LazySync: A New Synchronization Scheme for Distributed Simulation of Sensor Networks

Zhong-Yi Jin and Rajesh Gupta

Department of Computer Science and Engineering
University of California, San Diego
{zhjin, rgupta}@cs.ucsd.edu

Abstract. To meet the demands for high simulation fidelity and speed, parallel and distributed simulation techniques are widely used in building 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 LazySync, a novel conservative synchronization scheme that can significantly reduce such overheads by minimizing the number of clock synchronizations during simulations. We implement and evaluate this scheme in a cycle accurate distributed simulation framework that we developed based on Avrora, a popular parallel sensor network simulator. In our experiments, the scheme achieves a speedup of 4% to 53% in simulating single-hop sensor networks with 8 to 256 nodes and 4% to 118% in simulating multi-hop sensor networks with 16 to 256 nodes. The experiments also demonstrate that the speedups can be significantly larger as the scheme scales with both the number of packet transmissions and sensor network size.

1 Introduction

Accurate simulation is critical to the design, implementation and evaluation of wireless sensor networks (WSNs). Numerous WSN simulators have been developed based on event driven simulation techniques [1] and the fidelities of WSN simulators are rapidly increasing with the use of high fidelity simulation models [2–5]. In event driven simulations, fidelity represents the bit and temporal accuracy of events and actions. Due to the need for processing a large number of events, high simulation fidelity often leads to slow simulation speed [6] which is defined as the ratio of *simulation time* to *wallclock time*. Simulation time is the virtual clock time in the simulated models [7] 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. Typically, real time speed is required to use simulations for interactive tasks such as debugging and testing.

To meet the demands for high simulation fidelity and speed, most of latest WSN simulators are based on parallel and distributed simulation techniques [8–11]. WSN simulators can be broadly divided into two types: sequential simulators

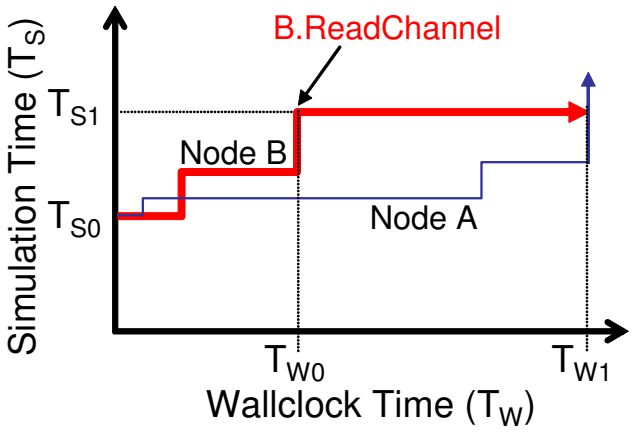

Fig. 1. The progress of simulating in parallel a wireless sensor network with two nodes that are in direct communication range of each other on 2 processors.

and parallel/distributed simulators. Sequential simulators simulate all the sensor nodes of a WSN in sequence on a single processor and therefore cannot benefit from running on a multi-core processor or on multiple processors. Parallel and distributed simulators seek to improve simulation speed by simulating different sensor nodes on different cores or processors in parallel.

A problem with existing distributed simulation techniques is the large overheads in synchronizing sensor nodes during simulations [10, 6]. When sensor nodes are simulated in parallel, their simulation speeds can vary due to the differences in simulated nodes, such as different sensor node programs, sensor node inputs and values of random variables, as well as the differences in simulation environments, such as different processor speeds and operating system scheduling policies. Because of this, simulated sensor nodes need to synchronize with each other frequently to preserve causality of events and ensure correct simulation results [9, 10, 6]. For example, as shown in Fig. 1, two nodes in direct communication range of each other are simulated in parallel on two processors. After T_{W0} seconds of simulation, Node B is simulated faster than Node A as indicated by the fact that the simulation time of Node B (T_{S1}) is greater than the simulation time of Node A at T_{W0}. At T_{S1}, Node B is supposed to read the wireless channel and see if there is an incoming transmission from Node A. However, after reaching T_{S1} at T_{W0}, Node B cannot advance any further because at T_{W0} Node B does not know whether Node A is going to transmit at T_{S1} or not. In other words, Node B cannot be simulated any further than T_{S1} until Node A reaches T_{S1}. There are two general approaches to handle cases like this: conservative [1] or optimistic [12]. The conservative approach works by ensuring no causality relationships among events are violated while the optimistic approach works by detecting and correcting any violations of causal relationships [6]. To the best of our knowledge, almost all distributed WSN sim-

ulators are based on the conservative approach as it is simpler to implement and has a lower memory footprint. With the conservative approach, simulated sensor nodes need to synchronize their clocks and coordinate their simulation orders frequently. These tasks introduce communication overhead and management overhead to distributed simulations respectively [13].

As described in [10, 6], the performance gains of existing distributed WSN simulators are often compromised by the rising overheads due to inter-node synchronizations. In this paper, we propose *LazySync*, a novel conservative synchronization scheme that can significantly reduce the number of clock synchronizations in parallel and distributed simulations of WSNs. We validate our approaches by their implementations in PolarLite, a cycle accurate distributed simulation framework that builds upon Avrora and serves as the underlying simulation engine [6].

We discuss related work in Sect. 2. The LazySync scheme is presented in Sect. 3 and its implementations are described in Sect. 4. In Sect. 5 we present the results of our experiments followed by the conclusion and future work in Sect. 6.

2 Related Work

Prior work on improving the speed and scalability of WSN simulators can be divided into two categories. The first category of work can be applied to both sequential and distributed simulators. It focuses on reducing the computational demands of individual simulation model without significantly lowering fidelity. For example, since emulating a sensor node processor is computationally expensive [4], TimeTOSSIM [5] 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 non-cycle-accurate TOSSIM [2] which TimeTOSSIM is based on. Our work can make this type of effort scalable on multiple processors/cores.

The second category of work focuses on distributed simulators only. There is a large body of research on improving the speed and scalability of distributed discrete event driven simulators in general. Among them, use of lookahead time [1] is a commonly used conservative approach [14, 15]. Our previous work [6, 13] follows this direction. In [6], we describe a technique that monitors the duty cycling of sensor nodes and uses the detected 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 the time when both the processor and radio of a sensor node are off for speedup because it is not possible to predict the exact radio wakeup time when the processor is running. In [13], we develop new techniques to address the limitation of [6]. By exploiting the radio wakeup latency and MAC backoff time, the new

techniques are effective even when the processor or radio of a node is active. Our LazySync approach is different from the lookahead approach as we do not explicitly exploit lookahead time.

As an alternative to the lookahead approach, the performance of distributed simulators can also be improved by reducing the overheads in performing synchronizations. DiSenS [10] 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. However, this technique only works well if most of the nodes are not within direct communication range as described in the paper. LazySync is very different from DiSenS in the sense that it works by reducing the total number of clock synchronizations in a simulation rather than by reducing the overhead of performing an individual clock synchronization. In addition, LazySync is particularly effective in improving the performance of simulating dense WSNs with a large amount of communication traffic.

LazySync is based on similar ideas as lazy evaluation which have been used in earlier work on very different problems from architectural designs to programming languages [16]. Though conceptually similar, we show how the notion of lazy evaluation can be applied in sensor networks for improved performance of distributed simulations.

3 Lazy Synchronization Scheme

In distributed simulations, each sensor node is commonly simulated in a separate thread or process. To maximize parallelism, a running node should try to prevent other nodes from waiting by communicating its simulation progress to those nodes as early as possible (AEAP). If a node has to wait for other nodes due to variations in simulation speeds, the thread/process simulating the waiting node should be suspended so the released physical resources can be used to simulate some other non-waiting nodes[1]. For maximum parallelism, suspended nodes need to be revived AEAP once the conditions that the nodes wait for are met. For example, Node A in Fig. 1 should synchronize with Node B immediately after it advances past T_{S1} to resume the simulation of Node B.

The AEAP synchronization scheme is adopted by most existing distributed WSN simulators [9, 10, 6]. It is commonly implemented [9, 6] by periodically sending the simulation time of every non-waiting node to all its neighboring nodes, which are nodes that are within its direct communication range. Ideally, the clock synchronization period should be as short as possible for maximum parallelism. However, due to the overheads in performing clock synchronizations [6], the synchronization period is commonly set to be the minimal lookahead time, which is the smallest possible lookahead time in the simulation. As mentioned, lookahead time is the maximum amount of simulation time that a simulated sensor node can advance freely without synchronizing with any other simulated

[1] For synchronization purposes, a non-waiting node refers to a node that is not waiting for any simulation events. It may still be ready, active or inactive in a given simulator.

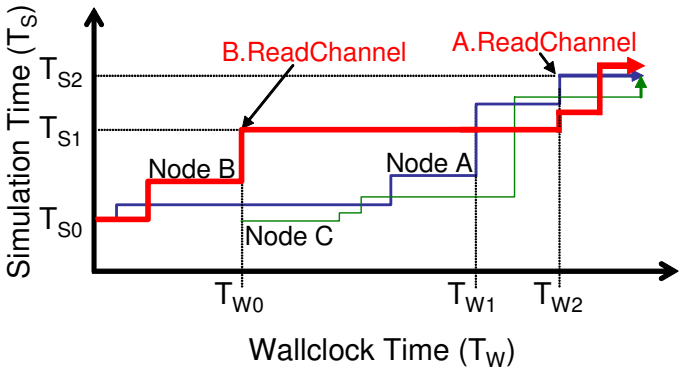

Fig. 2. The progress of simulating in parallel a wireless sensor network with three nodes that are in direct communication range of each other on 2 processors.

sensor nodes [7]. For example, in the case of simulating Mica2 nodes [17], the minimal lookahead time is the lookahead time of nodes with radios in the listening mode. It is equal to the amount of time to receive one byte over Mica2's CC1000 radio [6] and is equivalent to 3072 clock cycles of the 7.32728MHz AVR microcontroller in Mica2. Therefore, when simulating a network of Mica2 nodes, every non-waiting node needs to send its simulation time to all its neighboring nodes every 3072 clock cycles. Depending on simulator implementations, once the simulation time is received by a neighboring node, some mechanisms will be triggered to save the received time and compute the earliest input time (EIT) [1]. EIT represents the safe simulation time that the neighboring node can be simulated to. If the neighboring node happens to be waiting, then it will also be revived if its EIT is not less than the wait time. To be revived AEAP, a waiting node commonly sends its waiting time to the nodes that it depends on before entering into the suspended state. By doing so, the depending nodes can send their simulation time to the waiting node immediately after they advance past the waiting time.

3.1 Limitations of AEAP Synchronization Scheme

While the AEAP synchronization scheme is sound in principle, its effectiveness is based on the assumption that there is always a free processor available to simulate every revived node. However, this is generally not the case in practice as the number of nodes under a simulation is usually a lot larger than the number of processors used to run the simulation. As a result, the AEAP synchronization scheme may slow simulations down in many simulation scenarios by introducing unnecessary clock synchronizations. For example, Fig. 2 shows the progress of simulating in parallel 3 nodes that are in direct communication range of each other on 2 processors. In the simulation, Node A and B are simulated first on the two available processors and Node B reaches T_{S1} at T_{W0}. Similar to the case in

Fig. 1, Node B has to wait at T_{S1} until the simulation time of both Node A and Node C reach T_{S1}. However, unlike the case in Fig. 1, while Node B is waiting, the simulation of Node C begins and both processors are kept busy. With the AEAP synchronization scheme, Node A should send its simulation time to Node B at T_{W1} so the simulation of Node B can be resumed. However, since both processors are busy simulating Node A and C at T_{W1}, reviving Node B at T_{W1} does not increase simulation performance at all. In fact, this may actually slow the simulation down due to the overhead in performing this unnecessary clock synchronization. For example, instead of synchronizing with Node B at T_{W1}, Node A can delay the synchronization until a free processor becomes available at T_{W2} when Node A needs to read the wireless channel and waits for Node B and C. By delaying the clock synchronization to T_{S2} at T_{W2}, Node A effectively reduces one clock synchronization.

Another area that existing AEAP synchronization algorithms [9, 10, 6] fail to exploit for synchronization reductions is the simulation time gaps among neighboring nodes. Due to the lack of processors to simulate all non-waiting nodes simultaneously, the potential simulation time gaps of different nodes can be quite large during a simulation. For example, an actively transmitting node cannot hear transmissions from other nodes and therefore can be simulated without waiting until it stops transmitting and reads the wireless channel. Given such time gaps, a node receiving the simulation time of a node in the future can compare the future node's time with its own simulation time and calculate potential dependencies between the two nodes in the future. Consequently, the node falling behind can skip clock synchronizations if there are no dependencies between the two nodes. For instance, as shown in Fig. 2, once Node A sends a clock synchronization message to Node B at T_{S2}, Node B knows implicitly that Node A does not depend on it before T_{S2} and therefore does not need to synchronize its clock with Node A until then. In other words, Node B no longer needs to send its simulation time to Node A every minimal lookahead time before T_{S2} as it does with the AEAP synchronization algorithms. By delaying clock synchronizations, we can fully extend the time gaps and as a result create more opportunities for nodes falling behind to act upon and reduce clock synchronizations. We will discuss this in detail in the following section.

3.2 Lazy Synchronization Algorithm

To address the performance issue of the AEAP synchronization scheme, we propose a novel conservative synchronization scheme: LazySync. The key idea of the LazySync scheme is to delay a synchronization even when it should be done according to conservative simulations. It is opposite of opportunistic synchronization in that the simulator seeks to avoid synchronization until it is essential and it is able to do it given simulation resource constraints. Together, we show that the concept of lazy evaluation can be extended to specifically benefit from the operational characteristics of sensor networks. By procrastinating synchronizations, delayed clock synchronizations may be safely discarded or substituted

by newer clock synchronizations in simulating WSNs. As a result, the total number of clock synchronizations in a simulation can be reduced.

Note that if free processors are available, our LazySync scheme must perform synchronizations AEAP so potential nodes can be revived to use the available physical resources. To make this possible, we track the number of non-waiting nodes and only procrastinate synchronizations when the number is below a threshold. Ideally, the threshold should be set to be the number of processors used to run the simulation in order to maximize clock synchronization reduction and processor usage. However, considering the frequency of checking the number of non-waiting nodes and the overheads in reviving waiting nodes and performing scheduling, the threshold should be set to a number slightly larger than that in practice. Tracking the number of non-waiting nodes on a computer should incur very little overhead since that is already done by the underlying thread/process library or OS as part of their scheduling functions. For distributed simulations on multiple computers, the number of non-waiting nodes on each computer can be exchanged as part of clock synchronization messages sent between computers. If a computer does not receive any clock synchronization messages from another computer for a predetermined period of time, the nodes on the first computer can revert back to the AEAP scheme.

Our proposed LazySync algorithm is presented in Algorithm 1. As shown in Algorithm 1, we design the LazySync algorithm to work differently on nodes in different states because nodes may have different synchronization needs. In a simulation, a sensor node can be in one of two states, the independent state and the dependent state.

A node is in the independent state if its radio is not in receiving mode. This happens when the radio is off, in transmission mode or in any one of the initialization and transition states. Since a node in the independent state (independent node) does not take inputs from any other nodes, it can be simulated without waiting for any other nodes until the state changes. However, if free processors are available, an independent node still needs to synchronize with neighboring nodes so that the nodes depending on the outputs of the independent node can be simulated. In the LazySync algorithm, an independent node checks the number of non-waiting nodes every minimal lookahead time and only sends a clock synchronization message to its neighboring nodes if the number of non-waiting nodes is below a threshold.

A node is in the dependent state if its radio is in receiving mode. Since any node in direct communication range of a dependent node (a node in the dependent state) can potentially transmit, a dependent node needs to meet Condition 1 before actually reading the wireless channel to ensure correct simulation results. In other words, a dependent node needs to evaluate Condition 1 to determine if it can read the wireless channel and continue the simulation or has to wait for some neighboring nodes to catch up for their potential outputs. Since a dependent node has its radio in receiving mode, it needs to read the wireless channel at least once every minimal lookahead time (ΔT) which is the lookahead time of a node in the dependent state. Therefore, Condition 1 is evaluated at

Algorithm 1 Lazy Synchronization Algorithm

Require: *syncThreshold* /*sync threshold*/
Require: ΔT /*minimal lookahead time, the lookahead time of a node in the dependent state*/
1: set *timer* to fire at every ΔT
2: $syncTime \Leftarrow 0$ /*the time a sync condition is verified*/
3: **while** simulation not end **do**
4: simulate the next *instruction*
5: **if** in independent state **then**
6: **if** $timer.fired$ **then**
7: $syncTime \Leftarrow$ current sim time
8: **if** $numLiveNode < syncThreshold$ **then**
9: send current sim time to all neighboring nodes not ΔT ahead
10: **else if** in dependent state **then**
11: **if** *instruction* needs to read the wireless channel **then**
12: $syncTime \Leftarrow$ current sim time
13: **if** $((Condition\ 1) == true)$ **then**
14: **if** $numLiveNode < syncThreshold$ **then**
15: send current sim time to all neighboring nodes not ΔT ahead
16: **else**
17: send current sim time to all neighboring nodes not ΔT ahead
18: wait until $((Condition\ 1) == true)$
19: read the wireless channel
20: **if** *syncTime* - (current sim time) $> \Delta T$ **then**
21: $syncTime \Leftarrow$ current sim time
22: **if** $numLiveNode < syncThreshold$ **then**
23: send current sim time to all neighboring nodes not ΔT ahead

Condition 1 *If a node N_i reads wireless channel C_k at simulation time T_{SN_i}, then for all nodes N_s that are in direct communication range of N_i, $(T_{SN_s} + \Delta T) \geq T_{SN_i}$, where T_{SN_s} is the simulation time of N_s and ΔT is the lookahead time of N_i which is in the dependent state.*

least once every ΔT. In the LazySync algorithm, a dependent node only performs clock synchronizations under two circumstances. The first circumstance happens when Condition 1 is evaluated to be false and as a result, a dependent node has to wait for neighboring nodes. To prevent deadlocks, a synchronization has to be performed in this case before suspending the node, regardless of the number of available processors. A deadlock occurs when nodes wait for each other at the same simulation time. For instance, it happens when nodes within direct communication range read the wireless channel at the same simulation time. The second circumstance occurs when Condition 1 is evaluated to be true so a dependent node can go ahead to read the wireless channel. If the number of non-waiting nodes is below a threshold at this point, a clock synchronization is required to revive some nodes to use the available processors. Note that the block of code from line 20 to 23 in Algorithm 1 is just a safety mechanism to

guard against the cases that a node does not stay in any of the two states long enough to check for synchronization conditions.

It is important to note that a dependent node may only perform clock synchronizations at the times it reads the wireless channel. This is very different from the case in a typical AEAP synchronization algorithm. A node in an AEAP synchronization algorithm may perform clock synchronizations at any time according to the waiting times of other nodes. The decision to limit dependent nodes to perform clock synchronizations at channel read time only is based on the assumption that there are no free processors available to simulate any other nodes until an actively running dependent node gives up its processor due to waiting. By procrastinating clock synchronizations to channel read time, we can eliminate all intermediate synchronizations that need to be performed otherwise in AEAP synchronization algorithms, as described in Sect. 3.1.

With the LazySync algorithm described above, a node can be simulated for a long period of time without sending its simulation time to neighboring nodes. As discussed in Sect. 3.1, the extended simulation time gaps of neighboring nodes can be exploited effectively to reduce clock synchronizations. According to Condition 1, a dependent node N_i can read the wireless channel only if the simulation time of all neighboring nodes are equal to or greater than the simulation time of N_i minus ΔT. If the simulation time of a neighboring N_s is more than ΔT ahead of the simulation time of N_i, there are no needs for N_i to send its simulation time to N_s until the simulation time of N_i is greater than $T_{SN_s} - \Delta T$. The same also applies if an independent node receives a simulation time that is more than ΔT ahead. Based on these, the LazySync algorithm uses a filter to remove unnecessary clock synchronizations.

It is important to see that our LazySync algorithm still follows the principles of conservative synchronization algorithms [1, 7, 8] to not violate any causality during simulations. We only delay and discard unnecessary clock synchronizations to improve the performance of distributed simulations of WSNs. Due to space limits, a formal correctness proof of the LazySync algorithm is not given here.

4 Implementation

The proposed LazySync scheme is implemented in PolarLite, a distributed simulation framework that we developed based on Avrora [6]. Our simulation framework provides the same level of cycle accurate simulations as Avrora but uses a distributed synchronization engine instead of Avrora's centralized one.

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. However, we cannot identify any Java APIs that allow us to check the number of suspended/blocked threads in a running program. As an alternative, we track that using an atomic variable. The *syncThreshold* in Algorithm 1 is configurable via a command line argument.

To implement the LazySync algorithm, we need to detect the state that a node is in. 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. We verify the correctness of our implementation by running the same simulations with and without the LazySync algorithm using the same random seeds.

5 Evaluation

To evaluate the performance of the LazySync scheme, we simulate some typical WSNs with PolarLite using both the AEAP synchronization algorithm from [6] and the LazySync algorithm from Sect. 3.2. The performance results are compared according to three criteria:

- $Speed_{avg}$: The average simulation speed.
- $Sync_{avg}$: The average number of clock synchronizations per node.
- $Wait_{avg}$: The average number of waits per node.

$Speed_{avg}$ is calculated using Equation (1) based on the definition specified in Sect. 1. Note that the numerator of Equation (1) is the total simulation time in units of clock cycles. $Sync_{avg}$ is equal to the total number of clock synchronizations in a simulation divided by the total number of nodes in the simulation. Similarly, $Wait_{avg}$ is equal to the total number of times that nodes are suspended in a simulation due to waiting divided by the total number of nodes in the simulation.

$$Speed_{avg} = \frac{total\ number\ of\ clock\ cycles\ executed\ by\ the\ sensor\ nodes}{(simulation\ execution\ time) \times (number\ of\ sensor\ nodes)} \quad (1)$$

The WSNs we simulate in this section consist of only Mica2 nodes [17] running either CountSend (sender) or CountReceive (receiver) programs. Both programs are from the TinyOS 1.1 distribution and are similar to the programs used by other WSN simulators in evaluating their performance [2, 9, 10]. For example, CountSend broadcasts a continuously increasing counter repeatedly at a fixed interval. If the interval is set to 250ms, it behaves exactly the same as CntToRfm which is used in [2, 9, 10] for performance evaluations. CountReceive listens for messages sent by CountSend and displays the received values on LEDS.

All simulation experiments are conducted on an SMP server running Linux 2.6.24. 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. In the simulations, 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 120 seconds of simulation time and for each experiment we take the average of three runs as the results. The synchronization threshold (Algorithm 1) of the LazySync algorithm is set to 9 (the number of processors plus one) for all experiments.

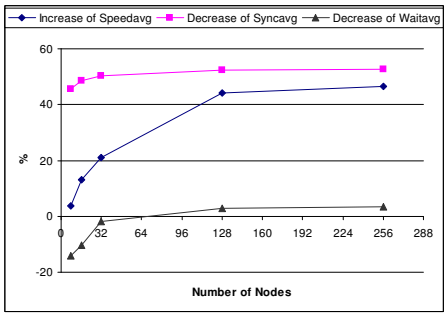

Fig. 3. Performance improvements of the LazySync scheme over the AEAP scheme in simulating one-hop WSNs. Senders transmit at a 250ms interval.

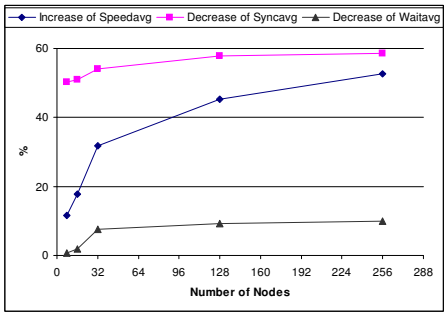

Fig. 4. Performance improvements of the LazySync scheme over the AEAP scheme in simulating one-hop WSNs. Senders transmit as fast as possible.

5.1 Performance in One-hop WSNs

In this section, we evaluate the performance of the LazySync scheme in simulating one-hop WSNs of various sizes. One-hop WSNs are sensor networks with all their nodes in direct communication range. All the one-hop WSNs that we simulate in this section have 50% of the nodes running CountSend and 50% of the nodes running CountReceive.

In the first experiment, we modify CountSend so that all senders transmit at a fixed interval of 250ms. Five WSNs with 8, 16, 32, 128 and 256 nodes are simulated and Fig. 3 shows the percentage improvements of the LazySync scheme compared to the AEAP scheme. As shown in Fig. 3, the LazySync scheme reduces $Sync_{avg}$ in all cases and the percentage reductions grow slowly with network sizes. It is important to see that the total number of clock synchronizations in a distributed simulation of a one-hop WSN is on the order of $N * (N - 1)$ where N is the network size [6]. So, although the percentage reductions of $Sync_{avg}$ increase slowly with network sizes in Fig. 3, the actual values of $Sync_{avg}$ decrease significantly with network sizes.

The significant percentage reduction of $Sync_{avg}$ in simulating 8 nodes with 8 processors is due to the time gap based filter and the fact that the synchronization threshold is only checked every ΔT or at channel read time. Since the threshold is not monitored at a finer time granularity, a processor may be left idle for a maximum of the amount of wallclock time to simulate a node for ΔT according to Algorithm 1. As a result, we can see in Fig. 3 that there are moderate increases of $Wait_{avg}$ when simulating small WSNs with 8 and 16 nodes. However, as the WSN size increases, the percentage reduction of $Wait_{avg}$ increases because processors are more likely to be kept busy by the extra nodes. In fact, the LazySync scheme performs better in terms of percentage reductions of $Wait_{avg}$ when simulating 128 and 256 nodes, as shown in Fig. 3. We believe this is because more CPU cycles become available for real simulations after significant reductions in the number of clock synchronizations. For the same reason,

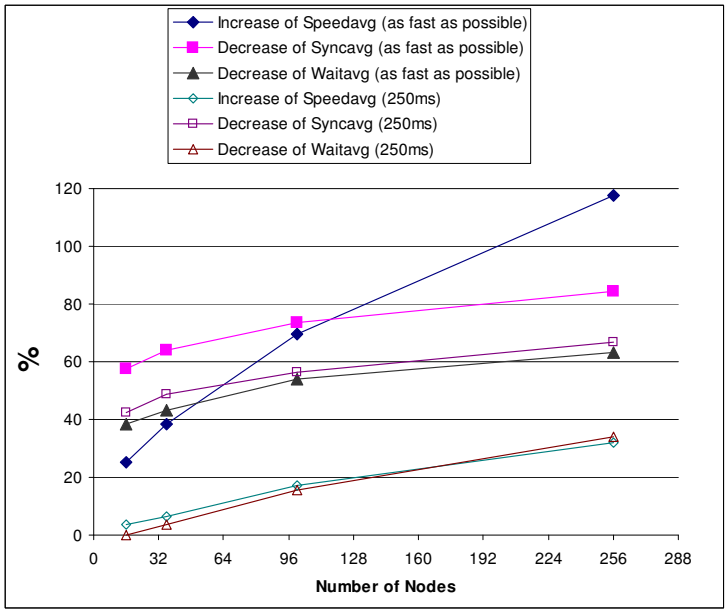

Fig. 5. Performance improvements of the LazySync scheme over the AEAP scheme in simulating multi-hop WSNs.

despite the increases of $Wait_{avg}$ in simulating small WSNs, we see increases of $Speed_{avg}$ in all cases, ranging from 4% to 46%.

Our second experiment is designed to evaluate the LazySync scheme in busy WSNs that have heavy communication traffic. It is based on the same setup as the first experiment except all senders transmit as fast as possible. As shown in Fig. 4, the LazySync scheme provides more significant percentage reductions of $Wait_{avg}$ in busier networks. This is because a busier network has more transmissions and consequently more independent states. The increased number of independent states in a busier network provides more opportunities for the LazySync scheme to exploit. It allows nodes to skip synchronizations and gives the filter larger gaps to exploit. As a result, the LazySync scheme brings a 12% to 53% increase of $Speed_{avg}$ in Fig. 4. We can also see in Fig. 4 that there are no increases of $Wait_{avg}$ in simulating small WSNs as in the first experiment. This is because it takes more CPU cycles to simulate all the communications in a busy network and that keeps the processors busy.

5.2 Performance in Multi-hop WSNs

In this section, we evaluate the performance of the LazySync scheme in simulating multi-hop WSNs of various sizes. Nodes are laid 15 meters apart on square grids of various sizes. Senders and receivers are positioned on the grids in such

a way that nodes of the same types are not adjacent to each other. By setting a maximum transmission range of 20 meters, this setup ensures that only adjacent nodes are within direct communication range of each other. This configuration is very similar to the two dimensional topology in DiSenS [10].

We simulate WSNs with 16, 36, 100 and 256 nodes. For each network size, we simulate both a quiet network with all the senders transmitting at a fixed 250ms interval and a busy network with all the senders transmitting as fast as possible. The results are shown in Fig. 5. We can see that the percentage decreases of $Sync_{avg}$ are more significant in the multi-hop networks than in the one-hop networks. The reason for this is that there are fewer dependencies among nodes in our multi-hop networks than in the one-hop networks, as a result of only having adjacent nodes in communication range in the multi-hop network setup. Having fewer dependencies brings two opportunities to the LazySync scheme. First, a node can be simulated for a longer period of time without waiting. Second, the increased number of non-waiting nodes keeps processors busy. Together, they enable nodes to skip clock synchronizations in LazySync. In addition, the increased simulation time gaps can also be exploited by LazySync to reduce clock synchronizations. As shown in Fig. 5, the percentage reductions of $Sync_{avg}$ are significantly higher in the busy multi-hop networks than in the quiet ones. This demonstrates once again that the LazySync scheme can exploit wireless transmissions in a WSN for synchronization reductions. As a result, we see significant percentage increases of $Speed_{avg}$ in simulating busy multi-hop networks, ranging from 25% to 118%.

6 Conclusion and Future Work

We have presented LazySync, a synchronization scheme that significantly improves the speed and scalability of distributed sensor network simulators by reducing the number of clock synchronizations. We implemented LazySync in PolarLite and evaluated it against an AEAP scheme inside the same simulation framework. The significant improvements of simulation performance on a multiprocessor 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.

As future work, we are planning to combine LazySync with some other performance increasing techniques that we developed in the past [6, 13]. Since these techniques exploit different aspects of WSNs for performance improvements, we believe combining the techniques can further improve the speed and scalability of distributed WSN simulators.

References

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

2. Levis, P., Lee, N., Welsh, M., Culler, D.: Tossim: accurate and scalable simulation of entire tinyos applications. In: SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems, New York, NY, USA, ACM Press (2003) 126–137
3. Shnayder, V., Hempstead, M., rong Chen, B., Allen, G.W., Welsh, M.: Simulating the power consumption of large-scale sensor network applications. In: SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems, New York, NY, USA, ACM (2004) 188–200
4. Polley, J., Blazakis, D., McGee, J., Rusk, D., Baras, J.: 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. (4-7 Oct. 2004) 145–152
5. Landsiedel, O., Alizai, H., Wehrle, K.: 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, Washington, DC, USA, IEEE Computer Society (2008) 344–355
6. Jin, Z., Gupta, R.: Improved distributed simulation of sensor networks based on sensor node sleep time. In: International Conference on Distributed Computing in Sensor Systems (DCOSS). (2008) 204–218
7. Fujimoto, R.M.: Parallel and distributed simulation. In: WSC '99: Proceedings of the 31st conference on Winter simulation, New York, NY, USA, ACM (1999) 122–131
8. Riley, G.F., Ammar, M.H., Fujimoto, R.M., Park, A., Perumalla, K., Xu, D.: A federated approach to distributed network simulation. ACM Trans. Model. Comput. Simul. **14**(2) (2004) 116–148
9. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks, Piscataway, NJ, USA, IEEE Press (2005) 477–482
10. Wen, Y., Wolski, R., Moore, G.: Disens: scalable distributed sensor network simulation. In: PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM Press (2007) 24–34
11. Henderson, T.: NS-3 Overview. (2008)
12. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. **7**(3) (1985) 404–425
13. Jin, Z., Gupta, R.: Improving the speed and scalability of distributed simulations of sensor networks. Technical Report CS2009-0935, UCSD (2009)
14. Filo, D., Ku, D.C., Micheli, G.D.: Optimizing the control-unit through the resynchronization of operations. Integr. VLSI J. **13**(3) (1992) 231–258
15. Liu, J., Nicol, D.M.: Lookahead revisited in wireless network simulations. In: PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation, Washington, DC, USA, IEEE Computer Society (2002) 79–88
16. Hughes, J.: Why functional programming matters. Comput. J. **32**(2) (1989) 98–107
17. Crossbow: MICA2 Datasheet. (2008)