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

Robert Sauter, Sascha Jungen, Richard Figura, and Pedro José Marrón
Networked Embedded Systems Group
University of Duisburg-Essen
Essen, Germany
robert.sauter@uni-due.de, sascha.jungen@uni-due.de,
richard.figura@uni-due.de, pjmarron@uni-due.de

ABSTRACT

Developing for wireless sensor networks is a challenging task due to the severe resource constraints of the devices, the uncertainties of the environment, and the distributed nature of the system. Therefore, simulation is an essential tool for developing systems and for evaluating and comparing protocols at scale in a reproducible manner. Cycle-accurate emulation of sensor networks allows the execution of platform target code and provides deep insight into the behavior of the overall system including the important aspect of energy consumption. However, the required fidelity incurs a significant overhead and limits the size of the emulated networks considerably. We investigate the use of binary-to-source translation, where the machine code of an executable for the target platform is transformed to source code for the host platform and compiled as part of the emulator. Additionally, as part of this transformation we perform static analysis and optimize the generated code. We have implemented our approach based on the well-established emulator Avrora and show in our evaluation that this approach can lead to significantly higher simulation speeds.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modelling**]: Applications; D.4.8 [**Operating Systems**]: Performance—*Simulation*

General Terms

Performance, Measurement, Design

Keywords

Emulation, Simulation, Wireless Sensor Networks

1. INTRODUCTION

The high cost of real deployments and the need for reproducibility and ease-of-use make simulation and emulation important tools for research and development in the area of wireless sensor networks. In contrast, the unreliability of the wireless medium, the resource constraints of the hardware, and the impact of the environment require the testing and evaluation of new protocols and systems in testbeds and experimental deployments. Therefore, cycle-accurate emulation that allows the simulated execution of platform target code including the application, the operating system, and device drivers, is an important simulation method. This simulation of the hardware itself facilitates the seamless transition from simulation, via testbeds, to deployments and back for development and evaluation. Additionally, this high-fidelity simulation provides detailed insight into the behavior and performance (including the important aspect of energy consumption) of the software under test.

However, the fidelity leads to significant overhead to ensure consistency and repeatability compared to more abstract simulation approaches and tools that focus on the high-level implementation of protocols building on existing components and models for evaluation purposes. While there is a significant body of work investigating the use of parallel and distributed machines to increase emulator performance, we focus on improving the core execution engine that is responsible for interpreting the individual instructions (e.g., an 'add' instruction) and updating the state of the simulated hardware. Instead of a main loop that decodes and interprets instructions, our approach adapts the simulator itself by translating the binary for the target platform to source code for the host platform of the simulator. This source code is then compiled and integrated with the simulator framework forming a dedicated adapted emulator for the specified firmware. This binary-to-source translation reduces overhead at runtime and more importantly provides the foundation for static analysis of the binary and further adaptation of the generated code to move decisions from runtime to generation time when possible.

Contribution and roadmap In this paper we provide a solution that improves the performance of cycle-accurate emulation by translating the binary for the target platform to code integrated in the simulator. We discuss the problem of cycle-accurate emulation of wireless sensor networks in detail in Section 2 and related work in Section 6. Our contribution in this paper is threefold.

In Section 3, we present our approach that focuses on the core interpreter of the simulator. Instead of a core loop decoding an instruction and executing the necessary code to

SIMUTOOLS 2015, August 24-26, Athens, Greece

DOI 10.4108/eai.24-8-2015.2261070

transform the simulated state of the hardware, we translate the target binary to the source language of the emulator (in this case Java) and compile and integrate the source into the emulator itself. Furthermore, this step provides the foundation for static analysis of the code.

In Section 4, we discuss several optimizations based on the static analysis that lead to further adaption of the generated code to the binary. This includes techniques from compiler construction and virtual machines using just-in-time compilation.

Third in Section 5, we study the impact of binary-to-source translation and the optimization techniques that build on it. We show that the impact is significant for small and medium networks but decreases with network size because the overhead for synchronization and simulating communication increases compared to the time the simulator requires for the actual execution of the program.

2. CYCLE-ACCURATE EMULATION

The development and evaluation of wireless sensor network systems and protocols make use of a spectrum of settings with regard to realism spanning the actual deployment on the one end, simulation on the other end, and testbeds with different levels of simplifications in between. Furthermore, within the simulation realm, solutions with three different abstraction levels exist. The highest abstraction level contains mostly existing traditional tools like OMNeT++ [23] and ns-2 [16] extended with support for WSN specific protocols and radio simulations. This abstraction level requires the implementation of a new protocol in the simulator specific language but allows building on existing and well tested components. Second, 'same-source' simulation allows the implementation of new components in the language of the operating system with compilation for the host CPU of the simulator. This requires simulating OS components close to the hardware (for example, a radio device driver) and connecting several instances via a simulated medium, but leaves protocol and application implementations unchanged. With TinyOS [15] and Contiki [2], at least two major research operating systems in this field support this abstraction level ([14], [17]). Finally, cycle-accurate emulation simulates the target hardware, i.e., the microcontroller, the radio, and possibly additional components of the node. In this case, the same firmware binary that is run on the target platform is loaded by the emulator and the complete software is interpreted and executed by the simulator, including the operating system and device drivers.

Besides the possibility to use well-tested existing models, e.g., for the simulation of the wireless medium, higher performance is the main advantage of traditional simulation tools. However, the unreliability of hardware, software, and especially the wireless medium as well as the impact of the environment that are hard to simulate with high fidelity led to the advent of testbeds and even real deployments as the primary evaluation setting. While simulation is still used as an evaluation tool, the primary focus is on supporting the creation and further development of systems and protocols. Therefore, the use of the exact same software for both testbeds/deployments and simulation while providing the important simulation benefits of reproducibility, scalability, and ease of use is the leading factor to use cycle-accurate emulation of wireless sensor networks.

The core process of simulating the target CPU is relatively simple: the simulator contains a model of the relevant parts, for example the RAM and the registers. Each instruction in the binary is interpreted and changes the associated state in the simulator. The main challenges derive from the simulation of all parts of the microcontroller on the one hand and the need to keep consistent and reproducible state for a number of emulated nodes forming a simulated network on the other hand. Therefore, each event triggered by the emulated hardware – usually emulating an interrupt on the hardware level – must be handled at the exact time in cycles as indicated by the hardware model. Furthermore, these events for one node can be induced by the emulation of the other nodes of the network. The simulated execution engine is, therefore, driven by the discrete event simulation method, where an event queue contains the already scheduled events, for example interrupts triggered by a hardware timer, that impact the flow of interpreted instructions.

To increase the performance and exploit the inherent parallelism in simulating a network of nodes, the emulator Avrora [21] – on which our work builds – uses one thread per simulated node. This requires synchronization of the threads to ensure deterministic behavior, for example, when the reception of a message from another node must be simulated at the correct global time with respect to the simulation of the whole network.

3. BINARY-TO-SOURCE TRANSLATION

Our work focuses on the core interpreter of the emulator. In the following, we will refer to 'binary' and 'instructions' when referring to information about the program-under-test compiled for the target platform. We use the terms 'code' consisting of 'statements' and 'expressions' when reasoning about the necessary operations on the host platform to emulate the binary. For example the simulation of an 'add' instruction for two registers requires several statements that do not only compute the result and store it in the simulated state of the target register, but also change the state of several simulated processor flags ('carry', etc.). Additionally, these statements are surrounded by 'housekeeping', for example checking the event queue and possibly changing the hardware state or determining which instruction has to be interpreted next.

Avrora already uses a technique that goes beyond a trivial decode-interpret-advance loop: there is a class for each instruction variant that contains fields for the operands (e.g., indicating register 3 or a relative address) and the code necessary to change the simulation state when interpreting the instruction. The complete binary is decoded in one step at the beginning of the simulation and the appropriate object for each instruction is instantiated with the respective operands and stored in an array at the corresponding address. The emulator then uses these instances when executing a step during the simulation (c.f., Fig. 1).

Our approach transforms the binary into the source code of a new Java class by translating each instruction into the corresponding code and replacing expressions in the code

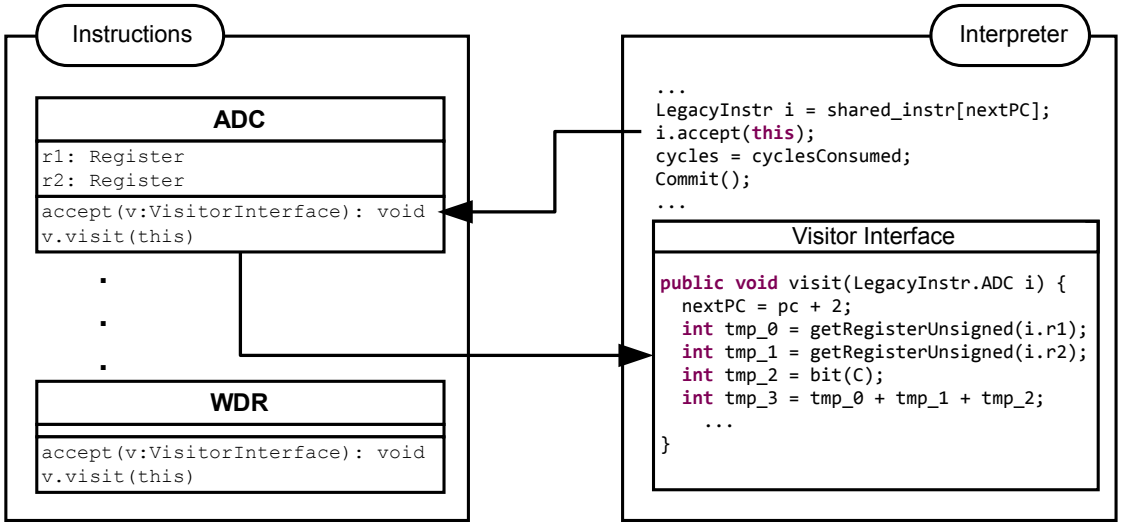


Figure 1: Avrora Instruction Classes and Interpreter Loop

with the appropriate operands of the instruction. This class is then compiled and loaded into the simulator to perform the core executions on the simulated state of a node, which eliminates part of the overhead of the interpreter. Additionally, this step allows static code analysis and optimizations of the generated code as described in the next section. In the following discussion we differentiate between 'compile time' – the compilation of the binary for the target platform –, 'runtime' – the execution of the emulator –, and 'generation time' – the phase where the binary is transformed to code and included in the emulator.

While the basic transformation process is straight-forward and the design of Avrora allows direct reuse of the source code for each instruction when generating the code for the binary, correct integration into the emulation requires additional steps as well as do the constraints of the runtime environment of the emulator (in this case Java). To simulate jumps, it must still be possible to continue execution at each statement corresponding to the start of the code for an instruction. This is handled by a 'switch' statement on the value of the program counter. Because the runtime limits the maximum method size to 64 kB of bytecode, the code for a binary has to be split into several pieces. This second hierarchy level performing a binary search for the correct method for the current program counter is generated using nested if-statements when transforming the binary. Besides the maximum size of a method, some optimization methods discussed in the next section work on sequences of instructions which impacts the optimal position for splitting the code. Finally, additional code that checks the event queue adds to the statements for each instruction handling the pure simulated execution of an instruction.

3.1 Lazy Timer Evaluation

The frequency of taking the direct path from the code of one instruction to another depends mostly on the number of events generated for the event queue at runtime. Most of the event types correspond to the simulation of hardware interrupts, for example, the notification of the radio of an incoming packet. The vast majority of events, however, is generated to facilitate the ticks of the simulated hardware timers. The simulated microcontroller has several configuration settings determining the behavior of its timers. Depending on this configuration – usually performed by the operating system – there are often just a few cycles between each increment for one of the timers, which in turn results in frequent 'housekeeping' operations between the code for individual instructions.

To reduce the number of generated events, we use the fact that the actual timer value at a given time usually has no relevance on the program flow with the exception of two cases: First, for the usual configuration, only certain values of the timer – again configured by the operating system – trigger a hardware interrupt, which requires the simulator to change the program flow of the binary. Second, the simulated microcontroller allows reading the current value of the timer by accessing a well-known address that instead of being backed by RAM is used to access hardware timer. Similarly, accessing dedicated addresses performs the configuration of the timer, which is already handled by the emulator and, for example, changes the frequency of inserting events to speed up the timer. We optimize this process by extending the configuration routine with dedicated functionality to handle the common configuration differently: instead of inserting an event for each increment of the timer, the component inserts events only when a hardware interrupt needs to be simulated. Additionally, when the software accesses the current value of the timer at runtime, the value is calculated by taking into account the last time a hardware interrupt was triggered, the configured frequency of the timer, and potential overflows.

4. OPTIMIZATIONS

The translation process in the previous chapter provides the foundation for additional optimizations based on static analysis of the binary. Our design includes an overall frame-

work where optimization steps can be added including utility functions to reason about the code. Additionally, several of the following optimizations consist of a generic part with one or more concrete implementations. On the one hand, they can be extended with additional implementations, for example, additional code replacement strategies. On the other hand, they provide a foundation for the implementation of comparable optimizations for other simulated hardware platforms.

4.1 Static Evaluation of Conditions and Removal of Runtime Checks

To simulate the behavior of the emulated hardware correctly, some superficially simple operations require checking of conditions and handling of special cases or border cases. The most prevalent example is access to the RAM of the device. These operations require boundary checks to make sure that the emulator only accepts valid addresses. Furthermore, both reads and writes to certain memory regions on the device do not simply read or change the value of a byte in RAM, but instead trigger effects on other subsystems of the micro controller. This includes components such as ADCs, I/O ports, and hardware timers. A write can, for example, change the frequency of a hardware timer, which requires the corresponding logic in the emulator to adapt. Then again a read may actually trigger the delivery of a byte from the simulated radio.

To correctly reflect the behavior of the hardware, these operations are handled by utility functions of various complexities. To remove unnecessary checks, we added alternative dedicated versions of some of these functions that contain optimized code paths. For example, if the target address of a RAM access is known to be in a non-critical region, the emulation code can be almost reduced to a simple array access. Finally, if the utility function used at a certain point is as trivial as an array access, the function definition is inlined and the code contains the access directly.

While the individual savings are sometimes rather small, i.e., mostly removing condition checks and the associated branches, the frequency of the use of this function is so high that the sum of the savings can be significant. We show in the evaluation that – compared with the other optimizations – this technique has often the biggest impact on the performance of the simulator.

4.2 Constant Folding and Constant Propagation of Immediates

The so-called immediates are constant operands of instructions. The simplest case is loading a register with a constant, but they are for example also operands in jump instructions – both absolute and relative. For our test programs, usually around a quarter of the instructions use an immediate. Since this value is known when code for a specific instruction is generated, this knowledge is used to apply constant folding and constant propagation: variables representing the immediate operand are replaced with their value and expressions are iteratively evaluated and in turn replaced by a constant value if possible.

A special case of this optimization that applies to all instructions is the current instruction pointer. Since this value is always the address of the current instruction, it can be consistently replaced at generation time with its value and directly used, for example, when emulating jump instructions or when incrementing at the transition to the code for the subsequent instruction.

4.3 Peephole Optimization of Individual Instructions

There are certain instruction/operand combinations that allow further reduction in the emulation code compared to the constant folding approach discussed above. A somewhat frequently occurring example is the use of the exclusive-or instruction where both operands use the same register to set the register to zero. As with other arithmetic instructions, the correct emulation does not only require the calculation of the result but also the computation of the values of the various registers of the simulated processor. However, in the case of this instruction/operand combination and a few others, the result as well as the value of the flags is already known at generation time and the statements can be reduced to simple assignments of constants.

4.4 Conditional Execution of Potentially Unnecessary Code

For arithmetic and logical instructions, the code necessary to calculate the resulting values of the simulated processor flags often exceeds the code to compute the value for the target register and, therefore, takes up a considerable amount of computation of the emulator. If the subsequent instruction also changes the values of a subset of the flags and does not rely on some of them, we can generate optimized code for the first instruction that omits the computation of unnecessary values. However, in general it is possible that after the emulation of each instruction, the emulator has to handle an event, for example, to generate a simulated interrupt. To ensure consistency and correctness, in this case the values of all simulated flags must be computed and stored. For this reason, the generated code still contains the calculation of all flags but moves them to a conditional block that is only executed if events of the queue are ready to fire. In Fig. 2, we show an example where an 'add-with-carry' (ADC) instruction is followed by a 'subtract-immediate' (SUBI) instruction. The left side shows the associated code of Avrora that simulates the instruction execution. The right side shows the optimized version generated by Notos where the majority of the statements necessary for the ADC instruction are moved to a conditional block that is only executed if an event is triggered before the subsequent SUBI instruction.

Aside from bookkeeping (for example, updating the instruction pointer), the code for an arithmetic or logical instruction can be divided into three parts: first, the values for the operands (e.g., register values) are fetched and stored in local temporary variables. Second, additional intermediate variables are calculated. Third, the destination operand and the flags are assigned various computation results combining these intermediate variables. The resulting code is a sequence of assignments with expressions of varying complexity. For example the 'add-with-carry' instruction uses 9 intermediate variables and comprises 15 statements.

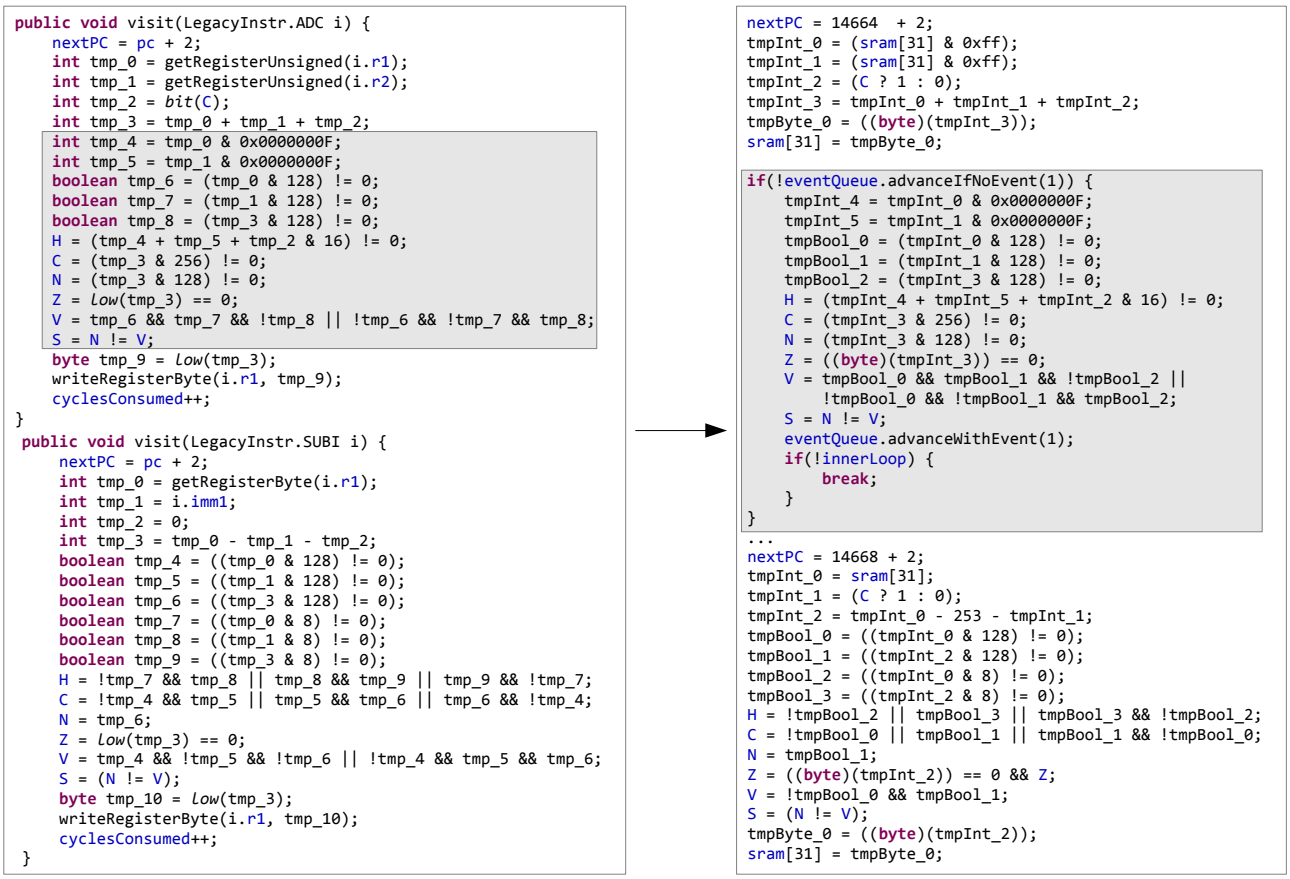


Figure 2: Example for conditional execution of potentially unnecessary code

For this optimization, the generation algorithm performs the following steps for two subsequent instructions A and B. First it constructs an inverse dependency graph: a directed acyclic graph (DAG) where each temporary variable of both instructions is a node and there is a node for each flag used in instruction A and for each flag used in instruction B. For each assignment, the algorithm adds an edge each node corresponding to a variable on the right to the node corresponding to the variable on the left. The graph is cycle free because a flag used in both instructions is represented by 2 dedicated nodes and otherwise the code could not compile for the temporary variables. The algorithm marks all nodes that represent flags that are not written by instruction B and then removes all nodes reachable from these marked ones. Finally, the algorithm has to filter out flags that are read in instruction B before written in B. For this, the algorithm checks for each flag read by B, if a path exists that reaches a flag written by A without a write by B in between. If such a path exists, the algorithm removes the node for the flag and all nodes reachable from it. The final graph contains the flags, that – assuming no simulation event occurs between the instructions as discussed above – do not have to be written in instruction A and also reachable from them the intermediate variables that are only used to calculate these.

This graph is used to split the code for instruction A into two parts: the first part – consisting of the statements not reflected in the remaining graph – always executes and the second part with the statements left in the graph is moved into an conditional block and only executes when the event queue fires. As a final constraint, the algorithm only divides the statements into two blocks if the number of statements in the conditional block is over a threshold. This constraint makes sure that the overhead by the condition check itself, which involves access to the data structure of the event queue, does not offset the possible savings.

4.5 Optimized Code Paths for Instruction Sequences

Following in the vein of the previous technique, the generator also optimizes code across multiple instructions. In this case, the analyzer looks for instruction sequences that do not contain any branches – so that the number of cycles spent in this block is constant – and provides an alternative code path. The first instruction of such a sequence checks if the lookahead indicates that no event can occur for the necessary number of cycles. If true, the emulator uses the code path that combines all instructions in this sequence with reduced code to only produce the end results and necessary intermediate steps.

Compared to the previous technique, these so-called shortcuts have constraints both at generation time and at run time that make them a much rarer occurrence than the optimization for subsequent instructions. Therefore, the further reduction in executed code is usually distinct but not extraordinary.

4.6 Replacing Common Instruction Sequences with Manually Optimized Code

While the two previous optimization techniques that tackle the reduction of executed code work automatically, we also included a framework for providing manually optimized functions to replace common instruction sequences. This targets functions of the standard library, artifacts of compilers, and operating system functionality. The framework supports the definition of a sequence of instructions with placeholders for operands, for example, to indicate that the same register must be used at certain places, but the actual register number is not specified. This code is then replaced by another manually defined code path where the placeholders are replaced by the actually used argument. This code path must then calculate the result. Additionally, it must also provide information about the number of cycles of the replaced code – possibly depending on the input values at runtime. This value is used to update the cycle counter at the end of the calculation and, similarly to the previous approach, to verify beforehand if the sequence can be executed without an event occurring in between.

As an example, we analyzed the instructions of the avr-gcc [20] standard library function for calculating an integer division. Since the target platform does not support this as a processor instruction, it takes between 193 and 209 cycles for the given implementation to calculate this. On the host platform, however, this is usually a simple operation of the CPU. Taking into account the further overhead of the emulator, is was possible to achieve a 99% reduction from around 2800 executed statements to around 20.

This example, however, also highlights the limitation of this approach. While the savings can be significant when just looking at the function itself, the function must be executed frequently to have impact of the overall execution time and most importantly this approach requires considerable manual effort. In this case, the calculation of the result – both registers and flags – was relatively simple, but the number of cycles necessary depends on the number of 1 bits in the result of the division, which was not obvious at first sight.

5. EVALUATION

In this section, we present the evaluation of the different algorithms with respect to a number of varying factors. We run all tests on desktop PCs with 2.8 GHz quad-core CPU supporting simultaneous multithreading. The systems are equipped with 8 GB of RAM and run a 64-bit Linux and a 64-bit Java virtual machine using 2 GB of RAM.

The bytecode approach with just-in-time compilation (JIT) adopted by the Java virtual machine makes it hard to obtain deterministic and predictable results in general. Especially, the performance differs greatly between the beginning of the execution and the state where most of the critical-path code has been transformed by the JIT compiler. For simulations that often run several minutes or hours, the performance of this so-called steady state is the determining factor for evaluating the performance of the simulator. Therefore, we follow the guidelines for evaluating the steady state of Java-based systems by Georges et al. [6] that recommend a warm-up phase where the code is executed before the actual measurements take place. Within the execution of a virtual machine, first we run the simulation twice to trigger the JIT system. Afterwards, we record the times of 5 sequential executions of 300 simulated seconds presenting unless otherwise noted the average, minimum, and maximum time of these tests in seconds.

We omit the time necessary for generating and compiling the class representing an application in the simulator. For complex applications, this is a constant overhead of approximately 20 seconds, which is significant for isolated short simulation runs and can outweigh any benefit of our approach. To alleviate this overhead to some extent, we included a caching mechanisms so that the simulator generates a new class only when the binary changes and the generation is a transparent process that does not require intervention from the user. Therefore, the numbers presented provide a better foundation for estimating the difference between the approaches for long-running simulations and for running multiple simulations of the same application, for example, varying network topologies – both cases where the simulator performance is a critical factor.

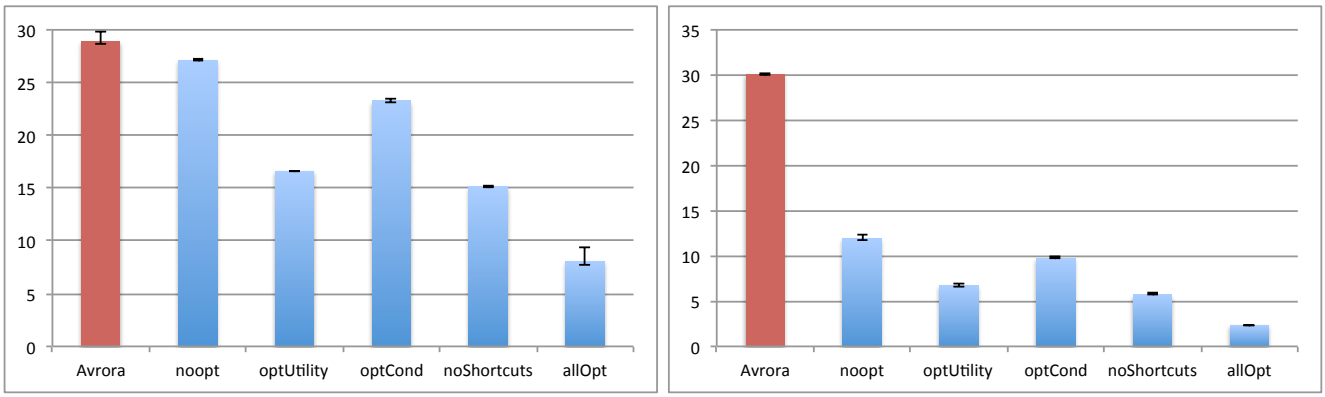
5.1 Factors

We vary the following factors in our evaluation: the execution model (Avrora and Notos), the Notos optimizations, the simulated application, and the number of simulated nodes.

We use the following terms in the graphs to refer to different execution models and enabled optimizations.

1. *Avrora* the baseline existing emulator.
2. *noOpt* Notos performing the binary-to-source translation as described in Section 3 without any further optimizations
3. *optUtility* Notos with the optimization of utility functions by static evaluation of conditions and removal of runtime checks
4. *optCond* Notos with conditional execution of potentially unnecessary code
5. *noShortcuts* Notos with all optimizations except shortcuts
6. *allOpt* Notos with all optimizations

For the applications, we include a simple infinite loop of multiplications as a simple test for the core execution engine. Additionally, we evaluate two different applications from the wireless sensor networks domain. As the foundation, we use the TinyOS example application MViz that periodically senses data and uses CTP [7] to forward them to a sink and, therefore, is a good representative for a typical sensor



(a) Infinite Loop

(b) MVizComplex

Figure 3: Single-node simulation performance (wall time in seconds)

network application. MViz does not use low power listening by default. We evaluate a second variant 'MVizComplex' that enables LPL and additionally performs long-running computations on the node, which represents advanced sensor network applications that make use of in-network processing.

Besides evaluating the performance for the simulation of a single node, we consider network sizes between 16 and 100 nodes.

5.2 Optimization Parameters

Before analyzing the performance of our approach with respect to Avrora, we discuss here two significant parameters of the translation process: the maximum method size for the generated code and the minimum shortcut length for the optimization discussed in Section 4.5.

As discussed in Section 3, the Java runtime puts a hard limit on the maximum size of a method and a second hierarchy level is required to transform the binary of larger applications. Since the second level requires more computations to reach the correct code for a given address, our initial design tried to maximum the size of each method. However, initial tests included some outliers where the simulation of some applications resulted in comparably weak performance. This is caused by implementation-defined limits on the complexity of a method that prevents some stages of the JIT process to take place. We analyzed the optimum method size with several applications and found that a maximum of 45 kB is safe while still limiting the overhead incurred by the second hierarchy level.

When choosing optimized code paths for instruction sequences, a tradeoff exists between the potential of skipping statements at runtime on the one hand and on the other hand the additional overhead for deciding this by inspecting the event queue – which incurs every time this piece of the code is executed. Furthermore, the likelihood of actually taking a shortcut decreases with its length because the probability that no event is queued for the requested time span decreases. Therefore, the optimization algorithm takes two parameter into account: the minimum shortcut length, i.e., how many instructions are compressed and the minimum reduction in size when taking the shortcut.

5.3 Single-Node Emulation Performance

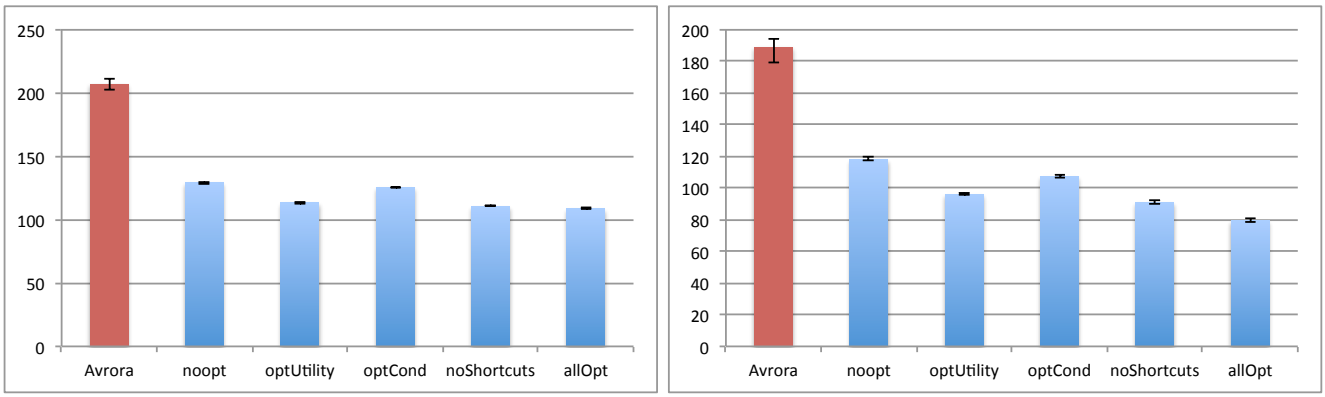
In Fig. 3, we show the the results for emulating a single node for the infinite loop application and the MVizComplex application. While Notos with all optimizations outperforms Avrora significantly in both cases, Notos without optimizations is just 5% faster than Avrora. Since the loop that requires almost all simulation time is rather small, the improved performance when just simulating instructions sequentially has little impact on the result, also because Avrora uses an already optimized execution engine that decodes all instructions once beforehand. However, especially the optimization of utility functions that, for example, streamlines the access to registers and RAM, significantly reduces the runtime of Notos.

For the more representative MVizComplex application, Notos without optimizations performs already significantly better than the baseline. Similarly to the infinite loop example, the optimization of the utility functions has the most significant impact as it influences the biggest part of the simulated binary.

While showing significant gains in this case that highlights the potential of this approach, these evaluation results are of limited practical value and we focus in the following on the performance when simulating small and big networks of sensor nodes.

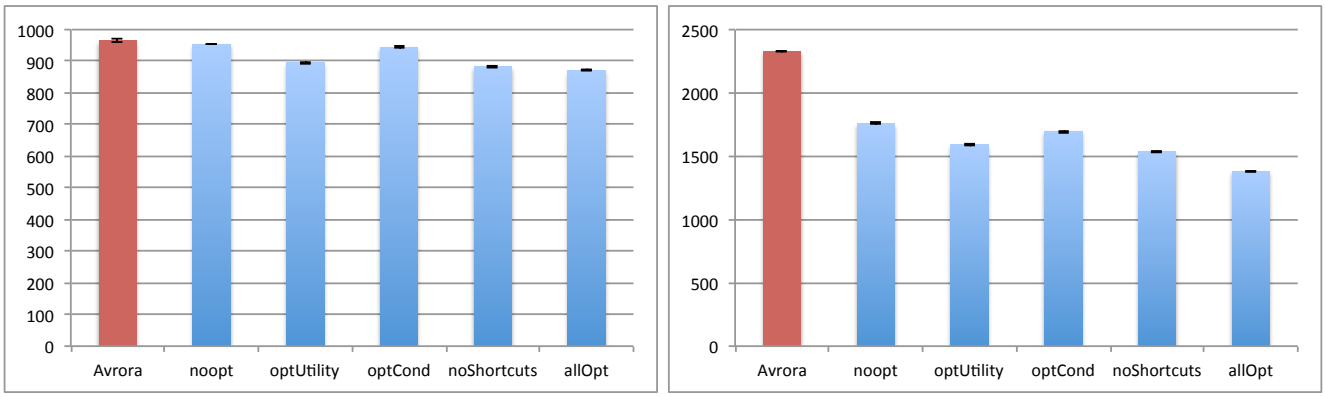
5.4 Network Emulation Performance

In Fig. 4, we show the result for a small network simulation using 16 nodes. MViz is a good representative for an almost pure network protocol application. While Notos performs considerably better than Avrora in all cases, the difference is significantly smaller than considering a single node and the gain by the optimizations decreases, also due to the impact of the lazy timer implementation on the synchronization overhead. It is noteworthy, that the impact of the shortcut optimization decreases in this case. This stems mostly from the application, since this optimization works on instruction sequences that are more common in computation-



(a) Mviz (b) MVizComplex

Figure 4: Network simulation performance for 16 nodes (wall time in seconds)



(a) Mviz (b) MVizComplex

Figure 5: Network simulation performance for 100 nodes (wall time in seconds)

heavy applications compared to the branch-focused protocol implementations.

MVizComplex combines the operation of a network protocol with additional computations as required, for example, in civil engineering monitoring systems [4]. This provides more potential for the optimizations which results in higher increase in speed compared to the Notos baseline. Additionally, the shortcut optimization has more impact. Nevertheless, also in these cases the static analysis of conditions and removal of runtime checks is the optimization with the biggest impact.

Fig. 5 highlights the shift in simulation overhead from interpreting the instructions of the individual nodes to ensuring consistency and handling the simulated communication in a larger simulated network of in this case 100 nodes. For MViz, while Notos with all optimizations still outperforms Avrora by around 10%, the difference is just 1% for the non-optimized version.

For MVizComplex the advantage is still much more pronounced as Notos can take advantage of the higher ratio between time spent by the emulator for interpreting the instructions of the target binary and the time spent for handling communication and synchronization.

6. RELATED WORK

The advent of research in wireless sensor networks has been accompanied by research on how to simulate them with high fidelity and high performance spurred by the difficulty and cost associated with the development and deployment of real systems.

ATEMU [18] was one of the first cycle-accurate emulators for wireless sensor networks. However, ATEMU does not support parallel emulation and, thus, does not benefit from multi processor or multi core systems. VMNet [27] enhances ATEMU with support for host-networking but does not address the scalability issue.

There are three major approaches to increase the speed of wireless sensor network simulators. First, parallel emulation uses a multithreaded approach to exploit multiple core or

CPUs in one system. Our work is based on Avrora [21][22], a popular cycle-accurate emulator that supports parallel simulation using threads and contains a large number of functions to support debugging and evaluation of sensor node software. With our approach, we replace the core execution engine for the interpretation of the target platform instructions. PolarLite enhances the Avrora emulator with an optimization that takes the sleep times into account [9] and with support for using software state from the TinyOS 1.x MAC protocol at runtime to compute bigger lookaheads [10]. With Boreas [19], we investigated alternative synchronization methods. Both approaches focus on improving the parallelism of the simulation and are largely orthogonal to our approach of improving the speed of simulating one node with binary-to-source translation.

A second approach uses distributed emulation to increase the performance. DiSenS [26] uses a cluster of hosts to emulate a network. However, the performance is strongly dependent on the topology as explained in the paper. LazySync [11] aims at reducing the number of clock updates in a distributed emulation. WorldSens [5] is another distributed emulator that uses an optimistic approach to increase efficiency. In general, binary-to-source translation can be used for any kind of emulation – sequential, parallel, and distributed – as it tackles the core execution engine that all emulators must implement.

A third approach to improve the speed of simulation is to increase the abstraction level. TOSSIM [14] provides 'same source' simulation for TinyOS by replacing hardware dependent parts with simulation components. This allows simulating the final application code of TinyOS applications but does not provide the fidelity, analysis details and operating system independence of cycle-accurate emulation. TimeTossim [13] extends TOSSIM with accurate timing by instrumenting the code based on debugging information gained from compiling for the target platform. This approach is suitable if the replacement of part of the platform code by simulation code and the provided accuracy is adequate for a scenario. COOJA [17] provides cross-level simulation, i.e., the simulation of a network of nodes where the abstraction levels of the nodes can range from emulation, over source code simulation to nodes developed in Java. Emulation is accomplished by using MSPsim [3] and source code simulation is coupled to the Contiki operating system. This approach allows for example emulating only a part of a network while the other nodes are simulated. This increases the speed but also decreases the fidelity for the overall simulation. SenQ [24] and [25] focus on the realism of simulation by combining a TOSSIM-like approach with an established simulation platform to benefit from established physical layer models, battery models and clock drift.

Binary-to-source translation and more generally code morphing and just-in-time compilation have been successfully used both in research and real-world applications. Widely used examples for the latter include the emulators provided by Mac OS for the transition from the 680x0 architecture to PowerPC and later from PowerPC to x86. Just-in-time compilation is a cornerstone for most recent programming languages and runtimes such as Java and the .NET framework. QEMU [1] transforms target binaries using small pieces of code that implement so-called micro-instructions for an abstract platform and uses the compilation of this to generate a dynamic translator. In [12], the authors discuss optimizing a translation solution to target simpler embedded processors. SPIRE [8] is just one recent example investigating optimization techniques for binary translation systems. Most approaches target running user mode applications from a different platform on a host operating system. However, to guarantee reproducibility of the simulation of a network, we target cycle-accurate emulation which constrains the possibility of optimizations. To some extent, our approach to transform the binary to source code also benefits from progress in this area via the evolution of the Java virtual machine, similarly to how QEMU benefits from progress of the optimizations performed by the compiler.

7. CONCLUSIONS AND FUTURE WORK

The evaluation highlights both the potential and the limits of improving the performance of cycle-accurate emulation of wireless sensor networks with binary-to-source translation. On the one hand, our approach that transforms a binary for the target platform to source code for the host platform of the simulator can increase the speed of the actual interpretation of the instructions considerably. More importantly, this provides the foundation for static analysis and binary-specific optimizations that can significantly decrease the simulation time – for networks with 16 nodes, by around 50%. On the other hand, the evaluation also highlights the shift of the overhead from the interpretation of the individual instructions to simulated communication when simulating networks. With increasing network size, the synchronization overhead increases and the impact of our approach decreases. Furthermore, due to the constant overhead for generating and compiling the source code for the application-specific class – around 20 seconds for complex applications –, our approach is not suitable for a single short simulation that only runs a few minutes. If, however, an application or protocol is evaluated with different settings, for example, network sizes or more general network topologies, even short simulations in sufficient numbers can benefit from our approach as the overhead incurs only once.

The amount of optimizations performed depends on the simulated application. In general, our approach works best with smaller number of branches which translates to applications that spend more time for computation heavy tasks, for example, performing in-network processing. Therefore, our approach is well suited for simulations of small to medium sized networks for applications that contain a noteworthy amount of arithmetic instructions.

For the future, we are planning to evaluate extending the overall optimization framework as well as adding more concrete implementations for the individual generic parts. Another interesting possibility is partially automating the process of generating fast alternatives for common code sequences such as standard library functions and operating system processes.

Acknowledgments

This work has been partially supported by the FP7 projects SMARTKYE (smartkye.eu) and BESOS (besos-project.eu) funded by the European Commission.

8. REFERENCES

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conf.*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association.

[2] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Nov. 2004.

[3] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. Mspsim – an extensible simulator for msp430-equipped sensor boards. In *Proc. of the European Conf. on Wireless Sensor Networks (EWSN), Poster/Demo session*, Jan. 2007.

[4] K. Flouri, O. Saukh, R. Sauter, K. E. Jalsan, R. Bischoff, J. Meyer, and G. Feltrin. A versatile software architecture for civil structure monitoring with wireless sensor networks. *Smart Structures and Systems*, 10(3):25–46, Sept. 2012.

[5] A. Fraboulet, G. Chelius, and E. Fleury. Worldsens: development and prototyping tools for application specific wireless sensors networks. In *Proc. of the 6th Intl. Conf. on Information processing in sensor networks*. ACM, 2007.

[6] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications*, OOPSLA '07, New York, NY, USA, 2007. ACM.

[7] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proc of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys'09)*. ACM, 2009.

[8] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. Spire: Improving dynamic binary translation through spc-indexed indirect branch redirecting. In *Proc. of the 9th ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments*, VEE '13, New York, NY, USA, 2013. ACM.

[9] Z.-Y. Jin and R. Gupta. Improved distributed simulation of sensor networks based on sensor node sleep time. In *Proc. of the 4th IEEE Intl. Conf. on Distributed Computing in Sensor Systems*. Springer-Verlag, 2008.

[10] Z.-Y. Jin and R. Gupta. Improving the speed and scalability of distributed simulations of sensor networks. In *Proc. of the 2009 Intl. Conf. on Information Processing in Sensor Networks*, IPSN '09. IEEE, 2009.

[11] Z.-Y. Jin and R. Gupta. LazySync: A New Synchronization Scheme for Distributed Simulation of Sensor Networks. In *Proc. of the 5th IEEE Intl. Conf. on Distributed Computing in Sensor Systems*. Springer, 2009.

[12] G. Kondoh and H. Komatsu. Dynamic binary translation specialized for embedded systems. In *Proc. of the 6th ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments*, VEE '10, New York, NY, USA, 2010. ACM.

[13] O. Landsiedel, H. Alizai, and K. Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *Proc. of the 7th Intl. Conf. on Information processing in sensor networks*. IEEE, 2008.

[14] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc of the 1st Intl. Conf. on Embedded networked sensor systems*. ACM, 2003.

[15] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*. Springer Berlin Heidelberg, 2005.

[16] The Network Simulator NS-2, http://www.isi.edu/nsnam/ns.

[17] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Proc. of the 31st Annual IEEE Conf. on Local Computer Networks*. IEEE, 2006.

[18] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras. ATEMU: a fine-grained sensor network simulator. In *Proc. of the 1st Annual IEEE Comm. Society Conf. on Sensor and Ad Hoc Communications and Networks (SECON 2004)*, October 2004.

[19] R. Sauter, R. Figura, O. Saukh, and P. J. Marrón. Boreas: Efficient synchronization for scalable emulation of sensor networks. In *Proc. of the 8th IEEE Intl. Conf. on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2011)*. Valencia, Spain, 2011.

[20] R. M. Stallman et al. *Using the GNU Compiler Collection*. Free Software Foundation, 2014.

[21] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. symposium on Information processing in sensor networks*. IEEE Press, 2005.

[22] B. L. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *Proc. of the 2005 ACM SIGPLAN/SIGBED Conf. on Languages, compilers, and tools for embedded systems*, LCTES '05. ACM, 2005.

[23] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proc. of the 1st Intl. Conf. on Simulation Tools and Techniques for Communications, Networks and Systems*, Simutools '08, Brussels, Belgium, 2008. ICST.

[24] M. Varshney, D. Xu, M. Srivastava, and R. Bagrodia. Senq: a scalable simulation and emulation environment for sensor networks. In *Proc. of the 6th Intl. Conf. on Information processing in sensor networks*. ACM, 2007.

[25] Y.-T. Wang and R. Bagrodia. Scalable emulation of tinyos applications in heterogeneous network scenarios. In *Proc. of the 6th Intl. Conf. on Mobile Adhoc and Sensor Systems*. IEEE, 2009.

[26] Y. Wen, R. Wolski, and G. Moore. Disens: scalable distributed sensor network simulation. In *Proc. of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007.

[27] H. Wu, Q. Luo, P. Zheng, and L. M. Ni. VMNet: Realistic Emulation of Wireless Sensor Networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(2), 2007.