When Timing Matters: Enabling Time Accurate & Scalable Simulation of Sensor Network Applications

Olaf Landsiedel, Hamad Alizai, Klaus Wehrle
Distributed Systems Group
RWTH Aachen University, Germany
{olaf.landsiedel, hamad.alizai, klaus.wehrle}@rwth-aachen.de

Abstract

The rising complexity of data processing algorithms in sensor networks combined with their severely limited computing power necessitates a in-depth understanding of their temporal behavior. However, today only cycle accurate emulation and test-beds provide a detailed and accurate insight into the temporal behavior of sensor networks.

In this paper we introduce fine grained, automated instrumentation of simulation models with cycle counts derived from sensor nodes and application binaries to provide detailed timing information. The presented approach bridges the gap between scalable but abstracting simulation and cycle accurate emulation for sensor network evaluation.

By mapping device-specific code with simulation models, we can derive the time and duration a certain code line takes to get executed on a sensor node. Hence, eliminating the need to use expensive instruction-level emulators with limited speed and restricted scalability. Furthermore, the proposed design is not bound to a specific hardware platform, a major advantage compared to existing emulators. Our evaluation shows that the proposed technique achieves a timing accuracy of 99% compared to emulation while adding only a small overhead. Concluding, it combines essential properties like accuracy, speed and scalability on a single simulation platform.

1 Introduction

In recent years, the number of sensor network deployments and applications in use have registered a sharp increase. Furthermore, from the basic deployments several years ago, that were mostly recording and transmitting simple measurements [23], the complexity of deployed applications has heavily increased [4, 20, 28]. This complexity requires a thorough evaluation of the application and the underlying operating system to ensure that it provides the required functionality. Similarly, it is of strong interest to evaluate how an application operates under heavy load and whether it can process the required number of events and tasks such as sensing, packet forwarding, and data aggregation.

Commonly, due to its high performance, scalability, and independence from the hardware platform, simulation is used to evaluate algorithmic functionality of sensor network applications. However, its high level of abstraction and therefore lack of detailed timing prohibits the use of simulation to evaluate the exact timing and computational load of algorithms and to determine limits of a hardware platform envisioned for deployment. Currently, only emulators and test-beds provide such detailed information about the behavior of the application in "real life". However, both have a set of specific limitations and disadvantages.

An emulator imitates a specific hardware platform, i.e., it provides a software based implementation of all platform components and executes the binary compiled for the specific sensor node. When an emulator implements all features of the emulated platform correctly, it is inherently cycle accurate and therefore of high use for a detailed evaluation of applications and operating systems. Exploiting the fact that the platform exists virtually, a user can explore states in registers, memory and corresponding transitions by stepping through individual instructions. However, as a result, emulation is heavyweight and compared to simulation it strongly limits performance and scalability. Additionally, an emulator is written for one specific platform. Commonly, adding new platforms results in major porting efforts. Avrora [24, 12] aimed to tackle this portability challenge by providing generic emulator building blocks such as registers, memory, and timers. However, even the subtle differences between various Atmel AtMega platforms made adding new microcontrollers from the same family work intense; not to mention the complexity of adding new microcontroller types or radios. Furthermore, as mentioned above, an emulator is only accurate when it implements all instructions and operation modes without any limitations or bugs. However, this seems to be challenging as Avrora still has trouble to emulate TinyOS 2.x [26] radio communications while TinyOS 1.x [14] workes fine.

Next to emulation, test-beds are typically used for a detailed evaluation of applications and operating systems. Compared to real world deployments, test-beds are limited in size, but provide functionality for event logging and injection as well as additional wired or wireless communication channels for feedback, interaction and debugging. Obviously, test-bed results are highly realistic. However, commonly test-beds contain some tens of nodes and therefore the scalability of the proposed design and operations with a large number of interactions are hard to evaluate. Furthermore, repeatability, controllability and system insight are limited and test-beds are quite cost and space intensive.

In this paper we show that by automatically instrumenting the simulation model with cycle counts, we can unite the advantages of simulation and emulation. We provide near cycle accurate timing combined with the scalability, flexibility and portability of simulation. Overall, we reach an accuracy of over 99% compared to emulation while adding only a small performance overhead compared to typical sensor network simulators. Furthermore, the presented design and implementation is independent from specific sensor network platforms and operating systems, ensuring easy adaptation to various platforms and systems.

Typical sensor network operating systems such as TinyOS, SOS [10], or Mantis [2] use the same code base for simulation and on the devices themselves. For simulation hardware specific device drivers are replaced with a slim simulation wrapper. The observation that in such a system large percentages of the source code in the device and simulation are identical provides the basis for the approach presented in this paper. It enables us to automatically instrument the simulation code with timing information, such as cycle counts, derived from the code compiled for the sensor node.

The reminder of this paper is structured as follows. Section 2 presents emulation and simulation systems that aim for accurate and detailed evaluation of sensor networks and compares our approach. Next, Section 3 introduces automated code-instrumentation to enable accurate timing in sensor network simulations. Section 4 discusses implementation details, limitations and integration into TinyOS. A detailed evaluation in Section 6 compares the achieved results with emulation. Next, we discuss future work in Section 7. Section 8 concludes the paper.

2 Related Work

In the past few years a great deal of effort has been invested in the design and development of simulators to embrace the special requirements imposed by the highly distributed and dynamic nature of sensor networks. Unfortunately, all of these efforts have made compromises over different attributes of simulation. For example, accuracy has been compromised over scalability and vice versa. SWAN [17], SensorSim [15], SENS [22], and TOSSIM [13] are examples of discrete event simulators for sensor networks which compromise accuracy over scalability by using non-figurative models of the sensor nodes. Such simulation models are the basis to quantify network delays, throughputs, and packet collisions. However, these models do not reveal the timing and interrupt properties of applications, operating systems, and hardware components which are extremely important for examining resource constrained sensor networks.

ATEMU [19], Avrora, Worldsens [3], DiSenS [27] and others [9, 25, 11] on the other hand are cycle-accurate instruction level emulators for sensor networks with the most expressive models. Nevertheless, they compromise scalability and performance. ATEMU is 30 times slower than TOSSIM, and its poor performance limits its scalability to about 120 nodes. Avrora, because of its multi-threaded architecture shows better performance measures than ATEMU when run on a multi-processor machine. However, on multi-processor machines it is still 50% slower than TOSSIM. Furthermore, Avrora shows typical performance bottlenecks of instruction level emulators when run on a customary end-user machine and can be up to a hundred times slower than simulation as shown by the results of our performance evaluation (see Section 6.3). DiSens and Worldsens suffer from similar scalability problems and use distributed simulation to address it. Thus, they do not only benefit form multi-processor environments, but can also be executed in clusters. Nonetheless, this distribution requires a high degree of fine grained synchronization and therefore limits scalability. Furthermore, such emulation environments have reached a complexity which is an order of magnitude higher than the system to evaluate, i.e. the sensor node. As a result, such cycle accurate emulators are hard to maintain, extend and debug. Furthermore, they are bound to one specific platform and hard to port.

Finally, test-beds such as Trio [6], Mirage [5], MoteLab [29], or Kansei [7] are commonly used for in-deep sensor network evaluation. To provide the user with detailed feedback, they use wired backchannels or even wireless connections via a second radio chip operating with a different technology or frequency band [1]. Nonetheless, high costs of test-beds and binding to a certain platform combined with limited scalability, insight, reproducibility and control cannot replace emulation or simulation as important means of evaluation.

Although designed for energy modeling instead of time accuracy, PowerTOSSIM [21], uses offline code instrumen-

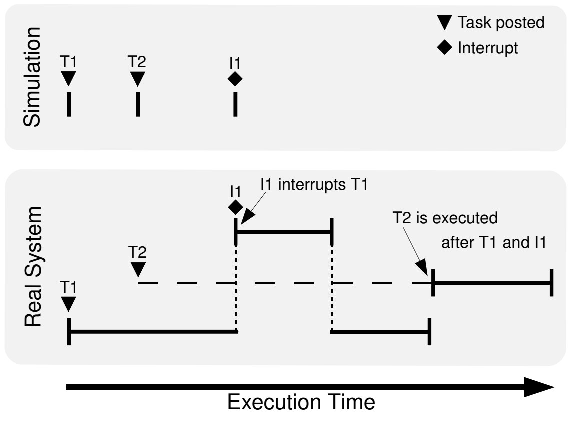


Figure 1. Simulation vs. real world event execution: In reality – and in contrast to simulation – event execution consumes time and events may interrupt or delay each other.

tation on basic block level to predict the power consumption of sensor nodes. The approach presented in this paper is based on similar techniques. However, we generalize it to provide online instrumentation and dynamic event queue adaptation compared to offline (after the simulation) modeling in PowerTOSSIM. Furthermore, we provide a more fine grained instrumentation level and features – such as energy models – can be easily derived from the detailed timing model presented in this paper.

Compared to existing work, automated simulation code instrumentation – as presented in this paper – provides the accuracy of emulation while perpetuating the key properties of simulation such as scalability and easy adaptation to new sensor node platforms and operating systems.

3 Enabling Time Accuracy

Classic simulation models the behavior of a system at event granularity. It translates all events, e.g. interrupts and tasks in TinyOS, into discrete simulator events. Events are executed one after another. Thus, time in simulation is handled discretely; at the beginning of an event the simulation time is set to the execution time of the event and remains unadjusted throughout the event execution. Therefore, events in simulation take zero execution time. However, in real life events have an execution time and may interrupt, interfere or delay each other (see Figure 1), resulting in different execution and completion order compared to simulation. Under peak loads, this may even load to event misses on interrupts and tasks.

Summarizing, simulation only contributes to testify the algorithmic functionality of an application. It is unable to provide any assistance in evaluating the performance of a hardware platform and modeling the much important timing and interrupt properties of applications. Especially, when the application is executed on a resource constrained embedded platform such as sensor nodes, timing of interrupts significantly impacts the performance of applications. Thus, due to the lack of time accuracy in modeling a system, false-positives about the performance of applications are inevitable in simulation.

3.1 Fine Granular Simulation Clock

As events can delay and even interrupt each other, modeling on a fine grained level is necessary to ensure the required accuracy, a property that today only cycle accurate emulation – executing hardware specific binaries – can provide. In this paper we propose automatic instrumentation of each source code line in the simulation model with its execution time. In our evaluation (see Section 6) we show that code instrumentation on source code line granularity reaches an accuracy of 99% compared to emulation while adding only a small overhead to the simulation performance.

We resolve timing discrepancy of sensor network simulation and emulation by enabling simulation to track the system time during event execution. Our proposed solution determines the execution time (clock-cycles) of each source-code line being executed inside a simulator event and then increments the simulation time accordingly. The underlying technique is to automate the mapping between simulation source-code and the platform specific executable. This is only possible when nearly identical application and operating system code is executed in simulation and on the hardware platform, which is typically the case in sensor network operating systems. Such a mapping enables us to identify the processor instructions corresponding to a source-code line. From the respective processor data-sheet we next retrieve the number of cycles consumed by each instruction and therefore can compute the time to execute each source code line on the sensor node platform.

The code mapping technique is particularly suited for embedded CPUs (such as in sensor nodes) employing sequential instruction execution without any pipelining and caching strategies. For such platforms, the execution time of a binary instruction is static and can be modeled without interpreting each individual instruction.

As our design only instruments code on source code line granularity and not on instruction level, it has one limitation; it does not completely model the instructions that are only partially executed such as logic operations. Patching the corresponding compilers to add further code annotations extends the accuracy to instruction level. However, due to following reasons, we decided to base our prototype implementation on source code line granularity without compiler modifications: (1) not to require compiler extensions ensures easy portability and adaptability to new sensor node platforms, (2) our evaluation shows, the impact on accuracy is limited on typical sensor network applications as external events cause an automatic re-synchronization.

3.2 Delaying and Interrupting Events

Tracking system time during event execution may result in overlapping events and only helps in determining the execution time of each event separately. However, the overall timing and interrupt behavior of an application still remains undetermined. For example, in TinyOS tasks are executed sequentially and therefore can delay each other's execution. However, interrupts are executed immediately and delay the execution of any currently active task (see Figure 1). To accurately model the behavior under peak loads, interrupts and tasks are dropped when their corresponding queues overflow. By extending the simulation queue with priorities representing tasks and the various interrupt levels, we can easily model such a behavior. Finally, adding atomic statements and the ability to disable interrupts even in the simulation model completes our extensions to the timing model and event queue.

Overall, these timing and rescheduling extensions to simulation models give a detailed insight into the performance of a system without the need for complex emulators or test-beds.

3.3 Automatic, Static and Manual Mapping

Although large percentages of the code in simulation and on the real platforms are identical, device drivers and other simulation specific parts such as the scheduler differ. Thus, automatic code instrumentation cannot be applied to these code sections.

For these sections we introduce static and manual mapping. We can apply static mapping to device driver code that does not contain conditional statements and therefore executes in a constant number of cycles. However, in more complex scenarios, for example in the scheduler, such a simple mapping fails.

Here we apply manual mapping, where we first match the functionalities of code sections in the device specific code and the simulation wrapper. Next, the code sections in the simulation are instrumented with the cycles counts of the matching device specific code. Although this process does not introduce inaccuracies in terms of cycles, it is not as fine granular as the commonly used source line granularity. Thus, interrupts may be delayed by a number of cycles.

4 Implementing Time Accuracy in TinyOS

After introducing the concepts to enable time accurate simulation of sensor networks, we discuss our prototype implementation in this Section. A special focus is put on automated code mapping on source line granularity, dynamic event queue adaptation and manual mapping for low-level device drivers.

We introduce TimeTOSSIM, as an extension of TinyOS-2.x based TOSSIM. We have chosen TinyOS as it is the de facto standard sensor network operating system. Furthermore, the layered platform abstraction of TinyOS results in slim low-level device drivers on the hardware presentation layer (HPL). Therefore, the simulation and platform specific code has very limited differences, making TinyOS 2.x a perfect candidate for automatic simulation code instrumentation. However, the presented approach can be applied to any operating system where simulation and device code share large sections. It is not even limited to sensor nodes.

4.1 Code Mapping & Clock Advancement

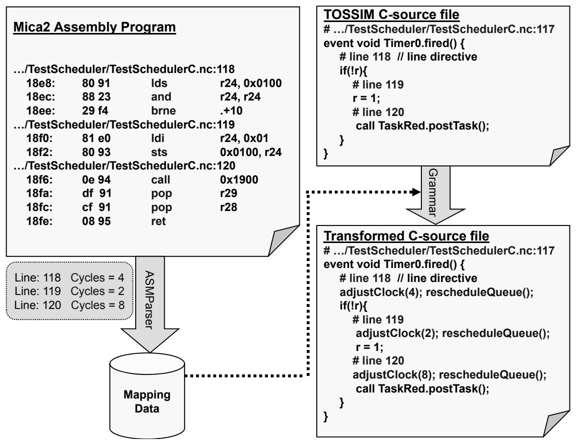
TOSSIM as many other sensor network simulation environments compiles directly from the hardware-platform dependent source-code and thereby enables us to create a mapping between the simulation-code and the platform dependent binary-code. The implementation of this technique is summarized in the following steps: (1) Determine the number of cycles needed by a source-code line to get executed on the original hardware and (2) increment simulation clock fine grained at runtime.

4.1.1 Determine execution time of a source-code line

First, we need to determine the number of cycles to execute each individual line of source code on the original hardware. We obtain this information from the debugging information of the assembly program compiled for a specific platform. Instructing the compiler to include debugging symbols in the assembly code allows us to map each instruction to its original source-code line. We implemented a grammar based parser in ANTLR [16] to analyze the object dumps of executables and retrieve source line information for each assembly instruction. Using this mapping and a look-up table storing the number of cycles required by individual assembly instructions, the execution time of each source code line can be computed (see figure 2).

4.1.2 Fine granular simulation clock incrementation

After computing the number of clock cycles needed by each source line to get executed on the original hardware, we need to instrument the simulation code with this knowledge.



```
.../TestScheduler/TestSchedulerC.nc:118
   18e8:      80 91      lds      r24, 0x0100
   18ec:      88 23      and      r24, r24
   18ee:      29 f4      brne     .+10
.../TestScheduler/TestSchedulerC.nc:119
   18f0:      81 e0      ldi      r24, 0x01
   18f2:      80 93      sts      0x0100, r24
.../TestScheduler/TestSchedulerC.nc:120
   18f6:      0e 94      call     0x1900
   18fa:      df 91      pop      r29
   18fc:      cf 91      pop      r28
   18fe:      08 95      ret
```

```
Line: 118   Cycles = 4
Line: 119   Cycles = 2
Line: 120   Cycles = 8
```

```
# .../TestScheduler/TestSchedulerC.nc:117
event void Timer0.fired() {
    # line 118  // line directive
    if(!r){
       # line 119
       r = 1;
       # line 120
        call TaskRed.postTask();
    }
}
```

```
# .../TestScheduler/TestSchedulerC.nc:117
event void Timer0.fired() {
    # line 118  // line directive
    adjustClock(4); rescheduleQueue();
    if(!r){
       # line 119
        adjustClock(2); rescheduleQueue();
        r = 1;
       # line 120
       adjustClock(8); rescheduleQueue();
        call TaskRed.postTask();
    }
}
```

Figure 2. Source-code mapping and instrumentation of TinyOS code

This process consists of three steps: (1) Parsing the application source code to identify each source code line, (2) instrumenting each line with corresponding execution time, and (3) building the simulation from the extended sources.

As input for the parsing process we use the C code generated by the NesC compiler [8] and not the NesC sources of TinyOS themselves. This has a number of advantages: First, the NesC compiler extends certain programming constructs into multiple lines of code. Thus, instrumenting the C sources instead of the NesC ones increases granularity. Second, the NesC compiler uses internal variables, macros and definitions for compilation, which are not available to external parses. Finally, by applying transformations to C-code, the instrumentation is kept independent from NesC and TinyOS and therefore can be easily ported to other operating systems. For parsing the implementation we use C grammars from the ANTLR parser framework.

In the abstract syntax tree (AST) generated from the C grammar, we identify each source code line and instrument it with the corresponding execution time by incrementing the simulation clock(see figure 2). The instrumented C-source file is then simply compiled to obtain an object file for TimeTOSSIM. This object file is later linked with a TinyOS 2.x simulation driver for executing the simulation.

4.1.3 Trading accuracy for performance

Although instrumentation on source line granularity promises fine grained timing information, such detailed timing modeling is not always necessary. Thus, TimeTOSSIM also allows to instrument code on basic block or function level. The resulting code has less overhead and allows to flexibly trade performance and accuracy based on application needs.

However, functions usually contain conditional statements and loops whose execution cannot be determined at compile time. Thus, function level granularity only gives a rough estimate on the execution time. A basic-block represents a sequence of instructions with single entry point, single exit point, and no internal branches. Therefore, code instrumentation on basic block level results in an accuracy equivalent to the source-code line instrumentation. However, the simulation clock is incremented less often and therefore as a side effect interrupts may be delayed. Thus, for evaluation we use source code line granularity to minimize the gap between simulation and emulation of a hardware platform.

4.2 Interleaving and Rescheduling Events

After instrumenting each source code line with the corresponding execution time, we need to adapt TOSSIM's event queue to handle overlapping events. Thus, interrupts should postpone current tasks and other interrupts based on their priority. Additionally, execution of tasks should be delayed until any current task has ended.

We assign execution priorities to different events. As events in the event-queue represent hardware interrupts or TinyOS tasks, it is possible to determine the type of an event and its execution priority from the processor datasheets. By assigning a priority to every event enables us to reschedule the event-queue and intensify the simulation models even further to exhibit timing and interrupt properties of a hardware platform. Correct ordering of events can be achieved by visiting the event queue at the start of every source line after incrementing the simulation clock. The idea is to reschedule events with lower priority, execute events with higher priority immediately, and thereby delay or interrupt the execution of currently active events.

As we extended TOSSIM's simulation models to represent events with a duration, these events can now be interrupted at any point in time by other events. Thus, it is required to incorporate the behavior of atomic statements into the simulation model of TOSSIM. Thereby we ensure the integrity of global data structures and model their temporal behavior and impact on the overall system. Access to the simulation code at the source-code line granularity also allows to accurately model the behavior of atomic statements in the code, as enabling and disabling interrupts itself takes a number of cycles.

4.3 Static and Manual Instrumentation

For simulation, TOSSIM replaces low-level device drivers on the hardware presentation layer (HPL) of TinyOS with simulation wrappers. Therefore, simulation and platform specific code differ on the hardware presentation layer and the presented code automated instrumentation techniques are of limited use for low-level device drivers. Further differences can be found in code that has been extended for TOSSIM to allow user interaction and the support for multiple sensor nodes, such as the scheduler. However, these layers are commonly quite slim. In this sub-section we present the implementation of two techniques to enable accurate timing even in these code section: (1) static code mapping and (2) manual code mapping.

We apply static code mapping in simple device drivers that do not contain any conditional statements and therefore execute in a constant number of cycles. For example, we applied this approach to model the time required to enable or disable pins of the microcontroller, timers and to integrate the Mica2 CC1000 radio into TimeTOSSIM. Here our design again benefits from the multi-layered hardware abstraction of TinyOS. Code on the HPL level simply presents direct hardware access and rarely contains complex statements such as loops and conditionals. Although this process does not introduce inaccuracies in terms of cycles, it is not as fine granular as the commonly used source line granularity. Thus, interrupts may get delayed a number of cycles. However, HPL code sections are usually 10 to 100 cycles and therefore executed in a couple of micro seconds.

Likewise, to model code sections that were extended for simulation in TOSSIM and to address that some code in the HPL layer may have a higher complexity, we use manual mapping. Based on the fact that the simulation model needs to reassemble the functionality of the device specific code, we manually map sections with equal functionality and instrument the simulation code with the corresponding number of cycles. We applied this approach to the TOSSIM scheduler. Its implementation strongly differs from the device specific one, but it reassembles the same functionality and therefore can be easily instrumented manually.

4.4 TimeTOSSIM: The Complete Process

After discussing the implementations of clock adaptation, event rescheduling and extensions to the code mapping process, we discuss their integration into TOSSIM and the TinyOS build process.

Figure 3 shows an overview of TimeTOSSIMS's build process. TimeTOSSIM extends the platform specific build process and the simulation specific one. On the platform specific side, we parse the assembly code to retrieve cycle counts from each source code line. On the simulation side, we first extend the simulation platform with statically and manually mapped code and the ability to model interrupts. After NesC compilation, we parse the resulting C-code to instrument it automatically with the cycle counts retrieved from the assembly code. The instrumented code is then

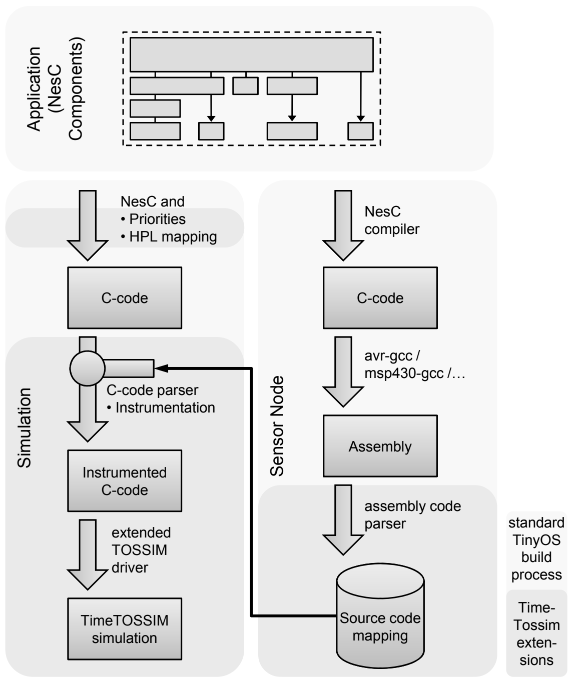


Figure 3. Integration of TimeTOSSIM into the TinyOS and TOSSIM build process.

combined with a TOSSIM simulation driver. Please note that this process is neither bound to a certain hardware platform nor to TinyOS and NesC. Therefore, it can easily be applied to any other sensor node architecture and operating system.

5 CC1000 Radio in TOSSIM

One of the most important functions performed by a sensor node is to communicate with other nodes in the network. Currently, TinyOS-2.x only provides simulation of the MicaZ sensor node platform which uses a packet level CC2420 radio chip. Our basic-block mapping technique successfully maps the communication related code of MicaZ sensor nodes with the corresponding simulation models. However, we are unable to evaluate the accuracy level achieved in radio communication and to profile the low level components of the MicaZ platform due to unavailability of a suitable emulator for CC2420 radio chip.

To overcome this limitation we provide our own simulation wrapper for the CC1000 radio chip (used in Mica2 sensor nodes) for TinyOS-2.x based TOSSIM simulation, as Mica2 and the CC1000 radio chip are supported by publically available emulation platforms such as Avrora. Our CC1000 radio chip implementation benefits from the platform abstraction architecture of TinyOS-2.x which only re-

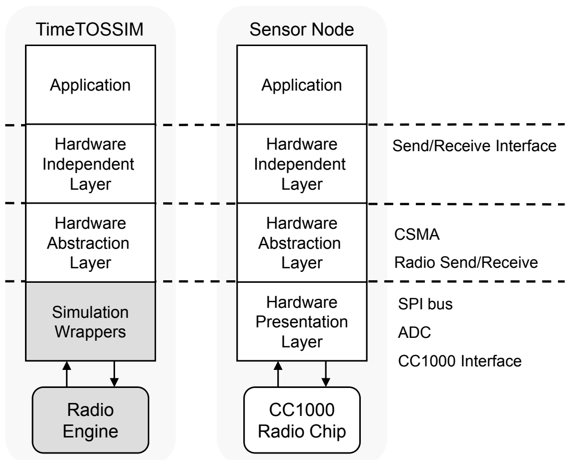


Figure 4. CC1000 radio simulation on HPL level in TimeTOSSIM

quires re-implementation of the low level hardware dependent code at the HPL layer. The original code at HIL and HAL layers remains unchanged for simulation, thus, enabling a detailed mapping and instrumentation of code and simulation models. [1]

Figure 4 shows the architecture of our CC1000 radio chip implementation. The hardware presentation layer (HPL) architecture of TinyOS requires the simulation to provide just four interfaces that expose the CC1000 radio hardware, making the simulation wrapper easy to integrate into TinyOS. Apart from this, we have implemented a small radio engine that interacts with the TOSSIM simulation core and stimulates the CC1000 specific radio simulation. It provides mandatory signals such as SPI interrupts and RSSI readings at the needed times and models the "air" between sensor nodes.

6 Evaluating TimeTOSSIM

In this section we thoroughly evaluate TimeTOSSIM both from performance and accuracy perspectives. We compare TimeTOSSIM with the cycle accurate emulator, Avrora, and to the original TOSSIM implementation. We achieve beyond 99% time accuracy for sensor network applications using TimeTOSSIM while adding only a small performance overhead compared to the original TOSSIM implementations. The evaluation is based on three types of benchmarks: (1) micro benchmarks, (2) evaluation of static and manual instrumentation, and (3) macro benchmarks. Micro-benchmarks evaluate the accuracy of TimeTOSSIM at the level of programming constructs such as conditional statements and loops. This evaluation gives a detailed insight into the accuracy and limitations of the presented design. Next, we evaluate the accuracy of the manually or statically instrumented low-level device drivers at the HPL level of TinyOS. Finally, we thoroughly evaluate the performance and accuracy of different off-the-shelf applications (when run on TimeTOSSIM) in our macro-benchmarks. Macro benchmarks present the accuracy level that can be expected from typical sensor network applications and show the performance of TimeTOSSIM in terms of CPU and memory overhead compared to TOSSIM and Avrora.

Although TimeTOSSIM supports both the MicaZ and Mica2 sensor nodes, Avrora is currently limited to the Mica2 sensor node with CC1000 ChipCon radio. For TinyOS 2.x even this radio is not fully supported by Avrora. Thus, our evaluation bases on the Mica2 platform with limited support from the Avrora side.

6.1 Micro Benchmarks

In our micro-benchmarks we evaluate the time accuracy of different types of mapped code-blocks (loops, control-structures etc.) independently from each other to give a deep insight into the timing properties of source-code. We start with simple executable statements and then discuss simple loops and conditionals. The presentation of nested constructs and short circuit operators completes the micro benchmarks.

Simple Executable Statements: With the term simple executable statements we refer to all statements that do not alter the execution sequence of the program unlike loops and control structures. These statements, for example, include variable initialization, assignment statements involving arithmetic expressions, and function calls. Commonly, simple statements make up for the largest part of a program. We achieve 100% time accuracy when the simulation is executing such statements. The reason is that these statements are compiled into a single basic-block of assembly instructions. Hence, it is possible to determine the exact clock-cycles consumed by such statements.

Loops: Loops can severely impact the timing of an application in simulation because any program spends most of its time in executing loops. We achieve 100% clock synchronization in the case of non-nested *while* and *do-while* loops. However, in the case of *for* loops and nested *while* loops the simulation clocks get de-synchronized just by a few clock cycles. Our stress tests on loops evaluate accuracy by monitoring the simulation clock for each single iteration of the loop

[1]After code cleanup we will contribute TimeTOSSIM and our CC1000 simulation model to the TinyOS 2.x community.

Code-block	Clock drift in cycles
Statements	0
While loops	0
Do loops	0
For loops	+4
Nested while loops	-1
If-else	0
Switch statement	±15

Table 1. Simulation clock drifts for the execution of a code block in cycles for Mica2.

instead of calculating the total number of lost cycles after the loop iterations are finished.

While loops: In the case of a single *while* loop (without any nested loops) having a conditional expression that doesn't include any short circuit operators, we achieve 100% clock synchronization between TimeTOSSIM and Avrora. However, for nested *while* loops we loose synchronization only by a single clock-cycle for each iteration of the loop as shown in Table 1. From our point of view, a single clock-cycle de-synchronization in nested-loops has a negligible impact on the timing of an application: Commonly, loops consume several hundred of cycles, reducing TimeTOSSIM's error to below 1%. Additionally, external events - such as interrupts - re-synchronize TimeTossim at the start of every new simulator-event as they are scheduled accurately (see section 6.3).

For loops: In the case of *for* loops the simulation clock of TimeTOSSIM gets de-synchronized typically by 4 clock-cycles for each iteration of the loop. It is because the *for* loops allow variable declaration and initialization inside the loop statement, which takes 4 clock-cycles in the case of an integer variable (which usually is the case). These clock cycles are counted for each iteration of the loop in TimeTOSSIM as debugging information in the assembly code combines all assembly instructions corresponding to a *for* loop declaration. However, the declaration and initialization takes place only once at the beginning of the loop. Our parser, as it abstracts from single instructions and operates on complete source code lines, reports the total number of cycles needed for variable initialization, condition check, and increment. Similarly to while loops, we consider the average four clock cycles of inaccuracy acceptable.

Control Structures: Control structures include *if-else* and *switch* statement clauses. Control structures are examples of such statements which may get compiled into several basic blocks of assembly instructions and get executed based on runtime decisions.

If-Else: We achieve 100% timing accuracy if the condition-check in the *if-else* statements do not include short circuit operators.

Switch clause: A *switch* clause jumps to one of the several *case* blocks depending on the value of the decision variable. Therefore, a *switch* statement also gets compiled into several basic-blocks of assembly instructions and the number of cycles consumed can only be determined at run time (i.e. it depends upon the case block the decision variable refers to). We take the average of the number of cycles reported by the assembly parser to increment the simulation clock for minimizing the clock de-synchronization. Our evaluation shows that the average corresponds to 75% of the total clock-cycles. For a *switch* clause containing five case blocks, the maximum clock drift is therefore only 15 cycles (2 microseconds).

Short circuit operator: Although TimeTOSSIM models most programming structures with no or just minimal inaccuracies, its accuracy is limited when evaluating short circuit operators. For example, the right-hand-side expression of an AND operator will only be executed if the left-hand-side expression is true and therefore requires an instrumentation granularity beyond source line level. Additionally, in contrast to loops and control statements we cannot bound the error introduced by these operations as they may be arbitrary complex. However, in practice – to insure code readability – most of these constructs turn out to be limited in complexity and therefore the error introduced by short circuit operators is acceptable.

Concluding the evaluation on micro benchmark level, it can be said that TimeTOSSIM models most programming structures with no or just minimal inaccuracies. Just short circuit operator show the limitations of the chosen approach. However, as the macro benchmarks show, the overall accuracy of TimeTOSSIM is only slightly influenced by these inaccuracies.

6.2 Hardware Components

After evaluating the accuracy of TimeTOSSIM regarding programming structures, we evaluate time accuracy of different operations performed on the most frequently used on-chip hardware components: LEDs and timers. Currently, apart from instruction execution, Avrora emulates only these two on-chip hardware components correctly for TinyOS-2.x based applications.

LEDs are the simplest example of a hardware component attached to a micro-controller pin. Evaluating LED

Component	Accuracy	Minimum Granularity
Led	100%	47 cycles
Timer	100%	same as emulation

Table 2. Accuracy and granularity of hardware components in TimeTOSSIM.

operations fully tests the functionality of our approach because any operation on LEDs involves automatic, static and manual code mapping and instrumentation, as all hardware components are accessed via the hardware abstraction layer of TinyOS. Profiling of the low level LED component of TinyOS shows that the minimum granularity (maximum clock advancement) achieved in LED operations is 47 clock cycles (6 microseconds).

Similar to the access to microcontroller pins, we evaluated the accuracy of Timer components in TimeTOSSIM. Our results show, that we achieve the same accuracy and granularity as emulation (see Table 2).

6.3 Macro Benchmarks

For our macro-benchmarks we evaluate TimeTOSSIM from two perspectives: time accuracy and scalability. We have a very limited choice of off-the-shelf applications to evaluate TimeTOSSIM. Firstly, because TinyOS 2.x is still in its active development phase and offers very few standard applications. Secondly, Avrora is still unable to emulate TinyOS-2.x based applications involving radio communication (which mostly is the case). Table 3 shows the accuracy level we achieve with different off-the-shelf applications. We compare the simulation traces of TimeTOSSIM with Avrora. Our measured results show beyond 99% time accuracy for most of the applications. Additionally, we use the *TestScheduler* application to stress-test the accuracy of TimeTOSSIM from the worst-case point of view. The *TestScheduler* application is a sanity check for TinyOS scheduler and has no hardware events that could re-synchronize the simulation-clock. Nonetheless, we still achieve 88% accuracy.

Table 3 depicts the accuracy for code instrumentation on different optimization and instrumentation levels. For source line granularity we show that we can achieve an accuracy beyond 99% for typical applications. This level of accuracy is independent from the compiler optimizations of the sensor node application. Basic block level instrumentation achieves similar timing results as source line instrumentation. However, it has a lower granularity and therefore may delay interrupts under high load. Function level instrumentation results in less accurate modeling compared to basic block and source line granularity. However, basic block and function level granularity result in less code instrumentation and therefore increase the simulation speed in TimeTOSSIM.

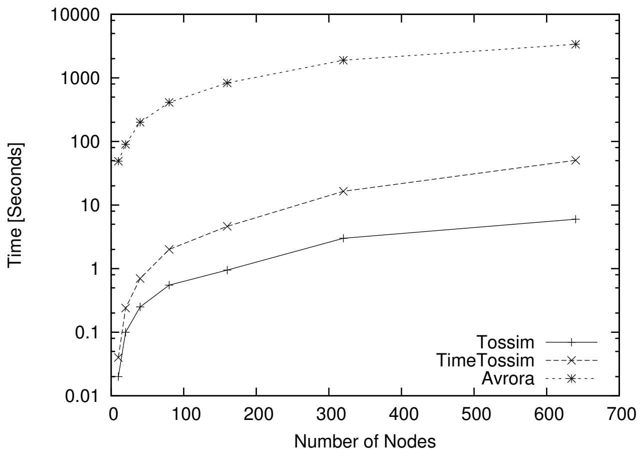


Figure 5. Scalability comparison for sensor network simulators and emulators. Please note the logarithmic scale on the y-axis.

After evaluating the accuracy achieved with TimeTOSSIM, we evaluate the speed and memory consumption of TimeTOSSIM by comparing it to TOSSIM and Avrora. All experimental results discussed in this section were executed on a customary end-user machine, a Pentium IV with 3 GHz clock frequency and 1GB of RAM. Our evaluations show that TimeTOSSIM when using instrumentation on source line granularity is up to 10 times slower than TOSSIM while being more than 100 times faster than Avrora, especially when using large numbers of nodes (see Figure 5). For single node simulations the overhead of TimeTOSSIM is reduced to a factor of 1 to 6, as the number of adaptations of the event queue gets reduced drastically (see Table 4). The results in Table 4 are based on simulation runs for 60 simulated seconds of 20 simulated sensor nodes; except for the *TestScheduler* application which is for one simulated sensor node. Furthermore, TimeTOSSIM consumes nearly the same amount of memory as TOSSIM.

In comparison to PowerTOSSIM, TimeTOSSIM shows a similar performance overhead. Thus, PowerTOSSIM and TimeTOSSIM need about the same time for simulation. However, TimeTOSSIM provides much more functionality and energy modeling can be easily added to TimeTOSSIM based on the derived cycle counts.

Concluding the performance and accuracy evaluation, it can be said that TimeTOSSIM, though slower than TOSSIM, provides a very accurate simulation of sensor networks. Although code instrumentation on source code line granularity introduces some inaccuracies, their overall impact seems to bee small. Furthermore, the fact that instrumentation of source lines does not require any special com-

Application	Instrumentation level and accuracy (in %)			
	Source line no optimization (-O0)	Source line space optimization (-Os)	Basic Block	Function
Blink	99.69	99.63	99.69	98.93
BlinkTask	99.73	99.55	99.73	98.84
CntToLeds	99.69	99.64	99.69	98.97
TestScheduler	87.7	81.44	87.7	NA

Table 3. Accuracy for different applications achieved in TimeTOSSIM (in comparison to Avrora) for different instrumentation granularities and compiler optimizations.

Application	TimeTOSSIM		TOSSIM		Avrora	
	Time (sec)	Memory (kB)	Time (sec)	Memory (kB)	Time (sec)	Memory (kB)
Blink	5.2	1064	1.7	1064	129	42892
BlinkTask	1.80	1064	1.5	1060	131	42504
Sense	3.1	1068	0.5	1064	NA	NA
CntToLeds	7.5	1068	4.0	1064	133	42604
RadioCountToLeds	20.3	1168	9.83	1168	NA	NA
TestScheduler	2.4	976	NA	976	29.4	20584

Table 4. Performance comparison for sensor network simulators and emulators: TOSSIM, Time-TOSSIM and Avrora

piler extensions, ensures that TimeTOSSIM can be easily ported to various sensor node platforms and operating systems.

7 Future Work

After implementing and evaluating the design of TimeTOSSIM a number of interesting questions remain open that we are addressing currently.

Although we were able to partially evaluate the accuracy of the CC1000 and CC2420 radios of Mica2 and MicaZ sensor node platforms, limitations in the current version of Avrora prohibited a full evaluation. As the radio in TinyOS is one of the most complex components, we think that the radio stack and its dynamic interaction with other sensor nodes requires a detailed evaluation to further explore the possibilities and limitations of the presented design. Thus, we are currently working on Avrora to provide the required radio support for TinyOS 2.x.

TimeTOSSIM is 5 to 10 times slower than TOSSIM. During evaluation, we observed that most of the performance overhead introduced by TimeTOSSIM is due to queue rescheduling mechanisms of TOSSIM. It stores events of all the simulated nodes in a single simulation queue. Therefore, the process to find target events - simulator events corresponding to the node currently being simulated - requires to search the whole queue. We believe that by implementing separate event-queues for each simulated node, the overhead of TimeTOSSIM simulation can be reduced significantly.

Furthermore, the grammar based code instrumentation allows for a flexible, plugin-based extension of TimeTOSSIM. Thus, features such as energy modeling and even shutting down nodes during simulation when their energy resources exceed, can easily be added to TimeTOSSIM.

Sensor network research offers a variety of sensor node platforms. Therefore, it is important to provide multi-platform support in sensor network simulations. Discrete event simulation, by virtue of its design, is easily extendable to multiple platforms. We plan to extend TimeTOSSIM to provide time accurate simulation of multiple sensor node platforms. Adding multi-platform support for AVR based sensor nodes only requires profiling the low-level hardware components. Similarly, adding support for the Texas Instruments MSP-430 based sensor nodes (e.g. Telos platform [18]) only requires – in addition to profiling – to extend the assembly parser to recognize the instruction set of the corresponding platform.

Finally, we can provide code instrumentation on opcode granularity to address the errors introduced by short circuit operators and loops (see section 6.1). We believe that it will enable 100% accuracy while still outperforming emulation, as instructions do not need to be interpreted. Although this would require compiler extensions, we expect the complexity of such extensions to be much less compared to designing and implementing a complete emulator.

8 Conclusions

The increasing complexity of sensor network applications combined with the resource constrained hardware of sensor networks requires a deep evaluation before deployment. For example, time consuming tasks such as data analysis or cryptography suffer significant delays and bottlenecks due to severely limited computation power of sensor nodes and periodic interrupts from external devices. Hence, it is important in simulations to model timing and interrupt properties of applications and operating systems at a fine granularity.

In this paper we present automated instrumentation of simulation models to enable the required time accurate simulation of sensor networks allowing deep analysis of applications. We show that this automated instrumentation on source-code line granularity provides an accuracy beyond 99% for typical sensor network applications while offering much higher performance, scalability and easy portability compared to today's emulators. This slight accuracy degradation was intentionally traded for easy portability and limited complexity of the overall system, as these were main design goals in addition to accurate timing. We believe, that with the strongly increasing number of different sensor node platforms, it is mandatory to provide accurate and flexibly portable tools – such as TimeTOSSIM – to easily evaluate applications for arbitrary platforms.

9 Acknowledgements

This work is supported by the DFG Research Cluster on Ultra High-Speed Mobile Information and Communcation (UMIC).

References

[1] J. Beutel, M. Dyer, L. Meier, M. Ringwald, and L. Thiele. Next-generation deployment support for sensor networks. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, Nov. 2004.

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *MONET, Special Issue on Wireless Sensor Networks*, 2004.

[3] G. Chelius, A. Fraboulet, and E. Fleury. Worldsens: a fast and accurate development framework for sensor network applications. In *The 22nd Annual ACM Symposium on Applied Computing (SAC 2007)*, Seoul, Korea, March 2007. ACM.

[4] K. Chintalapudi, T. Fu, J. Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Monitoring civil structures with a wireless sensor network. *IEEE Internet Computing*, 10(2), 2006.

[5] B. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat. Mirage: a microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (EmNets)*, May 2005.

[6] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, 2006.

[7] E. Ertin, A. Arora, R. Ramnath, V. Naik, S. Bapat, V. Kulathumani, M. Sridharan, H. Zhang, H. Cao, and M. Nesterenko. Kansei: a testbed for sensing at scale. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, 2006.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[9] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[10] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *Proceedings of the Third International Conference on Mobile Systems, Applications, and Services*, June 2005.

[11] O. Landsiedel, K. Wehrle, and S. Gotz. Accurate prediction of power consumption in sensor networks. In *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, 2005.

[12] O. Landsiedel, K. Wehrle, B. L. Titzer, and J. Palsberg. Enabling Detailed Modeling and Analysis of Sensor Networks. *PIK Journal*, 2005.

[13] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st ACM Conference on Embedded Networked Sensor Systems*, November 2003.

[14] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004.

[15] S. Park, A. Savvides, and M. B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, 2000.

[16] T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Software: Practice and Experience*, July 1995.

[17] F. Perrone, D. Nicol, J. Liu, C. Elliot, and D. Pearson. Simulation modeling of large-scale ad-hoc sensor networks. In *Proceedings of the 2001 Simulation Interoperability Workshop.*, 2001.

[18] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, 2005.

[19] J. Polley, D. Blazakis, J. Mcgee, D. Rusk, and J. S. Baras. Atemu: a fine-grained sensor network simulator. In *Proceedings of the First IEEE Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, October 2004.

[20] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. Luster: Wireless sensor network for environmental research. In *SenSys '07: Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems*, November 2005.

[21] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200, 2004.

[22] S. Sundresh, W. Kim, and G. Agha. Sens: A sensor, environment and network simulator. In *37th Annual Simulation Symposium (ANSS37)*, 2004.

[23] R. Szewczyk, J. Polastre, A. M. Mainwaring, and D. E. Culler. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, January 2004.

[24] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, 2005.

[25] M. Varshney, D. Xu, M. Srivastava, and R. Bagrodia. Senq: a scalable simulation and emulation environment for sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, 2007.

[26] V.Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler. Flexible hardware abstraction for wireless sensor networks. In *Proc. of 2nd European Workshop on Wireless Sensor Networks (EWSN 2005)*, Feb. 2005.

[27] Y. Wen, R. Wolski, and G. Moore. Disens: scalable distributed sensor network simulation. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007.

[28] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, November 2006.

[29] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: a wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, 2005.