Strength Reduction for Loop-Invariant Types

Phung Hua Nguyen and Jingling Xue

Compiler Research Group
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2032, Australia

Abstract

Types are fundamental for enforcing levels of abstraction in modern high-level programming languages and their lower-level representations. However, some type-related features such as dynamic method calls and dynamic type casts can contribute substantially to the performance of a program. Loop-invariant type is a concept relating to an object whose dynamic type never changes inside a loop. In this case, operations on the type of the object may be redundant in the loop. As these operations often cause exceptions, existing redundancy elimination techniques usually fail to optimise them. This paper proposes a new approach to reducing the cost of two important operations on loop-invariant types: method tests and dynamic type checking. We demonstrate its usefulness and benefit in IBM's Jikes RVM, a dynamic compilation system for Java.

Keywords: Loop-Invariant Type, Strength Reduction, PRE, Type Checking, Inlining

1 Introduction

Types are fundamental for enforcing levels of abstraction in modern programming languages and their lower-level representations (Morrisett, Walker, Crary & Glew 1998). However, some type-related features pose challenges for language implementors to remove these abstraction boundaries to permit efficient execution. In object-oriented languages, two examples are dynamic method dispatching and dynamic type checking. To reduce the overhead in the former case, the compiler typically applies a number of devirtualisation techniques (Calder & Grunwald 1994, Chambers & Ungar 1990, Detlefs & Agesen 1999, Goldberg & Robson 1983, Chambers, Dean & Grove 1996, Hölzle, Chambers & Ungar 1991, Ishizaki, Kawahito, Yasue, Komatsu & Nakatani 2000, Krasner 1983, Deutsch & Schiffman 1984, Sundaresan, Hendren, Razafimahefa, Vallée-Rai, Lam, Gagnon & Godin 2000) in tandem with inlining (Hazelwood & Grove 2003). To reduce the cost in the latter case, various type encoding techniques have been devised (Alpern, Cocchi & Grove 2001, Krall, Vitek & Horspool 1997). For example, IBM's Jikes RVM (Alpern, Attanasio, Barton, Burke, P.Cheng, Choi, Cocchi, Fink, Grove, Hind, Hummel, Lieber, Litvinov, Mergen, Ngo, Russell, Sarkar, Serrano, Shepherd, Smith, Sreedhar, Srinivasan, & Whaley 2000), an adaptive compilation system for Java, consists of several devirtualisation techniques including class test (Calder & Grunwald 1994), method test (Detlefs & Agesen 1999) and code patching (Ishizaki et al. 2000) for researchers to experiment with. In the class (method) test approach, a guard test is generated to test the receiver of the class (method) to ensure that it is valid to execute an assumed target method. In addition, Jikes maintains several data structures operationally close to every object to facilitate dynamic type checking required by Java's type system.

Precise type information is invaluable for analysis and optimisation of programs. For example, both dynamic method dispatching and dynamic type checking benefit from class hierarchy analysis (Dean, Grove & Chamber 1995) and type analysis (Sundaresan et al. 2000, Bacon, Wegman & Zadeck 1996, Bacon & Sweeney 1996). By deducing the set of plausible dynamic (or concrete) types of an object at compile time, the compiler can reduce the number of class and method tests required in devirtualisation and eliminate some redundant dynamic type checks. In the special case when the receiver `x` at a call site `x.f()` has been proved to have a single implementation, the callee `f` can be dispatched as a direct call or directly inlined if the "closed-world" assumption is made. Otherwise, in languages that embrace dynamic class loading such as Java, only one single class/method test is needed to guard the compile-time "unique" implementation if it is inlined. But the caller will be recompiled (Detlefs & Agesen 1999) or patched (Ishizaki et al. 2000) later if the single-implementation assumption is invalidated due to, for example, dynamic class loading.

Obviously, it is imperative for the tests required in devirtualisation and dynamic type checking to be executed as efficiently as possible, especially when they are executed frequently. This paper proposes a new approach to reducing the cost of these tests in a loop in an important special case that has not been addressed before. A variable (i.e., object) in a loop is said to have a *loop-invariant dynamic (i.e., concrete) type* if the dynamic type never changes during the execution time of the loop. In this paper, such an object is said to have a loop-invariant type. Our proposed compiler technique, called *strength reduction for loop-invariant types*, aims at reducing the cost of the type operations whose operands have loop-invariant types. In this case, we will execute these operations only once, at the first iteration of the loop and cache its value in a new temporary. The executions of these operations in the remaining iterations of the loop are redundant; they will be replaced with cheaper operations that perform a test on the new temporary only.

We apply our technique to reduce the overhead of two important operations: method tests in devirtualisation and dynamic type checking. Our technique is particularly useful in a compilation system that sup-

```
 1 public void printRec() {
 2   ...
 3   Enumeration i = entry.items.elements();
 4   Enumeration f = fmt.elements();
 5
 6   while ( f.hasMoreElements() ) {
 7     s = ((String)f.nextElement());
 8
 9     if ( s != null )
10       spec.harness.Context.out.print(s);
11     else {
12       s = (String)i.nextElement();
13       spec.harness.Context.out.println(s);
14       }
15     }
16   ...
17   }
```

Figure 1: An example from `db` in SPECjvm98 where our strength reduction succeeds but PRE fails.

ports (a) adaptive recompilation, (b) dynamic class loading and (c) precise exception semantics. Because of (a), the address of an inlined method must be loaded dynamically at run time, making a method test more expensive than a class test. For this reason and since class tests are rarely used (as Table 1 shows), we do not also optimise class tests in this work. Because of (b), class and method tests are frequently necessary, especially if a call site is known to have more than one implementation. Because of (c), existing partial redundancy elimination (PRE) techniques (Bodik, Gupta & Soffa 1998, Knoop, Rüthing & Steffen 1994, Cai & Xue 2003) usually fail to remove the partial redundancy removed by our technique. Method tests and dynamic type checks involve *potential exception instructions* (PEIs). Consider a `while` loop shown in Figure 1, which is contained in a method taken from `db`, a program from SPECjvm98. Some irrelevant lines indicated by the "..." have been deleted. In line 12, `i` has a loop-invariant type. By applying our technique, the type check on `i` will be performed only once during the first iteration of the `while` loop. PRE cannot move this operation out of the loop since doing so may cause an exception even though it may not be thrown in the original code. Converting the loop into a `do-while` would not enable the desired code motion since the `else` branch may or may not be executed as before. If the call site at line 12 is inlined and guarded by method test(s), our strength reduction technique will perform these expensive tests only once. Again PRE fails for the above reasons. In addition, PRE fails in both cases because some earlier instructions are PEIs. We will discuss this again in Section 2.1.2.

The proposed technique fills a gap between PRE, which optimises loops but is usually ineffective in handling exception-throwing operations, and the work on call devirtualisation and dynamic type checking, which normally does not consider loop-oriented optimisations. We have implemented our technique in Jikes, which is an adaptive compilation for Java that supports dynamic class loading and precise exception semantics. We give the statistical evidence about the existence of loop-invariant types in benchmark programs. We present some experimental results demonstrating the usefulness of our technique.

The rest of this paper is organized as follows. Section 2 introduces our strength reduction technique. Section 3 discusses the type analysis required for identifying loop-invariant types. Section 4 presents some experimental results. Section 5 discusses the related work. Section 6 concludes the paper.

```
R0 = <receiver object>
R1 = load(R0 + <offset-of-VMT-in-object>)
R2 = load(R1 + <offset-of-method-in-VMT>)
R3 = load(VMT of expected class 1)
R4 = load(R3 + <offset-of-method-in-VMT>)
if (R2 == R4) {       // test case 1
   ...                // inlined method 1
} else {
   R3 = load(VMT of expected class 2)
   R4 = load(R3 + <offset-of-method-in-VMT>)
   if (R2 == R4> {    // test case 2
   ...                // inlined method 2
   } else
     CALL R2
}
```

Figure 2: Pseudo code of a method test (assuming that there are two target implementations inlined).

2 Strength Reduction for Loop-Invariant Types

In this section, we present our compiler technique for performing strength reduction on loop-invariant types. We show how to reduce the overhead of two important operations on loop-invariant types: method tests (Section 2.1) and dynamic type checking (Section 2.2). In each case, we introduce the operation in pseudo code, discuss Jikes's implementation, and finally, present our optimisation. In addition, we will use examples to explain in detail why existing PRE techniques fail to optimise these two cases.

2.1 Method Tests

Object-oriented languages such as Java complicate inlining, because methods are usually *virtual*. A virtual method is defined in one class, but may be overridden in subclasses of that class. The method actually invoked at a call site depends on the dynamic type of the *receiver* object.

Method test (Detlefs & Agesen 1999) is one of many devirtualisation techniques (Calder & Grunwald 1994, Chambers & Ungar 1990, Detlefs & Agesen 1999, Goldberg & Robson 1983, Chambers et al. 1996, Hölzle et al. 1991, Ishizaki et al. 2000, Krasner 1983, Deutsch & Schiffman 1984, Sundaresan et al. 2000) proposed to reduce the overhead of dynamic method calls for various object-oriented languages. A method test is generated to test the target method of a virtual call with an expected method to ensure that it is valid to make a direct call to the expected method. Java is a statically-typed language. The overhead of a dynamic method call is low. Thus, the expected method is usually inlined to boost performance.

Figure 2 shows the pseudo code of a method test. Every object keeps a reference to a *virtual method table* (VMT), which contains the addresses of all virtual methods of the class that the object belongs to. When a call site is inlined, the address of the VMT of the receiver and the address of the target method are loaded. Then, the address of the target method is compared with that of the inlined method (i.e., expected method) to ensure that it is valid to execute the inlined method. Note that if the address of the inlined method can change at run time – as is the case in dynamic recompilation, the address of the expected method must be loaded as illustrated by the

pseudo code.

In Jikes, bytecode is translated into high-level intermediate representation (HIR), low-level intermediate representation (LIR), machine-specific intermediate representation (MIR), and finally, machine code. At each level, many different optimisations are applied. Inlining is applied when bytecode is translated into HIR. Consider some method invocation `x.f()`, where `f` has one implementation in class `A`, a second implementation in class `B` and possibly other implementations. Figure 3 gives the HIR code for inlining the targets `A.f()` and `B.f()` using a method test. `IG_METHOD_TEST`, known as an *inline guard*, makes sure that all targets of the receiver `x` are called correctly. The operands of such a method test are the receiver, the expected method, the label for the next test and a guard variable. From the receiver, Jikes obtains the address of the invoked method and compares it with the address of the expected method. If the test succeeds, the inlined method is executed. Otherwise, the next inline guard is tried. Jikes uses the guard variable `guard0` to enforce the implicit dependences among the four guarded instructions. Thus, the null-pointer check on `x`, which may potentially cause a `NullPointerException`, must be performed first.

Figure 4 shows the pseudo assembly code translated from the HIR code given in Figure 3. The machine code can be understood easily once Figures 2 and 3 are. In the nomenclature of Jikes, VMT is referred to as the *type information table (TIB)*. A reference to each TIB is kept in an array of static values called the *JikesRVM table of contents (JTOC)*. Note that the null-pointer check for the object reference `x` does not appear in the assembly code. *The Jikes uses a hack to force a hardware interrupt when* `x=null=0` *since* `TIBOffset=-12` (Alpern et al. 2000). As Jikes is an adaptive compilation system, the address of a compiled method can change at run time. Hence, the addresses of `A.f` and `B.f` must be loaded at run time during the method tests. Note that the instruction `L R1,TIBOffset(R0)` appears in lines 2, 9 and 16. The last two can be removed by the compiler since it is is fully redundant with respect to the first.

2.1.1 Strength Reduction

To boost performance, redundant type operations (`NULLCHECK` and method tests) at a call site should be eliminated whenever possible, especially when they are executed in a loop. Obviously, if the dynamic type of the receiver is invariant in a loop, all the type operations should be executed once during the entire execution of the loop.

Figure 5 gives our optimised version of Figure 3 for performing a method test when the method test code sequence is contained in a loop such that the receiver has a loop-invariant dynamic type. (The optimised version of Figure 4 can be obtained easily from Figures 4 and 5 and is thus omitted.) A temporary variable, `tmp`, is initialized to 0 outside the loop. Thus, the original tests are executed during the first iteration, at which `tmp` will be set to the appropriate value depending on which case is successful. In all subsequent iterations, `tmp` is guaranteed not to be 0. Thus, the original tests are skipped. Instead, each original method test is now replaced by a cheaper test against `tmp` that *memorises* (i.e., *caches*) the result of the tests in the first iteration.

One seemingly limitation with our technique is that the uninlined case, `CALL f`, takes one more test to execute than before. This may not be a problem since the inlined methods are expected to be called frequently. In addition, recall that in the unoptimised code given in Figure 4, the two loads are required to obtain the address of an inlined method before each method test. In that case, the compiler may or may not move them out of the loop depending on the availability of physical registers. In our optimised version, these loads are guaranteed to be executed only once. Finally, our technique also generalises well when there are guard tests for a number of inlined methods. By mapping all the cases to consecutive integers, we can implement all required tests on the temporary in the form of a binary test. If hardware supports relative indirect jump, we can replace all the tests more cheaply with only *one* such an instruction.

In Jikes, the interface calls are implemented such that they can be treated similarly as the virtual calls as far as (Alpern, Cocchi, Fink, Grove & Lieber 2001) inlining is concerned. Thus, our technique also applies to this case.

2.1.2 Why PRE Fails?

PRE (Partial redundancy elimination) (Morel & Renvoise 1979) removes partial redundancies along some paths through a flow graph. It subsumes global common subexpression elimination and loop-invariant code motion. In order to avoid introducing additional computations on an execution path, the classic PRE techniques (Bodik et al. 1998, Knoop et al. 1994, Morel & Renvoise 1979) move a computation e to a point p in a flow graph only if all paths emanating from p must evaluate e before any operand of e is modified. This guarantees the so-called down-safety: (a) the number of evaluations of e cannot be larger than before, and (b) if e may throw an exception in the original code, then the same exception may occur a bit sooner in the transformed code. PRE must be applied so as not to change the order of PEIs, which is disallowed in Java.

Recently, our compiler group has developed a profile-guided PRE that solves the PRE problem optimally with respective to a given edge profile (Cai & Xue 2003). However, our technique has considered only the exception-free operations as code motion candidates.

It should be clear now why the existing PRE techniques usually fail to eliminate the partial redundancy in a method test. Consider the two examples in Figure 6. In the two CFGs, `B3` represents a basic block that contains a method test, where the receiver object `x` has a loop-invariant type. The null-pointer check on the receiver `x` is a PEI but it is partially redundant. In the `while` loop depicted in Figure 6(a), PRE will not move the null-pointer check and associated instructions in the method test outside the loop. Because `B3` does not post-dominate the loop header `B1`, doing so may cause an exception on the execution path `B1-B2-B4`. That is, such a code motion is not *down-safe*. In the `do-while` loop shown in Figure 6(b), PRE will not attempt to hoist the method test(s) in `B3` across the call site in `B2` since the execution of `B2` may potentially throw an exception. Doing so may change the order in which some PEIs are executed. Finally, we recall our discussions earlier about the PRE limitation using the example given in Figure 1.

These three examples collectively suggest that the conventional trick used for converting `while` loops into `do-while` loops does not usually help hoist exception-throwing expressions out of a loop. It is possible to try other kinds of control flow restructuring. For example, one may duplicate the so-called code-motion-preventing (CMP) regions in the CFG to enable a complete removal of all redundancy computations (Bodik et al. 1998). However, the large increase in code size can hardly justify the benefit obtained.

```
                MOV R0,x    // get address of receiver x
                NULLCHECK guard0,R0

                IG_METHOD_TEST R0,A.f(),NEXT_METHOD1,guard0
                ...         // inlined code for A.f()

NEXT_METHOD1:   IG_METHOD_TEST R0,B.f(),NEXT_METHOD2,guard0
                ...         // inlined code for B.f()

NEXT_METHOD2:   CALL f,guard0
```

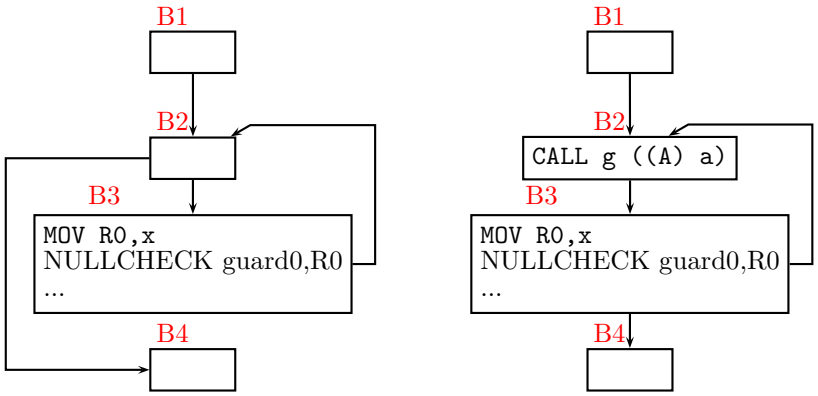
Figure 3: HIR code of a method test in Jikes.

```
 1     L    R1,TIBOffset(R0)    // get TIB address of receiver x
 2                              // whose address is kept in R0
 3     L    R2,A'sOffset(JTOC)  // get TIB address of class A
 4     L    R2,MethodOffset(R2) // get address of method A.f()
 5     CMP  MethodOffset(R1),R2
 6     JNE  NEXT_METHOD1
 7      ...                     // execute inlined method A.f()

 8 NEXT_METHOD1:
 9     L    R1,TIBOffset(R0)    // get TIB address from receiver
10     L    R2,B'sOffset(JTOC)  // get TIB address of class B
11     L    R2,MethodOffset(R2) // get address of method B.f()
12     CMP  MethodOFfset(R1),R2
13     JNE  NEXT_METHOD2
14     ...                      // execute inlined method B.f()

15 NEXT_METHOD2:
16     L    R1,TIBOffset(R0)   // get TIB address from receiver
17     CALL MethodOffset(R1)
```

Figure 4: Pseudo assembly code for the HIR code given in Figure 3.

```
           tmp = 0
 loop:
           ...
           MOV R0,x             // get address of receiver x
           NULLCHECK guard0,R0
           if (tmp == 1)
INLINE1:        ...             // inlined code for A.f()
           else if (tmp == 2)
INLINE2:        ...             // inlined code for B.f()
           else if (tmp == 3)
UNINLINE:       CALL f
           else {                // tmp == 0
                IG_METHOD_TEST R0,A.f(),NEXT_METHOD1,guard0
                tmp = 1
                goto INLINE1
NEXT_METHOD1:
                IG_METHOD_TEST R0,B.f(),NEXT_METHOD2,guard0
                tmp = 2
                goto INLINE2
NEXT_METHOD2:
                tmp = 3
                goto UNINLINE
                }
           ...
endloop
```

Figure 5: Optimised code of a method test when the receiver has a loop-invariant type.



(a) `while` loop (b) `do-while` loop

Figure 6: An illustration of PRE's failure in optimising method tests.

2.2 Dynamic Type Checking

Java is a strongly-typed programming language. Almost all type checking can be done at compile time and then verified by a Java Virtual Machine when classes are loaded. However, some runtime type checks are required by Java's type system. Several Java's dynamic type-checking bytecodes are: `checkcast` (casting a value of one type to another), `instanceof` (testing whether such a cast will succeed), `invokeinterface` (dispatching a method through an interface), `aastore` (storing an object in an array) and `athrow` (catching an exception).

A *type check* verifies whether a value of one type (*RHS*) can be legally assigned to a variable of another type (*LHS*). If so, the RHS type is said to be a *subtype* of the LHS type and the LHS type is said to be a *supertype* of the RHS type.

At its HIR level, Jikes makes use of instructions such as `InstanceOf` (for `instanceof`), `TypeCheck` (for `checkcast` and `invokeinterface`), `StoreCheck` (for arrays) and some instructions for exceptions to implement dynamic type checks. `InstanceOf` and `TypeCheck` share the same implementation code except that the former returns a boolean value while the latter returns no values. The instructions for exceptions are irrelevant since there are no redundancies to eliminate. `StoreCheck` can be optimised similarly as the remaining two cases except for the subtleties to be discussed in Section 3. Below we discuss how to eliminate redundant type checks specified as an instance of the `TypeCheck` instruction called *an interface test*.

Figure 7(a) depicts a loop that contains such an interface test that checks to see if the class of the object reference has implemented the specified interface. Let `I` be an interface type. Consider the three operations `x.f()`, `x instanceof I` and `(I)x`. In each case, the interface test is to ensure that the RHS `x` has implemented the interface `I`. If the object reference `x` has a loop-invariant type, Figure 7(b) will be the optimised version we generate. All instructions in **bold font** are introduced by the optimisation. A temporary, `tmp`, is initialized to 0 before the loop and set to 1 during the first iteration, where the original interface test is executed. In all other iterations of the loop, the interface test has been replaced by a simple test against the temporary `tmp`.

In Jikes, high-level type checking instructions are implemented using the techniques described in (Alpern, Cocchi & Grove 2001). Figure 8 gives the assembly code for an interface test taken from (Alpern, Cocchi & Grove 2001). Note that in our optimised code, all these instructions will be executed only once during the first iteration of the loop. It is irrelevant how these instructions work except for the following two points. First, the third, fourth and fifth instructions serve to extend the so-called *implements trits* array. Whether they are needed or not for the interface `I` is determined at compile time. Second, the MAYBE handler will always return the control to the instruction immediately after each `BLE MAYBE` instruction.

Several other checks can be optimised similarly. The number of instructions required depends on the underlying type-checking bytecode, the LHS type and RHS type. In the best case, two loads and one compare are required for an equality test (Alpern, Cocchi & Grove 2001).

For the same reasons we explained before with respect to method tests (Section 2.1.2), PRE cannot usually remove partial redundancy on dynamic type checks.

3 Loop-Invariant Type Analysis

Recall that a variable (or object) in a loop has a loop-invariant type if its dynamic type never changes during the execution of the loop. Obviously, a loop-invariant variable (as defined in the standard manner in (Aho, Sethi & Ullman 1986, p. 639)) has a loop-invariant type but the converse is not true. A variable can be assigned to different objects in the loop but if the type of these objects is the same, then the variable still has a loop-invariant type. This point is illustrated in Figure 9. Other cases can be constructed along this line.

As will be explained in Figure 10, we perform all required type analysis just before HIR is translated to LIR. At this stage, all object references subject to our type analysis are in symbolic registers. They can be classified into six categories: (a) `this`, (b) (scalar) local variables, (c) parameters, (d) (scalar) fields, (e) array elements, and (f) objects returned from method calls. Note that an object returned from a method invocation can be either a field reference or a reference to a temporary object created by the callee.

In Cases (a) – (c), we can find (with good accuracy) if a variable has a loop-invariant type or not by using the conventional intra-procedural reaching definition (or type) analysis. In the special case, `this` is always loop-invariant.

In Cases (d) – (f), some inter-procedural type and alias analysis is generally required. Fields (static or instance) are global variables. Techniques such as immutability analysis (Pechtchanski & Sarkar 2002, Porat, Biberstein, Koved & Mendelson 2000) try to prove if a field is immutable or not. Other techniques such as rapid type analysis (Bacon et al. 1996) and

```
loop:
          ...
          MUST_IMPLEMENT_INTERFACE ref,type,guard0
NEXT:
          ...
endloop
```

(a) An interface test inside a loop

```
                    tmp = 0

loop:
                    ...
                    if (tmp == 0)
                       goto TYPECHECK
NEXT:
                    ...
endloop

TYPECHECK:
                    MUST_IMPLEMENT_INTERFACE ref,type,guard0
                    tmp = 1
                    goto NEXT
```

(b) Optimised interface test

Figure 7: Strength reduction for an interface test when the object `ref` has a loop-invariant type.

```
L    R1,TIBOffset(RHS)     // get TIB address from RHS object
L    R1,ImplementsTrits(R1)// get array of trits from TIB
L    R2,R2,LengthOffset(R1)// get length of trits array
CMPI R2,LHSInterfaceId     // can trits contain this interface?
BLE  MAYBE                 // trits array too small => MAYBE
L    R1,LHSInterfaceId(R1) // get trit for this interface
CMPI R1,1                  // 1=>yes (class implements interface)
BLT  NO                    // 0=>no
BGT  MAYBE                 //>1=>maybe (further checking needed)
```

Figure 8: Assembly code for an interface test.

```
for (i=0; i<N; i++) {
  x = new A(i);
  x.f();
}
```

```
for (i=0; i<N; i++) {
  if (...)
    x = new A(i);
  else
    x = new A(i,i);
  x.f();
}
```

Figure 9: An illustration of loop-invariant types.

variable-type analysis (Sundaresan et al. 2000) may provide sharper analysis to detect variables that are *loop-variant* but have loop-invariant types. Our technique applies directly to array elements if they are treated as individual scalars. Consider:

```
1  for (i=0; i < N; i++) {
2    x = new A[N];
3    x[i].f();
4    for (j=0; j < M; j++) {
5      y[j].g();
6    }
7  }
```

In line 3, each element `x[i]` is accessed only once. So there is no redundancy to eliminate on the method test if the call site is inlined. In line 5, each element `y[j]` is accessed N times during the outer loop `i`. If this call site is inlined, the redundant method tests can be optimised as suggested in Figure 5 except that M different temporaries need to be introduced, one for each element of `y`. This does not appear to be practical. In fact, Jikes will probably inline a call site in a loop whose receiver is an array element only when all objects in the array have a common type. The call site at line 3 is one such an example. We recommend to apply our technique only in this case to remove the redundant method tests among the elements of the array `x`. In this case, only a single temporary variable is required as suggested in Figure 5. Similar subtleties exist for dynamic type checks required in assignments such as `a[i][j] = b` in a loop.

4 Experiments

Jikes has two compilers: the *baseline* compiler and the *optimising* compiler (Alpern et al. 2000). The optimising compiler translates bytecode into high-level intermediate representation (HIR), low-level intermediate representation (LIR), machine-specific intermediate representation (MIR), and finally, machine code. At each level, many different optimisations are applied. Figure 10 shows the implementation of our strength reduction technique in Jikes. Our technique is added as the third box in the pipeline. In order to evaluate the effectiveness of our technique, we have also modified Jikes so that code patching is replaced by a class test and a method test whichever is appropriate. In general, a class test is used only when the class that contains an inlined method is final.

We created the boot image using the FastAdaptiveSemiSpace configuration. Thus, the optimising compiler (at optimisation level 2) is used to statically compile all methods in the boot image. We ran the (modified) Jikes optimising compiler at optimisation level 1. Thus, the adaptive compilation is disabled. Our objective is to measure the effectiveness of our technique in replacing expensive type-related operations with cheaper ones.

We apply our technique just after Jikes has applied all its optimisations at the HIR level. These optimisations include common subexpression elimination, inlining and intra-procedural flow-sensitive optimisations. Jikes has not implemented any inter-procedural type analysis. In our current implementation, we conduct our loop-invariant type analysis only on objects that are `this`, (scalar) local variables, and parameters. Since Jikes performs flow-sensitive intra-procedural type analysis, the exact type for the receiver `x` in each of the two examples illustrated in Figure 9 can be found. So each call can be replaced with a direct call. These situations also carry over into dynamic type checks. Therefore, we have reduced our loop-invariant type analysis to the conventional loop-invariant analysis. Tables 1 and 2 present the (conservative) evidence about the existence of loop-invariant types in the SPECjvm98 benchmark suite. Table 1 gives the *static* number of call sites and type checks while Table 2 highlights their *dynamic* execution counts. (Interestingly, there are a lot of uninlined call sites with loop-invariant types.) We expect to see more optimisation opportunities if some inter-procedural type and alias analysis is available.

Table 3 demonstrates the effectiveness of our technique in reducing the "strength" of expensive operations on loop-invariant types. Columns 2 – 4 are for method tests while Columns 5 – 7 are for dynamic checks. Figure 11 duplicates Columns 4 and 7 in graphic form. In Column 2 (named "Before"), we list the total number of method tests contained inside the loops. In Column 3 (named "After"), we give the number of remaining such tests after our technique has been applied. In Column 4, we show the percentage of reduced tests in each benchmark. The percentages for `db` and `mpeg` are 94% and 96%, respectively. In the case of `compress`, there is only one loop-invariant type inside a loop with a single iteration. Thus, our technique is not beneficial here. For `javac`, there are a lot of loops with small iterations. Despite of this, the percentage reduction is still 7%. The data for dynamic checks given in Columns 5 – 7 for dynamic checks can be understood in a similar manner. In particular, the percentage reductions for `mpeg` and `compress` are 61% and 64%, respectively.

5 Related Work

This work overlaps with the prior work on PRE, call devirtualisations, dynamic type checking and type analysis.

PRE. PRE (Bodik et al. 1998, Knoop et al. 1994, Cai & Xue 2003, Morel & Renvoise 1979) is a powerful transformation for removing partial redundancies in computations on a path in a program's CFG. It has been primarily applied to eliminate redundant arithmetic operations and loads. As we discussed earlier in Sections 1 and 2.1.2, PRE cannot hoist a PEI out of a loop unless it post-dominates the header of the loop and doing so does not change the order in which all exceptions are thrown. Thus, converting `while` loops to `do-while` loops can only allow a limited number of PEIs to be hoisted outside a loop.

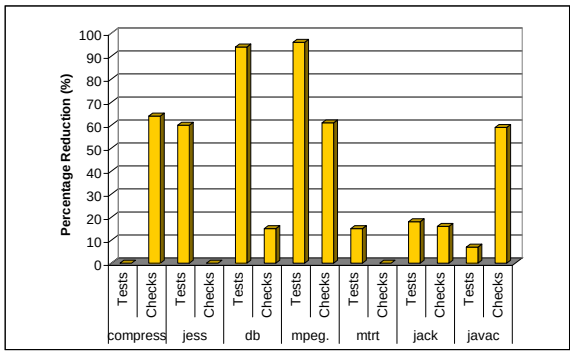


Figure 11: Columns 4 and 7 of Table 3.

Call Devirtualisation. A number of techniques have been invented in order to dispatch a virtual call efficiently, including Smalltalk-80's global cache (Goldberg & Robson 1983, Krasner 1983), monomorphic inline cache (Deutsch & Schiffman 1984), polymorphic inline cache (Hölzle et al.

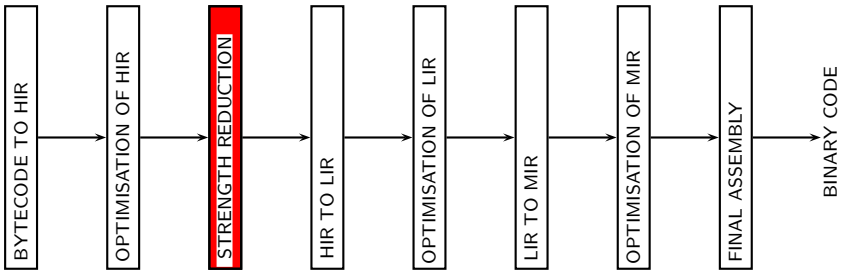


Figure 10: A framework for performing strength reduction in Jikes.

Benchmark		Uninlined Call Sites	Class Tests	Method Tests	Dynamic Checks
compress	OutLoop	2223	1	208	51
	LoopVar	470	0	81	10
	LoopInv	60	0	1	4
jess	OutLoop	3645	1	379	175
	LoopVar	853	0	149	61
	LoopInv	184	0	46	7
db	OutLoop	2404	1	261	52
	LoopVar	552	0	112	24
	LoopInv	98	0	2	18
mpegaudio	OutLoop	2630	6	224	64
	LoopVar	498	0	82	10
	LoopInv	72	1	2	5
mtrt	OutLoop	2506	1	717	51
	LoopVar	524	0	163	16
	LoopInv	98	0	27	4
jack	OutLoop	4590	7	797	129
	LoopVar	1070	0	137	107
	LoopInv	255	0	24	60
javac	OutLoop	5946	3	537	316
	LoopVar	945	0	99	126
	LoopInv	492	0	75	63

Table 1: Static counts of call sites and dynamic type checks obtained for objects that are `this`, locals and parameters in the strength reduction pass shown in Figure 10. OutLoop gives the number of a measured category outside a loop. LoopVar (LoopInv) quantifies loop-variant(-invariant) types, resp.. By combining the 3rd and 4th columns, we obtain "Inlined Call Sites" inside loops.

Benchmark		Uninlined Calls	Class Tests	Method Tests	Dynamic Checks
compress	OutLoop	7457	27	561	199
	LoopVar	451	0	51	44
	LoopInv	19726404	0	1	90
jess	OutLoop	12481109	153	2155003	9988762
	LoopVar	15805547	0	1232641	4346221
	LoopInv	4530442	0	5384768	51
db	OutLoop	56115	18	5808774	136
	LoopVar	67802248	0	824	53204281
	LoopInv	28952	0	15333	14923908
mpegaudio	OutLoop	15635547	96051	2020906	143147
	LoopVar	9346678	0	8877	20
	LoopInv	9392989	1	999385	42
mtrt	OutLoop	13746301	40	223483108	290
	LoopVar	7214491	0	14558714	1699226
	LoopInv	1001697	0	3275285	44
jack	OutLoop	6313916	497487	459770	2115415
	LoopVar	6123017	0	3623573	3670085
	LoopInv	2814698	0	1271958	2751762
javac	OutLoop	29364184	1044	2648809	3888563
	LoopVar	14310164	0	451644	2096883
	LoopInv	5226740	0	558977	3370808

Table 2: Dynamic execution counts of the same categories in Table 1.

Benchmark	Method Tests			Dynamic Checks		
	Before	After	Reduce (%)	Before	After	Reduce (%)
compress	52	52	0	134	48	64
jess	6617409	2582358	60	4346271	4346233	0
db	16157	826	94	68128189	57560119	15
mpegaudio	1008263	40123	96	62	24	61
mtrt	17833999	14999335	15	1699270	1699230	0
jack	4895531	3970374	18	6421847	5389520	16
javac	1010621	937849	7	5467691	2214247	59

Table 3: Strength reduction for method tests and dynamic checks in loops.

1991), class test (Calder & Grunwald 1994), method test (Detlefs & Agesen 1999), code patching (Ishizaki et al. 2000), and loop splitting (Chambers & Ungar 1990, Chambers et al. 1996).

The monomorphic inline cache approach (Deutsch & Schiffman 1984) is very effective at a monomorphic call site and even at a call site whose receiver has a loop-invariant type. However, the technique is based on self-modification and is not applicable to multi-threaded languages like Java. In the case of polymorphic inline cache (Hölzle et al. 1991), a call site is modified so that a dynamically generated stub is invoked first. This routine will execute class tests to see if the receiver class matches one of the classes seen previously. If it succeeds, the assumed method is executed. If it fails, the method lookup is performed to obtain the address of the invoked method and add one more class test to the stub routine. As compared to these caching techniques, our approach exploits loop-invariant types *statically* to reduce the cost of the required tests.

Direct inlining (Detlefs & Agesen 1999) and code patching (Ishizaki et al. 2000) inline a call site without a guard test provided there is only a single implementation of the callee at the time when the caller is compiled. If the assumption becomes invalid due to dynamic class loading, the caller will be recompiled or *patched* so that the call site may be inlined with a guard test. Our approach is complementary to these techniques in that it can be called upon whenever a call site, which is loop-invariant, becomes polymorphic.

Loop splitting (Chambers et al. 1996, Chambers & Ungar 1990) restructures the control flow to move all guard tests out of a loop at the cost of exponential code explosion. In addition, the technique is not effective if the call site does not post-dominate the header of the loop.

Dynamic Type Checking. We only mention some type encoding techniques devised (Alpern, Cocchi & Grove 2001, Krall et al. 1997). Our technique complements these techniques to avoid redundant checks.

Type Analysis. Type analysis helps reduce the set of dynamic types of an object. In the extreme case when the set is a singleton, the guard tests are unnecessary. Class hierarchy analysis (CHA) (Dean et al. 1995) uses the statically declared types of variables and the class hierarchy of the program to determine the set of possible dynamic types of an object. Rapid type analysis (Bacon & Sweeney 1996, Bacon et al. 1996) offers more accurate results by combining CHA and the information about instantiated classes in the program. Variable-type and declared-type analysis techniques (Sundaresan et al. 2000) make the results even more precise by propagating the information about class instantiation and type declaration, respectively, through implicit and explicit assignments. Other more powerful but also more expensive type analysis techniques also exist (DeFouw, Grove & Chambers 1998, Lenart, Sadler & Gupta 2000).

6 Conclusion

In this paper, we present a new compiler optimisation technique, called strength reduction for loop-invariant types, for eliminating redundant type-related operations in loops. We describe its application to two important operations in object-oriented languages: method tests employed in devirtualisation and dynamic type checking. Our technique fills a gap between partial redundancy elimination (PRE), which is usually limited to optimising exception-free operations, and the work on devirtualisation and type checking, which does not normally address the issue of loop-oriented optimisations.

We have implemented our technique in Jikes, a state-of-the-art adaptive compilation system for Java that supports dynamic class loading and precise exception semantics. Our statistics gathered on SPECjvm98 show convincingly the existence of loop-invariant types even after Jikes's optimising compiler has applied its extensive optimisations at the HIR level. Our experimental results on SPECjvm98 demonstrate the effectiveness of our technique in replacing expensive operations on loop-invariant types with cheaper ones.

References

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compilers: Principles, Techniques and Tools*, Addison-Wesley.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., P.Cheng, Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., & Whaley, J. (2000), 'The Jalapeño virtual machine', *IBM System Journal* **39**(1).

Alpern, B., Cocchi, A., Fink, S., Grove, D. & Lieber, D. (2001), Efficient implementation of Java interfaces: Invokeinterface considered harmless, *in* '16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', pp. 108–124.

Alpern, B., Cocchi, A. & Grove, D. (2001), Dynamic type checking in Jalapeño, *in* 'USENIX Java Virtual Machine Research and Technology Symposium'.

Bacon, D. F. & Sweeney, P. F. (1996), Fast static analysis of C++ virtual function calls, *in* '11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', SIGPLAN Notices, ACM Press, pp. 324–341.

Bacon, D. F., Wegman, M. & Zadeck, K. (1996), Rapid type analysis for C++, Technical report, IBM Thomas J.Watson Research Center.

Bodik, R., Gupta, R. & Soffa, M. L. (1998), Complete removal of redundant computations, *in* 'ACM SIGPLAN' 98 Conference on Programming Language Design and Implementation', pp. 1–14.

Cai, Q. & Xue, J. (2003), Optimal and efficient speculation-based partial redundancy elimination, *in* '1st IEEE/ACM International Symposium on Code Generation and Optimization'.

Calder, B. & Grunwald, D. (1994), Reducing indirect function call overhead in C++ programs, *in* '21st Annual ACM Symposium on Principles of Programming Languages', pp. 397–408.

Chambers, C., Dean, J. & Grove, D. (1996), Whole-program optimization of object-oriented languages, Technical Report TR-96-06-02, University of Washington, Seattle, Washington 98195-2350 USA.

Chambers, C. & Ungar, D. (1990), Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs, *in* 'ACM SIGPLAN '90 Conference on Programming Language Design and Implementation', pp. 150–164.

Dean, J., Grove, D. & Chamber, C. (1995), Optimization of object-oriented programs using static class hierarchy analysis, *in* '5th European Conference on Object-Oriented Programming', Vol. 952, Springer, pp. 77–101.

DeFouw, G., Grove, D. & Chambers, C. (1998), Fast interprocedural class analysis, *in* '25th Annual ACM Symposium on Principles of Programming Languages', San Diego, CA, pp. 222–236.

Detlefs, D. & Agesen, O. (1999), Inlining of virtual methods, *in* '13th European Conference on Object-Oriented Programming', pp. 258–278.

Deutsch, L. & Schiffman, A. (1984), Efficient implementation of the Smalltalk-80 system, *in* '11th Annual ACM Symposium on Principles of Programming Languages', Salt Lake City, Utah, United States.

Goldberg, A. & Robson, D. (1983), *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

Hazelwood, K. & Grove, D. (2003), Adaptive online context-sensitive inlining, *in* '1st IEEE/ACM International Symposium on Code Generation and Optimization', San Francisco, California.

Hölzle, U., Chambers, C. & Ungar, D. (1991), Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, *in* 'ECOOP'91 European Conference on Object-Oriented Programming', Springer Verlag Lecture Notes in Computer Science 512, Springer-Verlag, Geneva.

Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. & Nakatani, T. (2000), A study of devirtualization techniques for a Java just-in-time compiler, *in* '15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', pp. 294–310.

Knoop, J., Rüthing, O. & Steffen, B. (1994), 'Optimal code motion: Theory and practice', *ACM Transactions on Programming Languages and Systems* **16**(4), 1117–1155.

Krall, A., Vitek, J. & Horspool, N. (1997), Near optimal hierarchical encoding of types, *in* '11th European Conference on Object-Oriented Programming', Springer, Finland, pp. 128–145.

Krasner, G. (1983), *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley.

Lenart, A., Sadler, C. & Gupta, S. K. S. (2000), SSA-based flow-sensitive type analysis: Combining constant and type propagation, *in* '2000 ACM Symposium on Applied computing', pp. 813–817.

Morel, E. & Renvoise, C. (1979), 'Global optimization by suppression of partial redundancies', *Communications of the ACM* **22**(2), 96–103.

Morrisett, G., Walker, D., Crary, K. & Glew, N. (1998), From system F to typed assembly language, *in* '25th ACM SIGPLAN Symposium on Principles of Programming Languages', pp. 85–97.

Pechtchanski, I. & Sarkar, V. (2002), Immutability specification and its applications, *in* 'Joint ACM Java Grande -ISCOPE 2002 Conference', Seatle, Washington, USA.

Porat, S., Biberstein, M., Koved, L. & Mendelson, B. (2000), Automatic detection of immutable fields in Java, *in* 'Proceedings of CASCON 2000'.

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E. & Godin, C. (2000), Practical virtual method call resolution for Java, *in* '15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications', pp. 264–280.