Instruction Scheduling in the Presence of Java's Runtime Exceptions

Matthew Arnold Michael Hsiao [†] Ulrich Kremer Barbara G. Ryder

Department of Computer Science
Rutgers University
Piscataway NJ 08855, USA
{marnold,uli,ryder}@cs.rutgers.edu

[†]Department of Electrical and Computer Engineering
Rutgers University
Piscataway NJ 08855, USA
mhsiao@ece.rutgers.edu

Abstract

One of the challenges present to a native code Java compiler is Java's frequent use of runtime exceptions. These exceptions affect performance directly by requiring explicit checks, as well as indirectly by restricting code movement in order to satisfy Java's precise exception model. Instruction scheduling is one transformation which is restricted by runtime exceptions since it relies heavily on reordering instructions to exploit maximum hardware performance. The goal of this study was to investigate the degree to which Java's runtime exceptions hinder instruction scheduling, and to find new techniques for allowing more efficient execution of Java programs containing runtime exceptions.

1 Introduction

Java programs are currently executed by compiling Java source into bytecode and interpreting this bytecode on a Java Virtual Machine. While this framework makes Java applications portable, it comes at the cost of performance. As Java is becoming more popular as a general-purpose programming language, the demand for a high performance native code Java compiler is increasing.

One of the challenges present to a native code Java compiler is Java's frequent use of exceptions. Several Java statements implicitly throw exceptions, include array accesses, pointer uses, casts, and integer division. These implicit exceptions, or *runtime exceptions*, directly affect performance by requiring the insertion of explicit runtime checks. Performance is also affected indirectly by Java's precise exception model which restricts code movement, making many optimizations and transformations more difficult.

Instruction scheduling is one example of a compiler transformation which is affected by Java's runtime exceptions since it relies heavily on reordering instructions to exploit maximum hardware performance, particularly on advanced architectures such as superscaler and VLIW. The instruction reordering restrictions imposed by runtime exceptions can hinder the effectiveness of the instruction scheduler, and as a result, substantially reduce the amount of instruction level parallelism.

One approach to reduce the overhead introduced by runtime exceptions is to use static analysis to identify instructions which are known never to throw exceptions, allowing the runtime checks for these instructions to be safely removed. While this technique important, it may not be sufficient if many of the checks cannot be eliminated.

The goal of this study was to investigate the degree to which Java's runtime exceptions hinder instruction scheduling, and to find new techniques for allowing more efficient execution of Java programs containing runtime exceptions.

The contributions of this paper are as follows:

- We discuss the issues related to instruction scheduling in the presence of Java's runtime exceptions and discuss existing techniques which can be used to improve performance without violating the Java exception model semantics.
- We offer experimental results showing the degree to which runtime exceptions hinder performance on both modern and future architectures. We also show results confirming that techniques such as superblock scheduling can be used to substantially reduce this penalty.
- We discuss interesting properties of the Java programming language that can be used to improve performance. We also introduce modifications to existing scheduling techniques which would allow these properties to be exploited.
- We discuss different interpretations of the Java specification and discuss the effect these interpretations have on instruction scheduling.

1.1 Exceptions in Java

Java's precise exception model imposes restrictions on optimizations in the presence of exceptions. The exception model is defined in [7], section 11.3.1, as follows

> *Exceptions in Java are precise: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the Java program.*

Possible interpretations of this exception model are discussed in 6. For now, we will assume that meeting the following three restrictions is sufficient to satisfy the exception model described above.

1. Exceptions occur in the same order, regardless of what optimizations are performed.
2. An exception is thrown if and only if it is thrown in the unoptimized program.
3. When an exception occurs, the user visible state of the program is the same as it would have been if no optimizations had been performed.

The most obvious solution to maintain exception order is to add the explicit exception checks *before* any optimizations take place. As long as the compiler does not re-order the checks themselves, the exceptions will occur in the original order. The compiler must then speculate code in such a way that state is guaranteed to be correct if an exception occurs.

The addition of runtime checks increases the number of jumps and reduces the size of the basic blocks. This greatly hinders traditional instruction schedulers which schedule basic blocks individually.

2 Advanced Instruction Scheduling

The following is a review of existing instruction scheduling techniques which were designed for programs written in C. One of the primary goals of these techniques is to allow speculation of instructions while ensuring that hardware exceptions occur correctly. At this point we must make it clear that this work deals with hardware exceptions, not Java's runtime exceptions. To avoid confusion, for the rest of this paper exceptions will always be referred to as either *hardware exceptions* or *Java exceptions*. There are also two types of hardware exceptions – those which terminate the program execution, such as segmentation fault and bus errors, and those which do not, such as page faults, cache misses and TLB misses. These non-terminating exceptions will be referred to as *transparent exceptions*.

In later sections we will discuss how the techniques described below can be used to improve instruction scheduling for Java programs.

2.1 Superblock Scheduling

Superblock scheduling [9] is an extension of trace scheduling [6]. The goal of superblock scheduling is to combine sets of basic blocks which are likely to execute in sequence, thus allowing more effective instruction scheduling. Flow of control may exit the superblock early along branches which are unlikely to be taken. Side entrances into the superblock are eliminated by using tail duplication.

To ensure proper program execution, two restrictions must be enforced when moving instructions within the superblock. An instruction, J , may be moved before a branch, BR if (1) the destination of J is not used before it is redefined when BR is taken, and (2) J will not cause a hardware exception which alters the execution result of the program when BR is taken.

Restricted Percolation The scheduler enforces both restrictions (1) and (2) when using the restricted percolation model. Only those instructions which are guaranteed not to cause a hardware exception can be speculatively executed; otherwise an incorrect, or *spurious exception* could occur.

General Percolation The scheduler completely ignores restriction (2) when using the general percolation model. This can be accomplished if the architecture supports non-excepting, or silent, instructions. When a terminating hardware exception occurs for a silent instruction, the exception is simply ignored and a garbage value is written into the destination register. This ensures that spurious exceptions will not affect program execution. If the exception is not spurious (i.e., it would have appeared in the original program) it will be ignored by general percolation and program execution will continue. This may lead to incorrect results, or a later exception being raised.

The obvious drawback of general percolation is the inability to detect hardware exceptions properly, making debugging and error detection very difficult. Failure to accurately detect hardware exceptions is unacceptable in many cases, and thus general percolation is used mostly as an upper bound to compare the performance of other speculation models.

2.2 Sentinel Scheduling

Sentinel scheduling [11] attempts to provide the scheduling freedom of general percolation while always detecting a hardware exception and identifying the excepting instruction. The basic idea behind sentinel scheduling is that each *potentially excepting instruction* (PEI) has a *sentinel* which reports any exception that was caused by the PEI. The sentinel resides in the PEI's original basic block, or *home block*. If the home block is never reached then the exception is not raised, thus eliminating spurious exceptions.

If a sentinel for an excepting instruction is reached, the process of *recovery* is begins. Program execution jumps back to the speculatively executed instruction and it is re-executed non-speculatively, thus causing the exception to occur. If the exception is non-terminating then the exception is processed and program execution continues. The sequence of instructions between the PEI and the sentinel are re-executed, thus the compiler must ensure that this sequence of instruction is always re-executable.

Sentinel scheduling requires architectural support, including an extra bit on all registers to record whether a speculative instruction caused an exception, as well an extra bit in the opcode to distinguish speculative and non-speculative instructions. It is claimed [14] that the Merced architecture will provide this hardware support for speculative loads, making sentinel scheduling and general percolation possible.

One of the problems associated with sentinel scheduling is that maintaining this sequence of re-executable instructions significantly increases register pressure. Techniques such as [4][3] were presented in an attempt to solve this problem and thus decrease the performance difference between sentinel scheduling and general percolation.

3 Combining Advanced Scheduling and Java

Superblock scheduling is very effective for code consisting of small basic blocks in combination with predictable branches; this is exactly the type of code that is created when runtime exception checks are added to a Java program. We propose that the following algorithm is effective for reducing the performance penalty of Java's runtime exceptions.

Algorithm 1

1. **Remove unnecessary checks.** If it can be determined at compile time that an exception can not occur, that exception check can be removed. Removing as many checks as possible eliminates the overhead of the checks and relaxes restrictions on code movement.

2. **Make all remaining exception checks explicit.**[1] For example, the code :

   `x = A[i];` becomes:

   ```
   if (A == null) throwNullPointerException();
   if (OUT_OF_BOUND(A,i)) throwArrayIndexException();
   x = A[i];
   ```

3. **Form superblocks.** Form superblocks and schedule using general percolation as the speculation model.

Notice that general percolation is used as the speculation model. This allows instructions to be speculated without concern for hardware exceptions. The result is that hardware exceptions may go undetected, but as long as these exceptions would not have occurred in the original code, the program is guaranteed to execute properly (as if no speculation had occurred). For now it should be assumed that the original program does not raise terminating hardware exceptions and thus the use of general percolation is acceptable. We discuss this issue in detail in section 5.1.

Superblock scheduling is particularly effective for code containing exceptions because it takes advantage of the fact that exceptions rarely occur. Each exception check becomes a branch exiting the superblock. By using general percolation, the compiler is given more leeway to speculate instructions within the superblock.

As for any Java transformation, the compiler must ensure that the Java exception model is not violated. As discussed in section 1.1, the following 3 properties must be satisfied.

1. **Java exception order is preserved.** The key to maintaining exception order is the fact that each potentially excepting instruction is guarded by an explicit exception check. Instructions such as array accesses may be speculated; however, since the bounds checks are executed in the original order, Java exceptions are guaranteed to occur in the same order.

[1] Some Java compilers do not explicitly test for null pointer exceptions but instead look for the hardware exception that occurs when the null pointer is dereferenced. This technique can still be used, but requires a more intelligent scheduler. This is discussed in section 5.3.

Programs	LOC (Java/C)	Description
matrix	106/1532	Matrix multiply
euler	250/1524	Calculate Eulerian circuits in randomized graph
graycodes	121/1560	Computes all graycodes for a list of bit strings
binary	31/427	Performs 10,000 binary searches on a list of 200,000 elements
mergesort	55/766	Sorts a list of 10,000 elements using the recursive mergesort

Table 1: Program Characteristics

2. **Java exceptions occur if and only if they occur in the original program.** Again, the Java exception checks were made explicit, so assuming that the superblock scheduler maintains the semantics of the original program, Java exceptions will occur as they did in the original program.

3. **State is maintained when a Java exception occurs.** Recall restriction (1) from section 2. Even in general percolation, an instruction, J , is not speculated above a branch if J 's destination register is used before defined when the branch is taken. Thus when a Java exception occurs, all variables and registers which are used during and after the handling of the exception will be the same as if superblock scheduling had not taken place.[2]

4 Experimental Results

To measure the performance impact of Java's runtime exceptions we modified Toba [13], a Java to C converter, so that the insertion of runtime exception checks could be turned on and off. This clearly works only for programs which do not throw runtime exceptions since the exceptions will not be detected when the checks are not inserted. The resulting C programs were then compiled using the Trimaran [1] system, a framework designed for research in instruction level parallelism. The Trimaran compiler performs many optimizations and transformations, including superblock scheduling, and allows the code to be simulated on a parameterized processor architecture called HPL-PD.

The C programs produced by Toba were compiled with and without superblock scheduling, and simulated on two of the sample machines provided in the Trimaran release. Both machines were specified using MDES, Trimaran's machine description language. The first machine, a VLIW machine capable of issuing 9 instructions per cycle, contains 4 integer units, 2 floating point units, 2 memory units, and 1 branch unit. The second machine, which will be referred to as the *single-issue* machine, is capable of issuing only one instruction per cycle and has one of each functional unit mentioned above. Each machine has 64 general purpose registers.

This framework allowed us to compare the cost of Java's runtime exception checks on different architectures as well as with different scheduling techniques. The only drawback to this framework is that Toba does not perform any advanced object-oriented optimizations; once the C code is generated it may be too difficult for the Trimaran compiler to perform these optimizations since some of the high level information is lost. If we can obtain a more advanced Java compilation framework, we plan to investigate the impact that object-oriented optimizations may have on the relative cost of runtime exceptions.

The programs included in the results are described in Table 1, and were chosen for their extensive use of runtime exceptions. We hope to have larger, more realistic object-oriented programs to report at the workshop if we can get them through our experimental framework. We are currently restrained by a bug in Trimaran's register allocator and are working with NYU to fix the problem.

[2]It is not clear whether this is sufficient to satisfy the Java specification of maintaining visible state. This is discussed further in section 6.

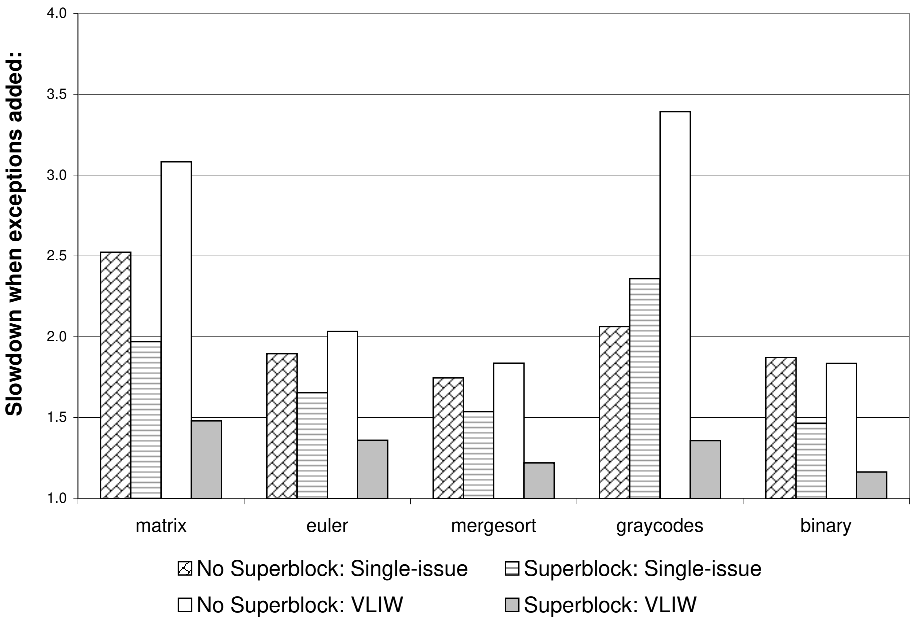

Figure 1: A comparison of the cost of Java's runtime exceptions across different scheduling techniques and architectures.

Results Figure 1 compares the performance impact of exceptions on different combinations of scheduling techniques and architectures. From left to right, the bars represent the following scheduling/architecture pairs: no superblock/single-issue, superblock/single-issue, no superblock/VLIW, superblock/VLIW, where "No superblock" means that superblock scheduling was not performed, and "Superblock" means that superblock scheduling was performed using general percolation as the speculation model.

With the scheduling technique and architecture fixed, the exception checks were inserted and removed. The y-axis represents the slowdown that occured when all checks were inserted. All measurements are normalized to 1, where 1 unit represents the execution time of the program with no checks on the particular scheduling/architecture pair. (Note that this graph does *not* show the overall performance gained by superblock scheduling. The programs ran faster with superblock scheduling regardless of whether the checks were on or off.)

There are two interesting things to notice in this graph. First, it compares the slowdown on both machines which occurs when superblock scheduling is not used (the first and third bars). For two programs, *matrix* and *graycodes*, the exceptions checks hurt performance significantly more on the VLIW machine than on the single-issue machine. This is due the the fact that the insertion of exception checks breaks up existing instruction level parallelism. However, to our surprise, for the remaining programs, the performance impact of exceptions was similar on both machines. We had expected all of the performance figures to be more like those of *matrix* and *graycodes*.

Secondly, this graph shows the slowdown which occurs on the VLIW machine, without and with superblock scheduling (third and fourth bars). The performance penalty of exceptions is drastically reduced when superblock scheduling is used. When used on single-issue machines, superblock scheduling helps reduce the penalty of exceptions, but not to the degree which it does on the VLIW machine. The code movement enabled by the superblock scheduler cannot be taken advantage of on the single-issue machine, since there is no room in the original schedule in which the checks can be hidden. When used on a VLIW architecture, superblock scheduling allows the runtime checks to be

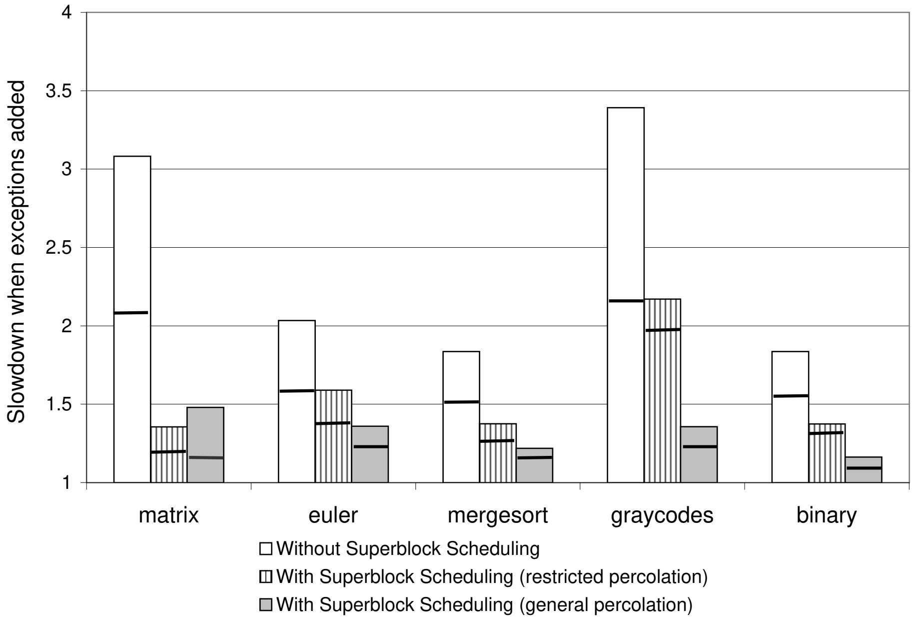

Figure 2: The cost of exceptions: with and without superblock scheduling on VLIW machine. The height of the full bar represents all exception checks turned on. The height of the line within each bar represents null pointer checks turned off, all others on.

moved into words which were not fully used, thus increasing the machine utilization and reducing the overall cost of the exception checks.

Figure 2 is similar to Figure 1 except it is used to compare the effect of restricted and general percolation as well as the impact of eliminating all null pointer checks. The graph shows the slowdown caused by runtime exceptions checks on the VLIW machine only. The bars represent, from left to right, no superblock scheduling, superblock scheduling with restricted percolation, and superblock scheduling with general percolation. The numbers are normalized to 1, where 1 unit represents the execution time of the program with no runtime exception checks using the particular scheduling method on the VLIW machine. The full height of the bar represents the slowdown which occurs when all checks are on, while the height of the line within each bar represents the slowdown which occurs when only null pointer checks are turned off (all others are on), thus representing the potential savings of implicit null pointer checks.

This graph shows that superblock scheduling in combination with general percolation is clearly most useful for reducing the cost of exceptions. Superblock scheduling with restricted percolation still reduces the penalty of the checks, but not as substantially as general percolation. For the program *matrix*, the smallest slowdown was obtained when using restricted rather than general percolation. (Again, this does not mean that *matrix* ran most quickly with restricted percolation, but rather that inserting exception checks had the smallest impact when restricted percolation was used.) A possible explanation is that the superblock scheduler creates a particularly tight schedule with general percolation, thus leaving no room to squeeze in the exception checks. Restricted percolation, on the other hand, produced a worse schedule and therefore had more room to hide the checks. This demonstrates that better instruction scheduling can potentially *increase* the cost of exceptions by creating schedules in which there is no space to hide extra instructions.

Figure 3 shows the overall speedup obtained by using superblock scheduling on the VLIW machine. The first bar shows the speedups of superblock scheduling when the program contains no

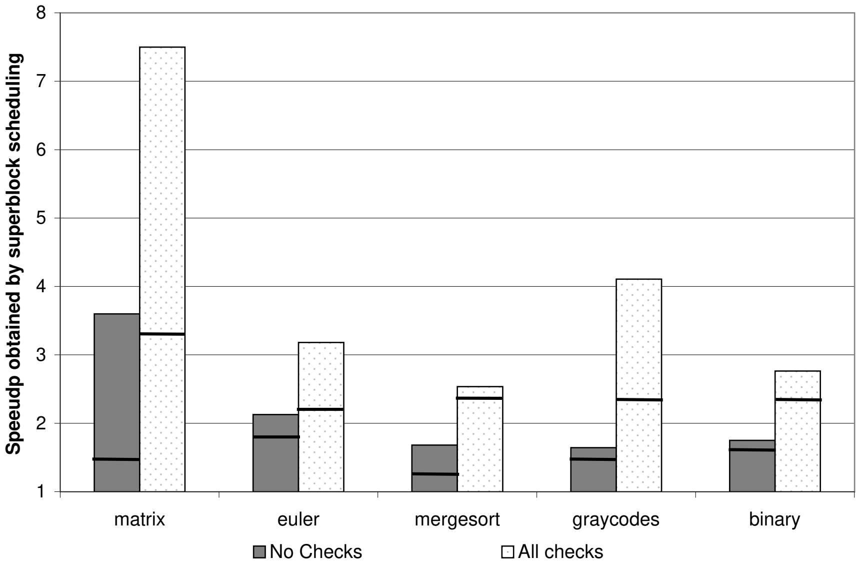

Figure 3: Speedups gained by using superblock scheduling on VLIW machine. The height of the full bar is with general percolation. The height of the line within each bar is restricted percolation.

checks. The second bar shows the speedup when the program contains all checks. Running times are normalized to 1, where 1 unit represents the running time without superblock scheduling for that particular exception scenario (checks or no checks). The height of each bar shows the speedup gained when general percolation is used as the speculation model, while the line within each bar represents the speedup obtained using restricted percolation.

The speedups obtained by superblock scheduling are substantially larger for programs containing runtime exceptions, demonstrating that superblock scheduling is even more important for Java programs than for languages without runtime exception checks. Superblock scheduling with restricted percolation also offers reasonable speedups, however they are significantly less than general percolation.

5 Specific Issues

5.1 Is General Percolation Sufficient?

The previous experiments were performed using general percolation, so proper execution is guaranteed assuming that no terminating hardware exceptions occur in the original program. However, if terminating hardware exceptions *do* occur in the original program, they may go undetected in the optimized program allowing execution to continue with unpredictable behavior, making debugging very difficult. Before we conclude that general percolation is unacceptable for Java programs, the following question must be answered: "When *do* terminating exceptions occur in Java programs?"

In fact, the Java language definition ensures that most common hardware exceptions *don't* occur. For example, consider the following:

- **Divide by zero -** Prevented by runtime check
- **Dereference null -** Prevented by runtime check

- **Dereference memory location out of segment** - Prevented by a combination of factors.
  1. Pointer use is restricted.
  2. Array bounds are explicitly checked.
  3. Memory storage management is left up to the compiler, most commonly via garbage collection.
  4. Casting is checked, preventing invalid casts.

  Using these facts it can be shown (via structural induction on program syntax) that the Java language specification prevents reference variables from referring to anything except valid objects or null. Thus, dereferencing a reference variable cannot cause a segmentation violation.

Hardware exceptions are architecture specific, so to prove that they *never* occur, every exception for the particular architecture would need to be considered, applying arguments similar to the above.

Of course, these hardware exceptions could easily occur if either the compiler or the operating system are malfunctioning. For example, the operating system could swap in the wrong page during a page fault, causing a reference to point to an invalid memory location. However, operating system error can cause improper detection of exceptions regardless of whether general percolation is being. If the operating system arbitrarily replaces values in memory, current Java compilers would fail to maintain proper exception detection as well.

The Java specification does not preclude the use of general percolation since it is architecture independent, and thus does not mention how the program should behave in the presence of a hardware exception. This supports the notion that hardware exceptions are expected *not* to occur, *making general percolation a viable option for the scheduling of Java programs.*

5.2 Is General Percolation Optimal?

To measure the performance of sentinel scheduling, previous work [11][4][3] has compared it to general percolation, treating general percolation as the upper limit on performance for sentinel scheduling. It is true that sentinel scheduling has several performance disadvantages compared to general percolation. First, sentinel scheduling requires the sequence of instructions between the speculated instruction and the sentinel to be re-executable. This eliminates speculation across irreversible instructions as well as increases register pressure by extending the live ranges of the registers used in the sequence. Sentinel scheduling also reduces performance by inserting extra instructions when no existing instruction can be used as a sentinel.

However, this assumption that general percolation *always* outperforms sentinel scheduling is not correct if the effects of transparent exceptions (cache misses, TLB misses, page fault, etc) are considered, because sentinel scheduling does have one potential performance *advantage* over general percolation. For speculated instructions, sentinel scheduling has the ability to delay the handling of transparent exceptions until the sentinel is reached, whereas general percolation must process all transparent exceptions immediately, even though the home block of the instruction may never be reached. The result is that general percolation may increase the number of transparent exceptions which occur while sentinel scheduling will not.

For example, consider the follow scenarios which are possible when an instruction is being speculated. A page fault is used as a sample transparent exception.

1. The instruction will cause a page fault regardless of whether it is speculated.
2. The instruction will *not* cause a page fault regardless of whether it is speculated.
3. The instruction will cause a page fault *only* if it is speculated.

General percolation wins in cases 1 and 2 since no spurious exceptions are introduced in either case. Sentinel scheduling is particularly poor in case 1 since it would delay the exception until the home block is reached, start recovery, then re-execute all needed instructions. However, sentinel scheduling wins in case 3, since it prevents the spurious page fault from being processed. If case 3 occurs frequently enough and the spurious exceptions are expensive to process, sentinel scheduling could outperform general percolation, leaving us with the question, "Is general percolation really optimal?"

A study combining sentinel scheduling with predicated execution for C programs [2] reported that for their benchmarks, 31% of cache misses and 13% TLB misses and page faults were spurious[3] and thus could be avoided by delaying their handling until the sentinel was reached. The study did not provide numbers showing the performance impact of avoiding these spurious exceptions nor mention any performance comparisons to general percolation.

Possible Solutions One strategy to reduce the number of spurious exceptions introduced by general percolation would be to use profiling to identify the instructions which are likely to be in case 3, and prevent these instructions from being speculated. Since it is not uncommon for a small number of instructions to cause a substantial number of the page faults [8], it may be possible to reduce the number of spurious transparent exceptions without overly restricting speculation.

Another possibility would be to encode a bit in the opcode representing whether or not a transparent exception should be processed immediately or delayed. Instructions which fit into case 2 should handle the transparent exception immediately, while those in case 3 could delay the handling. This would allow the best possible performance for all scenarios which were compile time predictable, although it is not clear whether the potential performance gains warrant such extreme measures as hardware modification.

5.3 How to Handle Null Pointer Checks?

Algorithm 1 from section 3 tests explicitly for null pointer exceptions; however, some Java compilers, such as *Harissa* [12], check for null pointer exceptions *implicitly* by looking for the hardware exception that occurs when the null pointer is dereferenced.

There are two reasons why null pointers cannot be checked implicitly in algorithm 1. First, the algorithm relies on the fact that all instructions which may throw Java exceptions are guarded by an explicit check. If these checks are removed, correct exception order is no longer guaranteed and state may not be properly maintained. These problems are not present when the checks are explicit because although the pointer uses may be speculated, the exception checks are guaranteed to remain in the same order. Maintaining state is also not a problem because care is taken when moving state-changing statements over branches.

Secondly, algorithm 1 uses general percolation which ignores terminating hardware exceptions on all speculated instructions, hence there is no way for the null pointer use to be detected. One possibility would be to use sentinel scheduling rather than general percolation in an attempt to keep track of hardware exceptions while allowing speculation. Unfortunately this also has problems because sentinel scheduling admittedly does not maintain proper exception order within a basic block, and therefore cannot guarantee that pointer uses in the same basic block will throw null pointer exceptions in the correct order.

Another solution is not to speculate pointer uses which may be null, but this restriction on code movement could hinder performance more than simply adding the runtime check.

Possible Solution Modifying sentinel scheduling to guarantee intra-basic block exception order would provide a safe solution, allowing null pointer exceptions to be checked implicitly yet enabling the scheduler to speculate potentially excepting pointer uses. However, precise exception order within a basic block is not needed for *all* instructions, but only for the potentially excepting pointer

[3] "Spurious exception" was defined as an exception by a speculative instruction whose home block was not reached.

```
a = p.x;
b = q.y;
```

(a)

```
if (!p)
  goto EXCEP;
a = p.x;
if (!q)
  goto EXCEP;
b = q.y;
```

(b)

```
b = q.y;
if (!p)
  goto EXCEP;
a = p.x;
if (!q)
  goto EXCEP;
sentinel(b)
```

(c)

```
b = q.y;
a = p.x;
sentinel(b)
```

(d)

Figure 4: Example of Algorithm 2. (a) Original program (b) Explicit checks added (c) Speculation of `b = q.y;` (d) Runtime checks removed

uses. The following algorithm demonstrates a slight modification to sentinel scheduling which would maintain the needed exception ordering, while not introducing any unnecessary overhead.

Algorithm 2

1. Add explicit checks for all statements which may throw runtime Java exceptions, including null pointer exceptions. Do not add checks for any pointer use which is known *not* to throw an exception.
2. Perform scheduling (in our case, superblock scheduling) using the following rules for speculation:
   (a) For potentially excepting pointer uses, a sentinel is required. If no instruction remaining in the pointer use's home block can be used as a sentinel, add an explicit sentinel, as is done with sentinel scheduling.
   (b) For all other speculative instructions, ignore hardware exceptions, as is done with general percolation.
3. Remove all null pointer exception checks.

An example of this algorithm is shown in figure 4. Inserting the explicit runtime checks for potentially excepting pointer uses ensures that the superblock scheduler considers the possible control flow which would take place if a Java null pointer exception occurs.[4] Once the pointer uses are speculated and the sentinels are identified, the runtime checks can be removed because if a null pointer is dereferenced, the sentinel will catch the hardware exception and raise the Java exception. Note that explicit sentinel checks (as shown in figure 4) are not necessarily needed for every speculated pointer use, but only when no instruction in the home block can be used.

Algorithm 2 allows speculation of potentially excepting instructions, guarantees exception order and visible state are maintained, and allows implicit detection of null pointer exceptions. It also raises an interesting point. Notice that pointer uses which may throw an exception require sentinels, but pointer uses which can *not* throw an exception can be speculated freely, avoiding the costs associated with the use of a sentinel. The result is that static analyses identifying non-excepting pointer uses can still be used to increase performance even when the null pointer runtime checks are implicit.

[4] This is essentially the same as modifying the superblock scheduler to treat potentially excepting pointer uses as branches.

Program 1

```
foo()  {
 int x=0; int y=0;

 try {
   x = 1;    // STMT 1
   x = A[i]; // STMT 2
   y = 3;    // STMT 3
   System.out.print(x + y);
 }
 catch(Throwable e) {
   // x and y not used
   return;
}}
```

Program 2

```
foo() {
 int a=0; int b=0;

 try{
   a = p.f;  // STMT 4
   b = q.f;  // STMT 5

   System.out.print(a + b);
 }
 catch (Throwable e) {
   // a and b not used
   return;
}}
```

Figure 5: Programs demonstrating different possible interpretations of the Java specification

6 Interpreting the Java Specification

The Java Language Specification [7] defines exceptions in Java as *precise* in an attempt to clarify exactly what transformation are legal in the presence of exceptions. Unfortunately, this definition (as quoted in section 1.1) still leaves us questioning the legality of several transformations. The English language phrases "must be prepared to hide" and "appear to have taken place" lead to the ambiguity which is made clear by the following examples.

Example 1 Assume that the program 1 in figure 5 is being compiled by a high performance, native code Java compiler with the debugging flag turned off. If `STMT 2` throws an exception, does it matter whether `x` and `y` contain the correct values during the execution of the catch clause? Since the values of `x` and `y` are not used within the exception handler and are locals so they cannot be used after the return, the user will never be able to know whether they contained the correct values.[5] `STMT 1` could then be removed since it is dead code, and `STMT 3` could be speculated before `STMT 2`.

Example 2 Consider program 2 in figure 5. Does it matter whether the order of `STMT 4` and `STMT 5` are interchanged? Suppose that both `p` and `q` are null. The program execution which follows the exception will be identical regardless of which exception is thrown. Once execution jumps to the catch clause, the only way to know which statement threw the exception is to examine the values of `a` and `b`. Since neither `a` nor `b` is referenced in (or after) the catch clause, there is no way for the user to detect that `STMT 4` and `STMT 5` were interchanged.

In the case where either `p` or `q` is null (but not both), reversing STMT 4 and STMT 5 is still not noticeable to the user. Control flow will correctly jump to the catch clause, and since `a` and `b` are not used, no incorrect state will be detectable to the user.

So the question remains: Are the transformations proposed in these examples legal according to the Java specification? If yes, then we are establishing that the specification is based on **program behavior observable by the user**.[6] This implies that Java's precise exception model is no more restrictive than the requirement traditionally imposed on optimizing compilers of preserving program semantics ; that is, transformations cannot affect program output visible to the user.

[5]If the user is using a source level debugger on the Java code, then all variables can potentially be viewed at all program points. We are presuming that this is not the case in these examples.

[6]Algorithm 1 from section 3 assumes this interpretation. It ensures that registers contain the correct value when an exception branch is taken *only* for registers which are used before defined. All dead variables may contain incorrect values.

However, if these transformations are *not* legal according to the Java specification, then we are establishing that it cannot be determined whether a compiled program meets the Java specification without **observing the internals of the machine on which the program is running**, something which directly violates one of the primary goals of the Java specification – architecture independence. A statement such as "to meet the specification, register X must contain value Y." cannot be made.

The first interpretation enables additional compiler transformations, while ensuring reasonable behavior in a debugging setting. However, both interpretations are possible given the Java specification, and neither is currently unanimously agreed upon. This issue needs to be resolved so that compiler writers know whether their compiler meets the specification.

7 Related Work

Le [10] presents a runtime binary translator which allows speculation of potentially excepting instructions. The translator produces native code on demand at runtime, processing basic blocks as they are reached and combining them into superblocks when appropriate. It takes advantage of the fact that the translator can intercept all exceptions reported by the native OS, and decides which should be made visible to the application. When an exception is raised by a speculated instruction, a recovery algorithm reverts flow of control back to a pre-computed *checkpoint* in the unspeculated code. The remainder of the basic block is then interpreted, raising any exceptions which may occur.

The main advantage of Le's work is that it does not require any hardware support, such as non-faulting loads. The disadvantage is that when an exception occurs, recovery must be used to determine whether the exception should actually be raised, or whether it is a *false exception*. In sentinel scheduling, recovery begins only if an exception is raised *and* the sentinel in the home block is reached, thus never processing false exceptions.

To avoid wasting time repeatedly recovering from false exceptions, any superblock in Le's work which raises a false exception is permanently rescheduled with no speculation. This is clearly a disadvantage compared to general percolation and sentinel scheduling.

Ebcioglu and Altman proposed an out-of-order translation technique called DAISY [5] to run RISC programs on VLIW processors. DAISY assumes the architecture modifications used by sentinel scheduling to allow speculation of potentially excepting instructions. The emphasis of this paper is how to produce VLIW code with significant levels of ILP while keeping compilation overhead to a minimum.

8 Conclusions

We have proposed using superblock scheduling in combination with general percolation to reduce the cost of Java's runtime exception checks (Algorithm 1), and have shown that doing so does not violate the Java specification. Our experimental results show that runtime exceptions hinder performance considerably on both sequential and VLIW machines. Superblock scheduling and general percolation significantly reduce this penalty on the VLIW machine by taking advantage of words which are not fully utilized to partially hide the cost of the exception checks. Superblock scheduling helped only moderately on the single-issue machine.

A modification to the sentinel scheduling algorithm was presented (Algorithm 2) which allows proper execution while implicitly checking for null pointer exceptions. The algorithm allows speculation while guaranteeing that Java exceptions are thrown in the correct order and that visible state is correct at the time of the exception.

We have also raised some questions regarding the precise exception model as defined in the Java language specification. Two possible interpretations of this exception model were presented, leaving us to question the legality of certain transformations which are particularly helpful to instruction scheduling. One interpretation needs to be chosen so that there is an agreed upon standard for compiler writers to follow.

Acknowledgments I would like to thank Ben Goldberg and Rodric Rabbah of New York University for their help facilitating the use of the Trimaran infrastructure.

References

[1] Trimaran: An infrastructure for research in instruction-level parallelism. Produced by a colaboration between Hewlet Packard, University of Illinois, and New York University. URL: http://www.trimaran.org.

[2] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 227–237, New York, June 27–July 1 1998. ACM Press.

[3] D. August, B. Deitrich, and S. Mahlke. Sentinel scheduling with recovery blocks. Computer Science Technical Report CRHC-95-05, University of Illinoiss, Urbana, IL, February 1995.

[4] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W.-M. W. Hwu. Speculative execution exception recovery using write-back suppression. In *Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO'93)*, volume 24 of *SIGMICRO Newsletter*, pages 214–224, Los Alamitos, CA, USA, December 1993. IEEE Computer Society Press.

[5] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100 In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 26–37, New YOrk, June 2–4 1997. ACM Press.

[6] J. A. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, 1981.

[7] J. Gosling, B. Joy, and G. L. Steele. *The Java^{TM} Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[8] J. L. Hennessy and D. A. Patterson. *Computer Archetecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., Santo Mateo, California, 1990. 3rd Printing.

[9] 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. *The Journal of Supercomputing*, 7:229–248, 1993.

[10] B. C. Le. An out-of-order execution technique for runtime binary translators. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 33, pages 151–158, November 1998.

[11] Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4):376–408, November 1993.

[12] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.

[13] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 41–54, Berkeley, June 16–20 1997. Usenix Association.

[14] A. Pylkin. Merced facts and speculations. URL: http://www.microprocessor.sscc.ru/Merced, 1998. Web Site.