Thin Guards: A Simple and Effective Technique for Reducing the Penalty of Dynamic Class Loading*

Matthew Arnold[1,2] and Barbara G. Ryder[1]

[1] Department of Computer Science, Rutgers University, Piscataway NJ 08855, USA
{marnold,ryder}@cs.rutgers.edu,
[2] IBM T.J. Watson Research Center, Hawthorne, NY, 10532, USA

Abstract. Dynamic class loading is an integral part of the Java™ programming language, offering a number of advantages such as lazy class loading and dynamic installation of software components. Unfortunately, these advantages often come at the cost of decreased performance because certain optimizations become more difficult to perform when an optimizing compiler cannot assume that it has seen the whole program. This paper introduces *thin guards*, a simple but effective technique that uses lightweight runtime tests to identify regions of code within which speculative optimizations can be performed. One application of thin guards is described in detail, demonstrating how they can be used to perform speculative inlining in the presence of dynamic class loading. Our experimental evaluation shows that when used in combination with other traditional compiler optimizations, thin guards can eliminate most of the penalty of dynamic class loading. Performance improvements of up to 27% are observed, eliminating up to 92% of the penalty imposed by dynamic class loading.

1 Introduction

Dynamic class loading [20] is an integral part of the Java programming language, offering a number of advantages such as lazy class loading and dynamic installation of software components. Unfortunately, these advantages often come at the cost of decreased performance because certain optimizations become more difficult to perform when an optimizing compiler cannot assume that it has seen the whole program.

Object-oriented languages encourage data encapsulation through the use of methods, resulting in frequent method invocations. Additionally, object-oriented languages support dynamically dispatched (virtual) calls, where the method called depends on the runtime type of the receiver object. Efficient implementations of virtual dispatch [12,15] help reduce the direct overhead of virtual method invocation, however, *method inlining* remains an important optimization for effective implementation of object-oriented languages.

* Funded, in part, by NSF grant CCR-9808607.

One technique for performing inlining at potentially polymorphic call sites is *guarded inlining* [7, 14, 17, 18] (also called *receiver class prediction* [14] and *guarded devirtualization* [18]), where a runtime test is inserted to check whether is safe to execute the inlined code; if not, a full virtual dispatch is invoked. Guarded inlining allows *some* of the benefits of direct inlining, but not all. The two main disadvantages are 1) executing the guard incurs runtime overhead, and 2) optimization of the caller method is restricted because of the possibility that the default virtual dispatch will be executed. Code patching [8,24] is a technique that removes the overhead of executing the guard itself, but does not address the second problem of restricted optimization.

Interprocedural analysis, such as *class hierarchy analysis* (CHA) [10,13], can help eliminate guards by identifying call sites that are provably monomorphic (have only one possible target method) and therefore can be directly inlined without a guard. However, dynamic class loading precludes the use of traditional interprocedural analysis because classes loaded in the future may cause the result of the analysis to become incorrect. Call sites will be referred to as *currently-monomorphic* if traditional analysis identifies them as monomorphic, but dynamic class loading may cause them to become polymorphic in the future.

Several existing techniques have addressed the goal of allowing direct inlining of currently-monomorphic call sites in presence of dynamic class loading. Such techniques include on-stack replacement [16], preexistence-based inlining [11], code patching [8,24], and extant analysis [23]. As discussed further in Sections 2 and 9, these techniques either 1) do not completely remove the penalty of dynamic class loading, or 2) have implementation complexities that limit their use in mainstream JVMs. The latter is made evident by the fact that many VM's [1,8,11,24] continue to perform guarded inlining at currently-monomorphic call sites.

This paper introduces *thin guards*, a simple but effective technique that uses lightweight runtime tests to identify regions of code within which speculative optimizations can be performed, even without the ability to perform on-stack replacement. One application of thin guards is described in detail, demonstrating how they can be used to perform speculative inlining in the presence of dynamic class loading. Thin guards are almost as easy to implement as traditional guarded inlining, yet offer two substantial advantages: thin guards are 1) more efficient (generally one machine load) and 2) test a more general condition, making it easier to guard multiple inlined call sites with a single test. Our experimental evaluation shows that when used in combination with other traditional compiler optimizations, such as control flow graph splitting [7], thin guards can eliminate most of the penalty of dynamic class loading.

The remainder of this paper is organized as follows. Section 2 reviews existing techniques for inlining in the presence of dynamic class loading. Section 3 presents a motivating example demonstrating the benefits of thin guards. Section 4 describes thin guards in full generality, while Section 5 shows one application of thin guards using class hierarchy analysis. Sections 6 and 7 describe our implementation and experimental results using the Jikes Research Virtual

Machine.[3] Section 8 describes applications of thin guards other than inlining in the presence of dynamic class loading. Sections 9 and 10 describe related work and conclusions.

2 Background: Dynamic Class Loading

Dynamic languages such as Self [7] and the Java programming language make it difficult, or even impossible, to determine the set of components that comprise the whole program. For example in Java, `Class.forName` can be used to load a class whose name is computed at runtime. Optimizations performed in the presence of such semantics must be conservative to ensure that all code produced will execute properly regardless of what classes are loaded in the future.

In practice, however, programs may execute for long periods of time without performing dynamic class loading. There may be many applications for which class loading occurs during program startup, and then either 1) never occurs again, 2) occurs only infrequently during software updates, 3) occurs frequently, but affects only a portion of the class hierarchy. For these types of programs it can be profitable to optimize *speculatively* (or *optimistically*) to obtain maximum performance, and rely on potentially expensive invalidation mechanisms to ensure safety when class loading occurs. Direct inlining at currently-monomorphic call sites is an example of such an optimization.

One approach for invalidation is to simply recompile any methods that contain unsafe code after class loading occurs. Although this will ensure that all future calls execute safely, currently running methods may be executing unsafe code, and may continue to do so indefinitely. Therefore, without a technique such as *on-stack replacement* to allow recompilation of currently executing methods, optimizations must be conservative to ensure safety in the presence of dynamic class loading.

The remainder of this section describes existing techniques for safely performing inlining of currently-monomorphic call sites.

2.1 Guarded inlining

Guarded inlining is one solution for safely inlining currently-monomorphic call sites. It is relatively easy to implement and is used by many systems to avoid incorrect execution in the presence of dynamic class loading [1, 8, 11, 18].

There are two common types of guards. A *class test* [7, 14] checks the type of the receiver object, while *method test* [11] checks which method will be called by the receiver. The advantages and disadvantages of each are discussed further in Section 3.1.

[3] Jikes RVM is an open source version of the Jalapeño Research Virtual Machine [1, 4]. Documentation and source code are available at `http://www.ibm.com/developerworks/oss/jikesrvm`

2.2 On-stack replacement

On-stack replacement or *dynamic deoptimization* [16] is a technique that allows a currently executing method (i.e., on the activation stack) to be stopped and replaced by another version of that method, presumably optimized in a different manner, or possibly not optimized at all. By allowing currently executing methods to be deoptimized, on-stack replacement is an effective mechanism for performing invalidation of unsafe code. In fact, Hotspot [21] uses on-stack replacement to allow direct inlining of currently-monomorphic methods.

However, there are disadvantages associated with on-stack replacement. It is a fairly complex technique that can require space consuming data structures [6,16]. Additionally, optimizations may need to be restricted to maintain the ability to later perform on-stack replacement [6]. The bottom line is that few JVMs today perform on-stack replacement; to our knowledge, Hotspot [21] is the only publicly described JVM to do so. An implementation of on-stack replacement in Jikes RVM is currently underway, although not yet complete.

2.3 Preexistence-based inlining

Preexistence-based inlining [11] is a technique that enables direct inlining at *some* currently-monomorphic call sites, without requiring on-stack replacement. The goal of preexistence is to identify call sites at which the receiver object must have been created before the caller method was invoked; thus the receiver object *preexists* the method invocation. If the receiver of a currently-monomorphic call site is guaranteed to preexist the method invocation, the call site can be directly inlined.

When dynamic class loading occurs, all invalid methods are recompiled so new calls to those methods execute safely. Currently executing methods are safe because receiver objects at directly inlined call sites preexist their caller's invocation. This implies that the receiver object also preexists the dynamic class loading that occurred, ensuring that the type of the receiver object cannot be one of the newly loaded classes.

Preexistence is a simple and effective technique for identifying candidates for direct inlining, and is complementary to the work presented here. The main limitations of preexistence-based inlining is that it is effective for only a subset of the currently-monomorphic call sites. As discussed further in Section 7.5, substantial performance can be gained by directly inlining the remaining call sites not covered by preexistence. Additionally, we show that the effectiveness of preexistence is reduced as more aggressive inlining techniques are applied.

2.4 Code Patching

Code patching [8, 24] is a technique used to eliminate the runtime overhead of executing guards. Inlining is performed just like traditional guarded inlining, where a guard is inserted during optimization and a backup virtual dispatch is included in case the guard should fail. However, with code patching the instructions to implement the guard are not included in the final generated code and the inlined body executes unconditionally. If the inlining decision is later invalidated by dynamic class loading, the first instruction of the inlined code is dynamically overwritten with an unconditional jump to the backup virtual dispatch.

Code patching is more efficient than traditional guarded inlining because it eliminates the overhead of executing the guards themselves. However, code patching does not address the restrictions placed on optimization by the presence of the backup virtual dispatch. As discussed in the next section, the performance penalty caused by restricted optimization can be as great, or greater than the penalty imposed by executing guards.

3 Motivating Example

This section presents a motivating example to convey the basic idea behind thin guards. Consider the partial program shown in Figure 1. Assume that currently there are no subclasses of `A` or `B`. Methods `getX()` and `getY()` are not `private` or `final` so dynamic class loading could cause them to be overridden at some point in the future; therefore, these methods cannot be inlined directly in method `bar()`. Even though they are currently-monomorphic, the directly inlined code could become incorrect if either method is overridden in the future.

Figure 2 shows how method `bar()` would be optimized using traditional guarded inlining. A method test is used to guard the inlined body of `getX()` because the receiver (`a`) could be of type `A` or `B`. A class test is used to guard the inlined body of `getY()` because the only possible type for the receiver (`b`) is `B`. Neither guard can be identified as redundant and eliminated because each method is dispatched on a different receiver object. Preexistence-based inlining would not help because the objects referenced by `a` and `b` may have been created after method `bar()` was invoked.

However, as long as no class loading occurs, directly inlining these calls would be safe; therefore, both guards in `bar()` could be replaced by a single test that checks whether any class loading has occured since `bar()` was compiled, as shown in Figure 3. This simple example illustrates the key idea behind *thin guards*: using a boolean condition bit to represent program-wide conditions. This technique decreases the overhead of the guards themselves, and also allows more optimization to occur because a larger area of code can be covered by a single guard.

3.1 Comparison of guards

Table 3.1 summarizes the relative strengths and weaknesses of thin guards compared to class tests and method tests. The first advantage of thin guards is the efficiency of the guard itself. Checking a condition bit can be performed with a single load, whereas methods tests typically require two [11] or three [2] loads,

```
class A {

  public int bar() {
     A a = getSomeA_or_B();
     B b = getSomeB();
     return a.getX() + b.getY();
  }

  public int getX() { return 99; }
  public int getY() { return 100; }
}

class B extends A {
}
```

Fig. 1. Example program fragment. Assume no classes override classes `A` or `B`.

```
void bar() {
   a = ...;
   b = ...;
   if ( METHOD_TEST(a,getX()) )
     t1 = 99;    // Inlined body
   else
     t1 = a.getX();
   if ( CLASS_TEST(b,B) )
     t2 = 100;   // Inlined body
   else
     t2 = b.getY();
   return t1 + t2;
}
```

Fig. 2. Method `bar()` from Figure 1 optimized using traditional guards (method and class tests).

```
void bar() {
   a = ...;
   b = ...;
   if (noClassLoadingHasOccured)
     return  199;
   else
     return a.getX() + b.getY();
}
```

Fig. 3. Method `bar()` from Figure 1 optimized using thin guards.

Table 1. Comparison of the advantages and disadvantages of each guarded inlining technique. The code in the column "Example" assumes the class hierarchy from Figure 1, and that variables `a` and `b` are of static type `A` and `B` respectively.

Characteristic	Example	Class Test	Method Test	Thin Guard
Runtime overhead of typical implementation (# loads)	—	1 (or 2)	2 (or 3)	1
Can one guard cover multiple call sites dispatching different methods on the same receiver object?	`b.getX()` `b.getY()`	Yes	No	Yes
Can one guard be used at call sites where multiple receiver types are possible, but call the same method?	`a.getX()`	No	Yes	Yes
Can one guard cover multiple call sites dispatched on different receiver objects?	`a.getX()` `b.getY()`	No	No	Yes
Effective for polymorphic call sites (assuming distribution is peaked).	—	Yes	Yes	No

depending on the implementation.[4] Similarly, class tests typically require one or two loads.

Choosing between a class test or a method test involves balancing tradeoffs because each has limitations. Class tests offer the advantage that multiple dispatches on the same receiver object can be guarded by a single test, even if the calls are to different methods; this is not true for method tests. However, a single method test can be used for call sites where multiple receiver types are possible, as long as all of these types result in a call to the same method; this is not the case for class tests, which would need to test each potential type individually.

Thin guards will handle both of the these scenarios with a single guard. Furthermore, thin guards can allow multiple calls to be covered by a single test *even if the calls are dispatched from different receiver objects*, as demonstrated in Figure 3 where the two calls on receiver objects `a` and `b` are both guarded by a single test.

The main disadvantage of thin guards (as discussed later in Section 4) is that they are effective only for currently-monomorphic call sites. Class and method tests can be effective for polymorphic call sites because they examine the receiver object of each dispatch. Thin guards, however, test whether a condition is true for *all possible receivers*, and therefore cannot be used to distinguish receivers at polymorphic call sites. It is for this reason that thin guards are most effective for infrequently changing, program-wide conditions, such as dynamic class loading.

[4] An additional load is necessary in Jikes RVM because the *type information block* (TIB) can potentially be moved by a copying garbage collector. See [2] for more details on dynamic type checking in Jikes RVM.

3.2 Effect of Code Patching

Code patching [8,18] (see Section 2.4) is a technique that can be used to reduce the overhead of executing the guards, and is orthogonal to the particular guard implementation being used. Code patching can be used with all of the guards shown in Table 3.1, including thin guards. Augmenting each guard type with code patching would reduce the runtime overhead (the first row of Table 3.1) to zero. However, code patching does not relax any of the restrictions that guards place on optimization, and thus does not affect the other characteristics shown in Table 3.1.

4 Thin Guards

Thin guards are lightweight runtime tests designed to efficiently monitor *optimistic assumptions* about the executing program. For the purpose of this paper, an optimistic assumption is some fact about the executing program that 1) is currently true, 2) is unlikely to change in the near future, and 3) enables additional optimization to be performed. For example, "Method `A::getX()` is not overridden" is an optimistic assumption. By providing a mechanism to efficiently verify optimistic assumptions, thin guards can be used to create regions of code within which these assumptions can be relied upon to perform more aggressive optimization.

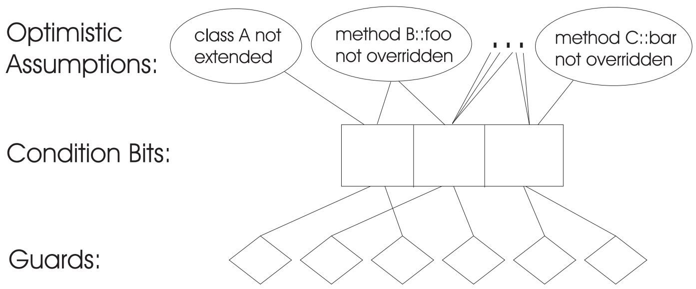

Fig. 4. The general structure of thin guards. Optimistic assumptions are mapped to condition bits, which are referenced by guards throughout the application.

The general structure of thin guards is shown in Figure 4. Optimistic assumptions are mapped to some number of *condition bits* that are used to record whether the optimistic assumptions are still true.[5] There may be more optimistic assumptions than condition bits, so multiple assumptions may map to the same

[5] Although referred throughout the paper as *condition bits*, the notion of a *bit* is abstract. The actual implementation of the conditions could use any representation that is most convenient and efficient.

bit. A condition bit is true only if *all* of its optimistic assumptions are true. A condition bit must be set to false if any of its optimistic assumptions become false.

During compilation, optimizations that may benefit from relying on an optimistic assumption can be performed as long as the region of code is guarded by a test that checks a condition bit associated with that optimistic assumption. The guards in Figure 4 represent conditional tests compiled into code throughout the application. Each of these guards tests a single condition bit to determine whether the optimistic assumptions mapped to that bit are still true.

This mechanism of mapping optimistic assumptions to condition bits serves two purposes.

1. **Efficiency.** An optimistic assumption can be tested by reading a single bit. Because there is a small number of condition bits, checking a bit can most likely be implemented with a single load.
2. **Generality.** Mapping several optimistic assumptions to a single condition bit allows multiple assumptions to be tested with a single guard, reducing the number of guards that need to be executed. The example in Figure 3 demonstrated this by using single condition bit to verify that neither method (`A::getX()` nor `A::getY()`) was overridden.

The main disadvantage of thin guards is that they are effective only for assumptions that can be mapped to a condition bit. Program-wide characteristics that change infrequently fit most naturally into this model. Dynamic class loading is the primary application discussed in this paper, however, other potential applications are presented in Section 8.

4.1 When Optimistic Assumptions Change

After a program has executed for some time, optimistic assumptions may change. When they do, all condition bits associated with the false optimistic assumptions must be set to false. Guards testing these condition bits will fail so execution will continue safely, but more slowly. To recover performance, the VM can choose to recompile all methods necessary to remove failing guards. Such an approach is not specific to thin guards. A system using traditional guarded inlining would likely choose to recompile certain methods after class loading occurs, to remove method or class tests that have begun failing.

However, thin guards introduce additional complexity because multiple optimistic assumptions may be hashed to the same condition bit. A guard G that reads a condition bit B becomes dependent on all B's optimistic assumptions, even if G was originally inserted to test only a subset of those assumptions. For example, the code from Figure 3 used a single bit (`classLoadingHasNotOccured`) to cover all inlined call sites. If some obscure class `Q` is loaded that does not extend `A` or `B`, the guard in `bar()` will begin failing even though the inlined code is still valid.

For ease of reference, some terminology will be established. A guard G is *intentionally-dependent* on an optimistic assumptions O if G was inserted specifically to test O. G is *unintentionally-dependent* on O if G was not inserted to test O, but becomes dependent on O only because multiple assumptions map to the same condition bit.

When a condition bit becomes false, guards reading that bit can be classified into two categories

1. A guard is *permanently-disabled* if it is intentionally-dependent on an optimistic assumption that has become false. Permanently-disabled guards must never succeed or invalid code would be executed.
2. A guard is *temporarily-disabled* if it is *not* permanently-disabled, but is reading a false condition bit. Temporarily-disabled guards are safe to succeed at any time, but fail because of an unfortunate mapping to the condition bit.

Additionally, any method that contains a permanently-disabled guard will be referred to as an *unsafe method*. The false condition bit cannot be reset to true while unsafe methods are executing, otherwise permanently-disabled guards could succeed and incorrect execution may result.

In terms of regaining performance it is desirable to re-enable the condition bit as soon as possible so that temporarily-disabled guards can begin succeeding again. There are several possible strategies that could used to recover performance. Below we discuss a recompilation-based approach, as well as a code patching approach.

Recompilation A failing condition bit can be re-enabled once all permanently-disabled guards dependent on that bit have been eliminated. To accomplish this, the VM could eliminate all unsafe methods as follows:

1. Recompile all unsafe methods to remove permanently-disabled guards so that new calls to these methods go to safe code. Alternatively, unsafe methods could be marked for lazy compilation [19] so that future calls trigger a new compilation of that method.
2. Wait for all currently executing unsafe methods to exit. This could be implemented in a variety of ways, one of which is to periodically scan the stack (possibly during garbage collection when the stack is already scanned) to see if unsafe methods are still active.

Once no unsafe method can possibly execute, the condition bit can be re-enabled; temporarily-disabled guards will begin succeeding again and performance will no longer be degraded. However, it may be the case that unsafe methods continue executing for a long period of time. During this time, temporarily-disabled guards throughout the application must fail to ensure safety. If the unsafe methods continue executing for an unacceptable period of time, the VM can use a backup plan to recover performance. A new condition bit could be introduced that tests only those optimistic assumptions that are still true; all methods that contain temporarily-disabled guards can then be recompiled to take advantage of the new condition bit.[6] This strategy is undesirable in the case that the unsafe methods exit quickly (because then the additional recompilation is not necessary), but can be used to prevent performance from being degraded indefinitely in extreme cases.

Code Patching Although the previous section presented a recompilation-only strategy for recovering performance, code patching can be used to simplify the process. When a condition bit becomes false, code patching can be used to replace all permanently-disabled guards with an unconditional jump to the backup path. Once the patching occurs, the condition bit can be reset so that the temporarily-disabled guards succeed once again.

In fact, code patching can also be used in combination with thin-guards to remove the guards altogether, similar to traditional code patching as described in Section 2.4. The combination of thin-guards and code patching would yield higher (peak) performance than either technique used in isolation: code patching eliminates the direct overhead of executing guards, while the thin guards approach of mapping multiple optimistic assumptions to a single guard creates larger regions of check-free code, thus reducing the indirect overhead overhead due to restricted optimization. When optimistic assumptions change, permanently-disabled guards would be be patched to be unconditionally false, while temporarily-disabled guards would be left alone (and thus would remain unconditionally true).

5 Example Application: Inlining

This section describes one particular application of thin guards, showing how they can be used to allow more effective inlining in the presence of dynamic class loading. The goal is reduce the overhead introduced by dynamic class loading, allowing performance to approach that which could be obtained if dynamic class loading did not exist.

5.1 Using thin guards with CHA

For our example application, class hierarchy analysis (CHA) [10] is used to identify currently-monomorphic call sites. Inlining at such call sites is guarded using thin guards to ensure proper execution if the class hierarchy should change in the future.

For simplicity, a single condition bit is used to represent all optimistic assumptions, similar to the example from Figure 3. However, using a single condition bit does not imply that the bit must represent *all possible* class loading (like the `noClassLoadingHasOccurred` bit did in Figure 3). Instead, optimistic

[6] The previous condition bit can remain false indefinitely, or potentially be reused if the unsafe methods ever exit.

assumptions can be selected carefully to minimize the chance of invalidation. Only parts of the class hierarchy that are relatively stable (i.e., have not been extended for some amount of time) should be associated with the condition bit. Methods in classes that are extended frequently can still be inlined with traditional techniques. Using this selective approach, even applications that perform frequent class loading can benefit from thin guards as long as some portion of the class hierarchy is stable.

The following steps can be followed during optimization, and when class loading occurs, to safely achieve the benefits of thin guards.

During optimization During compilation the optimizing compiler can make optimistic assumptions about call sites using class hierarchy analysis. To inline a call to method `A::foo()` using thin guards, the compiler needs to perform the following steps.

1. Use class hierarchy analysis to confirm that the call to `A::foo()` is currently-monomorphic and therefore a candidate for direct inlining.
2. Let O represent the optimistic assumption "`A::foo()` is not overridden". If O is already mapped to the condition bit, goto step 5.
3. Decide whether class `A`, or more specifically method `A::foo()`, is stable enough to justify using a thin-guard (i.e., has not been extended for a sufficient amount of time). If this is not the case, use traditional guarded inlining and skip steps 4–6.
4. Insert O into the set of optimistic assumptions mapped to the condition bit. If class loading later violates O, then the condition bit must be set to false.
5. Record that the method being compiled (M) contains a guard that is intentionally-dependent on the optimistic assumption O. This allows M to be identified as an unsafe method if O should become false.
6. Guard the inlined code with a conditional that tests the condition bit.

At this stage, a separate guard is inserted for each call site, with the assumption that redundant checks can be eliminated at a later stage. Section 5.2 describes techniques for increasing the amount of code covered by each guard.

During class loading When class loading occurs, the following steps must be performed before the loaded class is made available.

1. Compute the set of optimistic assumptions that were violated by the classes that were loaded. If no assumptions were violated, no further action is necessary.
2. For each optimistic assumption that is now false, compute the set of unsafe methods. This is just a lookup of the information recorded in step 3 during optimization. If there are no unsafe methods, no further action is necessary.
3. Change the condition bit to false so that all thin guards begin to fail.

At this point, the loaded class can be made available and execution will continue safely, but with degraded performance because guards will be failing. Performance can be regained using the techniques described in Section 4.1.

5.2 Eliminating redundant guards

One of the main advantages of thin guards is that they test more general conditions than traditional guards, allowing a larger region of code to be covered by a single guard. The algorithm described in 5.1 placed one guard per call site. This section describes how to safely identify guards that are redundant and can be removed.

Guard removal becomes slightly more complex in a multi-threaded environment, so the single-threaded case is considered first. In a single-threaded environment, redundant guards can be eliminated by using a slight variant of conditional constant propagation [25] to forward propagate condition bit values. The value of the condition bit can be assumed to be true along the "true" path after being tested by a thin guard, and false along the "false" path. These values can be propagated forward and guards that can be proven to test a constant condition can be removed.

The only special treatment necessary is to account for the fact that class loading may change the value of any condition bit; therefore, the analysis needs to ensure that all instructions that may cause dynamic class loading, either directly or indirectly, kill the condition bit values being propagated. In our implementation (as discussed further in Section 6.2) calls are among the instructions that act as thin guard barriers because interprocedural analysis is not performed to identify methods that are guaranteed not to perform class loading.

Multiple threads In a multi-threaded environment removing guards becomes more difficult because while one thread (`T1`) is executing, a second thread `T2` could load classes at any time. In such a scenario, none of the guards in `T1` can safely be removed.

Fortunately, this problem can be solved by adding logic to the class loader. Section 5.1 described the actions taken by the class loader to trigger invalidation. In a multi-threaded environment a 4th step can be performed by the class loader before making the loaded classes available:

4. Wait for all threads to exit any region of code that is guarded by a false condition bit.

With this addition, guards can be removed as in the single-threaded case; by the time the newly loaded class is made available, all execution will have exited the unsafe regions, and unsafe regions cannot be re-entered because the condition bit has been set to false.

The question is how can this 4th step be implemented? It is actually quit easy in any system that uses some variation of *safe points* such as *interrupt points* in Self [16], or *yieldpoints* in Jikes RVM [1]. Safe points are simply program locations at which the VM can preempt the currently executing thread; they are typically placed on method entries and loop backedges to ensure that only a finite amount of execution can occur between them. Given such safe points, step 4 can be implemented by

– During redundant guard elimination, ensure that guards are not eliminated across safe points (i.e., safe points kill during the propagation of condition bit values).
– During class loading, wait for all threads to reach a safe point before making the newly loaded class available.

Together, these two conditions ensure that all threads will have existed unsafe code by reaching a safe point, therefore class loading can continue. Making safe points barriers to guard elimination may increase the number of guards executed; however, if safe points are placed at method entries and loop backedges, guard removal can still be quite liberal. The entire body of a loop, or the entire body of a loop-free method can potentially be covered by one guard (assuming the loop or method does not contain other barriers to guard elimination).

Increasing opportunities for guard elimination Although the previous section described how to identify guards that are redundant and can be eliminated, it is easy to construct simple code fragments containing guards that cannot be eliminated. For example, consider the code fragment from Figure 2. Even if the method test and class test are replaced by thin guards, the second guard cannot automatically be removed. The true and false paths of the first guard merge before reaching the second guard, therefore the condition bit must be retested. However, several well known transformations can help increase the opportunities for removing redundant guards.

Splitting Splitting [7] is a transformation originally designed to reduce the overhead of message sends in Self. Splitting exposes optimization opportunities by specializing sections of code within a method. Specialization is achieved by performing tail-duplication of conditional control flow to eliminating control flow merges where data flow information would have been lost.

For the example from Figure 2, splitting can be used to duplicate the second guard (along with the code it protects) by placing a copy along each path of the first guard. After splitting, both copies of the second guard can be identified as redundant, producing the code in Figure 3. Splitting is effective for exposing a variety of optimization opportunities, but is particularly effective for thin guards, as discussed further in Section 7.4.

Inlining Calls are likely to be barriers for guard elimination; eliminating guards across a call would require interprocedural analysis to prove that the dynamic class loading could not occur during at any point during the call. Aggressive inlining will reduce the number of (non-inlined) call sites, thus reducing interference with guard removal.

Loop transformations Loop transformations, such as loop cloning [7], or loop unrolling, can be used to expose opportunities for removing redundant guards. Consider a tight loop that executes only a single call site that is inlined with a guard. The guard cannot simply be removed; the condition bit must be tested at some point to ensure that class loading has not occured. However, by unrolling the loop N times it may be possible for guards 2–N to be identified as redundant on the first guard.

5.3 Multiple condition bits

Our example application uses a single condition bit to represent all optimistic assumptions. This approach has advantages besides that of simplicity. Having all guards test the same condition bit maximizes the number of guards that can be identified as redundant and eliminated. As long as class loading does not occur, a single condition bit offers the highest possible performance.

The disadvantage of using a single condition bit is recovery cost, because all thin guards in the program will begin failing when any optimistic assumptions become false. Performance is not recovered until the temporarily-disabled guards can be re-enabled as described in Section 4.1. Invalidation is expected to occur infrequently, although it could be the case that this potential temporary performance degradation is undesirable or even unacceptable.

However, in scenarios where temporary performance degradation is unacceptable, it is likely that on-stack replacement is also not a viable solution. With on-stack replacement the program must *stop completely* until all unsafe methods have been recompiled and replaced on the stack. With thin guards, the failing guards allow execution to continue safely, but more slowly, until the unsafe code has been recompiled and is no longer executing. Better yet, if code patching is used in combination with thin guards (see Section 4.1) then the only cost of recovery is the time needed to patch the permanently-disabled guards.[7]

However, in cases where code patching is not being used *and* the potential performance degradation associated with using a single condition bit is too high, multiple bits could be used to reduce the number of guards that become temporarily-disabled when class loading occurs. There are many possible ways to incorporate multiple condition bits, each of which has its own advantages and disadvantages. We mention only a few examples below.

One possibility is to allocate one condition bit per optimistic assumption. This would ensure that guards are never temporarily-disabled because guards would fail only if their own optimistic assumption becomes false. Unfortunately, this approach reduces the opportunity for identifying and eliminating redundant guards; guards that were previously identified as redundant may now be testing different condition bits.

Another extreme is to allocate a separate bit for each method that is compiled. All optimistic assumptions relied upon during compilation of a method could be mapped to the condition bit for that method. This ensures that guards within a method all test the same condition bit, maximizing the possibility of removing redundant guards within the method. Additionally, each method has

[7] Methods containing permanently-disabled guards (which have been patched to be permanently false) will not regain their performance until they are recompiled and re-entered, but all other methods will execute at full speed once the patching completes.

its own condition bit, so class loading will disable guards only in methods that are legitimately affected. However, the disadvantage of this approach is that increasing the number of condition bits will eventually degrade performance due to decreased cache behavior of reading the bits.

A more practical strategy would be to use a fixed number of bits that is small enough to provide desirable cache behavior, yet large enough to help reduce the performance penalty incurred when classes are loaded. Such a scenario creates many new potential strategies for deciding which optimistic assumptions map to which condition bits, with the goal of maximizing guard removal. However, we did not explore this approach because combining code patching with thin guards is a simpler, yet effective option. By using code patching, temporarily-disabled guards are no longer an issue, so the recovery performance and scalability concerns associated with the single bit strategy are no longer a concern.

6 Implementation

To validate the ideas presented in this work, we implemented thin guards in the Jikes Research Virtual Machine. Section 6.1 provides a brief overview of Jikes RVM and Section 6.2 describes details of the implementation.

6.1 Jikes RVM

Jikes RVM [1] is an open source Research Virtual Machine developed at IBM T.J. Watson Research Center. Jikes RVM is written almost entirely in Java, and takes a compile only approach (no interpreter). Methods are compiled with a non-optimizing compiler upon their first execution, and an aggressive optimizing compiler [5] is applied selectively by the adaptive optimization system [4]. Although not a fully complete JVM, the performance of Jikes RVM has been shown to be competitive with that of commercial JVMs on the PowerPC platform.

Jikes RVM has the ability to perform inlining based on static heuristics, as well as adaptive inlining based on a call-edge profile collected via time-based sampling [4]. Direct inlining (without a guard) is performed when inlining monomorphic `private` and `final methods`, as well as for call sites proven monomorphic by local type propagation. Preexistence-based inlining [11] based on *invariant argument analysis* [11] is also performed to eliminate guards when possible. All other inlined methods are guarded with either a class test or method test.

Our version of Jikes RVM also performs some optimizations not yet available in the open source version, including feedback-directed splitting [3] and loop unrolling.

6.2 Thin guards implementation

Our thin guards implementation is the example application described in Section 5.1; class hierarchy analysis is used to identify candidates for direct inlining, and a single condition bit is used to represent all optimistic assumptions regarding dynamic class loading. To prevent invalidation from occurring unnecessarily, our implementation does not insert thin guards until a method has reached the highest level of optimization ("O2" in Jikes RVM). Traditional guards are used for all lower levels of optimization. This approach allows the application to execute for some time before any optimistic assumptions are made. Using this technique, no invalidation occurred for any of the benchmarks used in this study.

Redundant guard elimination To identify and eliminate redundant thin guards, a separate optimization pass was added to the Jikes RVM optimizing compiler. The elimination pass is not performed until after inlining, splitting, and loop unrolling have been performed, to allow the elimination to benefit from the effects of those transformations. However, the elimination is performed fairly early in the optimization process to allow the remaining optimizations to benefit from the simplified control flow.

Jikes RVM's *yieldpoints* were used as the mechanism to ensure safety in a multi-threaded environment, as described in Section 5.2. Jikes RVM's yieldpoints are simply a small sequence of instructions that checks a timer-set bit to determine whether the currently executing thread should yield to the thread scheduler. Yieldpoints are placed on all method entries and backedges to ensure that only a finite amount of execution can occur before a yieldpoint is executed.[8]

Yieldpoints act as a barrier to guard elimination to ensure safety in a multi-threaded environment, as described in Section 5.2. Additionally, all instructions that can potentially cause dynamic class loading also act as a barrier to guard elimination;[9] this includes all calls, because interprocedural analysis is not performed to identify methods that are guaranteed not to perform dynamic class loading.

Class loader modifications When class loading causes an optimistic assumption to be violated, the newly loaded class is not made available until all threads have executed at least one yieldpoint (see Section 5.2). In Jikes RVM this is achieved by stopping all threads at the next executed yieldpoint.[10] After all

[8] Jikes RVM also places yieldpoints on method epilogues to improve the accuracy of the time-based samples collected by the adaptive optimization system. This placement is not necessary for correctness.

[9] The following bytecode instructions have the potential to cause a class to be loaded: anewarray, checkcast, getfield, getstatic, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, multianewarray, new, putfield, and putstatic. However, in many cases it can be identified that class loading cannot occur. For example, getstatic cannot cause class loading once the class of the static field being accessed has been loaded.

[10] The current implementation simply triggers a garbage collection, which automatically stops all threads the next yieldpoint when using one of Jikes RVM's *stop-the-world* collectors.

threads have stopped, the thin guard condition bit is set to false and execution continues.

All unsafe methods are marked for recompilation to remove all permanently-disabled guards. Our current implementation does not have the capability to identify when all active unsafe methods have finished executing (are no longer on the stack), although this functionality is currently being implemented. Therefore, to safely recover performance after class loading, our current implementation immediately employs the "backup" mechanism described in Section 4.1; a new condition bit is introduced and all methods containing temporarily-disabled guards are marked for lazy compilation so they will be recompiled on their next invocation. As mentioned previously, invalidation never occurs during the execution of our benchmarks, although the invalidation mechanism and recovery process is fully implemented.

7 Experimental Evaluation

This section describes our experimental evaluation of the thin guards implementation described in Section 6.

7.1 Methodology

Our experimental evaluation investigates the effect of thin guards on the SPECjvm98 benchmarks suite [9]. A standard *autorun* execution of the SPECjvm98 benchmarks consists of n executions of each benchmark in the same JVM. For our experiments, n was chosen separately for each benchmark to allow all benchmarks to run for approximately 4 minutes (using the size 100 inputs). This experimental setup was used to ensure that each benchmark executed long enough for the adaptive optimization system to approach a steady state. The time for the best run was chosen as the representative time for each autorun sequence. The goal of this study is to evaluate inlining effectiveness, so the startup performance of Jikes RVM is of no particular interest.

The Jikes RVM adaptive optimization system is nondeterministic, and sometimes this nondeterminism causes visible performance differences, even when comparing two runs of the same configuration. To eliminate noise, the final time used for each benchmark was computed by collecting 10 representative times (described above) and taking the median.

All experiments were performed on a 500 MHz 6 processor IBM RS/6000 Model S80 with 4 GB of RAM running IBM AIX 4.3.2. Jikes RVM was run using 1 processor and a 400 MB heap.

7.2 Benchmark characterization

Figure 5 characterizes these benchmarks by showing the rate at which they execute virtual call sites (`invokevirtual` bytecodes). All call sites (both inlined and not inlined) were instrumented to record dynamic execution frequencies. These execution frequencies were then divided by the running time of the benchmark to compute executions per second. Call sites were divided into 5 categories.

1. Non-inlined virtual dispatch.
2. Guarded inlining of call site that CHA identifies as potentially polymorphic.[11]
3. Guarded inlining of a call site that CHA identifies as monomorphic.
4. Direct inlining performed using preexistence.
5. Direct inlining performed without needing preexistence (callee was `private`, `final`, or belongs to a `final` class).

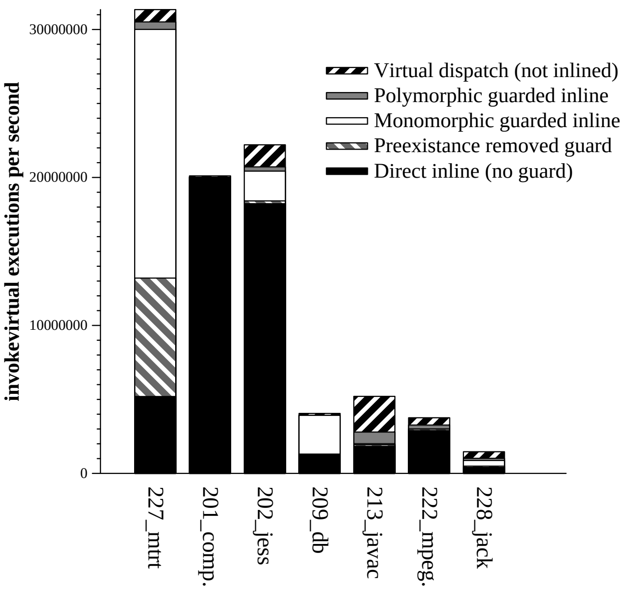

Fig. 5. Execution rate of invokevirtual call sites (both inlined and not inlined).

Our thin guards implementation can affect only the third category, shown in white in Figure 5. `_201_compress` `_213_javac` and `_222_mpegaudio` have virtually no such dynamic calls; therefore, it can be expected that thin guards will have no effect for these benchmarks. Many of the benchmarks have a large portion of their call sites directly inlined (the solid black bar). As an experiment, the benchmarks were also included in the study in a modified form, where the `private` and `final` modifiers were removed from all methods.[12] This creates

[11] These call sites must have been inlined by the profile-driven adaptive inlining because the static heuristics inline only call sites that are monomorphic according to CHA.

[12] For benchmarks whose source was not available, Jikes RVM was modified to ignore the `final` and `private` modifiers when making inlining decisions. This will not catch

the most challenging scenario possible for the optimizing compiler because all methods in the program may potentially be extended.

This is clearly a somewhat artifical modification because the `private` and `final` are language constructs that can be used to enforce information hiding and avoid undesirable software updates. However, programmers are often afraid of the overhead associated with data encapsulation, and specify methods `private` or `final` solely for the purpose of improving performance (allowing them to be directly inlined by the optimizing compiler), or even worse, avoid using data encapsulation altogether. This practice defeats the point of having dynamic class loading as part of the language, and would occur less frequently if programmers trusted their optimizing compiler to provide more competitive performance. Our goal for evaluating the modified benchmarks is simply to see how much these language constructs affect performance, and how much of this performance penalty can be regained by using our technique.

7.3 Effectiveness of thin guards

To evaluate the effectiveness of thin guards, timings were collected using the following 3 configurations of Jikes RVM:

1. **Base**: This configuration is the original Jikes RVM performing its full suite of optimizations [1,5]. Guarded inlining is performed using traditional method tests and class tests. Relevant optimizations performed include preexistence-based inlining and local type propagation (to eliminate guards whenever possible), adaptive inlining [4], feedback directed splitting [3] and loop unrolling.
2. **Thin guards**: Identical to `Base` but thin guards are used for currently-monomorphic call sites when methods reach the highest optimization level (O2).
3. **Ideal:** Guards are removed (unsafely) from all inlined call sites that are currently-monomorphic, thus representing the performance that could be achieved if dynamic class loading did not exist.[13] Incorrect execution can occur if classes are dynamically loaded.

Figure 6 shows the performance of these three configurations. The black bar shows the performance improvement of `Thin guards` over `Base` (higher bar means more improvement over `Base`), and the full height of the bar represents the upper bound on performance using the `Ideal` configuration. The number above each bar represents the percentage of the `Ideal` improvement that is achieved

all cases because the Java to bytecode compiler may have performed direct inlining while compiling the bytecodes.

[13] It should not be assumed that a JVM implementing on-stack replacement (see Section 2.2) would safely achieve the performance of Ideal. Optimization may need to be restricted to support on-stack replacement, resulting in degraded performance relative to Ideal. There is currently little published data quantifying the degree to which these potential optimization restrictions affect performance in practice.

by using thin guards. This number is not meaningful (and thus not reported) for benchmarks with `Ideal` performance less than 2%.

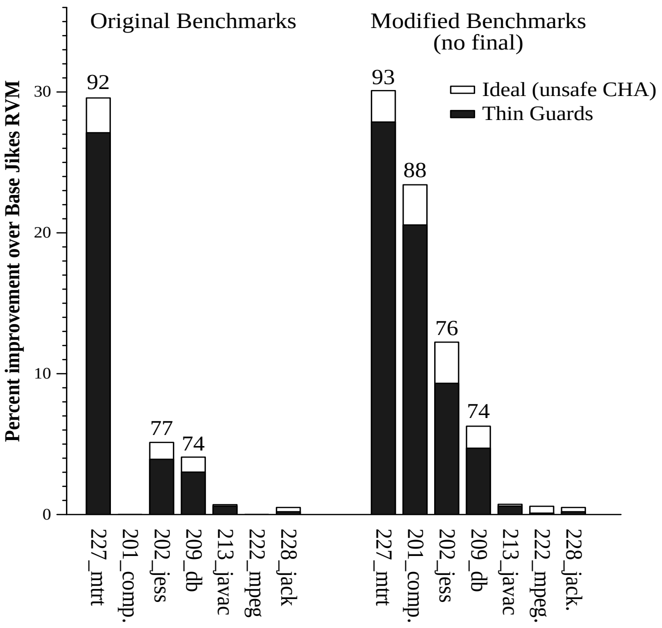

Fig. 6. Performance improvement of thin guards. The full bar shows the `Ideal` improvement using (unsafe) class hierarchy analysis. The black bar shows the performance improvement obtained safely using thin guards. The number above each bar shows the percent of the `Ideal` improvement that was achieved using thin guards.

The benchmarks showing the most substantial `Ideal` improvements are `_227_mtrt` (in its original form) and the modified version of `_201_compress`. `_202_jess` and `_209_db` show reasonable improvement, particularly `_202_jess` in its modified form. These improvements are mostly consistent with the breakdown of invocation types previously shown in Figure 5.

The total performance obtained by `Ideal` is *not* a contribution of this paper; it is simply an observation of the potential performance that could be obtained using well-known techniques, assuming dynamic class loading did not exist. The contribution of this paper is the percentage of `Ideal` performance that is obtained using thin guards. For `_227_mtrt` and the modified `_201_compress`, thin guards achieved 92% and 88% of the `Ideal` performance. `_202_jess` and `_209_db` showed reasonable performance, achieving over 70% of the possible `Ideal` win. The authors were pleasantly surprised by the substantial fraction of `Ideal` performance that could be obtained by using thin guards, and that such a simple technique could eliminate the majority of the penalty imposed by dynamic class loading.

7.4 Importance of splitting

The results of Figure 6 were achieved using a system that performs feedback-directed splitting of intraprocedural control flow paths. As discussed in Section 5.2, splitting exposes optimization opportunities allowing additional guards to be identified as fully redundant and removed. This section takes a closer look at the interactions between thin guards and splitting. To evaluate the importance of splitting, the experiments of the previous section were repeated using the identical version of Jikes RVM, but with splitting disabled. The results are shown in Figure 7. Splitting was disabled for all 3 configurations shown in Figure 7, so the results show how much performance improvement can be expected when thin guards are added to a system that does not perform splitting.

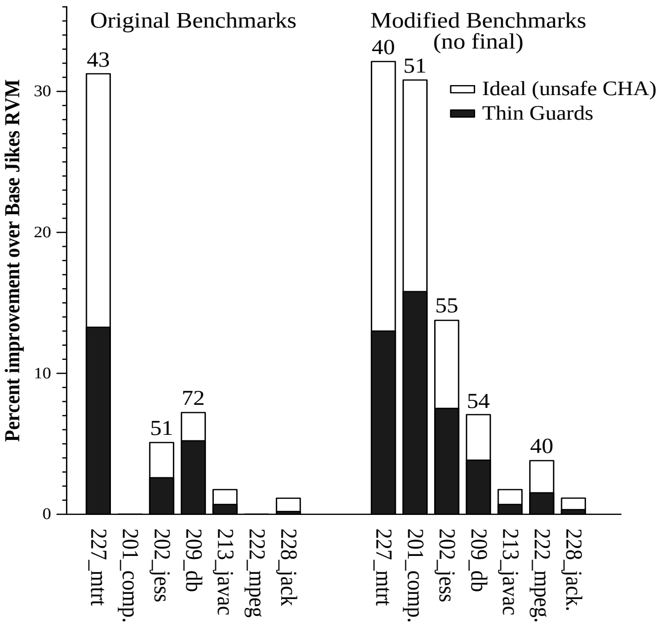

Fig. 7. Performance of thin guards in VM that does not perform splitting. Data is collected identically to that of Figure 6, but all 3 configurations (`Base`, `Thin guards`, and `Ideal`) were run with splitting disabled.

Thin guards are clearly less effective when splitting is not performed, achieving roughly 50% of the `Ideal` win in most cases. It may be tempting to conclude from this data that thin guards did not actually provide the performance improvements previously reported in Figure 6, but that splitting is responsible instead; however, this is not the case. Recall that splitting was used in *all three* configurations of Figure 6, including the base configuration. Traditional guards often cannot be eliminated because of the limitations discussed in Section 3, *even* *when splitting is applied.* It is the synergy between splitting and the more general condition tested by thin guards that accounts for the increased performance of Figure 6.

Clearly, thin guards provide increased motivation for incorporating some form of splitting into the optimizing compiler. However, even without splitting the performance improvements offered by thin guards can be substantial (13.5% for `_227_mtrt` and 17% for the modified version of `_201_compress`), particularly given the implementation simplicity of thin guards.

7.5 Effectiveness of preexistence

Preexistence-based inlining is implemented in Jikes RVM and was included in the `Base` configuration for our experiments. This section makes two observations about the effectiveness of preexistence when used without thin guards.

The first observation involves the bottom line effectiveness of preexistence-based inlining. The first row of Table 2 (labeled "Orig. RVM") shows the effect of preexistence (based on invariant argument analysis) on the `_227_mtrt` benchmark. The runtime is improved from 10.3 seconds to 8.6 seconds, a 16.2% improvement. However, the ideal performance is 5.9 seconds, meaning that preexistence achieved roughly a third of the ideal win (42.2%), leaving plenty of performance opportunity remaining for techniques such as thin guards.

Table 2. Effectiveness of preexistence-based inlining for `_227_mtrt`, both with and without aggressive adaptive inlining. "Orig. RVM" is an unmodified version of Jikes RVM. "No A.I." stands for "No Adaptive inlining", and represents a version of Jikes RVM with adaptive inlining disabled.

	Performance (seconds)			Dynamic Counts (millions)		
	Without	Argument	Ideal	Guarded	Unguarded Inlined	
	Preexistence	Preexistence	(unsafe CHA)	Inlined	Non-Preex	Preex
Orig. RVM	10.3	8.6 (-16.2%)	5.9 (-42.2%)	643	83	532
No A.I.	10.6	8.5 (-19.2%)	7.0 (-34.0%)	500	27	706
Difference	—	—	—	-143	-56	+147

Our second observation involves the interaction between aggressive inlining and preexistence. By default, Jikes RVM performs adaptive inlining using profiling information to aggressively inline hot call edges. Although effective at improving performance for most programs, adaptive inlining has consistently, and mysteriously, degraded the performance of `_227_mtrt` (previously reported in [4]). Upon further investigation, the problem turned out not to be a deficiency of the adaptive inlining implementation, but a fundamental characteristic of preexistence-based inlining.

Preexistence relies on method boundaries as a mechanism for performing invalidation of unsafe code without the need for on-stack replacement. As more aggressive inlining is performed, methods become larger and method invocations become less frequent; the result is that fewer objects preexist the most recent method invocation, and preexistence-based inlining becomes less effective.[14] Consider an extreme example where the entire program is inlined into `main()`. The call to `main()` will be the only call executed during the program, so no objects can preexist this call;[15] therefore, no inlined call sites will be candidates for direct inlining based on preexistence.

To confirm that this effect occurs in practice, the row labeled "No A.I." in Table 2 repeats the measurements from the previous row but using a version of Jikes RVM with adaptive inlining disabled. Adaptive inlining improves performance in both scenarios where preexistence was *not* used (columns "Without preexistence" and "Ideal"), but degrades performance when preexistence is used (column "Argument preexistence").

The final three columns of the table provide dynamic inlining counts to confirm our hypothesis. The first of the three columns reports the dynamic number of call sites that were executed with a guard, while the remaining two columns represent call sites that were inlined directly. The directly inlined call sites are broken into two categories to distinguish whether or not the guard was removed by preexistence.

As expected, disabling adaptive inlining for `_227_mtrt` decreased the number of guarded inline sites (-143 million) and also decreased the number of call sites directly inlined for reasons other than preexistence (-56 million). However, disabling adaptive inlining *increased* the dynamic number of call sites that were directly inlined due to preexistence (+147 million), confirming that preexistence was hindered by adaptive inlining.

These observations are not meant to suggest that preexistence-based inlining is not an effective technique for improving performance. Preexistence is a simple idea that offers large speedups in many cases. Even though adaptive inlining's negative effect on `_227_mtrt` is related to preexistence, both configurations *with* preexistence are much faster than either configuration *without* preexistence. Our goal is only to argue that preexistence cannot be relied upon to eliminate all of the performance penalty of dynamic class loading, particularly in systems that perform aggressive inlining.

8 Other Applications

Although this paper focused on performing inlining in the presence of dynamic class loading, thin guards are general mechanism for allowing speculative optimizations to be performed safely without the need for on-stack replacement. One obvious extension is to use analyses other than class hierarchy analysis to identify currently-monomorphic call sites. The only requirement for incorporating analyses with thin guards is the ability to identify the set of call sites that become invalid after a class has been loaded.

[14] True for preexistence in general, not just invariant argument preexistence.

[15] Except possibly objects created by static initializers.

Thin guards also have potential for testing other infrequently changing conditions not related to dynamic class loading. For example, thin guards could be used to reduce the performance penalty of non-final static fields. Guards could be inserted to create regions of code within which optimizations can exploit the values of infrequently changing static fields. Any code modifying the value of such fields would need to trigger invalidation. Other examples include using thin guards to remove synchronization for single-threaded programs by testing whether more than one application thread has been created. Similarly, thin guards could be used to keep track of whether reflection has been used; if not, then it is safe to apply optimizations that take advantage of private fields (which can be modified using reflection).

9 Additional Related Work

Sreedhar et al. [23] describe *extant analysis*, an offline, interprocedural static analysis that operates on a predefined set of classes that are defined to be the *closed world*. The goal of extant analysis is to categorize references into two categories, 1) *unconditionally extant* references guaranteed to remain in the closed world, and 2) *conditionally extant* references *not* guaranteed to stay in the closed world. For the first category, optimizations such as inlining can be directly applied. For the second category, runtime tests called *extant safety tests* can be used to create regions of code within which direct optimizations can be applied.

As described, extant analysis is an offline analysis that would be non-trivial to incorporate into an online JVM. The extant safety test proposed in their work examines the receiver of the method being dispatched, and therefore is more expensive and offers less opportunity for guard elimination than thin guards. The importance of optimizing extant safety tests (to minimize the dynamic number of tests executed) is discussed in their work, but no algorithm for doing so is presented; other variants of extant safety tests are mentioned but not thoroughly discussed. Thin guards could be considered to be one particular kind of extant safety test that is simple, yet effective in practice. Finally, their experimental evaluation admittedly relied on several unsafe assumptions about entry points into the closed world; our experimental evaluation makes no unsafe assumptions.

Pechtchanski and Sarkar [22] present a framework for performing optimistic interprocedural analysis. Their work describes a general dependency tracking mechanism for speculative optimizations, and presents an optimistic interprocedural type analysis as an application of their framework. Although addressing a similar problem, there are significant differences between their work and ours. Their framework is designed for systems that have on-stack replacement available as a mechanism for invalidation; the goal of our work is to avoid the need for on-stack replacement. Additionally, because on-stack replacement was not available in their system, the potential restrictions on optimization it imposes were not present in their experimental evaluation. Our evaluation used a complete implementation that ensures correct execution when class loading occurs. For some benchmarks, thin guards based on class hierarchy analysis offered speedups larger than those reported for their interprocedural type analysis.

Sealed calls [26] take advantage of *sealed packages* to identify additional methods that can not be overridden, even if these methods are not `final`. Such methods can be inlined directly without a guard.

10 Conclusions and Future Work

Dynamic class loading is important aspect of the Java programming language, although it complicates many optimizations such as inlining. Thin guards are a simple technique that can be used to reduce the performance penalty imposed by dynamic class loading. They are almost as easy to implement as traditional guards, but offer two advantages: they are 1) more efficient and 2) test a more general condition, allowing a larger region of code to be covered by a single guard.

For the benchmarks used in our study, thin guards were able to eliminate the majority of the penalty imposed by dynamic class loading. For the `_227_mtrt` benchmark, thin guards improved performance by 27% achieving 92% of ideal performance. Intraprocedural path splitting was shown to substantially improve the effectiveness of thin guards, although even without splitting, thin guards yielded reasonable performance improvements (13.5% for `_227_mtrt`).

For future work we are planning to evaluate the effectiveness of combining code patching and thin guards, to further improve performance. We intend to move our implementation into the open source release of Jikes RVM, as well as continue exploring the effectiveness of our technique on a larger suite of benchmarks. We hope to experiment with benchmarks that perform enough class loading to trigger invalidation; such benchmarks will allow us to evaluate the performance penalty incurred by failing guards, as well as evaluate various recovery strategies.

11 Acknowledgments

We would like to thank IBM Research and the entire Jikes RVM team for providing the infrastructure to make this research possible. We would also like to thank David Grove, Michael Hind, and Stephen Fink for helpful discussions and feedback on an earlier draft of this paper. Finally, we thank Michael Hind and Vivek Sarkar for their support of this work.

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), 2000.

2. B. Alpern, A. Cocchi, and D. Grove. Dynamic typechecking in Jalapeño. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, Apr. 2001.
3. M. Arnold. Online instrumentation and feedback-directed optimization of Java. Technical Report DCS-TR-469, Department of Computer Science, Rutgers University, 2001.
4. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.
5. M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.
6. C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Comiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, 1992.
7. C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Nov. 1991. *SIGPLAN Notices* 26(11).
8. M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.
9. T. S. P. E. Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.
10. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *9th European Conference on Object-Oriented Programming*, 1995.
11. D. Detlefs and O. Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming*, 1999.
12. K. Driesen and U. Hoelzle. Minimizing row displacement dispatch tables. *ACM SIGPLAN Notices*, 30(10):141–155, Oct. 1995.
13. M. Fernandez. Simple and effective link-time optimizations of modula-3 programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.
14. D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Oct. 1995.
15. U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.
16. U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 32–43, June 1992.
17. U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, Orlando, Florida, 20–24 June 1994. *SIGPLAN Notices* 29(6), June 1994.
18. 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*, Oct. 2000.

19. C. Krintz, D. Grove, D. Lieber, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software – Practice and Experience*, 31(8):717–738, Dec. 2000.
20. S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.
21. M. Paleczny, C. Vic, and C. Click. The Java Hotspot(tm) server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, 2001.
22. I. Pechtchanski and V. Sarkar. Dynamic optimistic whole program analysis: a framework and an application. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.
23. V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.
24. T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2001.
25. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.
26. A. Zaks, V. Feldman, and N. Aizikowitz. Sealed calls in Java packages. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'00)*, pages 83–92, 2000.