# **Eliminating Exception Constraints of Java Programs for IA-64**

Kazuaki IshizakiISHIZAKI@TRL.IBM.COMTatsushi InagakiE29253@JP.IBM.COMHideaki KomatsuKOMATSU@ JP.IBM.COMToshio NakataniNAKATANI@ JP.IBM.COMIBM Research, Tokyo Research Laboratory1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan

### 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 optimizations for Java programs. Although some effective techniques have been proposed to eliminate exception checks, many exception checks still remain. We propose a new framework to perform speculation effectively for Java programs using a direct acyclic graph representation (DAG) 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 estimate the potential reduction of the critical path length accurately for applying speculation. The new framework also allows us to avoid extra copy instructions and to generate efficient code with minimum register pressure. We have implemented it in the IBM Java Just-In-Time compiler, and observed performance improvements of 30% for one of the micro-benchmark programs, up to 13% (with an average of 1.7%) for the Java Grande Benchmark Suite, and up to 12% (with an average of 1.2%) for SPECjym98 on an Itanium processor. Our speculation technique was particularly effective for those programs that access multidimensional arrays.

## 1 Introduction

The type safety feature of Java calls for the bytecode to make a large number of runtime exception checks in order to eliminate error-prone situations. These runtime checks ensure 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 thrown by the hardware as the result of program execution such as segmentation faults and invalid operations. Java exceptions are those software exceptions that 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* (PEI) if it can cause a hardware exception. 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. Such an exception 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, 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 how we can allow instruction reordering optimizations in Java programs on the IA-64 [3]. In the past, some effective techniques have been proposed to eliminate S-PEIs [4, 5, 6], but many S-PEIs still remain. Therefore, we propose a new framework to eliminate the constraints between S-PEIs and H-PEIs by using speculative code motion. This allows a compiler to schedule H-PEIs before dependent S-PEIs. Our approach depends upon special hardware features of the 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, 7, 8, 9, 10, 11], which is a technique to alleviate the control dependence between a branch and the following instructions by estimating the branch direction. Control speculation allows the compiler to aggressively schedule instructions across a branch. The other is called *data speculation* [12, 8, 10, 11], which is a technique to alleviate the data dependence between a store instruction and the following load instructions by determining when 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 *excep*tion speculation as a third speculation technique to alleviate the exception dependence between the S-PEI and the following H-PEIs by determining 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 [13]. A DAG consists of nodes that represent instructions in a program and edges that represent dependencies among instructions. We use an edge in the DAG to represent an exception dependence relationship between each S-PEI and each of its succeeding H-PEIs [14]. We call this an *exception dependence edge*. Every exception dependence edge can also be 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 the approach of Arnold et al. [15], 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 the compiler avoids global optimizations such as percolation 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* [10], to move across an S-PEI, and how to generate the recovery code. We propose a selection mechanism for a speculative chain that does not require 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 the recovery code, and to minimize the impact on the code scheduling (bundle formation) phase by duplicating instructions in the recovery code.

We implemented these approaches for eliminating exception checks and exception constraints using the IBM Java Just-In-Time (JIT) compiler [4] for the IA-64. We conducted experiments by running micro-benchmarks, and two industry standard benchmark suites, the Java Grande Benchmark Suite and SPECjvm98. Our preliminary results show that exception speculation improves the performance, by 30% for one of the micro-benchmark programs, up to 13% (with an average of 1.7%) for the Java Grande Benchmark Suite, and up to 12% (with an average of 1.2%) for SPECjym98, with only a modest code size growth on an Itanium processor that is an implementation of the IA-64 architecture. In a few cases in the Java Grande Benchmark Suite and SPECjvm98, we observed a small performance degradation. We believe that this is due to the execution of recovery code when exception speculation fails because of TLB misses at runtime. In addition, we also observed that exception check eliminations using forward dataflow analysis [5], backward dataflow analysis [6], and loop versioning [4] are also effective with exception speculation. In some cases, our results show that exception speculation increases the performance even after exception check eliminations have been applied. The experiments show that our speculation technique is particularly effective for those programs that access multi-dimensional arrays frequently when those exception elimination techniques cannot reduce the number of exception checks. Thus, exception speculation is complementarily effective even when known exception elimination techniques are applied. We also observed that our framework saves 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 checks and exception constraints in Java programs using IBM's Java JIT compiler for the IA-64.

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 gives a 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

We categorize approaches for eliminating exception constraints into two categories. One is speculation to move H-PEIs over S-PEIs. The other is elimination of S-PEIs using compiler optimization techniques. Section 2.1 describes related works regarding to the elimination of S-PEI. Section 2.2 describes related work regarding the elimination of S-PEIs.

#### 2.1 Speculation of H-PEI

Ebcioglu and Altman [16] 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. These extra copies increase the critical path length. In contrast, since our technique estimates the critical path length using DAG representation, the critical path length is not increased. Le [2] also described a runtime binary translation technique that moves instructions speculatively without any special architectural support. When an exception is raised by a speculated instruction, it is caught by the OS and the OS goes back to a generated checkpoint in the normal code. Thus, the translator requires the generation of some checkpoint code, which increases the critical path length even when no exception occurs. Its advantage is in not requiring any hardware support such as non-faulting instruction. In contrast to this approach, our speculation technique does not increase the critical path length.

Ju et al. [10] described a unified framework for control and data speculation on the IA-64. Their framework uses an SSA-based DAG to move instructions speculatively. It is similar to ours, in the sense that a DAG is used and extra copy instructions are avoided. Their target language is C, and thus they do not deal with runtime exception speculation for a type-safe language. Their experiments were limited to an IA-64 architecture simulator.

Arnold et al. [15] described the impact of Java exceptions on a VLIW architecture based on superblock scheduling and a general percolation technique, and assumed architectural support for non-faulting load instructions. Their technique converts an S-PEI to a pair of compare and branch instructions. Instructions are then moved speculatively by superblock scheduling and a general percolation technique. The advantage is in applying general instruction scheduling techniques. The disadvantage is in increasing the number of control dependence edges for subsequent instructions by converting the S-PEI. Again, their experiments were limited to a simulator.

### 2.2 Elimination of S-PEI

Whaley [5] described forward dataflow analysis to eliminate S-PEI. This technique eliminates redundant S-PEIs for validated values in a control flow graph. Since this technique eliminates only redundant S-PEIs going forward, many S-PEIs remain after applying this technique. Thus, our speculation technique offers additional improvements.

Kawahito et al. [6] described an algorithm using backward dataflow analyses such as partial redundancy elimination [17] to move S-PEIs backward in a control flow graph to the earliest points they can reach. Since this technique moves redundant S-PEIs to an infrequently executed path such as a location outside of a loop, it is quite effective. However, since several S-PEIs remain after applying this technique, our speculation technique is still effective.

Suganuma et al. [4] described loop versioning that 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. This technique doubles the code size. It also fails to eliminate S-PEIs for two or more dimensional arrays. Our speculation technique is effective even when applying loop versioning.

Manish et al. [18] described some compiler optimizations to reorder different S-PEIs or reorder an S-PEI with respect to an instruction affected by precise exception semantics using software solutions. Their technique first finds the visible state in a method, and then relaxes exception constraints between S-PEIs that do not change the visible state. It has an advantage that no hardware support is required. However, it cannot move an H-PEI over an S-PEI speculatively. We propose a complementary technique to reorder S-PEIs and H-PEIs with appropriate architectural support.

#### **3** System Overview

This section provides an overview of the IA-64 architecture and our compiler. Section 3.1 de-

scribes 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 [9]:

**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 a hardware exception but does propagate 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 recovery code.

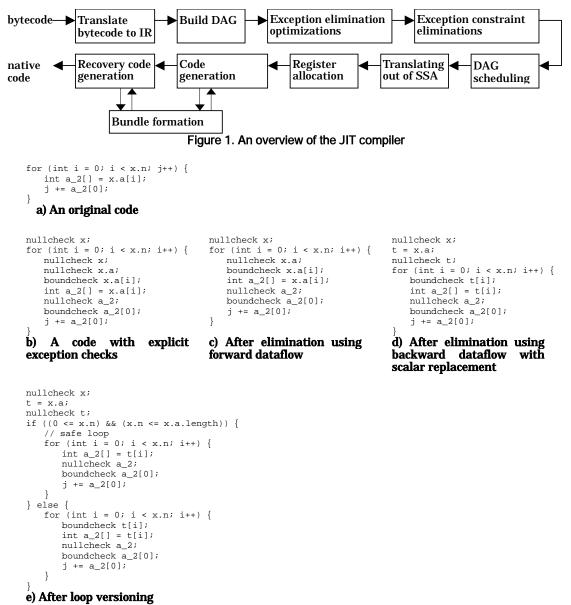
Feature 3: Many registers to hold the values the recovery code can use.

These features could also be implemented in software using the support of an operating system. However, the IA-64 architecture [3] supports the above features in hardware. For Feature 1, the IA-64 architecture provides non-faulting load instructions such 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. When the NaT bits of one or more source registers are set, the referring instruction propagates only the NaT bits to the 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 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. The DAG consists of nodes that represent instructions in a program and edges that represent dependencies among instructions.

Second, it performs exception elimination optimizations such as loop versioning and nullcheck elimination to remove redundant runtime exception checks. The nullcheck elimination algorithm using forward dataflow analysis [5] and backward dataflow analysis [6] removes most of the nullchecks for references to instance variables. Loop versioning [4] removes most of the bound-checks for the first dimension of arrays. All of these are software approaches eliminate exception checks as described in Section 2. Although these optimizations are effective, the exception checks against the higher dimensions of arrays cannot be eliminated and many exception checks have already been eliminated during this phase. Figure 2 shows an example of eliminating exception checks. Figure 2 a) is the original code. Figure 2 b) represents a code block with explicit exception checks such as nullcheck and boundcheck. Nullcheck elimination using forward dataflow analysis eliminates a nullcheck for the instance variable x in the loop. The result is shown in Figure 2 d). Finally, loop versioning eliminates the boundcheck for t [i] by creating a safe loop.



However, exception checks for the array a\_2 cannot be eliminated. The result is shown in Figure 2 e).

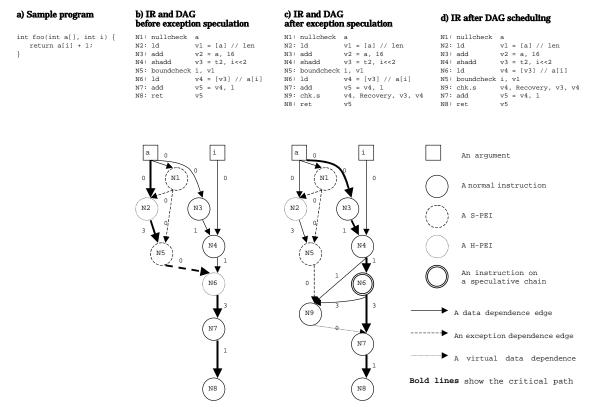
Figure 2: An example of eliminating exception checks

Third, it eliminates the exception constraints against the remaining exception checks, as described in Section 4, and it performs DAG scheduling based on list scheduling [19]. In Figure 2 e), the elimination of the exception constraints enables the scheduling of a load instruction for the element  $a_2[0]$  before the nullcheck and boundcheck in a safe loop. After it translates out of the SSA form, it allocates physical registers and generates the native code. The code generation works with the bundle formation, which identifies groups of instructions that should be executed simultaneously. Since VLIW machines such as the IA-64 processor issue several instructions si-

multaneously, bundle formation is important to extract instruction parallelism from programs. In the code generation phase, the S-PEI is converted to a pair of native compare and branch instructions. Finally, it generates recovery code, as described in Section 5, corresponding to each speculative chain.

### 4 Exception Speculation

In this section, we present a technique to eliminate an exception constraint between an S-PEI and an H-PEI by using speculation 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. An example of eliminating an exception constraint between H-PEIs and S-PEIs

Example 1 shows an example of our technique. It shows an IR and a DAG. The DAG consists of nodes that represent instructions in a program and edges that represent dependencies among instructions. Each edge is weighted with the latency between instructions. Here, we assume that each 1d instruction (N2 and N6) and each 1d.s instruction (N6) in Example 1 c) and d) take three cycles and all other instructions take one cycle each, except for the chk instruction (N9). A chk instruction takes 0 cycles. Unlike the system of Arnold et al. [15], 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. An exception dependence edge has zero latency since S-PEI and H-PEI can be executed in parallel due to the

high ILP of the IA-64. The exception dependence edge allows the compiler to accurately estimate the potential change in 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 moved speculatively.
- **Step 2:** Reconnect incoming exception dependence edges from a load instruction to a chk instruction.
- Step 3: Select a sequence of instructions as a speculative chain.
- **Step 4:** Build edges for the live-in and live-out sets of the recovery code.
- **Step 5:** Build edges among the selected instructions and the chk instructions.

We give detailed explanations for each step of applying exception speculation below.

**Step 1:** The compiler compares the two earliest times to initiate the execution of the load instruction. One time is constrained by a data dependence, and the other is constrained by an exception dependence. If the longest time is set only by the exception dependence, the compiler will move the load instruction speculatively by replacing the faulting load instruction with a non-faulting one. This constraint can avoid unnecessary speculation that degrades performance by executing recovery code due to TLB misses.

In the case of N2 of Example 1, the time set for the data dependence edge is zero, and the time set for the exception dependence is zero. Therefore, N2 is not selected as a candidate for speculation. This is because the critical path length would not be reduced even if the compiler eliminated an exception dependence edge from N2. In the case of N6, the time set for the data dependence edge is two and the time set for the exception dependence edge is three. Therefore, N6 is selected as a candidate for speculation, and N6 is replaced with a non-faulting load instruction.

**Step 2:** A chk instruction is inserted at the original position of the corresponding load. Then, the exception dependence edges from the S-PEIs to the 1d 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. Data speculation can eliminate the access order edge, but this is beyond the scope of this paper. The exception edges eliminated from the S-PEIs are connected to a chk instruction. This safeguards the execution order of the original program by executing the recovery code after executing the S-PEIs. The recovery code is generated by the compiler to recover the correct program state 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 N5 to N6 is eliminated in Example 1 b), and then an edge is added from N5 to N9 in Example 1 c).

**Step 3:** 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 any instructions with a side effect such as a store instruction on the IA-64 architecture. The compiler selects instructions by excluding any instruction that causes a side effect while it is traversing the instructions that depend on the result of the load instruction through the true dependence chains.

Furthermore, the compiler takes keep track of the earliest time to initiate the execution of each instruction in a speculative chain. If the earliest time to initiate the execution of an instruction in the speculative chain is later than the time set for the exception dependence as calculated in Step 1, the compiler excludes the instruction from the speculative chain. This avoids utilizing excessive resources by unnecessary instruction hoisting. Then the compiler marks the selected instructions as a speculative chain. We discuss the selection process in more details in Section 5.1.

In the case of N6 of Example 1, since the earliest time to initiate the execution is three, it is earlier than the time set for the exception dependence. In the case of N7, since the earliest time to initiate the execution is five, it is later than the time set for the exception dependence. Therefore, only N6 is selected as a speculative chain.

**Step 4:** If the recovery code were represented explicitly in the DAG, a phi ( $\Phi$ ) instruction would be inserted after the chk instruction. The phi instruction may cause 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. In Example 1, the compiler adds operands v3 and v4 to N9 as live-in and live-out sets, respectively.

```
Procedure determine_liveinout(sc, scoth, li, lo)
         in set of statements in the speculative chain
sc:
scoth: in set of statements in other speculative chains
1i:
         out set of src operands for live-in set
10:
        out set of dst operands for live-out set
begin
             : a statement
 st
 o, s, p : an operand
 li = lo = \phi
 for (st c statements(sc)) {
   for (o \subset dst operands(st))
      s = Sucessive use(o)
     if ((s \cap src operands(sc) \neq \phi) || // destination is used out of the chain <math>(s \cap src operands(scoth) \neq \phi)) // destination is used in other chain
        lo ∪= o
    for (o \subset src operands(st)) {
     p = Previous def(o)
      if (p \cap dst operands(sc) == \phi) li \cup= o
   }
end
```

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

**Step 5:** 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 those 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 N4 to N9 is added to the live-in set of that block of recovery code. A data dependence edge from N6 to N9 is added in order to check whether an exception has been deferred. Then, a virtual data dependence edge from N9 to N7 is added since N7 is a consumer of the data generated in the recovery code corresponding to N9. Although the virtual data dependence edge preserves the correct order between N9 and N8, it is not handled as a data dependence edge. If it were handled as a data dependence edge, a new phi instruction would be produced and the instruction could decrease the efficiency of some optimizations.

After exception speculation, the compiler performs DAG scheduling. As a result, N6 can be scheduled before N5. Since the chk.s instruction for N9 takes 0 cycles on the IA-64 architecture if no exception has been deferred, exception speculation reduces the critical path length (in **bold**) from 7 to 6. 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 translation out of SSA form [13, 20]. The other is how to generate the recovery code to minimize the live-in register set needed by the recovery code and the impact on 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 [21]. 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 cyclic graphs, and excluding instructions with side effects such as store instructions on the IA-64 architecture. We assert four conditions to generate the correct recovery code, along with two additional conditions to prevent the insertion of extra copy instructions. One of the conditions was already proposed by Ju et al. [10]. 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. There are four conditions as follows:

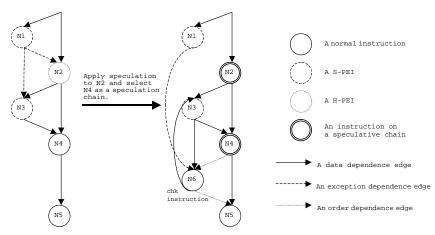
Condition 1: Do not choose any instruction that would result in a cyclic graph.

Condition 2: Do not select any instruction that causes a side effect in a speculative chain.Condition 3: Do not choose any instruction that overrides another version of the same variable involved in a phi congruence class.

**Condition 4:** Avoid interference between the destination registers of speculative instructions.

We give detailed explanations for each condition below.

**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 ensure 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 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 recovery code is referred to in N3. At that point, nodes N3 and N6 form a cyclic path, so N6 is excluded from the speculative chain consisting of N3 and N4.

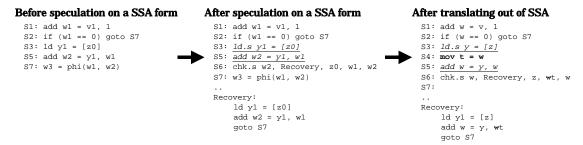


Example 2. An example of making a cyclic path by speculation

**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 chains.

**Condition 3:** Do not choose any instruction that overrides another version of the same variable involved in a phi congruence class: A phi congruence class is defined as a maximal union of true dependence chains that consist of the following two types of links [20]. One type of link is one from the definition operand in a phi or non-phi instruction to the use operand in another phi instruction. The other type of the link is one from the definition operand of another non-phi instruction, but this use operand of the non-phi instruction to the use operand of the non-phi instruction to the use operand of the non-phi instruction. Any variable that consists of a phi congruence class 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 instruction included in a phi congruence class, then the live ranges of the two versions for the variable interfere with each other at the chk instruction. An extra copy instruction is needed to avoid the interference between multiple versions of the variable during translation out of SSA form. Therefore, the compiler should not select any instruction that causes this interference. To simplify the explanation in Example 3, we do not speculatively move any instructions across S-PEIs.



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

In Example 3, S1, S2, S5, and S7 form a phi congruence class involving the variable w. When speculation is applied to S3 and S5 (in *italics*), w1 must be live at the 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 the extra copy instruction.

**Condition 4:** Avoid interference between the destination registers of speculative instructions: This was already suggested in [10]. A compiler must not move the definitions in the same phi congruence class if their live ranges interfere with one another. Normally, only one path is speculatively moved 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 register set, the generation of recovery code from the speculative chain, and avoidance of constraints when forming bundles. 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 from 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.
- **Step 2:** Generate the recovery code.
- Step 3: Duplicate instructions within the same bundle of a chk instruction.

**Step 4:** Generate a branch instruction to return to the address of the next bundle.

We give detailed explanations for each step of the code generation below.

**Step 1:** 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 register spill set*, denoted as sp. Algorithm 2 shows an algorithm to determine the spill set sp.

Example 4 is an example to explain the recovery code generation. We assume that three instructions within the same bundle can be executed in parallel. Here, the compiler made two speculative chains. One involves I1 and I4, and the other contains I2, I3, and I4. There are opportunities to minimize the number of chk instructions, but they are 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.

```
Procedure calc_spillset(sc, scoth, sp)
sc: in set of instructions in the speculative chain
scoth: in set of instructions in all the speculative chains except sc
         out set of registers for spill set
sp:
begin
                 : set of registers for live-in set
 li
 lo
                 : set of registers for live-out set
                 : set of registers for kill set
 k1
                  : a statement
 st
 r, s, p
                 : a register
 kl = li = lo = \phi
 for (st \subset instructions(sc)) \{
for (r \subset dst registers(st)) \{
     s = Successive use(r)
    kl \cup= r _____ if ((s \cap src operands(sc) \neq \phi) || (s \cap src operands(scoth) \neq \phi)) lo \cup= r
    for (r \subset src registers(st)) {
     p = Previous def(r)
      if (p \cap dst registers(sc) == \phi) li \cup= r
   }
 }
 sp = kl \cap (li \cup lo)
end
```

Algorithm 2. Calculation of the spill set

Before speculation A		After speculation				
12: 13: 14: 17:	<pre>ld r2 = (r1) ld r12 = (r11) add r13 = r12, 1 add r21 = r2, r13 mov, r12 mov, r21</pre>	I2: I3: I4: I5: I6: I7:		          	bundle 0 bundle 1 bundle 1 bundle 1 bundle 2	
		Reco 19: 110: 111:	very1: ld r2 = (r1) add r21 = r2, r13 chk.s r21, Recovery2 goto I7	//	copy of I1 copy of I4	
		I13: I14: I15: I16: I17:	<pre>very2: spill r13 ld r12 = (r11) add r13 = r12, 1 add r21 = r2, r13 fill r13 goto I7</pre>	//	copy of I3	

Example 4. Recovery code generation

**Step 2:** 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 the NaT bit in the recovery code, even if the non-faulting instruction sets the NaT bit in the destination register in the main code.

In Example 4, I9 and I10 are generated from I1 and I4, while I14, I15, and I16 are generated from I2 to I4.

**Step 3:** Since bundle formation is important to extract instruction parallelism from a program, speculation should not impose any constraints on it. Since a branch instruction can specify only the first instruction in the first bundle as a target address, the return address from a recovery code block is the first instruction in the second bundle. If the recovery code for a chk instruction returns to the second bundle in the main code, the successor instructions for the 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:** The compiler terminates the recovery code with the branch instruction, which goes back to the next bundle following the one containing the 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 our exception speculation framework in the Just-In-Time Compiler. All speculation features except for exception speculation were disabled. The measurements were performed on an IBM IntelliStation Mpro (Intel Itanium dual processor [21] 800MHz) with 1 GB of RAM, under Windows XP Advanced Server.

#### 6.1 Efficiency of Representation

Table 1 shows the space efficiency of the DAG representations with and without exception edges. For the DAG representation without exception edges, explicit control flow edges are used to represent exception dependencies. We ran seven programs from the SPECjvm98 Suite [22] with a size of 100. During the execution, 1,070 methods were compiled. A *BB node* represents a basic block, while a *statement node* represents a statement in a basic block.

Table 1. Companson of DAG representations with and without exception cuges						
	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				

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

The second row of the table shows that the DAG with exception edges drastically reduced the total number of nodes. The third row shows an average BB node with an exception edge contains over four instructions, while one without exception edges has slightly over one. Thus, the opportunity to apply local (intra-block) optimizations is increased. 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. A similar result has been obtained by Choi et al. in [23].

### 6.2 Performance Improvement by eliminating exception checks and exception constraints

We measured the effectiveness of eliminating exception checks and exception constraints by running micro-benchmarks, the Java Grande Benchmark Suite [24] using the kernels of Section II

with SizeA, and SPECjvm98 with the size of 100. Table 2 describes the Java programs used in the experiments.

In all micro-benchmarks, two-dimensional arrays are frequently accessed in kernel loops. A two-dimensional array consists of linked pointer vectors for simplicity of the language design and flexibility of the declaration of an array in Java. However, no optimization can remove the exception checks for the elements in the second and higher dimensions. From the point of the view of performance, the design is poor. Good performance improvement was expected by applying exception speculation that moves S-PEIs over H-PEIs in any dimension speculatively. We choose the Java Grande Benchmark Suite for measuring the effectiveness for computationally intensive programs and SPECjvm98 for measuring the effectiveness for general-purpose benchmark programs.

Туре	Benchmarks	Description
Micro-benchmarks	Matrix Multiply	Multiply two two-dimensional dense matrices.
Micro-Denchmarks	All-pairs Shortest-path	Find the shortest path from the start vertex to each of the other vertices.
	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.
Java Grande Benchmark Suite	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.
	compress	LZW compression and decompression.
	jess	NASA's CLIP expert system.
	db	Search and modify a database.
SPECjvm98 benchmarks	javac	Source to bytecode compiler.
	mpegaudio	Decompress audio file.
	mtrt	Multi-threaded image rendering.
	jack	Parser generator generating itself.

Table 2.	Description	of benchmarks	characteristics
----------	-------------	---------------	-----------------

Table 3 shows the runtime performance for the micro-benchmarks, and Figure 3 shows the performance improvements of the micro-benchmarks, 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 exception check eliminations (EE). Our baseline is to avoid applying exception speculation and exception elimination optimizations (denoted as "no ES and no EE"). Our exception check elimination algorithms are based on forward dataflow analysis, backward dataflow analysis, and loop versioning. Loop versioning [4] is a technique to eliminate all the exceptions in the nested loop by creating a safe region, within which the compiler is free to apply optimizations such as instruction reordering. They are software approaches for eliminating S-PEIs as described in Section 2. Exception speculation is effective regardless of whether or not any exception elimination optimizations are applied.

When exception check eliminations are not performed, exception speculation improves the performance by 28% and 30%. It is effective for all benchmarks, since there are many S-PEIs in most of these kernel loops, as is typical in Java programs. When exception check eliminations are performed, exception speculation improves the performance by 0% and 30%.

(unit: sec)	Matrix Multiply	All-pairs Shortest-path		
No ES and no EE	2.77	10.38		
ES and no EE	2.16	7.99		
No ES and EE	2.34	5.34		
ES and EE	1.80	5.35		

Table 3: Runtime performance measurements for the micro-benchmarks

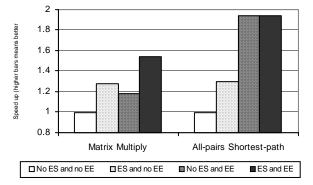


Figure 3. Runtime performance improvements for the micro-benchmarks

Table 4 shows the runtime performance for the Java Grande Benchmark Suite, and the left half in Figure 4 shows the performance improvements for the Java Grande Benchmark Suite for four possible combinations of the optimizations relative to our baseline (no ES and no EE).

When exception check eliminations are not applied, exception speculation alters the performance from -27% to 15% (with an average of 0.0%). It is particularly effective for SOR and SparseMatmul. This is because there are many opportunities to move H-PEI over S-PEI speculatively since no S-PEI is eliminated. When exception check eliminations are applied, exception speculation changes the performance from -10% to 13% (with an average of 1.7%). It is also effective for SOR and SparseMatmul. This is because the kernel loop of SOR accesses twodimensional arrays frequently, and the kernel loop of SparseMatmul has indirect accesses to the array element. Since known exception elimination techniques cannot eliminate S-PEIs for multidimensional arrays and indirect arrays, there are still rooms to move S-PEIs over H-PEIs speculatively.

In HeapSort, the performance is degraded by 27% when exception check eliminations are not performed, despite the fact that the critical path length in a kernel loop can be reduced by 17%. In Crypt, the performance is degraded by 10% when exception check eliminations are performed. Since NullPointerException or IndexOutOfBoundsException do not occur in these benchmarks, no non-faulting load instruction with an invalid effective address will ever be executed. Thus, the recovery code is not executed due to deferred exceptions by a memory fault. Both performance degradations are probably due to deferred exceptions from the TLB misses, leading to the execution of the recovery code.

Table 5 shows the runtime performance for SPECjvm98, and the right half in Figure 4 shows the performance improvements of SPECjvm98 for four possible combinations of the optimizations relative to our baseline (no ES and no EE).

When exception check eliminations are not applied, exception speculation alters the performance from -2% to 5% (with an average improvement of 1.1%). It is particularly effective for compress and mpegaudio, both of which include many S-PEIs. When exception check eliminations are applied, exception speculation changes the performance from -2% to 12% (with an average of 1.3%). It is effective for mpegaudio and mtrt. In the case of mpegaudio, its kernel loop frequently accesses two-dimensional arrays. 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 arrays or instance variables frequently.

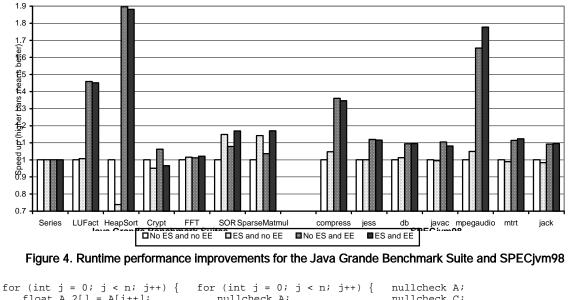
Example 5 a) shows a part of the kernel loop from mpegaudio. Example 5 b) is an intermediate representation compiled from Example 5 a) before applying exception check eliminations. Our exception check eliminations move the exception checks for the arrays A and C out of a loop, because the complete ranges of i and j within the loop can be determined by compile time analysis and the compiler can generate a safe loop where all exception checks on the arrays A and C are eliminated. The result is shown in Example 5 c). Since the value of the array C is loop-variant and a two-dimensional array consists of linked pointer vectors, no optimization can remove the exception checks for the element accesses of the array C 2. Thus, exception speculation effectively eliminates the exception dependence associated with these accesses. In fact, exception speculation reduces the critical path length from 37 to 35 cycles for the entire kernel loop.

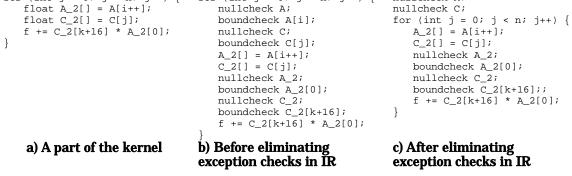
Since NullPointerException or IndexOutOfBoundsException do not occur in the program. any non-faulting load instruction with an invalid effective address will never be executed. Thus, the recovery code is not executed due to deferred exceptions from a memory fault. The performance degradation is probably due to deferred exceptions by the TLB misses, leading to the execution of the recovery code. We also observed that exception speculation is effective for programs that accesses indirect arrays or multi-dimensional arrays in the cases where those known exception elimination techniques cannot reduce the number of exception checks. Thus, exception speculation is still effective even when known exception elimination techniques are applied.

Table 4: Runtime performance measurements for the Java Grande Benchmark Suite								
(unit: sec)	Series	LUFact	HeapSort	Crypt	FFT	SOR	SparseMatmul	
No ES and no EE	33.58	2.86	2.37	3.07	23.08	10.39	8.96	
ES and no EE	33.56	2.84	3.21	3.23	22.73	9.05	7.85	
No ES and EE	33.59	1.96	1.25	2.89	22.81	9.64	8.65	
ES and EE	33.57	1.97	1.26	3.18	22.60	8.89	8.89	

Table 5: Runtime performance measurements for SPECJVM98							
(unit: sec)	compress	jess	db	javac	mpegaudio	mtrt	jack
No ES and no EE	23.04	10.57	38.41	13.22	20.53	6.67	9.68
ES and no EE	22.00	10.57	37.93	13,29	19.56	6.74	9.84
No ES and EE	16.93	9.45	35.11	11.97	12.41	5.99	8.86
ES and EE	17.11	9.47	35.08	12.23	11.55	5.94	8.84

Table 5: Runtime performa	ance measurements for SPECjvm98
---------------------------	---------------------------------





Example 5. A part of the kernel loop of mpegaudio

#### 6.3 Code Size Expansion

We also measured the static code size expansion when exception check eliminations and exception speculation are performed.

Figure 5 shows the static code size expansion of each kernel method relative to that of our baseline (no ES and no EE) for four possible combinations. Exception check eliminations reduce the code size by an average of 8.8% when exception speculation is not performed. In Matrix Multiply, the code size is larger than that of the baseline even when exception check eliminations are performed. This is because loop versioning duplicates the kernel loop to create a safe region even for the case in which our exception check eliminations do not remove exception checks.

Exception speculation increases the static code size from 55% to 79% without exception check elimination, and it increases the static code size from 27% to 44% with exception check eliminations.

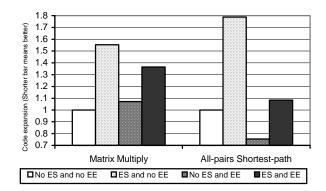


Figure 5. Static code size expansion of a kernel method for the micro-benchmarks

The left half in Figure 6 shows the static code size expansion of the Java Grande Benchmark Suite relative to our baseline (no ES and no EE) for four possible combinations. Exception speculation increases the static code size from 7.3% to 11.0% (with an average of 8.5%) when exception check eliminations are not performed. Exception speculation increases the static code size from 2.9% to 5.6% (with an average of 3.6%) when exception check eliminations are not performed. Unlike the micro-benchmarks, the code growth is not significant.

The right half in Figure 6 shows the static code size expansion for SPECjvm98 relative to our baseline (no ES and no EE) for four possible combinations. Exception speculation increases the static code size from 3.4% to 10.0% (with an average of 5.9%) when exception check eliminations are not performed. Exception speculation increases the static code size from 0% to 7.9% (with an average of 3.0%) when any exception check elimination techniques are not performed. Unlike the micro-benchmarks, the code growth is not significant. Since SPECjvm98 consists of non-numerical programs, code growth is smaller than for the Java Grande Benchmark Suite. The largest code growth (7.9%) was observed for mpegaudio along with its significantly better performance. This indicates that exception speculation is applied aggressively here.

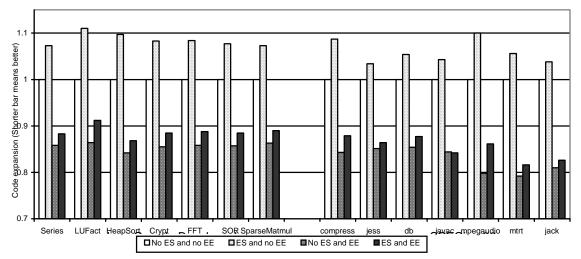


Figure 6. Static code size expansion for the Java Grande Benchmark Suite and SPECjvm98

#### 7 Conclusions

We have proposed a new exception speculation framework 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 can be effectively selected to schedule a sequence of the most profitable instructions across every Java exception check (S-PEI), without generating any extra copy instructions. Unlike the previous work of Arnold et al. [15], our framework does not introduce any conditional branches. Our experimental results confirm that our framework reduces the space for the DAG representation during the compilation. For the recovery code generation, our proposed method does not increase the register pressure on the critical path nor the restrictions on the final code scheduling (bundle formation) phase.

Our preliminary results on an IA-64 Itanium processor show that exception speculation improves the performance by 30% for one of the micro-benchmark programs, and up to 13% (with an average of 1.7%) for the Java Grande Benchmark Suite, and up to 12% (with an average of 1.2%) for SPECjvm98, even when exception check eliminations are performed. The experiments show that our speculation technique is effective particularly for those programs that are computationally intensive or that access multi-dimensional arrays frequently in the cases where those known exception elimination techniques cannot reduce the number of exception checks. Thus, exception speculation is complementarily effective even when known exception elimination techniques are applied. We also observe some performance degradations that are probably due to deferred exceptions from the TLB misses, leading to the execution of the recovery code.

#### Acknowledgement

We thank the staff of Network Computing Platform at Tokyo Research Laboratory for implementing our JIT compiler. We would like to thank the anonymous reviewers for their useful comments.

#### References

- 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] Intel Corp. "IA-64 Application Developer's Architecture Guide." http://developers.intel.com/design/ia64/downloads/adag.htm.
- [4] 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.
- [5] J. Whaley. "Dynamic Optimization through the Use of Automatic Runtime Specialization." M. Eng. thesis, Massachusetts Institute of Technology, 1999.

- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] J. Knoop, O. Ruthing, and B. Steffen. "Optimal Code Motion: Theory and Practice." ACM Transactions on Programming Languages and Systems, Vol.17, No. 5, pp. 777-802, 1995.
- [18] 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.
- [19] 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.
- [20] V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. "Translating Out of Static Single Assignment Form." In Proceedings of Static Analysis Symposium, LNCS 1694, pp. 194-210, Springer Verlag, 1999.

- [21] Intel Corp. "Itanium<sup>TM</sup> Processor Microarchitecture Reference." http://developers.intel.com/design/ia64/downloads/245474.htm.
- [22] Standard Performance Evaluation Corp. SPEC JVM98 Benchmarks, available at http://www.spec.org/osg/jvm98/.
- [23] J.-D. Choi, D. Grove, M. Hind, and V. Sarker. "Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs." In Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 21-31, 1999.
- [24] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. "A Methodology for Benchmarking Java Grande Applications." In Proceedings of ACM 1999 Java Grande Conference, pp. 81-88, 1999.