Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund
Informatik 12
Germany

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

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

Design repository

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

Design
8: Test

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
Compilers for embedded systems

- Introduction
- Energy-aware compilation
- Memory-architecture-aware compilation
- Reconciling compilers and timing analysis
- Compilation for digital signal processors
- Compilation for VLIW processors
- Compiler generation, retargetable compilers, design space exploration
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
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:

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

Exploitation of the memory hierarchy

```plaintext
R2:=a[0];
for i:= 1 to 10 do
begin
    R1:= a[i];
    C:= 2 * R1 + R2;
    R2 := R1;
end;
```
Energy-aware compilation (3)

- Energy-aware **scheduling**: the order of the instructions can be changes 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

Address space

ARM7TDMI cores, well-known for low power consumption

Example

0

scratch pad memory

no tag memory

FFF..
Very limited support in ARMcc-based tool flows

1. Use pragma in C-source to allocate to specific section:
For example:
   #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

Compilers for embedded systems

Book section 7.3

- Introduction
- Energy-aware compilation
- Memory-architecture-aware compilation
- Reconciling compilers and timing analysis
- Compilation for digital signal processors
- Compilation for VLIW processors
- Compiler generation, retargetable compilers, design space exploration
Migration of data & instructions, global optimization model (TU Dortmund)

Example:

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 $G = \sum g_k$, respecting size of SPM $SSP \geq \sum s_k$.

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

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

**Integer programming formulation:**

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

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

Multi_sort (mix of sort algorithms)

Feasible with standard compiler & postpass optimization

Measured processor / external memory energy + CACTI values for SPM (combined model)

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.
Taking consecutive basic blocks into account

**Approach:**

- Consider sets of consecutive BBs as a new kind of basic blocks ("multi blocks")
- Add a constraint preventing the same block from being selected twice:

\[
x(BB_b) + x(F_i) + \sum_{j \in \text{multiblocks}(b)} x(BB_j) \leq 1 \\
\forall b \in \{\text{blocks}\} \cup \{\text{multi blocks}\}
\]

- Block \( b \) is either moved individually, as part of a function, as part of one of its enclosing multi-blocks or not at all.
Allocation of basic blocks, sets of adjacent basic blocks and the stack

Requires generation of additional jumps (special compiler)

![Graph showing cycle and energy consumption with scratch pad size in bytes.](Image)
Savings for memory system energy alone

Combined model for memories
How much better can we get?

\[
\text{improvement} = \frac{1}{(1 - P) + \frac{P}{S}} \quad \text{(Amdahl’s law)}
\]

where

- \( P \): fraction of memory references replaced by faster/more energy efficient memory

and

- \( S \): speed/energy improvement

Important not to have many “untouchable” references \((1-P)\), otherwise even \( S \to \infty \) does not help.
Scratch-pad 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

![Diagram of the process]

1. C program
2. SPM size
3. Memory-aware compiler
4. ARMulator
5. Executable
6. aiT
7. Actual performance
8. Worst case execution time
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:

![Graph showing cycles vs. Scratchpad size for G.721](image1)

Using Unified Cache:

![Graph showing cycles vs. Cache size for G.721](image2)

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?
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.
Overlaying/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

\[ \text{MO} = \{A, T1, T2, T3, T4\} \]

\[ \text{SP Size} = |A| = |T1| \ldots = |T4| \]

Solution:
- \[ \text{SPILL_STORE}(A); \]
- \[ \text{SPILL_LOAD}(T3); \]
- \[ \text{A} \rightarrow \text{SP} \text{ & } T3 \rightarrow \text{SP} \]

\[ \text{SPILL_LOAD}(A); \]
Overlaying/dynamic replacement within SPM

Edge detection relative to non-overlaying/static allocation

![Bar chart showing Processor Energy, Memory Energy, Total Energy, and Execution Time with Scratchpad Size (Bytes) on the x-axis and energy values on the y-axis. The chart includes data for Scratchpad Sizes of 64, 100, 128, 200, and 256 bytes, as well as an average.]
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 -

```c
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] * W[k][j]
```

Tiling

```c
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];
         U[it:it+Sb-1,jt:jt+Sb-1]=U[it:it+Sb-1,jt:jt+Sb-1]
             + V[it:it+Sb-1,1:n]
             * 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

Cache

- Dynamic
- Static
- Hybrid

Kandemir@DAC01

Kandemir@DATE06
Hierarchical memories: Memory hierarchy layer assignment (MHLA) (IMEC)

n layers with "partitions" consisting of modules

Partition n.1

... 

Partition 2.1

Partition 2.2

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

Processor

Memory hierarchy layer assignment (MHLA) - Copy candidates -

### Copy candidate

<table>
<thead>
<tr>
<th>int A[250]</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;10; i++)</td>
</tr>
<tr>
<td>for (j=0; j&lt;10; j++)</td>
</tr>
<tr>
<td>for (k=0; k&lt;10; k++)</td>
</tr>
<tr>
<td>for (l=0; l&lt;10; l++)</td>
</tr>
<tr>
<td>f(A[j*10+l])</td>
</tr>
<tr>
<td>size=10; reads(A)=10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int A[250]</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;10; i++)</td>
</tr>
<tr>
<td>for (j=0; j&lt;10; j++)</td>
</tr>
<tr>
<td>for (k=0; k&lt;10; k++)</td>
</tr>
<tr>
<td>for (l=0; l&lt;10; l++)</td>
</tr>
<tr>
<td>f(A'[j*10+1])</td>
</tr>
<tr>
<td>size=100; reads(A)=100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int A[250]</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;10; i++)</td>
</tr>
<tr>
<td>for (j=0; j&lt;10; j++)</td>
</tr>
<tr>
<td>for (k=0; k&lt;10; k++)</td>
</tr>
<tr>
<td>for (l=0; l&lt;10; l++)</td>
</tr>
<tr>
<td>f(A'[j*10+1])</td>
</tr>
<tr>
<td>size=100; reads(A)=100</td>
</tr>
</tbody>
</table>

### Copy candidate

<table>
<thead>
<tr>
<th>int A[250]</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;10; i++)</td>
</tr>
<tr>
<td>for (j=0; j&lt;10; j++)</td>
</tr>
<tr>
<td>for (k=0; k&lt;10; k++)</td>
</tr>
<tr>
<td>for (l=0; l&lt;10; l++)</td>
</tr>
<tr>
<td>f(A'[j*10+1])</td>
</tr>
<tr>
<td>size=100; reads(A)=100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int A[250]</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;10; i++)</td>
</tr>
<tr>
<td>for (j=0; j&lt;10; j++)</td>
</tr>
<tr>
<td>for (k=0; k&lt;10; k++)</td>
</tr>
<tr>
<td>for (l=0; l&lt;10; l++)</td>
</tr>
<tr>
<td>f(A'[j*10+1])</td>
</tr>
<tr>
<td>size=100; reads(A)=100</td>
</tr>
</tbody>
</table>

### Graph

- **Reads(A)**
- **Size**

- **Copy candidate A', A'' in small memory**
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

---

[Content of the bottom part of the slide as a footnote]

---

Memory hierarchy layer assignment (MHLA) - Approach -

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

Current assignment
- NOT assigned copy candidates
- assigned copy candidates

Next assignment
- NOT assigned copy candidates
- assigned copy candidate

Platform A' A A''
- L3
- L2
- L1
- L0

1250 11000 1000 10000

250 1250

100 1000 10000

100 350

1250 1100

Step 1

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

Saving/Restoring at context switch

Saving/Restoring at context switch

Saving Context Switch (Saving)

- Utilizes SPM as a common region shared all processes
- Contents of processes are copied on/off the SPM at context switch
- Good for small scratchpads
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

- 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
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 on execution times & energy consumption

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

- Allocation strategies
  - Non-overlaying/static allocation
    - Variables, functions, basic blocks, stack, heap, partitioning
    - Timing predictability
  - Overlaying/dynamic allocation
    - CFG-based, tiling
    - Multiple hierarchy levels
    - Multiple processes
    - Dynamic sets of processes

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