Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund
Informatik 12
Germany

2011年 01 月 12 日

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

Numbers denote sequence of chapters
Compilers for embedded systems: Why are compilers an issue?

- Many reports about low efficiency of standard compilers
  - Special features of embedded processors have to be exploited.
  - High levels of optimization more important than compilation speed.
  - Compilers can help to reduce the energy consumption.
  - Compilers could help to meet real-time constraints.

- Less legacy problems than for PCs.
  - There is a large variety of instruction sets.
  - Design space exploration for optimized processors makes sense
Energy-aware compilation (1): Optimization for low-energy the same as for high performance?

No!
- High-performance if available memory bandwidth fully used; low-energy consumption if memories are at stand-by mode
- Reduced energy if more values are kept in registers

```c
int a[1000];
c = a;
for (i = 1; i < 100; i++) {
    b += *c;
    b += *(c+7);
    c += 1;
}
```

LDR r3, [r2, #0]
ADD r3,r0,r3
MOV r0,#28
LDR r0, [r2, r0]
ADD r0,r3,r0
ADD r2,r2,#4
ADD r1,r1,#1
CMP r1,#100
BLT LL3

2096 cycles 19.92 µJ

ADD r3,r0,r2
MOV r0,#28
MOV r2,r12
MOV r12,r11
MOV r11,rr10
MOV r0,r9
MOV r9,r8
MOV r8,r1
LDR r1, [r4, r0]
ADD r0,r3,r1
ADD r4,r4,#4
ADD r5,r5,#1
CMP r5,#100
BLT LL3

2231 cycles 16.47 µJ

No!
- High-performance if available memory bandwidth fully used; low-energy consumption if memories are at stand-by mode
- Reduced energy if more values are kept in registers
Energy-aware compilation (2)

- Operator strength reduction: e.g. replace * by + and <<
- Minimize the bitwidth of loads and stores
- Standard compiler optimizations with energy as a cost function

E.g.: Register pipelining:

for i := 0 to 10 do
  C := 2 * a[i] + a[i-1];

R2 := a[0];
for i := 1 to 10 do
  begin
    R1 := a[i];
    C := 2 * R1 + R2;
    R2 := R1;
  end;

Exploitation of the memory hierarchy
Energy-aware compilation (3)

- Energy-aware scheduling: the order of the instructions can be changed as long as the meaning does not change. Goal: reduction of the number of signal transitions. Popular (can be done as a post-pass optimization with no change to the compiler).

- Energy-aware instruction selection: among valid instruction sequences, select those minimizing energy consumption.

- Exploitation of the memory hierarchy: huge difference between the energy consumption of small and large memories.
3 key problems for future memory systems

1. (Average) Speed
2. Energy/Power
3. Predictability/WCET
Hierarchical memories using scratch pad memories (SPM)

Hierarchy

main

SPM

processor

Example

ARM7TDMI cores, well-known for low power consumption

Address space

0

scratch pad memory

FFF..

no tag memory
Very limited support in ARMcc-based tool flows

1. **Use pragma in C-source to allocate to specific section:**
   For example:
   ```c
   #pragma arm section rwdata = "foo", rodata = "bar"
   int x2 = 5; // in foo (data part of region)
   int const z2[3] = {1,2,3}; // in bar
   ```

2. **Input scatter loading file to linker for allocating section to specific address range**

   ![Diagram](http://www.arm.com/documentation/Software_Development_Tools/index.html)
Which memory object (array, loop, etc.) to be stored in SPM?

Non-overlaying (“Static”) allocation:

Gain $g_k$ and size $s_k$ for each object $k$. Maximise gain $G = \Sigma g_k$, respecting size of SPM $SSP \geq \Sigma s_k$.

Solution: knapsack algorithm.

Overlaying (“dynamic”) allocation:

Moving objects back and forth
ILP representation
- migrating functions and variables-

Symbols:

\( S(var_k) \) = size of variable \( k \)

\( n(var_k) \) = number of accesses to variable \( k \)

\( e(var_k) \) = energy *saved* per variable access, if \( var_k \) is migrated

\( E(var_k) \) = energy *saved* if variable \( var_k \) is migrated (= \( e(var_k) \cdot n(var_k) \))

\( x(var_k) \) = decision variable, =1 if variable \( k \) is migrated to SPM, =0 otherwise

\( K \) = set of variables; similar for functions \( I \)

**Integer programming formulation:**

Maximize \( \sum_{k \in K} x(var_k) \cdot E(var_k) + \sum_{i \in I} x(F_i) \cdot E(F_i) \)

Subject to the constraint

\( \sum_{k \in K} S(var_k) \cdot x(var_k) + \sum_{i \in I} S(F_i) \cdot x(F_i) \leq SSP \)
Reduction in energy and average run-time

Multi_sort (mix of sort algorithms)

Feasible with standard compiler & postpass optimization

Numbers will change with technology, algorithms remain unchanged.
Using these ideas with an gcc-based tool flow

Source is split into 2 different files by specially developed memory optimizer tool *.

* Built with tool design suite ICD-C available from ICD (see www.icd.de/es)
Allocation of basic blocks

Fine-grained granularity smoothens dependency on the size of the scratch pad.

Requires additional jump instructions to return to "main" memory.
Allocation of basic blocks, sets of adjacent basic blocks and the stack

Requires generation of additional jumps (special compiler)
Savings for memory system energy alone

![Graph showing energy savings for different memory systems](graph.png)

Combined model for memories

- Multi_Sort
- Encodecombined
- Fast_idct
- FFT_Viva
- ref_idct
Scratch-pad/tightly coupled memory based predictability

Pre run-time scheduling is often the only practical means of providing predictability in a complex system. [Xu, Parnas]

- Time-triggered, statically scheduled operating systems
- Let’s do the same for the memory system
- Are SPMs really more timing predictable?
- Analysis using the aiT timing analyzer
Architectures considered

ARM7TDMI with 3 different memory architectures:

1. **Main memory**
   - LDR-cycles: (CPU,IF,DF)=(3,2,2)
   - STR-cycles: (2,2,2)
   - * = (1,2,0)

2. **Main memory + unified cache**
   - LDR-cycles: (CPU,IF,DF)=(3,12,6)
   - STR-cycles: (2,12,3)
   - * = (1,12,0)

3. **Main memory + scratch pad**
   - LDR-cycles: (CPU,IF,DF)=(3,0,2)
   - STR-cycles: (2,0,0)
   - * = (1,0,0)
Results for G.721

Using Scratchpad:

Using Unified Cache:

References:
- Wehmeyer, Marwedel: Influence of Onchip Scratchpad Memories on WCET: 4th Intl Workshop on worst-case execution time (WCET) analysis, Catania, Sicily, Italy, June 29, 2004
- Second paper on SP/Cache and WCET at DATE, March 2005
Multiple scratch pads

Small is beautiful:
One small SPM is beautiful (😊).
May be, several smaller SPMs are even more beautiful?

addresses

0

- scratch pad 0, 256 entries
- scratch pad 1, 2 k entries
- scratch pad 2, 16 k entries
- background memory
Optimization for multiple scratch pads

Minimize \[ C = \sum_{j} e_j \cdot \sum_{i} x_{j,i} \cdot n_i \]

With \( e_j \): energy per access to memory \( j \),
and \( x_{j,i} = 1 \) if object \( i \) is mapped to memory \( j \), \( = 0 \) otherwise,
and \( n_i \): number of accesses to memory object \( i \),
subject to the constraints:

\[ \forall j : \sum_{i} x_{j,i} \cdot S_i \leq SSP_j \]

\[ \forall i : \sum_{j} x_{j,i} = 1 \]

With \( S_i \): size of memory object \( i \),
\( SSP_j \): size of memory \( j \).
Considered partitions

Example of considered memory partitions for a total capacity of 4096 bytes

<table>
<thead>
<tr>
<th># of partitions</th>
<th>number of partitions of size:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4k</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Results for parts of GSM coder/decoder

A key advantage of partitioned scratchpads for multiple applications is their ability to adapt to the size of the current working set.

energy savings in %

Total scratchpad capacity [bytes]

„Working set“
Dynamic replacement within scratch pad

- Effectively results in a kind of *compiler-controlled segmentation/paging* for SPM
- Address assignment within SPM required (paging or segmentation-like)

Dynamic replacement of data within scratch pad: based on liveness analysis

MO = \{A, T1, T2, T3, T4\}
SP Size = |A| = |T1| \ldots = |T4|

Solution: \(A \Rightarrow SP \& T3 \Rightarrow SP\)

SPILL_STORE(A);
SPILL_LOAD(T3);

SPILL_LOAD(A);
Dynamic replacement within SPM
Edge detection relative to static allocation

![Bar chart showing Processor Energy, Memory Energy, Total Energy, and Execution Time versus Scratchpad Size (Bytes)]
Hardware-support for block-copying

The DMA unit was modeled in VHDL, simulated, synthesized. Unit only makes up 4% of the processor chip.
The unit can be put to sleep when it is unused.

Code size reductions of up to 23% for a 256 byte SPM were determined using the DMA unit instead of the overlaying allocation that uses processor instructions for copying.

References to large arrays (1)
- Regular accesses -

for (i=0; i<n; i++)
for (j=0; j<n; j++)
for (k=0; k<n; k++)
\[ U[i][j] = U[i][j] + V[i][k] \times W[k][j] \]

Tiling

for (it=0; it<n; it=it+Sb)
{read_tile V[it:it+Sb-1, 1:n]
for (jt=0; jt<n; jt=jt+Sb)
{read_tile U[it:it+Sb-1, jt:jt+Sb-1];
read_tile W[1:n, jt:jt+Sb-1];
write_tile U[it:it+Sb-1, jt:jt+Sb-1];
}}

References to large arrays
- Irregular accesses -

for each loop nest \( L \) in program \( P \) { 
  apply loop tiling to \( L \) based on the access patterns of regular array references;
  for each assignment to index array \( X \)
    update the block minimum and maximum values of \( X \);
  compute the set of array elements that are irregularly referenced in the current inter-tile iteration;
  compare the memory access costs for using and not using SPM;
  if (using SPM is beneficial)
    execute the intra-tile loop iterations by using the SPM
  else
    execute the intra-tile loop iterations by not using the SPM
}

[G. Chen, O. Ozturk, M. Kandemir, M. Karakoy: Dynamic Scratch-Pad Memory Management for Irregular Array Access Patterns, DATE, 2006]
Results for irregular approach

![Bar chart showing normalized cycles for different cache approaches: Dynamic, Static, Hybrid. The bars are labeled with benchmarks such as 'dither', 'power-x', 'ur-direct', 'med 3.0', 'mesh', 'trio-enfr', 'terpa 1.1'. The chart compares the performance across these benchmarks highlighting the differences between the cache approaches.]
Hierarchical memories: Memory hierarchy layer assignment (MHLA) (IMEC)

\[ n \text{ layers with "partitions" consisting of modules} \]

Partition 1.1
- SPM-module 1.1.1
- SPM-module 1.1.2

Partition 1.2
- Cache-module 1.2.1
- Cache-module 1.2.2

Partition 2.1
Partition 2.2

Processor

Memory hierarchy layer assignment (MHLA)
- Copy candidates -

int A[250]
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        for (k=0; k<10; k++)
            for (l=0; l<10; l++)
                f(A'[j*10+l])
size=100; reads(A)=10000

int A[250]
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        for (k=0; k<10; k++)
            for (l=0; l<10; l++)
                f(A'[j*10+1])
size=0; reads(A)=10000

int A[250]
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        for (k=0; k<10; k++)
            for (l=0; l<10; l++)
                f(A'[j*10+1])
size=10; reads(A)=1000

int A[250]
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        for (k=0; k<10; k++)
            for (l=0; l<10; l++)
                f(A'[j*10+1])
size=10; reads(A)=1000

int A[250]
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        for (k=0; k<10; k++)
            for (l=0; l<10; l++)
                f(A'[j*10+1])
size=0; reads(A)=10000

int A[250]
A'[0..99]=A[0..99];
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        for (k=0; k<10; k++)
            for (l=0; l<10; l++)
                f(A'[l])
size=10; reads(A)=1000

int A[250]
A'[0..99]=A[0..99];
for (i=0; i<10; i++)
    for (j=0; j<10; j++)
        for (k=0; k<10; k++)
            for (l=0; l<10; l++)
                f(A'[l])
size=0; reads(A)=10000

reads(A)

- 32 -
Memory hierarchy layer assignment (MHLA) - Goal -

**Goal**: For each variable: find permanent layer, partition and module & select copy candidates such that energy is minimized.

Conflicts between variables

[Image showing graphs for variables A[] and B[] with miss rates vs. copy size, and conflicting layer size values.

Memory hierarchy layer assignment (MHLA)

- Approach -

Approach:
- start with initial variable allocation
- incrementally improve initial solution such that total energy is minimized.

Platform

Current assignment

NOT assigned copy candidates  assigned copy candidates

A' 250 A 1250 L3
100
1000
10000

A" 1000 11000

Next assignment

NOT assigned copy candidates  assigned copy candidate

A' 250 A 1250 L3
100
1000
10000

A" 1000 11000

A L3
A L2
A L1
A L0

More general hardware architecture than the Dortmund approach, but no global optimization.
Saving/Restoring Context Switch

- **Saving Context Switch (Saving)**
  - Utilizes SPM as a common region shared by all processes
  - Contents of processes are copied on/off the SPM at context switch
  - Good for small scratchpads

Diagram:
- Scratchpad
- Process $P_2$
- Saving/Restoring at context switch
Non-Saving Context Switch

- Partitions SPM into disjoint regions
- Each process is assigned a SPM region
- Copies contents during initialization
- Good for large scratchpads
Hybrid Context Switch

Hybrid Context Switch (Hybrid)

- Disjoint + Shared SPM regions
- Good for all scratchpads
- Analysis is similar to Non-Saving Approach
- Runtime: $O(nM^3)$
Multi-process Scratchpad Allocation: Results

- For small SPMs (64B-512B) Saving is better
- For large SPMs (1kB- 4kB) Non-Saving is better
- Hybrid is the best for all SPM sizes.
- Energy reduction @ 4kB SPM is 27% for Hybrid approach

Energy Consumption (mJ)

- Energy (SPA)
- Energy (Saving)
- Energy (Hybrid)
- Energy (Non-Saving)
- CopyEnergy (Saving)
- CopyEnergy (Hybrid)

edge detection, adpcm, g721, mpeg
Dynamic set of multiple applications

Compile-time partitioning of SPM no longer feasible

- Introduction of SPM-manager
  - Runtime decisions, but compile-time supported

Approach overview

- 2 steps: compile-time analysis & runtime decisions
- No need to know all applications at compile-time
- Capable of managing runtime allocated memory objects
- Integrates into an embedded operating system

Using MPArm simulator from U. Bologna
Results

► MEDIA+ Energy
- Baseline: Main memory only
- Best: Static for 16k → 58%
- Overall best: Chunk → 49%

► MEDIA+ Cycles
- Baseline: Main memory only
- Best: Static for 16k → 65%
- Overall best: Chunk → 61%
Comparison of SPMM to Caches for SORT

- Baseline: Main memory only
- SPMM peak energy reduction by 83% at 4k Bytes scratchpad
- Cache peak: 75% at 2k 2-way cache
- SPMM capable of outperforming caches
- OS and libraries are not considered yet

Chunk allocation results:

<table>
<thead>
<tr>
<th>SPM Size</th>
<th>Δ 4-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>74.81%</td>
</tr>
<tr>
<td>2048</td>
<td>65.35%</td>
</tr>
<tr>
<td>4096</td>
<td>64.39%</td>
</tr>
<tr>
<td>8192</td>
<td>65.64%</td>
</tr>
<tr>
<td>16384</td>
<td>63.73%</td>
</tr>
</tbody>
</table>
Summary

Impact of memory architecture on execution times & energy consumption

- The SPM provides
  - Runtime efficiency, energy efficiency, timing predictability

- Allocation strategies
  - Static allocation
    - Partitioning
    - Timing predictability
  - Dynamic allocation
    - Tiling
    - Multiple hierarchy levels
    - Multiple processes
    - Dynamic sets of processes

- Savings dramatic, e.g. ~ 95% of the memory energy