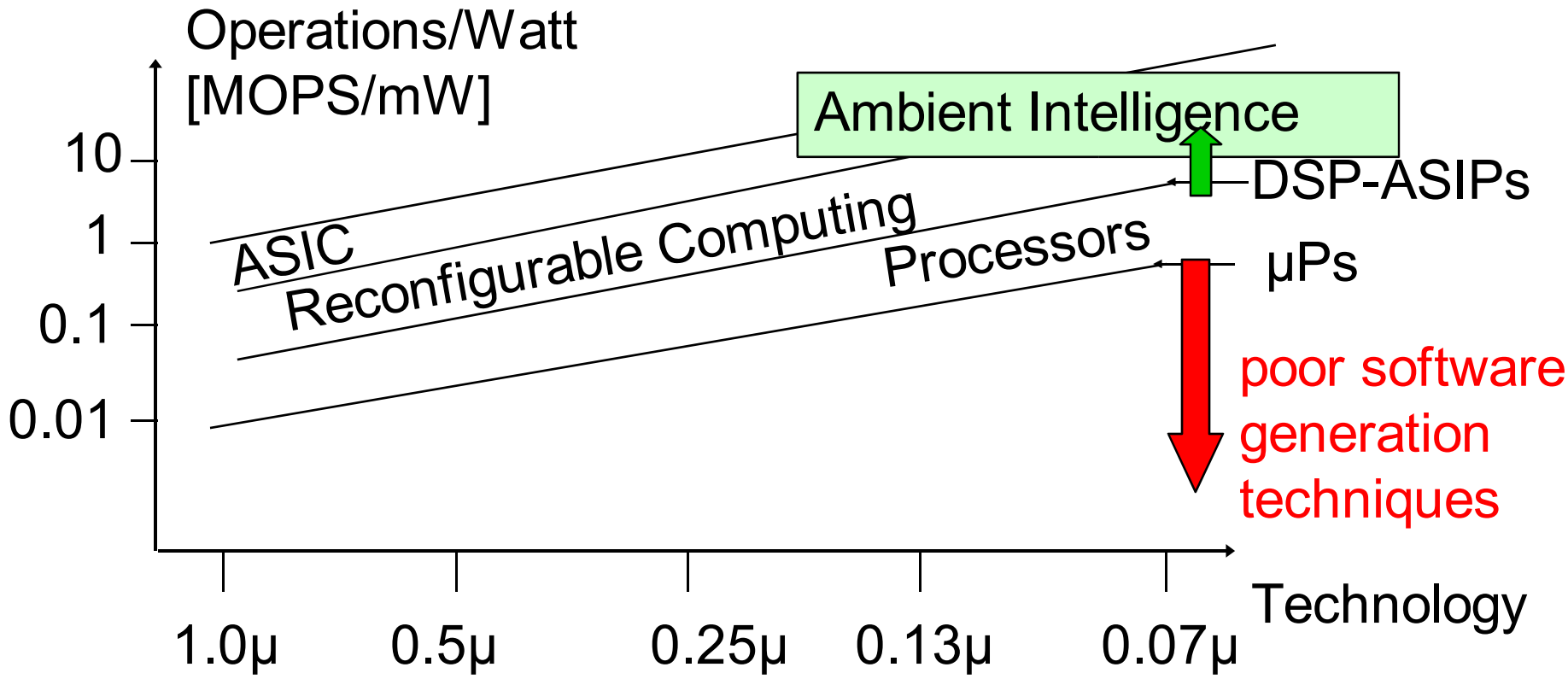
Efficiency improving transformations

Peter Marwedel
University of Dortmund + ICD
Dortmund, Germany

Software efficiency extremely important



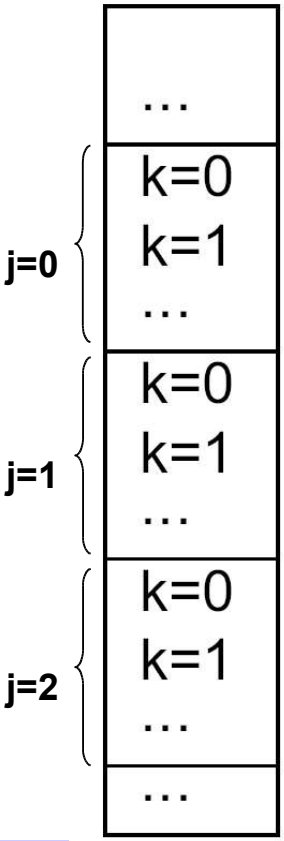
Necessary to optimize software; otherwise the price for software flexibility cannot be paid!

[H. de Man, Keynote, DATE'02; T. Claasen, ISSCC99]

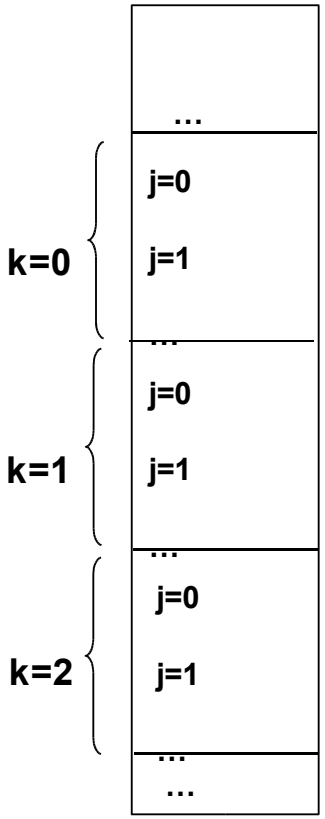
Impact of memory allocation on efficiency

Array p[j][k]

Row major order (C)



Column major order (FORTRAN)



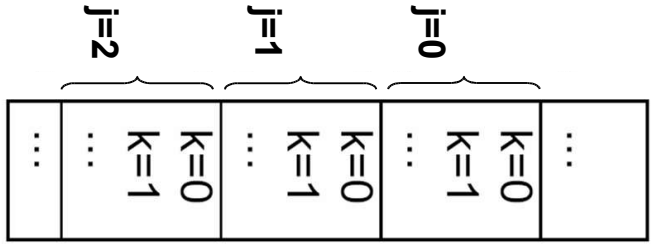
Best performance if innermost loop corresponds to rightmost array index

Two loops, assuming row major order (C):

```
for (k=0; k<=m; k++)          |  for (j=0; j<=n; j++)
  for (j=0; j<=n; j++) )      |    for (k=0; k<=m; k++)
    p[j][k] = ...             |     p[j][k] = ...
```

Same behavior for homogenous memory access, but:

For row major order



↑ Poor cache behavior Good cache behavior ↑

☞ memory architecture dependent optimization

☞ Program transformation “Loop interchange”

Example:

```
…#define iter 400000
int a[20][20][20];
void computeijk() {int i,j,k;
  for (i = 0; i < 20; i++) {
      for (j = 0; j < 20; j++) {
          for (k = 0; k < 20; k++) {
              a[i][j][k] += a[i][j][k];}}}}
void computeikj() {int i,j,k;
  for (i = 0; i < 20; i++) {
      for (j = 0; j < 20; j++) {
          for (k = 0; k < 20; k++) {
              a[i][k][j] += a[i][k][j] ;}}}}…
start=time(&start);for(z=0;z<iter;z++)computeijk();
  end=time(&end);
  printf("ijk=%16.9f\n",1.0*difftime(end,start));
```

☞ Improved locality

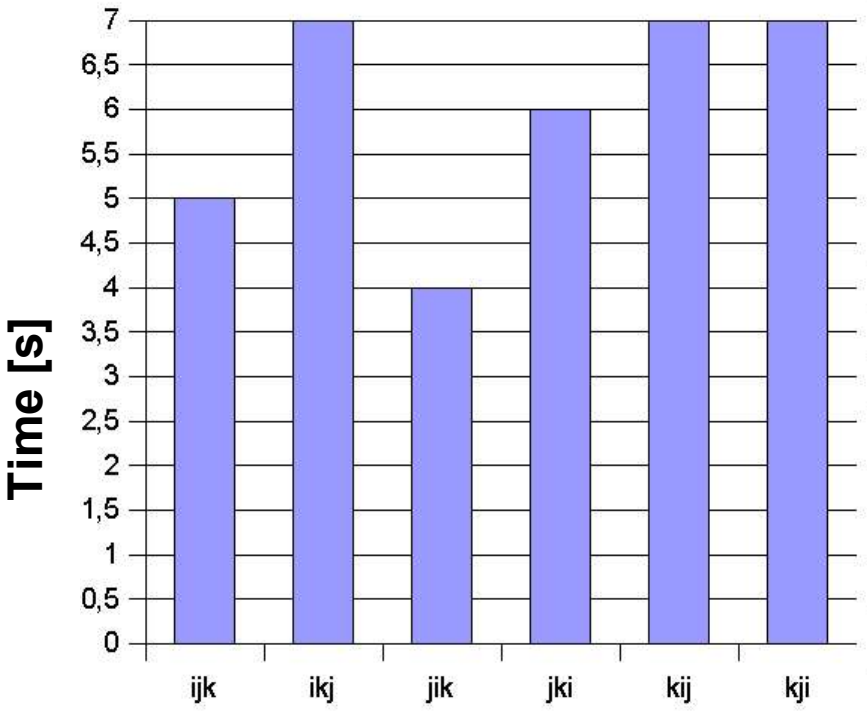
(SUIF interchanges array indexes instead of loops)

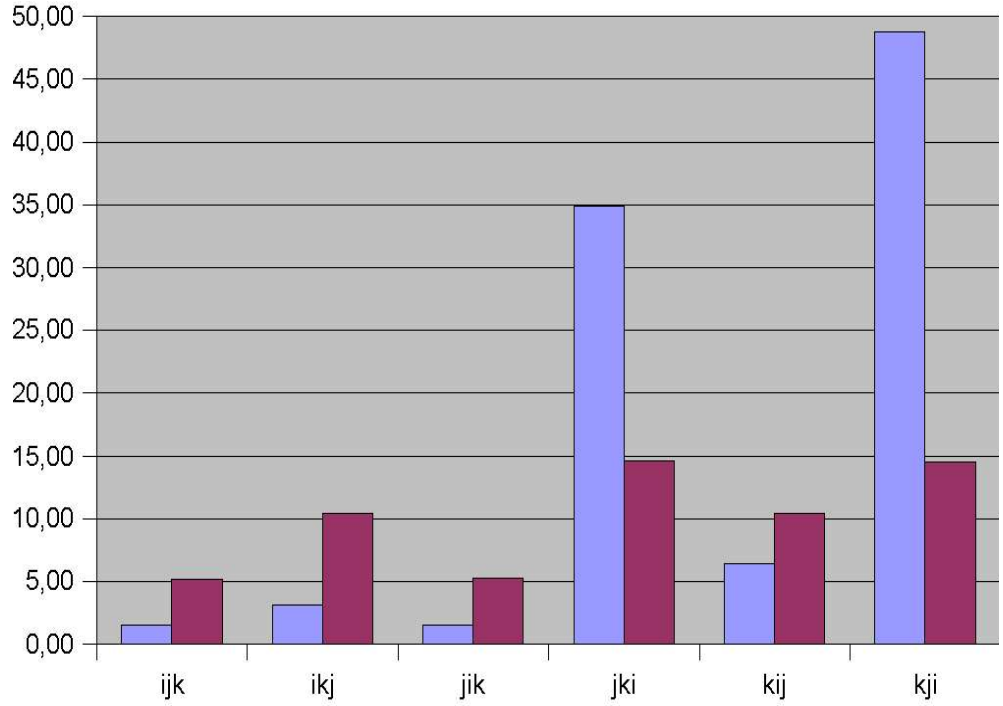
Results:
strong influence of the memory architecture

Loop structure: i j k

Dramatic impact of locality

Processor	Ti C6xx	Sun SPARC	Intel Pentium
reduction to [%]	~ 57%	35%	3.2 %



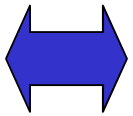


Not always the same impact ..

[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

Transformations
"Loop fusion" (merging), "loop fission"

```
for(j=0; j<=n; j++)
  p[j]= ... ;
for (j=0; j<=n; j++) ,
  p[j]= p[j] + ...
```



```
for (j=0; j<=n; j++)
  {p[j]= ... ;
        p[j]= p[j] + ...}
```

Loops small enough to allow zero overhead Loops

Better locality for access to p.
Better chances for parallel execution.

Which of the two versions is best?
Architecture-aware compiler should select best version.

Example: simple loops

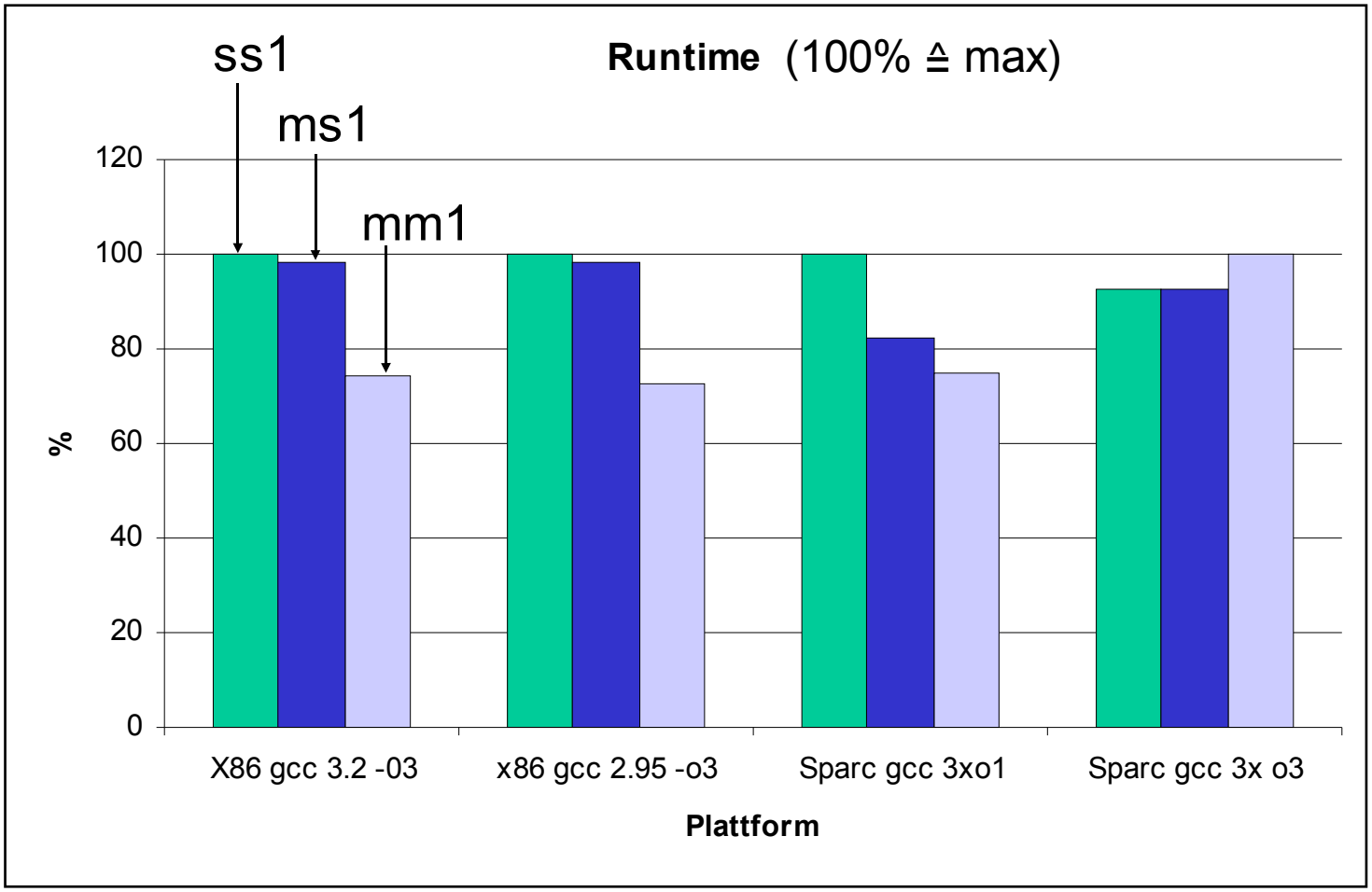
```
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];
```

```
void ss1() {int i,j;
 for (i=0;i<size;i++){
  for (j=0;j<size;j++){
   a[i][j]+= 17;}}
 for(i=0;i<size;i++){
  for (j=0;j<size;j++){
   b[i][j]-=13;}}}
```

```
void ms1() {int i,j;
 for (i=0;i< size;i++){
  for (j=0;j<size;j++){
   a[i][j]+=17;    }
  for (j=0;j<size;j++){
   b[i][j]-=13; }}}
```

```
void mm1() {int i,j;
 for(i=0;i<size;i++){
  for(j=0;j<size;j++){
   a[i][j] += 17;
   b[i][j] -= 13;}}}
```

Results: simple loops



☞ Merged loops superior; except Sparc with –o3

Example: loops with reuse

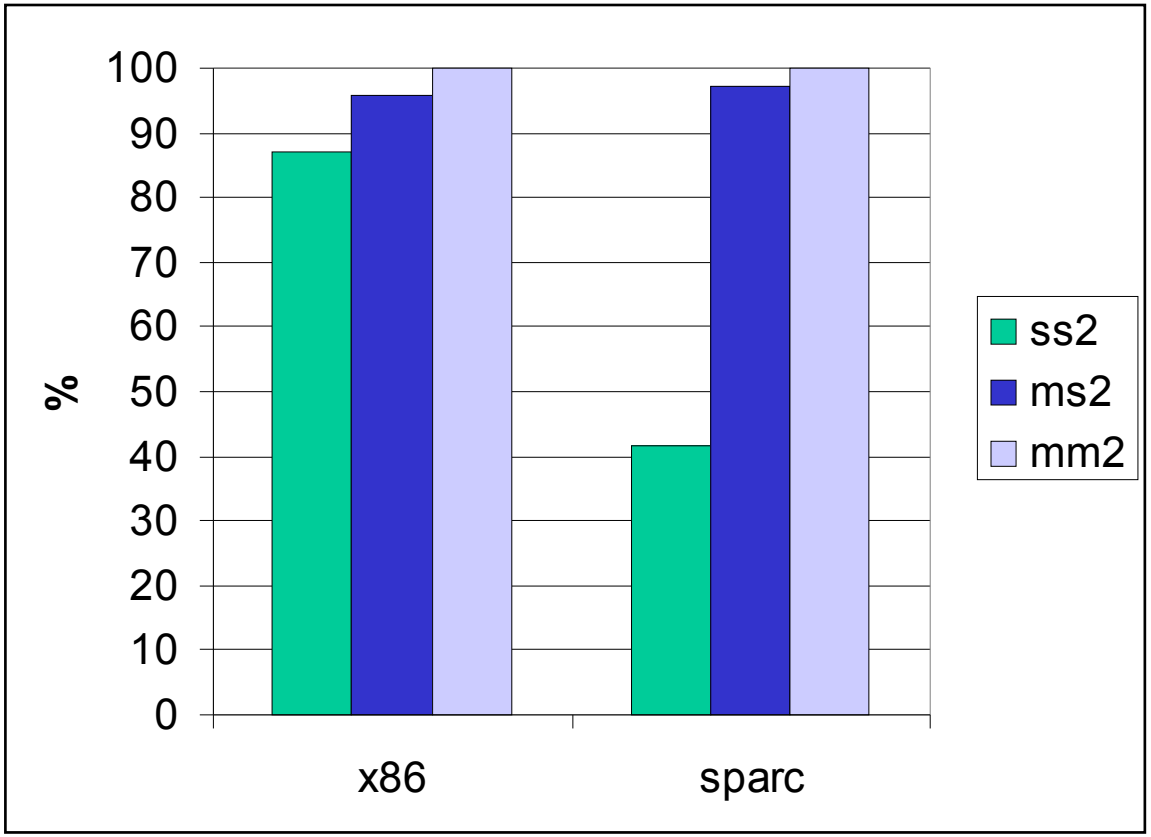
```
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];
```

```
void ss2() {int i,j;
 for(i=0;i<size;i++) {
  for(j=0;j<size;j++) {
   a[i][j] += 17;}}
 for(i=0;i<size;i++){
  for(j=0;j<size;j++) {
   b[i][j]-=a[i][j];}}}
```

```
void ms2() {int i,j;
 for(i=0;i<size;i++) {
  for(j=0;j<size;j++) {
   a[i][j]+= 17;}
  for(j=0;j<size;j++){
   b[i][j]-=a[i][j];}}}
```

```
void mm2() {int i,j;
 for(i=0;i<size;i++) {
  for(j=0;j<size;j++) {
   a[i][j]+=17;
   b[i][j]-=a[i][j];}}}
```

Results



gcc 3.x –o3

Split loops superior (reason: high level of optimization?)

Loop unrolling

```
for (j=0; j<=n; j++)
p[j]= ... ;
```

```
for (j=0; j<=n; j+=2)
    {p[j]= ... ; p[j+1]= ...}
```

factor = 2

Better locality for access to p.

Less branches per execution of the loop. More opportunities for optimizations.

Tradeoff between code size and improvement.

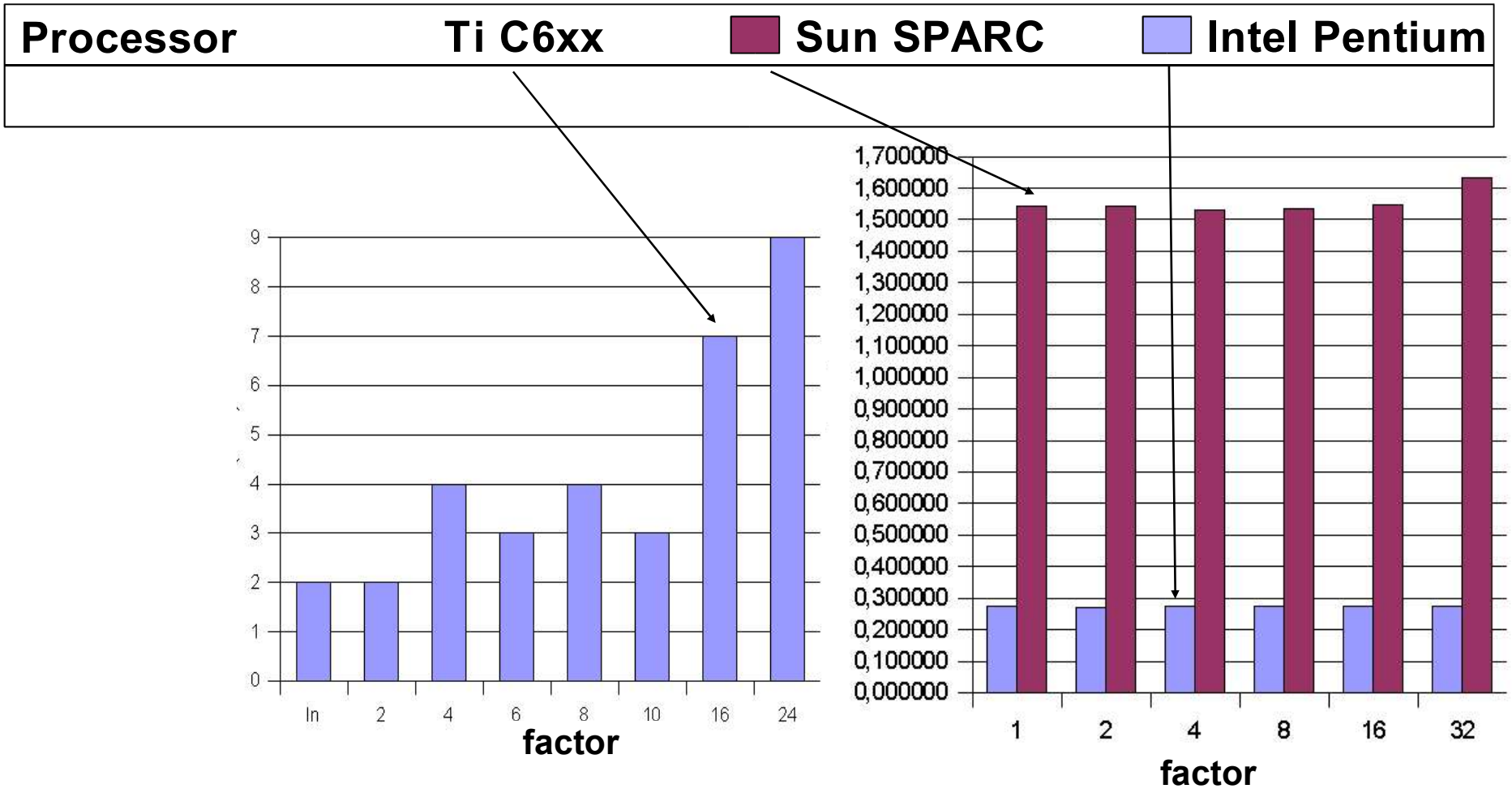
Extreme case: completely unrolled loop (no branch).

Example: matrixmult

```
#define s 30
#define iter 4000
int a[s][s],b[s][s],
   c[s][s];
void compute(){int
   i,j,k;
 for(i=0;i<s;i++){
  for(j=0;j<s;j++){
   for(k=0;k<s;k++){
    c[i][k]+=
       a[i][j]*b[j]
   [k];
}}}}
```

```
extern void compute2()
 {int i, j, k;
  for (i = 0; i < 30; i++) {
   for (j = 0; j < 30; j++)  {
    for (k = 0; k <= 28; k += 2)
     {{int *suif_tmp;
     suif_tmp = &c[i][k];
     *suif_tmp=
     *suif_tmp+a[i][j]*b[j][k];}
     {int *suif_tmp;
     suif_tmp=&c[i][k+1];
     *suif_tmp=*suif_tmp
            +a[i][j]*b[j][k+1];
 }}}}
return;}
```

Results



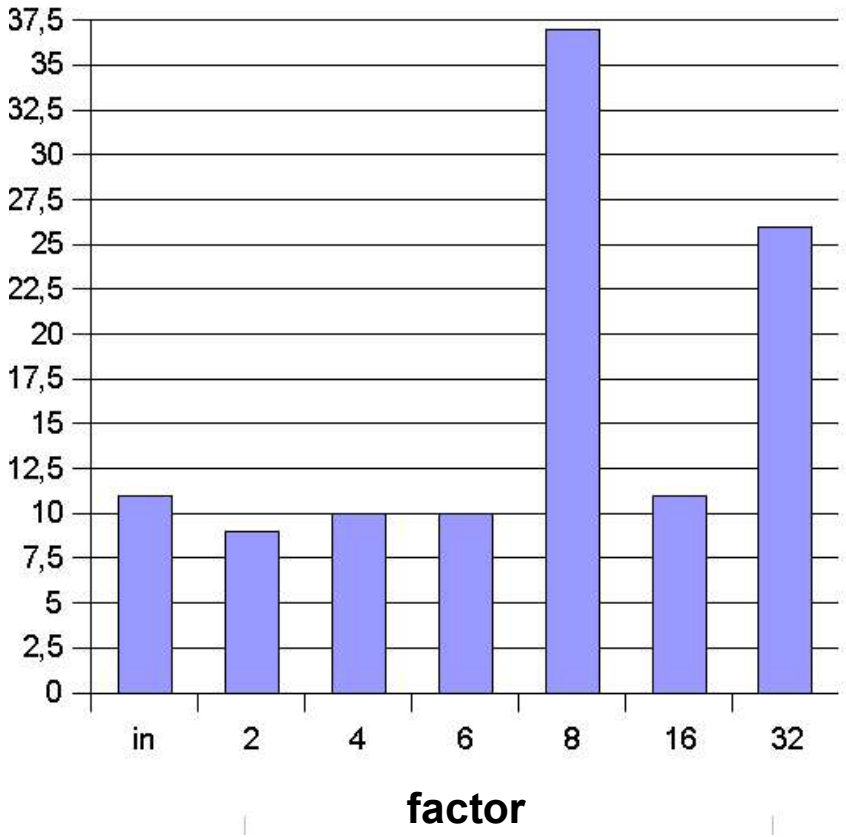
Benefits quite small; penalties may be large

[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

Results: benefits for loop dependences

Processor	Ti C6xx
reduction to [%]	

```
#define s 50
#define iter 150000
int a[s][s], b[s][s];
void compute() {
 int i,k;
  for (i = 0; i < s; i++) {
   for (k = 1; k < s; k++) {
    a[i][k] = b[i][k];
    b[i][k] = a[i][k-1];
}}}
```

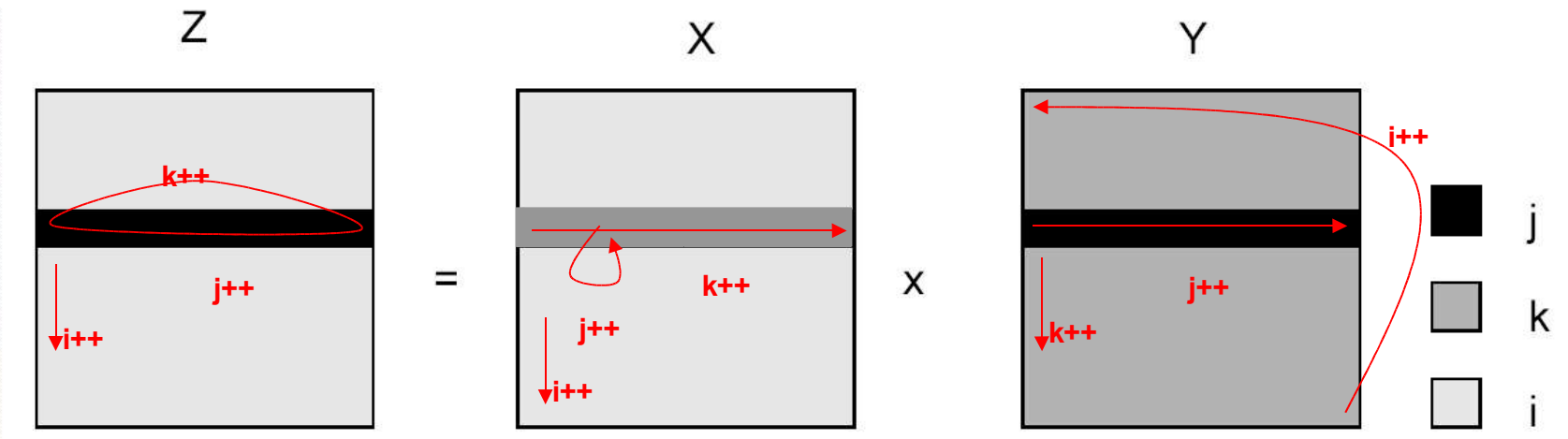


Small benefits, but results may vary

[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

Program transformation
Loop tiling/loop blocking: - Original version -

```
for (i=1; i<=N; i++)
    for(k=1; k<=N; k++){
        r=X[i,k]; /* to be allocated to a register*/
        for (j=1; j<=N; j++)
            Z[i,j] += r* Y[k,j]
} % Never reusing information in the cache for Y and Z if N
    is large or cache is small (2 N³ references for Z).
```



Loop tiling/loop blocking - tiled version -

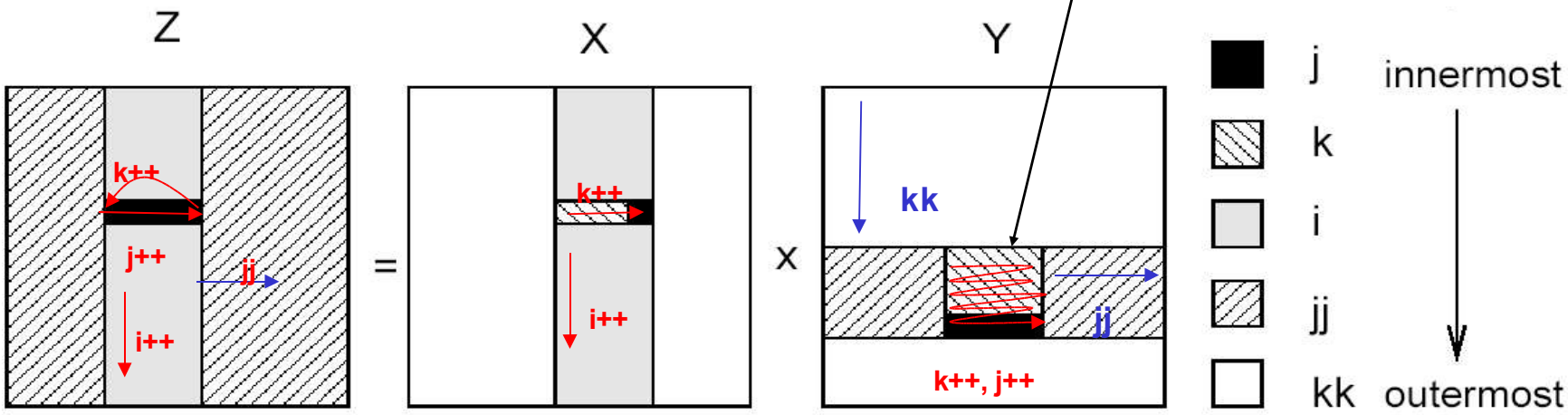
```
for (kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
    for (i=1; i<= N; i++)
      for (k=kk; k<= min(kk+B-1,N); k++){
        r=X[i][k]; /* to be allocated to a register*/
        for (j=jj; j<= min(jj+B-1, N); j++)
          Z[i][j] += r* Y[k][j]
        }
```

Reuse factor of B for Z, N for Y

$O(N^3/B)$ accesses to main memory

Compiler should select best option

Same elements for next iteration of i

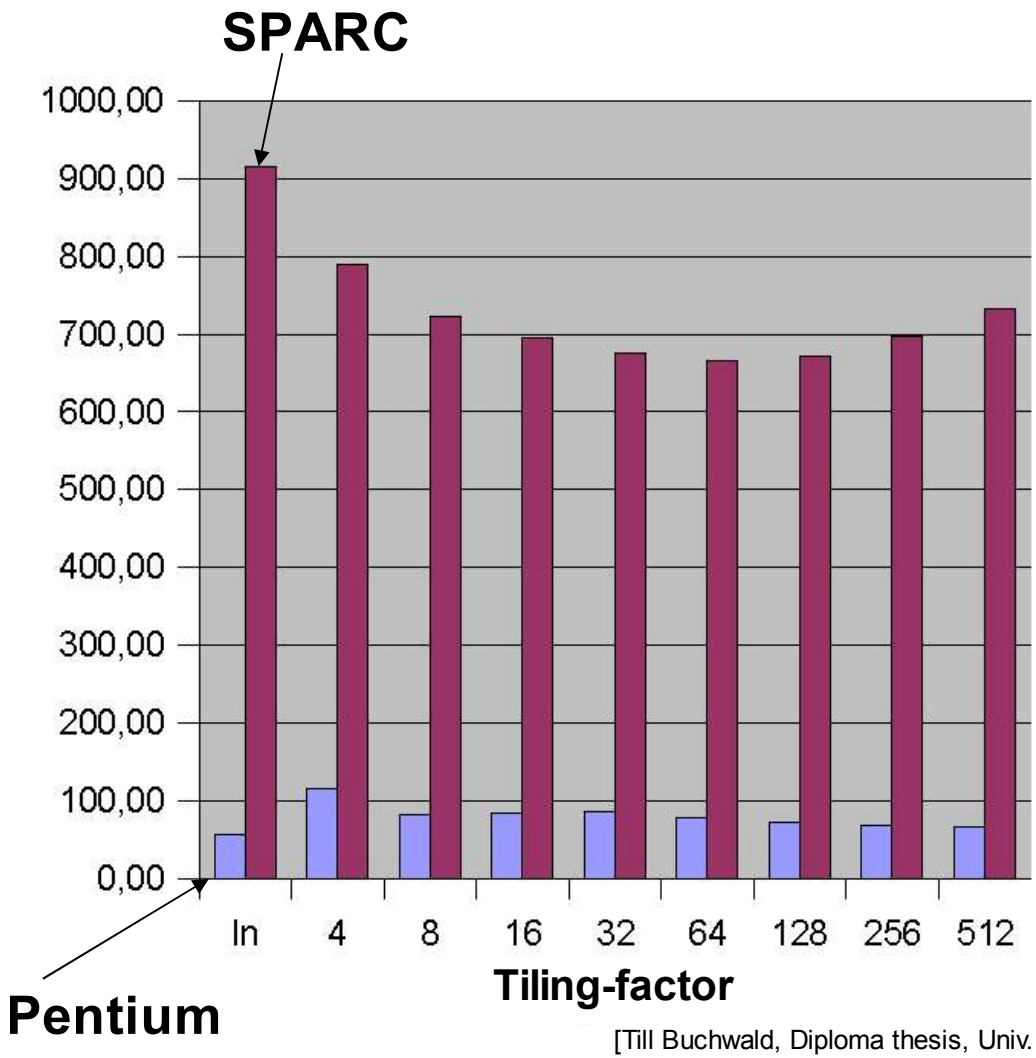


Example

In practice, results by Buchwald are disappointing.

One of the few cases where an improvement was achieved:

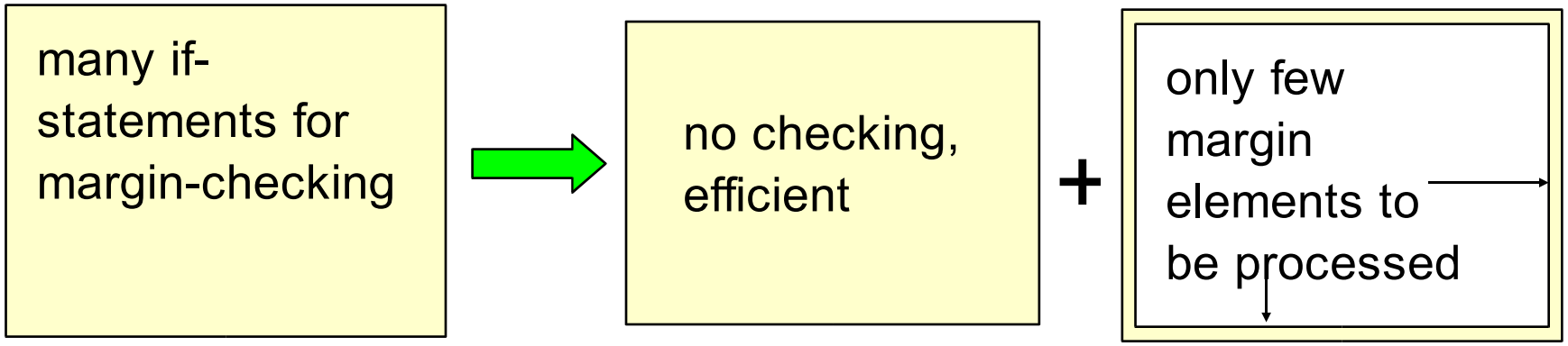
Source: similar to matrix mult.



[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

Transformation “Loop nest splitting”

Example: Separation of margin handling



Loop nest splitting at University of Dortmund

Loop nest from MPEG-4 full search motion estimation

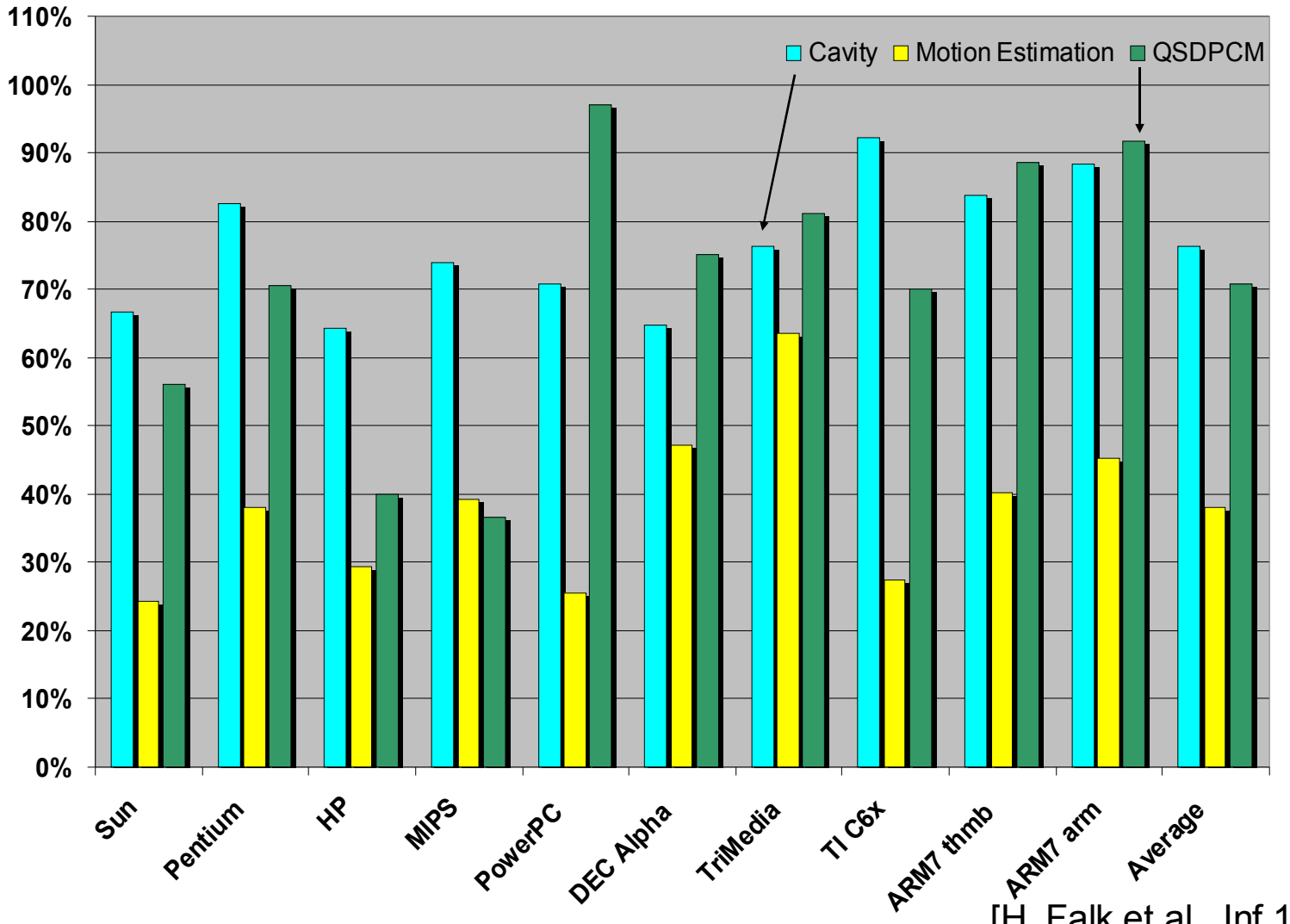
```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3||y3<0||48<y3)
        then_block_1; else else_block_1;
       if (x4<0|| 35<x4||y4<0||48<y4)
        then_block_2; else else_block_2;
}}}}}}
```

analysis of polyhedral domains,
selection with genetic algorithm

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++)
if (x>=10||y>=14)
  for (; y<49; y++)
    for (k=0; k<9; k++)
     for (l=0; l<9;l++ )
      for (i=0; i<4; i++)
       for (j=0; j<4;j++) {
        then_block_1; then_block_2}
else {y1=4*y;
  for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (0 || 35<x3 ||0 || 48<y3)
        then-block-1; else else-block-1;
       if (x4<0|| 35<x4||y4<0||48<y4)
        then_block_2; else else_block_2;
}}}}}}
```

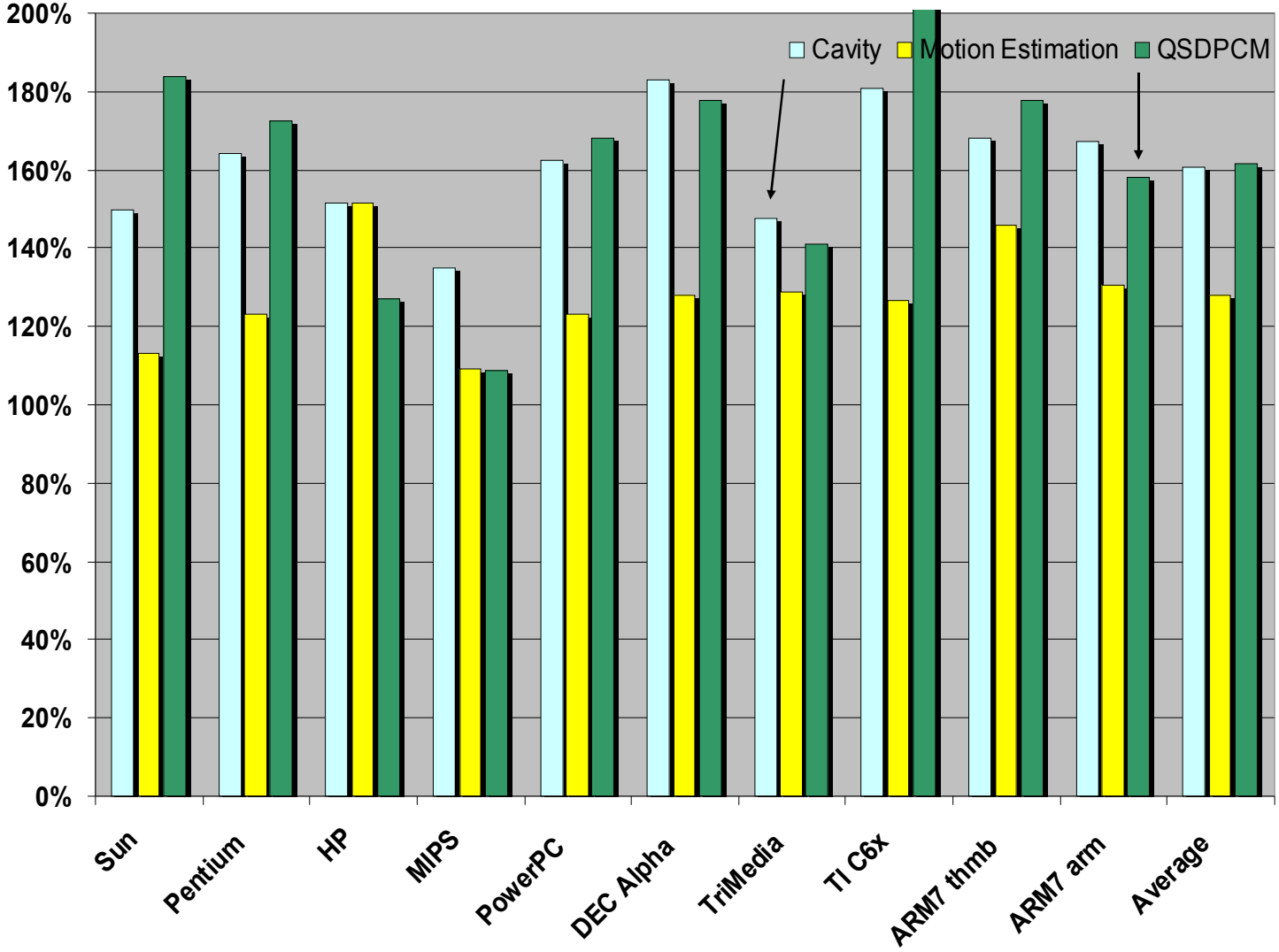
[H. Falk et al., Inf 12, UniDo, 2002]

Results for loop nest splitting
- Execution times -



[H. Falk et al., Inf 12, UniDo, 2002]

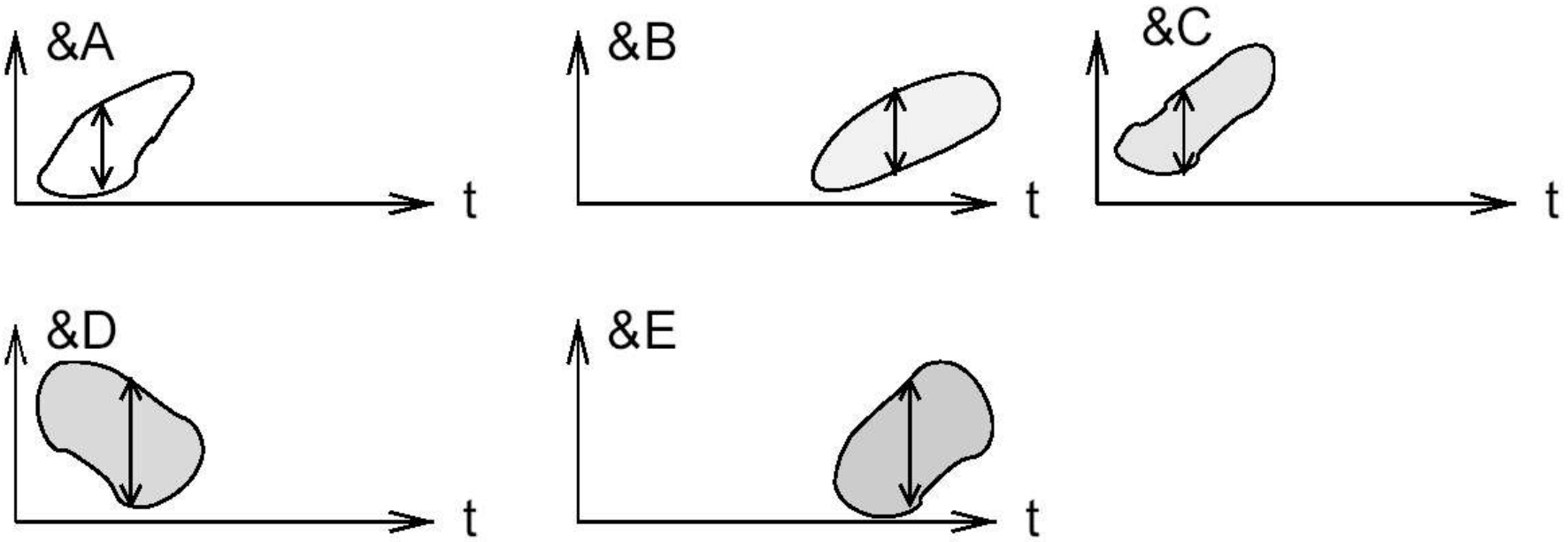
Results for loop nest splitting
- Code sizes -



[Falk, 2002]

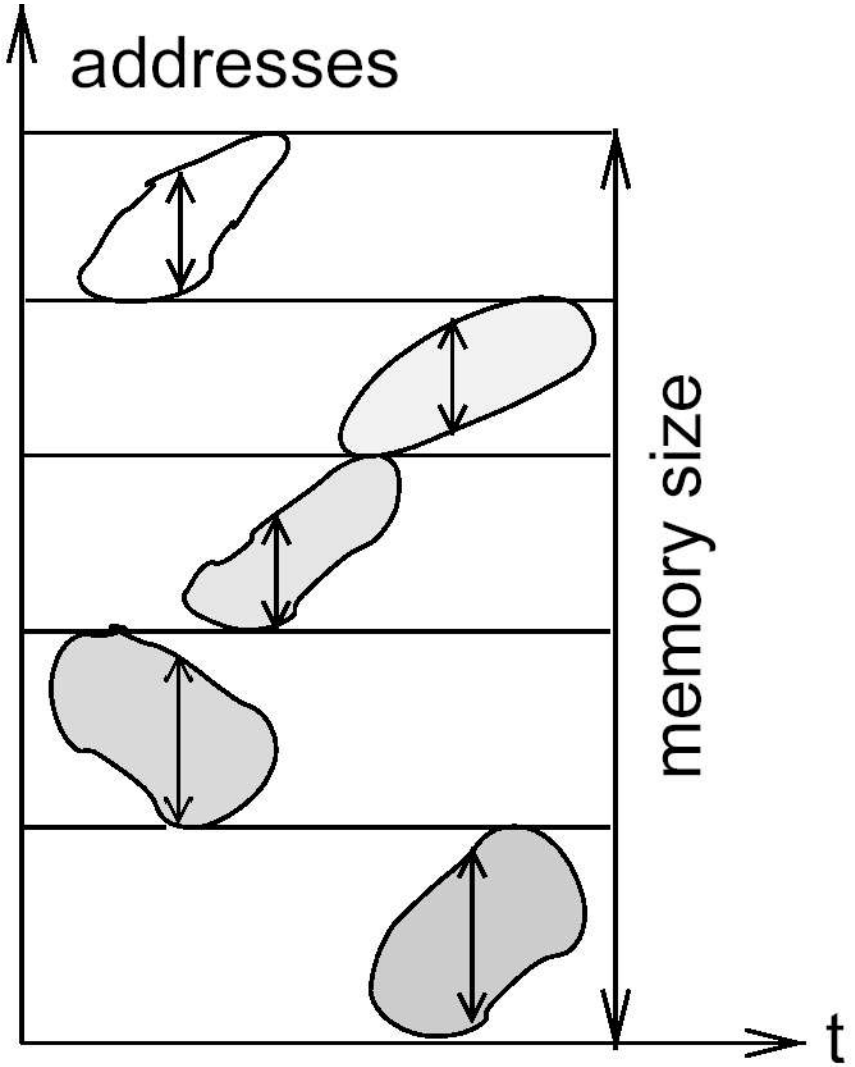
Array folding

Initial arrays



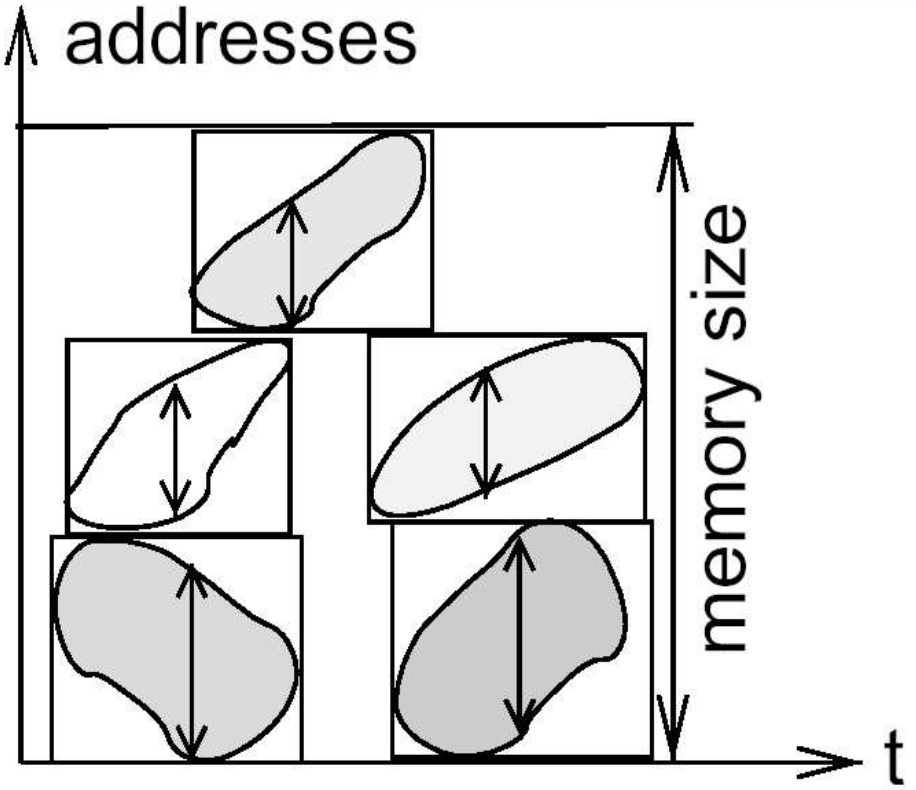
Array folding

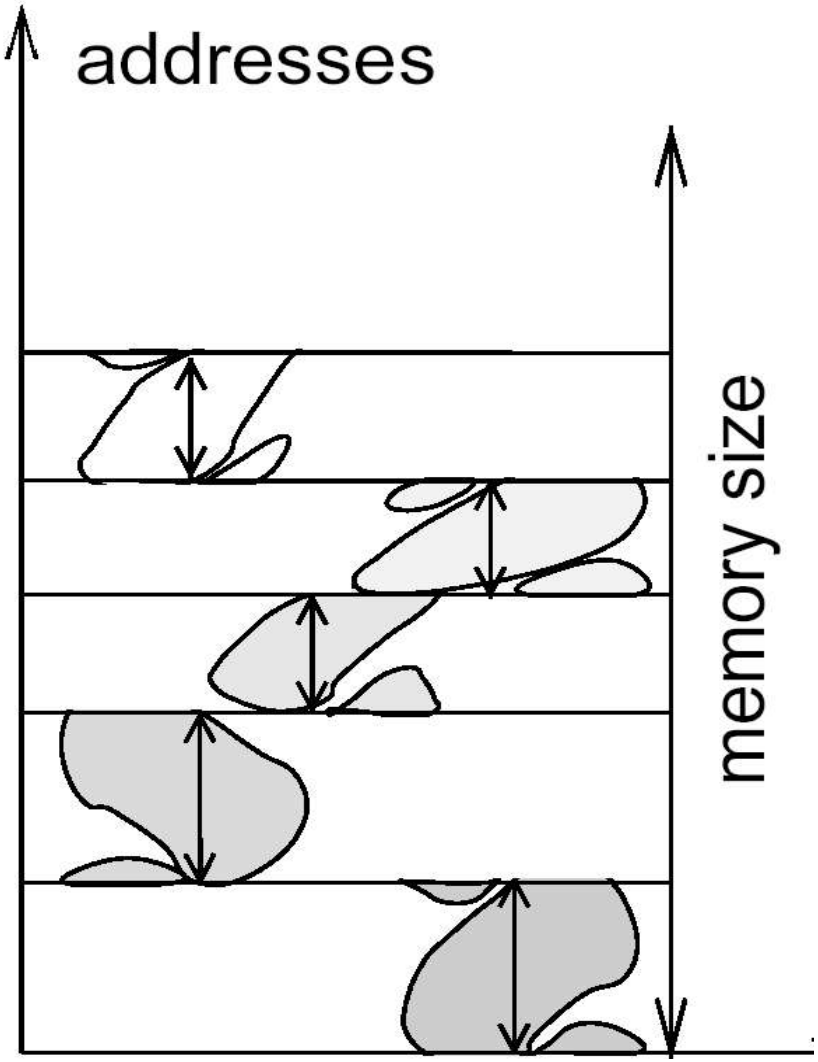
Unfolded arrays



Intra-array folding

Inter-array folding





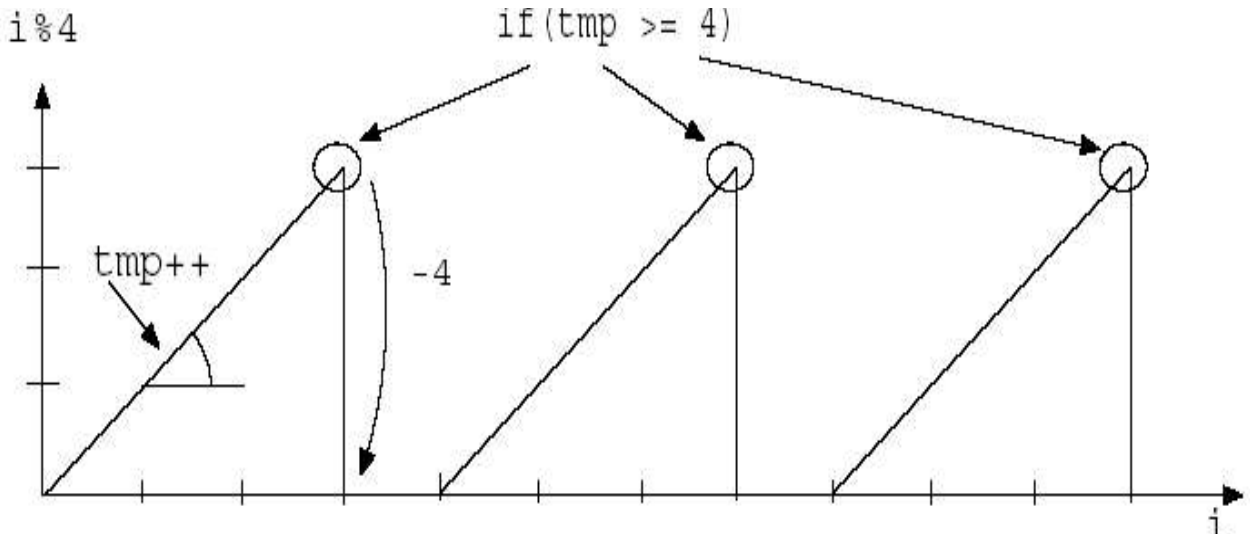
Application

Array folding is implemented in the DTSE optimization proposed by IMEC. Array folding adds div and mod ops. Optimizations required to remove these costly operations.

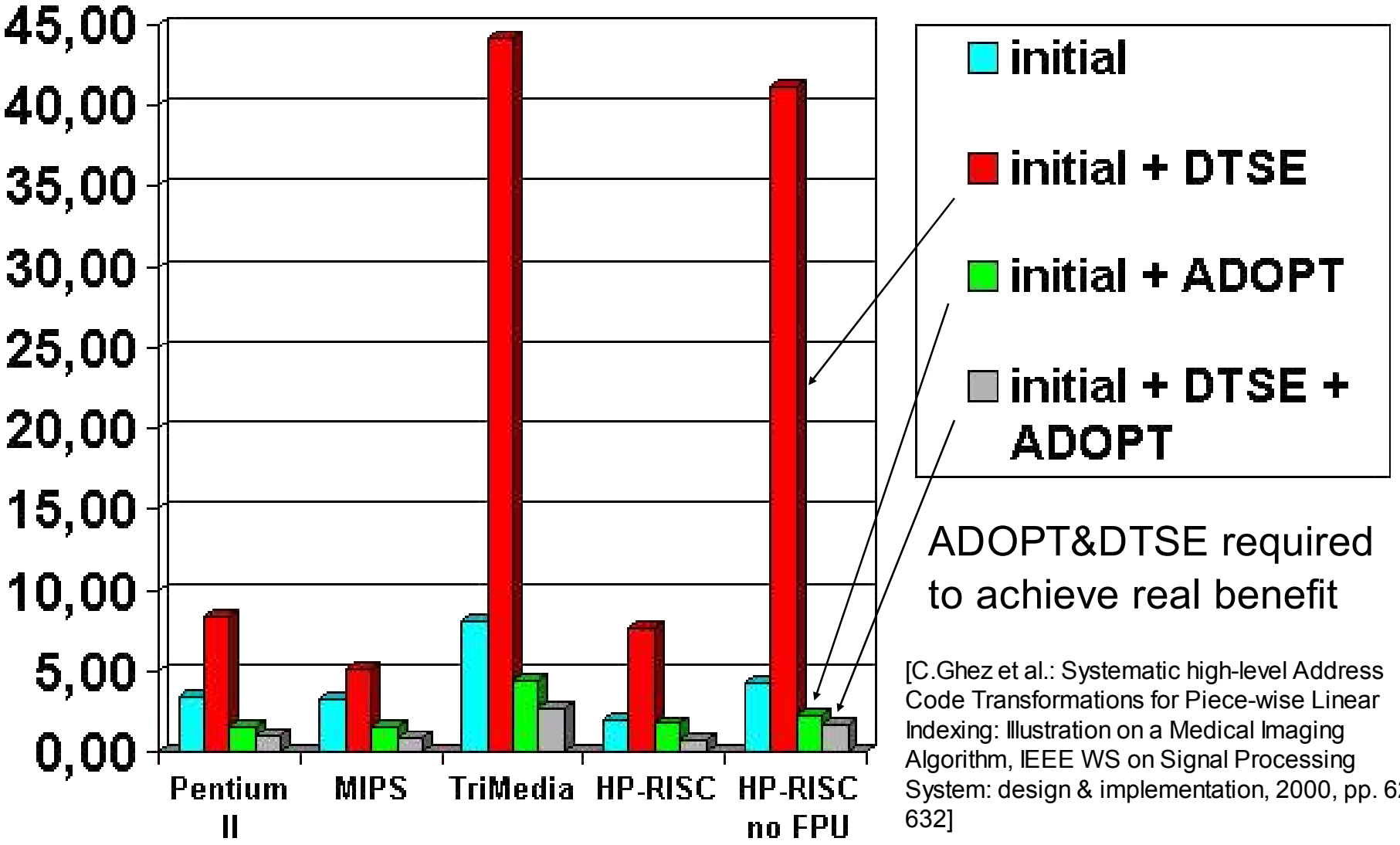
At IMEC, ADOPT address optimizations perform this task. For example, modulo operations are replaced by pointers (indexes) which are incremented and reset.

```
for(i=0; i<20; i++)
      B[i % 4];

tmp=0;
for(i=0; i<20; i++)
      if(tmp >= 4)
         tmp -=4;
      B[tmp];
      tmp ++;
```



Results (Mcycles for cavity benchmark)



ADOPT&DTSE required to achieve real benefit

[C.Ghez et al.: Systematic high-level Address Code Transformations for Piece-wise Linear Indexing: Illustration on a Medical Imaging Algorithm, IEEE WS on Signal Processing System: design & implementation, 2000, pp. 623-632]

Function inlining: advantages and limitations

Advantage: low calling overhead

```
Function sq
(c:integer)
 return:integer;
begin
 return c*c
end;
....
a=sq(b);
....
```

branching

Inlining

```
push PC;
push b;
BRA sq;
 pull R1;
 mul R1,R1,R1;
 pull R2;
 push R1;
 BRA (R2)+1;
pull R1;
ST R1,a;
....
LD R1,b;
MUL R1,R1,R1;
ST R1,a
....
```

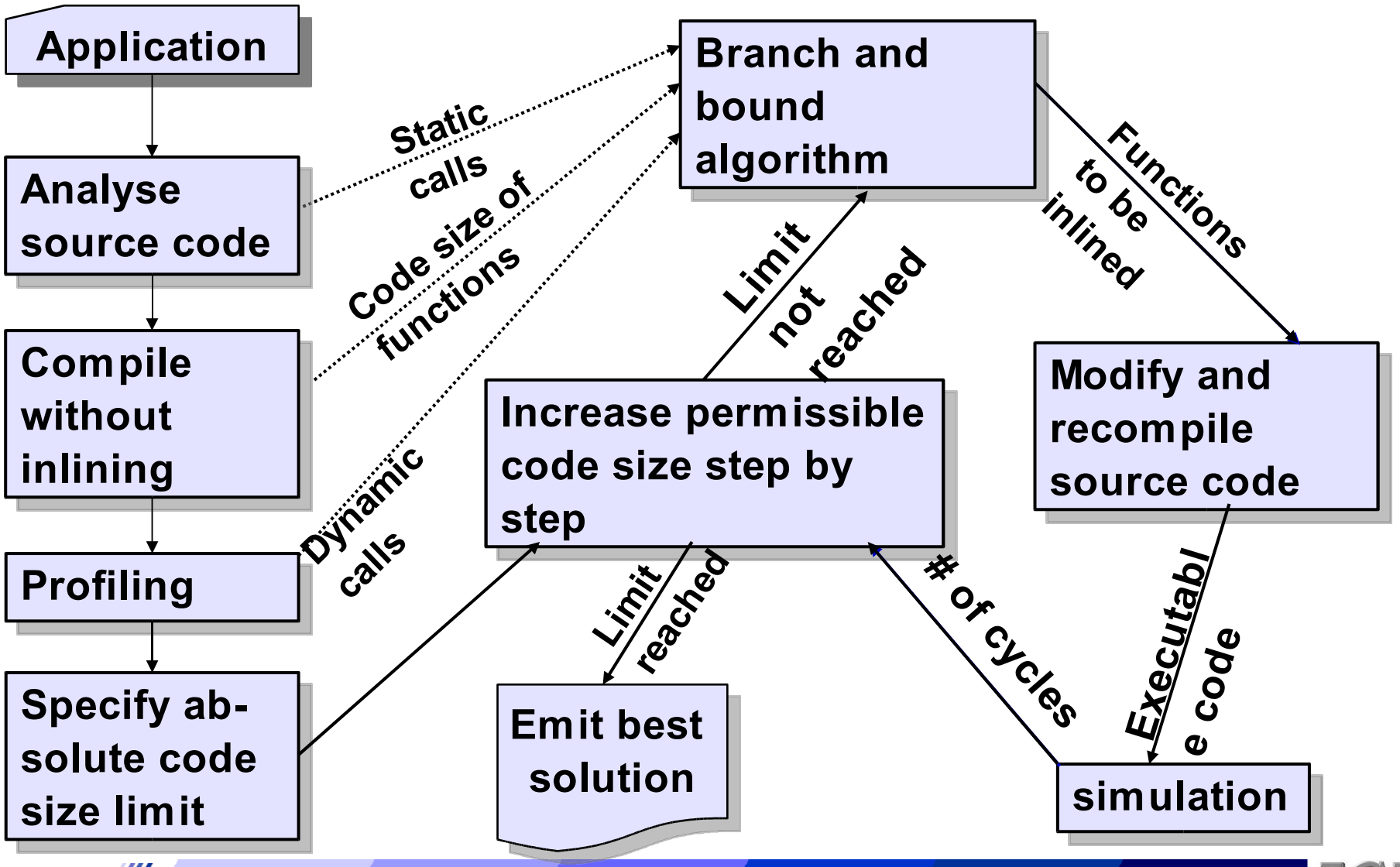
Limitations:

- Not all functions are candidates.
- Code size explosion.
- Requires manual identification using 'inline' qualifier.

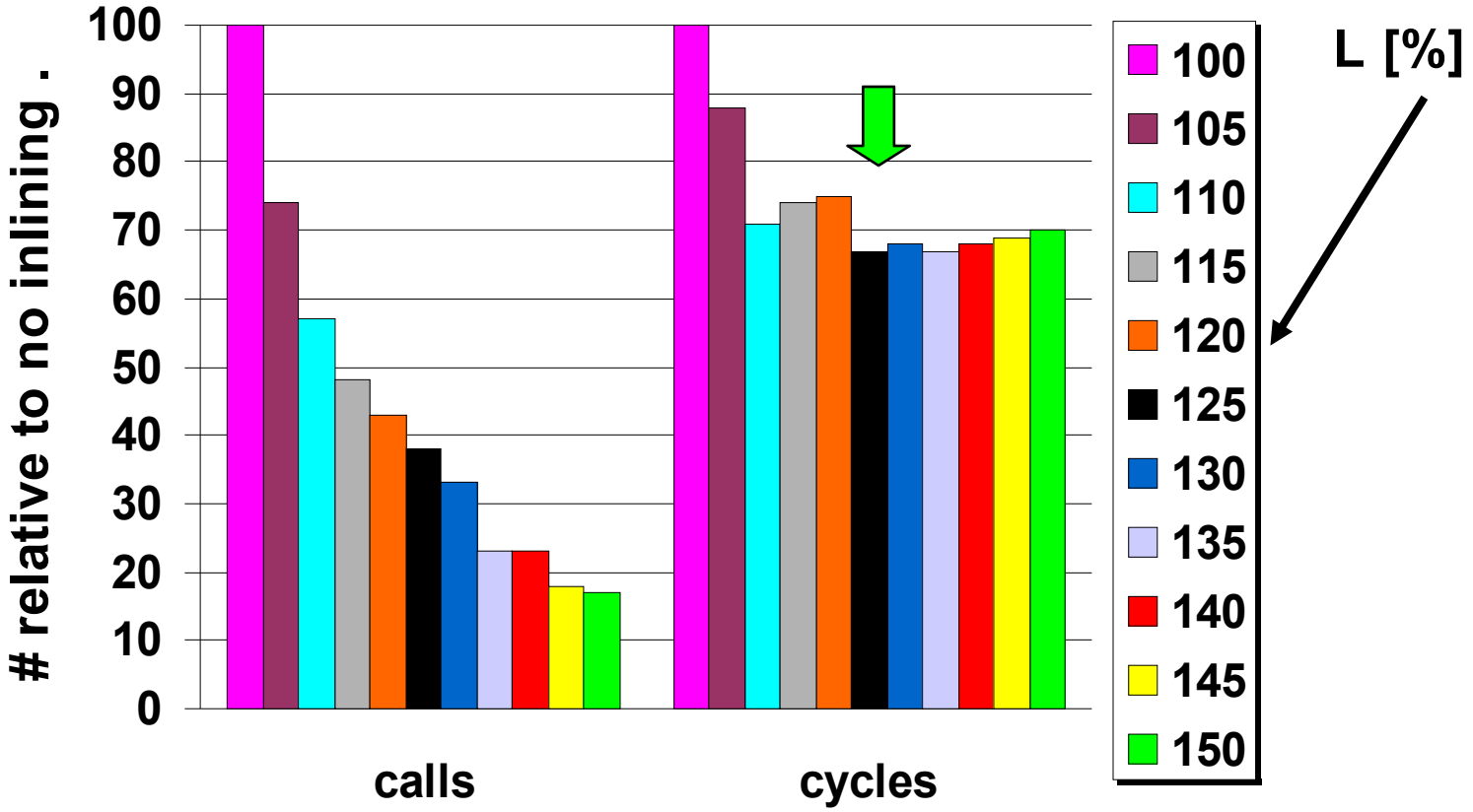
Goal:

- Controlled code size
- Automatic identification of suitable functions.

Design flow


Application
Analyse source code
Compile without inlining
Profiling
Specify ab-solute code size limit
Static calls
Code size of functions
Dynamic calls
Branch and bound algorithm
Limit not reached
Increase permissible code size step by step
Limit reached
Emit best solution
Functions to be inlined
Modify and recompile source code
# of cycles
Executable code
simulation

Results for GSM speech and channel encoder: #calls, #cycles (TI 'C62xx)



33% speedup for 25% increase in code size.

# of cycles not a monotonically decreasing function of the code size!

Inline vectors computed by B&B algorithm

size limit (%)	inline vector (functions 1-26)
100	00000000000000000000000000
105	00100000001100001110111111
110	10111001011100001111111111
115	10110000000001001000111001
120	10110100101000100110111101
125	10110000001010000100111101
130	00110000000010100100111000
135	10110010001110101110111101
140	10111011111110101111111111
145	10110110101010100110111101
150	10110110000010110110111101

Major changes for each new size limit. Difficult to generate manually.

References:

• J. Teich, E. Zitzler, S.S. Bhattacharyya. 3D Exploration of Software Schedules for DSP Algorithms, CODES'99

• R. Leupers, P.Marwedel: Function Inlining under Code Size Constraints for Embedded Processors ICCAD, 1999

Floating-point to fixed point conversion

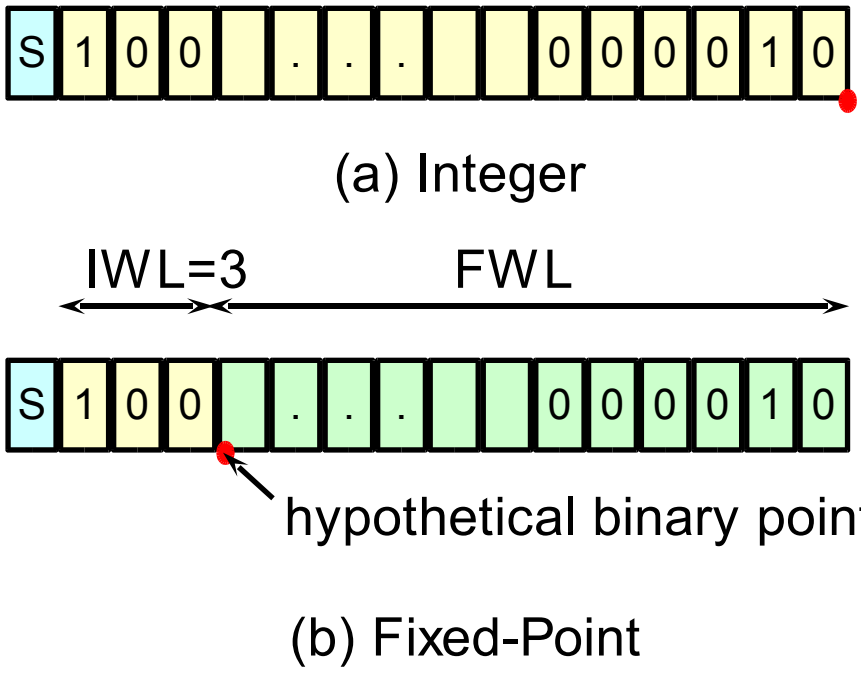
- **Pros**:
  - Lower cost
  - Faster
  - Lower power consumption
  - Sufficient SQNR, *if properly scaled*
  - Suitable for portable applications
- **Cons:**
  - Decreased dynamic range
  - Finite word-length effect, *unless properly scaled*
    - Overflow and excessive quantization noise
  - Extra programming effort

© *Ki-Il Kum, et al. (*Seoul National University*):* A Floating-point To Fixed-point C Converter For Fixed-point Digital Signal Processors, 2nd SUIF Workshop, 1996

Fixed-Point Data Format

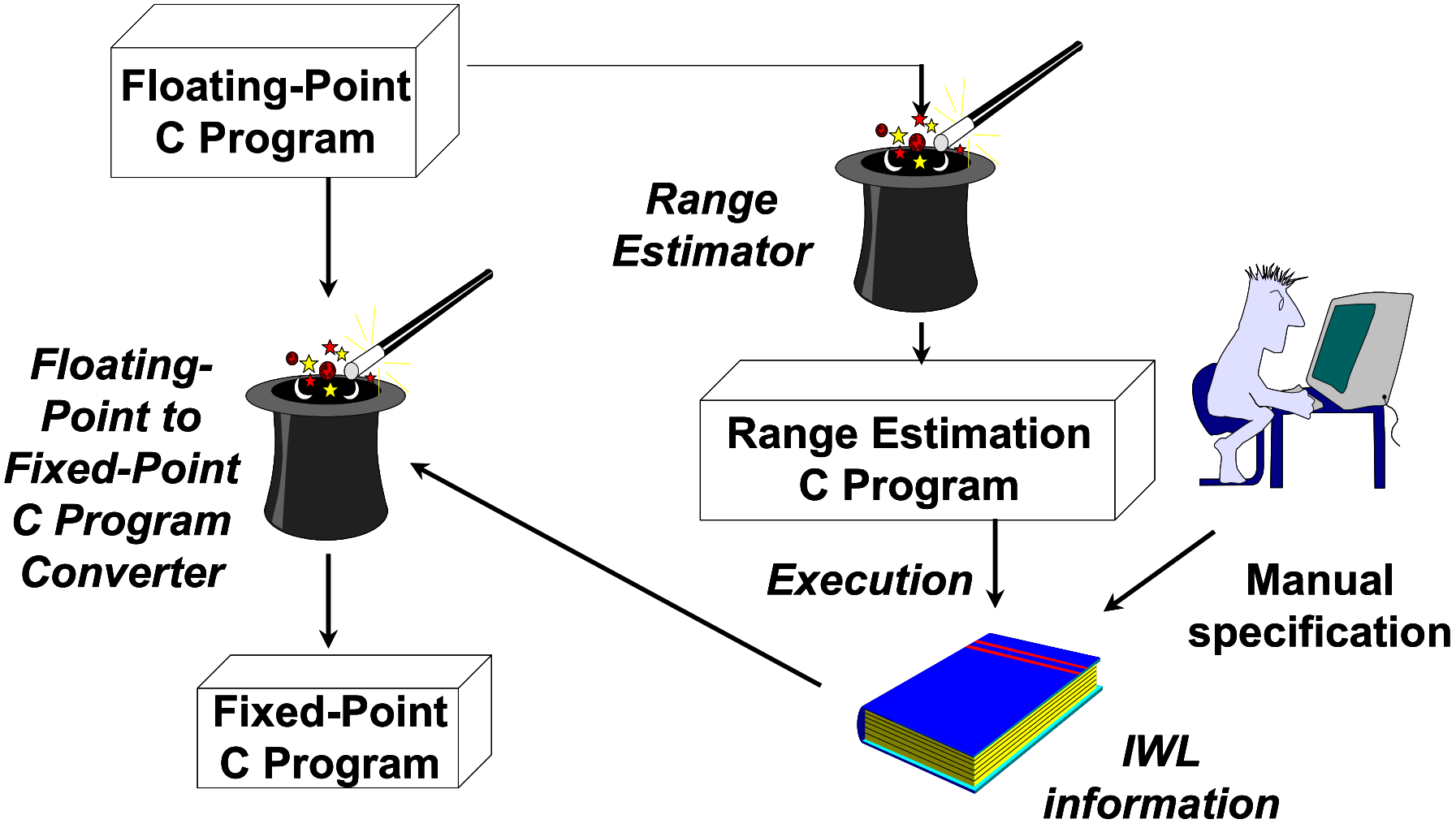
- Floating-Point vs. Fixed-Point
  - *exponent*, mantissa
  - Floating-Point
    - automatic computation and update of each exponent at run-time
  - Fixed-Point
    - implicit exponent
    - determined off-line

- Integer vs. Fixed-Point

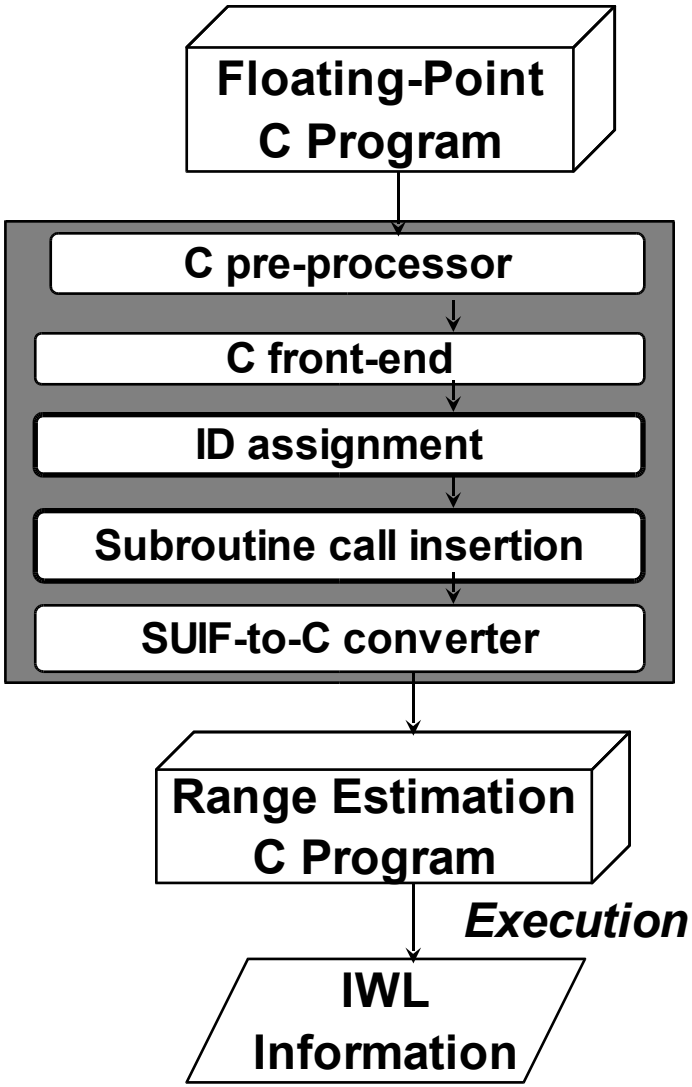


© *Ki-Il Kum, et al*

Development Procedure



Range Estimator



SUIF Compiler System

Range Estimation C Program

```
float iir1(float x)
{
    static float s = 0;
    float y;

    y = 0.9 * s + x;
    range(y, 0);
    s = y;
    range(s, 1);

    return y;
}
```

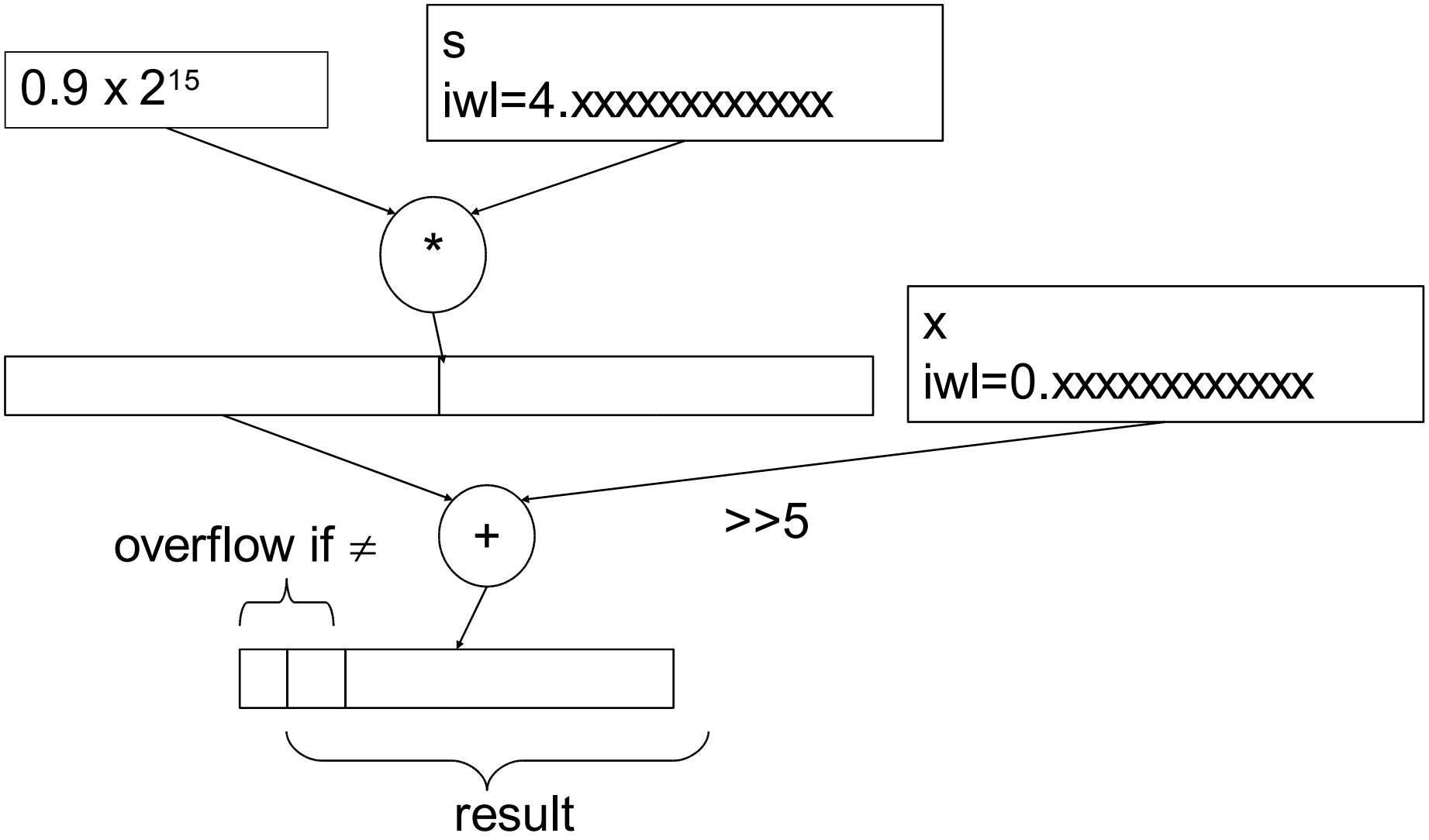
Floating-Point to Fixed-Point Program Converter

Fixed-Point C Program

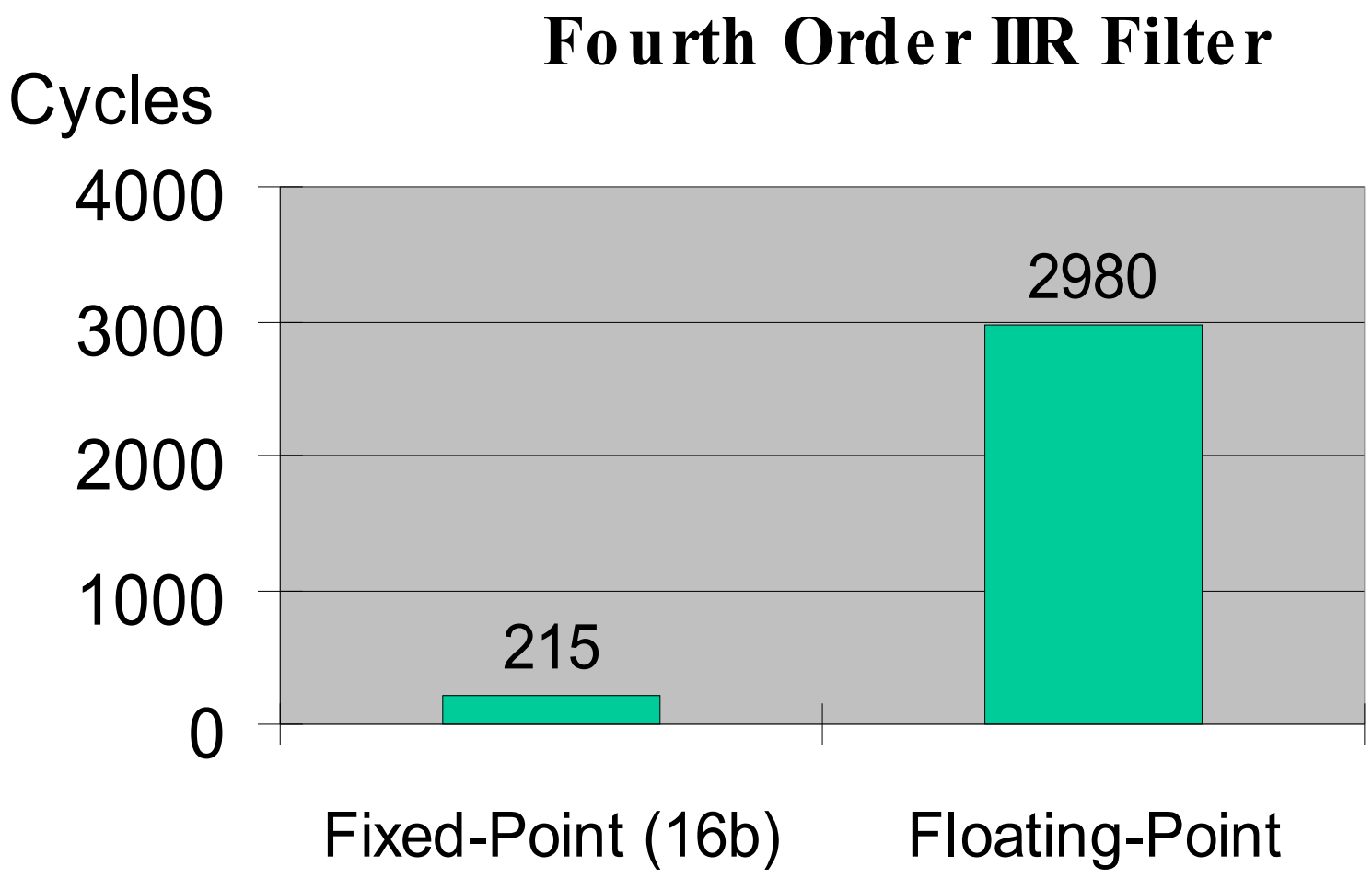
```
int iir1(int x)
{
 static int s = 0;
 int y;
 y=sll(mulh(29491,s)+ (x>> 5),1);
 s = y;
 return y;
}
```

- *mulh*
  - to access the upper half of the multiplied result
  - target dependent implementation
- *sll*
  - to remove 2nd sign bit
  - opt. overflow check

Operations in fixed point program

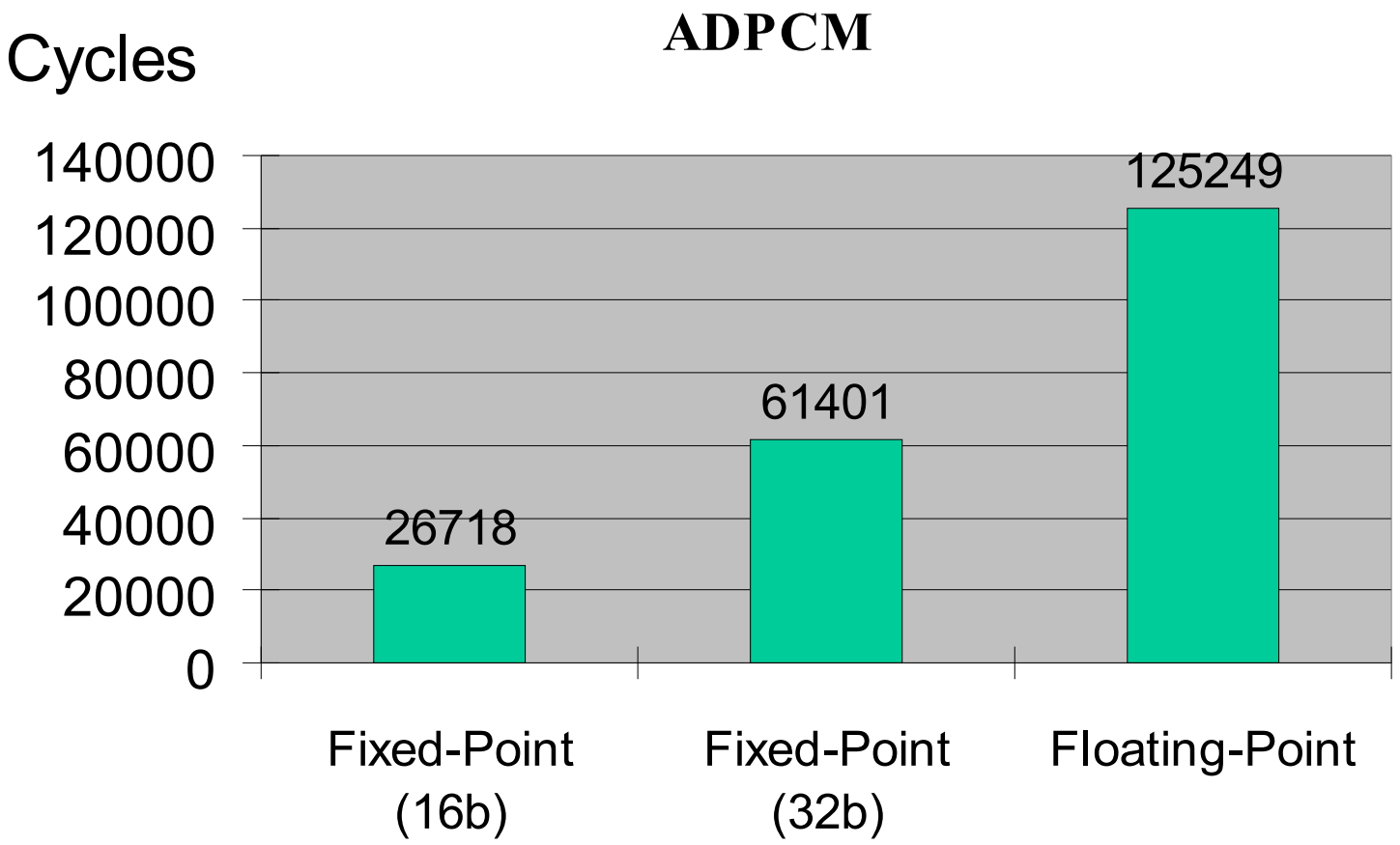


Performance Comparison
- Machine Cycles -



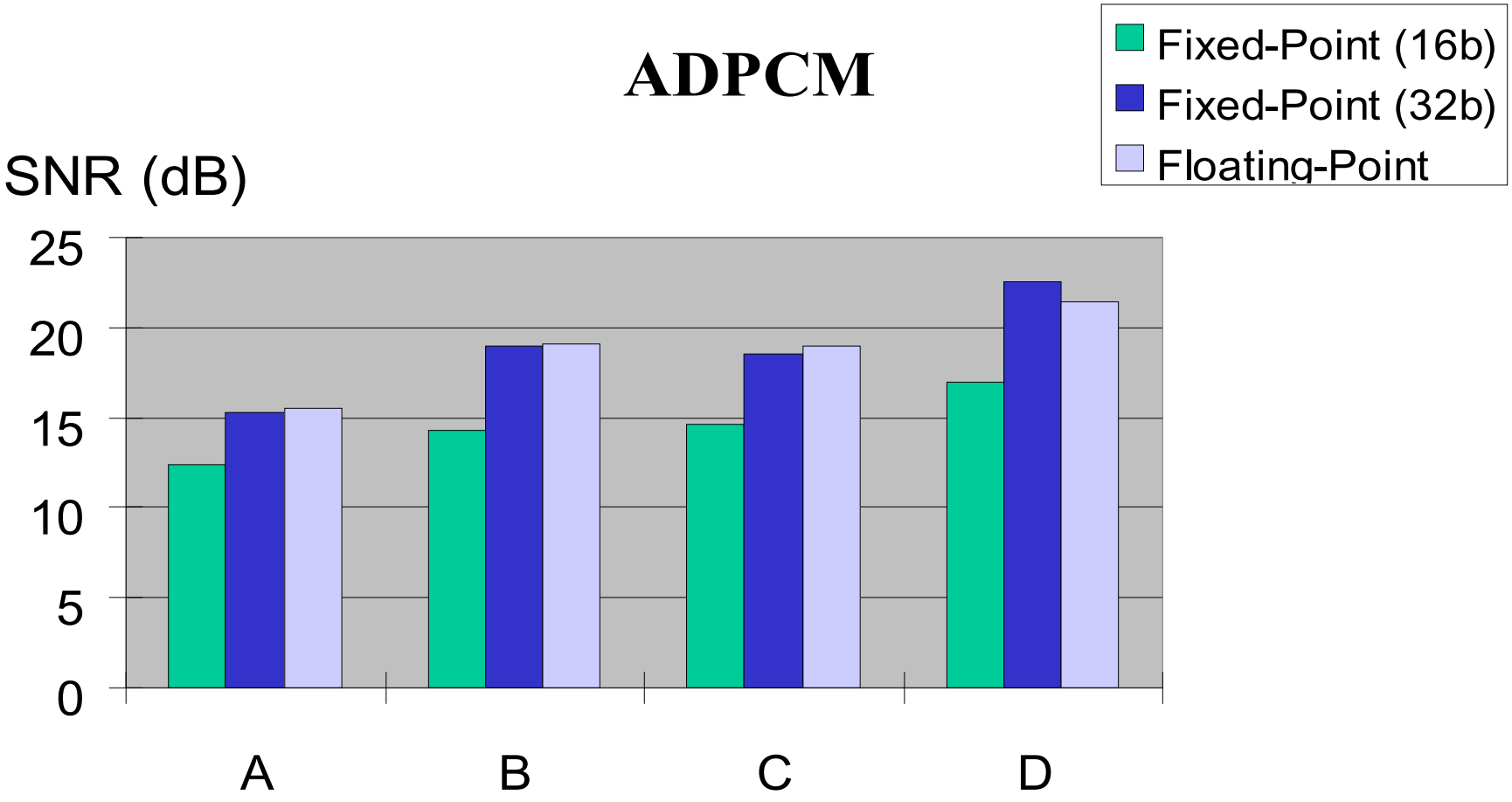
© Ki-Il Kum, et al

Performance Comparison
- Machine Cycles -



© *Ki-Il Kum, et al*

Performance Comparison
- SNR -



© *Ki-Il Kum, et al*

Summary

Efficiency improving transformations

- Loop interchange
- Loop fusion / loop fission
- Loop unrolling
- Loop tiling
- Loop nest splitting
- Inter- and intra array folding
- Controlled function inlining
- Floating to fixed point conversion