Automatic Firmware Design for Application-specific Electronic Systems: Opportunities, Challenges and Solutions

Daniel Große (Univ. Bremen / DFKI)
Joscha Benz (Univ. of Tuebingen)
Vladimir Herdt (Univ. of Bremen)
Martin Dittrich (TU Munich)

This contribution is funded as part of the CONFIRM project (project label 16ES0564-70) within the research program ICT 2020 by the German Federal Ministry of Education and Research (BMBF) and supported by the industrial partners Infineon Technologies AG, Robert Bosch GmbH, Intel Deutschland AG, and Mentor Graphics GmbH.
Outline

1. Overview and Challenges for Firmware Design under Timing and Power Budgets
2. Context-Sensitive Source-Level Timing Simulation
3. Validation of Firmware-based Powermanagement using Constrained-Random Techniques
4. Driver Generation and Optimization of the SW/HW Interface
Application Scenario: Automotive

- Highly integrated SiP solutions: ASIC + MEMS (multicore architectures + DSPs)

- **Challenges:**
  - Modelling and generation of timing- and power-predictable application software
  - ECU firmware has to ensure power and timing intervals
  - Sensors have to be queried in application-specific time frames under fixed power budgets
  - Automatic firmware optimization of MCUs (e.g. in sensors)
Application Scenario: Smartphone

• Heterogeneous multi-cores (CPUs, GPUs, DSPs, sensors)

• Challenges:
  – Tasks of platform firmware:
    • ensures specified power and timing intervals
    • minimizes temporal variance
    • exploits dynamic load conditions
  – Prevention of temporal and spatial temperature peaks
  – Improvement of energy efficiency
  – Ensuring timing intervals for real-time critical functions
Application Scenario: Power Control

- Low Power
- Strong resource constraints
- Real-time
- ICs with simple processors/MCUs

**Challenges:**
- Firmware generation under timing/power awareness
- Firmware optimization

Examples:
- Power Converter
- LED Control
- Sensor SIP
Context-Sensitive Source-Level Timing Simulation

Joscha Benz (Univ. of Tuebingen)
Motivation

Is this calculation fast enough?
Agenda

• Source-Level Timing Simulation
  – Principle & Framework
  – Instrumentation

• Loop Acceleration
  – Results

• Conclusion & Future Work
Source-Level Timing Simulation (SLTS)

1. Source/Binary Code

```c
int foo() {
    for(...) {
        for(...) {
            ...
        }
    }
}
```

2. Matching

![Diagram showing matching process]

3. Instrumentation

```c
... int i = 0; bb (0x8000);
for (int c = 0; c <= pow; c++) {
    i = i * 2;  bb (0x8010);
}
... pot = i;  bb (0x801C);
...
```

void bb (int address)
{
    ...
    if (lastBlock == 0x8000 && nextBlock == 0x8010)
        delay (0x8000, 0x8010);
    ...
    lastBlock = nextBlock;
}

Path simulation code + Timing
Basic Block Timing – How to determine?

- By annotating statically analyzed timings
- By annotating dynamically measured timings

Timing is context-dependent!
Instrumentation

Complete Instrumentation

- Precise ✓
- Slower

Partial Instrumentation I

- Not precise
- Fast ✓

Partial Instrumentation II

- Precise ✓
- Fast ✓
int c = 200;

while(c-->=0) {
    a[c] = rand() % c;
}

Instrumentation
int c = 200;

while(c-->=0&&simulate_bb_s_25()) {
    a[c] = rand() % c;
}

Instrumentation
```c
int simulate_bb_s_24() {
    switch(last_block) {
        case 22:
            cycles += 15;
            break;
        case 25:
            cycles += 4;
            break;
        case 26:
            cycles += 8;
            break;
    }
    last_block = 24;
    return 1;
}
```

```c
int c = 200;

while(c-->=0&&simulate_bb_s_25()) {
    a[c] = rand() % c;
}
```

```c
int simulate_bb_s_27();
```
Instrumentation

• Partial instrumentation
  – Already very fast
    • Context-Sensitive: 2019 MIPS average throughput [1]
  → Can we go further?

• Loop Acceleration
  – Much simulation time consumed
  – Especially for simple Loops
Loop Acceleration in SLTS
Loop Acceleration in SLTS – Pro & Con

- Reduction of instrumentation points
  - Enable compiler optimizations
  - Decrease simulation run-time

- Decrease of Accuracy
  - Conservative loop bounds
  - Virtual unrolling
Loop Acceleration in SLTS - Pro & Con

• Decrease of Accuracy
  – How to handle?

• Heuristically accelerate Loops
  – User-provided percentage

• Calculate Expected Inaccuracy
  – Loop bounds: \[ \frac{L_{upper} - L_{lower}}{L_{upper}} \cdot 100 \]
  – Loop paths: \[ \frac{\max(len(path)) - \min(len(path))}{\max(len(path))} \cdot 100 \]

• Accumulated Expected Inaccuracy must not exceed user-provided percentage
Loop Acceleration in SLTS - Evaluation

• Cortex M0+
• Benchmarks
  – Mälardalen (Selection)
• Reference Times
  – Measured using GPIO-Pin
• Experiments
  – Execution-Time Predictions
  – Simulation Run-Time Measurement
Loop Acceleration in SLTS - Results

Run-time of Simulation

Execution Time
Conclusion & Future Work

• Further improvement of simulation run-time possible
  – Using a Heuristic to contain loss of accuracy

• Use context-sensitive flow-facts
  – Derive tighter loop bounds

• CONFIRM: Language Extension
  – Constructs allowing to define tighter bounds
References/Further Reading


2. S. Schulz, O. Bringmann, “Accelerating Source-Level Timing Simulation”, 2016 Design Automation & Test in Europe (DATE)
Validation of Firmware-based Powermanagement using Constrained-Random Techniques

Vladimir Herdt (Univ. of Bremen)
Motivation

• Efficient power management is very important for modern SoC’s
• Conflicting demands: high performance and power consumption

Errors in the power management functionality can have fatal consequences:
– Excessive power consumption
– Error in functionality
Power Aware Design Flow

• Power management/optimization techniques should be considered early in the design flow

• ESL offers much more opportunities for power optimization than RTL

• ESL features early availability of SW development and fast simulation speed

• Power-aware SystemC-based Virtual Prototypes (VPs)
VP-based Power Management

Validation of firmware-based power management is important
Power Aware Co-Simulation

- Cross-compile and run on VP
- Execute SW in FW/HW co-simulation
- Track and report power/performance characteristics

© Accellera Systems Initiative
Scenarios

- Missing corner-cases violating power/performance budgets
- Production-level SW might be unavailable yet
- Constraint-based workload description
- Specific power consumption profile
Validation of VP-based PM

- With power management
- SW
- FW-1
- Report
- FW-2
- Report
- SW
- All in full power mode

Power/performance budgets:
+20% power save
-50% performance

© Accellera Systems Initiative
Scenario to Software

Workload constraints
(specified in DSL built on top of Python)

Arith. IO

1. Constrained Random Generator

2. Application Generator

Scenario

Concrete Workload

Application (SW)
### Scenario to SW

1: \( A = \text{Select}(\lambda x: x.\text{pos} \leq 5) \)

2: \( \text{Ensure}(A, \lambda x: x.\text{type} == \text{InstrType.Arithmetic}) \)

3:

4: \( B = \text{Select}(A, \lambda x: x.\text{irq.scaler} \neq 0 \&\& x.\text{irq.scaler} \leq 0x50) \)

5: \( \text{Assert}(\text{Size}(B) == 2) \)

6:

7: \( \text{Exists}(\lambda x: x.\text{type} == \text{InstrType.Arithmetic} \&\& x.\text{arithmetic}.\text{num_instr} > 10000) \)

8: …

---

**Scenario**

\[
\text{int f3()} \{
    \star \text{APB_IRQ_SCALER_ADDR} = 0x40;
    \text{int acc} = 0;
    \text{for (int i=0; i<20000; ++i)}
        \text{acc} = \text{acc} \times i;
    \star \text{APB_IRQ_SCALER_ADDR} = 0;
    \text{return acc;}
\}
\]

---

**SW**

- **type = arithmetic**
  - num-instr = 20000
  - op-type = int-add
  - irq-scaler = 0
  - pos = 1

- **type = arithmetic**
  - num-instr = 10000
  - op-type = int-add
  - irq-scaler = 0x40
  - pos = 2

- **type = arithmetic**
  - num-instr = 20000
  - op-type = int-mult
  - irq-scaler = 0x40
  - pos = 3
  
  ...
Constrained Random Generator

Scenario

Concrete Workload

Symbolic IB list with N blocks

SMT Solver

try gen. next workload

no

Symbolic IB with N blocks

depend on type (arithmetic, IO, memory)

type = ???
num-instr = ???
op-type = ???
um-chars = ???
io-scaler = ???
irq-scaler = ???
pos = ???

yes

Concrete Workload

try gen. next workload

no

Symbolic IB with N blocks

SMT Solver

try gen. next workload

no

Symbolic IB list with N blocks

SMT Solver

try gen. next workload

no

Symbolic IB with N blocks

SMT Solver

try gen. next workload

no

Symbolic IB list with N blocks

SMT Solver

try gen. next workload

no

Symbolic IB with N blocks
Constrained Random Generator

<table>
<thead>
<tr>
<th>type-1 = ???</th>
<th>type-2 = ???</th>
</tr>
</thead>
<tbody>
<tr>
<td>num-instr-1 = ???</td>
<td>num-instr-2 = ???</td>
</tr>
<tr>
<td>op-type-1 = ???</td>
<td>op-type-2 = ???</td>
</tr>
<tr>
<td>num-chars-1 = ???</td>
<td>num-chars-2 = ???</td>
</tr>
<tr>
<td>io-scaler-1 = ???</td>
<td>io-scaler-2 = ???</td>
</tr>
<tr>
<td>irq-scaler-1 = ???</td>
<td>irq-scaler-2 = ???</td>
</tr>
<tr>
<td>pos-1 = ???</td>
<td>pos-2 = ???</td>
</tr>
</tbody>
</table>

1: \( \exists x. x.\text{type} == \text{Arith.} \)

2: \( \exists x. x.\text{type} == \text{Mem.} \)

3: \( \exists x. x.\text{type} == \text{IO} \)

\[
\begin{align*}
\text{(type}_1 &= \text{Arith.} \vee \text{type}_2 = \text{Arith.}) \\
\land \\
\text{(type}_1 &= \text{Mem.} \vee \text{type}_2 = \text{Mem.}) \\
\land \\
\text{(type}_1 &= \text{IO} \vee \text{type}_2 = \text{IO})
\end{align*}
\]

Ask SMT Solver

No solution (unsat)
Constrained Random Generator

Scenario

Concrete Workload

Symbolic IB list with N blocks

SMT Solver

try gen. next workload

Symbolic IB

valid fields depend on type (arithmetic, IO, memory)

CRG

no, add constraints to block last workload

enough workloads?

no

yes

num-instr = ???

op-type = ???

num-chars = ???

io-scaler = ???

irq-scaler = ???

pos = ???

Done

yes

SMT Solver

type = ???

try gen. next workload

enough blocks?

no

yes

enough blocks?
Case Study: SoCRocket

- Based on Aeroflex Gaisler GRLib (RTL)
- AHB/APB: TLM-based AMBA-bus
- Memory Controller
- Various Peripherals

“Transaction-Level Modeling Framework for Space Applications”
SoCRocket Power Modelling

• Every component has three different power values:
  – Static,
  – Internal
  – Switching

• Static and internal power is application independent
  – Only depends on active power state

• Switching power depends on the components activity
  – Needs to be traced periodically
  – Use SystemC thread with periodic wait delay
Power Management

- So far: VP provides power states and allows changing
- Now: Firmware-based power control
- HW interface unit (attached to bus) forwards request

```c
volatile uint *pm_control = (uint *)0xB0000000;
//...
pm_control[0] = PM_ID_LEON3 |
(PM_STATE_LEON3_PS2 << 8);
```
typedef int8_t PM_STATE;

typedef struct {
    PM_STATE pm_state;
    _Bool wait_for_io;
    int num_rtm;
} leon3_stat_t;

typedef struct {
    uint32_t scaler;
    uint32_t local_num_recv;
    uint32_t activity;
    _Bool wait_for_io;
    PM_STATE pm_state;
} uart_stat_t;
DEMO

SoC Rocket

Sensor (ahbin)

LEON3

MCTRL

SDRAM

ROM

IRQMP

Timer

UART

AHB

APB

Timer

#include hello_world.h

int main()

printf("Hello world\n");

Return 0;
Experiments

• Consider five scenarios:
  1. High CPU load
  2. Interrupt intensive workload
  3. Alternating instruction blocks
  4. Memory and IO intensive
  5. Many small tasks

• 8,000,000 instructions executed on SoCRocket in avg. per workload
• 15 minutes wall time in avg. per scenario (Linux with 2,4 GHz Intel)
## Experiments

### Power Cons. (uJ)

<table>
<thead>
<tr>
<th>Power Cons. (uJ)</th>
<th>Without PM</th>
<th>With PM</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) CPU Load</td>
<td>254969</td>
<td>94405</td>
<td>-62.97%</td>
</tr>
<tr>
<td>2) Interrupts</td>
<td>161274</td>
<td>129345</td>
<td>-19.80%</td>
</tr>
<tr>
<td>3) Alternating</td>
<td>397375</td>
<td>210988</td>
<td>-46.90%</td>
</tr>
<tr>
<td>4) Memory/IO</td>
<td>1004561</td>
<td>278397</td>
<td>-72.29%</td>
</tr>
<tr>
<td>5) Small Tasks</td>
<td>270656</td>
<td>208755</td>
<td>-22.87%</td>
</tr>
</tbody>
</table>

### Sim. Time (sec.)

<table>
<thead>
<tr>
<th>Sim. Time (sec.)</th>
<th>Without PM</th>
<th>With PM</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) CPU Load</td>
<td>2.03</td>
<td>2.42</td>
<td>+19.21%</td>
</tr>
<tr>
<td>2) Interrupts</td>
<td>1.09</td>
<td>1.68</td>
<td>+54.13%</td>
</tr>
<tr>
<td>3) Alternating</td>
<td>2.77</td>
<td>3.74</td>
<td>+35.02%</td>
</tr>
<tr>
<td>4) Memory/IO</td>
<td>7.63</td>
<td>10.88</td>
<td>+42.60%</td>
</tr>
<tr>
<td>5) Small Tasks</td>
<td>1.82</td>
<td>2.88</td>
<td>+58.24%</td>
</tr>
</tbody>
</table>

Save in power consumption

Loss in performance
Conclusion

• Approach for early validation of firmware-based power management

• Using power aware SystemC-based Virtual Prototypes

• Workload generation using constrained random techniques

• Check that power/performance budgets are satisfied
Next Steps

• Incorporate Coverage Metrics
  – Cross coverage: power configuration / source code metrics
  – Add feedback loop

• Extend Constraint Language
  – Simplifies symbolic description
  – Insert power management calls (e.g. need RTM)

• Improve Constrained Random Solutions
  – Currently the last solution is blocked
  – “intelligent” guessing of solutions, use SMT solver to complete partial solutions
References


• http://www.systemc-verification.org/

Driver Generation and Optimization of the SW/HW Interface

Martin Dittrich (TU Munich)
Motivation

- **Small** MCUs such as ARM Cortex M0 used in many applications
- **Advantages:**
  - Flexibility by programmability
  - Late design patches / Firmware updates possible
- **Challenges:**
  - Very resource constrained
  - AREA, power, timing overheads vs. a fixed-HW implementation
  - Main cost factor is embedded memory
  - Complex task to design memory-footprint/performance optimized FW
State of the Art

• Program Memory Footprint
  – HW Code Compression
    • Huffmann Codes
    • Dictionary-based Compression
    • Bit Masks Dictionaries
  – Link-time Optimization
  – Binary Rewriting

• Data Memory Footprint*
  – Bit Packaging
  – Pointer Tables
  – New calculation of values instead of storing values
  – Delta Coding

*Small Memory Software Patterns for System with Limited Memory
C. Weir, J. Noble 2000
Goals

• Automatic Generation of optimized MCU Firmware
• Optimization of the HW/SW Interface of the MCU for better Firmware in terms of Performance and Memory Requirements
MCU HW/SW Interface + Driver Generator

MCU HW/SW Interface + Driver Spec (Pseudo C)

Parser

Driver Model (CDFG)

HW/SW Interface Model (IP-Xact)

HAL CodeGen

Driver Units using HAL (C)

HW/SW IF Optimizer

Memory Footprint

# Static Accesses

Static Analysis

Firmware Drivers (ASM)

Cross-Compiler

© Accellera Systems Initiative
Driver Spec. Parameters (Pseudo-C)

• Simple Example:
  – Software Input/Output Parameters of Driver Functions
  – Hardware Input/Output Parameters (Hardware Device Params)

**Software I/O Parameters**

**Inputs:**
- `uint32_t id`
- `enum {modA, modB} mode`
- `uint32_t aNum`
- `bool rStat`

**Outputs:**
- `uint16_t out[4]`

**Hardware I/O Parameters**

**Inputs:**
- `uint9_t inputLength_i;`
- `uint32_t seed_i;`
- `uint32_t poly_i[2];`
- `uint1_t chkT_i[4];`
- `uint1_t chkA_i[3];`

**Outputs:**
- `uint32_t out_o[2];`
Driver Spec. Behavior (Pseudo-C)

Driver Unit

Software I/O Parameters
Hardware I/O Parameters
Behavior

Behavior: program_device

inputLength_i =

if (rStat) {
    (mode == modB)
    ? 100 : 500;
    seed_i = id;
    poly_i[0] = 35263098;
    poly_i[1] = 10031374;
    chkT_i[0] = 1;
    chkT_i[1] = 1;
    chkT_i[2] = 1;
    chkT_i[3] = 1;
}
else {
    chkA_i[0] = aNum;
    chkA_i[1] = 0;
    chkA_i[2] = 0;
    chkA_i[0] = 1;
    chkA_i[1] = 1;
    chkA_i[2] = 1;
}

© Accellera Systems Initiative
Control-data-flow-graph (CDFG) analysis

Behavior: program_device

```plaintext
inputLength_i = (mode == modB) ? 100 : 500;
seed_i = id;
poly_i[0] = 35263098;
poly_i[1] = 10031374;
chkT_i[0] = 1;
chkT_i[1] = 1;
chkT_i[2] = 1;
chkT_i[3] = 1;

if (rStat) {
  chkA_i[0] = aNum;
  chkA_i[1] = 0;
} else {
  chkA_i[0] = 1;
  chkA_i[1] = 1;
  chkA_i[2] = 1;
}
```

- Control flow dependencies
- No data dependencies
- No Write-before-read dependencies
Step 1: Generation of Register Interface

Hardware I/O Parameters

**Inputs:**
- `uint9_t inputLength_i;`
- `uint32_t seed_i;`
- `uint32_t poly_i[2];`
- `uint1_t chkT_i[4];`
- `uint1_t chkA_i[3];`

**Outputs:**
- `uint32_t out_o[2];`
Register-Compatibility Graph

program_device

seed_i \( \rightarrow \) id \( \rightarrow \) REG1

poly_i[0] \( \rightarrow \) 35263098 \( \rightarrow \) REG3

cpoly_i[1] \( \rightarrow \) 10031374 \( \rightarrow \) REG4

inputLength \( \rightarrow \) 500

chkA_i[0] \( \rightarrow \) 1

chkA_i[1] \( \rightarrow \) 1

chkA_i[2] \( \rightarrow \) 1

chkT_i[0] \( \rightarrow \) 1

chkT_i[0] \( \rightarrow \) 1

chkT_i[0] \( \rightarrow \) 1

chkT_i[0] \( \rightarrow \) 1

Can be written/read within one Load/Store Access
Otherwise Read-Modify-Store
**Behavior: program_device**

```c
inputLength_i = (mode == modB) ? 100 : 500;
seed_i = id;
poly_i[0] = 35263098;
poly_i[1] = 10031374;
chkT_i[0] = 1;
chkT_i[1] = 1;
chkT_i[2] = 1;
chkT_i[3] = 1;

if (rStat) {
    chkA_i[0] = aNum;
    chkA_i[1] = 0;
    chkA_i[2] = 0;
} else {
    chkA_i[0] = 1;
    chkA_i[1] = 1;
    chkA_i[2] = 1;
}
```

```c
void program_device1(enum mode, uint32_t id, bool rstat, uint32_t aNum) {
    if (mode == modB) set_inputLength_i(100);
    else set_inputLength_i(500);
    set_seed_i(id);
    set_poly_i_0(35263098)
    set_poly_i_1(10031374)
    set_chkT_i_0(1);
    set_chkT_i_1(1);
    set_chkT_i_2(1);
    set_chkT_i_3(1);
    if (rStat) {
        set_chkA_i_0(aNum);
        set_chkA_i_1(0);
        set_chkA_i_2(0);
    } else {
        set_chkA_i_0(1);
        set_chkA_i_1(1);
        set_chkA_i_2(1);
    }
}
```

<table>
<thead>
<tr>
<th>Performance</th>
<th># Read</th>
<th># Write</th>
<th>#Read-Modify Write</th>
<th>#HW Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>19</td>
</tr>
</tbody>
</table>
Step 2: Firmware-Generation

void program_device1(enum mode, uint32_t id, bool rStat, uint32_t aNum) {
    if (mode == modB) set_inputLength_i(100);
    else set_inputLength_i(500);
    set_seed_i(id);
    set_poly_i_0(35263098);
    set_poly_i_1(10031374);
    set_chkT_i_0_3(1,1,1,1);
    if (rStat) {
        set_chkA_i_0_2(aNum,0,0);
    } else {
        set_chkA_i_0_2(1,1,1);
    }
}

Performance

<table>
<thead>
<tr>
<th># Read</th>
<th># Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

# Read-Modify Write 3

#HW Accesses 9
Step 2: Firmware-Generation

void program_device1(enum mode, uint32_t id, bool rstat, uint32_t aNum) {
    set_seed_i(id);
    set_poly_i_0(35263098);
    set_poly_i_1(10031374);

    if (mode == modB && rStat) {
        set_inputLength_chkT_chkA(100,1,1,1,1,aNum,0,0);
    }
    if (mode == modA && rStat) {
        set_inputLength_chkT_chkA(500,1,1,1,1,aNum,0,0);
    }
    if (mode == modB && ! rStat) {
        set_inputLength_chkT_chkA(100,1,1,1,1,1,1,1);
    }
    if (mode == modA && ! rStat) {
        set_inputLength_chkT_chkA(500,1,1,1,1,1,1,1);
    }
}

Performance

# Read 0
# Write 4
#Read-Modify Write 0
#HW Accesses 4
Control-data-flow-graph (CDFG) analysis

Behavior: program_device

```
inputLength_i = (mode == modeB) ? 100 : 500;
seed_i = id;
poly_i[0] = 35263098;
poly_i[1] = 10031374;
chkT_i[0] = 1;
chkT_i[1] = 1;
chkT_i[2] = 1;
chkT_i[3] = 1;
if (rStat) {
    chkA_i[0] = aNum;
    chkA_i[1] = 0;
    chkA_i[2] = 0;
} else {
    chkA_i[0] = 1;
    chkA_i[1] = 1;
    chkA_i[2] = 1;
}
```

//Barrier

```
chkT_i[0] = 1;
chkT_i[1] = 1;
chkT_i[2] = 1;
chkA_i[0] = aNum;
chkA_i[1] = 1;
chkA_i[2] = 1;
```
Step 2: Firmware-Generation

Behavior: program_device

inputLength_i = (mode == modB) ? 100 : 500;
seed_i = id;
poly_i[0] = 35263098;
poly_i[1] = 10031374;

//Barrier

chkT_i[0] = 1;
chkT_i[1] = 1;
chkT_i[2] = 1;

if (mode == modB) set_inputLength_i(100);
else set_inputLength_i(500);
set_seed_i(id);
set_poly_i(0,35263098)
set_poly_i(1,10031374)

if (rStat) {
    set_chkT_chkA(1,1,1,1,aNum,0,0);
} else {
    set_chkT_chkA(1,1,1,1,1,1,1);
}

Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td># Read</td>
<td>0</td>
</tr>
<tr>
<td># Write</td>
<td>3</td>
</tr>
<tr>
<td># Read-Modify Write</td>
<td>2</td>
</tr>
<tr>
<td>#HW Accesses</td>
<td>7</td>
</tr>
</tbody>
</table>
Conclusion

• Automatic FW Code-Generation
• Optimization of the HW/SW Interface

• Big research question:
  – What is the best model and design abstraction for specifying the FW?
  – C is already very compact
Tutorial Conclusion

• Automatic firmware design for application-specific electronic systems is very challenging

• Three solutions
  – Context-Sensitive Source-Level Timing Simulation
  – Validation of Firmware-based Powermanagement using CRV
  – Driver Generation and Optimization of the SW/HW Interface