Technique for obtain data address for model a data cache in native source-code HW/SW co-simulation

Luis Díaz, Hector Posadas & Eugenio Villar
University of Cantabria

{luisds, posadash, villar}@teisa.unican.es
Index

- Introduction
- Data cache modeling solution
  - Memory access annotations
  - Data cache model
- Test and Results
- Conclusions
Introduction

- Fast mechanisms for performance estimation required early in the design flow
  - Lots of simulations required for design space exploration
  - Speed more important than accuracy
Simulator Requirements

- Provide sufficiently accurate metrics
  - Time and power estimation
  - Consider SW & HW effects
    - Accuracy limited by the early design flow step
- As Fast as possible
  - Close to native, functional execution
  - Instruction Set Simulators (ISS) too slow
    - Intermediate solution required
Fast simulation technique

- Native co-simulation
  - Virtual HW platform model
    - Model the architecture and HW functionality
      - Both functional and performance effects
    - No Processor model included
  - Native execution of annotated SW code
    - Annotate SW with time and power information
    - Requires a RTOS model

- Minimal SW annotation overhead required
Improvements needed

- Previous work on SW native modeling
  - Effects of the processor core
    - Time/Power metrics
  - SW platform
    - Operating System model, middleware, drivers, …
  - Caches
    - Instruction cache misses solved
    - Lack of data cache models
Fast simulation technique

- Replace ISS
- Add Performance information to SW code
  - Instruction execution
  - Bus accesses
    - I/O (RTOS)
    - I-Cache misses

Diagram:

- Annotated application SW (C/C++)
- OS model
- SystemC models
  - Peripheral model
  - Bus model
  - Memory model
- Peripheral model

Host computer
Cache modeling

- Common ISS data cache model
  - Based on memory access traces

- Native simulation
  - Address traces are not directly available
  - Required a mechanism to produce the traces
    - Detect data accesses
    - Obtain data addresses
Instruction-cache modeling

- Solved in GLSVLSI’10

- Solution not valid for data cache
  - Spatial locality of instructions
  - Source-code instructions only once in memory
  - Instruction address known in compilation time
Instruction-cache Modeling

- Data structure with cache line information declared in the SW code

```
struct icache_line {
  char num_set;
  char hit;
}
```

```
while (a){
  static icache_line line = {addr%8,0};
  b = c+d[a];
  c += 2;
  a -= 1;
  if(!line.miss) insert_line(line);
}
```

```
insert_line(icache_line *line) {
  icache_line *victim;
  victim = get_victim_line(line->set);
  if (victim != NULL) victim->hit = 0;
  line -> hit = 1;
  bus_model_transfer(line);
}
```

Cache miss
Data cache annotation process

- Cache effects
  - Mixed static/dynamic solution
- Add marks to the SW code
  - Analyze the code with a grammar
  - Extract data accesses
  - Include cache annotations
- Simulate
Obtaining data access addresses

- Absolute addresses in host != addresses in target platform
  - Target platform address obtained from corrected native ones
- Data storage order is specified by
  - Linkage order
  - Order in which variables are declared
- Using the same compiler front-end and the same linkage order, the order of the variables within each section is maintained.
Obtaining data access addresses

- Ensure host data have same size than in target
  - Source code is modified using equivalent data type
- Different data types stored in different sections
  - E.g. ELF format
    - Global variables in “data”, “rodata” and “bss” sections
    - Local variables in the “stack” section
    - Dynamic data are stored in the “heap”
- Sections start in different address in host & target
  - Relative addresses within each section are correct
Data addresses corrections

- Change the base address of the variable
  - Identify the section of the variable from his address
  - Subtract to the address the base of the section in the host
  - Add base address of the same section in the target.
  - Host addresses from elf format, “/proc/self/stat” & “/proc/self/maps”
  - Target addresses provided by the designer.
Detecting data accesses

- Variable reads/writes detected in the grammar
  - A variables used in the code -> load/store
  - A write is performed when
    - Variable is in the left side of an equal expression (=, +=…)
    - when there is an operator such as ++ or --.
    - For arrays, the write access modelled corresponds to single element access, not to the entire array.
  - Arrays and pointers imply multiple accesses
    - Array index is a variable
    - Multiple indirection levels
Annotating accesses in code

- **How:** Modeling data cache accesses
  - `dcache_read(void *address)`
  - `dcache_write(void * address)`

- **Where:**
  - Accesses inside basic blocks annotated at the end of the block
    - Control operators that modify the data
      - E.g. `v[i++]` requires checking `v[i]`, not `v[i+1]`
    - `a?b:c` operator is analyzed as a control statement
  - Accesses in control statements require specific annotations
Fast Data-Cache Modeling

Instruction Set Simulator

Annotated SW code

Search in the cache

Provoke miss

Update cache

HW Platform model

Addresses

Cache model

Search in the cache

Provoke miss

Update cache

HW Platform model

Addresses
Fast data-cache model

- Create an array with all possible cache lines
  - Complete memory: $2^{32}$ addresses
  - Line size: e.g. 32 bytes
    - Total lines: $2^{27}$
  - Use a bit to indicate is line is in cache or not
    - Array of 16Mbytes for cache

- Checking
  - `#define CACHED(addr) mem[(addr>>8)]&1<<(addr>>5)&7`
  - `if(CACHED(&variable)) dcache_insert_line(&addr);`
Data cache model

- Search a data in the cache
- Round robin replacement
- Check the bit dirty
Data cache model

- Data cache model like ISS models
  - \( n \) rows \( \rightarrow \) one for each line that can accommodate the actual cache
  - two columns \( \rightarrow \)
    - tag cache line
    - bit dirty

\[
\text{int cache[dcache_size/line_size][2];}
\]
Data cache model

- A configurable data cache model has been implemented
- Parameters to configure:
  - total size (dcache_size)
  - degree of associativity (assoc)
  - line size in words (line_size)
  - word size (word_size)
  - number of lines (n_lines=dcache_size/line_size)
  - number of sets (n_sets=n_lines/assoc).

- This model is only for simulate misses and hits in the cache, so the data are not stored.
Test and Results

- Examples checked:
  - Simple examples
  - C implementation of 12.2 Kbps GSM Vocoder
- Target processor → ARM920T
  - Data cache size: 16KB
  - 64-associative
  - Line size: 32 bytes
Test and Results

<table>
<thead>
<tr>
<th>Example</th>
<th>Proposed</th>
<th>Search-based</th>
<th>ISS</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble 1000 elements</td>
<td>127</td>
<td>127</td>
<td>126</td>
<td>0.8</td>
</tr>
<tr>
<td>Bubble 10000 elements</td>
<td>5207383</td>
<td>5207383</td>
<td>5199310</td>
<td>0.16</td>
</tr>
<tr>
<td>Hanoi</td>
<td>44</td>
<td>44</td>
<td>41</td>
<td>7.32</td>
</tr>
<tr>
<td>Factorial</td>
<td>500</td>
<td>500</td>
<td>375</td>
<td>33.33</td>
</tr>
</tbody>
</table>

- Main source of error:
  - Number of registers saved in function calls
  - Error compensated in large codes
Comparison in simulation speed
Test and Results

- GSM Vocoder
  - ISS: Skyeye

<table>
<thead>
<tr>
<th>Frames</th>
<th>Data cache misses</th>
<th>Simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Proposed</td>
<td>ISS</td>
</tr>
<tr>
<td>1</td>
<td>660</td>
<td>670</td>
</tr>
<tr>
<td>4</td>
<td>2370</td>
<td>2452</td>
</tr>
<tr>
<td>7</td>
<td>4104</td>
<td>4235</td>
</tr>
<tr>
<td>10</td>
<td>5915</td>
<td>6026</td>
</tr>
</tbody>
</table>
Conclusions

- Data cache effects can be accurately modeled in native execution
- Data accesses can be obtained analyzing the code with a grammar
- Native addresses can be used
  - Ensure data type sizes are valid
  - Adjust the base address of each section