Lecture 23: Optimization I

Seyed-Hosein Attarzadeh-Niaki

Based on slides by Peter Marwedel

Review

- Worst-case execution time problem
- Programs as Graphs
- Challenges of Execution Time Analysis
- Current Approaches
- Limitations and Future Directions

Outline

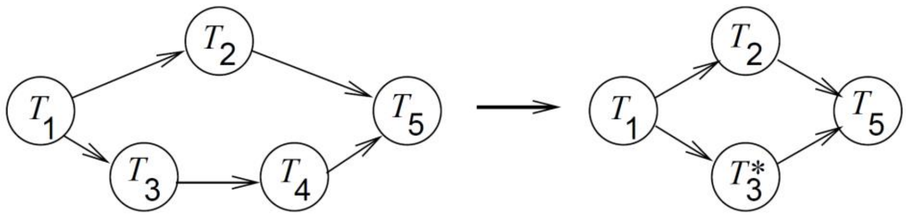
- Task level concurrency management
- High-level optimizations
  - Floating-point to fixed-point conversion
  - Simple loop transformations
  - Loop tiling/blocking
  - Loop splitting
  - Array folding

Task-level concurrency management

- Granularity: size of tasks (e.g. in instructions)
- Readable *specifications* vs. efficient *implementations*
  - possibly require different task structures.

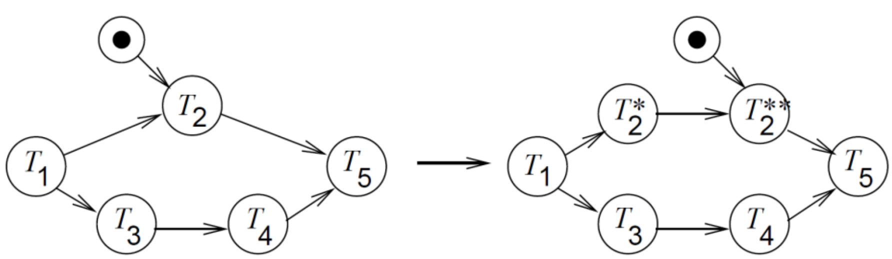
☞ Granularity changes

Merging of tasks



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

Splitting of tasks

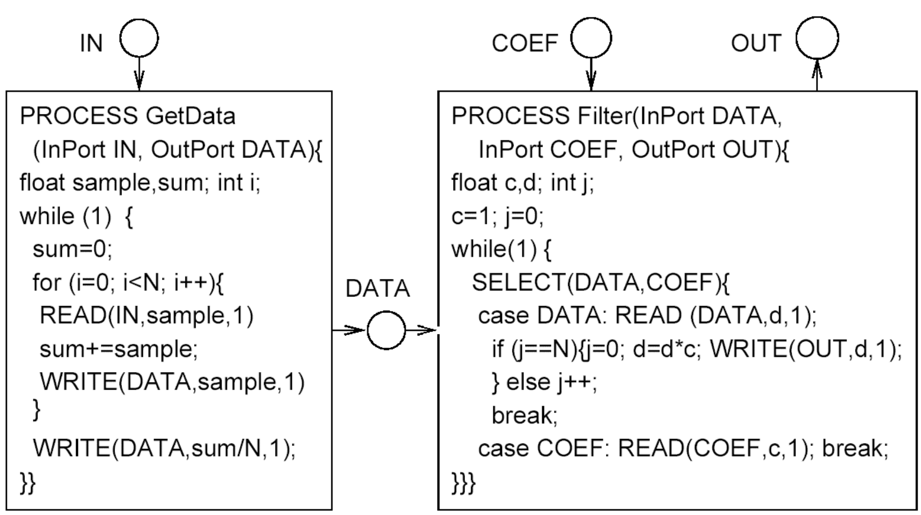


- No blocking of resources while waiting for input,
- more flexibility for scheduling, possibly improved result.

Merging and splitting of tasks

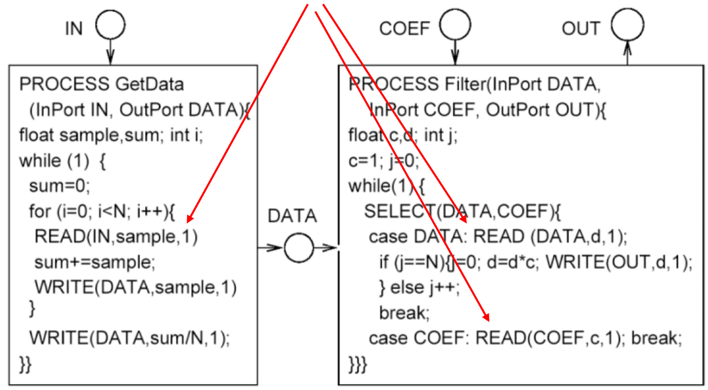
- The most appropriate task graph granularity depends upon the context
  - ☞ merging and splitting may be required.
- Merging and splitting of tasks should be done automatically, depending upon the context.

Automated rewriting of the task system Example (in FlowC language)



Parts of a system that need rewriting

Tasks blocking after they have already started running



Work by Cortadella et al.

- Transform each of the tasks into a special formalism (Petri net),
- Generate one global Petri net from the nets of the tasks,
- Partition global net into “sequences of transitions”
- Generate one task from each such sequence

Result, as published by Cortadella

Reads only at the beginning

Initialization task

IN OUT

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

COEF

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

```
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
}
```

Never true

Always true

Optimized version of Tin

Never true

IN OUT

```
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

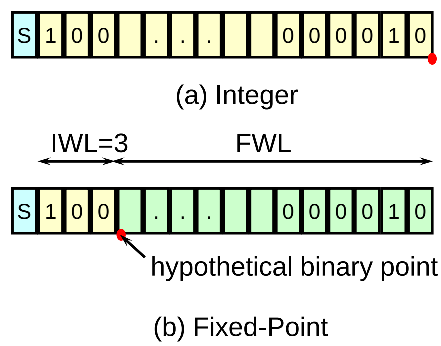
Always true

```
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;
}
```

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

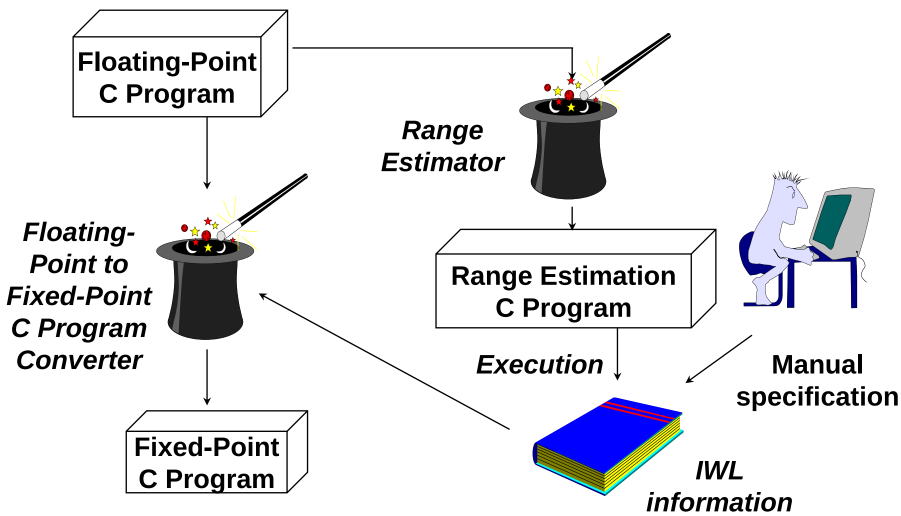
- Lower cost
- Faster
- Lower power consumption
- Sufficient SQNR, *if properly scaled*
- Suitable for portable applications

Cons

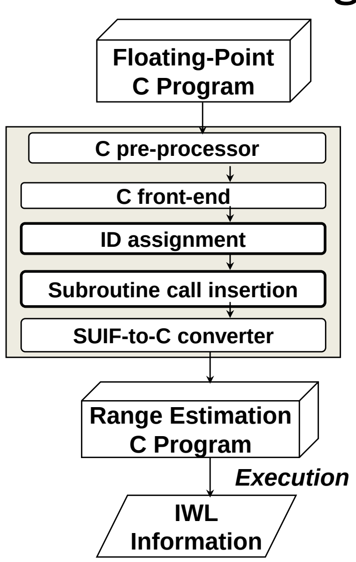
- Decreased dynamic range
- Finite word-length effect, *unless properly scaled*
  - Overflow and excessive quantization noise
- Extra programming effort

© *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



Range Estimator



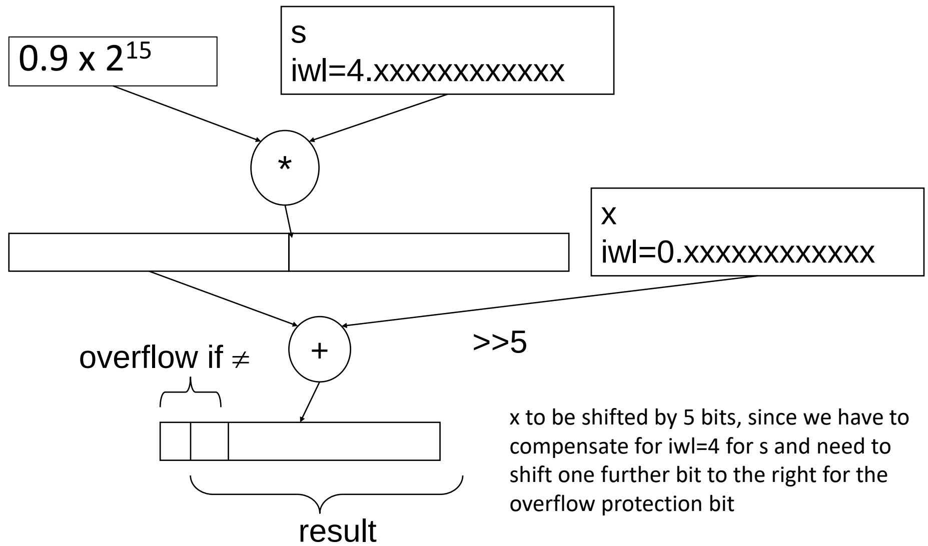
Range Estimation C Program

```
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



x to be shifted by 5 bits, since we have to compensate for iwl=4 for s and need to shift one further bit to the right for the overflow protection bit

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

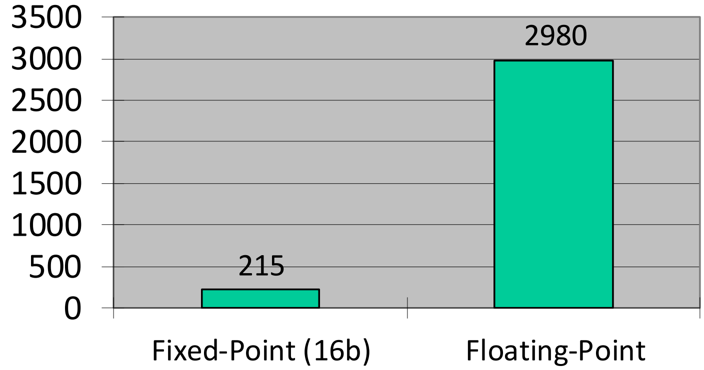
IWL of variable x, y, s are 0, 4, 4

Performance Comparison – Machine Cycles -

Fourth Order IIR Filter

Cycles

TMS 320C50 DSP processor



Embedded Real-Time Systems

© *Ki-Il Kum, et al*

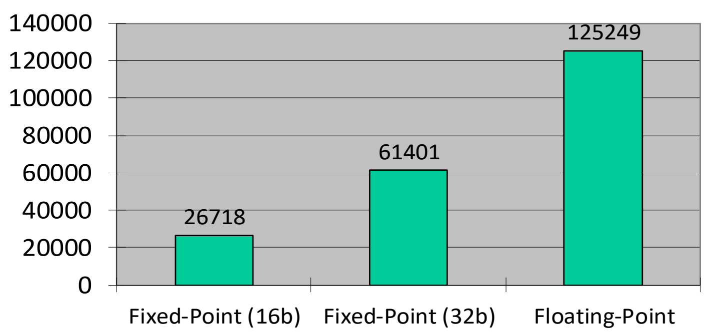
19

Performance Comparison – Machine Cycles -

Cycles

ADPCM

TMS 320C50 DSP processor

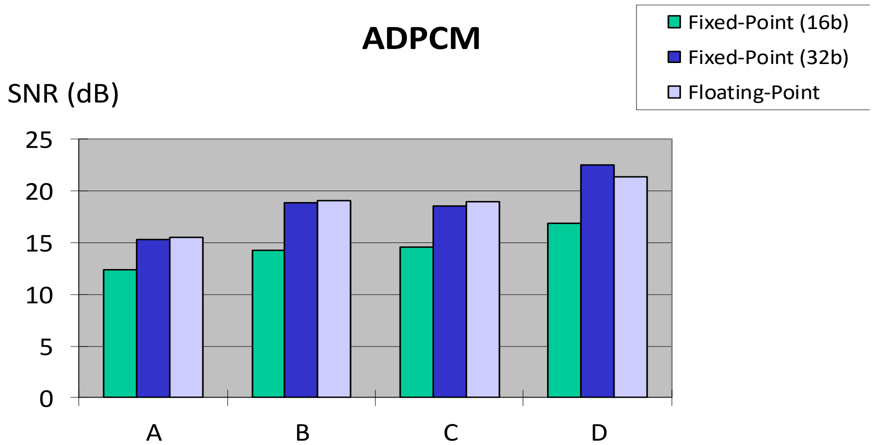


Embedded Real-Time Systems

© *Ki-Il Kum, et al*

20

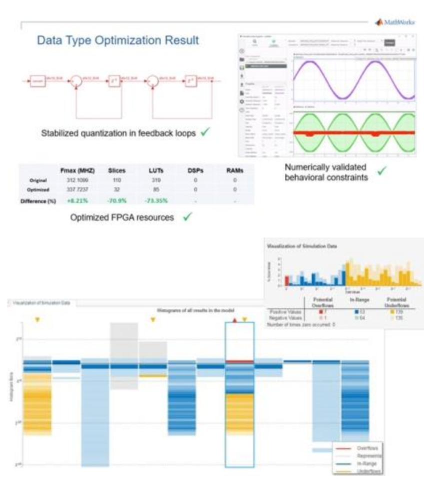
Performance Comparison - SNR -



Matlab Fixed-Point Designer

Provides data types and tools for optimizing and implementing fixed-point and floating-point algorithms on embedded hardware.

- Data Type Exploration
- Automated Data Typing
- Embedded Implementation
- Testing and Debugging



Simple Loop Transformations

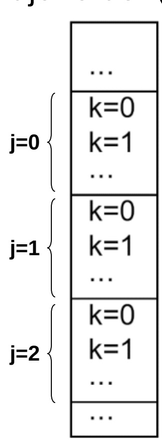
- Loop permutation
- Loop fusion, loop fission
- Loop unrolling

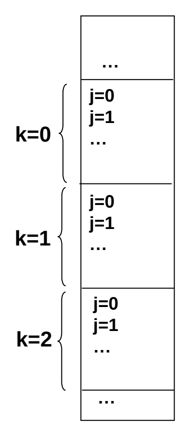
Impact of Memory Allocation on Efficiency

Row major order (C)

Array p[j][k]

Column major order (FORTRAN)





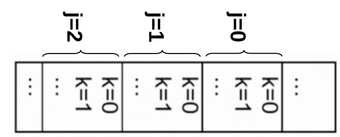
Best performance of innermost loop corresponds to rightmost array index

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] ;}}}}…
start=time(&start);for(z=0;z<iter;z++)computeijk();
  end=time(&end);
  printf("ijk=%16.9f\n",1.0*difftime(end,start));
```

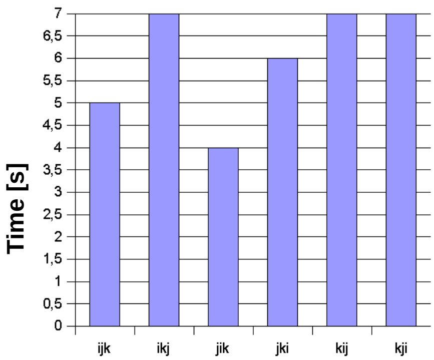
(SUIF interchanges array indexes instead of loops)

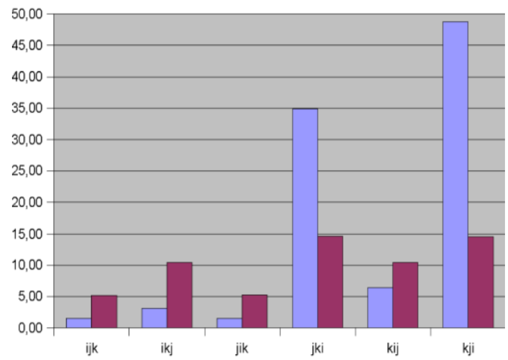
Results: Strong Influence of The Memory Architecture

Dramatic impact of locality

Loop structure: i j k

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]

Program Transformation: "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

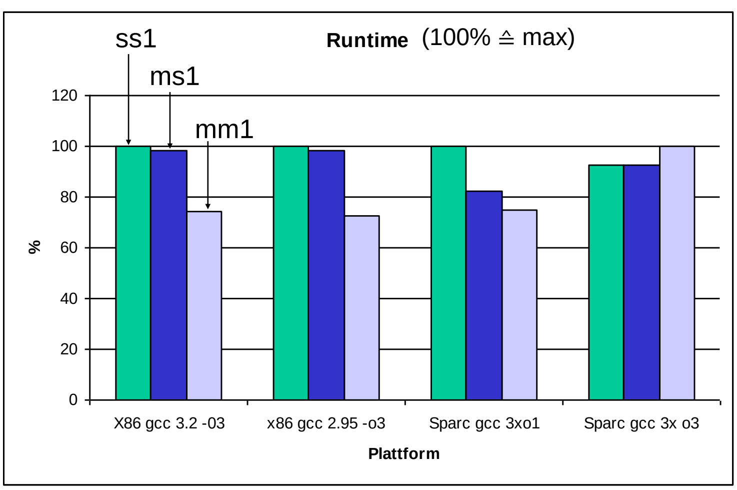
```
#define size 30
#define iter 40000
int a[size][size];
float b[size][size];
```

```
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

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).

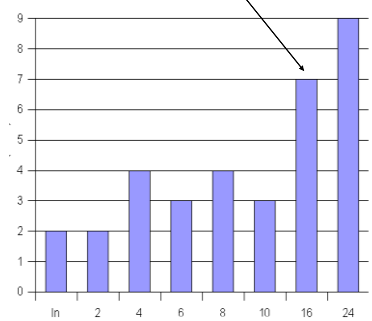
Example: matrixmult

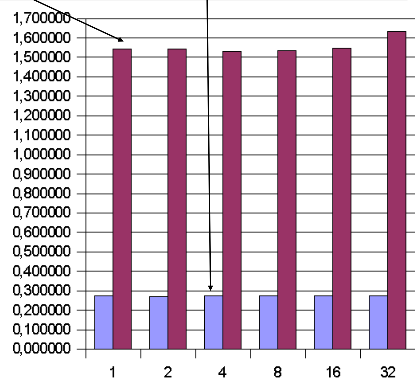
```
#define s 30
#define iter 4000
int
a[s][s],b[s][s],c[s][s];
void compute(){int i,j,k;
 for(i=0;i<s;i++){
  for(j=0;j<s;j++){
   for(k=0;k<s;k++){
    c[i][k]+=
    a[i][j]*b[j][k];
}}}}
```

```
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

Processor	Ti C6xx	Sun SPARC	Intel Pentium





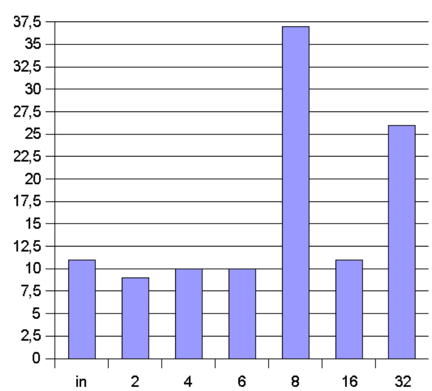
Benefits quite small; penalties may be large

[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]

Loop Tiling/Blocking

- Speed of memories is increasing at a slower rate than that of processors.
- Small memories are faster than large memories -> use memory hierarchy
- Possible “small” memories include caches and scratch-pad memories.
  - Significant *reuse factor* for the information in these memories is required
  - Try to fit the active data into fast memories

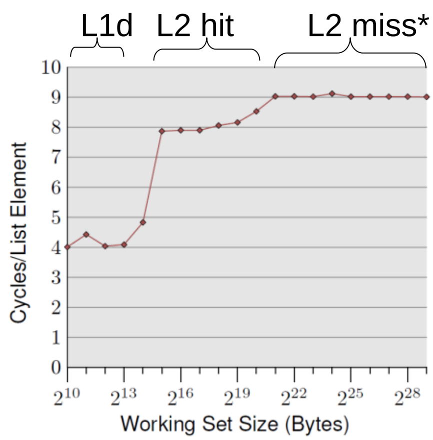
Impact of Caches on Execution Times?

- Execution time for traversal of linked list, stored in an array, each entry comprising NPAD*8 Bytes

- Pentium P4
- 16 kB L1 data cache, 4 cycles/access
- 1 MB L2 cache, 14 cycles/access
- Main memory, 200 cycles/access

U. Drepper: *What every programmer should know about memory**, 2007, http://www.akkadia.org/drepper/cpumemory.pdf; Dank an Prof. Teubner (LS6) für Hinweis auf diese Quelle
* In Anlehnung an das Papier „David Goldberg, *What every programmer should know about floating point arithmetic, ACM Computing Surveys*, 1991 (auch für diesen Kurs benutzt).

Cycles/access as a function of the size of the list



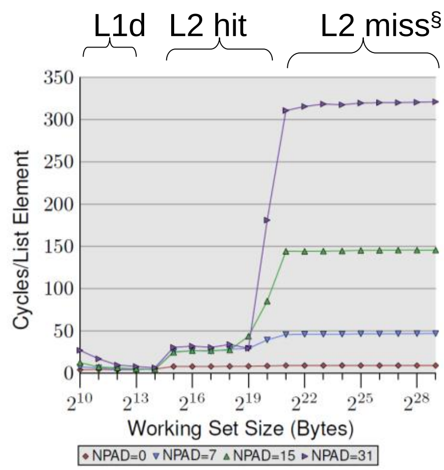


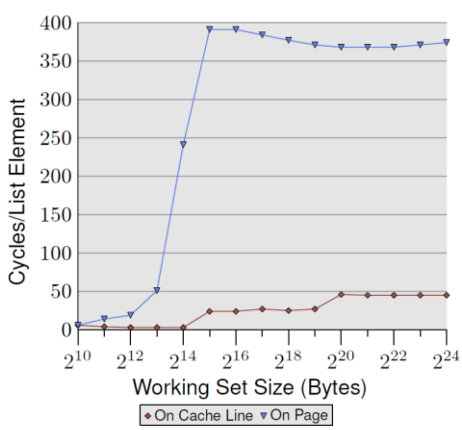
Figure 3.10: Sequential Read Access, NPAD=0

* prefetching succeeds

§ prefetching fails

Impact of TLB misses and larger caches

Elements on different pages; run time increase when exceeding the size of the TLB



Larger caches are shifting the steps to the right

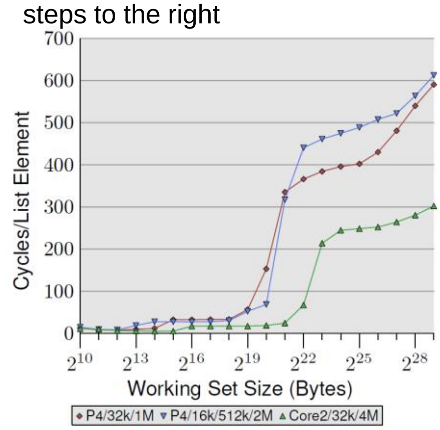
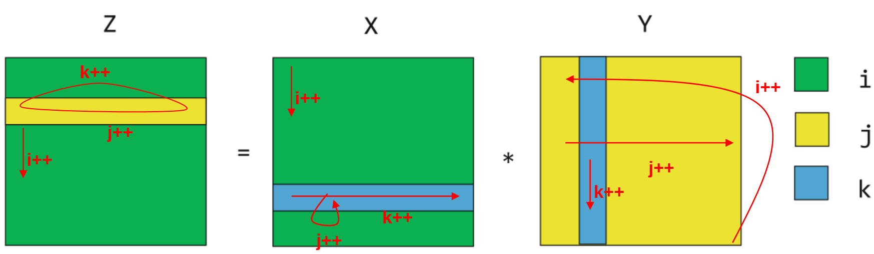


Figure 3.14: Advantage of Larger L2/L3 Caches

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

```
for  (i=0; i<N; i++)
   for(j=0; j<N; k++){
      r=0; /* to be allocated to a register*/
      for  (k=0; k<N; j++)
         r += X[i,k] * Y[k,j]
      Z[i][j]=r;
}
```

% Never reusing information in the cache for Y and Z if N is large or cache is small (2 N^3 references for Z).



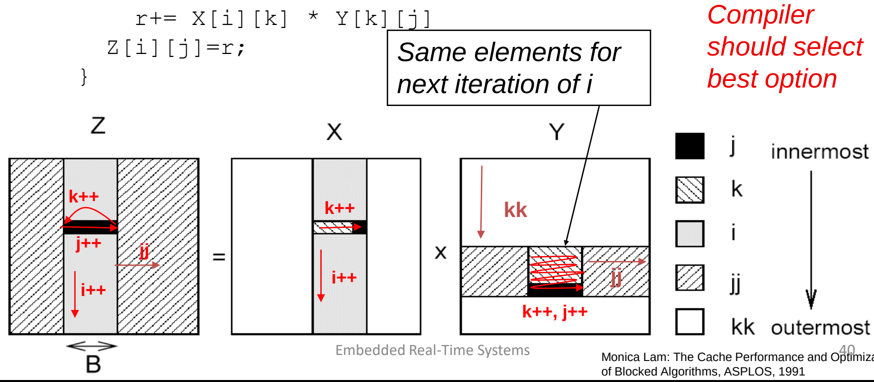
Loop tiling/loop blocking
- tiled version -

```
for  (ii=0; ii<N; ii+=B)
  for  (jj=0; jj<N; jj+=B)
    for  (kk=0; kk<N; kk+=B)
      for  (i=ii; i<min(ii+B-1,N); i++){
        for  (j=jj; j<min(jj+B-1, N); j++)
          r=0; /* to be allocated to a register*/
          for  (k=kk; k<min(kk+B-1, N); k++)
            r+= X[i][k] * Y[k][j]
          Z[i][j]=r;
      }
```

Reuse factor of B-1 for Z, N for Y

O(N^3/(B-1)) accesses to main memory

Compiler should select best option



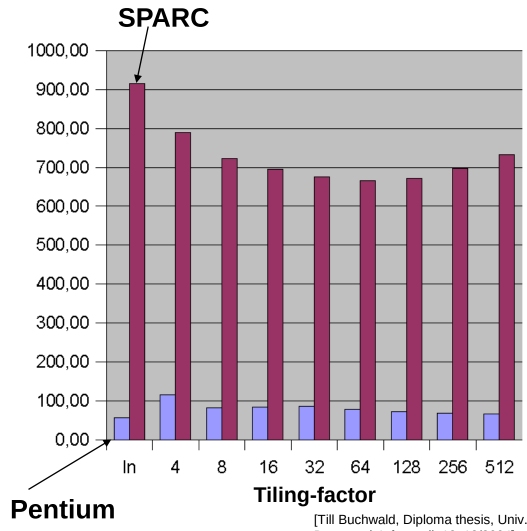
Monica Lam: The Cache Performance and Optimization of Blocked Algorithms, ASPLOS, 1991

Example

In practice, results by Buchwald are disappointing.

One of the few cases where an improvement was achieved:

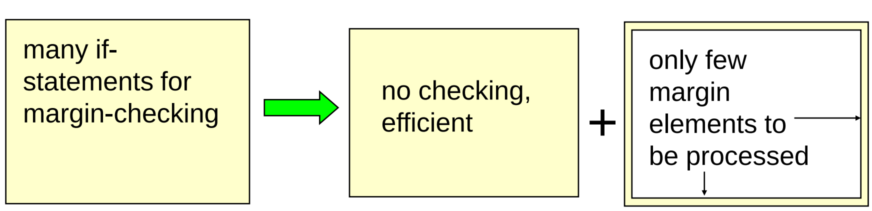
Source: similar to matrix mult.



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

Loop (Nest) Splitting

- Example: Separation of margin handling
  - Regular computation for pixels in image filtering
  - At image margins, neighbors do not exist
  - Straightforward implementation: check boundaries in the inner loop
  - Efficient implementation: Split loops



Loop nest from MPEG-4 full search motion estimation

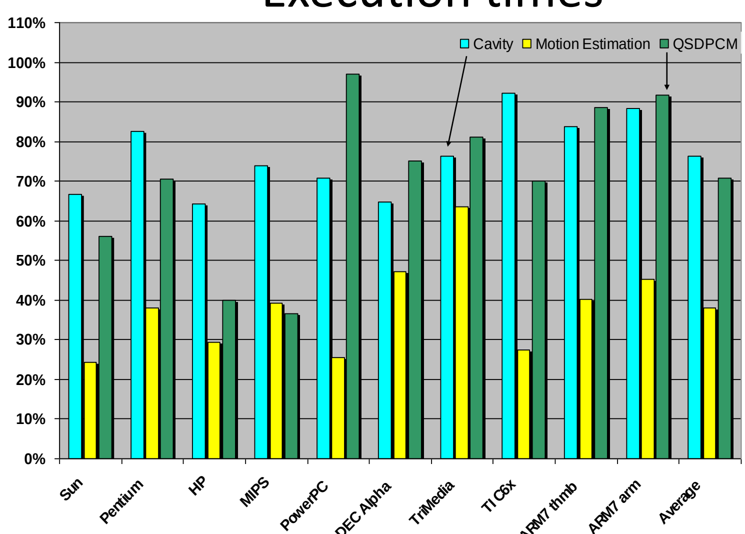
```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3||y3<0||48<y3)
        then_block_1;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

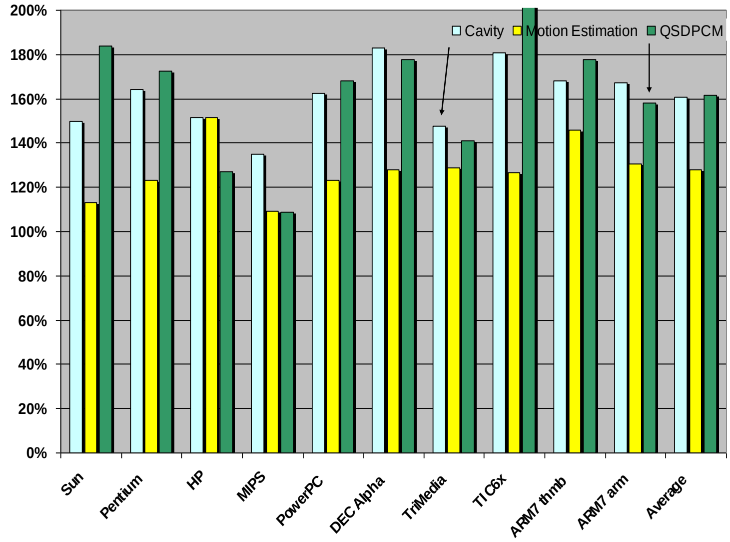
Results for loop nest splitting - Execution times -



Cycle counts can be reduced by up to about 75% (to 25 % of original value).

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

Results for loop nest splitting - Code sizes -



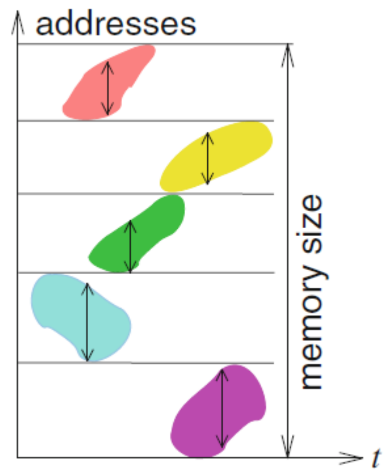
[Falk, 2002]

Array Folding

- Some applications (e.g., multimedia) include large arrays.
- At any particular time only a subset of array elements is needed.
- Maximum number of elements needed: *address reference window*

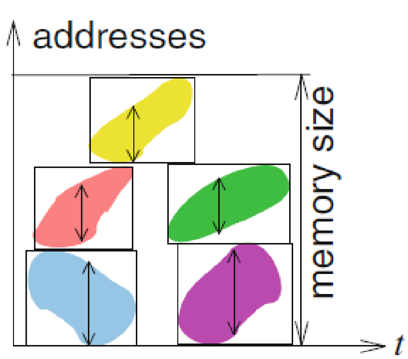
Classical Memory Allocation

- Each array is allocated the maximum of the space it requires during the entire execution time
- Unfolded array

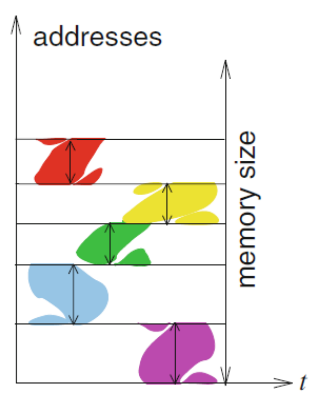


Improved Memory Allocation

Inter-array folding



Intra-array folding

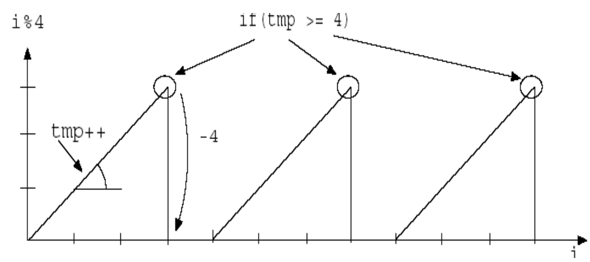


Application

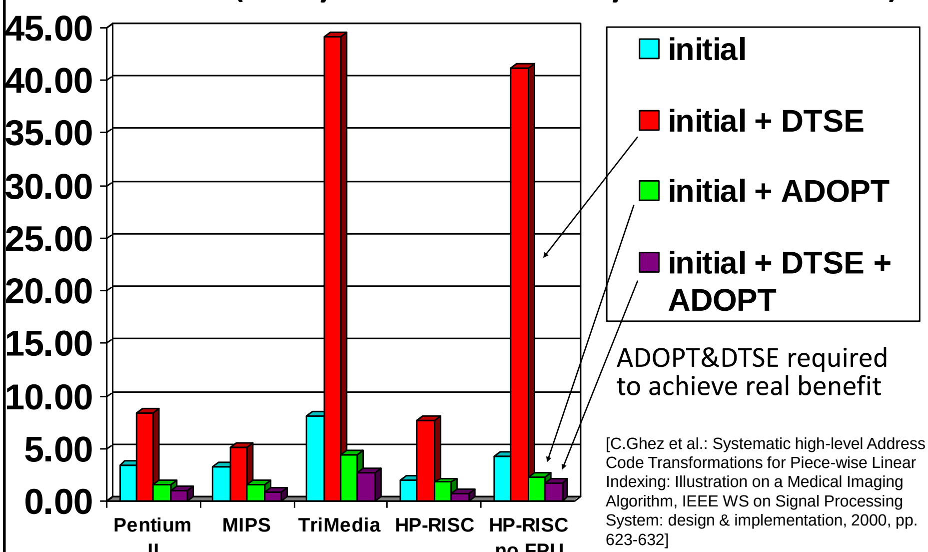
- Array folding is implemented in the DTSE optimization proposed by IMEC. Array folding adds div and mod ops. Optimizations required to remove these costly operations.
- At IMEC, ADOPT address optimizations perform this task. For example, modulo operations are replaced by pointers (indexes) which are 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)



[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]