

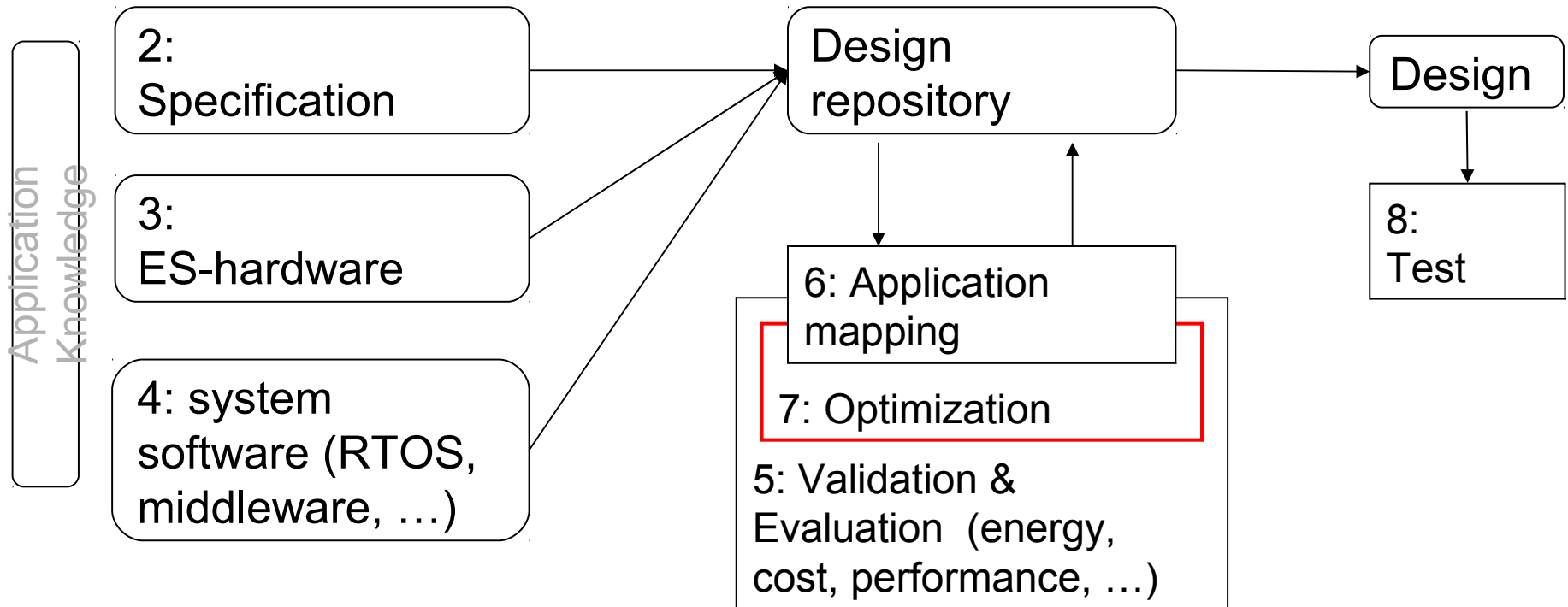
Optimizations

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2010/01/10



Structure of this course



Numbers denote sequence of chapters

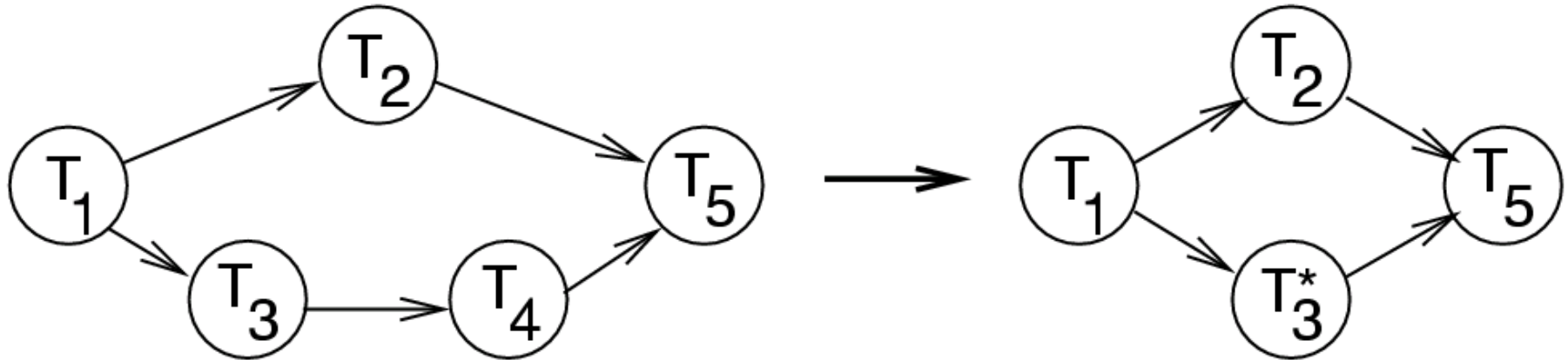
Task-level concurrency management

Granularity: size of tasks (e.g. in instructions)

Readable specifications and efficient implementations can possibly require different task structures.

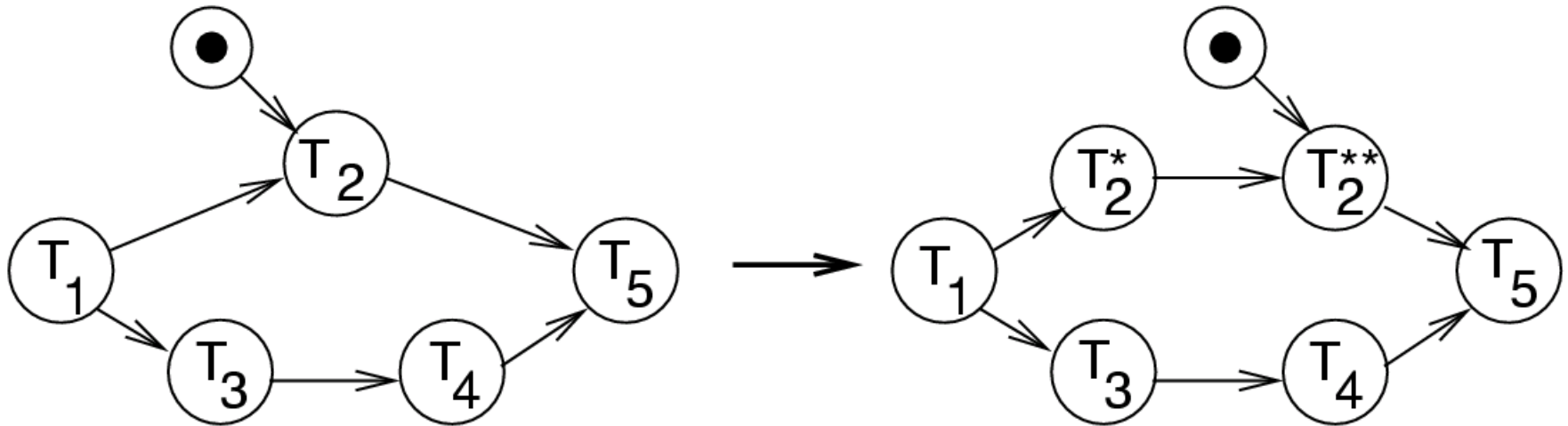
Granularity changes

Merging of tasks



Reduced overhead of context switches,
More global optimization of machine code,
Reduced overhead for inter-process/task communication.

Splitting of tasks



No blocking of resources while waiting for input,
more flexibility for scheduling, possibly improved result.

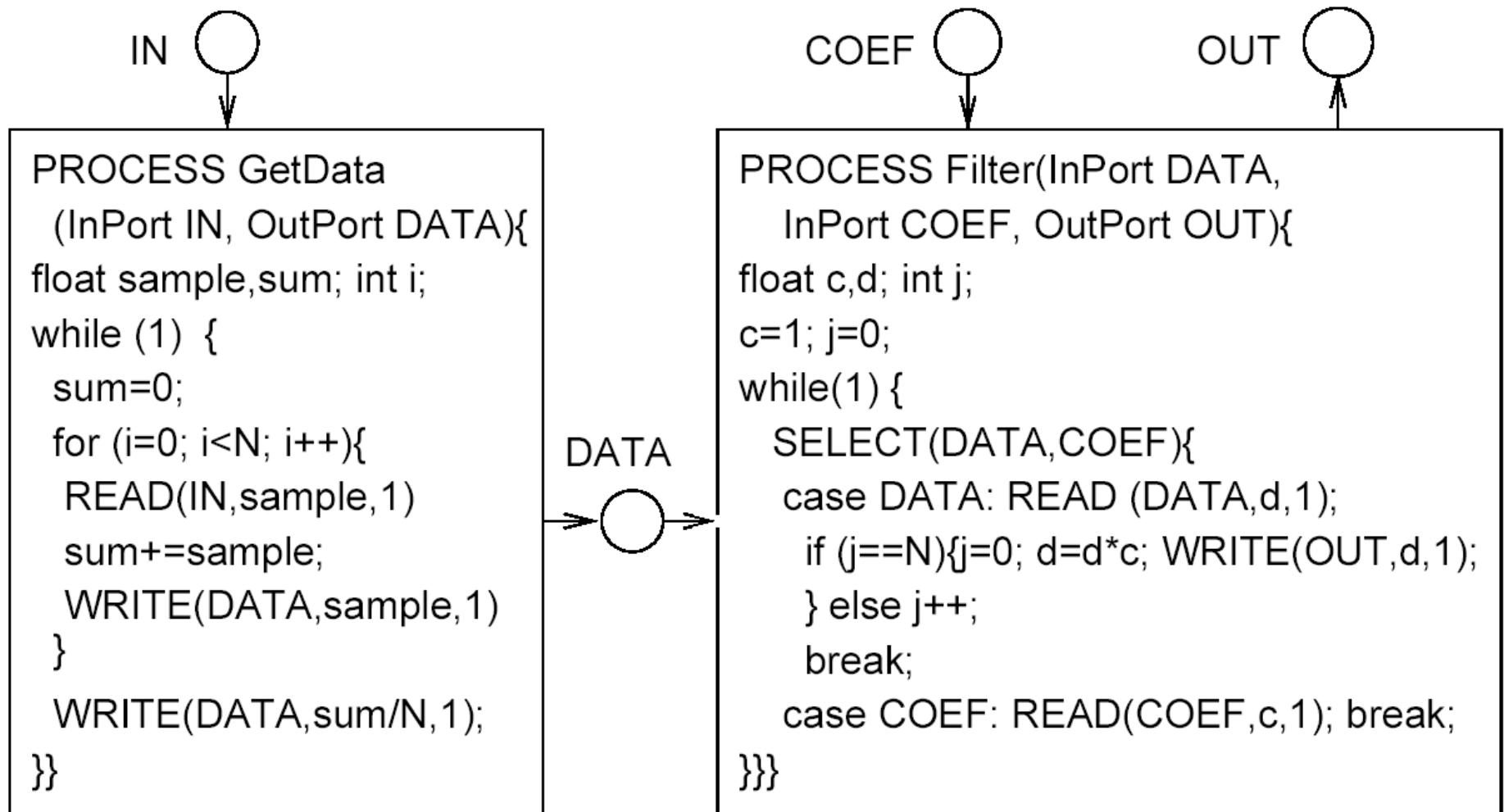
Merging and splitting of tasks

The most appropriate task graph granularity depends upon the context merging and splitting may be required.

Merging and splitting of tasks should be done automatically, depending upon the context.

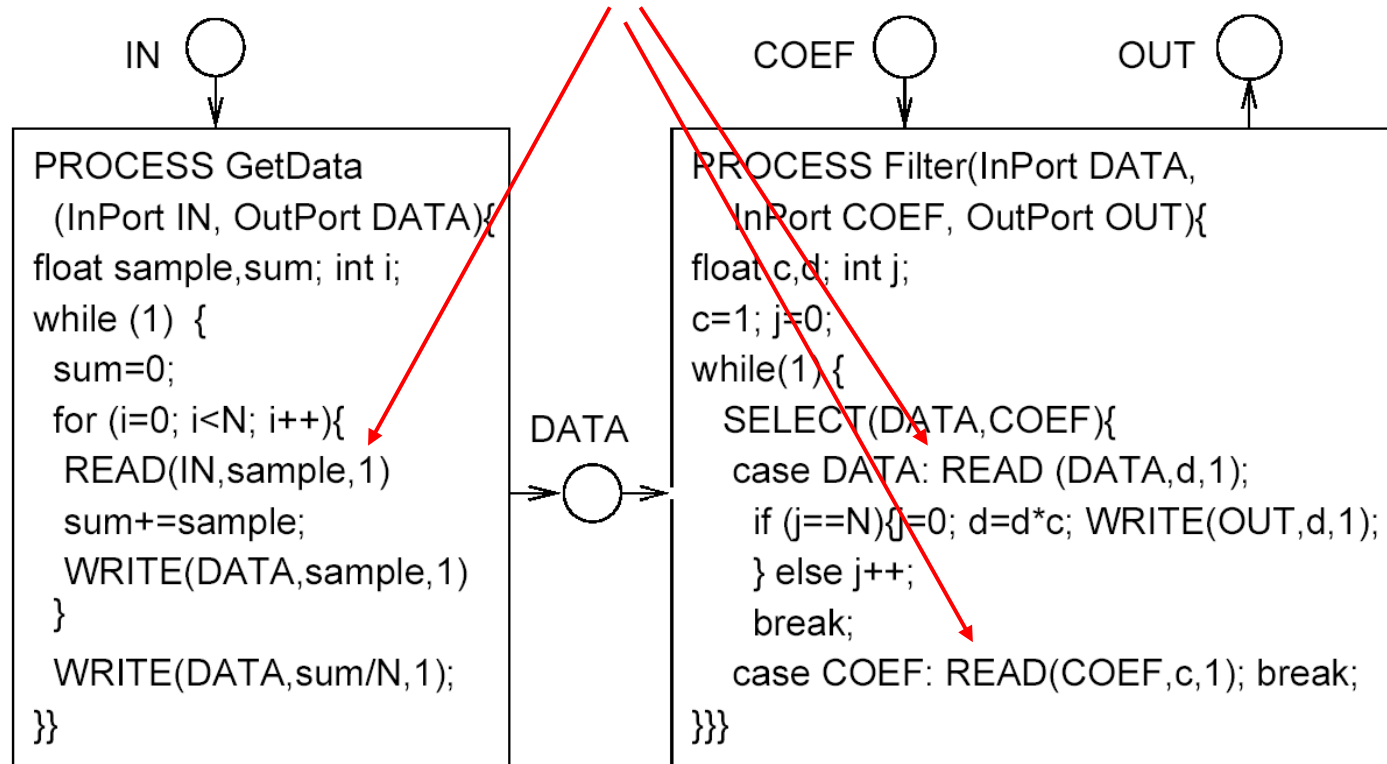
Automated rewriting of the task system

- Example -



Attributes of a system that needs rewriting

Tasks blocking after they have already started running



Work by Cortadella et al.

1. Transform each of the tasks into a Petri net,
2. Generate one global Petri net from the nets of the tasks,
3. Partition global net into “sequences of transition”
4. Generate one task from each such sequence

Mature, commercial approach not yet available

Result, as published by Cortadella

Reads only at the beginning

Initialization task

```
Init(){  
  sum=0;i=0;c=1;j=0;  
}
```

```
Tcoef(){  
  READ(COEF,c,1);  
}
```

```
Tin(){  
  IN ○  
  READ(IN,sample,1);  
  sum+=sample; i++;  
  DATA=sample, d=DATA;  
  if (j==N) {j=0; d=d*c; WRITE(OUT,d,1);  
             }else j++;  
  L0: if (i<N) return;  
  DATA=sum/N; d=DATA;  
  if (j==N) {j=0; d=d*c; WRITE(OUT,d,1);  
             }else j++;  
  sum=0; i=0; goto L0  
}
```



Never true

Always true

Optimized version of Tin

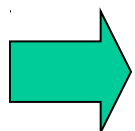
Never true



```
Tin(){
  READ(IN,sample,1);
  sum+=sample; i++;
  DATA=sample; d=DATA;
  if (j==N) {j=0; d=d*c; WRITE(OUT,d,1);
    }else j++;
L0: if (i<N) return;
  DATA=sum/N; d=DATA;
  if (j==N) {j=0; d=d*c; WRITE(OUT,d,1);
    }else j++;
  sum=0; i=0; goto L0
}
```

$j == i - 1$

j i



```
Tin () {
  READ (IN, sample, 1);
  sum += sample; i++;
  DATA = sample; d = DATA;
  L0: if (i < N) return;
  DATA = sum/N; d = DATA;
  d = d*c; WRITE(OUT,d,1);
  sum = 0; i = 0;
  return;
}
```

Always true

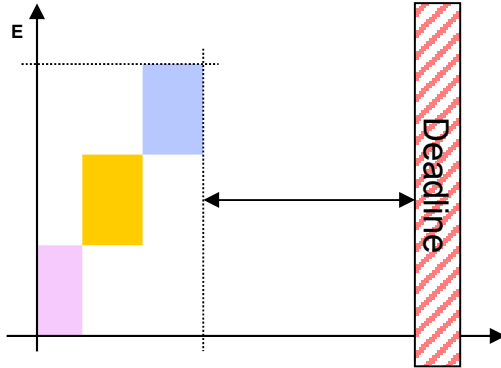
Task-level concurrency management (2)

- The dynamic behavior of applications getting more attention.
- Energy consumption reduction is the main target.
- Some classes of applications (i.e. video processing) have a considerable variation in processing power requirements depending on input data.
- Static design-time methods becoming insufficient.
- Runtime-only methods not feasible for embedded systems.

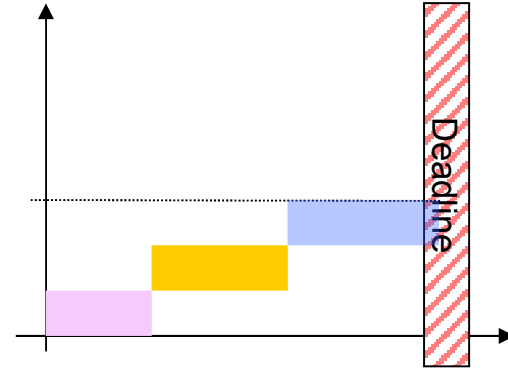
→ How about mixed approaches?

Example of a mixed TCM

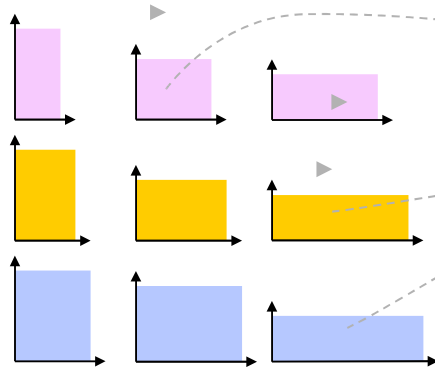
- Task1
- Task2
- Task3



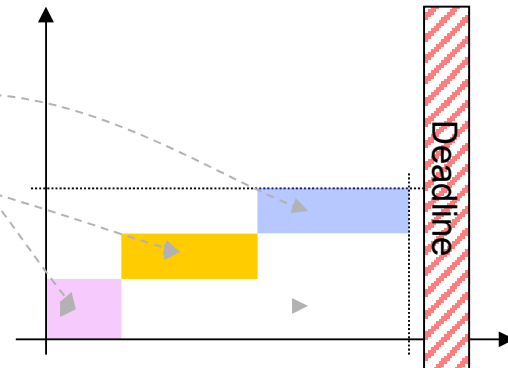
Static (compile-time) methods can ensure WCET feasible schedules, but waste energy in the average case.



...or they can define a probability for violating the deadline.



Mixed methods use compile-time analysis to define a set of possible execution parameters for each task.



Runtime scheduler selects the most energy saving, deadline preserving combination.

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

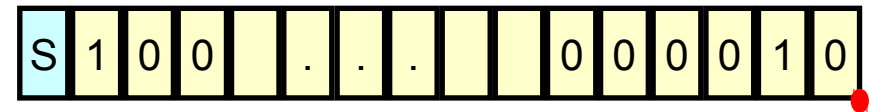
© Ki-Il Kum, et al. (Seoul National University): A Floating-point To Fixed-point C Converter For Fixed-point Digital Signal Processors, 2nd SUIF Workshop, 1996

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



(a) Integer



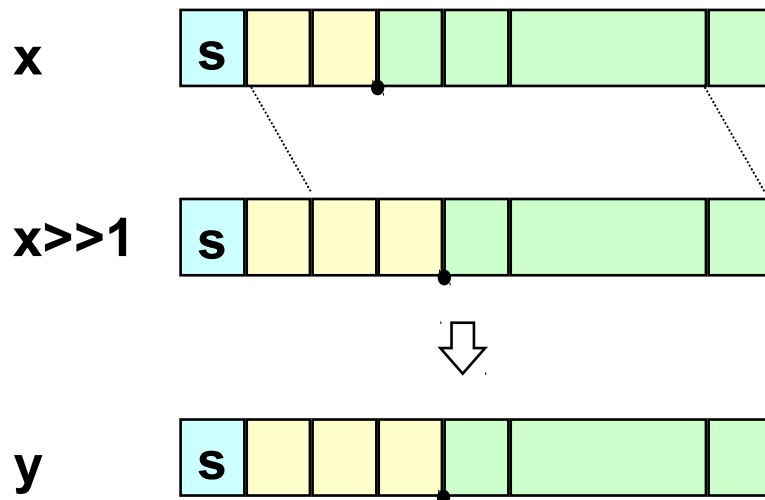
hypothetical binary point

(b) Fixed-Point

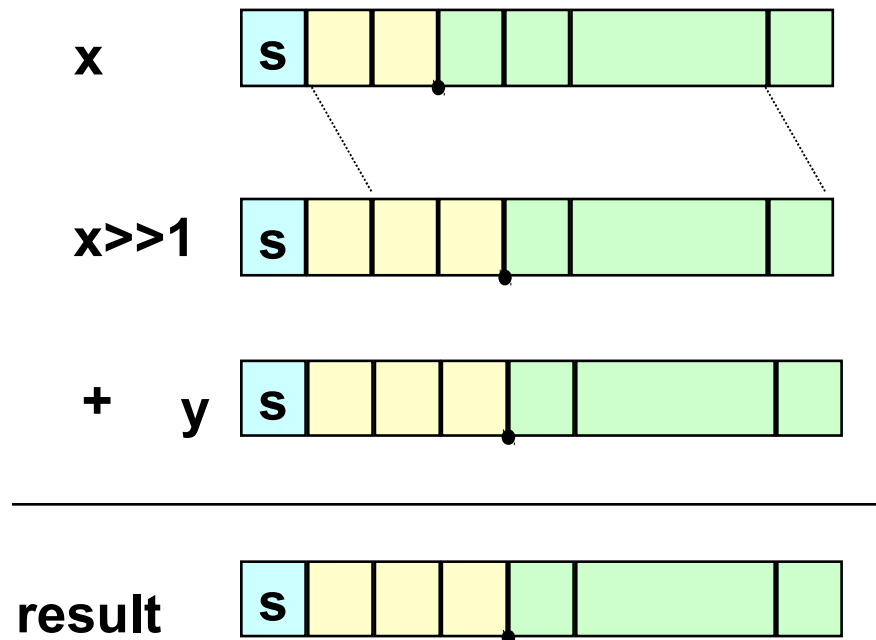
© Ki-Il Kum, et al

Assignment and Addition/Subtraction

Assume $y = x$, with
 $-x$ (IWL=2) and
 $-y$ (IWL=3):



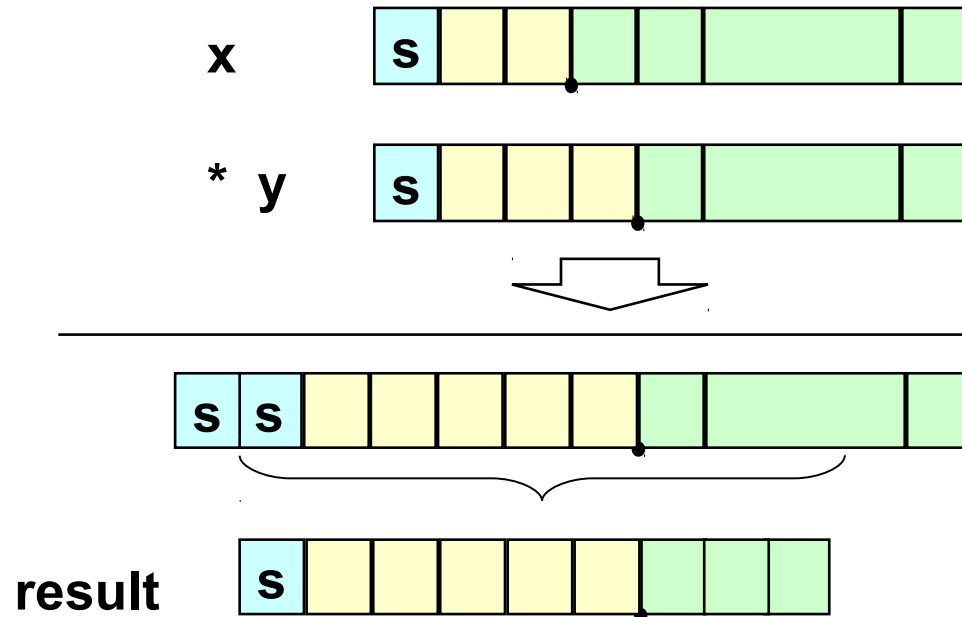
Let $\text{result} = x + y$:
 equalizing each IWL



© Ki-Il Kum, et al

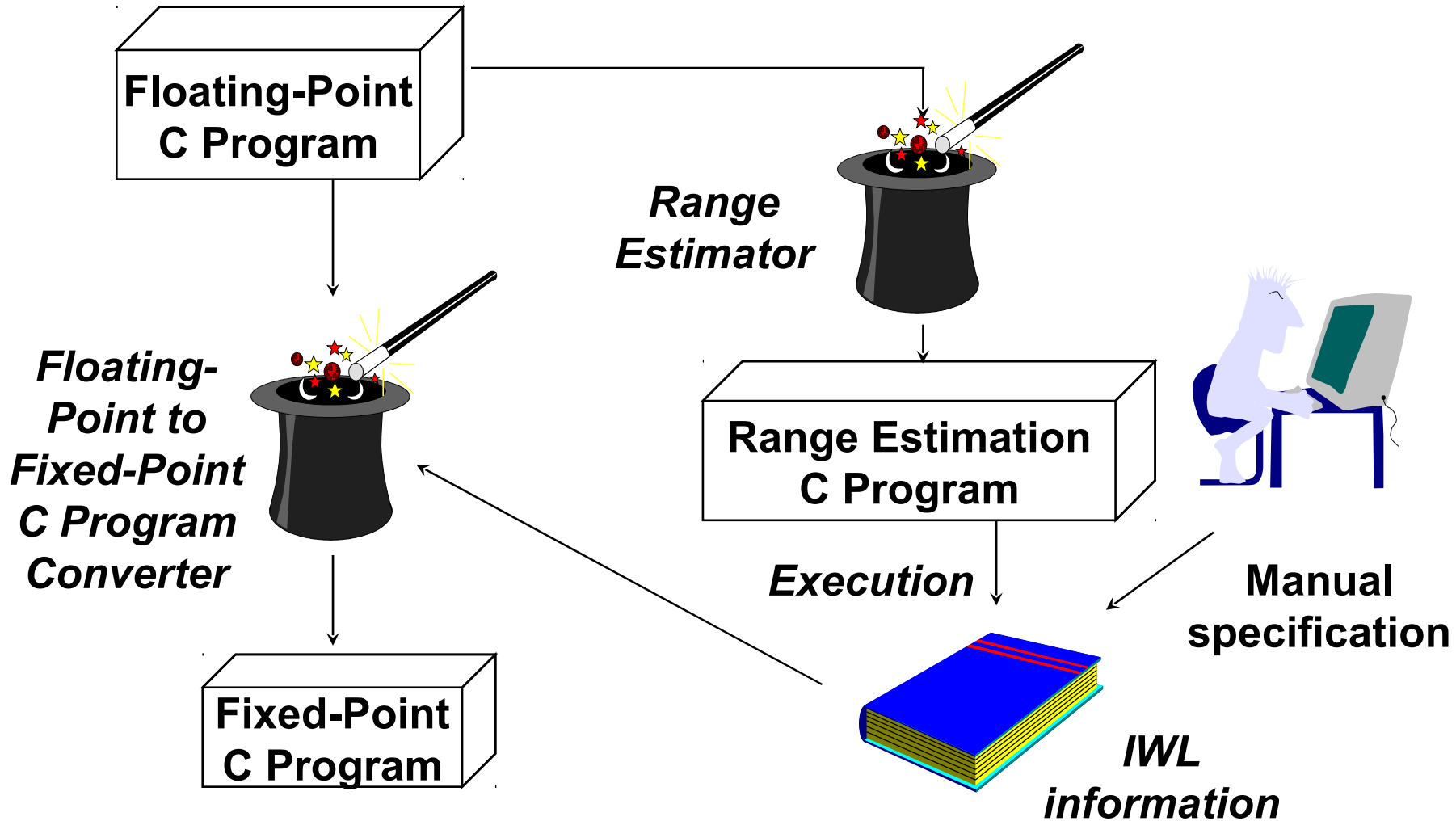
Multiplication

Assume result = $x * y$, with
_x (IWL=2) and
_y (IWL=3)
-> result (IWL=2+3)

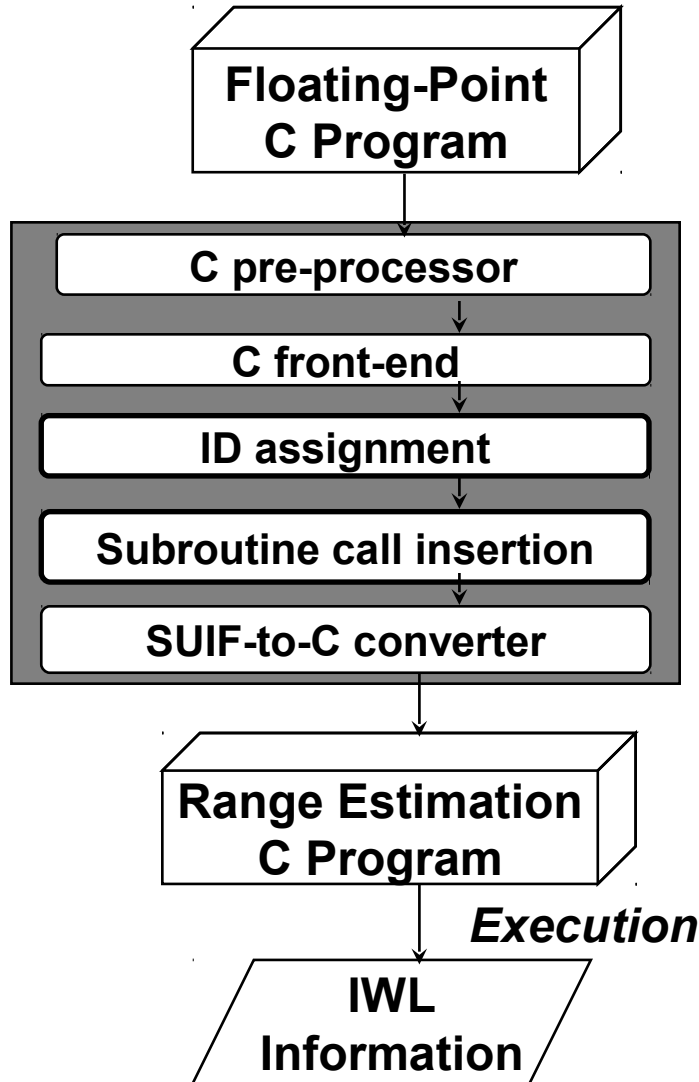


© Ki-Il Kum, et al

Development Procedure



Range Estimator



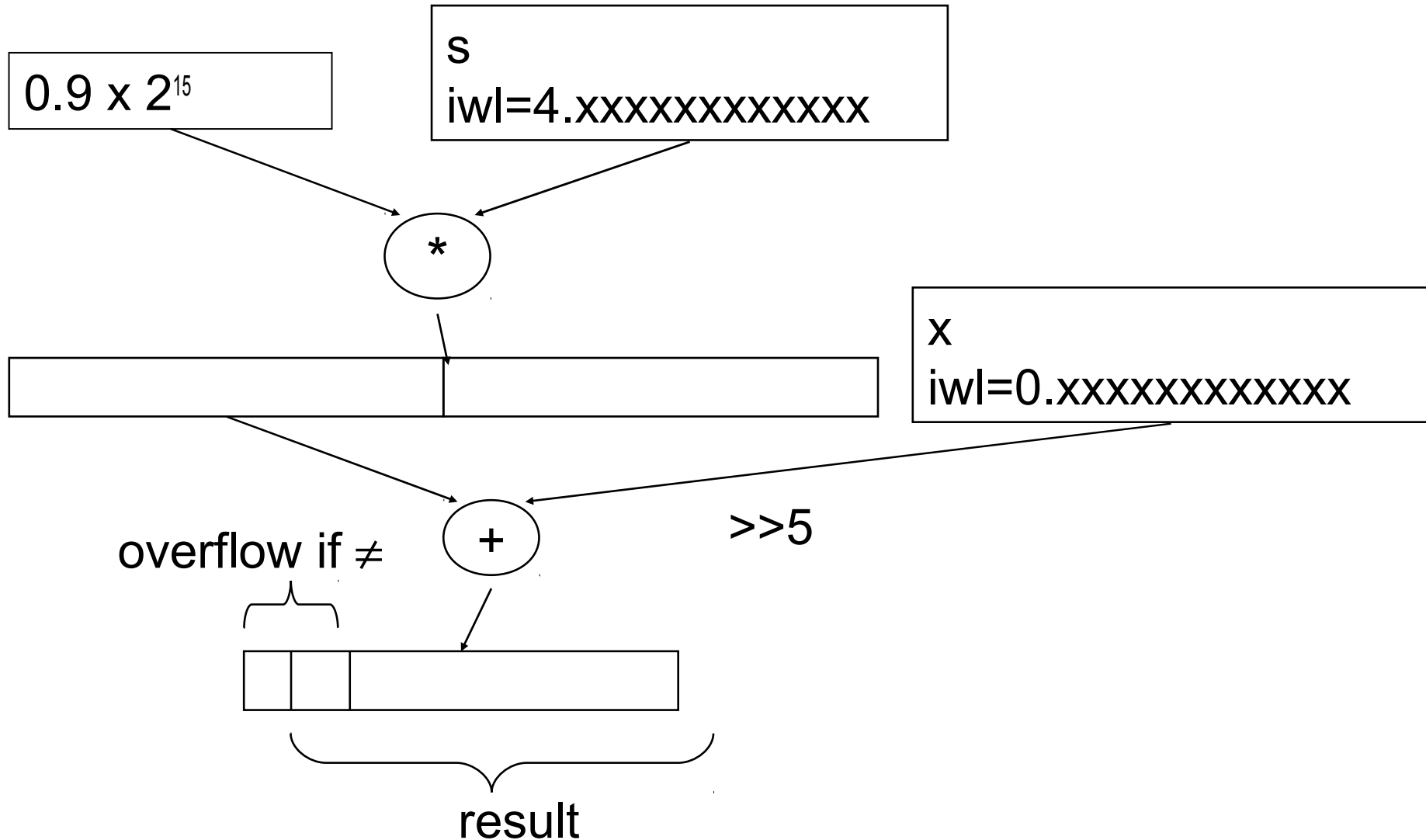
Range Estimation C Program

```
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;
}
```

Operations in fixed point program



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

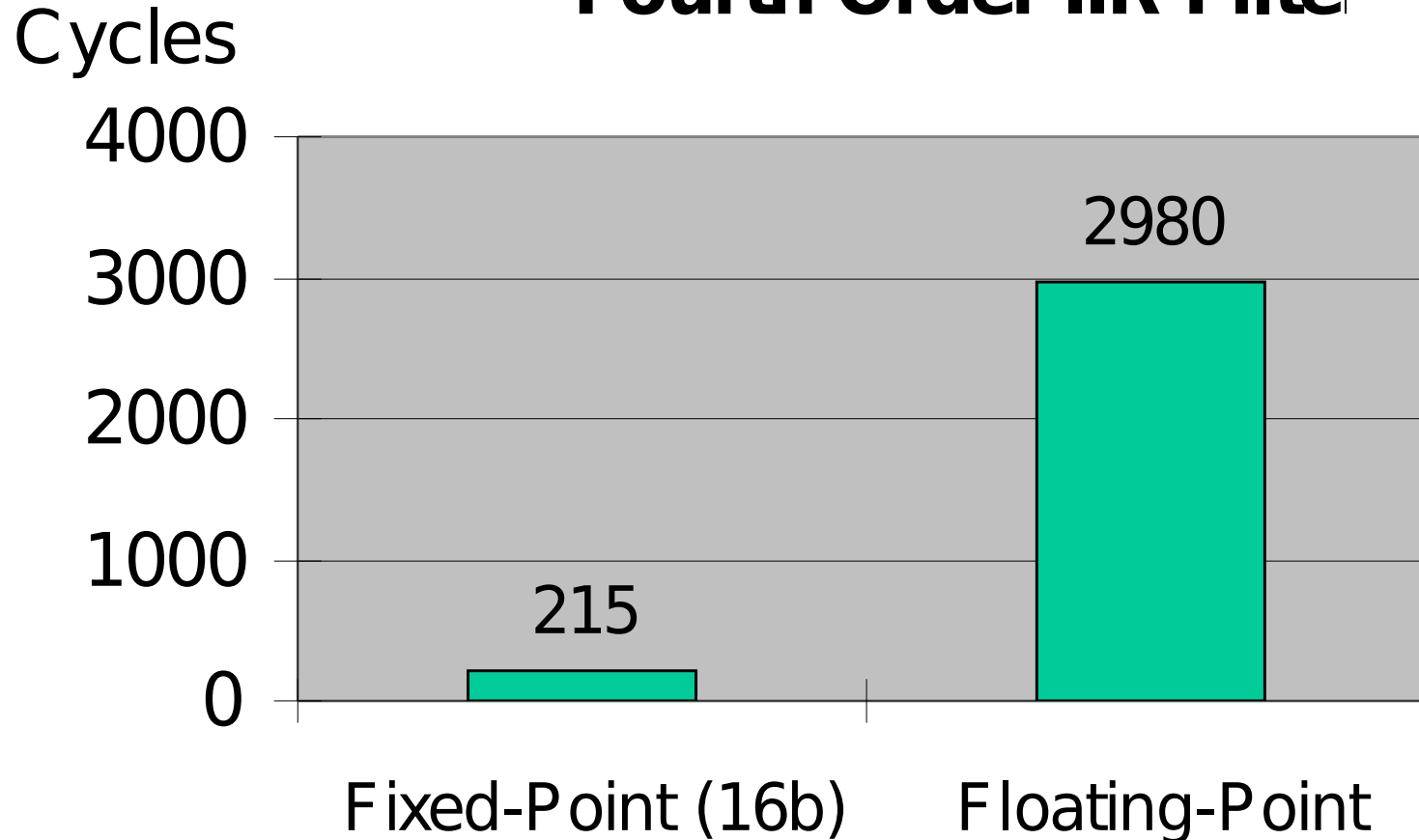
sll

- to remove 2nd sign bit
- opt. overflow check

© Ki-Il Kum, et al

Performance Comparison - Machine Cycles -

Fourth Order IIR Filter

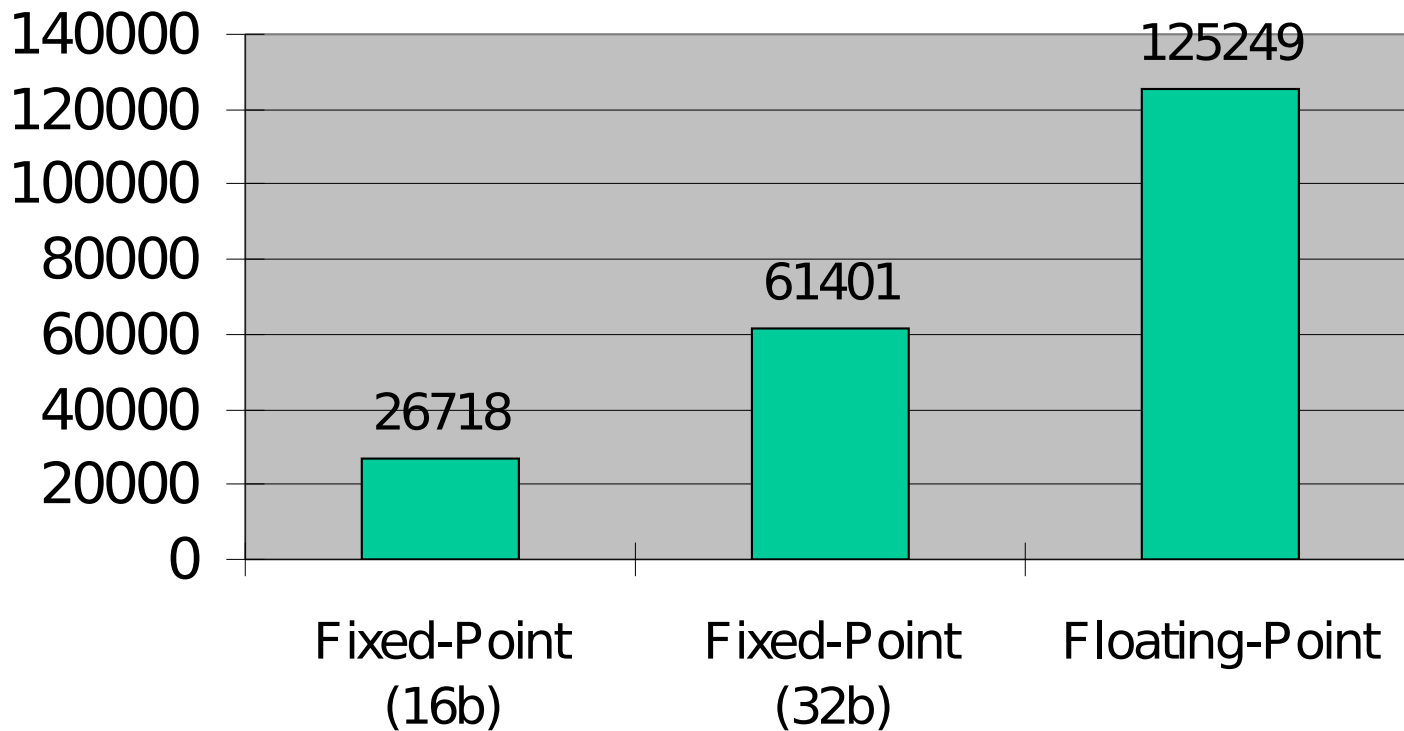


© Ki-Il Kum, et al

Performance Comparison - Machine Cycles -

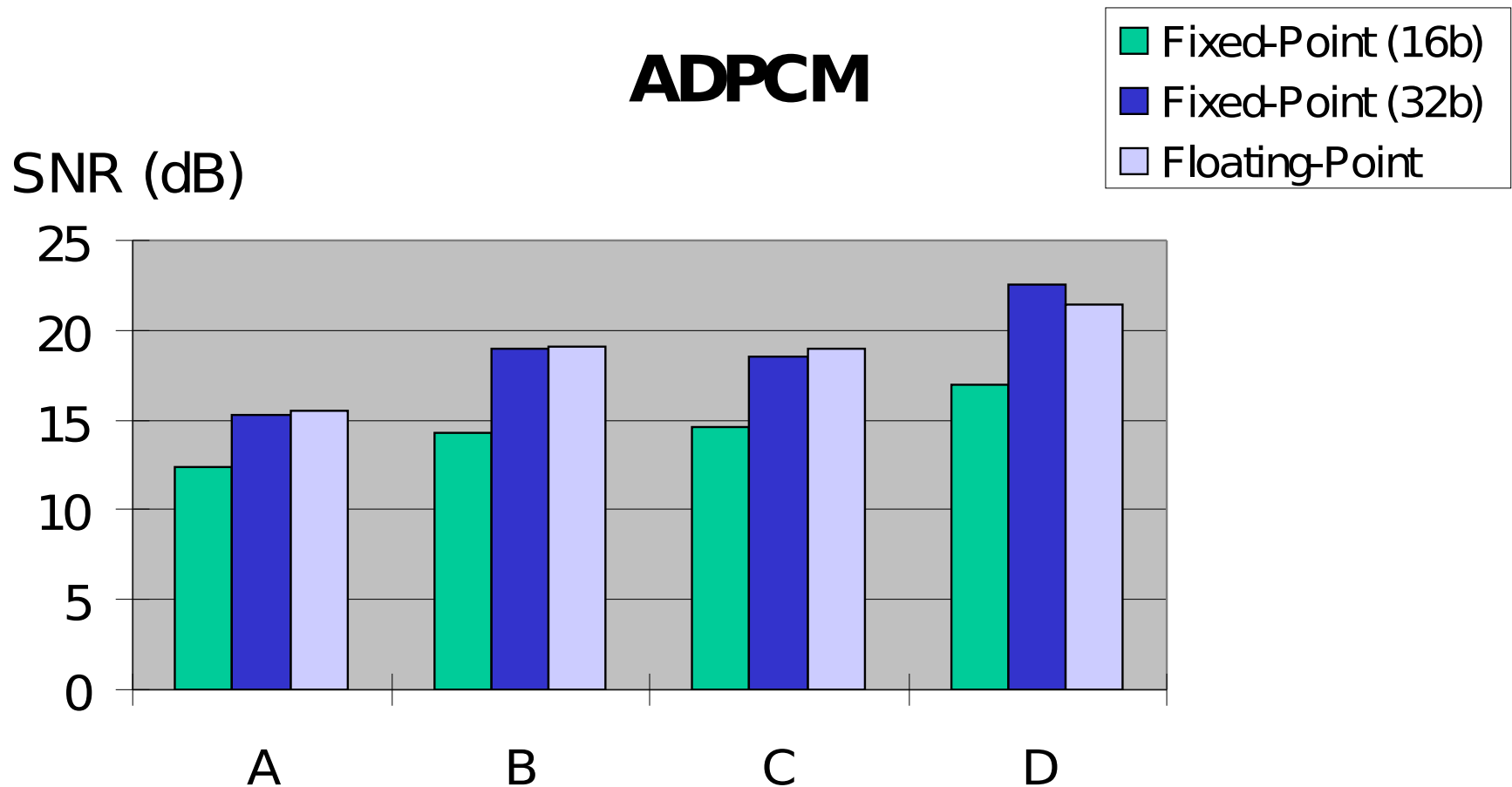
Cycles

ADPCM



© Ki-Il Kum, et al

Performance Comparison - SNR -



© *Ki-Il Kum, et al*

High-level software transformations

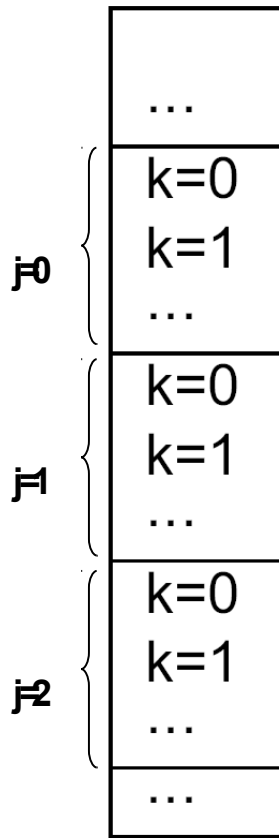
Peter Marwedel
TU Dortmund
Informatik 12
Germany



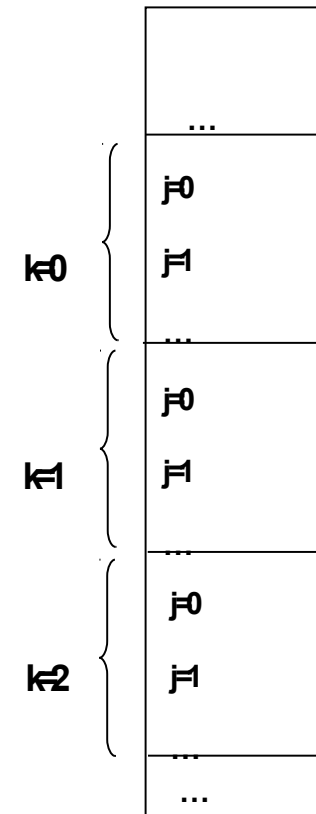
Impact of memory allocation on efficiency

Array $p[j][k]$

Row major order (C)



Column major order (FORTRAN)



Best performance if innermost loop corresponds to rightmost array index

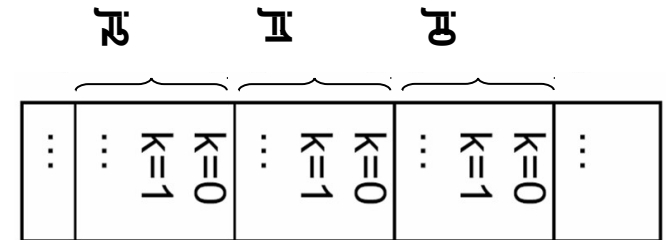
Two loops, assuming row major order (C):

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

```
for (j=0; j<=n; j++)
  for (k=0; k<=m; 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)

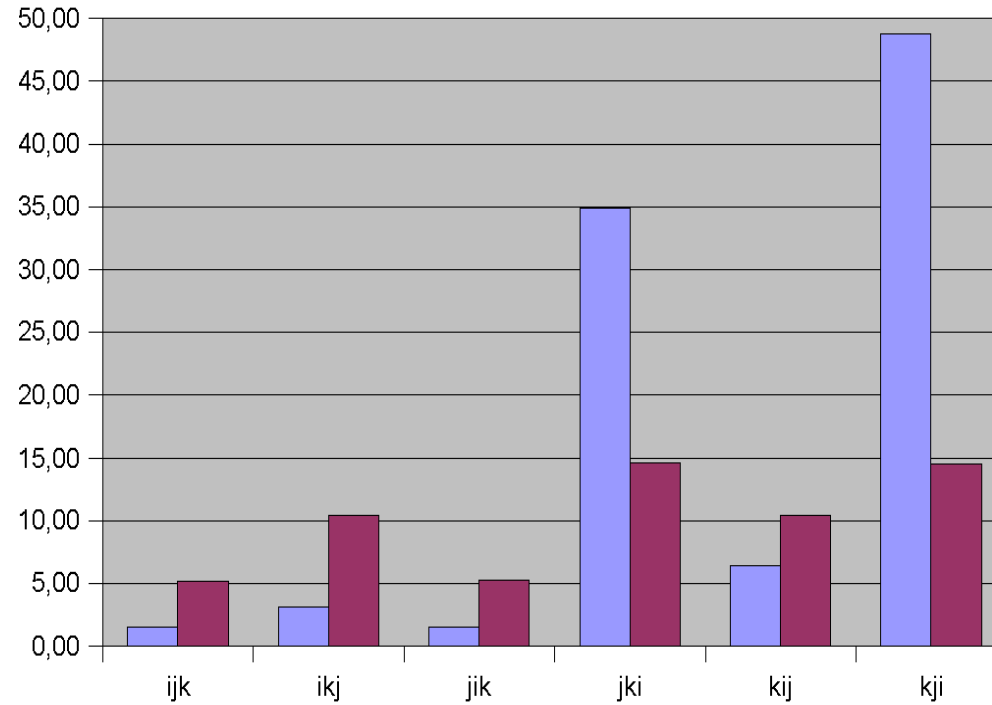
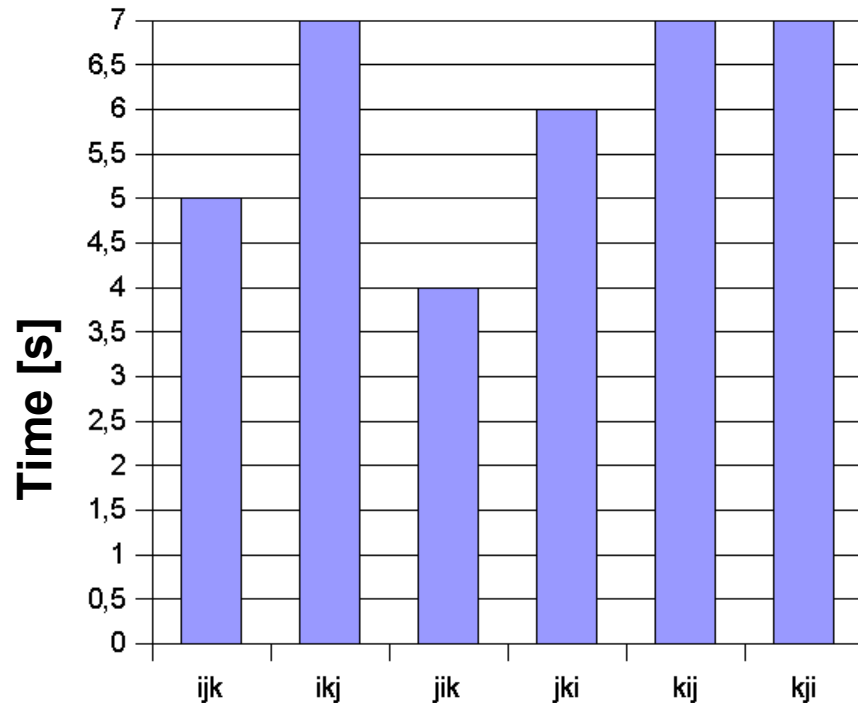
Results:

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 %



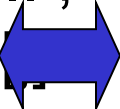
Not always the same impact ..

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

Transformations

“Loop fusion” (merging), “loop fission”

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



Loops small enough to **Better locality** for
allow zero overhead access to p.

Loops Better chances for
parallel execution.

Which of the two versions is best?

Architecture-aware compiler should select best version.

Example: simple loops

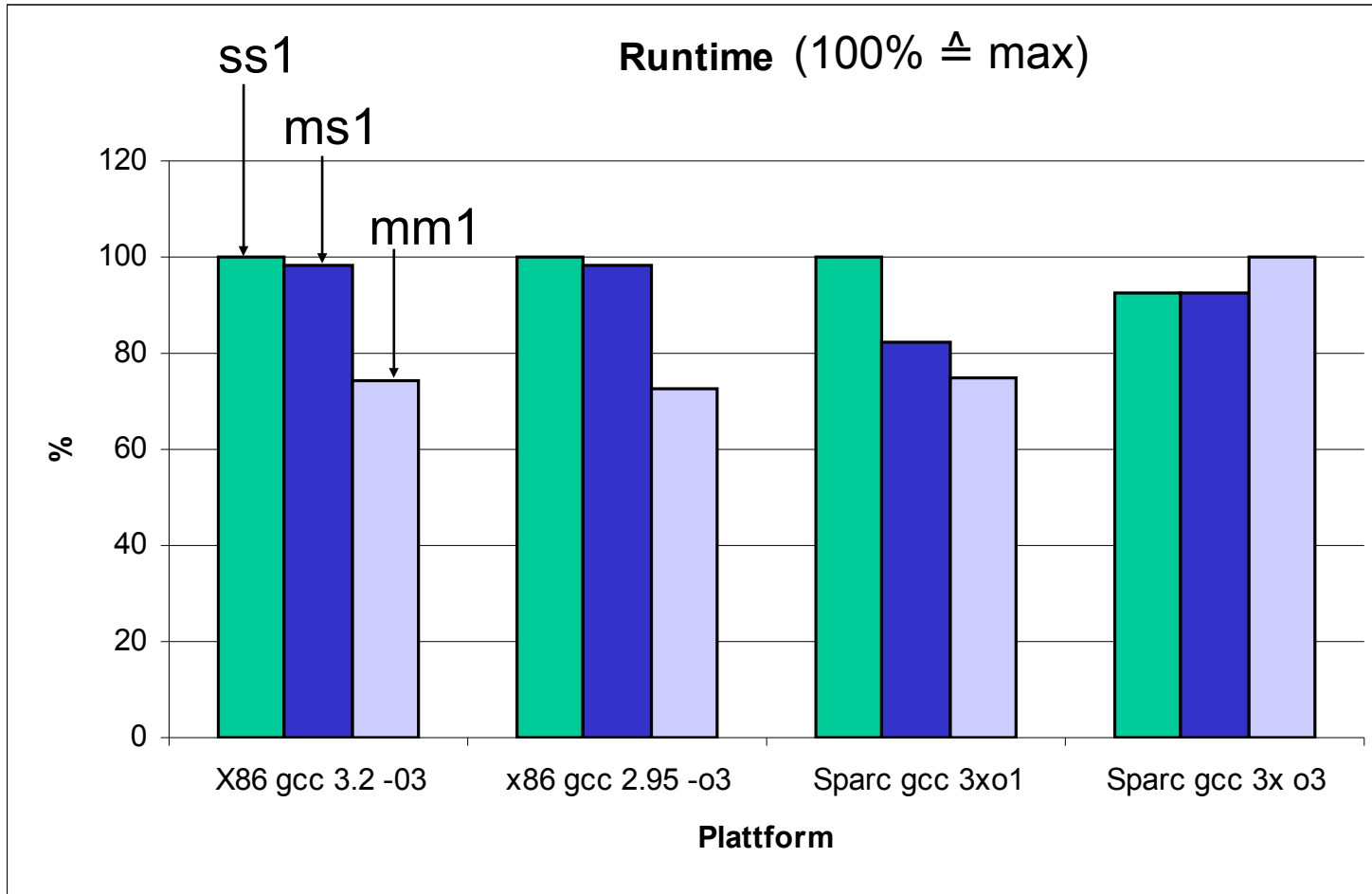
```
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];
```

```
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;}}}
```

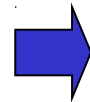
Results: simple loops



Merged loops superior; except Sparc with -o3

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.

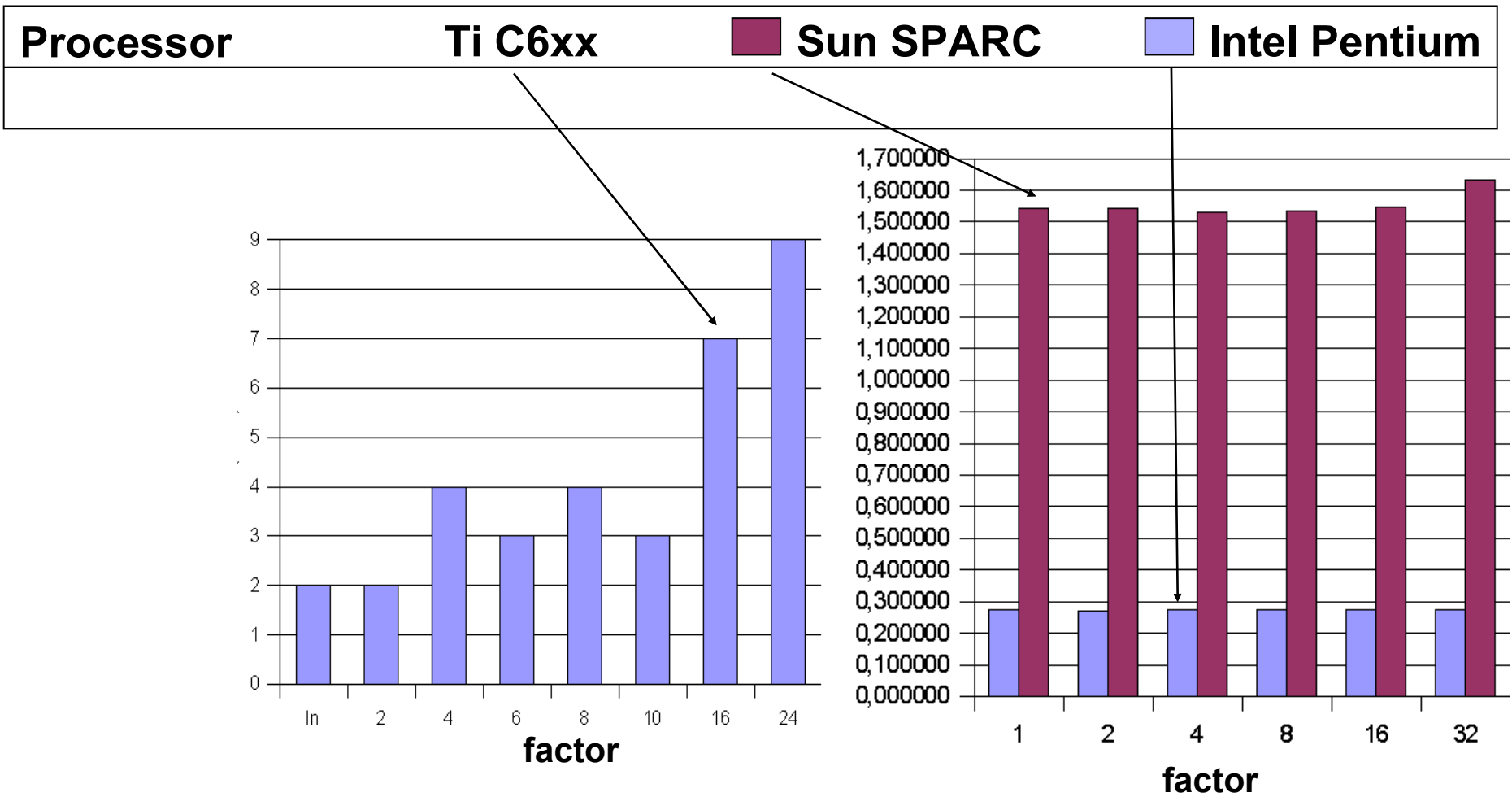
Tradeoff between code size
and improvement.

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];
*suiif_tmp=
*suiif_tmp+a[i][j]*b[j][k];}
{int *suif_tmp;
suif_tmp=&c[i][k+1];
*suiif_tmp=*suiif_tmp
+a[i][j]*b[j][k+1];
}}}}
return;}
```

Results



Benefits quite small; penalties may be large

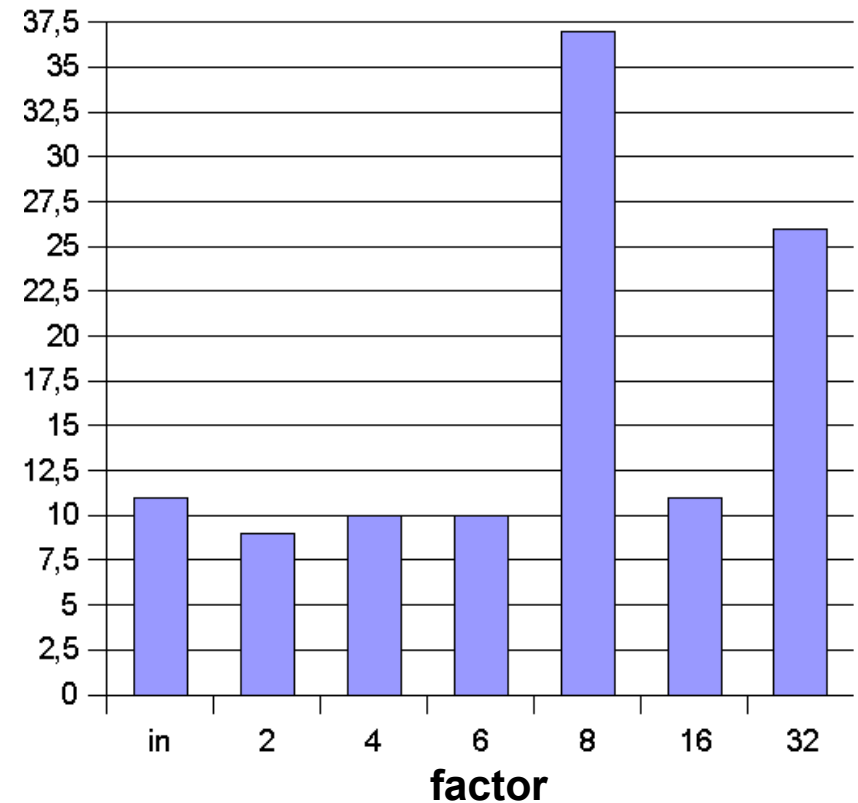
[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

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;



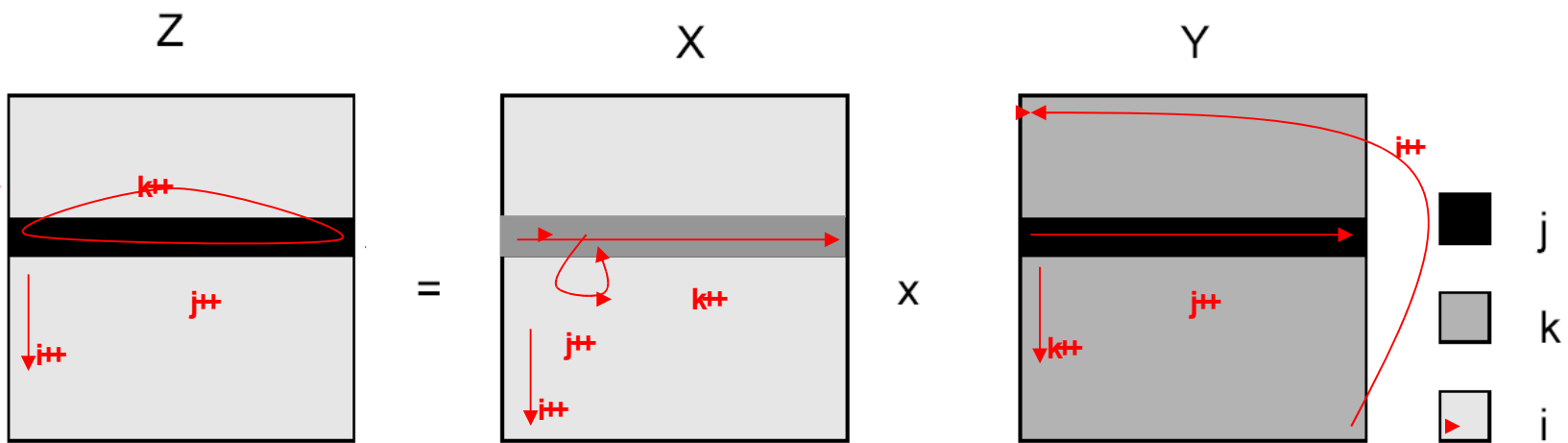
[Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004]

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 -

```

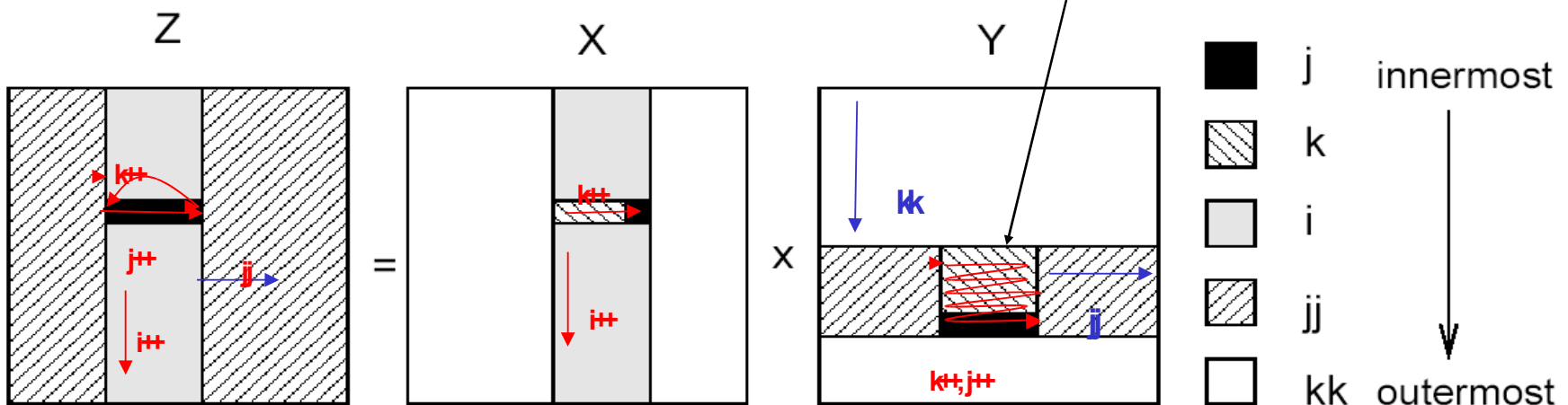
for (kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
    for (i=1; i<= N; i++)
      for (k=kk; k<= min(kk+B-1,N); k++){
        r=X[i][k]; /* to be allocated to a register*/
        for (j=jj; j<= min(jj+B-1, N); j++)
          Z[i][j] += r* Y[k][j]
      }
  
```

Reuse factor of
B for Z, N for Y

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

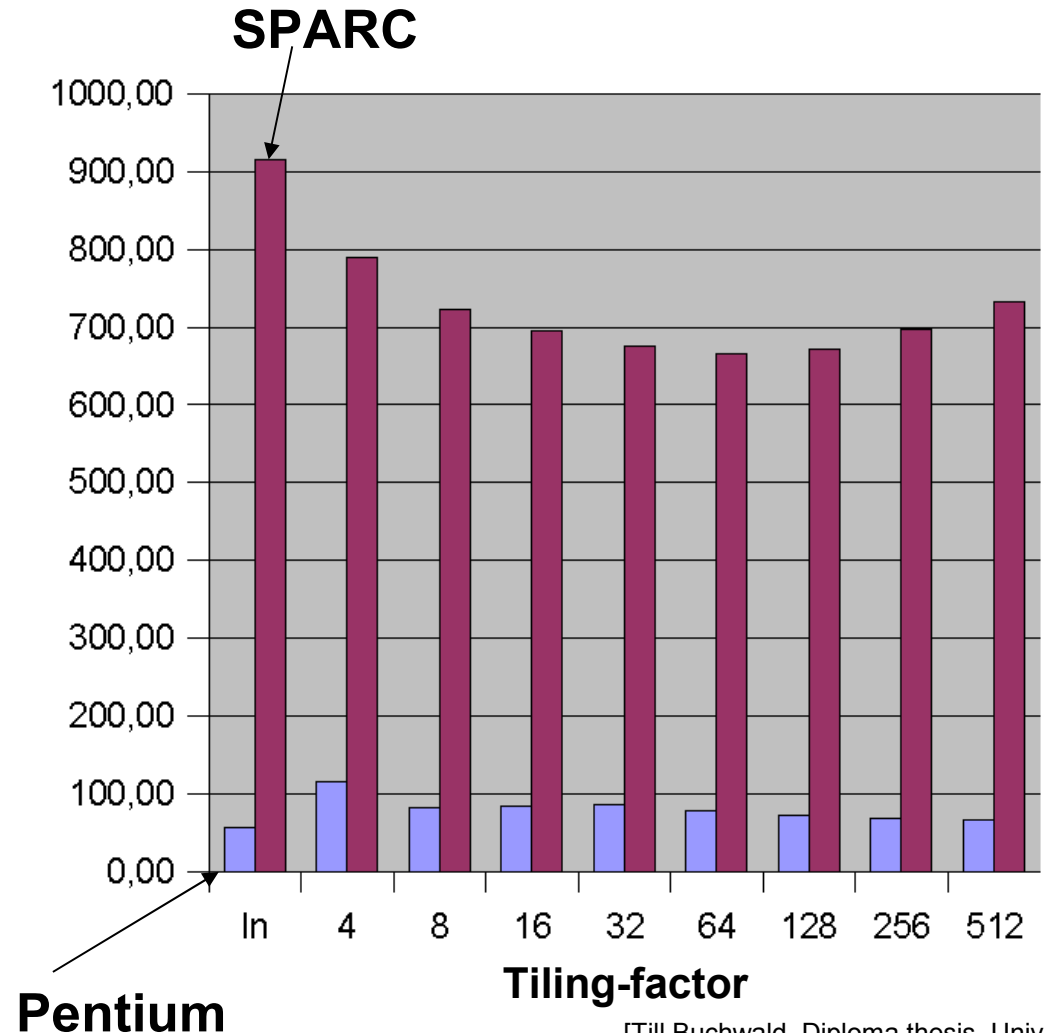
*Compiler
should select
best option*

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]

Summary

- Task concurrency management
 - Re-partitioning of computations into tasks
 - Dynamic exploitation of slack
- Floating-point to fixed point conversion
 - Range estimation
 - Conversion
 - Analysis of the results
- High-level loop transformations
 - Fusion
 - Unrolling
 - Tiling