Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2014 年 01 月 17 日

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.
Structure of this course

2: Specification & Modeling
3: ES-hardware
4: system software (RTOS, middleware, …)

Design repository
6: Application mapping
7: Optimization
5: Evaluation & validation & (energy, cost, performance, …)

Application Knowledge

8: Test

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

Book section 7.1
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 -

```
PROCESS GetData
   (InPort IN, OutPort DATA){
      float sample, sum; int i;
      while (1) {
         sum=0;
         for (i=0; i<N; i++){
            sum+=sample;
            WRITE(DATA, sample, 1)
         }
         WRITE(DATA, sum/N, 1);
      }
   }
```

```
PROCESS Filter(InPort DATA, InPort COEF, OutPort OUT){
   float c, d; int j;
   c=1; j=0;
   while(1) {
      SELECT(DATA, COEF){
         case DATA: READ (DATA, d, 1);
            if (j==N) {j=0; d=d*c; WRITE(OUT, d, 1);}
         } else j++;
         break;
         case COEF: READ(COEFF, c, 1); break;
      }
   }
```
Attributes of a system that needs rewriting

Tasks blocking after they have already started running

```
PROCESS GetData(InPort IN, OutPort DATA){
    float sample,sum; int i;
    while (1) {
        sum=0;
        for (i=0; i<N; i++){
            READ(IN,sample,1)
            sum+=sample;
            WRITE(DATA,sample,1)
        }
        WRITE(DATA,sum/N,1);
    }
}

PROCESS Filter(InPort DATA, InPort COEF, OutPort OUT){
    float c,d; int j;
    c=1; j=0;
    while(1) {
        SELECT(DATA,COEF){
            case DATA: READ (DATA,d,1);
            if (j==N) c=0; d=d*c; WRITE(OUT,d,1);
            else j++;
            break;
            case COEF: READ(COEF,c,1); break;
        }
    }
```
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 transitions”
4. Generate one task from each such sequence

Mature, commercial approach not yet available
Result, as published by Cortadella

Initialization task

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

```
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
}
```
Optimized version of Tin

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
High-level software transformations

Peter Marwedel
TU Dortmund
Informatik 12
Germany

These slides use Microsoft clip arts. Microsoft copyright restrictions apply.
High-level optimizations

Book section 7.2

- 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**

```
S 1 0 0 . . . 0 0 0 0 1 0
```

(a) Integer

```
S 1 0 0 . . . 0 0 0 0 1 0
```

hypothesical binary point

(b) Fixed-Point

© Ki-II 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-II 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

Floating-Point C Program

Floating-Point to Fixed-Point C Program Converter

Range Estimator

Range Estimation C Program

Execution

Manual specification

Fixed-Point C Program

IWL information
Range Estimator

Range Estimation C Program

```c
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

\[ 0.9 \times 2^{15} \]

\[ \text{s } \]
\[ \text{iwl}=4.\text{xxxxxxxxxxxxxxx} \]

\[ \ast \]

\[ \text{x } \]
\[ \text{iwl}=0.\text{xxxxxxxxxxxxxxx} \]

\[ \text{overflow if } \neq \]
\[ \text{result} \]

\[ \gg 5 \]
Floating-Point to Fixed-Point Program Converter

Fixed-Point C Program

```c
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 2\(^{nd}\) sign bit
- opt. overflow check
Performance Comparison - Machine Cycles -

Fourth Order IIR Filter

Cycles

Fixed-Point (16b)  Floating-Point

0  1000  2000  3000  4000

© Ki-II Kum, et al
Performance Comparison - Machine Cycles -

Cycles

ADPCM

Fixed-Point (16b): 26718
Fixed-Point (32b): 61401
Floating-Point: 125249
Performance Comparison - SNR -

ADPCM

SNR (dB)

Fixed-Point (16b)
Fixed-Point (32b)
Floating-Point

A B C D

© Ki-Il Kum, et al
High-level optimizations

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

Book section 7.2
Impact of memory allocation on efficiency

Array \( p[j][k] \)

Row major order (C)

\[
\begin{array}{cccc}
  & k=0 & k=1 & \ldots \\
  j=0 & \ldots & \ldots & \ldots \\
  j=1 & k=0 & k=1 & \ldots \\
  j=2 & k=0 & k=1 & \ldots \\
\end{array}
\]

Column major order (FORTRAN)

\[
\begin{array}{cccc}
  & k=0 & j=0 & j=1 & \ldots \\
  & \ldots & j=0 & j=1 & \ldots \\
  & \ldots & j=0 & j=1 & \ldots \\
  & \ldots & j=0 & j=1 & \ldots \\
\end{array}
\]
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”

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

Loop structure: i j k

<table>
<thead>
<tr>
<th>Processor</th>
<th>Ti C6xx</th>
<th>Sun SPARC</th>
<th>Intel Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduction to [%]</td>
<td>~ 57%</td>
<td>35%</td>
<td>3.2 %</td>
</tr>
</tbody>
</table>

Dramatic impact of locality

Not always the same impact ..

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

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

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

```c
# 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 (i=0;i<size;i++){
        for (j=0;j<size;j++){
            b[i][j] -= 13;
        }
    }
}

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

void mm1() {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;
        }
    }
}
```

Results: simple loops

Merged loops superior; except Sparc with –o3
Loop unrolling

\begin{align*}
\text{for } (j=0; j<=n; j++) & \quad \text{for } (j=0; j<=n; j+=2) \\
p[j]= \ldots & \quad \{p[j]= \ldots ; p[j+1]= \ldots\}
\end{align*}

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

```c
#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 < s; i++) {
        for (j = 0; j < s; 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];
            }
        }
    }
    return;
}
```
Results

<table>
<thead>
<tr>
<th>Processor</th>
<th>Ti C6xx</th>
<th>Sun SPARC</th>
<th>Intel Pentium</th>
</tr>
</thead>
</table>

Benefits quite small; penalties may be large

Results: benefits for loop dependences

<table>
<thead>
<tr>
<th>Processor</th>
<th>Ti C6xx</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduction to [%]</td>
<td></td>
</tr>
</tbody>
</table>

```c
#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;

High-level optimizations

Book section 7.2

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


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 -

```plaintext
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]

Same elements for next iteration of i

Reuse factor of B for Z, N for Y
O(N³/B) accesses to main memory

Compiler should select best option

In practice, results by Buchwald are disappointing. One of the few cases where an improvement was achieved: Source: similar to matrix mult.

[Graph showing performance comparison between SPARC and Pentium processors for different tiling factors.]

High-level optimizations

Book section 7.2

- 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

many if-statements for margin-checking  \rightarrow  no checking, efficient  \rightarrow  only few margin elements to be processed
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;
            }}}}}}

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]
High-level optimizations

Book section 7.2

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

Initial arrays

&A

&B

&C

&D

&E
Array folding

Unfolded arrays

addresses

memory size

t
Inter-array folding

Intra-array folding

addresses

memory size

t

addresses

memory size
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.

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

int tmp=0;
for(i=0; i<20; i++)
    if(tmp >= 4)
        tmp -=4;
    B[tmp];
    tmp ++;
```
Results (Mcycles for cavity benchmark)

- Pentium II
- MIPS
- TriMedia
- HP-RISC
- HP-RISC no FPU

ADOPT&DTSE required to achieve real benefit

Summary

- Task concurrency management
  - Re-partitioning of computations into tasks
- 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