Improving the Speed and Scalability of Distributed Simulations of Sensor Networks

Zhong-Yi Jin, Rajesh Gupta
Dept. of Computer Science & Eng, UCSD
{zhjin, rgupta}@cs.ucsd.edu

ABSTRACT

Distributed simulation techniques are commonly used to improve the speed and scalability of wireless sensor network simulators. However, accurate simulations of dynamic interactions of sensor network applications incur large synchronization overheads and severely limit the performance of existing distributed simulators. In this paper, we present two novel techniques that significantly reduce such overheads by minimizing the number of sensor node synchronizations during simulations. These techniques work by exploiting radio and MAC specific characteristics without reducing simulation accuracy. In addition, we present a new probing mechanism that makes it possible to exploit any potential application specific characteristics for synchronization reductions. We implement and evaluate these techniques in a cycle accurate distributed simulation framework that we developed based on Avrora. In our experiments, the radio-level technique achieves a speedup of 2 to 3 times in simulating 1-hop networks with 32 to 256 nodes. With default backoffs, the MAC-level technique achieves a speedup of 1.1 to 1.3 times in the best case scenarios of simulating 32 and 64 nodes. In our multi-hop flooding tests, together they achieve a speedup of 1.5 to 1.8 times in simulating networks with 36 to 144 nodes. The experiments also demonstrate that the speedups can be significantly larger as the techniques scale with the number of processors and radio-off/MAC-backoff time.

Categories and Subject Descriptors

I.6.7 [**SIMULATION AND MODELING**]: Simulation Support Systems—*Sensor Networks*

IPSN'09, April 13–16, 2009, San Francisco, California, USA.

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Distributed simulation, Sensor network simulator

1. INTRODUCTION

Wireless sensor network (WSN) simulators are important in developing and debugging WSN applications. By running sensor network programs on top of simulated sensor nodes inside simulated environments, the states and interactions of sensor network programs can be inspected and studied easily and repeatedly. In addition, the properties of the simulated entities (simulation models) such as the locations of sensor nodes and the inputs to the sensor nodes can be readily changed before or during simulations. By choosing appropriate simulation models, one can build an entire WSN application, including the underlying operating system, using simulations.

There are two key requirements to WSN simulators: fidelity and speed. Fidelity reflects bit and temporal accuracy of events and actions. High fidelity often leads to low simulation speed [10], defined as the ratio of simulation time to wallclock time. Simulation time is the virtual clock time in the simulated models [2] while wallclock time corresponds to the actual physical time used in running the simulation program. A simulation speed of 1 indicates that the simulated sensor nodes advance at the same rate as real sensor nodes and this type of simulation is called real time simulation. In general, real time speed is required to use simulations for interactive tasks such as debugging and testing.

The fidelity of WSN simulators is rapidly increasing with the use of high fidelity simulation models [13, 18, 17, 11]. However, most of these improvements of fidelity are largely achieved at the cost of decreased simulation speed and scalability because significant computational resources are required to execute high fidelity simulation models. For example, even with TOSSIM [13], a pop-

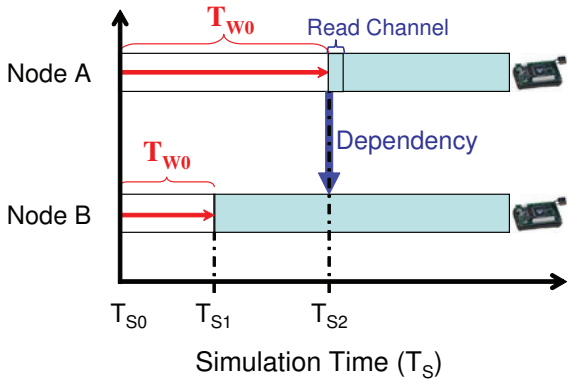

Figure 1: The progress of simulating in parallel two nodes that are in direct communication range of each other

ular and one of the fastest sequential WSN simulators, one can only simulate, without cycle accuracy, about 32 nodes in real time on fast computers [13, 20]. As alternatives, parallel and distributed WSN simulators [20, 21, 3] leverage the combined computing resources of multiple processors/cores on the same computer and on a network of computers, respectively. They can significantly improve simulation speed and scalability because sensor nodes may be simulated in parallel on different processors or computers. However, when sensor nodes are simulated in parallel, their simulation speeds can vary [20]. This is either caused by the differences in sensor node programs, sensor node inputs and values of random variables or due to the differences in simulation environments, such as different processor speeds and operating system scheduling policies. Since different nodes may get simulated at different speeds, simulated nodes often need to synchronize with each other to preserve causality of events and to ensure correct simulation results.

Synchronization of two sensor nodes is illustrated in Figure 1 which shows the progress of simulating in parallel two sensor nodes that are within direct communication range of each other. After starting the simulation for T_{W0} seconds of wallclock time, Node A advances to simulation time T_{S2} while Node B only advances to T_{S1}. The speeds of simulating the two nodes could be different for any of the reasons described earlier. At T_{S2}, Node A is supposed to read the wireless channel and continue its execution along different paths based on whether there are active wireless transmissions or not. However, at T_{W0}, Node A may not be simulated any further than T_{S2} because Node A does not know, at T_{W0}, whether Node B is going to transmit at T_{S2} or not (since $T_{S1} < T_{S2}$). In other words, at T_{S2}, the input of Node A depends on the output of Node B.

To maintain such dependencies, simulated sensor nodes often need to synchronize with each other during simulations. There are two general approaches to handle synchronizations: conservative [2] or optimistic [9]. Conservative synchronization requires that Node A waits at T_{S2} until the simulation time of Node B reaches T_{S2}. The optimistic approach, on the other hand, would allow Node A to advance assuming there will be no transmissions from Node B at T_{S2}. However, the entire simulation state of Node A at T_{S2} has to be saved. If Node A later detects that Node B actually transmits to Node A at T_{S2}, it can correct the mistake by rolling back to the saved state and start again. To the best of our knowledge, almost all distributed WSN simulators are based on the conservative approach as it is simpler to implement and has a smaller memory footprint.

Synchronizations bring significant overheads to distributed simulations. With the conservative approach, the overheads can be divided into management overheads and communication overheads. Management overheads come from managing the threads or processes that simulate sensor nodes. For example, to maximize parallel use of computational resources, the thread or process simulating Node A in Figure 1 needs to be suspended while waiting for Node B so another thread or process simulating another node can be swapped in for execution. Suspended nodes also need to be swapped back in for simulation later on. These usually involve context switches and large numbers of them would significantly reduce simulation speed and scalability. Communication overheads arise because nodes need to communicate their progresses to each other during simulations. For example, in Figure 1, Node B must notify Node A after it advances past T_{S2} so that Node A can continue. Communicating across processes or threads is generally expensive. In the case where nodes are simulated on different computers, the communication overheads could be very high as messages have to be sent across slow networks.

The performance gains of existing distributed WSN simulators are often compromised by the rising overheads due to inter-node synchronizations. For example, with Avrora [20], a cycle accurate parallel WSN simulator, it is faster to simulate 32 nodes with 2 processors than using all 8 processors of a parallel computer in some of our tests (Figures 4 and 11) because of large overheads in synchronizing threads (1 node/thread) across processors. In the case of DiSenS [21], a cycle accurate distributed WSN simulator, if all nodes are within communication range, DiSenS needs 16 computers to simulate 16 nodes in real time despite the fact that each of the computers can simulate 8 nodes in real time [21]. This sub-linear performance in DiSenS is due to the large communication overheads in synchronizing nodes

that are simulated on different computers.

In this paper, we propose two novel techniques that greatly reduce synchronization overheads by cutting the number of synchronizations. The techniques work by exploiting radio and MAC specific characteristics without reducing simulation fidelity. They are effective even when node processors or radios are active and can significantly improve simulation speed and scalability. To support the MAC-level technique, we develop a new probing mechanism that tracks the internal states of any WSN applications during simulations for synchronization reductions. We validate the proposed techniques by their implementations in PolarLite, a cycle accurate distributed simulation framework that builds upon Avrora and serves as the underlying simulation engine [10].

We discuss related work in Section 2. The speedup techniques are described in Section 3 and their implementations in Section 4. In Section 5 we present the results of evaluating the techniques followed by the conclusion and future work in Section 6.

2. RELATED WORK

Previous work in improving the speed and scalability of WSN simulators can be broadly divided into two categories. The first category focuses on reducing the computational demands of individual simulation models without significantly reducing fidelity. For example, it is very computationally expensive to emulate an actual sensor node processor in a simulation model for cycle accurate simulations [17]. To reduce the large computational needs, TimeTossim [11] automatically instruments applications at source code level with cycle counts and compiles the instrumented code into the native instructions of simulation computers for fast executions. This significantly increases simulation speed while achieving a cycle accuracy of up to 99%. However, maintaining cycle counts also slows down TimeTossim to about 1/10 the speed of TOSSIM [13] which TimeTossim is based on. Our work makes this type of effort scalable on multiple processors/cores.

The second category of work focuses on reducing overheads in parallel and distributed simulations. DiSenS reduces the overheads of synchronizing nodes across computers by using the sensor network topology information to partition nodes into groups that do not communicate frequently and simulating each group on a separate computer [21]. However, this technique only works well if most of the nodes are not within direct communication range as described in the paper. In [10], we describe a technique that uses sensor node sleep time to reduce the number of synchronizations. As demonstrated in the paper, using the node-sleep-time-based technique can significantly increase speed and scalability of distributed WSN simulators. However, the technique is only able to exploit for speedup the time when both the processor and the radio of a sensor node are off. This is because it is not possible to predict the exact radio wakeup time when the processor is running. As a result, we can not apply the node-sleep-time-based technique if the radio is on or if the sensor node processor is kept alive by tasks such as reading and monitoring sensor inputs.

The techniques we propose in this paper are new and are orthogonal to the previous techniques. They are not bounded by the number of neighboring nodes and are effective even when the processor or the radio of a node is active. While the techniques are developed specifically for simulating WSNs, they can also be applied to any general distributed network simulators such as NS-3 [7] for improved performance in simulating wireless networks.

There is a large body of work on improving the speed and scalability of distributed discrete event driven simulators in general. Among them, exploiting lookahead time is a commonly used conservative approach [5, 6, 14]. Our techniques belong to this category in the sense that we also improve speed and scalability by increasing the lookahead time. However, our techniques are fundamentally different as we use application specific characteristics in a different context to increase lookahead time.

3. REDUCING SYNCHRONIZATIONS IN DISTRIBUTED SIMULATIONS OF WSNS

Sensor node synchronizations are required for enforcing dependencies between simulated sensor nodes due to their interactions over wireless channels. Our approach to increase simulation speed and scalability by reducing synchronizations is based on identifying and exploiting parallelism between nodes according to their communication capabilities. A node can "lookahead" for a minimum guaranteed interval to identify periods when no transmitting or receiving events in a discrete event driven simulator would occur. Such a lookahead is often closely tied to the event scheduling algorithm used in the simulators. Based on the distributed scheduling algorithm in [10], we present two techniques for reducing the number of synchronization events at the radio-level or at the MAC-level.

3.1 Radio-level Speedup Technique

Our radio-level speedup technique exploits the radio off time when a sensor node radio is duty cycled. Radio-level duty cycling works by selectively turning radios on and off. Since radios are one of the most power consuming components of sensor nodes, radio-level duty cycling is ubiquitously used in WSNs to reduce energy consumption and extend working life of energy constrained sensor nodes. Due to its wide applications and the complex

tradeoffs in energy savings and communication overheads, radio-level duty cycling is commonly built into energy efficient WSN MACs such as S-MAC [22] and B-MAC [15].

Our radio-level speedup technique is illustrated in Figure 2 which shows the progress of simulating two sensor nodes in parallel. In this simulation, Node B turns its radio off at time T_{S1} and puts it back on at time T_{Sx}. With existing distributed simulators, after running the simulation for T_{W0} seconds of wallclock time, Node A has to wait at T_{S3} for Node B to catch up from T_{S2}, despite the fact that node B will not transmit any packets at T_{S3}. Ideally, we can avoid this unnecessary synchronization by having Node B notify node A at time T_{S1} that its radio is off until T_{Sx}. However, this will not work as it is not possible for Node B to predict the exact radio wakeup time T_{Sx} at T_{S1}. This is because while the radio is off at T_{S1}, the sensor node processor is still running and it can turn the radio back on at any time based on current states, application logics and sensor readings. In other words, it is just not possible for Node B to predict when the radio will be turned on in the future.

Instead of predicting the exact radio wakeup time, our radio-level speedup technique exploits the radio off period by calculating the earliest possible communication time, $T_{EarliestCom}$. $T_{EarliestCom}$ is the earliest time that a turned off radio can be used to send or receive data over wireless channels and can be calculated based on T_{Act}, the amount of time to fully activate a turned off radio. A turned off radio can not be activated instantly for sending or receiving data. It takes time for the radio to be initialized and become fully functional [16]. For example, the CC1000 radio of Mica2 nodes [4] needs 2.45ms to be activated and the CC2420 radio of Telos nodes [16] needs about 1.66ms without counting the SPI acquisition time [12]. The exact delays in terms of numbers of clock cycles are hard coded into WSN MAC protocols and can be easily identified in the source code. For example, in TinyOS 1.1 [8, 19], B-MAC waits for a total of 34300 clock cycles for the CC1000 radio of MICA2 by calling the `TOSH_uwait` function. While the delays seem to be small, they are significantly larger than typical lookahead times in simulating WSNs. For example, it is about 11 times larger than the 3072 clock cycle lookahead time in simulating Mica2 nodes [20, 10]. As mentioned, lookahead time is defined as the maximum amount of simulation time that a simulated sensor node can advance freely without synchronizing with other simulated sensor nodes [10].

Our radio-level speedup technique works by tracking when sensor node radios are turned on and off. When we detect that a sensor node radio is turned off, we immediately send its $T_{EarliestCom}$ in a clock synchronization message to all neighboring nodes and then repeatedly send the latest $T_{EarliestCom}$ every T_{Act} time until the radio is detected to be turned on. $T_{EarliestCom}$ can be calculated as the sum of current simulation time and T_{Act}. As a result, neighboring nodes no longer need to synchronize with the radio off node until the latest $T_{EarliestCom}$. For example, as shown in Figure 2, when we detect that Node B turns its radio off at T_{S1}, we immediately send its $T_{EarliestCom}$ to Node A and repeat that every T_{Act} time which is fixed according to the radio of Node B. The clock synchronization messages are shown as arrows from Node B to Node A in the figure. Upon receiving the second $T_{EarliestCom}$, the lookahead of node A increases to a time beyond T_{S3} and therefore it no longer needs to wait at T_{S3} after T_{W0} seconds of simulation. In other words, Node A knows before T_{W0} that Node B will not be able to transmit any packet at T_{S3}. Since the increase of lookahead time ($T_{Act} - OldLookAheadTime$) may just be a very small fraction of the total radio off period, it is critical to repeatedly send $T_{EarliestCom}$ every T_{Act} time to fully exploit the entire radio off period.

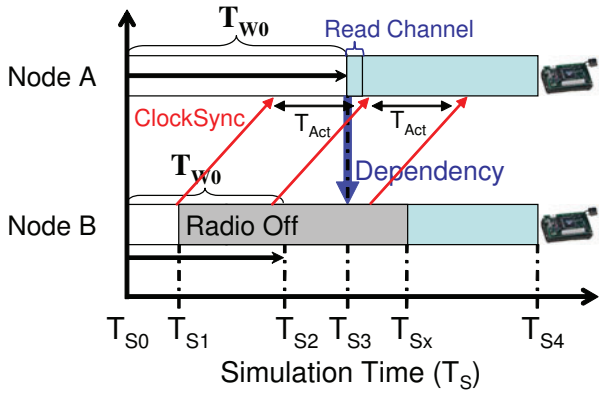

Figure 2: The progress of simulating two nodes that are in direct communication range with the radio-level speedup technique

The radio-level speedup technique also reduces the number of clock synchronizations. In distributed simulations, clock synchronization messages are used to send the simulation time of a node to all its neighboring nodes so causality can be maintained and suspended waiting nodes can be revived. To maximize parallelism in simulations, a node needs to send 1 clock synchronization message for every lookahead time of its neighboring nodes [20, 21, 10]. Since our radio-level speedup technique increases the lookahead times of neighboring nodes, the number of clock synchronizations can be greatly reduced. For example, in the case of simulating Mica2 nodes, one $T_{EarliestCom}$ message can increase lookahead time by a factor of 11 and therefore eliminates

10 clock synchronization messages.

3.2 MAC-level Speedup Technique

While the radio-level speedup technique takes advantage of physical delays in WSN radios, our MAC-level speedup technique exploits the random backoff behaviors of WSN MACs. Almost all WSN MACs need to perform random backoffs to avoid concurrent transmissions [22, 15]. For example, before transmitting a packet, B-MAC would first perform an initial backoff. If the channel is not clear after the initial backoff, B-MAC needs to repeatedly perform congestion backoffs until the channel is clear. Because a MAC will not transmit any data during backoff periods, we are able to exploit the backoff times for speedups. Although the backoff times are random and MAC specific, they are usually a lot longer than typical lookahead times in simulating WSNs. For example, in the case of B-MAC, the default initial backoff is 1 to 32 times longer than the 3072 clock cycle lookahead time in simulating Mica2 nodes. The default congestion backoff is 1 to 16 times longer in B-MAC.

Our MAC-level speedup technique is illustrated in Figure 3 which is similar to Figure 2 except Node B enters into a backoff period from T_{S1} to T_{S4}. The MAC-level speedup technique works by detecting the start and the duration of a backoff period. When the start of a backoff period is identified, the end time of the backoff period is first calculated based on the duration of the period and then sent to the neighboring nodes. This effectively increases the lookahead times of neighboring nodes and helps to eliminate unnecessary synchronizations. For example, in order to avoid the unnecessary synchronization of Node A at T_{S3} after running the simulation for T_{W0} seconds of wallclock time, our MAC-level speedup technique first detects at T_{S1} the start of Node B's backoff period as well as the duration of the backoff period. Then we compute the end time of the backoff period and send that in a clock synchronization message to Node A. Once Node A knows that Node B will not transmit until T_{S4}, it no longer needs to wait at T_{S3}. Similar to the radio-level speedup technique, the MAC-level technique also reduces the number of clock synchronizations, which provides additional speedup. We discuss how to detect the start and the duration of a random backoff period in Section 4.

The MAC-level speedup technique is a good complement to the radio-level technique as WSNs usually have very bursty traffic loads. Nodes in a WSN usually do not communicate frequently and can duty cycle their radios extensively until certain triggering events occur. Once triggered by those events, nodes need to actively communicate and interact with each other to accomplish certain tasks. Our MAC-level technique is most effective when wireless channels are busy.

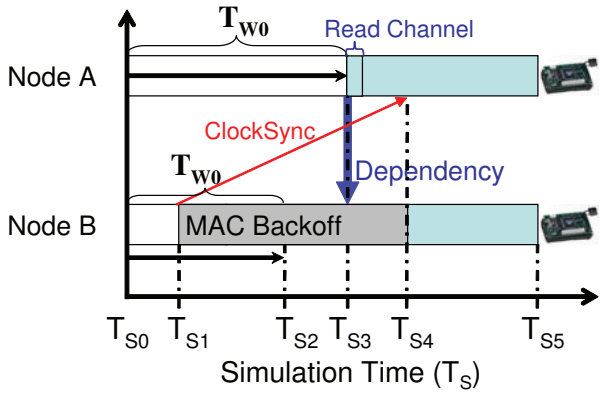

Figure 3: The progress of simulating two nodes that are in direct communication range with the MAC-level speedup technique

4. IMPLEMENTATION

The proposed speedup techniques are implemented in PolarLite, a distributed simulation framework that we developed based on Avrora [10]. PolarLite provides the same level of cycle accurate simulation as Avrora but uses a distributed synchronization engine instead of Avrora's centralized one. This makes it possible to implement and evaluate our speedup techniques in a distributed simulation environment. The synchronization engine of PolarLite is based on the distributed synchronization algorithm described in [10]. With this algorithm, nodes can be synchronized separately according to their own lookahead times. In other words, if a node is not accessing the wireless channel, it only needs to communicate to neighboring nodes its simulation time and does not need to wait for any other nodes. Some minor code optimizations are made to the PolarLite synchronization engine in [10] and the speedup techniques are implemented on the newer version. As a result, we evaluate the speedup techniques by comparing with the newer version in Section 5. As with Avrora, PolarLite allocates one thread for each simulated node and relies on the Java virtual machine (JVM) to assign runnable threads to any available processors on an SMP computer.

To implement the radio-level speedup technique, we need to detect when radios are turned on and off. In discrete event driven simulations, the changes of radio states are triggered by events and can be tracked. For example, in our framework, we detect the radio on/off time by tracking the IO events that access the registers of simulated radios.

Detecting MAC backoff times and durations for the MAC-level speedup technique are considerately more difficult. The backoffs are MAC and application specific and generally do not correlate to any unique events

or actions that can be easily tracked. In addition, the backoff durations are completely random. One possible solution to this problem is to insert special code into the source code of an application that is to be simulated. These special pieces of code are compiled into the application and used to report to the simulator the MAC backoff times and durations during simulations. However, this technique does not work for cycle accurate simulators like ours.

Cycle accurate sensor network simulators [17, 20, 21, 10] offer the highest level of fidelity among all types of sensor network simulators. They provide simulation models that emulate the functions of major hardware components of a sensor node, mainly the processor. Therefore, they take as inputs the same binary code (images) that are executed by real sensor nodes. To detect backoff times and durations without changing the source code of the applications under simulation, we develop a generic probing mechanism based on pattern matching to expose the internal states of sensor network applications during simulations.

Our probing mechanism works by first using patterns to pinpoint from compiled applications the machine instructions that represent events of interest during the start of a simulation. The identified instructions are then replaced by corresponding "hook" instructions to report the internal states of the applications, such as the backoff durations, to a simulator during simulations. Hook instructions are artificial instructions that behave exactly the same as the original instructions they replace except they will pass to a simulator the memory locations or registers accessed by the original instructions during simulations. The values stored in those locations are the internal states of the applications that correspond to the events of interest. Since an instruction may access multiple locations, we associate with each pattern a list that indicates the operands of interest based on their order in the instruction. Therefore, an instruction may be translated into different hook instructions according to the list. To maintain cycle accuracy, our simulator ensures that the hook instructions consume the same number of clock cycles as the original instructions.

Our current implementation of the probing mechanism is largely based on existing constructs from Avrora. In Avrora, a compiled program (object file) is disassembled first before simulation and each disassembled instruction is loaded into a separate instruction object (Java object). Once an instruction of interest is identified with pattern matching, we encapsulate the corresponding instruction object in a new hook instruction object and attach to the hook instruction object a probe object created specifically for the pattern. When executed, the hook instruction invokes the original instruction first and then calls the probe attached to it. It is the specific probe that turns a regular hook instruction into a unique hook instruction and reports values of interest to a simulator during simulations.

We do not use addresses to identify instructions of interest because addresses tend to vary across compilations after source code changes, even if the changes are at places not related to the instructions. With pattern matching, we only need to create a set of patterns once if the corresponding source code does not change. For example, if an application is written with TinyOS, the instructions that assign backoff durations to B-MAC are part of the OS, regardless of whether the backoffs are calculated by default functions in the OS or user supplied functions in the application. Therefore, we only need to create a set of patterns once for each version of TinyOS to track the backoff times in B-MAC during simulations.

We use regular expressions for pattern matching. To uniquely identify an instruction, we need to match additional instructions before or after that instruction as well. In the current implementation, the backoff matching process for B-MAC is hard coded in our simulator. To match the initial backoff, we first locate the block of code that corresponds to the *send* function in a disassembled program by using the function name (symbol name). The *send* function is a part of TinyOS and is where the initial backoff calculation function is invoked. Then we match within this code block a continuous sequence of instructions (*sts,sts,sts,lds,out,and,brne,rjmp*) which are instructions that immediately follow the initial backoff calculation code. Note that we only need to match the names of the instructions in the sequence. Once this pattern is found, the value for initial backoff can be tracked via the first 2 *sts* instructions in the matched code. Similarly, we can identify the instructions that store congestion backoffs. For simplicity, we consider that MAC backoffs start at the times that the hook instructions report the backoff durations. It is safe to do so as no data will be sent from this point on until the end of the backoff periods.

Note that we can not simply use the symbol names of the backoff calculation functions for pattern matching because these functions are in-lined by the compiler. However, there are always some caller functions in TinyOS, such as the *send*, that are not in-lined due to space and other constrains. Based on these functions, we can create patterns that remain the same as long as the functions do not change.

5. EVALUATION

We conduct a series of experiments to evaluate the performance of our speedup techniques. All experiments are carried out on an SMP server running Linux 2.6.24.

Table 1: Radio off periods under different duty cycling modes of B-MAC

Duty cycling Mode	Radio off Time (ms)
0	0
1	20
2	85
3	135
4	185

The server features a total of 8 cores on 2 Intel Xeon 3.0GHz CPUs and 16GBytes of RAM. Sun's Java 1.6.0 is used to run all experiments. Simulation speed is calculated using Equation (1) according to the definition in Section 1. Note that the numerator of Equation (1) is the total simulation time in units of clock cycles and consequently the calculated speed is in units of Hz.

$$Speed = \frac{total\ number\ of\ simulated\ clock\ cycles}{(execution\ time) \times (number\ of\ nodes)} \quad (1)$$

All of the experiments are based on the CountSend (sender) and CountReceive (receiver) programs from the TinyOS 1.1 distribution. They are similar in nature to the programs used by other WSN simulators in evaluating their performance [13, 20, 21]. CountSend broadcasts at a fixed interval the value of a continuously increasing counter. CountReceive simply listens for messages sent by CountSend and displays the last received value on LEDs. The programs are executed on simulated Mica2 nodes [4] and the starting time of each node is randomly selected between 0 and 1 second of simulation time to avoid any artificial time locks. All simulations are run for 300 seconds of simulation time and for each experiment we take average of three runs as the results.

5.1 Performance of Radio-level Technique

For experiments in this section, we modify the sender and receiver programs slightly to enable B-MAC's built-in radio-level duty cycling feature. This can be done by calling the SetListeningMode and SetTransmitMode functions of TinyOS 1.1 at start. B-MAC supports a total of 7 radio-level duty cycling modes in TinyOS 1.1 and the 5 modes used in our experiments are shown in Table 1. Once enabled, B-MAC turns a radio off periodically for a duration corresponding to the duty cycling mode. The radio is turned back on either when there are data to transmit or a radio off period ends. The radio is turned off again once there are no pending packets to transmit and the channel is clear for a fixed period of time. In the case of TinyOS 1.1 and Mica2 nodes, the channel clear time is the amount of time to transmit 8 bytes over the radio [8, 19].

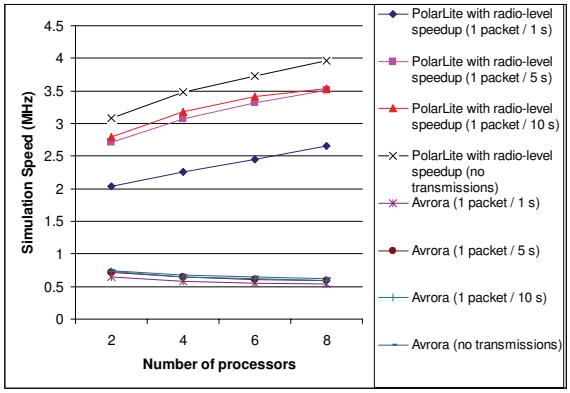

Figure 4: Speed of simulating with Avrora and PolarLite running the radio-level speedup (1 sender 31 receivers, mode 3)

5.1.1 *Speed and Scalability with respect to the number of processors*

Our first set of experiments evaluates the performance of the radio-level speedup technique over different numbers of processors, or cores in this case. For these experiments, we simulate a WSN of 32 nodes that are within direct communication range using 2 to 8 processors. The 32 nodes are set up in such a way that one node is configured as a sender and the rest as receivers. Since all nodes are within direct communication range, any one of the nodes can be chosen as the sender. The frequency that the sender transmits packets is varied for different experiments. The radio-level duty cycling modes of all nodes are set to 3. For comparisons, we conduct the same experiments using Avrora, PolarLite without any speedups and PolarLite with the radio-level speedup.

As a baseline, Figure 4 compares the speeds of simulating with Avrora and PolarLite running the radio-level speedup technique. We can see that PolarLite running the radio-level speedup technique is considerably faster than Avrora (up to 544% or 6.44 times) and scales with the number of processors. In contrast, the speeds of simulating with Avrora decrease with increasing number of processors in this set of experiments due to larger synchronization overheads[1].

We can also see in Figure 4 that the speeds of simulating with our radio-level speedup technique increase with transmission intervals. This is because our radio-level speedup technique is based on exploiting radio off time and large transmission intervals increase that. Note that at a given radio-level duty cycling mode, increasing the transmission intervals will also increase the radio off time of the receivers because radios have to be left on

[1]Our results are different from results of similar expriments in [20] as our experiments use faster 3.0GHz CPUs, compared to 900MHz ones of theirs.

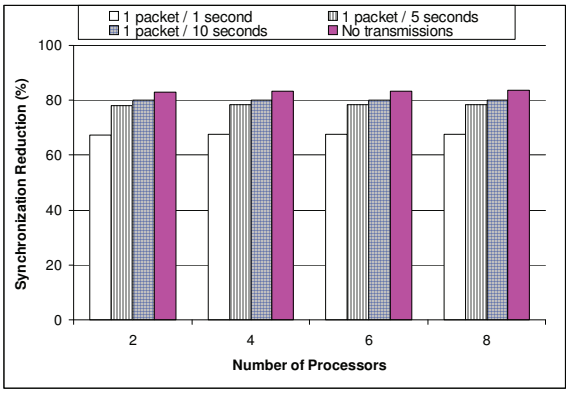

Figure 5: Percentage reductions of synchronizations using the radio-level speedup technique in PolarLite (1 sender 31 receivers, mode 3)

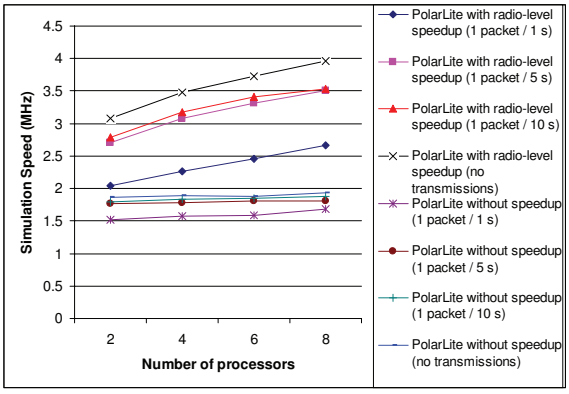

Figure 6: Speed of simulating with and without the radio-level speedup technique in PolarLite (1 sender 31 receivers, mode 3)

when receiving packets. However, as shown in Figure 4, the percentage increases of simulation speed with the radio-level speedup technique decrease quickly with increasing transmission intervals. This is due to the fact that when transmission intervals increase, the radio off time is determined more by the radio-level duty cycling mode than by the transmission intervals.

Figure 5 shows the percentage reductions of synchronizations based on numbers collected by running with and without the radio-level speedup technique in PolarLite. For accurate evaluations, we only show synchronization reductions within our PolarLite framework because PolarLite and Avrora are based on different synchronization algorithms and our speedup techniques are only implemented in PolarLite.

As shown in Figure 5, the percentage reductions of synchronizations are significant in all cases and actually grow very slowly with the number of processors. This is because more nodes can be simulated in parallel when the number of processors increases. As a result, our radio-level speedup technique has more radio sleep time to exploit at a given time. Although the reduction numbers are very close with respect to the number of processors, simulation speeds increase significantly with the number of processors in Figure 6 which shows the speed of simulating with and without the radio-level speedup technique in PolarLite using different number of processors. The reason is that per-synchronization overheads increase with the number of processors due to high inter-processor communication overheads.

As shown in Figure 6, using the radio-level speedup technique increases simulation speeds significantly (up to 111%) in PolarLite. Comparing with Figure 4, we observe that PolarLite alone without any speedup techniques is faster than Avrora in these experiments. This is because our distributed synchronization algorithm (Section 4) can provide more parallelism by allowing nodes to be synchronized separately according to their own lookahead times. Avrora on the other hand synchronizes all nodes together at a fixed time interval. However, even using the distributed synchronization algorithm, PolarLite alone does not scale well with the number of processors as shown in Figure 6. Using the radio-level speedup technique significantly improves scalability.

5.1.2 *Speed and Scalability with respect to network sizes and radio off times*

We also evaluate the radio-level speedup technique over WSNs of different sizes and radio sleep durations (radio-level duty cycling modes). Similar to the setups in Section 5.1.1, nodes in these experiments are within direct communication range and only one node is configured as the sender. The sender transmits a packet every 10 seconds to the rest of receiver nodes. Figures 7, 8 and 9 show the results of simulating with or without the radio-level speedup technique in PolarLite using all 8 processors.

Figure 7 shows significant percentage reductions of synchronizations using the radio-level speedup technique. There are no reductions when the radio is constantly on because the radio-level speedup technique works by exploiting radio off time. Figure 7 also shows that the reduction percentages scale with radio off durations since larger durations bring more radio off time. While the reduction percentages are about the same for all network sizes at a given radio off duration, the percentage increases of simulation speed actually grow with network sizes according to Figure 7. This is because in a network where all nodes are in direct communication range, the total number of synchronizations in a distributed sim-

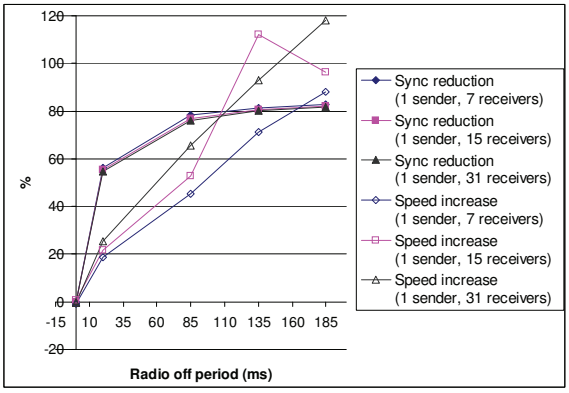

Figure 7: Percentage reductions of synchronizations and percentage increases of simulation speed using the radio-level speedup technique in simulating WSNs of different sizes and radio off times in PolarLite (1 packet/10 seconds, 8 processors)

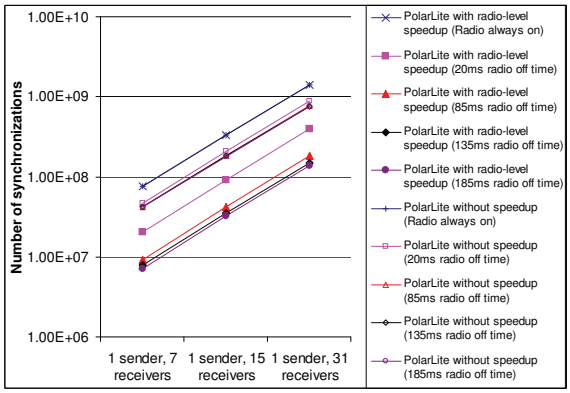

Figure 8: Total number of synchronizations in simulating WSNs of different sizes and radio off times with and without the radio-level speedup technique in PolarLite (1 packet/10 seconds, 8 processors)

ulation is in the order of $N * (N - 1)$ where N is the network size. Therefore, the total number of reductions in the experiments grows with network sizes. This can be seen clearly in Figure 8 which shows the total number of synchronizations in logarithmic scale.

We can also see from Figure 7 that the percentage increases of simulation speed scale well with radio off durations. The case of simulating 16 nodes with 135ms radio off duration appears to be an outlier, showing a much higher increase in simulation speed than normal. The figure also shows that in this case, the reduction in synchronizations is not unusual to cause a higher simulation speedup. We find that this outlier point is actually caused by a slow simulation speed in PolarLite without using the radio-level speedup technique. In fact, when increasing the sleep duration from 85ms to 135ms, the simulation speed actually decreases from 37.45MHz to 37.10MHz. So, when we apply radio-level speedup to this case, the relative increase becomes larger than ordinary. This shows that the proposed radio-level speedup is effective even when the baseline simulation in PolarLite can not benefit from the increased radio off time.

Finally, we evaluate the scalability of the radio-level speedup technique over larger WSNs under a transmission rate of 1 packet/10 seconds and a radio-level duty cycling mode of 3, using all 8 processors. The results are shown in Figure 9. We can see that the radio-level speedup technique increases simulation speed in large WSNs as well. It provides a 197% increase of simulation speed when simulating 256 nodes in PolarLite. That is an additional 106% improvement over the 91% speed increase in simulating 32 nodes under the same setup in Figure 7. In other words, although simulation

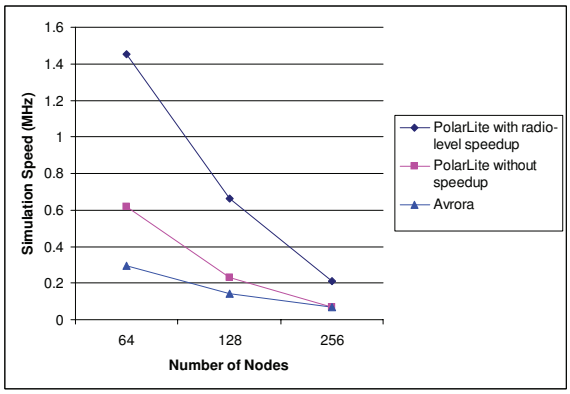

Figure 9: Speed of simulating large WSNs (1 packet/10 seconds, 8 processors, mode 3)

speeds decrease with network sizes due to the limited computational power of our server, the percentage increases of simulation speed using the radio-level speedup technique still grow with network sizes.

5.2 Performance of MAC-level Technique

The performance of our MAC-level speedup technique depends on how busy wireless channels are and how often sensor nodes transmit around the same time. Instead of evaluating with a large number of scenarios, we study the maximum speedup that can be achieved in simulating a WSN with the MAC-level speedup technique. For experiments in this section, we enable CountSend to send as fast as possible by modifying CountSend such that it sends out a new packet as soon as it is notified by the MAC that the previous packet is sent. We also disable the radio-level duty cycling for both CountSend and CountReceive.

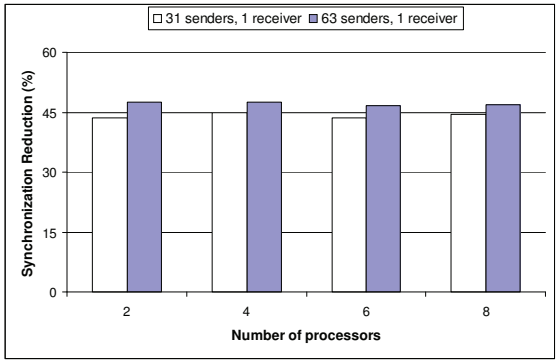

Figure 10: Percentage reductions of synchronizations with MAC-level speedup on WSNs of different sizes in PolarLite (No duty cycling)

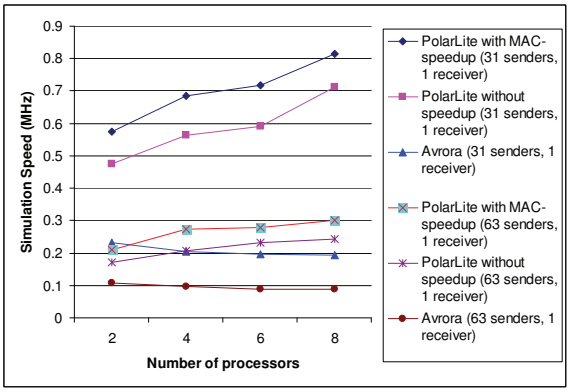

Figure 11: Speed of simulating 2 WSNs with Avrora, PolarLite and PolarLite+MAC-speedup (No duty cycling)

We simulate two WSNs that have 1 receiver and 31 or 63 senders using Avrora, PolarLite without speedups and PolarLite with the MAC-level speedup. Unless explicitly specified, the default backoff calculation functions in TinyOS 1.1 are used for the senders. The results are shown in Figures 10, 11 and 12.

5.2.1 *Speed and Scalability with respect to the number of processors and backoff times*

We can see from Figure 10 that the MAC-level speedup technique reduces synchronizations by about 44% to 47% in PolarLite. As a result, it brings a speedup of 14% to 31% (96% to 323% compared to Avrora) using the default backoff calculation functions of TinyOS 1.1 as shown in Figure 11. However, the default backoff windows are not large enough for our experiments since we observe a significant amount of colliding transmissions causing dropped packets. This limits the performance of our MAC-level speedup technique as nodes may transmit at the same time without backoffs. To further investigate this, we perform the same experiments by doubling the sizes of the default backoff windows and the results are shown in Figure 12. We can see that our MAC-level speedup technique brings more significant increases of simulation speed with larger backoff windows. We can also see that as the number of nodes increases, the speeds of simulating with the MAC-level speedup technique drop faster than with Avrora. This is because given the small backoff window sizes, the number of colliding transmissions increases quickly with the network size in these setups where nodes transmit as fast as possible.

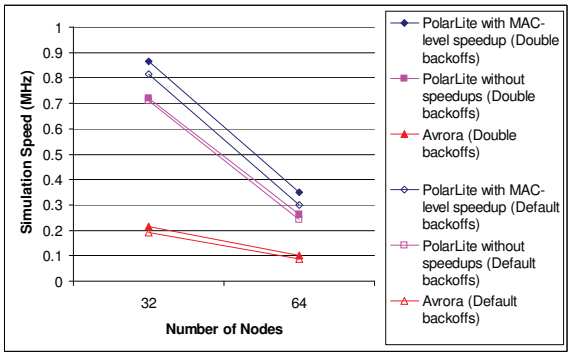

Figure 12: Speed of simulating with MAC-level speedup on WSNs using default and double sized backoff windows (No duty cycling, 8 processors)

5.2.2 *Speed and Scalability with respect to network sizes*

Figure 10 also shows that the percentage reductions of synchronizations using the MAC-level speedup technique increase with network sizes in PolarLite. As explained in Section 5.1.2, the total number of synchronizations in these experiments is in the order of the square of the network size. Therefore, the total number of reductions is very significant when the network size doubles from 32 to 64. We can see in Figure 11 that the percentage increases of simulation speed using the MAC-level speedup technique scale with network sizes even with the default backoff windows. We notice the unusually low increase of speed in simulating with 6 processors. Since the percentage reductions of synchronizations are consistent according to Figure 10, we believe this is caused by the asymmetrical use of all 4 cores of 1 CPU and 2 cores of another CPU in our server.

5.3 Performance with Both Techniques

We evaluate the combined performance of our speedup techniques with a real world scenario. In this scenario, we simulate a WSN service that floods data to every

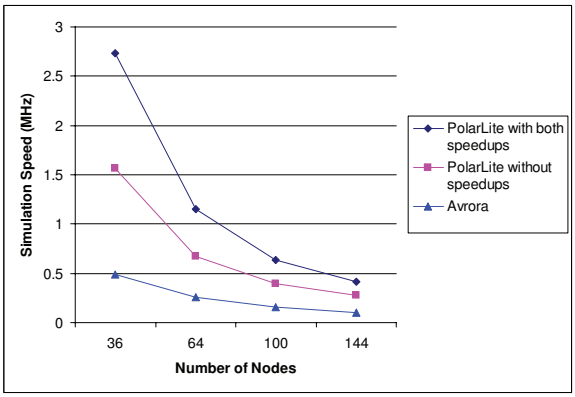

Figure 13: Speed of simulating with Avrora, PolarLite without speedups and PolarLite with both speedup techniques (8 processors)

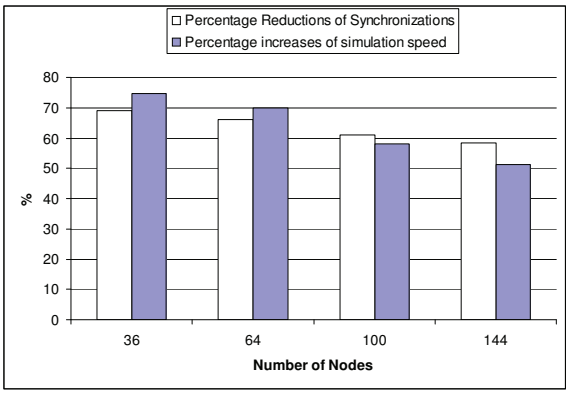

Figure 14: Percentage increases of simulation speed and percentage reductions of synchronizations with both speedup techniques in PolarLite (8 processors)

node in a WSN. This service works by having every node in a WSN relay, by broadcasting, messages it receives. To avoid sending duplicate messages, a node only relays messages with IDs greater than the largest IDs of the messages it has already sent. For experiments in this section, we modify CountReceive to relay messages the way we just described.

In our experiments, we simulate WSNs that have nodes laid 3 meters apart on square grids of different sizes. For each of the WSNs, a corner node is configured as the sender and the rest of nodes are configured as relaying nodes running the modified CountReceive program. The sender transmits a new packet every 20 seconds with an increasing ID. The radio-level duty cycling modes of all nodes are set to 4 (Table 1) and the backoff windows are doubled from TinyOS 1.1 defaults. The transmit range of all nodes is set to 19 meters. We conduct the experiments with all 8 processors and the results are shown in Figures 13 and 14.

As shown in Figures 13 and 14, PolarLite running both speedup techniques is significantly faster and provides a speedup of 51% to 75% over PolarLite alone and 289% to 462% compared to Avrora. We can also see from Figure 14 that the speedup techniques reduce synchronizations significantly by 58% to 70%. However, we observe that the reduction percentages decrease with increasing network sizes. This is caused by our simple flooding protocol. As the network size increases, the number of relaying messages grows as well. Although a node does not transmit the same message twice, it can be forced to receive the same message multiple times from different neighboring nodes. In other words, a transmitting node can keep all nodes within its communication range from turning off their radios. This significantly reduces the sleep time of the nodes and lowers the performance of our radio-level speedup technique. However, even under this setup, our speedup techniques still provide significant increases of simulation speed. This is because our MAC-level speedup technique benefits from an increasing number of backoffs in the larger networks. In practice, more advanced protocols are usually used to reduce the number of unnecessary relaying messages. Therefore, we expect significant better performance with the speedup techniques in those cases.

6. CONCLUSION AND FUTURE WORK

We have described two speedup techniques that significantly improve the speed and scalability of distributed sensor network simulators by reducing the number of sensor node synchronizations during simulations. We implemented the techniques in PolarLite, a cycle accurate distributed simulation framework based on Avrora. The significant improvements of simulation performance on a multi-processor computer in our experiments suggest even greater benefits in applying our techniques to distributed simulations over a network of computers because of their large overheads in sending synchronization messages across computers during simulations.

We have also developed a general probing mechanism that can expose the internal states of any sensor network applications during simulations. By knowing the internal states during simulations, one can exploit any application specific characteristics for the increase of lookahead time and as a result, improve simulation speed and scalability.

As future work, we plan to use the probing mechanism to exploit scheduled transmission slots in TDMA type MACs such as S-MAC [22]. With this type of MAC, a node can only send data in scheduled transmission slots. By knowing the time of the slots during simulations, we can identify the periods that a node does not trans-

mit and therefore increase the lookahead time. We also plan to merge our implementation with the latest development branch of Avrora. This would make it possible to simulate TinyOS 2.0 [1] based applications with our speedup techniques. Although the techniques are currently implemented in a cycle accurate simulator, they can also be applied to other simulators like TOSSIM to make them more scalable over multiple processors.

7. REFERENCES

[1] Tinyos 2.0. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 320–320, New York, NY, USA, 2005. ACM.

[2] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[3] G. Chelius, A. Fraboulet, and E. Fleury. Worldsens: a fast and accurate development framework for sensor network applications. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 222–226, New York, NY, USA, 2007. ACM.

[4] Crossbow. *MICA2 Datasheet*, 2008.

[5] D. Filo, D. C. Ku, and G. D. Micheli. Optimizing the control-unit through the resynchronization of operations. *Integr. VLSI J.*, 13(3):231–258, 1992.

[6] R. M. Fujimoto. Parallel and distributed simulation. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 122–131, New York, NY, USA, 1999. ACM.

[7] T. Henderson. *NS-3 Overview*, 2008.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[9] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[10] Z. Jin and R. Gupta. Improved distributed simulation of sensor networks based on sensor node sleep time. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 204–218, 2008.

[11] O. Landsiedel, H. Alizai, and K. Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *IPSN '08: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (IPSN 2008)*, pages 344–355, Washington, DC, USA, 2008. IEEE Computer Society.

[12] P. Levis. *TinyOS Programming*, June 2006.

[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 international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[14] J. Liu and D. M. Nicol. Lookahead revisited in wireless network simulations. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 79–88, Washington, DC, USA, 2002. IEEE Computer Society.

[15] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM.

[16] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *the Fourth International Conference on Information Processing in Sensor Networks*, 2005.

[17] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. Baras. Atemu: a fine-grained sensor network simulator. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 145–152, 4-7 Oct. 2004.

[18] 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, New York, NY, USA, 2004. ACM.

[19] TinyOS-Alliance. Tinyos 1.1.15.

[20] 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*, page 67, Piscataway, NJ, USA, 2005. IEEE Press.

[21] 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*, pages 24–34, New York, NY, USA, 2007. ACM Press.

[22] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Infocom '02*, pages 1567–1576, New York, NY, 2002.