Rechnerarchitektur (RA)

Sommersemester 2016

Architecture-Aware Optimizations
-Software Optimizations-

Jian-Jia Chen
Informatik 12
Jian-jia.chen@tu-..
http://ls12-www.cs.tu-dortmund.de/daes/
Tel.: 0231 755 6078

2016/07/13

High-level optimizations

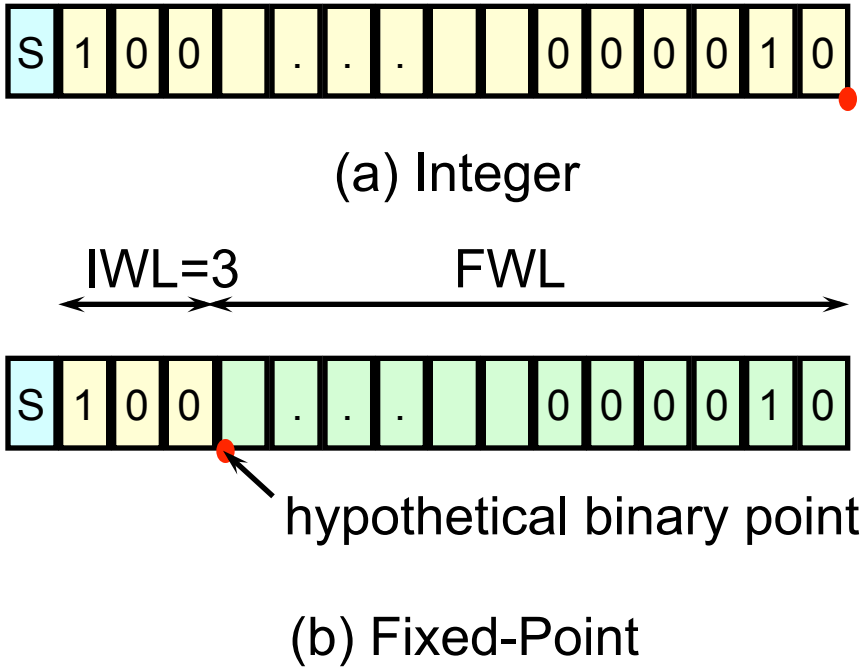
- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

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*

Floating-point to fixed point conversion

Pros:

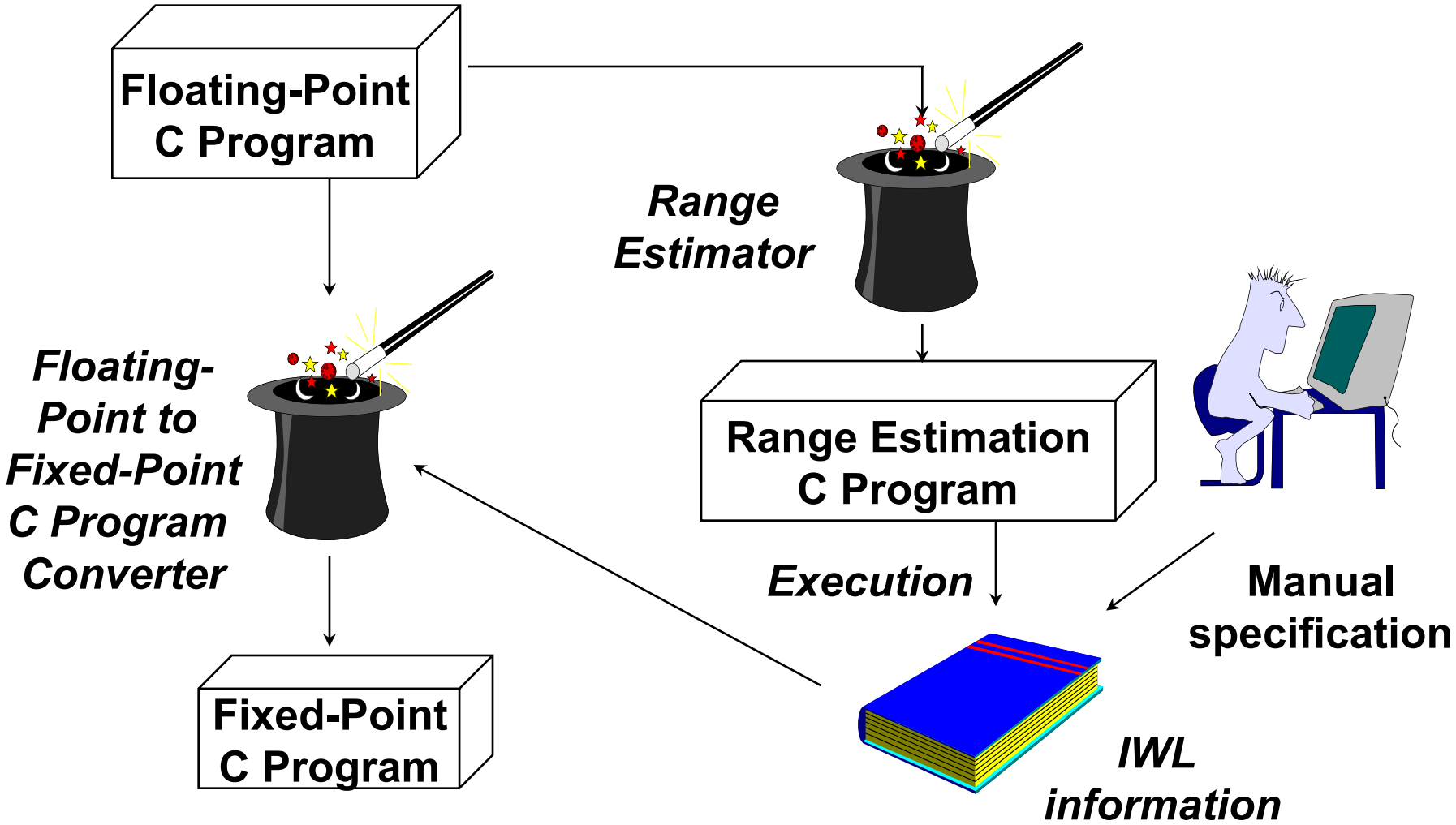
- Lower cost
- Faster
- Lower power consumption
- Sufficient SQNR, *if properly scaled*
- Suitable for portable applications

Cons:

- Decreased dynamic range
- Finite word-length effect, *unless properly scaled*
    - Overflow and excessive quantization noise
- Extra programming effort

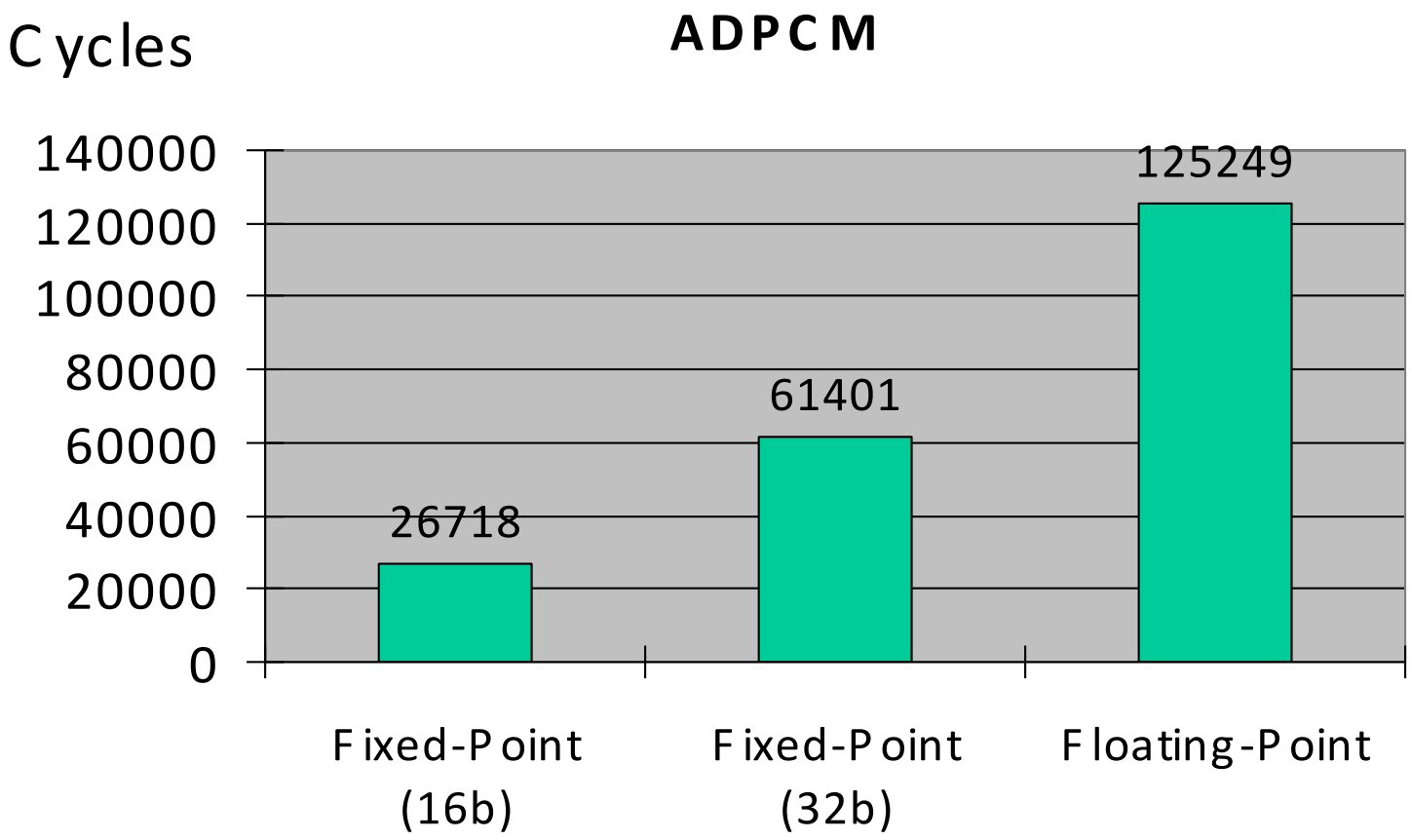
© *Ki-Il Kum, et al. (*Seoul National University*):* A Floating-point To Fixed-point C Converter For Fixed-point Digital Signal Processors, 2nd SUIF Workshop, 1996

Development Procedure

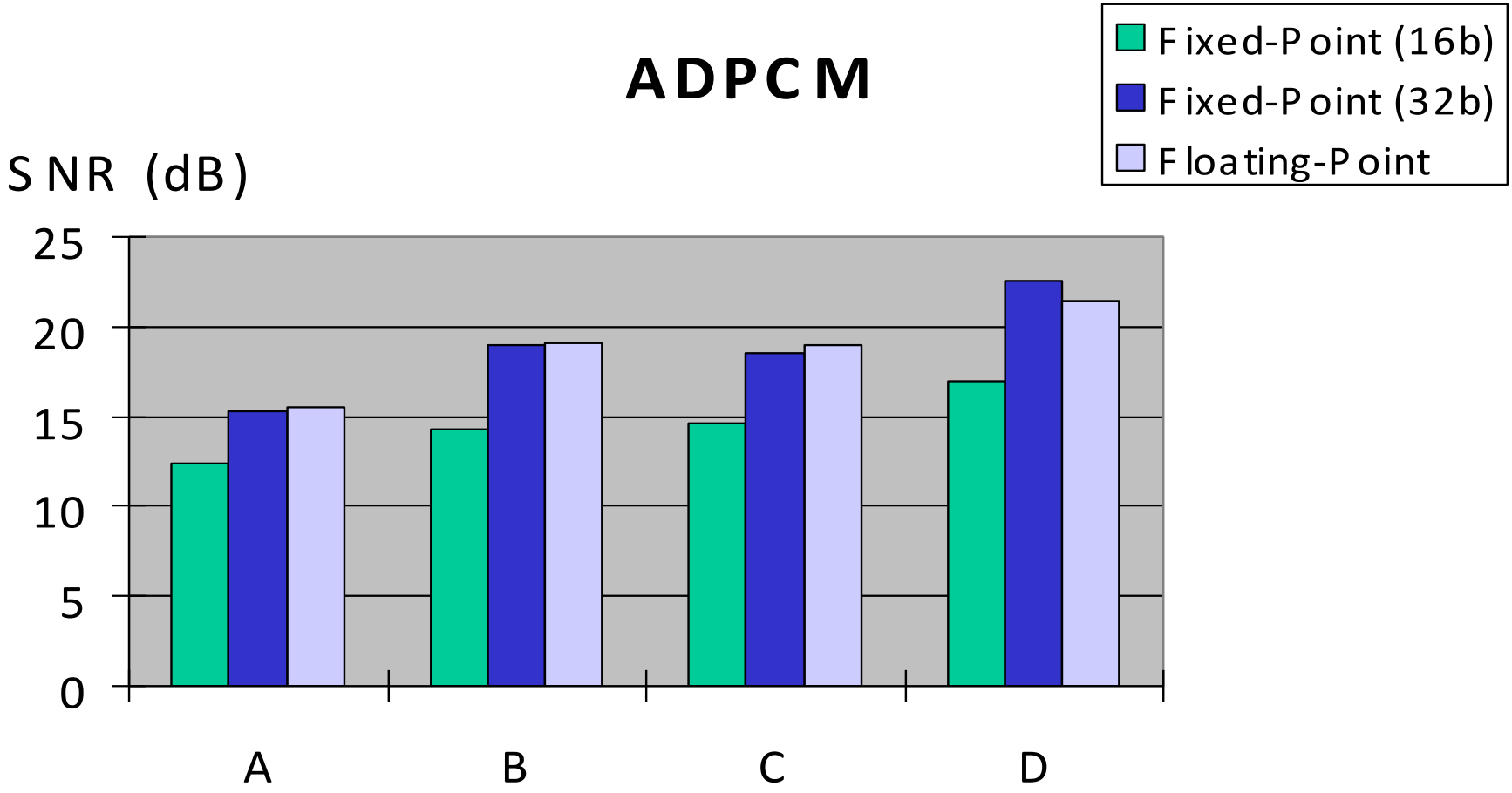


© Ki-Il Kum, et al

Performance Comparison - Machine Cycles -



Performance Comparison - SNR -



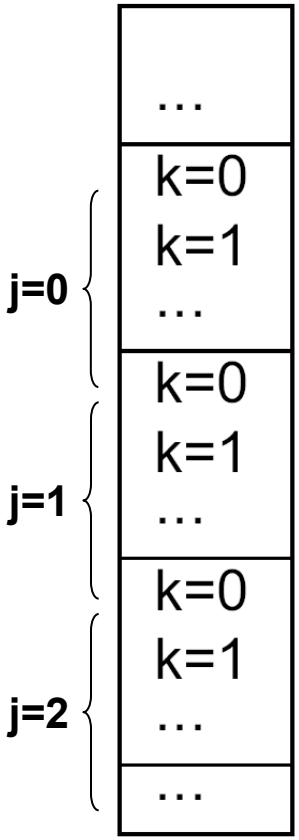
High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

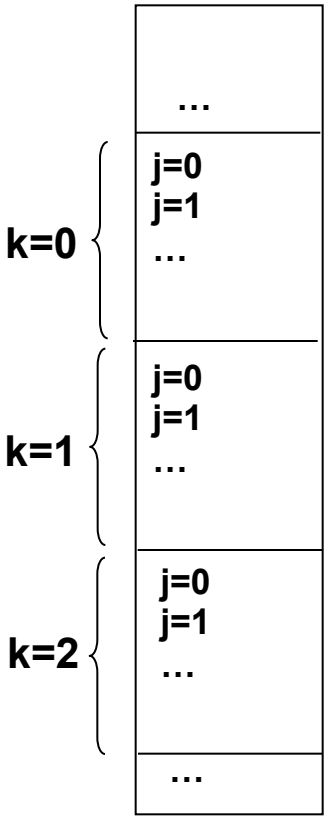
Impact of memory allocation on efficiency

Array p[j][k]

Row major order (C)



Column major order (FORTRAN)



Best performance of innermost loop corresponds to rightmost array index

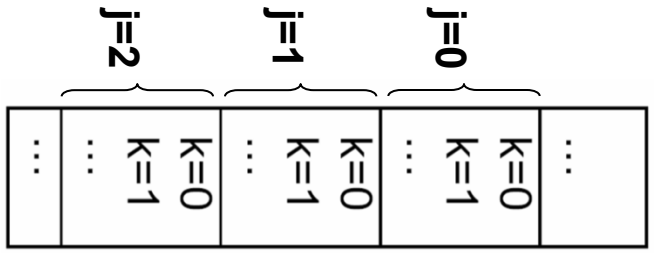
Two loops, assuming row major order (C):

```
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 homogeneous memory access, but:

For row major order



↑ Poor cache behavior Good cache behavior ↑

☞ memory architecture dependent optimization

☞ Program transformation “Loop interchange”

☞ Improved locality

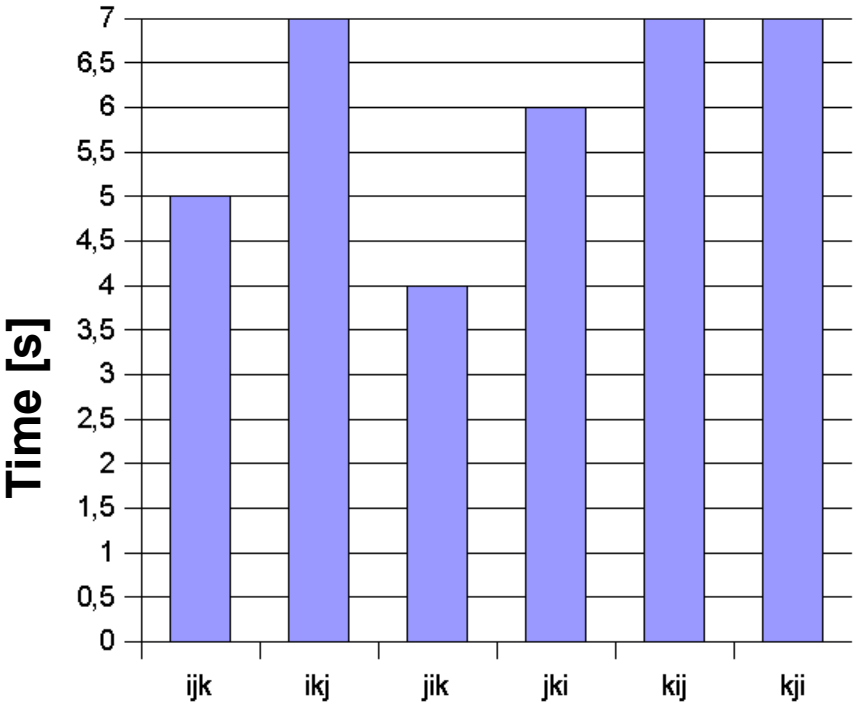
```
Example:
…#define iter 400000
int a[20][20][20];
void computeijk() {int i,j,k;
   for (i = 0; i < 20; i++) {
       for (j = 0; j < 20; j++) {
             for (k = 0; k < 20; k++) {
                   a[i][j][k] += a[i][j][k];}}}}
void computeikj() {int i,j,k;
   for (i = 0; i < 20; i++) {
       for (j = 0; j < 20; j++) {
             for (k = 0; k < 20; k++) {
                   a[i][k][j] += a[i][k][j] ;}}}}…
```

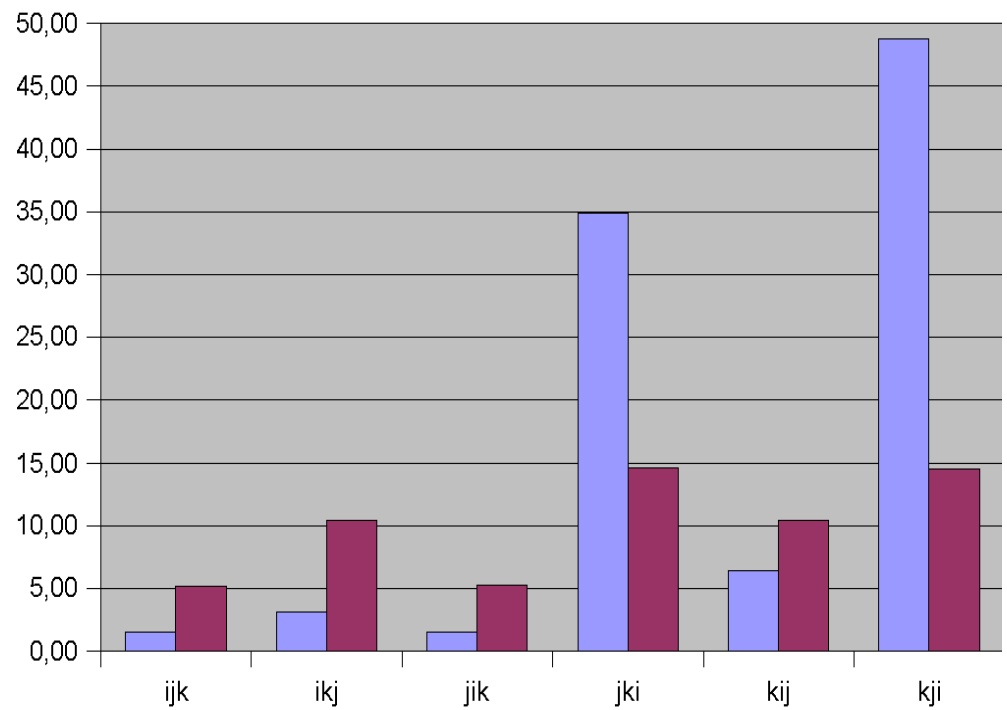
Results:
strong influence of the memory architecture

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

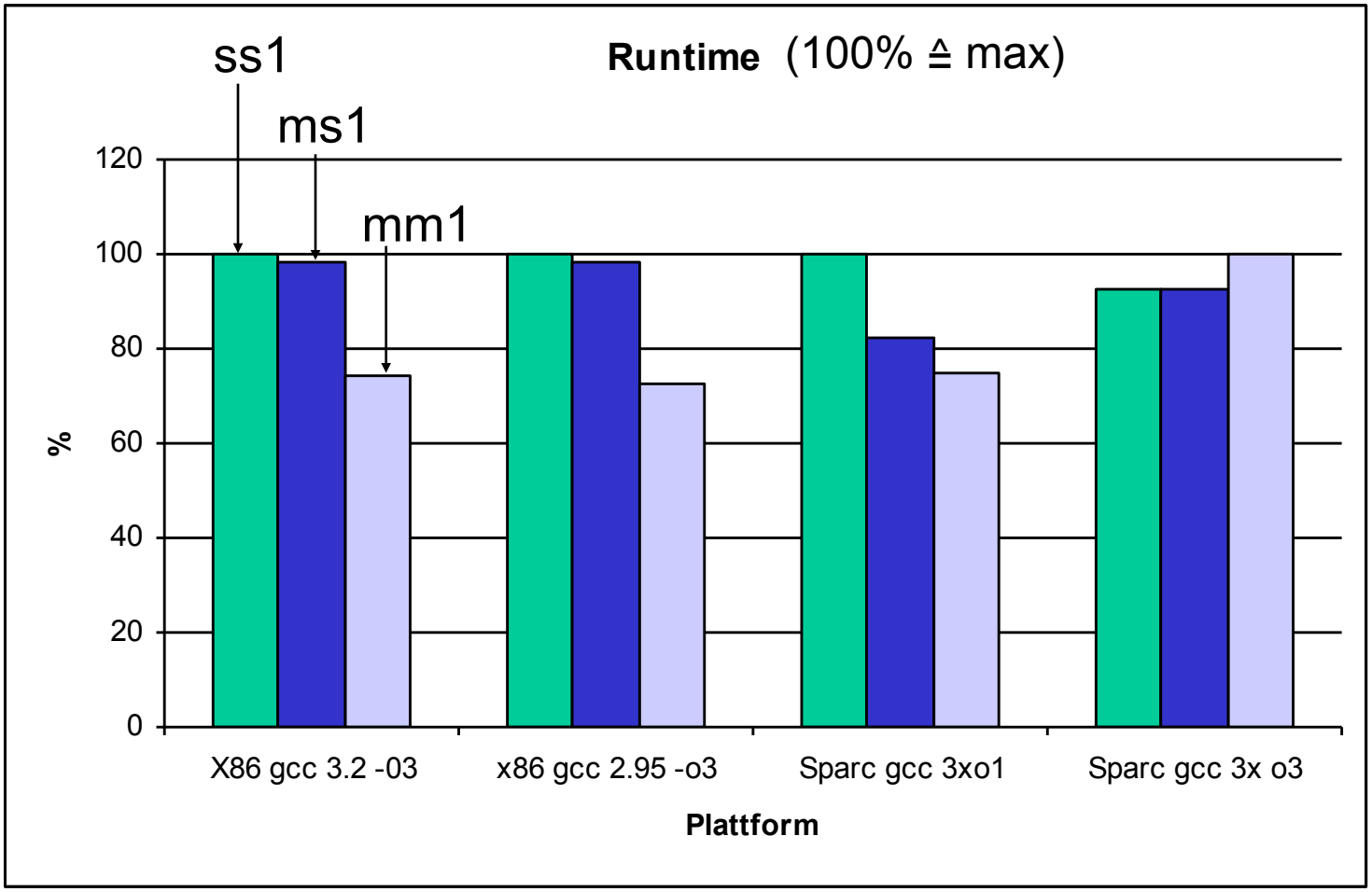
```
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];
```

```
void 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 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 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.

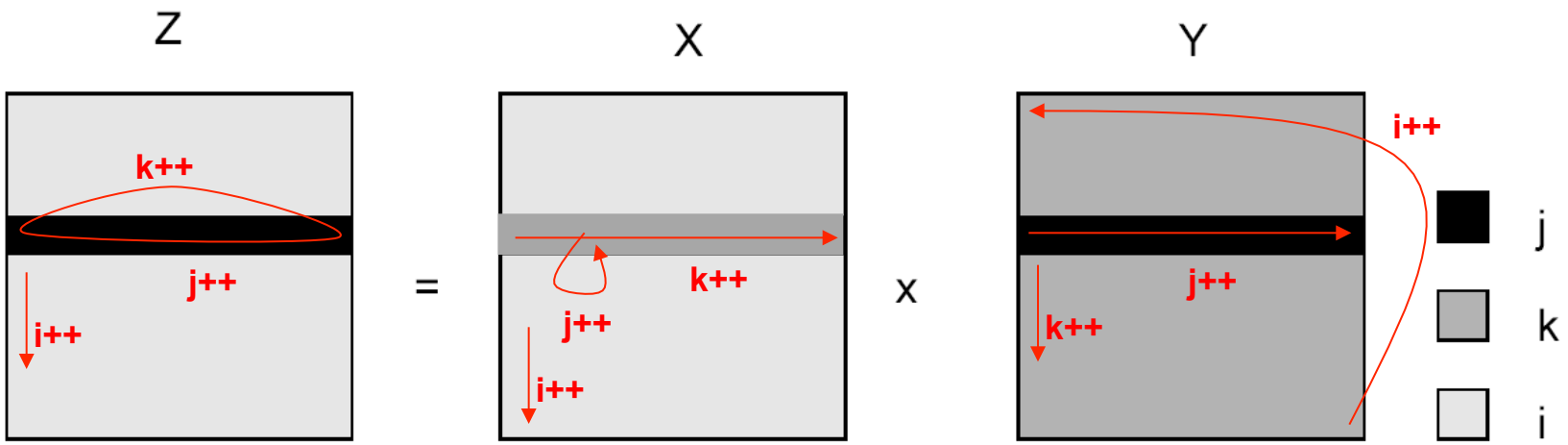
Extreme case: completely unrolled loop (no branch).

High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

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 (O(N³) references for Z).
```



Loop tiling/loop blocking
- tiled version -

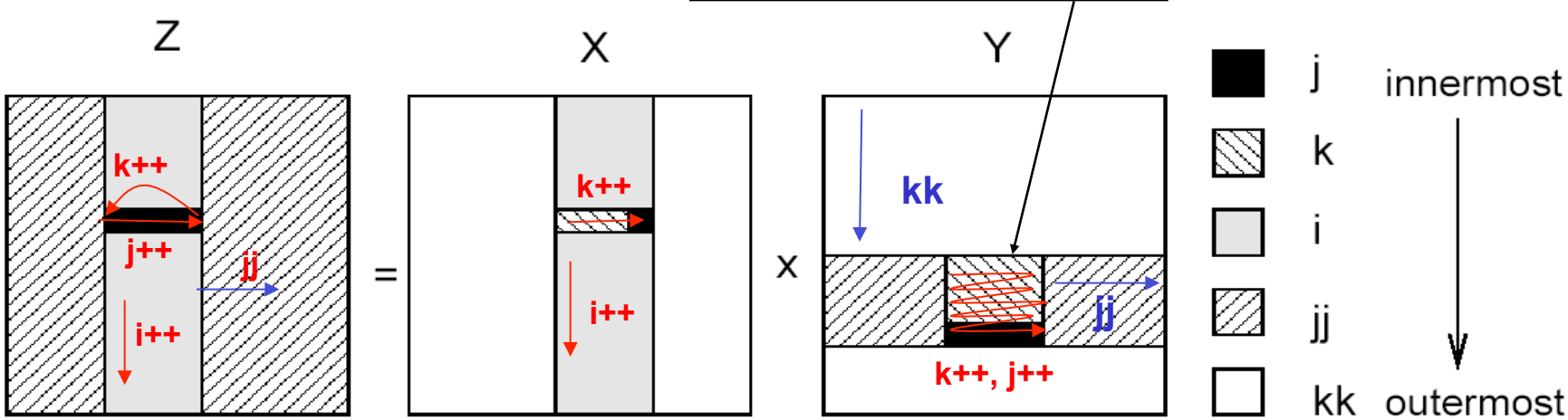
```
for (kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
    for (i=1; i<= N; i++)
      for (k=kk; k<= min(kk+B-1,N); k++){
        r=X[i][k]; /* to be allocated to a register*/
        for (j=jj; j<= min(jj+B-1, N); j++)
          Z[i][j] += r* Y[k][j]
      }
```

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



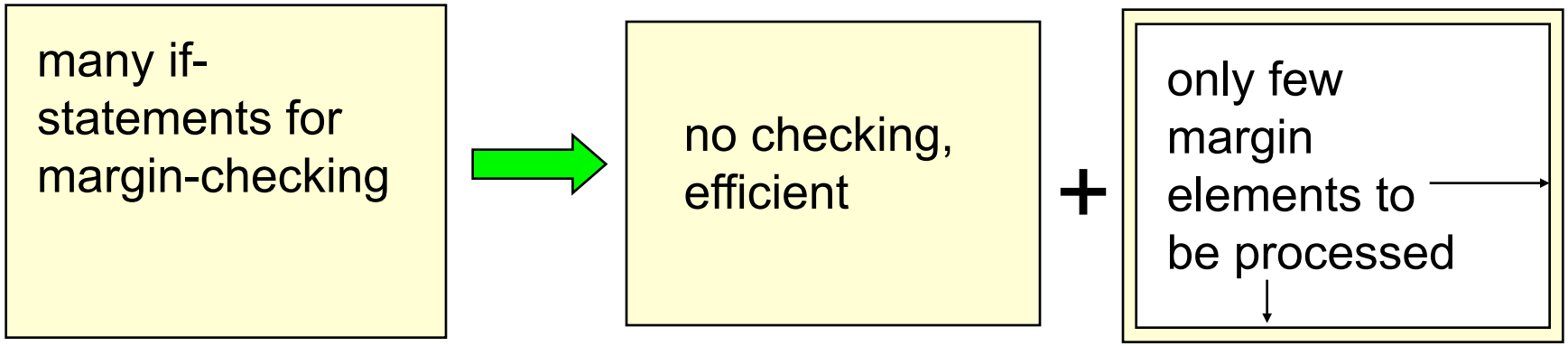
Monica Lam: The Cache Performance and Optimization of Blocked Algorithms, ASPLOS, 1991

High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

Transformation “Loop nest splitting”

Example: Separation of margin handling

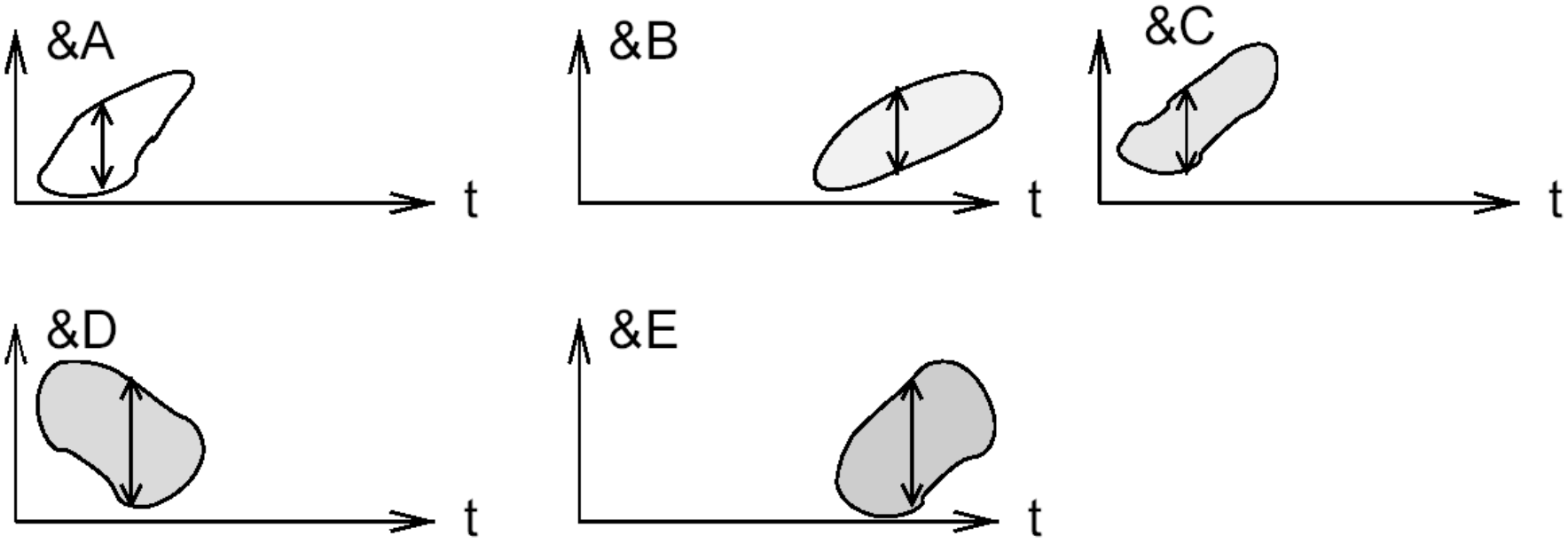


High-level optimizations

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

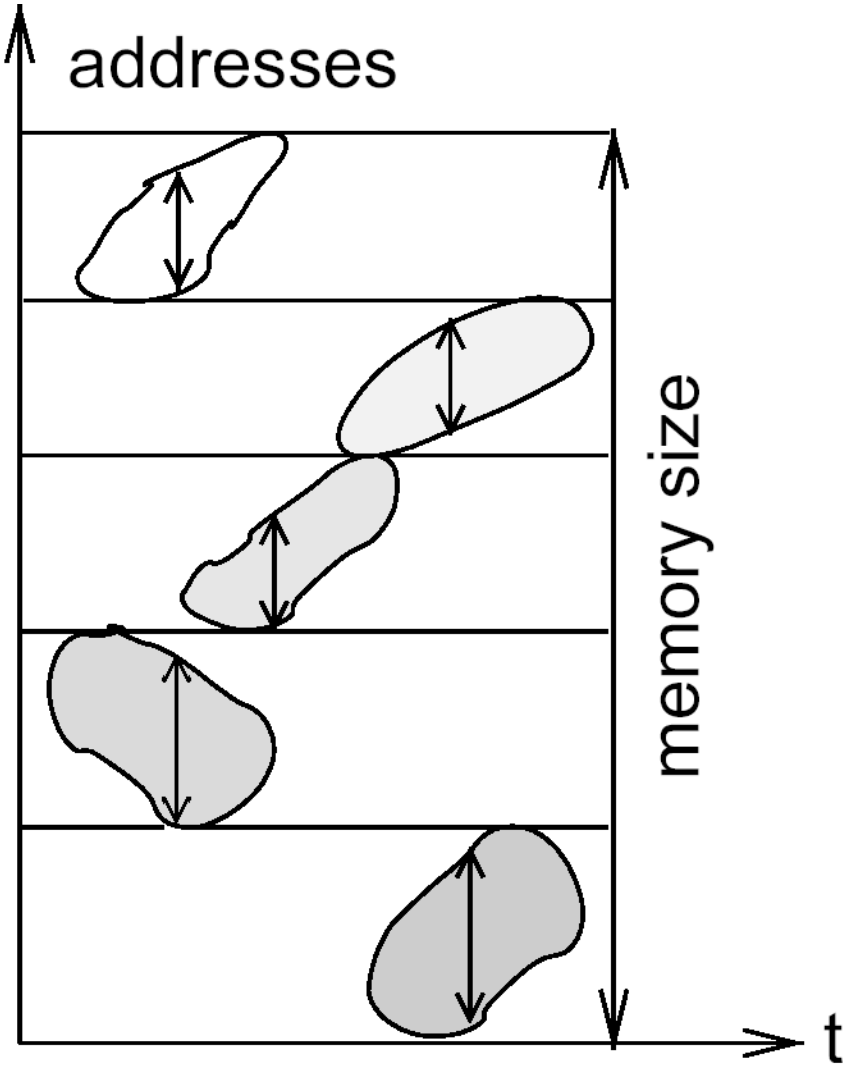
Array folding

Initial arrays



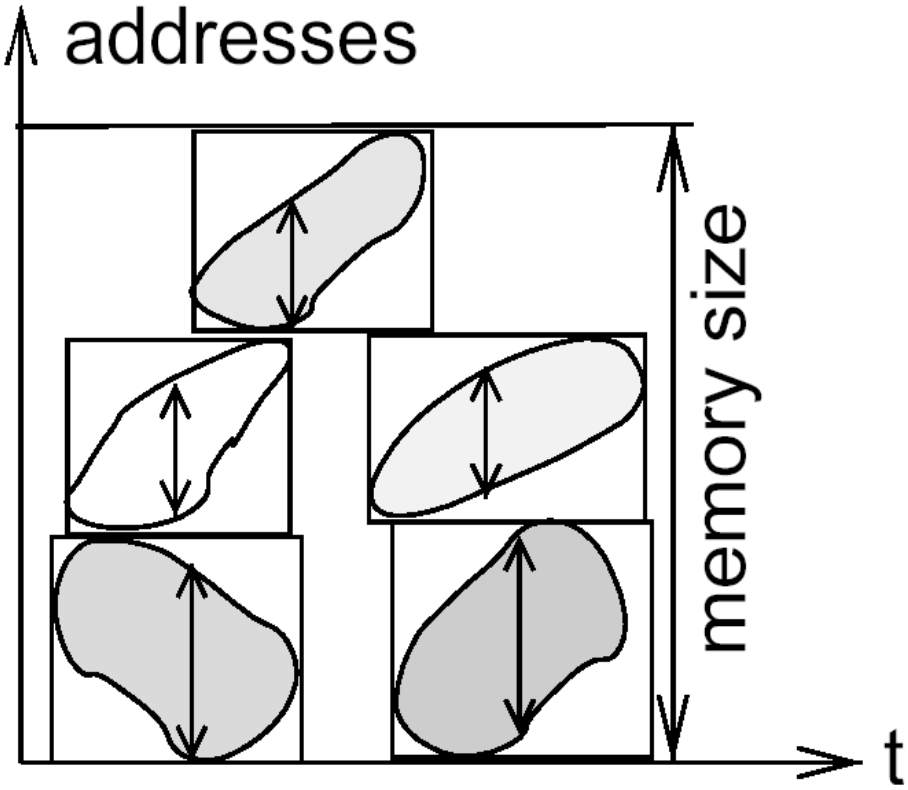
Array folding

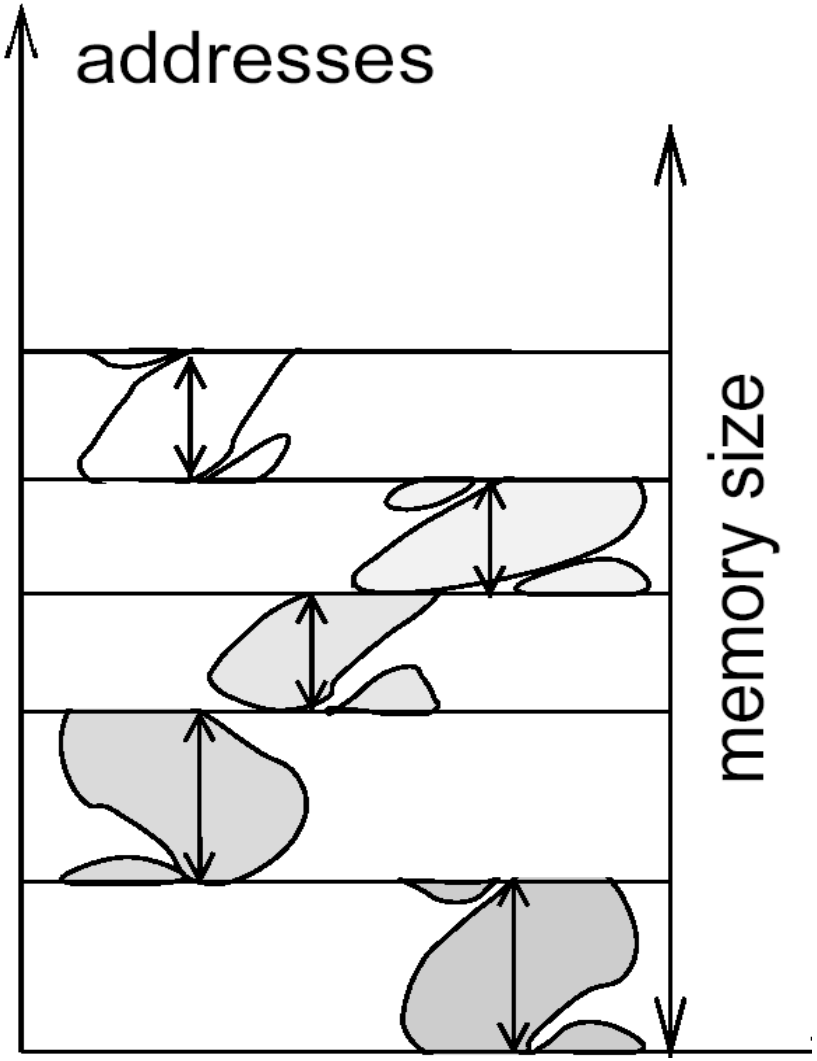
Unfolded arrays



Intra-array folding

Inter-array folding





Summary

- Floating-point to fixed point conversion
  - Range estimation
  - Conversion
  - Analysis of the results
- High-level loop transformations
  - Fusion
  - Unrolling
  - Tiling
  - Loop nest splitting
  - Array folding