Sentinel PRE: Hoisting beyond Exception Dependency with Dynamic Deoptimization

Rei Odaira Kei Hiraki

University of Tokyo
Hongo, Bunkyo-ku, 113-0033 Tokyo, Japan
E-mail: {ray,hiraki}@is.s.u-tokyo.ac.jp

Abstract

Many excepting instructions cannot be removed by existing Partial Redundancy Elimination (PRE) algorithms because the ordering constraints must be preserved between the excepting instructions, which we call exception dependencies. In this work, we propose Sentinel PRE, a PRE algorithm that overcomes exception dependencies and retains program semantics. Sentinel PRE first hoists excepting instructions without considering exception dependencies, and then detects exception reordering by fast analysis. If an exception occurs at a reordered instruction, it deoptimizes the code into the one before hoisting. Since we rarely encounter exceptions in real programs, the optimized code is executed in almost all cases. We implemented Sentinel PRE in a Java just-in-time compiler and conducted experiments. The results show 9.0% performance improvement in the LU program in the Java Grande Forum Benchmark Suite.

1. Introduction

Exception mechanisms have been widely used in Java and other execution environments to ensure the runtime robustness of programs. Figure 1 shows examples in Java. In (a), the object reference "a" is nullcheck'ed immediately before the value of its field is loaded. In (b), the nullcheck of "a" and the boundcheck of an index "i" are executed immediately before the load from an array element. In this manner, illegal accesses and unexpected errors are avoided.

However, from the point of view of execution speed, these check instructions, which we call Potentially Excepting Instructions (PEIs), cause performance degradation because they are converted into compare-and-branch instructios or function calls in the final code. Moreover, many PEIs in a program are redundant because a lot of memory accesses, which are accompanied with PEIs, are known to be redundant. In this work, we aim to remove such redundant PEIs by compiler optimization.

```
(a)                 (b)
nullcheck a         nullcheck a
x:=a.field1         t:=arraylength a
                    boundcheck t, i
                    x:=a[i]
```

Figure 1. Examples of exceptions in Java.

Of the redundancy elimination algorithms used in optimizing compilers, Partial Redundancy Elimination (PRE) is known to be effective [20, 13, 14], because it subsumes loop-invariant code motion, and can remove redundancy on some (but not necessarily all) incoming paths. In Fig. 2(a), a nullcheck instruction 3 and a load instruction 4 are redundant only when executed from the left. In this case, we can remove the redundancy by hoisting the computations at 3 and 4 up to 5 and 6, respectively, as shown in Fig. 2(b). As this example illustrates, it is the hoisting that allows PRE to eliminate partially redundant instructions.

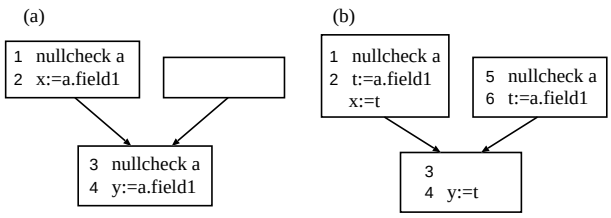

Figure 2. Example of optimization by PRE.

However, when an exception mechanism is introduced, hoisting of PEIs is restricted because optimization is not permitted to change an exception thrown at runtime. In Fig. 3(a), there is another PEI *X* inside the range of the hoisting. Assuming that "a" is null, an exception *X* occurs at Instruc-

tion 3, and the program is executed from the right path, then the actual exception to be thrown is *X*. However, if PRE has transformed the program into the one shown in Fig. 3(b), a nullcheck exception for "a" would be thrown instead of the exception *X*. Existing PRE algorithms that deal with PEI [11, 12] do not perform this kind of hoisting so as not to change the order of exceptions. We call the dependency to preserve semantics concerning exceptions *exception dependency*.

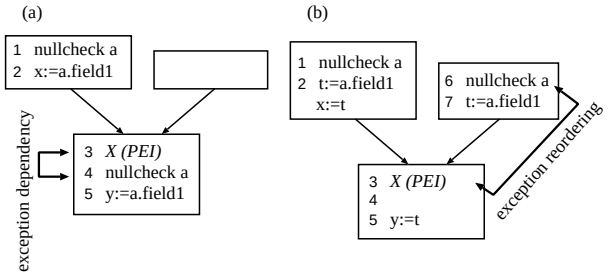

Figure 3. Example of exception dependency and exception reordering.

However, keeping exception dependencies means that many redundancies remain unremoved in a program. This is because PEIs, as described above, are accompanied with all memory accesses, and such PEIs prevent another PEI from being hoisted at many points. In addition, nullchecks and boundchecks account for nearly all of the PEIs in a program, but those checks rarely cause exceptions in programs encountered in real life. For example, the standard benchmarks we used in this work do not throw any single nullcheck or boundcheck exception. In other words, existing PRE algorithms conservatively give priority to correctness at the sacrifice of execution speed in almost all cases where exceptions do not occur. Therefore, to speed up programs using an exception mechanism, what is really needed is a new PRE algorithm that can remove redundancies beyond exception dependencies.

In this work, we propose a PRE algorithm called Sentinel PRE, which achieves redundancy elimination beyond exception dependency, and at the same time preserves program semantics. Sentinel PRE first hoists instructions without considering exception dependencies, and then detects exception reordering by analyzing the transformed program. Even if an exception occurs at one of the PEIs hoisted beyond execution dependencies, Sentinel PRE does not throw any actual exception. Instead, it dynamically patches the original place (which we call the *sentinel*) of the hoisted PEI by writing a PEI that is semantically the same as the hoisted one, and then continues with the execution. In other words, the program is deoptimized into the state before the transformation by PRE, which ensures the correct order of exceptions at runtime. Note that the hoisted PEI rarely causes an exception, as described above, so that the program remains optimized in almost all cases. Since runtime deoptimization heavily depends on the information provided by the compile-time analysis, Sentinel PRE is suited to runtime optimizing compilers.

The main contributions of our work are as follows.

- Sentinel PRE achieves partial redundancy elimination of PEIs beyond exception dependency through cooperation of compiler analysis and dynamic code patching.
- We developed a new fast algorithm that detects exception reordering to enable implementation in runtime compilers.
- We implemented Sentinel PRE in our runtime compiler, and present its effectiveness by conducting experiments using standard benchmarks.

The rest of the paper is organized as follows. Section 2 presents the overview of Sentinel PRE. Section 3 details its algorithm, and Section 4 shows the experimental results. Section 5 reviews related works, and Section 6 sets out our conclusions.

2. Overview of Sentinel PRE

Figure 4 illustrates the overview of optimization and deoptimization in Sentinel PRE.

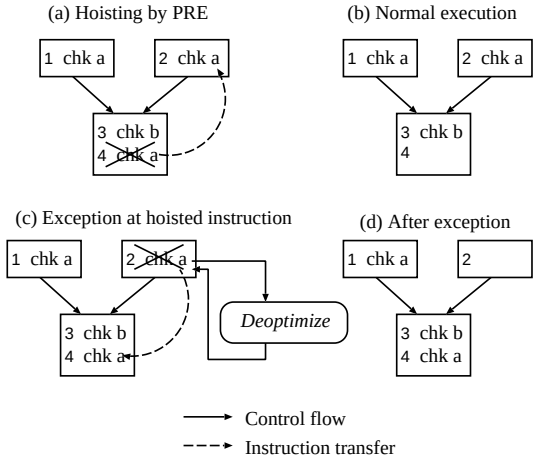

Figure 4. Overview of Sentinel PRE.

(a) Perform PRE without taking account of the exception dependency between Instructions 3 and 4. Then detect the exception reordering between Instructions 2 and 3 by program analysis.

For instructions m and n, and an expression x,

$$ANTIC_{out}(m,x) = \bigwedge_{\forall n \in Succ(m)} ANTIC_{in}(n,x)$$
$$ANTIC_{in}(n,x) = (ANTIC_{out}(n,x) \wedge TRANSP_{up}(n,x)) \vee (x \text{ is computed at } n).$$

Figure 5. Equation system for anticipatability.

(b) Execute the optimized code. It is worth noting that the same exception is to be thrown as the one thrown before the transformation by PRE, providing that Instruction 2 does not cause an exception.

(c) If it does, jump into a special exception handler, in which the PEI is put back into its sentinel (Instruction 4). Then return to immediately after the PEI.

(d) Execute the code in which the hoisting is canceled. If the exception check at Instruction 3 fails, throw an exception for "chk b." If it succeeds, then throw an actual exception for "chk a" at Instruction 4. Hereafter, when this function is reinvoked, execute this unoptimized code.

In this manner, we can realize the correct semantics in respect to exceptions.

3. Sentinel PRE Algorithm

In this section, we describe the compile-time analysis and the code generation of Sentinel PRE.

The main purpose of the compile-time analysis is to determine which of the hoisted PEIs are to be handled by a deoptimization mechanism. The reasoning is that a hoisted PEI does not necessarily cause exception reordering, and that the deoptimization should be applied to as small a set of PEIs as possible because it leaves restrictions on instruction scheduling and register allocation at sentinels as described in Section 3.3. Therefore, the compile-time algorithm of Sentinel PRE is as follows.

(1) Perform PRE beyond exception dependency.

(2) Determine the hoisted PEIs to be (possibly) deoptimized, and at the same time gather information necessary for the deoptimization.

(3) Generate optimized code with special exception handlers, which are in charge of the actual deoptimization.

In the following subsections, we discuss how the PRE framework beyond exception dependency is different from the traditional one, how our analysis algorithm detects exception reordering, why we chose code patching to implement dynamic deoptimization, and what happens if load instructions are also hoisted beyond exception dependency.

In this work, we deal with PRE of nullchecks and boundchecks in Java because they account for nearly all the PEIs in Java programs; but Sentinel PRE can be applied to any kind of PEI the PRE framework can eliminate. Since we do not handle explicitly excepting instructions such as "throw" in Java, the effectiveness of Sentinel PRE is not compromised even if explicit exceptions occur frequently.

3.1. PRE framework

PRE solves three data-flow equation systems to determine which instructions can be eliminated, which cannot be eliminated, and up to where instructions should be hoisted. Before solving the equation systems, we must compute *transparency*, which is a local predicate that determines whether or not a computation can be moved upward or downward beyond an instruction.

Hoisting is particularly related to upward transparency $TRANSP_{up}$, which is necessary for the equation system of anticipatability $ANTIC$ (Fig. 5). Existing PRE algorithms[11, 12] strictly observe exception dependencies, so that they do not allow "nullcheck a" to be hoisted across other PEIs (including a function call) and memory stores, as well as assignments to "a":

$$\begin{aligned} &TRANSP_{up}(n, \text{nullcheck a}) \Leftrightarrow \\ &\quad n \text{ does not overwrite "a"} \\ &\quad \wedge n \text{ is not a PEI (including a function call)} \\ &\quad \wedge n \text{ is not a memory store.} \end{aligned}$$

The same goes for boundchecks [1]. The reason hoisting is prohibited by a store is that the value stored can be referred to inside or after the exception handler corresponding to the PEI. On the other hand, Sentinel PRE allows PRE beyond exception dependencies; hence, hoisting is prohibited by assignments to "a" only:

$$TRANSP_{up}(n, \text{nullcheck a}) \Leftrightarrow$$

[1] For brevity, we deal with functions in which there is no exception handler block (try–catch block in Java). If there is any, block boundaries and assignments to any variables also prevent PEIs from being hoisted.

n does not overwrite "a".

3.2. Analysis algorithm

When we consider an execution path, exception reordering can occur in the following two cases (Fig. 6(1) and (2)).

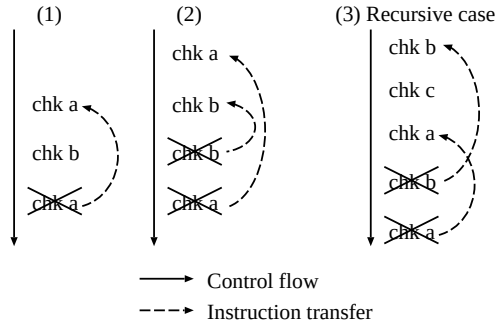

Figure 6. Hoisting beyond exception dependency.

(1) Hoisting across another PEI (or store) that is not eliminated or hoisted. This case corresponds to the right path of the program in Fig. 7(1-a). Note that its left path does not suffer from exception reordering.

(2) Hoisting across the entire hoisting range of another PEI. Figure 7(3-a) exemplifies this case. Figure 7(2-a) is similar to (3-a), but has no exception reordering in it.

The analysis algorithm in Fig. 8 detects both cases. Let $HoistedExcps$ be the set of all the hoisted PEIs. The algorithm in Lines 1 – 24 searches for the sentinels ($h.Sentinels$) for each hoisted PEI h, and at the same time identifies its hoisting region $h.Region$ and other PEIs ($h.InnerHoisted$) hoisted into $h.Region$. The principle of the algorithm is to traverse the control flow forward from h to its sentinels, using a work list W. Since the PRE algorithm we use does not hoist a PEI up to a path on which there was no PEI of the same type before the optimization, we can always find the corresponding sentinels by traversing forward from h. The algorithm stops within a finite number of steps because each instruction is traversed at most once for each h. In Fig. 7, thick lines denote forward traversals originated from hoisted PEIs.

$EquivalentInPRE$ (Line 14) is a function to determine whether n is a sentinel of h or not by consulting the information from PRE, which was performed immediately before this analysis. We cannot rely on the lexical appearances of n and h because PEIs at sentinels have already been removed by PRE. What PRE data structures are consulted

```
1  SpeculativeExcps := ∅
2  for each h ∈ HoistedExcps do
3    h.Region := ∅
4    h.InnerHoisted := ∅
5    h.Sentinels := ∅
6    W := Succ(h)
7    while W ≠ ∅ do
8      n ∈ W; W := W \ n
9      if n ∈ h.Region then
10       continue
11     else
12       h.Region ∪:= n
13     end
14     if EquivalentInPRE(n, h) then
15       h.Sentinels ∪:= n
16       continue
17     else if n ∈ HoistedExcps then
18       h.InnerHoisted ∪:= n
19     else if n is excp. insn. or store then
20       SpeculativeExcps ∪:= h
21     end
22     W ∪:= Succ(n)
23   end
24 end
25
26 for each h ∈ HoistedExcps do
27   if h ∈ SpeculativeExcps then
28     continue
29   end
30   for each e ∈ h.InnerHoisted do
31     if ∃s ∈ e.Sentinels s.t. s ∈ h.Region then
32       SpeculativeExcps ∪:= h
33     end
34   end
35 end
36
37 do
38   changed := false
39   for each h ∈ HoistedExcps do
40     if h ∈ SpeculativeExcps then
41       continue
42     end
43     for each i ∈ SpeculativeExcps do
44       if ∃s ∈ i.Sentinels s.t. s ∈ h.Region then
45         SpeculativeExcps ∪:= h
46         changed := true
47       end
48     end
49   end
50 while changed end
```

Figure 8. Algorithm to find hoisting beyond exception dependency.

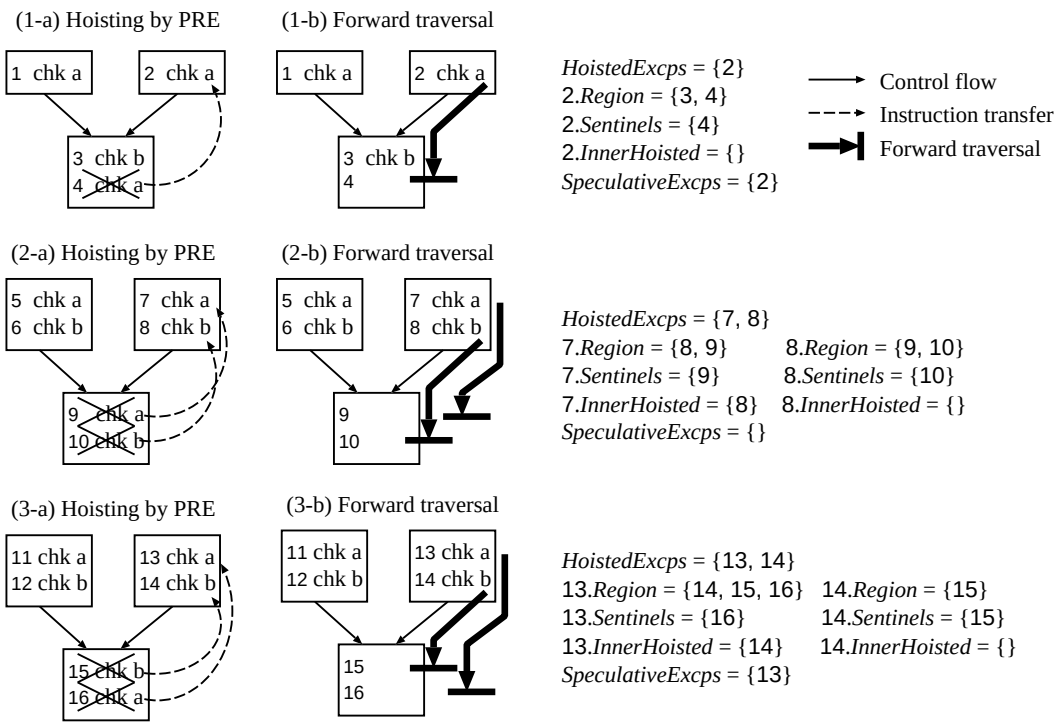

Figure 7. Examples of exception reordering detection.

in $EquivalentInPRE$ depends on the implementation of PRE. In our implementation, PRE assigns "expression numbers" to each lexical appearance of PEIs to detect redundancies. PRE replaces a redundant PEI with a "nop" instruction, but its expression number is still associated with the "nop." Sentinel PRE can discover the sentinel of a hoisted PEI h by comparing its expression number with that of n in $EquivalentInPRE(n, h)$.

If we encounter another hoisted PEI during the forward traversal, we record it in $InnerHoisted$ (Line 18). If we come across a PEI that is not eliminated or hoisted by PRE, we add h to $SpeculativeExcps$ (Line 20) because this case corresponds to the one in Fig. 6(1). In Fig. 7(1-b), we go through "chk b" (Instruction 3) during forward traversal, so that we add Instruction 2 to $SpeculativeExcps$.

The case in Fig. 6(2) is detected in Lines 26 – 35. In Fig. 7(2-b) and (3-b), "chk b" (Instructions 8 and 14) is included in $InnerHoisted$ of "chk a" (Instructions 7 and 13). If $Region$ of "chk a" contains at least one of the sentinels of "chk b," then we consider "chk a" as a PEI hoisted beyond exception dependency. Therefore, Instruction 13 is added to $SpeculativeExcps$, but Instruction 7 is not.

Additionally, sentinels that correspond to a PEI hoisted beyond exception dependency must also be treated as a source of exception dependency. For example, in Fig. 6(3), the hoisting of "chk b" leads to reordering against "chk c," but the hoisting of "chk a" does not against "chk c" nor "chk b." However, once "chk b" is deoptimized and returns to its sentinel, "chk a" in turn must be considered to be hoisted beyond the exception dependency. That is to say, with PEIs of the types shown in Fig. 6(1) and (2) being the basis, other PEIs that are hoisted across their sentinels must be recursively added to $SpeculativeExcps$. Lines 37 – 50 in Fig. 8 deal with this case.

$|SpeculativeExcps|$ increases monotonically and is less than or equal to $|HoistedExcps|$, thus the algorithm is guaranteed to stop.

3.3. Dynamic deoptimization

For PEIs in $SpeculativeExcps$, we must generate code for dynamic deoptimization.

We have three choices on how to implement the deoptimization, that is, code versioning, runtime recompilation, and runtime code patching. Code versioning generates two versions of a function; hoisted and non-hoisted versions. However, the increase in code size in the middle of runtime compilation should be avoided because it increases the compilation time of subsequent phases such as register allocation. Runtime recompilation requires on-stack replacement, by which the code of a function is switched in the middle of its execution, so that the implementation tends to

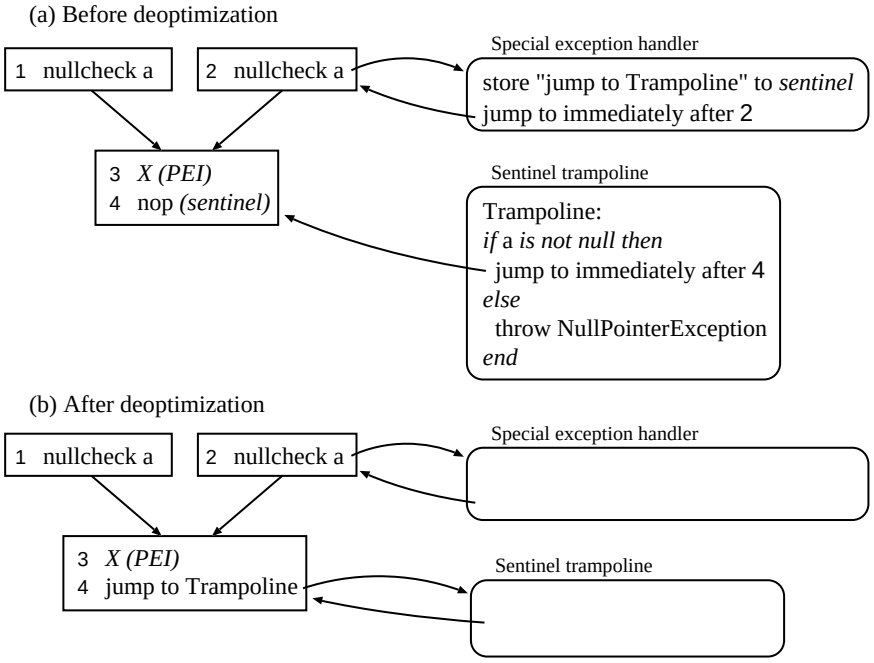

Figure 9. Example of deoptimization code.

be complicated.

We chose runtime code patching, it being the easiest one to implement. A *sentinel trampoline*, a code block semantically equal to the original PEI, is generated for each sentinel. At the time of deoptimization, a jump instruction to the trampoline is written into the sentinel. Figure 9 shows an example; assume that "nullcheck a" at Instruction 2 is in $SpeculativeExcps$ and its sentinel is Instruction 4.

We simplified our implementation in two ways.

- A "nop" instruction is inserted at a sentinel to reserve the space for code patching. It is possible to implement it without a "nop" by copying the overwritten instruction to the head of the corresponding trampoline in advance.

- A sentinel is not converted back to a "nop" after the corresponding trampoline is executed. This does not hurt performance because most Java programs do not throw any single nullcheck or boundcheck exception.

Our code patching method leaves the following restrictions on normal execution paths.

- A sentinel has the same constraints about instruction scheduling as the hoisted PEI used to have. For example, we cannot move other PEIs across the sentinel. This is because an exception might occur at the sentinel as a result of deoptimization.

- A sentinel is considered to use the same registers as the hoisted PEI. For example, in Fig. 9, "a" must be live at Instruction 4 because it is nullcheck'ed in the sentinel trampoline.

3.4. Special exception handler

It is special exception handlers that are in charge of the code patching. These handlers and sentinel trampolines are generated and appended to the tail of the function code during instruction emission, which is the very last phase of compilation. Therefore, the increase in code size due to handlers and trampolines is negligible in terms of compilation time.

In multi-threaded environments, code patching can conflict with another thread running exactly at the sentinel point. It does not matter on most architectures, where a jump instruction can be written atomically, but a three-phased technique as used in JUDO [5] is needed for use on architectures with variable-length instructions, like IA-32.

Once having performed deoptimization, a special handler need not be executed any more, so that the corresponding PEI (for example, Instruction 2 in Fig. 9) can be replaced with a "nop." However, since there is no harm in executing the handler twice or more, we leave it unchanged, giving priority to simple implementation.

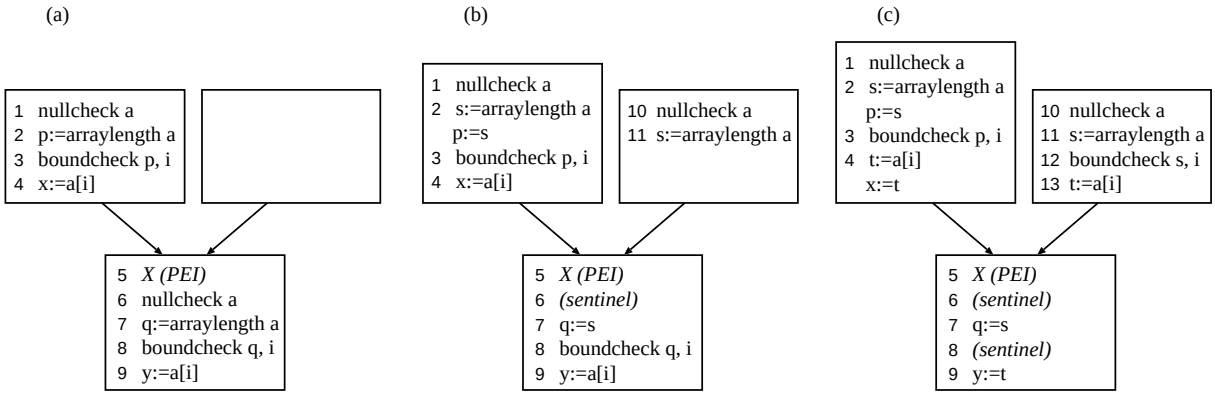

Figure 10. Elimination of a partially redundant load from an array element.

3.5. Data dependency and load instruction

So far in this section, we have dealt with just the dependencies between PEIs. However, in a program there are redundancies of load instructions protected by PEIs, and redundancies of PEIs that are data-dependent on those load instructions. We show an example of PRE for an array element load in Fig. 10. If we apply PRE just once to the original program (Fig. 10(a)), we can eliminate only the redundancies of the nullcheck and the load (arraylength), as shown in Fig. 10(b). Thus, in this case we need to iterate PRE twice [11] (Fig. 10(c)).

A load instruction comes into question when it is hoisted beyond exception dependency. In Fig. 10, load instructions at 7 and 9 are hoisted beyond Instruction 5, on which the protecting PEIs (Instructions 6 and 8) have exception dependencies. If Instruction 10 in Fig. 10(c) causes a nullcheck exception, the execution jumps to a special exception handler, and then to Instruction 11. Since "a" is still null, Instruction 11 can be an illegal memory access, depending on the memory mapping of the system. In the same way, if Instruction 12 causes a boundcheck exception, Instruction 13 can be illegal, depending on the value of "i."

It is worth noting that these illegal accesses can be ignored. For example, if we ignore the illegal accesses at 11 and 13, a nullcheck exception at 10 results in unknown values of "s" and "t." However, since either Instruction 5 or 6 must throw an exception, those unknown values cannot be referred to from outside this function; in other words, they cannot be a return value, or an argument of a store or a function call.

That is, an unknown value of a hoisted load instruction protected by a PEI h (which is also hoisted beyond exception dependency) can be referred to by

(1) instructions hoisted beyond exception dependency, or

(2) instructions below the sentinel of h.

Stores, function calls, and return instructions do not belong to the first type because PRE does not handle them. Further, execution does not reach the instructions of the second type, because an exception must be thrown not later than the sentinel. Therefore, we can ignore the illegal accesses.

We can use a non-faulting load supported by IA-64 and other architectures, or make the corresponding signal (SIGSEGV, in UNIX) handler return immediately if the faulting instruction is to be ignored. To find load instructions to be ignored, we also search for loads protected by a hoisted PEI h during the traversal in Fig. 8.

In a certain type of system where a signal is thrown by dereferencing a null pointer, we can substitute signaling at a load instruction for a nullcheck protecting it [11]. If such a load is hoisted beyond exception dependency, the corresponding signal handler must take care of runtime code patching.

4. Experimental results

We implemented Sentinel PRE in *RJJ*, a Java just-in-time (JIT) compiler that we are now developing. RJJ is invoked from Kaffe 1.0.7, a free implementation of a Java virtual machine (JVM). As a PRE algorithm, we used Partial Value Number Redundancy Elimination [21, 22], which is an extension to the PRE framework described in Section 3.1. We did not substitute signaling for nullchecks; we used compare and conditional branch instructions to implement them.

All measurements were collected on a Solaris 9 machine with 8x 900MHz UltraSPARC III processors and 40 GB main memory. We specified "-ms700m -mx700m" as options to JVM in order to avoid garbage collection. We adopted the shortest execution time of five sequential executions.

4.1. Microbenchmark

To measure how the deoptimization affects execution time, we made the microbenchmark shown in Fig. 11. We used this microbenchmark because the real benchmarks used in the next section, in which nullcheck and boundcheck exceptions never occur, did not allow us to measure the effect of the deoptimization.

```
1  class T {
2  int f1, f2;
3
4  static int meth(T obj1, T obj2, int iter) {
5    int i, ret;
6    for (i = 0, ret = 0; i < iter; i++) {
7      obj2.f2 = iter;
8      ret += obj1.f1;
9    }
10   return ret;
11 }
12
13 public static void main(String args[]) {
14   T obj1, obj2;
15   obj1 = new T(); obj1.f1 =1; obj1.f2 = 0;
16   obj2 = new T(); obj2.f1 =0; obj2.f2 = 0;
17   // 1st measurement
18   meth(obj1, obj2, 2000000000);
19   try { meth(null, obj2, 1); }
20   catch (NullPointerException ex) {}
21   // 2nd measurement
22   meth(obj1, obj2, 2000000000);
23 }
24 }
```

Figure 11. Microbenchmark.

The nullcheck and the load instruction that compose the field access at Line 8 are loop-invariant, but have exception dependency on the store instruction at Line 7. However, by using Sentinel PRE, we can hoist them out of the loop. First, at Line 18 we measured the execution time of "`meth`" in which the loop-invariants are optimized beyond the exception dependency. Then we brought about the deoptimization by the invocation at Line 19, and conducted the second measurement at Line 22.

The results are shown in the upper row of Table 1. The deoptimized method is more than two times slower than the optimized one. The performance degradation is due to the increase in code inside the loop; that is, a jump to a sentinel trampoline, a nullcheck, and another jump to immediately after a sentinel. We also show in the lower row the results of an execution in which Sentinel PRE was not used. Although the nullcheck and the load instruction remain unoptimized in the loop, it is faster than the deoptimized version.

From these results, we can confirm that for programs in which a PEI causes an exception even once, we should not use Sentinel PRE, or should convert a sentinel back to a "nop" after the corresponding trampoline is executed.

Table 1. Execution times (sec) for the microbenchmark.

	Optimized	Deoptimized
Sentinel PRE used	11.138	24.504
Sentinel PRE not used	16.707	16.706

4.2. Macrobenchmark

The real macrobenchmarks we used were 10 programs [2] from SPECjvm98 [23], and Java Grande Forum Benchmark Suite [10]. We executed each of them separately with a problem size of 100 for compress and mpegaudio, SizeB for series, lu, heapsort, fft, and sor, and SizeA for the others. These programs do not throw any single nullcheck or boundcheck exception.

In Table 2, the "Eliminated" column presents the static numbers of all PEIs eliminated by the redundancy elimination algorithm; that is, the numbers of not only the partially redundant but also the totally redundant PEIs. The "Hoisted" column shows the total numbers of hoisting, which include the numbers of hoisting beyond exception dependency shown in the "Speculative" column. We can see that approximately half of the hoisted nullchecks are beyond exception dependency. The static numbers of the hoisted boundchecks are relatively smaller than those of nullchecks except for euler, which contains many loop-invariant array accesses.

Table 3 presents the execution times and Fig. 12 shows the percentage of performance improvement achieved by Sentinel PRE. Note that we made a comparison between "PRE with Sentinel PRE" and "PRE without Sentinel PRE"; in other words, "PRE beyond exception dependency" and "PRE within exception dependency." We found that Sentinel PRE is effective for lu, heapsort, and sor. In lu and sor, some loop-invariant nullchecks, boundchecks, and subsequent load instructions, which have exception dependencies on upper PEIs, are moved out of loops by Sentinel PRE. In heapsort, a boundcheck and a subsequent load instruction, which are partially redundant, but not loop-invariant, are hoisted beyond another array load instruction inside the innermost loop. In euler, although Sentinel PRE can hoist more than 200 PEIs beyond exception dependency, it has no effect on overall performance, because far more instructions remain unremoved in the innermost loops.

[2] Since our JIT compiler currently does not support several Java bytecode instructions, it cannot compile the other programs in SPECjvm98 and Java Grande Forum Benchmark Suite.

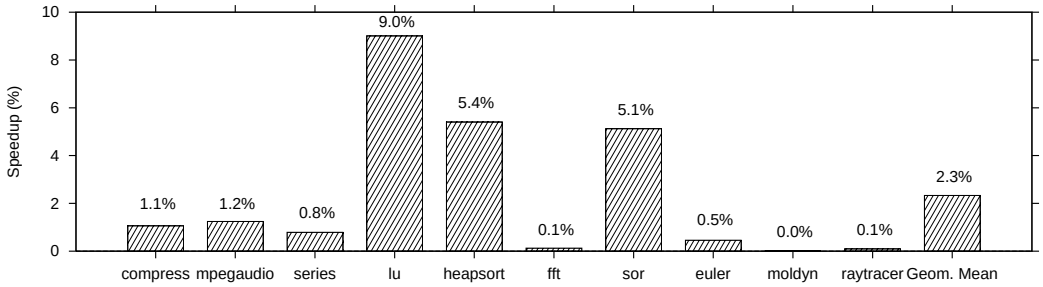

Figure 12. Improvements of the macrobenchmarks.

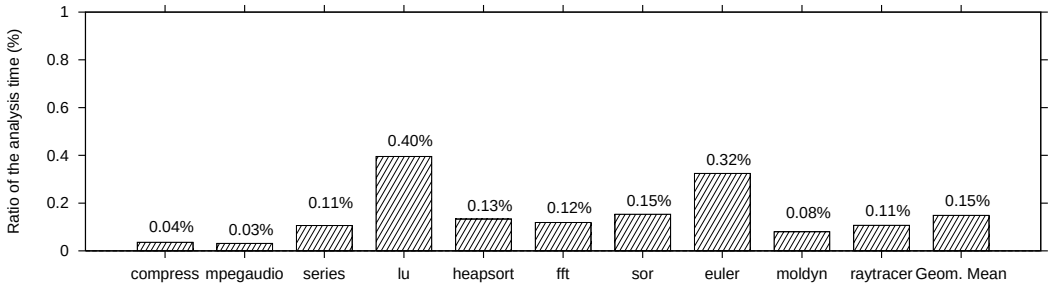

Figure 13. Ratios of the analysis time of Sentinel PRE to that of RJJ.

Table 2. Static number of eliminated / hoisted / speculatively hoisted PEIs (n.c. = nullcheck, b.c. = boundcheck).

	Eliminated		Hoisted		Speculative	
	n.c.	b.c.	n.c.	b.c.	n.c.	b.c.
compress	1788	32	42	2	22	0
mpegaudio	3252	377	106	13	40	5
series	533	31	14	3	7	1
lu	672	60	54	6	30	3
heapsort	550	36	16	3	7	1
fft	558	46	15	2	8	0
sor	533	35	19	6	9	1
euler	3039	944	181	108	132	84
moldyn	1151	30	22	2	10	0
raytracer	823	29	15	2	8	0

Figure 13 shows the ratios of the analysis time of the algorithm in Fig. 8 to that of the entire JIT compilation. Sentinel PRE does not increase the analysis time by more than 1%, which means there is no deficit using Sentinel PRE in terms of analysis time.

Table 3. Execution times (sec) for the macrobenchmarks.

	Sntl. PRE not used	Sntl. PRE used
compress	20.673	20.456
mpegaudio	20.371	20.121
series	294.406	292.101
lu	9.235	8.472
heapsort	7.318	6.943
fft	94.336	94.216
sor	16.742	15.926
euler	50.599	50.367
moldyn	18.306	18.304
raytracer	40.862	40.823

5. Related work

5.1. Speculative hoisting

Sentinel PRE is a kind of speculatively hoisting optimization, something that has long been studied in the context of speculative PRE and speculative instruction scheduling.

General percolation [8] speculatively hoists instructions

including loads beyond conditional branches in order to exploit instruction level parallelism. However, it does not ensure correct semantics when a speculatively hoisted load causes an exception.

Sentinel scheduling [4, 19] also hoists load instructions beyond control and data dependency, and at the same time retains program semantics, assuming the support of special hardware instructions. A special check instruction is inserted into the original place (sentinel) of a speculatively hoisted load. If the speculated load turns out to be invalid, the corresponding check instruction fails and execution jumps into a recovery code. Lin et al. [16, 17, 18] proposed speculative PRE for loads, using the same mechanism as sentinel scheduling.

The most significant difference between these works and Sentinel PRE is their targets: these other works deal with load instructions, while Sentinel PRE focuses on PEIs, like nullchecks. Although both may cause exceptions, we must treat them differently. First, speculative hoisting of loads needs special hardware support to efficiently detect invalid speculation. Although it can be implemented without hardware support, the overhead would spoil the performance gain. In contrast, speculative hoisting of PEIs needs no hardware support, and can be best realized by dynamic deoptimization, as we proposed in this work. Secondly, load speculation requires recovery code generation, while PEI speculation does not. When a speculative PEI causes an exception and execution reaches the corresponding sentinel, we do not have to recover from the invalid speculation, but just throw an actual exception. Thirdly, in speculative PRE for loads, a load is hoisted beyond branches and stores, which themselves are not hoisted. On the other hand, in speculative PRE for PEIs, a PEI is hoisted beyond other PEIs, which can also be hoisted, as we illustrated in Fig. 7 (2-a) and (3-a). Therefore, we need to detect exception reordering through the algorithm we developed in Fig. 8.

Gupta et al. [6] proposed an instruction scheduling algorithm beyond exception dependency without any need for special hardware support. It focuses on the reordering of PEIs within a basic block, while Sentinel PRE hoists PEIs beyond basic blocks. In addition, their method needs code duplication, which results in an increase in compilation time.

Also of note is that existing speculative PRE beyond control dependency [2, 3, 7] is an optimizing method that hoists an instruction up to a path on which there was no computation of the same type (Fig. 14(a) and (b)). Since speculative PRE does not necessarily reduce the entire count of executed instructions, the profile of execution frequency is often used to estimate the cost-benefit of speculative hoisting. The PRE framework we use forbids this kind of speculation, but if we consider a PEI to be a conditional branch as shown in Fig. 14(c) and (d), Sentinel PRE can be thought of as

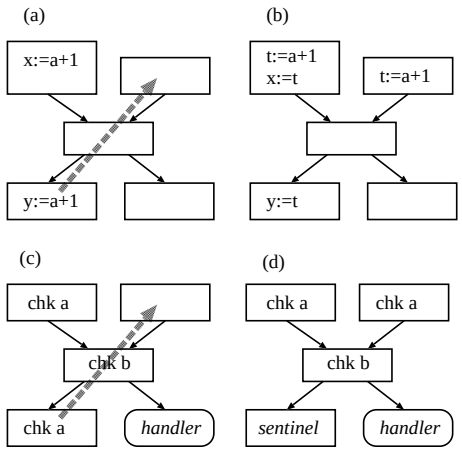

Figure 14. Speculative PRE beyond control dependency.

speculative PRE beyond control dependency. However, we can assume that exceptions rarely occur in real programs, which means that Sentinel PRE does not need profile information to estimate the cost-benefit.

5.2. Redundancy elimination for PEIs

Previous work on PRE that focuses on PEIs includes those that eliminate nullchecks [11], and boundchecks [1, 15]. However, they hoist PEIs only within exception dependency, or they do not preserve exception semantics. On the other hand, Sentinel PRE can optimize a program beyond exception dependency, and at the same time, preserve the correct semantics concerning exceptions.

Redundant boundchecks can also be eliminated speculatively by loop versioning [5], which combines a range of boundchecks inside a loop into one speculative check outside the loop. It is more powerful than Sentinel PRE in that respect, but it is specialized only to a loop structure, and suffers from the overhead of code duplication. In contrast, Sentinel PRE can be applied to any control-flow structure. Thus, they are complementary in optimizing power.

5.3. Runtime code patching and deoptimization

Runtime code patching and deoptimization are already used in inline expansion of virtual function invocation [5, 9]. In these works, caller sites are deoptimized back into virtual function invocation if dynamic class loading breaks the presupposition of the optimization. However, except for our Sentinel PRE, there are no existing works that apply runtime code patching and deoptimization to speculative hoisting.

6. Conclusions

In this work, we proposed Sentinel PRE, a PRE algorithm that overcomes exception dependency and at the same time preserve program semantics. Sentinel PRE first hoists PEIs without considering exception dependencies, and then detects exception reordering by fast analysis. When an exception occurs at one of the reordered PEIs, it deoptimizes part of the code back into the state before the hoisting. Since we rarely encounter exception in real programs, the optimized code is executed in almost all cases.

We implemented Sentinel PRE in a JIT compiler and conducted experiments. We found that Sentinel PRE achieved a maximum 9.0% speedup. Its analysis time accounts for less than 1% of the entire compilation time, which means there is no deficit using Sentinel PRE in terms of analysis time. Our results demonstrate that by using Sentinel PRE, exception dependency does not hamper aggressive redundancy elimination in programs where an exception mechanism is used.

References

[1] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 321–333. ACM Press, 2000.

[2] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 1998.

[3] Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the international symposium on Code generation and optimization*, pages 91–102. IEEE Computer Society, 2003.

[4] R. D. ching Ju, K. Nomura, U. Mahadevan, and L.-C. Wu. A unified compiler framework for control and data speculation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 157. IEEE Computer Society, 2000.

[5] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

[6] M. Gupta, J.-D. Choi, and M. Hind. Optimizing java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 422–446. Springer-Verlag, 2000.

[7] R. N. Horspool and H. C. Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *Proceedings of the 8th Israeli Conference on Computer-Based Systems and Software Engineering*, page 111. IEEE Computer Society, 1997.

[8] W.-M. 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. *J. Supercomput.*, 7(1-2):229–248, 1993.

[9] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310. ACM Press, 2000.

[10] Java Grande Benchmarking Project. Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/.

[11] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. *SIGPLAN Not.*, 35(11):139–149, 2000.

[12] M. Kawahito, H. Komatsu, and T. Nakatani. Eliminating exception checks and partial redundancies for Java just-in-time compilers, 2000. IBM Research Report RT0350.

[13] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *ACM SIGPLAN Notices*, 27(7):224–234, 1992.

[14] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1117–1155, 1994.

[15] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices*, 30(6):270–278, 1995.

[16] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew. Speculative register promotion using advanced load address table (alat). In *Proceedings of the international symposium on Code generation and optimization*, pages 125–134. IEEE Computer Society, 2003.

[17] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 289–299. ACM Press, 2003.

[18] J. Lin, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, and T.-F. Ngai. A compiler framework for recovery code generation in general speculative optimizations. In *Proceedings of the Parallel Architecture and Compilation Techniques, 13th International Conference on (PACT'04)*, pages 17–28. IEEE Computer Society, 2004.

[19] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Trans. Comput. Syst.*, 11(4):376–408, 1993.

[20] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[21] R. Odaira and K. Hiraki. Partial value number redundancy elimination. Technical report, Dept. of CS, Univ. of Tokyo, 2004. TR04-01.

[22] R. Odaira and K. Hiraki. Partial value number redundancy elimination. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing. (LCPC '04)*, 2004. To appear.

[23] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/.