Effective Null Pointer Check Elimination Utilizing Hardware Trap

Motohiro Kawahito Hideaki Komatsu Toshio Nakatani
IBM Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan
{ jl25131, komatsu, nakatani }@jp.ibm.com

ABSTRACT
We present a new algorithm for eliminating null pointer checks from programs written in Java™. Our new algorithm is split into two phases. In the first phase, it moves null checks backward, and it is iterated for a few times with other optimizations to eliminate redundant null checks and maximize the effectiveness of other optimizations. In the second phase, it moves null checks forward and converts many null checks to hardware traps in order to minimize the execution cost of the remaining null checks. As a result, it eliminates many null checks effectively and exploits the maximum use of hardware traps. This algorithm has been implemented in the IBM cross-platform Java Just-in-Time (JIT) compiler. Our experimental results show that our approach improves performance by up to 71% for jBYTEmark and up to 10% for SPECjvm98 over the previously known best algorithm. They also show that it increases JIT compilation time by only 2.3%. Although we implemented our algorithm for Java, it is also applicable for other languages requiring null checking.

1. INTRODUCTION
The Java language [4] has a powerful exception-handling mechanism, which is useful for error handling, program control, and safety preservation. However, because of the support for precise exceptions in Java, any instruction potentially throwing an exception inhibits a compiler's ability to optimize the program. In general, a program written in Java tends to have many such instructions, which become barriers to code motion and thus significantly reduce the scope of optimizations.

For example, null pointer checks are required for every instance variable access, method call, and array access. In fact, these operations are quite common in typical Java programs. In practice, the implementation of null checking can take advantage of hardware traps [2, 13, 15]. For typical operating systems, accessing the zero address (page) will throw an exception to the application, and thus no explicit instruction has to be generated to check the null pointer.

Even with such an implementation, null check elimination is still important for two reasons. The first is that null checks become barriers to optimizations, even with the hardware trap support, and thus significantly reduce the scope of optimizations. The second is that all the null checks cannot necessarily rely on the hardware support mechanism.

For example, some operating systems do not generate an interrupt when the offset of the address is larger than a certain size. As another example, AIX does not generate an interrupt for reading from the first page at the address zero. A more subtle example is that when devirtualization [1, 3, 6, 7, 13] is applied, an explicit null check instruction must be generated for an object access to the method table, since this object access will be eliminated by transforming the dynamic (virtual) call to a static (non-virtual) call or inlining its method body. Here, the execution cost of the generated null check instruction may not be negligible since the inlined method body can often be just a few instructions.

Previous null check elimination techniques, such as the one by forward data-flow analysis [14], have two drawbacks. The first is that they cannot remove loop invariant null checks from the loop and thus significantly limit the effectiveness of other optimizations. The second is that they do not exploit the maximum use of the hardware trap to minimize the cost of null checks.

Our null check elimination algorithm solves these two issues using a two-phase approach. In both phases, a partial redundancy elimination algorithm (PRE) [8, 9, 11] is enhanced to reduce the number of null checks. For each instruction that can potentially throw a null pointer exception, we split it into a null check and the original operation to allow us to move the null check separately from its original location.

In the first phase, as an architecture independent optimization, null checks are moved backward in the control flow graph to the earliest points they can reach without violating the precise exception support in Java. That is, our optimization prevents null checks from moving across side-effecting instructions, which can potentially throw exceptions other than a null pointer exception or perform memory write operations. This phase is iterated for a few times with other optimizations to eliminate redundant null checks and maximize the effectiveness of other optimizations.

In the second phase, as an architecture dependent optimization, null checks are moved forward in the control flow graph to the latest points they can reach without violating the precise exception support in Java. Then they are converted to hardware traps wherever possible in order to minimize the execution cost of null checking. Finally, those remaining null checks, which are not converted to hardware traps, are eliminated if they are redundant.

As a result, our algorithm eliminates many null checks effectively and exploits the maximum use of hardware traps. To the best of our knowledge, this is the first algorithm to optimize null checking in two phases and to provide such powerful null check elimination.

We implemented our new algorithm in the IBM cross-platform Java Just-in-Time (JIT) compiler. Our JIT compiler supports Intel IA32, PowerPC, and S/390, and our algorithm is applicable for all

Copyright © A.C.M. 2000 1-58113-317-0/00/0011...$5.00
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.
ASPLOS 2000 Cambridge, MA Nov. 12-15 , 2000

these architectures. We conducted experiments by running jBYTEmark and SPECjvm98 benchmark programs on both a Pentium III 600 MHz machine (with Windows NT 4.0) and a PowerPC 604e 332 MHz machine (with AIX 4.3.3). Our preliminary performance results show significant improvements over previous approaches.

1.1 Our Contributions

- **A New Null Check Elimination Algorithm:** Our two-phase null check optimization algorithm can maximize the effectiveness of other compiler optimizations unlike previously known algorithms, and yet it can take full advantage of the hardware trap mechanism. Although we implemented our algorithm for Java, it is also applicable for other languages requiring null checking.
- **Empirical Evaluation:** Our experimental results show that our approach improves performance by up to 71% for jBYTEmark and up to 10% for SPECjvm98 over the previously known best algorithm. They also shows that our approach increases JIT compilation time by only 2.3%.

The rest of the paper is organized as follows. Section 2 summarizes previous work. Section 3 gives an overview of our approach. Section 4 presents the details of our algorithm. Section 5 shows the performance results obtained in our experiments. Section 6 offers some concluding remarks.

2. PREVIOUS WORK

2.1 Implementation of Null Check

Some JIT compilers, such as the Jalapeño Dynamic Optimizing Compiler [2] from the IBM T.J. Watson Research Center, LaTTe JIT compiler [15], and our JIT compiler [6, 13], utilize hardware traps and the associated OS support functions for null check implementation. Jalapeño's object layout is designed in order to trap in hardware for memory reads and writes, although the target architecture (AIX on PowerPC) can trap only for writes to memory in the first page. Jalapeño accesses the object's slots (including object headers) using a negative offset from an object reference (pointer). The designers rely on the fact that reading from the last page triggers this hardware trap. LaTTe (whose target architecture is SPARC) relies on all memory reads and writes causing hardware traps. They assume that all null checks cause hardware traps.

However, such an assumption can not be used in applying some code transformations, such as method inlining by devirtualization. This is because the object's slots are not always accessed in the invoked method. **Figure 1** shows such an example. If the null check instruction of *'a'* in Figure 1(2) is not generated (by relying on the hardware trap) and *'i'* has a negative value, no slot of *'a'* is accessed. In this case, if *'a'* in (2) is a null pointer, no exception occurs and the program continues to be executed. This violates

1) Before inlining

```
int func(int s1) {
  if (s1 < 0){
    return s1;
  } else {
    return this.field1;
  }
}

result = a.func( i );
```

2) After inlining

```
nullcheck a; // check instruction
             // must be generated
if (i < 0){
  result = i;
} else {
  result = a.field1;
}
```

(***Italic*** denotes the actual exception sites)

Figure 1. Nullcheck with method inlining

the Java language specification. In this case, null checking code must be generated explicitly. Such null checks with method inlining appear frequently in typical Java programs, and thus their overhead is not negligible.

2.2 Elimination by Forward Analysis

Previous JIT compilers, such as the Jalapeño compiler [14] and the previous version of our JIT compiler [6, 13], eliminate null pointer checks by using forward data-flow analysis. This algorithm eliminates redundant null checks that appear again in the control flow graph. However, there are two drawbacks to this approach:

- Forward data-flow analysis cannot move loop invariant null checks out of the loop. For example, when the first object access lies inside of the loop, its null check must remain in the loop body. Such a null check becomes a barrier and thus significantly limits the effectiveness of other optimizations.
- This elimination algorithm does not exploit the maximum use of the hardware trap to minimize the cost of null checks.

3. OVERVIEW OF OUR APPROACH

This section describes how we solve the issues described in Section 2. Section 3.1 describes the high-level flow diagram of our null check optimization. Section 3.2 describes an overview of the architecture independent optimization phase, which solves the first issue in Section 2.2. Section 3.3 gives an overview of the architecture dependent optimization phase, which solves the issue in Section 2.1 and the second issue in Section 2.2.

3.1 High-level View of Our Algorithm

We begin by explaining the high-level flow diagram of our null check optimization using **Figure 2**. Our algorithm is split into two phases; the first is an architecture independent optimization (1), and the second is an architecture dependent optimization (5). The architecture independent optimization (1) moves null checks backward and eliminates redundant null checks. This optimization increases opportunities for both array bound check optimization (2) and scalar replacement (3), and these optimizations also increase opportunities for the null check optimization in phase 1 (1). Therefore a few iterations (4) can achieve more optimizations. In previous approaches, scalar replacement (3) is iterated in itself. In our approach, however, phase 1 (1) is iterated with other optimizations ((2) and (3)), providing a powerful optimization effect.

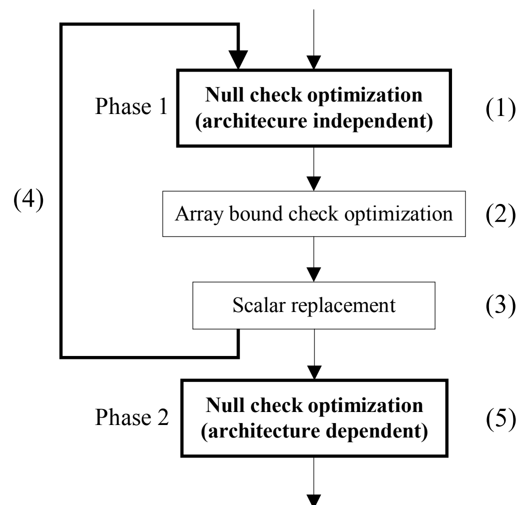


Figure 2. High-level flow diagram of null check optimization

The architecture dependent optimization (5) moves null checks forward and minimizes the execution cost of null checking by utilizing the hardware trap.

3.2 Architecture Independent Optimization

We enhance partial redundancy elimination (PRE) [8, 9, 11] to eliminate null checks and move them out of loops. **Figure 3** shows an example of partially redundant null checks. In Figure 3 (1), the null check located at the merge point cannot be eliminated without code motion, because the right path does not include any null check. Therefore, a null check will be executed twice along the left path. The architecture independent optimization first computes the movable areas of null checks backward by taking into account side-effecting instructions and finds insertion points, which are at the end of basic blocks in our algorithm (2). It next computes the non-null areas of the target variables assuming null checks are inserted at the insertion points, and then eliminates redundant null checks within the non-null areas (3). Finally, it inserts null checks at the insertion points (4). As a result, a null check will be executed only once along each path in this example.

Figure 4 shows an example where the architecture independent null check optimization and scalar replacement assist each other.

1) Before optimization

Nullcheck a
t += a.field1
a = func( )
Nullcheck a
t += a.field2

2) Compute insertion points of null checks

Nullcheck a
t += a.field1
a = func( )
Movable area of **Nullcheck a** backward (●:Insertion Points)
Nullcheck a
t += a.field2

3) Eliminate null checks

Nullcheck a
t += a.field1
a = func( )
eliminated
Nullcheck a
t += a.field2
Non-null area of 'a'

4) Optimization result

Nullcheck a
t += a.field1
a = func( )
Nullcheck a
Insert **Nullcheck a** at insertion point
t += a.field2

Figure 3. Architecture independent optimization

The first "nullcheck a" in (2) cannot be eliminated from the loop by the previous approach [14] using forward data-flow analysis, because the outer path does not include any null check. However, our approach moves "nullcheck a" out of the loop in (3). The result of (4) cannot be achieved without the null check optimization in (3), since a null check becomes a barrier to moving the related memory accesses backward. The result of (5) also cannot be achieved without the scalar replacement in (4), since a write to a variable becomes a barrier to moving its null check. Therefore combining these optimizations in this manner is particularly effective for loop invariant code motion.

3.3 Architecture Dependent Optimization

This phase converts as many null checks as possible to those utilizing hardware traps, while we generate an explicit null check instruction for each of the remaining null checks. A distinguished feature of our approach is to use a PRE algorithm in the opposite direction to maximize the number of null checks detected by hardware traps.

3.3.1 Implicit Null Check and Explicit Null Check

We first explain our implementation of the null checks. We define two kinds of null checks:

- ***Implicit Null Check***, which does not need to generate an actual null check instruction, but relies on the hardware trap.
- ***Explicit Null Check***, which needs to generate an actual null check instruction.

To implement null checking, we use implicit null checks wherever possible. However, in some cases we have to use explicit null checks in order to meet the Java language specification. For example, when an instruction requiring null checking does not cause a hardware trap, its null check must be an explicit null check.

For **Figure 5** (1), as the offset of the memory access for the BigOffset is not within the (protected) trap area, its null check must be an explicit null check. Fortunately, this case is rare in Java. For any array access, the array length is required for bounds checking and its offset is typically zero from the top of the object, though this depends on the implementation of the object layout. For a field access, its offset is usually within the trap area. In extreme cases the offset can be larger than the trap area since Java Virtual Machine Specification [10] states that it can be as large as 512 KB (which is 65534 * 8 = 524272). Our JIT compiler for Windows has implemented both explicit and implicit null checks.

For Figure 5 (2), if the operating system detects hardware traps

1) Original program

```
do {
  i += a.b.c + a.b.d;
} while(some cond);
```

2) Intermediate representation

```
do {
  nullcheck a
  T1 = a.b
  nullcheck T1
  T2 = T1.c

  nullcheck a
  T3 = a.b
  nullcheck T3
  T4 = T3.d

  i += T2 + T4
}while(some cond)
```

3) After null check optimization

```
nullcheck a
do {
  T1 = a.b
  nullcheck T1
  T2 = T1.c

  T3 = a.b
  nullcheck T3
  T4 = T3.d

  i += T2 + T4
} while(some cond)
```

4) After scalar replacement

```
nullcheck a
Tb = a.b
do {
  nullcheck Tb
  T2 = Tb.c

  nullcheck Tb
  T4 = Tb.d

  i += T2 + T4
} while(some cond)
```

5) After null check optimization

```
nullcheck a
Tb = a.b
nullcheck Tb
do {
  T2 = Tb.c

  T4 = Tb.d

  i += T2 + T4
} while(some cond)
```

6) After scalar replacement

```
nullcheck a
Tb = a.b
nullcheck Tb
Tc = Tb.c
Td = Tb.d
do {

  i += Tc + Td
} while(some cond)
```

(assumption : ‘a’ and ‘i’ are local variables)

Figure 4. Architecture independent null check optimization and scalar replacement

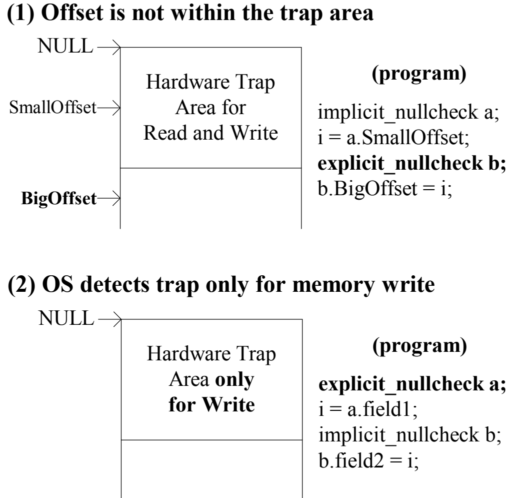
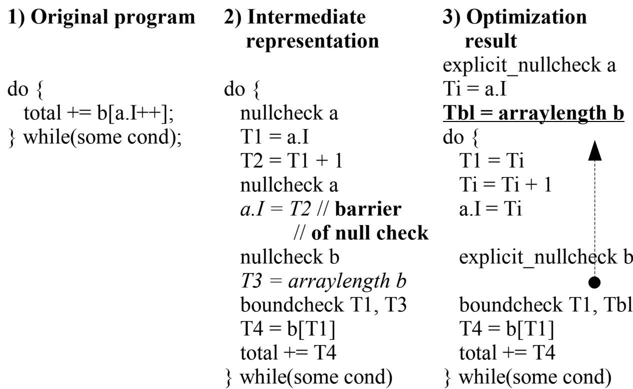


Figure 5. Examples where an explicit null check is required

only for memory writes to the (protected) trap area as in the case of AIX, the null check for a memory read must be an explicit null check. However, such an operating system has an advantage in that the compiler can apply speculation to memory reads. If a memory read with a null pointer is guaranteed not to cause a hardware trap, it can be moved across its null check speculatively. Furthermore, it can be moved out of the loop as a loop invariant instruction.

Figure 6 shows an example of such a case. In (2), "nullcheck b" cannot be moved across the memory write, "a.I = T2." However, if the operating system does not generate a trap for the memory read, "arraylength b" can be moved up across "nullcheck b." Finally "arraylength b" can be moved out of the loop as shown in (3).

Our JIT compiler for AIX could use implicit null checks for the memory writes, but we have not implemented it yet. At present, we skip the architecture dependent optimization for AIX. In the code generation phase, we generate a conditional trap instruction (which requires only one cycle if it is not taken) for each explicit nullcheck corresponding to a memory read or a write. Instead, we apply speculation for memory reads in the scalar replacement phase. Our JIT compiler for Windows cannot use speculation because a memory read causes a hardware trap on this platform.



(assumption : 'a', 'b', and 'total' are local variables)

Figure 6. An example of speculation

3.3.2 *Architecture Dependent Optimization*

We explain this algorithm by using **Figure 7**, which is essentially the same as the method inlining example in Figure 1(2). A null check in Figure 7(1) must be implemented as an explicit null check even if the previous approach in Section 2.1 is applied, because no slot of object *'a'* is accessed along the right-hand path. We use the PRE algorithm in the opposite direction in order to minimize the execution cost of null checks by utilizing hardware traps.

At first, the architecture dependent optimization treats all null checks as explicit null checks in the input code, and it computes the movable areas of null checks in a forward direction by taking into account side-effecting instructions and finds insertion points (1). Second, it inserts either an explicit null check or an implicit null check depending on the next instruction following each insertion point. If the next instruction is known to cause a hardware trap by accessing an slot of object referred by the null check, then an implicit null check (which does not generate actual code) is inserted (2). The instruction following an implicit null check should be the actual exception site, and therefore we must mark such an instruction as an exception site. This is to prevent instruction-level optimizations (such as code scheduling) from applying code motion illegally beyond the exception site in the later phase. Third, it computes the substitutable areas of null checks to eliminate them wherever possible (3). As a result, we can reduce the execution cost along the left path (4), and thus we can optimize the explicit null checks generated by method inlining.

As another example, if this optimization is applied to the result of Figure 4 (6), all the null checks are replaced by implicit null checks.

4. OUTLINE OF OUR ALGORITHM

This section describes the outline of our algorithm for null check optimization. Section 4.1 describes two transformations for the architecture independent optimization. Section 4.2 describes two transformations for the architecture dependent optimization.

4.1 Architecture Independent Optimization

4.1.1 *Algorithm for Null Check Insertion*

The goal of this stage is to compute the earliest points null checks

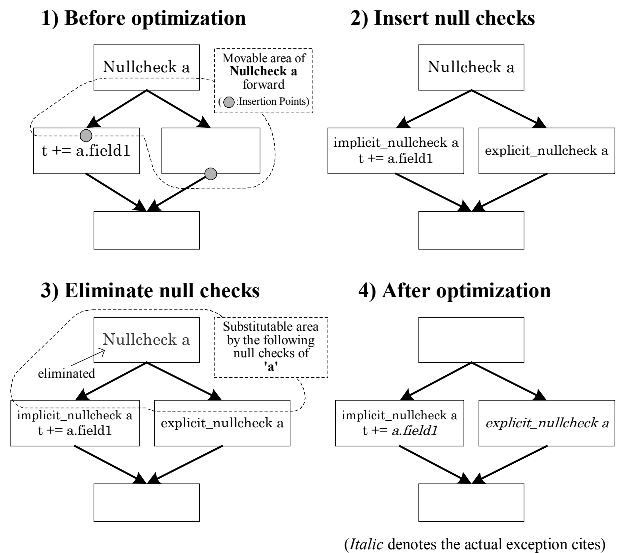


Figure 7. Architecture dependent optimization

can reach when they are moved backward in the control flow graph. First, we compute the Out_bwd(n), which is the set of null checks that can be moved up to the exit point of basic block *n* from succeeding basic blocks, by solving the following backward data-flow equations:

$$\text{Out_bwd}(n) = \bigcap_{m\, =\, \text{Succ}(n)} (\, \text{In_bwd}(m) - \text{Edge_try}(m, n)\,)$$

$$\text{In_bwd}(n) = (\text{Out_bwd}(n) - \text{Kill_bwd}(n)) \cup \text{Gen_bwd}(n)$$

Here, Gen_bwd(n), Kill_bwd(n), and Edge_try(m, n) are defined as follows:

Gen_bwd(n) : The set of null checks that are located in basic block *n* and can be moved up to the entry point of basic block *n*.

Kill_bwd(n) : The set of null checks that cannot be moved up beyond basic block *n* in the backward direction because one of the following conditions holds:

- There is an instruction that overwrites a variable targeted by the null check.
- There is a side-effecting instruction, which can potentially throw an exception other than a null pointer exception or perform a memory write (including a local variable write in a try region).

Edge_try(m, n) : The set of null checks that cannot be moved on the edge from basic block *m* to basic block *n*, because basic block *m* and basic block *n* are not in the same try region.

Finally, Earliest(n) is computed by the following equation:

Earliest(n) : The set of null checks that can reach the exit of the basic block *n* but cannot beyond basic block *n* when they are moved backward in the control flow graph.

$$\text{Earliest}(n) = \Big(\bigcup_{m\, =\, \text{Pred}(n)} \overline{\text{Out_bwd}(m)} \Big) \cap \text{Out_bwd}(n)$$

We call Earliest(n) as the *insertion points* for basic block *n*, since they are the set of null checks which will be inserted at the exit of basic block *n*. They are not yet inserted at this time, because some of them might be eliminated by the following optimization phase described in Section 4.1.2.

4.1.2 Algorithm for Null Check Elimination

The goal of this stage is to eliminate null checks that are known to be non-null. First, we compute the In_fwd(n) by solving the following forward data-flow equations:

In_fwd(n), Out_fwd(n) : The set of null checks that are known to be non-null at the entry/exit point of basic block *n*.

$$\text{In_fwd}(n) = \bigcap_{m\, =\, \text{Pred}(n)} (\text{Out_fwd}(m) \cup \text{Earliest}(m) \cup \text{Edge}(m, n)\,)$$

$$\text{Out_fwd}(n) = (\text{In_fwd}(n) - \text{Kill_fwd}(n)) \cup \text{Gen_fwd}(n)$$

Here, Gen_fwd(n), Kill_fwd(n), and Edge(m, n) are defined as follows:

Gen_fwd(n) : The set of null checks that are known to be non-null at the exit of basic block *n*, because there is a null check to the same target variable, or there is a *new* instruction that creates a new object pointed by the same target variable.

Kill_fwd(n) : The set of null checks whose target variables are overwritten in basic block *n*.

Edge(m, n) : The set of null checks that are on the edge from basic block *m* to basic block *n* and that are known to be non-null because one of the following conditions holds:

- There is an instruction such as *ifnull*, *ifnonnull*, or *instanceof-if*<*cond*>, which tells if the target variable is null or not.
- Basic block *m* is the entry of an instance method and basic block *n* is its first basic block, in which the target variable points to the "this" object.

Now we are ready to eliminate null checks from each basic block. At any point of basic block *n*, the set of null checks that are known to be non-null can be computed from In_fwd(n). For any null check ***C*** in basic block *n*, ***C*** is eliminated if it is part of the computed set at the point immediately preceding ***C*** and thus it is known to be non-null.

Next, we eliminate unnecessary null checks from Earliest(n) by computing the following equation, and insert those null checks, which appear in the computed Earliest(n), at the exit of basic block *n*.

$$\text{Earliest}(n) = \text{Earliest}(n) - \text{Out_fwd}(n)$$

4.2 Architecture Dependent Optimization

4.2.1 Algorithm for Null Check Insertion

The goal of this stage is to compute the latest points null checks can reach when they are moved forward in the control flow graph. First, we compute the In_fwd(n), which is the set of null checks that can be moved down to the entry point of basic block *n* from preceding basic blocks, by solving the following forward data-flow equations:

$$\text{In_fwd}(n) = \bigcap_{m\, =\, \text{Pred}(n)} (\, \text{Out_fwd}(m) - \text{Edge_try}(m, n)\,)$$

$$\text{Out_fwd}(n) = (\text{In_fwd}(n) - \text{Kill}(n)) \cup \text{Gen_fwd}(n)$$

Here, Gen_fwd(n) and Kill(n) are defined as follows:

Gen_fwd(n) : The set of null checks that are located in basic block *n* and can be moved down to the exit point of basic block *n*.

Kill(n) : The set of null checks that cannot be moved down beyond the basic block *n* in the forward direction because one of the following conditions holds:

- There is an instruction that overwrites a variable targeted by the null check.
- There is an instruction that accesses a slot of the object referred by the null check and causes the hardware trap if its target variable is a null pointer.
- There is a side-effecting instruction, which can potentially throw an exception other than a null pointer exception or perform a memory write (including a local variable write in a try region).

Finally, Latest(n) is computed by the following equation:

Latest(n) : The set of null checks that can reach the entry of the basic block *n* but cannot beyond basic block *n* when they are moved forward in the control flow graph.

$$\text{Latest}(n) = \Big(\bigcup_{m\, =\, \text{Succ}(n)} \overline{\text{In_fwd}(m)} \Big) \cap \text{In_fwd}(n)$$

The *insertion points* inside basic block *n* are determined from Latest(n) by means of the following algorithm.

```
Inner = Latest(n)
for (each I from the first to the last instruction in basic block n) {
   if (I is a null check ) {
      C = null check by instruction I;
      Inner = Inner ∪ C;
   } else {
      if (I accesses object's slots &&
         I will cause a hardware trap if object reference is null){
         C = null check for instruction I;
         if (C ∈ Inner) {
            Insert implicit null check for C before I; // this step is optional.
            Mark I as exception site;
            Inner = Inner - C;
         }
      }
      if (I might cause other kinds of exceptions ||
         I might write to memory ||
         (I writes to local variable && basic block n is in a try region)) {
         for (each C ∈ Inner) {
            Insert explicit null check for C before I;
         }
         Inner = φ;
      } else if (I overwrites a local variable that has object) {
         C = null check of the local variable;
         if (C ∈ Inner) {
            Insert explicit null check for C before I;
            Inner = Inner - C;
         }
      }
   }
}
for (each C ∈ Inner) {
   Insert explicit null check for C at the exit of basic block n;
}
```

4.2.2 Algorithm for Explicit Null Check Elimination

The goal of this stage is to eliminate explicit null checks that are known to be substitutable. That is, they are redundant because they can be checked later in the control graph. First, we compute the Out_bwd(n), which is the set of null checks that are known to be substitutable at the exit point of basic block *n*, by solving the following backward data-flow equations:

$$Out_bwd(n) = \bigcap_{m = Succ(n)} (In_bwd(m) - Edge_try(m, n))$$

$$In_bwd(n) = (Out_bwd(n) - Kill(n)) \cup Gen_bwd(n)$$

Here, Gen_bwd(n) and Kill(n) are defined as follows:

Gen_bwd(n): The set of null checks that are known to be substitutable at the entry point of basic block *n*, because there is a null check to the same target variable or there is an instruction accessing the object's slot pointed by the same target variable and causing a hardware trap if it is a null pointer.

Kill(n) : The set of null checks that cannot be substitutable above basic block *n*. This set is the same as Kill(n) in Section 4.2.1.

Now we are ready to eliminate null checks from each basic block. At any point of basic block *n*, the set of null checks that are known to be substitutable can be computed from Out_bwd(n). For any explicit null check ***C*** in basic block *n*, ***C*** is eliminated if it is part of the computed set at the point immediately succeeding ***C*** and thus it is known to be substitutable.

5. EXPERIMENTAL RESULTS

We chose two benchmark programs for the evaluation of our optimizations: jBYTEmark version 0.9 (from BYTE Magazine) and SPECjvm98 [12]. For SPECjvm98, the measurements were performed in the test mode (not in SPEC-compliant mode) with the count of 100 (as specified for the SPEC-compliant mode). All the experiments described in Section 5.1 through 5.3 were conducted on an IBM IntelliStation M Pro (Pentium III 600 MHz with 384 MB of RAM), Windows NT 4.0 Service Pack 5, and IBM Developer Kit for Windows, Java Technology Edition, Version 1.2.2. The experiment described in Section 5.4 was conducted on a PowerPC 604e 332 MHz with 128 MB of RAM, AIX 4.3.3.

To show our baseline performance, we disabled all the null check optimizations and always generated explicit null checks for all the required null checks (denoted as "No Null Opt. (No Hardware Trap)" in **Table 1** and **Table 2**). To compare the effectiveness of implicit null checks (that is with the hardware trap) over explicit null checks, we disabled all the null check optimizations and utilized the hardware trap (denoted as "No Null Opt. (Hardware Trap)" in Table 1 and Table 2). To compare the performance improvement of previous approach over our baseline, we implemented Whaley's algorithm [14] (denoted as "Old Null Check" in Table 1 and Table 2) for null check elimination. To compare the performance improvement of the architecture independent optimization over our baseline, we disabled the architecture dependent optimization and enabled only the architecture independent optimization (denoted as "New Null Check (Phase1 only)" in Table 1 and Table 2). To compare the performance improvement of the new algorithm over our baseline, we enabled all the null check optimizations (denoted as "New Null Check (Phase1+Phase2)" in Table 1 and Table 2).

In order to validate the competitiveness of the compilation time and overall performance of our optimizations, we also measured these benchmarks by running the HotSpot™ Server VM 2.0 beta [5] (called the HotSpot for the rest of this paper) under the same software environment.

5.1 Performance Improvement

Figure 8 shows the percentage of performance improvement over our baseline for jBYTEmark v.0.9 achieved by the new null check optimization described in Section 4. We found that the architecture independent optimization is very effective for ***Assignment***, ***Neural Net***, and ***LU Decomposition***. This is because these benchmarks use multidimensional arrays, which are optimized effectively by the iterative use of null check optimization, array bounds check optimization, and scalar replacement. As a result, some loop invariant array accesses are moved out of loops, and thereby performance is greatly improved.

Figure 9 shows the percentage of performance improvement over our baseline for SPECjvm98 achieved by the new null check optimization. We found that the architecture dependent optimization is particularly effective for ***mtrt*** after method inlining is performed. This is because ***mtrt*** has small methods (to access data in a class) which are called frequently and many explicit null checks associated with these calls can be eliminated only after they are inlined.

5.2 Performance Compared with the HotSpot

Figure 10 shows the performance comparison for jBYTEmark between our JIT compiler (with our null check optimization) and the HotSpot. Our JIT compiler shows significantly better performance for the five benchmark programs. The average relative performance of our JIT compiler is 69% better than the HotSpot.

Figure 11 shows the performance comparison for SPECjvm98 between our JIT compiler (with our null check optimization) and the HotSpot. Our JIT compiler shows slightly better performance. The average relative performance of our JIT compiler is 6% better than the HotSpot.

Table 1. Performance for jBYTEmark v.0.9 (Larger numbers are better)

(unit : index)	Numeric Sort	String Sort	Bitfield	FP Emulation	Fourier	Assignment	IDEA encryption	Huffman Compression	Neural Net	LU Decom-position
New Null Check (Phase1+Phase2)	201.96	54.41	258.86	219.64	22.75	207.41	67.46	159.33	200.50	205.90
New Null Check (Phase1 only)	202.10	54.46	258.89	219.64	22.74	181.75	67.49	158.49	200.10	203.64
Old Null Check	160.78	49.87	245.25	186.12	22.74	130.10	63.27	156.08	130.82	158.31
No Null Opt. (Hardware Trap)	157.01	49.58	245.13	170.18	22.74	125.31	63.14	151.88	130.42	119.91
No Null Opt. (No Hardware Trap)	156.94	49.08	227.85	163.87	22.68	107.87	62.99	134.40	116.81	112.57
HotSpot	207.13	44.73	234.00	206.56	8.06	114.74	25.69	145.24	88.87	106.62

Table 2. Performance for SPECjvm98 (Smaller numbers are better)

(unit : sec)	mtrt	jess	compress	db	mpegaudio	jack	javac
New Null Check (Phase1+Phase2)	6.44	7.67	17.38	24.42	11.32	9.39	14.18
New Null Check (Phase1 only)	6.89	7.71	17.45	24.43	11.33	9.45	14.31
Old Null Check	7.05	7.86	17.49	24.70	11.33	9.77	14.30
No Null Opt. (Hardware Trap)	7.09	7.95	17.55	24.71	11.39	9.80	14.33
No Null Opt. (No Hardware Trap)	7.38	8.25	18.70	25.33	12.00	10.02	15.17
HotSpot	5.73	6.53	20.13	24.61	14.78	9.25	17.50

New Null Check (Phase1+Phase2): New null check optimization. Enable both the architecture independent and dependent optimizations. It utilizes hardware traps.
New Null Check (Phase1 only): New null check optimization. Disable the architecture dependent optimization. It utilizes hardware traps.
Old Null Check Optimization: Use Whaley's algorithm[14] for null check elimination. It utilizes hardware traps.
No Null Opt. (Hardware Trap): Disable all the null check optimizations. It utilizes hardware traps.
No Null Opt. (No Hardware Trap): Disable all the null check optimizations. It does not utilize hardware traps.
HotSpot: HotSpot Server VM 2.0 beta

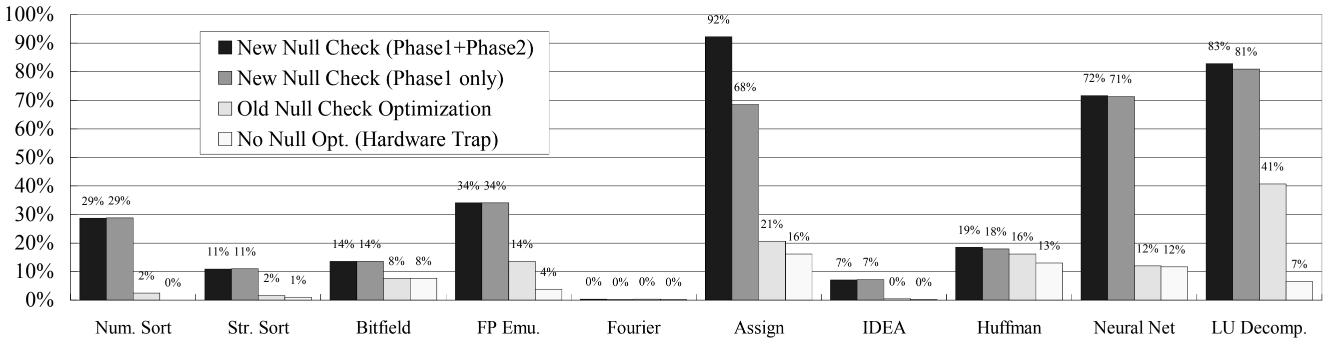


Figure 8. Improvement for jBYTEmark v.0.9

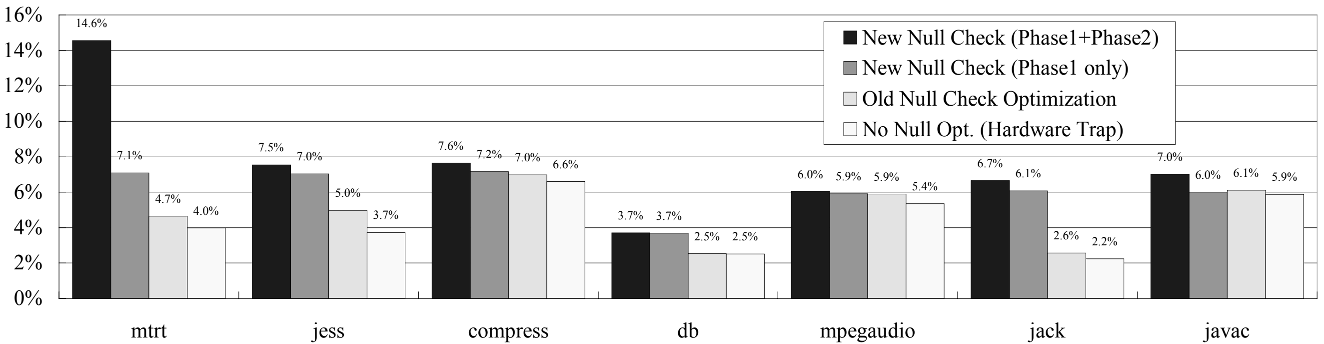


Figure 9. Improvement for SPECjvm98

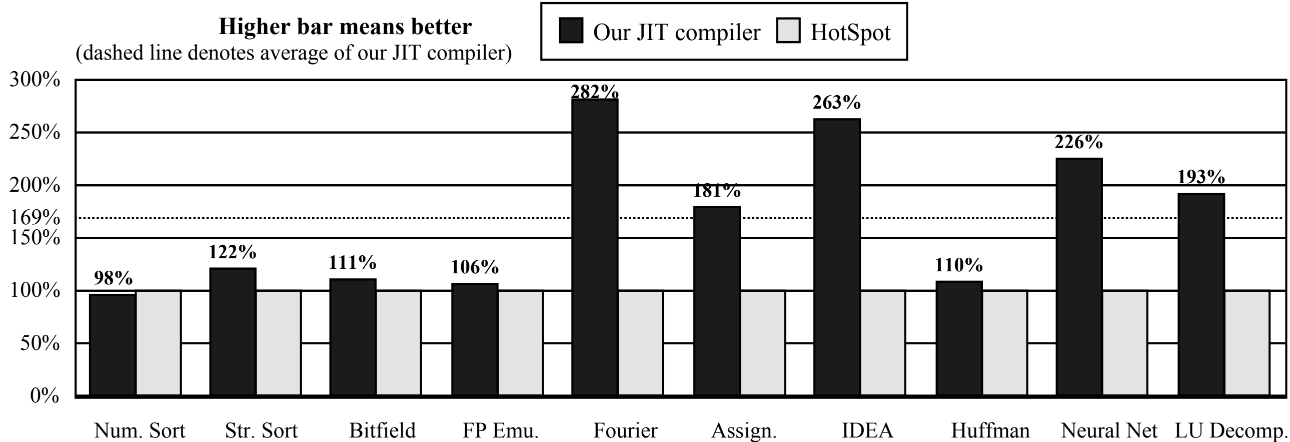


Figure 10. Performance comparison for jBYTEmark (HotSpot=100%)

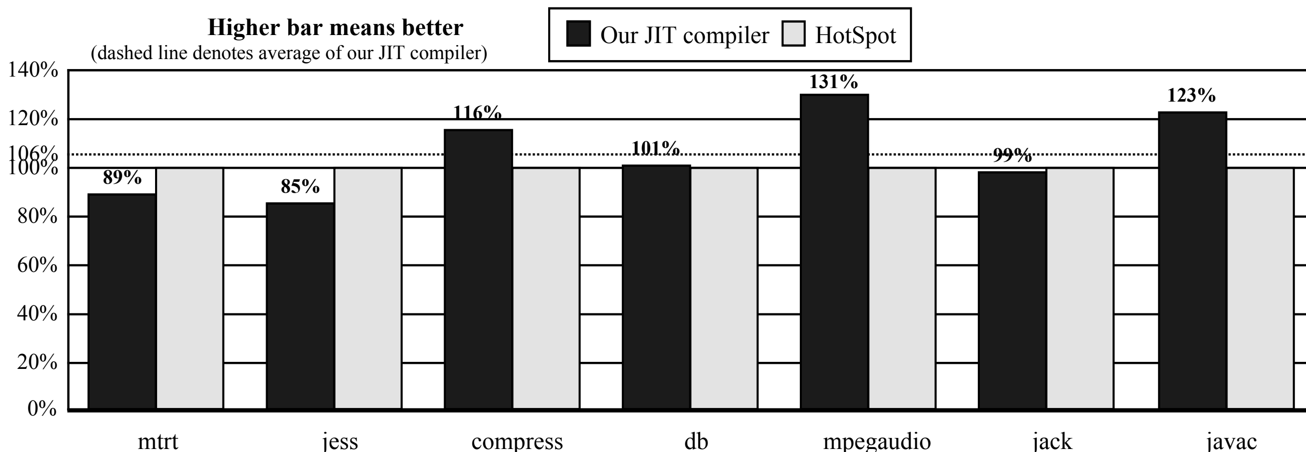


Figure 11. Performance comparison for SPECjvm98 under the test mode (HotSpot=100%)

5.3 JIT Compilation Time

In this section, we compare the JIT compilation time by our new algorithm with that by the previous approach (Whaley's algorithm). We measured the compilation time of jBYTEmark (1.35 sec), but the number of instructions executed by jBYTEmark seems to depend on the execution speed. When the execution speed is faster, jBYTEmark seems to use a larger iteration count for measurement because of accuracy. It means that the ratio of the compilation time over the total execution time of jBYTEmark is different on each environment. For instance, the ratio becomes smaller on a faster machine. In our environment, the ratio of the compilation time over the total execution time was 2.6% for the total of jBYTEmark. Because we think the ratio of the compilation time of jBYTEmark does not have much significance for the above reason, we did not put it in **Figure 12**.

We assume that the difference between the first run and the best run of SPECjvm98 is essentially due to the compilation time. In order to reduce cache and file system effects between the first run and the best run, we executed SPECjvm98 once before starting measurements. However, the assumed compilation time may still include cache and file system effects. **Table 3** shows the time for the first run, best run, and the assumed compilation time of our JIT compiler and the HotSpot. This shows that our JIT compiler spends significantly less time in compilation when compared with the HotSpot. Figure 12 shows the ratio of the compilation time over the whole execution time (that is, the time spent for the first run) of our JIT compiler. In summary, ***javac*** is the benchmark program that the JIT compiler took the longest time to compile.

We further measured the breakdown of our JIT compilation time by using a trace tool available in AIX, and computed the compilation time by taking into account platform differences. **Table 4** and **Figure 13** show the results. (We truncated the graph below 90% in Figure 13.) For ***compress***, ***db***, and ***mpegaudio***, we measured the breakdown of the total compilation time of these three benchmark programs because each of them spent little time for compilation. The new null check optimization itself takes about 3 times longer in the compilation time than the old one (Whaley's algorithm), and it also slightly increases the compilation time for the other optimizations. This is because the new null check optimization increases the opportunities for the other optimizations, such as scalar replacement, to be applied.

Table 5 shows the increase in the compilation time by our new algorithm relative to that by the old one. In summary, the new algorithm increased the total compilation time by 2.3% (on average).

Table 3. JIT compilation time of SPECjvm98 (seconds)

		mtrt	jess	compress	db	mpegaudio	jack	javac
Our JIT compiler	first run	9.47	10.37	17.43	24.62	12.56	11.95	22.33
	best run	6.44	7.67	17.38	24.42	11.32	9.39	14.18
	compilation time	3.03 (32.00%)	2.70 (26.04%)	0.05 (0.29%)	0.20 (0.81%)	1.24 (9.87%)	2.56 (21.42%)	8.15 (36.50%)
HotSpot	first run	11.50	18.06	20.75	26.80	19.23	21.88	57.38
	best run	5.73	6.53	20.13	24.61	14.78	9.25	17.50
	compilation time	5.77 (50.17%)	11.53 (63.84%)	0.62 (2.99%)	2.19 (8.17%)	4.45 (23.14%)	12.63 (57.72%)	39.88 (69.50%)

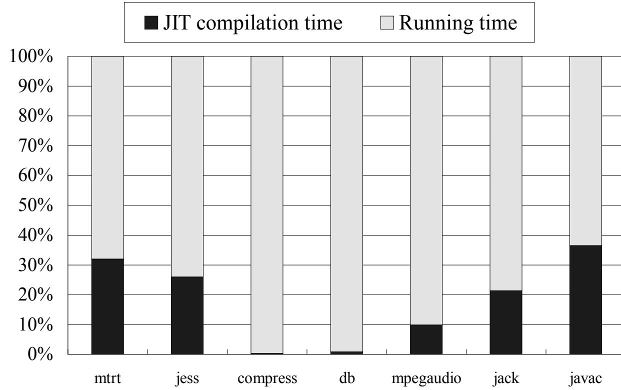


Figure 12. Ratio of our JIT compilation time (100% = time spent for the first run of each program)

Table 4. Breakdown of our JIT compilation time (seconds)

		Null check optimization	Others
mtrt	NEW	0.07 (2.31%)	2.96 (97.69%)
	OLD	0.02 (0.66%)	2.93 (96.70%)
jess	NEW	0.06 (2.22%)	2.64 (97.78%)
	OLD	0.02 (0.74%)	2.62 (97.04%)
db + compress + mpegaudio	NEW	0.035 (2.35%)	1.455 (97.65%)
	OLD	0.012 (0.81%)	1.454 (97.58%)
jack	NEW	0.06 (2.34%)	2.50 (97.66%)
	OLD	0.02 (0.78%)	2.49 (97.27%)
javac	NEW	0.17 (2.09%)	7.98 (97.91%)
	OLD	0.06 (0.74%)	7.86 (96.44%)
jBYTEmark	NEW	0.023 (1.70%)	1.327 (98.30%)
	OLD	0.008 (0.59%)	1.305 (96.67%)

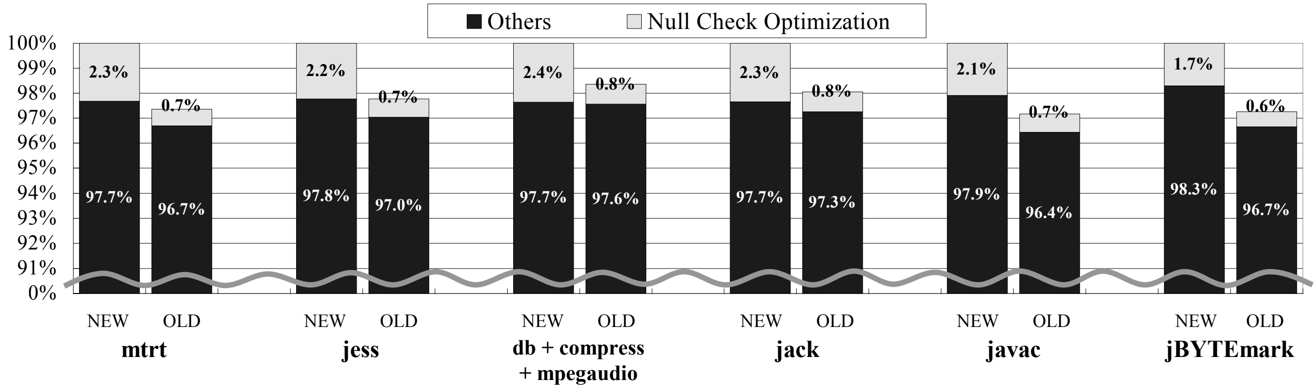


Figure 13. Breakdown of JIT compilation time (100%=new null check optimization)

Table 5. Increases in JIT compilation time in our approach

	Increase in total compilation time (second)	Increase in total compilation time (%)
mtrt	0.07	2.31%
jess	0.06	2.22%
db + compress + mpegaudio	0.02	1.61%
jack	0.05	1.95%
javac	0.23	2.82%
jBYTEmark	0.04	2.74%

5.4 Speculation versus Implicit Null Check

As we described in Section 3.3.1, we use conditional trap instructions available for the PowerPC to detect null pointers associated with memory reads and writes on AIX. For memory reads, we used speculation. To find the performance gain over the baseline (denoted as "No Null Check Optimization" in **Table 6** and **Table 7**), we disabled the whole phase of our null check optimization. To see the effectiveness of speculation (denoted as "Speculation" in Table 6 and Table 7), we disabled speculation (denoted as "No Speculation" in Table 6 and Table 7) in the scalar replacement phase. To compare the effectiveness of implicit null checks (denoted as "Illegal Implicit" in Table 6 and Table 7), we applied the architecture dependent null check optimization for Intel as the phase 2 optimization for AIX after the architecture independent optimization is applied in the phase 1. We note here that this violates the Java language specification since a NullPointerException may not be thrown correctly on AIX. This is purely for our experimental purpose.

Table 6. Performance for jBYTEmark v.0.9 on AIX (Larger numbers are better)

(unit : index)	Numeric Sort	String Sort	Bitfield	FP Emulation	Fourier	Assignment	IDEA encryption	Huffman Compression	Neural Net	LU Decomposition
Speculation	186.12	30.01	84.45	87.46	13.26	96.47	45.14	97.35	86.03	92.08
No Speculation	181.09	29.77	83.65	86.16	13.25	94.76	45.14	97.20	75.94	91.66
No Null Check Optimization	173.92	28.17	83.42	79.89	13.23	81.71	44.68	97.14	73.93	79.98
Illegal Implicit (No Speculation)	183.28	29.91	84.40	86.62	13.25	95.66	45.60	100.74	77.35	92.66

Table 7. Performance for SPECjvm98 on AIX (Smaller numbers are better)

(unit : sec)	mtrt	jess	compress	db	mpegaudio	jack	javac
Speculation	20.34	25.92	43.80	72.08	20.16	44.56	47.14
No Speculation	20.56	26.28	44.21	72.39	20.33	44.66	47.26
No Null Check Optimization	21.00	26.28	44.25	72.85	20.42	45.36	47.34
Illegal Implicit (No Speculation)	19.94	26.09	43.75	71.86	19.87	44.71	46.90

Speculation: Enable new null check optimization. Enable speculation. All null checks are explicit null checks.
No Speculation: Enable new null check optimization. Disable speculation. All null checks are explicit null checks.
No Null Check Optimization: Disable all the null check optimizations and speculation. All null checks are explicit null checks.
Illegal Implicit (No Speculation): Apply the architecture dependent optimizations for Intel, assuming memory accesses cause hardware traps. Disable speculation.

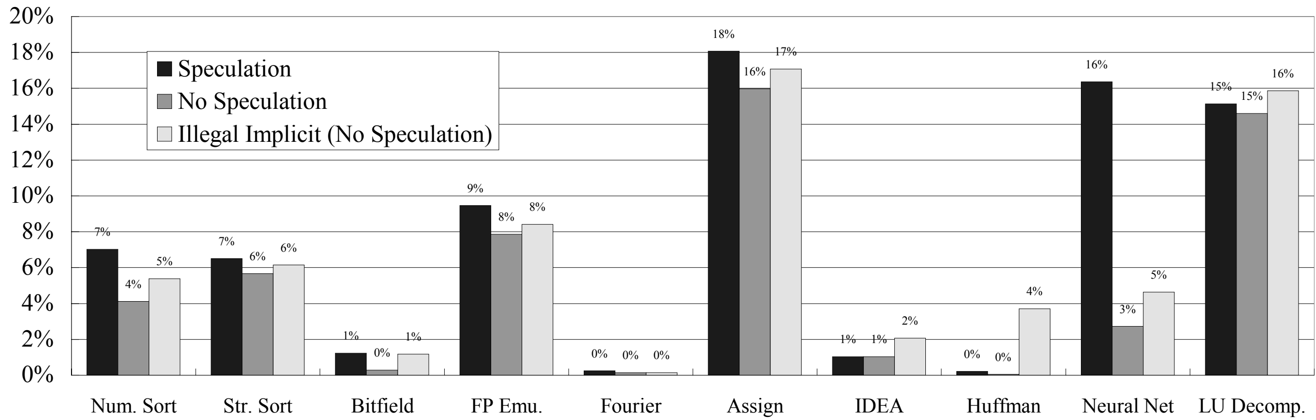


Figure 14. Improvement for jBYTEmark v.0.9 on AIX

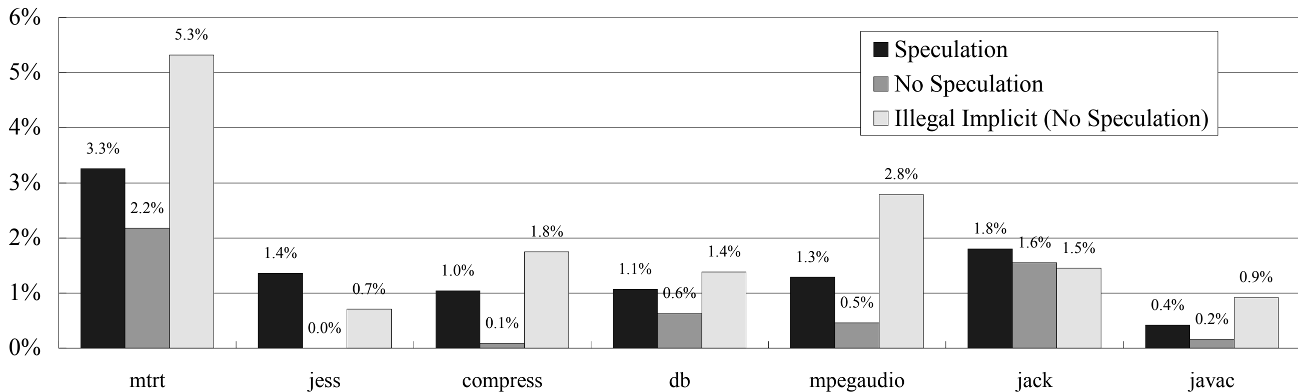


Figure 15. Improvement for SPECjvm98 on AIX

Figure 14 shows the percentage of performance improvement over our baseline (No Null Check Optimization) for jBYTEmark v.0.9. We found that speculation is very effective for ***Neural Net***. This is because four instructions were moved out of the innermost loop (across their null checks) with speculation.

Implicit null checking for ***Neural Net*** is least effective among all the benchmarks when the result is compared to that of the Intel platform. On the Intel platform, the method java.lang.Math.exp was inlined, but it was not on the PowerPC. This is because our JIT compiler for Intel converts this method to an exponential instruction. However, it does not for the PowerPC, since the PowerPC does not have such an instruction. Therefore, this method call became a barrier for scalar replacement, and the improvement from implicit null checking is particularly limited on the PowerPC relative to other benchmarks.

Figure 15 shows the percentage of performance improvement over our baseline (No Null Check Optimization) for SPECjvm98. We noticed that implicit null checking is especially effective for ***mtrt***. This is also true for Intel, though the improvement is smaller on the PowerPC platform than that on the Intel platform. This is because the execution cost for an explicit null check on the PowerPC platform (using a conditional trap) is smaller than that on the Intel platform.

6. CONCLUSIONS

In this paper, we have presented a new algorithm for null pointer check elimination, which has been implemented in the IBM Java Just-in-Time compiler. The architecture independent optimization moves null checks backward, and it is iterated for a few times with other optimizations to eliminate redundant null checks. This optimization maximizes the effectiveness of other optimizations. Then the architecture dependent optimization converts null checks to hardware traps in order to minimize the execution cost of null checking. Preliminary performance results show a significant performance improvement over the previously known best approach. Although we implemented our algorithm for Java, it is also applicable for other languages requiring null checking. We expect the importance of the techniques presented in this paper to grow further.

7. ACKNOWLEDGMENTS

We would like to thank the members of the TRL JIT Compiler team for helpful discussions and analysis of possible performance improvements. We also thank the IBM Java JIT Compiler Development group in Toronto for their helpful discussion.

8. REFERENCES

[1] G. Aigner, and U. Holzle. Eliminating Virtual Function Calls in C++ Programs, In *Proceedings of the 10th European Conference on Object-Oriented Programming - ECOOP '96*, Volume 1098 of Lecture Notes in Computer Science, Springer-Varlag, pp. 142-166, 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. Russel, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, J. Whaley, The Jalapeño virtual machine, *IBM Systems Journal*, Vol. 39, No. 1, 2000.

[3] J. Dean, D. Grove, C. Chambers. Optimization of object-oriented programs using static class hierarchy, In *Proceedings of the 9th European Conference on Object-Oriented Programming - ECOOP '95*, Volume 952 of Lecture Notes in Computer Science, Springer-Varlag, pp. 77-101, 1995.

[4] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).

[5] HotSpot homepage is http://java.sun.com/products/hotspot/.

[6] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. "Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler." In *Proceedings of the ACM SIGPLAN Java Grande Conference*, June 1999.

[7] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. "A Study of Devirtualization Techniques for a Java Just-In-Time Compiler." To appear in *the Conference on Object Oriented Programming Systems, Languages & Applications – OOPSLA*, 2000.

[8] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, Vol. 27, No. 7, pp. 224-234, San Francisco, CA, June 1992.

[9] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 5, pp. 777-802, 1995.

[10] T. Lindholm, Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).

[11] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies, *CACM*, Vol. 22, No. 2, Feb. 1979, pp. 96-103.

[12] Standard Performance Evaluation Corp. "SPEC JVM98 Benchmarks," http://www.spec.org/osg/jvm98/

[13] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, T. Nakatani. Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol. 39, No. 1, 2000.

[14] J. Whaley. Dynamic optimization through the use of automatic runtime specialization. M.Eng., Massachusetts Institute of Technology, May 1999.

[15] B.S. Yang, S.M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y.C. Chung, S. Kim, K. Ebcioglu, E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation, *Conference on Parallel Architectures and Compilation Techniques*, October 1999.