Generic Software pipelining at the Assembly Level

Markus Pister
pister@cs.uni-sb.de

Daniel Kästner
kaestner@absint.com
Embedded Systems (ES) are widely used
- Many systems of daily use: handy, handheld, …
- Safety critical systems: airbag control, flight control system,…
- Rapidly growing complexity of software in ES
Embedded Systems (2)

- Hard real time scenarios:
  - Short response time
    - Flight control systems, airbag control systems
  - Low power consumption and weight
    - Handy, handheld, …

- Urgent need for fast program execution under the constraint of very limited code size
Code Generation for ES

- Program execution times *mostly spent in loops*
- Modern processors offer massive instruction level parallelism (ILP)
  - VLIW architecture: e.g. Philips TriMedia TM1000
  - EPIC architecture: e.g. Intel Itanium
Code Generation for ES (2)

- Many existing compilers cannot generate satisfactory code (cannot exploit ILP)
- High effort enhancing them to cope with advanced ILP

- Improving the quality of legacy compilers by
  - Starting at the assembly level
  - Building flexible postpass optimizers
    - Can be quickly retargeted
    - Improve generated code quality significantly
PROPAN-Overview

- Postpass-oriented Retargetable Optimizer and Analyzer
In this talk

- **Software Pipelining** as a post pass optimization
  - Important technique to exploit ILP while trying to keep code size low
- Static cyclic and global instruction scheduling method
- Idea: overlap the execution of consecutive iterations of a loop
Software Pipelining

- Computes new (shorter) loop body
- Overlapping loop iterations
- Exploits ILP

- **Modulo Scheduling**
  - Initiation interval (II)
  - divides loop into Stages
  - Schedule operations modulo II
- Iterative Modulo Scheduling
Minimum Initiation Interval

- Resource based: $MII_{res}$
  - Determined by the resource requirements
  - Approximation for optimal bin packing

- Data dependence based: $MII_{dep}$
  - Delays imposed by cycles in DDG

$MII = \text{Max} (MII_{res}, MII_{dep})$

- Basis for Kernel (modulo) computation
Scheduling Phase

- **Flat Schedule**
  - Maintain partial feasible schedule
  - Algorithm:
    - Pick next operation
    - Compute slot window \([E_{\text{Start}}, L_{\text{Start}}]\)
    - Search feasible slot within \([E_{\text{Start}}, L_{\text{Start}}]\)
    - Conflict: unschedule some operations and force current operation into partial schedule

- **Kernel**
  - Schedule operations from the Flat Schedule modulo \(I\)
Prologue / Epilogue

- „fills up“ or „drains down“ the pipeline respectively
Characteristics of the Post pass approach

- Integration of the pipelined loop into the surrounding control flow
  - Modification of branch targets needed

- Reconstruction of the CFG is complex and difficult
  - Resolving targets of computed branches/calls and switch tables

```
1d32d(20) r4 → r34

ijmpt r1 r34
```
Characteristics of the Post pass approach (2)

- Register allocation is already done
  - Assignment can be changed with Modulo Variable Expansion
  - Liveliness properties must be checked before register renaming

- Applicable for inline assembly and library code

- Data dependencies at the assembly level are more general
  - More generality leads to a more complex DDG
  - One single array access → multiple assembler operations
Data dependences at the assembly level

```plaintext
i = 0;
...
i = i + 1;
...
j = array[i];
...
```

```plaintext
ld32d(8) r6 → r7
...
iadd(1) r7 → r8
...
ld32d(20) r4 → r10
iadd r10 r8 → r9
ld32d r9 → r11
```
DDG at the assembly level
TriMedia TM1000 - Overview

- Digital Signal Processor for Multimedia Applications designed by Philips
- 100 MHz VLIW-CPU (32 Bit)
- 128 General Purpose Registers (32 Bit)
- 27 parallel functional units
TM1000 — VLIW-Core

SDRAM

Instruction Cache

Issue Slot 1
- CONST
- ALU
- SHIFTER
- FALU
- DSPALU

Issue Slot 2
- CONST
- ALU
- SHIFTER
- DSPMUL
- BRANCH
- IFMUL
- FTOUGH

Issue Slot 3
- CONST
- ALU
- FCOMP
- DSPMUL
- BRANCH
- IFMUL
- DSPALU

Issue Slot 4
- CONST
- ALU
- DMEM
- FALU

Issue Slot 5
- CONST
- ALU
- DMEM
- DMEMSPEC
TriMedia TM1000 - Properties

- Instruction set
  - Register-based addressing modes
  - Predicative execution: register-based
  - Load/store architecture
  - Special multimedia operations

- 5 Issue Slots, 5 Write-Back Busses

- Irregular execution times for operations
  - Write-Back Bus has to be modeled independently
Experimental Results

- Files from DSPSTONE- and Mibench-Benchmark
- Best performance gains for chain like DDG’s (up to 3,1)

Performance increase for the execution of 100 iterations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Original</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic math 1</td>
<td>1100</td>
<td>796</td>
</tr>
<tr>
<td>basic math 2</td>
<td>800</td>
<td>792</td>
</tr>
<tr>
<td>bitco count</td>
<td>1000</td>
<td>798</td>
</tr>
<tr>
<td>chain</td>
<td>800</td>
<td>292</td>
</tr>
<tr>
<td>chain 2</td>
<td>900</td>
<td>295</td>
</tr>
<tr>
<td>dfir</td>
<td>1200</td>
<td>1200</td>
</tr>
<tr>
<td>dlims</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>dmat 1x3</td>
<td>2500</td>
<td>799</td>
</tr>
<tr>
<td>dmat rix1</td>
<td>4000</td>
<td>1985</td>
</tr>
<tr>
<td>FFT</td>
<td>5400</td>
<td>2386</td>
</tr>
<tr>
<td>mam u2</td>
<td>1100</td>
<td>895</td>
</tr>
</tbody>
</table>
Experimental Results (2)

- Moderate code size increase (average: 1,42)

![Code size increase chart]

<table>
<thead>
<tr>
<th>Instructions</th>
<th>basic math1</th>
<th>basic math2</th>
<th>bitcount</th>
<th>chain</th>
<th>chain2</th>
<th>dfir</th>
<th>dlms</th>
<th>dmat1 x3</th>
<th>dmatr x1</th>
<th>FFT</th>
<th>mamu 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>11</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>10</td>
<td>25</td>
<td>40</td>
<td>54</td>
<td>11</td>
</tr>
<tr>
<td>Kernel</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>20</td>
<td>24</td>
<td>9</td>
</tr>
<tr>
<td>Overall</td>
<td>20</td>
<td>8</td>
<td>22</td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>39</td>
<td>45</td>
<td>58</td>
<td>31</td>
</tr>
</tbody>
</table>
Experimental Results (3)

- Computed MII mostly is already feasible (73%)

Feasibility of the MII

<table>
<thead>
<tr>
<th></th>
<th>basic math1</th>
<th>basic math2</th>
<th>bitcount</th>
<th>chain</th>
<th>chain2</th>
<th>dfir</th>
<th>dlms</th>
<th>dmat1 x3</th>
<th>dmat1 x1</th>
<th>FFT</th>
<th>mamu 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MII</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>8</td>
<td>5</td>
<td>12</td>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>II</td>
<td>7</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>19</td>
<td>5</td>
</tr>
</tbody>
</table>
Future Work

- Nested loops:
  - Process loops from innermost to outermost one
  - Treat an inner loop as one instruction (”meta-instruction”)
- Parallelize Prologue and Epilogue code with surrounding code
  - Can be done by existing acyclic scheduling techniques like list scheduling
- Delay Slot filling
Conclusion

- Embedded Systems creates need for fast program execution under constraint of very limited code size
- Overcome limitation of existing compilers by retargetable postpass optimizer
- Fast program execution by exploiting ILP with Software Pipelining
- Iterative Modulo Scheduling at the Assembly level
  - Characteristics of the Postpass approach
- Experimental results show
  - a speedup of up to 3.1 with
  - an average code size increase of 1.42