

# Effective Sign Extension Elimination

Motohiro Kawahito

Hideaki Komatsu

Toshio Nakatani

IBM Tokyo Research Laboratory

1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan

{ j125131, komatsu, nakatani }@jp.ibm.com

## ABSTRACT

Computer designs are shifting from 32-bit architectures to 64-bit architectures, while most of the programs available today are still designed for 32-bit architectures. Java™, for example, specifies the frequently used “int” as a 32-bit data type. If such Java programs are executed on a 64-bit architecture, many 32-bit values must be sign-extended to 64-bit values for integer operations. This causes serious performance overhead. In this paper, we present a fast and effective algorithm for eliminating sign extensions. We implemented this algorithm in the IBM Java Just-in-Time (JIT) compiler for IA-64™. Our experimental results show that our algorithm effectively eliminates the majority of sign extensions. They also show that it significantly improves performance, while it increases JIT compilation time by only 0.11%. We implemented our algorithm for programs in Java, but it can be applied to any language requiring sign extensions.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – compilers, optimization.

## General Terms

Algorithms, Performance, Design, Experimentation

## Keywords

Sign extension, 64-bit architectures, Java, JIT compilers, IA-64

## 1. INTRODUCTION

When a program is compiled, values whose size is defined to be smaller than the architectural register size must be adjusted to the register size. For example, on a 64-bit architecture, values defined as signed 8, 16, and 32-bit values in a program must be sign-extended to make them 64-bit values (Figure 1). Today, many systems and applications are still designed for 32-bit architectures. For example, Java specifies the frequently used “int” as a 32-bit data type [5]. If such programs are executed on a 64-bit architecture, 32-bit values must be sign-extended to 64-bit values for many integer instructions. This will cause serious performance degradation.

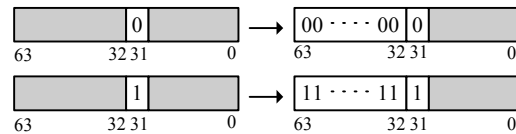
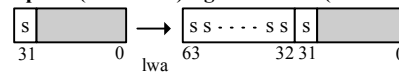


Figure 1. Sign extension of a 32-bit value to a 64-bit value

### 1) Implicit (automatic) sign extension (PPC64: *lwa*)



(Load Word Algebraic Instruction)

### 2) Explicit sign extension (PPC64: *exts*, IA64: *sxt*)

```
i = mem;          - (1)
i = i + 1;        - (2)
i = extend(i);    - (3) // explicit sign extension is required
t = (double) i;  - (4) // i must be sign-extended
```

(*extend()* denotes a sign extension instruction from 32-bit to 64-bit)

Figure 2. Two types of sign extension

Some 64-bit architectures have an instruction that reads from memory and extends the sign in the same instruction automatically. We call this “implicit sign extension.” For example, the PowerPC architecture [8] has such an instruction, called the *load word algebraic (lwa)* instruction (Figure 2(1)). Other 64-bit architectures have no such instruction. For example, on IA-64, values are zero-extended during memory reads, and thus sign extension instructions are required. Sign extension elimination is important for these architectures.

Even for those architectures that have an implicit sign extension instruction, when the register size and the size of the values defined in a program are different, sign extensions are required during calculations. For example, in Figure 2(2), since *i* must be sign-extended on statement (4), a sign extension instruction (statement (3)) is required in addition to the implicit sign extension instruction (statement (1)). We call this an “explicit sign extension” and denote it as *extend()*. For example, the PowerPC architecture has such an instruction, called the *extend sign (exts)* instruction. The IA-64 architecture [9] also has such an instruction, called the *sign extend (sxt)* instruction.

In principle, a sign extension instruction can be eliminated if its source operand is already sign-extended or if the upper 32 bits of its destination operand do not affect the correct execution of the succeeding instructions [3]. We implemented the first algorithm for sign extension elimination by using backward dataflow analysis. This algorithm first generates a sign extension instruction immediately following every instruction *I* with a 32-bit destination operand unless the destination operand of the instruction *I* is guaranteed to be sign-extended. Next, it eliminates a sign extension instruction if the backward dataflow analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'02, June 17-19, 2002, Berlin, Germany.

Copyright 2002 ACM 1-58113-463-0/02/0006...\$5.00.

```

int j; // j is a 32-bit variable.
int t = 0; // t is a 32-bit variable.
int i = mem; // i is a 32-bit variable.
int C = 0xffffffff; // C is a 32-bit variable.
i = extend(i);           - (1) (can be eliminated)
do {
  i = i - 1;             - (2)
  i = extend(i);        - (3)
  j = a[i];              - (4)
  j = extend(j);        - (5) (can be eliminated)
  j = j & C;             - (6)
  j = extend(j);        - (7) (can be eliminated)
  t += j;                - (8)
  t = extend(t);        - (9)
} while(i > start);
// need sign extension for t
d = (double) t;         - (10)

```

**Figure 3. Limitations of the first algorithm**

proves that the upper 32 bits of the destination operand do not affect the correct execution of the following instructions. When we applied this algorithm to the example shown in **Figure 3**, we can only eliminate the sign extensions (1), (5), and (7)<sup>1</sup>. We found the following four limitations of this algorithm:

The first limitation is that a sign extension for an array index (e.g. (3) in Figure 3) cannot be eliminated using this algorithm. This is because an effective address computation for an array access requires a sign extension. **Figure 4** shows examples of effective address computations on IA-64 and PPC64. For IA-64 (Figure 4 (b)), if the sign extension can be eliminated, an effective address can be computed using one instruction, called the *shift left and add (shladd)* instruction. Array accesses often appear inside a loop, and thus leaving these sign extension instructions in the loop causes major performance degradation. Additionally, when an array index in a loop is not loop invariant, loop invariant code motion techniques cannot move a sign extension for that array index out of the loop. On the other hand, we can utilize an instruction that computes an effective address without an explicit

**(a) Original program**

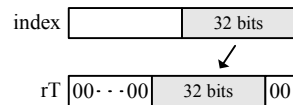
```
base[index] = 0;           # array access
```

**(b) IA-64 architecture**

```
sxt4   index = index      # sign extension
shladd rEA = index, 2, base # shift and add : (index << 2) + base
st4    [rEA] = 0          # memory write
```

**(c) PowerPC64 architecture**

```
addi   rS, 0, 0          # rS = 0
rldic  rT, index, 2, 30  # shift and clear
```



```
stwx   rS, base, rT      # memory write : [base+rT] = rS
```

**Figure 4. Examples of effective address computations**

<sup>1</sup> Statement (10) requires a sign extension, and thus (9) cannot be eliminated. Statements (8) and (6) do not require any sign extension because the upper 32 bits of their source operands do not affect their correct execution, and thus (7) and (5) can be eliminated, respectively. Statement (4) again requires a sign extension, and thus (3) cannot be eliminated. Finally, since (2) does not require a sign extension, (1) can be eliminated.

sign extension on some architectures. For example, PPC64 has such an instruction, called the *rotate left doubleword immediate then clear (rldic)* instruction. If the *index* never has a negative value according to the language specification, we can utilize this instruction to compute an effective address. We will discuss this assumption in Section 3. For another example, on AMD’s SledgeHammer [2], values are zero-extended during 32-bit operations, and thus we can simply use the 64-bit *shift-left (shl)* instruction to compute an effective address without an explicit sign extension.

The second limitation is that elimination using only backward dataflow analysis may miss some opportunities for eliminating sign extensions. In Figure 3, when (8) “t += j” is replaced by an instruction requiring a sign extension for *j*, such as “d += (double) j”, (7) is no longer eliminated by this algorithm.

The third limitation is that a sign extension inside a loop may fail to be eliminated using this algorithm if there are sign extension instructions both inside and outside of the loop for the same variable. This is because elimination using backward dataflow analysis leaves the last sign extension of the affected variable in the flow graph. In Figure 3, for example, there are two sign extensions for the variable *i*, (1) and (3). It is better to eliminate (3) since it is in the loop, but this algorithm results in eliminating (1).

The fourth limitation is that a sign extension inside a loop may fail to be eliminated using this algorithm even when that sign extension instruction can be moved out of the loop. For example, in Figure 3, the sign extension (9) is not required inside the loop, but only before (10) outside of the loop.

We present a new algorithm solving these problems. Our approach has the following characteristics:

- It eliminates sign extension for the effective address computation of an array access based on our assumption that a negative array index is not allowed by the language specification.
- It eliminates sign extensions selectively, starting with the most frequently executed region.
- It utilizes UD/DU chains [1] for the above two goals.
- It inserts sign extensions before elimination. A combination of insertion and elimination can effectively move sign extensions to less frequently executed regions, and particularly out of loops.

We implemented our algorithm in our production-level Java Just-in-Time (JIT) compiler for IA-64. In order to improve portability of our JIT compiler, many optimizations [10, 11, 12, 16, 19, 20] are performed at the intermediate language level. By porting these optimizations to the IA-64 JIT compiler, we achieved high performance quickly. We measured the effectiveness of our algorithm using jBYTEmark and SPECjvm98, both on an IBM IntelliStation Z Pro with two Intel Itanium™ processors. Our experimental results show our algorithm can effectively eliminate the majority of sign extensions. They also show that it significantly improves performance, while increasing the JIT compilation time by only 0.11%.

<sup>TM</sup> Java is a trademark of Sun Microsystems. IA-64 and Itanium are trademarks of Intel Corporation.

The rest of the paper is organized as follows. Section 2 gives an overview of our approach. Section 3 describes the sign extension elimination for array indices. Section 4 shows the performance results obtained in our experiments. Section 5 summarizes previous work. Section 6 offers some concluding remarks.

## 2. OUR APPROACH

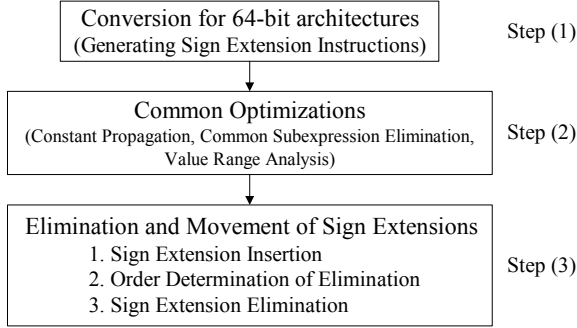


Figure 5. Flow diagram of our algorithm

Figure 5 shows a flow diagram of our algorithm for sign extension elimination, which consists of three steps. Step (1) translates the intermediate representation of the target program from a 32-bit architecture form to a 64-bit architecture form. There are two approaches to generate sign extension instructions. One is to generate a sign extension instruction at a definition point, that is immediately following every instruction  $I$  with a 32-bit destination operand, unless the destination operand of the instruction  $I$  is guaranteed to be sign-extended. The other is to generate a sign extension instruction at a use point, that is immediately before every instruction  $I$  that requires a sign extension, unless the source operand of the instruction  $I$  is guaranteed to be sign-extended. We use the first approach in order to most effectively optimize sign extensions. Figure 6 is an example to show these two approaches. In this example, if the compiler generates a sign extension at a use point as in (c), no sign extension can be eliminated. In contrast, if the compiler generates a sign extension at a definition point as in (b), one sign extension can be eliminated. (See Section 3 for the theorems that justify this elimination.)

Step (2) also optimizes sign extensions. For example, when a constant is propagated as the source operand of a sign extension,

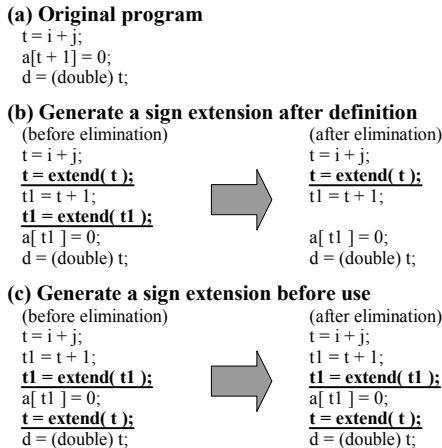


Figure 6. Two approaches to generate sign extensions

the sign extension will be changed to a copy instruction by constant folding. Common sub-expression elimination is also applied to sign extensions. Here we employ a variant of the partial redundancy elimination algorithm [12, 13, 14] for common sub-expression elimination. This optimization moves an expression backward in the control flow graph, and thus loop-invariant sign extensions can be moved out of the loop. A value range analysis [4, 7], used for some optimizations such as array bounds check elimination, also helps sign extension elimination.

Step (3) is the main part of our algorithm, and it has three phases. In the first phase ((3)-1), a sign extension is inserted immediately before every instruction that requires a sign extension. In the second phase ((3)-2), order determination is performed to eliminate sign extensions selectively beginning from the most frequently executed region. Finally, in the third phase ((3)-3), redundant sign extensions are eliminated using UD/DU chains. The following three sections describe each phase.

### 2.1 Sign Extension Insertion

In the first phase, we insert two kinds of sign extensions. To eliminate sign extensions effectively from loops, we insert sign extension instructions. In the example of Figure 7, (10) is the only instruction that requires a sign extension for  $t$ . If sign extension elimination is applied here without insertion, the sign extension (9) will still remain in the loop as shown in Figure 8(a). To avoid this inefficiency, we insert a sign extension instruction immediately before every instruction where sign extension is necessary unless its variable is obviously sign-extended. To balance

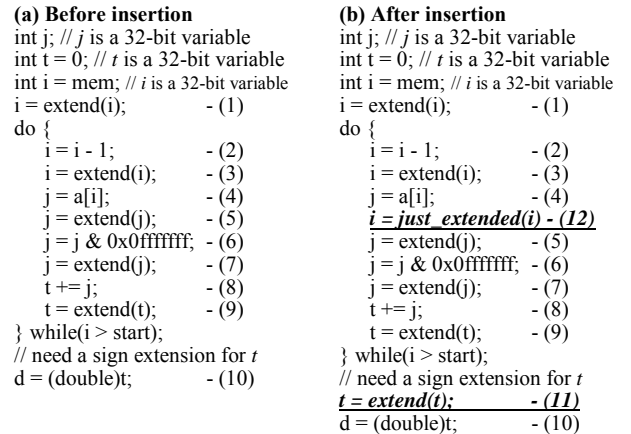


Figure 7. Example of inserting a sign extension

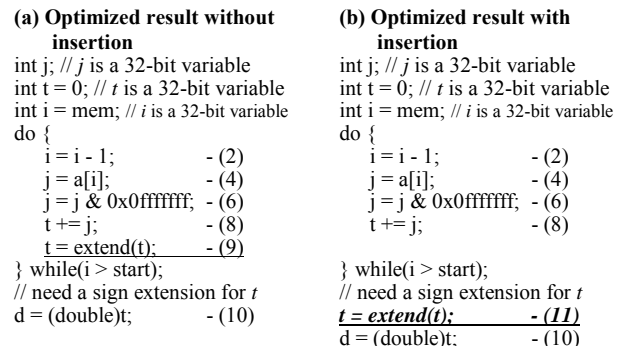


Figure 8. The optimized result of Figure 7

compilation time and effectiveness, we apply this insertion only to those methods that include a loop.

We tried another insertion algorithm that is a variant of the partial dead code elimination (PDE) algorithm [15]. This algorithm inserts a sign extension at the latest point on every possible path where each sign extension can be reached when it is moved forward in the control flow graph. However, the simple insertion algorithm turned out to work better than this algorithm, as shown in Figure 12 and Figure 13 (“all, using PDE” vs. “new algorithm (all)”), and therefore we decided to use the simple insertion algorithm.

We also insert a dummy sign extension instruction just after every array access to indicate that it is guaranteed to be sign-extended, unless an array index is overwritten immediately, as in the case of “ $i = a[i]$ ”. Dummy sign extension instructions are used to eliminate other sign extension instructions, and they will then be eliminated after the elimination in Section 2.3 is performed.

In Figure 7(b), this transformation inserts both a sign extension at (11) and a dummy sign extension (denoted as *just\_extended*( )) at (12). After the sign extension elimination (Figure 8), all the sign extensions except for (11) can be successfully eliminated as shown in Figure 8(b).

## 2.2 Order Determination for Elimination

It is best to eliminate sign extensions starting from the most frequently executed region. When there are sign extensions both inside and outside of a loop, some sign extensions inside the loop may not be eliminated if the sign extensions outside the loop are eliminated first. In Figure 7(b), if we eliminate the sign extension (11) first, (9) will still be required. To take another example, there are two candidates for elimination in Figure 9(a). In this example, while only one of them can be eliminated, Result 1 in (b) is obviously better than Result 2 in (c). To achieve the results of Figure 8(b) and Figure 9(b), we eliminate sign extensions starting from the most frequently executed regions. We sort basic blocks in the order of their execution frequency. For each basic block  $B$ , this can be estimated from both the loop nesting level of  $B$  and the execution frequency of  $B$  within its acyclic region based on the probability of each conditional branch [21]. Additionally, we use profile information collected for conditional branches by our combined interpreter and dynamic compiler [20] in order to enhance the accuracy of branch probabilities. The interpreter gathers statistical data on conditional branches. When the interpreter finds that a method is executed frequently, the dynamic compiler is called. At that time, the interpreter provides the statistical data to the dynamic compiler.

(a) Before elimination	(b) Result 1	(c) Result 2
$i = j + k;$	$i = j + k;$	$i = j + k;$
<u><math>i = \text{extend}(i);</math></u>	<u><math>i = \text{extend}(i);</math></u>	
do {	do {	do {
$i = i + 1;$	$i = i + 1;$	$i = i + 1;$
<u><math>i = \text{extend}(i);</math></u>		<u><math>i = \text{extend}(i);</math></u>
$a[i] = 0;$	$a[i] = 0;$	$a[i] = 0;$
} while( $i < \text{end}$ );	} while( $i < \text{end}$ );	} while( $i < \text{end}$ );

Figure 9. Example requiring order determination

## 2.3 Sign Extension Elimination

The goal of this optimization is to analyze and eliminate each sign extension starting from the most frequently executed region, de-

termined as described in Section 2.2. As mentioned in Section 1, our algorithm is very powerful for eliminating sign extensions for array subscripts. We will discuss this distinguished feature in the next section, while here we focus on basic cases of eliminations.

The algorithm `EliminateOneExtend` analyzes and eliminates one sign extension by using UD/DU chains. We assume that each instruction has three flags, USE, DEF, and ARRAY, to indicate that the instruction has been traversed for each check. We note here that 8-bit and 16-bit sign extensions are also eliminated based on the same algorithm, although we describe the algorithm only for eliminating 32-bit sign extensions in this section.

```

EliminateOneExtend(EXT) {
  initialize all flags (USE,DEF,ARRAY) for all instructions;
  required = FALSE;
  /* use DU-chain */
  for (I ∈ all instructions that use the destination operand of EXT) {
    required = AnalyzeUSE(EXT, I, TRUE);
    if (required) break;
  }
  if (required) {
    /* use UD-chain */
    for (I ∈ all instructions that define the source operand of EXT) {
      required = AnalyzeDEF(I);
      if (required) break;
    }
  }
  if (!required) eliminate EXT;
}

```

In principle, a sign extension can be eliminated if its source operand is already sign-extended or if the upper 32 bits of its destination operand do not affect the correct execution of the following instructions. The algorithm `AnalyzeDEF` checks the first condition. An example of Case 1 is a bit-wise “AND” instruction where either operand is known to have a positive value. An example of Case 2 is a copy operation.

```

/* if it returns FALSE, sign extension is unnecessary for I
   if it returns TRUE, sign extension is necessary for I */
AnalyzeDEF(I) {
  if (a flag of DEF for I has been already set) return FALSE;
  set a flag of DEF for I;
  switch(I) {
  case The destination operand of I is known to be sign-extended:
    /* Case 1 */
    return FALSE;

  case The destination operand of I can be determined to be
    sign-extended if the source operand of I is sign-extended:
    /* Case 2 */
    /* use UD-chain */
    for (J ∈ all instructions that define the source operand of I) {
      if (AnalyzeDEF(J)) return TRUE;
    }
    return FALSE;
  }
  return TRUE;
}

```

The following algorithm `AnalyzeUSE` checks the second condition. An example of Case 1 is a 32-bit memory write operation. An example of Case 2 is an addition. We will explain the algorithm `AnalyzeARRAY` in the next section.

This phase of sign extension elimination ends with one trivial operation that eliminates all the dummy sign extensions.

```

/* if it returns FALSE, sign extension is unnecessary for I
if it returns TRUE, sign extension is necessary for I */
AnalyzeUSE(EXT, I, ANALYZE_ARRAY) {
if (a flag of USE for I has already been set) return FALSE;
set a flag of USE for I;
switch(I){
case The upper 32 bits of the source operand do not affect I:
/* Case 1 */
return FALSE;

case I computes an effective address of an array:
if (ANALYZE_ARRAY) return AnalyzeARRAY(EXT, I);
break;

case The source operand of I can be determined to be unnecessary
if the destination operand of I is determined to be unnecessary:
/* Case 2 */
if (it is impossible to analyze array's address computation via I){
ANALYZE_ARRAY = FALSE;
}
/* use DU-chain */
for (J ∈ all instructions that use the destination operand of I){
if (AnalyzeUSE(EXT, J, ANALYZE_ARRAY) ) return TRUE;
}
return FALSE;
}
return TRUE;
}

```

### 3. HANDLING OF ARRAY SUBSCRIPTS

The most serious problem with our first algorithm is that a sign extension for an effective address computation of an array access cannot be eliminated. Here, we observed that these sign extensions could be eliminated if we know that an array cannot be accessed with a negative array index. This is true for Java since Java programs throw an `ArrayIndexOutOfBoundsException` if an array is accessed using a negative array index [5]. Note that the implementation of array bounds checking may require a sign extension. If the target architecture has 32-bit compare (including trap) instructions to compare only the lower 32 bits of registers, we can implement array bounds checking without any sign extension. This is because 32-bit compare instructions ignore the upper 32 bits of registers. For example, SledgeHammer [2], PPC64 [8], and IA-64 [9] have such an instruction. Based on this language specification (LS), the following predicate will hold for a subscript expression  $e$ .

$$\text{LS}(e) \equiv 0 \leq \text{lower 32 bits of } e \leq 0x7fffffff$$

In this section, we discuss preconditions for  $e$  to satisfy  $0 \leq e \leq 0x7fffffff$  by using  $\text{LS}(e)$ . In general, if the language specification rules out any array access with a negative array index, then the following four theorems can be derived. Note that the theorems assume 64-bit wrap-around on overflow. These theorems depend on the knowledge of the value range, which can be determined at compile time using one of the value range analysis techniques (Figure 5(2)). We use these theorems to effectively eliminate sign extensions for array indices.

#### Theorem 1

If a variable  $i$  satisfies the following two conditions,  $i$  does not need a 32-bit sign extension for the effective address computation of an array access whose subscript expression is  $i$ .

1. The upper 32 bits of  $i$  are initialized to zero.
2.  $\text{LS}(i)$  holds. <sup>†</sup>

**Proof.** Because of the two conditions,  $0 \leq i \leq 0x7fffffff$  will always hold. Thus,  $i$  does not need a 32-bit sign extension.  $\square$

#### (a) Original program

```

index = obj.field;          # memory read
base[index] = 0;           # array access

```

#### (b) IA-64 Instructions

```

ld4      index = [obj.field]    # upper 32 bits are cleared
sxt4     index = index         # can be eliminated
cmp4.ltu.unc p0, p6 = index, base.Len # array bounds check
(p6) br.spnt.many ExceptionHandler # LS( index ) holds
shladd   rEA = index, 2, base   # shift and add
st4      [rEA] = 0             # memory writes

```

Figure 10. An example of Theorem 1

Figure 10 shows an example of Theorem 1. On IA-64, since a value is zero-extended during any memory read, the upper 32 bits of  $index$  are cleared. Next, the array bounds check guarantees  $\text{LS}(index)$ . As a result, the sign extension in Figure 10(b) can be eliminated.

#### Theorem 2

If variables  $i, j$  satisfy the following three conditions,  $i+j$  does not need a 32-bit sign extension for the effective address computation of an array access whose subscript expression is  $i+j$ .

1. Both  $i$  and  $j$  have already been sign-extended from 32 bits.
2. Either  $i$  or  $j$  satisfies the following inequality:  
 $0 \leq i$  or  $j \leq 0x7fffffff$
3.  $\text{LS}(i+j)$  holds. <sup>†</sup>

**Proof.** Since  $i$  and  $j$  are commutable for  $i+j$ , it is sufficient to prove only the case in which  $i$  satisfies the second condition.

Case  $0 \leq i \leq 0x7fffffff$ :

Case 1 ( $0 \leq j \leq 0x7fffffff$ ): Because the upper 32 bits of  $i+j$  must be zero, Theorem 2 holds using Theorem 1.

Case 2 ( $0xffffffff80000000 \leq j \leq 0xffffffffffffff$ ): The range of  $i+j$  will be  $0xffffffff80000000 \leq i+j \leq 0xffffffffffffff$  or  $0 \leq i+j \leq 0x7ffffffe$ . Because of the third condition,  $0 \leq i+j \leq 0x7ffffffe$  will hold. Thus,  $i+j$  does not need a 32-bit sign extension.  $\square$

By using Theorem 2, we can eliminate the sign extension in the loop of Figure 9(a).

#### Theorem 3

If variables  $i, j$  satisfy the following three conditions,  $i-j$  does not need a 32-bit sign extension for the effective address computation of an array access whose subscript expression is  $i-j$ .

1. The upper 32 bits of  $i$  are initialized to zero.
2.  $j$  satisfies the following inequality:  
 $0 \leq j \leq 0x7fffffff$
3.  $\text{LS}(i-j)$  holds. <sup>†</sup>

<sup>†</sup> This condition always holds for Java.

**Proof.** The range of  $i$  is  $0 \leq i \leq 0x\text{fffffff}$ . The range of  $i-j$  will be  $0x\text{fffffff80000001} \leq i-j \leq 0x\text{fffffff}$  or  $0 \leq i-j \leq 0x\text{fffffff}$ . Because of the third condition,  $0 \leq i-j \leq 0x7\text{ffffff}$  will hold. Thus,  $i-j$  does not need a 32-bit sign extension.  $\square$

Theorem 3 is useful on IA-64 since zero extension is performed for every memory read. We can enhance sign extension elimination for subtraction by using Theorem 3. When Theorem 3 is applied to Figure 7, the sign extension at (1) can be eliminated.

If the maximum array size can be limited to a certain size or if an array size is known at compile time, the following theorem can be derived from Theorem 2.

#### Theorem 4

If variables  $i, j$  satisfy the following four conditions,  $i+j$  does not need a 32-bit sign extension for the effective address computation of an array access whose subscript expression is  $i+j$ .

1. The array size can be limited to  $maxlen$ , and the following inequality holds:  $\dagger$   
 $0 \leq maxlen \leq 0x7\text{ffffff}$
2. Both  $i$  and  $j$  have already been sign-extended from 32 bits.
3. Either  $i$  or  $j$  satisfies the following inequality:  
 $(maxlen-1) \cdot 0x7\text{ffffff} \leq i$  or  $j \leq 0x7\text{ffffff}$
4. The lower 32 bits of the array index  $i+j$  satisfy the following inequality for the language specification:  $\dagger$   
 $0 \leq \text{lower 32 bits of } (i+j) < maxlen \leq 0x7\text{ffffff}$

**Proof.** Since  $i$  and  $j$  are commutable for  $i+j$ , it is sufficient to prove only the case in which  $i$  satisfies the third condition.

Case  $0 \leq i \leq 0x7\text{ffffff}$ . Theorem 4 holds using Theorem 2.

Case  $(maxlen-1) \cdot 0x7\text{ffffff} \leq i \leq 0x\text{fffffff}$ .

Case 1 ( $0 \leq j \leq 0x7\text{ffffff}$ ): Theorem 4 holds using Theorem 2.

Case 2 ( $0x\text{fffffff80000000} \leq j \leq 0x\text{fffffff}$ ):  $i+j$  will be  $0x\text{fffffff00000000} + maxlen \leq i+j \leq 0x\text{fffffff}$ ; that is,  $maxlen \leq \text{lower 32 bits of } (i+j) \leq 0x\text{ffffffe}$ . The fourth condition excludes this case.  $\square$

Note that Theorems 2 and 4 can be applied to subtractions like  $i-k$  by computing the range of  $k$ , which can be computed by assigning  $(-k)$  to  $j$ .

We can eliminate more sign extensions by using Theorem 4. For example, the Java language specification defines the maximum array size as  $0x7\text{ffffff}$ . Therefore, we can eliminate a sign extension for the effective address computation of an array access whose subscript expression is  $i+j$ , if both  $i$  and  $j$  have been sign-extended and if  $-1 \leq i$  or  $j \leq 0x7\text{ffffff}$  holds. This will cover count down loops. In Figure 7, we can eliminate the sign extension (3) using Theorem 4. Figure 11 shows an example where a sign extension can be removed depending on the array size. This example is the same as the one in Figure 7(a) except that statement (2) is replaced by “ $i = i - 2$ ”. In this example, if  $mem$  is assumed to be  $0x80000000$  (the minimum value of a signed 32-bit representation), the lower 32 bits of  $i$  at the time of the first array access are  $0x7\text{ffffffe}$ . If this array size is  $0x7\text{ffffff}$ , the array element  $a[i]$  can be accessed. In this case, the sign extension (3) cannot be eliminated, because the upper 32 bits must be set correctly. On the

```

int j; // j is a 32-bit variable
int t = 0; // t is a 32-bit variable.
int i = mem; // i is a 32-bit variable.
// assuming mem = 0x80000000.
i = extend(i); - (1)
// i == 0xffffffff80000000.
do {
    i = i - 2; - (2)
    // i == 0xffffffff7ffffffe
    i = extend(i); - (3)
    // i == 0x7ffffffe. a[i] can be accessed depending on array size.
    j = a[i]; - (4)
    j = extend(j); - (5)
    j = j & 0x0ffffff - (6)
    j = extend(j); - (7)
    t += j; - (8)
    t = extend(t); - (9)
} while(i > start);
// need sign extension for t
d = (double)t; - (10)

```

Figure 11. A removable sign extension depending on array size

other hand, if the size of this array is known to be smaller than  $0x7\text{ffffff}$ , the access of element  $a[i]$  must be invalid. This is because the condition “the lower 32 bits of  $i$  ( $0x7\text{ffffffe}$ ) < the array’s size <  $0x7\text{ffffff}$ ” is never satisfied. Therefore, if it is known that the size of this array is always smaller than  $0x7\text{ffffff}$  at compile time, the sign extension (3) can be eliminated.

As another example, when we can limit the maximum array size to  $0x7\text{fff0001}$  based on limitation of the configurable memory resources, we can eliminate the sign extension if either  $i$  or  $j$  is larger than  $-65536$  and the other conditions (2 and 4) are met.

```

/* if it returns FALSE, sign extension is unnecessary for AOP
if it returns TRUE, sign extension is necessary for AOP */
AnalyzeARRAY(EXT, AOP) {
    if (all instructions that use the destination operand of AOP are
    array accesses) {
        for (D ∈ all instructions that define the source operand of EXT) {
            if (a flag of ARRAY for D has not been set yet) {
                set a flag of ARRAY for D;
                required = TRUE;
                switch(D) {
                    case The upper 32 bits of the destination operand of D are
                    always zero: /* Theorem 1 */
                        required = FALSE;
                        break;
                    case subtraction:
                    case addition:
                        required = AnalyzeTheorem2_3_4(D);
                        break;
                    case copy:
                        /* use UD-chain */
                        for (J ∈ all instructions that define the source operand of D) {
                            if (AnalyzeARRAY(J, AOP) return TRUE;
                        }
                        required = FALSE;
                        break;
                }
            }
            if (required) return TRUE;
        }
    }
    return FALSE;
}
return TRUE;
}

```

The algorithm AnalyzeARRAY analyzes the effective address computation of an array access to see whether the sign extension

can be eliminated by checking if any of Theorems 1, 2, 3, or 4 can be satisfied for all the instructions that define the source operand of the given sign extension.

The algorithm `AnalyzeTheorem2_3_4` analyzes if any of Theorems 2, 3, or 4 can be satisfied for  $D$ . Here, `upper32( $e$ )` and `lower32( $e$ )` denote the upper and lower 32 bits of  $e$ , respectively.

```

/* if it returns FALSE, sign extension is unnecessary for D
   if it returns TRUE, sign extension is necessary for D */
AnalyzeTheorem2_3_4(D){
  if (the second operand of D is sign-extended && /* use UD-chain */
      upper32(the first operand of D) are zero && /* use UD-chain */
      ((D is subtraction && 0 ≤ lower32(second operand of D) ≤ 0x7fffffff) ||
       (D is addition && -0x7fffffff ≤ lower32(second operand of D) ≤ 0))) {
    return FALSE; /* Theorem 3 */
  } else {
    arraylen = the maximum array size;
    if (the array size is known) arraylen = the array size;
    min_theorem4 = (arraylen - 1) - 0x7fffffff;
    if (both operands of D are sign-extended && /* use UD-chain */
        ((D is addition &&
          (min_theorem4 ≤ lower32(first operand of D) ≤ 0x7fffffff ||
           min_theorem4 ≤ lower32(second operand of D) ≤ 0x7fffffff) ||
          (D is subtraction &&
           (-0x7fffffff ≤ lower32(first operand of D) ≤ -min_theorem4 ||
            -0x7fffffff ≤ lower32(second operand of D) ≤ -min_theorem4)))) {
      return FALSE; /* Theorem 4 */
    }
  }
  return TRUE;
}

```

We use UD chains and the results of value range analysis to see if either Theorem 3 or 4 is satisfied. For Java, the conditions 1 and 4 of Theorem 4 always hold, and thus Theorem 2 always holds when Theorem 4 holds (`min_theorem4` is always zero for Theorem 2). Therefore, we can skip Theorem 2 here. We analyze the value range of the lower 32 bits for the “int” type in Figure 5 Step (2). Even if a value range analysis is not performed, our algorithm is still effective because one of the two operands is often a constant value (e.g. Figure 3(2)).

## 4. EXPERIMENTAL RESULTS

We used jBYTEmark and SPECjvm98 [18] benchmarks for the evaluation of our optimizations. To measure every result under the same environment, we ran each benchmark program from the command line. For jBYTEmark, we specified the benchmark size to measure them consistently. For SPECjvm98, we ran each benchmark program with the count set to 100 from the command line, instead of running all of the benchmarks continuously as suggested by the official SPEC run rules. We implemented our algorithm in the IBM Java Just-in-Time (JIT) compiler for IA-64. All the experiments were conducted on an IBM IntelliStation Z Pro model 689412X (two Intel Itanium 800 MHz processors with 2 GB of RAM) and Windows. The architectural characteristics related to sign extension optimizations for this machine are [9]:

- IA-64 has a 32-bit compare instruction to compare only the lower 32 bits of registers, so array bounds checking can be implemented with no sign extension.
- Since a value is zero-extended during any memory read on IA-64, there are many opportunities to utilize Theorems 1 and 3 as described in Section 3.

## 4.1 Performance Improvement

We instrumented the compiled code to count the remaining sign extension instructions to see the effectiveness of sign extension elimination. **Table 1** and **Table 2** show the dynamic counts of 32-bit sign extensions and the percentages during a sample run of each benchmark program of jBYTEmark and SPECjvm98 for IA-64, respectively. In these tables, white circles denote improved results and black circles denote worsened results. The shaded cells denote important cells, and others denote breakdowns of our new algorithm. We measured the following algorithm variants:

**Disable opt.:** Disable all the sign extension optimizations in Figure 5(3).

**Baseline:** Generate a sign extension instruction at the code generation phase immediately before every instruction that requires a sign extension.

**First algorithm (bwd flow):** Our first algorithm. It eliminates sign extensions using backward dataflow analysis.

**Basic ud/du:** New algorithm, but disable sign extension insertion, order determination, and elimination for array indices.

**Insert:** Enable sign extension insertion, but disable both order determination and elimination for array indices.

**Order:** Enable order determination, but disable both sign extension insertion and elimination for array indices.

**Insert, order:** Enable both “insert” and “order”, but disable elimination for array indices.

**Array:** Enable elimination for array indices, but disable both sign extension insertion and order determination.

**Array, insert:** Enable both “array” and “insert”, but disable sign extension insertion.

**Array, order:** Enable both “array” and “order”, but disable order determination.

**All, using PDE (reference):** All optimizations in this paper are applied, but the insertion algorithm is a variant of the partial dead code elimination algorithm.

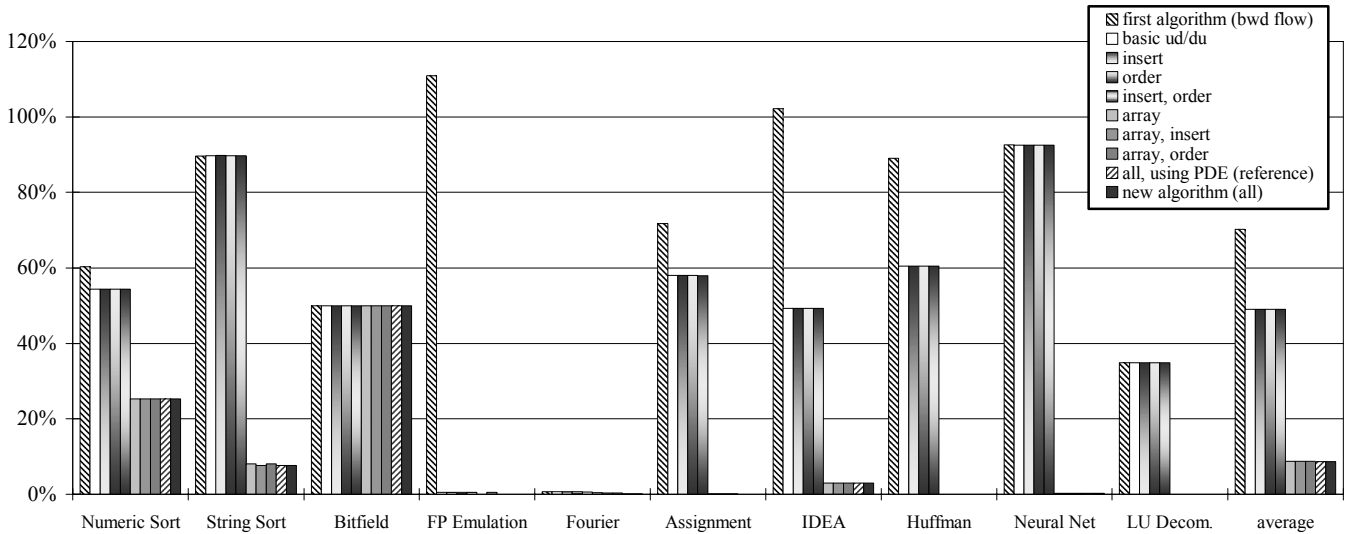
**New algorithm (all):** All optimizations in this paper are applied.

For our baseline, we generate a sign extension instruction at the code generation phase immediately before every instruction that requires a sign extension. We use this algorithm as our baseline because it is a simple and effective approach without performing any optimization. Any version that disables the order determination performs the sign extension elimination in the reverse depth first search order, the same order in which backward dataflow analysis is performed. **Figure 12** and **Figure 13** plot the percentages of dynamic counts of remaining sign extensions over our baseline.

Overall, our algorithm (denoted as “new algorithm (all)”) eliminates between 60.96% and 99.99% of sign extensions over our baseline. The difference between “first algorithm (bwd flow)” and “basic ud/du” shows how often the second problem, described in Section 1, occurs, and how often other optimizations increase the opportunity for eliminating sign extensions. Sign extension elimination for array indices is most effective for all the benchmark programs. Regarding the order determination and the sign extension insertion, we can make the following observations:

**Table 1. Dynamic counts of remaining 32-bit sign extensions for jBYTEmark (○: improved result, ●: worsened result)**

	Numeric Sort	String Sort	Bitfield	FP Emu.	Fourier	Assignment	IDEA	Huffman	Neural Net	LU Decom.	average
disable opt.	3758195644 (203.53%)	998928087 (244.20%)	1449826010 (175.51%)	1714076540 (829.37%)	14430930 (100.04%)	1531993239 (103.25%)	1458959514 (210.26%)	1143451491 (155.66%)	324884747 (93.63%)	738877762 (34.87%)	(215.03%)
baseline	1846480139 (100.00%)	409059333 (100.00%)	826057022 (100.00%)	206672979 (100.00%)	14424738 (100.00%)	1483839474 (100.00%)	693886034 (100.00%)	734563912 (100.00%)	346982678 (100.00%)	2118828282 (100.00%)	(100.00%)
first algorithm (bwd flow)	1113353493 ○ (60.30%)	366540417 ○ (89.61%)	413060614 ○ (50.00%)	229190109 ● (110.90%)	98150 ○ (0.68%)	1064297572 ○ (71.73%)	709112140 ● (102.19%)	653693676 ○ (88.99%)	321123167 ○ (92.55%)	738398682 ○ (34.85%)	○(70.18%)
basic ud/du	1004325701 ○ (54.39%)	366876481 (89.69%)	413057597 (50.00%)	1129233 ○ (0.55%)	95474 (0.66%)	861401297 ○ (58.05%)	341860302 ○ (49.27%)	444498974 ○ (60.51%)	320996644 (92.51%)	738370555 (34.85%)	○(49.05%)
insert	1004329535 (54.39%)	367129292 ● (89.75%)	413039165 (50.00%)	1128654 (0.55%)	94863 (0.66%)	861403261 (58.05%)	341859947 (49.27%)	444498283 (60.51%)	320996048 (92.51%)	738568088 ● (34.86%)	(49.05%)
order	1004325701 (54.39%)	366876481 (89.69%)	413057597 (50.00%)	1129232 (0.55%)	95474 (0.66%)	861401297 (58.05%)	341860299 (49.27%)	444498974 (60.51%)	320996644 (92.51%)	738370555 (34.85%)	(49.05%)
insert, order	1004320308 (54.39%)	366867867 (89.69%)	413052165 (50.00%)	1123692 ○ (0.54%)	90139 ○ (0.62%)	860117895 ○ (57.97%)	341854867 (49.27%)	444477957 (60.51%)	320991237 (92.51%)	738365142 (34.85%)	○(49.03%)
array	466217696 ○ (25.25%)	32825823 ○ (8.02%)	412965502 ○ (49.99%)	37083 ○ (0.02%)	58188 ○ (0.40%)	1433880 ○ (0.10%)	20496734 ○ (2.95%)	114906 ○ (0.02%)	814294 ○ (0.23%)	53711 ○ (0.00%)	○ (8.70%)
array, insert	466217905 (25.25%)	31422376 ○ (7.68%)	412997440 ● (50.00%)	1082443 ● (0.52%)	54158 ○ (0.38%)	1432224 (0.10%)	20492756 (2.95%)	110281 (0.02%)	810069 (0.23%)	247626 ● (0.01%)	● (8.71%)
array, order	466213149 (25.25%)	32821279 (8.02%)	412960937 (49.99%)	32095 (0.02%)	53855 ○ (0.37%)	1429335 (0.10%)	20492173 (2.95%)	109923 ○ (0.01%)	809749 (0.23%)	49152 (0.00%)	(8.70%)
all, using PDE (reference)	466212816 (25.25%)	31167734 ○ (7.62%)	412960608 (49.99%)	31756 (0.02%)	17543 ○ (0.12%)	151010 ○ (0.01%)	20491806 (2.95%)	104414 (0.01%)	808814 (0.23%)	48826 (0.00%)	○ (8.62%)
new algorithm (all)	466205975 (25.25%)	31158251 ○ (7.62%)	412953766 (49.99%)	24770 ○ (0.01%)	10759 ○ (0.07%)	144163 ○ (0.01%)	20484964 (2.95%)	87114 (0.01%)	802576 (0.23%)	41980 (0.00%)	○ (8.62%)



**Figure 12. Dynamic counts of remaining 32-bit sign extensions for jBYTEmark (baseline=100%)**

1. Combining sign extension insertion or elimination for array indices with order determination enhances the effectiveness of elimination.
2. Sign extension insertion is ineffective without order determination.

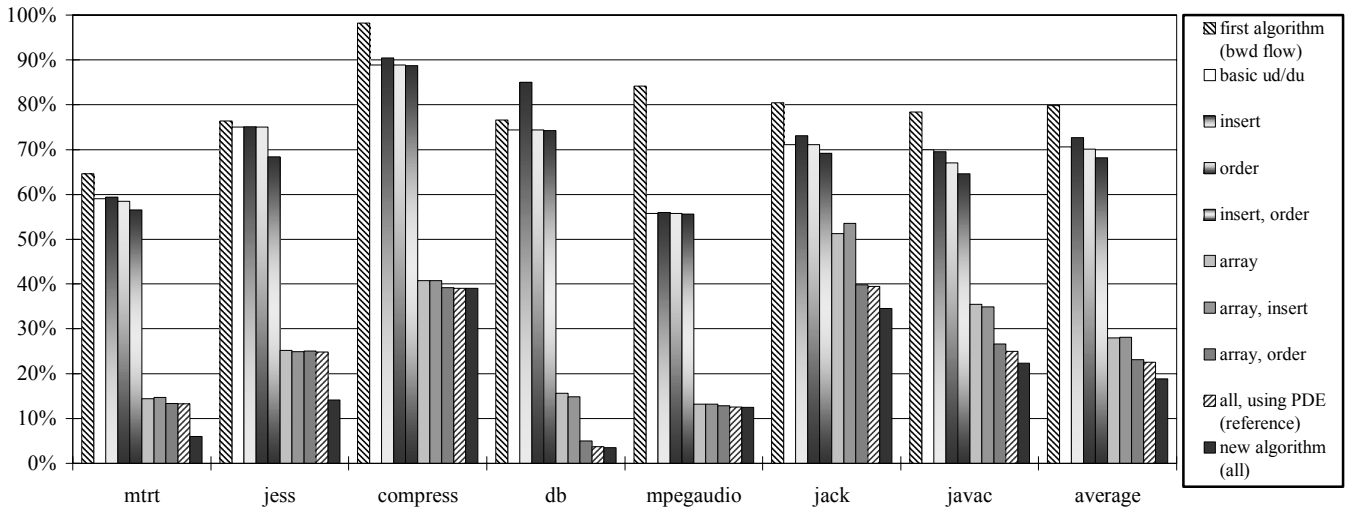
Regarding the first observation, when either sign extension insertion or elimination for array indices is done, there are often several sign extensions that are potential candidates for elimination. Therefore, using order determination enhances their effectiveness. Order determination alone (denoted as “order” in Table 2) is not

effective for any of the programs except for the *mtrt* and *javac* benchmarks. Moreover, many sign extensions that cannot be eliminated without a combination of “array”, “order”, and “insert” (denoted as “all”) are found in most of the benchmark programs.

Regarding the second observation, when order determination is disabled, the possibility of sign extensions remaining in frequently executed regions is increased. The combination of the order determination and the sign extension insertion is particularly effective for FP Emulation in jBYTEmark (1,082,443 vs. 24,770). As we mentioned in Section 2.1, we also tried another insertion algo-

**Table 2. Dynamic counts of remaining 32-bit sign extensions for SPECjvm98 (○: improved result, ●: worsened result)**

	mtrt	jess	compress	db	mpegaudio	jack	javac	average
disable opt.	19162530 (140.70%)	165578180 (193.42%)	1831002215 (284.42%)	298338613 (140.73%)	807174869 (189.30%)	95753436 (175.38%)	234753067 (174.85%)	(185.54%)
baseline	13619419 (100.00%)	85606229 (100.00%)	643756102 (100.00%)	211989358 (100.00%)	426398800 (100.00%)	54596337 (100.00%)	134261024 (100.00%)	(100.00%)
first algorithm (bwd flow)	8800008 ○ (64.61%)	65389295 ○ (76.38%)	632370673 ○ (98.23%)	162360645 ○ (76.59%)	358862336 ○ (84.16%)	43946795 ○ (80.49%)	105245327 ○ (78.39%)	○ (79.84%)
basic ud/du	8042506 ○ (59.05%)	64268121 ○ (75.07%)	572297310 ○ (88.90%)	157713624 ○ (74.40%)	237743297 ○ (55.76%)	38839843 ○ (71.14%)	93920998 ○ (69.95%)	○ (70.61%)
insert	8091775 ● (59.41%)	64282490 ● (75.09%)	582159879 ● (90.43%)	180226907 ● (85.02%)	238661074 ● (55.97%)	39919110 ● (73.12%)	93400183 ○ (69.57%)	● (72.66%)
order	7962959 ○ (58.47%)	64268123 (75.07%)	572297310 (88.90%)	157713624 (74.40%)	237743297 (55.76%)	38839851 (71.14%)	90060780 ○ (67.08%)	○ (70.12%)
insert, order	7702613 ○ (56.56%)	58537108 ○ (68.38%)	571386557 ○ (88.76%)	157417853 ○ (74.26%)	237099838 ○ (55.61%)	37781407 ○ (69.20%)	86758551 ○ (64.62%)	○ (68.20%)
array	1961456 ○ (14.40%)	21544766 ○ (25.17%)	262105810 ○ (40.72%)	33105544 ○ (15.62%)	56354143 ○ (13.22%)	27999218 ○ (51.28%)	47560306 ○ (35.42%)	○ (27.98%)
array, insert	1997325 ● (14.67%)	21330983 ○ (24.92%)	262095499 ○ (40.71%)	31514492 ○ (14.87%)	56287323 ○ (13.20%)	29245657 ● (53.57%)	46826978 ○ (34.88%)	● (28.12%)
array, order	1815980 ○ (13.33%)	21451357 ○ (25.06%)	252222150 ○ (39.18%)	10557513 ○ (4.98%)	54729584 ○ (12.84%)	21714796 ○ (39.77%)	35732666 ○ (26.61%)	○ (23.11%)
all, using PDE (reference)	1811531 ○ (13.30%)	21249333 ○ (24.82%)	251323510 ○ (39.04%)	7794544 ○ (3.68%)	53638690 ○ (12.58%)	21533665 ○ (39.44%)	33544130 ○ (24.98%)	○ (22.55%)
new algorithm (all)	816153 ○ (5.99%)	12104429 ○ (14.14%)	251303625 ○ (39.04%)	7472849 ○ (3.53%)	53079482 ○ (12.45%)	18844058 ○ (34.52%)	29942890 ○ (22.30%)	○ (18.85%)



**Figure 13. Dynamic counts of remaining 32-bit sign extensions for SPECjvm98 (baseline=100%)**

rithm that is a variant of partial dead code elimination algorithm. Our experiments (“all, using PDE” vs. “new algorithm (all)”) show that the simple insertion algorithm is slightly better for all the benchmarks.

**Figure 14** and **Figure 15** show the performance improvement over our baseline for jBYTEmark and SPECjvm98, respectively. Our sign extension elimination is effective for all benchmarks, particularly for the *Huffman* and *compress* benchmarks.

## 4.2 JIT Compilation Time

This section describes how our approach affects the JIT compilation time. We measured the breakdown of the JIT compilation time for IA-64, as shown in **Table 3**, by using a trace tool. In summary, both sign extension optimizations and UD/DU chain creation increased the total compilation time by 3.03% on average.

Since we used the UD/DU chains for other optimizations, these chain creations are still necessary even if sign extension optimizations are not performed. Excluding the time for UD/DU chain

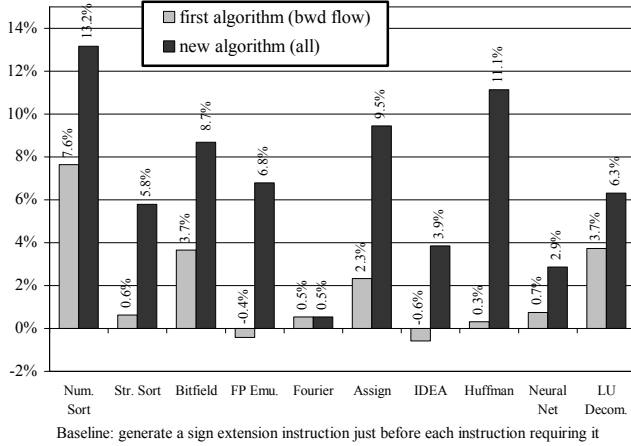


Figure 14. Performance Improvement for jBTEmark

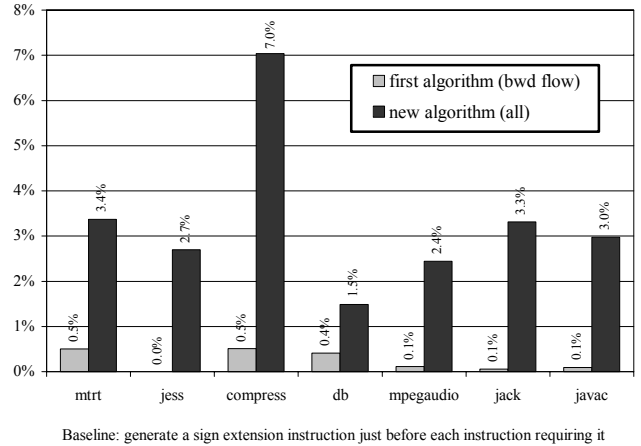


Figure 15. Performance Improvement for SPECjvm98

Table 3. Breakdown of JIT compilation time

	Sign extension optimizations (all)	UD/DU chain creation	Others
mrtt	0.20%	2.31%	97.49%
jess	0.20%	2.44%	97.36%
compress	0.13%	3.05%	96.82%
db	0.08%	2.56%	97.35%
mpegaudio	0.10%	2.41%	97.49%
jack	0.10%	2.53%	97.37%
javac	0.13%	2.52%	97.35%
num. sort	0.09%	3.14%	96.77%
str. sort	0.09%	2.63%	97.28%
Bitfield	0.10%	3.56%	96.34%
FP emu.	0.07%	3.19%	96.73%
fourier	0.11%	3.51%	96.38%
assignment	0.10%	3.30%	96.61%
IDEA	0.13%	3.23%	96.64%
huffman	0.10%	3.32%	96.58%
neural net	0.08%	2.97%	96.95%
lu decom.	0.09%	2.92%	96.99%
average	0.11%	2.92%	96.97%

creation, the sign extension optimizations increased the total compilation time only by 0.11% on average, while they achieved significant performance improvements as shown in Figure 14 and Figure 15.

## 5. PREVIOUS WORK

PentiumGCC [17], a version of the GNU C compiler customized for Pentium, performs sign extension elimination, but there is no description of it. The AS/400 Optimizing Translator [3] also performs sign extension elimination. It describes two concepts of sign extension elimination, but unfortunately no detailed algorithms are presented.

A partial redundancy elimination approach [13, 14] is also effective for eliminating sign extensions. In fact, our PRE phase (Figure 5(2)) eliminated some sign extensions. A partial dead code elimination (PDE) approach [15] turns out to be less effective than the simple insertion algorithm. Figure 16 shows a draw-

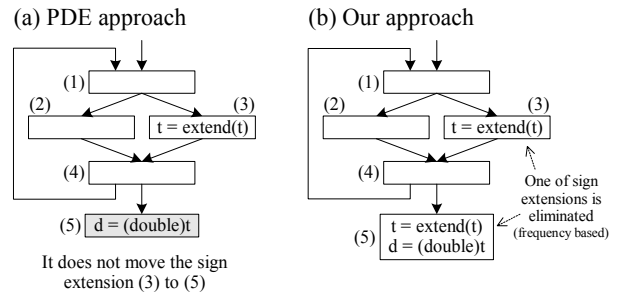


Figure 16. Drawback of the PDE approach

back of the PDE approach. In this example, PDE does not move the sign extension (3) to (5). In contrast, our approach first inserts a sign extension (5) and next eliminates sign extensions selectively starting with the most frequently executed region. Therefore, if the compiler judges that (5) is more frequently executed than (3), the sign extension (5) will be eliminated. If the compiler judges that (3) is more frequently executed than (5), the sign extension (3) will be eliminated. A path-profile-guided partial dead code elimination approach [6] could be more effective than the simple insertion algorithm at the cost of a much longer compilation time, but we believe it is not practical for dynamic compilers.

## 6. CONCLUSIONS

In this paper, we have presented a new algorithm for sign extension elimination. To the best of our knowledge, this is the first algorithm to provide fast and effective sign extension elimination. Sign extensions are eliminated selectively based on the order of the most to the least frequently executed code. Our approach can eliminate sign extensions for effective address computation of array accesses based on our assumption that a negative index is not allowed by the language specification. Our experiments show that the majority of sign extensions can be eliminated at a small cost in compilation time. Although we implemented our algorithm for Java, it should also be applicable for other languages requiring sign extensions.

## 7. ACKNOWLEDGMENTS

We would like to thank all the members of the IBM JIT Compiler Research team in Tokyo for helpful discussions and analysis of possible performance improvements. We also thank Stephen Fink and Arvin Shepherd of IBM T.J. Watson Research Center for their helpful comments on this paper. Finally, we thank the anonymous reviewers for their helpful comments.

## 8. REFERENCES

- [1] A.V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Reading, MA, 1986.
- [2] AMD x86-64™ Architecture Programmers Overview, [http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/x86-64\\_overview.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/x86-64_overview.pdf)
- [3] K. V. Besaw, R. J. Donovan, E. C. Prosser, R. R. Roediger, W. J. Schmidt, and P. J. Steinmetz. "The Optimizing Translator." <http://www-1.ibm.com/servers/eserver/iserries/beyondtech/translator.htm>
- [4] W. Blume and R. Eigenmann. "Symbolic range propagation" In *Proceedings of the 9th International Parallel Processing Symposium*, pp.357-363, 1995.
- [5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, Addison-Wesley Publishing Co., Reading, 1996.
- [6] R. Gupta, D.A. Berson, and J.Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pp.102-115, San Francisco, California, November 1997.
- [7] W. Harrison. "Compiler Analysis of the Value Ranges for Variables" In *IEEE Transactions on Software Engineering*, Vol. 3, No. 3, pp.243-250, 1977.
- [8] IBM Corp.: PowerPC Homepage <http://www.chips.ibm.com/products/powerpc/>
- [9] Intel Corp.: Itanium Architecture - Manuals. <http://www.intel.com/design/itanium/manuals/>
- [10] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. "Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler." In *Proceedings of the ACM SIGPLAN Java Grande Conference*, June 1999.
- [11] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. "A Study of Devirtualization Techniques for a Java Just-In-Time Compiler." In *Conference on Object Oriented Programming Systems, Languages & Applications – OOP-SLA*, 2000.
- [12] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap, In *Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems*, pp. 139-149, 2000.
- [13] 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.
- [14] 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.
- [15] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 147-158, Orlando, Florida, June 1994.
- [16] T. Ogasawara, H. Komatsu, and T. Nakatani. "A Study of Exception Handling and Its Dynamic Optimization in Java", In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001
- [17] PentiumGCC Frequently asked Questions <http://www.goof.com/pcg/pgcc-faq.html>
- [18] Standard Performance Evaluation Corp. "SPEC JVM98 Benchmarks," <http://www.spec.org/osg/jvm98/>
- [19] 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.
- [20] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. "A Dynamic Optimization Framework for a Java Just-In-Time Compiler", In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001
- [21] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. "Accurate Static Estimators for Program Optimization", In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, pp. 85-96, 1994.

## APPENDIX

Table 4 and Table 5 show the effectiveness of each theorem. In these tables, "Theorem 1" corresponds to the case in which only Theorem 1 is used for elimination of array indices for "new algorithm (all)". In summary, Theorem 4 is particularly effective among all, and combining several theorems is most effective for many benchmarks.

**Table 4. Effectiveness of each theorem (dynamic counts of remaining 32-bit sign extensions for jBYTEmark)**

	Numeric Sort	String Sort	Bitfield	FP Emu.	Fourier	Assignment	IDEA	Huffman	Neural Net	LU Decom.
insert_order	1004320308	366867867	413052165	1123692	90139	860117895	341854867	444477957	320991237	738365142
Theorem 1	532527189	37591283	413030875	1098500	33606	742647172	40909148	145939732	9644225	20104474
Theorem 2	520223880	56143191	412959054	30162	16003	149439	20491056	277594795	807852	437009
Theorem 3	532529340	59045953	413033014	1100680	35730	742649295	40911300	342776487	9646366	20300531
Theorem 4	466208541	56140454	412956310	27367	13283	146711	20487519	196924291	805127	240368
new algorithm (all)	466205975	31158251	412953766	24770	10759	144163	20484964	87114	802576	41980

**Table 5. Effectiveness of each theorem (dynamic counts of remaining 32-bit sign extensions for SPECjvm98)**

	mtrt	jess	compress	db	mpegaudio	jack	javac
insert. order	7702613	58537108	571386557	157417853	237099838	37781407	86758551
Theorem 1	2465530	22954568	324555558	57947218	105862753	32209121	50888050
Theorem 2	1677055	28970272	468027675	58933166	81541931	24794609	47888369
Theorem 3	3197882	39772061	495377600	108379551	134199846	35651120	64421260
Theorem 4	1588599	28961899	412269219	57911643	81440630	22966525	45041715
new algorithm (all)	816153	12104429	251303625	7472849	53079482	18844058	29942890