CS 7600 Survey Paper: On-Stack Replacement

Ming-Ho Yee
Northeastern University

December 14, 2018

Abstract

On-stack replacement (OSR) is a programming language implementation technique that allows a running program to switch to a different version of code. For example, a program could start executing optimized code, and then transfer to and start executing unoptimized code. This was the original use case for OSR, to facilitate debugging of optimized code.

After its original use was established, OSR shifted to a different use case: optimizing programs. OSR allows the run-time system to detect if a program is executing an inefficient loop, recompile and optimize the method that contains the loop, and then transfer control to the newly compiled method. Another strategy is to optimize code based on some assumptions, then, if the assumptions are invalidated at run-time, transfer control back to the original, unoptimized code.

In this survey paper, we study how OSR was first introduced as a means for debugging, how it came to be used for program optimizations, its implementation as a reusable library, and other directions of research.

1 Introduction

On-stack replacement (OSR) is a technique used in some programming language implementations, such as Java and JavaScript, to improve program performance. In these implementations, code is generated at run time, in a process known as just-in-time (JIT) compilation. The run-time system, called a virtual machine (VM), executes programs, monitors program execution, and decides when to optimize a method or roll back an optimization. Throughout this process, OSR allows the VM to switch from one version of a method to another—while the method is still executing. For instance, if an inefficient version of a method is executing, the VM can compile a more optimized version, and then use OSR to switch to the new version and continue executing. In this example, OSR allows an implementation to improve the performance of a program as it runs. As another example, the VM could speculatively optimize a method based on some assumption, and if it observes the assumption to be invalid, use OSR to revert to the original method version.

However, OSR is a complex mechanism, with many difficult implementation details to overcome: the compiler must generate special metadata, the VM needs to map the local state of one method version to another version, low-level stack manipulation and swapping must be performed, and stack frames corresponding to inlined methods must be reconstructed. Because of these challenges, the significant implementation and maintenance costs of OSR means few production programming language implementations use OSR, even if the performance benefits are desirable. Nonetheless, OSR has been developed over many years, and is vital to enable significant optimizations in the implementations that do use it.

In this survey paper, we examine three advances in the development of OSR: the original OSR paper [Hölzle, Chambers, and Ungar, 1992], an early approach that uses OSR for optimizations [Fink and Qian, 2003], and an implementation of OSR in a popular compiler framework [D'Elia and Demetrescu, 2016]. We discuss how OSR was first introduced to facilitate debugging, how its primary use case shifted from debugging to program optimization, its implementation and use as a reusable library, and other directions of research.

2 Dynamic Deoptimization for Debugging

Hölzle et al. [1992] first introduced the OSR technique, under the name of *dynamic deoptimization*, to provide source-level debugging in the Self language. Self programs need to be highly optimized, otherwise they run too slowly to be practical. However, this makes debugging more difficult because the executable code no longer corresponds to the source code: instructions may be reordered, variables may be removed, or methods may be inlined. For example, the debugger may not be able to single-step through the program, examine and modify variables, or display a physical stack trace. To address these challenges, the authors developed a new strategy: when the user wants to interrupt and debug a running program, the VM generates an unoptimized version of the currently executing method and then switches to it.

The Self implementation demonstrated that OSR was feasible, and it pioneered many of the OSR techniques still in use today. For instance, OSR transitions can only occur at special *interrupt points* in the program—these interrupt points contain the necessary metadata for mapping optimized program states to unoptimized program states. The metadata describe the corresponding call tree and source position in the unoptimized program, as well as the locations and values of local variables and subexpressions. Another technique Hölzle et al. introduced was *lazy deoptimization*, where deoptimization is deferred for a method until that method's stack frame is at the top of the stack. This avoids the difficulty of modifying stack frames in the middle of the stack and then having to adjust other frames.

Most of the related work at the time concerned debugging optimized code, rather than switching between two versions of a method. The other debugging approaches often had compromises: either debugging ability was hindered for better optimizations, or the optimized code was slowed down by including debugging information. Only Self was able to provide full source-level debugging of optimized programs, by using OSR to switch from optimized code to unoptimized code.

The most similar prior work was Smalltalk-80 [Deutsch and Schiffman, 1984], which introduced an earlier version of interrupt points as well as *dynamic compilation*, more commonly known as JIT compilation. The interrupt points meant that a method was compiled only at its first invocation; furthermore, the interrupt points contained metadata to map compiled code back to source code. Compared to Self, however, Smalltalk did not perform optimizations, so there was no need for deoptimization.

3 On-Stack Replacement for Optimizations

Although OSR was originally developed as a means for debugging, it is now used as a tool for program optimization. Fink and Qian [2003] generalized the term *on-stack replacement* to refer to any kind of transition from one version of code to another, for example, a transition from optimized code to unoptimized code.[1] They prototyped this kind of transition in the Jikes Research Virtual Machine (RVM) by combining run-time *profiling* with *deferred compilation*. The RVM collects profiling data about a program's execution, identifies the frequently executed code branches, and then compiles only those branches. Uncommon branches are compiled as stubs; if a stub is actually reached during execution, it triggers re-compilation of the entire method as well as an OSR transition to the new code. This strategy reduces both compilation time as well as the size of the generated code. Furthermore, the RVM can choose between three levels of optimizations, based on estimating the potential compilation costs and performance speedups. Fink and Qian evaluated their system and observed modest, but promising results: they found an average improvement of 2.6% in the SPECjvm98 benchmarks, with 8% being the largest improvement.

Some of the ideas in the RVM originated from the Self [Chambers and Ungar, 1991, Hölzle and Ungar, 1994] and Java HotSpot [Paleczny, Vick, and Click, 2001] virtual machines. Chambers and Ungar introduced *deferred compilation* of uncommon branches in the Self virtual machine; however, the compilation decisions were based on static (*i.e.*, not run-time) information. Hölzle and Ungar extended this idea with *adaptive optimization*, where frequently executed methods are optimized and recompiled; this process also requires an OSR transition to the recompiled code. Paleczny et al. introduced an even more aggressive strategy, called *speculative optimization*: code is optimized under certain assumptions, but events may invalidate those assumptions and trigger a deoptimization. For example, many optimizations in HotSpot assume the class hierarchy does not change; however, dynamic class loading will invalidate this assumption. In this situation, HotSpot must regenerate the unoptimized code and continue execution there. Fink and Qian's approach is a form of speculative optimization, as the RVM assumes certain code branches are not taken.

The work by Fink and Qian was later extended by Soman and Krintz [2006] to unblock more optimizations. Specifically, Soman and Krintz decoupled the OSR metadata from the program code: instead of inserting OSR-specific instructions and metadata at every point in the program that could initiate an OSR transition, the VM maintains the metadata separately for each method, in a structure called a *variable map*. Furthermore, the variable map could be updated by the compiler, which means it no longer restricts optimizations.

Today, many production VMs implement the optimization techniques discussed above, which are only possible with OSR. These implementations include virtual machines for JavaScript (*e.g.*, Google V8,[2] Mozilla SpiderMonkey,[3] Apple JavaScriptCore,[4] and Microsoft ChakraCore[5]), the Java HotSpot VM, and implementations on top of the GraalVM.[6] All these implementations use OSR to switch from unoptimized code to optimized code (if a method is frequently executed), or from optimized code to unoptimized code (if optimization assumptions are invalidated). Unlike Self, none of these implementations use OSR for debugging.

4 Reusable Implementations of On-Stack Replacement

With multiple independent implementations of OSR, one question is whether OSR could be implemented in a library and reused. D'Elia and Demetrescu [2016] address this by extending the work

[1] Older work did not recognize deoptimization as a form of OSR.

[2] `https://v8.dev`

[3] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`

[4] `https://developer.apple.com/documentation/javascriptcore`

[5] `https://github.com/microsoft/ChakraCore`

[6] `https://www.graalvm.org/`

of Lameed and Hendren [2013] to implement OSR in the LLVM compiler framework. LLVM is a popular framework for implementing optimizers and compilers; it also provides the MCJIT library for developing JIT compilers. However, MCJIT lacks a mechanism for performing OSR, meaning LLVM users must develop their own OSR implementations. To remedy this situation, Lameed and Hendren designed a reusable OSR implementation in LLVM. Later, D'Elia and Demetrescu updated and extended that implementation: they support OSR transitions from any point in the program, and they allow targets of OSR transitions to initiate their own OSR transitions.

Implementing OSR in the LLVM framework has two significant benefits. First, the implementation is platform-independent and reusable. Other developers can use LLVM to implement JIT compilers without having to implement OSR on their own; they also get support for all hardware architectures that LLVM targets. Second, since the OSR mechanism is entirely in LLVM, it can be optimized by LLVM's extensive optimization passes.

Two other approaches include Truffle/Graal and the Mu micro virtual machine. Truffle [Wimmer and Würthinger, 2012, Würthinger, Wöß, Stadler, Duboscq, Simon, and Wimmer, 2012] is a framework for implementing programming languages on the Graal VM [Würthinger, Wimmer, Wöß, Stadler, Duboscq, Humer, Richards, Simon, and Wolczko, 2013]. The language implementer only needs to write an interpreter for their programming language, and can take advantage of Graal's optimization and deoptimization mechanisms—the author does not need to be concerned with the details of OSR. Mu [Wang, Lin, Blackburn, Norrish, and Hosking, 2015] takes a different approach, by providing a minimal virtual machine abstraction that supports, among other features, OSR [Wang, Blackburn, Hosking, and Norrish, 2018].

5 Other Related Work

Aside from addressing OSR in terms of debugging, optimization, and implementation, researchers have considered other directions of study. In this section, we briefly summarize a few of those topics.

Flückiger, Scherer, Yee, Goel, Ahmed, and Vitek [2017] studied compiler correctness with deoptimizations. Prior work had studied compiler correctness, as well as the correctness of JIT compilation; however, no other work had considered speculative optimizations. The challenge is that speculative optimizations are inherently unsound, as they depend on assumptions that could be wrong—but OSR allows the program to recover when assumptions are invalid.

D'Elia and Demetrescu [2018] proposed to decouple OSR from VMs: they suggested that OSR could be treated as a general mechanism for transferring execution between two versions of a program, without support from a virtual machine. For instance, a program about to crash could use OSR to switch to an unoptimized version of code, making it easier for developers to examine the core dump. As another example, OSR could be used to obfuscate a program by randomly switching execution between two program versions. D'Elia and Demetrescu presented a formal framework to reason about these transformations, implemented a prototype in LLVM, and revisited debugging of optimized code as a case study.

Essertel, Tahboub, and Rompf [2018] continued these ideas and examined how OSR could be used in metaprogramming, *i.e.*, programs that generate other programs or source code. Again, the goal was to use OSR to implement speculative optimizations and deoptimizations, without needing a VM. They demonstrated the feasibility of their approach by using OSR to optimize an SQL-to-C query compiler.

So far, all the discussion in this paper, and mentioned implementations, concerned *method-based* JIT compilers, where the unit of compilation is an entire method. An alternate approach is a *tracing* or *trace-based* JIT, where the unit of compilation is a sequence of instructions. As a program runs, the VM records frequently-executed sequences of instructions—which typically correspond to hot loops and may cross method boundaries—and generates faster, more optimized code. Subsequent executions of the trace must execute the optimized code, otherwise the VM will deoptimize and resume execution in the original code, using a transition similar to OSR. For a further discussion on tracing JITs, but with minimal attention to OSR, refer to the survey by Yee [2017].

6 Conclusions

The on-stack replacement technique has evolved since it was first introduced in the implementation of the Self programming language. Originally designed as a means for debugging optimized code, today OSR is mainly used for program optimizations. The ability to change the version of code that is executed grants an enormous amount of flexibility: it means a program is never forced to commit to a single version of code. For example, a program can always transfer execution to a more optimized version of code, or a program can be optimized under certain assumptions and roll back the optimization if the assumptions are invalidated.

Today, OSR is used for optimizations in a number of programming language implementations, such as Java HotSpot, Google V8, Mozilla SpiderMonkey, Apple JavaScriptCore, and Microsoft ChakraCore. Researchers are continuing to explore reusable implementations of OSR, such as the Truffle/Graal project, and also how OSR can be generalized as a program transformation mechanism without relying on a virtual machine. Just as the main use case for OSR has changed since it was first introduced, it is possible that more applications of OSR will continue to be discovered.

References

Craig Chambers and David Ungar. Making Pure Object-oriented Languages Practical. In *Proc. of the International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1991. doi: 10.1145/117954.117955. URL `http://bibliography.selflanguage.org/_static/practical.pdf`.

Daniele Cono D'Elia and Camil Demetrescu. Flexible On-stack Replacement in LLVM. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, 2016. doi: 10.1145/2854038.2854061. URL `https://www.authorea.com/users/55853/articles/66046-flexible-on-stack-replacement-in-llvm/_show_article`.

Daniele Cono D'Elia and Camil Demetrescu. On-Stack Replacement, Distilled. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 2018. doi: 10.1145/3192366.3192396. URL `https://season-lab.github.io/papers/osr-distilled-pldi18.pdf`.

L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*, 1984. doi: 10.1145/800017.800542.

Grégory Essertel, Ruby Tahboub, and Tiark Rompf. On-Stack Replacement for Program Generators and Source-to-Source Compilers. Preprint, 2018. URL `https://www.cs.purdue.edu/homes/rompf/papers/essertel-preprint201811b.pdf`.

Stephen J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, 2003. doi: 10.1109/cgo.2003.1191549.

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. Correctness of Speculative Optimizations with Dynamic Deoptimization. *Proc. ACM Programming Languages*, 2(POPL), 2017. doi: 10.1145/3158137. URL `https://www.o1o.ch/sourir.pdf`.

Urs Hölzle and David Ungar. A Third-generation SELF Implementation: Reconciling Responsiveness with Performance. In *Proc. of the International Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA)*, 1994. doi: 10.1145/191080.191116. URL `http://bibliography.selflanguage.org/_static/third-generation.pdf`.

Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*, 1992. doi: 10.1145/143095.143114. URL `http://bibliography.selflanguage.org/_static/dynamic-deoptimization.pdf`.

Nurudeen A. Lameed and Laurie J. Hendren. A Modular Approach to On-stack Replacement in LLVM. In *Proc. of the International Conference on Virtual Execution Environments (VEE)*, 2013. doi: 10.1145/2451512.2451541. URL `http://www.sable.mcgill.ca/publications/papers/2013-1/vee13-lameed.pdf`.

Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot^{TM} Server Compiler. In *Proc. of the Symposium on Java^{TM} Virtual Machine Research and Technology (JVM)*, 2001. URL http://static.usenix.org/event/jvm01/full_papers/paleczny/paleczny.pdf.

Sunil Soman and Chandra Krintz. Efficient and General On-Stack Replacement for Aggressive Program Specialization. In *Proc. of the International Conference on Programming Languages and Compilers (PLC)*, 2006. URL https://www.cs.ucsb.edu/~ckrintz/papers/osr.pdf.

Kunshan Wang, Yi Lin, Stephen M. Blackburn, Michael Norrish, and Antony L. Hosking. Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development. In *Summit on Advances in Programming Languages (SNAPL)*, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.321.

Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Hop, Skip, & Jump: Practical On-Stack Replacement for a Cross-Platform Language-Neutral VM. In *Proc. of the International Conference on Virtual Execution Environments (VEE)*, 2018. doi: 10.1145/3186411.3186412. URL https://wks.github.io/downloads/pdf/osr-vee-2018.pdf.

Christian Wimmer and Thomas Würthinger. Truffle: A Self-optimizing Runtime System. In *Proc. of the Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, 2012. doi: 10.1145/2384716.2384723.

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST Interpreters. In *Proc. of the Symposium on Dynamic Languages (DLS)*, 2012. doi: 10.1145/2384577.2384587. URL http://lafo.ssw.uni-linz.ac.at/papers/2012_DLS_SelfOptimizingASTInterpreters.pdf.

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proc. of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2013. doi: 10.1145/2509578.2509581. URL http://lafo.ssw.uni-linz.ac.at/papers/2013_Onward_OneVMToRuleThemAll.pdf.

Ming-Ho Yee. Tracing JITs for Dynamic Languages, 2017. URL https://prl.ccs.neu.edu/blog/2017/03/15/tracing-jits-for-dynamic-languages/.