Optimizations

Peter Marwedel
TU Dortmund, Informatik 12
Germany
Structure of this course

1. Application Knowledge

2. Specifications

3. Embedded System HW


5. Application mapping: scheduling, HW/SW-Partitioning

6. Evaluation

7. Optimization of Embedded Systems

8. Testing

New clustering
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 -

**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) { j = 0; d = d * c; WRITE(OUT, d, 1); }
      else j++;
      break;
      case COEF: READ(COEF, 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){j=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 transition”
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 == i - 1) {j = 0; d = d * c; WRITE(OUT, d, 1);
} else j++;
L0: if (i < N) return;
DATA = sum/N; d = DATA;
if (j == i - 1) {j = 0; d = d * c; WRITE(OUT, d, 1);
} else j++;
sum = 0; i = 0; goto L0

Never true

Always true

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

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

(b) Fixed-Point

© Ki-Il Kum, et al
Assignment and Addition/Subtraction

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

$x$

$x >> 1$

$y$

Let result $= x + y$: equalizing each IWL

$x$

$x >> 1$

$+ y$

result

© Ki-Il Kum, et al
Multiplication

Assume result = x * y, with
-x (IWL=2) and
-y (IWL=3)
-> result (IWL=2+3)
Development Procedure

Floating-Point C Program

Range Estimator

Floating-Point to Fixed-Point C Program Converter

Fixed-Point C Program

Range Estimation C Program

Execution

Manual specification

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 x 2^{15}

s
iwl=4.xxxxxxxxxxxx

x
iwl=0.xxxxxxxxxxxx

overflow if \neq result

result

\Rightarrow 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\textsuperscript{nd} sign bit
  - opt. overflow check

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

Fourth Order IIR Filter

Cycles

4000
3000
2000
1000
0

Fixed-Point (16b)  Floating-Point

215  2980

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

Cycles

Fixed-Point (16b) 26718
Fixed-Point (32b) 61401
Floating-Point 125249

© Ki-II Kum, et al
Performance Comparison
- SNR -

ADPCM

SNR (dB)

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

© Ki-II Kum, et al
High-level software transformations

Peter Marwedel,
TU Dortmund, Informatik 12
Germany

2007/12/24
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):

\[
\begin{align*}
\text{for } & (k=0; k<=m; k++) & \text{for } & (j=0; j<=n; j++) \\
\text{for } & (j=0; j<=n; j++) & \text{for } & (k=0; k<=m; k++) \\
p[j][k] & = ... & p[j][k] & = ...
\end{align*}
\]

Same behavior for homogenous memory access, but:

- Poor cache behavior
- Good cache behavior

.memory architecture dependent optimization
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

Dramatic impact of locality

<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>

Not always the same impact ..
Transformations
“Loop fusion” (merging), “loop fission”

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

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;
        }
    }
}

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

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

```c
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.
Extrem 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 < 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

Results: benefits for loop dependences

`#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;

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

```c
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(N³/B) accesses to main memory

Compiler should select best option

Example

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

Tiling-factor

[Source: similar to matrix mult.]

[Sparc]

Pentium

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