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

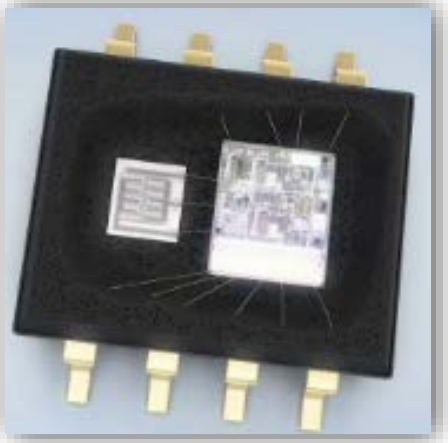
Daniel Große (Univ. Bremen / DFKI)
Joscha Benz (Univ. of Tuebingen)
Vladimir Herdt (Univ. of Bremen)
Martin Dittrich (TU Munich)

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

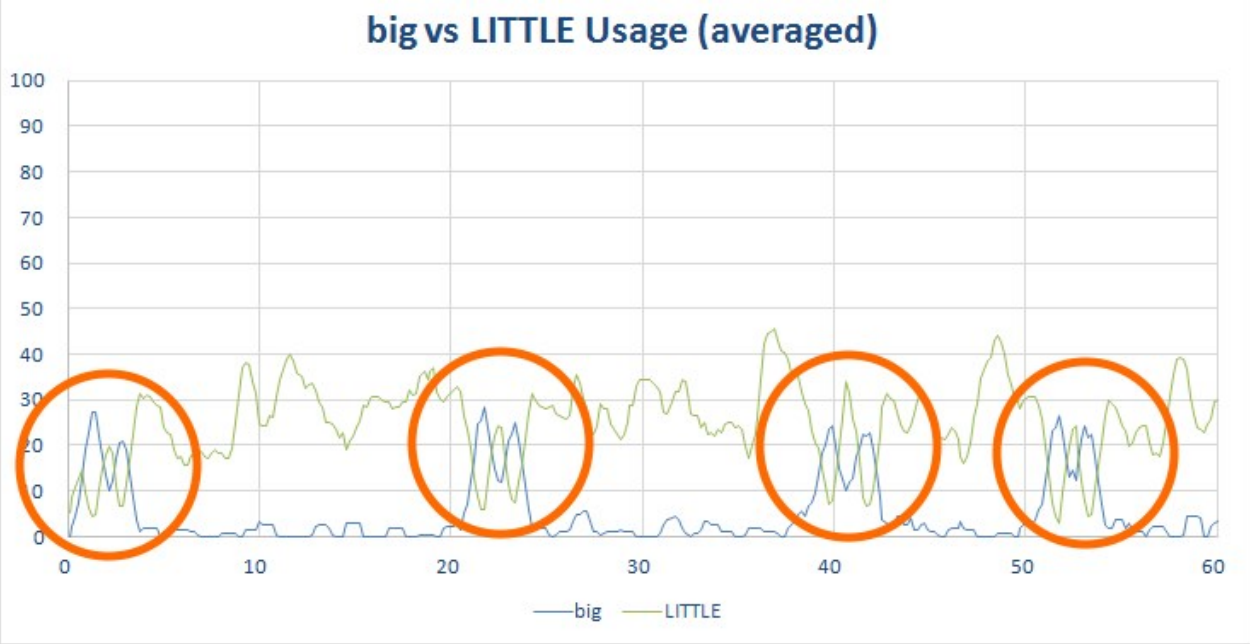


Src: Bosch

- 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

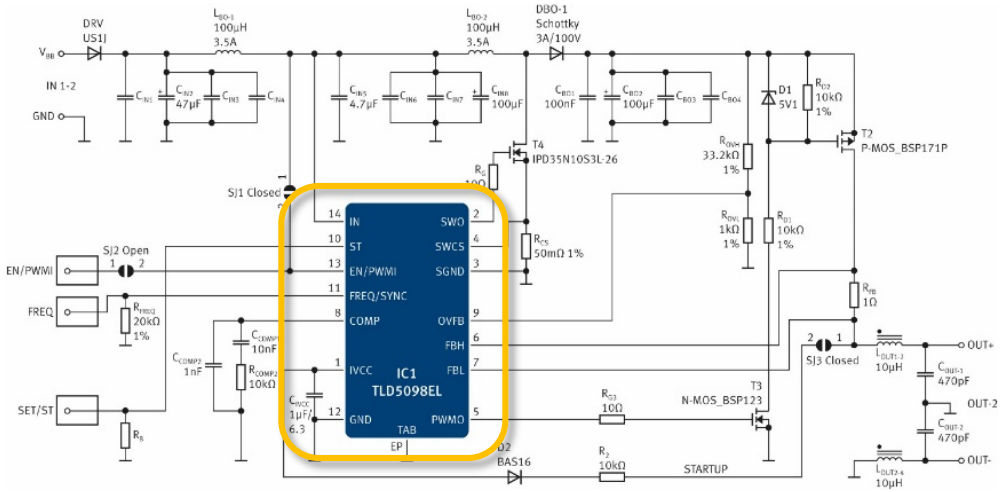


Src: www.androidauthority.com

- 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



Src: IFX

Examples:

- Power Converter
- LED Control
- Sensor SIP

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

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

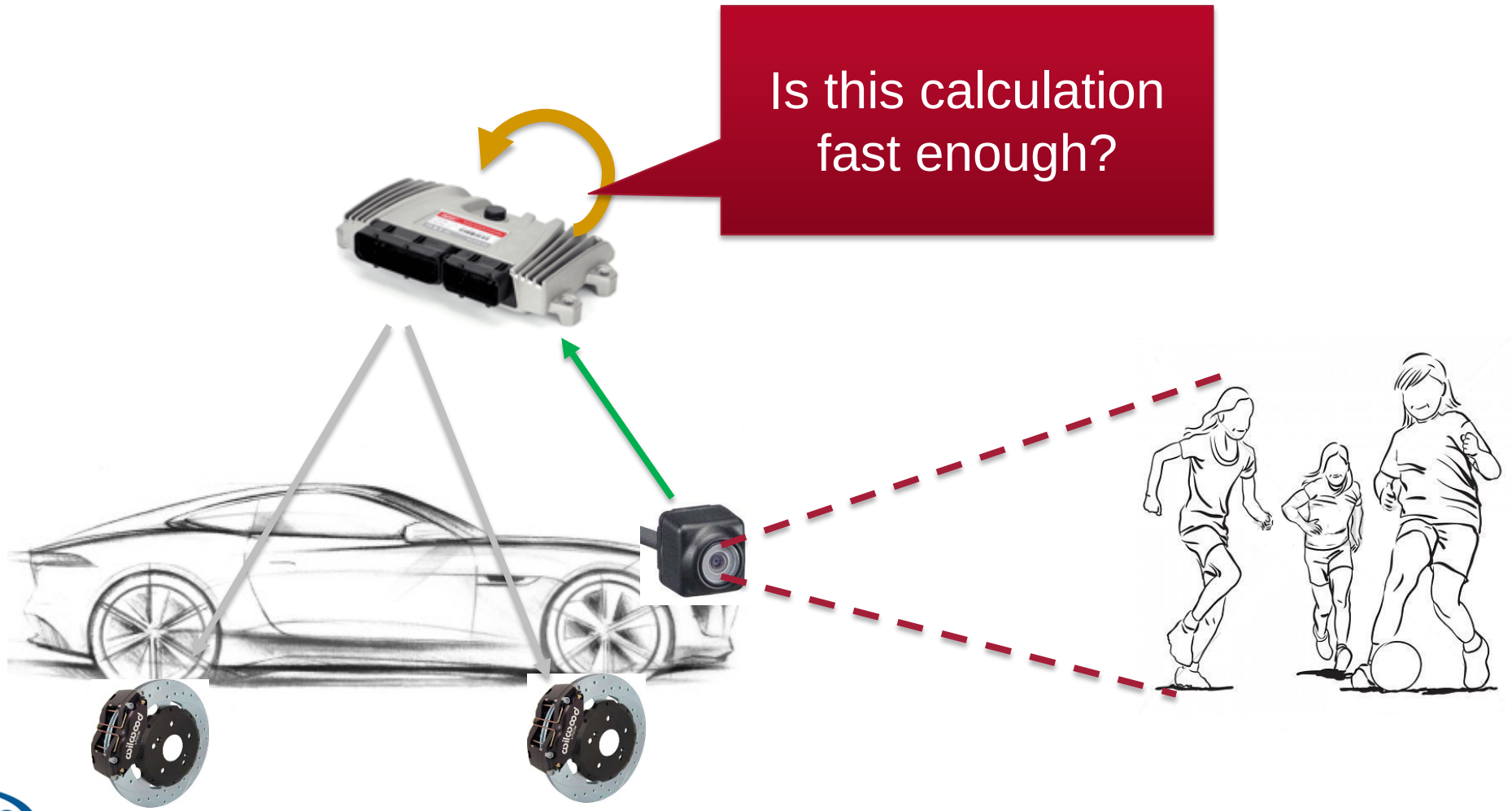
Context-Sensitive Source-Level Timing Simulation

Joscha Benz (Univ. of Tuebingen)

accellera SYSTEMS INITIATIVE

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Motivation



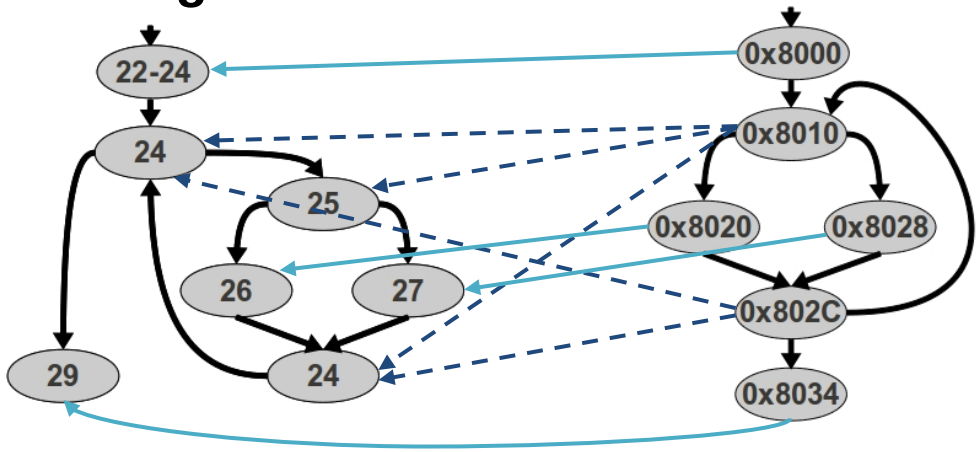
Agenda

- Source-Level Timing Simulation
  - Principle & Framework
  - Instrumentation
- Loop Acceleration
  - Results
- Conclusion & Future Work

Source-Level Timing Simulation (SLTS)

1. Source/Binary Code

2. Matching



3. Instrumentation

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

Annotated C/C++ Source Code

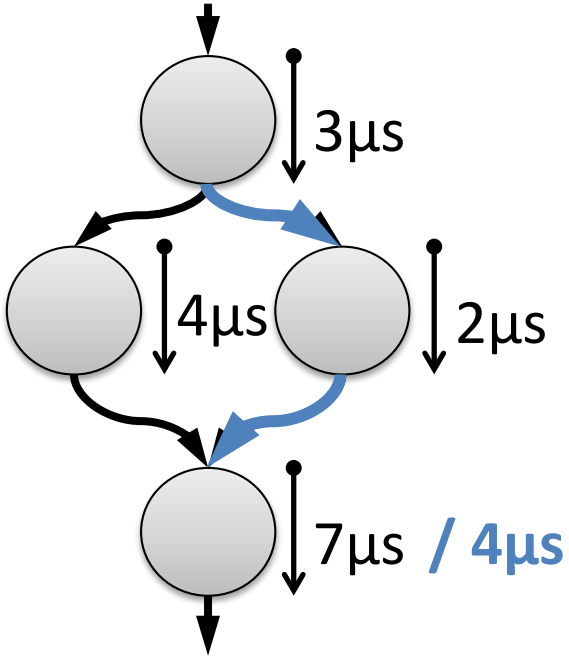
```
void bb (int address)
{
...
      if (lastBlock == 0x8000
            && nextBlock == 0x8010)
            delay (0x8000, 0x8010);
...
      lastBlock = nextBlock;
}
```

Path Simulation Code

Path simulation code + Timing

Basic Block Timing – How to determine?

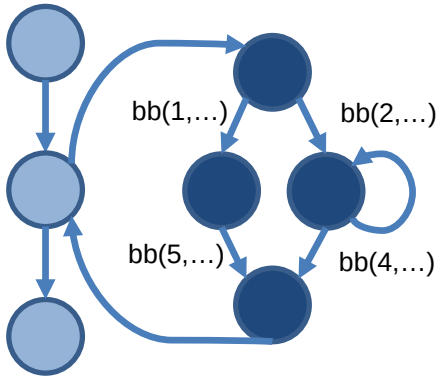
- By annotating statically analyzed timings
- By annotating dynamically measured timings



Timing is context-dependent!

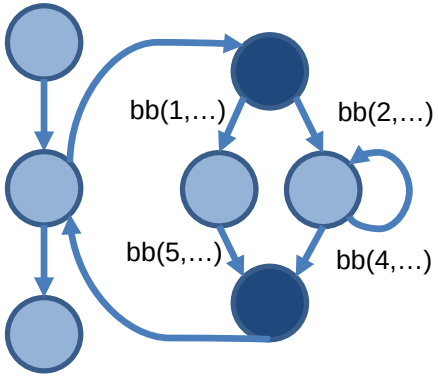
Instrumentation

Complete Instrumentation



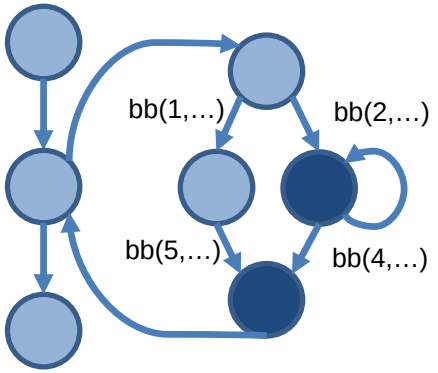
- Precise ✓
- Slower

Partial Instrumentation I



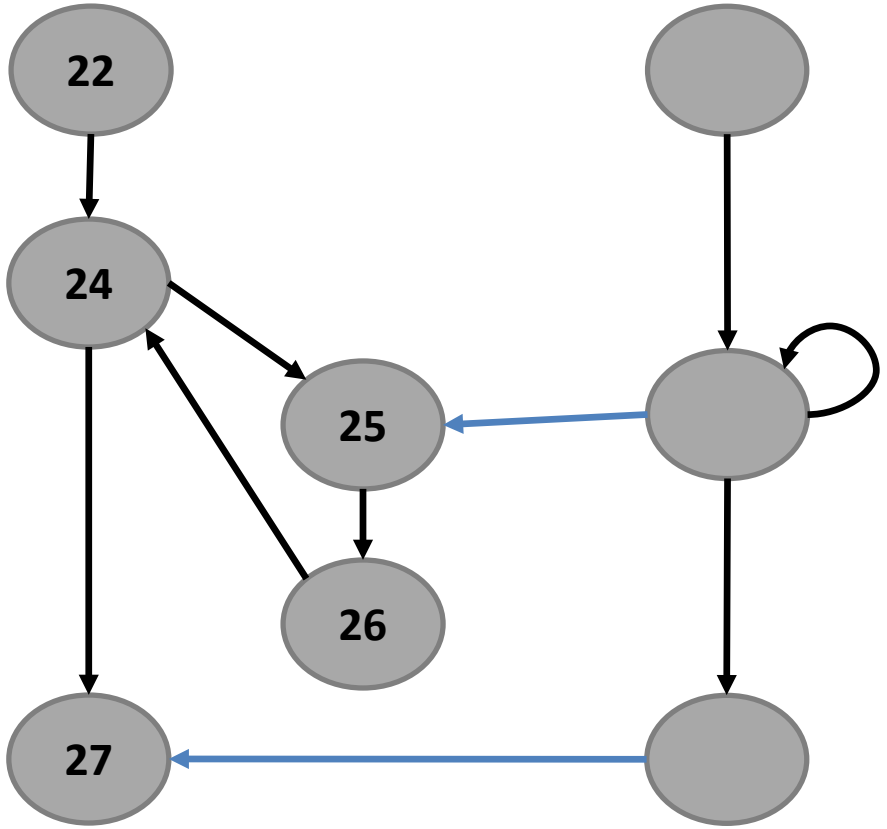
- Not precise
- Fast ✓

Partial Instrumentation II



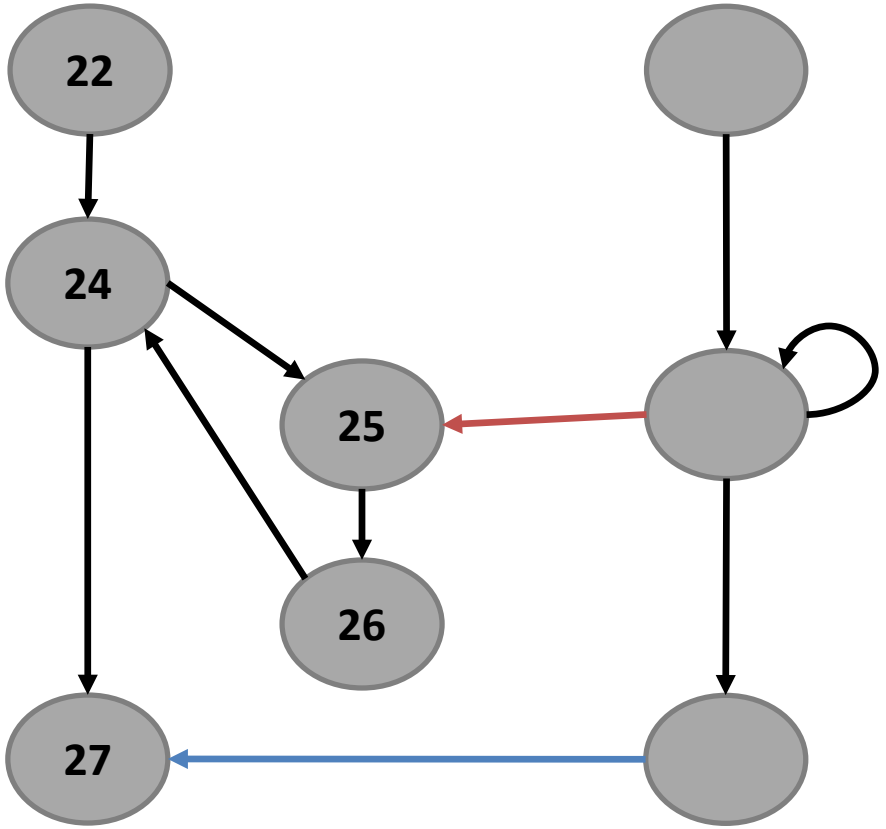
- Precise ✓
- Fast ✓

Instrumentation




```
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

```
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;
}
```

```
int c = 200;


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

simulate_bb_s_27();
```

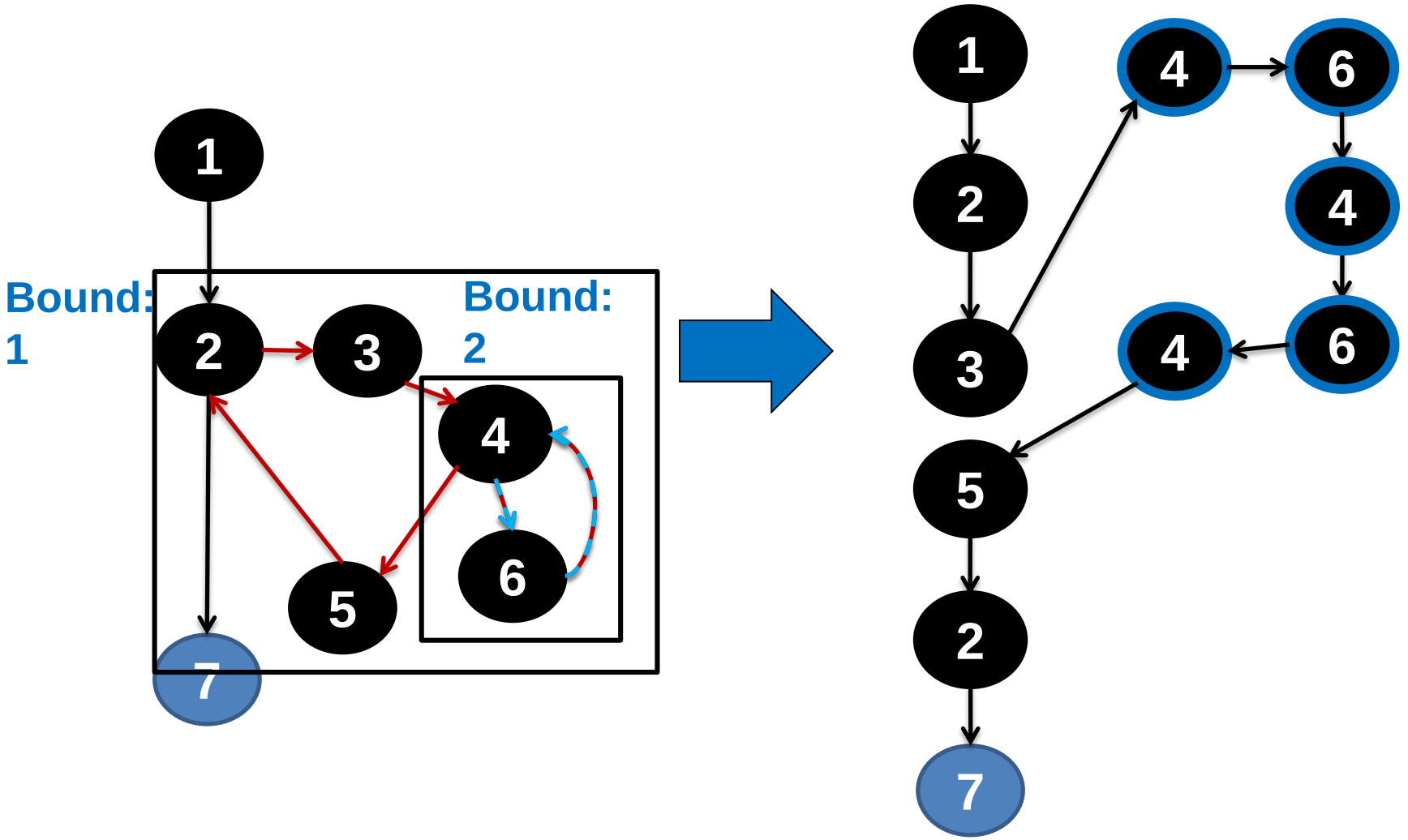
Instrumentation

- Partial instrumentation
  - Already very fast
    - Context-Sensitive: 2019 MIPS average throughput [1]

  →Can we go further?

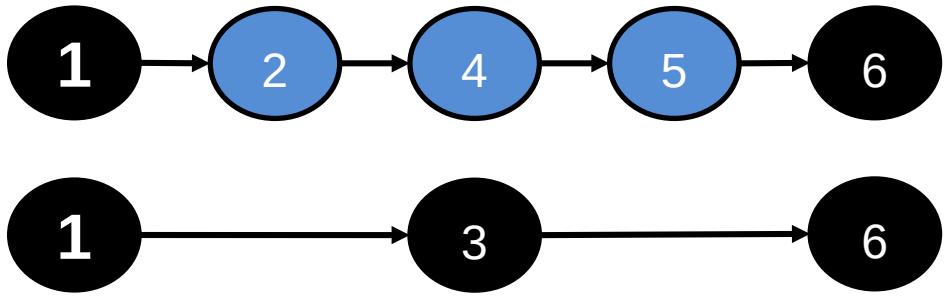
- Loop Acceleration
  - Much simulation time consumed
  - Especially for simple Loops

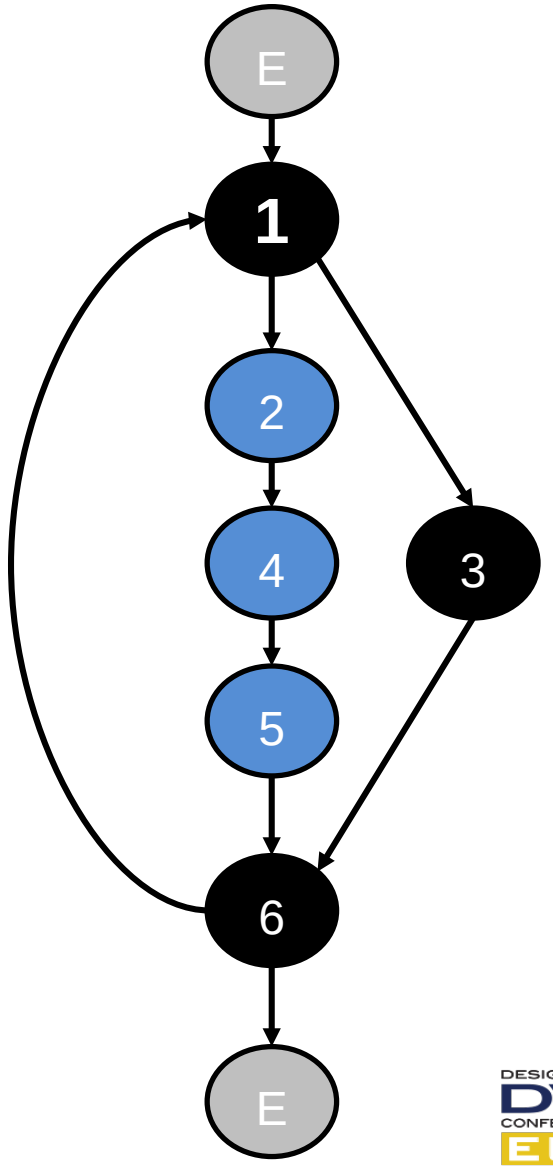
Loop Acceleration in SLTS



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

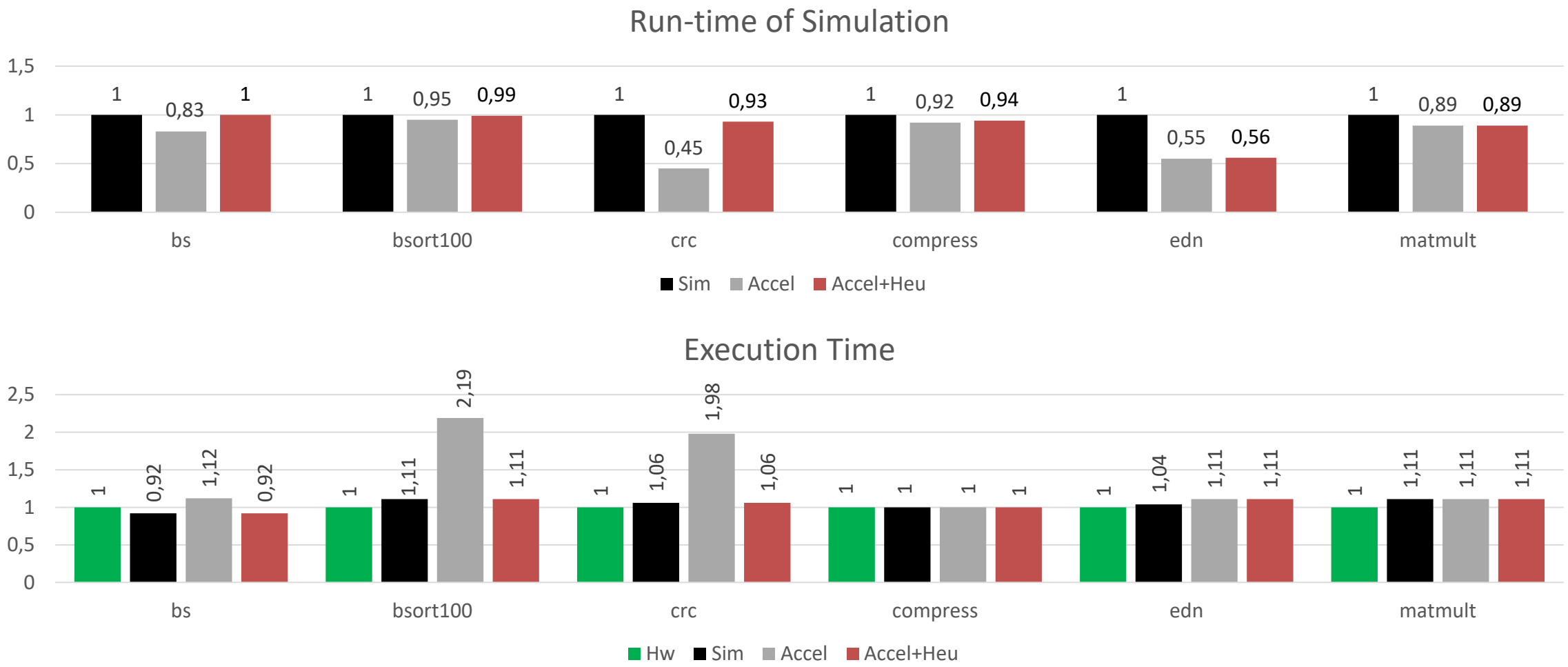
- Calculate Expected Inaccuracy
  - Loop bounds: $\frac{L.upper - L.lower}{L.upper} \cdot 100$
  - Loop paths: $\frac{\max(len(path)) - \min(len(path))}{\max(len(path))} \cdot 100$

- 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



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

1. S. Ottlik et al., “Context-Sensitive Timing Automata for Fast Source Level Simulation”, 2017 Design Automation & Test in Europe (DATE)
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

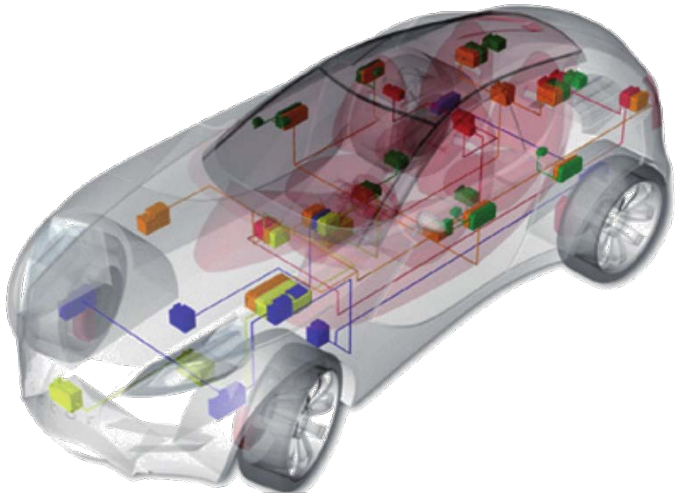
Vladimir Herdt (Univ. of Bremen)

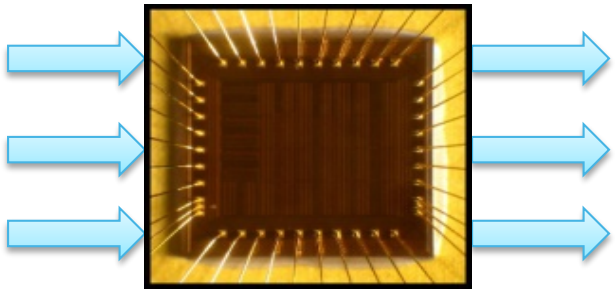
accellera SYSTEMS INITIATIVE

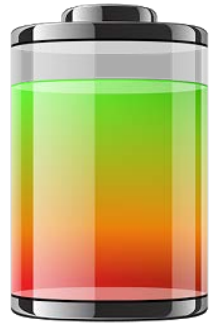
2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

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:

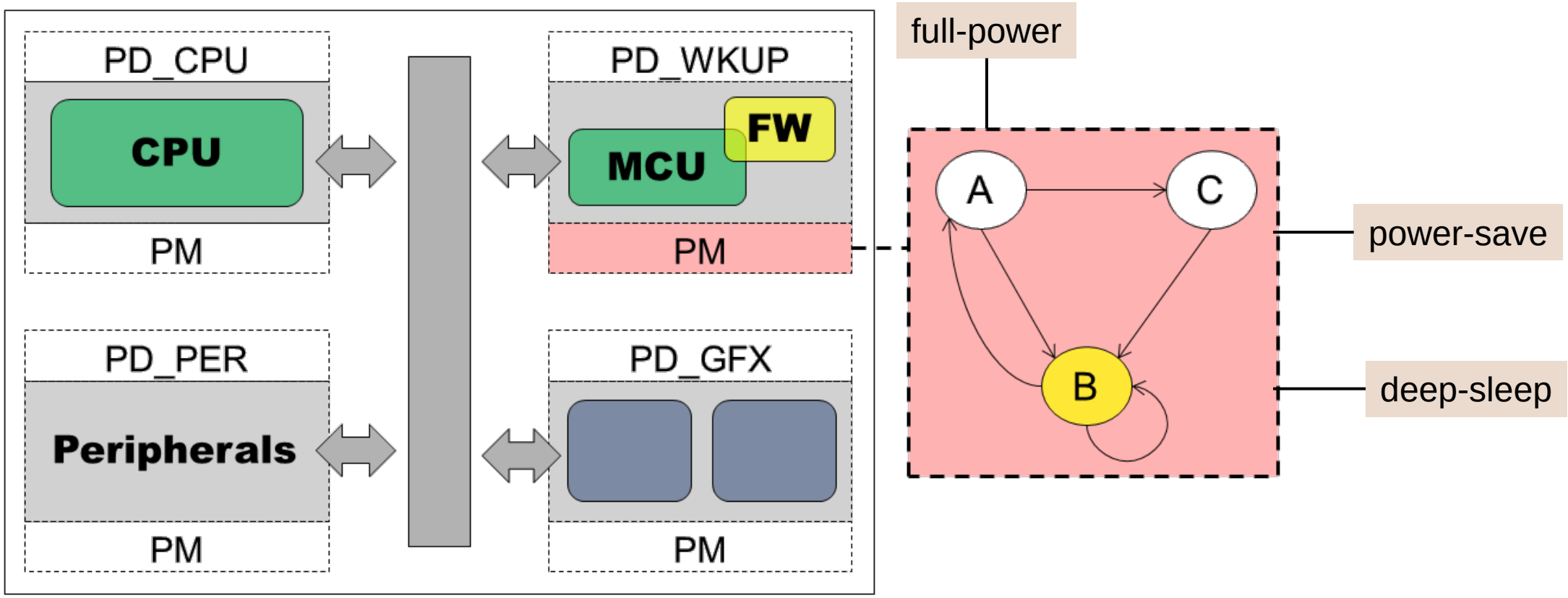
- Excessive power consumption
- Error in functionality

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)

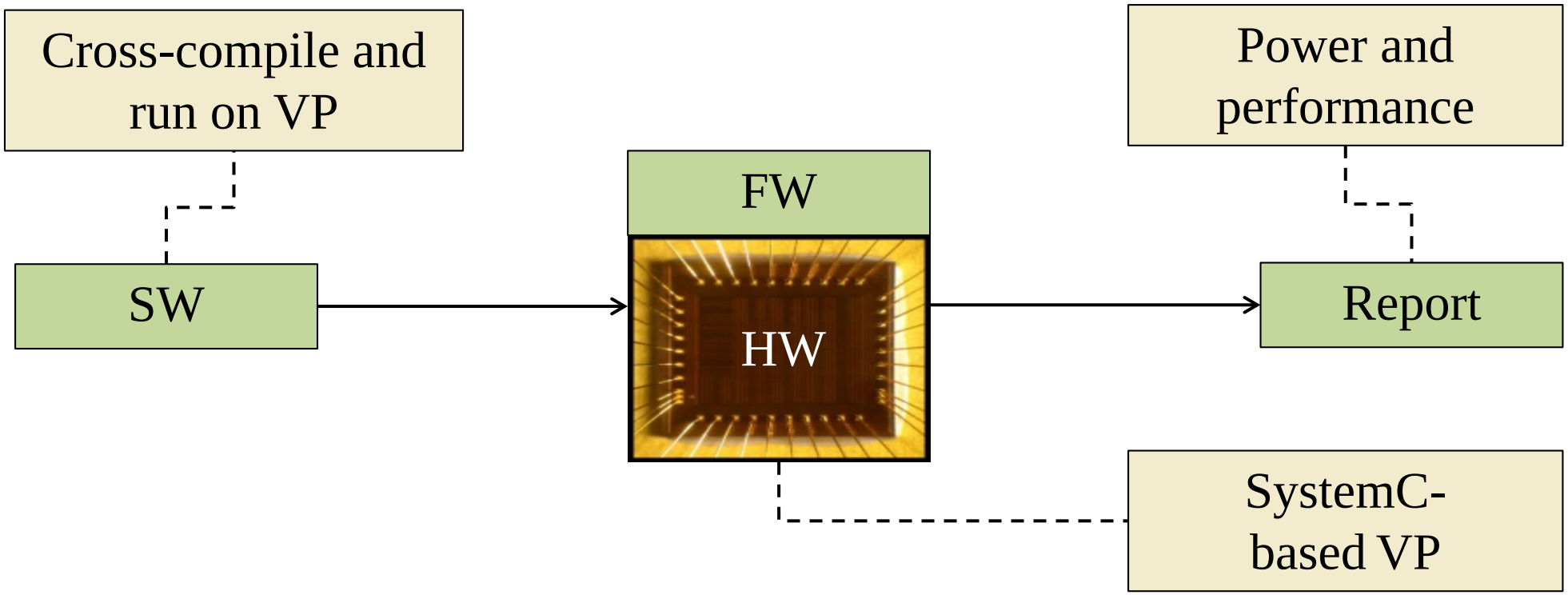
www.systemc.org

VP-based Power Management



Validation of firmware-based power management is important

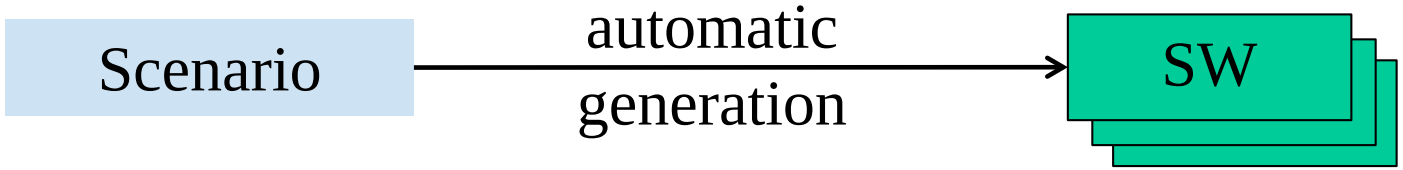
Power Aware Co-Simulation



- Execute SW in FW/HW co-simulation
- Track and report power/performance characteristics

Scenarios

- Missing corner-cases violating power/performance budgets
- Production-level SW might be unavailable yet



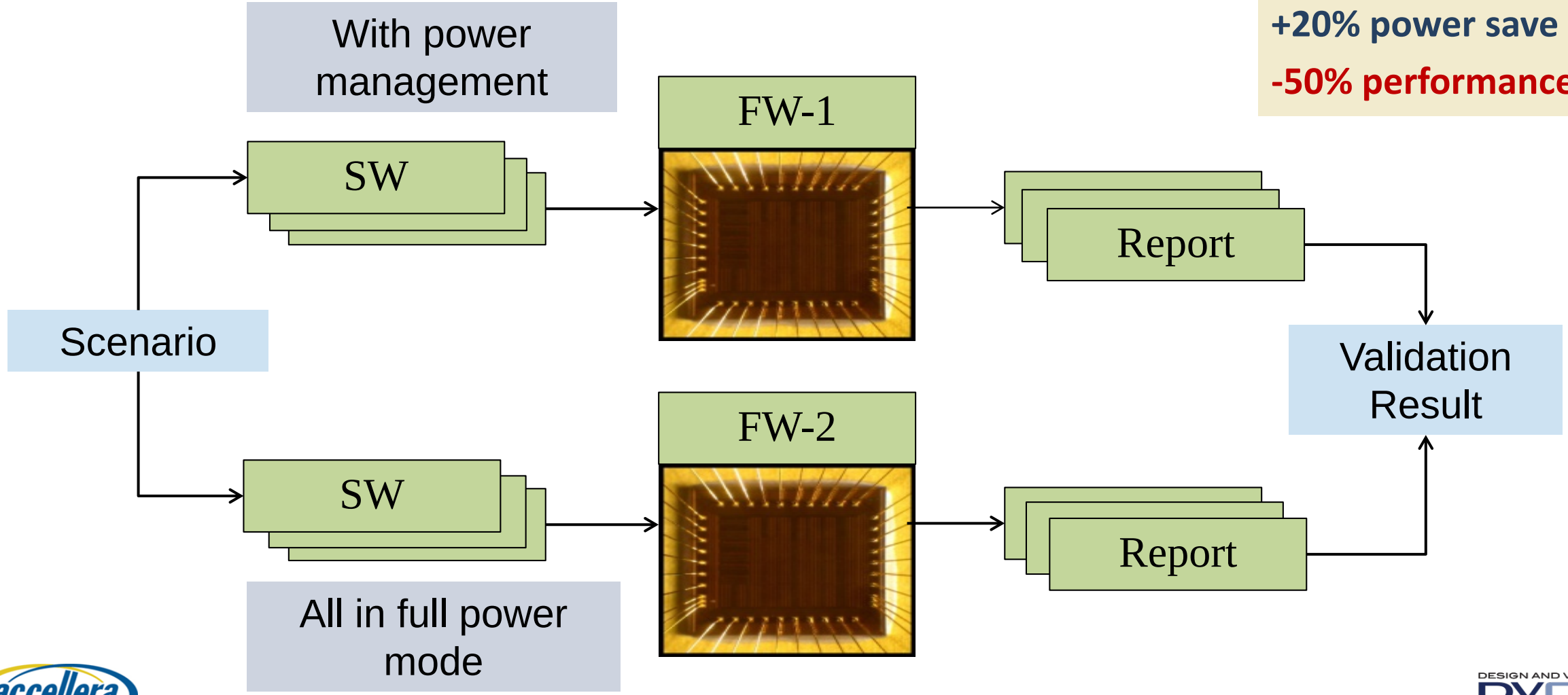
- Constraint-based workload description
- Specific power consumption profile

Validation of VP-based PM

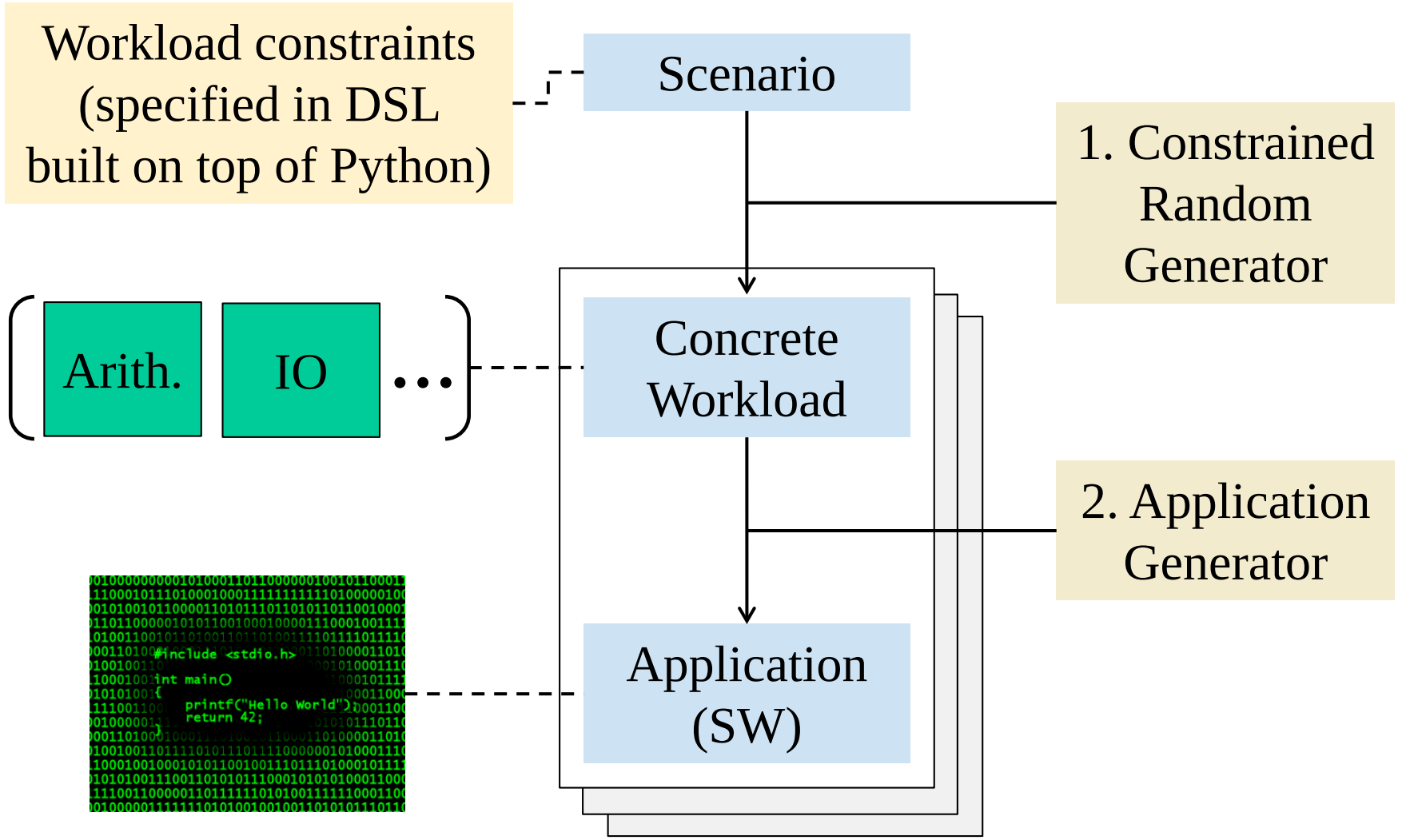
Power/performance budgets:

+20% power save

-50% performance



Scenario to Software



Scenario to SW

```
1: A = Select(lambda x: x.pos <= 5)
2: Ensure(A, lambda x: x.type == InstrType.Arithmetic)
3:
4: B = Select(A, lambda x: x.irq.scaler != 0 && x.irq.scaler <= 0x50)
5: Assert(Size(B) == 2)
6:
7: Exists(lambda x: x.type == InstrType.Arithmetic
                    && x.arithmetic.num_instr > 10000)
8: …
```

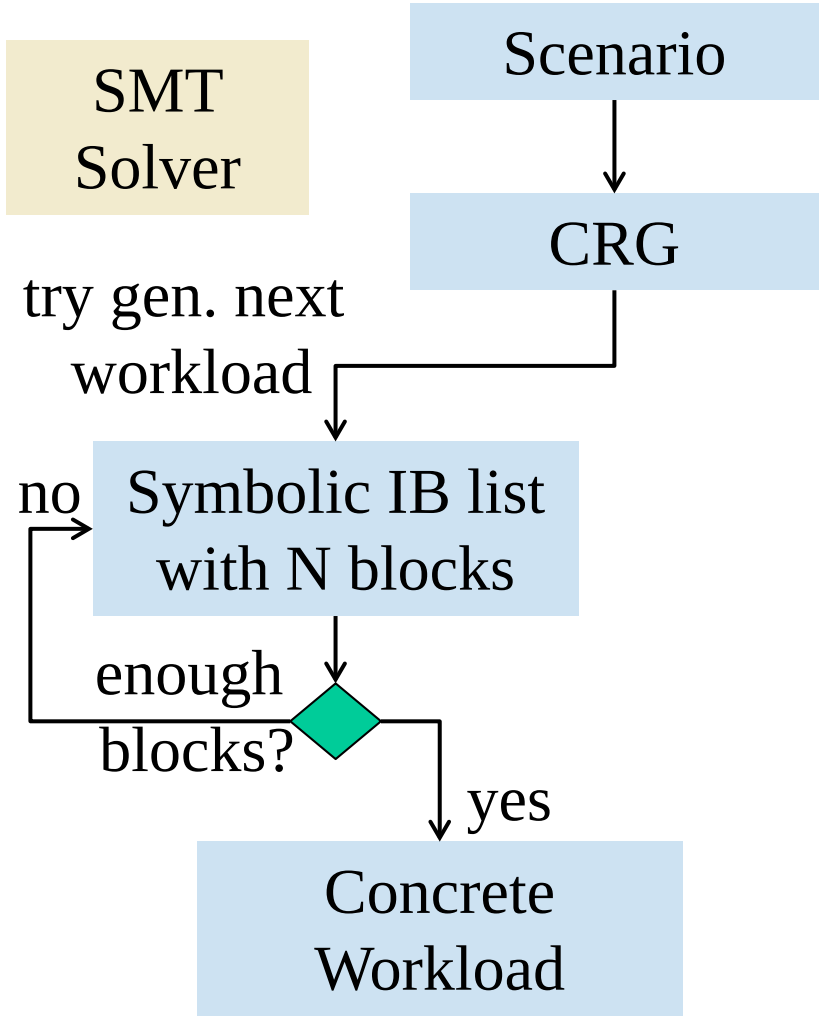
Scenario

```
int f3() {
    *APB_IRQ_SCALER_ADDR = 0x40;
    int acc = 0;
    for (int i=0; i<20000; ++i)
        acc = acc * i;
    *APB_IRQ_SCALER_ADDR = 0;
    return acc;  }
```

SW

Workload
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



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

Symbolic IB

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

Constrained Random Generator

type-1 = ???
num-instr-1 = ???
op-type-1 = ???
num-chars-1 = ???
io-scaler-1 = ???
irq-scaler-1 = ???
pos-1 = ???

type-2 = ???
num-instr-2 = ???
op-type-2 = ???
num-chars-2 = ???
io-scaler-2 = ???
irq-scaler-2 = ???
pos-2 = ???

No solution (unsat)

Ask SMT Solver

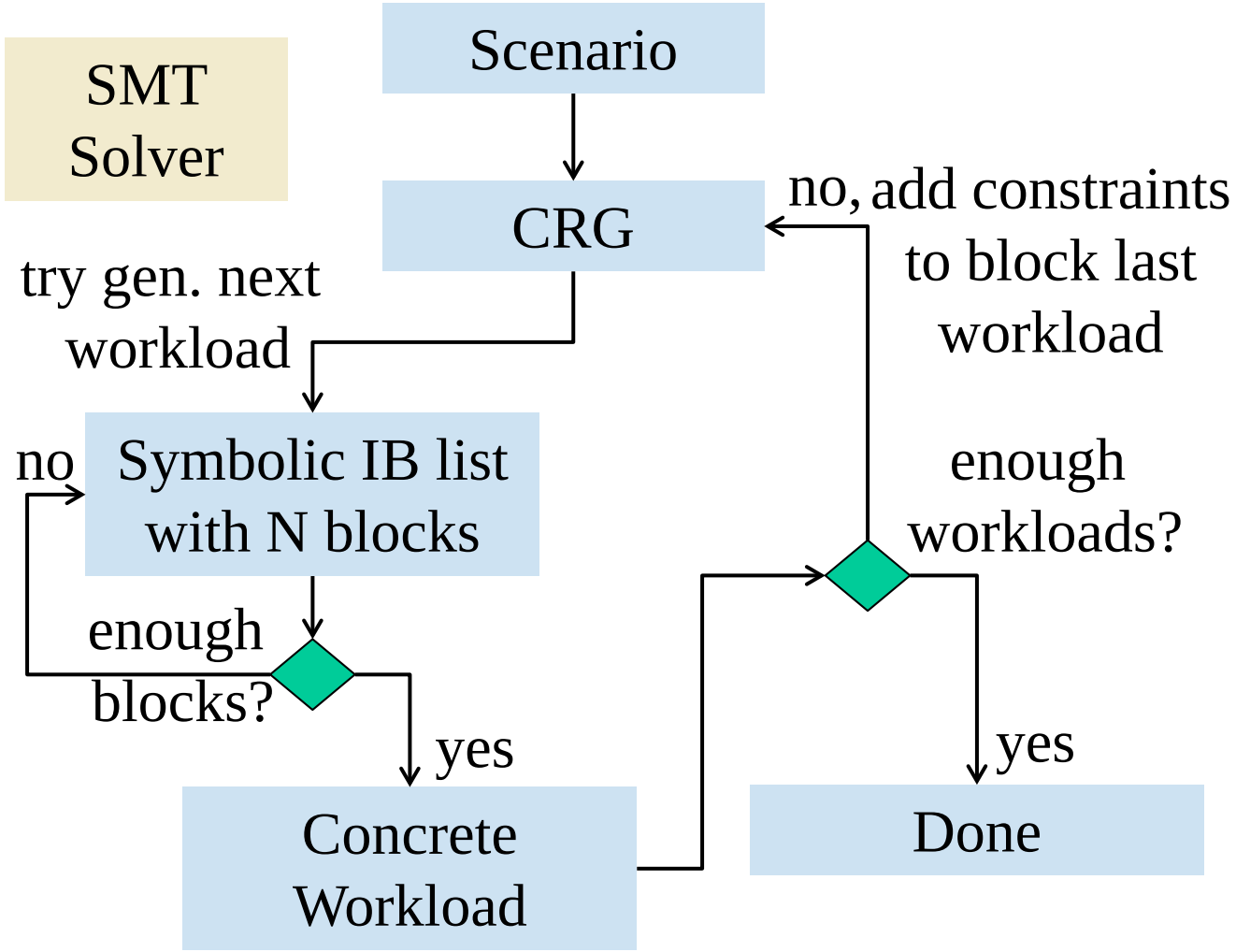
1: $\boldsymbol{Exists}(lambda\ x{:}\ x.type == Arith.)$

2: $\boldsymbol{Exists}(lambda\ x{:}\ x.type == Mem.)$

3: $\boldsymbol{Exists}(lambda\ x{:}\ x.type == IO)$

$$(type_1 = Arith. \vee type_2 = Arith.) \wedge (type_1 = Mem. \vee type_2 = Mem.) \wedge (type_1 = IO \vee type_2 = IO)$$

Constrained Random Generator



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

Symbolic IB

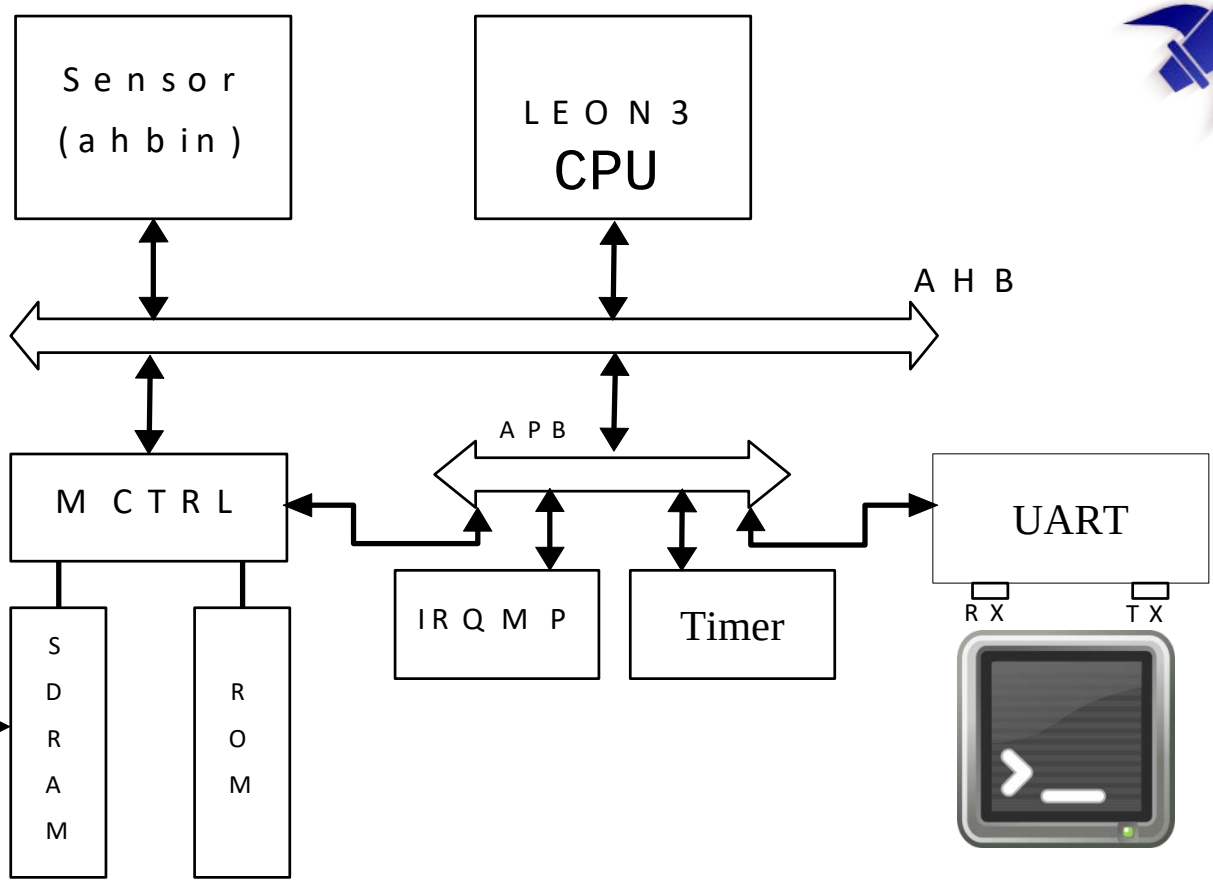
valid fields depend on type (arithmetic, IO, memory)

Case Study: SoCRocket

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

SW + FW

cross-compile



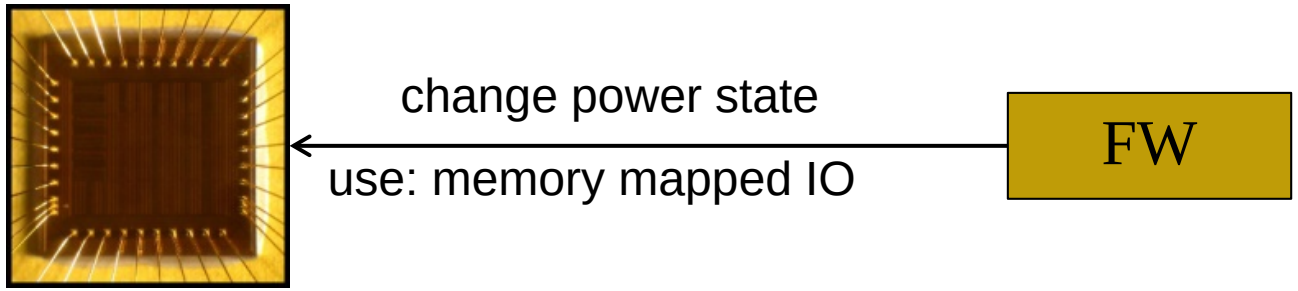
"Transaction-Level Modeling Framework for Space Applications"

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

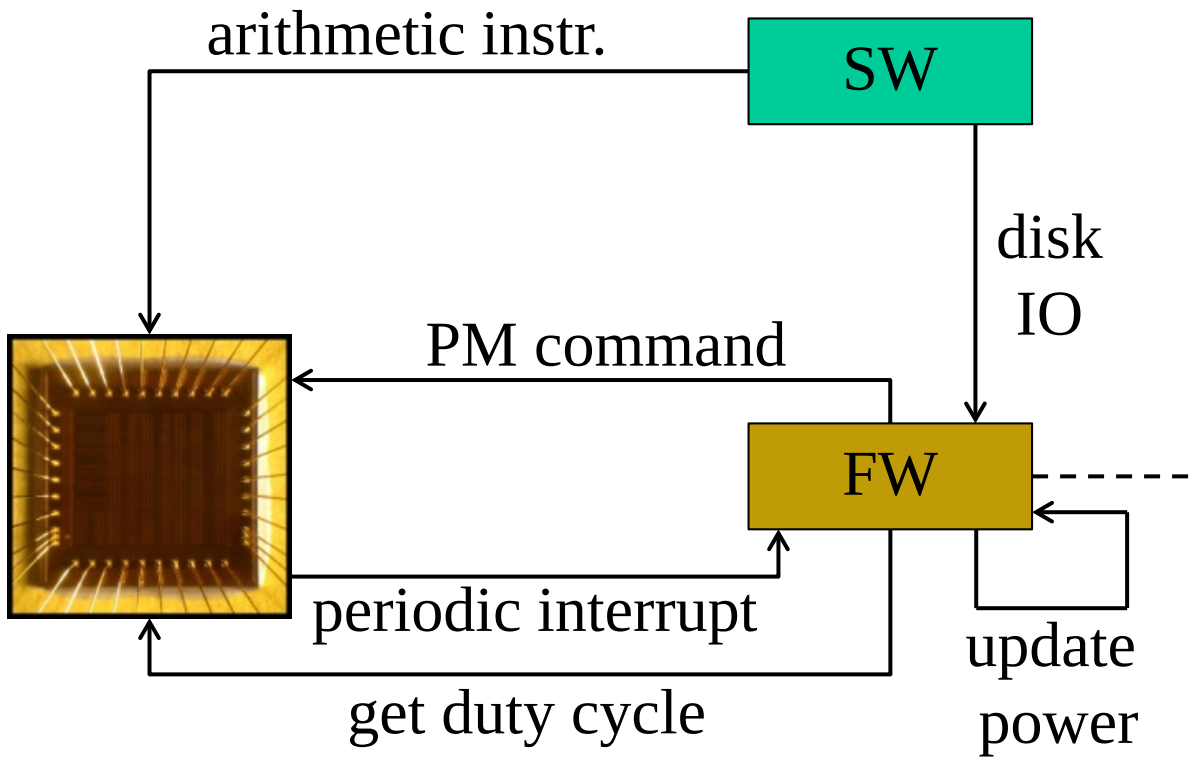
Power Management

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



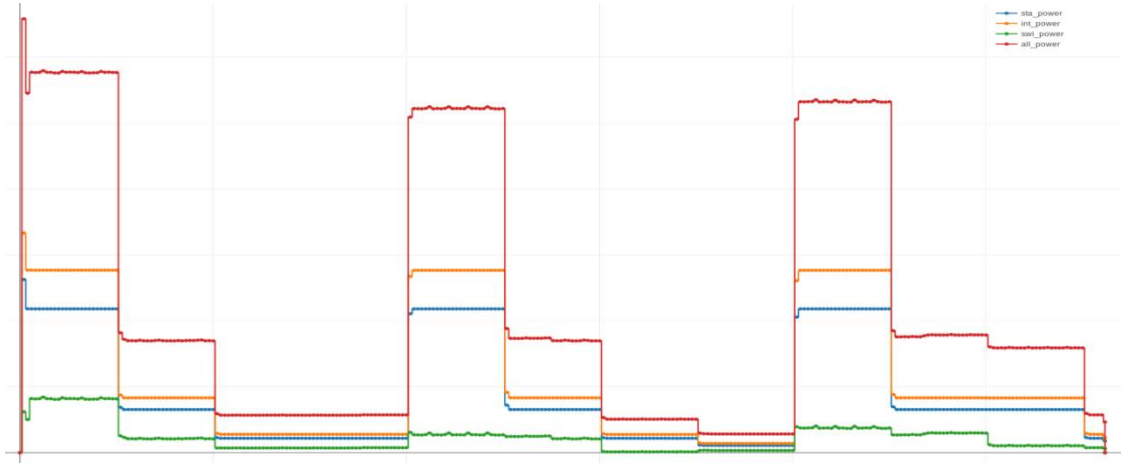

```
volatile uint *pm_control = (uint *)0xB0000000;
//…
pm_control[0] = PM_ID_LEON3 |
                    (PM_STATE_LEON3_PS2 << 8);
```

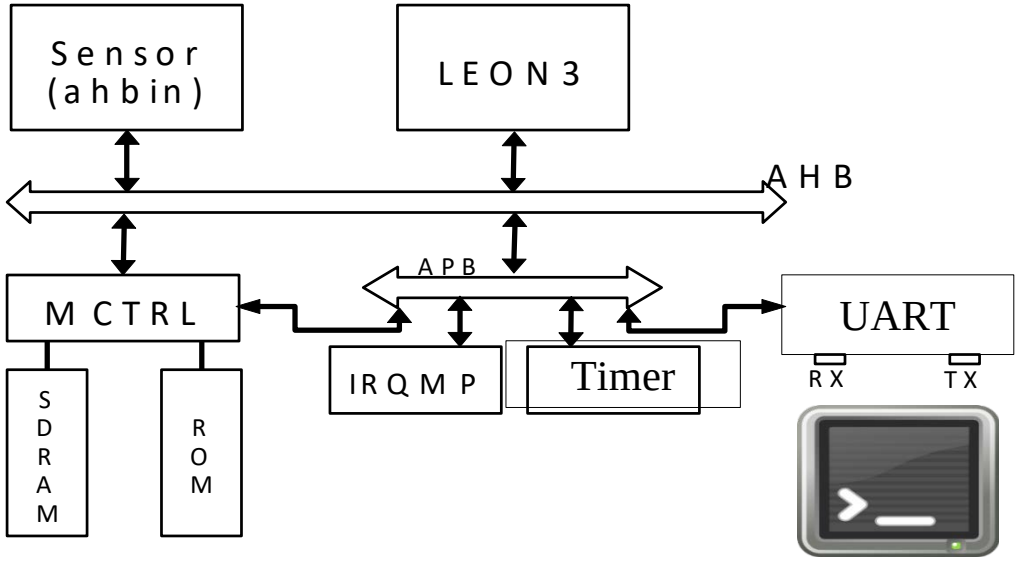
Power Management

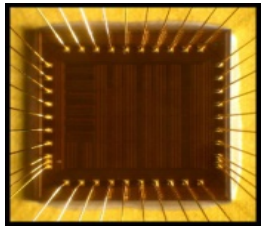


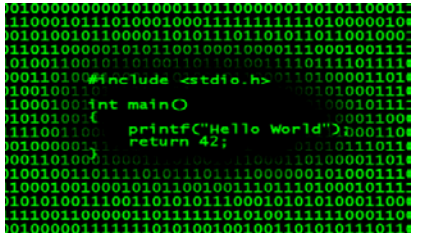

```
typedef  int8_t PM_STATE;

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









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)	Without PM	With PM	Difference
1) CPU Load	254969	94405	-62.97%
2) Interrupts	161274	129345	-19.80%
3) Alternating	397375	210988	-46.90%
4) Memory/IO	1004561	278397	-72.29%
5) Small Tasks	270656	208755	-22.87%

Save in power consumption

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

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

- V. Herdt, H. M. Le, D. Große, and R. Drechsler , Towards Early Validation of Firmware-Based Power Management using Virtual Prototypes: A Constrained Random Approach (FDL, 2017)
- http://www.systemc-verification.org/
- T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic , Socrocket - A virtual platform for the European Space Agency's SoC development (ReCoSoC, 2014), https://socrocket.github.io/

Driver Generation and Optimization of the SW/HW Interface

Martin Dittrich (TU Munich)

accellera SYSTEMS INITIATIVE

2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

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/perfomance optimized FW

State of the Art

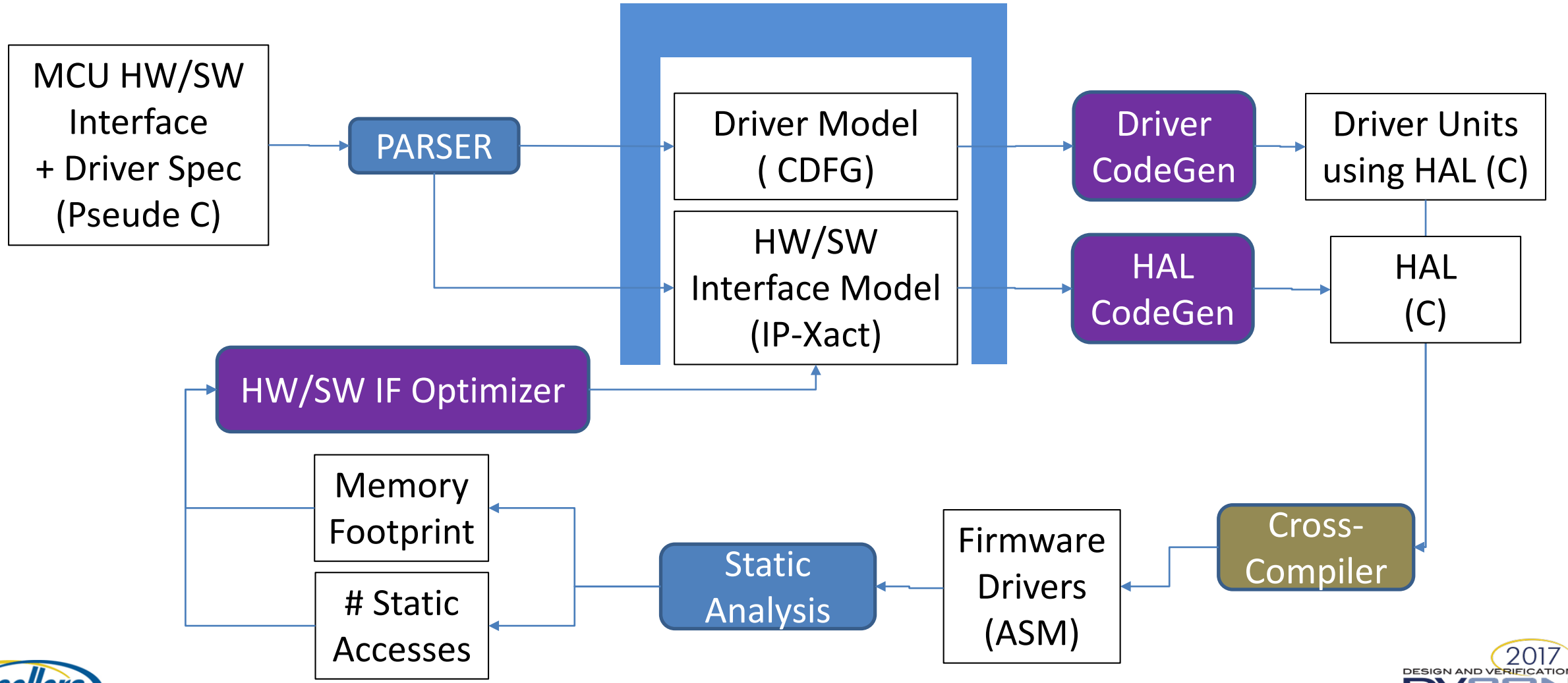
- Program Memory Footprint
  - HW Code Compression
    - Huffmann Codes
    - Dictionary-based Compression
    - Bit Masks Dictionaries
  - Link-time Optimization
  - Binary Rewriting
- Data Memory Footprint*
  - Bit Packaging
  - Pointer Tables
  - New calculation of values instead of stroring values
  - 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



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 =
  (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;
}
```

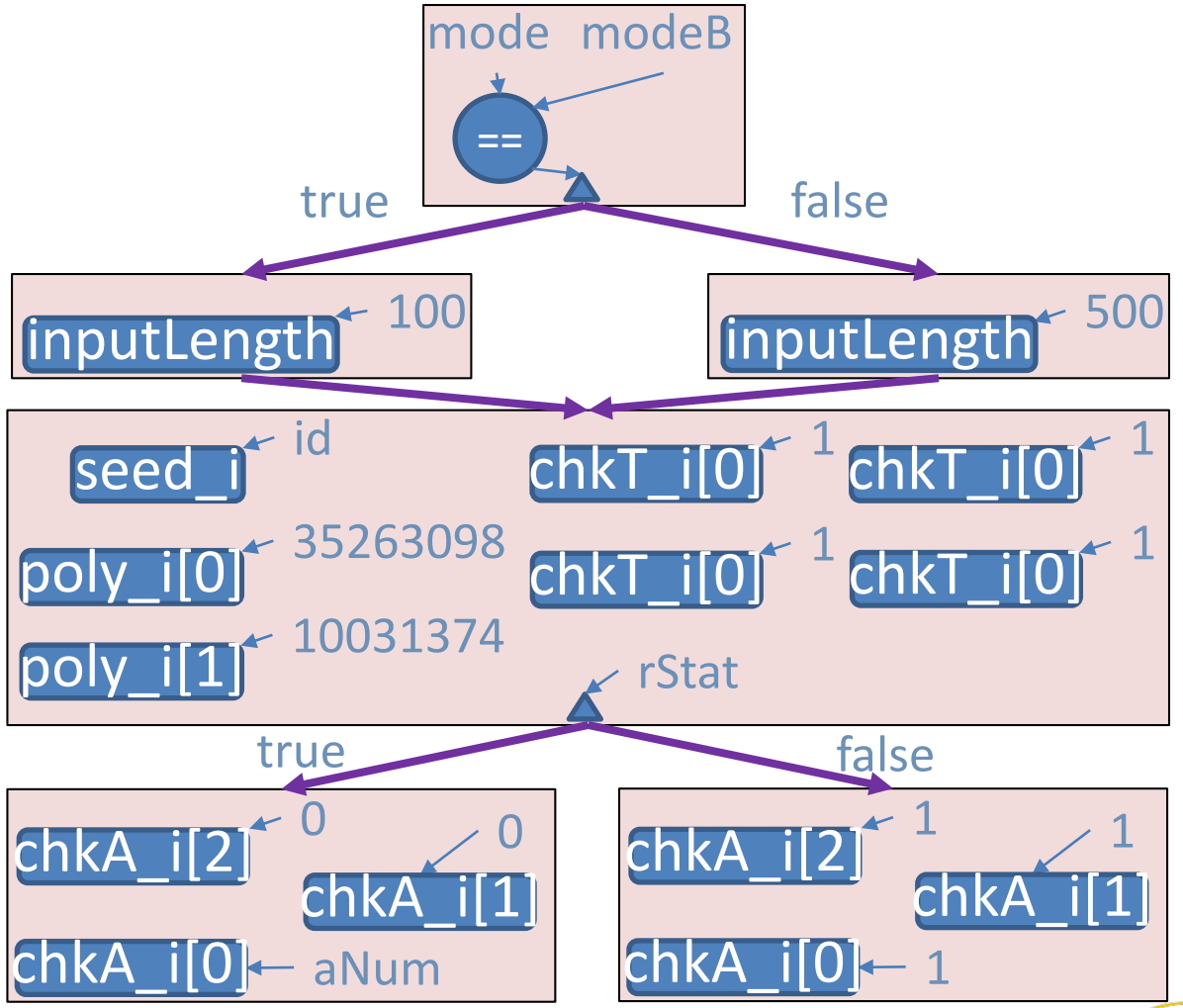
Control-data-flow-graph (CDFG) analysis

Behavior: program_device

```
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;
}
```

- Control flow dependencies
- No data dependencies
- No Write-before-read dependencies



Step 1: Generation of Register Interface

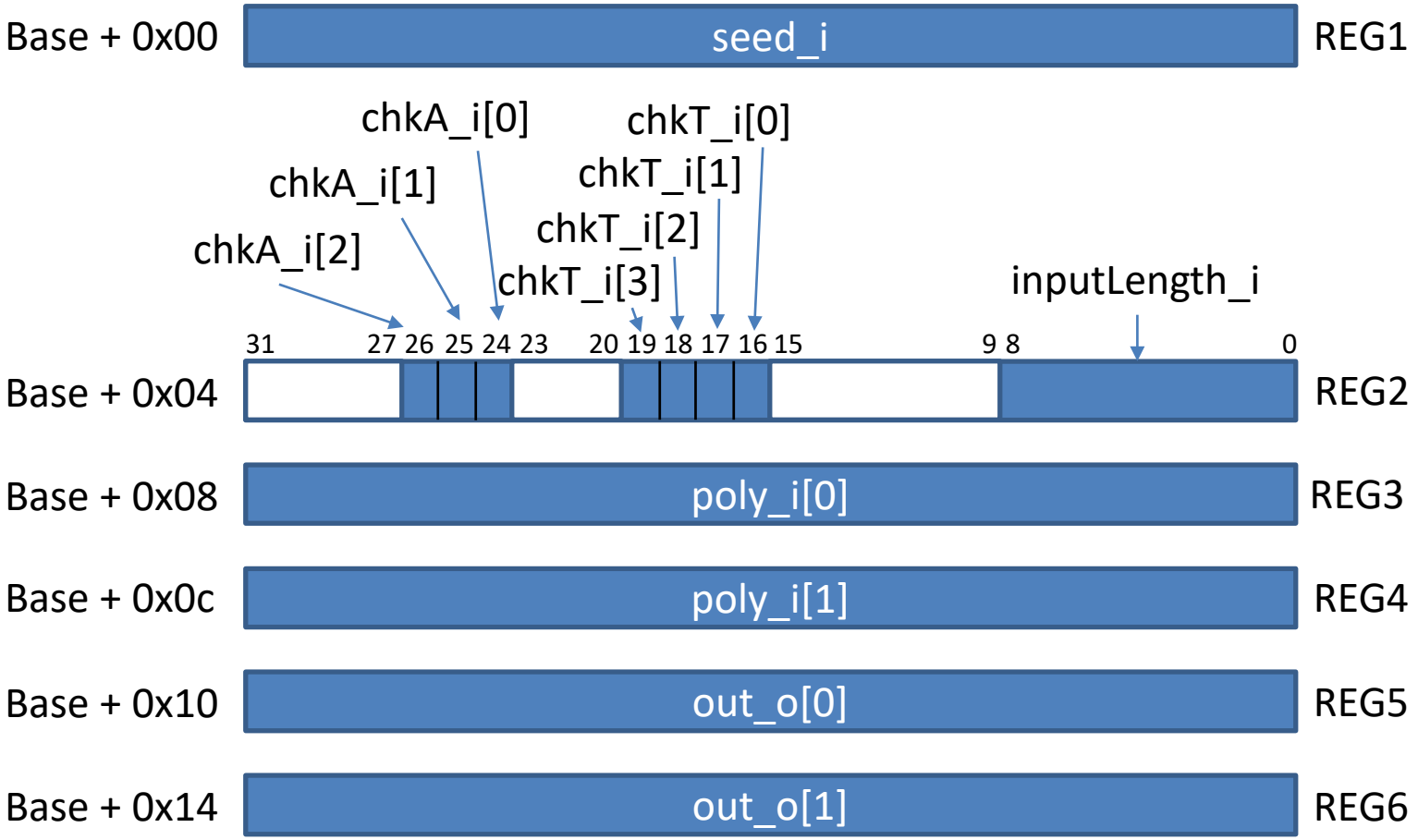
Hardware I/O Parameters

Inputs:

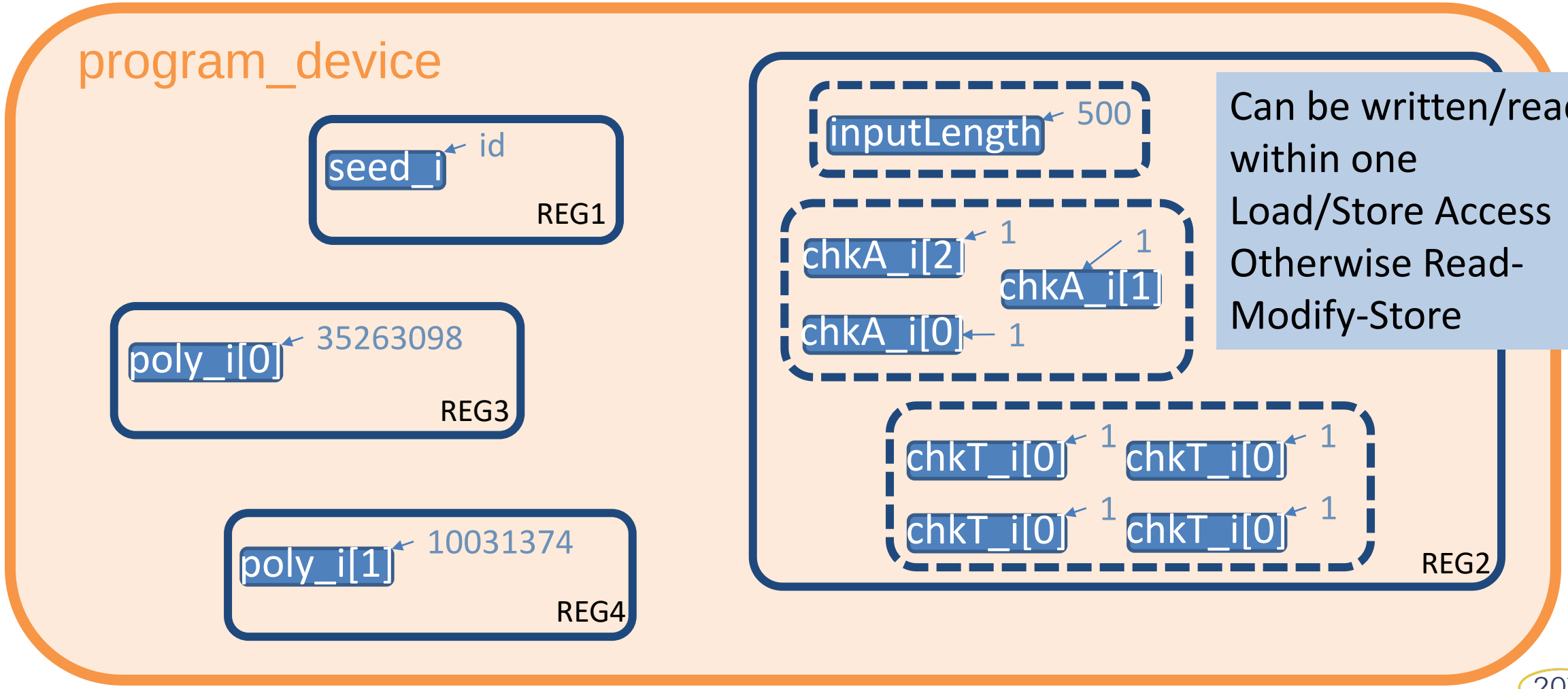
uint9_t inputLength_i;
uint32_t seed_i;
uint32_t poly_i[2];
uint1_t chkT_i[4];
uint1_t chkA_i[3];

Outputs:

uint32_t out_o[2];



Register-Compatibility Graph



Step 2: Firmware-Generation

Behavior: program_device

```
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;
}
```

Performance	# Read	0
	# Write	3
	#Read-Modify Write	8
	#HW Accesses	19

```
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);
}}
```

Step 2: Firmware-Generation

Behavior: program_device

```
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;
}
```

Performance	# Read	0
	# Write	3
	#Read-Modify Write	3
	#HW Accesses	9

```
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);
   }
}
```

Step 2: Firmware-Generation

Behavior: program_device

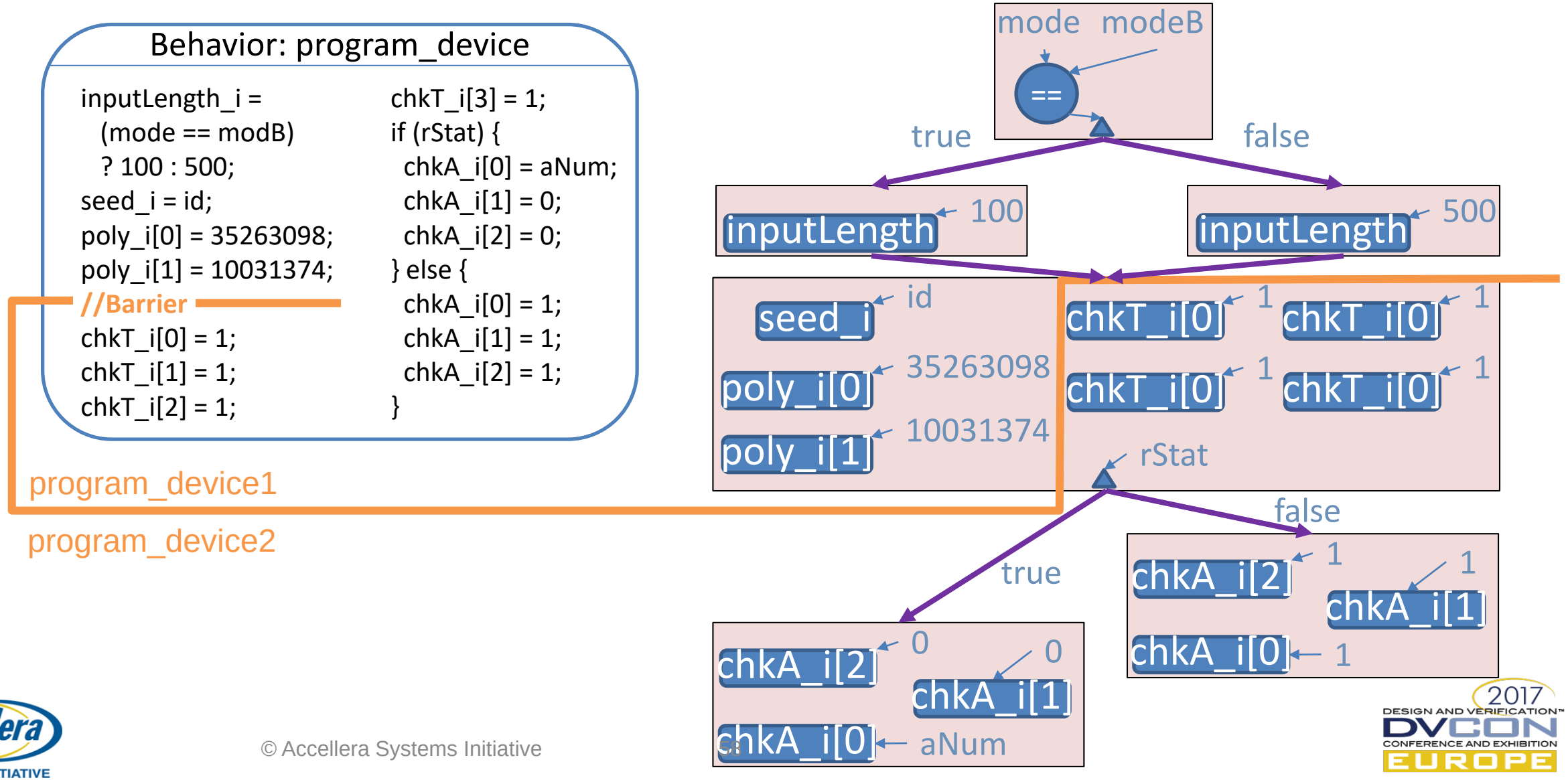
```
inputLength_i =
  (mode == modB)
  ? 100 : 500;
seed_i = id;
poly_i[0] = 35263098;
poly_i[1] = 10031374;
chkT_i[0] = 1;
chkT_i[1] = 1;
chkT_i[2] = 1;
chkT_i[3] = 1;
```

```
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;
}
```

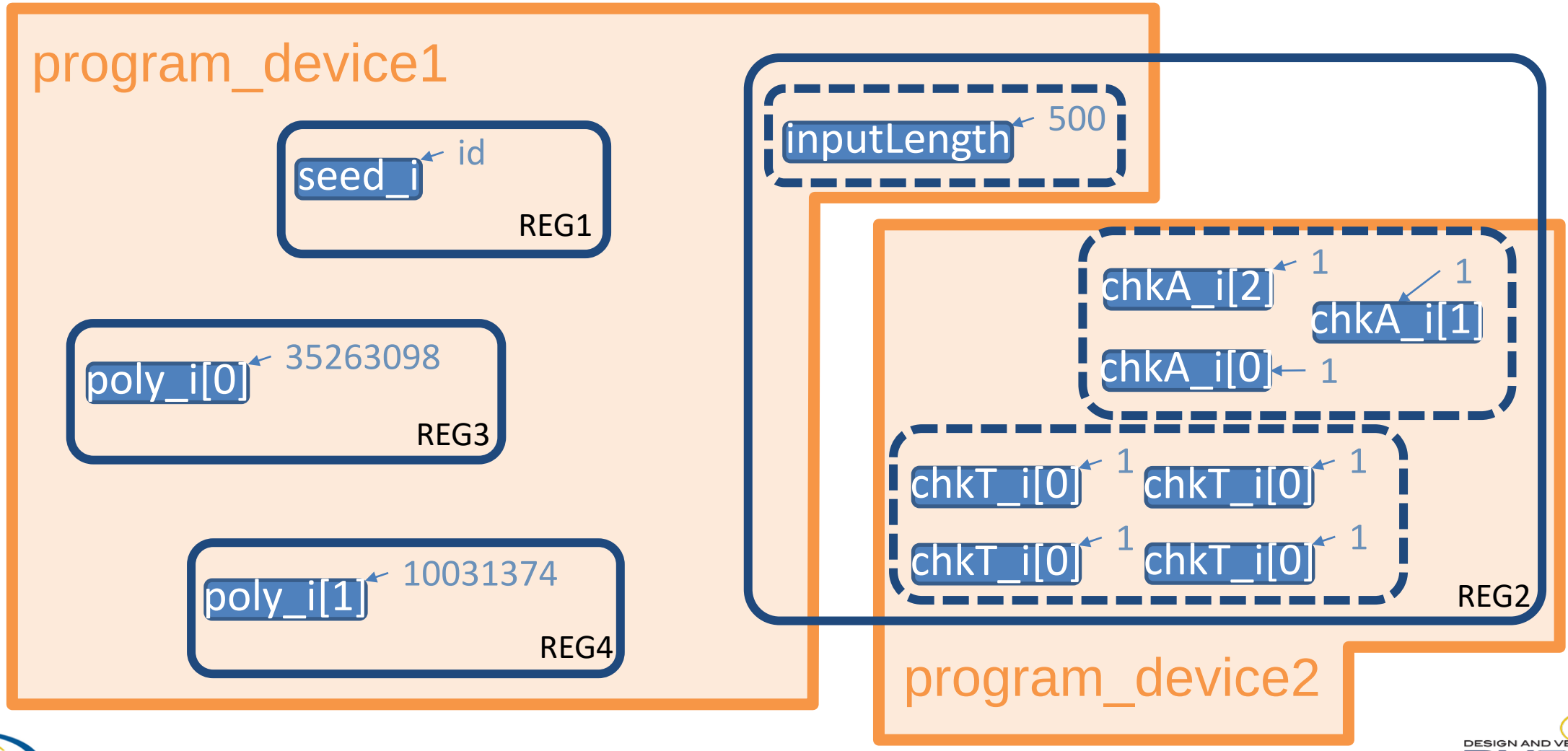
Performance		
	# Read	0
	# Write	4
	#Read-Modify Write	0
	#HW Accesses	4

```
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);
  }
}
```

Control-data-flow-graph (CDFG) analysis



Register-Compatibility Graph



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;
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;
}
```

Performance		
	# Read	0
	# Write	3
	#Read-Modify Write	2
	#HW Accesses	7

```
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)

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

   } else {
     set_chkT_chkA(1,1,1,1,1,1,1);
  }
}
```

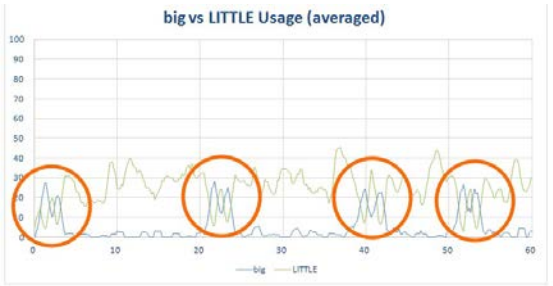
Conclusion

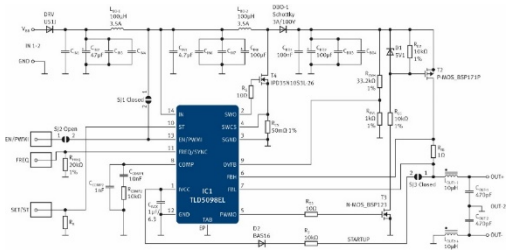
- Automatic FW Code-Generation
- Optimization of the HW/SW Interface

- 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





Questions?