Eliminating Exception Constraints of Java Programs for IA-64

Kazuaki Ishizaki, Tatsushi Inagaki, Hideaki Komatsu, Toshio Nakatani
IBM Research, Tokyo Research Laboratory
ishizaki@trl.ibm.co.jp

Abstract

Java exception checks are designed to ensure that any faulting instruction causing a hardware exception does not terminate the program abnormally. These checks, however, impose some constraints upon the execution order between an instruction potentially raising a Java exception and a faulting instruction causing a hardware exception. This reduces the effectiveness of instruction reordering optimization. We propose a new framework to effectively perform speculation for the Java language using a direct acyclic graph representation based on the SSA form. Using this framework, we apply a well-known speculation technique to a faulting load instruction to eliminate such constraints. We use edges to represent exception constraints. This allows us to accurately estimate the potential reduction of the critical path length for applying speculation. We also propose an approach to avoid extra copy instructions and to generate efficient code with minimum register pressure. We have implemented the technique in the IBM Java Just-In-Time compiler, and observed performance improvements up to 25% for micro-benchmark programs, up to 10% for Java Grande Benchmark Suite, and up to 12% for SPECjvm98 on an Itanium processor.

1 Introduction

The type safety feature of Java language calls for its bytecode to make quite a few runtime exception checks in order to eliminate error-prone situations. These runtime checks ensure that a Java program cannot execute unsafe operations that may cause the program to be terminated abnormally. Exceptions in Java programs can be categorized into two types: *hardware exceptions* and *Java exceptions*. Hardware exceptions are those which the hardware may throw as the result of program execution such as segmentation faults and invalid operations. Java exceptions are those software exceptions which Java bytecode may throw such as NullPointerException and IndexOutOfBoundsException. An instruction that may throw an exception is called a *potentially excepting instruction* (PEI) [1]. Furthermore, we call the instruction a *hardware-initiated potentially excepting instruction* (H-PEI) if it can cause a hardware exception. Although a hardware-initiated exception does not occur in normal Java programs, it may occur when a compiler applies speculation. In contrast, we call the instruction a *software-initiated potentially excepting instruction* (S-PEI) if it may throw a Java exception as the result of its execution. A Java exception is always recoverable in the sense that it can be caught by a catch clause in the Java program. It must be thrown before any hardware exception occurs. A load instruction that may cause a hardware exception by reading an invalid effective address or a TLB miss, called a *faulting load instruction* [2] (one of the H-PEIs), cannot be scheduled across an S-PEI. This limitation prevents instruction reordering optimizations from being applied in a wider region.

This paper focuses on the problem of preventing instruction reordering optimizations of Java programs on IA-64. We propose a technique to eliminate the constraints between S-PEIs and H-PEIs by using speculative code motion. Our approach depends upon special hardware features of IA-64 to support speculative code motion. Here, speculative execution refers to executing an instruction before execution is required. Two speculative execution techniques have been proposed. One is called *control speculation* [1, 2, 3, 4, 5, 6, 7] which is a technique to alleviate control dependence, between a branch and its succeeding instructions, by estimating the branch direction. Control speculation allows the compiler to aggressively schedule instructions across a branch. The other is called *data speculation* [8, 4, 6, 7] which is a technique to alleviate data dependence, between a store instruction and its succeeding load instructions, by estimating that the load will access a different memory location from the store. Data speculation allows the compiler to aggressively schedule instructions across a store. We propose *exception speculation* as a third speculation technique to alleviate exception dependence between the S-PEI and its succeeding H-PEIs by estimating that an exception does not occur when moving the H-PEI across the S-PEI. Exception speculation allows the compiler to aggressively schedule instructions across an S-PEI.

Our approach uses a *directed acyclic graph* (DAG) representation based on the *static single assignment* (SSA) form [9]. We use an edge in the DAG to represent an exception dependence relationship between each S-PEI and each of its succeeding H-PEIs [10]. We call this an *exception dependence edge*. Every exception dependence edge can be also included in the critical path length to estimate the gain when the target instruction is scheduled earlier across the S-PEI. After the decision is made to apply exception speculation, the faulting load instruction is replaced with the non-faulting load instruction to prevent it from throwing a hardware exception when it is scheduled earlier across the S-PEI.

Supporting exception speculation in an *intermediate representation* (IR) to distinguish exception speculation from control speculation yields the following two advantages:

1. Unlike Arnold's approach [11], which converts an S-PEI to a pair of compare and branch instructions and thus increases the number of control dependence edges for subsequent instructions, our approach needs only one exception dependence edge between an S-PEI and an H-PEI. This reduces the size of the IR as well as the number of

edges to be traversed for determining speculation and performing instruction scheduling. The compilation time can be reduced since it avoids global optimizations such as percolation scheduling or trace scheduling.

2. Unlike general percolation, by introducing exception speculation we can easily estimate the benefit of exception speculation along an exception dependence edge.

We also address two compilation issues for exception speculation: how to select a sequence of instructions, called a *speculative chain* [6], to move across an S-PEI, and how to generate the recovery code. We propose a selection mechanism for a speculative chain without inserting any additional copy instructions. We also propose code generation techniques to minimize the register pressure in the original code by saving registers killed in recovery code, and to minimize the impact on the code scheduling (bundle formation) phase by duplicating instructions in the recovery code.

We implemented our approach for eliminating exception constraints using the IBM Java Just-In-Time (JIT) compiler [12] for IA-64. We conducted experiments by running micro-benchmarks, and two industry standard benchmarks, the Java Grande Benchmark Suite and SPECjvm98. Our preliminary results show that exception speculation improves the performance, by up to 25% (with an average of 12.8%) for micro-benchmark programs, up to 10% (with an average of 1.7%) for Java Grande Benchmark Suite, and up to 12% (with an average of 2.3%) for SPECjvm98, with only a modest code size growth on an Itanium processor. In a few cases in the Java Grande Benchmark Suite and SPECjvm98, we observed a small performance degradation. We suspect that it is possibly due to the performance penalty caused by a small increase in TLB misses. Furthermore, we also observed that creating a partially exception-eliminated (safe) loop, using the *loop versioning* [12] technique, is also effective with exception speculation. In some cases, our results show that exception speculation increases the performance even after loop versioning has been applied. We also observed that our framework can save space for the DAG-based IR.

This paper makes the following contributions:

- A new framework to handle exception speculation using the SSA-based DAG representation.
- An efficient method to select speculative chains and generate the corresponding recovery code.
- Experimental results to validate the effectiveness of eliminating exception constraints in Java programs using IBM's Java JIT compiler for IA-64.

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 gives the system overview. Section 4 describes the technique to eliminate the exception constraints between S-PEIs and H-PEIs using exception speculation. Section 5 describes compilation issues for speculation and their solutions. Section 6 gives our experimental results. Finally, section 7 presents our conclusions.

2 Related Work

Ebcioglu et al. [13] proposed an out-of-order translation technique. It assumed architectural support for non-faulting load instructions. Since it renames a target register when the original load instruction is speculatively replaced with a non-faulting load instruction, an extra copy instruction must be inserted to recover the value in the original register at the original position of the corresponding load instruction. Extra copies increase the critical path length.

Le [2] also described a runtime binary translation technique that reorders instructions without any special architectural support. However, it requires the generation of some checkpoint code, which increases the critical path length even when no exception occurs.

Ju et al. [6] described a unified framework for control and data speculation on IA-64. Their target language is C, thus they do not deal with runtime exception speculation for a type-safe language. Their framework is similar to ours, in the sense that an SSA-based DAG is used and extra copy instructions are avoided. Their experiments were limited to an IA-64 architecture simulator.

Arnold et al. [11] described the impact of Java exceptions on a VLIW architecture based on superblock scheduling and a general percolation technique, by assuming architectural support for non-faulting load instructions. His experiments were also limited to a simulator.

Manish et al. [14] described some optimizations to reorder different S-PEIs or reorder an S-PEI with respect to an instruction affected by precise exception semantics using software solutions. We also propose a complementary technique to reorder S-PEIs and H-PEIs with an architectural support.

3 System Overview

This section describes an overview of the IA-64 architecture and our compiler. Section 3.1 describes the important architectural features which the compiler relies on. Section 3.2 describes the optimization framework for our compiler.

3.1 Architectural Features of IA-64

In this paper, we assume the following three architectural features to effectively support speculation using recovery blocks [5]:

Feature 1: A non-faulting load instruction that defers the exception and sets deferred bits to destination registers if the instruction causes hardware exceptions. When any instruction reads source registers with the deferred bits, it does not cause hardware exceptions but propagates the deferred bits.

Feature 2: A low-overhead instruction that checks whether the deferred bits are set in a register and if so, then executes the provided recovery code.

Feature 3: Many registers to hold the values the recovery code can use.

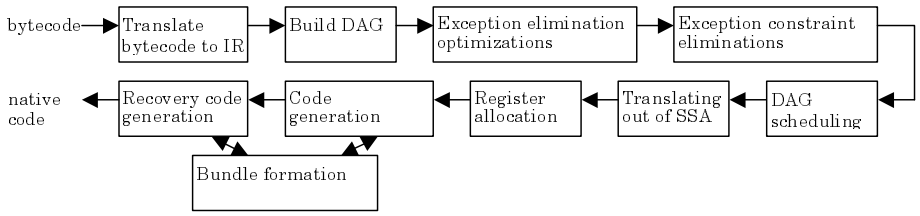


Figure 1. An overview of the JIT compiler

These features could also be implemented by software using the support of an operating system. However, the IA-64 architecture [15] supports the above features by hardware. For Feature 1, the IA-64 architecture provides non-faulting load instructions as `ld.s` instructions in the instruction set architecture. A *Not a Thing* (NaT) bit of the destination register is set if an `ld.s` instruction reads an invalid address or causes a TLB miss. Wehn the NaT bits of one or more source registers are set, the instruction propagates only the NaT bits to a destination register. For Feature 2, the IA-64 architecture provides a check instruction (we refer to it as `chk.s`) to quickly see whether a hardware exception has been deferred. If no exception has been deferred, the instruction takes no cycles. If an exception has been deferred, the execution branches to recovery code. For Feature 3, the IA-64 provides 128 general purpose registers and 128 floating point registers. Therefore, registers are rarely spilled out even if their liveness is extended by recovery code.

3.2 Optimization Framework

In Figure 1, we show an overview of our Java JIT compiler.

First, the compiler builds a DAG in the SSA form after it translates the bytecode to our IR. Second, it performs exception elimination optimizations such as loop versioning [12] and nullcheck elimination [16] to remove redundant runtime exception checks. The nullcheck elimination algorithm using forward dataflow and partial redundancy elimination removes most of the nullchecks for references of instance variables. Loop versioning elevates an individual array index exception check outside a loop by creating a safe loop and an unsafe (original) loop. The code for exception checks is generated at the entry of the loop by examining the whole range of the index within the loop. Therefore, all the array bound checks against the first dimension of the array are eliminated in the safe loop. As a result, many exception checks have already been eliminated in this phase. Although these optimizations are effective, the exception checks against the higher dimensions of arrays cannot be eliminated.

Third, it eliminates the exception constraints against remaining exception checks, as described in Section 4, and it performs DAG scheduling based on the list scheduling [17]. After it translates out of the SSA form, it allocates physical registers and generates the native code. In the code generation phase, the S-PEI is converted to native compare and branch instructions. Finally, it generates a recovery code, as described in Section 5, corresponding to each speculative chain.

4 Exception Speculation

In this section, we propose a technique to eliminate an exception constraint between an S-PEI and an H-PEI using speculation in order to reduce the critical path length. After eliminating the constraint and replacing the H-PEI with a non-faulting instruction, the non-faulting instruction and the succeeding instructions can be executed earlier than the S-PEI. We call this *exception speculation*.

Example 1 shows an example of our technique. It shows an IR and a DAG. Here, we assume that each `ld` instruction (N2 and N6) takes three cycles and all other instructions take one cycle each, except for the `chk` instruction (N9). A `chk` instruction takes 0 cycles. Unlike Arnold's work [11], our DAG representation does not generate a pair of compare and jump instructions corresponding to an S-PEI. Instead, we represent an S-PEI as a single instruction and add an exception dependence edge. This allows the compiler to accurately estimate the potential change of the critical path length. Exception speculation can eliminate all the exception dependence edges between S-PEIs and H-PEIs. Exception speculation consists of the following five steps:

Step 1: Decide whether a load instruction can be speculated: The compiler compares the two earliest times to initiate the execution of the load instruction. One time is constrained by data dependence, and the other is constrained by an exception dependence. If the time by exception dependence is longer, the compiler decides to speculate the instruction by replacing a faulting load instruction with a non-faulting one. This decision can avoid unnecessary speculation that executes recovery code caused by a small increase in TLB misses.

In Example 1, the time set for the data dependence edge at N6 is three, and the time set for the exception dependence edge at N6 is four. Therefore, N6 is selected as the candidate for speculation, and N6 is replaced with a non-faulting load instruction.

Step 2: Reconnect incoming exception dependence edges of a load instruction to a `chk` instruction: A check instruction is inserted at the original position of the corresponding load. Then, the exception dependence edges from the S-PEIs to the `ld` instruction are eliminated. If there is an edge constrained by the access order from a write instruction to a read instruction, it will not be eliminated. Although data speculation can eliminate the access order edge, that is beyond the scope of this paper. These exception edges eliminated from the S-PEIs are connected to a `chk` instruction. This safeguards the execution order of the original program by the execution of the

recovery code after executing the S-PEIs. The recovery code is generated by the compiler to recover the correct program status if the speculative load instruction defers any exception associated with the instruction.

In example 1, a `chk` instruction (N9) is inserted into the DAG. The edge from N3 to N6 is eliminated in the Example 1 b), and then an edge is added from N3 to N9 in the Example 1 c).

Step 3: Select a sequence of instructions as a speculative chain: We define a speculative chain as an instruction sequence that is constructed by those data dependence chains preceded by the speculative load instruction which do not include the instructions with side effects such as a store instruction on the IA-64 architecture. The compiler selects instructions by excluding any instruction that causes side effects while traversing instructions that depend on the result of the load instruction through the true dependence chains. The compiler marks the selected instructions as a speculative chain. We discuss the selection process in more details in Section 5.1.

In example 1, N6 and N7 are selected as a speculative chain.

Step 4: Build edges for the live-in and live-out sets of the recovery code: If the recovery code is to be represented explicitly in the DAG, a Φ instruction is inserted after a `chk` instruction. The Φ instruction causes problems for other optimizations including register allocation. Therefore, our representation marks the instructions on a speculative chain, and then the compiler generates the recovery code from the marked instructions in the last phase. The compiler has to add the live-in and live-out sets of the recovery code as operands for the `chk` instruction. Algorithm 1 shows the algorithm to determine the live-in set `li` and live-out set `lo` from the corresponding speculative chain.

```
sc<IN> : set of statements in the speculative chain
li<OUT>: set of src operands for live-in set
lo<OUT>: set of dst operands for live-out set
SCOTH  : set of statements in other speculative chains

li = lo = ϕ
for (s ⊂ statements(sc)) {
  for (o ⊂ dst operands(s))
    if ((Succ(o) ∩ src operands(sc) ≠ ϕ ) ||
        //  destination is used out of the chain
        (Succ(o) ∩ src operands(SCOTH) ≠ ϕ ))
        //  destination is used in other chain
      lo ∪= o
  for (o ⊂ src operands(s))
    if (Pred(o) ∩ dst operands(sc) == ϕ) li ∪= o
}
```

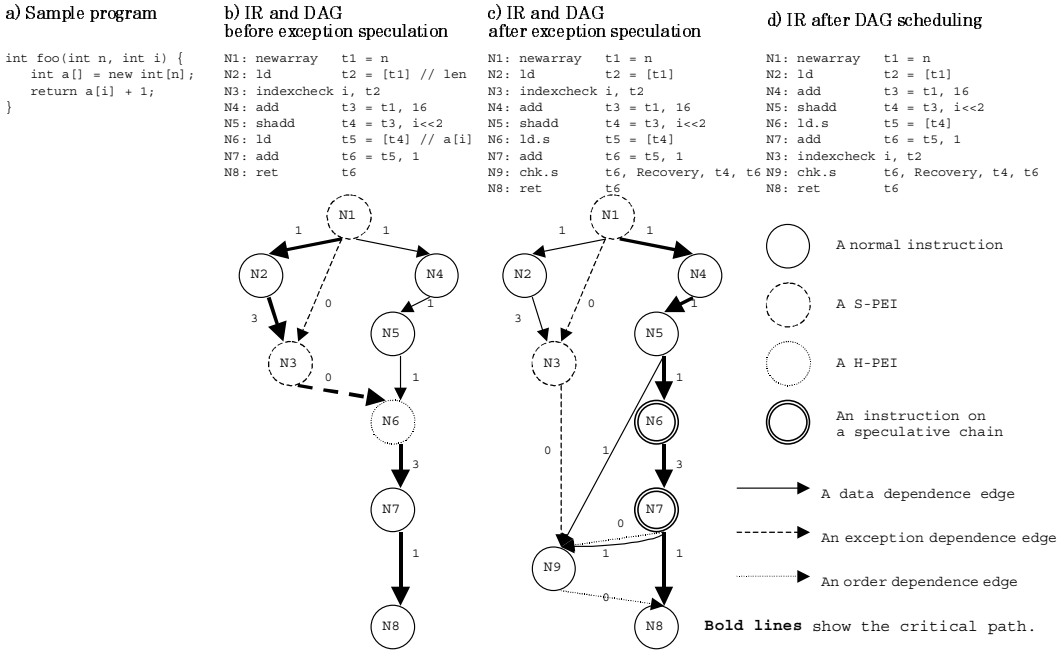
Algorithm 1. Determination of the live-in and live-out sets

In example 1, the compiler adds operands t4 and t6 to N9 as live-in and live-out sets, respectively.

Step 5: Build edges among selected instructions and the `chk` instructions: Instructions on a speculative chain have to have edges to a `chk` instruction so that all speculated instructions precede the `chk` instruction. The `chk` instruction has to have edges to all the instructions that use the live-out set to schedule the `chk` instruction before these instructions. These edges insure the correct execution when a deferred exception occurs and the recovery code recalculates the live-out set.

In example 1, a data dependence edge from N5 to N9 is added for the live-in set of that block of recovery code. Then, an order dependence edge from N7 to N9 is added to execute a `chk` instruction after N6 and N7 are executed, and an order dependence edge from N9 to N8 is also added. A data dependence edge from N7 to N9 is added in order to test whether an exception has been deferred.

After exception speculation, the compiler performs DAG scheduling. As a result, N4, N5, N6, and N7 can be scheduled



Example 1. An example of eliminating an exception constraint between H-PEIs and S-PEIs

before N3. Since a `chk.s` instruction takes 0 cycles on the IA-64 architecture if no exception has been deferred, exception speculation reduces the critical path length (in **bold**) from 9 to 8. Since the example is simplified for the sake of explanation, the reduction is small. In general, for an access to an array element, two load instructions for the array length and the array element can be moved speculatively across S-PEIs such as NullPointerException and IndexOutOfBoundsException checks. Therefore, the reduction in the critical path length will be more significant.

5 Compilation Issues for Speculation

In this section, we address two compilation issues using speculation. One is how to select a sequence of instructions as a speculative chain to avoid extra copy instructions during the phase of translating out of the SSA form [9, 18]. The other is how to generate the recovery code to minimize the live-in registers set needed by the recovery code and the impact on the bundle formation.

5.1 Select a Speculative Chain

Here, we describe a method to select a sequence of instructions as a speculative chain. The method is guaranteed not to insert extra copy instructions during the phase of translating out of the SSA form. If extra copy instructions are inserted, they might cancel out the advantages of speculation, since each such copy would require an additional hardware execution unit. For example, a move instruction between floating-point registers takes 5 cycles on the Itanium processor [19]. Therefore, this selection method is important.

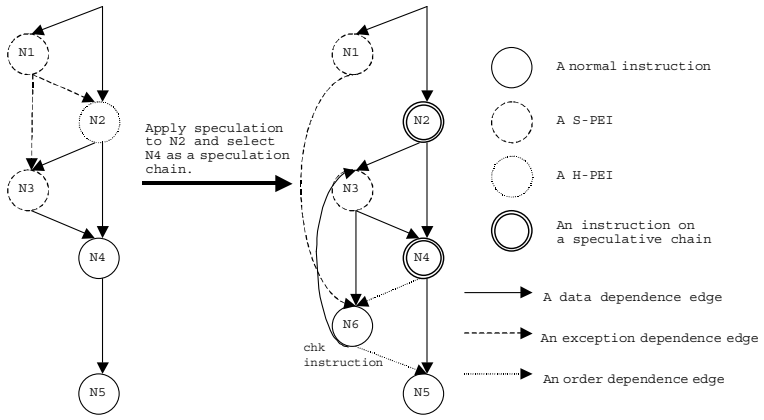
We propose a method to generate the longest possible speculative chain, while avoiding the generation of extra instructions, preventing non-cyclic graphs, and excluding instructions with side effects such as store instructions on the IA-64 architecture. We assert two conditions to generate the correct recovery code, along with two additional conditions to prevent the insertion of extra copy instructions. One of them was already proposed by Ju et al. [6]. If all of the conditions are satisfied while traversing the instructions depending on the results of the load instructions or preceding from the source of the load instructions through the true dependence edges, the compiler can add instructions to the speculative chain.

Condition 1: Do not choose any instruction that would result in a cyclic graph: If the graph becomes cyclic, it would preclude list scheduling. Since the IR of a Java program has many exception dependence edges, the compiler must insure this condition when selecting any sequence of instructions as a speculative chain. Example 2 shows an example of generating a cyclic graph by speculation. We could decide to apply speculation to N2 and select N2 and N4 as a speculative chain. A node N6 is created for a `chk` instruction. An exception edge from N1 to N2 is eliminated, and an exception edge from N1 to N6 is added to retain the original exception semantics. A data dependence edge from N3 to N6 is added since the recovery code refers to a variable defined in N3, and a data dependence edge from N6 to N3 is added since a variable defined in the recover code is referred to in N3. At that point, nodes N3 and N6 form a cyclic path.

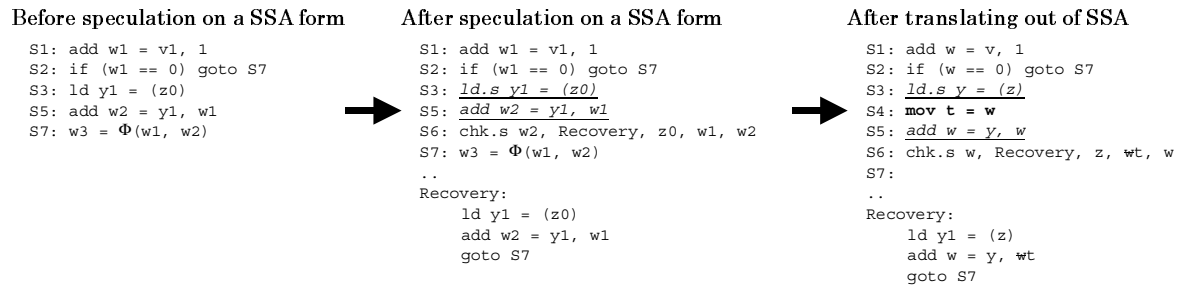
Condition 2: Do not select any instruction that causes a side effect in a speculative chain: A NaT bit in the registers of a speculative chain is propagated if a speculative load instruction defers an exception. Except for store instructions, no instructions will raise a hardware exception even when a NaT bit is encountered in their source operands. Thus, store instructions must be excluded from the speculative chain.

Condition 3: Do not choose any instruction that overrides another version of the same variable involved in a web: A web is defined as a maximal union of true dependence chains that consist of the following two types of links. One type of the link is the one from the definition operand in Φ or non-Φ instruction to the use operand in another Φ instruction. The other type of the link is the one from the definition operand in Φ or non-Φ instruction to the use operand of another non-Φ instruction, but this use operand of the non-Φ instruction terminates a web. A web must include one or more Φ instructions. Any variable that consists of a web will be assigned to the same name during the translation out of the SSA form.

If the compiler selects an instruction that extends the lifetime of another version of that variable, which will be part of the live-in set of a recovery code block, up to a `chk` instruc-



Example 2. An example of making a cyclic path by speculation



Example 3. Examples in which a compiler generates extra copy instructions

tion within a web, then the two live ranges of the variable interfere with each other at the chk instruction. It requires an extra copy instruction to avoid the interference between multiple versions of the variable during the translation out of the SSA form. Therefore, the compiler should not select any instruction that causes this interference. We give an example in Example 3. To simplify the explanation, we do not speculatively move any instructions across S-PEIs in this figure.

In Example 3, S1, S2, S5, and S7 form a web involving the variable w. When speculation is applied to S3 and S5 (in *italics*), w1 must be live at a chk instruction (S6) since w1 will be accessed in the recovery code. The variables w1, w2, and w3 are assigned the same name as w during the translation out of the SSA form. Since w1 is live at S6 and w2 overrides w1 at S5, w1 and w2 interfere at S6. Therefore, S4 (in **bold**) is generated to save the value of w1 as t, and t is used in the recovery code. In this case, S5 should not be chosen as an instruction in a speculative chain to avoid inserting an extra copy instruction.

Condition 4: Avoid interference between the destination registers of speculative instructions: This was already suggested in [6]. A compiler must not move the definitions in the same web if their live ranges interfere with one another. Normally, only one path is speculated over a particular branch based on the execution probability of each path.

5.2 Recovery Code Generation

Here, we describe a method to generate recovery code. The compiler should minimize negative effects on the main code since the recovery code is rarely executed when exceptions have been deferred. To achieve this goal, there are three issues to address: minimization of the live-in registers set, the generation of recovery code from the speculative chain, and avoiding constraints when forming bundles. The bundle formation identifies a group of instructions, which should be executed simultaneously, for instruction scheduling. In this section, we offer solutions for these issues.

In our framework, we identify the instructions on a speculative chain by traversing the DAG since the recovery code is not represented explicitly. The instructions on a speculative chain are marked in the main code. To generate recovery code, the compiler duplicates the marked instructions in Section 5.1 while converting non-faulting load instructions to faulting load instructions. The recovery code is generated after the main code has been generated. The register assignment for the recovery code is the same as that for the main code. Bundles within the recovery code can be formed using the information from the code generation phase of the main code. Therefore, the compiler can minimize the increase in the code size. The code generation consists of the following four steps:

Step 1: Generate the prolog and epilog for the recovery code: In the recovery code, all killed registers except for the live-in and live-out register sets are written to persistent memory in a prolog, and the registers are restored from memory by the epilog. We call this *the registers spill set*, denoted as sp. The Algorithm 2 shows an algorithm to determine the spill set sp.

```
sc<IN>  : set of instructions in the speculative chain
li      : set of registers for live-in set
lo      : set of registers for live-out set
kl      : set of registers for kill set
sp<OUT> : set of registers for spill set
SCOTH   : set of instructions in all the speculative
          chains except sc

kl = li = lo = ϕ
for (s ⊂ instuctions(sc)) {
  for (r ⊂ dst registers(s)) {
    kl ∪= r
    if ((Succ(r) ∩ ¬src operands(sc) ≠ ϕ) ||
        (Succ(r) ∩ src operands(SCOTH) ≠ ϕ)) lo ∪= r
  }
  for (r ⊂ src registers(s))
    if (Pred(r) ∩ dst registers(sc) == ϕ) li ∪= r
}
sp = kl ∩ ¬(li ∪ lo)
```

Algorithm 2. Calculation of the spill set

In Example 4, the compiler made two speculative chains. One involves I1 and I4, and the other contains I2, I3, and I4. There is an opportunity to minimize the number of chk instructions, but this is beyond the scope of this paper. In Recovery Block 2, r2 and r11 are the live-in set, and r12 and r21 are the live-out set. Then, r12, r13, and r21 are the kill set. As a result, r13 is the spill set to be saved in the prolog and restored in the epilog. Therefore, r13 is free for register allocation after I4.

Step 2: Generate the recovery code: The compiler duplicates the marked instructions from the corresponding speculative chain between the prolog and the epilog while converting non-faulting load instructions to faulting load instructions. A valid value will be written in the destination register since the faulting load reloads the value without setting a NaT bit in the

Before speculation

```
I1:  ld r2 = (r1)
I2:  ld r12 = (r11)
I3:  add r13 = r12, 1
I4:  add r21 = r2, r13


I7:  mov .., r12
I8:  mov .., r21
```

After speculation

```
I1:  ld.s r2 = (r1)          // bundle 0
I2:  ld.s r12 = (r11)        // bundle 0
I3:  add r13 = r12, 1        // bundle 0
I4:  add r21 = r2, r13       // bundle 1
I5:  chk.s r2, Recovery1     // bundle 1
I6:  chk.s r12, Recovery2    // bundle 1
I7:  mov .., r12             // bundle 2
I8:  mov .., r21             // bundle 2

Recovery1:
I9:  ld r2 = (r1)            // copy of I1
I10: add r21 = r2, r13       // copy of I4
I11: chk.s r21, Recovery2    // copy of I6
I12: goto I7

Recovery2:
I13: spill r13
I14: ld r12 = (r11)          // copy of I2
I15: add r13 = r12, 1        // copy of I3
I16: add r21 = r2, r13       // copy of I4
I17: fill r13
I18: goto I7
```

Example 4. Recovery code generation

recovery code, even if the non-faulting instruction sets the NaT bit in the destination register in the main code.

In example 2, I9 and I10 are generated from I1 and I4, while I14, I15, and I16 are generated from I2 to I4.

Step 3: Duplicate instructions within the same bundle of the corresponding `chk` instruction: Since VLIW machines such as the IA-64 processor issue several instructions simultaneously, bundle formation is important to extract instruction parallelism from a program. Therefore, speculation should not impose any constraints on bundle formation. Since a branch instruction can specify only the first instruction of a bundle as a target address, the return address from a recovery code block is the first instruction of the next bundle. If the recovery code of the corresponding `chk` instruction returns to the next bundle in the main code, the successor instructions for that `chk` instruction within the same bundle would be skipped. To avoid this situation, the successor instructions within the same bundle are duplicated at the end of the recovery code block.

In the example, we assume that I1, I2, and I3 are included within a bundle, and I4, I5, and I6 are linked within another bundle. Recovery1 cannot return to the address of I6 since it can only return to the addresses of I1, I4, or I7 as the tops of bundles. Therefore, I6 within the same bundle is duplicated as I11 at the end of the recovery code.

Step 4: Generate a branch instruction to return to the address of the next bundle: The compiler terminates the recovery code with the branch instruction, which goes back to the address of the next bundle containing a `chk` instruction. In the example of Recovery1, the compiler generates a branch instruction I12 to the address of I7.

6 Experiments

Our experiments were performed using a product version of the IBM Developers Kit for IA-64, Java Technology Edition, Version 1.3. We implemented the exception speculation that we described here in the Just-In-Time Compiler. The measurements were performed on an IA-64 Itanium processor [19] with 1GB of RAM.

Table 1. Comparison of DAG representations with and without exception edges

	DAG without exception edges	DAG with exception edges
Total number of BB nodes	142679	37327
Average number of statement nodes per BB node	1.23	4.71
Total number of edges between BB nodes	262791	54669
Total number of edges between statement nodes	80190	145952

6.1 Efficiency of Representation

Table 1 shows the space efficiency of the DAG representations with and without exception edges. For the case of the DAG representation without exception edges, explicit control flow edges are used to represent exception dependence. We ran seven programs from the SPECjvm98 suite [20] with the size of 100. The 1070 methods were compiled. The statistics for DAG without exception edges were estimated by increasing or decreasing nodes and edges that would be required. A *BB node* represents a basic block, while a *statement node* represents a statement in a basic block.

The second row of the table shows that the DAG with exception edges drastically reduced the total number of nodes. The third row shows a BB node with exception edges includes more than four instructions while a BB node without exception edges includes nearly one instruction. Thus, it increased the opportunity to apply local (intra-block) optimizations. We can also apply speculation using only local optimizations, as we described in Section 1. It is important for a dynamic compiler to avoid applying time-consuming global optimizations such as trace scheduling or percolation scheduling.

The fourth and fifth rows show that the DAG with exception edges reduced the total number of edges between BB nodes while it increased the number of edges between statement nodes in a graph. This reduces the space for the DAG representation during the compilation. The similar result has been obtained in other work [21].

6.2 Micro-Benchmarks

We measured the effectiveness of exception speculation using two micro-benchmarks, described in Table 2. In all these benchmarks, two-dimensional arrays are frequently accessed in kernel loops. Figure 3 shows the performance improvements, where the four bars in each benchmark present the performance improvement relative to our baseline for four possible combinations of enabling exception speculation (ES) and loop versioning (LV). Our baseline is to avoid applying

Table 2. Benchmarks characteristics of the micro-benchmarks

Benchmarks	Description
All-pairs Shortest-path	Find the shortest path from the start vertex to each of the other vertices.
Matrix Multiply	Multiply two two-dimensional matrices.

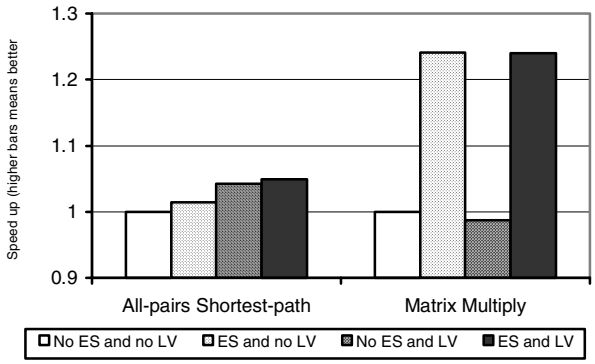


Figure 3. Runtime performance measurements for the micro-benchmarks

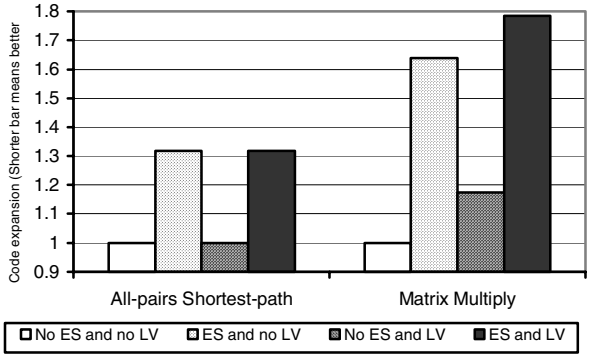


Figure 4. Static code size expansion of a kernel method for the micro-benchmarks

exception speculation and loop versioning (denoted as "no ES and no LV"). Loop versioning [12] is a technique to eliminate all the exceptions in the nested loop. Within a safe region, the compiler is free to apply optimizations such as instruction reordering.

When loop versioning is not performed, exception speculation improves the performance from 1% to 24% (with an average of 12.8%). It is effective for all benchmarks, since there are many S-PEIs in these kernel loops, as typical in Java programs.

When loop versioning is performed, exception speculation improves the performance from 5% to 25% (with an average of 14.5%). The improvement by exception speculation is almost the same as that without loop versioning. The reason is simple in that loop versioning eliminates almost all S-PEIs that may throw Java exceptions in the kernel loops.

Figure 4 shows the static code size increase for each kernel method relative to that of our baseline (no ES and no LV) for four possible combinations. The size grew from 32% to 64% without loop versioning, and the size grew from 32% to 61% with loop versioning. The largest code growth is observed for Matrix Multiply with the best performance improvement. The code growth is mainly due to recovery code, and the critical path is not affected.

6.3 Java Grande Benchmark Suite

We also measured the effectiveness of exception speculation using the Java Grande Benchmark Suite [22]. We choose the kernels of Section II with SizeA. Each benchmark the kernels of Section II with SizeA. Each benchmark is described in Table 3. Figure 5 shows the performance improvement for the optimizations relative to our baseline (no ES and no LV) for four possible combinations.

When loop versioning is not applied, exception speculation alters the performance from -1% to 10% (with an average of 1.8%). It is especially effective for SOR and SparseMatmul.

When loop versioning is applied, exception speculation improves the performance from -2% to 10% (with an average of 1.7%). It is effective for SOR and SparseMatmul. This is because the kernel loop of SOR accesses two-dimensional arrays frequently, and the kernel loop of SparseMatmul has indirect accesses to the array element. In LUFact, loop versioning improve the performance by 25% even without exception speculation.

In Crypt, the performance degrades a little. The reason is that the check code to guarantee a safe region is executed at the loop entry, and results in significant overhead. In FFT, the performance also degrades slightly. The reason is that the TLB misses cause deferred exceptions, which lead to the execution of the recovery code. Since NullPointerException and IndexOutOfBoundsException never occur in these benchmarks, no non-faulting load instruction with an invalid effective address will be executed. Figure 6 shows the static code size increase relative to our baseline (no ES and no LV) for four possible combinations. The size grew from 0% to 7% when applying loop versioning. Unlike the micro-benchmarks, the code growth is not significant.

Table 3. Benchmarks characteristics of the Java Grande Benchmark Suite

Benchmarks	Description
Series	Compute the first N Fourier coefficients.
LUFact	Solves an N x N linear system using LU factorization.
HeapSort	Sorts an array of N integers using a heap sort algorithm.
Crypt	Performs IDEA encryption and decryption.
FFT	Computes FFT's of complex, double precision data.
SOR	Solve an equation by Successive Over Relaxation.
SparseMatmul	Multiply two one-dimensional sparse matrices.

Table 4. Benchmarks characteristics of the SPECjvm98 benchmarks

Benchmarks	Description
compress	LZW compression and decompression.
jess	NASA's CLIP expert system.
db	Search and modify a database.
javac	Source to bytecode compiler.
mpegaudio	Decompress audio file.
mtrt	Multi-threaded image rendering.
jack	Parser generator generating itself.

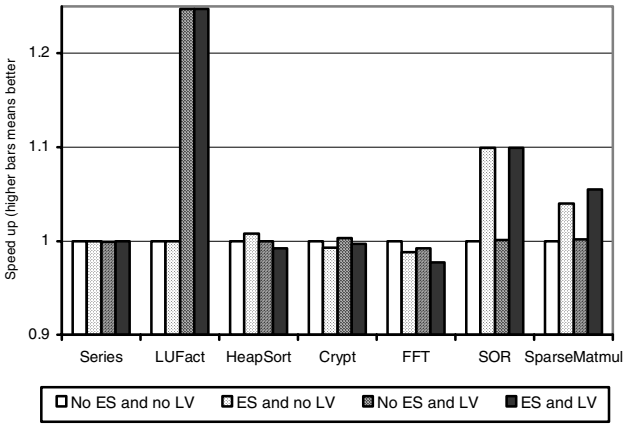


Figure 5. Runtime performance measurements for the Java Grande Benchmark Suite

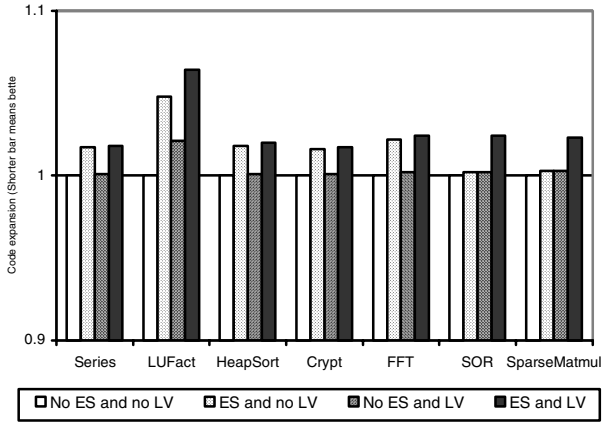


Figure 6. Static code size expansion for the Java Grande Benchmark Suite

6.4 SPECjvm98

We measured the effectiveness of exception speculation using SPECjvm98 with the size of 100. Each benchmark is described in Table 4. Figure 7 shows the performance improvement for the optimizations relative to our baseline (no ES and no LV) for four possible combinations.

When loop versioning is not applied, exception speculation improves the performance from 0% to 10% (with an average of 2.4%). It is particularly effective for compress, mpegaudio, and mtrt, which include many S-PEIs.

When loop versioning is applied, exception speculation changes the performance from -1% to 12% (with an average of 2.3%). It is effective for compress, mpegaudio, and mtrt. For mpegaudio, this is because its kernel loop accesses two-dimensional arrays frequently. As we described in Section 3.2, our production compiler aggressively eliminates exception checks against the references to instance variables and array elements in the first dimension. Therefore, exception speculation is not effective in our implementation for a program that accesses one-dimensional array or instance variables frequently. Example 5 shows a part of the kernel loop from mpegaudio. Since the value of the first dimension of the array `C` is loop-variant and a two-dimensional array consists of linked pointer vectors, no optimization can remove the S-PEIs for the element accesses of the second dimension of the array `C`. As a result, exception speculation effectively eliminates the exception dependence associated with these accesses. In fact, exception speculation reduces the critical path length from 31 to 25 cycles for the entire kernel loop. In jess, the performance degrades a little. The reason is that TLB misses cause deferred exceptions, which lead to the execution of the recovery code. Since NullPointerException and IndexOutOfBoundsException never occur in these benchmarks, no non-faulting load instruction with an invalid effective address is ever executed.

```
for (int j = 0; j < n; j++) {
   float  B[] = A[i++];
   f += C[j][k+16] * B[0];
}
```

Example 5. A part of the kernel loop of mpegaudio

Figure 8 shows the static code size increase relative to our

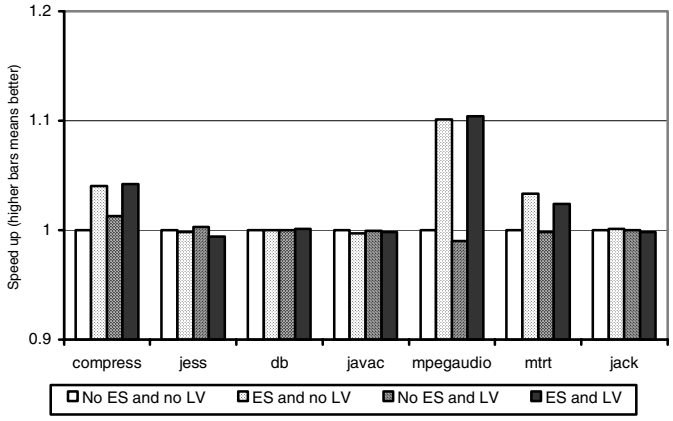


Figure 7. Runtime performance measurements for the SPECjvm98 benchmarks

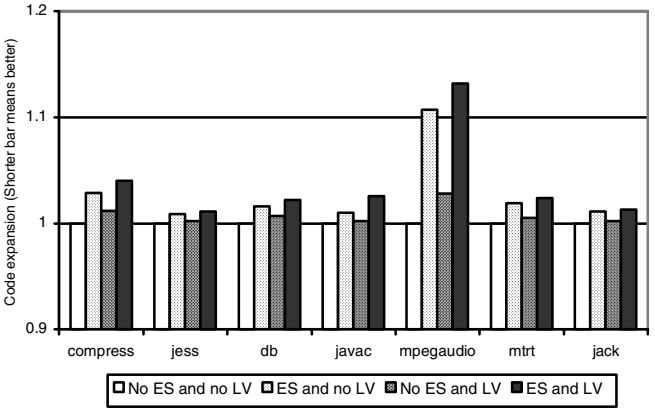


Figure 8. Static code size expansion for the SPECjvm98 benchmarks

baseline (no ES and no LV) for four possible combinations. The size grew from 0% to 10% with the application of loop versioning. Unlike the micro-benchmarks, the code growth is not significant. The largest code growth was observed for mpegaudio along with its significantly better performance improvement. This indicates that exception speculation is applied aggressively here.

7 Conclusions

We have presented an exception speculation technique that shortens the critical path length constrained by exception dependence for Java programs. By using exception dependence edges and associating the critical path length with each basic block in the SSA-based DAG representation, speculative chains are effectively selected to schedule a sequence of the most profitable instructions across every Java exception check, without generating any extra copy instructions. Unlike the previous work [11], our approach does not introduce any conditional branches. Our proposed method for the recovery code generation does not impose any register pressure on the critical path nor any restrictions to the final code scheduling (bundle formation) phase. Our preliminary results on an IA-64 Itanium processor show that exception speculation improves the performance by up to 25% (with an average of 12.8%) for micro-benchmarks, and up to 10% (with an average of 1.7%) for Java Grande Benchmark Suite, and up to 12% (with an average of 2.3%) for SPECjvm98. The experiments show that our speculation technique is especially effective for programs that are computation intensive and access multiple dimensional arrays frequently.

8 Acknowledgement

We thank the people in Network Computing Platform at Tokyo Research Laboratory for implementing our JIT compiler. We also thank Manish Gupta for his useful feedback on earlier drafts of this work. We also thank Shannon Jacobs for his editorial assistance. We appropriate the insightful comments from the anonymous reviewers.

References

[1] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. ACM Transactions on Computer Systems, 11(4), pp. 376-408, 1993.

[2] B. C. Le. An Out-of-Order Execution Technique for Runtime Binary Translators, In Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems, pp.151-158, 1998.

[3] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 344-354, 1990.

[4] 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 technique for VLIW and Superscalar Compilation. Journal of Supercomputing, 7(1), pp. 229-248, 1993.

[5] D. I. August, B. L. Deitrich, and S. A. Mahlke. Sentinel Scheduling with Recovery Blocks. Computer Science Technical Report CRHC-95-05, University of Illinois, Urbana, 1995.

[6] R. D. Ju, K. Nomura, U. Mahadevan, and L. We. A Unified Compiler Framework for Control and Data Speculation. In Proceedings of the Conference on Parallel Architectures and Compilation Techniques, pp. 157-168, 2000.

[7] R. Zahir, D. Morris, J. Ross, and D. Hess. OS and Compiler Considerations in the Design of the IA-64 Architecture, In Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems, pp. 212-221, 2000.

[8] D. M. Gallagher and W. Y. Chen and S. A. Mahlke and J. C. Gyllenhaal and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 183-193, 1994.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems, 13(4), pp. 451-490, 1991.

[10] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan. Dependence Analysis for Java, In 12th International Workshop on Languages and Compilers for Parallel Computing, pp. 35-52, 1999.

[11] M. Arnold, M. S. Hsiao, U. Kremer, and B. Ryder. Exploring the interaction between Java's implicitly thrown exceptions and instruction scheduling. International Journal of Parallel Programming, 29(2), pp. 111-137, 2001.

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

[13] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In Proceedings of the International Symposium on Computer Architecture, pp. 26-37, 1997.

[14] M. Gupta, J.-D. Choi, and M. Hind. Optimizing Java Programs in the Presence of Exceptions, In Proceedings of the 14th European Conference on Object-Oriented Programming, pp. 422-446, 2000.

[15] Intel Corp. IA-64 Application Developer's Architecture Guide, http://developers.intel.com/design/ia64/downloads/adag.htm.

[16] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap, In Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems, pp. 139-149, 2000.

[17] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In Proceedings of SIGPLAN '86 Symposium on Compiler Construction, pp. 11-16, 1986.

[18] V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating Out of Static Single Assignment Form. In Static Analysis Symposium, LNCS 1694, pp. 194-210, 1999.

[19] Intel Corp. Itanium™ Processor Microarchitecture Reference, http://developers.intel.com/design/ia64/downloads/245474.htm.

[20] Standard Performance Evaluation Corp. SPEC JVM98 Benchmarks, available at http://www.spec.org/osg/jvm98/.

[21] J.-D. Choi, D. Grove, M. Hind, and V. Sarker. Efficient and precise modeling of exceptions for the analysis of Java programs. In ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 21-31, 1999.

[22] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking Java Grande applications. In ACM 1999 Java Grande Conference, pp. 81-88, 1999.