Avrora: Scalable Sensor Network Simulation with Precise Timing

Ben L. Titzer
UCLA
titzer@cs.ucla.edu

Daniel K. Lee
Cornell University
dkl25@cornell.edu

Jens Palsberg
UCLA
palsberg@ucla.edu

***Abstract*— Simulation can be an important step in the development of software for wireless sensor networks and has been the subject of intense research in the past decade. While most previous efforts in simulating wireless sensor networks have focused on protocol-level issues utilizing models of the software implementation, a significant challenge remains in precisely measuring time-dependent properties such as radio channel utilization. One promising approach, first demonstrated by ATEMU, is to simulate the behavior of sensor network programs at the machine code level with cycle-accuracy, but poor performance has so far limited its scalability. In this paper we present Avrora, a cycle-accurate instruction-level sensor network simulator which scales to networks of up to 10,000 nodes and performs as much as 20 times faster than previous simulators with equivalent accuracy, handling as many as 25 nodes in real-time. We show how an event queue can enable efficient instruction-level simulation of microcontroller programs and allow the hidden parallelism in fine-grained sensor network simulations to be extracted, once two core synchronization problems are identified and solved. Avrora's ability to measure detailed time-critical phenomena can shed new light on design issues for large-scale sensor networks.**

I. Introduction

A. Background

Sensor networks have tremendous potential to monitor, study, and analyze phenomena in the physical world in detail never before available, in places too far, too deep, too high, or too dangerous for researchers to go, from monitoring oceanic microorganisms to industrial processes to volcanic activity. Simulation of sensor networks has been studied intensively this decade [1], [2], [3], [4], [5], [6], [7]. The primary goal of simulation is to enable rapid exploration and validation of system designs before deployment, by providing a controlled environment for evaluating design and configuration alternatives, even for systems that may be impossible to physically realize, and allowing detailed inspection of interactions that may be too fast, too infrequent, or too complex to observe in a real system. Distributed behavior and subtle timing interactions in sensor networks present significant challenges for accurate, fine-grained simulation.

For single processors, efficient, cycle-accurate, instruction-level simulation is standard. Such simulations are often used when acquiring the actual hardware is difficult, infeasible, or impossible, such as when prototyping and evaluating proposed microprocessor designs, emulating legacy architectures, developing compiler backends, virtualizing operating systems, etc. Instruction-level simulation provides the highest behavioral and timing accuracy for software and is both language and operating system independent. For a sensor network with many nodes, building an accurate and scalable simulator is more challenging.

A sensor network simulator can trade accuracy for scalability by running *models* of the nodes rather than simulating an actual software implementation. For example, SensorSim [1], SWAN [2], and SENS [3] all use handcrafted models of the sensor nodes. The modeling-based approach was sufficient to gauge network delays, throughputs, packet collisions, and node localization errors for 10,000 nodes [2], [3], as well as power usage and the effect of several power management schemes [1], [3].

Two recent sensor network simulators provide better accuracy by using little or no modeling of the sensor nodes. Both TOSSIM [5] and ATEMU [7] are simulators for sensor networks in which the nodes are Crossbow AVR/Mica2 motes, which we will simply refer to as "motes". Motes have limited resources; at the core is an 8-bit 7.3728MHz AVR microcontroller with 4KB main memory for stack and heap, 128KB program storage for code and pre-initialized data, 4KB non-volatile EEPROM storage, and internal devices such as clocks and a serial port for controlling external devices. Software for the motes is generally built with either TinyOS [8], a set of components for building sensor networks programs written in the nesC [9] programming language, or SOS [10], a lightweight, modular operating system designed for dynamic flexibility.

TOSSIM [5] provides a high degree of accuracy by using models of only a few low-level components, and otherwise running the application source code unchanged. Specifically, TOSSIM compiles nesC source code together with TinyOS libraries into a binary for the development workstation, replacing the software modules that interface hardware with emulation libraries, including timers, communication channels, sensors, and the radio. TOSSIM's level of detail was sufficient to measure packet losses, packet CRC failure rates, and the length of the send queue for up to 8,192 nodes [5]. However, TOSSIM's compilation step loses the fine-grained timing and interrupt properties of the code that can be important when the application runs on the hardware and interacts with other nodes.

ATEMU [7] was the first and, until now, only instruction-level simulator that could simulate sensor network programs with accuracy down to the clock cycle of each individual node. ATEMU's fine-grained accuracy enabled a reliable count of the number of backoffs after transmission collisions for up to 120 nodes [7]. However, ATEMU is 30 times slower than TOSSIM.

Our goal: better scalability without sacrificing cycle accuracy.

B. The Problem

ATEMU uses a cycle-by-cycle implementation strategy where each node and each device are advanced by one clock cycle every round to ensure nodes, their internal devices, and the radio communication are correctly synchronized. Synchronization is essential because much of the microcontroller program's execution depends on the timing and behavior of devices. For example a timer device might increment a timer count register every 1024 clock cycles and trigger an interrupt when the counter reaches a maximum value. In ATEMU, the timer does work each cycle and internally counts up to 1024; it then increments the special register and checks whether an interrupt needs to be fired. The cycle-by-cycle implementation strategy achieves synchronization for free but scales poorly, as each device adds work to be done each clock cycle.

Is it necessary to synchronize every clock cycle?

C. Our Results

We have implemented Avrora, an instruction-level sensor network simulator which scales better than ATEMU and approaches the performance of TOSSIM while preserving cycle accuracy. Like ATEMU, Avrora simulates a network of motes, runs the actual microcontroller programs (rather than models of the software), and runs accurate simulations of the devices and the radio communication. Avrora is implemented in Java, which helps flexibility and portability; TOSSIM and ATEMU are implemented in C. Avrora and ATEMU gain language and operating system independence by simulating machine code, while TOSSIM can simulate only TinyOS programs.

Avrora = cycle-accurate simulation
+ multi-threading
+ event queue
+ efficient synchronization.

Avrora runs one thread per node; those threads synchronize only when necessary. Synchronization is needed only to ensure that the global timing and order of radio communications are preserved during simulation. We show that synchronization can happen much less often than every clock cycle, which leads to a substantial performance gain. To enable efficient execution of programs, we use the well-known idea of an *event queue* [11]. For efficient synchronization that preserves correctness and cycle-accuracy, we have formulated and solved two synchronization problems, namely the *Send-Receive Problem* and the *Sampling Problem.*

In single node performance, Avrora's core interpreter can simulate CPU-intensive microcontroller programs at up to 25MHz on our test machine, a 3.06ghz Xeon. This is more than 3 times real-time speed for the 7.3728MHz ATMega128L microcontroller that drives the motes. With the sleep optimization made possible by the event-queue model, programs that sleep most of the time can run many times faster.

In scalability over the number of nodes, Avrora can handle up to 1750 nodes on our dual processor server. On a large Sun Enterprise machine, we have successfully simulated networks as large as 10,000 nodes. Avrora achieves linear scalability in terms of simulation time across the number of nodes in our experiments, achieving greater than real-time performance for networks less than 25 nodes. We have found that the number of nodes that can be simulated is limited only by the operating system's limit on the number of threads per process.

Avrora outperforms ATEMU in terms of raw performance when run on a state-of-the-art Java virtual machine, scales to more nodes, and scales well over number of processors available to the simulator. Avrora and ATEMU achieve the same level of accuracy, as both model time down to the clock cycle. While less precise, TOSSIM scales the best in our experiments.

Avrora shows that cycle-accurate simulation of thousands of nodes is possible with reasonable performance and that networks of tens of nodes can be simulated in real-time. Avrora allows sensor network program behavior to be analyzed in precise detail on a scale that has not been previously achieved. Recently, Landsiedel et al. [12] have added a highly accurate energy model to Avrora, enabling power profiling and lifetime prediction of sensor networks.

In Section II we describe how Avrora simulates a single node. In Section III we discuss radio communication and synchronization of multiple nodes. Finally, in Section IV we present experimental results for single node simulations and networks up to thousands of nodes.

II. Avrora's Event Queue

The event queue is key in implementing an efficient cycle-accurate simulation of microcontrollers an external devices and provides for a natural extension from single node simulation to simulating an entire network of sensor nodes. One of the motivations for the event queue is that energy-aware microcontroller programs tend to sleep a large proportion of the time. The motes have a software-enabled sleep mode that puts the microcontroller in a low-power mode where no instructions are executed and much less energy is consumed. In sensor networks, the need for conserving energy is heightened by sensing, processing and communicating in remote areas where batteries cannot be replaced often. Consequently, programs tend to sleep as often as they can, waking infrequently to sense and communicate. This interval may vary from a few times per second to a few times per hour. For example, the program CntToRfm (from the TinyOS distribution) increments a counter and broadcast its value 4 times per second; with simulation we found it sleeps 96.8% of the time.

Avrora's event queue takes advantage of the sleep-often property of sensor programs for a significant boost in performance. We observe that if the program puts the microcontroller into sleep mode, then only a time-triggered event that causes an interrupt can wake the microcontroller. Such an event may be generated on the chip or in the environment, both of which are implemented by inserting time-triggered events into the event queue of the node to be executed at the appropriate time in the future. Consequently, events must be queued in advance of when they should occur and only the event at the head of the queue, regardless of how far in the future it might be, can influence the simulation when the microcontroller is sleeping, as there are no instructions being executed. Avrora therefore only needs to process the events in the queue in order until one of them causes a hardware interrupt, which resumes normal microcontroller execution. Large amounts of simulated time can be passed in this way with little simulator work. An extreme case, Blink, a TinyOS program that toggles an LED once a second, sleeps 99.96% of the time. Without the sleep optimization, the simulation runs at 31MHz on Avrora. With the sleep optimization, the simulation runs more than 500 times faster.

The pseudo-code below outlines the main simulation loop:

```
infinite loop {
    if (sleeping)
        eventqueue.advance(head.delta);
    else {
        execute instruction;
        eventqueue.advance(cyclesconsumed);
    }
}
eventqueue.advance(cycles) {
    if (head) {
        head.delta -= cycles;
        if (head.delta == 0) {
            head.fire();
            head = head.next;
        }
    }
}
```

Interrupt processing, sleep mode processing, and event queue processing are handled in the main loop. At the end of each instruction, the interpreter advances the event queue by the number of cycles consumed by the instruction just executed, maintaining the correct local simulation time.

The event queue is implemented using an efficiently maintained delta queue which keeps the earliest event to be fired at the head. After each instruction, the main loop of the interpreter simply

subtracts the number of cycles consumed by that instruction from the event at the head of the queue. When that delta reaches zero, the event is fired, and the next link in the queue becomes the new head. This imposes minimal cost on the interpreter; often there are no events in the queue, which requires only one comparison to detect. When there are events in the queue, one extra integer subtraction and one test for zero are all that is required after executing each instruction. We found insertion to be fast because most often the queue is small or empty.

The device implementations take advantage of the event queue, leading to significant simplifications. For example, a timer implementation no longer needs to do work each cycle to scale the main clock, but does work by inserting an event into the queue to be fired at the appropriate time. The simulator will execute instructions at full speed until it reaches that event, at which time the event can update the count of the register and trigger an interrupt if necessary. For devices such as the UART and EEPROM, where correct timing of reads, writes, and transmissions is important, the event queue model makes implementation far more simple and straightforward, while still maintaining timing accuracy down to the clock cycle. Another important advantage is that, byte-level communication devices need not be simulated down to individual bits; devices such as the UART and SPI use a single delivery event for the reception of the entire byte, rather than delivering each bit individually, while preserving the correct timing. For devices that are present on the microcontroller but 'off', no events are inserted into the queue, and thus there is no overhead on simulation, in contrast to ATEMU.

III. Efficient Synchronization

We show that efficient synchronization is key to extracting the hidden parallelism in cycle-accurate simulation of sensor network programs. In our architecture, each node in a sensor network is 1) represented as a Simulator object, 2) run in its own thread, and 3) treated as an event-generating black box. For simplicity and generality, we show how the problem can be solved when each node can communicate with all other nodes over the radio. This presents the worst case in terms of radio channel contention and communication possibilities. Since our goal is to demonstrate the effectiveness of our architecture for simulation and its performance, this simple radio model suffices. In our simulator implementation, more complex radio models can be easily substituted, as the model is clearly separated and modularized in the software architecture, and neither the simulator implementation, event queue, or radio device implementation is directly dependent upon it. Next, we will describe the details of the hardware radio, list the requirements for correct radio simulation, and then present our formulation and solution to the two central synchronization problems.

A. Hardware Software Interface

TinyOS software implements a simple medium access control (MAC) protocol that allows link-layer transport of packets. These network packets are decomposed into individual bytes or bits by the lowest level of the software radio stack and sent to the radio device which transmits them at a programmable frequency, power level, and bit rate. In the case of the mica2 motes, a CC1000 AM radio is connected to the microcontroller through a byte-oriented interface. The software that controls the CC1000 is delicately timed and contains micro-second level wait() loops to achieve correct synchronization with the radio device. In order to test, debug, and tune this level of the stack, the CC1000 radio and its characteristics must be faithfully simulated. Since both the MAC and the protocol stack are implemented in software, a simulator cannot make assumptions about their behavior, but must emulate the low-level behavior of the devices in order to validate and study how the software interacts with them.

In addition to sending and receiving bytes, the radio device allows software to measure a receive signal strength indication (RSSI) value. An RSSI value represents the power level of the signal that the radio is currently receiving which is essentially the sum of all transmissions the radio is receiving at the current time. An RSSI value is often used for collision avoidance; software samples the RSSI value and if it is too high, the the radio is likely receiving a signal from another source or some noise. The software will wait to try again later to avoid interfering with the transmission in progress. An RSSI value can also be used by sensor software for range finding and location discovery; an accurate RSSI value is important in this case.

B. Radio Simulation Requirements

An accurate simulator should preserve the relative ordering of communications and be faithful as to when the communications occur, when nodes transmit, and when the data is received. In our simulator, we treat each node as a separate thread which runs concurrently with other nodes in other threads. Therefore, efficient solutions to synchronizing concurrent processes are needed.

Let us introduce a bit of notation. If n, m are nodes, and s, t are points in time ($s > t$), we use the notation $wait(n, s) until(m, t)$ to denote the requirement that a node n should not pass point s until node m has passed point t. We can now crystallize the requirements for radio simulation into two core synchronization problems.

- **The Send-Receive Problem:** Suppose a node n sends a byte over the radio at global time T, with a latency of L before that packet reaches other nodes. For every node m, ($m \neq n$), we have the requirement $wait(m, T + L) until(n, T)$. The requirement ensures that each node that is able to receive a packet that is sent will not proceed beyond the point in time where that packet should have been delivered.
- **The Sampling Problem:** Suppose a node n requests the RSSI value of its own radio at global time T, with a sampling time of S before the RSSI value is available. For every node m, ($m \neq n$), we have the requirement $wait(n, T + S) until(m, T)$. The requirement ensures that node n does not inspect the computed RSSI value before all transmissions that can influence the RSSI value have been made.

We have estimated the latency L and sampling time S, as follows. The minimum latency L for transmission of a single byte can be calculated from parameters in the hardware specification manual for the radio. Data is transmitted over the radio at 38.4 kBaud using Manchester encoding, equivalent to 19.2 kbps, or 2.4 kilobytes per second. Using the clockrate of the microcontroller, we can calculate that 3072 cycles pass for each byte transmitted over the radio.

$$\begin{aligned} L &= \frac{\text{\# cycles per second}}{\text{radio data rate in bytes per second}} \times 1 \text{ byte} \\ &= \frac{7372800 \text{ cycles per second}}{2400 \text{ bytes per second}} \times 1 \text{ byte} = 3072 \text{ cycles} \end{aligned}$$

The sampling time S corresponds to the latency of the analog to digital (ADC) hardware device that converts the power level of an analog signal to a digital value. We have from the hardware manual that one conversion takes 13 ADC-cycles, and from experimentation we have determined that one ADC-cycle takes 64 machine cycles, so:

$$S = 13 \text{ ADC-cycles} \times 64 \text{ cycles per ADC-cycle} = 832 \text{ cycles}$$

The easiest way of satisfying the requirements is to use the cycle-by-cycle implementation strategy of ATEMU, but unfortunately it affords no parallelism. We now explain how to extract the "hidden" parallelism by leveraging the event queue of each node in Avrora.

C. Avrora's Two Synchronization Strategies

For a requirement $wait(n, t + d)until(m, t)$, we have considered two different ways of ensuring that it be satisfied.

- **Synchronization Intervals:** In this approach, the nodes n, m synchronize periodically in intervals of at most d cycles. In other words, each node runs to the end of the current interval and waits until all the other nodes reach the same point in simulation time. The longer the interval, the less often nodes synchronize, allowing them to run concurrently for a longer time, resulting in overall better simulation performance. It would be unsafe to have the interval be longer than d, because it could allow a node to run past a point in time at which the node should have been influenced by another node's actions.
- **Wait for Neighbors:** In this approach, each node periodically notifies a global data structure how far it has progressed, measured in number of cycles. A requirement $wait(n, t + d)until(m, t)$ is met by n asking the global data structure whether m has passed t, and n blocks until it gets the answer Yes. Deadlock is impossible if for every wait...until requirement, n never waits for m to reach a point in n's future.

We observe that the cycle-by-cycle implementation strategy of ATEMU corresponds to using synchronization intervals of length 1. To solve the Sampling Problem, we observe that since S is small, using a *Synchronization Intervals* strategy would be only a small improvement over the cycle-by-cycle implementation strategy, because we would have to choose the interval length to be no more than S; a node could execute at most S cycles before it was forced to wait for all other nodes. We instead chose to solve the Sampling Problem with a *Wait for Neighbors* approach since a node typically samples the RSSI value much less frequently than it sends packets. Therefore nodes block only infrequently when they attempt to sample the RSSI.

For the Send-Receive problem, we find that each TinyOS packet consists of 40 bytes of data, so sending of bytes can occur much more frequently. Moreover, given that the latency L is large, each node can execute for a longer period before it must synchronize with the others. Therefore, Avrora solves the solves the Send-Receive problem using the Synchronization Interval approach with an interval of L, the maximal safe choice.

The class GlobalClock implements the Synchronization Intervals solution by inserting events into each Simulator's event queue periodically. GlobalClock inserts an event e_i at the beginning of each interval, where e_i is to be fired after L cycles. All threads are then started, and each simulates its program, executing its instructions, which might interact with devices, the radio, etc. Each thread consumes clock cycles until it reaches the synchronization event e_i in the event queue. When e_i fires, it will block the thread on the GlobalClock and wait until all of the simulation threads have entered the GlobalClock. After all threads have blocked, each is guaranteed to have stopped at the same local simulation time. Global information about the simulation can be computed, and the synchronization point e_{i+1} is inserted into the event queues of each node L cycles in the future. If no node has transmitted in this time interval, all threads are released to run again in parallel.

However, if any bytes have been transmitted, they will be received in the next interval, because the synchronization interval has been chosen to be L. Avrora uses this fact to compute the point in time in the next interval at which receivers will receive the radio transmission and inserts a new synchronization point d for delivery into the event queues of all nodes. The second synchronization at delivery time is needed because other nodes may begin new transmissions in the next interval which could collide with the byte being transmitted before it is delivered. When the Simulators meet at the delivery synchronization d, the data to be received is calculated by considering all recent transmissions; overlapping transmissions can be handled according to the radio model. The simplest radio model performs an arithmetic *or* on each of the bytes appropriately shifted by their relative offsets in time and discards stray bits before and after a complete byte transmission. After the data is delivered to the receiving nodes' radios, the threads are freed from the synchronization point d to run in parallel again. They will meet again at the next synchronization point e_{i+1} and repeat the process.

Avrora provides flexible instrumentation points for inserting more complex radio models than simple full-byte reception, while retaining timing accuracy. For example, a more complex radio model can be used to simulate partial preamble loss. This happens because although the software sends and receives individual bytes from the CC1000 radio, a receiver radio may not receive the first few bits from a transmission as it attempts to lock on to the signal, causing subsequent bytes delivered to the software to be shifted over by some number of bits. The software radio stack must be written to deal with this by scanning for a start symbol and then realigning the bytes. This behavior can be simulated by discarding the first few bits or bytes of a radio transmission and then delivering data shifted-over by some number of bits. Correct reception time can still be preserved by delivering the shifted byte at the correct time to the radio using a delivery event inserted into the event queue.

IV. Experimental Results

The four graphs on the next page display our experimental results. The first graph compares the scalability and performance of three sensor network simulators (ATEMU, Avrora, and TOSSIM) over the number of nodes in the network. The second graph compares the performance of three AVR simulators (ATEMU, Avrora, and Simulavr) for executing three AVR assembly benchmarks. The third graph shows measurements of radio channel utilization obtained with Avrora, and the fourth graph evaluates Avrora's scalability over the number of processors. We performed the first two experiments on a dual 3.06ghz Xeon machine with hyperthreading enabled with 4GB of RAM running Linux 2.4.20. The last two experiments were performed on a Sun V880 with 8 Ultrasparc III 900MHz processors and 16GB of RAM running Solaris 9. On the Linux machine, we used Sun's HotSpot JVM 1.4.2 to run Avrora, and on the Solaris machine, we used Sun's HotSpot 64-bit JVM 1.4.2. The performance results for Avrora on multi-node simulations are fairly consistent across VM implementations; experiments with IBM's Java 1.4.2 and Sun's Beta 1.5.0 produced results that are very similar. Each of the results is an average over three runs. In multi-node simulations, each node's starting time is randomly perturbed between 0 and 1 second in order to avoid artificial lock-step timing phenomenon.

A. Comparison of ATEMU, Avrora, and TOSSIM

The first graph illustrates the scalability of three sensor network simulators as measured by their performance varying over the number of nodes in the network. In pure event-based simulation, the number of events is often the primary variable of interest, but for sensors that can sense, compute, and communicate, events do not capture the

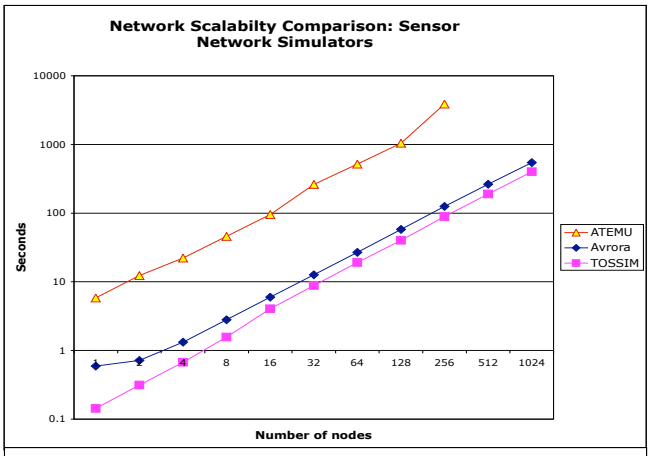
cost of simulating program execution, which can be significant. In this experiment, half of the nodes run the CntToRfm TinyOS program and half run RfmToLeds, each running ten simulated seconds. CntToRfm was described earlier; RfmToLeds listens for radio packets and displays them as a binary value on the LEDs attached to the physical node. TOSSIM compiles the program for the host machine (Linux) and runs it, while ATEMU and Avrora execute the same AVR binary that would be loaded onto the actual hardware device. We compare all three simulators in their most accurate configurations; for TOSSIM this means a bit-level radio without precise timing; ATEMU simulates the byte-oriented interface to the radio and its transmissions at the bit level with precise timing; Avrora works at the byte level with precise timing. For 512 nodes, ATEMU did not complete within 8 hours. We can see that all three simulators scale linearly with the number of nodes, and TOSSIM has the best performance. We found that Avrora is approximately 50% slower than TOSSIM, while ATEMU lags behind Avrora by a factor of 20 and behind TOSSIM by a factor of 30. Notice that Avrora's performance improves quickly on small numbers of nodes; this is partly due to the multithreaded architecture, and partly due to amortizing some startup costs associated with Java class loading and JIT compilation.

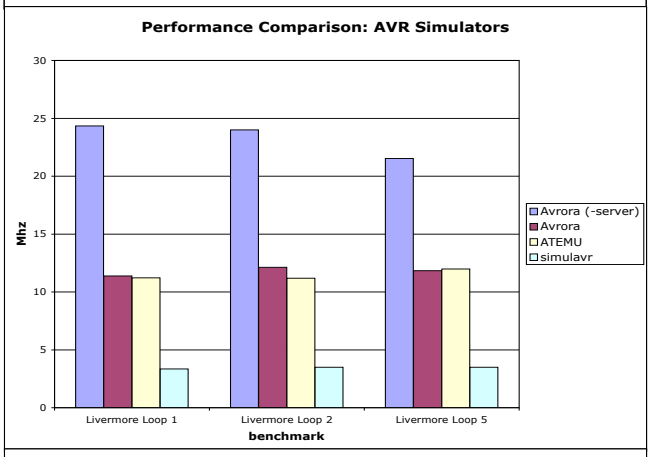
B. Comparison of ATEMU, Avrora, and Simulavr

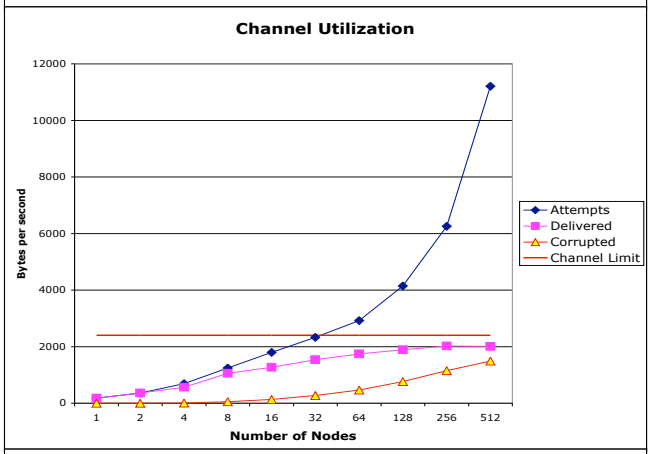
The second graph compares Avrora with ATEMU and Simulavr [13], to compare the raw performance of interpreting AVR instructions. We measure megahertz (rather than seconds or MIPS) to compare with the speed of an actual AVR processor and capture the cost of simulating device operation or sleeping, which a MIPS measure cannot. We use three computationally intensive Livermore Loops as benchmarks; these are not sensor network programs. They are coded in C [14] and compiled to AVR machine code by avr-gcc. We can see that Avrora and ATEMU are closely matched when Avrora is run on the default Hotspot VM, but Avrora runs much faster on the server Hotspot VM. Simulavr performs poor in comparison, suggesting it is not suitable for large-scale simulation of sensors. ATEMU and Simulavr were compiled with gcc 3.2.2 with -O2 optimization level; using -O9 did not measurably impact the runtimes of either ATEMU or Simulavr. Running on the HotSpot server VM, Avrora gives the best performance due to the event queue organization and fast VM.

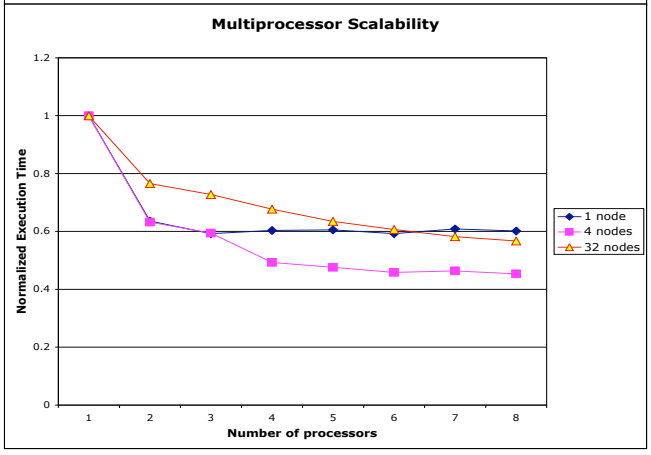
C. Radio channel utilization

Precise network measurements on a fine-grained timing scale are necessary for radio stack development and tuning, which often has delicate timed code. It would be useful to measure and tune this layer of the stack in the presence of applications with varying behaviors, including significant computation and bi-directional communication. Simulating such applications with precise time, which cannot be done with TOSSIM, is a pre-requisite for an accurate picture of MAC layer behavior. The third graph shows measurements obtained with Avrora of the radio channel utilization for a network of nodes. In this experiment, each node runs the CntToRfm application. We were able to identify and measure a race condition present in the MAC protocol of TinyOS where a delay between sensing a clear channel and beginning transmission admits an opportunity for collisions to occur. We can see that for a network of smaller than 32 nodes, the RSSI sampling and exponential backoff restrain the number of attempted byte transmissions to be logarithmic in the network size, but when the network becomes dense, the number of attempts rises dramatically, and the number of bytes corrupted by colliding transmissions rises sharply. From this we see that as the density of the network increases,









more nodes will attempt to "pile-on" a quiet period when they sample the RSSI and find it to be quiet, resulting in massive collisions. Our experiments with larger numbers of nodes confirm that.

D. Scalability over number of processors

The fourth graph illustrates Avrora's scalability over the number of processors available on the machine. In this experiment we use the Sun V880 server with 8 Ultrasparc III processors to evaluate the benefits from Avrora's highly multithreaded architecture. As in our first graph, half of the nodes run the CntToRfm application and half run the RfmToLeds application. For each run, we restrict the number of processors available to the Java Virtual Machine and normalize with respect to the running time for the simulation on a single processor. The fourth graph validates our expectation by showing that Avrora's performance increases with the number of processors. For one node, we can see that the simulation benefits from more processors by allowing the Java virtual machine to run the JIT compiler thread in the background. For four nodes, we see the simulation time is cut in half on four processors. However, when the number of nodes is greater than the number of processors, contention for global data structures and OS thread switching overhead limit the performance benefits significantly, as is seen in the much smaller gains for the 32 node network. On the Sun V880 machine, we successfully simulated a network of 10,000 nodes with one CntToRfm and the remaining nodes running RfmToLeds for 2 simulated seconds in a total of 30 minutes. For sensor network programs with more in-network processing such as acoustic processing or data compression, we expect the benefits from multithreaded simulation to be more pronounced, since simulating the computation happens concurrently, without needing frequent synchronization. Neither CntToRfm or RfmToLeds are particularly computationally intensive, yet the simulation performance still benefits from Avrora's highly multithreaded architecture.

V. Conclusion

We have implemented a sensor network simulator which is cycle-accurate like ATEMU and scalable like TOSSIM, while merely 50% slower than TOSSIM. This solves an open problem stated by Levis et al. [5, Section 6]. The event-queue model for cycle-accurate simulation of device and communication behavior allows improved interpreter performance and enables an essential sleep optimization. Avrora enables us to validate time-dependent properties of large-scale networks. Sensor network deployments can now be carried out with more confidence after detailed instruction-level simulation has been performed.

One limitation of Avrora is that it does not model clock drift, a phenomenon where nodes may run at slightly different clock frequencies over time due to manufacturing tolerances, temperature, and battery performance. In recent work, we model distance-attenuation for multi-hop scenarios, but we do not yet model mobility [15], [16], [17]. Validating our timing results with real-world systems down to the clock cycle level for all devices remains as future work; until now we have verified our timings results against those of ATEMU for large programs with radio communication and against real hardware only for small, simple programs.

Although described in this paper, we have not yet implemented partial preamble loss. We expect that its implementation will allow even larger synchronization intervals, since the latency between the transmission of the first byte of a packet and the reception of the first byte will be larger because the first few bytes can be lost. Additionally, we expect our approach to scale well to new, packet-oriented radio devices. Since the software sends the entire packet to the radio to be buffered and then instructs the radio to transmit it, and the radio buffers packets it is receiving, the latency value L is much larger (however, the bitrate of the new radios is also higher).

Avrora is available from http://compilers.cs.ucla.edu/avrora.

Acknowledgments. We thank Kevin Chang, Simon Han, Krishna Nandivada, and Vidyut Samanta for helpful comments on a draft of the paper. We thank Olaf Landsiedel for his energy model work in Avrora and Simon Han for work on timing validation against real hardware. We thank the NSF Center for Embedded Networked Sensing (CENS) for access to the Sun V880 server, which was a generous donation from Sun Microsystems. We were partially supported by the NSF ITR award #0427202.

References

[1] S. Park, A. Savvides, and M. Srivastava, "Sensorsim: a simulation framework for sensor networks," in *Proceedings of MSWiM'00, 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2000, pp. 104–111.

[2] J. Liu, L. F. Perrone, D. M. Nicol, M. Liljenstam, C. Elliott, and D. Pearson, "Simulation modeling of large-scale ad-hoc sensor networks," in *Proceedings of Euro-SIW'01, European Simulation Interoperability Workshop*, 2001.

[3] S. Sundresh, W. Kim, and G. Agha, "SENS: A sensor, environment and network simulator," in *Proceedings of 37th Annual Simulation Symposium*, 2004, pp. 221–230.

[4] L. F. Perrone and D. Nicol, "A scalable simulator for TinyOS applications," in *Proceedings of WSC'02, Winter Simulation Conference*, 2002.

[5] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proceedings of SenSys'03, First ACM Conference on Embedded Networked Sensor Systems*, 2003.

[6] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "Emstar: a software environment for developing and deploying wireless sensor networks," in *Proceedings of the USENIX Technical Conference*, 2004.

[7] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, and M. Karir, "ATEMU: A fine-grained sensor network simulator," in *Proceedings of SECON'04, First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. Pister, "System architecture directions for networked sensors," in *Proceedings of ASPLOS'00, International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104.

[9] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of PLDI'03, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003, pp. 1–11.

[10] C.-C. S. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "SOS operating system," http://nesl.ee.ucla.edu/projects/sos.

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] O. Landsiedel, K. Wehrle, S. Rieche, S. Gotz, and L. Petrak, "Accurate prediction of power consumption in sensor networks," 2004, manuscript.

[13] T. A. Roth, "Simulavr: an AVR simulator," http://savannah.nongnu.org/projects/simulavr.

[14] T. Peters, "Livermore loops coded in C," 1992, www.netlib.org/benchmark/livermorec.

[15] L. Bajaj, M. Takai, R. Ahuja, and R. Bagrodia, "Simulation of large-scale heterogeneous communication systems," in *Proceedings of MILCOM'99, Military Communications Conference*, 1999, pp. 1396–1400, Volume 2.

[16] V. Naoumov and T. Gross, "Simulation of large ad hoc networks," in *Proceedings of MSWiM'03, Sixth ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2003.

[17] J. Liu, Y. Yuan, D. Nicol, R. Gray, C. Newport, D. Kotz, and L. Perrone, "Simulation validation using direct execution of wireless ad-hoc routing protocols," in *Proceedings of PADS'04, 18th Workshop on Parallel and Distributed Simulation*, 2004.