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

読んだ人: みよしたけふみ

sign extension

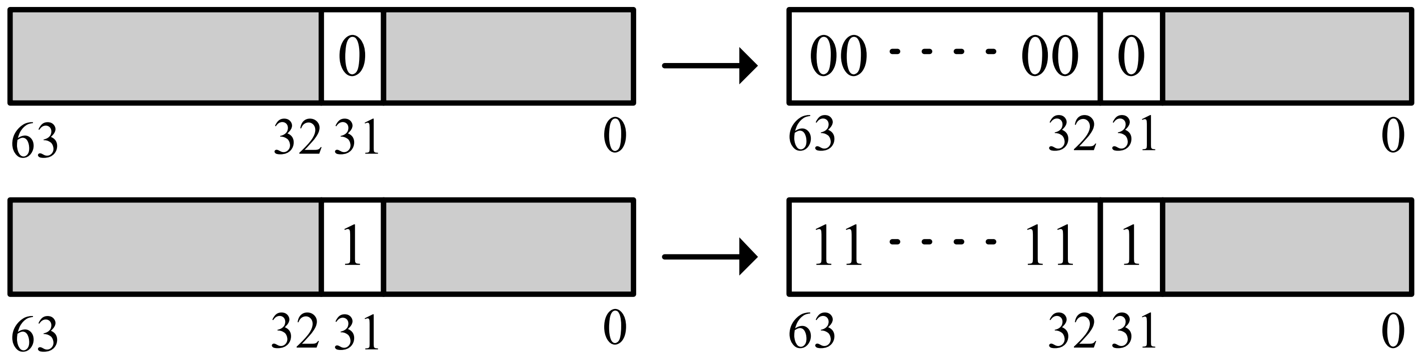
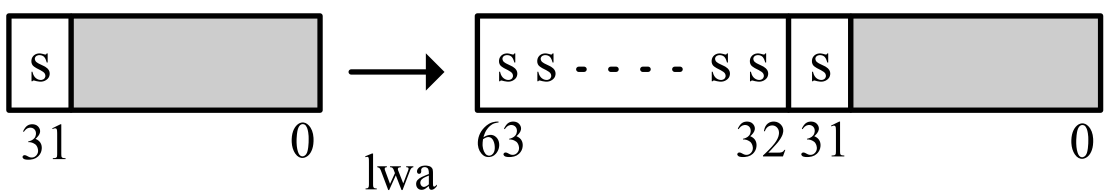


Figure 1. Sign extension of a 32-bit value to a 64-bit value

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-exetended
```

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

only no-affected extension

```
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

Contributions

- 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.

Flow Design

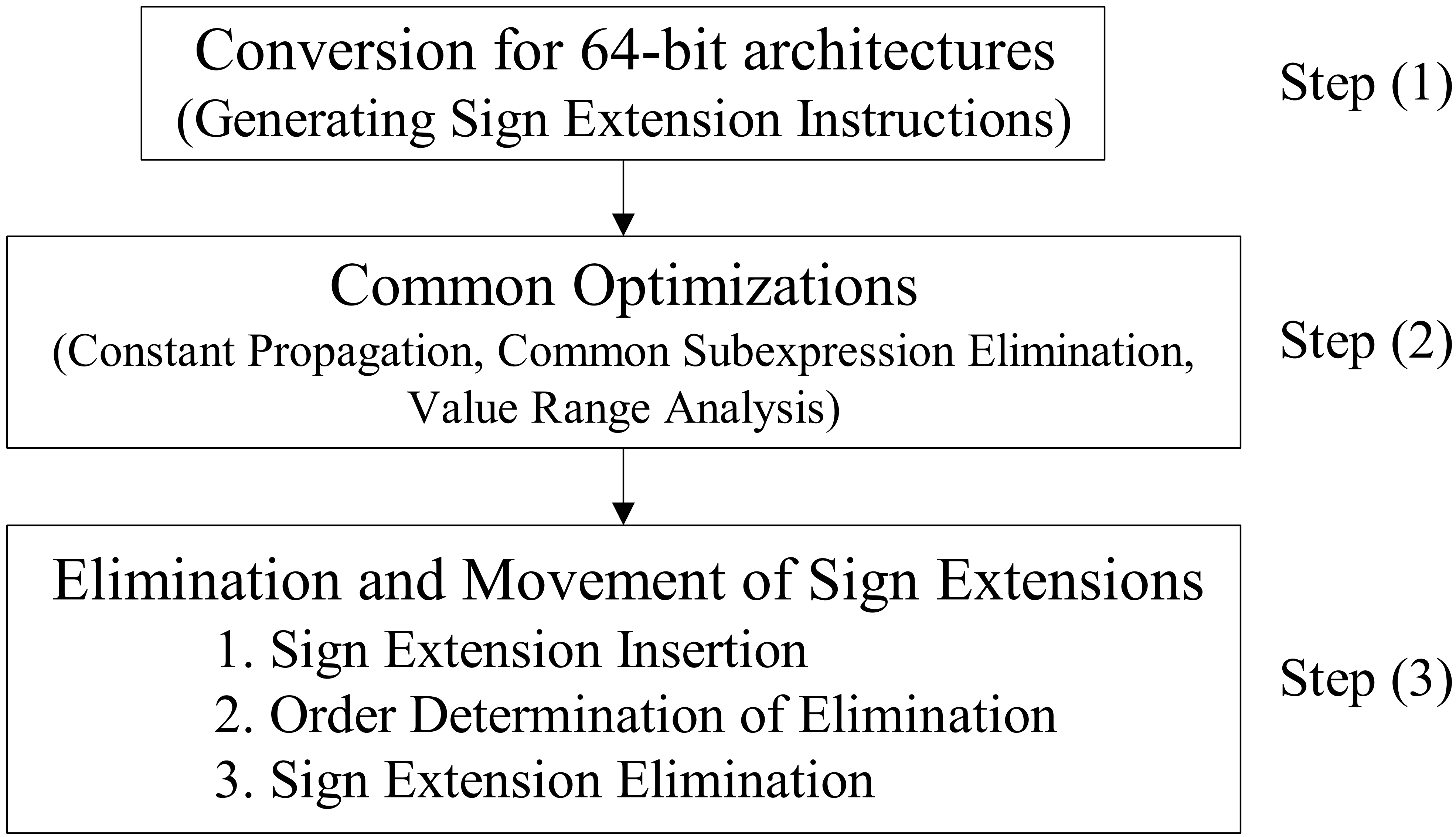


Figure 5. Flow diagram of our algorithm

Approaches

(a) Original program

```
t = i + j;
a[t + 1] = 0;
d = (double) t;
```

(b) Generate a sign extension after definition

(before elimination)	(after elimination)
t = i + j;	t = i + j;
t = extend(t);	**t = extend(t);**
t1 = t + 1;	t1 = t + 1;
t1 = extend(t1);	
a[t1] = 0;	a[t1] = 0;
d = (double) t;	d = (double) t;

(c) Generate a sign extension before use

(before elimination)	(after elimination)
t = i + j;	t = i + j;
t1 = t + 1;	t1 = t + 1;
t1 = extend(t1);	**t1 = extend(t1);**
a[t1] = 0;	a[t1] = 0;
t = extend(t);	**t = extend(t);**
d = (double) t;	d = (double) t;

Figure 6. Two approaches to generate sign extensions

(3)-1 Sign Extension Insertion

(a) Before insertion

```
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
i = extend(i);            - (1)
do {
    i = i - 1;            - (2)
    i = extend(i);        - (3)
    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 a sign extension for t
d = (double)t;            - (10)
```

(b) After insertion

```
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
i = extend(i);            - (1)
do {
    i = i - 1;            - (2)
    i = extend(i);        - (3)
    j = a[i];             - (4)
    i = just_extended(i) - (12)
    j = extend(j);        - (5)
    j = j & 0x0fffffff;   - (6)
    j = extend(j);        - (7)
    t += j;               - (8)
    t = extend(t);        - (9)
} while(i > start);
// need a sign extension for t
t = extend(t);            - (11)
d = (double)t;            - (10)
```

Figure 7. Example of inserting a sign extension

(a) Optimized result without insertion

```
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
do {
    i = i - 1;            - (2)
    j = a[i];             - (4)
    j = j & 0x0fffffff;   - (6)
    t += j;               - (8)
    t = extend(t);        - (9)
} while(i > start);
// need a sign extension for t
d = (double)t;            - (10)
```

(b) Optimized result with insertion

```
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
do {
    i = i - 1;            - (2)
    j = a[i];             - (4)
    j = j & 0x0fffffff;   - (6)
    t += j;               - (8)

} while(i > start);
// need a sign extension for t
t = extend(t);            - (11)
d = (double)t;            - (10)
```

Figure 8. The optimized result of Figure 7

(3)-2 Order determination

(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 requiring order determination

we eliminate sign extensions starting from the most frequently executed regions.

Sign Extension Elimination

The algorithm EliminateOneExtend analyzes and eliminates one sign extension by using UD/DU chains.

```
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;
}
```

Handling of Array Subscripts

#. Eliminated sign extensions

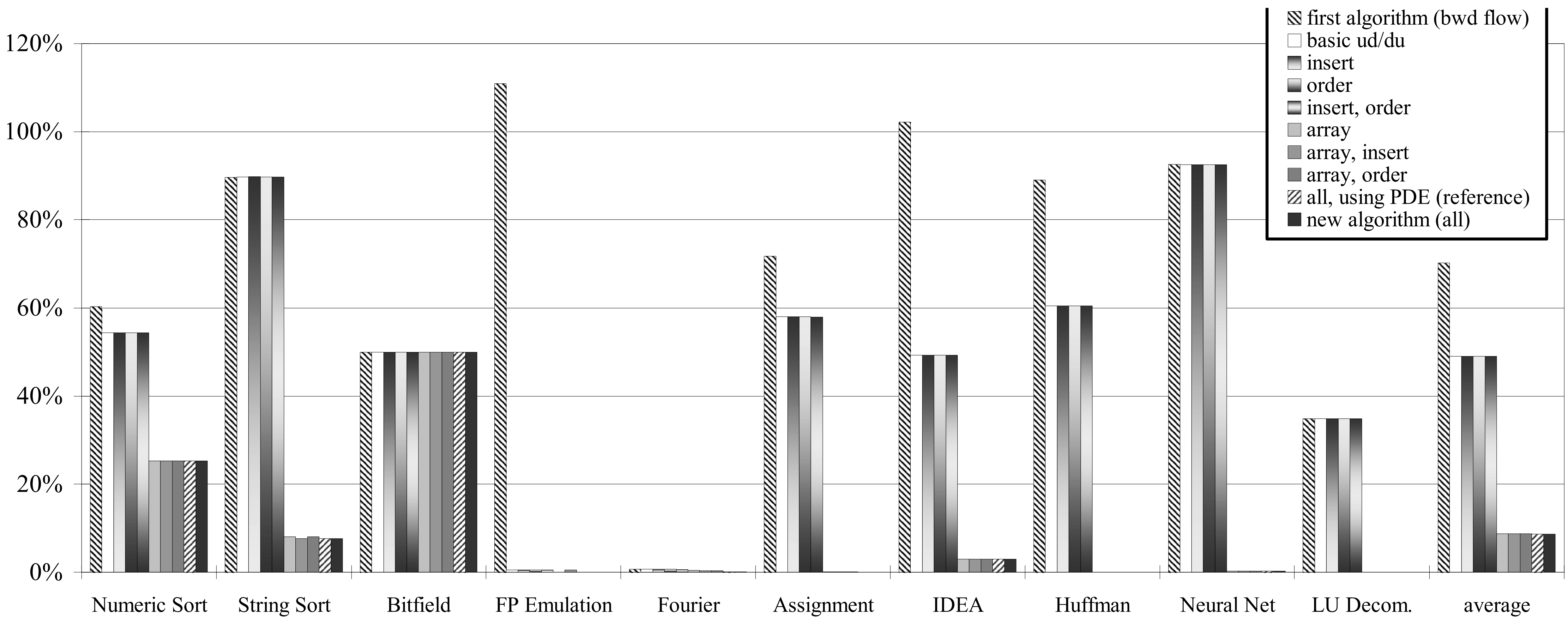


Figure 12. Dynamic counts of remaining 32-bit sign extensions for jBYTEmark (baseline=100%)

#. Eliminated sign extensions

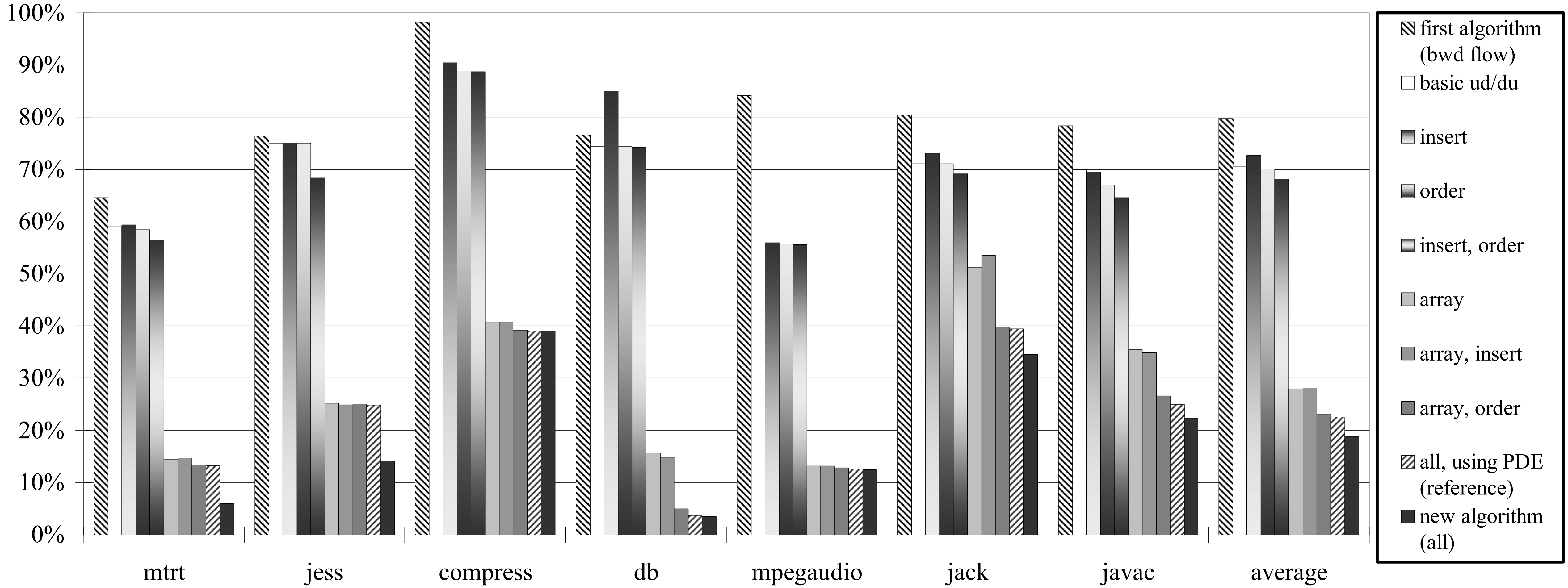


Figure 13. Dynamic counts of remaining 32-bit sign extensions for SPECjvm98 (baseline=100%)

Performance Improvement

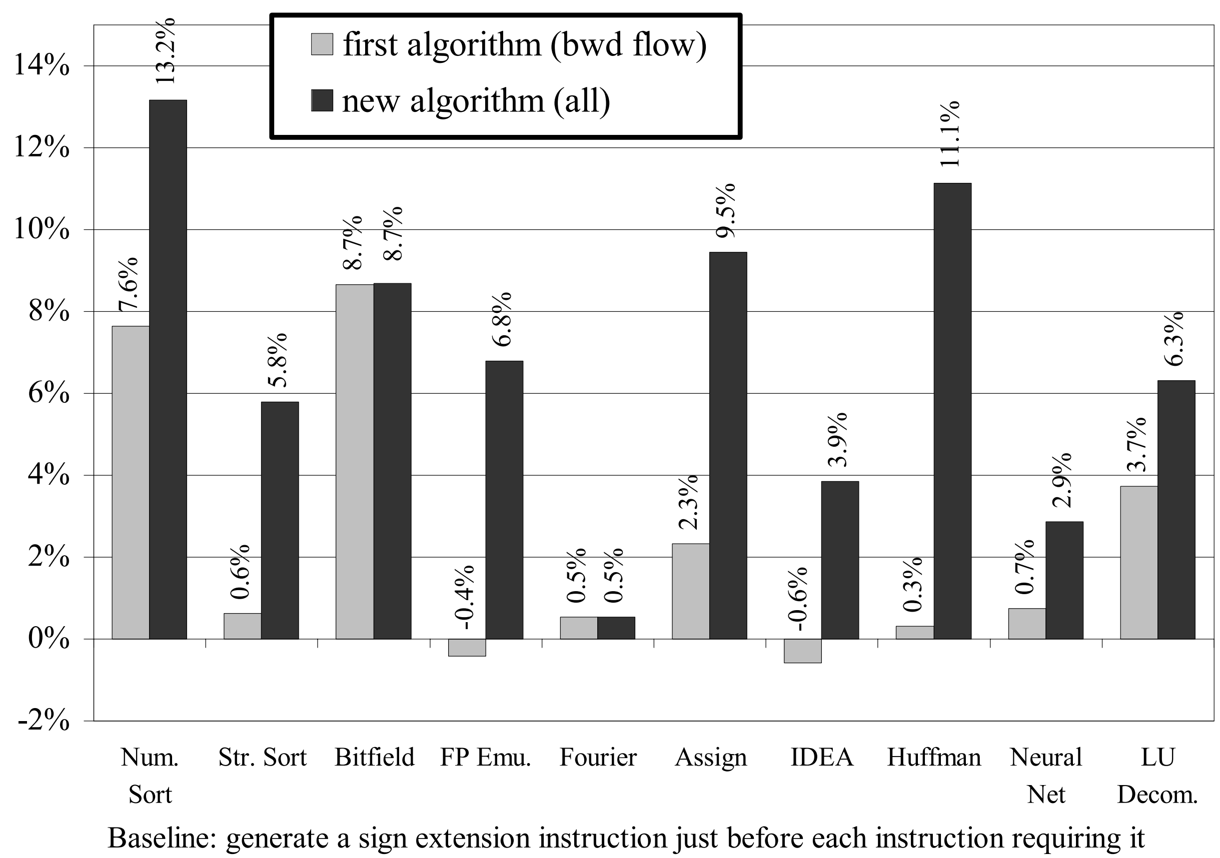


Figure 14. Performance Improvement for jBYTEmark

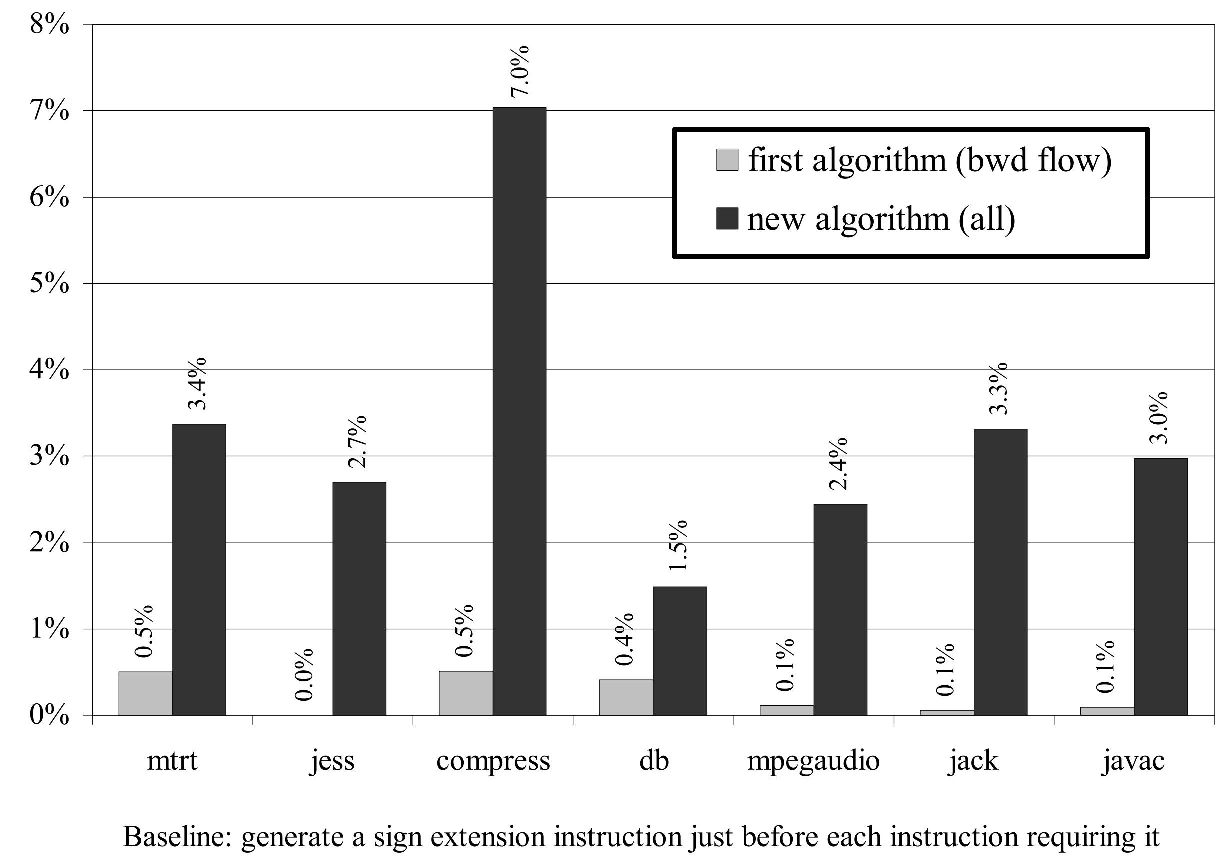


Figure 15. Performance Improvement for SPECjvm98