

Fault-Safe Code Motion for Type-Safe Languages

Brian R. Murphy
Google
Beijing, China
brm@acm.org

Vijay Menon
Intel Labs
Santa Clara, CA 95054
vijay.s.menon@intel.com

Florian T. Schneider
Dept. of Computer Science
ETH Zurich, Switzerland
florian.schneider@inf.ethz.ch

Tatiana Shpeisman
Intel Labs
Santa Clara, CA 95054
tatiana.shpeisman@intel.com

Ali-Reza Adl-Tabatabai
Intel Labs
Santa Clara, CA 95054
ali-reza.adl-
tabatabai@intel.com

ABSTRACT

Compilers for Java and other type-safe languages have historically worked to overcome overheads and constraints imposed by runtime safety checks and precise exception semantics. We instead exploit these safety properties to perform code motion optimizations that are even more aggressive than those possible in unsafe languages such as C++.

We present a novel framework for speculative motion of *dangerous* (potentially faulting) instructions in safe, object-oriented languages such as Java and C#. Unlike earlier work, our approach requires no hardware or operating system support. We leverage the properties already provided by a safe language to define *fault safety*, a more precise notion of safety that guarantees that a dangerous operation (e.g., a memory load) will not fault at a given program point.

We illustrate how typical code motion optimizations are easily adapted to exploit our safety framework. First, we modify the standard SSAPRE partial redundancy elimination (PRE) algorithm [18, 20] to use fault safety, rather than the traditional down safety property. Our modified algorithm better exploits profile information by inserting of dangerous instructions on new paths when it is profitable and *provably* safe. Second, we extend an instruction trace scheduler to use fault safety to safely schedule load instructions across branches to better tolerate memory latency and to more compactly target instruction slots.

We implemented these optimizations in StarJIT [1], a dynamic compiler, and show performance benefits of up to 10% on a set of standard Java benchmarks.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: compilers, optimization

General Terms: Performance, Reliability, Languages, Algorithms

Keywords: code motion, speculative code motion, safety dependences, intermediate representations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.
Copyright 2008 ACM 978-1-59593-978-4/08/04 ...\$5.00.

1. INTRODUCTION

Optimizing compilers have long relied on code motion to enable key program transformations such as redundancy elimination, dead store elimination, and instruction scheduling. Partial redundancy elimination (PRE), for example, performs code motion to place expressions at new program points to eliminate redundant computations. Traditionally, compilers have relied on the *down safety* [19] property to guide code motion, and, in particular, to ensure that a transformation did not introduce a computation onto a program path where it did not previously exist. Down safety is very useful for two reasons. First, it implies an optimality property: on any given program path, a transformed program would execute no more instructions than the original program. The compiler does not optimize one program path to the detriment of another. Second, it provides a crucial safety property: a transformed program will not fault when the original program did not. This guarantees that the transformed program preserves partial correctness of the original program for execution paths.

```
for (i = 0; i < n; ++i)
  if (i & mask)
    k += a + b;
(a) Partially redundant expression
```

```
tmp = a + b;
for (i = 0; i < n; ++i)
  if (i & mask)
    k += tmp;
(b) Redundant expression hoisted expression
```

Figure 1: Speculative code motion

With the advent of just-in-time compilation and profile-guided optimization, the optimality property of down safety has proven unnecessarily restrictive. With precise profile information, it is often advantageous for a compiler to optimize a frequently executed path at the cost of penalizing a rarely taken one [20, 5, 2]. For example, in Figure 1(a), the expression $a+b$ is conditionally computed inside the loop. If our profile information indicates that the condition typically holds for more than one loop iteration, it is worth hoisting the computation outside the loop as shown in Figure 1(b), even though it is not down safe at that point. This code motion is clearly speculative. In cases where the condition never holds (e.g., the mask is 0), we have introduced an extra computation. However, our profile information tells us that this is rare, and the extra computation is innocuous.

In Figure 2, we show an almost identical example. In this case, however, the redundant expression involves a memory

<pre> for (i = 0; i < n; ++i) if (i & mask) k += a.x + b; </pre>	<pre> tmp = a.x + b; for (i = 0; i < n; ++i) if (i & mask) k += tmp; </pre>
(a) Partially redundant expression	(b) Risky transformation

Figure 2: Dangerous speculation of memory access

access. The subexpression `a.x` dereferences a field from an object in the heap and may fault if `a` is an invalid reference.¹ Hoisting the expression is no longer innocuous. If the object is invalid but the condition never holds, the transformed program would fault when the original would not. Because of this, speculative code motion has generally been restricted to computations that are intrinsically safe—such as the arithmetic expression in Figure 1—or has relied upon specialized hardware support.

<pre> if (a != null) { for (i = 0; i < n; ++i) if (i & mask) k += a.x + b; } </pre>	<pre> if (a != null) { tmp = a.x + b; for (i = 0; i < n; ++i) if (i & mask) k += tmp; } </pre>
(a) Partially redundant expression	(b) Safe speculation of memory access

Figure 3: Provably safe speculative code motion

In this paper, we observe that we can systematically apply speculative code motion to potentially faulting instructions under certain conditions. The example in Figure 3 is similar to that in Figure 2, but with a test for null at the beginning. In a language such as C++, this new test still does not allow us to hoist the load. Even though the reference `a` is non-null, it may still be invalid. In a type-safe language such as Java, however, the reference `a` *cannot* be invalid. The static verification of the type of `a` and the runtime test for a null value are, in combination, sufficient to prove that an access to a field `a` cannot fault. A compiler that can leverage this safety information can place the load at any point after the null test. In this paper, we present a framework for utilizing this safety information. In particular, we make the following novel contributions:

- We define fault safety, a new safety property for code motion that leverages type-safe languages to enable provably safe speculative code motion on dangerous operations. [Section 2]
- We describe how fault safety is computed, and we discuss how to use abstract *tau* variables (an untyped variant of proof variables [22]) to maintain this information in an optimizing compiler’s intermediate representation. We demonstrate how null, type, and bounds check elimination extend the scope of fault safety and the range of safe speculation. [Section 3]
- We present a modified partial redundancy elimination algorithm, based upon SSAPRE [18, 20], that uses

¹The load may fault if the compiler hoists it but not its corresponding runtime check. If the compiler also hoists the check, it may incorrectly generate an exception instead. Neither behavior is consistent with the semantics of the original Java program [14, 2].

fault safety to perform safe speculative redundancy elimination. The algorithm can speculatively hoist dangerous operations, such as memory loads, outside loops in a provably safe manner. [Section 4]

- We present a modified instruction scheduling algorithm that exploits fault safety to speculatively schedule memory loads across branch operations. We demonstrate that our approach is more effective than previous approaches that relied upon special speculative hardware such as the Itanium® Processor Family’s (IPF) control speculation feature. [Section 4]

We implemented our framework and optimizations within the StarJIT [1] Just-In-Time Java compiler, and measure the effectiveness of fault-safety-based speculative code motion on the SPEC JVM98 benchmarks [25].

2. DANGEROUS COMPUTATION SAFETY

A compiler’s ability to legally move a computation to a new program point depends upon the side effects of that computation. To preserve the semantics of the original program, the compiler must consider the observable side effects of a computation, including writes to memory or faults that could cause a program to crash.

We define a *safe* computation as one with no observable effects on program control flow. Safe computations include arithmetic calculations that do not write to memory and that never fault.² From a correctness perspective, safe computations are easy for a compiler to reason about. Introducing safe computations at new program points may change performance, but will not affect the correctness of a program as long as the compiler respects data dependences. For example, the computation `a+b` in Figure 1 is safe and may be hoisted out of the loop.

We define an *unsafe* computation as one where observable side effects, such as writes or faults, may occur. For example, a write to a field of an object (e.g., `a.x = 0`) will have side effects: a fault if the object reference is invalid, or a change to shared data observable to other threads.

A compiler must be careful about moving or introducing new unsafe computations. Traditionally, such computations are never newly introduced to a program path. In the literature [19], an expression *e* is defined to be *up-safe* at a program point *p* if the computation of *e* at *p* would not introduce a new computation on any path from the program entry to *p*. The expression *e* is defined as *down-safe* at a program point *p* if the computation of *e* at *p* would not introduce a new value on any path from *p* to the program exit. A new placement of an expression does not introduce a new computation onto any program path if the expression is either up-safe (for sinking) or down-safe (for hoisting) at that point.

The control flow graph in Figure 4 illustrates safety. Variable `a` is local to the method and points to an object on the heap. Load expression `a.x` is computed in nodes D and E. This expression is down-safe at node C: any path exiting C must traverse either D or E; thus, the expression is fully anticipated.³ Placing the load at C will not introduce

²In this context, we do not consider writes to virtual registers or private stack variables to be observable side effects.

³As discussed in the next section, we require that any exceptional control flow is represented explicitly.

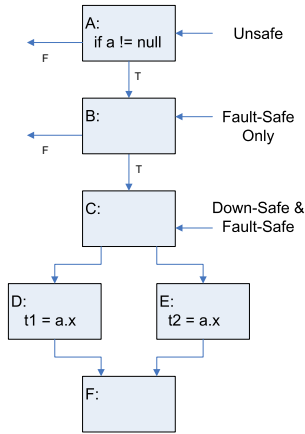


Figure 4: Different notions of safety

it on a new path. On the other hand, the load expression is not down-safe at nodes A or B, since from each node a path exists in which the load expression `a.x` was not computed. Placing the load at either node introduces it on a new path. Traditional code motion algorithms would consider these placements illegal.

The down-safety property may be conservative, but is a useful constraint for unsafe computations, as was shown in Figure 2.

2.1 Dangerous Computations

This constraint can be relaxed for an important class of unsafe operations. We define *dangerous* computations as operations that may fault but will otherwise have no observable side effects. Such computations include indirect loads or divides. Programmers typically do not want their programs to fault, so they usually protect memory accesses or divide operations to avoid them. For example, in Figure 4, the load operation is protected by a test in node A that ensures the object is not null. As faults are rare, dangerous computations are usually safe in terms of run-time behavior.

In a language such C, a compiler cannot treat a dangerous instruction differently from any other unsafe instruction. In Figure 4, the programmer may know that the load will never fault, but the compiler cannot reason at this level. Even if the object is not null, it may point to invalid memory.

2.2 Safety Dependences and Fault Safety

In a language such as Java, the load in Figure 4 can never fault. The runtime check in node A combined with Java’s static type checking ensures that the load will not fault [14, 2]. In fact, Java compilers routinely use such information to avoid adding runtime checks for many load instructions.

Following [22], we say that the loads in nodes D and E are *safety-dependent* on the branch in node A. More generally, a computation is safety dependent on any control flow point that establishes its safety. An intrinsically safe instruction has no safety dependences. A dangerous instruction, such as the loads in our example, is safety-dependent on one or more preceding control flow points.

We consider a computation to be *fault-safe* at any point where it respects its safety dependences. In Figure 4, the

```

aload_0
checkcast #5 <Class sometype>
getfield #19 <Field sometype::x>

```

(a) Input Bytecode

```

I1: tau0 = checktype t0, sometype → handler
I2: tau1 = checknull t0 → handler
I3: t4 = ldflda t0, sometype::x
I4: t5 = ldind [t4], tau1, tau0

```

(b) Intermediate Representation

Figure 5: Lowered safety representation

computation `a.x` is fault-safe at node B even though it is not down-safe. The test that establishes its safety is available in the dataflow sense. Thus, the load could be safely moved to node B without violating the original program semantics. In contrast, the load is not fault-safe in node A, preceding the test. A placement of the load at this point would violate the load’s safety dependence.

3. INTERMEDIATE REPRESENTATION

A safe language compiler can leverage fault safety to perform safe speculative code motion. To do this, however, the compiler must explicitly represent and preserve the safety dependences in the original program throughout the compilation process. Our starting point is the program verification work of [22]. That work focused on verifying the result of traditional optimizations; our focus here is to exploit safety dependences to *enhance* optimizations. We represent safety dependences as value dependences in the form of abstract *tau values* (shorthand for *proof variables*).⁴ We do not require the sophisticated type system of [22] for our purpose—constraining, not verifying, optimizations—so we omit this additional mechanism here.

As in prior work [7, 15, 8], we lower Java bytecodes into one or more lower level operations in our IR. In particular, for operations such as loads or divides, we separate runtime checks from the underlying dangerous computation. For example, a field load requires a test that the dereferenced object is non-null. An array element load requires a similar test as well as a bounds test to establish that index is within the array’s bounds.

3.1 Representing Safety in Lowered Bytecode

Our dynamic compiler uses tau values to represent the safety dependences of dangerous instructions in the lowered IR. The Java bytecode shown in Figure 5(a) becomes the lowered IR of Figure 5(b). The stack slots in the bytecode have been converted into static single assignment (SSA) virtual registers in our IR. The arrows represent exception control flow that is triggered when a check fails. All exceptions are represented explicitly, and check operations (along with any other potentially excepting instructions) end basic blocks.

The `checkcast` operation is lowered to a `checktype` operation. The `tau0` operand returned by this operation establishes that the check succeeded and is defined only on the fall-through path (i.e., I2 and beyond). This tau value is abstract and used only as a safety constraint. The field

⁴Tau values can also be viewed a more general form of condition registers [7], representing constraints globally rather than within a block.

```

I1: tau0 = checktype t0, sometype → handler
I2: tau1 = checknull t0 → handler
I3: t4 = ldflda t0, sometype::x
I4: t5 = ldind [t4], tau1, tau0
I5: tau3 = checktype t0, sometype → handler
I6: tau4 = checknull t0 → handler
I7: t6 = ldflda t0, sometype::y
I8: t7 = ldind [t6], tau4, tau3

```

(a) Two field loads with checks

```

I1: tau0 = checktype t0, sometype → handler
I2: tau1 = checknull t0 → handler
I3: t4 = ldflda t0, sometype::x
I4: t5 = ldind [t4], tau1, tau0
I6: tau3 = checktype t0, sometype = tau0
I5: tau4 = checknull t0 = tau1
I7: t6 = ldflda t0, sometype::y
I8: t7 = ldind [t6], tau4, tau3

```

(b) Redundant checks replaced by copies

```

I1: tau0 = checktype t0, sometype → handler
I2: tau1 = checknull t0 → handler
I3: t4 = ldflda t0, sometype::x
I4: t5 = ldind [t4], tau1, tau0
I7: t6 = ldflda t0, sometype::y
I8: t7 = ldind [t6], tau1, tau0

```

(c) After copy propagation

Figure 6: Redundant checks eliminated

load is lowered into three operations. First, a `checknull` operation verifies that the object (`t0`) is non-null and, as before, defines a `tau` to represent this constraint. Second, a `ldflda` operation performs a simple address calculation, requiring no safety constraint. Third, a `ldind` represents the actual load. As arguments, this operation takes not only the address, but the `tau` operands that make the load safe.

Despite the distinct names used here, `tau`-valued operands are ordinary SSA operands in the compiler IR, distinguished only by type. Importantly, they represent safety constraints on other local SSA variables. As such, they cannot be killed and are valid wherever their definitions are available.

We are able to handle redundant check operations as ordinary redundant expressions. Figure 6(a) shows two sequential accesses to fields of an operand `t0`. Each redundant check is replaced by a copy from the corresponding previous check, as in Figure 6(b). After copy propagation, the code is simplified to the final form in Figure 6(c).

3.2 Instructions to facilitate check elimination

In our compiler, we preserve fault safety in the presence of aggressive check elimination. To facilitate this, we introduce new `tau`-defining instructions in the following subsections.

3.2.1 Representing branch dependence: `tauedge`

To allow safe check elimination based on branch conditions, we introduce a `tauedge` instruction, which is associated with the preceding conditional branch and yields a `tau` value indicating that the preceding conditional branch was taken or not, as needed, to reach the `tauedge` instruction.

Figures 7(a) and 7(b) show the original bytecode and internal representation of a check made redundant by a preceding branch. To remove redundant check I2, we insert a `tauedge` instruction immediately following the conditional branch that makes the check redundant. The `tau` defined by a `tauedge` instructions can be used in any successor of a conditional branch which is dominated by the branch edge.

```

aload_0
ifnull L1
checkcast #5 <Class sometype>
getfield #19 <Field sometype::x>

```

(a) Redundant check

```

I0: ifnull t0 goto L1 → L1
...
I1: tau0 = checktype t0, sometype → handler
I2: tau1 = checknull t0 → handler
I3: t4 = ldflda t0, sometype::x
I4: t5 = ldind [t4], tau1, tau0

```

(b) Lowered representation

```

I0: ifnull t0 goto L1 → L1
tau2 = tauedge
...
I1: tau0 = checktype t0, sometype → handler
I2: tau1 = checknull t0 = tau2
I3: t4 = ldflda t0, sometype::x
I4: t5 = ldind [t4], tau2, tau0

```

(c) Check replaced by a use of `tauedge`

Figure 7: `tauedge` represents conditional safety

Note that when a redundant branch is folded, the compiler must update its corresponding `tauedge` instruction (if one exists). In the example in Figure 8(b), if additional context becomes available which allows folding of the branch I4, the `tauedge` instruction I9 must be replaced by a copy from a `tau` value providing the “reason” why the branch was folded. To accomplish this, we have modified any optimization pass (e.g., constant propagation, copy propagation, value numbering) which might fold branches based on control flow context to maintain enough information during its analysis to allow any `tauedge` associated with a folded branch to be converted to a copy from a `tau` representing the control flow which allows the branch to be folded.

3.2.2 Effect of multiple conditions: `tauand`

In some cases, a check is eliminated based on multiple preceding conditions. This is most common with array bounds check elimination [4], but can occur in other circumstances.

Consider the code segment shown in Figure 8(a), which safely reads element `t0` of array `t1`. Instruction I7: `checkbounds t0, t2` throws an exception unless $(0 \leq t0)$ and $(t0 < t2)$. From the preceding explicit tests, it is clear⁵ that the check will never fail. In this case, we have two prior conditions contributing to the safety of the check, so we insert `tauedge` instructions to represent the branch conditions, and use a `tauand` instruction to combine them, as shown in Figure 8(b). Note that a `tauand` always combines `tau` values along a single control-flow path.

3.2.3 Safety past extended basic blocks: `phi`

As described earlier, `taus` are SSA operands. As with any other SSA operands, they may be combined at merge points in the control flow via `phi` instructions. In this manner, we can correctly represent the safety dependence of a dangerous instruction on separate checks from different paths leading to that instruction. The compiler uses `phi` instructions to (1) allow precise safety dependence information in the case of a check eliminated by a global analysis (such as array bounds check elimination analysis), and (2) preserve correct safety

⁵Many typical “redundant bounds check” analyses, such as the ABCD algorithm we use, can discover this.

```

I1: tau1 = checknull t1
I2: tau2 = checktype t1, int[]
I3: t2 = length t1, tau1, tau2
I4: if (t0 < 0) goto L1 → L1
I5: if (t0 >= t2) goto L1 → L1
I6: t3 = ldelemaddr t1, t0, int[]
I7: tau3 = checkbounds t0, t2 → handler
I8: t5 = ldind [t3], tau1, tau2, tau3

```

(a) Check redundant on multiple conditions.

```

I1: tau1 = checknull t1
I2: tau2 = checktype t1, int[]
I3: t2 = length t1, tau1, tau2
I4: if (t0 < 0) goto L1 → L1
I9: tau3 = tauedge
I5: if (t0 >= t2) goto L1 → L1
I10: tau4 = tauedge
I6: t3 = ldelemaddr t1, t0, int[]
I7: tau3 = checkbounds t0, t2 = tauand tau3, tau4
I8: t5 = ldind [t3], tau1, tau2, tau3

```

(b) Redundant check replaced by tauand

Figure 8: Use of tauand

```

t0 = newobject sometype → handler
tau1 = checktype t0, sometype → handler
tau2 = checknull t0 → handler
t1 = ldfla t0, sometype::x
t2 = ...
t5 = ldind [t1], tau2, tau1

```

(a) Unconditionally safe check

```

t0 = newobject sometype → handler
tau3 = tausafe
t1 = ldfla t0, sometype::x
t2 = ...
t5 = ldind [t1], tau3, tau3

```

(b) Check replaced by tausafe

Figure 9: tausafe represents global safety

dependence information in the case of code duplicating code transformations (such as tail duplication, loop peeling, etc.).

3.2.4 Eliminating globally valid checks: tausafe

In some cases, a safety check can be eliminated based on other value flow. For example, a `new` operation in Java bytecode is guaranteed to return a non-`null` reference to an object of a particular type or to throw an exception. Once the corresponding IR operation `newobject` has completed, returning an object reference, subsequent use may safely rely on the reference not being `null`. A load from this object needs no checks to be certain that it will not fault.

Figure 9(a) shows such an example, where the `checknull` and the `checktype` will always succeed. In Figure 9(b) the check results are replaced by a `tausafe` instruction, signifying that the operation is safe due to value dependences. The operation is globally safe. The `tausafe` instruction itself has no effect and can be freely moved or reordered. `tausafe` is the unity for the `tauand` operation.

3.2.5 Check elimination based on method interfaces: tauhastype, taunonnull

When a check is eliminated based on the properties of a formal parameter or a value returned from a called method, we are first tempted to use a `tausafe` operation, but may not do so if there is any possibility that a later method inlining pass might expose the method body, allowing code motion to or from the call site. We make the use of interface dec-

```

public static sometype Foo(sometype a) {
    ...
}

```

(a) Static method declaration

```

t0 = invokestatic Foo → handler
tau1 = checknull t0 → handler
tau2 = tauhastype t0, sometype
t1 = ldfla t0, sometype::x
t2 = ...
t5 = ldind [t1], tau1, tau2

```

(b) Unambiguous call site return type is known

Figure 10: Returned-type redundant check

```

class String {
    ...
    public String substring(int begin);
    ...
}

```

(a) Declaration of substring

```

t0 = arg0 // this
tau1 = checknull t0 = taunonnull t0
tau2 = tauhastype t0, String
t1 = ldfla t0, String::data

```

(b) substring IR prologue

```

taux = checknull x
tauy = checktype x, String
call x.substring(..), taux, (tauy, ...)

```

(c) Site of call to substring

Figure 11: Parameter-dependent safety

larations explicit with the `tauhastype` and `taunonnull` instructions. Static method `Foo` of Figure 10(a) is declared to return a reference of type `sometype`, so any call to `Foo` must return a (possibly `null`) reference to `sometype`. Figure 10(b) shows a call site where a `checktype` has been replaced by a `tauhastype` operation to represent this dependence.

Another case occurs with formal parameters to a method: each parameter may be assumed to have its declared type, and the `this` parameter may be assumed to be non-`null`. Any dangerous instruction whose safety depends on these properties must use a tau value produced by the `taunonnull` or `tauhastype` applied to the corresponding formal parameter. For example, given a method `substring` as declared in Figure 11(a), the implementation IR might look like Figure 11(b). If the method is ever inlined, then the `taunonnull` and `tauhastype` safety dependences must be replaced by corresponding tau operands to the call instruction, as at the callsite shown in Figure 11(c). When inlining, the `taux` parameter is copied to any `taunonnull t0` operations in `substring`, and the type check parameter `tauy` replaces any corresponding `tauhastype`.

When the compiler has finished all inlining, any remaining instructions `taunonnull` and `tauhastype` applied to parameter values may be converted to `tausafe`, since the specified properties are guaranteed to hold throughout the method.

3.2.6 Effect on checks of branch folding: tauguard

In some cases, program transformations which eliminate a safety check based on branches may be spread across several compiler passes. In this case, it is particularly important that the compiler IR retains enough information to allow the right reason (tau value) to be substituted for the eliminated

```

I0: if (t0 != 1) goto L1 → L1
I1: if (t1 != t0) goto L2 → L1
I2: tau1 = checkzero t1 → handler
I3: t4 = div t2, t1, tau1

```

(a) Initial IR

```

I0: if (t0 != 1) goto L1 → L1
I1: if (t1 != 1) goto L2 → L2
I2: tau1 = checkzero t1 → handler
I3: t4 = div t2, t1, tau1

```

(b) IR after context-based constant folding

```

I0: if (t0 != 1) goto L1 → L1
I1: if (t1 != 1) goto L2 → L2
I2: tau1 = checkzero 1 =tausafe? NO
I3: t4 = div t2, t1, tau1

```

(c) Illegal check elimination

Figure 12: Motivation for tauguard

```

I0: if (t0 != 1) goto L1 → L1
I4: tau2 = tauedge
I5: t3 = tauguard tau2, 1
I1: if (t1 != t3) goto L2 → L2
I2: tau1 = checkzero t1 → handler
I3: t4 = div t2, t1, tau1

```

(a) Context-dependent optimization with tauguard

```

I0: if (t0 != 1) goto L1 → L1
I4: tau2 = tauedge
I5: t3 = tauguard tau2, 1
I1: if (t1 != t3) goto L2 → L2
I6: tau3 = tauedge
I7: tau4 = tauand tau2, tau3
I2: tau1 = checkzero t1 = tau4
I3: t4 = div t2, t1, tau1

```

(b) tauguard-enabled check elimination

Figure 13: Safe check elimination with tauguard

check. (This sort of thing happens more frequently in a dynamic compiler, which may apply additional optimizations to a “hot” method after the program has run for a while.)

In Figure 12(a), a `checkzero` instruction validates the divisor operand to a dangerous division operation. A constant-propagation pass which takes advantage of branch information might observe that `t0` is always 1 at instruction I1, and transform the code to Figure 12(b). At some later point we may decide that the method is hot and do some additional optimization. A later round of clever constant propagation might see that `t0` is always 1 at instruction I2. At this point the `checkzero` will never fail, but we cannot indicate that the `div` operation is always safe, lest we unsafely hoist it above I0. We need to replace `tau1` by the reason why the check is known to be safe, and that information was lost.

The `tauguard` instruction addresses this problem. Any optimization which replaces one value by another based on control flow transfers (branches or checks) introduces a `tauguard` operation, tying the optimized value to a tau value representing the safety dependences of the transformation. To perform context-based constant folding as in the example, a `tauguard` must be introduced to constrain use of that value to the region whose context makes it valid, as shown in Figure 13(a). Later optimization passes see `t3` as a constant at instruction I1, but guarded by the tau value `tau2`. They may conclude at instruction I2 that `t1` is also a constant, but guarded by both `tau2` and a new tau value `tau4` representing I1’s fall-through case. A new tau value `tau4`

<i>checks</i>	define a tau-valued result operand
<i>dangerous instrs</i>	additional tau-valued operands
<i>calls/return/stores</i>	additional tau-valued operands prove type properties of operands/results
<i>phi</i>	also handles tau-valued operands
<i>tau-generating instructions:</i>	
tausafe	always safe: uses are freely moveable
taupoint	unknown dependence: can never move
tauand tau2, tau3	depends on both tau2 and tau3
tauedge	depends on preceding branch (<code>tauedge</code> is tied to the edge)
tauastype t1,Type	depends on interface guarantee that t1 is of a subclass of Type
taunonnull t1	depends on interface guarantee that t1 is nonnull
<i>guarded optimized value:</i>	
t1=tauguard tau1, t2	any dangerous instruction using t1 also depends on tau1 for its safety.

Figure 14: Summary of IR Changes

represents the conditions guaranteeing that the `checkzero` will succeed, and eliminate the check in Figure 13(b). The resulting code is safe: the dangerous `div` operation cannot be hoisted above the branches which make it safe.

3.2.7 Unknown safety dependences: taupoint

In some cases, the safety dependences of a dangerous instruction are unknown or are too difficult to represent precisely. To accommodate these cases, we use the `taupoint` instruction, which indicates the dependence of a dangerous instruction on its placement at particular point in the code. The `taupoint` instruction may not be moved, so a dangerous instruction using the value produced by it must remain in the region dominated by its original location. This prevents a new fault from being introduced onto any program path.

For Java programs, our compiler does not require `taupoint` operations as it is able to preserve the safety constraints from the original program. However, it does allow us to model unsafe computation in our IR. From an engineering perspective, we have also found it useful during the development and debugging of optimizations.

3.3 From IR to Optimizations

To represent constraints on code motion imposed by fault safety involves a small set of changes to a typical dynamic compiler IR, summarized in Figure 14. In the following sections, we show how tau operands can be used to perform safe speculative code motion. Each code motion algorithm can rely on the fact that a dangerous computation is fault-safe wherever all its tau arguments are available.

4. PARTIAL REDUNDANCY ELIMINATION

Traditional code motion techniques based on PRE, such as LCM [19] and SSAPRE [18], consider only down-safe insertions to eliminate partial redundancies. This ensures correctness, but precludes speculative placements such as that of Figure 1(b). Previous work on speculative PRE [20] limits speculation to *safe* computations or relies on hardware support [2]. By using fault-safety to constrain placement of expressions, it is simple to extend speculative PRE to dangerous expressions such as field or array loads and divides, without such hardware support.

We started with an implementation of the work-list driven version of SSAPRE [18] in our compiler, then modified it to

respect fault safety via tau variables. It works on one expression at a time and handles compound expressions by processing each sub-expression incrementally. To track memory dependences, we use an SSA-based representation similar to HSSA [9]. We leverage the type and context information available in Java [12, 11] to obtain more precise alias information and further refine our memory representation.

4.1 Modified SSAPRE algorithm

To support speculative code motion within the SSAPRE framework, we extended the *DownSafety* [20] step in the SSAPRE algorithm. For each expression, SSAPRE represents potential insertion points as Φ -nodes and then determines which of these insertion points are down-safe. Speculation essentially entails marking some non-down-safe insertion points as down-safe. We calculate profitable places for speculative code motion using the *conservative speculation* heuristic from [20]. We only hoist loop-invariant expressions.

To expand opportunities for hoisting, we detect store-to-load redundancies to handle definitive updates (like `a.x++`) inside a loop body and, when possible and profitable, hoist the load out of the loop.⁶ Preceding PRE, the optimizer performs superblock formation [16] to split loops into a hot-path inner loop and an outer loop with the cold-path. This enables code motion from the hot path even when killing instructions (e.g., method calls) occur along the cold path.

The algorithm treats the tau operands of loads as normal source operands. As PRE must already respect value dependences, it respects the safety dependences encoded by taus automatically. PRE will perform a safe placement due to an explicit dependence between the tau-defining instruction and the load instruction. We can automatically hoist loads that are provably safe (e.g. by `tauedge`) speculatively.

The Φ -nodes marked by the speculation are considered for insertion in the following code motion steps in addition to the down-safe points selected by standard PRE.

Handling of Check Instructions

Fully redundant checks are eliminated before PRE, as outlined in Section 3. If a load is still guarded by an immediately preceding check, our ability to move the load is limited: the check may throw an exception and acts as a barrier to the load. For those loads, we are limited to down-safe placements.

We extend the range of code motion by applying PRE to check operations as well. However, we do not allow speculative insertions for tau-generating instructions (`checknull`, `checkzero`, `checktype`) as this would introduce a potential exception onto a path where it did not exist before. We optimize partially redundant check instructions only as long as we perform only down-safe insertions and do not violate Java’s precise exceptions [15]. After hoisting check instructions, we update the control-flow graph, the dominator tree and the SSA form accordingly to reflect the changes in the (exceptional) control flow.

4.2 Using Taus for Speculative Code Motion

Figure 15 illustrates our approach to speculative code motion. This code is a partially lowered form of our earlier example in Figure 3 (for clarity we leave the `for` loop header in

⁶At this time, we do not perform partial dead store elimination to sink stores and obtain the full benefit of register promotion [20].

<pre> if (a != null) { t2 = tauedge for (i = 0; i < n; ++i) if (i & mask) { t3 = ldflda a, A::x t4 = ldind [t3], t2 t5 = t4 + b k += t5 } } </pre>	<pre> t3 = ldflda a, A::x if (a != null) { t2 = tauedge t4 = ldind [t3], t2 t5 = t4 + b for (i = 0; i < n; ++i) if (i & mask) k += t5 } </pre>
<p>(a) Partially redundant load with safety information</p>	<p>(b) Optimized version: Safe motion of memory access</p>

Figure 15: Speculative PRE with taus moves load out of loop, address calculation even further

source form). We can hoist the indirect load (`ldind`) out of the inner loop because it is guarded by the tau variable generated after `a!=null` by the `tauedge`. Note that the address calculation (`ldflda`) can be moved outside the outer branch because it is a safe (arithmetic) operation. The heuristic that drives speculation always tries to hoist expressions to the outer-most loop level. Each definition of a tau variable defines the region in the CFG where it is safe to place any instruction that consumes that tau variable as an operand (in our case the load `a.x`). This way we achieve a profitable placement that is provably safe.

An operation may depend on multiple safety conditions. Our representation of safety allows PRE to move a load operation to any point where all of its tau operands are available (and value/memory dependences are not violated).

The key to our technique is the explicit representation of safety information in a form that the compiler can reason about. Without tau variables, a compiler could still eliminate checks, and would know that each load was non-faulting (as the input language is safe), but would not know why, so would not be able to safely move the load above any branch.

4.3 Post-passes to PRE

Our experimental PRE pass is unaware of architectural features such as the number of physical registers and the latency of memory operations. We explored two optional post-passes which consider such factors.

The *address resinking* post-pass addresses register pressure: it is well known that redundancy elimination increase register pressure by increasing the live range of variables. This post-pass sinks address calculations back into loops whenever we fail to hoist the corresponding load operation. As address calculations are always safe, this post-pass does not need to be aware of fault-safety.

The *early placement* post-pass addresses memory latency. Typical PRE passes place redundant computations at the latest point that is sufficient to eliminate any redundancy, reducing register pressure. However, load operations frequently have extremely high latency. As an early placement is not computed naturally in the general SSAPRE algorithm, we use this post-pass to place loads (and instructions that they depend upon) earlier in the control flow. The placement relies upon profile information to ensure that it does not move computations to hotter paths. It also relies upon fault-safety to safely place loads above branch points.

5. INSTRUCTION SCHEDULING

In this section, we describe modifications to an instruction scheduler to leverage fault safety for safe speculation.

```

I0: (p0, p1) = cmp t1, 0      I2: t3 = t2 + 16
I1: (p0) br B1               I0: (p0, p1) = cmp t1, 0
I2: t3 = t2 + 16            I1: (p0) br B1
I3: t4 = ld [t3]             I3: t4 = ld [t3]

```

(a) Code before scheduling (b) Code after scheduling

Figure 16: Scheduling with unsafe load

```

I4: tau1 = tauedge           I4: tau1 = tauedge
...
I0: (p0, p1) = cmp t1, 0     I2: t3 = t2 + 16
I1: (p0) br B1               I3: t4 = ld [t3], tau1
I2: t3 = t2 + 16            I0: (p0, p1) = cmp t1, 0
I3: t4 = ld [t3], tau1     I1: (p0) br B1

```

(a) Code before scheduling (b) Code after scheduling

Figure 17: Safe load can be hoisted above a branch

An instruction scheduler reorders instructions in a program region, aiming to improve instruction-level parallelism. High latency, critical path instructions are placed early, so that their execution can overlap that of other instructions. A global instruction scheduler operates on a region containing multiple basic blocks such as a superblock, a trace, a hyperblock or an arbitrary program region.

A global instruction scheduler’s ability to reorder instructions within a region is constrained by instruction safety, which is typically approximated by down safety, effectively assuming that a dangerous instruction may depend on any branch. Such an assumption severely limits the scheduler’s ability to reorder instructions. As an example, consider the Itanium® Processor Family (IPF) code fragment shown in Figure 16. I0 compares `t1` with `null`, setting `p0` to true or false (and `p1` to the complement). The predicated branch instruction I1 jumps to B1 if the preceding compare succeeds. I2 is an address calculation, and I3 is a load.

An instruction scheduler would ideally place the high-latency load instruction I3 before the compare I0 and branch I1, but it cannot do so in general as speculative execution of a load might result in a fault. With traditional scheduling, the load I3 is still executed last. Under certain circumstances, the scheduler may conclude that the load is safe (for example, when load is moved into a control-equivalent block, or is dominated by another load from a related address), but such situations are not common enough to enable general hoisting of loads.

Safe speculative instruction scheduling uses our more precise representation of fault safety to enable code motion of dangerous instructions above some branches. As with PRE in the previous section, it uses `tau` operand dependencies to constrain the ordering of branches and loads. With safe speculative instruction scheduling, a load is independent of any preceding branch unless such a dependence is explicitly encoded in the IR. Figure 17(a) shows the code fragment from Figure 16(a) augmented with the load safety information. Here load I3 has an explicit argument `tau` defined by an earlier `tauedge` instruction I4. With this information, the scheduler knows that load I3 does not have a dependency on branch I1 and may hoist it above the branch as shown in Figure 17(b).

Some computer architectures provide hardware support for speculatively moving loads above the branches. For example, IPF features hardware support for control speculation [21]. A special *speculative load* instruction (`ld.s`) does

```

I2: t3 = t2 + 16              I2: t3 = t2 + 16
I3: t4 = ld.s [t3]           I0: (p0, p1) = cmp t1, 0
I0: (p0, p1) = cmp t1, 0     I3: (p1) t4 = ld [t3]
I1: (p0) br B1               I1: (p0) br B1
I5: chk.s t4, RecoveryCode
Next:
...
RecoveryCode:
I6: t4 = ld [t3]
I7: br Next

```

(a) Control speculation (b) Partial predication

Figure 18: Load speculation using hardware support

not fault when the address operand is invalid, but instead sets a special NAT (not-a-thing) bit on the result register to indicate failure of the load. NAT bits are propagated through most IPF instructions (e.g., arithmetic and loads), allowing the compiler to build chains of speculative execution. Eventually, a speculative check instruction (`chk.s`) transfers control flow to a given address if its source register has a NAT bit set. Figure 18 shows how the load I3 in Figure 16 can be hoisted above the branch I1 using control speculation. In addition to the now speculative load I3, the code contains a speculative check instruction I5 that transfers control to recovery code when speculation fails. The recovery code re-executes speculative computation found to be invalid (in this case a single load I6) and transfers control back to the main flow of execution.

Control speculation has several drawbacks compared to safe speculation. First, it requires speculative check instructions and recovery code, increasing code size (including the code size of the hot code due to check instructions) and consuming additional resources (such as address registers used in recovery code). Second, `ld.s` instruction may fail even when it happens to be not speculative—for example, due to a page miss. Safe speculative scheduling avoids these overheads, although its applicability is more limited. The benefits of both are obtained by eliminating safe control dependencies using safe load speculation and then breaking the remaining ones using hardware-supported control speculation.

Another way to move a load above a branch using hardware support is partial predication [3]. Figure 18(b) illustrates a load I3 moved above branch I1 by predicating it on the complimentary predicate `p1`. The limitation of partial predication is that it does not allow a load to move above the compare instruction (in this example, instruction I0).

We implemented safe speculative instruction scheduling as part of a trace scheduling algorithm [13]. The baseline scheduler operates on a low-level IR similar to machine code for the IPF architecture. It uses both control speculation [21, 2] and partial predication [3] to speculatively hoist loads above the branches.

We extended the scheduler to leverage fault-safety using our existing scheduling heuristics. As with the higher-level IR in Sections 3 and 4, the scheduler represents safety dependencies using abstract `tau` instructions and virtual registers in IPF load instructions.⁷ When the scheduler decides to speculate a dangerous instruction to a certain point, it first determines whether that instruction is fault-safe at that

⁷The low-level IR currently has safety information only for loads. Other potentially faulting instructions are constrained by down-safety.

I0: (p0, p1) = cmp t1, 0	I7: t3 = t2 + 16
I1: (p0) br B1	I2: (p2, p3) = cmp t1, 1
...	I8: (p3) t4 = ld [t3], tau1
I2: (p2, p3) = cmp t1, 1	I0: (p0, p1) = cmp t1, 0
I3: (p2) br B2	I1: (p0) br B1
I4: tau1 = tauedge	...
...	I3: (p2) br B2
I5: (p4, p5) = cmp t1, 2	...
I6: (p4) br B3	I5: (p4, p5) = cmp t1, 2
I7: t3 = t2 + 16	I6: (p4) br B3
I8: t4 = ld [t3], tau1	

(a) Code before scheduling (b) Code after scheduling.

Figure 19: Hoisting a load using combination of safe speculation and partial predication

point. If it is, then hardware speculation support can be omitted. If not, it uses either control speculation or partial predication as directed by the heuristics.

Figure 19 illustrates the utility of the `tauedge` instruction for enabling safe hoisting after partial predication. In this example, load `I8` is preceded by 3 branches: `I1`, `I3` and `I6`. The `tauedge` instruction `I4` placed between branches `I3` and `I5` indicates that the load depends only on branch `I3`. Figure 19(b) shows the code after scheduling. Because load `I8` depends only on a single branch the scheduler can use partial predication to break the dependency. It predicates load `I8` on the complimentary predicate `p3` to hoist it above branch `I3`. It then hoists load `I8` even further up, before branch `I1` and compare `I0`. This is possible because instruction `I4` pinpointed the exact dependence between the load and just one of its preceding branches.

6. EXPERIMENTAL RESULTS

In this section, we study the performance effects of fault-safe code motion algorithms on the standard Java SPEC JVM98 benchmark suite [25]. Our platform is the ORP Java virtual machine [10] and the StarJIT dynamic compiler [1] running on a 4-processor, 1.4 GHz Itanium® 2-based system with 6 MB L3 cache and 8 GB physical memory running Windows Server 2003, 64-bit edition. .

Our system compiles each method immediately with an initial set of optimizations and with method and edge instrumentation. This initial set of optimizations is fairly extensive. It is applied after Java bytecodes are lowered into the machine independent representation shown in Section 3 and includes dead and unreachable code elimination, dominator tree-based dead common subexpression elimination (including redundant null/type check elimination), dominator tree-based redundant load-store elimination, guarded devirtualization of virtual calls, type and copy propagation, constant folding, and redundant branch merging. Following these optimizations, the code is lowered to IPF instructions and a fast basic block-level instruction scheduling is performed. No speculative code motion is performed during initial compilation.

Frequently executed methods are compiled a second time, with the collected profile information annotated onto the program representation. The initial optimizations are reapplied, then a set of expensive, profile-guided optimizations. These include (traditional) partial redundancy elimination, array bounds check elimination, hot path splitting, method inlining, operator strength reduction, and loop peeling. At the machine level, speculative (traditional) trace-based instruction scheduling is performed that uses IPF’s control

speculation features to hoist load operations above down-safe points and to insert compensation code when speculation fails.

For our experiments, we modified our recompilation pass to apply our fault-safety-aware variants of SSAPRE and the speculative IPF instruction scheduler. To factor out the costs of compilation and instrumentation⁸, we run each JVM98 benchmark three times within the same JVM instance and measure the execution time of the third run. By the third run, the vast majority of recompilation has already taken place.

In Figure 20(a), we show the effect of our optimizations on the SPEC JVM98 suite. For each benchmark, the first bar represents the normalized execution time of the standard optimization path, described above. This baseline is already heavily optimized. It includes speculative SSAPRE (bounded by down-safety for loads) and speculative trace-based instruction scheduling that already leverages IPF’s built-in control speculation and predication support. The remaining bars are normalized against the first. The second bar (+ FS PRE) represents the effect of using fault-safe speculation in SSAPRE over the baseline. The third bar (+ FS IS) represents the effect of using fault-safe speculation in the instruction scheduler over the baseline. The last bar (+ Both) represents the cumulative effect of both.

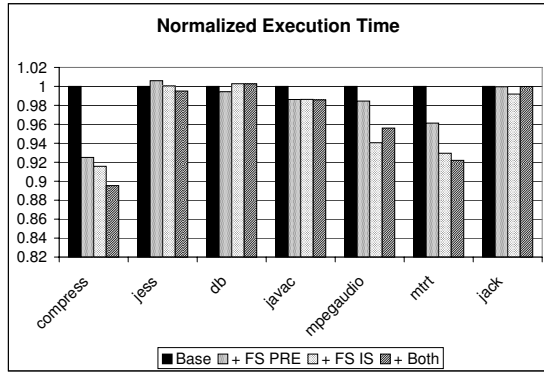
We measured no appreciable benefit to fault-safe speculation in five of the SPEC JVM98 benchmarks. In the remaining three, however, the effects were significant. For `compress`, we see a 7% benefit from FS SSAPRE, an 8% benefit from FS scheduling, and a 10.5% overall benefit. For `mpegaudio`, we see a 2% benefit from FS SSAPRE, a 6% benefit from FS scheduling, and a 4.5% overall benefit. Finally, for `mtrt`, we see a 4% benefit from FS SSAPRE, a 7% benefit from FS scheduling, and an 8% benefit overall.

In Figure 20(b), we isolate the benefits of the FS PRE post-passes described in Section 4. The first and last bars are the same as the first and second bars of Figure 20(a). The second bar includes FS PRE with no post-passes. The third adds address resinking, and the fourth adds early placement of loads. Interestingly, FS PRE alone (i.e., with a latest placement) helps only `compress`. Address resinking helps as much as it hurts (not surprising as IPF has a wealth of registers), but early placement is particularly effective.

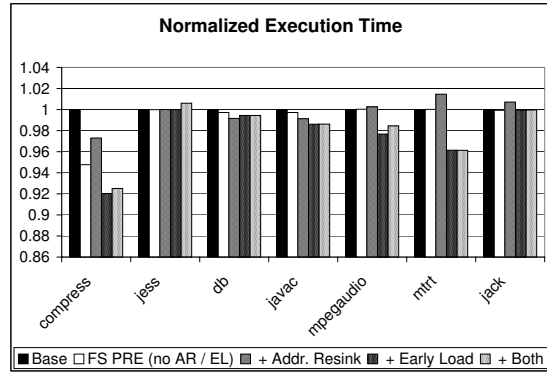
In Figure 20(c), we isolate the benefits of fault-safety on the IPF scheduler. In particular, we compare the effects of fault-safe speculation (FS IS) with no hardware support against hardware-based control speculation (CS) and partial predication (PP). In this graph, the fourth and sixth bars are the same as the first and third bars in Figure 20(a). The first bar is a lowered baseline that disables usage of the IPF hardware features. The effects of partial predication alone (the second bar) are modest. Control speculation, on the other hand, is quite effective. In some cases, however, fault-safe speculation is able to best hardware-based control speculation.

In Figure 20(d), we again measure the overall effects of fault-safe speculation, but this time we use the lowered baseline with no hardware speculation support. In this case, we see performance benefits across the board demonstrating that hardware speculation masks some of the benefits of fault-safe speculation.

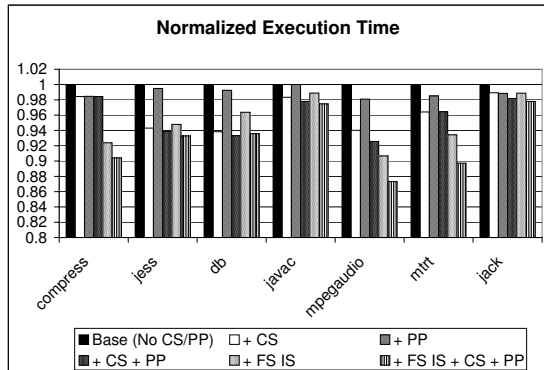
⁸The effects of our fault-safety-aware variants on compilation time appears to be negligible.



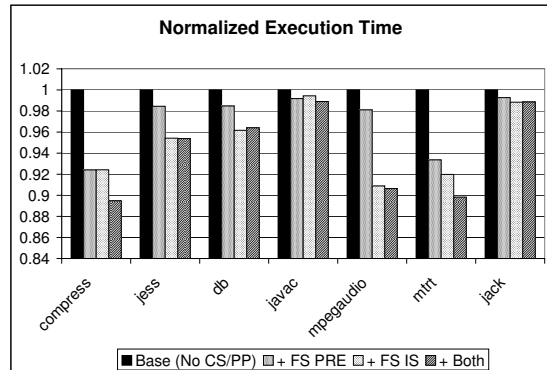
(a) Fault-safety-aware speculative optimization



(b) Fault-safety-aware partial redundancy elimination



(c) Fault-safety-aware scheduling vs. control speculation and partial predication



(d) Effects of fault-safety-aware optimizations without IPF control speculation and partial predication

Figure 20: Experimental results

The benefits of fault-safe speculation in both PRE and instruction scheduling overlap to some extent. This is not surprising as both code motion optimizations have similar effects on the many instructions. However, the two are also complementary. Fault-safety-aware SSAPRE is performed at the machine-independent level and is not aware of IPF memory latencies and available instruction slots. Fault-safety-aware instruction scheduling, on the other hand, is performed only at the level of instruction traces and cannot move code out of loops as in Figure 3. In some benchmarks, we see a cumulative effect from both.

7. RELATED WORK

Classical partial redundancy elimination [23] and its variants such as LCM [19] and SSAPRE [18] perform a computational optimal placement of expressions constrained by down-safety. Lo, et al. [20] extended SSAPRE further to handle load expressions and stores as well. Their work also performs speculative code motion for inherently safe operations including arithmetic expressions or direct loads by relaxing down-safety. We adopted the heuristics from this work to decide when to perform speculative insertions. Our approach generalizes this idea by also allowing speculation for unsafe operations like indirect loads.

There are several other profile-guided algorithms for eliminating redundant expressions using edge-profiles [5, 6, 24]. Xue et. al. [26] present a computationally and life-time op-

timal placement for speculative PRE. These approaches are orthogonal to our work as they focus on code motion for arithmetic expressions.

Safe languages like Java offer interesting opportunities and challenges for the compiler because they guarantee type safety [14]. On the one hand, they allow more precise alias analysis [12, 7] based upon types that, in turn, can permit greater freedom to reorder memory operations. This is an important but orthogonal benefit to fault safety that we utilize in our base system.

On the other hand, there are also additional constraints imposed by precise exception semantics. Because of exceptional control flow, computations that would have been considered down-safe in the context of C or Fortran are no longer. Gupta, et. al. [15] discuss how Java’s precise exception model affects typical optimizations. They describe how to circumvent some of these constraints by generating compensation code for the exceptional case, and show that eliminating those dependencies allows the optimizer to be much more aggressive. Chambers, et. al. [7] describe dependence analysis in the presence of such a precise Java exception representation. Their method uses condition registers, which are similar in spirit to our tau variables but are used in a much more limited context—that of extended basic blocks. In particular, they have no equivalent to most of our tau-generating instructions that permit a global notion of fault safety.

Arnold, et. al. [2] discuss instruction scheduling in the context of Java, and make the key observation that Java's static and runtime checks prevent loads from actually faulting. They propose a general percolation strategy that requires the hardware to suppress all faults, and, thus, permits dangerous instructions to be placed speculatively. They show significant performance advantages of this approach. Their approach does require specialized hardware support, and it does not easily allow Java code to be mixed with native code.

Kawahito, et. al. [17] describe an approach to the same high-level problem of enabling safe speculative code motion in type-safe languages like Java. Their work is more ad-hoc, treating check instructions specially, and requires "dummy" checks to be inserted to mark safety dependence sources when checks are eliminated. Indirect representation of safety dependence through the operands of check and load/store instructions prevents the separation and independent optimization of address calculations. The special treatment of exception instructions makes it difficult to envision adapting PRE to handle dangerous instructions. Since their presentation only describes null check removal in detail, it is difficult to tell if it is compatible with certain aggressive bounds check elimination techniques, or even certain kinds of simple type check elimination.

Finally, there has been much work to model safety information in a compiler's IR primarily for program verification, including our previous paper [22]; please see that paper for more related work in this area.

8. CONCLUSION

We have presented a novel framework for speculative motion of dangerous instructions such as load operations to memory. Our framework is able to leverage the properties already provided by a safe language such as Java to perform speculation with no hardware or system support.

We have demonstrated how two different compiler optimizations can be adapted to exploit our framework for code motion: a partial redundancy elimination (PRE) algorithm and an Itanium® instruction scheduling algorithm. In both algorithms, our framework allows code motion of potentially faulting instructions that is not constrained by traditional notions of safety. Experiments demonstrate performance benefits of up to 10% on standard Java benchmarks.

In this work, we demonstrate that safe language compilers can perform optimizations that are not possible in unsafe languages such as C. Moving forward, we believe that researchers and implementers should rethink classical notions of safety as they adapt and develop optimizations for safe languages.

9. REFERENCES

- [1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. S. Menon, B. R. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1), February 2003.
- [2] M. Arnold, M. Hsiao, U. Kremer, and B. Ryder. Exploring the interaction between java's implicitly thrown exceptions and instruction scheduling. *International Journal of Parallel Programming*, Fall 2000.
- [3] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. mei W. Hwu. Integrated predicated and speculative execution in the impact epic architecture. In *ISCA 1998*.
- [4] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI 2000*.
- [5] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *PLDI 1998*.
- [6] Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *CGO 2003*.
- [7] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan. Dependence analysis for Java. In *LCPC 1999*.
- [8] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *PASTE '99: Program analysis for software tools and engineering*.
- [9] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Sixth International Conference on Compiler Construction (CC'96)*.
- [10] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003.
- [11] K. D. Cooper and L. Xu. An efficient static analysis algorithm to detect redundant memory operations. In *ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*.
- [12] A. Diwan, K. McKinley, and E. Moss. Type-based alias analysis. In *PLDI 1998*.
- [13] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30:478-490, July 1981.
- [14] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [15] M. Gupta, J.-D. Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In *ECOOP 2000*.
- [16] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective structure for vliw and superscalar compilation. Technical report, Center for Reliable and High-Performance Computing, University of Illinois, February 1992.
- [17] M. Kawahito, H. Komatsu, and T. Nakatani. Partial redundancy elimination for access expressions by speculative code motion. *Softw. Pract. Exper.*, 34(11):1065-1090, 2004.
- [18] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627-676, 1999.
- [19] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *PLDI 1992*.
- [20] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. *ACM SIGPLAN Notices*, 33(5):26-37, 1998.
- [21] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Trans. Comput. Syst.*, 11(4):376-408, 1993.
- [22] V. S. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A.-R. Adl-Tabatabai, and L. Petersen. A verifiable SSA program representation for aggressive compiler optimization. In *POPL2006*.
- [23] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96-103, Feb. 1979.
- [24] B. Scholz, N. Horspool, and J. Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *LCTES '04: Languages, compilers, and tools for embedded systems*.
- [25] Standard Performance Evaluation Corporation. SPEC JVM98, 1998. See <http://www.spec.org/jvm98>.
- [26] J. Xue and Q. Cai. A lifetime optimal algorithm for speculative pre. *ACM Trans. Archit. Code Optim.*, 3(2):115-155, 2006.