Fast and Accurate Source-Level Simulation Considering Target-Specific Compiler Optimizations

Oliver Bringmann
Outline

- Embedded Software – Challenges
- TLM2 Platform Modeling
- Source-Level Timing Instrumentation
- Consideration of Compiler Optimizations
- Experimental Results
Trend Towards Multi-Core Embedded Systems

Example: Automotive Domain

- Transition from passive to active safety
- Active systems: Innovation by interaction of ECUs, added-value by synergetic networking
- Multi-sensor data fusion and image recognition for automated situation interpretation in proactive cars

Well-tailored Embedded Platforms

- Increasing computation and energy requirements
- Distributed embedded platforms with energy-efficient multi-core embedded processors

Challenges

- Early verification of global safety and timing requirements
- Consideration of the actual software implementation w.r.t. the underlying hardware
- Scalable verification methodology for multi-core & distributed embedded systems
Modeling techniques providing a holistic system

- Derivation of an optimized network architecture
- Generation of abstract executable models (virtual prototypes)
TLM Timing and Platform Model Abstractions

Timing abstractions

- **Untimed (UT) Modeling**
  - notion of simulation time is not required, each process runs up to the next explicit synchronization point before yielding

- **Loosely Timed (LT) Modeling**
  - Simulation time is used, but processes are temporally decoupled from simulation time until it reaches an explicit synchronization point

- **Approximately Timed (AT) Modeling**
  - Processes run in lock-step with SystemC simulation time. Annotated delays are implemented using timeouts (wait) or timed event notifications

Platform model abstractions

- **CP** = Communicating Processes; parallel processes with parallel point-to-point communication
- **CPT** = Communicating Processes + Timing
- **PV** = Programmers View; scheduled SW-computation and/or scheduled communication
- **PVT** = Programmers View + Timing
- **CC** = Cycle Callable; cycle count accurate timing behavior for computation and communication
SystemC TLM 2.0 Loosely Timed Modeling Style

- **SystemC (lock-step sync.)**

  ```
  ...
  wait (1, SC_MS);
  ...
  wait (1, SC_MS);
  do_communication();
  wait (1, SC_MS);
  ...
  ```

- **SystemC + TLM 2.0 Loosely Timed Modeling Style (LT)**

  ```
  ...
  local_offset += sc_time(1, SC_MS);
  ...
  local_offset += sc_time(1, SC_MS);
  do_communication(local_offset);
  local_offset += sc_time(1, SC_MS);
  if (local_offset >= local_quantum) {
    wait (local_offset);
    local_offset = SC_ZERO_TIME;
  }
  ...
  ```

advance simulation time
Inaccuracies induced by Temporal Decoupling

- parallel accesses to shared resources (cache, bus)
- conflicts may delay concurrent accesses
- temporally decoupled simulation (LT)

Simulation: Core 1 \( t=0 \) \( t=1 \) Core 2
Reality: Core 1 \( t=0 \) \( t=1 \) \( t=2 \) Core 2

- higher priority access simulated after lower priority access
  \( \Rightarrow \) preemption not detected
- explicit synchronization entails severe performance penalty
- *alternative approach: early completion with retro-active adjustments*
Conflict Resolution in TLM Platforms

- **TLM+ Resource Model**
  - access arbitration for each relevant simulation step despite temporal decoupling
  - delayed activation of a core’s simulation thread upon conflict
  - arbitration induces no additional context switches in SystemC simulation kernel
  - based on SystemC TLM-2.0 (downward compatible)

- **Universal approach for fast and accurate TLM simulation**
  - Arbitration using „Resource Model“ shared by all users of a resource
  - synchronization of bus accesses
  - simulation of parallel RTOS software tasks
Simulation-Based Timing Analysis

Interpretation of Binary Code

- Software and hardware model separated
- Independent Compilation
- HW: RTL model or instruction set simulator
- Software timing induced by hardware model
- Problem: Long simulation time

Software Simulation

- Common system model for SW and HW
- Combined compilation of HW and SW
- High simulation speed
- Problem: Precise timing analysis is difficult at source-code level
Source-Level Timing Instrumentation

**Goal**
- Static timing prediction of basic blocks with dynamic error correction

**Proposed Approach**
- **Compilation into binary code** enriched with debugging information
- **Static execution time analysis** with respect to architectural details (e.g. pipeline mode, cache model, ...)
- **Back-annotation of the analyzed timing information** into the original C/C++ source code

**Advantages**
- Consideration of **architectural details**
- **Efficient compilation** onto simulation host
- Considering the influences of **dynamic timing effects**

```c
int f( int a, int b, int c, int d )
{
    int res;
    res = (a + b) << c - d;
    delay( 3, ms );
    return res;
}
```

**Important:**
- Requires accurate relation between source code and binary code
- Run-Time Models for Branch Prediction and Caching have to be incorporated
Combined Source-Level Simulation and Target Code Analysis: State of the Art

- Schnerr, Bringmann et al. [DAC 2008]
  - static pipeline analysis to obtain basic block execution times
  - instrumentation code to determine cache misses dynamically
  - no compiler optimizations

- Wang, Herkersdorf [DAC 2009]; Bouchhima et al. [ASP-DAC 2009]; Gao, Leupers et al. [CODES+ISSS 2009]
  - use modified compiler backend to emit annotated „source code“
  - supports compiler optimizations as binary code and annotated source have same structure

- Lin, Lo, Tsay [ASP-DAC 2010]
  - very similar to approach of [DAC2008]
  - claims to support compiler optimizations, no details

- Castillo, Villar et al. [GLSVLSI 2010]
  - improves cache simulation method of [DAC2008]
  - supports compiler optimizations without control flow changes
Timing Instrumentation and Platform Integration

**Cycle Calculation Functions**
- Use an architectural model of the processor for the cycle calculation

**C code corresponding to a basic block**
- Main function for cycle calculation

- \( \text{delay(statically predicted number of cycles);} \)
- \( \text{delay(cycleCalculationICache(iStart,iEnd));} \)
- \( \text{delay(cycleCalculationForConditionalBranch());} \)

**Function consume**
- VP synchronization with respect to accumulated delays

- \( \text{consume(cycles collected with delay);} \)
- \( \text{e.g. I/O access} \)

**Architectural Model**
- Cache Model
- Branch Prediction Model

**Usage of the Loosely-Timed (LT) Modeling Approach**
Compiler Optimizations and the Relation between Source Code and Binary Code

- **Dead Code Elimination**
  - binary-level control flow gets simpler
  - no real problem for back-annotation

- **Moving Code (e.g. Loop Invariant Code Motion)**
  - not necessarily modifies binary-level control flow
  - blurs relation between binary-level and source-level basic blocks

- **Loop Unrolling**
  - complete unrolling is simple (annotate delays in front of loop)
  - partial unrolling requires dynamic delay compensation

- **Function Inlining**
  - may induce radical changes in control flow graph
  - introduces ambiguity as several binary-level basic reference identical source locations

- **Complex Loop Optimizations**
  - basic block structure may change completely (Loop Unswitching)
  - execution frequency of basic blocks due to transformation of iteration space (Loop Skewing)
Effects of Compiler Optimizations

**main.c**

```c
main.c
... 42  int mul (int a, int b)
43  {
44      return a * b;
45  }
46
47  void arrayMul (int *A, int *B, int *C, int s)
48  {
49      for (int i = 0; i < (s - 1); i++) {
50          C[i] = mul (A[i], B[i]);
51      }
52  }
...```

**main ir**

```c
main ir
... 42  int mul (int a, int b)
43  {
44      return a * b;
45  }
46
47  void arrayMul (int *A, int *B, int *C, int s)
48  {
49      int i = 0;
50      int tmp = s - 1;
51      while (i < tmp) {
52          C[i] = A[i] * B[i];
53              i++;
54      }
55  }
...```

**Code Transformations**
Effects of Compiler Optimizations

```c
main.c
...
42 int mul (int a, int b)
43 {
44    return a * b;
45 }
46
47 void arrayMul (int *A, int *B, int *C, int s)
48 {
49    for (int i = 0; i < (s - 1); i++) {
50        C[i] = mul (A[i], B[i]);
51    }
52 }
...
```

```c
main.ir
...
47 void arrayMul (int *A, int *B, int *C, int s)
48 {
49    int i = 0;
49    int tmp = s - 1;
49    int tmp2 = tmp - (tmp % 4);
49    while (i < tmp2) {
44        C[i] = A[i] * B[i];
44        C[i+1] = A[i+1] * B[i+1];
44        C[i+2] = A[i+2] * B[i+2];
44        C[i+3] = A[i+3] * B[i+3];
49        i += 4;
49    }
49    while (i < tmp) {
44        C[i] = A[i] * B[i];
44        i++;
49    }
52 }
...
```

```c
a.out
...
0x8000 addi r1 r0 0x0      main.c:49
0x8004 subi r2 r9 0x1      main.c:49
0x8008 modi r3 r2 0x4      main.c:49
0x800C sub r3 r2 r3        main.c:49
0x8010 zol r3 0x2          main.c:49
0x8014 mul r8 r7 r6        main.c:49
0x8018 inc r8 r7 r6        main.c:49
0x801C zol r2 0x2          main.c:49
0x8020 mul r8 r7 r6        main.c:49
0x8024 inc r8 r7 r6        main.c:49
...
```
Using Debug Information to Relate Source Code and Optimized Binary Code

- Compilers usually do not generate accurate debug information for optimized code
- Structure of source code and binary code can be completely different
  ➔ No 1:1 relation between source-level and binary-level basic blocks
  ➔ Simply annotating delay attributes does not work

To perform an accurate source-level simulation without modifying the compiler
- relation between source code and binary code must be reconstructed from debug information
- binary-level control must be approximated during source-level simulation
Constructing the dominator homomorphism relation

Remaining ambiguities (caused by multiple function inlining) can be resolved dynamically using path simulation code
Generating Annotated Source Code

- Reconstruct line references
- Low-level analysis
  - Analyze basic block execution times using proven commercial tool AbsInt aiT
- Instrumentation and back-annotation
  - Add reference markers to original source code
  - Generate path simulation code to determine binary control flow dynamically
  - Path simulation code simulates execution through binary-level control flow graph
  - Control flow reconstruction allows:
    - Precise consideration of branch penalties, branch prediction model can be included
    - Not matching all basis blocks to a source-level statement without losing information

```
main.c
...
23  int i = 0;
24  for (int c = 0; c <= pow; c++) {
25      i = i * 2;
26  }
27  pot = i;
...

main-instrumented.c
...
23  int i = 0;  sim (0x8000);
24  for (int c = 0; c <= pow; c++) {
25      i = i * 2;  sim (0x8010);
26  }
27  pot = i;  sim (0x801C);
...

path-simulation.c
void sim (int address)
{
...
    if (previous == 0x800C  
        && address == 0x801C)
        cycles += 3;
...
    previous = address;
}
```
Instrumented source code provides functionality

Reconstruction considers structure of source code and binary

Arbitrary properties can be simulated: timing, memory accesses, power,...
Results

The graph illustrates the runtime in milliseconds for different benchmarks (crc, edn, matmul, nsichneu, statemate) compared between Annotation and Open Virtual Platforms. The x-axis represents the benchmarks, and the y-axis represents the runtime in milliseconds. The darker bars indicate the runtime for Annotation, while the lighter bars represent the runtime for Open Virtual Platforms.
Application Example
Traffic Sign Recognition

ADAS Function:
Traffic sign recognition

Functional Network & Hardware Architecture

Virtual & real Prototypes

Images -> Image Capturing -> Region detection

- Preprocessing
- Circle detection
- Segmentation

Classification
- Feature Extraction
- Classification

Traffic signs -> CAMERAS

DISPLAY

RECOGNIZE

CLASSIFY

Architectural Model

Virtual Prototype

Implementation

ADAS Function:
Traffic sign recognition

Images

Traffic signs

Traffic information

Display
Application Example
Traffic Sign Recognition
Conclusion

- Timing analysis for embedded software considering the target software implementation and the influences of the underlying hardware
- Fast and accurate solution by combining the advantages of formal analysis and simulation
  - Timing relations are annotated to the original source code even though code optimizations have been applied
  - Effects of branch prediction and basic block interleaving are easily supported by considering the basic block transitions in the target code
- TLM2 platform modeling provides efficient simulation with late timing corrections using the TLM2 resource model
  - TLM2 resource model controls the synchronization of temporal decoupled platform models
  - Cache accesses are optimistically performed and are corrected afterwards (only timing corrections have to be applied, data corrections not needed)
- Simulation performance is quite similar to native execution of the pure software functionality at the simulation host
  - Highly scalable in terms of the number of processors/processor cores
Thank you very much for your attention!

Questions?

Oliver Bringmann
FZI Forschungszentrum Informatik
Microelectronics System Design
Email: bringmann@fzi.de