Efficient On-Stack Replacement for Aggressive Specialization of Java Programs

In submission, please do not distribute

Sunil Soman   Chandra Krintz

UCSB Technical Report #2004-24, September, 2004
Computer Science Department
University of California, Santa Barbara
{sunils,ckrintz}@cs.ucsb.edu

Abstract

On-stack replacement (OSR) is a technique used by dynamic and adaptive compilation systems to enable program specialization. Using OSR, an executing method can be recompiled (de- or re-optimized), and its runtime stack invocation frame dynamically replaced with that of the new version, whenever assumptions made for specialization are invalidated. Despite its potential, OSR is only used to a very limited degree in production Java Virtual Machines (JVMs). Two reasons for this are the limited forms of OSR-based specializations available and the restrictions on compiler optimizations (and hence execution performance) imposed by extant OSR designs.

In this paper, we address these limitations by extending an existing JVM OSR implementation so that it is more amenable to optimization. Moreover, our OSR version can be used to implement aggressive specializations that are invalidated by events external to the executing code. We present a novel specialization for write-barrier avoidance for generational garbage collection (GC) using our system that reduces program startup time by 6% on average. We also evaluate our approach for specialization in a JVM that implements multiple GCs in a single system and dynamically switches between them. Our OSR implementation reduces the overhead of this system by 6% on average. Finally, we show that our system has the potential to significantly improve performance if used to avoid the checks that guard speculatively inlined virtual method calls. We detail our OSR implementation and its empirical analysis in the open source Jikes Research Virtual Machine (JikesRVM) for Java from IBM Research.

1 Introduction

On-stack replacement (OSR), originally developed for Self-91 [9], is a technique that enables a method to be automatically replaced by the system *while it is executing*. In particular, the system replaces the runtime stack activation frame of the method with that of the new version, and continues execution at the same point within the new version. Past work has effectively employed OSR within dynamic compilation systems to enable optimized code to be debugged dynamically [14], to defer compilation of method regions to avoid compilation overhead (and improve dataflow) [9, 23, 11, 19], and to optimize (i.e. promote) methods that execute unoptimized for a prolonged period within a single invocation [15, 11, 19].

We are interested in using OSR for an aggressive specialization system that we are developing for Java. Our end goal is to enable significantly higher levels of performance than are available today from extant Java Virtual Machines (JVMs) [1, 19, 22], by customizing the VM for an executing application. Prior work in this area has shown that program performance can be significantly improved using application-specific garbage collection [21]. To enable efficient execution *and* the ability to switch between GC systems automatically, the VM in this prior work specializes the program for the underlying GC system (e.g. inlines allocation sites) and uses OSR (and method invalidation [1]) to replace methods if a GC switch occurs.

This prior work shows, however, that significant overhead is required to enable automatic GC switching – negating the benefits of the switching system in many cases. The primary overhead in this system is imposed by OSR: OSR limits optimization opportunity by not allowing optimization to occur *across* OSR points. This limitation is not specific to the GC switching system; it also is imposed by all prior approaches to OSR implementation [14, 11, 23]. For specializations for which invalidation can be triggered by an external event, e.g., GC switch or class loading, this limitation is particularly severe – resulting in very poor code quality – since OSR points are frequent (at every thread-switch point [2]).

To address this limitation, we present a new OSR implementation that is more amenable to optimization and that can be employed for both existing OSR uses, e.g., method promotion and debugging, and for uses for which OSR is triggered in response to an external event. Our implementation facilitates optimization through the use of a novel state recording system that is independent of the code (and so does not restrict dataflow) and that is updated incrementally by the compiler during optimization.

In addition to employing our OSR system to improve the performance of automatic GC switching, we also present two other aggressive, OSR-based, specialization techniques enabled by our implementation. The first is a novel technique in which we specialize code for JVMs that employ generational garbage collection, a popular GC type that enables good performance for a wide range of programs [24, 18]. Generational GCs segregate the heap based on object lifetimes and collect the younger generations independently of the older generations. One disadvantage of generational GCs is that they require write barriers for each pointer assignment. Write barriers are checks that record whether the assignment results in a reference

[1]Method invalidation is the process of recompiling and replacing *future* invocations of a method with a new version.
[2]We assume a single processor, multi-threaded JVM system.

from an older-generation object to one in a younger generation. Such references must be remembered and considered when the younger generations are collected (so that all live objects are correctly identified).

Our specialization relies on the observation that until there are objects in the older generation (i.e. a collection has occurred and objects have been promoted), write-barriers are unnecessary and introduce startup overhead. For programs that do not allocate enough to cause a garbage collection, write-barriers are pure overhead. We therefore optimize code without write-barriers. If a garbage collection occurs and objects are promoted to an older generation, we recompile the code to include write-barriers. We employ OSR to ensure that currently executing methods are replaced with correct versions.

Our second specialization is an OSR-based implementation of a well-known technique for improving virtual method dispatch performance, called guarded inlining [12]. Using this technique, if the compiler can determine the likely runtime type of the receiver object, it inlines the appropriate method. The compiler inserts checks (guards) immediately prior to the inline points that verify that the runtime object type is as expected. If it is not, control transfers to code that invokes dynamic dispatch. For our specialization, we omit the checks thereby improving the performance of methods for which compiler assumption holds. If class loading occurs that invalidates an assumption, we replace any executing methods using OSR.

Our OSR system is an extension of an implementation of OSR for deferred compilation [11]. In addition, we couple existing techniques for OSR implementation used in an automatic GC switching system [21] and in Self-93 for dynamic debugging of optimized code [14] with a novel state collection process that together enable improved code quality and efficient on-stack replacement of methods. We employ the state-of-the-art, open-source, Jikes Research Virtual Machine from IBM T. J. Watson Research Center as our implementation infrastructure. We detail our extensions and present results from our empirical evaluation of the performance of the system. Our results indicate that OSR introduces very little overhead and in combination with our specializations can enable significant performance improvements. In summary, **we make the following contributions with this paper:**

- The design and implementation of a freely-available, general-purpose OSR implementation that enables extant uses of OSR to be implemented as well as those for more-aggressive specialization (in which external events can trigger OSR).
- A set of techniques that enable OSR state to be extracted in the presence of compiler optimization.
- A novel specialization technique that enables methods to be optimized as if the underlying GC system was not generational (avoiding the overhead of write-barriers) and replaced when a GC

occurs and write-barriers are required. This technique improves program performance by 2-12% using a heap size of 500MB for the programs studied.

- An empirical evaluation of our system and specializations that includes measurements of all OSR overheads. We show that our OSR implementation improves the performance of an existing approach to an automatic GC switching system by 6% and has the potential to enable additional performance benefits (1-19%) when used for aggressive guard-free inlining of virtual methods.

2 Background and Related Work

On-stack replacement (OSR) [9, 15] is the process of (1) extracting the runtime state from a currently executing method, (2) recompiling a new version of the method, (3) generating a stack activation frame for the new version and updating it with the runtime values computed by the old version, (4) replacing the stack activation frame of the old version with that from the new version, and (5) continuing execution at the same point within the new version. Given OSR functionality, a dynamic compilation system can implement very aggressive specializations based on temporary conditions. Then, when these conditions change as a result of an external event or a runtime check, the compilation system can re-compile (and possibly re-optimize) the code and replace the currently executing version [14, 13, 11, 23, 19].

Invalidation is a related technique in which future invocations of a method use a different version of the code. Invalidation is much simpler than OSR since invalidation entails only that a correct version of the method is generated and that future invocations use the new copy. This is commonly implemented by replacing the address of the original version with that of the new version in a lookup table.

Both OSR and invalidation have been employed by extant dynamic compilation systems for dynamic debugging of optimized code [14], to defer compilation of method regions to avoid compilation overhead (and improve dataflow) [9, 23, 11, 19], and to optimize (promote) methods that executed unoptimized for a long time without returning [15, 11, 19]. The open-source Jikes Research Virtual Machine (JikesRVM) [1] from IBM T. J. Watson Research Center is one such system. JikesRVM uses OSR and invalidation for deferred compilation and method promotion. We extend this system in our work. We first describe the system and then present our extensions to it.

JikesRVM was designed to enable high-performance Java execution in server environments. The system dynamically compiles Java bytecode at the method-level, to x86 (or PowerPC) code. Upon initial invocation, each method is *baseline* compiled without optimization. JikesRVM also implements extensive runtime services (garbage collection, thread scheduling, synchronization, etc.) and an adaptive optimization system (AOS) [2]. The AOS samples the execution of programs to identify frequently executed, i.e., "hot", methods. The AOS optimizes (and performs any specialization on) hot methods at increasing levels of optimization; there are currently implements three levels of optimization [7]. To implement sampling, the system increments a counter for the currently executing method (regardless of how it was compiled) at each thread switch (approximately every 10ms). Thread-switch points occur at at method prologs, epilogs, and loop back-edges.

2.1 OSR and Method Invalidation in JikesRVM

To implement method invalidation, JikesRVM simply replaces the address of the method (located in a shared table in the VM) with that of a compilation stub when an assumption used to specialize a method is found to be invalid. Stubs are used in JikesRVM to enable lazy compilation of methods; all method entries in the system initially contain the stub address. When a method is invoked, the compilation stub is invoked instead which causes the *baseline* compiler to compile the target method. The stub then replaces its own address in the table with that of the compiled target method (for future target method invocations) and continues execution in the method. If the adaptive system later identifies the method has hot (and its optimization is predicted to be beneficial), the system will optimize the method as described above.

For currently executing methods that were specialized using now-invalid assumptions, JikesRVM replaces the method using OSR. To implement OSR, JikesRVM inserts an *OsrPoint* instruction into the application code at the point at which OSR should be performed. This instruction is implemented similarly to a call instruction: Execution of an *OsrPoint* causes the method's execution to be suspended and control to pass out of the current method into code that handles OSR.

The *OsrPoint* records the execution state of the method at a particular program point in native code. The execution state consists of values for bytecode-level local variables, stack variables, and the current program counter. The execution state is a map that provides the OSR system with runtime values at the bytecode-level (source-level) so that the method can be recompiled and re-started using another version. JikesRVM employs a stack resizing mechanism to allow each frame (from a recompiled method) to be correctly inserted into the stack. All methods that require OSR are replaced when OSR occurs.

An *OsrPoint* can be guarded by a check inserted into the application code, so that the *OsrPoint* is executed only when the check fails, e.g., for guarded inlining of virtual methods using a particular object type. Alternately, an *OsrPoint* can be inserted without a guard, e.g., for deferred compilation where it is inserted in place of code that has not yet been compiled. In either case, once an *OsrPoint* instruction is executed, the current method will be *unconditionally* replaced. The use of an *unconditional* instruction to initiate OSR requires that checks be added at every point at which OSR is to be performed *conditionally*, i.e. whenever some runtime system property changes. Consequently, OSR can be used only in a limited sense, since every check inserted into the code imposes a performance penalty.

The JikesRVM OSR implementation is similar to that used in other systems for similar purposes [14, 23, 13, 19], although different names are used for *OsrPoint*, e.g., interrupt points [14] OPC_RECOMPILE instructions [23], and uncommon traps [19]. The implementation, although simple, severely restricts compiler optimization. First, all method variables (locals as well as stack) are considered live at an *OsrPoint*; doing so artificially extends the live ranges of variables and can significantly limit the applicability of compiler optimization such as dead code elimination, load/store elimination, and copy/constant propagation. In addition, *OsrPoint* instructions are designated as *yieldpoints*, i.e., points at which control may be transferred out of the method to ensure that variable definitions are not moved around the *OsrPoints*. By "pinning" these instructions, the compiler is unable to perform optimizations across *OsrPoints*. Therefore, compiler insertion of many OsrPoints or of OsrPoints along the critical path has the potential to result in very poor code quality and thus, performance.

2.2 OSR for JikesRVM with Automatic GC Switching Functionality

Recently, Soman et. al proposed an extension to JikesRVM that enables application-specific garbage collection customization [21]. The system implements multiple garbage collection systems within a single JikesRVM execution image and can switch between them dynamically in an effort to improve performance. This prior work showed that such a system can produce significant performance improvements (11-14% for standard benchmarks and up to 44% for short running codes) over a system without dynamic GC switching functionality. However, these results were for a system that switched immediately prior to the start of an executing program. For such a configuration, the system can specialize the program (inline allocation sites and include or omit generational write-barriers as appropriate) without the need for OSR and method invalidation – since the system never switches *while* the program was executing.

This prior work presented preliminary results on the performance of switching between GC systems *automatically, after* the program has started. This configuration requires OSR and invalidation since there may be methods specialized for the previous GC. The authors of this work showed that the overhead imposed by the use of OSR is significant and in some cases can negate the benefits enabled by switching.

OSR is implemented in the GC switching system using an extension of the *OsrPoints* described above, called *OsrInfoPoints*. *OsrInfoPoints* are place-holder instructions inserted in the application code at GC safe-points (which are the points at which GC can occur – and a GC Switch may be performed). *OsrInfoPoints* are similar to *OsrPoints* except that they are removed from the code at the end of the compilation and do not appear in the final machine code. As such, they are not unconditional and do not require runtime checks. Since *OsrInfoPoints* (like *OsrPoints*) are intermediate instructions until the final machine code is generated, the compiler treats them as it does *OsrPoints*, i.e., it considers them "pinned" and it assumes that all variables are live, severely restricting compiler optimization.

Such an implementation (for *OsrPoints* or *OsrInfoPoints*) works reasonably well when there are only a small number of OSR points or when OSR is used for deferred compilation (since all variables must be considered live in the uncompiled code). However, any OSR-based specialization for which invalidation is triggered by an external event, e.g., GC switch, class loading, etc., requires that there be state recorded at every point in the method at which a thread-switch can occur (we assume a single processor system in this work). Since thread-switch points are commonly very frequent (method prologs, epilogs, call sites, loop back-edges, and exceptions) [19, 1], an instruction-based OSR state extraction mechanism has the potential for causing very poor code quality for frequently executed (optimized) methods resulting in poor program performance and possibly negating the benefits enabled by specialization.

3 Improving the Efficacy of On-Stack Replacement

Our extension to the JikesRVM OSR system enables OSR to occur an any point in a method at which a thread-switch can occur (in JikesRVM at method prologues, method epilogues, loop back-edges, call sites, exception throws, and explicit yieldpoints) – without incurring the performance penalty of prior

OSR implementations. Moreover, our approach does not require that explicit or place-holder instructions be inserted into the instruction stream to cause OSR. The key to our approach is an implementation of execution state collection called a *VAR_MAP* which maintains information similar to that in an *OsrPoint*, but which is independent of the application code.

A *VAR_MAP* is a per-method list of of bytecode variables that are live at each thread-switch point. This list is independent of the code and does not impact the live-ness information of the program point. To ensure that we maintain accurate information in the VAR_MAP, we update it incrementally as compiler optimizations are performed. Figure 1(a) shows an example of a VAR_MAP entry.

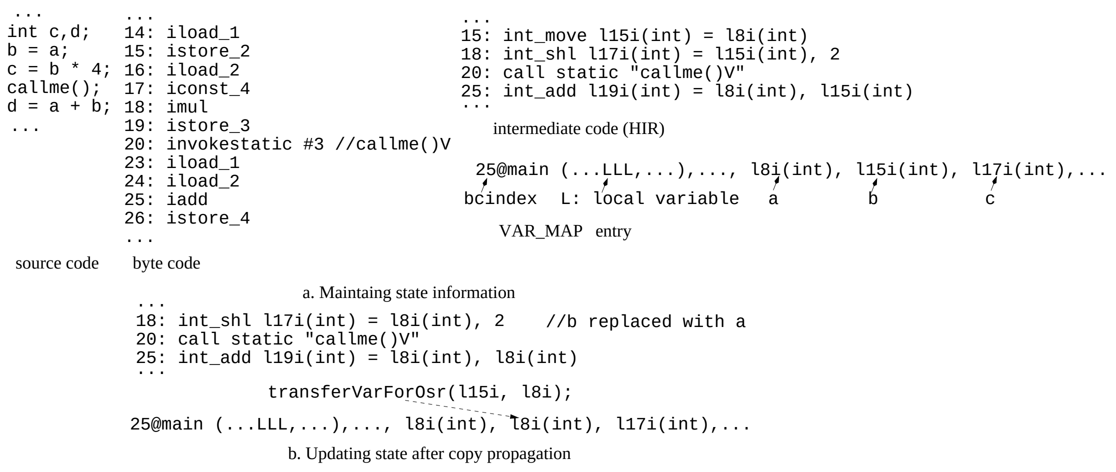
To update the VAR_MAP entries, we defined the following system methods:

- *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)
- *removeVariableForOsr (var)*: Record that *var* is no longer live/valid in the code. Note that, even though a variable may not be live, we must still remember it's relative order among the set of method variables.
- *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 copy and constant propagation, dead-code elimination (DCE), and escape analysis optimization. During copy and constant propagation, whenever a use of a variable (rvalue) is replaced by another variable (or constant), we use *transferVarForOsr* to record this in the VAR_MAP. Figure 1(b) shows a simple example of a VAR_MAP update following copy propagation.

We handle DCE and escape analysis similarly. When definitions of dead variables are removed, we need to ensure that if a dead variable is present in the VAR_MAP, it is replaced by its rvalue (using *transferVarForOsr*). If the definition uses more than one right-hand side variable, we record all uses along with the operators used to derive the lvalue (we currently only handle simple unary or binary operations). The JikesRVM optimization that employs escape analysis eliminates variable definitions for those that do not "escape" a method (or a thread). We replace such eliminated variables (as we do dead variables) with their rvalues in the VAR_MAP.

Similarly to [14], we are unable to ensure accurate recovery of state information for recursive method calls that have been optimized away using tail recursion elimination. This is because the call frame is completely eliminated for recursive frames.



```
 ...          ...
int c,d;     14: iload_1
b = a;       15: istore_2
c = b * 4;   16: iload_2
callme();    17: iconst_4
d = a + b;   18: imul
...          19: istore_3
             20: invokestatic #3 //callme()V
             23: iload_1
             24: iload_2
             25: iadd
             26: istore_4
             ...
```

```
...
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)
...
```

```
 25@main (...LLL,...),..., l8i(int), l15i(int), l17i(int),...
bcindex  L: local variable   a        b          c
```

```
...
18: int_shl l17i(int) = l8i(int), 2    //b replaced with a
20: call static "callme()V"
25: int_add l19i(int) = l8i(int), l8i(int)
...
             transferVarForOsr(l15i, l8i);

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

source code
byte code
intermediate code (HIR)
VAR_MAP entry
a. Maintaing state information
b. Updating state after copy propagation

Figure 1: (a) Shows how the VAR_MAP is maintained for a snippet of Java source (its bytecode and high-level intermediate representation (HIR) is included). We show the VAR_MAP entry for the *callme()* call site. The entry contains the next bytecode index (25) after the call site *callme*, and three local variables (*a: l8i*, *b: l15i*, *c: l17i*) along with their types. (b) Shows how the VAR_MAP is updated after copy propagation. Variable *b: l15i* is replaced with *a: l8i*.

We also update the VAR_MAP during live variable analysis. We record variables that are no longer live at each potential OSR point (i.e. each thread switch point), while still remembering their relative positions in the map. We set every variable that live-analysis discovers as dead, to a *void type* in the VAR_MAP. We identify local and stack variables by their relative positions in the Java bytecode. Maintaining the relative positions of variables in the VAR_MAP allows us to restore a variable's runtime value to the correct variable location. In addition, we load dummy (i.e. null) entries on to the the operand stack for dead stack operands since, although a dead stack variable will not be used after the current program point in the new version of the method, we must preserve the stack height and the order of variables on the stack for correct execution.

During register allocation, we update the VAR_MAP with the actual register and spill locations for the variables, so that they can be restored from these locations during on-stack replacement. The VAR_MAP contains symbolic registers corresponding to each variable. We update symbolic registers with a physical register or a stack location upon allocation by querying the map maintained by the register allocator for every symbolic register that has been allocated to a physical register. We record spilled variables via the spill location that the allocator encodes as a field in the symbolic register object.

Upon completion of compilation, we encode the VAR_MAP of the method using the compact encoding used for OsrPoints in the original system [11]. The encoded map contains an entry for each potential OSR point. Each entry consists of the *register map*, which is a bit map that indicates which physical registers contain references (which a copying 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 remains in the system throughout the lifetime of the the program and all other data structures required for OSR-aware compilation (including the original VAR_MAP) are reclaimed during GC.

3.1 Handling Special Cases

There are several cases that we must handle specially to enable correct state collection and to enable efficient code generation. These cases include call site return values, inlined allocation sites in the GC switching system, and machine-specific instruction selection and optimization.

One of the types of program points at which OSR can occur in an optimized method is the call site. Control transfers out of the method at the call site and returns to the program point immediately following the call when control transfers back to the method. Therefore, if OSR is required for the calling method, our system must enable execution to occur in the new version of the method at the instruction following the call. If the called method returns a value, the OSR value must correctly extract that value from the execution state so that it is available in the new version of the method.

Commonly, return values are stored in specific registers (as dictated by the compiler's calling convention). For example for the x86 architecture, JikesRVM stores the return value in registers *eax* and *edx* for optimized code. For such calling conventions, we restore the return value from the appropriate register into the correct local or stack value (in the execution state) during OSR. However, we cannot simply insert code into the specialized (original version of the) method after the call to record this state since these instructions will be skipped when control is transferred to the machine-code equivalent of the bytecode instruction that follows the method invocation bytecode. We, therefore, keep track of the return value type and the variable in which the value should be restored in the VAR_MAP. During OSR, we restore the value directly from the appropriate register(s).

We handle exception throws and yieldpoints as we do call sites. Moreover, we do not perform OSR for methods in their epilogues since no further processing within the method body will be performed.

In the GC switching system, allocation sites are inlined (if the class type has been resolved) using the allocation routines of the underlying GC system as part of the specialization. Since OSR can be triggered during garbage collection, we handle OSR for inlined allocation sites specially. In JikesRVM, an inlined allocation routine is divided into a *fast path* and a *slow path* [3, 5]. The fast path contains the actual inlined allocation sequence, however, the slow path is a call to a system allocation method which will cause a GC if memory is constrained. Control may transfer out of the application thread only at the slow path allocation call, however, we need to ensure that the OSR process will restore execution to the "correct" program point (i.e. the fast path), so that the new instruction is re-executed. To enable this, we update the OSR VAR_MAP program counter entry so that control is restored to the new instruction (and not the following bytecode instruction as is done for the call).

We also handle platform (x86) specific compiler transformations and optimizations. JikesRVM uses the BURS (bottom-up rewriting system) [7] to convert the intermediate code (after all optimizations have been performed on it) to final architecture-dependent code. BURS will encode floating point instructions on the Intel architecture, to make use of floating point registers. We need to update our *var map* to record stores of operands into the x86 floating point registers. We use *transferVarForOsr* for this purpose, passing in the operand that is to be stored along with the floating point register that it is to be stored in.

On the Intel architecture, the optimizing compiler tries to use the accumulator register for simple binary arithmetic and logic operations (e.g., ADD, SUB, MUL, OR, so on). We use *transferVarForOsr* to record any variable definitions that are being replaced by an implicit use of the accumulator register.

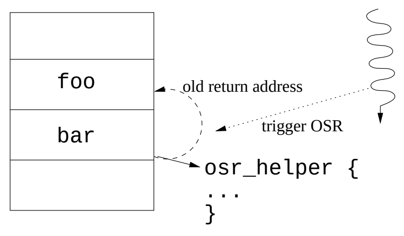
3.2 Triggering On-Stack Replacement

The other component required to enable on-stack replacement is the use of an appropriate OSR-triggering mechanism. There are two ways to trigger OSR (shown in Figure 2), depending on how OSR is to be used.

- Compiler-driven or Eager. The compiler inserts a call that is guarded by a condition that checks the validity of the specializing assumption to a system method that performs OSR. This is the approach employed by the JikesRVM OSR implementation.
- External or Lazy. OSR is triggered externally by a runtime event that renders a specializing assumption invalid. An OSR method is invoked by the runtime event which in turn either either patches the code of the executing method(s) with code that invokes the rest of the OSR process, or that reroutes the return address of each method that is the callee of a method to be OSR'd. The return is rerouted to invoke the rest of the OSR process. This approach does not require the insertion of conditional calls into the application code, nor does it require OSR to be performed on all methods

```
void foo(ptype1 param1, ptype2 param2, ...) {
 ltype1 local1 = new Object();
 bar();
 if(!specialization_cond_valid)   {
  //osr will be unconditionally triggered here
  OsrPoint this, param1, param2, local1, ...
  }
  ...
 }
```

a. OSR triggered by compiler inserted OsrPoint



b. OSR triggered lazily by external event

Figure 2: Triggering On-Stack Replacement. In (a), the compiler inserts a special OsrPoint instruction which is guarded by a condition. OSR will be triggered when this instruction executes. In (b), OSR is triggered *lazily* by an external event (thread) that will reset *bar*'s return address to "call" the *osr helper*.

(that require it) at once. However, it does require inspection of the runtime call stack. This approach was employed in Self-91 for debugging optimized code [14].

We extended JikesRVM to implement the lazy OSR trigger since it eliminates the need for runtime checks in the application code. To enable this, we modify the return address of the specialized method's callee so that it will jump to a special system method (a *OSR helper*) that performs OSR for the specialized method. This *OSR helper* method suspends the application thread, extracts the execution state from the specialized methods stack frame, and sets up the new stack frame. To preserve register values contained in registers for the execution of specialized methods, the *OSR helper* saves all registers (volatiles and non-volatiles) into its stack frame. Since the *OSR helper* is not directly called from the specialized code, we must "fake" a call to the *OSR helper*. This involves setting the return address of the *OSR helper* to point to the current instruction pointer in the specialized code upon entry to the *OSR helper*. This process also requires that we update the stack pointer for the *OSR helper* appropriately.

4 Aggressive OSR-Based Program Specialization

In addition to the empirically evaluating our OSR extensions for a GC switching system for JikesRVM, we also implemented two aggressive OSR-based specializations. The first is a novel specialization for generational garbage collection systems and the second is a specialization in which we remove the runtime checks that guard inlined call sites of de-virtualized methods.

4.1 Specialization for Generational Garbage Collection Systems

Generational garbage collectors make use of the weak generational hypothesis [18, 24] which states that most dynamically allocated objects (between 80 to 90%) have very short lifetimes. Consequently, objects can be segregated into separate parts of the heap, a *nursery* region for young (newly created) objects, and, a *mature space* for old objects. As objects age in the heap they are promoted from the nursery to the *mature space*. This separation enables nursery objects to be garbage collected more frequently than mature space objects.

Generation collectors require write barriers [25, 16, 4] (instructions in the execution stream) that record of references from mature objects to nursery objects. This record enables the nursery to be collected independently of the rest of the heap since it identifies nursery objects that are live solely because they are pointed-to by a mature object. Write-barriers are inserted into the application code by the compiler at every pointer store (i.e. putfields, array stores in Java bytecode) and thus, can degrade program performance. Researchers have found, however, that with intelligent write-barrier code, the benefits that result from maintaining separate generations outweigh the write barrier cost for many programs [4, 6].

However, for applications that do not allocate enough memory during their lifetime to cause a GC, e.g., short running programs or programs with small to medium dynamic memory requirements, the write-barrier cost (no matter how efficiently write-barriers are implemented) is pure overhead that can be significant. Such applications have become more plentiful as the cost of memory for modern processors has plummeted and larger and larger memory sizes have become available.

To address this limitation, we have developed a novel specialization technique in which we avoid inserting write-barriers until the first garbage collection commences. That is, we optimize code without write-barriers so that the program executes at full speed. If a GC occurs and objects are promoted to the mature space, our system invalidates methods that are optimized and require write-barriers and OSRs any such method that is currently executing. During compilation, we identify methods with pointer stores as possible OSR/invalidation candidates. We use this information to avoid needlessly processing methods without pointer stores.

Our compiler checks the heap size, heap residency, and whether any GCs have occurred to decide when to apply write-barrier specialization. The compiler does not perform this optimization when GC has occurred (objects have been promoted to the mature space) or when the maximum heap size is below 500MB (set arbitrarily but based on our experience with Java programs). Heap residency is the ratio of allocated pages to total pages in the system. When this ratio exceeds 0.6 (identified empirically), the compiler inserts write-barriers, i.e., does not specialize the method. Otherwise, the compiler optimizes the method without write-barriers and logs whether the method contains pointer stores. Only hot methods are optimized (and hence specialized) in our system using the adaptive optimization system of JikesRVM described previously. All unoptimized, cold methods have write-barriers inserted by the baseline compiler. Using this specialization, programs that perform no GC have no write-barrier overhead imposed on them; however programs that do exercise the GC system can have their startup time reduced and are able to reap the benefits of generational collection (with only minor OSR overhead imposed at the first GC).

4.2 Specialization for Virtual Method Dispatch

To investigate the generality of our OSR system, we used it to evaluate a well-known form of specialization for virtual method dispatch for object-oriented programs. Virtual method dispatch is a technique for dynamically binding method call sites to their implementations. The cost of executing virtual methods is higher than for static methods since the underlying object type must be looked up at runtime to find the method to dispatch. The authors in [17] overview many such techniques.

One such optimization is to specialize the caller code with (possibly inlined) statically dispatched version(s) of the call for the likely object type(s) [12, 15, 7]. To ensure that this code executes correctly, these systems insert checks (guards) into to the code just prior to the specialized code. The guard tests whether the runtime type of the object is that which was assumed. If the guard fails, another branch is taken that implements dynamic dispatch. Regardless of the correctness of the assumption, the guard is executed. This imposes overhead on the executing code that can be significant if guards are executed frequently. Efficient guard tests [17, 10, 8] and analysis techniques that identify when the guards can be omitted [10] have been proposed in the literature and employed in production systems to reduce this overhead.

OSR can also be used to remove guard tests. To evaluate its efficacy, we removed all guard tests for optimized methods that inline virtual methods that currently (given the classes loaded) have a single implementation. During class loading, we check to see if any loading event violates the assumptions used to remove the guards. When a violation occurs, we invalidate the method and perform OSR as necessary.

JikesRVM currently implements preexistence [10], a technique originally described for the HotSpot VM, in which guards are removed from methods when the compiler can determine that the receiver object cannot be of an as-yet-unknown class type. Preexistence precludes the need for OSR of executing method since it guarantees that once a method begins execution, the object type will not change. The method must be invalidated, however, so that future invocations of the methods are correct. OSR-based check-removal specialization is complementary to preexistence and enables additional checks to be removed. We extended the JikesRVM implementation of preexistence to enable OSR-based check removal and evaluated the efficacy of combining the two techniques.

5 Empirical Evaluation

To evaluate the efficacy of our OSR implementation, we measured the performance of our system for a number of different aggressive specializations (GC switching specialization, generational write-barrier removal, and guard-free dynamic dispatch). We first detail the experimental methodology we used for our experiments and then present our results.

5.1 Experimental Methodology

We gathered our experimental results using a dedicated 2.4GHz x86-based single-processor Xeon machine (with hyperthreading enabled) running RedHat Linux 9. We implemented our version of on-stack replacement into the JikesRVM version 2.2.0 with jlibraries R-2002-11-21-19-57-19 and the JikesRVM GC switching system described in [21] (and made freely available via the authors' web site).

We ran all of our experiments using the *adaptive optimization* JikesRVM system (described in Section 2). Since the hot-method selection process using this configuration is non-deterministic, the set of methods optimized and consequently, execution time across runs using the same input, shows significant variability. To reduce this non-determinism and to ensure that our results can be reproduced, we profiled each program off-line 100 times and collected the list of methods selected by the JikesRVM and their optimization level. We then composed the intersection set of these methods into a file which is read by the VM system at startup. The compiler consults this list of methods whenever a method is to be compiled, and if found, it optimizes the method directly at the indicated optimization level. This setup is similar to that used in other JikesRVM studies [20, 21]. and is called a *pseudo-adaptive* configuration.

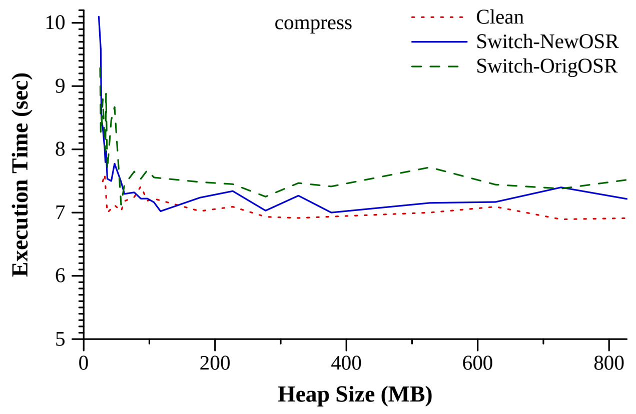
We execute each benchmark 10 times and turn off hot-method sampling for the final three runs. The values we report are the average of these last three runs. As such, the results we report for the clean system (without our extensions) contain no compilation overhead. For the experiments that exercise OSR and invalidation, the compilation time introduced by each *is* included in the results. We report total (first run) compilation overhead separately . We compiled the JikesRVM boot image methods at the highest optimization level and used the Generational Mark-Sweep (GMS) garbage collector in each system.

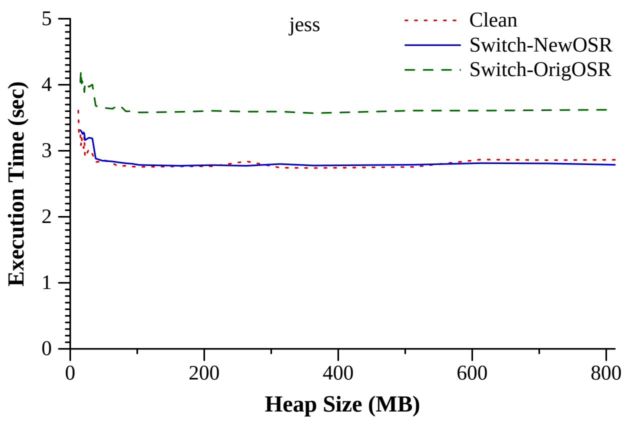
5.2 Results

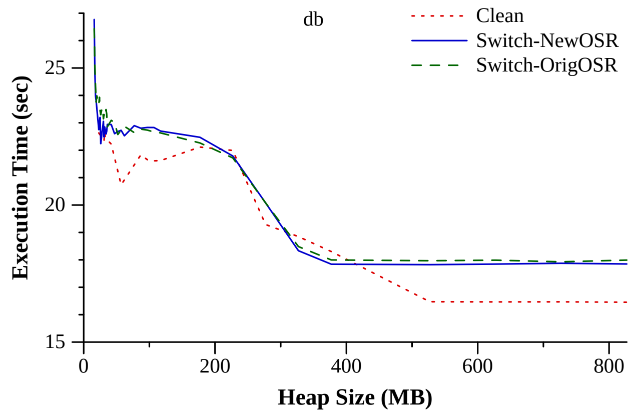
In Figure 3, we present the performance impact of our OSR system for the JikesRVM GC switching system (described in Section 2). The graphs in the figure compare the execution time performance for a clean version of JikesRVM system without the GC switching functionality (*Clean*), for the JikesRVM GC switching system without our extensions (*Switch-OrigOSR*), and for the GC switching system with our OSR extensions (*Switch-NewOSR*). All configurations use the generational mark-sweep (GMS) garbage collector and the switching systems never switches. The x-axis is heap size; the y-axis is time in seconds. The y-axis range for db and mtrt is a little over 10 seconds, as opposed to 5 seconds for the rest of the benchmarks.

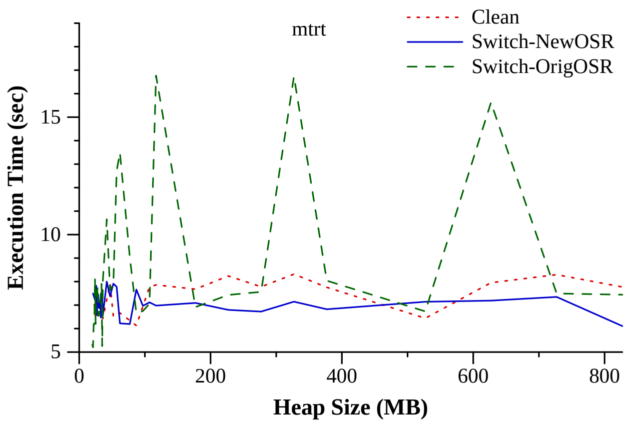
Although, the dynamic GC switching system has significant potential to outperform a VM built with a single garbage collector, its benefits are greatly reduced in the presence of support for on-stack replacement and invalidation, as reported in [21]. Our goal, therefore, is to improve the performance of the GC switching system when it *does not switch*, i.e. to obtain performance levels similar to that of a clean version of JikesRVM when both systems employ the same GC. This will enable the GC switching system to significantly improve performance when actually switching GCs at runtime. We therefore, compare our system to the clean system running the GMS collector, which is considered to enable the best performance in the JikesRVM, and to the original switching system with the same collector, without switching.

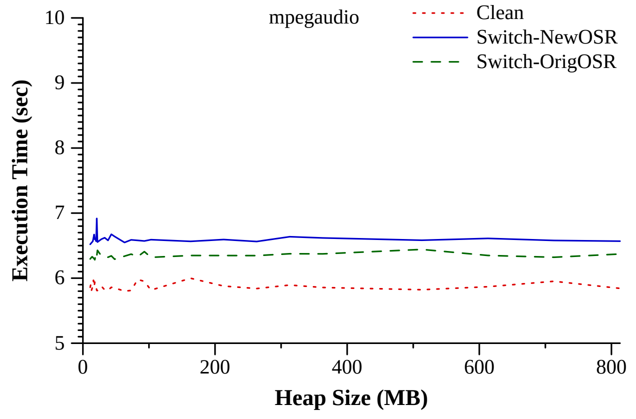
For most benchmarks and heap sizes, our system (Switch-NewOSR) outperforms the original switching system. Moreover, it enables much less performance variance across heap sizes. The most significant improvements are enabled for the jess, db, compress, and mtrt benchmarks. Over all heap sizes for jess, compress, and db, Switch-NewOSR improves performance by over 26%, 3%, and 1.5% on average, re-

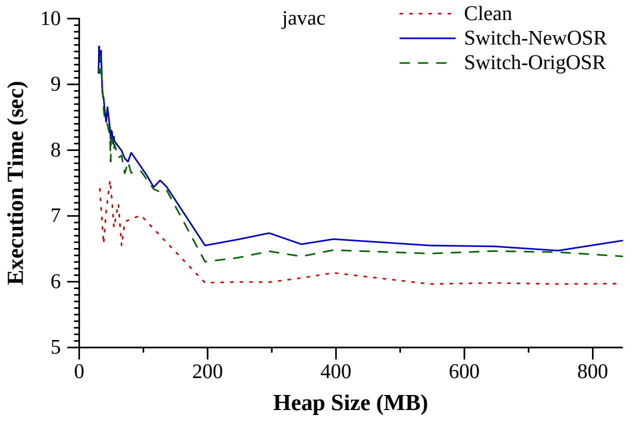


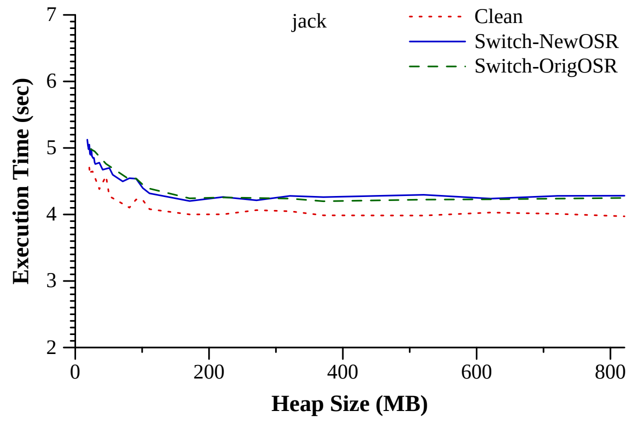












	Pct. Degredation	Pct. Improvement
	Switch-NewOSR	Switch-NewOSR
Benchmark	over Clean	over Switch-OrigOSR
compress	3.71%	3.22%
jess	3.06%	25.74%
db	3.09%	1.54%
javac	13.05%	-1.70%
mpegaudio	11.07%	-2.13%
mtrt	-1.80%	18.70%
jack	5.88%	1.0%
Geo. Mean	5.33%	6.16%

Figure 3: Execution performance of the baseline JikesRVM (Clean), the JikesRVM GC switching system using the original OSR version (Switch-OrigOSR), and the switching system using our general-purpose OSR implementation (Switch-NewOSR). The table summarizes the data (i.e. the impact of Switch-NewOSR) across heap sizes.

spectively. For the mtrt benchmark, Switch-NewOSR significantly reduces the performance fluctuation of the original switching system, improving overall average performance by 19%.

Switch-NewOSR performs similarly to the original switching system for javac, jack, and mpegaudio. At some heap sizes for these benchmarks, the original system performs slightly better than Switch-NewOSR. We believe that this is due to measurement variance (described in the previous subsection).

We also include a table in the figure that summarizes the impact of our new OSR implementation. Column 2 is the average (mean) percent degradation (Switch-NewOSR over Clean). Column 3 is the average (mean) percent improvement (Switch-NewOSR over Switch-OrigOSR). On average, the our system degrades performance of JikesRVM (without switching functionality) by 5%. However, it improves performance by 6% over the original switching system on average – and so it is a significant step toward our goal.

The results in Figure 3 measure the execution time for the benchmarks, that each of the systems enables. However, compilation time is also an important factor in the performance of a dynamic and adaptive compilation system. We present the compilation overhead introduced by our new version of OSR to the right in Table 4. We show data for the clean system (without GC switching functionality) and the Switch-NewOSR system. The data we present is the bytecode bytes per millisecond (BCB/ms) required for optimized compilation. The level of optimization is the same for both systems (and specified in the profile used as described in Section 5.1). On average across benchmarks, there are 12,193 bytecode bytes optimized. Since our system must track OSR state information at every thread switch point throughout the optimization process, it introduces 33% more overhead per bytecode byte over the clean system. On average across the benchmarks studied, this translates into an additional 440 milliseconds of overhead.

	BCB/ms			OSR Space Ohead	
Benchmark	Clean	NewOSR	% Degred.	(KB)	(KB)
compress	19.56	14.25	27.15%	4.50	3.71
db	11.15	7.00	37.22%	7.30	6.04
jack	15.47	9.10	41.18%	50.16	37.86
javac	18.75	10.75	42.67%	179.77	148.23
jess	11.15	7.64	31.48%	43.84	33.30
mpegaudio	17.90	10.94	38.88%	36.14	28.46
mtrt	8.51	7.57	11.05%	27.44	36.62
Geo. Mean	14.57	9.58	32.38%	42.20	36.44

Figure 4: Compilation overhead. BCB/ms is bytecode bytes per millisecond required for optimization of the hot methods. The final two columns show the compilation (collectable) and runtime space overhead, respectively, introduced by our system.

The final two columns of the table show the space overhead required for by our general-purpose OSR implementation. Column 5 shows the total space in KB required during compilation for all of the methods optimized in each program; this space is available for garbage collection once compilation completes for each method. This space consists of the un-encoded data structures that our system uses for the VAR_MAP. Column 6 shows the total space in KB that is kept in the system to enable OSR. This space consists of the compactly encoded OSR map that is used to map runtime values in a currently executing method to storage locations in the bytecode. This information is required to enable recompilation and execution continuation within a new version of an OSR'd method. On average, our system introduces 50KB of collectable space overhead and 42KB of permanent space overhead. The javac benchmark requires significantly more space overhead than the others due to a much larger number of optimized methods.

5.2.1 OSR-Based Specialization for Generational Write-Barrier Removal

We next present results for our OSR-based specialization for write-barrier removal for generational garbage collectors. For this specialization, we employ the popular JikesRVM Generational Mark/Sweep (GMS) collector as we did in the previous results. During compilation, the optimizing compiler checks the maximum heap size value to ensure that it is large enough to warrant specialization ($>=$500MB) and that the heap residency (ratio of total pages allocated) is low ($<=$0.6). In addition, the compiler checks whether any garbage collections have occurred and, if not, optimizes the code without write-barriers.

If a GC occurs (requiring write-barrier execution due to promotion of nursery objects), we perform OSR and invalidation to update optimized code with the correct version – both for future invocations and for currently executing methods. However, in case of OSR, we replace the currently executing specialized method with its baseline compiled equivalent. Consequently, OSR introduces only a very small runtime penalty. The compiler marks methods that have been specialized so that OSR and invalidation only occurs for those methods. Our goal with this specialization is to reduce the overhead of write-barriers for programs that require no garbage collection and to improve the startup performance of those programs that do.

We present our results in Table 1. We compare the performance of the Switch-NewOSR system (without switching) with and without write-barrier specialization using a heap size of 500MB. Column 2 and 3 show the execution time performance in seconds without and with the specialization (which we refer to as WBSpec), respectively. Column 3 shows the percent improvement enabled by WBSpec. The final two columns show the OSR overhead imposed (column 4 is the number of OSRs and column 5 is the total time

	Switch-NewOSR (w/o Switching)					
	w/o WBSpec	with WBSpec				
Benchmark	ET (secs)	ET (secs)	% Impr.	# WBs Elim	# OSRs	OSR Time (ms)
compress	7.39	7.24	2.03 %	1277	0	0.00
jess	3.09	2.88	6.80 %	7307524	7	3.29
db	17.01	15.87	6.70%	26851799	0	0
javac	6.54	6.41	1.99 %	2561614	1	0.60
mpeg	6.72	6.49	3.42 %	5464574	0	0.00
mtrt	6.70	5.99	10.60 %	2511488	0	0.00
jack	4.24	4.14	2.36 %	4974796	5	2.65
Em3d	1.07	0.98	8.41 %	1561718	0	0.00
MST	0.26	0.24	7.69 %	1744291	0	0.00
Perimeter	0.26	0.23	11.54 %	3170977	0	0.00
Geo. Mean	5.23	4.96	6.10 %	1810558	1.27	0.65
Geo. Mean Spec98	7.31	6.94	4.80 %	1716051	1.82	0.93

Table 1: Performance impact of write-barrier specialization for a heap size of 500MB using the GMS garbage collection system

for all OSRs). For many of the benchmarks, no OSR is required since 500MB is sufficient for each of the benchmarks. For those benchmarks that require OSR, the overhead is very small.

On average for the SPECJVM benchmarks (first 7 in the table) WBSpec improves performance by 5% on average. For benchmarks that require garbage collection, this 5% benefit occurs during the initial part of the program's execution, i.e., during program startup. We also include data for benchmarks from the Olden-Java Benchmark Suite (JOlden) in the table (Em3d, MST, and Perimeter). These benchmarks are very short running and require no garbage collection given a maximum heap size of 500MB – hence, these benchmarks are ideal candidates for write-barrier specialization. The average improvement in execution time for the JOlden benchmarks is 9% and 6% across all of the benchmarks in the table.

5.2.2 OSR-Based Specialization for Guard-Free Dynamic Dispatch

We next present results for removing inline guards from virtual methods. To protect against executing incorrect code, we employ OSR to invalidate code without checks when class loading occurs that causes invalidation of the assumptions made by the compiler.

Figure 5 shows the results. The figure presents the results for the three benchmarks that require inline guards; we also include jack input size 10 since it also uses a guard. The number of static guards removed is shown at the top of each bar. The y-axis is the percent reduction in execution time due to guard removal. In all of our experiments, we turn on preexistence (described in Section 4.2), which eliminates many of the guards. However, even with only a few static guards to remove, our system enables improvements of

19% and 16% respectively, for jess and mtrt. Removing only a single guard enables an improvement of 1-2% for the two inputs of jack.

Only three benchmarks required guards for inlined virtual methods. Moreover, class loading did not trigger OSR or invalidation. This is due to our experimental setup and the implementation of the benchmarks themselves. Since we average the last 3 of 10 runs, all classes are loaded during measured runs. Therefore, the compiler removes guards for methods that are guaranteed not to require OSR. Moreover, the benchmarks themselves do not implement a large number of virtual methods that are overridden. We plan to include other, more "object-oriented" benchmarks and identify other ways of measuring the impact of our system (yet that generate results with low variance that are repeatable) for the final version of this paper, should it be accepted. However, we believe that the results for mtrt and jess indicate the considerable potential of this specialization.

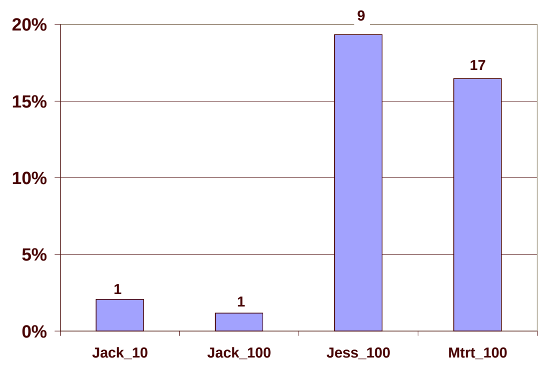


Figure 5: Pct. improvement due to OSR-protected inline guard removal. Above each bar is the static number of guards removed.

6 Conclusion

With this paper, we describe a novel implementation for on-stack replacement (OSR) for JikesRVM an adaptive optimization system for Java. Our version of OSR is amenable to optimization yet enables replacement of any executing method at any point at which a thread-switch can occur. Our implementation enables compiler optimization to occur across OSR points since, during optimization, we update a map that maintains the execution state necessary for OSR. We use the system to improve the performance of a GC switching system available for JikesRVM, by 6% on average. The switching system, without switching, is now within 5% of the clean system without the switching functionality.

We also show that our version of OSR is useful for implementing aggressive specializations. We present a novel OSR-based write-barrier specialization for generational garbage collectors that improves performance by an additional 6% on average. In addition, we use our system to implement and measure the potential of OSR-based, guard-free, aggressive inlining of dynamically dispatched methods. For the benchmarks with guards available for removal, our system enables performance improvements of 1-19%.

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.J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the jalapeño jvm. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2000.

[3] S. Blackburn, P. Cheng, and K. McKinley. A garbage collection design and bakeoff in jmtk: An effi cient extensible java memory management toolkit. Technical Report TR-CS-03-02, Department of Computer Science, FEIT, ANU, Feb 2003. `http://eprints.anu.edu.au/archive/00001986/`.

[4] S. Blackburn and K. McKinley. In or out? putting write barriers in their place. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2002.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java With MMTk. In *International Conference on Software Engineering*, Edinburgh, Scotland, May 2004.

[6] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *International Symposium on Memory Management (ISMM04), To appear*, October 2004.

[7] 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 Java Grande Conference*, pages 129–141, June 1999.

[8] B. Calder and D. Grunwald. Reducing indirect function call overhead in c++ programs. In *ACM Symposium on Principles of Programming Languages (POPL94)*, 1994.

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

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

[11] 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)*, March 2003.

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

[13] U. Hölzle. Optimizing Dynamically Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.

[14] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, June 1992.

[15] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness With Performance. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'94)*, 1994.

[16] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *SIGPLAN Notices*, Vancouver, British Columbia, October 1992. Association for Computing Machinery.

[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 *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, October 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 Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[20] N. Sachindran, J. Eliot, and B. Moss. Mark-copy: Fast copying gc with less space overhead. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 326–343, 2003.

[21] S. Soman, C. Krintz, and D. F. Bacon. Dynamic Selection of Application-Specifi c Garbage Collectors. In *International Symposium on Memory Management (ISMM04), To appear*, October 2004.

[22] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

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

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

[25] Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Notices*, 24(5):87–92, 1989.