January 24, 2002
RT0442
Computer Science 10 pages

Research Report

Effective Sign Extension Elimination

Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani

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

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

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 Sign Extension Elimination

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

ABSTRACT

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 "int" as a 32-bit data type, which is used frequently. 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 IA64. 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.

Keywords

sign extension, Java, JIT, compilers, IA64, Itanium

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 "int" as a 32-bit data type [5], and this type is used frequently. 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.

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)).

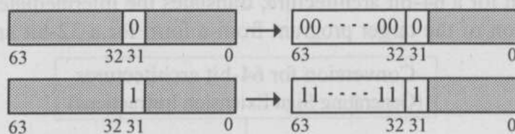
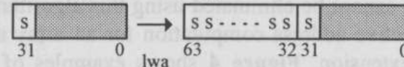


Figure 1. Sign extension of a 32-bit value to a 64-bit value

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)
```

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

Figure 2. Two types of sign extension

Even for the PowerPC, 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), a sign extension instruction is required in addition to the implicit sign extension instruction (statement (1)). We call this an "*explicit sign extension*" and write it as *extend( )*. The PowerPC has the *extend sign* (***exts***) instructions, while the IA64 architecture [9] has the *sign extend* (***sxt***) instructions.

Sign extension elimination is even more important for those architectures lacking any implicit sign extension instruction. For example, the IA64 is such an architecture and values are zero-extended during memory reads, thus requiring explicit sign extension instructions.

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 analy-

```
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 = 0x0fffffff; // 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

sis. 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 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)[1]. 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 IA64 and PPC64. For IA64 (Figure 4 (b)), if the sign extension can be eliminated, an effective address can be computed in one instruction (shladd). 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 of that array index out of the loop. On the other hand, we can utilize an instruction that computes an effective address without an explicit sign extension on some architectures. For example, PPC64 has an ***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.

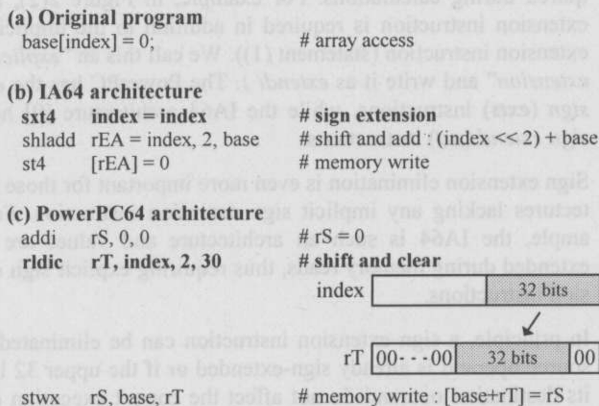


Figure 4. Examples of effective address computations

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 latest sign extension 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 IA64. Our JIT compiler is designed to work on many platforms, and thus many optimizations [10, 11, 12, 16, 19, 20] are performed at the intermediate language level in order to improve portability. By also porting these optimizations to the IA64 JIT compiler, we could achieve 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 that our algorithm can effectively eliminate the majority of sign extensions. They also show that this significantly improves performance, while increasing the JIT compilation time by only 0.11%.

2. OUR APPROACH

Figure 5 shows a flow diagram of our algorithm for sign extension elimination, which consists of three steps. Step (1), conversion for a 64-bit architecture, translates the intermediate representation of the target program from a form for a 32-bit architecture

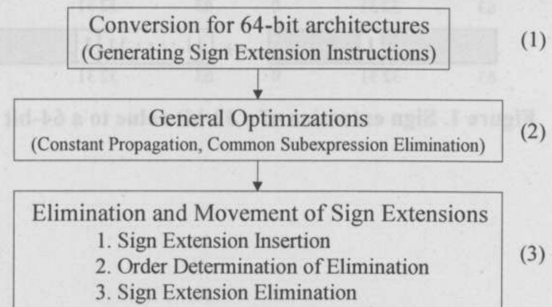


Figure 5. Flow diagram of our algorithm

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

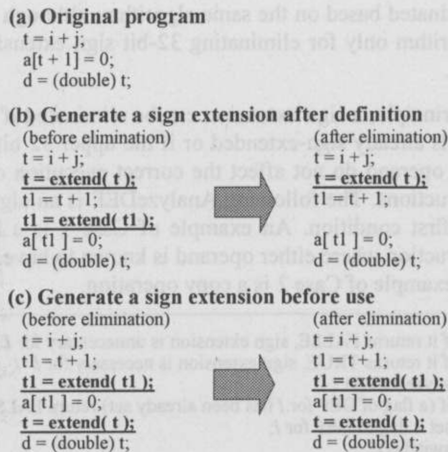


Figure 6. Two approaches to generate sign extensions

to a 64-bit architecture form. There are two approaches to generate sign extension instructions. One is to generate 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. The other is to generate sign extension instructions immediately before every instruction ***I*** that requires sign extensions 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 before a use point as in (c), no sign extension can be eliminated. In contrast, if the compiler generates a sign extension after a definition point as in (b), one sign extension can be eliminated. (See Section 3 for the theorems that justify this elimination.).

Step (2), general optimizations (Figure 5(2)), also optimize sign extensions. For example, when a constant is propagated as the source operand of a sign extension, the sign extension will be changed to a copy instruction by constant folding. Sign extension is also applied to common sub-expression elimination. 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.

Step (3), elimination and movement of sign extension (Figure 5(3)), has three phases. In the first phase ((3)-1), a sign extension is inserted immediately before every instruction that requires sign extensions. 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

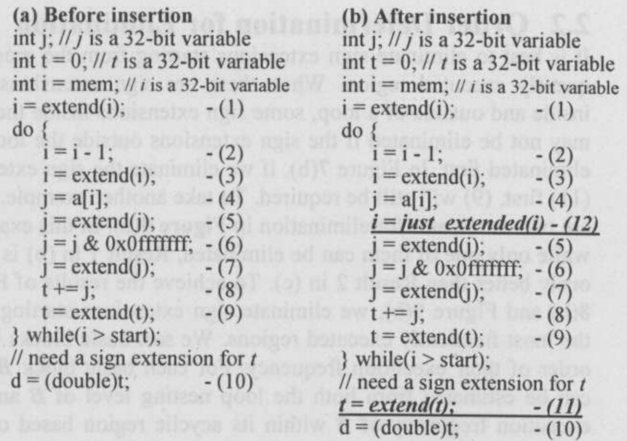


Figure 7. Example of inserting a sign extension

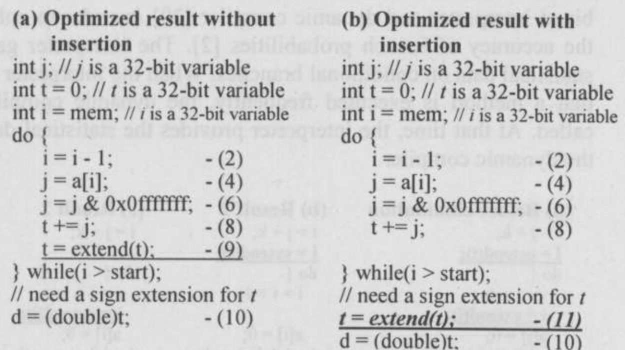


Figure 8. The optimized result of Figure 7

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

We tried another insertion algorithm that is a variant of 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 11 and Figure 12 ("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. 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 [2]. 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
```
i = j + k;
i = extend(i);
do {
  i = i + 1;
  i = extend(i);
  a[i] = 0;
} while(i < end);
```

(b) Result 1
```
i = j + k;
i = extend(i);
do {
  i = i + 1;

  a[i] = 0;
} while(i < end);
```

(c) Result 2
```
i = j + k;

do {
  i = i + 1;
  i = extend(i);
  a[i] = 0;
} while(i < end);
```

Figure 9. Example of requiring the 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, determined as described in Section 2.2. As mentioned in Section 1, our algorithm is so powerful to eliminate sign extensions for array subscripts. We will discuss this distinguished feature in the next section, while we focus on basic cases of eliminations here.

```
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;
}
```

The EliminateOneExtend shows an algorithm that 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.

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 following AnalyzeDEF is an algorithm to check 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 AnalyzeUSE algorithm 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.

```
/* 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 been already 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;
}
```

This phase of sign extension elimination ends with one trivial operation; that is, to eliminate all the dummy sign extensions.

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. Since both PPC64 and IA64 have such instructions [8, 9], we think this is reasonable. Based on this language specification, the following predicate will hold for a subscript expression *e*.

LS(*e*) ≡ 0 ≦ lower 32 bits of *e* ≦ 0x7fffffff

In general, if the language specification rules out any array access with a negative array index, then the following four theorems can be derived. These theorems depend on knowledge of the value range, which can be determined at compile time using one of the value range analysis techniques [4, 7]. 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***.

- The upper 32 bits of ***i*** are initialized to zero.
- LS(***i***) holds.

Proof. Because of the two conditions, 0 ≦ ***i*** ≦ 0x7fffffff will always hold. Because ***i*** must not have a negative value as a signed 32-bit representation, ***i*** is already sign-extended. Thus, ***i*** does not need a 32-bit sign extension. □

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***.

- Both ***i*** and ***j*** have already been sign-extended from 32 bits.
- Either ***i*** or ***j*** satisfies the following inequality:
  0 ≦ ***i*** or ***j*** ≦ 0x7fffffff
- LS(***i+j***) holds.

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 ≦ ***i*** ≦ 0x7fffffff:

- ➢ Case 0 ≦ ***j*** ≦ 0x7fffffff: Because the upper 32 bits of ***i+j*** must be zero, Theorem 2 holds using Theorem 1.
- ➢ Case 0xffffffff80000000 ≦ ***j*** ≦ 0xffffffffffffffff: The range of ***i+j*** will be 0xffffffff80000000 ≦ ***i+j*** ≦ 0xffffffffffffffff or 0 ≦ ***i+j*** ≦ 0x7ffffffe. Because of the third condition, 0 ≦ ***i+j*** ≦ 0x7ffffffe will hold. Because ***i+j*** must not have a negative value for a signed 32-bit representation, ***i+j*** is already sign-extended. Thus, ***i+j*** does not need a 32-bit sign extension. □

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***.

- The upper 32 bits of ***i*** are initialized to zero.
- ***j*** satisfies the following inequality:
  0 ≦ ***j*** ≦ 0x7fffffff
- LS(***i-j***) holds.

Proof. The range of ***i*** is 0 ≦ ***i*** ≦ 0xffffffff. The range of ***i-j*** will be 0xffffffff80000001 ≦ ***i-j*** ≦ 0xffffffffffffffff or 0 ≦ ***i-j*** ≦ 0xffffffff. Because of the third condition, 0 ≦ ***i-j*** ≦ 0x7fffffff will hold. Because ***i-j*** must not have a negative value as a signed 32-bit representation, ***i-j*** is already sign-extended. Thus, ***i-j*** does not need a 32-bit sign extension. □

Theorem 3 is useful on IA64 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***.

- The maximum array size can be limited to ***maxlen***, and the following inequality holds:
  0 ≦ ***maxlen*** ≦ 0x7fffffff
- Both ***i*** and ***j*** have already been sign-extended from 32 bits.
- Either ***i*** or ***j*** satisfies the following inequality:
  (***maxlen***-1)-0x7fffffff ≦ ***i*** or ***j*** ≦ 0x7fffffff
- The lower 32 bits of the array index ***i+j*** satisfy the following inequality for the language specification:
  0 ≦ lower 32 bits of (***i+j***) < ***maxlen*** ≦ 0x7fffffff

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 ≦ ***i*** ≦ 0x7fffffff: Theorem 4 holds using Theorem 2.

Case (***maxlen***-1)-0x7fffffff ≦ ***i*** ≦ 0xffffffffffffffff:

- ➢ Case 0 ≦ ***j*** ≦ 0x7fffffff: Theorem 4 holds using Theorem 2.
- ➢ Case 0xffffffff80000000 ≦ ***j*** ≦ 0xffffffffffffffff: ***i+j*** will be 0xffffffff00000000+***maxlen*** ≦ ***i+j*** ≦ 0xfffffffffffffffe; that is, ***maxlen*** ≦ lower 32 bits of (***i+j***) ≦ 0xfffffffe. The fourth condition excludes this case. □

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 0x7fffffff. 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 ≦ *i*** or ***j* ≦ 0x7fffffff** holds. This will cover count down loops. In Figure 7, we can eliminate the sign extension (3) using Theorem 4.

As another example, when we can limit the maximum array size to 0x7fff0001 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 are met.

Figure 10 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 is 0x7ffffffe. If this array size is 0x7fffffff, 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 other hand, if the size of this array is known to be smaller than 0x7fffffff, the access of element ***a*[*i*]** must be invalid. This is because the condition "the lower 32 bits of ***i*** (0x7ffffffe) < the array's size < 0x7fffffff" is never satisfied. Therefore, if it is known that the size of this array is always smaller than 0x7fffffff at compile time, the sign extension (3) can be eliminated.

```
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 & 0x0fffffff                  - (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 10. A removable sign extension depending on array size

The following AnalyzeARRAY algorithm 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. We also use UD/DU chains to analyze if any of Theorems is satisfied.

```
/* 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:
          /* use UD/DU chains */
          if (conditions of either Theorem 2, 3, or 4 met for
              operands of D){
            required = FALSE;
          }
          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;
}
```

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 IA64. All the experiments were conducted on an IBM IntelliStation Z Pro model 689412X (two Intel Itanium 800 MHz processors with 2 GB of RAM), Windows. The architectural characteristics related to sign extension optimization for this machine are [9]:

- IA64 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 values are zero-extended during memory reads on IA64, there are many opportunities to utilize Theorems 1 and 4 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

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
baseline	3758195644	998928087	1449826010	1714076540	14430930	1531993239	1458959514	1143451491	324884747	738877762	
	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)
gen use (reference)	1846480139	409059333	826057022	206672979	14424738	1483839474	693886034	734563912	346982678	2118828282	
	○ (49.13%)	○ (40.95%)	○ (56.98%)	○ (12.06%)	○(99.96%)	○ (96.86%)	○ (47.56%)	○ (64.24%)	●(106.80%)	● (286.76%)	○(86.13%)
first algorithm (bwd flow)	1113353493	366540417	413060614	229190109	98150	1064297572	709112140	653693676	321123167	738398682	
	○ (29.62%)	○ (36.69%)	○ (28.49%)	○ (13.37%)	○ (0.68%)	○ (69.47%)	○ (48.60%)	○ (57.17%)	○ (98.84%)	○ (99.94%)	○(48.29%)
basic ud/du	1004325701	366876481	413057597	1129233	95474	861401297	341860302	444498974	320996644	738370555	
	○ (26.72%)	(36.73%)	(28.49%)	○ (0.07%)	(0.66%)	○ (56.23%)	○ (23.43%)	○ (38.87%)	(98.80%)	(99.93%)	○(40.99%)
insert	1004329535	367129292	413039165	1128654	94863	861403261	341859947	444498283	320996048	738568088	
	(26.72%)	● (36.75%)	(28.49%)	(0.07%)	(0.66%)	(56.23%)	(23.43%)	(38.87%)	(98.80%)	● (99.96%)	●(41.00%)
order	1004325701	366876481	413057597	1129232	95474	861401297	341860299	444498974	320996644	738370555	
	(26.72%)	(36.73%)	(28.49%)	(0.07%)	(0.66%)	(56.23%)	(23.43%)	(38.87%)	(98.80%)	(99.93%)	(40.99%)
insert, order	1004320308	366867867	413052165	1123692	90139	860117895	341854867	444477957	320991237	738365142	
	(26.72%)	(36.73%)	(28.49%)	(0.07%)	(0.62%)	○ (56.14%)	(23.43%)	(38.87%)	(98.80%)	(99.93%)	○(40.98%)
array	466217696	32825823	412965502	37083	58188	1433880	20496734	114906	814294	53711	
	○ (12.41%)	○ (3.29%)	(28.48%)	○ (0.00%)	○ (0.40%)	○ (0.09%)	○ (1.40%)	○ (0.01%)	○ (0.25%)	○ (0.01%)	○ (4.63%)
array, insert	466217905	31422376	412997440	1082443	54158	1432224	20492756	110281	810069	247626	
	(12.41%)	○ (3.15%)	● (28.49%)	● (0.06%)	○ (0.38%)	(0.09%)	(1.40%)	(0.01%)	(0.25%)	● (0.03%)	(4.63%)
array, order	466213149	32821279	412960937	32095	53855	1429335	20492173	109923	809749	49152	
	(12.41%)	(3.29%)	(28.48%)	○ (0.00%)	○ (0.37%)	(0.09%)	(1.40%)	(0.01%)	○ (0.25%)	○ (0.01%)	(4.63%)
all, using PDE (reference)	466212816	31167734	412960608	31756	17543	151010	20491806	104414	808814	48826	
	(12.41%)	○ (3.12%)	(28.48%)	(0.00%)	○ (0.07%)	○ (0.01%)	(1.40%)	(0.01%)	(0.25%)	(0.01%)	○ (4.58%)
new algorithm (all)	466205975	31158251	412953766	24770	10759	144163	20484964	87114	802576	41980	
	(12.41%)	○ (3.12%)	(28.48%)	○ (0.00%)	○ (0.07%)	○ (0.01%)	○ (1.40%)	○ (0.01%)	(0.25%)	○ (0.01%)	(4.58%)

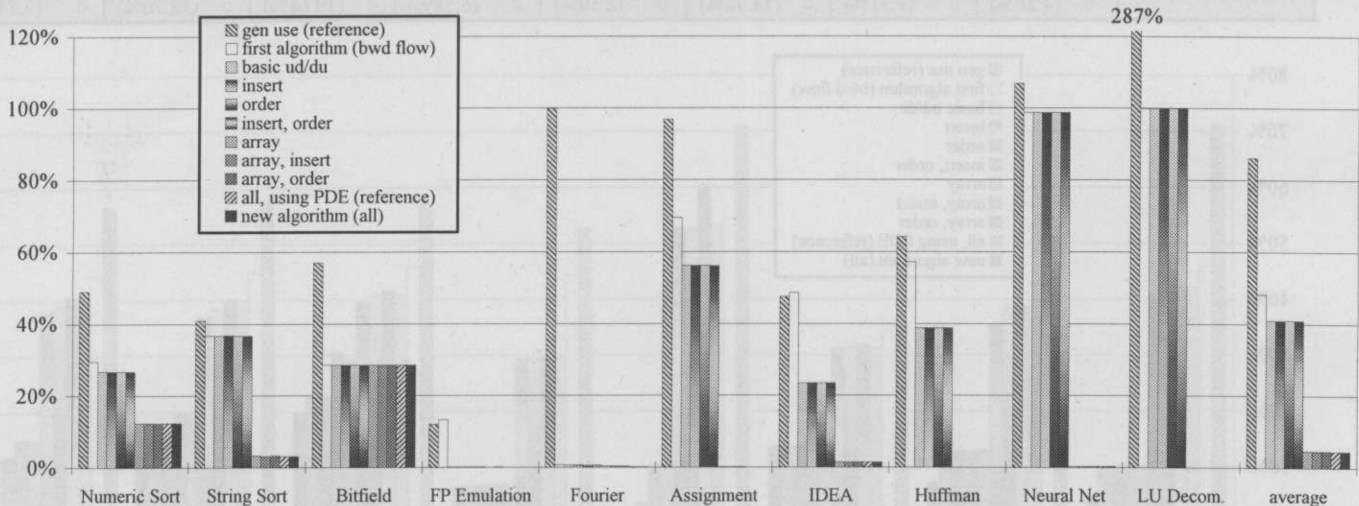


Figure 11. Dynamic counts of remaining 32-bit sign extensions for jBYTEmark (baseline=100%)

IA64, 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:

- Baseline: Disable sign extension optimizations (Figure 5(3)).
- Gen use (reference): Generate a sign extension before a use point at the code generation phase.
- First algorithm (bwd flow): Our first algorithm. It eliminates sign extensions using backward dataflow analysis.
- Basic ud/du: New algorithm, but disable all of sign extension insertion, order determination, and elimination for array indices.
- Insert: Enable sign extension insertion; that is, disable both order determination and elimination for array indices.
- Order: Enable order determination; that is, disable both sign extension insertion and elimination for array indices.
- Insert, order: Enable both "insert" and "order"; that is, disable elimination for array indices.
- Array: Enable elimination for array indices; that is, disable both sign extension insertion and order determination.
- Array, insert: Enable both "array" and "insert"; that is, disable sign extension insertion.
- Array, order: Enable both "array" and "order"; that is, disable order determination.

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

	mtrt	jess	compress	db	mpegaudio	jack	javac	average
baseline	19162530 (100.00%)	165578180 (100.00%)	1831002215 (100.00%)	298338613 (100.00%)	807174869 (100.00%)	95753436 (100.00%)	234753067 (100.00%)	(100.00%)
gen use (reference)	13619419 ○ (71.07%)	85606229 ○ (51.70%)	643756102 ○ (35.16%)	211989358 ○ (71.06%)	426398800 ○ (52.83%)	54596337 ○ (57.02%)	134261024 ○ (57.19%)	○ (56.58%)
first algorithm (bwd flow)	8800008 ○ (45.92%)	65389295 ○ (39.49%)	632370673 ○ (34.54%)	162360645 ○ (54.42%)	358862336 ○ (44.46%)	43946795 ○ (45.90%)	105245327 ○ (44.83%)	○ (44.22%)
basic ud/du	8042506 ○ (41.97%)	64268121 ○ (38.81%)	572297310 ○ (31.26%)	157713624 ○ (52.86%)	237743297 ○ (29.45%)	38839843 ○ (40.56%)	93920998 ○ (40.01%)	○ (39.28%)
insert	8091775 ● (42.23%)	64282490 ● (38.82%)	582159879 ● (31.79%)	180226907 ● (60.41%)	238661074 ● (29.57%)	39919110 ● (41.69%)	93400183 ○ (39.79%)	● (40.61%)
order	7962959 ○ (41.55%)	64268123 (38.81%)	572297310 (31.26%)	157713624 (52.86%)	237743297 (29.45%)	38839851 (40.56%)	90060780 ○ (38.36%)	○ (38.98%)
insert, order	7702613 ○ (40.20%)	58537108 ○ (35.35%)	571386557 ○ (31.21%)	157417853 ○ (52.76%)	237099838 ○ (29.37%)	37781407 ○ (39.46%)	86758551 ○ (36.96%)	○ (37.90%)
array	1961456 ○ (10.24%)	21544766 ○ (13.01%)	262105810 ○ (14.31%)	33105544 ○ (11.10%)	56354143 ○ (6.98%)	27999218 ○ (29.24%)	47560306 ○ (20.26%)	○ (15.02%)
array, insert	1997325 ● (10.42%)	21330983 ○ (12.88%)	262095499 (14.31%)	31514492 ○ (10.56%)	56287323 (6.97%)	29245657 ● (30.54%)	46826978 ○ (19.95%)	● (15.09%)
array, order	1815980 ○ (9.48%)	21451357 ○ (12.96%)	252222150 ○ (13.78%)	10557513 ○ (3.54%)	54729584 ○ (6.78%)	21714796 ○ (22.68%)	35732666 ○ (15.22%)	○ (12.06%)
all, using PDE (reference)	1811531 ○ (9.45%)	21249333 ○ (12.83%)	251323510 ○ (13.73%)	7794544 ○ (2.61%)	53638690 ○ (6.65%)	21533665 ○ (22.49%)	33544130 ○ (14.29%)	○ (11.72%)
new algorithm (all)	816153 ○ (4.26%)	12104429 ○ (7.31%)	251303625 ○ (13.72%)	7472849 ○ (2.50%)	53079482 ○ (6.58%)	18844058 ○ (19.68%)	29942890 ○ (12.76%)	○ (9.54%)

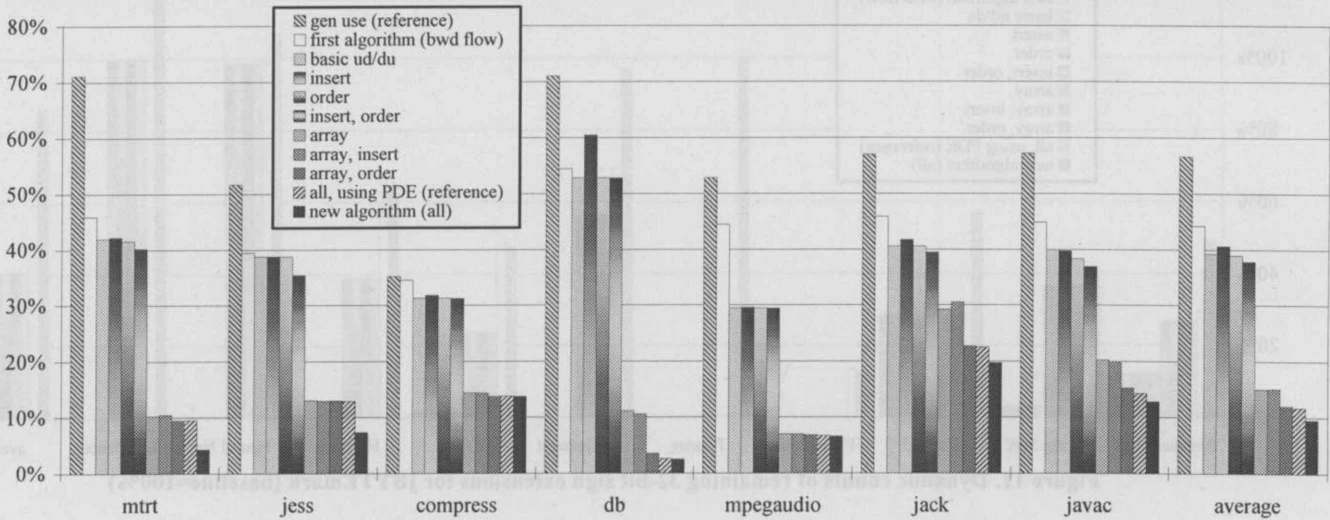


Figure 12. Dynamic counts of remaining 32-bit sign extensions for SPECjvm98 (baseline=100%)

- 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.

All the versions that disable the order determination perform the eliminations in the reverse depth first search order, the same order in which backward dataflow analysis is performed. **Figure 11** and **Figure 12** plot the percentages of dynamic counts over our baseline. As explained in Section 2, we generate a sign extension instruction after a definition point in order to improve the effectiveness of our sign extension elimination. For reference, we measured another method that generates a sign extension instruction before a use point (denoted as "gen use") at the code generation phase.

Overall, our algorithm (denoted as "new algorithm (all)") eliminates between 71.52% and 99.999% 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:

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 region is increased. The combination of the order determination and the sign extension insertion is particularly effective (1082443 vs. 24770) for FP Emulation in jBYTEmark. As we mentioned in Section 2.1, we also tried another insertion algorithm 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 13 and **Figure 14** show the performance improvement over our baseline for jBYTEmark and SPECjvm98, respectively.

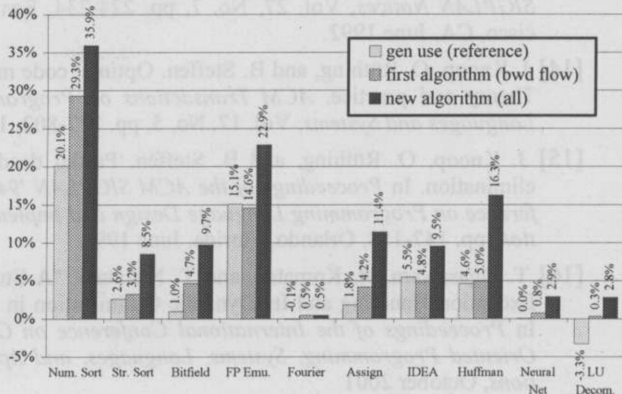


Figure 13. Performance Improvement for jBYTEmark

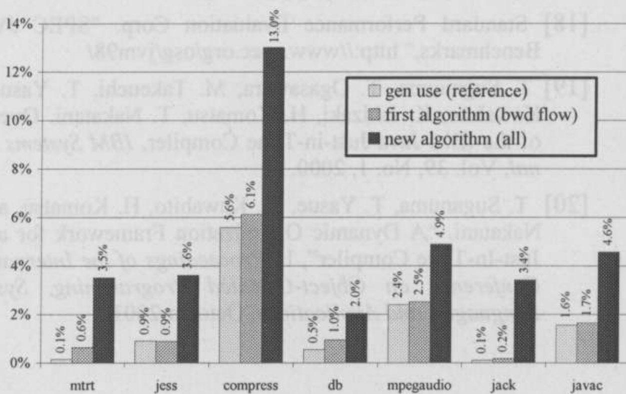


Figure 14. Performance Improvement for SPECjvm98

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 IA64, 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 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 13 and Figure 14.

Table 3. Breakdown of JIT compilation time

	Sign extension optimizations (all)	UD/DU chain creation	Others
mtrt	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%

5. PREVIOUS WORK

The Pentium version of GCC [17] 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 for our baseline. A partial dead code elimination (PDE) approach [15] turns out to be less effective than the simple insertion algorithm. **Figure 15** shows drawbacks 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-

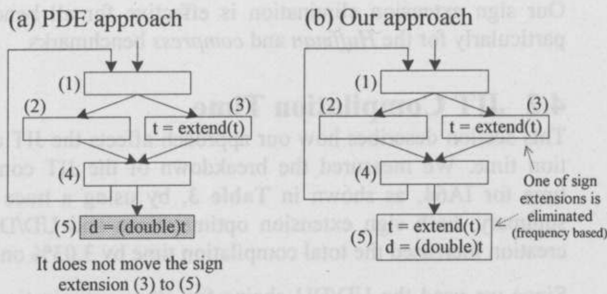


Figure 15. Drawbacks of the PDE approach

guided partial dead code elimination approach [6] would 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 is also applicable for other languages requiring sign extensions.

7. ACKNOWLEDGMENTS

We would like to thank 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.

8. REFERENCES

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

[2] T. Ball and J. R. Larus. "Optimally Profiling and Tracing Programs" In *Principles of Programming Languages*, 1992.

[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/iseries/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 – OOPSLA*, 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