Efficient and General On-Stack Replacement for Aggressive Program Specialization

Sunil Soman Chandra Krintz
Computer Science Department
University of California, Santa Barbara, CA 93106

E-mail: {sunils,ckrintz}@cs.ucsb.edu

Abstract

Efficient invalidation and dynamic replacement of executing code – on-stack replacement (OSR), is necessary to facilitate effective, aggressive, specialization of object-oriented programs that are dynamically loaded, incrementally compiled, and garbage collected. Extant OSR mechanisms restrict the performance potential of program specialization since their implementations are special-purpose and restrict compiler optimization.

In this paper, we present a novel, general-purpose OSR mechanism that is more amenable to optimization than prior approaches. In particular, we decouple the OSR implementation from the optimization process and update the program state information incrementally during optimization. Our OSR implementation efficiently enables the use of code specializations that are invalidated by any event – including those external to program code execution. We improve code quality over the extant, state-of-the-art, resulting in performance gains of 1-31%, and 9% on average.

Keywords
Program specialization, on-stack replacement, code invalidation, virtual machines, Java.

1 Introduction

Advanced implementations of modern object-oriented languages, such as Java [12], depend on runtime environments that employ "just-in-time" compilation for performance. These language environments dynamically compile program code that is loaded incrementally, and possibly from a remote target. Although optimization has the potential to significantly improve program performance, on-the-fly and incremental compilation makes optimization implementation challenging. In particular, the compiler must predict which optimizations will be most successful for the remainder of program execution.

One successful approach to enable more aggressive optimizations for such systems is to *specialize* the program for a particular behavior or set of execution conditions, and then to *undo* the optimization when conditions change. This enables the compiler to identify optimizations that are likely to be successful in the near-term, and then to apply others as it learns more about future program and resource behavior. Examples of such optimizations include virtual call inlining, exception handler removal, and memory management system optimizations.

Key to the success of such specializations is the presence of an efficient, general-purpose, mechanism for undoing optimization *on-the-fly* when associated assumptions are rendered invalid. Future method invocations can be handled by recompilation. Replacing the currently executing method and re-initiating execution in the new version is termed *on-stack replacement* (OSR). OSR has previously been successfully employed for debugging optimized code [14], for deferred compilation [7, 23, 10, 19], and method optimization-level promotion [15, 10, 19].

Each of these prior approaches, however, is specific to the particular optimization being performed. Moreover, extant approaches inhibit compiler optimizations by employing special instructions for state collection, in-line with generated program code. Such implementations, in addition to increasing code size, increase variable live ranges and restrict code motion.

We present a general-purpose OSR implementation, which can be used for any OSR-based specialization, and is more amenable to compiler optimization. We decouple OSR from the program code, and consequently, from compiler optimizations. This significantly improves code quality over an extant, state-of-the-art approach, and enables OSR to occur at any point at which control can transfer out of an executing method. We can, therefore, support existing OSR-based specializations as well as other, more aggressive, optimizations that are triggered by events *external* to the executing code, including, class loading, changes in program and JVM behavior, exception conditions, resource availability, and user events, such as dynamic software updates.

In the sections that follow, we provide background on OSR (Section 2) and detail our extensions and implementation in the Jikes Research Virtual Machine (JikesRVM) [1] (Section 3). We then describe three OSR-based specializations (Section 4). The first two are existing specializations for virtual method dispatch and for dynamic GC switching. Our OSR implementation enables performance improvement by 6% for the former technique, and by 9% for the latter. We also present a novel specialization for generational garbage collection in which we avoid adding write-barriers to generated code until they are required (i.e. until there are objects in the old generation(s)). Our results indicate that we improve startup time for programs by 6% on average; for short running programs we reduce execution time by 8-14%. Following our empirical evaluation and analysis (Section 5), we present our conclusions and plans for future work (Section 6).

2 Background and Related Work

On-stack replacement (OSR) [7, 15] is a four step process. The runtime extracts the execution state (current variable values and program counter) from a particular method. The compilation system then recompiles the method. Next, the compiler generates a stack activation frame and updates it with the values extracted from the previous version. Finally, the system replaces the old activation frame with the new and restarts execution in the new version.

Given OSR functionality, a dynamic compilation system can implement aggressive specializations based on conditions that may change as execution progresses. If these conditions change as a result of an external or a runtime event, the compilation system can recompile (and possibly re-optimize) the code, and replace the currently executing version [7]. OSR has been employed by extant virtual execution environments for dynamically de-optimizing code for debugging [14], for deferred compilation of method regions to avoid compilation overhead and improve data flow [7, 23, 10, 19], and to optimize methods that execute unoptimized for a long time without returning [15, 10, 19].

These OSR implementations were specially designed for a target purpose. For example, OSR for deferred compilation is implemented as an unconditional branch to the OSR invocation routine. OSR for method promotion and de-optimization is implemented as part of the thread yield process. The JikesRVM from IBM Research is a virtual machine that currently employs a state-of-the-art, special-purpose, OSR for deferred compilation and method promotion [10]. It is this system that we extend in this work.

The JikesRVM optimizing compiler inserts an instruction, called an OSRPoint, into the application code at the point at which OSR should be unconditionally performed. This instruction is implemented similarly to a call instruction – execution of an OSRPoint inserted in a method causes OSR to be performed *unconditionally* for that method. The OSRPoint records necessary execution state which consists of values for bytecode-level, local and stack variables, and the current program counter. The execution state is a mapping that provides information about runtime values at the bytecode-level so that the method can be recompiled and execution restarted with another version. The JikesRVM OSR implementation is similar to that used in other systems for similar purposes [14, 23, 13, 19], although the authors of each refer to the instruction using different names, e.g., interrupt points [14], OPC_RECOMPILE instructions [23], and uncommon traps [19].

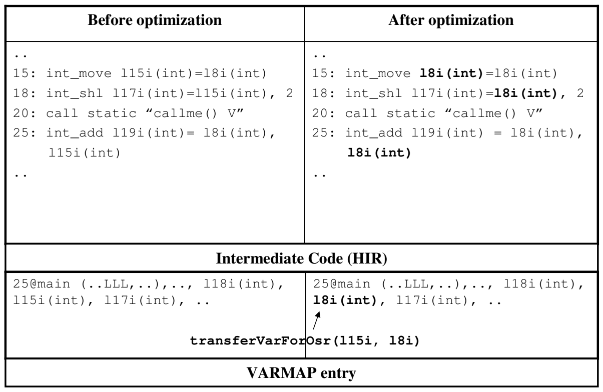
Prior OSR implementations restrict compiler optimization. Since the compiler considers all method variables (locals and stack) live at an OSRPoint, it artificially extends the live ranges of variables and limits the applicability of optimizations such as dead code elimination, load/store elimination, alias analysis, and copy/constant propagation. In addition, the compiler cannot move variable definitions around OSRPoints [14, 10].

3 Extending On-Stack Replacement

Existing implementations for OSR do not significantly degrade code quality when there are a small number of OSRPoints [10, 14]. However, our goal is to enable more aggressive, existing and novel, specializations including those triggered by events external to the executing code, e.g., class loading, exception handling, garbage collection optimization, and dynamic software updating. For such specializations, we may be required to perform OSR at all locations in a method at which execution can be suspended. Since there are a large number of such locations, many along the critical path of the program, we require an alternative implementation that enables optimization at, and across, these locations.

To this end, we extended the current OSR implementation in the JikesRVM with a mechanism that decouples the OSR implementation from the optimization process. In particular, we maintain execution state information without inserting extra instructions at every point at which control may transfer out of a method. This includes implicit yield points (method prologues, method epilogues, and loop back-edges), call sites, exception throws, and explicit yield points. The data structure we employ for state collection is called a VARMAP (short for *variable map*).

A VARMAP is a per-method list of thread-switch points and bytecode variables that are live at each point. We maintain this list independent from compiled code so it does not affect liveness of code variables. We update the VARMAP incrementally as the compiler performs its optimizations. Our VARMAP is somewhat similar in form to the data structure described in [9]


Before optimization

```
..
15: int_move l15i(int)=l8i(int)
18: int_shl l17i(int)=l15i(int), 2
20: call static "callme() V"
25: int_add l19i(int)= l8i(int),
    l15i(int)
..
```

After optimization

```
..
15: int_move l8i(int)=l8i(int)
18: int_shl l17i(int)=l8i(int), 2
20: call static "callme() V"
25: int_add l19i(int) = l8i(int),
    l8i(int)
..
```

Intermediate Code (HIR)

```
25@main (..LLL,..),.., l18i(int),
l15i(int), l17i(int), ..
```

```
25@main (..LLL,..),.., l18i(int),
l8i(int), l17i(int), ..
```

```
transferVarForOsr(l15i, l8i)
```

VARMAP entry

Figure 1. Shows how the VARMAP is maintained and updated.

for tracking pointer updates in the presence of compiler optimizations, to support garbage collection in Modula-3. Our implementation is different in that we track all stack, local, and temporary variables across a wide range of compiler optimizations automatically and transparently, and do so online, during dynamic optimization of Java programs.

To update VARMAP entries during optimization, we defined the following system methods:

- *transferVarForOsr(var1, var2)*: Record that `var2` will be used in place of `var1`, henceforth in the code (e.g. as a result of copy propagation)

- *removeVarForOsr (var)*: Record that `var` is no longer live/valid in the code.

- *replaceVarWithExpression(var, vars[], operators[])*: Record that variable `var` has been replaced by an expression that is derivable from the set of variables `vars` and `operators`.

Our OSR-enabled compilation system handles *all* JikesRVM optimizations that impact liveness except tail call and tail recursion elimination (which eliminate stack frames entirely). This set of optimizations includes copy and constant propagation, common subexpression elimination (CSE), branch optimizations, dead-code elimination (DCE), and local escape analysis optimization.

When a variable is updated, the compiler calls a wrapper function that automatically invokes the appropriate VARMAP functions. This enables users to easily extend the compiler with new optimizations, without having to manually handle VARMAP updates. For example, when copy/constant propagation or CSE replaces a use of a variable (*rvalue*) with another variable or constant, the wrapper performs the replacement in the VARMAP entry by invoking the `transferVarForOsr` function.

We handle DCE and local escape analysis using a similar wrapper function for updates to variable definitions. When definitions of dead variables that are present in the VARMAP are removed during optimization, the wrapper replaces its entry with the *rvalue* in the instruction, or records all of the right-hand variables along with operators used to derive the *lvalue* in case of multiple *r-values* (we currently only handle simple unary or binary expressions). Similarly, we replace variables eliminated by escape analysis with their *rvalues* in the VARMAP.

The compiler automatically updates the VARMAP during live variable analysis. We record variables that are no longer live at an OSR point, and the relative position of each in the map. Every variable that live analysis discovers as dead, is set to a `void` type in the VARMAP. We cannot simply drop the variable from the entry since we must track the relative positions of variables in order to enable their restoration in the correct location during OSR. During register allocation, we update the VARMAP with physical register and spill locations. This enables us to restore these locations during OSR as well.

Figure 1 shows how we maintain and update VARMAP entries. We show the VARMAP before and after copy propagation. We show a snippet of Java bytecode (left), and high-level, JikesRVM intermediate code representation (HIR). We also show the VARMAP entry (bottom) for the `callme()` call site which contains the bytecode index (25) of the instruction that follows the call site, as well as three local (l) variables with integer (i) types (`a: l8i, b: l15i, c: l17i`). The index identifies the place in the code at which execution resumes following the call (in this example), if OSR occurs during `callme()`. The VARMAP tracks method variables (`l8i, l15i, and l17i`) at this point; this entry is used by the system to update the frame of the new version of `meth()` during OSR.

In the HIR before optimization, the instruction at bytecode index 15 copies variable `l8i` to `l15i` then uses both in the subsequent code. The optimization replaces all `l15i` uses with `l8i` during copy propagation. During this optimization, the replacement invokes a wrapper which automatically updates the VARMAP, i.e., the wrapper calls `transferVarForOsr(l15i,l8i)` prior to replacement. DCE will remove bytecode instruction 15 in a later pass. DCE of the instruction at index 15 will cause the VARMAP to record that `l15i` is no longer live.

When the compilation of a method completes, we encode the VARMAP using a compact encoding for OSRPoints from the existing implementation [10]. The encoded map contains an entry for each OSR point, which consists of the *register map* – a bitmap that indicates which physical registers contain references (which the garbage collector may update). In addition, the map contains the current program counter (bytecode index), and a list of pairs *(local variable, location)* (each pair encoded as two integers), for every inlined method (in case of an inlined call sequence). The encoded map is maintained in the system throughout the lifetime of the the program. All other data structures required for OSR-aware compilation (including the VARMAP) are reclaimed during GC.

We also handle certain cases specially to enable correct state collection and to enable efficient code generation. These cases include call site return values, exception throws and yield points, and machine-specific instruction selection and optimization.

If the callee of a method to be replaced returns a value, the OSR process extracts that value from the execution state so that it is available in the new version of the method. Return values are typically stored in specific registers as dictated by the compiler's calling convention and the target architecture, and must be extracted from these registers. We handle exception throws and yield points as we do call sites. Moreover, we do not perform OSR for methods for which execution has reached their epilogue.

3.1 Triggering On-Stack Replacement

Our system requires an appropriate trigger to initiate OSR. The current OSR implementation in JikesRVM uses a compiler-driven, eager approach. The compiler inserts a call to a system method, which is guarded by a condition that checks the validity of the specializing assumption. Execution of this method causes OSR for the callee.

The alternative, an external, lazy trigger, is more appropriate for enabling OSR due to events in the environment, out of line with application execution, such as is done in Self-91 for debugging optimized code [14]. The runtime triggers OSR when it deems invalid an assumption previously used by the compiler for specialization. The runtime invokes a helper method (called `OSR helper`) that performs OSR. The method either patches the code of the executing method(s) with code that invokes the rest of the OSR process, or modifies the return address of each callee of the method to be replaced. This approach does not require the insertion of conditional calls into the application code or all invalidated methods to be replaced at once. As a result, it enables a fast call stack traversal.

The `OSR helper` reroutes the return address of the specialized method's callee to itself so that OSR it can perform OSR on the method incrementally – when the callee returns. When this occurs, the `OSR helper` sets up a new stack frame with execution state extracted from the specialized method's stack. To preserve values the contained in registers during the execution of specialized method, the `OSR helper` saves all registers (volatiles and non-volatiles) into its stack frame. The `OSR helper` expects to find the specialized method by traversing the stack, and hence, we "fake" a call to the `OSR helper` from the specialized method. More precisely, apart from re-routing the return address of the specialized method's callee, we set the return address of the `OSR helper` to point to the current instruction pointer in the specialized code, since this value will be used during OSR to calculate the location to resume execution at the corresponding location in the new version of the method. We also update `OSR helper`'s stack pointer appropriately.

4 Aggressive Program Specialization

We compare our OSR implementation to the current state-of-the-art as part of our empirical evaluation. In addition, we employ our system for novel specialization for generational garbage collection (GC) and to improve two existing specialization systems, one for dynamic dispatch of virtual method calls and the other for dynamic switching between GCs.

Write-Barrier Specialization for Generational GC A generational garbage collector (GC) [18, 24] segregates objects into young and old parts (generations), based on their lifetimes. This separation enables young objects to be garbage collected independently and more frequently than the old, without collecting the entire heap. Once objects age in the young generation, they are promoted to the old. A generational GC must record references from mature objects to nursery objects on every field assignment. The compiler inserts extra instructions called write-barriers [25, 16, 3] into application code for this purpose, for every pointer mutation (putfields and array stores in Java). These extra instructions generally degrade program performance. Researchers have recently found that with intelligent write-barrier code and modern architectures, write-barrier cost for many programs is small [3, 4], compared to the benefits of generational GC.

However, for applications that do not allocate enough memory to require a GC, or that trigger GC later in their lifetime, write-barrier execution, regardless of how efficiently the write-barriers are implemented, is pure overhead. Such applications have become more plentiful as the cost of memory for modern processors has plummeted and memory has become abundant.

To address this issue, we introduce a novel specialization technique that avoids inserting write-barriers. That is, we *specialize* frequently executing (hot) methods without write-barriers. The compiler inserts write-barriers for all cold methods. If a GC occurs and objects are promoted to the mature space, we invalidate and replace hot methods that require write-barriers. We do not specialize hot methods once GC has occurred and objects have been promoted to the mature space, or when the maximum heap size is below 500MB, or if heap residency exceeds 60% (identified empirically).

Consequently, programs that perform no GC experience minimal write-barrier overhead (due to execution of cold, unoptimized methods). Also, programs that do not require GC at startup, experience improved startup and reap the benefits of generational collection, with only minor OSR overhead imposed at the first GC.

Specialization of Virtual Method Dispatch
Virtual method dispatch is a technique for aggressively binding virtual call sites to their implementations. The cost of executing virtual methods is higher than static method invocations since the underlying object type must be extracted at runtime, to identify the method to dispatch. [17] discusses many such techniques.

JikesRVM employs pre-existence [8], code patching [17], and deferred compilation [10] to reduce the overhead of dynamic dispatch. The optimizing compiler speculatively inlines a virtual method using a heuristic to predict the likely underlying type of an object [5, 11, 15]. The compiler omits a check to protect inlined code if it can determine pre-existence – if given the currently loaded class files, the object type does not change once method execution commences. If a class loaded later violates these assumptions, only future invocations of the method are recompiled, obviating the need for OSR [8].

If the compiler cannot establish a pre-existence guarantee, then it must insert a "guard" to protect the speculatively inlined code. JikesRVM uses code patching for this purpose. The compiler generates code for the inlined sequence, and instructions that will be executed if inline specialization is invalidated. If class loading invalidates the assumption used for inlining, the VM overwrites the first instruction of the inlined sequence that occurs after the current point in the execution, with a branch to the alternative path. The compiler inserts an OSRPoint at the alternative path to avoid compilation of this unlikely case. Program execution of an OSRPoint causes the current method to be replaced with a new version that implements only the dynamic dispatch code sequence. As an alternative to code patching, we extended this JikesRVM implementation to use our VARMAP-based OSR for speculatively inlined call sites for which pre-existence cannot be guaranteed.

Specialization of Automatic GC-Switching System
In prior work, we developed an extension to JikesRVM that enables application-specific dynamic GC selection [21]. This system employs multiple garbage collectors within a single JikesRVM execution image and can switch between them at runtime in an effort to improve performance. This prior work indicates the potential for significant performance improvements (11-14% for standard benchmarks and up to 44% for short running applications).

To enable these improvements, the compilation system must specialize the code for the underlying GC, i.e., inline allocation sites and include or omit generational write-barriers, as appropriate. When the underlying GC changes, the system must invalidate and if necessary, OSR specialized methods to ensure correctness. The system only performs OSR for optimized methods that may execute specialized code after the current program point. Support for OSR however significantly inhibits dynamic GC selection from achieving its full potential. We modified the switching system to use our VARMAP extension, in place of the extant (default JikesRVM) approach.

5 Empirical Evaluation

To evaluate the efficacy of our OSR implementation, we measured the performance of our system for the aggressive specialization techniques described previously. We first detail the experimental methodology and then present results.

5.1 Experimental Methodology

We used a dedicated 2.4GHz x86-based single-processor Xeon machine (with hyperthreading) running RedHat Linux 9. We focus on the x86 system since it is currently the most popular processor for desk-side systems and Internet computing. We extended JikesRVM version 2.2.0 with jlibraries R-2002-11-21-19-57-19.

We ran all of our experiments using the Fast Adaptive JikesRVM configuration with a Generational Mark-Sweep (GMS) garbage collector. This GC uses a copying young generation and a mark-sweep collected mature generation. We generated and employed off-line profiles to guide hot-method optimization to reduce non-determinism resulting from adaptive selection of methods to be optimized (our profiles are available upon request) as is done in similar studies [20, 21]. Our benchmarks are a subset of the the SpecJVM98 [22] and the JOlden [6] benchmark suites. We executed each benchmark multiple times and report the average of all but the first run. We report the compilation overhead introduced by our system separately.

5.2 Results

We first present results from experiments that compare our VARMAP-based implementation to a state-of-the-art extant OSR implementation. In order to create a valid reference to compare our implementation against, we extended the JikesRVM OSR system to remove OSRPoints (described in Section 2) at the end of compilation. By doing so, we are able to insert OSRPoints at every point in a method at which execution can be suspended, thus potentially allowing OSR at these points, without actually triggering OSR unconditionally.

Figure 2 shows the results from this comparison. The y-axis is percent reduction in execution time enabled by VARMAP OSR over OSRPoints. The Average bar shows the average across all benchmarks, and Average Spec98 shows the average for only the SpecJVM benchmarks. Both bars are averages across heap sizes (minimum to 12 times the minimum), to include the impact of any space overhead introduced by VARMAPs.

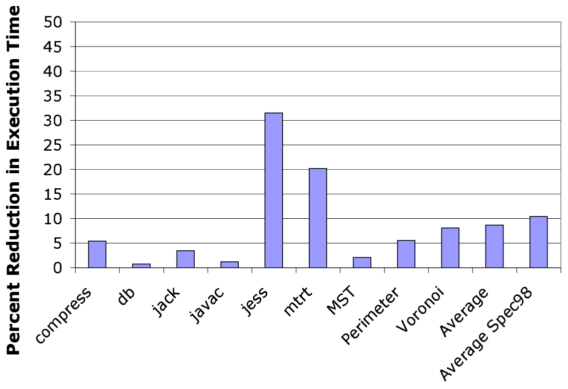


Figure 2. Performance of our VARMAP OSR Implementation vs the JikesRVM Reference System, across heap sizes.

	Compilation Time (ms)		Space Added (KB)	
Benchmark	Clean	VARMAP	Compile Time	Runtime
compress	68	79	14.52	3.16
db	91	117	24.57	5.26
jack	445	543	139.67	30.00
javac	1962	2540	629.94	136.98
jess	504	656	136.80	29.20
mtrt	595	746	154.38	33.50
MST	50	66	17.03	3.73
Perimeter	86	66	15.82	3.47
Voronoi	96	129	62.06	13.49
Avg.	433	549	132.75	28.75
Avg. Spec98	611	780	183.31	39.68

Figure 3. Overhead of our OSR-VARMAP Implementation in the JikesRVM reference system. Cols 2 & 3 indicate compilation times and Cols 5-6 show space overhead.

VARMAP shows significant improvement in application performance – by 9% on average across all benchmarks, and by over 10% across the SpecJVM benchmarks. jess and mtrt show the most benefit, 31% and 20% respectively. For these benchmarks, the original implementation severely inhibits optimization, particularly due to increased register pressure by artificially extending live ranges of variables, past their last actual use. This results in a large number of variable spills to memory. With our implementation, we do not need to maintain conservative liveness estimates, since we track liveness information accurately.

Other benchmarks show benefits of 5% or less. For these benchmarks, improved code quality does not impact overall performance significantly. Since these programs are short running, they are not heavily optimized. In addition, OSR VARMAP does impose some GC overhead, since we maintain information for each possible OSR point – which for short running codes is not fully amortized, especially for small heap sizes.

Figure 3 details the space compilation overhead of

	Clean	With WBSpec				
Benchmark	ET (s)	ET (s)	% Impr.	WBs Elim.	OSRs	OSR Time (ms)
compress	6.96	6.72	3.45	1209	0	0
jess	2.97	2.96	0.37	6902267	13	3.75
db	17.03	15.83	7.05	26852028	0	0
javac	6.71	6.56	2.24	2598303	1	0.38
mtrt	6.23	5.92	4.98	1851952	0	0
jack	4.11	4.08	0.73	4685883	7	3.94
MST	0.87	0.75	13.79	4683081	0	0
Perimeter	0.25	0.23	8	3170646	0	0
Voronoi	1.44	1.21	15.97	16047220	0	0
Avg.	5.17	4.92	6.29	7421399	2.33	0.90
Avg. Spec98	7.34	7.01	3.14	7148607	3.50	1.35

Figure 4. Performance of write-barrier specialization (WBSpec) for a heap size of 500MB using the popular Generational Mark Sweep GC.

our system. Columns 2 and 3 show the compilation time for the reference system and our VARMAP, respectively. Column 4 shows the percentage degradation in compilation time imposed by VARMAP. Columns 5 and 6 show the space overhead introduced by VARMAP during compilation time (collectable) and runtime (permanent), respectively. On average, our system increases compilation time by just over 100ms, and adds space overhead of 133KB that is collected, and 29KB that is not collectable.

Specialization for Generational Garbage Collection

We next present results for a novel OSR-based specialization for write-barrier removal for generational GCs. Prior to the initial GC when there are no objects in the mature space, write-barriers are not required, and thus, impose pure overhead. Our goal with this specialization is to reduce the overhead of write-barriers for programs that do not require GC, and to improve the startup performance of those programs that do.

For this specialization, we employ the popular Generational Mark Sweep (GMS) collector. The compiler checks the maximum heap size to ensure that it is large enough to warrant specialization (>=500MB), and that heap residency (pages used/pages allocated) is low (<=60%). We identified both values empirically. If no GCs have occur, the compiler elides write-barriers.

This specialization must be invalidated when objects are first promoted to the mature space. Our system only performs OSR for methods that require write barriers and when there are field assignments that will execute after the point at which execution has been suspended.

We present the performance of write-barrier specialization (referred to as WBSpec) in Figure 4. We used a heap size of 500MB for these experiments. Column 2 and 3 show the execution time performance in seconds without and with WBSpec, respectively. Column 4 shows the percent improvement enabled by WBSpec. Column 5 shows the number of write barriers eliminated. The final two columns show the OSR overhead

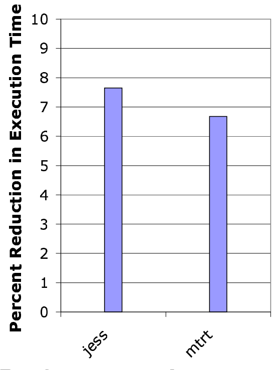


Figure 5. Performance improvement from using OSR VARMAP for speculative devirtualization (dynamic dispatch) of virtual methods. We omit benchmarks which showed no significant change.

imposed – column 6 is the number of OSRs that occur and column 7 is the total time for all OSRs. For many of the benchmarks, no OSR is required since a minor GC is not triggered for these. For those that require OSR, the overhead is very small.

On average, WBSpec improves performance by 6% across benchmarks. For the SPECJVM benchmarks (first 6 in the table), WBSpec improves performance by 3% on average. For benchmarks that require GC, this benefit is during program startup. The JOlden benchmarks require no GC when we use a heap size of 500MB. We believe that such applications are ideal candidates for specialization presented. The average improvement in execution time for these benchmarks is 13%.

OSR VARMAP for Existing Specializations

For guard-free dynamic dispatch, we replace code patching and OSRPoints for deferred compilation with our VARMAP implementation to guard speculatively inlined virtual method calls. The compiler inlines calls that meet size constraints and for which a single object target can be predicted [2]. To preserve correctness, we employ OSR to replace code that does not have checks inserted, when class loading invalidates assumptions made by the compiler. Upon recompilation, the compiler generates a new version of the method that implements dynamic dispatch at the previously inlined call site. We only perform OSR for methods for which the compiler cannot establish pre-existence guarantees (see Section 4).

Figure 5 shows the impact of using VARMAP over using the original OSRPoint and code patching. We only presents the results for the benchmarks (2) that implemented call sites for which the compiler was unable to make pre-existence guarantees. For jess, a single inlined method with an eliminated guard (*_202_jess.jess.ValueVector.size()*) constitutes 8% of the total number of method invocations. In the case of mtrt, 3 such methods constitute over 50% of the total invocations.

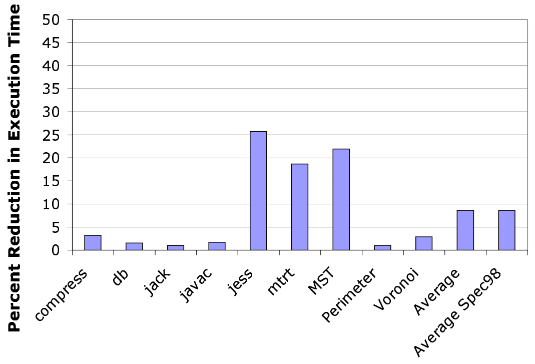


Figure 6. Performance improvement from using OSR VARMAP for OSR within an automatic GC switching system.

We also used our OSR implementation to improve the performance of a dynamic, garbage-collection switching system within JikesRVM that we developed in prior work. This system enables performance gains (including all overheads) of 11-14% on average. However, this system incurs an overhead of 10% on average, even when no dynamic switching is triggered. This overhead inhibits the full performance potential of the system, the primary source of which is the need for OSR support. Our OSR implementation reduces the overhead of the system by almost half.

Figure 6 shows the benefits on total execution time due to the use of our OSR implementation in the GC switching system. The data is the average improvement across a range of heap sizes for each benchmark (from the minimum to 12x the minimum). VARMAP improves performance most significantly for jess (26%), mtrt (19%), and MST (22%). This is due to the number of OSR points in the hot methods for these benchmarks, and due to the fact that MST is very short running. Across benchmarks, VARMAP shows a 9% improvement. This result is similar to the performance improvement enabled by VARMAP versus the extant state-of-the-art that we presented previously.

6 Conclusions and Future Work

We present a new implementation of on-stack replacement (OSR) that decouples dynamic state collection from compilation and optimization. Unlike existing approaches, our implementation does not inhibit compiler optimization, and enables the compiler to produce higher quality code. Our empirical measurements within JikesRVM show a performance improvement by 9% on average (from 1% to 31%), over a commonly used implementation. We implement a novel, OSR-based specialization for write-barrier removal in generational GC that improves performance by 6% on average. Moreover, we empirically confirm that our system is effective for extant specializations: dynamic dispatch of virtual methods, and for automatic GC switching.

As part of future work, we plan to employ our OSR system for other aggressive specializations. These include removing infrequently executing instructions, e.g. exception handling code. In addition, OSR can be used to trigger dynamic software updates in highly available server systems. In such environments, control never leaves a particular method (typically, a program loops forever listening for service requests, and issues requested work to slave processes). Using OSR, we can, hence, upgrade code without affecting service availability. Finally, we plan to investigate the use of OSR in aggressive incremental alias, and escape analysis to perform speculative pointer-based optimizations, such as, stack allocation of objects, memory layout optimizations, and synchronization removal.

Acknowledgements

We thank Kathryn McKinley and Steve Fink for their invaluable input and suggestions on this work. This work was funded in part by NSF grant Nos. CAREER-CNS-0546737, ST-HEC-0444412, and EHS-0209195 and grants from Intel, Microsoft, and UCMicro.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–221, 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[3] S. Blackburn and K. McKinley. In or Out? Putting Write Barriers in Their Place. In *International Symposium on Memory Management (ISMM)*, June 2002.

[4] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *International Symposium on Memory Management (ISMM)*, Oct. 2004.

[5] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM JavaGrande Conference*, June 1999.

[6] B. Cahoon and K. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java Controller. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2001.

[7] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1991.

[8] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1999.

[9] A. Diwan, E. Moss, and R. Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *ACM Conference on Programming language design and implementation*, June 1992.

[10] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

[11] G. Aigner and U. Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *European Conference on Object-Oriented Programming (ECOOP)*, July 1996.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] U. Hölzle. Optimizing Dynamically Dispatched Calls with Run-Time Type Feedback. In *Conference on Programming Language Design and Implementation*, June 1994.

[14] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *ACM Conference on Programming language design and implementation*, June 1992.

[15] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness With Performance. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1994.

[16] A. Hosking, J. E. Moss, and D. Stefanović. A Comparative Performance Evaluation of Write Barrier Implementations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1992.

[17] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[18] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, Apr. 2001.

[20] N. Sachindran, J. Eliot, and B. Moss. Mark-copy: Fast Copying GC with less Space Overhead. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2003.

[21] S. Soman, C. Krintz, and D. F. Bacon. Dynamic Selection of Application-Specific Garbage Collectors. In *International Symposium on Memory Management (ISMM)*, Oct. 2004.

[22] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[23] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *Conference on Programming Language Design and Implementation*, June 2003.

[24] D. Ungar. Generation scavenging: A non-disruptive high performance storage recalamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, Apr 1992.

[25] P. Wilson and T. Moher. A Card-Marking Scheme for Controlling Intergenerational References in Generation-Based Garbage Collection on Stock Hardware. *SIGPLAN Notices*, 24(5):87–92, 1989.