Optimizing Java Programs in the Presence of Exceptions

Manish Gupta, Jong-Deok Choi, and Michael Hind

IBM Thomas J. Watson Research Center, P.O. Box 704,
Yorktown Heights, NY 10598, USA
{mgupta,jdchoi,hindm}@us.ibm.com

Abstract. The support for precise exceptions in Java, combined with frequent checks for runtime exceptions, leads to severe limitations on the compiler's ability to perform program optimizations that involve reordering of instructions. This paper presents a novel framework that allows a compiler to relax these constraints. We first present an algorithm using dynamic analysis, and a variant using static analysis, to identify the subset of program state that need not be preserved if an exception is thrown. This allows many spurious dependence constraints between potentially excepting instructions (PEIs) and writes into variables to be eliminated. Our dynamic algorithm is particularly suitable for dynamically dispatched methods in object-oriented languages, where static analysis may be quite conservative. We then present the first software-only solution that allows dependence constraints among PEIs to be completely ignored while applying program optimizations, with no need to execute any additional instructions if an exception is not thrown. With a preliminary implementation, we show that for many benchmark programs, a large percentage of methods can be optimized (while honoring the precise exception requirement) without any constraints imposed by frequent runtime exceptions. Finally, we show that relaxing these reordering constraints can lead to substantial improvements (up to a factor of 7 on small codes) in the performance of programs.

1 Introduction

Java [14] continues to gain importance as a popular object-oriented programming language for general-purpose programming. Although some aspects of Java, such as strong typing, simplify the task of program analysis and optimization, other aspects, such as support for precise exceptions, can hamper program analysis and optimizations. The Java language specification requires that exceptions be *precise*, which implies that

1. when an exception is thrown, the program state observable at the entry of the corresponding exception handler must be the same as in the original program; and
2. exception(s) must be thrown in the same order as specified by the original (unoptimized) program.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 422–446, 2000.

To satisfy the precise exception requirement, Java compilers disable many important optimizations across instructions that may throw an exception (we refer to these *potentially excepting instructions* as PEIs [19]). This hampers a wide range of program optimizations such as instruction scheduling, instruction selection (across a PEI), loop transformations, and parallelization. Furthermore, PEIs are quite common in Java programs – frequently occurring operations such as reads and writes of instance variables, array loads and stores, method calls, and object allocations may all throw an exception. Hence, the ability of the compiler to perform any program transformation that requires instruction reordering is severely limited, which impedes the performance of Java programs. This paper presents a framework that enables aggressive program transformations to be applied in the presence of precise exceptions. Our approach relies on the following intuition. First, the program state that needs to be preserved when an exception is thrown is often a very small subset of the program state that is conservatively preserved by compilers to support precise exceptions. By identifying this subset, many spurious constraints on instruction reordering can be removed. Second, exceptions are rarely thrown by correctly executing Java programs,[1] and so it is desirable to optimize the program for this expected case even at the expense of some inefficiency when an exception is thrown.

This paper makes the following contributions:

- It presents both dynamic and static analyses to identify the subset of program state that needs to be preserved if an exception is thrown. In particular, it presents a novel approach to optimize procedures based on runtime propagation of relevant properties of the calling environment to callee procedures. This allows dependences between PEIs and other instructions that modify the value of variables not live at the exception handler to be ignored during instruction-reordering optimizations.
- It presents a framework that allows dependences among PEIs to be completely ignored during program transformations that reorder instructions. This framework is based on a novel algorithm to generate compensation code, which ensures that the same exception is raised as in the original unoptimized program when an exception is thrown. Our algorithm does not rely on any special hardware support and does not incur the overhead of executing *any* additional instructions in the expected case when no exception is thrown. To the best of our knowledge, no other approach has this property.

This paper also presents measurements of relevant benchmark characteristics and preliminary experimental results. A significant percentage of instructions in the benchmarks are PEIs, pointing to the importance of our technique for overcoming dependences among PEIs, which can be applied to any code with PEIs. In 11 out of 13 benchmarks, over 65% of methods are recognized as targets of our aggressive optimization techniques for ignoring program-state dependences

[1] This is particularly true for the predefined *runtime* exceptions, which typically account for most of the exception checks in Java programs. User-defined exceptions are sometimes used for normal control flow.

at PEIs; and using dynamic analysis, over 96% of the method invocations can be aggressively optimized in 9 of those benchmarks. We also demonstrate that significant speedups, up to a factor of 7 on small programs, can be obtained using our techniques and hand-application of well-known program transformations.

The rest of the paper is organized as follows. Section 2 describes background for this work. Section 3 describes the first part of our framework which eliminates spurious dependence constraints by identifying the subset of program state that needs to be preserved if an exception is thrown. Section 4 describes the second part of our framework which allows dependences among PEIs to be completely ignored for performing program optimizations. Section 5 demonstrates empirically that our techniques are applicable to real Java programs and suggests that these techniques can lead to substantial improvement in the performance of programs. Section 6 describes related work and Section 7 presents conclusions.

2 Background

This section reviews background material on the specification of exceptions in Java and on the modeling of exceptions in our compiler's intermediate program representation.

2.1 Exceptions in Java

Java defines `try-catch` blocks that govern control flow when an exception is thrown by a statement – execution proceeds to the closest dynamically enclosing `catch` block that handles the exception [14]. Java exceptions (represented by `Throwable` classes) fall into one of four categories: *runtime exceptions*, *checked exceptions*, *errors*, and *asynchronous exceptions*. Of these, only the first two require a compiler to generate code that precisely preserves the program state when the exception is thrown. Runtime exceptions include several language-defined exceptions, such as `NullPointerException`, `IndexOutOfBoundsException`, and `ArithmeticException`. Several bytecode instructions can potentially throw a runtime exception, such as field-access instructions, array-access instructions, integer-division instructions, and method-call instructions.

If a thread fails to catch a thrown exception, it executes the `uncaughtException` method of its *thread group*, and then terminates [14]. If the `uncaughtException` method is not overridden by the application for the thread group of the excepting thread, eventually the `uncaughtException` method of the `system` thread group is invoked. This default method simply prints the stack trace at the point of exception.[2] Any other threads associated with the application are not directly affected. It is possible for the user to override the exception handler that catches an exception not caught by any handler in the user code, by specifying the `uncaughtException` method for a thread group.

[2] It does not use any other part of the program state and is followed by the termination of the thread, a fact that we exploit, as described in Section 3.

2.2 Basic Framework to Model Exceptions

We use a form of the *program dependence graph* [7] to model control and data dependences, with a special representation for exception-related dependences. This representation exploits ideas from the *factored* control flow graph (FCFG) [9] to model control flow due to exceptions. The FCFG representation does not terminate basic blocks at PEIs, which results in larger basic blocks and fewer edges than a regular control flow graph. The low-level intermediate representation (LIR) of the Jalapeño optimizing compiler [6] we employ for our optimization uses *condition registers* to represent the *exception-conditional dependences*. An instruction that is exception-conditionally dependent on exception-check instructions should execute only if none of these exception-check instructions indicates that an exception should be thrown. (We treat exception-conditional dependences in the same manner as data or control dependences in this paper.)

In addition to the usual control and data dependence edges, our program dependence graph (which we subsequently refer to as the *dependence graph*) has edges for two kinds of precise-exception related dependences: (i) *program-state* dependences, which ensure that a write to a nontemporary variable is not moved before or after a PEI, in order to maintain the correct program state if an exception is thrown, and (ii) *exception-sequence* dependences among PEIs, which ensure that the correct exception is thrown by the code. Our goal is to eliminate as many of these precise-exception related dependences as possible, so that a program can be optimized without paying a heavy performance penalty for the precise exception semantics of Java. In the following, we will refer to these precise-exception related dependences simply as *exception-related dependences*.

Program Example: Fig. 1 shows a simple Java class containing a method and a low-level intermediate representation, LIR, with optimization. Condition registers, such as $c1$ at dependence graph node $P1$, represent the exception-conditional dependences of instructions on exception-check instructions. For example, instructions of nodes $n2$ and $n3$ should be executed only if the `null_check` instruction of $P1$ succeeds without throwing the `NullPointerException`. As shown in the figure, the Jalapeño optimizing compiler removes redundant instructions for exception checking [7].[3] However, the compiler does not move stores across PEIs or reorder PEIs.

Fig. 2 shows the dependence graph (before applying our techniques) for the LIR shown in Fig. 1. The regular data and control dependence edges are shown as thin lines, the program-state dependences are shown as dotted lines, and the exception-sequence dependences are shown as dashed lines. We have not shown the dependences that are transitively satisfied by other sequences of dependences (e.g., the program state dependence from $P7$ to $n9$ is covered by dependences from $P7$ to $n8$ and from $n8$ to $n9$). The critical path of the graph shown in the figure has a length of 10: $P1$, $P5$, $n6$, $P7$, $n8$, $n9$, $P10$, $n11$, $P12$, $n13$, and $n14$.

[3] For example, only one null check is performed on variable `p`, even though it is dereferenced three times in the original program.

```
public class EXCEPT {
   int f, g;
   static void foo(int a[], int b[], int i, EXCEPT p) {
       a[i] = p.f + p.g;
       b[i] = p.g;
   }
}
```

```
Bytecode                                                          Dependence
Offset    Operator    Operands                                    Graph Node
------    --------    --------                                    ----------
          label       B0
  3   PEI null_check  c1 = p                                        P1
  3       int_load    t0 = @{p, -16}, <EXCEPT.f>, [c1]            n2  // t0 = p.f
          int_load    t1 = @{p, -20}, <EXCEPT.g>, [c1]            n3  // t1 = p.g
 10       int_add     t2 = t0, t1                                 n4  // t2 = t0 + t1
 11   PEI null_check  c2 = a                                      P5
          int_load    t5 = @{a, -4}, [c2]                         n6  //
 11   PEI trap_if     c3 = t5, i, <=U, [c2]                       P7  // bounds_check
          int_shl     t6 = i, 2, [c2, c3]                         n8
 11       int_store   t2, @{a, t6}, [c2, c3]                      n9  // a[i] = t2
 18   PEI null_check  c4 = b                                      P10
          int_load    t7 = @{b, -4}, [c4]                         n11 //
 18   PEI trap_if     c5 = t7, i, <=U, [c4]                       P12 // bounds_check
          int_shl     t8 = i, 2, [c4, c5]                         n13
 18       int_store   t1, @{b, t8}, [c4, c5]                      n14 // b[i] = t1
 19       return                                                  n15
          end_block   B0
```

Fig. 1. An example Java code segment and its LIR. Instructions introduced by optimization do not have a bytecode offset. Operands beginning with "c" are condition registers.

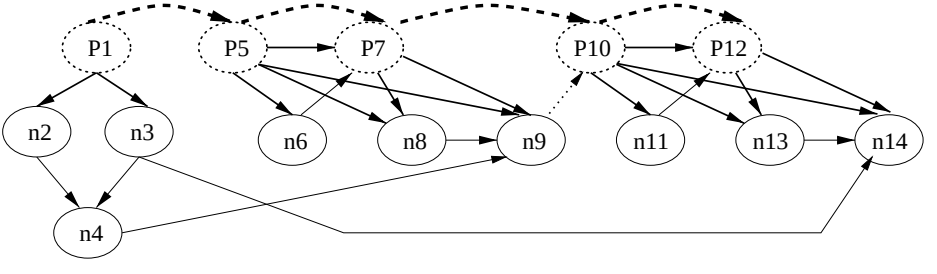

Fig. 2. Dependence Graph for LIR in Fig. 1

.

3 Elimination of Program State Dependences

This section describes an algorithm to identify the subset of program-state dependences that need not be preserved at each PEI. For the purpose of dependence analysis, each PEI, p, represents a use of all variables, i.e., memory locations, that are live (possibly used before defined) on entry to any handler for the exception potentially thrown by p. We refer to this set as $use(p)$. (Section 4 presents a technique to allow exception-sequence dependences to be effectively ignored for the purpose of optimizations.) Section 3.1 describes an analysis that identifies the set of variables that are live on entry to a given exception handler. This analysis takes multithreading into account. Although it is presented separately for clarity, the analysis is applied as part of the algorithm, described in Section 3.2, to propagate information about live variables at enclosing exception handlers to various program points. We describe two algorithms, one using runtime analysis and the other using compile-time analysis, to perform this propagation. Any of these algorithms may be used in a static or a dynamic compiler. Section 3.2 also describes how the liveness information is used to eliminate spurious program state dependences. Section 3.3 describes an extension to improve the quality of the analysis.

3.1 Liveness Analysis at Exception Handler

In this work, liveness analysis is performed at exception handlers in a very coarse-grain manner, so that the information can be compactly represented and propagated along the calling chain of methods in an efficient manner. In particular, we try to capture the common case where only the exception object and some I/O-related variables, such as files and stream objects, are live on entry to an exception handler. We refer to these variables as *exception status-reporting* (ESR) variables. The set of live variables at entry to an exception handler H is

$$L_{IN}(H) = UE(H) \cup [L_{OUT}(H) - MustDef(H)],$$

where $UE(H)$ is the set of variables with *upward-exposed use* (i.e., a use not dominated by a must definition) in H, $L_{OUT}(H)$ is the set of live variables at exit from H, and $MustDef(H)$ is the set of variables definitely written in H. $L_{OUT}(H)$ must include variables that are live not only in the current thread, but also in other threads.

We compute $UE(H)$ by scanning the `catch` block code to check if it only uses ESR variables or local variables declared within the `catch` block (local variables don't contribute to an upward-exposed use for the block). Otherwise, $UE(H)$ is conservatively set to include all nonlocal variables (i.e., variables not locally declared in H) that are visible in H[4] – in this case, further computation of $L_{OUT}(H)$ is not needed. For simplicity, one can conservatively use $MustDef(H) = \emptyset$ in the liveness computation.

[4] Local variables of a method are not visible to an exception handler appearing outside that method.

$L_{OUT}(H)$ is computed as follows:

$$L_{OUT}(H) = LE_{OUT}(H) \cup LNE_{OUT}(H).$$

These two components, $LE_{OUT}(H)$ and $LNE_{OUT}(H)$, respectively account for the following two possibilities: (1) execution of the handler H itself ends abnormally with an exception being thrown (possibly due to a PEI in H), or (2) the execution of the handler ends normally.

We obtain the first component as

$$LE_{OUT}(H) = \bigcup_{e \in E} L_{IN}(EEH(H, e)),$$

where E denotes the set of exceptions possibly thrown by H, and $EEH(H, e)$ denotes the set of enclosing exception handlers that can catch the exception e thrown by H. This component is computed as part of the algorithm, described in Section 3.2, to identify the dynamically enclosing exception handlers for different exceptions at an arbitrary point in the program.

To compute $LNE_{OUT}(H)$ (liveness at exit from H, when no exception is thrown by H), we use a simple algorithm that checks for one of the following cases: (i) H ends with a system exit, (ii) H ends with an exception being thrown by an explicit `throw` statement, (iii) H is followed by termination of the current thread, or (iv) default, when none of the other cases apply. In the first case, where the `catch` block code terminates by calling the `System.exit` method, $LNE_{OUT}(H) = \emptyset$, because the entire application is terminated after H is executed. In the second case, again, $LNE_{OUT}(H) = \emptyset$, because the execution of H does not end normally.[5] This case includes a common idiom where the `catch` block code rethrows the same exception after performing some bookkeeping action, such as releasing a lock. In the third case, which holds for the default `uncaughtException` method of the `System` thread group, $LNE_{OUT}(H)$ is conservatively estimated as the set of variables that are not local to the current thread. *Escape analysis* [10,4,5] can be used to identify thread-local objects. In particular, the algorithm presented in [10] has been shown to often identify a large percentage of objects as thread-local. In the fourth case, $L_{OUT}(H)$ is conservatively set to include all variables visible on exit from H.

Finally, computing the union of $L_{OUT}(H)$ with $UE(H)$, conservatively assuming $MustDef(H) = \emptyset$, leads to one of the following possible values of $L_{IN}(H)$: (i) only ESR variables, (ii) ESR variables and variables that are not thread-local, or (iii) all nonlocal variables (variables not declared in H) visible on exit from H – we shall refer to such an exception handler as *nontrivial*. Two bits are sufficient to encode this liveness information for a given handler. We can use one of the bits (say, bit 0) to encode the liveness of thread-local variables, and the other bit to encode the liveness of variables that are not thread-local (always implicitly assuming that the ESR variables are live). Therefore, the three possible values above are encoded as, respectively, 00, 10, and 11. This encoding has the advantage that a union operation can be performed using a bitwise *or* operation,

[5] The computation of $LE_{OUT}(H)$ completely accounts for $L_{OUT}(H)$ in this case.

which is convenient when a single word is used to record liveness information for multiple handlers for various exception types. It is conceptually straightforward to extend our framework to use more precise (and expensive) liveness analysis.

3.2 Identification of Enclosing Exception Handlers and Propagation of Liveness Information

A simple control flow analysis of `try-catch` blocks, together with static type information, is sufficient to identify the possible enclosing exception handlers, if any, for a PEI within its method. If the type of exception thrown by a PEI can be assigned to (is either the same or a subtype of) the declared exception type of the `catch` clause, we regard the exception handler (corresponding to that `catch` block) as enclosing the PEI. In a `try-catch-finally` construct [14], we regard the `finally` block as enclosing each PEI in the `try` block.

An instruction that does not appear in a `try` block within a method may still be dynamically enclosed in a `try` block due to a calling ancestor of the given method. We describe two interprocedural algorithms to obtain information about the dynamically enclosing exception handler(s) that may catch an exception thrown by an instruction: one is a dynamic analysis, i.e., it uses runtime information, and the other is a static analysis; it uses purely compile-time information.

Consider a call by method A to method B inside a *try* block with k *catch* clauses, where each pair $[H_i, E_i], 1 \leq i \leq k$ represents the exception handler H_i and the exception type E_i caught by it. The basic idea is to propagate the liveness information about the enclosing exception handler(s) (LEEH), $([L_{IN}(H_i), E_i], 1 \leq i \leq k)$, to the callee node B and to each method transitively called by B. Our algorithm using dynamic analysis performs this propagation precisely to the relevant methods, while the one using static analysis may conservatively propagate this information to methods where it is not necessary. In both algorithms, an initial compile-time analysis is performed separately to identify any method that overrides the `uncaughtException` method of any thread group. If such a method H' exists, the pair $[L_{IN}(H'), Exception]$ is conservatively added to the LEEH information for the main method, where *Exception* denotes the Java `Exception` class from which all exceptions that need to be handled precisely are derived.

Dynamic Analysis The key idea behind our dynamic analysis is to propagate runtime liveness information about enclosing exception handlers to each method activation. This information is passed in the form of an extra LEEH parameter. The set of exceptions tracked by this analysis can be determined by a compile-time linear scan of the complete program to identify the set of exceptions potentially thrown by any instruction in the program.[6] For simplicity, in this work, we do not perform such a scan, but instead keep separate liveness information only for the predefined runtime exceptions in Java (i.e., we use two bits,

[6] In general, the complete program is required for such an analysis.

as described in Section 3.2, for each distinct runtime exception and its superclasses). Information on all other handlers, which catch checked, asynchronous, or user-defined exceptions, is summarized with two additional bits, without further distinction among exceptions. Our results, presented in Section 5, suggest that runtime exceptions account for most of the exception checks in typical Java programs.

The total number of predefined runtime exceptions (e.g., the `NullPointerException`) and their superclasses (`RuntimeException` and `Exception`) is small enough that the LEEH parameter can be encoded in a 32-bit word. The LEEH parameter for the `main` method of the application is initialized to all "10" values for each exception type (representing the liveness of only nonthread-local variables and ESR variables at the default `uncaughtException` method provided by the JVM), unioned (using a bit-wise *or* operation) with the liveness encoding for each overriding `uncaughtException` method declared in the program. At each call site not appearing in a `try` block, the LEEH formal parameter value of the caller is passed, without any change, to the callee as the LEEH actual parameter. At a call site inside a `try` block, we require two values to accurately compute the LEEH value passed to the callee: *GEN*, which denotes the encoding of liveness information at the corresponding `catch` blocks (computed as described in Section 3.1), and *KILL*, which is used to override the LEEH information from the caller for exceptions that are definitely caught by one of the given `catch` blocks, and therefore will not be caught by a handler dynamically enclosing the caller. *KILL* is simply encoded as another 32-bit word with the bits "00" appearing for each runtime exception type that is caught by one of the given `catch` blocks, and 1's appearing in all other positions. The LEEH parameter value for the caller is obtained as

$$LEEH_{callee} = (LEEH_{caller} \ \ \& \ \ KILL) \ \ | \ \ GEN,$$

where "&" and "|" represent the bit-wise *and*, *or* operations respectively.

In a dynamic compiler, the runtime information about the actual exception handlers enclosing a method call can be used to optimize the code generation of that method at runtime. Therefore, in a dynamic compiler, no program state dependences are imposed between a PEI and write statements for variables that are not live on entry to the exception handler for that PEI. Alternatively, in a static or dynamic compiler, *cloning/specialization* [21] can be used to obtain different versions of the method, optimized to different degrees based on the incoming LEEH information. At one extreme, only a single additional version of the method may be created to handle the best case where none of the PEIs in the method has an enclosing exception handler with live variables other than ESR variables. We used this approach in obtaining the results presented in Section 5. Furthermore, execution history information about a method may be used to create the specialized version that is most likely to be selected at runtime. We do not discuss these techniques further in this paper.

Consider the example in Fig. 3, where method `main` calls `leaf1` twice, once from within a `try-catch` block and once outside of any such catch block. Because the `catch` block inspects static data, we consider all variables to be live at

```
 public class LIVENESS {
   static int glob;
   static void main() {                    static void leaf1() {
     try {                                   ...
S1:    leaf1();                            }
     }
     catch (NullPointerException e) {      static void leaf2() {
S2:    ... = glob                            ...
     }                                     }
S3:  leaf1();
S4:  leaf2();
   }
 }
```

Fig. 3. Example for static and dynamic liveness analysis

the entry to the handler. However, the second call (at `S3`) does not have such a handler. The dynamic analysis will augment the calls to encode this liveness information at the different calls, allowing for the elimination of spurious program state dependences when `leaf1` is called from `S3`.

Static Analysis The static analysis is performed on the *call graph* representation of the program. In the call graph, each node represents a method, and an edge e_i from node A to node B represents a call site in A that invokes B. For a virtual method call, there is a separate edge for each potential target of the method. Given a call to method B inside a `try` block, the liveness information about enclosing exception handlers (LEEH) is propagated to the callee node(s) for B and to each node reachable from that node along the call graph edges. This ensures, conservatively, that liveness information is propagated to each method with an activation that is dynamically enclosed by the given `try-catch` block. At each method, liveness sets from its different call sites are combined using a union operation.

Finally, for each PEI, the compiler obtains information (using interprocedural analysis described above and using intraprocedural analysis of local `try-catch` blocks) about the live variables at each potentially enclosing exception handler. This information is used to restrict the program state dependences that are imposed between PEIs and write statements in that method.

Our static analysis is conservative relative to the dynamic analysis in two ways. First, for a call to a virtual method inside a `try` block, the enclosing exception handler information is propagated to all methods considered to be targets of that virtual call based on static type analysis. Second, information on call paths is ignored. If a method is called without an enclosing handler (and hence, with the propagation of liveness set of only the default `uncaughtException` handler) along many call paths, but called with a nontrivial handler along even a single call path, all the execution instances of the method are considered to have a nontrivial enclosing handler. On the other hand, the advantage of static analy-

sis is that no extra method parameter is needed to record the runtime liveness information.

Once again consider the example in Fig. 3. Using static analysis, liveness information for all calls is unioned. For example, method `leaf1` in Fig. 3 will be regarded as being enclosed by a catch block of `NullPointerException` due to the call at `S1` although the call at `S3` is not in any catch block. This results in the need to honor program state dependences in `leaf1` (because of the call from `S1`). Program state dependences involving thread-local variables need not be honored in `leaf2` because it does not have a call enclosed by a catch block.

3.3 Impact of Java Memory Model

Our basic analysis, described in Section 3.1, regards all variables visible to other threads as live at an exception handler, unless the exception handler leads to termination of the entire application. Thus, it forces all write statements for those variables to have dependences with respect to the corresponding PEIs. This is a consequence of the current Java memory model not requiring conflicting data accesses in different threads to be synchronized in order to guarantee defined behavior. However, the current memory model has some "serious flaws" in this regard [22] and is in the process of being revised [22,17]. If, as expected, it is revised to require synchronization between writes and reads of a variable in different threads, a write operation performed in an unsynchronized region may be regarded as not necessarily visible to any other thread. (We refer to statements that are dynamically enclosed in a synchronized method or synchronized statement block as constituting a synchronized region). Note that when a PEI p throws an exception within a synchronized region, the default exception handler releases the lock held by the thread, and in that case, we must regard the program state reflecting the effect of instructions up to the excepting instruction as visible to other threads. We identify synchronized regions in our preliminary implementation to demonstrate the additional precision that can be obtained if the revised Java memory model requires synchronization between writes and reads of a variable in different threads.

4 Ignoring Dependences Among PEIs

This section describes how we completely eliminate exception-sequence dependences between PEIs for the purpose of applying program optimizations. The key contribution of this section is an algorithm to generate two sets of code: the *optimized code* that runs ignoring the dependences among PEIs (without relying on any special hardware and without any check-pointing overhead); and the *compensation code* that runs only when an exception occurs during the execution of the optimized code. The role of the compensation code is to intercept any exception thrown by the optimized code, and throw the same exception that would be thrown by the unoptimized code. Section 4.1 describes a transformation that is performed on the dependence graph. Section 4.2 proves the correctness of this

transformation. Section 4.3 describes a modified code generation algorithm and proves its correctness.

The dependence graph that is the basis of the transformation described in this section reflects the results of applying the techniques described in Section 3: the dependence graph contains, along with the regular control and data dependences, any remaining program-state dependences for precise exception semantics that are not eliminated by the techniques described in Section 3.

4.1 Transformation of the Dependence Graph

We perform three transformation steps on the unoptimized dependence graph. The first step splits each PEI node, PEI_i, into two separate nodes: (a) an *exception-monitoring node*, em_i, that determines whether an exception should be thrown and sets its *exception flag*, called ef_i, to `true` or `false`, accordingly; and (b) an *exception-throwing node*, et_i, that actually throws an exception if ef_i is set `true` by em_i. After this PEI-node splitting, the original exception-sequence dependences are preserved by the second step, which inserts an exception-sequence edge from em_i to et_i for each PEI_i, and replaces each exception-sequence edge from PEI_i to PEI_j with an exception-sequence edge from et_i to em_j. Finally, the third step replaces each exception-sequence edge from et_i to em_j with an exception-sequence edge from et_i to et_j. This step, therefore, has the effect of eliminating the exception-sequence dependences among exception-monitoring nodes.

The transformed dependence graph consists of two components: the *optimized dependence subgraph*, which does not have any exception-throwing nodes, and the *exception-thrower subgraph*, which has only exception-throwing nodes. The transformed dependence graph allows two additional optimizations not possible with the unoptimized dependence graph: (1) delaying throwing exceptions by et_i after em_i sets ef_i true, and (2) reordering exception-monitoring nodes (modulo control/data dependences). The first comes from splitting em_i and et_i, and the second comes from deleting the exception-sequence edges among em nodes.

Let us revisit the LIR shown in Fig. 1. The *transformed dependence graph* in Fig. 4 is the result of this transformation applied to the unoptimized dependence graph in Fig. 2. Of the two subgraphs of the transformed dependence graph, the optimized subgraph (in Fig. 4-(B)) becomes the basis for our optimization. The critical paths (there are two) of the optimized subgraph alone have lengths of 4 each: $em5$, $n6$, $em7$, $n8$, $n9$; and $em10$, $n11$, $em12$, $n13$, $n14$. Shorter critical paths in the dependence graph result in more freedom in code reordering. (In the figure, the program-state dependence from $n9$ to $em10$ has been deleted after liveness analysis.)

The next section shows that generating optimized code, based on honoring the dependences in the transformed dependence graph and skipping the execution of certain statements when an exception flag is set, leads to no violation of the precise exception semantics. In Section 4.3, we describe a modified algorithm for code generation that leads to more efficient optimized code, and prove its correctness as well.

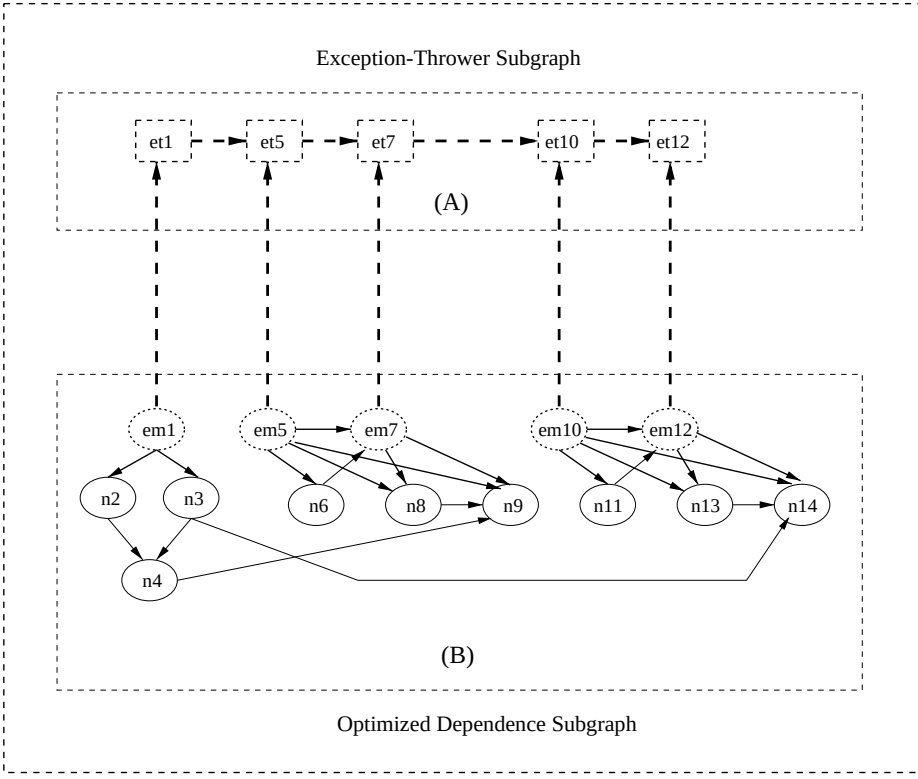

Fig. 4. Transformed dependence graph for LIR in Fig. 1. (A) is the exception-thrower subgraph. (B) is the optimized dependence subgraph. (A) and (B) combined constitute the transformed dependence graph. Rectangular nodes in (A) are exception-thrower nodes, and dotted oval nodes in (B) are exception-monitoring nodes.

4.2 Correctness

This section proves the correctness of the optimized code for the case when the control flow graph (CFG) is acyclic, i.e., with no loops in the procedure. We show the correctness by proving that the code generated from the transformed dependence graph throws the same exception that the code generated from the original dependence graph. Our transformation is valid in the presence of arbitrary control flow, and the proof can indeed be extended to a cyclic CFG by employing the concepts of loop iteration vectors and loop-carried dependences.[7] We omit the more general proof due to its complexity.

We first define a few terminologies to be used below as follows:

– P_o denotes the optimized code, and P_u denotes the unoptimized code.

[7] In general, the control flow graph (CFG) need not be preserved in order to have a program transformation that preserves the program semantics, as long as the control/data dependences are preserved by the transformation.

- $n_i \xrightarrow{*} n_j$ indicates that there exists a CFG path, possibly zero length, from n_i to n_j. In other words, n_i executes before n_j (because we assume an acyclic CFG).
- $n_i \xrightarrow{es} n_j$ indicates that there exists an exception-sequence dependence edge from n_i to n_j.
- $n_i \xrightarrow{es,*} n_j$ indicates that there exists an *exception-sequence path* (a path over exception-sequence dependence edges), possibly zero length, from n_i to n_j.
- $Slice(n_i)$ denotes the set of nodes in the transformed dependence graph that are reachable from n_i. In other words, $n_j \in Slice(n_i)$ if and only if there exists a path from n_i to n_j in the transformed dependence graph.
- $Slice_{ec}(n_i)$, a subset of $Slice(n_i)$, denotes the set of nodes in the transformed dependence graph that are reachable from n_i over *non*-exception-sequence (dependence) edges. In other words, $n_j \in Slice_{ec}(n_i)$ if and only if there exists a path over control/data dependence edges, possibly zero length, from n_i to n_j in the transformed dependence graph. For example, $Slice_{ec}(em1) = \{n2, n3, n4, n9, n14\}$. None of the exception-thrower nodes belongs to $Slice_{ec}(x)$, for any x, because they are reachable only via exception-sequence edges.

For convenience, PEIs are indexed from 1 to N based on a topological ordering of the PEI nodes: if $i < j$, there is no execution path from n_j to n_i (because we assume an acyclic CFG). We also use min_{EX} to denote the minimum number among the indices of the exception-flags that are `true`, i.e.,

$$min_{EX} = k \text{ if } \forall j, 1 \leq j < k, ef_j = \texttt{false} \wedge ef_k = \texttt{true}.$$

Note that if $min_{EX} = k$ in P_u, PEI_k is the (first) one that throws the exception in P_u.

Our optimization scheme ensures that the correct exception is thrown by P_o when multiple *em* nodes in P_o set their exception flags (to `true`) by skipping the execution of $Slice_{ec}(em_j)$ in P_o if $ef_j = \texttt{true}$. The following theorem shows the correctness of this scheme.

Theorem 1. *Let the execution of $Slice_{ec}(em_j)$ in P_o be skipped if $ef_j = \texttt{true}$. Then, $min_{EX} = k$ in $P_o \Longleftrightarrow min_{EX} = k$ in P_u.*

Before proving the theorem, we first present Property 1, which states the condition for a correct out-of-order execution of instructions in an optimized code. We assume that any out-of-order optimization performed by the compiler observes this property.

Property 1. Let $n_i \xrightarrow{*} n_j$ in P_u. Execution of n_j can precede that of n_i in P_o only if $n_j \notin Slice(n_i)$.

We prove Theorem 1 based on the following lemmas.

Let $EM = em_{m(1)} \xrightarrow{*} em_{m(2)} \cdots \xrightarrow{*} em_{m(k)}$ in P_o be an out-of-order execution of exception-monitors such that $m(1) > m(2) > \cdots > m(k)$ in P_u.

Lemma 1. *Let the execution of* $Slice_{ec}(em_{m(i)} \in EM), 1 \leq i < k$, *in* P_o *be skipped. Then,*

$$ef_{m(k)} = \texttt{true} \text{ in } P_o \Longleftrightarrow ef_{m(k)} = \texttt{true} \text{ in } P_u.$$

Proof: Let j be any $m(i), 1 \leq i < k$. The only side effect of $em_j \in EM$ is setting ef_j. Therefore, executing em_j before $em_{m(k)}$ does not affect the program state of $em_{m(k)}$. Also, $PEI_{m(k)} \xrightarrow{*} PEI_j$ means there exists an exception-sequence path from $em_{m(k)}$ to et_j, which prevents et_j from prematurely throwing an exception before $em_{m(k)}$ executes. Finally, any node $n_q \in Slice(em_j)$ comes after $em_{m(k)}$ in P_u because $PEI_{m(k)} \xrightarrow{*} PEI_j$, and skipping them should not affect the program state of $em_{m(k)}$. □

Lemma 2. *Let* $ef_{m(i)} = \texttt{false}, 1 \leq i < k$. *Then,*

$$ef_{m(k)} = \texttt{true} \text{ in } P_o \Longleftrightarrow ef_{m(k)} = \texttt{true} \text{ in } P_u.$$

Proof: Again, let j be any $m(i), 1 \leq i < k$. Any node $n_q \in Slice_{ec}(em_j) \cup em_j$, that is executed before $em_{m(k)}$ must not affect $em_{m(k)}$ for correct optimization, as stated by property 1. Since $ef_j = \texttt{false}$, executing $n_q \in Slice_{ec}(em_j)$ will not result in any illegal operation, either. □

Proof (of Theorem 1): According to Lemma 1 and Lemma 2, either skipping the execution of $Slice_{ec}(em_{m(i)} \in EM)$ or $ef_{m(i)} = \texttt{false}, 1 \leq i < k$, ensures that

$$min_{EX} = m(k) \text{ in } P_o \Longleftrightarrow min_{EX} = m(k) \text{ in } P_u.$$

Therefore, skipping the execution of $Slice_{ec}(em_{m(i)})$ when $ef_{m(i)} = \texttt{true}$ still ensures the same. □

Theorem 1 also shows how optimized code might be generated to guarantee the same exception to be thrown that an unoptimized code will throw: when ef_i is `true` all the statements that are transitively control/data dependent on em_i should not be executed (in order not to cause any illegal operations). This scheme, however, penalizes an exception-free execution because execution of statements need to be guarded by code that checks whether the corresponding exception flag is `true` or `false`. The code-generation scheme we employ eliminates this unwanted penalty for exception-free execution at an increased cost of *exceptional execution*, i.e., when an exception is thrown. This shifting of additional cost from an exception-free execution to exceptional execution, we believe, is the right approach for optimization because exceptions are in general regarded as being *exceptional*. The following section describes our code-generation scheme in detail.

4.3 Generation of Optimized Code and Compensation Code

The core idea of the code generation algorithm is generating two sets of code: the *optimized code* that runs ignoring the dependences among PEIs; and the

compensation code that runs only when an exception occurs during the execution of the optimized code. The optimized code is generated from the optimized dependence subgraph, such as the one shown in Fig. 4-(B). The compensation code is generated from the fully transformed dependence graph, such as the one shown in Figs. 4-(A) and 4-(B) combined, and its behavior follows Theorem 1. The role of the compensation code is to intercept any exceptions thrown by the optimized code, and to throw the same exception that would be thrown by the unoptimized code. The compensation code does not attempt to recover any "lost" program state: the dependence graph prevents code generation that might require any such recovery of the program state when an exception occurs.

Our strategy is to run the optimized code until an exception is thrown. This exception might not be the same (correct) as the exception thrown by the unoptimized code, but will be caught by the compensation code. Then, the compensation code runs as described in the previous section until an exception-throwing node throws the real exception, which will be the correct exception to be thrown by the unoptimized code. This way, performance of the optimized code does not suffer when no exception is thrown by the optimized code. Transfer of execution from the optimized code to the compensation code when an exception occurs is performed by an exception handler: we enclose the optimized code in a `try` block, whose `catch` block contains the compensation code. In a dynamic compiler, the compensation code can be generated on demand when an exception is thrown. In a static compiler, the compensation code may be placed such that it does not pollute the instruction cache during normal execution when no exception is thrown. The following steps describe the details of code generation:

1. We generate code from the transformed dependence graph. This code is the basis of the compensation code.
2. A copy is created of the compensation code, excluding the portion corresponding to the exception-thrower graph. This code, equivalent to code generated from the optimized dependence subgraph, is the basis of the optimized code. Both the optimized code and the compensation code share the same program state, reading from and writing to the same variables.
3. In the optimized code, we replace each exception-monitor code with the corresponding PEI code.
4. Exception-monitors in the compensation code *only* set the exception flags if the results of their exception checks indicate that an exception should be thrown. Guards are introduced in the compensation code so that if the exception flag of an exception-monitor is set, execution of all the statements transitively control/data dependent on the exception-monitor is skipped.
5. The compensation code is put into a catch block that catches any exception thrown by the optimized code. The catch block of the optimized code starts with code, called the *handler prologue*, that examines the exception thrown by the optimized code. The handler prologue sets the exception flag corresponding to the PEI that threw the exception, and transfers the control flow to the instruction immediately after the exception-monitoring node in the compensation code – additional labels are introduced in the compensation code for this purpose.

The following theorem states that the code generated as described above guarantees throwing the same exception that an unoptimized code will throw.

Theorem 2. *The compensation code executes if and only if an exception is thrown by the unoptimized code. Furthermore, the exception thrown by the compensation code is the same as that thrown by the unoptimized code.*

Proof: We prove the theorem by showing that the execution behavior of the optimized code, the compensation code, the handler prologue, and the `try-catch` structure inserted by the code generator, all combined, is the same as that of the compensation code, denoted as P_c, alone. Theorem 1 already showed that P_c and P_u are equivalent in execution behavior.

Let PEI_t be the PEI that throws the exception in P_o. The execution behavior of P_o up to PEI_t is the same as that of P_c up to em_t because (1) they are identical by construction (of the code generation as described above) except for the replacement of PEI nodes in P_o with em nodes in P_c, and (2) no ef_k in P_c, such that $em_k \xrightarrow{*} em_t$, would have been set (otherwise, PEI_t would not be the first PEI to throw an exception). At PEI_t, the exception handler catches the exception thrown by PEI_t and passes the control flow to the handler prologue, which in turn passes the control flow to right after em_t in P_c after setting ef_t to `true`. After that, the execution continues in P_c, which is already shown to be equivalent to P_u in execution behavior by Theorem 1. □

Fig. 5 shows a high-level view of the generated code for method `foo` in Fig. 1. Let us consider an example situation where the original program throws an exception at P1, but during the execution of BODY_A (the optimized code), P5 throws the first exception, which is caught by the generated catch block. The handler prologue sets flag_p5 to `true`, and branches to LABEL_5 in the compensation code, which appears immediately after the instruction corresponding to P5 in the optimized code. Since the execution behaviors of the optimized code and the compensation code are identical (except for the PEIs), the execution continues with the compensation code as it would with the optimized code. At LABEL_5, since the flag is set for the PEI, the compensation code skips the statements in the program that are transitively control/data dependent on $em5$ (i.e., in $Slice_{ec}(em5)$), and continues its execution (eventually) to LABEL_1. Exception-thrower $et5$ can execute only after Exception-thrower $et1$ because of the exception-sequence dependence from $et1$ to $et5$. Exception-thrower node $et1$ can, in turn, execute only after $em1$, due to the exception-sequence dependence from $em1$ to $et1$ in the exception dependence graph. Therefore, even though P5 initially throws an exception during execution of the optimized code, $et1$ will throw the "real" exception to be caught by the user application or by the Java virtual machine.

While our example uses straight-line code for simplicity, our algorithm for compensation code generation works correctly in the presence of arbitrary control

```
foo() {
    try {
      BODY_A; // optimized code of foo from the optimized graph
    } catch (... e) {

      // handler prologue
      all flags = false;
      if (e was thrown by PEI1) {
         flag_p1 = true;   branch to LABEL_1;
      } else if (e was thrown by PEI5) {
         flag_p5 = true;   branch to LABEL_5;
      } else if (e was thrown by PEI7) {
         flag_p7 = true;   branch to LABEL_7;
      }

      // Compensation code from (B) + (C).
      // Note that code for PEI5 has been code-motioned before PEI1.
      . . .
      LABEL_5: // code immediately after PEI5 monitor in optimized code
      . . .
      LABEL_1: // code immediately after PEI1 monitor in optimized code
      . . .
      if (flag_p1) throw exception(PEI1);
      . . .
      if (flag_p5) throw exception(PEI5);
      . . .

    }
}
```

Fig. 5. A high-level view of the generated code for method `foo` in Fig. 1

flow, including loops. Furthermore, the technique described above can be applied to any region of the method, including a `try` block.

5 Empirical Measurements

This section describes empirical evidence of the effectiveness of some of the techniques described in the paper. It includes (1) evidence that PEIs occur often in practice, particularly those related to runtime exceptions; (2) measurements of the effectiveness of static and dynamic analyses for detecting methods with desirable exception handler properties;[8] and (3) experimental results that demonstrate the potential for performance improvement from our techniques to

[8] We report the effectiveness of our algorithm for overcoming program-state dependences. Our techniques for overcoming the exception-sequence dependences among PEIs are *always* applicable to any code with PEIs.

eliminate exception-related dependences. The static analysis (Section 6) is implemented in an extended version of Jax [25], an application extractor for Java. Liveness information for exception handlers is currently implemented as a conservative version of the analysis described in Section 3.1.

The dynamic analysis is implemented in the Jalapeño Java virtual machine [2] by instrumenting each method using the Jalapeño optimizing compiler [6] to dynamically propagate exception handler information as described in Section 3.2. This implementation does not currently perform liveness analysis of exception handlers, and treats all exception handlers, except for the default `uncaughtException` method, as nontrivial. Neither analysis performs the "kill" computation described in Section 3.2.

Because the Jalapeño optimizing compiler does not yet contain sophisticated code transformations, performance improvements from eliminating exception dependences allowed by our techniques are not yet realized in the system. To illustrate that this is not a shortcoming of our techniques, we hand-optimized two applications at the Java source level, applying transformations that are enabled because of the dependences eliminated by our techniques. The transformations we applied, loop tiling and outer loop unrolling (also known as loop unroll-and-jam) [26,23], are well-known compiler transformations that are expected to be included in the Jalapeño optimizing compiler in the future. Both the original and optimized versions of the applications were executed using Jalapeño.

Table 1 presents some characteristics of the benchmark suite, which includes 13 programs from the SPEC JVM98 suite and other sources. The second column of Table 1 presents the number of statically reachable methods in the original application (including libraries) as determined by Jax's static type analysis. The next column reports the number of HIR instructions for those reachable methods. The next column report the percentage of HIR instructions that can throw a predefined runtime exception, such as `ArithmeticException`, `NullPointerException`, `IndexOutOfBoundsException`. The last column reports this percentage for all PEIs. These results suggest that PEIs are quite frequent — on an average 42.5% of HIR instructions are PEIs, which confirms the importance of overcoming exception-sequence dependences among PEIs. Furthermore, a large percentage of PEIs (on average 99.5%) throw only runtime exceptions predefined by the Java language.[9]

Table 2 presents the effectiveness of the static and dynamic analyses for the techniques described in Section 3. The first two columns after the benchmark name report the results of the static analysis. The next three columns report the results of the dynamic analysis. The first static analysis column reports the percentage of methods that (1) neither contain an exception handler nor have an enclosing exception handler (EEH) in the program, or (2) all such handlers are *trivial*, in that they do not contribute to the liveness set. The next column reports the same metric as the previous column, except it checks only for the

[9] None of benchmarks override the default implementation of `uncaughtException` mentioned in Section 2.

Table 1. Benchmark characteristics sorted by number of HIR instructions. HIR, the high-level intermediate representation of the Jalapeño optimizing compiler, is converted (with optimization) to LIR, which is closer to machine code.[10]

		HIR	Pct of PEIs	
Benchmark	Methods	Instructions	Runtime Only	Total
Cholesky Factorization	24	330	31.2%	32.1%
Matrix Multiply	21	381	35.2%	36.0%
201.compress	140	2,654	37.3%	38.4%
209.db	157	3,069	44.5%	45.7%
227.mtrt	212	4,621	51.6%	51.9%
toba	167	6,422	55.2%	55.6%
pBOB 1.1	190	6,682	45.0%	45.7%
JavaLex	201	8,752	41.4%	41.7%
222.mpegaudio	341	10,316	42.8%	43.2%
228.jack	415	10,868	50.6%	51.3%
202.jess	586	11,228	50.5%	51.4%
JavaParser	654	12,242	20.3%	20.8%
213.javac	1,155	23,742	44.9%	45.4%
Average			42.3%	42.5%

Table 2. Applicability results, "RT" – Runtime, "EEH" – Enclosing Exception Handler. Our instrumentation failed to produce dynamic measurements for `228.jack`.

	Static Analysis		Dynamic Analysis		
	% Methods with		% of Executed Methods with		
	No EEH	No RT EEH	No EEH	No RT EEH	No RT EEH and not Sync Region
Cholesky Factorization	100.0%	100.0%	100.0%	100.0%	100.0%
Matrix Multiply	100.0%	100.0%	100.0%	100.0%	100.0%
201.compress	52.9%	65.7%	99.996%	100.0%	99.998%
209.db	29.9%	69.4%	2.428%	99.968%	72.847%
227.mtrt	67.0%	93.4%	49.218%	96.024%	96.020%
toba	24.6%	88.0%	3.005%	99.995%	96.735%
pBOB 1.1	76.3%	99.5%	6.479%	10.957%	7.933%
JavaLex	1.2%	100.0%	1.238%	100.0%	96.687%
222.mpegaudio	21.1%	25.2%	0.012%	0.034%	0.023%
228.jack	12.3%	80.5%	—	—	—
202.jess	12.3%	69.6%	99.243%	99.996%	99.768%
JavaParser	4.6%	99.8%	0.000%	100.0%	100.0%
213.javac	5.1%	6.3%	0.037%	0.064%	0.042%
Average	39.0%	76.7%	30.135%	67.253%	64.171%

presence of nontrivial *runtime* exception handlers, thereby potentially increasing the percentage of methods.

[10] The number of HIR instructions of a program is slightly smaller than that of LIR instructions of the program. For example, the total numbers of HIR and LIR in-

The static results show that, except for a few benchmarks, the majority of methods either contain an exception handler or are enclosed in a nontrivial exception handler (Static No EEH column). However, as shown by the next (Static No RT EEH) column, in 11 of 13 benchmarks, over 65% of methods neither contain, nor are enclosed in, a nontrivial *runtime* exception handler.[11]

The metrics used for dynamic analysis are the same as those for static analysis, except that the analysis is performed at runtime for each method *activation*. In 9 of 12 benchmarks, over 96% of method activations are not enclosed in a runtime exception handler inside the program, and do not themselves contain a runtime exception handler. Hence, these methods can be optimized quite aggressively, particularly with further application of techniques presented in Section 4. The decrease in effectiveness for the `pBOB` benchmark is mostly due to the conservative liveness analysis used in our current implementation, given the manner in which Java bytecode generators implement synchronized statements. Namely, the compiler translates synchronized statements into a `try-catch` block, where the `catch` block unlocks the locked object and rethrows the exception. Although the static analysis implementation determines that such blocks are trivial, the current dynamic analysis implementation does not have this feature.

The extension described in Section 3.3, as shown in the last column under dynamic analysis, finds a very high percentage of the "optimizable" methods (those without enclosing or local runtime exception handlers, from the second column) as being unsynchronized themselves and appearing in unsynchronized regions. For example, the percentage of methods in `JavaLex` that are not enclosed by a runtime exception handler decreased only marginally from 100.0% to 96.687% when methods appearing in synchronized regions are excluded. With the expected extensions to the Java memory model [22,17] (or by simply not honoring those features of the current model that apply to programs with data races, as all commercial JVMs seem to do [22,17]),[12] these methods could be optimized completely ignoring *all* runtime exception-related dependences.

Table 3 provides an indication of how the techniques presented in this work can affect execution performance for `Matrix Multiply` and `Cholesky Factorization`. It reports the execution time in seconds for the original application and a hand-optimized version that applied well-known transformations, such as loop tiling and outer loop unrolling [26,23]. These transformations are enabled because of the dependences eliminated by our techniques. The results show speedups of 6.88 and 3.34 for `Matrix Multiply` and `Cholesky Factorization`, respectively. Although such speedups may not be realized on larger applications, these results do suggest that promising performance gains can be obtained once

structions of toba are about 23K and 30K, respectively. We have found no significant difference in the number of PEIs between HIR and LIR.

[11] The static and dynamic analysis of the SPEC JVM98 programs begins at the application's main, not the common harness. The harness contains a catch block that catches the `Throwable` exception to allow for robust functionality.

[12] All JVMs tested by Pugh [22,17] were found to break the Java memory model.

Table 3. Performance improvement by eliminating exception dependences and performing hand-transformations

Benchmark	Unoptimized Time (sec)	Optimized Time (sec)	Speedup
Matrix Multiply	115.5	16.8	6.88
Cholesky Factorization	58.8	17.6	3.34

exception-related dependences are eliminated using the techniques described in this work.

6 Related Work

Previous work on speculative code motion for superscalar and VLIW processors [8,24,19,13] has some similarities with our work, in that it involves aggressive code motion and recovery from exceptions thrown by speculative instructions. Broadly, our work differs in at least two ways. First, it does not require any hardware support, while these approaches rely on special hardware to support silent exceptions or to store the results of speculative instructions. Second, (since we are looking at a different problem, that of handling precise exceptions in Java) our work is unique in addressing the problem of reducing the program state that must be preserved at a possible exception point.

The problem of debugging optimized programs also has some similarities with optimizing programs in the presence of exceptions, in that the notion of preserving the *currency* of variables at a breakpoint [15] is similar to that of preserving program state at exception points. Holzle et al. [16] restrict optimizations across *interrupt points* and use dynamic deoptimization of code to support debugging. Other researchers have proposed various approaches to detect variables whose value may not be current at a breakpoint due to program optimizations [11,1,12], and possibly recovering the original value of variables that are not current [12]. In contrast, our work must ensure that the value of each relevant variable is "current" at an exception point and focus on determining the minimal set of such relevant variables. Furthermore, we deal with the problem of ensuring the correct ordering of exceptions (without adversely affecting program optimizations), which does not occur in the context of debugging optimized programs.

Hennessy [15] describes program optimizations in the presence of exception handling, where he essentially deals with the altered control flow due to an exception being thrown. His work does not deal with precise exceptions.

Le [18] describes a runtime binary translator that supports reordering of instructions for architectures that support precise exception semantics. However, this work requires the generation of checkpointing code, which contributes to the overhead of executing extra instructions even when no exception is thrown. Therefore, the benefits of increased flexibility in instruction scheduling have to be weighed against the overhead of checkpointing, which can potentially be high.

Moreira et al. [20] describe, for numeric computations with simple array subscripts, a program transformation to create *safe* regions in which no (out-of-bounds array index or null-pointer) exceptions may take place. They show that the loop-reordering program transformations can contribute up to a factor of 18 improvement in performance on an IBM POWER2 workstation, over and above the improvement from eliminating the exception checks in safe regions. Our work handles the more commonly occurring case where it is not possible to create exception-free regions, for example, if there is memory being allocated for a new variable in the region.

Arnold et al. [3] describe the impact of Java runtime exceptions on a VLIW and single-issue architecture using multiple instruction scheduling techniques. Their simulation results show that exception overhead can be substantially reduced using advanced scheduling techniques, most notably on the VLIW architecture. The smallest overhead was achieved when using known hardware mechanisms for increased speculation. Their simulation results confirm the adverse impact of precise exceptions on program performance, particularly on non-VLIW machines, thus supporting the need for work like ours to enable aggressive optimizations in the presence of precise exceptions.

7 Conclusions

Precise exception support imposes constraints on optimizations such as instruction reordering. The novel framework presented in this paper enables the relaxation of these constraints. By identifying the subset of program state that needs to be preserved if an exception is thrown, the framework enables the removal of many spurious dependences between writes and potentially excepting instructions. The static and dynamic analysis algorithms we presented can be used separately or together to improve the effectiveness of the analysis. Our framework further allows aggressive optimization of the program, ignoring all dependences between potentially excepting instructions, by generating compensation code that is executed only when an exception is thrown. This code ensures that the same exception is raised as in the original unoptimized code. The algorithms are applicable to any language that constrains program transformations due to dependences involving exceptions. The preliminary implementation using a conservative version of our algorithm shows promising results: using the static analysis, in 11 out of 13 benchmarks, over 65% of methods are recognized as targets of our aggressive optimization techniques for ignoring program state dependences at potentially excepting instructions; using the dynamic analysis, over 96% of the method invocations can be aggressively optimized in 9 of those benchmarks. We have also demonstrated that significant speedups, up to a factor of 7 on small programs, can be obtained using our techniques and well-known transformations.

We expect the importance of techniques presented in this paper to grow further, as Java is used more heavily in application areas that require high performance, and also as Java compilers become more mature, and run into the

limitations imposed by precise exception semantics while applying aggressive optimizations that involve code reordering.

Acknowledgments. We thank Pat Gallop, David Grove, and Marc Snir for useful technical discussions. We thank Dave Streeter, Frank Tip, and Doug Lorch for their help with the Jax infrastructure and Igor Pechtchanski and Chandra Krintz for help with the code generation used for the dynamic analysis instrumentation. We also thank David Grove, Harini Srinivasan, Frank Tip, and Laureen Treacy for their useful feedback on earlier drafts of this work and Vivek Sarkar for his support of this work.

References

1. A.-R. A.-Tabatabai and T. Gross. Source-level debugging of scalar optimized code. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.
2. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.
3. M. Arnold, M. Hsiao, U. Kremer, and B. Ryder. Instruction scheduling in the presence of Java's runtime exceptions. In *12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.
4. B. Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.
5. J. Bodga and U. Hölzle. Removing unnecessay synchronization in Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.
6. M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.
7. 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*, August 1999.
8. P. Chang, S. Mahlke, W. Chen, N. Warter, and W.-M. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proc. 18th International Symposium on Computer Architecture*, pages 266–275, 1991.
9. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. 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*, September 1999.
10. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.
11. M. Cooperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, 1994.

12. D. Dhamdhere and K. Sankaranarayanan. Dynamic currency determination in optimized programs. *ACM Transactions on Programming Languages and Systems*, 20(6), November 1998.
13. K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proc. 24th International Symposium on Computer Architecture*, pages 26–37, June 1997.
14. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
15. J. Hennessy. Program optimization and exception handling. In *8th Annual ACM Symposium on the Principles of Programming Languages*, pages 200–206, 1981.
16. U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.
17. Java memory model mailing list. Archive at http://www.cs.umd.edu/p̃ugh/java/memoryModel/archive/.
18. B. C. Le. An out-of-order execution technique for runtime binary translators. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–158, October 1998.
19. S. Mahlke, W. Chen, R. Bringmann, R. Hank, W.-M. Hwu, B. Rau, and M. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4):376–408, November 1993.
20. J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 1999. (to appear); Also available as IBM Research Report RC 21166.
21. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
22. W. Pugh. Fixing the Java memory model. In *ACM 1999 Java Grande Conference*, pages 89–98, June 1999.
23. V. Sarkar. Automatic selection of high-order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3), May 1997.
24. M.D. Smith, M.S. Lam, and M.A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. 17th International Symposium on Computer Architecture*, pages 344–354, May 1990.
25. F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.
26. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.