April 3, 2000
RT0350
Computer Science 18 pages

Research Report

Eliminating Exception Checks and Partial Redundancies for Java Just-in-Time Compilers

Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Eliminating Exception Checks and Partial Redundancies for Java Just-in-Time Compilers

Motohiro Kawahito
jl25131@jp.ibm.com

Hideaki Komatsu
komatsu@jp.ibm.com

Toshio Nakatani
nakatani@jp.ibm.com

IBM Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan

Abstract

We present new algorithms for eliminating null pointer checks, array bound checks, and partial redundancies from programs written in Java. These algorithms have been implemented in the latest IBM Java™ Just-In-Time (JIT) compiler. Preliminary performance results using SpecJVM98 show significant improvements over previously-known algorithms.

1. Introduction

The Java language[11] has a powerful exception-handling mechanism, which is useful for error handling, program control, and safety preservation. However, any instruction capable of 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. Also, array bound checks are required for every array access. In fact, these operations are quite common in typical Java programs.

A Just-in-Time (JIT) compiler, which generates native code from Java bytecode on the fly, must optimize and generate the native code for the best runtime performance without compromising the safety of Java. Exception check elimination is particularly important for the JIT compiler, not only because it improves the quality of the generated code by reducing the code size, but also because it increases the opportunity for other optimizations to be applied in a wider region by eliminating barriers to code motion.

JIT compilers can optimize Java programs more effectively if runtime trace information is used for optimizations. In the latest version of our JIT compiler, we modified the interpreter to collect the outcome of the first execution of every conditional branch and to pass that information to the JIT compiler to enhance partial redundancy elimination. It is not an ideal solution, in that the interpreter does not always pass accurate branch statistics but only the outcome of the first occurrence, but this is a trade off for increased efficiency of interpretation. Collecting precise branch statistics by the interpreter may significantly slow down its execution.

In this paper, we present a new algorithm for null pointer check elimination, array bound check elimination, and trace-based partial redundancy elimination, all of which have been implemented in the latest IBM Java™ Just-in-Time (JIT) compiler. Our new algorithm moves null pointer checks out of loops and utilizes the hardware trap mechanism. We enhanced Gupta's algorithm[2] to eliminate more array bound checks. We use runtime trace information and eliminate common subexpressions more aggressively on the first-taken path, employing a variant of the partial redundancy elimination algorithm[5]. We conducted experiments by running SpecJVM98 benchmark programs on Pentium III 600MHz, Windows NT 4.0, and IBM Developer Kit for Windows(R), Java™ Technology Edition, Version 1.2.2. Our preliminary performance results show significant improvements over previous approaches.

The rest of the paper is organized as follows. Section 2 summarizes previous work on each type of optimization. Section 3 gives an overview of our approach for each type of optimization. Section 4 presents the details of our

algorithms. Section 5 shows the performance results obtained in our experiments. Section 6 offers some concluding remarks and outlines future work.

2. Previous Work

2.1 Nullcheck Elimination using Forward Data-Flow Analysis

Previous JIT compilers, such as the Jalapeño Dynamic Optimizing Compiler[1, 9] from the IBM T.J. Watson Research Center or previous version of our JIT compiler[10, 13], eliminate null pointer checks (called ***nullcheck***s in this paper) by using forward data-flow analysis. However, there are two drawbacks to this approach:

- The implementation of *nullcheck* takes advantage of the hardware trap and the associated OS support function, but this elimination algorithm does not take into account this hardware support.
- Forward data-flow analysis cannot move loop invariant *nullcheck*s out of the loops. For example, when the first object access lies inside of a loop, the *nullcheck* for it must remain in the loop body.

2.2 Array Bound Check Elimination

Previous JIT compilers, such as the Intel JIT compiler[8], used a simple mechanism to eliminate array bound checks (called ***boundcheck***s in this paper) of only constant indices. In addition, when the array is created (using the ***newarray*** byte code), it used the size specified in the ***newarray*** to eliminate bounds checking on subsequent array accesses. The main advantage of this algorithm is its fast compilation. However, it has two limitations. First, the optimization scope is local, that is, the elimination algorithm is only applied within each basic block. Second, the algorithm eliminates only redundant *boundcheck* for the arrays with a constant index; more bound checks could be eliminated if a symbolic representation of indices were supported.

Gupta's algorithm[2], which was developed for a static compiler, eliminates the array bound checks in two steps. We explain Gupta's algorithm using an example in Figure 1. Here, we use an array ***a***, and ***ub*** as an upper bound of the array ***a***. For an array ***a***, boundcheck(0<=i) means that an array bound must be checked to see if **i** is no smaller than 0 and an exception will be thrown otherwise.

1) Original program	2) After first step	3) After second step
(a) boundcheck(0 <= i-1);	(a) boundcheck(0 <= i-1);	(a) boundcheck(0 <= i-1);
(b) boundcheck(i-1 < ub);	**(b) boundcheck(i < ub);**	(b) boundcheck(i < ub);
(c) t = a[i-1];	(c) t = a[i-1];	(c) t = a[i-1];
(d) i = i + 1;	(d) i = i + 1;	(d) i = i + 1;
(e) boundcheck(0 <= i);	**(e) boundcheck(0 <= i-1);**	**(e)**
(f) boundcheck(i < ub);	(f) boundcheck(i < ub);	(f) boundcheck(i < ub)
(g) boundcheck(0 <= i-1);	(g) boundcheck(0 <= i-1);	**(g)**
(h) boundcheck(i-1 < ub);	(h) boundcheck(i-1 < ub);	**(h)**
(i) t += a[i] + a[i-1];	(i) t += a[i] + a[i-1];	(i) t += a[i] + a[i-1];

Figure 1: Gupta's algorithm

In the first step, Gupta's algorithm will move *boundcheck*s up in the backward data-flow analysis as in the following:

1. The *boundcheck* (e) is replaced by (g), because (e) will always hold if (g) holds for **i**.
2. The lower *boundcheck*s (e) and (g) cannot be moved up across (d), because these *boundcheck*s may not be valid any more if they are moved.
3. The upper *boundcheck* (f) can be moved up across (d), and (b) is replaced by (f) because (b) will always hold if (f) holds for **i**.

In the second step, Gupta's algorithm will move *boundcheck*s down and will eliminate the redundant *boundcheck*s in the forward data-flow analysis as in the following:

1. The upper *boundcheck* (b) cannot be moved down across (d), because this *boundcheck* may not be valid any more if it is moved.
2. The lower *boundcheck* (a) can be moved down across (d), and (e) and (g) are eliminated because (a), (e), and (g) are identical checks.
3. The *boundcheck* (h) is eliminated because (h) will always hold if (f) holds for **i**.

However, Gupta's algorithm has the following drawback:

- Either the lower or the upper *boundcheck* cannot be moved across the index update. If both lower and upper *boundcheck*s could be moved, then more *boundcheck*s could be eliminated.

2.3 Common Sub-Expression Elimination

Previous JIT compilers, such as the Intel JIT compiler[8], used a fast common sub-expression elimination (CSE) algorithm that eliminates common sub-expressions from each basic block. The main advantage of this algorithm is its fast compilation. However, it has two limitations. First, the optimization scope is local; that is, the elimination algorithm is applied only within a basic block. Second, backward code motion, such as loop-invariant code motion, is not applied.

Partial-redundancy elimination (PRE) is an optimization that combines CSE and loop-invariant code motion. PRE inserts and deletes computations in the flow graph in such a way that after the transformation each path contains a fewer computations than before. PRE was originated by Morel and Renvoise[3]. Later Knoop and others developed a new method[4, 5] to avoid any unnecessary register pressure and code motion. They describe two methods of moving code in [5]: "busy code motion" (BCM), which is essentially the same as the original method developed by Morel and Renvoise, and "lazy code motion" (LCM). BCM inserts an expression at every first point where the expression can be moved, and eliminates common expressions that can be reached by the inserted expressions. LCM inserts an expression as late as possible, to reduce the lifetime of the variable that holds the result of the expression; moreover, it avoids unnecessary code motion. However, LCM has the following limitation, particularly when it is applied to the JIT compiler, which uses run-time trace information for aggressive optimization:

- When there is a path on which the expression is not executed, the expression cannot be moved even on a frequently executed path. Gupta and others[15] solved this problem, but their algorithm is too slow to use for JIT compilers.

3. Our Approach

We use the following approach to solve the issues described in the previous section:

3.1 Enhancement of *Nullcheck* Elimination

We begin by explaining our implementation of the *nullcheck*. We define two kinds of *nullcheck*s:

- *Explicit Nullcheck*s, which need to generate actual checking code.
- *Implicit Nullcheck*s, which do not need to generate actual checking code, but rely on the hardware trap and its associated OS support function.

To implement *nullcheck*, we use *implicit nullcheck* wherever possible. However, in some cases we use an *explicit nullcheck* in order to maintain the Java language specification. For example, when an instruction requiring *nullcheck* is a dead store, the instruction can be eliminated by using an *explicit nullcheck* to replace the dead store instruction.

To take another example, when method inlining is applied to a call site and an invoked method is specified at

compile-time, say *invokenonvirtual*, an *explicit nullcheck* needs to be generated.

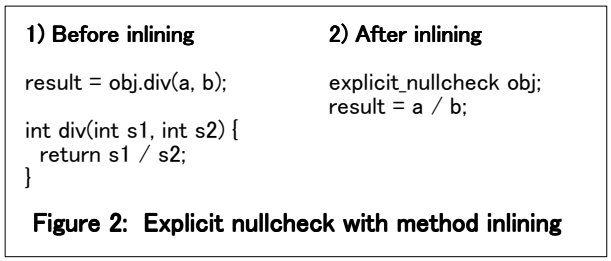

Figure 2: Explicit nullcheck with method inlining

Figure 2 shows an example. If the *explicit nullcheck* in Figure 2 (2) is replaced by an *implicit nullcheck*, no content of *obj* is accessed. Therefore, when *obj* is a null pointer, no exception occurs and execution continues. This violates the Java language specification. *Explicit nullcheck*s with method inlining appear frequently in typical Java programs, and thus *explicit nullcheck* elimination is an effective optimization. *Implicit nullcheck* elimination is also effective, since an *implicit nullcheck* will become a barrier to the application of other optimizations across the *nullcheck*.

3.1.1 *Explicit Nullcheck* Elimination

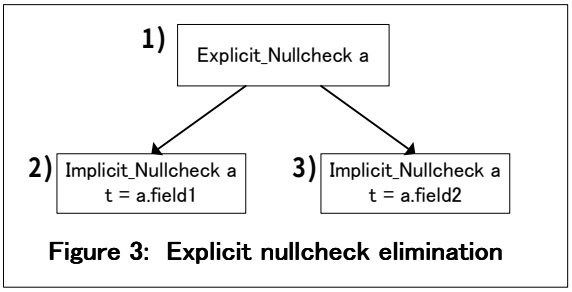

Figure 3: Explicit nullcheck elimination

We explain *explicit nullcheck* elimination with reference of Figure 3. In this case, if the *explicit nullcheck* in (1) is deleted, execution on both paths becomes faster. However, traditional approaches using forward flow analysis cannot delete this *explicit nullcheck* in (1), but only the *implicit nullcheck*s in (2) and (3). This is because the *explicit nullcheck* in (1) is the first point at which the contents of the object are accessed. In contrast, our method can eliminate *explicit nullcheck*s wherever possible if the following instruction that requires a *nullcheck* can be used as a substitute for the *explicit nullcheck*. To minimize *explicit nullcheck*s, *explicit nullcheck* elimination needs to be performed before the optimizations described in section 3.1.2.

3.1.2 Code Motion of *Nullcheck*

We enhance busy code motion (BCM)[5] in order to eliminate partially-redundant *nullcheck*s and also to move *nullcheck*s out of loops.

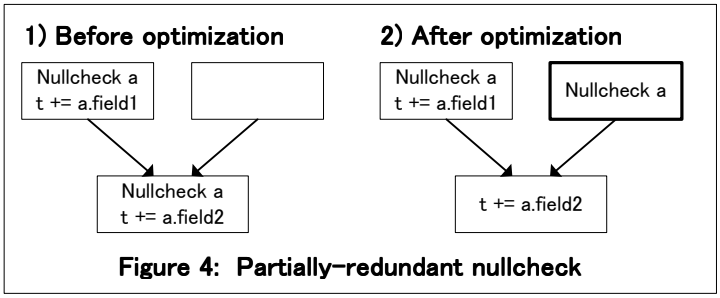

Figure 4: Partially-redundant nullcheck

Figure 4 shows an example of a partially-redundant *nullcheck*. In Figure 4 (1), the *nullcheck* located at the junction cannot be eliminated without code motion, because the right path does not include any *nullcheck*. Therefore, *nullcheck* will be executed twice along the left path. This optimization inserts a *nullcheck* in the right basic block and eliminates

the *nullcheck* at the junction. As a result, a *nullcheck* will be executed only once along each path.

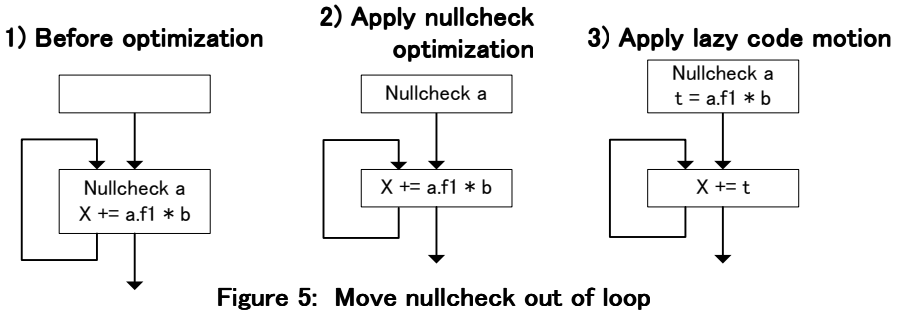

Figure 5: Move nullcheck out of loop

Figure 5 shows an example of moving *nullcheck*s out of loops. In Figure 5(1), the *nullcheck* cannot be eliminated by the previous approach using forward data-flow analysis, because the outer path does not include any *nullcheck*. When a *nullcheck* remains in a loop, the field access (a.f1) cannot be moved out of the loop. Figure 5(3) shows the final result, which cannot be achieved without moving the code of the *nullcheck* out of the loop.

We note here that the attribute (explicit/implicit) must be determined for the *nullcheck* inserted by the code motion as follows. If one of the original *nullcheck*s for the inserted one has the *explicit* attribute, then the attribute of the inserted *nullcheck* is determined to be an *explicit*. We note here that the optimization described in section 3.1.1 should be executed before the optimization in section 3.1.2, in order to eliminate as many *explicit nullcheck*s as possible.

3.2 Enhancement for Array Bound Check Elimination[1]

We enhance Gupta's array bound check elimination algorithm[2, 12] in order to eliminate more array bound checks. Our algorithm is basically the same as Gupta's, which modifies *boundcheck*s by backward data-flow analysis as the first step and eliminates *boundcheck*s by forward data-flow analysis as the second step. However, our algorithm is improved over Gupta's algorithm in the following four areas:

- Our algorithm can move both the upper and the lower *boundcheck*s when an index variable is updated by adding a constant.

  While Gupta's algorithm propagates the *boundcheck* without modifying the expression of the *boundcheck*, our algorithm will modify the subscript expression of the *boundcheck* to allow the *boundcheck* to be moved across the definition of the relevant variable. Therefore, our algorithm can propagate more *boundcheck*s than Gupta's. We explain this enhancement using the same example as before (Figure 1).

1) Original program	2) After first step	3) After second step
(a) boundcheck(0 <= i-1);	(a) boundcheck(0 <= i-1);	(a) boundcheck(0 <= i-1);
(b) boundcheck(i-1 < ub);	**(b) boundcheck(i+1 < ub);**	(b) boundcheck(i+1 < ub);
(c) t = a[i-1];	(c) t = a[i-1];	(c) t = a[i-1];
(d) i = i + 1;	(d) i = i + 1;	(d) i = i + 1;
(e) boundcheck(0 <= i);	**(e) boundcheck(0 <= i-1);**	**(e)**
(f) boundcheck(i < ub);	(f) boundcheck(i < ub);	**(f)**
(g) boundcheck(0 <= i-1);	(g) boundcheck(0 <= i-1);	**(g)**
(h) boundcheck(i-1 < ub);	(h) boundcheck(i-1 < ub);	**(h)**
(i) t += a[i] + a[i-1];	(i) t += a[i] + a[i-1];	(i) t += a[i] + a[i-1];

Figure 6: Enhancement for array bound check elimination (1)

In the first step, our algorithm will move *boundcheck*s up in the backward data-flow analysis. The *boundcheck* (e) is replaced by (g), because (e) will always hold if (g) holds for **i**. In our algorithm, both the upper and the lower

[1] This algorithm was briefly introduced in our previous paper[10, 13].

*boundcheck*s can be moved up across (d) by modifying the expressions of the *boundcheck*s in the following. Here we assume that $\mathbf{i_1}$ is the value of **i** at (c) and $\mathbf{i_2}$ is the value of **i** at (e).

(d) $i_2 = i_1 + 1$
(e) boundcheck($0 <= i_2 - 1$) ≡ boundcheck($0 <= i_1$) -- (e')
(f) boundcheck($i_2 < ub$) ≡ boundcheck($i_1 + 1 < ub$) -- (f')

Finally, (b) is replaced by (f'), because (b) will always hold if (f') holds for **i**. Figure 6(2) is the result of the first step.

In the second step, our algorithm will move *boundcheck*s down and eliminate *boundcheck*s in the forward data-flow analysis. In our algorithm, both the upper and the lower *boundcheck*s can be moved down across the (d) by modifying the expressions of the *boundcheck*s as in the following.

(d) $i_2 = i_1 + 1$ ≡ $i_1 = i_2 - 1$
(a) boundcheck($0 <= i_1 - 1$) ≡ boundcheck($0 <= i_2 - 2$) -- (a')
(b) boundcheck($i_1 + 1 < ub$) ≡ boundcheck($i_2 < ub$) -- (b')

The *boundcheck*s (e) and (g) can be eliminated because (e) and (g) will always hold if (a') holds for **i**. The *boundcheck*s (f) and (h) can also be eliminated because (f) and (h) will always hold if (b') holds for **i**. Figure 6(2) is the result of the second step. Notice that our algorithm eliminated more array bound checks than the original algorithm.

- Our algorithm can create a new *boundcheck* with a constant index based on the maximum and minimum constant offset from the index variable.

```
boundcheck( 0 <= i-1 );
boundcheck( i+1 < ub );
(We create a new available information, that is  boundcheck( 2 < ub ) , at this point.)
a[i+1] = a[i] + a[i-1];
boundcheck( 2 < ub );   /* This array bound check can be eliminated */
a[2] = 0;
```

Figure 7: Enhancement for array bound check elimination (2)

For example in Figure 7, our algorithm will create a new *boundcheck* with the constant offset equivalent to the value of (maximum offset − minimum offset). In Figure 7, since the maximum constant offset of ***i*** is 1 and the minimum constant offset is −1, our algorithm will add the new boundcheck(1−(−1) < ub) = boundcheck(2 < ub). Therefore, the next boundcheck(2 < ub) can be eliminated. We note here that the new *boundcheck* is not actually generated but it is only used for eliminating the following *boundcheck*s.

- If the maximum (or minimum) value of the subscript expression is known and the *boundcheck* with the maximum (or minimum) index value has already been performed earlier in the program, then our algorithm can eliminate all the following *boundcheck*s with the known indices.

```
boundcheck( 10 < ub );
a[10] = 1;
for (i = 0; i < 10; i++) {
  boundcheck( 0 <= i );  /* This array bound check can be eliminated */
  boundcheck( i < ub );  /* This array bound check can be eliminated */
  a[i] = 0;              /* min. value of i is 0, max. value of i is 9 */
}
```

Figure 8: Enhancement for array bound check elimination (3)

For example, in Figure 8, the minimum value of i is known to be 0 and the maximum value of i is known to be 9 inside the loop[2]. Therefore, both the lower and the upper *boundcheck*s in the loop can be eliminated after the boundcheck(10 < ub) is performed.

---

[2] We have used the same approach as Chambers's algorithm[14] to compute range of local variables.

- Our algorithm can eliminate *boundcheck*s whose subscript expression is equivalent to the average of those index variables whose *boundcheck*s are already performed. This is based on the following inequalities.

  Minimum(variables) ≤ Average(variables) ≤ Maximum(variables)

```
boundcheck( left < ub );
boundcheck( 0 <= left );
w1 = a[left];
boundcheck( right < ub );
boundcheck( 0 <= right );
w2 = a[right];
boundcheck( (left+right)/2 < ub );   /* This array bound check can be eliminated */
boundcheck( 0 <= (left+right)/2 );   /* This array bound check can be eliminated */
center = a[(left+right)/2];
```

Figure 9: Enhancement for array bound check elimination (4)

For example, in Figure 9, the following inequalities will hold. Therefore, the two *boundcheck*s involving (left+right)/2 can be eliminated.

Minimum(left, right) ≤ Average(left, right) ≤ Maximum(left, right)

3.3 Enhancement for Lazy Code Motion (LCM)

We enhance lazy code motion (LCM) to allow scalar replacement and common subexpression elimination by moving redundant expressions aggressively along frequently executed paths, even if they cannot be moved on all of the paths in the original LCM algorithm.

For example, in Figure 10(1), there are two instances of "a * b * c" on one side of the conditional branch. The traditional LCM algorithm cannot treat them as common, since there is no common expression on the other path. Our algorithm will move the common expression on the "frequently taken edge" if it does not cause any side effects (that is, if the expression is potentially throwing an exceptions or writing the result into memory). As a result, we can improve the runtime performance on the frequently executed path. Figure 10(2) shows the result of the transformation, where "a * b * c" is optimized along the "frequently taken edge." If common subexpressions are eliminated from the loop body, the performance improvement will become even more significant. Let us note here that the machine code for "i=t" will not be generated in a later phase, because the lifetime of ***t*** ends at the trivial copy "i=t", and therefore register allocator will treat ***i*** and ***t*** as the same register.

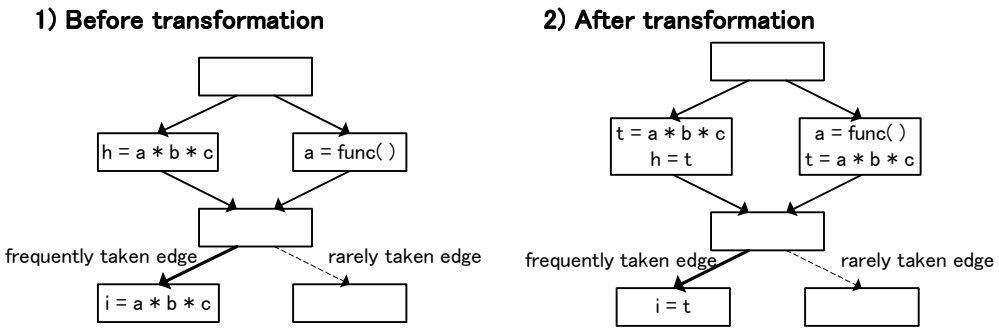

Figure 10: Enhancement of lazy code motion

Gupta's PRE algorithm[15] uses path profiles and computes the cost and benefit along each path. In contrast, our algorithm uses only edge profiles created by the interpreter. Owing to this simplification, as shown in Table 3 in Section 5.3, there was no noticeable increase in compilation time compared to the original LCM algorithm.

4. Outlines of Our Algorithms

4.1 Algorithm for *Nullcheck* Elimination

We first describe the outline of our algorithm for *nullcheck* elimination.

4.1.1 Algorithm for *Explicit Nullcheck* Elimination

This optimization is performed when both *implicit nullcheck*s and *explicit nullcheck*s are generated. Out(n) is the set of substitutes for *nullcheck* at the exit of the basic block. This set is computed by solving the backward data-flow equations given below.

Gen(n): The set of substitutes for *nullcheck* at the entry of basic block *n*. Both *nullcheck* and accessing the contents of an object are included in the set.

Kill(n) : The set of *nullcheck*s killed through basic block *n* in the backward direction. Concrete instructions are definitions of a variable that is used in *nullcheck* or that causes side effects, such as other kinds of exceptions or writes to memory.

$$Out(n) = \bigcap_{m = Succ(n)} In(m)$$

$$In(n) = (Out(n) - Kill(n)) \cup Gen(n)$$

The set of substitutes for *nullcheck* at each point inside a basic block *n* is determined from Out(n). An *explicit nullcheck* is eliminated if we determine that a substitute for *nullcheck* can reach the *explicit nullcheck* in the backward direction.

4.1.2 Algorithm for *Nullcheck* Insertion

This optimization can be performed at any time; but when both *implicit nullcheck* and *explicit nullcheck* are generated, an algorithm that determines the attribute for the inserted *nullcheck* is necessary.

Out(n) is the set of movable *nullcheck*s at the exit of the basic block. This set is computed by solving the backward data-flow equations given below.

Gen(n) : The set of movable *nullcheck*s at the entry of basic block *n*. Only *nullcheck*s are included in the set.

Kill(n) : The set of *nullcheck*s killed through basic block *n* in the backward direction. It is the same as Kill(n) in section 4.1.1.

$$Out(n) = \bigcap_{m = Succ(n)} In(m)$$

$$In(n) = (Out(n) - Kill(n)) \cup Gen(n)$$

Next, Earliest(n) is the set of the first points of *nullcheck* in the region where *nullcheck* can be moved toward the backward. This set is computed by means of the following equation:

$$Earliest(n) = \left(\bigcup_{m = Pred(n)} \overline{Out(m)} \right) \cap Out(n)$$

The *nullcheck*s in Earliest(n) are inserted at exit of basic block n.

Algorithm for attribute determination

When both an *implicit nullcheck* and an *explicit nullcheck* are generated, the attribute for the inserted *nullcheck* must be determined.

Out_explicit(n) is the set of *explicit nullcheck*s at the exit of the basic block. This set is computed by solving the

backward data-flow equations given below.

Gen(n) : The set of movable *explicit nullcheck*s at the entry of basic block *n*. Only *explicit nullcheck*s are included in the set.

Kill(n) : The set of *nullcheck*s killed through basic block *n* in the backward direction. It is the same as Kill(n) in section 4.1.1.

$$Out\_explicit(n) = \bigcup_{m = Succ(n)} In\_explicit(m)$$

$$In\_explicit(n) = (Out\_explicit(n) - Kill(n)) \cup Gen(n)$$

The set of *explicit nullcheck*s at each point inside a basic block *n* is determined from Out_explicit(n). The attribute of an inserted *nullcheck* is determined to be *explicit nullcheck* when the set of an *explicit nullcheck* contains the inserted *nullcheck*. The *implicit nullcheck* inserted by the optimization described in 4.1.2 is located at a different point from that at which the NullPointerException actually occurs. This difference is corrected by the transaction of 4.1.4; the reason we perform *implicit nullcheck* optimization, which consumes compilation time, is that *nullcheck* becomes barrier to the application of other optimizations, such as scalar replacement or code scheduling.

4.1.3 Algorithm for *Nullcheck* Elimination

This optimization can be performed any time. Its purpose is to eliminate *nullcheck*s that have already been checked somewhere along the data-flow.

In(n) is the set of available *nullcheck*s at the entry of the basic block. This set is computed by solving the forward data-flow equations given below.

Gen(n) : The set of available *nullcheck*s at the exit of basic block *n*. Both *nullcheck* and accessing the contents of an object are included in the set.

Kill(n) : The set of *nullcheck*s killed through basic block *n* in the forward direction. A concrete instruction is a definition of a variable used in *nullcheck*.

Non_null(n) : The set of non-null objects that are determined by a conditional branch or a "this" object in the original Java program at the entry of the basic block *n*. Concrete examples are *ifnull*, *ifnonnull*, and the "this" object for an instance method.

$$In(n) = \left(\bigcap_{m = Pred(n)} Out(m)\right) \cup Non\_null(n)$$

$$Out(n) = (In(n) - Kill(n)) \cup Gen(n)$$

The set of available *nullcheck*s at each point inside a basic block *n* is determined from In(n). A *nullcheck* ***C*** is eliminated if we determine that there is an available *nullcheck* that can reach ***C***.

4.1.4 Algorithm for *Implicit Nullcheck* Propagation

This phase must be executed after the optimization described in section 4.1.2 has been applied and an *implicit nullcheck* has been generated. It should be executed after the optimizations of code motion. Figure 11 explains why it must be applied. We assume that all *nullcheck*s in Figure 11 are *implicit nullcheck*s. As we explained before, we cannot move "**a.f1**" in Figure 11(a) out of the loop without *implicit nullcheck* optimization. Therefore, *implicit nullcheck* optimization should be applied. In Figure 11 (b), the *implicit nullcheck* is placed at a different point from that at which the NullPointerException actually occurs. This might cause an invalid optimization result, and therefore it is necessary to apply *implicit nullcheck* propagation, which propagates the *implicit nullcheck* to the exception occurrence point.

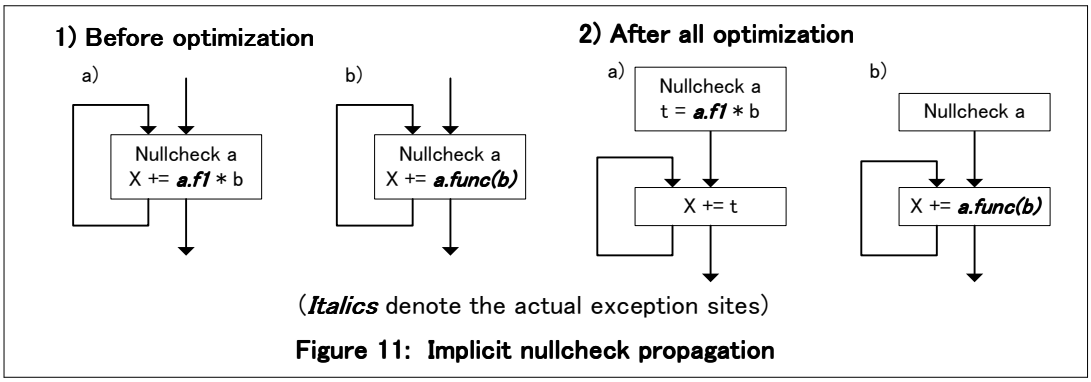

Figure 11: Implicit nullcheck propagation

In(n) is the set of *implicit nullcheck*s whose objects have not yet been accessed at the entry of the basic block. This set is computed by solving the forward data-flow equations given below.

Gen(n) : The set of *implicit nullcheck*s whose object has not yet been accessed at the exit inside the basic block *n*. Only *implicit nullcheck*s are included in the set.

Kill(n) : The set of *nullcheck*s that contains Kill(n) described in section 4.1.3 and instructions for accessing the contents of its object.

$$In(n) = \bigcup_{m = Pred(n)} Out(m)$$

$$Out(n) = (In(n) - Kill(n)) \cup Gen(n)$$

The set of a *nullcheck* whose object has not yet been accessed at each point inside a basic block *n* is determined from In(n). Information on exception occurrence is added if it is determined that the set can reach the instruction by accessing the contents of the object. Finally, all *implicit nullcheck*s are eliminated.

4.2 Algorithm for Array Bound Check Elimination

In this section, we outline our algorithm for array bound check elimination.

4.2.1 Algorithm for Modifying *Boundcheck*s

The purpose of this optimization is to combine as many *boundcheck*s as possible. Out(n) is the set of movable *boundcheck*s at the exit of the basic block. This set is computed by solving the backward data-flow equations given below.

Gen(n) : The set of movable *boundcheck*s at the entry of basic block *n*.

Kill(n) : The set of *boundcheck*s killed through basic block *n* in the backward direction. Concrete instructions are definitions of an object that is used in *boundcheck* or instructions that cause side effects, such as other kinds of exceptions or a write to memory. A definition of a variable of the subscript used in *boundcheck* is not included in the set.

Effect(n, v) : This summarizes the effect of ***n*** on variable ***v***. It consists of a changed state and a value. The ***state*** has four bits denoting "change," "constant," "increase," and "decrease." The ***value*** is valid when the "constant" bit of ***state*** is true.

$$Out(n) = \bigcap_{m = Succ(n)} In(m)$$

$$In(n) = backward(Out(n) - Kill(n), n) \cup Gen(n)$$

backward( ) denotes the following transaction, which returns a set. In the description of backward(), I(u) denotes that f(v) in I is replaced by f(u).

```
backward(INPUT, n) {
  RET = φ
  for (each boundcheck I ∈ INPUT){
    v = index variable in I
    if ("change" bit in Effect(n, v) == false){
      RET = RET ⋃ I(v)
    } else if ( ("increase" bit in Effect(n, v) == true) || ("decrease" bit in Effect(n, v) == true) ) {
      f(v) = subscript expression in I
      if (f(v) is monotone by v){
        if ("constant" bit in Effect(n, v) == true){
          C = changed value in Effect(n, v)
          RET = RET ⋃ I(v + C)
        } else {
          switch(kind of comparison in I){
          case compared with lower bound :
            if (  ("increase" bit in Effect(n, v) == true && f(v) is a decreasing expression  by v) ||
                  ("decrease" bit in Effect(n, v) == true && f(v) is a increasing expression  by v) ){
              RET = RET ⋃ I(v)
            }
            break;
          case compared with upper bound :
            if (  ("increase" bit in Effect(n, v) == true && f(v) is a increasing expression  by v) ||
                  ("decrease" bit in Effect(n, v) == true && f(v) is a decreasing expression  by v) ){
              RET = RET ⋃ I(v)
            }
            break;
          }
        }
      }
    }
  }
  return (RET)
}
```

The set of a movable *boundcheck* at each point inside a basic block *n* is determined from Out(n). A bound check ***C*** is modified if we determine that there is another check ***C'*** that is a movable *boundcheck* at the point immediately following ***C*** and that ***C'*** contains ***C***. In this case, we replace ***C*** by ***C'***.

4.2.2 Algorithm for *Boundcheck* Elimination

The purpose of this optimization is to eliminate *boundcheck*s that have already been checked somewhere along the data-flow. In(n) is the set of available *boundcheck*s at the exit of the basic block. This set is computed by solving the forward data flow equations given below.

Gen(n) : The set of available *boundcheck*s at the exit of basic block *n*. We can determine the following already checked *boundcheck*s:

- The *boundcheck* itself
- The constant index obtained by computing (maximum offset − minimum offset) from the index variable
- The constant index obtained by computing (creation size − 1) at the instruction that is created by kind of ***new*** with constant size.

Kill(n) : The set of *boundcheck*s killed through basic block *n* in the forward direction. Concrete instructions are definitions of an object used in *boundcheck*. A definition of a variable of the subscript used in *boundcheck* is not included in the set.

Effect(n, v) : The same as Effect(n, v) in section 4.2.2.

$$In(n) = \bigcap_{m\ =\ Pred(n)} Out(m)$$

Out(n) = forward(In(n) − Kill(n), n) ⋃ Gen(n)

forward( ) denotes the following transaction, which returns a set. In the description of forward(), I(u) denotes that f(v) in **I** is replaced by f(u).

```
forward(INPUT, n) {
  RET = φ
  for (each boundcheck I ∈ INPUT){
    v = index variable in I
    if ("change" bit in Effect(n, v) == false){
      RET = RET ∪ I(v)
    } else if ( ("increase" bit in Effect(n, v) == true) || ("decrease" bit in Effect(n, v) == true) ) {
      f(v) = subscript expression in I
      if (f(v) is monotone by v){
        if ("constant" bit in Effect(n, v) == true){
          C = changed value in Effect(n, v)
          RET = RET ∪ I(v − C)
        } else {
          switch(kind of comparison in I){
          case compared with lower bound :
            if (  ("increase" bit in Effect(n, v) == true && f(v) is a increasing expression  by v) ||
                  ("decrease" bit in Effect(n, v) == true && f(v) is a decreasing expression  by v) ) {
              RET = RET ∪ I(v)
            }
            break;
          case compared with upper bound :
            if (  ("increase" bit in Effect(n, v) == true && f(v) is a decreasing expression  by v) ||
                  ("decrease" bit in Effect(n, v) == true && f(v) is a increasing expression  by v) ) {
              RET = RET ∪ I(v)
            }
            break;
          }
        }
      }
    }
  }
  return (RET)
}
```

The set of available *boundcheck*s at each point inside a basic block *n* is determined from In(n). A *boundcheck* is eliminated if we determine as follows that the *boundcheck* ***C*** has already been checked:

- There is an available boundcheck ***C'*** that contains ***C*** at the point immediately preceding ***C***.
- The maximum (or minimum) value ***V*** of the subscript expression in ***C*** is known, and there is an available boundcheck with ***V***.
- The subscript expression of ***C*** consists of the average of previously checked index variables.

4.3 Algorithm for Enhanced Lazy Code Motion

We use lazy code motion, which minimizes the lifetime of temporary variables, to perform scalar replacement and common sub-expression elimination. There is a problem in implementing LCM in a JIT compiler because of its time complexity. To reduce the compilation time, we limit the number of expressions to 32 or 64, since these can be represented by non-array bit vectors. We count expressions in a method according to the specified weight of its loops and limit the number of expressions according to the count.

We modify the original algorithm as follows:

1. We determine a set of expressions that do not have any side effects (exceptions or writes to memory).
2. When the ***Down-Safety*** described in [2, 3] is computed, we ignore the edge of the "rarely taken edge" for the set not causing any side effects.

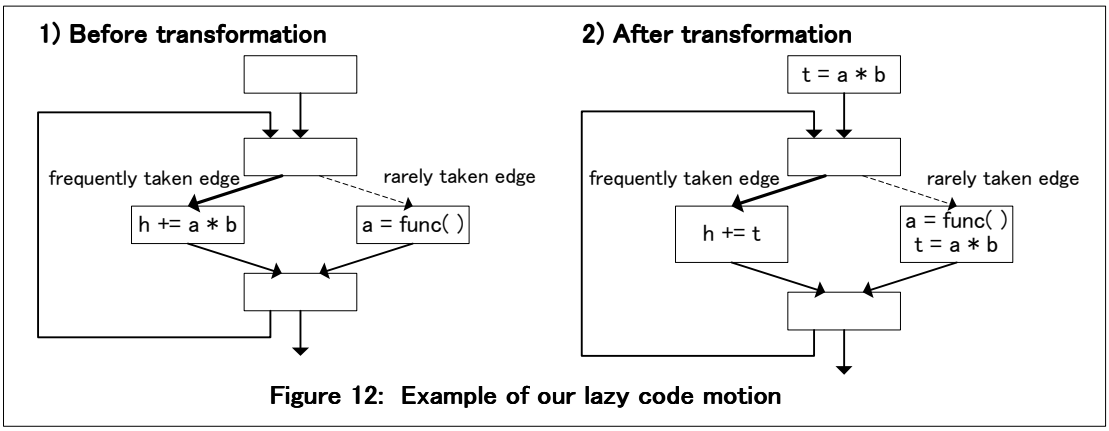

Figure 12: Example of our lazy code motion

If the optimization applies to expressions in loops, the effect is significant. Figure 12 shows an example of the results obtained by our algorithm. Here the expression "a * b" is moved off of the frequently executed path and therefore will be executed much fewer times.

5. Experimental Results

We chose SPECjvm98[7], which is a set of industry-standard client benchmark programs, for the evaluation of our individual optimizations. The measurements were performed in test mode with a count of 100, as specified for the SPEC-compliant mode. All the experiments described below were conducted on an IBM IntelliStation M Pro (Pentium III 600MHz with 384 MB of RAM), Windows NT 4.0 Service Pack 5, and IBM Developer Kit for Windows(R), Java™ Technology Edition, Version 1.2.2. Our implementation of *nullcheck* utilized the hardware trap (*implicit nullcheck*), and our implementation of *boundcheck* used compare and branch. The runtime trace on the conditional branch, which is created by the interpreter, was recorded only on the first execution.

We disabled each optimization (denoted as "No Null", "No Array", and "No Lazy" in Table 1) individually to show the effectiveness over the full optimizations that enabled all optimizations (denoted as "All New" in Table 1). To compare the performance improvement over previous approach, we also implemented Whaley's algorithm[1] (denoted as "Old Null" in Table 1) for *nullcheck* elimination, Gupta's algorithm[2] (denoted as "Old Array" in Table 1) for *boundcheck* elimination, and Knoop's algorithm[5] (denoted as "Old Lazy" in Table 1) for lazy code motion. Table 1 shows the results (time in seconds) for each case.

Table 1: Improvement of individual optimizations

(unit : sec)	mtrt	jess	compress	db	mpeg	jack	javac
ALL NEW	7.03	8.08	18.03	23.95	12.03	11.69	14.83
No Null	7.61	8.16	18.25	24.45	12.14	11.86	14.86
Old Null	7.56	8.13	18.11	24.31	12.14	11.76	14.85
No Array	7.14	8.31	18.59	24.42	13.94	11.94	15.27
Old Array	7.09	8.14	18.19	24.11	12.16	11.73	15.07
No Lazy	7.09	8.48	17.93	24.22	13.81	12.39	15.13
Old Lazy	7.02	8.08	17.94	23.97	12.34	12.00	14.86

ALL NEW: All new optimizations are applied.
No Null: Disable *nullcheck* optimization.
Old Null: Use Whaley's algorithm[1] for null check elimination. Enable other optimizations.
No Array: Disable *boundcheck* optimization.
Old Array: Use Gupta's algorithm[2] for array bound check elimination. Enable other optimizations.
No Lazy: Disable optimization by lazy code motion.
Old Lazy: Use Knoop's algorithm[5] for lazy code motion. Enable other optimizations.

5.1 Improvement of Our Exception Check Elimination

Figure 13 shows the percentage of performance improvement achieved by the *nullcheck* optimization described in section 4. It has been noticed that our *explicit nullcheck* elimination is particularly effective for ***mtrt*** after method inlining is performed. This is because ***mtrt*** has those small methods (to access data in a class) which are called frequently and many *explicit nullcheck*s associated with them can be eliminated only after they are inlined.

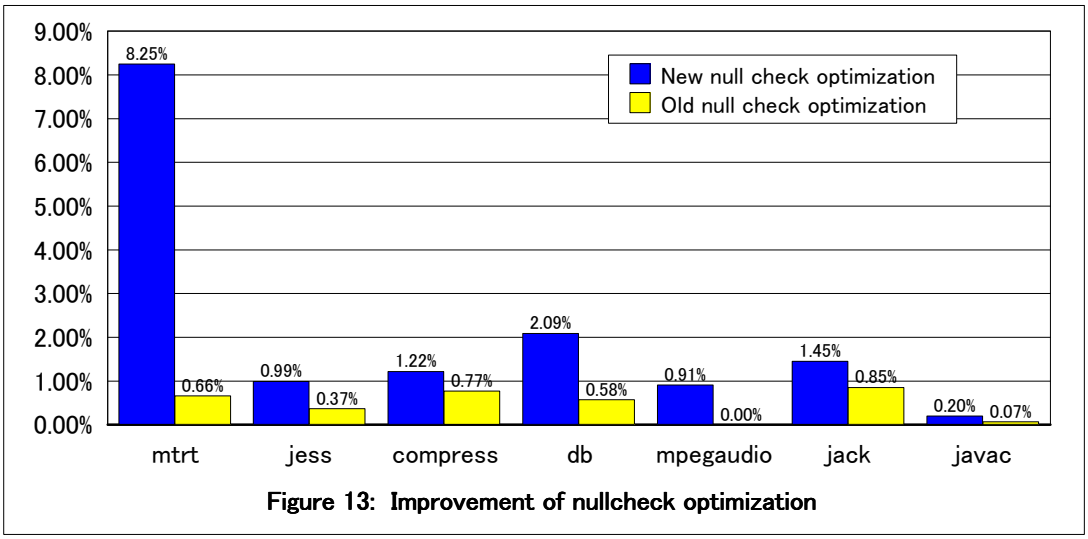

Figure 13: Improvement of nullcheck optimization

Figure 14 shows the percentage of performance improvement achieved by the *boundcheck* optimization described in section 4. This optimization is most effective for ***mpegaudio***, because it performs a large number of array accesses.

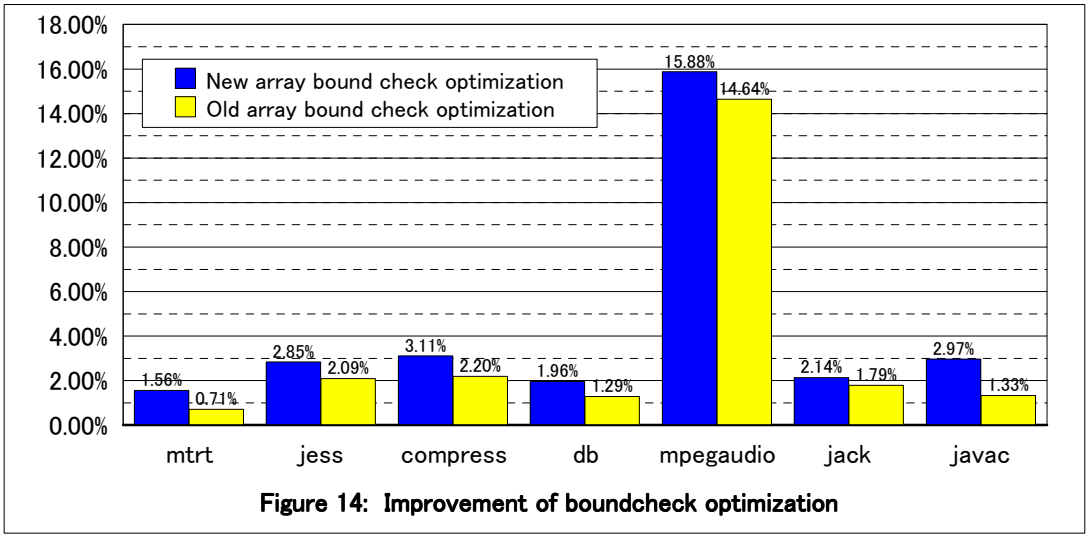

Figure 14: Improvement of boundcheck optimization

5.2 Improvement of Our Lazy Code Motion

Figure 15 shows the percentage of performance improvement achieved by our lazy code motion algorithm described in section 4.

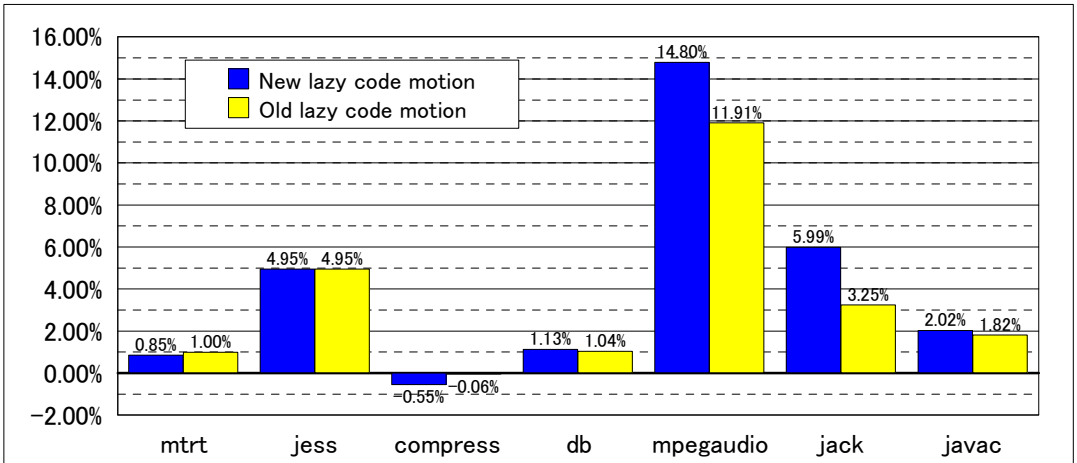

Figure15: Improvement of lazy code motion

This optimization is most effective for ***mpegaudio***. As shown in Figures 14 and 15, we must implement both array bound check elimination and enhanced LCM to achieve the best performance of ***mpegaudio***. It has been noticed that optimizing multidimensional arrays is important to improve performance of ***mpegaudio***. In contrast, this optimization is least effective for ***compress***, which is slightly worse (0.49%) than that with the original LCM. This might be caused by inaccurate runtime trace information. To validate this, we collected branch statistics for each program. Figure 16 shows a hit ratio of the trace information, that is the ratio of the outcome of the first execution of every conditional branch over the outcomes of all the branches actually executed in the whole life of each program. In summary, ***compress*** shows the worst hit ratio among all the benchmark programs.

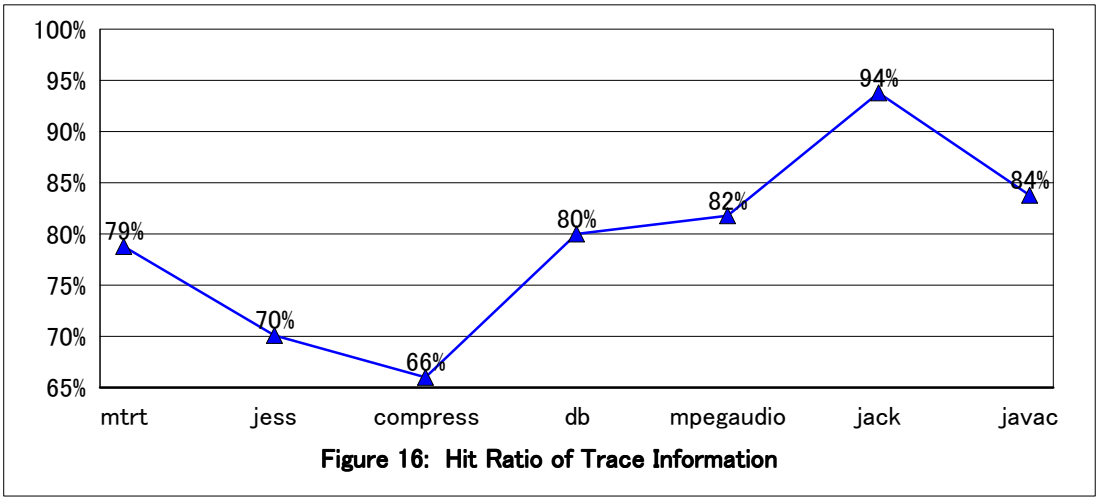

Figure 16: Hit Ratio of Trace Information

5.3 Compilation Time

In this section, we compare the compilation time in our approach with that in the previous one. We assume that the difference between the first run and the best run is essentially due to compilation time. Table 2 shows the time for the first run, best run, and the compilation time. Figure 17 shows the percentage of the compilation time over the whole execution time (that is, the time spent for the first run). In summary, ***javac*** spends the greatest percentage of its time for compilation among the benchmark programs in SPECjvm98.

Table 2: Compilation time (seconds)

	mtrt	jess	compress	db	mpegaudio	jack	javac
first run	9.01	10.79	18.18	24.41	12.89	12.92	20.75
best run	7.03	8.08	18.03	23.95	12.03	11.69	14.83
compilation time	1.98 (21.98%)	2.71 (25.12%)	0.15 (0.83%)	0.46 (1.88%)	0.86 (6.67%)	1.23 (9.52%)	5.92 (28.53%)

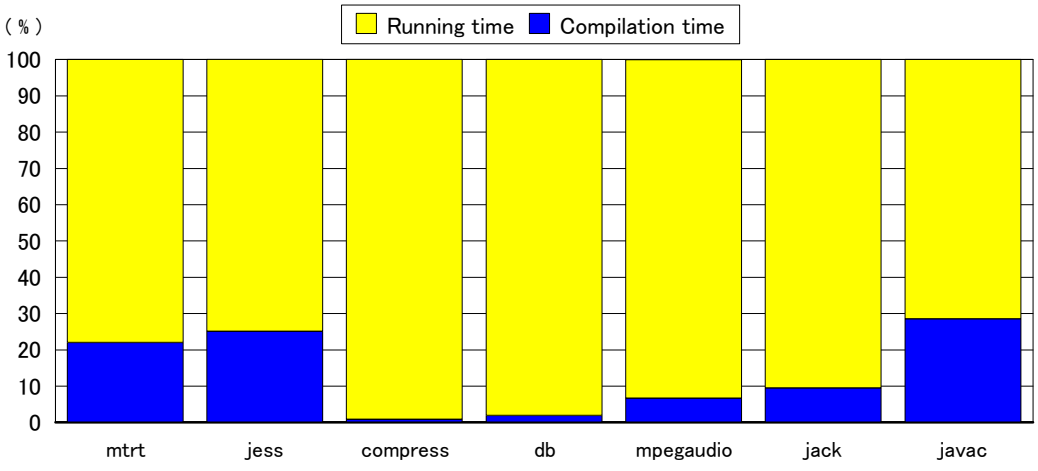

Figure 17: Percentage of compilation time (100% = first run)

We further measured the breakdown of the compilation time by using a trace tool available in AIX, and computed the compilation time by taking into account platform differences. Table 3 and Figure 18 show the results. We were not able to measure the breakdown of the compilation times for ***compress***, ***db***, and ***jess*** because they were very short.

Table 3: Breakdown of compilation times (seconds)

		Null check optimization	Bound check optimization	Lazy code motion	Others
mtrt	NEW	0.06 (3.03%)	0.05 (2.53%)	0.11 (5.56%)	1.76 (88.89%)
	OLD	0.02 (1.01%)	0.05 (2.53%)	0.11 (5.56%)	1.76 (88.89%)
jess	NEW	0.06 (2.21%)	0.09 (3.32%)	0.11 (4.06%)	2.45 (90.41%)
	OLD	0.02 (0.74%)	0.08 (2.95%)	0.11 (4.06%)	2.45 (90.41%)
compress	**We could not measure the breakdown because the compilation time was very short.**				
db	**We could not measure the breakdown because the compilation time was very short.**				
mpegaudio	**We could not measure the breakdown because the compilation time was very short.**				
jack	NEW	0.04 (3.25%)	0.02 (1.63%)	0.04 (3.25%)	1.13 (91.87%)
	OLD	0.02 (1.63%)	0.02 (1.63%)	0.04 (3.25%)	1.13 (91.87%)
javac	NEW	0.20 (3.38%)	0.12 (2.03%)	0.32 (5.41%)	5.28 (89.19%)
	OLD	0.05 (0.84%)	0.11 (1.86%)	0.32 (5.41%)	5.28 (89.19%)

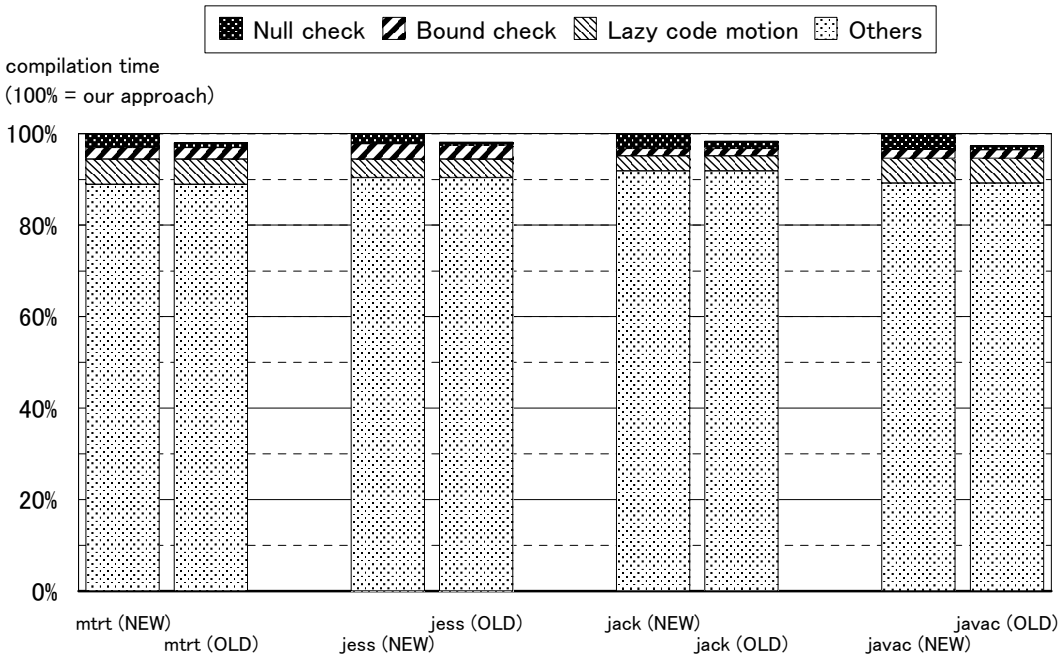

Figure 18: Breakdown of compilation times

Table 4: Increases in compilation time in our approach

	Increase in total compilation time (second)	Increase in total compilation time (%)	Increase in total execution time (%)
mtrt	0.04	2.02%	0.44%
jess	0.05	1.85%	0.46%
jack	0.02	1.63%	0.15%
javac	0.16	2.70%	0.77%

Table 4 shows the increase in the compilation time in our new approach relative to the compilation time in the old one. In summary, three enhancements described in this paper increased the total compilation time by approximately 2.7%.

6. Conclusions and Future Work

In this paper, we have presented a new algorithm for null pointer check elimination, array bound check elimination, and trace-based partial redundancy elimination (enhanced LCM), all of which have been implemented in the latest IBM Java Just-in-Time compiler. Preliminary performance results show a significant performance improvement over previous approaches. There are several areas in which further improvements are needed. First, we need more accurate runtime trace information for conditional branches. Currently, the runtime trace uses only the outcome of the first execution of every conditional branch, whose accuracy is approximately 70% according to our experiments. We can enhance the accuracy by using more samples from several runtime executions. Second, we can further improve the performance by applying runtime trace information to other optimizations, such as exception check elimination. Third, we should run larger collections of application programs to evaluate our new algorithm with respect to the performance improvement and the overhead of the compilation time.

Acknowledgment

We would like to thank the members of the TRL JIT team for helpful discussions and analysis of possible performance improvements.

References

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

[2] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, Vol. 2, Nos. 1–4, pp.135–150, March–December 1993.

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

[4] 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.

[5] 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.

[6] A.V.Aho, R.Sethi, and J.Ullman, *Compilers: Principles, Techniques, and Tools*, Addison–Wesley Publishing Co., Reading, MA(1986).

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

[8] A–R. Adl–Tabatabai, M. Cierniak, G–Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a Just–In–Time Java compiler. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[9] M. G. Burke, J–D. Choi, S. Fink, D. Grove, M. Hind, V. Sarker, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. "The Jalapeño dynamic optimizing compiler for Java," *In Proceedings of the ACM SIGPLAN Java Grande Conference*, June 1999.

[10] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. "Optimizations to reduce overheads of the Java language in a Just–in–Time Java compiler." *In Proceedings of the ACM SIGPLAN Java Grande Conference*, June 1999.

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

[12] R. Gupta. A fresh look at optimizing array bound checking, *In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 272–282, June 1990.

[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] C. Chambers, D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically–typed object–oriented programs. *In Proceedings of the ACM SIGPLAN' 90 Conference on Programming Language Design and Implementation*, pp. 150–164, June 1990.

[15] R. Gupta, D. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*, May 1998.