Notos: Efficient Emulation of Wireless Sensor Networks with Binary-to-Source Translation

Robert Sauter, Sascha Jungen, Richard Figura, and Pedro José Marrón

University of Duisburg-Essen, Germany
Motivation (Short Version)

Experts Agree

“In Simulation, Faster is Better”
Sensor Network Nodes

16 Bit CPU, 4-8 MHz
10 KB RAM, 48 KB ROM
Low-Power Low-Bandwidth
Wireless Communication
Battery Powered
WSN Applications
Motivation (2)

- Software development for WSN is challenging
  - Severe resource constraints $\rightarrow$ optimized code necessary
  - Very dynamic; unreliable nodes; unreliable links
  - Large-scale distributed system

$\rightarrow$ Simulation to the Rescue!
Motivation (2)

- Software development for WSN is challenging
  - Severe resource constraints → optimized code necessary
  - Very dynamic; unreliable nodes; unreliable links
  - Large-scale distributed system

→ Simulation to the Rescue!

- But

  - Application code, protocols, and operating system tightly coupled
  - Cross-layer optimizations
  - Large impact of environment on protocol/system behavior
  - Switch among simulation, testbeds and exploratory deployments

→ Cycle-accurate emulation to the Rescue!
Cycle-Accurate Emulation

- Code written and compiled for target platform
- Emulator simulates each cycle of the target platform
- Emulator contains model of the micro-controller …
  - Core CPU: supported instructions, flags and registers
  - RAM and program memory (flash)
  - Timers, ADC, DAC, I/O ports
- … and the relevant periphery
  - Radio
  - External Flash, sensors
- High fidelity and usability, fully deterministic, but computationally expensive
Core Engine

- Core engine uses discrete-event simulation
- Events simulate hardware interrupts
  - E.g., by timers, radio, …
- Main loop
  - Read current instruction
  - Modify simulated state
  - Check event queue

```java
public void visit(LegacyInstr.ADC i){
    nextPC = pc + 2;
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    int tmp_4 = tmp_0 & 0x0000000F;
    int tmp_5 = tmp_1 & 0x0000000F;
    boolean tmp_6 = (tmp_0 & 128) != 0;
    boolean tmp_7 = (tmp_1 & 128) != 0;
    boolean tmp_8 = (tmp_3 & 128) != 0;
    int H = (tmp_4 + tmp_5 + tmp_2 & 16) != 0;
    int C = (tmp_3 & 256) != 0;
    int N = (tmp_3 & 128) != 0;
    int Z = low(tmp_3) == 0;
    byte V = tmp_6 && tmp_7 && !tmp_8 || !tmp_6 && !tmp_7 && tmp_8;
    byte tmp_9 = low(tmp_3);
    writeRegisterByte(i.r1, tmp_9);
    cyclesConsumed++;
}
```
Avrora Main Loop

Instructions

ADC
r1: Register
r2: Register
accept(v:VisitorInterface): void
v.visit(this)

WDR
accept(v:VisitorInterface): void
v.visit(this)

Interpreter

... LegacyInstr i = shared_instr[nextPC];
i.accept(this);
cycles = cyclesConsumed;
Commit();
...

Visitor Interface

public void visit(LegacyInstr.ADC i) {
    nextPC = pc + 2;
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    ...
}
Binary-to-Source Translation

- Adapt emulator to program
  - Transform AVR instructions to corresponding simulator code and compile for host platform (here: Java)
  - Replace main loop and integrate with event handling and simulation of components

- Java limitations
  - Partition code into methods with switch statements
  - Higher level constructed binary search tree

```java
while(innerLoop && nextPC >= 0 && nextPC <= 690) {
    switch (nextPC) {
    case 0:  // jmp 436
        nextPC = 436 * 2;
        pc = nextPC;
        eventQueue.advance(3);
        inst_692_to_1338();
        break;
    ...
    case 140:  // adc r3, r9
        nextPC = 140 + 2;
        tmpInt_0 = getRegisterUnsigned(R3);
        tmpInt_1 = getRegisterUnsigned(R9);
        tmpInt_2 = bit(C);
        ...
        C = (tmpInt_3 & 256) != 0;
        tmpByte_0 = low(tmpInt_3);
        writeRegisterByte(R3, tmpByte_0);
        pc = nextPC;
        eventQueue.advance(1);
        if(!innerLoop) {
            break;
        }
    }
    ...
}
```
Binary-to-Source Translation (2)

- Reduce method calling overhead
  - Best case: next instruction begins with next statement
  - Usually: within switch statement without use of the BST

- Foundation for static analysis and optimizations
  - Individual code per instruction (vs. per instruction type)
  - Use additional knowledge, e.g., addresses
  - Consider instruction sequences

- Lazy timer evaluation
  - Calculate timer value on demand and only translate simulated interrupts to events instead of every tick
  - Significantly increase occurrences of sequence optimizations
Optimization for Individual Instructions

- Optimized utility functions, for example:
  - Hardware maps I/O ports to RAM: accessing RAM can involve multiple simulated components
  - If address is known, runtime checks can be removed (done by dedicated streamlined utility functions)
  - Small savings but very frequently used $\rightarrow$ evaluation shows highest impact

- Constant folding and constant propagation of immediates
  - Allows pre-calculation and removal of intermediate expressions
  - Iterative optimization algorithm
  - Simple example: instruction pointer

- Peephole optimizations
  - For some instructions, result is already known (e.g., XOR R1, R1)
Conditional Execution of Code

- Emulation of arithmetic and logical instructions
  1. Bookkeeping (instruction pointer, event queue, ...)
  2. Fetch input operands (e.g., registers)
  3. Calculate intermediate expressions
  4. Store result and update flags

```java
public void visit(LegacyInstr.ADC i) {
    nextPC = pc + 2;
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    int tmp_4 = tmp_0 & 0xFFFFFFFF;
    int tmp_5 = tmp_1 & 0xFFFFFFFF;
    boolean tmp_6 = (tmp_0 & 128) != 0;
    boolean tmp_7 = (tmp_1 & 128) != 0;
    boolean tmp_8 = (tmp_3 & 128) != 0;
    H = (tmp_4 + tmp_5 + tmp_2 & 16) != 0;
    C = (tmp_3 & 256) != 0;
    N = (tmp_3 & 128) != 0;
    Z = low(tmp_3) == 0;
    V = tmp_6 && tmp_7 && !tmp_8 || !tmp_6 && !tmp_7 && tmp_8;
    S = N != V;
    byte tmp_9 = low(tmp_3);
    writeRegisterByte(i.r1, tmp_9);
    cyclesConsumed++;
}
```
Conditional Execution of Code

- Emulation of arithmetic and logical instructions
  1. Bookkeeping (instruction pointer, event queue, ...)
  2. Fetch input operands (e.g., registers)
  3. Calculate intermediate expressions
  4. Store result and update flags

```java
public void visit(LegacyInstr.ADC i) {  
  nextPC = pc + 2;
  int tmp_0 = getRegisterUnsigned(i.r1);
  int tmp_1 = getRegisterUnsigned(i.r2);
  int tmp_2 = bit(C);
  int tmp_3 = tmp_0 + tmp_1 + tmp_2;
  int tmp_4 = tmp_0 & 0x0000000F;
  int tmp_5 = tmp_1 & 0x0000000F;
  boolean tmp_6 = (tmp_0 & 128) != 0;
  boolean tmp_7 = (tmp_1 & 128) != 0;
  boolean tmp_8 = (tmp_3 & 128) != 0;
  H = (tmp_4 + tmp_5 + tmp_2 & 16) != 0;
  C = (tmp_3 & 256) != 0;
  N = (tmp_3 & 128) != 0;
  Z = low(tmp_3) == 0;
  V = tmp_6 && tmp_7 && !tmp_8 || !tmp_6 && !tmp_7 && tmp_8;
  S = N != V;
  byte tmp_9 = low(tmp_3);
  writeRegisterByte(i.r1, tmp_9);
  cyclesConsumed++;
}
```
Conditional Execution of Code

- Emulation of arithmetic and logical instructions
  1. Bookkeeping (instruction pointer, event queue, ...)
  2. Fetch input operands (e.g., registers)
  3. Calculate intermediate expressions
  4. Store result and update flags

```java
public void visit(LegacyInstr.ADC i){
    nextPC = pc + 2;
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    int tmp_4 = tmp_0 & 0x0000000F;
    int tmp_5 = tmp_1 & 0x0000000F;
    boolean tmp_6 = (tmp_0 & 128) != 0;
    boolean tmp_7 = (tmp_1 & 128) != 0;
    boolean tmp_8 = (tmp_3 & 128) != 0;
    H = (tmp_4 + tmp_5 + tmp_2 & 16) != 0;
    C = (tmp_3 & 256) != 0;
    N = (tmp_3 & 128) != 0;
    Z = low(tmp_3) == 0;
    V = tmp_6 & tmp_7 & !tmp_8 || !tmp_6 & tmp_7 & tmp_8;
    S = N != V;
    byte tmp_9 = low(tmp_3);
    writeRegisterByte(i.r1, tmp_9);
    cyclesConsumed++;
}
```
Conditional Execution of Code

- **Emulation of arithmetic and logical instructions**
  1. Bookkeeping (instruction pointer, event queue, …)
  2. Fetch input operands (e.g., registers)
  3. Calculate intermediate expressions
  4. Store result and update flags

```java
public void visit(LegacyInstr.ADC i){
    nextPC = pc + 2;
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    int tmp_4 = tmp_0 & 0x0000000F;
    int tmp_5 = tmp_1 & 0x0000000F;
    boolean tmp_6 = (tmp_0 & 128) != 0;
    boolean tmp_7 = (tmp_1 & 128) != 0;
    boolean tmp_8 = (tmp_3 & 128) != 0;
    byte tmp_9 = low(tmp_3);
    cyclesConsumed++;
}
```
Conditional Execution of Code

- Emulation of arithmetic and logical instructions
  1. Bookkeeping (instruction pointer, event queue, …)
  2. Fetch input operands (e.g., registers)
  3. Calculate intermediate expressions
  4. Store result and update flags

```java
public void visit(LegacyInstr.ADC i){
    nextPC = pc + 2;
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    int tmp_4 = tmp_0 & 0x0000000F;
    int tmp_5 = tmp_1 & 0x0000000F;
    boolean tmp_6 = (tmp_0 & 128) != 0;
    boolean tmp_7 = (tmp_1 & 128) != 0;
    boolean tmp_8 = (tmp_3 & 128) != 0;
    H = (tmp_4 + tmp_5 + tmp_2 & 16) != 0;
    C = (tmp_3 & 256) != 0;
    N = (tmp_3 & 128) != 0;
    Z = low(tmp_3) == 0;
    V = tmp_6 && tmp_7 && !tmp_8 || !tmp_6
        && !tmp_7 && tmp_8;
    S = N != V;
    byte tmp_9 = low(tmp_3);
    writeRegisterByte(i.r1, tmp_9);
    cyclesConsumed++;
}
```
Conditional Execution of Code

- Emulation of arithmetic and logical instructions
  1. Bookkeeping (instruction pointer, event queue, ...)
  2. Fetch input operands (e.g., registers)
  3. Calculate intermediate expressions
  4. Store result and update flags

- Often: significant overhead for computation of flags

```java
public void visit(LegacyInstr.ADC i){
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    int tmp_4 = tmp_0 & 0x000000FF;
    int tmp_5 = tmp_1 & 0x000000FF;
    boolean tmp_6 = (tmp_0 & 128) != 0;
    boolean tmp_7 = (tmp_1 & 128) != 0;
    boolean tmp_8 = (tmp_3 & 128) != 0;
    int H = (tmp_4 + tmp_5 + tmp_2 & 16) != 0;
    int C = (tmp_3 & 256) != 0;
    int N = (tmp_3 & 128) != 0;
    int Z = low(tmp_3) == 0;
    boolean V = tmp_6 && tmp_7 && !tmp_8 || !tmp_6 && !tmp_7 && tmp_8;
    byte tmp_9 = low(tmp_3);
    writeRegisterByte(i.r1, tmp_9);
    cyclesConsumed++;
}
```
Conditional Execution of Code

- Emulation of arithmetic and logical instructions
  1. Bookkeeping (instruction pointer, event queue, …)
  2. Fetch input operands (e.g., registers)
  3. Calculate intermediate expressions
  4. Store result and update flags
- Often: significant overhead for computation of flags

```java
public void visit(LegacyInstr.ADC i) {
    nextPC = pc + 2;
    int tmp_0 = getRegisterUnsigned(i.r1);
    int tmp_1 = getRegisterUnsigned(i.r2);
    int tmp_2 = bit(C);
    int tmp_3 = tmp_0 + tmp_1 + tmp_2;
    int tmp_4 = tmp_0 & 0x0000000F;
    int tmp_5 = tmp_1 & 0x0000000F;
    boolean tmp_6 = (tmp_0 & 128) != 0;
    boolean tmp_7 = (tmp_1 & 128) != 0;
    boolean tmp_8 = (tmp_3 & 128) != 0;
    int H = (tmp_4 + tmp_5 + tmp_2 & 16) != 0;
    int C = (tmp_3 & 256) != 0;
    int N = (tmp_3 & 128) != 0;
    int Z = low(tmp_3) == 0;
    boolean V = tmp_6 && tmp_7 && !tmp_8 || !tmp_6 && !tmp_7 && tmp_8;
    byte tmp_9 = low(tmp_3);
    writeRegisterByte(i.r1, tmp_9);
    cyclesConsumed++;
}
```
Conditional Execution of Code (2)

- Flags may be overwritten by subsequent instruction
  - Only calculate flags (and necessary intermediate values) that are read or unmodified by subsequent instruction
- But: event may be scheduled at any time
  - Move calculation to conditional block and only execute if emulation continues uninterrupted

- At generation time
  - Build graph with flags and intermediates
  - Identify flags for which computation may be unnecessary
  - Identify intermediates that are not used by necessary computations
Optimized Paths for Instruction Sequences

- Extend optimization to sequences
  - May not have branches (amount of cycles must be constant)
  - No events may be scheduled for duration of sequence
    - Checking this has significant overhead

- Rare occurrence at runtime
Manually Constructed Alternative Code Paths

- **Idea:** replace *common and often used instruction sequences* with *optimized code variants*
  - OS functions, compiler artifacts and standard library functions
- **Framework** to specify patterns (e.g., to handle varying register allocations)
- **Advantage:** savings can be huge
  - Example: integer division of the avr-gcc standard library (193-209 cycles)
  - 99% reduction from ~2800 statements to ~20
- **Disadvantage:** significant manual overhead
  - Calculate result, flags, and *number of cycles*
Evaluation Setup

- Hosts:
  - Desktop PCs with quad-core (+SMT) CPUs @ 2.8 GHz, 8 GB RAM
  - 64-Bit JVM (2 GB RAM) on 64 Bit Linux

- Applications (based on TinyOS examples)
  - MViz: “canonical” WSN app: periodic sensing + collection routing
  - MVizComplex: enable low power listening radio stack and in-network processing with computation load

- Grid topologies with 16, and 100 nodes
Evaluation Methodology

- Challenge: evaluate performance of Java software
  - Significant impact of just in-time compilation
  - Performance varies over time especially compared to startup
- Goal: evaluate so-called "steady-state" performance
  - Better indication for long-running simulations
- Approach: within one JVM session …
  - Run simulation twice as warm-up
  - Run simulation 5 times for 300 simulated seconds
  - Plots show average, minimum, and maximum time
- Generation overhead
  - ~20 seconds for a complex application
  - Caching alleviates overhead for multiple simulations (e.g., varying network sizes)
Network Emulation Performance (1)

![Graph showing network emulation performance with Avrora, noopt, optUtility, optCond, noShortcuts, and allOpt configurations. The x-axis represents Mviz, 16 nodes, and the y-axis represents wall time in seconds. The graph indicates Avrora is significantly better than the other configurations.](image-url)
Network Emulation Performance (2)

Wall time in seconds

- Avrora
- noopt
- optUtility
- optCond
- noShortcuts
- allOpt

Mviz, 100 nodes
Network Emulation Performance (3)

![Bar chart showing wall time in seconds for different network emulation options on MvizComplex, 100 nodes. Avrora has the highest wall time, followed by noopt, optUtility, optCond, noShortcuts, and allOpt. The chart indicates that allOpt performs better than the other options.]
Conclusions

- Binary-to-source translation for the emulation of WSN
  - Analysis and optimization framework
  - Instruction and instruction sequence optimizations

- Most advantageous when simulating computation ...
  - Small networks
  - Complex applications with in-network processing

- ... smaller benefit when simulating “waiting”
  - Large networks
  - Translation overhead must be offset by number of simulations or simulation time
Thank you for your attention!