April 14, 2000
RT0356
Computer Science 19 pages

Research Report

Effective Null Pointer Check Elimination Utilizing Hardware Trap

Motohiro Kawahito, Hideaki Komatsu, and 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

Limited Distribution Notice
This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

Effective Null Pointer Check Elimination Utilizing Hardware Trap

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 a new algorithm for eliminating null pointer checks from programs written in Java™. However, the same approach should work for any languages requiring null checking. Our new algorithm moves null checks backwards and eliminates redundant null checks. This increases the opportunities for other optimizations to be applied by eliminating many null checks that impede code motion. In a separate pass, it moves null checks forwards and converts many null checks to hardware traps in order to minimize the execution costs for the remaining null checks. This algorithm has been implemented in the IBM 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 previously known algorithms.

1. Introduction

The Java language[9] 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. In fact, these operations are quite common in typical Java programs.

In general, the implementation of null checking can take advantage of hardware traps[10, 13, 14]. 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 implementation, null check elimination is still important for two reasons. The first is that null checks become barriers to optimizations even with 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 does not raise an interrupt when the offset of the address is larger than a certain size. As another example, AIX does not raise an interrupt for the case of reading the zero address. A more subtle example is that, when devirtualization[8, 10, 15, 16] 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 as small as a few instructions.

To convert null checks to hardware traps in earlier phases of compilation inhibits other optimizations, since any instruction

capable of throwing an exception becomes a barrier to code motion and thus significantly reduces the scope of optimizations. Our null check optimization is split into two phases in order to solve this issue. In the first phase, as an architecture independent optimization, a partial redundancy elimination algorithm is employed to reduce null checks in order to increase the opportunity for applying other optimizations in a wider region. Previous null check optimization techniques, such as forward data-flow null check elimination, cannot maximize other optimizations' effect, since the null check does not move and remains a barrier to code motion. In the second phase, as an architecture dependent optimization, null checks will be converted to the hardware traps available for the target hardware and operating system in order to minimize the execution cost of null checking. 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 these architectures. We conducted experiments by running jBYTEmark and SPECjvm98 benchmark programs on both a Pentium III 600MHz machine (with Windows NT 4.0 and IBM Developer Kit for Windows(R), Java Technology Edition, Version 1.2.2) and a PowerPC 604e 332 MHz machine (with AIX 4.3.1 and IBM Developer Kit for AIX, Java Technology Edition, Version 1.2.2). 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 effect of other compiler optimizations, unlike previously known algorithms, and yet it can take full advantage of the hardware trap mechanism. Although we have 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 previously known algorithms. It also shows that our approach increases JIT compilation time an average of 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 algorithms. 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[14] from the IBM T.J. Watson Research Center, LaTTe JIT compiler[13], and the previous version of our JIT compiler[8, 10] utilize hardware traps and the associated OS support functions for null check implementation. Jalapeño's object layout was 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. If the pointer to an object is null, Jalapeño accessed the contents of the object using a negative address. The designers relied on the fact that reading from the last page triggered this hardware trap. LaTTe (the target architecture is SPARC) relied on all memory reads or writes causing hardware traps. They assumed that all null checks caused hardware traps. However, such an assumption can not be used in applying some code transformations, such as method inlining by devirtualization. Our JIT compiler for Windows on Intel IA32 partly used such an implementation. Our JIT compiler for AIX on the PowerPC could use such an implementation for the memory writes, but we have not implemented it yet. Instead, we used a conditional trap instruction (it requires only one cycle if it is not taken) to check null pointers for all memory reads and writes, and we applied speculation for memory reads. That is, we moved memory reads speculatively above the original location where the trap instruction is placed to catch the null pointer.

2.2 Null Pointer Check Elimination using Forward Data-Flow Analysis

Previous JIT compilers, such as the Jalapeño compiler[1] or the previous version of our JIT compiler[8, 10], eliminate null pointer checks by using forward data-flow analysis. This algorithm eliminates null checks that have already been checked somewhere along the data-flow. 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, the null check for it must remain in the loop body. Such null checks become barriers to optimizations and thus significantly reduce the scope of optimizations.
- This elimination algorithm does not take into account the utilization of hardware support.

3. Overview of Our Approach

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

3.1 High-level Flow Diagram of Null Check Optimization

We begin by explaining the high-level flow diagram of null check optimization using Figure 1. Our null check optimization is split into two phases; the first is an architecture independent optimization (1), the second is an architecture dependent optimization (5). The architecture independent optimization (1) moves null checks backwards and eliminates redundant null checks. This optimization increases opportunities for array bound checking optimization (2) and scalar replacement (3), and these optimizations also increase opportunities for null checking optimization; therefore some iterations (4) achieve more optimization. An architecture dependent optimization (5) moves null check forwards and minimizes the execution costs of null checking by utilizing the hardware trap.

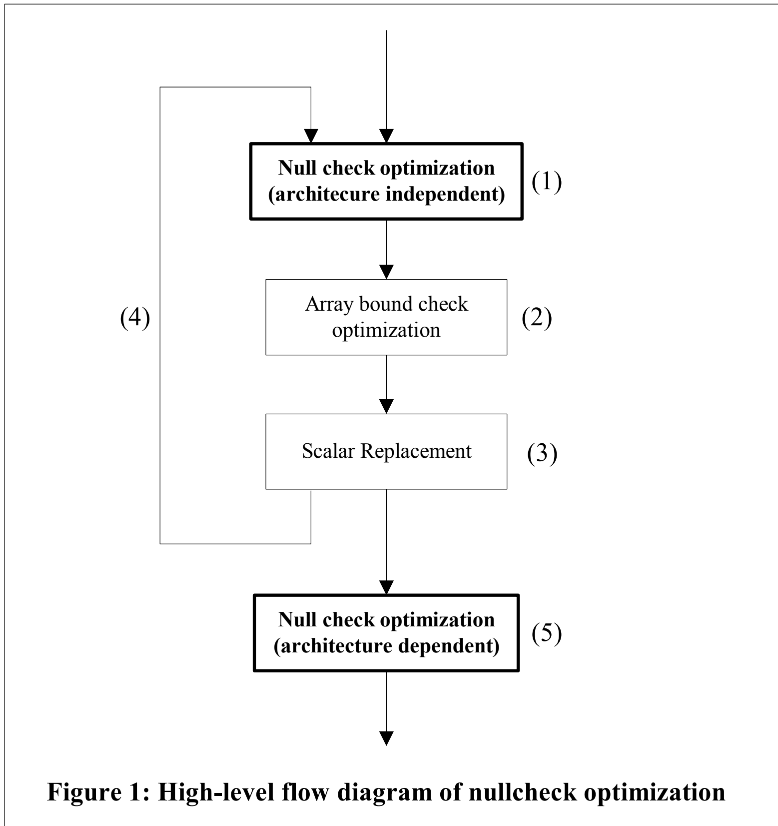


Figure 1: High-level flow diagram of nullcheck optimization

3.2 Architecture Independent Optimization

We enhance partial redundancy elimination (PRE)[2, 3, 4] in order to eliminate partially redundant null checks and also to move null checks out of the loop.

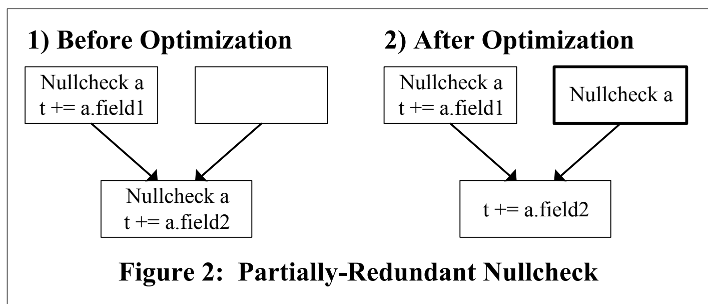


Figure 2: Partially-Redundant Nullcheck

Figure 2 shows an example of partially redundant null checks. In Figure 2 (1), the null check located at the junction 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. This optimization inserts a null check in the right basic block and eliminates the null check at the junction. As a result, a null check will be executed only once along each path.

1) original program	2) Intermediate Representation	3) After Null check Optimization	4) After Scalar Replacement	5) After Null check Optimization	6) After Scalar Replacement

do {
i += a.B.C + a.B.D;
} while(some cond);
	do {
	nullcheck a;
	s1 = load[a.B];
	nullcheck s1;
	s1 = load[s1.C];
	
	nullcheck a;
	s2 = load [a.B];
	nullcheck s2;
	s2 = load [s2.D];
	
	i += s1 + s2;
	} while(some cond);
		nullcheck a;
		do {
		s1 = load [a.B];
		nullcheck s1;
		s1 = load [s1.C];
		
		s2 = load [a.B];
		nullcheck s2;
		s2 = load [s2.D];
		
		i += s1 + s2;
		} while(some cond);
			nullcheck a;
			b = load [a.B];
			do {
			nullcheck b;
			s1 = load [b.C];
			
			nullcheck b;
			s2 = load [b.D];
			
			i += s1 + s2;
			} while(some cond);
				nullcheck a;
				b = load [a.B];
				nullcheck b;
				do {
				s1 = load [b.C];
				
				s2 = load [b.D];
				
				i += s1 + s2;
				} while(some cond);
					nullcheck a;
					b = load [a.B];
					nullcheck b;
					c = load [b.C];
					d = load [b.D];
					do {
					i += c + d;
					} while(some cond);

(assumption : 'a' and 'i' are local variables)

Figure 3: The architecture independent null check optimization and scalar replacement

Figure 3 shows an example where the architecture independent null check optimization and scalar replacement assist each other. The first *"nullcheck a"* in (2) cannot be eliminated by the previous approach 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 the null check becomes a barrier against backward movement of related memory accesses. The result of (5) also cannot be achieved without the scalar replacement in (4), since writes of an object becomes a barrier to moving its null check; therefore these optimizations are particularly effective for loop invariant code motion.

3.3 Architecture Dependent Optimization

We enhance partial redundancy elimination (PRE) in order to reduce the execution costs of null checking by utilizing the 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*s, which do not need to generate actual checking code, but rely on the hardware traps.
- *Explicit Null check*s, which need to generate actual checking code.

To implement null checking, we use implicit null checks wherever possible. However, in some cases we have to use an explicit null check in order to maintain conformance to 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.

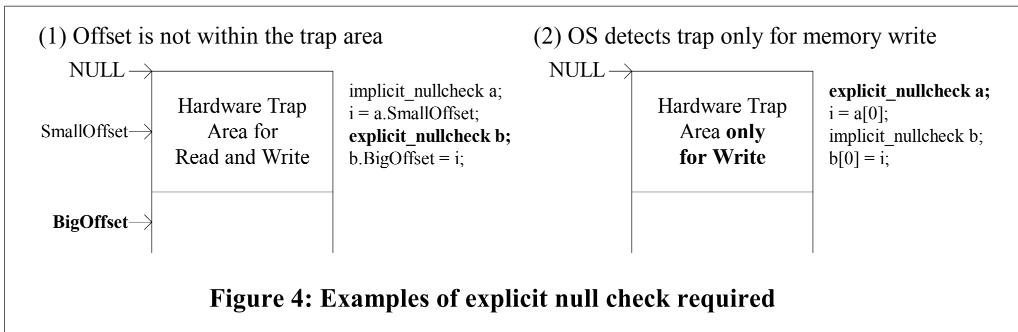


Figure 4: Examples of explicit null check required

For Figure 4 (1), the offset of a memory access is not within the (protected) trap area, so its null check must be an explicit null check. Fortunately, this case is rare for Java language. For any array access in Java, the array length is required for bound 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 it can be as large as 512 KB (which is 65534 * 8 = 524272 based on the Java Virtual Machine Specification[12]).

For Figure 4 (2), if the operating system detects hardware traps only for memory writes to the (protected) trap area, 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 move across its null check speculatively. Furthermore, it can be moved out of the loop as a loop invariant instruction.

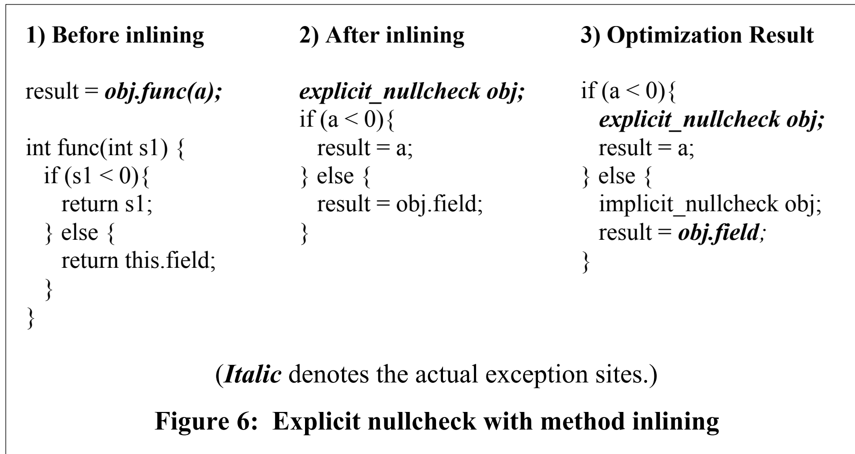
Figure 5 shows an example of such case. In (2), "*nullcheck b*" cannot move across the memory write, "*a.I = S2*." However, if the operating system does not throw the trap for memory reads, "*arraylength b*" can move up across "*nullcheck b*." Finally "*arraylength b*" can be moved out of the loop as shown in (3).

1) original program	**2) Intermediate Representation**	**3) Optimization Result**
do {	do {	explicit_nullcheck a;
total += b[a.I++];	nullcheck a;	i = a.I;
} while(some cond);	s1 = a.I;	**bl = arraylength b;**
	s2 = s1;	do {
	s2 = s2 + 1;	s1 = i;
	nullcheck a;	i = i + 1;
	a.I = s2; // **barrier of nullcheck**	a.I = i;
	nullcheck b;	explicit_nullcheck b;
	s2 = arraylength b;	
	boundcheck s1, s2;	boundcheck s1, bl;
	s1 = b[s1];	s1 = b[s1];
	total += s1;	total += s1;
	} while(some cond);	} while(some cond);

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

Figure 5: An example of speculation

As another example, when method inlining of a dynamic method call is applied to a call site by devirtualization, an explicit null check needs to be generated as shown in Figure 6.

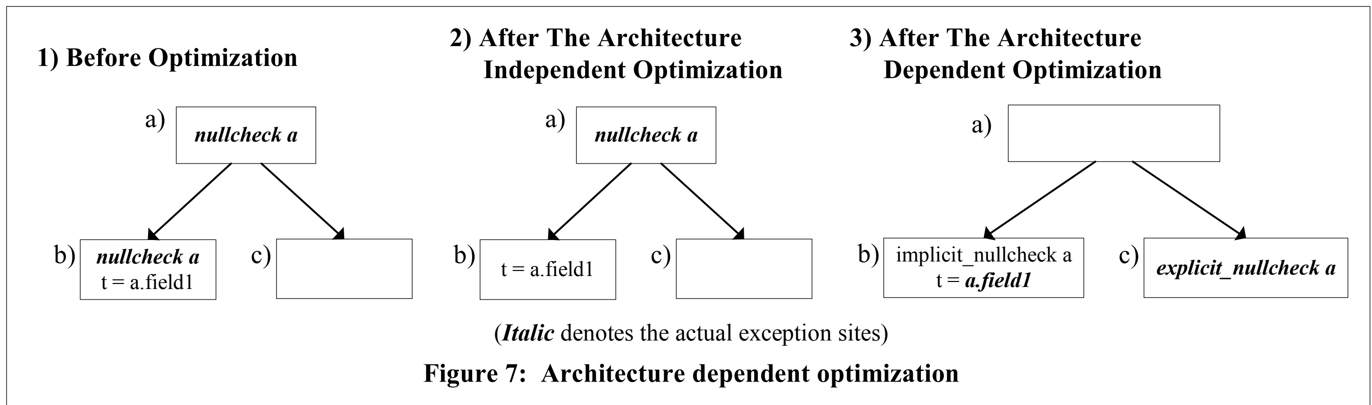


(***Italic*** denotes the actual exception sites.)

Figure 6: Explicit nullcheck with method inlining

If the *explicit_nullcheck* in Figure 6(2) is replaced by an *implicit_nullcheck*, the content of *obj* is not always accessed. Therefore, if *obj* is a null pointer and '*a'* has a negative value, no exception occurs and execution continues. This violates the Java language specification. Explicit null checks with method inlining appear frequently in typical Java programs, and thus, explicit null check elimination is a particularly effective optimization. We can minimize the execution cost for the null check as shown in optimization result (3).

3.3.2 Overview of the Architecture Dependent Optimization



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

Figure 7: Architecture dependent optimization

We explain the algorithm using Figure 7. At the beginning, we treat all null checks as explicit null checks in input code. In the first phase, the architecture independent optimization (as explained in Section 3.2) cannot delete the null check in (a), but only the null check in (b). This is because the null check in (a) is the first point at which the contents of the object are accessed. Therefore, a null check remains in (a). In the second phase, the architecture dependent optimization moves *explicit_nullcheck*s forward. If it reaches an instruction that accesses the content of an object and that is known to cause a hardware trap, then the *explicit_nullcheck* is replaced by an *implicit_nullcheck* (which does not generate actual code). The instruction following an *implicit_nullcheck* 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. As the result, we can get optimization result (3), and can reduce the execution cost along the left path.

As another example, if this optimization is applied against result of Figure 3 (6), all null checks are replaced by implicit null

checks.

4. Outlines of Our Algorithms

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

4.1 Algorithm for Architecture Independent Optimization

4.1.1 Algorithm for Null Check Insertion

The purpose of this stage is to compute the first checkpoints of null checks in the region where null checks can be moved backwards. Out_bwd(n) is the set of movable null checks at the exit point of the basic block. This set is computed by solving the backward data-flow equations given below.

Gen_bwd(n) : The set of movable null checks at the entry of basic block *n*. Only null checks are included in the set.

Kill_bwd(n) : The set of null checks that cannot be moved beyond the basic block *n* in the backward direction. The specific instructions that can be candidates for the kill set are as follows:

- Any instruction that overwrites a variable that is used in null check.
- Any instruction that causes side effects, such as raising other kinds of exceptions or writing to memory.

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

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

Next, Earliest(n) is calculated as the set of the first points for null checks in the region where null checks can be moved backwards. This set is computed by means of the following equation.

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

Earliest(n) has insertion points for null checks at the end of basic block. A null check is not yet inserted at this time, since the next optimization might eliminate some insertion points for null checks.

4.1.2 Algorithm for Null Check Elimination

The purpose of this stage is to eliminate null checks that have already been checked somewhere along the data-flow. In_fwd(n) is the set of available null checks at the entry of the basic block. This set is computed by solving the forward data-flow equations given below.

Gen_fwd(n) : The set of available null checks at the exit of basic block *n*. Both null checks and accesses of the contents of objects are included in the set.

Earliest(n) : The set of null checks that will be inserted at the end of basic block *n*. This set was computed by Section 4.1.1.

Kill_fwd(n) : The set of null checks that cannot be moved beyond the basic block *n* in the forward direction. The specific instruction that can be candidates for the kill set is a definition of a variable used in null check.

Non_null(m, n) : The set of null checks that are known to be non-null by either a succeeding conditional branch or a "this" object in the original Java program at the entry of the basic block *n* from basic block *m*. More specifically, the followings are the candidates for this set:

- null checks on the edge from *m* to *n* can be determined as non-null by *ifnull*, *ifnonnull*, or *instanceof-if<cond>*
- the first basic block for the "this" object for an instance method

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

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

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

Next, we delete unnecessary insertion points from Earliest(n) by computing the following equation, and insert null checks at the end of the basic block from the computed Earliest(n).

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

4.2 Algorithm for Architecture Dependent Optimization

4.2.1 Algorithm for Null Check Insertion

The purpose of this stage is to compute the last points in the region for which each null check can be moved forwards. In_fwd(n) is the set of movable null checks at the entry of the basic block. This set is computed by solving the forward data-flow equations given below.

Gen_fwd(n) : The set of movable null checks at the entry of basic block *n*. Only null checks are included in the set.

Kill(n) : The set of null checks that cannot be moved beyond the basic block *n* in the forward direction. The specific instructions that can be candidates for the kill set are as follows:

- Any instruction that overwrites a variable that is used in null check.
- Any instruction that accesses the contents of an object, where it is known that the access will cause a hardware trap if the object is null.
- Any instruction that causes side effects, such as raising other kinds of exceptions, or writing to memory.

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

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

Next, Latest(n) is calculated as the set of the last points for null checks in the region where the null checks can be moved forwards. This set is computed by means of the following equation.

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

Latest(n) has insertion points for null checks at the entry point of the basic block. The insertion points inside a basic block *n* are determined from Latest(n) by means of the following algorithm.

```
for (each I from the first instruction to the last instruction in the basic block n) {
   if (I is a null check ) {
      C = null check by instruction I;
      Latest(n) = Latest(n) ∪ C;
   } else {
      if (I accesses contents of object && I will cause hardware trap if object is null) {
         C = null check for instruction I;
         if (C ∈ Latest(n)) {
            Insert implicit null check for C before I;    // this transaction is optional.
            Mark I as exception sites;
            Latest(n) = Latest(n) - C;
         }
      }
      if (I might cause other kinds of exceptions || I might write to memory) {
         for (each C ∈ Latest(n)) {
            Insert explicit null check for C before I;
         }
         Latest(n) = ϕ;
      } else if (I overwrites a local variable which has object) {
         C = null check of the local variable;
         if (C ∈ Latest(n)) {
            Insert explicit null check for C before I;
            Latest(n) = Latest(n) - C;
         }
      }
   }
}
for (each C ∈ Latest(n)) {
   Insert explicit null check for C at the end of basic block n;
}
```

4.2.2 Algorithm for Explicit Null Check Elimination

The purpose of this stage is to eliminate explicit null checks wherever possible if the following instructions that require null checks can be used as substitutes for the explicit null checks. Out_bwd(n) is the set of substitutes for null checks at the exit of the basic block. This set is computed by solving the backward data-flow equations given below.

Gen_bwd(n): The set of substitutes for null checks at the entry of basic block *n*. Both null checks and accesses of the contents of objects are included in the set.

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

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

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

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

5. Experimental Results

We chose two benchmark programs for the evaluation of our optimization: jBYTEmark version 0.9 from BYTE Magazine and SPECjvm98[6]. For SPECjvm98, the measurements were performed in test mode (not in SPEC-compliant mode) with a 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 600MHz 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 332MHz with 128 MB of RAM, AIX 4.3.1.

We disabled null check optimization and always generated explicit null checks (that is software checks only) for all null checks (denoted as "No Opt. with Software Check" in Table 1 and Table 2) to show the effectiveness over the enabled null check optimization (denoted as "New Null Check Optimization" in Table 1 and Table 2). To compare the performance improvement over previous approach, we implemented Whaley's algorithm[1] (denoted as "Old Null Check Optimization" in Table 1 and Table 2) for null check elimination. To compare the effectiveness of implicit null checks (that is with the hardware trap) over explicit null checks, we disabled null check optimization and utilized the hardware trap (denoted as "No Opt. with Software Check" in Table 1 and Table 2). To compare with another Java system, we also measured these benchmarks by using a HotSpot™ Server VM 2.0 beta[11] under the same software environment.

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 Decomposition
New Null Check Optimization	201.96	54.41	258.86	219.64	22.75	207.41	67.46	159.33	200.50	205.90
Old Null Check Optimization	160.78	49.87	245.25	186.12	22.74	130.10	63.27	156.08	130.82	158.31
No Opt. with Hardware Trap	157.01	49.58	245.13	170.18	22.74	125.31	63.14	151.88	130.42	119.91
No Opt. with Software Check	156.94	9.08	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 Optimization	6.44	7.67	17.38	24.42	11.32	9.39	14.18
Old Null Check Optimization	7.05	7.86	17.49	24.70	11.33	9.77	14.30
No Opt. with Hardware Trap	7.09	7.95	17.55	24.71	11.39	9.80	14.33
No Opt. with Software Check	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 Optimization: New null check optimization is applied utilizinghardware traps.
Old Null Check Optimization: Use Whaley's algorithm[1] for null check elimination. It utilizeshardware traps.
No Opt. with Hardware Trap: Disable null check optimization. However, implementation of null check utilizes thehardware traps.
No Opt. with Software Check: Disable null check optimization. All null checks are explicit null checks.
HotSpot: HotSpot Server VM 2.0 beta

5.1 Improvement from Our Null Check Elimination

Figure 8 shows the percentage of performance improvement over our baseline (No Opt. with Software Check) achieved for jBYTEmark v.0.9 by the new null check optimization described in section 4. We found that architecture independent optimization is very effective for ***Assignment***, ***Neural Net***, and ***LU Decomposition***. This is because these benchmarks use multidimensional arrays, and therefore null check optimization, array bound check optimization, and scalar replacement assist each other. Some loop invariant array accesses are moved out of loops and performance is greatly improved.

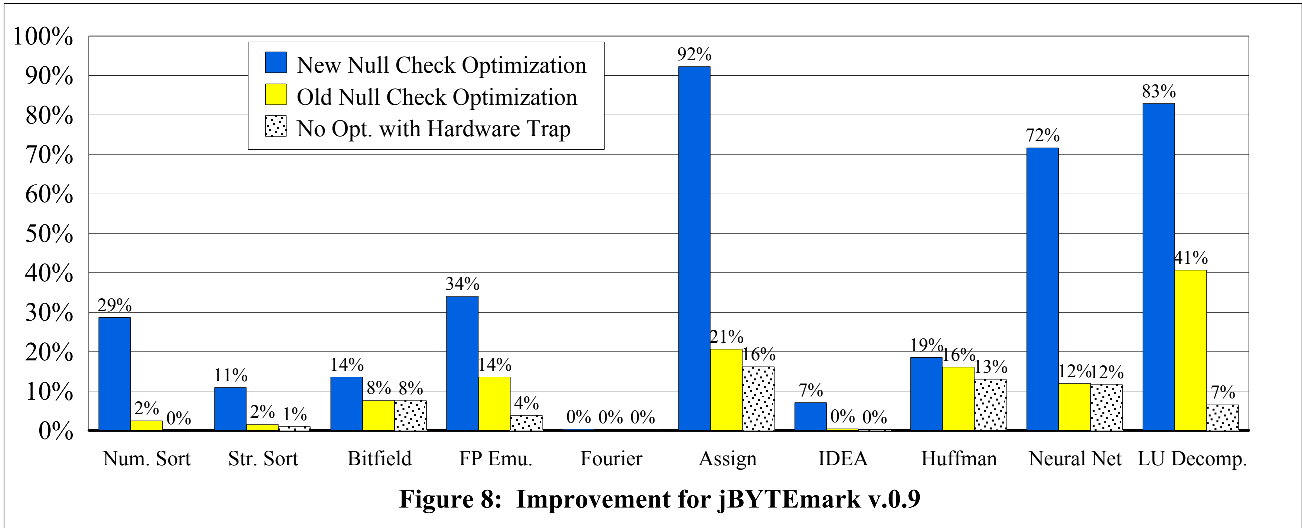


Figure 8: Improvement for jBYTEmark v.0.9

Figure 9 shows the percentage of performance improvement over our baseline (No Opt. with Software Check) achieved for SPECjvm98 by the new null check optimization. We found that our explicit null check elimination 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.

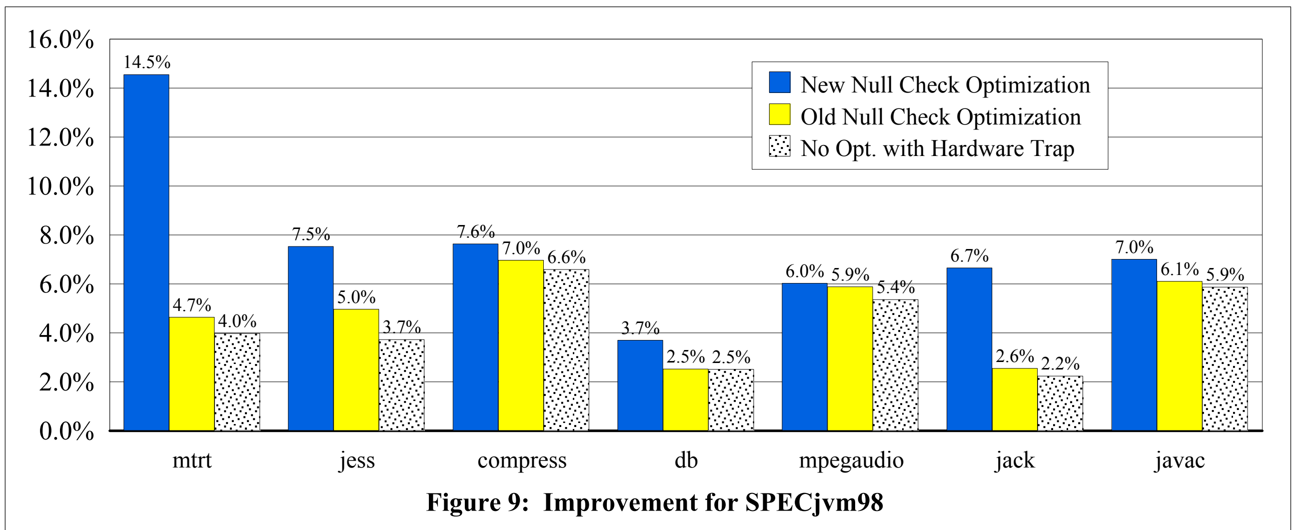


Figure 9: Improvement for SPECjvm98

5.2 Performance Compared with HotSpot

Figure 10 shows the performance comparisons for jBYTEmark, comparing our JIT compiler including new null check optimizations with HotSpot. Our JIT compiler shows better performance for almost of all benchmarks, and the average relative performance of our JIT compiler is 169% better than HotSpot.

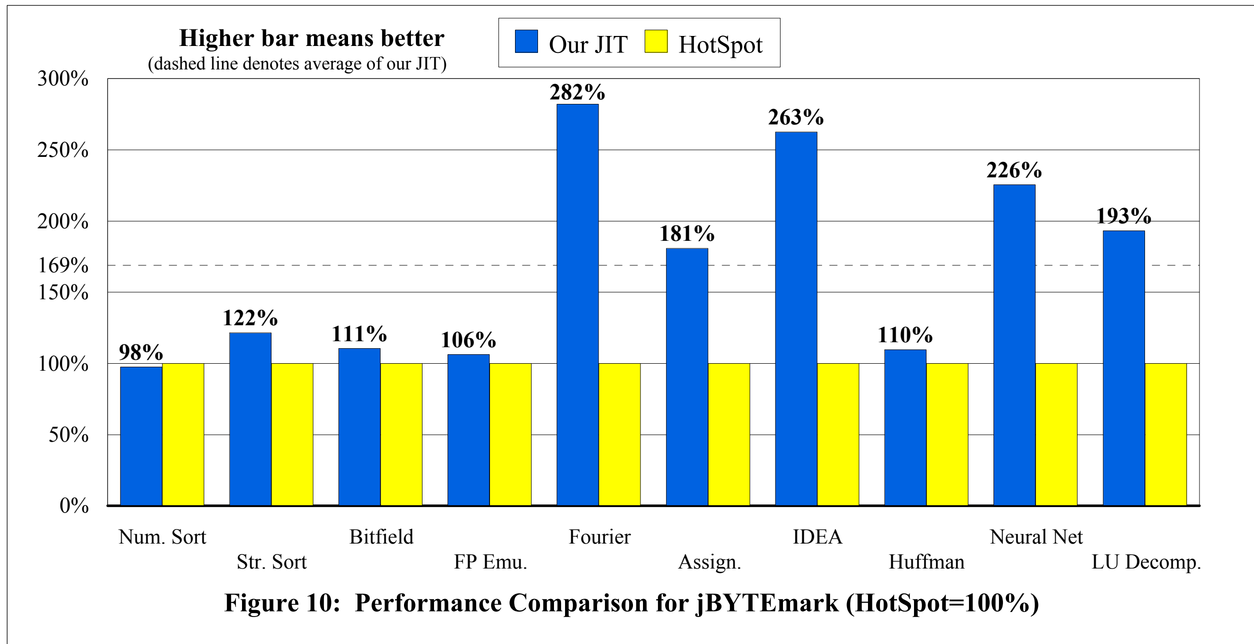


Figure 10: Performance Comparison for jBYTEmark (HotSpot=100%)

Figure 11 shows the performance comparisons for SPECjvm98, comparing our JIT compiler including new null check optimizations with HotSpot. Our JIT compiler shows slightly better performance, since the average relative performance of our JIT compiler is 106% compared to HotSpot.

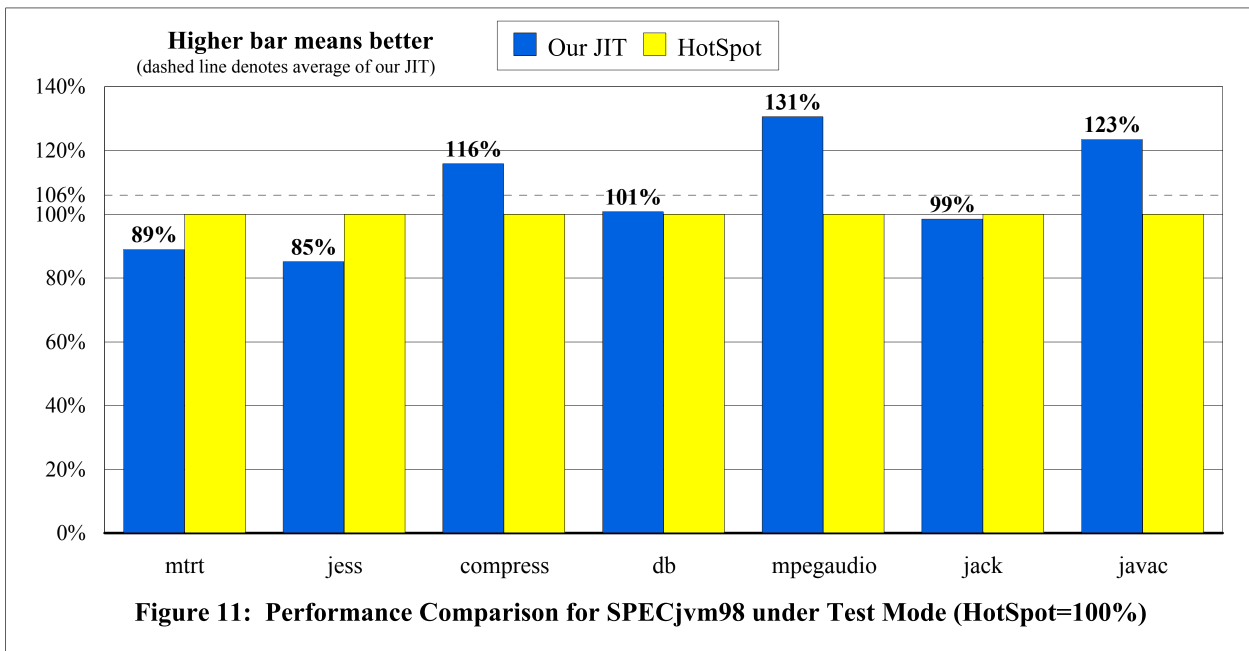


Figure 11: Performance Comparison for SPECjvm98 under Test Mode (HotSpot=100%)

5.3 JIT Compilation Time

In this section, we compare the JIT compilation time in our approach with that of the previous approach. We could not measure compilation time for jBYTEmark because it was very short. We assume that the difference between the first run and the best run of SPECjvm98 is essentially due to compilation time. Table 3 shows the time for the first run, best run, and the assumed compilation time of our JIT compiler and HotSpot. This shows that our JIT compiler performs very fast compilation when compared with HotSpot, if our assumption about compilation time is correct. Figure 12 shows the percentage of the our JIT compilation time over the whole execution time (that is, the time spent for the first run). In summary, ***javac*** is the benchmark that the JIT compiler took the most time to compile.

Table 3: JIT compilation time (seconds)

		mtrt	jess	compress	db	mpegaudio	jack	javac
Our JIT	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%)

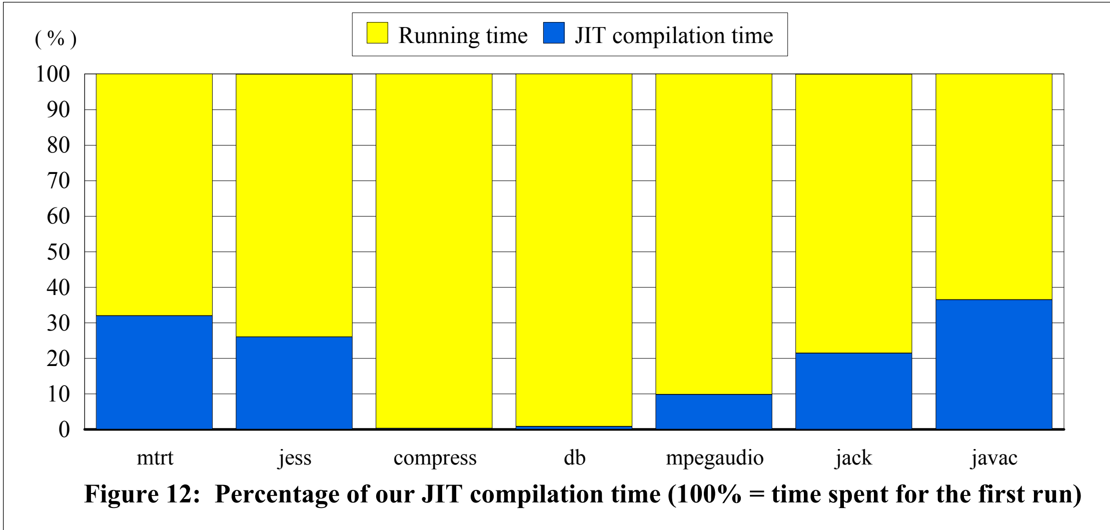


Figure 12: Percentage of our JIT compilation time (100% = time spent for the first run)

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 were not able to measure the breakdown of the compilation times for ***compress***, ***db***, and ***jess*** because they were very short. The new null check optimization itself takes about 3 times longer in compilation time than the old one, and it also slightly increases the compilation time of other optimizations. This is because the new null check optimization also increases opportunities for other optimizations, such as scalar replacement, to be applied.

Table 4: Breakdown of our JIT compilation times (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%)
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	0.06 (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%)

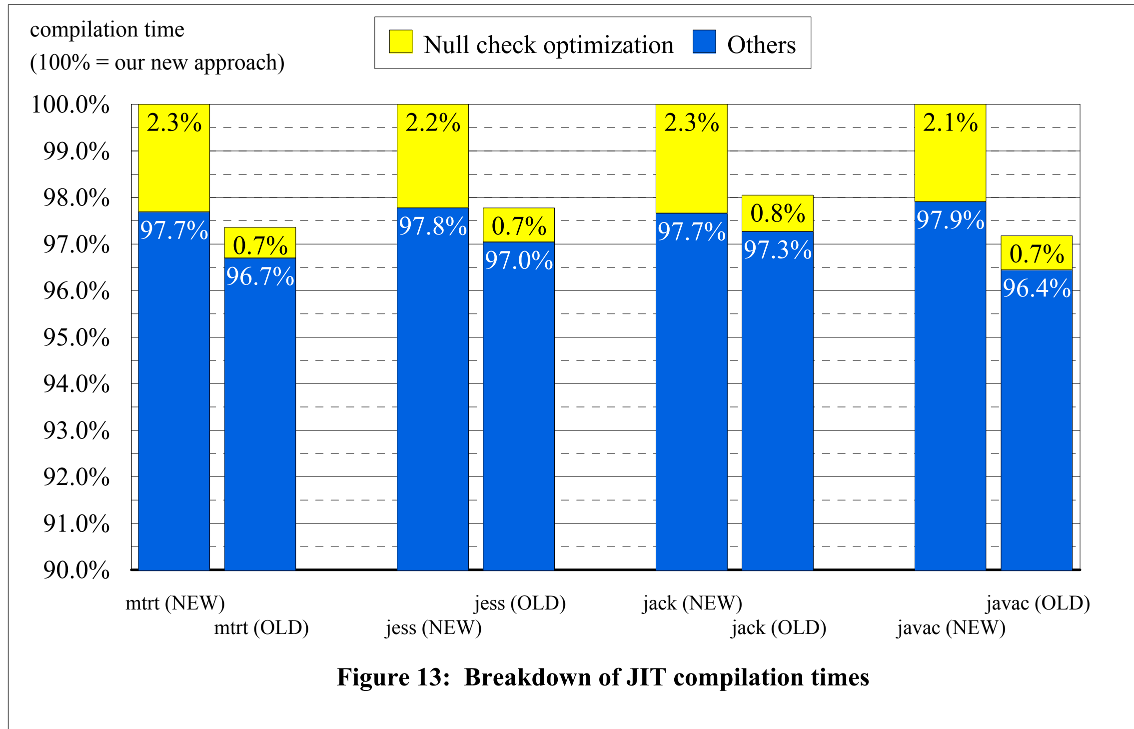


Figure 13: Breakdown of JIT compilation times

Table 5 shows the increase in the compilation time in our new approach relative to the compilation time in the old one. In summary, the enhancements described in this paper increased the total compilation time by approximately 2.3% (on average).

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%
jack	0.05	1.95%
javac	0.23	2.82%

5.4 Relative Effectiveness of Speculation versus Implicit Null Check

As we described in Section 2.1, we have used 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 compare 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 "Implicit Null Check" in Table 6 and Table 7), we applied the architecture dependent null check optimizations for Intel to the optimizations for AIX. We note here that this violates the Java language specification since a NullPointerException may not be thrown correctly on AIX. This is purely for experimentation purposes. To find the effectiveness of the overall performance gain over the baseline (denoted as "No Opt. with Software Check" in Table 6 and Table 7), we disabled the whole phase of our null check optimization.

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
Implicit Null Check	183.28	29.91	84.40	86.62	13.25	95.66	45.60	100.74	77.35	92.66
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 Opt. with Software Check	173.92	28.17	83.42	79.89	13.23	81.71	44.68	97.14	73.93	79.98

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

(unit : sec)	mtrt	jess	compress	db	mpegaudio	jack	javac
Implicit Null Check	19.94	26.09	43.75	71.86	19.87	44.71	46.90
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 Opt. with Software Check	21.00	26.28	44.25	72.85	20.42	45.36	47.34

Implicit Null Check: Apply architecture dependent optimizations for Intel, assuming memory accesses cause hardware traps.
Speculation: Enable speculation as our current JIT compiler including new null check optimization. We use conditional trap in order to check null.
No Speculation: Disable speculation in our current JIT compiler.
No Opt. with Software Check: Disable null check optimization. All null checks are software checks.

Figure 14 shows the percentage of performance improvement over our baseline (No Opt. with Software Check) achieved 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 its null checks) with speculation, and this made a significant improvement.

Implicit null checking for ***Neural Net*** is least effective among all the benchmarks when the results are compared on the Intel platform. This is because the input code to the null check optimization on the PowerPC was different. On the Intel platform, the method java.lang.Math.exp was inlined, but it was not inlined on the PowerPC. This is because our JIT compiler for Intel converts this method to an exponential instruction. However, our JIT compiler for the PowerPC does not, 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 for the PowerPC relative to other benchmarks.

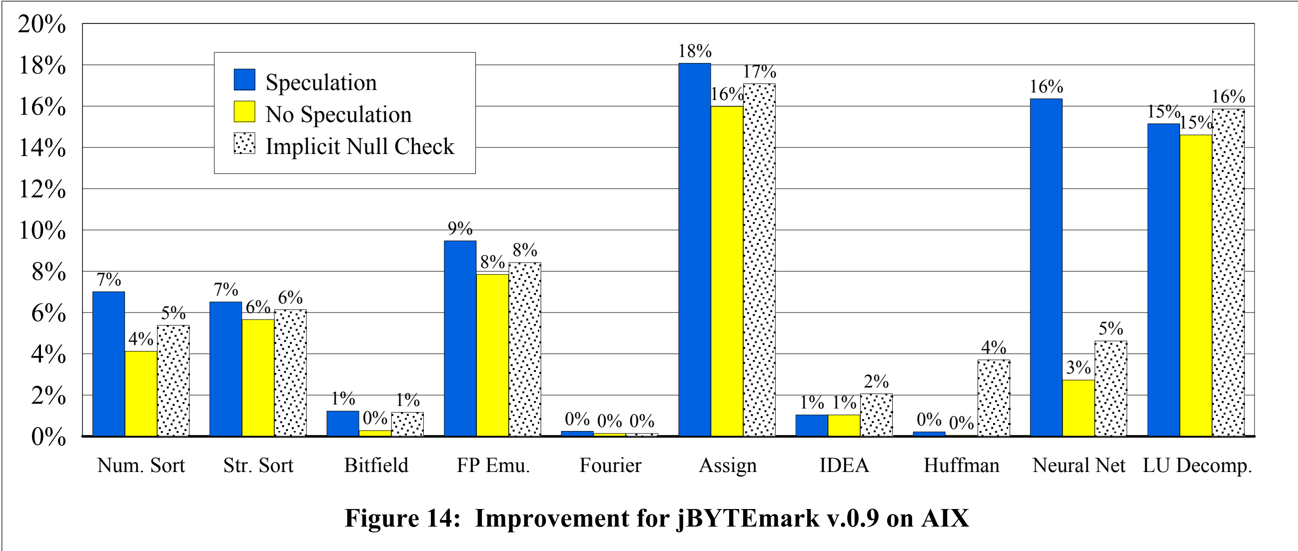


Figure 14: Improvement for jBYTEmark v.0.9 on AIX

Figure 15 shows the percentage of performance improvement over our baseline (No Opt. with Software Check) achieved for SPECjvm98. It has been noticed that implicit null checks are especially effective for ***mtrt***. This is also true for Intel, though the improvement is smaller on the PowerPC platform than for the Intel platform. This is because the original execution costs for null checks (using conditional trap) are smaller than that on the Intel platforms.

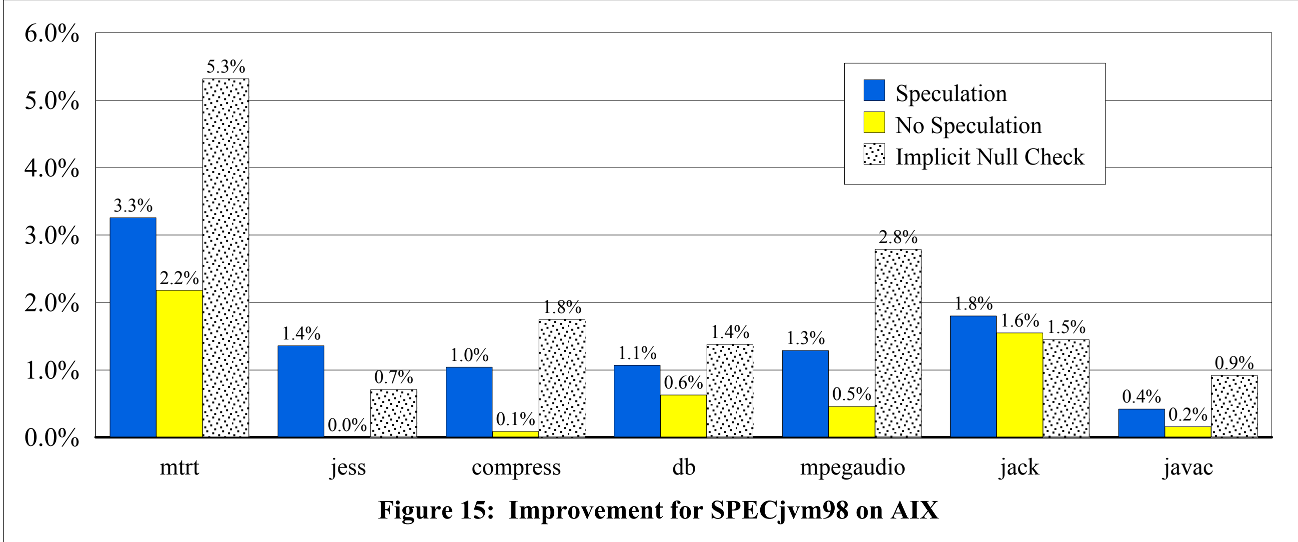


Figure 15: Improvement for SPECjvm98 on AIX

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. An architecture independent optimization moves null checks backwards and eliminates redundant null checks. This optimization maximizes other optimizations' effectiveness. Then an 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 previous approaches. Our algorithm can apply not only for Java, but also for other languages requiring null checks. We expect the importance of the techniques presented in this paper to grow further.

Acknowledgment

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

References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.