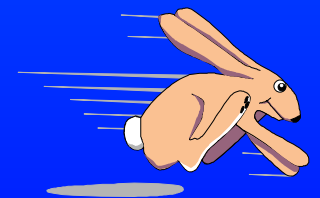


Effective Sign Extension Elimination

Motohiro Kawahito,
Hideaki Komatsu, and Toshio Nakatani

IBM Tokyo Research Laboratory



Many Misprints in the Paper!

- Many symbols lost

 $0 \square i \square 0x7fffffff$

- ▶ Lost symbols are like:

- \leq in many cases

$0 \square i \square 0x7fffffff \rightarrow 0 \leq i \leq 0x7fffffff$

- \in in some cases ("for loops" in algorithms)

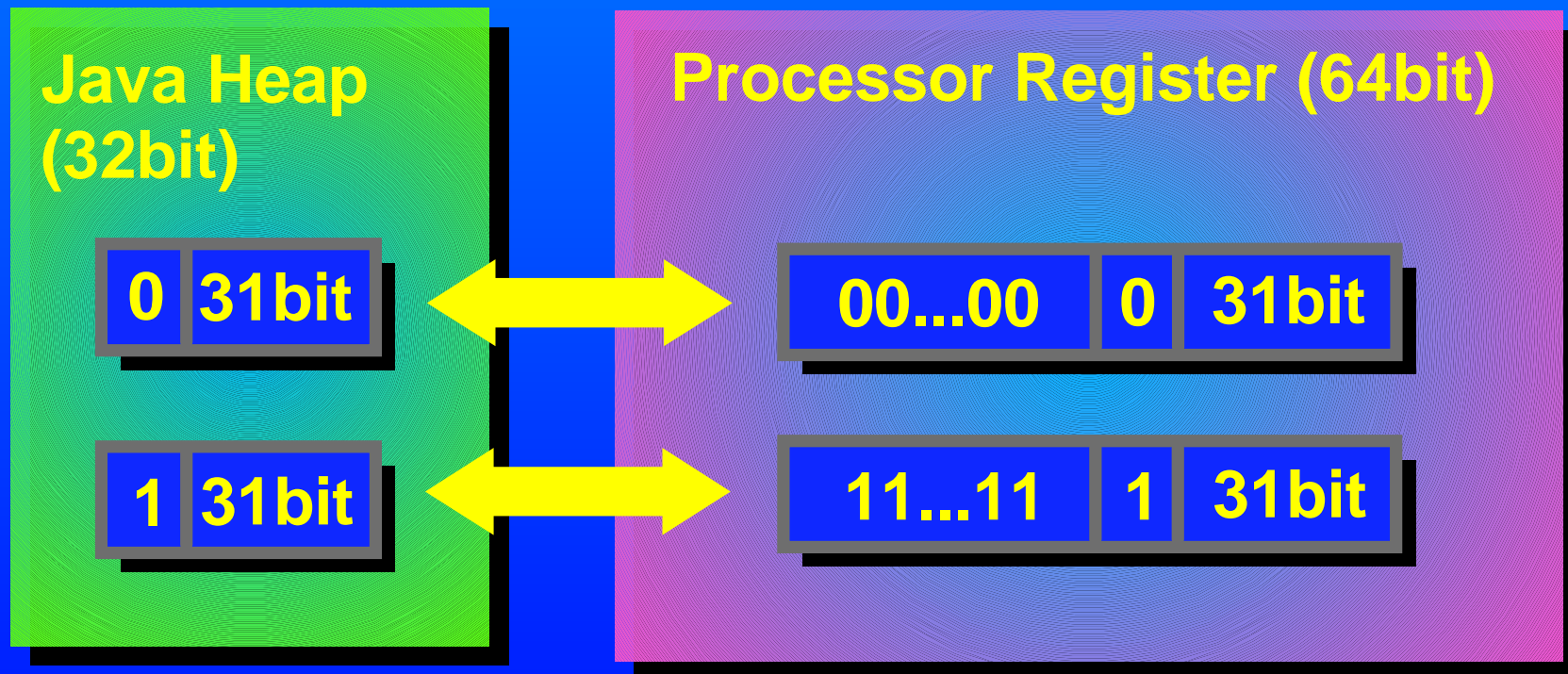
for ($J \square$ all ... \rightarrow for ($J \in$ all ...

Table of Contents

- **Motivation**
- **Our approach**
 - ▶ **Maximize the chance to place sign extensions at the right points**
 - ▶ **Eliminate sign extensions for array indices**
- **Experiments**
 - ▶ **jBYTEmark and SPECjvm98 measurements on IA64**
- **Conclusion**

What is Sign Extension?

- 32-bit applications on 64-bit architecture
 - ▶ Java specifies "int" as a 32-bit data type
 - ▶ Load or calculation requires sign extension



Calculation is Wrong w/o Sign Extension

$i = 7\text{FFFFFFF}$

00....00

0

11....11

$t = i + 1;$

$t = 80000000$

00....00

1

00....00

$s = t / 2;$

$s = 40000000$

00....00

0

10....00

Incorrect!!

Calculation is Wrong w/o Sign Extension

$i = 7FFFFFFF$

00....00	0	11....11
----------	---	----------

$t = i + 1;$

$t = 80000000$

00....00	1	00....00
----------	---	----------

$t = \text{extend}(t);$ $t = FFFFFFFF80000000$

11....11	1	00....00
----------	---	----------

$s = t / 2;$

$s = FFFFFFFFC0000000$

11....11	1	10....00
----------	---	----------

Correct!!

Simple Insertion Strategy

- Def approach : Insert a sign extension **after** each **definition** point
- Use approach : Insert a sign extension **before** each **use** point

Basic Concepts of Elimination

- A sign extension can be eliminated, if
 - ▶ Upper 32 bits of the output variable do not affect any use of the variable, or
 - ▶ Input variable is already sign-extended

First case

```
i = extend(i); // eliminated  
:  
a[0] = i; // 32-bit store
```

Second case

```
i = j & 0xFF;  
:  
i = extend(i); // eliminated
```

Our First Elimination Ideas

- Def approach
 - ▶ Using a **backward** dataflow analysis
 - ▶ Upper 32 bits of the output variable do not affect any use of the variable
- Use approach
 - ▶ Using a **forward** dataflow analysis
 - ▶ Input variable is already sign-extended

Example 1: Definition Points & Backward

■ Original program

```
int t = 0;
int i = j + k;
do {
    i = i + 1;
    t += a[ i ];
} while(i < end);
d = (double) t;
```

■ After insertion

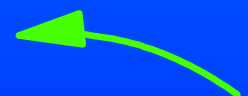
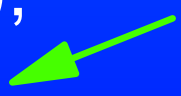
```
int t = 0;
int i = j + k;
i = extend ( i );
do {
    i = i + 1;
    i = extend( i );
    t += a[ i ];
    t = extend( t );
} while(i < end);
d = (double) t;
```

Example 1: Definition Points & Backward

▪ After insertion

```
int t = 0;  
int i = j + k;  
i = extend( i );  
do {  
    i = i + 1;  
    i = extend( i );  
    t += a[ i ];  
    t = extend( t );  
} while(i < end);  
d = (double) t;
```

▪ After backward elimination

```
int t = 0;  
int i = j + k;  
  
do {  
    i = i + 1;  
    i = extend( i );  
    t += a[ i ];   
    t = extend( t );  
} while(i < end);  
d = (double) t; 
```

**Require
sign extension**

Example 2: Use Points & Forward

■ Original program

```
int t = 0;
int i = j + k;
do {
    i = i + 1;
    t += a[ i ];
} while(i < end);
d = (double) t;
```

■ After insertion

```
int t = 0;
int i = j + k;
do {
    i = i + 1;
    i = extend( i );
    t += a[ i ];
} while(i < end);
t = extend( t );
d = (double) t;
```

Example 2: Use Points & Forward

■ After insertion

```
int t = 0;
int i = j + k;
do {
    i = i + 1;
    i = extend( i );
    t += a[ i ];
} while(i < end);
t = extend( t );
d = (double) t;
```

■ After forward elimination

```
int t = 0;
int i = j + k;
do {
    i = i + 1;
    i = extend( i );
    t += a[ i ];
} while(i < end);
t = extend( t );
d = (double) t;
```

Results are not sign-extended

Further Optimization is Possible!

■ Def & Backward

```
int t = 0;
int i = j + k;

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

d = (double) t;
```

■ Use & Forward

```
int t = 0;
int i = j + k;

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

■ Optimization Goal

```
int t = 0;
int i = j + k;
i = extend( i );
do {
    i = i + 1;
    t += a[ i ];
} while(i < end);
t = extend( t );
d = (double) t;
```

Our Approach

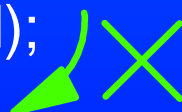
- **Maximize the chance to place sign extensions at the right points**
 - ▶ **Insert sign extensions at both def and use points**
 - ▶ **Sort the basic blocks in the execution frequency**
- **Eliminate sign extensions for array indices by using Java semantics**
 - ▶ **Array bounds checking excludes a negative index**

Insert Sign Extensions at both Def and Use Points

- To insert sign extensions at either def or use points makes fewer candidates

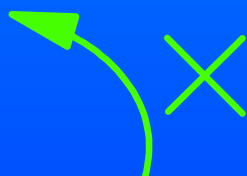
(Def Points Only)

```
int t = 0;  
int i = j + k;  
i = extend( i );  
do {  
    i = i + 1;  
    i = extend( i );  
    t += a[ i ];  
    t = extend( t );  
} while(i < end);  
d = (double) t;
```



(Use Points Only)

```
int t = 0;  
int i = j + k;  
  
do {  
    i = i + 1;  
    i = extend( i );  
    t += a[ i ];  
} while(i < end);  
t = extend( t );  
d = (double) t;
```



(Optimization Goal)

```
int t = 0;  
int i = j + k;  
i = extend( i );  
do {  
    i = i + 1;  
    t += a[ i ];  
} while(i < end);  
t = extend( t );  
d = (double) t;
```

Example: Inserting Sign Extensions

After Insertion

```
int t = 0;  
int i = j + k;  
i = extend ( i );  
do {  
    i = i + 1;  
    i = extend( i );  
    t += a[ i ];  
    t = extend( t );  
} while(i < end);  
t = extend ( t );  
d = (double) t;
```

- To insert sign extensions at both definition and use points increases candidates of elimination

*One of them
can be eliminated*

*One of them
can be eliminated*

Sort Basic Blocks in the Execution Frequency

- **Eliminate sign extensions from the most frequently executed region**
 - ▶ **Estimate frequency for each basic block by:**
 - loop nest level
 - probability of each conditional branch
(enhanced with the runtime profile information)
 - ▶ **Use UD/DU-chain to selectively eliminate sign extensions**
 - Dataflow analysis cannot selectively eliminate them

Example: Order Determination

- In the loop first, others second

```
int i = j + k;
```

```
i = extend( i );
```

```
do {
```

```
    i = i + 1;
```

```
    i = extend( i );
```

```
    t += a[ i ];
```

```
    t = extend( t );
```

```
} while(i < end);
```

```
t = extend( t );
```

```
d = (double) t;
```

(1)

(2)

Result of Frequency-based Elimination

- Sign extension for the array index still remains in the loop

(Before Elimination)

```
int i = j + k;  
i = extend( i );  
do {  
    i = i + 1;  
    i = extend( i );  
    t += a[ i ];  
    t = extend( t );  
} while(i < end);  
t = extend( t );  
d = (double) t;
```

(After Elimination)

```
int i = j + k;  
  
do {  
    i = i + 1;  
    i = extend( i );  
    t += a[ i ];  
  
} while(i < end);  
t = extend( t );  
d = (double) t;
```

Still
Remains!!



Use Java Semantics for Array Indices (First semantic)

- *i* is considered sign-extended after **a[i]**
 - ▶ We insert a dummy sign extension after each array access

```
t += a[ i ];
```

```
// i is considered sign-extended here
```

Insert Dummy Sign Extensions

- Insert a dummy sign extension after each array access to tell that an array index is sign-extended

```
int i = j + k;
i = extend( i );
do {
    i = i + 1;
    i = extend( i );
    t += a[ i ];
    i = dummy_extend( i ); // i is considered sign-extended here
    t = extend( t );
} while(i < end);
t = extend( t );
d = (double) t;
```

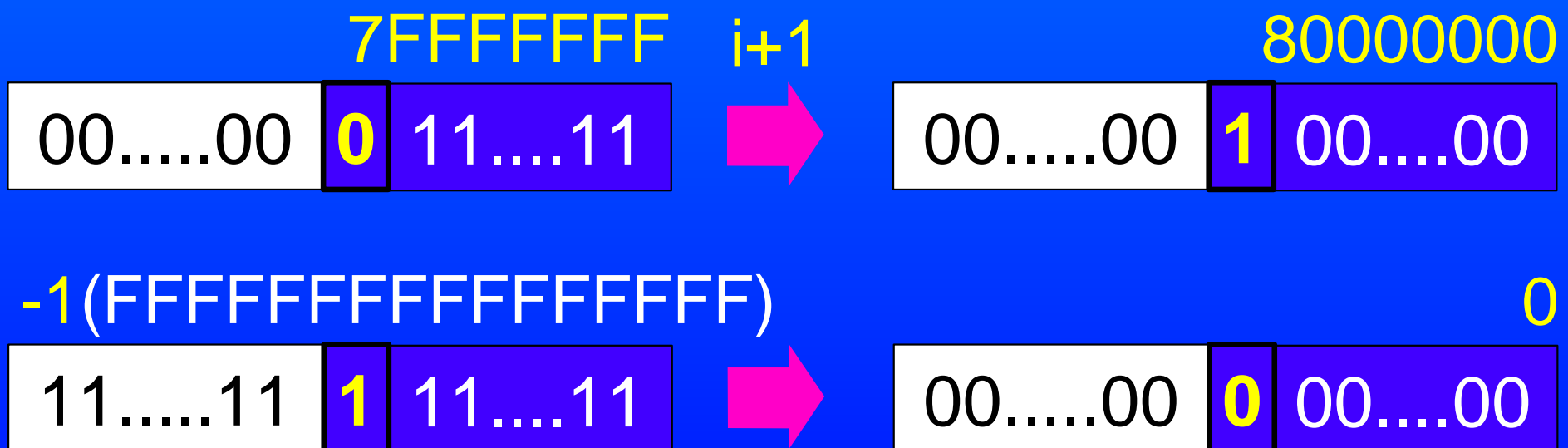
Insert Dummy Sign Extensions

- Insert a dummy sign extension after each array access to tell that an array index is sign-extended

```
int i = j + k;  
i = extend( i );  
do {  
    i = i + 1;  
    i = extend( i ); ← Our goal is to eliminate this!  
    t += a[ i ];  
    i = dummy_extend( i ); // i is considered sign-extended here  
    t = extend( t );  
} while(i < end);  
t = extend( t );  
d = (double) t;
```

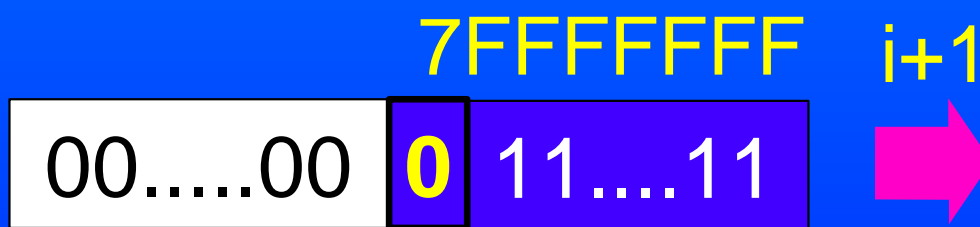
Use Java Semantics for Array Indices (Second semantic)

- Sign extension for the array index "**i+1**" is not necessary if **i** is sign-extended, because:
 - ▶ Sign extension is necessary only when $i = 7\text{ffffff}$
 - ▶ Array bounds checking using a 32-bit compare instruction excludes a negative index

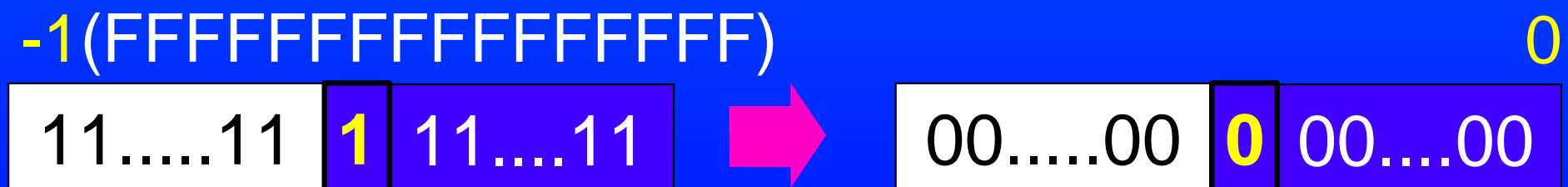


Use Java Semantics for Array Indices (Second semantic)

- Sign extension for the array index "**i+1**" is not necessary if **i** is sign-extended, because:
 - ▶ Sign extension is necessary only when $i = 7\text{ffffff}$
 - ▶ Array bounds checking using a 32-bit compare instruction excludes a negative index



Array bounds checking
excludes this case



Example: Elimination for Array Indices

- Sign extension for the array index "**i+1**" is not necessary if **i** is sign-extended

```
int i = j + k;
```

```
i = extend( i );
```

```
do {
```

```
    i = i + 1;
```

```
    i = extend( i );
```

```
    t += a[ i ];
```

```
    i = dummy_extend( i );
```

```
    t = extend( t );
```

```
} while( i < end );
```

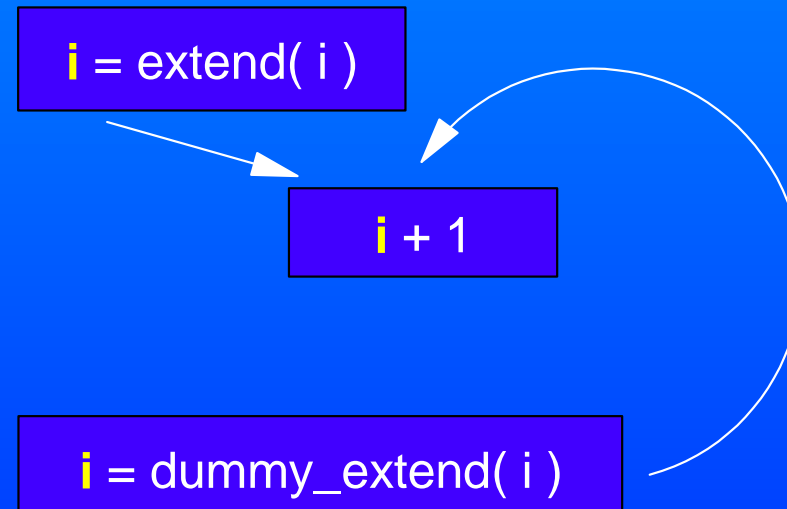
```
t = extend( t );
```

```
d = (double) t;
```

- (1) **i = extend(i)**

- (2) **i + 1**

- (3) **i = dummy_extend(i)**



Example: Our Optimization

■ Final result

```
int i = j + k;
```

```
i = extend( i );
```

```
do {
```

```
    i = i + 1;
```

```
    i = extend( i ); ←
```

```
    t += a[ i ];
```

```
    t = extend( t ); ←
```

```
} while( i < end );
```

```
t = extend( t );
```

```
d = (double) t;
```

Eliminated using
Java semantics for array indices
and frequency-based elimination

Eliminated using
frequency-based elimination

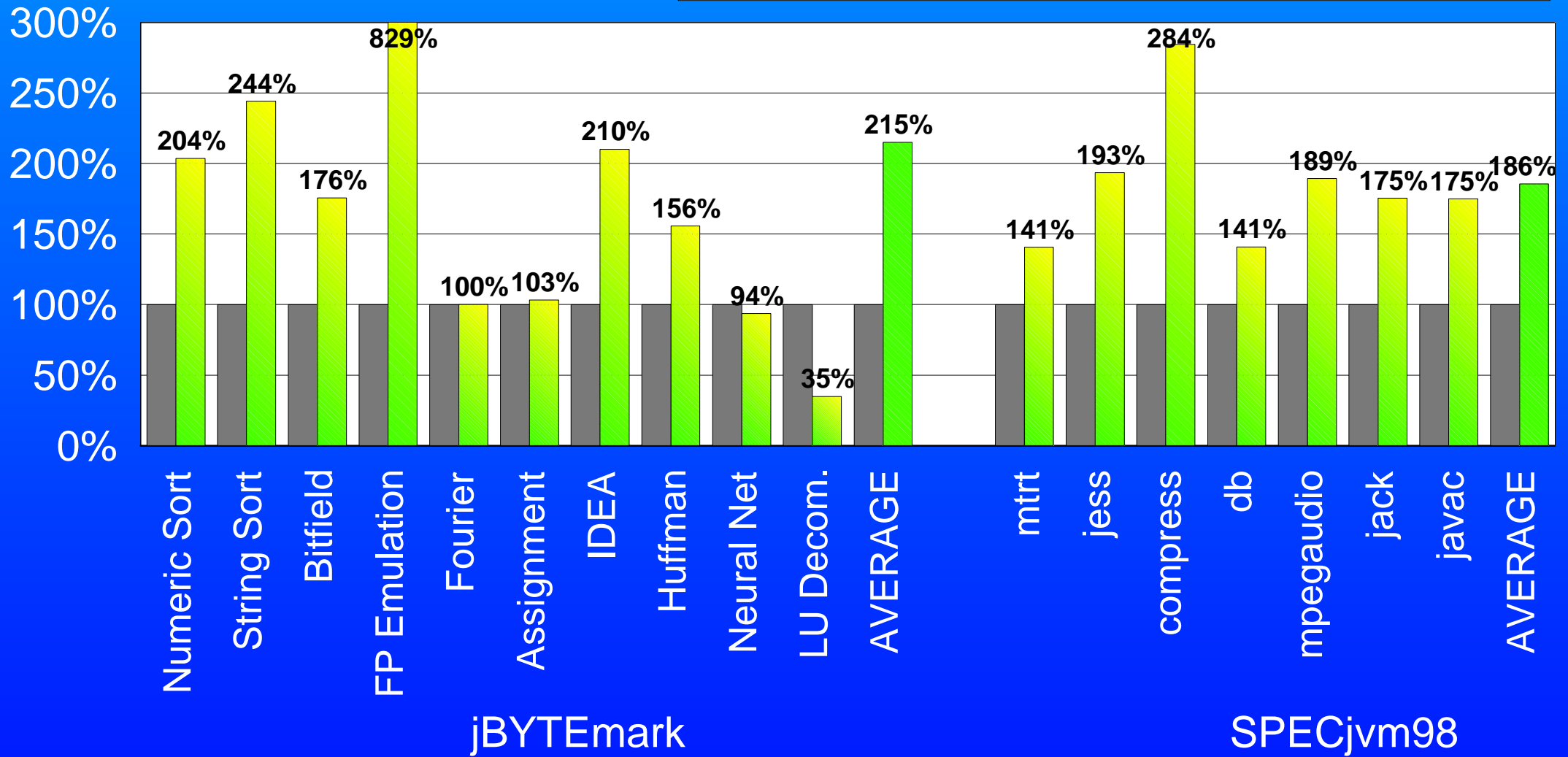
Experimental Result

- Measure jBYTEmark and SPECjvm98 on IA64 (Itanium)
- **Our baseline:**
Insert a sign extension before each use point whenever necessary

Dynamic Counts of Sign Extensions Executed by Two Insertion Strategies (w/o Elimination)

Lower bars are better
(Baseline=100%)

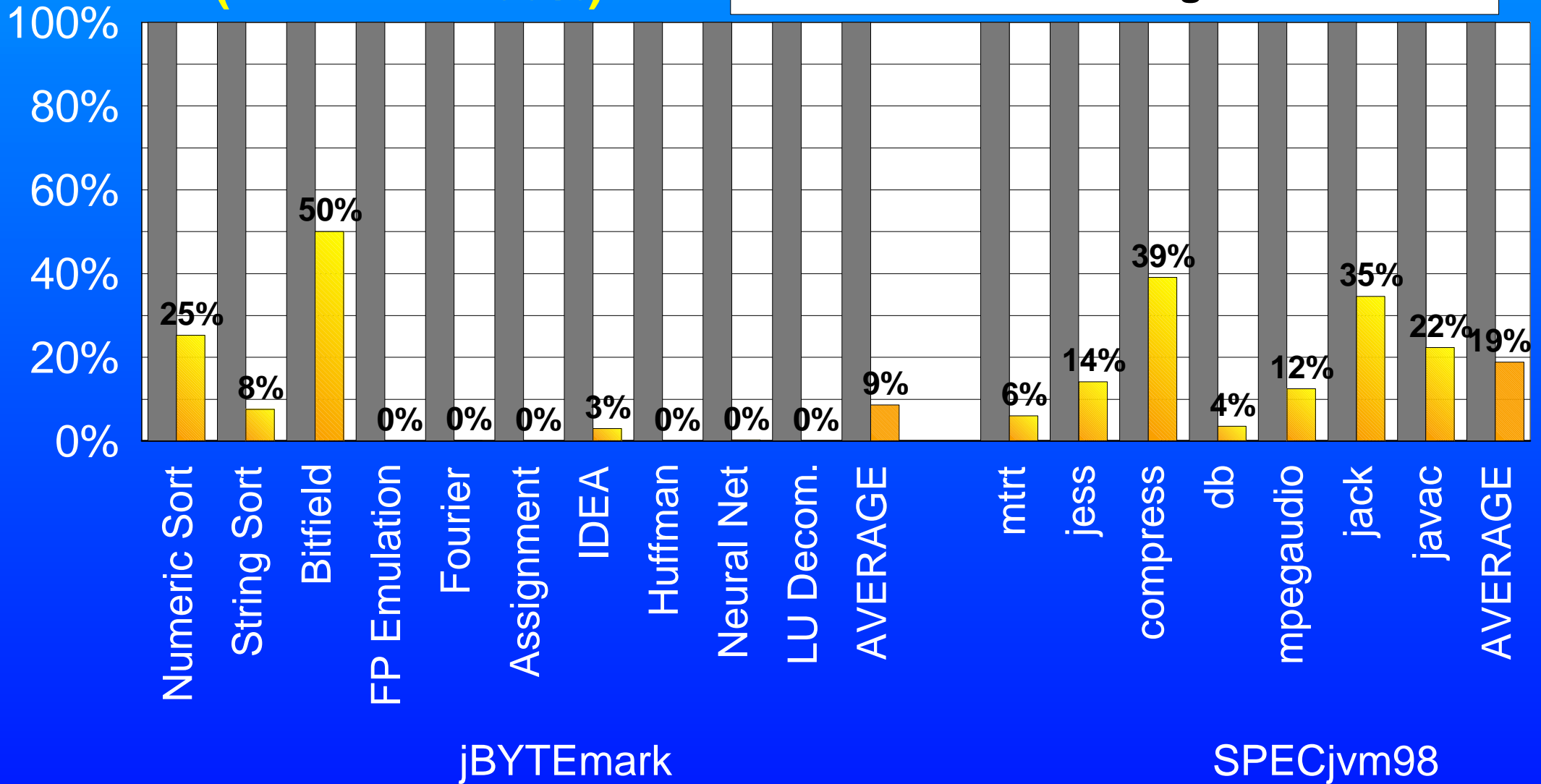
Baseline (Use point)
Definition point



Dynamic Counts of Sign Extensions Executed by Our New Algorithm

Lower bars are better
(Baseline=100%)

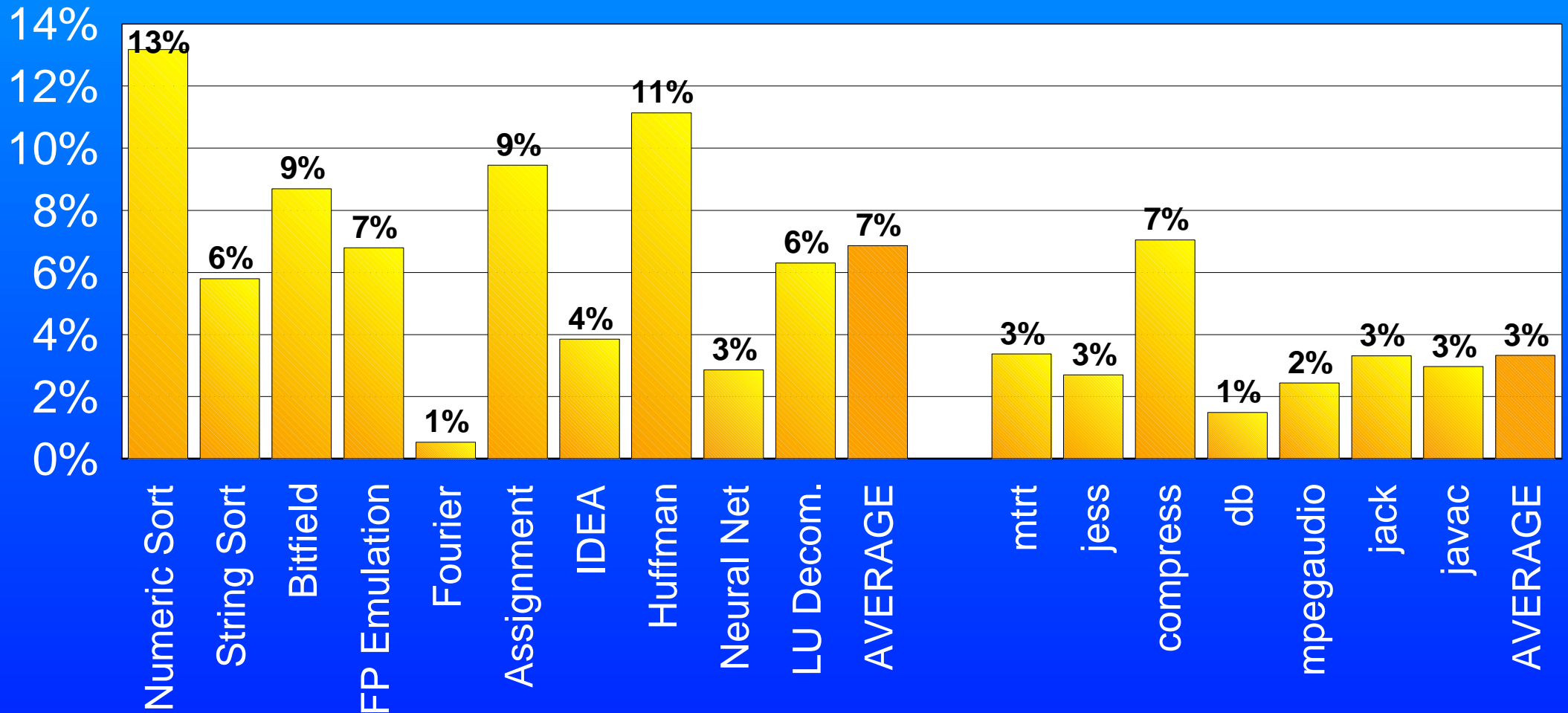
Baseline
Our New Algorithm



Performance Improvement

Taller bars are better
(Baseline=0%)

Our New Algorithm



jBYTEmark

SPECjvm98

Breakdown of JIT Compilation Time

	Average for jBYTEmark	Average for SPECjvm98
Sign Extension Optimizations	0.10%	0.13%
UD/DU chain creations	3.18%	2.55%
Others	96.73%	97.32%

Conclusion

- Our approach eliminates many sign extensions **(86%)** with little additional compilation time overhead **(0.11%)**
 - ▶ Maximize the chance to place sign extensions at the right points
 - ▶ Eliminate sign extensions for array indices by using Java semantics

Backup

Array Bound Checking Excludes a Negative Index

- Bound checking excludes "i = 0x7fffffff" case
 - ▶ We use a 32-bit compare instruction for the check
 - This instruction allows us to implement array bounds checking without sign extension

```
i = i + 1; // result = 0x80000000
```

```
i = extend(i);
```

```
boundcheck i, a[ ]; // 32-bit compare instruction
```

```
... = a[ i ]; // i is always sign-extended here
```

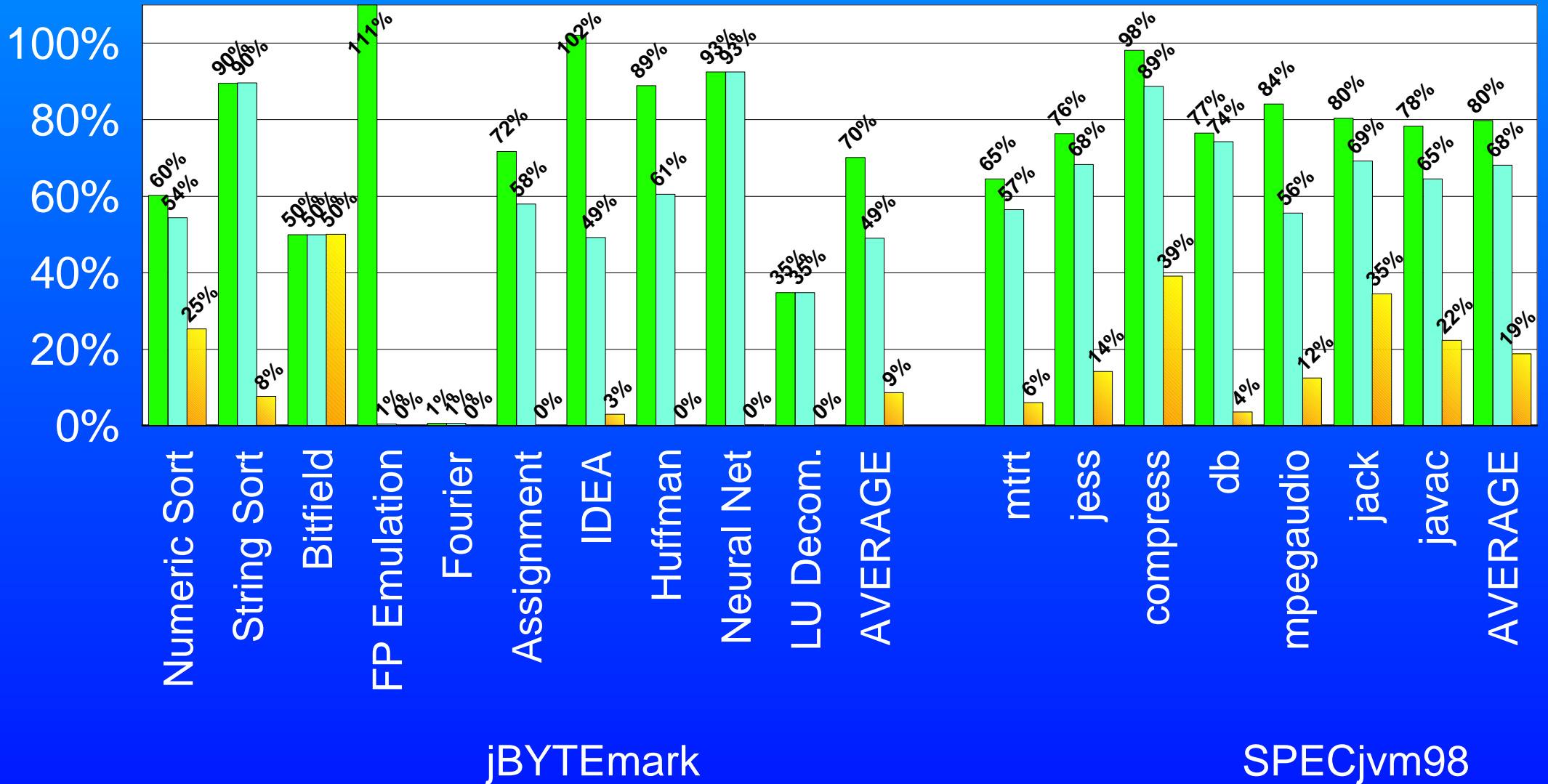
Exception
handler



Dynamic Counts of Sign Extensions Executed by Some Algorithms

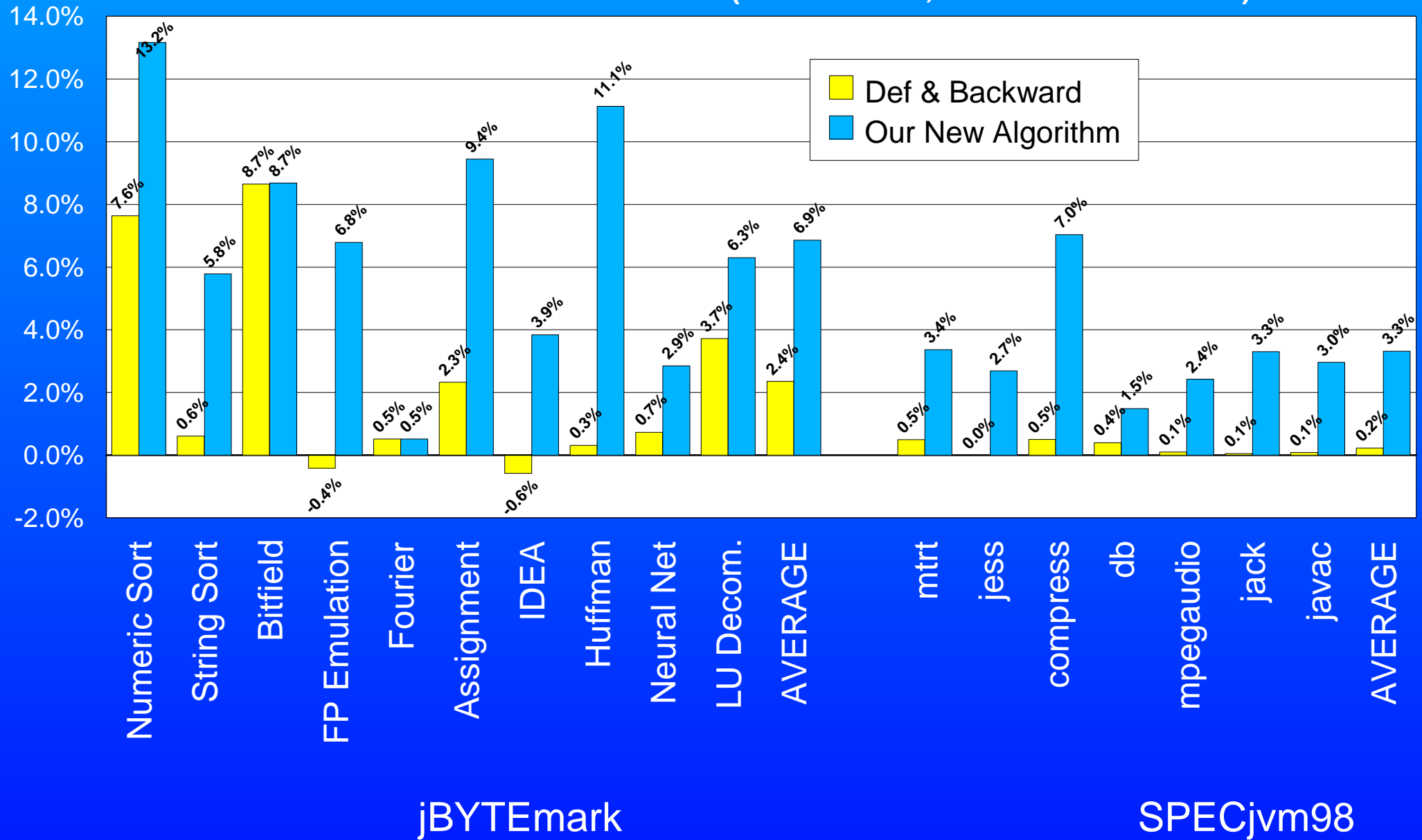
Lower bars are better
(Baseline=100%)

■ Def & Backward
 ■ No Array
 ■ Our New Algorithm



Performance Improvement

(Baseline=0%, Taller bars are better)



Flow Diagram of Our Optimizations

Sign Extension Insertion at Definition Points

```
graph TD; A[Sign Extension Insertion at Definition Points] --> B[General Optimizations]; B --> C["Optimizing Sign Extensions  
1. Sign Extension Insertion at Use Points  
2. Order Determination of Elimination  
3. Sign Extension Elimination"];
```

General Optimizations

Optimizing Sign Extensions

1. Sign Extension Insertion at Use Points
2. Order Determination of Elimination
3. Sign Extension Elimination

Insertion Algorithm: PDE vs Use

Sign Extension Insertion at Definition Points



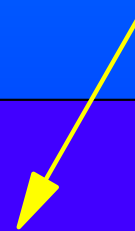
General Optimizations



Optimizing Sign Extensions

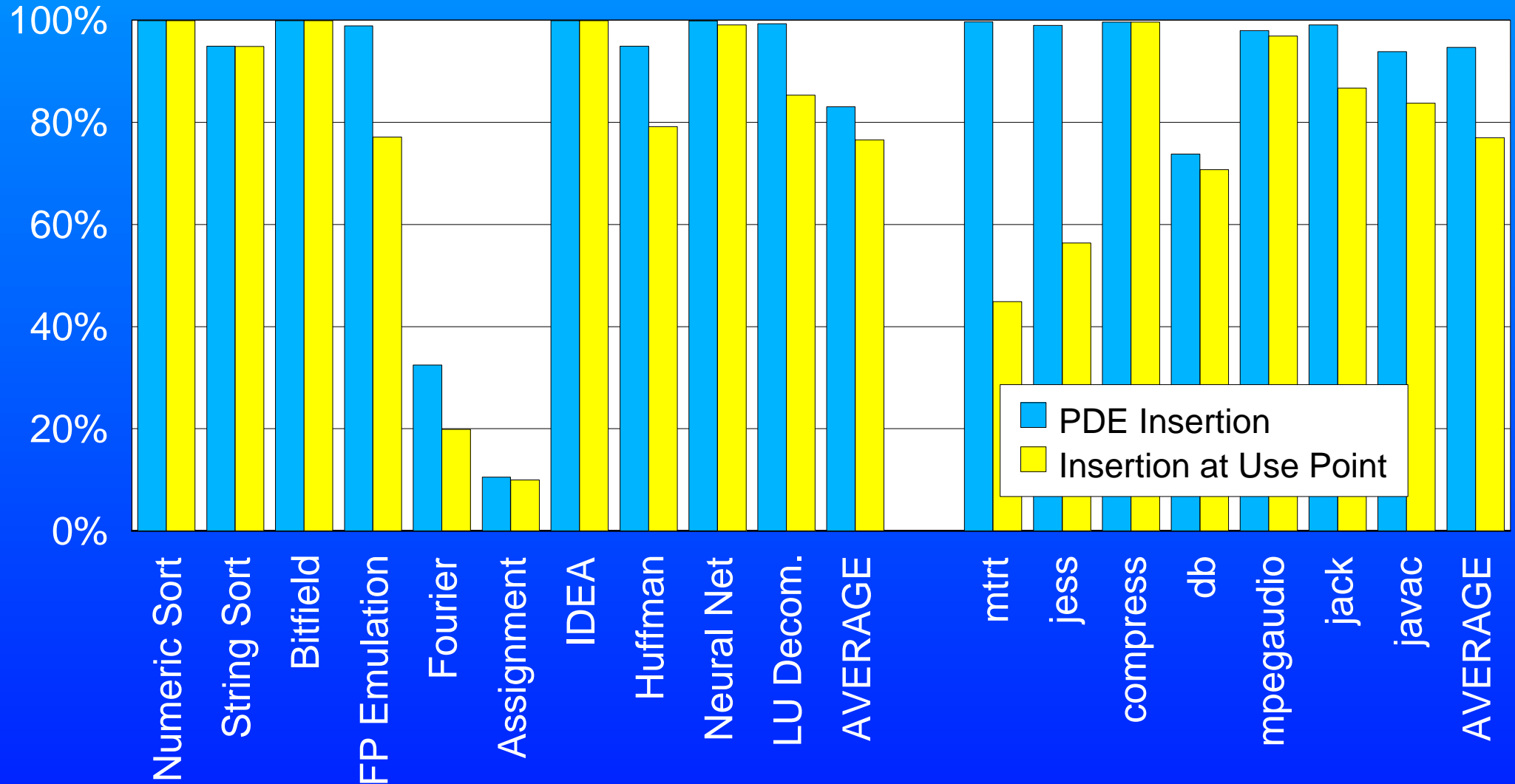
1. Sign Extension Insertion at Use Points
2. Order Determination of Elimination
3. Sign Extension Elimination

We tried insertion algorithm of partial dead code elimination



Insertion Algorithm: PDE vs Use (dynamic counts)

(No Insertion At Use=100%, Lower bars are better)

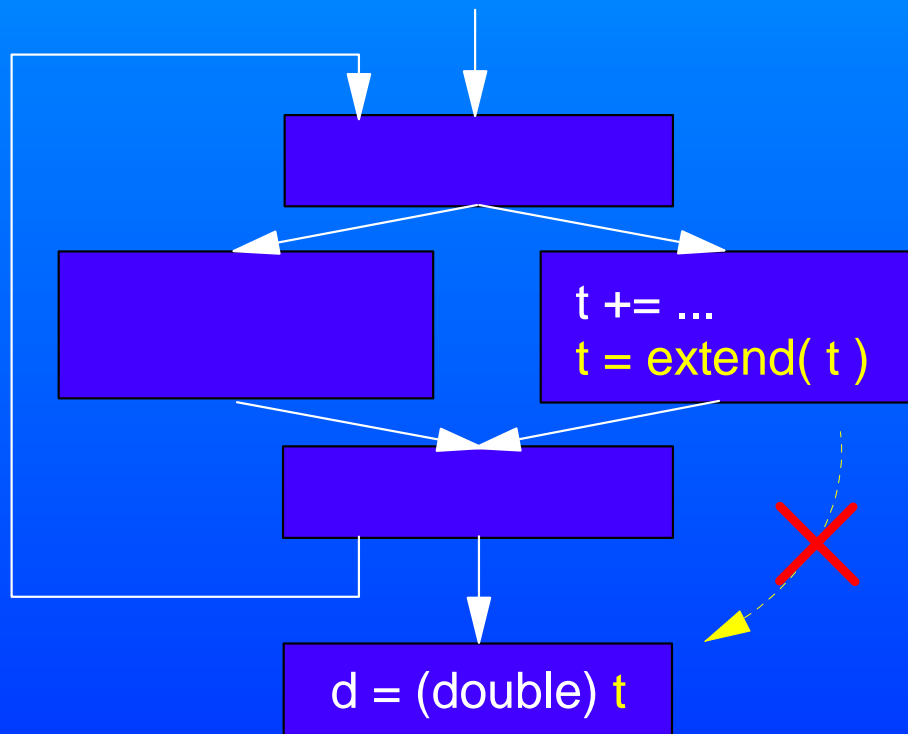


jBYTEmark

SPECjvm98

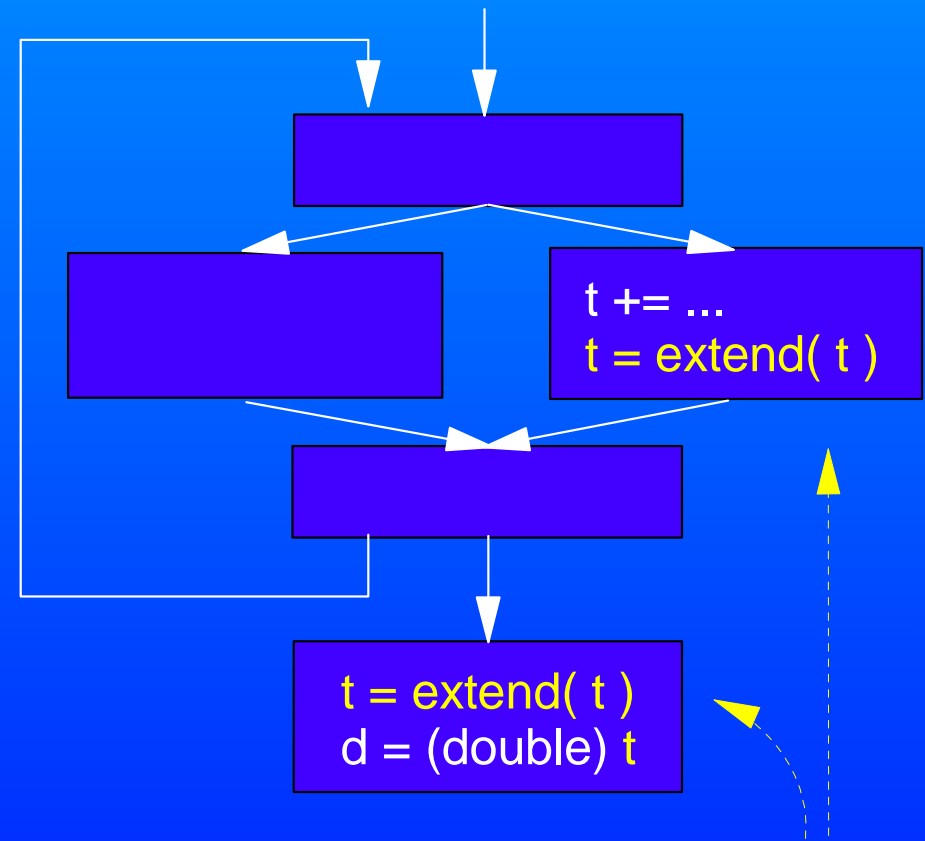
Limitation of a PDE approach

■ PDE approach



It does not move the sign extension out of the loop

■ Our approach



One of sign extensions is eliminated (frequency based)