DiSenS: Scalable Distributed Sensor Network Simulation *

Ye Wen
University of California, Santa Barbara
wenye@cs.ucsb.edu

Rich Wolski
University of California, Santa Barbara
rich@cs.ucsb.edu

Gregory Moore
University of California, Santa Barbara
gmoore@umail.ucsb.edu

Abstract

Simulation is widely used for developing, evaluating and analyzing sensor network applications, especially when deploying a large scale sensor network remains expensive and labor intensive. However, due to its computation intensive nature, existent simulation tools have to make trade-offs between fidelity and scalability and thus offer limited capabilities as design and analysis tools. In this paper, we introduce DiSenS (DIstributed SENsor network Simulation) – a highly scalable distributed simulation system for sensor networks. DiSenS does not only faithfully emulates an extensive set of sensor hardware and supports extensible radio/power models, so that sensor network applications can be simulated transparently with high fidelity, but also employs distributed-memory parallel cluster system to attack the complex simulation problem. Combining an efficient distributed synchronization protocol and a sophisticated node partitioning algorithm (based on existent research), DiSenS achieves greater scalability than even many discrete event simulators. On a small to medium size cluster (16-64 nodes), DiSenS is able to simulate hundreds of motes in realtime speed and scale to thousands in sub-realtime speed. To our knowledge, DiSenS is the first full-system sensor network simulator with such scalability.

Categories and Subject Descriptors I.6 [*Simulation and Modeling*]

General Terms Experimentation, Performance

Keywords Distributed Simulation, Sensor Network, Simulation

1. Introduction

Recent interest in sensor networks, in which tiny sensor devices are interconnected by unreliable wireless radio for non-invasive and inexpensive instrumentation and analysis of our living environment, have provided a chance to revisit many interesting traditional problems in the parallel and distributed computation area as applied to a new discipline. One such opportunity comes in the form of the need to be able to run large-scale ensemble sensor network simulations consisting of many independent-but-communicating individual device emulations. As in other distributed research areas, simulation is a useful tool for the system development, evaluation and analysis, and is widely used in sensor network research. In this context, simulation is particularly necessary since large-scale physical deployments are difficult to execute and expensive to maintain.

To be useful as a program development and debugging tool, these ensemble simulations require the precise emulation of internal functioning inside each entity to provide accurate event timing of the system. These machine-accurate emulations then communicate over a simulated network to provide a full scale simulation however the computation and communication requirements necessary to support the simulation (particularly in real time) are substantial.

In this work, we explore the use of parallel cluster computers to support ensemble sensor network simulations. Previous work in this area either implements such simulations sequentially [12] or using shared-memory parallelism [30]. In our approach, we treat the problem of implementing ensemble simulation as a task-parallel cluster computing problem and borrow several techniques from high-performance parallel scientific computing to achieve execution efficiency.

The state of art research in sensor network simulation employs two general approaches. Discrete-event systems such as those described in [12, 18, 28] model device functionality and communication as a set of partially ordered events modifying distributed state, much like in the traditional network simulation [16]. Often, these systems have focused on communication interactions (which takes place via unreliable and difficult-to-model communication radios) and only roughly approximate the behavior of the constituent devices themselves. By sacrificing device fidelity, discrete event simulators can achieve very high performance and scale well.

Full-system simulators [23, 26, 30, 19] take an alternative approach. They simulate the internal device functionality in detail and allow ensemble behavior to emerge from the interactions of independent-but-communicating simulated devices. These systems achieve sufficient fidelity levels, but the tightly coupled coordination among multiple simulated devices has limited their scalability.

Our work attempts to extract and combine the benefits of both approaches by employing distributed-memory parallel cluster system to attack the scalability problem while maintaining the simulation fidelity. This is essentially to map one distributed system with large number of emulations of simple, unreliable computational devices (tasks) to a cluster with a smaller number of much more powerful reliable hosts. The difference in computational power between physical sensor network devices and commonly available cluster nodes is so great that it is possible for a single cluster node to emulate multiple sensors in real time. To simulate inter-device communication, we intercept packets generated by each emulation and translate them into network packets for transmission across the cluster's network fabric. Our work, in the form of DiSenS (DIstributed SENsor network Simulation) – a distributed software infrastructure for scalable sensor network simulation, has achieved satisfying results towards this goal.

* This work was supported by grants from Intel/UCMicro, Microsoft, and the National Science Foundation (No. EHS-0209195 No. CNF-0423336, and No. NGS-0204019).

PPoPP'07 March 14–17, 2007, San Jose, California, USA.

DiSenS is first a sensor device emulator with high-fidelity and high-performance. It faithfully executes the program instruction by instruction and maintains the correct device state cycle by cycle. It also models the radio communications among sensor devices and the power consumption of each device. More importantly, DiSenS provides the distributed framework for the simulation of coordination among sensor devices (homogeneous or heterogeneous) on a distributed-memory cluster system so that scalability can be achieved by utilizing available computation resources. Our ultimate goal in developing DiSenS is to build a simulation framework that permits exploration of fidelity, completeness, scalability, and bridging, as outlined in [12]. We report on the degree to which we currently achieve this goal with DiSenS using both benchmarks designed to exercise various component features, and publically available sensor network operating system and application code that we treat as inviolate. In so doing, we believe that this work makes the following research contributions.

- We describe the distributed implementation methodology we have chosen for DiSenS, with a particular emphasis on the protocol we use to synchronize the emulated device clocks, and the partitioning strategy for mapping simulation components to cluster processors.
- We report on the fidelity that our full-system device emulations are able to achieve.
- We provide a detailed exposition and analysis of DiSenS's efficacy in terms of simulation performance, completeness, and most importantly, scalability.

As a whole, we believe these contributions extend the state-of-the-art in distributed sensor network simulation.

In rest of paper, Section 2 gives a brief overview of the device emulation framework, including the hardware emulation core and the pluggable radio and power models. Section 3 studies the synchronization problem and presents DiSenS distributed simulation algorithms. We present measurements and analysis of DiSenS functionality in Section 4, survey the related work in Section 5 and finally conclude in Section 6.

2. Parameterizable Device Simulation Framework

In this section, we give a brief overview of the device simulation framework, which faithfully emulates the sensor device hardware and lays the foundation for distributed simulation. We also describes the pluggable fidelity enhancing models, e.g. radio model, power model, etc., which allow experimentation with different fidelity levels and modes of investigation.

2.1 Cycle-Accurate Hardware Emulator

As the basis for accurate simulations, we have stressed the development of simulation tools that achieve timing accuracy. While accurate power and radio simulation techniques are the subject of much current research activity, we believe that successful approaches will depend, ultimately, on the ability to simulate device cycle timings correctly.

At the core of our device simulation framework is a hardware emulator with extensive support for various popular sensor network devices. In the current implementation, we emulate the mote [15] devices (the Mica2 and MicaZ platforms), Stargate devices [27], and iPAQ devices [10] and we are adding the support of other devices, like Telos [29]. Thus, the system is capable of heterogeneous sensor network simulations. In this work, however, we focus only on simulating ensembles of Mica2 and MicaZ devices exclusively.

The emulation core supports the following sensor node functionality and components by emulating

- the AVR instruction set,
- the ATmega128L microcontroller, including most on-chip functions: program memory, RAM, EEPROM, timers, serial devices (UARTs), SPI (Serial Peripheral Interface), ADC (Analog/Digital Converter), Watch Dog Timer and fuse bit setting (for boot loader and self programming),
- the 512KB on-board flash,
- the Serial ID chip,
- the CC1000 (Mica2) and CC2420 (MicaZ) radio chips,
- the LEDs and the sensor boards.

The heart of hardware emulator is a cycle-accurate AVR instruction interpreter. Hardware emulation is a mature area yielding several good technologies for simulating one architecture on another with high efficiency [23, 3]. However, we choose to use a fairly simple switch-based interpreter, that is similar to SimpleScalar [1]. The biggest reason is for portability. Since we intend to implement simulations using collections of machines, the ability to run on a broad range of architectures is essential. Moreover, the relatively simple nature of the AVR architecture and the high clock speeds available from commodity powered workstations makes it possible to achieve faster-than-real time emulations of many sensor devices. For example, our system is able to emulate motes using a 3.2GHz x86 processor at approximately 9-times real-time speed.

The interpreter emulates each instruction, changes the state of microcontroller and drives an internal clock cycle by cycle, which in turn fires the asynchronous events in an event queue, generated by hardware components like timers, UARTs and ADCs. The collection of emulated devices is rich enough and accurate enough to boot and execute unmodified TinyOS [8] binaries. Thus applications and operating systems cannot distinguish execution on the emulator from execution on the actual hardware.

2.2 Pluggable Models

Our device simulation framework provides a set of common interfaces for integrating the core hardware emulator to various extensions for power and communication. Our intention is to provide a platform for experimentation with different "plug-in" models, both to support the development of new models as well as to provide a way to trade simulation speed for fidelity using a suite of models. Since our focus is scalability, we only discuss the radio models in the following, which has great impact on the design of distributed synchronization protocol, and skip the power models.

2.2.1 Radio Models

Our system includes a "simple" or "ideal" radio model in which radio packets are sent losslessly to all the neighbor nodes within its radio range. While the ideal model is typically highly inaccurate, it is often used for initial code development and debugging as well as to achieve an upper bound on potential performance. Under this model, each sensor node buffers the packets sent to it even if it is not in receiving mode. Packets are time stamped and when a sensor node receives, it checks the packet buffer and reads the packets that match its current clock time. In addition, packets from different nodes may conflict with each other. When conflicting transmissions interfere, the ideal model performs a bit-wise *OR* of the bits received during the conflict period. As a result, this basic radio model is able to simulate transmission conflicts and thus the "hidden terminal" effect [31]. Also, packet loss due to the partial reception of packet preamble (because of the mis-synchronization of packet receiving and packet transmitting) is naturally modelled as part of the radio chip emulation logic.

The ideal model can be made more realistic through the addition of channel loss models. There are different ways to model the chan-

nel loss. Analytical techniques use a mathematical description of a physical electromagnetic radiation propagation. Thus, loss or signal perturbation is based on the "physics" of the intervening communication medium. There is a large body of literature on such physical models [21]. Despite their accuracy, however, their complexity and potential computational expense make them difficult to use in sensor network simulations.

A more popular approach is based a statistical description of channel loss, often derived from measurement trace data [4, 34, 33]. In this approach, a large set of radio transmission data is collected using different parameters. The trace data is then "mined" using statistical methods to derive distributional descriptions of characteristics such as reception rate. Cerpa et al. [4] explored this approach and achieved some noteworthy results. They have also proposed methods of generating realistic network instances based on the discovered feature distribution. In our work, we have developed a plug in that uses a loss rate distribution generated from our own measurement trace data using a similar methodology as in [4]. Thus, using the basic model and the trace-derived loss model, our system can incorporate both deterministic models based on mechanism and statistical models based on off-line analysis of trace data.

3. Distributed Simulation

One of the primary motivations for the development of our system is the ability to simulate "large" ensembles of sensors so that potential problems of scale can be studied. While previous work [30, 12, 17, 2] has addressed the issue of scalability using different approaches (cf. Section 5 for a review of related work), our goal is to support binary transparency with respect to the applications and operating system (similar to Avrora [30]) in a way that maximizes the size of the ensemble that can be simulated.

There are two measures of scalability DiSenS attempts to maximize. The first, analogous to the standard notion of speedup used to characterize parallel programs, is to maximize the ratio of wall-clock time that elapses for a complete sensor network simulation on a single processor to that for the same simulation running on multiple processors. This ratio characterizes the benefit of parallelism in terms of reduced execution time for a given simulation.

In addition, we also consider speedup (or more probably slowdown) in terms of the clock periods of the sensor network devices under study. By calculating the number of device clock cycles that have been simulated in a given wall-clock period, we can compute the speedup or slowdown of the simulation relative to the clock cycles that the real device experiences in real time.

Notice that these two notions of speedup are related but distinct. For example, it is possible for our system to achieve excellent speedup using the first measure (the parallel time is much faster than the sequential time) but poor speedup or even large slowdown using the second measure (devices are simulated only a small fraction of their real time speeds). While we have designed DiSenS to attempt to optimize both measures, we focus on the latter measure – the relationship to real time device speed – in this work as we believe it is the more challenging of the two.

Clearly, the degree to which these measures can be optimized depends on both the structure, constituent devices, and topology of the ensemble simulation and the characteristics of the computational resources. For the latter, we believe a distributed memory cluster computing environment has the largest potential. However, the typically close coupling of simulation systems makes distributed implementation challenging.

3.1 Background and Approach

Our approach is to simulate ensembles of sensor devices by executing individual cycle-accurate device simulations which communicate via simulated radios. Notice that this approach is distinct from an event-driven methodology in that we do not decompose the collection of simulations into explicit events that must then be time ordered. Rather, we use individual device emulations and a simulated radio communication environment as a virtual deployment of a complete sensor network, and run the same operating system and applications on the virtual sensor network as if they were running on an actual deployment[1]. To coordinate between individual device emulations, when radio communication occurs, the two communicating sensor devices must be synchronized with respect to their relative internal clocks.

Previous work that takes a similar approach includes ATEMU [19] and Avrora [30]. ATEMU [19] is a cycle-accurate sensor network simulator. It maintains a global clock and emulates one instruction a time for each simulated device. In this way, the sensor nodes are automatically synchronized and no extra facility is necessary to maintain the correct order of radio events. However, ATEMU is limited to a single process and can not scale to larger systems. Avrora [30] extends the simulation to a multi-threaded shared memory system. It scales on multi-processor machines. In Avrora, each device is simulated in a separate thread. Avrora loosens ATEMU's cycle-to-cycle synchronization requirement by extending the synchronization period to the length of a byte transfer time – 3072 ATmega128L cycles – since packets are always transmitted in byte unit. A thread barrier is used to achieve its lock-step style synchronization, which stops all the threads periodically to ensure every radio byte will be correctly received during the correct time period.

In a clustering computing environment, relatively large and variable network latencies make direct extensions of these two approaches difficult. Cluster network latency is measured in milliseconds while a desktop PC can easily emulate one device instruction in the 0.1 micro-second range. If lock-step global synchronization is used, the simulated clock speed will be determined by the all-to-all network communication latency.

3.2 Synchronizing Ensemble Emulations

Our approach to synchronizing multiple device emulations relies on an abstraction of the radio communication protocol. To illuminate the nature of this abstraction, we begin by discussing sensor network radio behavior in some detail.

Currently two types of radio chips are emulated in our device emulator, the CC1000 chip and CC2420 Zigbee radio chip [5], both manufactured by Chipcon. The CC1000 is the radio chip used by the Mica2 senor mote [13] and the CC2420 is used in the more recent MicaZ [14] platform. The CC1000 is a rather simple radio chip. It has two working modes: transmitting and receiving (ignoring power saving features of the chip for the moment). In transmitting mode, data bits are pumped in from the SPI line, modulated, and emitted through the antenna. In receiving mode, the radio signal is amplified, demodulated and converted into digital bits which will be assembled into radio packets by software protocol stack. The mode transition is controlled by the software. CC1000 also has a receive signal strength indication (RSSI) measurement function. This analog value of the signal strength is output via a chip pin, and converted into digital value by the ADC module of the microcontroller. The RSSI value is used by the software MAC layer to perform collision detection.

CC2420 is a more advanced radio chip that implements the low level function of Zigbee (IEEE 802.15.4) standard. The major difference between CC1000 and CC2420 from the simulation point

[1] Our style of simulation might more properly be termed an "emulation" as a way of emphasizing the distinction between our approach and an event-driven one. Because the radio environment is purely simulated, however, we have chosen to term our approach as a "simulation" since we believe that term to be more general.

of view is that CC2420 performs the packet assembly in the chip and has a much faster transmission speed. CC2420 has a similar signal sampling function and also measures RSSI value. However CC2420 uses a pin called CCA (clear channel access) to indicate whether the radio channel is clear based on a preset threshold. This provides a simpler interface for MAC layer collision detection.

The typical radio activity paradigm of TinyOS sensor applications can be described as follows. Normally, the radio stays in receiving mode (it may be turned off for power saving). When a preamble of a packet is recognized, the complete packet payload is to be assembled and uploaded to the application. When a packet needs to be sent, the MAC layer checks the channel using RSSI value or CCA value. If the channel is busy, it backs off for a random period of time and tries again. Otherwise, the radio chip is switched into transmitting mode and a complete packet is sent out.

Thus, packet receiving and signal sampling are very similar operations: they both read a value from the channel. The only difference is the length of time they use to access the channel. As a result, radio communication behavior can be abstracted into two operations: *read_channel* and *write_channel*. The *read_channel* represents the packet receiving and the signal sampling. The *write_channel* represents the packet transmitting.

As discussed previously, global clock is not feasible in a distributed environment since every clock access needs to traverse the network thereby incurring a large overhead. Instead, we use a peer-to-peer design in which each sensor node maintains its own local clock, clocks are synchronized before message rendezvous, and each node is otherwise simulated independently.

When a communication between nodes occurs, the causal relationship that exists between sender and receiver is rectified at the receiver so that packets are received in order, and that local clock values roughly correspond to arrival timings. We formalize this synchronization problem in abstract terms and then discuss our proposed solution.

We first define the simulation:

DEFINITION 1 (Simulation). *If we define a radio node N_i as a tuple $(clock_i, read\_channel, write\_channel)$, where $clock_i$ is the internal clock of node N_i, $read\_channel$ and $write\_channel$ are the only two operations performed on a shared resource, C, representing the channel, we can define a* simulation *S as a set: $(N_0, N_1, ..., N_k, C)$.*

We have to distinguish the *simulation time* and *simulated time*. The former is the wall clock time in real world that is used to measure the simulation. The latter is the virtual clock time in simulated world that is shared by simulated motes.

Then we define the correctness of a simulation:

DEFINITION 2 (Correctness). *A* simulation *is correct if the following relationship is ensured: $\forall$* simulated time *period $[vt_{i_1}, vt_{i_2}]$ (corresponding* simulation time *period $[rt_{i_1}, rt_{i_2}]$), at which node N_i is scheduled to $write\_channel(C)$, and its neighbor node N_j is to $read\_channel(C)$ during $[vt_{j_1}, vt_{j_2}]$ (*simulation time $[rt_{j_1}, rt_{j_2}]$*); if $[vt_{i_1}, vt_{i_2}] \cap [vt_{j_1}, vt_{j_2}] \neq \emptyset$, $[rt_{i_1}, rt_{i_2}] \cap [rt_{j_1}, rt_{j_2}] \neq \emptyset$.*

Intuitively, a *correct* simulation requires any receiver to receive any data that it is meant to receive according to the causality in simulated time space. In our simulation structure, given that sent data is transferred in byte unit and buffered at the receiver side, correct simulation can be achieved if each receiving node delays the delivery of each message byte until the local clock on the receiver is past the local clock on the sender.

Conservatively,

PROPERTY 1 (Safe Receive). *if whenever a node N_i invokes operation $read\_channel$, it waits until synchronized with its neighbors, which means $\forall k$, if N_k and N_i are neighbors, $clock_k >= clock_i$, the simulation S is* correct.

Note that we have to be conservative by waiting all the neighbors since we can not predict which neighbor will transmit at the time when we receive. We term this property the *safeness property.*

3.3 Distributed Synchronization Protocol

Based on the safeness property we design the complete synchronization protocol for distributed simulation. We first introduce a primitive, *wait_on_recv*.

DEFINITION 3 (wait_on_recv). wait_on_recv *is a primitive operation. If it is called by a node N_i, it waits until $\forall k$, N_k is a neighbor of N_i, $clock_k >= clock_i$.*

wait_on_recv has to be called every time the radio channel is accessed (receiving or sampling).

Since *wait_on_recv* relies on the clock information of neighboring nodes, each node has to be informed of its neighbors' local clock value. We use a clock update protocol in which each node broadcasts its local clock time periodically. The length of update interval does not affect correctness but does have effect on performance. There are two requirements on when to send updates. First, clock updates can not be sent during the transmission of a byte. This is because if it is sent, a neighbor waiting on a receive will believe it is time to proceed (if it does not wait for others) and may miss a partial byte. Updates, then, can only be sent between bytes during a transmission. Second, before a node starts to wait by calling *wait_on_recv*, it must first send an update. Without notifying its neighbors of its intention to wait, a node's silent wait will cause a deadlock if some other nodes are going to wait for it.

In summary, any receiver *wait_on_recv*s to block and wait for neighbors' clock updates before it receives a message or samples the radio medium. Before blocking, however, it must reliably inform its neighbors of its local clock value to prevent deadlock.

Using the above synchronization protocol, we implement our distributed simulation system. Given a set of nodes, we first partition them into groups. Each group is simulated on one machine and each node is simulated in one thread. In each group, a clock table is maintained to keep the updated clock time for all local nodes and their neighbor nodes. Whenever an clock update is sent, it first updates the local neighbors and then multicasts to the remote neighbors if it has. Our synchronization protocol treats the local and remote synchronization in the same way. The following pseudo code demonstrates the synchronization algorithm of a sensor node.

```
do_for_every_byte_transfer_time() {
    switch (mode) {
    case RECEIVING:
        send my clock update;
        wait_on_recv();
        retrieve data byte from packet buffer;
        break;
    case TRANSMITTING:
        send my clock update and data byte;
        break;
    default:
        send my clock update;
        break;
    }
}
```

The above code doesn't show the algorithm for signal sampling (RSSI) operation, which is the same as receiving (the "RECEIVING" section in *switch* statement). The code shows that we send at least one clock update for every byte transfer time regardless of radio modes. For transmission, data byte is "piggy-backed" on the clock update messages to reduce the message traffic. Notice that

there is no constraint for senders. Senders send data bytes at any time they want. The sent data bytes are buffered at receivers' side. And it is receiver's responsibility to ensure the correct reception of radio packets. Notice also that there is a great deal of overhead in this protocol. If this overhead cannot be amortized or ameliorated by the performance of the network interconnect within the cluster, the overall performance of the ensemble simulation will be low. Our results seem to indicate that these issues are addressable, however.

Here is the code for *wait_on_recv*:

```
wait_on_recv(nodei) {
    for (all nodej as a neighbor of nodei) {
        if (nodej's time < nodei's time) {
            put nodei on nodej's waiting list;
        }
    }
    if (nodei waits on any node)
        wait();
}
```

The following code shows what happens when a clock update is received, regardless locally or remotely.

```
update_clock(nodei, clock) {
    nodei.clock = clock;
    for (all nodej waiting on nodei) {
        if (clock >= nodej's time) {
            decrease nodej's waiting count;
            if (its waiting count is 0)
                wake up nodej;
        }
    }
}
```

3.4 Node Partitioning for Parallel Execution

As indicated, the major potential source of overhead comes from the network synchronization necessary to keep the various emulations synchronized. To get maximal performance, we must reduce the remote synchronization as much as possible. Thus partitioning the sensor nodes into groups plays an important role in the making of an efficient simulation.

The amount of remote network synchronization is determined by the number of remote neighborhood links between sensor nodes. Local updates to neighbors co-located on the same machine are relatively inexpensive (because they can use a shared data structure in memory) compared to remote clock update synchronization. As such our nodes partition algorithm has two goals. First, we need to evenly distribute the node workload among groups if we are running simulation on a homogeneous system like a dedicated cluster. This need for load balancing is because any slow host will become the bottleneck of the whole simulation due to the implicit dependency among nodes. Second, we want to minimize the number of links among remote neighbors that are assigned to processors that can only communicate via network messages.

We find that we can actually convert this optimization problem into a "classical" graph partition problem that is well studied in parallel computing area [24, 25, 20]. Formally, the partition problem is as follows. Given a weighted, undirected graph $G = (V, E)$, the k-way graph partition problem is to split the vertices of V into k disjoint subsets such that each subset has roughly equal amount of vertex weight while minimizing the sum of the weights of the edges whose incident vertices belong to different subsets (an edge cut) [24].

In our case, given a node map, which specifies the node coordinates in a 2D or 3D space, and the maximal transmission range of a typical sensor node, we can build up a graph called potential neighboring graph (PNG). Each vertex of the graph is a node. Each edge represents that the connected two nodes have the potential to communicate. Then the node partition problem is exactly a graph partition problem with both edge and vertex weights to be unitary.

The graph partition problem is known to be NP-complete in general. A large body of research explores heuristic algorithms. There are geometric algorithms, like recursive inertial bisection that uses coordinate axes to partition the graph; combinatorial algorithms, like K-L algorithm that optimizes an partition locally; spectral methods, which transform the discrete optimization into a continuous one using linear algebra; and multilevel algorithms featuring a coarsening-refining process. In our simulator, we use a general graph partitioning package for parallel computing from Sandia National Lab, called Chaco [7], which combines these techniques based on graph topology and vertex and edge weights. We use Chaco without modification and plan to report on its effectiveness in a future effort. Anecdotally, however, we are quite pleased with the quality of the partitions it generates for the simulations we have investigated.

3.5 Scalability Analysis

Before looking at the experimental results generated by our implementation, we attempt to describe the potential scalability of the system analytically. The simulation performance is determined by the pure device simulation speed and the synchronization overhead. Ultimately, the computational and memory cost of emulating individual devices will dominate performance, but the machine and memory speeds of the cluster hosts are so much more powerful than the devices simulated on them, it is the network synchronization that poses the greatest impediment to scalability.

We define the following property that describes the scalability of our algorithm in the ideal case.

PROPERTY 2 (Scalability). *Given fixed map density d and node density D and node number N_h on each host, when the number of hosts H increases, and thus the simulated nodes N increase, the communication cost for each host is constant if the partition of nodes to hosts is optimal.*

Here the map density d is defined as the ratio between the sum of areas of node range circles (the circle centered at the node with maximal radio range as radius) and the area of the map (maximal area that the nodes occupy). It is a good indication of nodes' average number of neighbors.

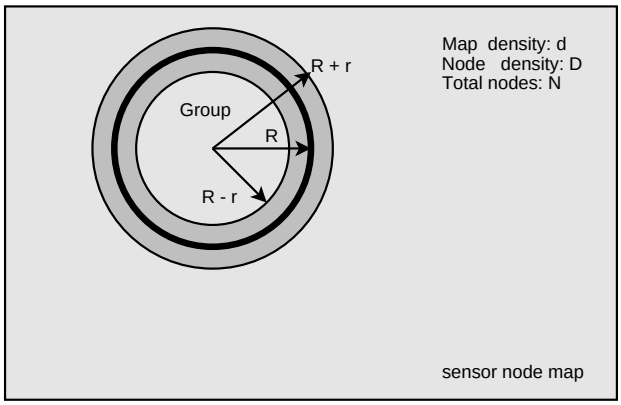

Figure 1. Illustration of Property 2

The property is illustrated in Figure 1. The circle with radius R represents a group. We can use circle is because, assuming an optimal partition, the group should have minimal contact with others and a circle is a good estimation of its boundary. Since the map density d is fixed and the number of nodes per group

is also fixed, the area of a group and thus radius R is constant. Moreover the ring area corresponding to the area between circles having radius $R - r$ and radius $R + r$ both having the same center is the area which nodes may have cross-group edges, if r defined to be the maximal radio transmission range. Then the number of cross-group edges for a group can be calculated as follows:

$$Num_{edge} \approx Area_{ring} * Density_{node} * Density_{map} = 4\pi RrdD \quad (1)$$

Since R, r, d and D are all constant with respect to the *total number of nodes*, the number of cross group edges is fixed. Thus the communication cost of each *host* is fixed. Although Property 2 corresponds to an ideal upper bound on communication overhead, it predicts that scalability will be affected most by the number of nodes assigned to each processor rather than the total number of nodes simulated or the total number of processors employed given fixed map density and nodes per host. Our experimental results described in the next section seem to reflect this outcome.

4. Evaluation

In this section, we examine the fidelity and scalability of DiSenS. As a measure of fidelity, we compare cycle counts generated by our simulator to those observed from individual real devices (using an oscilloscope to maximize measurement accuracy). This cycle count comparison is for full-device emulation (CPU and memory, flash storage, radio, etc.) using a set of benchmarks designed to exercise all sensor hardware subsystems.

We also have investigated the transparency and completeness of our system by booting unmodified TinyOS images on the simulator and executing popular large sensor network applications: TinyDB, Surge and Deluge [9]. However, due to the length limit and our focus on scalability, these results are not provided in this paper.

Finally we examine the scalability of DiSenS using a single benchmark (employed previously in the literature for such studies) and compare the results to those generated by previous efforts.

4.1 Experimental Framework

The results presented in the following exposition have been generated using two different machine clusters to which we have access at UCSB. CLUSTER1 is a 16 host [2] dual-processor 3.2GHz Intel Xeon cluster that uses switched gigabit Ethernet as its communication interconnect. CLUSTER2 is a larger, departmental cluster composed of 64 dual-processor 2.6GHz Intel Xeon hosts, again interconnected via a gigabit Ethernet switch. Both systems are used in dedicated mode to remove the effects of network or host contention by other executing applications.

For all the scalability experiments, we use TinyOS application *CntToRfm* as the benchmark. *CntToRfm* has been used as the touchstone in previous scalability studies [12, 30]. *CntToRfm* periodically sends out radio packets and keeps the radio channel busy. Note that although it does not actually receive packets, the radio chip is still in receiving mode when it is not transmitting so it does in effect exercise all radio activities. Our experiences with other applications in scalability are similar as *CntToRfm*.

For most of the experiments, we use Mica2 as the target sensor device. At the end of this section, however, we briefly discuss scalability results for MicaZ to show how the effect of radio transfer speed on simulation performance.

[2] The term "node" is rather unfortunately common to both the sensor network and cluster computing communities occasionally leading to confusion when discussing sensor network emulation on clusters. In the remainder of this paper, we will use the term "host" to refer to a node in a cluster, and the term "node" to refer to a sensor network device.

4.2 Cycle-Accuracy

We use four benchmarks to test the cycle-accuracy, exercising important components on the mote device. *cpu* benchmark runs CPU intensive computations. *flash_read* performs small reads from the on-board flash chip. *flash_write* writes to the on-board flash. *radio* exercises the CC1000 radio chip and transfers a small amount of data.

The execution time on real device is measured using an oscilloscope, Agilent 54621A (accurate up to 10 nanoseconds). Each benchmark starts by writing an "1" to a pin in I/O port C and ends by writing a "0" to the pin. The pin is connected to the oscilloscope probe. The oscilloscope measures the pulse width. The measured time is then converted into cycle numbers using a division with ATmega128L's clock speed (7372800 Hz). The numbers are compared with our simulation result.

We run the *radio* benchmark in an environment with minimal interferences and make the antennas of two motes in close distance to minimize the effects of noise and communication channel instability on cycle timings in an best effort, since we compare to the ideal radio model.

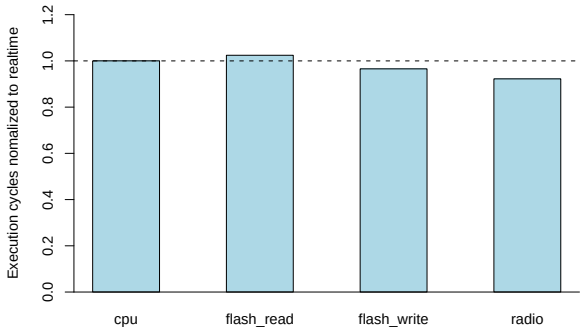

Figure 2. Normalized average cycles for benchmarks.

Figure 2 gives the average of 20 measurements as the ratio of simulated execution cycles to cycles measured from the actual devices (a ratio of 1.0 would indicate perfect accuracy). That is, we normalize the data using actual measured cycle counts. For CPU emulation, the simulator closely approximates empirical measurement. Flash and radio emulation have slightly larger errors, but the size of these errors is of limited statistical significance. In [32] we provide a more complete statistical analysis of this comparison which we omit from this work due to space considerations. Instead, by way of summary, we note that in general the simulation error is relatively small.

4.3 Scalability

For each scalability experiment, we vary two experimental parameters independently: the number of sensor *nodes* simulated on each *host* in a cluster, and the number of *hosts* used for each experiment. Thus, for example, the value in row 2, column 4 shows the results for two nodes per host and four hosts. For each node-count-host-count pair, we run *CntToRfm* for 60 seconds and record the average simulated clock speed. Except where noted explicitly, all the experiments are run on CLUSTER1.

4.3.1 Best Case: One Dimensional Topology

Our first experiment simulates a one dimensional topology. All the nodes are laid on a straight line, 50 meters apart (again assuming

Nodes per host	Hosts number				
	1	2	4	8	16
1	9.28	2.26	1.96	1.72	1.67
2	6.68	2.12	1.82	1.68	1.68
4	2.18	1.83	1.70	1.68	1.67
8	1.20	1.21	1.18	1.16	1.15
16	0.78	0.61	0.60	0.60	0.60
32	0.35	0.36	0.31	0.31	0.31
64	0.18	0.15	0.17	0.15	0.14
128	0.09	0.09	0.09	0.08	0.08

Table 1. Simulated clock speed for 1-D topology. Each row has fixed number of nodes per host and each column has fixed number of hosts. All value is normalized to real time clock speed.

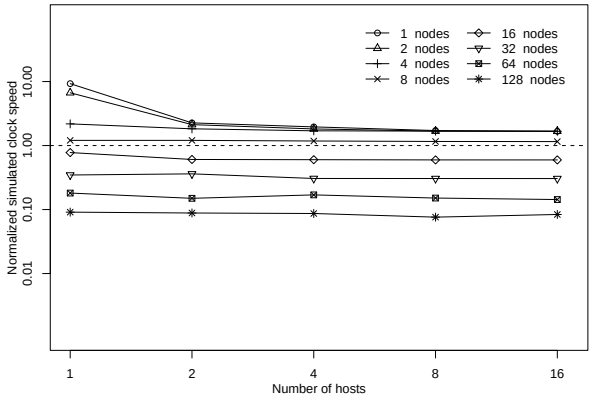

Figure 3. Scalability of 1-D topology. X-axis is number of hosts and Y-axis is clock speed. Each curve represents the performance with a fixed number of nodes per host. Dashed line shows real time speed.

the maximal radio range is 60 meters). This gives us the minimal cross group edges (given an optimal partition). It constitutes the best possible case for the distributed simulation and as such provides a rough upper bound on the performance.

Table 1 presents the results. Each cell of the table shows the ratio of the simulated average clock speed to the real time clock speed, of 7372800 cycles per second. To compute the average simulated clock speed, the simulator records the number of clock cycles each mote executed during the 60 second execution run. The sum of the cycles is divided by the number of motes, and that number is divided by 60. Thus each cell depicts the average slowdown or speedup factor relative to native execution speed.

From the table, the best performance is a speedup of **9.28** times real time speed when simulating one node on one host (the upper lefthand corner in the table). Notice that as expected, simulating more nodes on a single host (column 1) yields a slower rate of decay in the speedup factor than does simulating one node on each of a successively larger number of hosts (row 1). When two nodes are co-located in the same host, the speedup factor drops to **6.68** whereas two nodes each located on a separate host generates a speedup factor of only **2.26**. This is due to the network communication overhead. The trend reverses when the host number is getting larger when the communication overhead is amortized among peer nodes on the host.

What is perhaps the most remarkable, however, is the similarity between the values for 2 through 16 hosts. While we expected a substantial fall off in speedup in moving from one host to two hosts, we expected that fall off to continue as the number of hosts increases. Indeed, starting with 8 nodes per host (the fourth row in

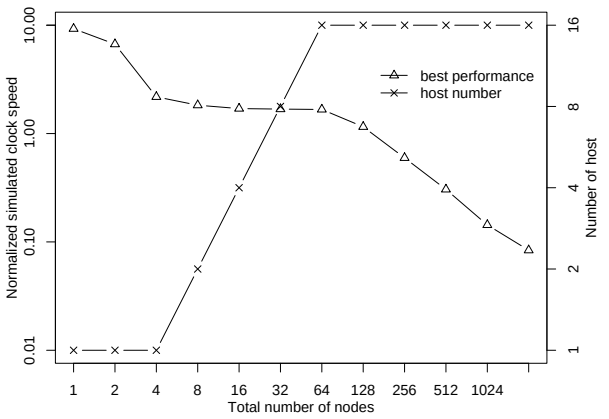

Figure 4. Gold curves for 1-D topology. X-axis is total number of nodes simulated. The left Y-axis is normalized performance and the right one is number of hosts. The decreasing curve is the fastest speed curve. The increasing curve gives the corresponding host number at each point.

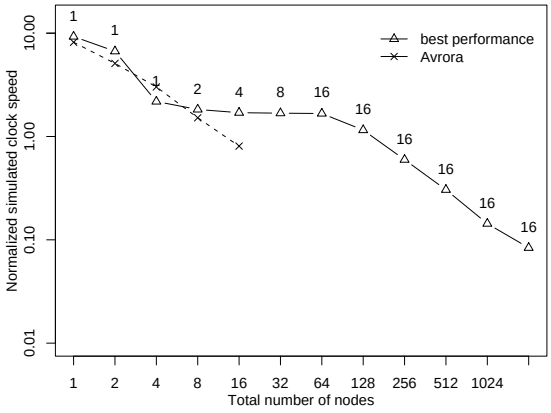

Figure 5. Best performance curve comparison. X-axis is total number of nodes simulated. The Y-axis is normalized performance. Compare our best performance (1-D topology) with Avrora's performance. The annotated number is the corresponding number of hosts.

the table) the speedup factors are remarkably similar regardless of host count. Further, the tipping point with respect to speedup and slowdown (the point where the ratio falls below 1.0) is between 8 and 16 nodes per host for *all* host counts.

Figure 3 shows this relationship graphically using a log-log scale. The speedup drops for small node counts from one host to two, but for the other data points, the number of nodes per host (and not the number of hosts) is the determining factor up to 16 hosts. This relationship is predicted by the analysis of Theorem 2 presented in the previous section but none the less, we found the degree to which it holds somewhat surprising.

By way of comparison to previous work, in this best case scenario **2048** nodes can be simulated at nearly a tenth of the real time speed using 16 hosts (lower righthand corner of Table 1), which is almost 8 times better than results reported for TOSSIM [12]. Also, nearly **160** nodes can be simulated in real time speed using 16 hosts, and improvement of almost a factor of 5 over previous TOSSIM results.

In Figure 4, we plot the best performance of simulating 1, a total of 2, 4, ..., and 2048 nodes respectively. The units of the y-axis on the lefthand side of the graph are for the ratio shown in Table 1. For each point, we also plot the corresponding "host number" at which the best performance is achieved (the host count is shown on the y-axis at the righthand side of the graph). We call the two curves "gold curves" since they show the number of hosts necessary to obtain the fastest simulation of a specific number of nodes. Note that the fall off in the best performance curve occurs when the number of hosts reaches 16 (the maximum number in CLUSTER1) and the total node count is increased beyond 64. Thus, in this best case example, scalability is limited by host availability through 2048 simulated nodes.

We compare Avrora [30]'s best performance curve with our "gold curve" in Figure 5. We run Avrora on a single host from CLUSTER1 (using both processors on that host) for up to 16 nodes (the implementation of Avrora we ported to our machine did not execute correctly with more than 16 nodes). Recall that Avrora is not designed to use distributed memory parallelism and message passing but it can take advantage of multiple processors in a single hosts that share memory. Despite the extra overhead we have in our system that is necessary to take advantage of multiple hosts, the performance comparison is favorable to our work. For up to 8 hosts, our system and Avrora achieve similar speedup factors. For the 8 node comparison, however, we require 2 hosts, using both processors on each host (the small integers next to each triangular graph feature in Figure 5 indicate how many hosts our system requires to achieve the corresponding speedup factor) where Avrora is using only one. Beyond 8 nodes, however, our methodology, using successively larger host counts, achieves a considerable scalability improvement over Avrora.

4.3.2 Common Case: Two Dimensional Topology

Nodes per host	Hosts number				
	1	2	4	8	16
1	9.14	2.52	1.83	1.66	1.64
2	6.65	2.12	1.58	1.38	1.18
4	2.09	1.49	1.27	1.12	1.10
8	1.25	1.07	1.01	0.96	0.92
16	0.82	0.63	0.62	0.59	0.57
32	0.32	0.38	0.31	0.30	0.30
64	0.16	0.17	0.16	0.15	0.15
128	0.10	0.08	0.07	0.07	0.07

Table 2. Simulated clock speed for 2-D topology. Each row has fixed number of nodes per host and each column has fixed number of hosts. All value is normalized to real time clock speed.

A two dimensional topology is more realistic for sensor network applications. Using the same configurations, we perform the experiments on a two dimensional node map. The nodes are again 50 meters apart and fill a grid whose shape is as close to a square as possible. Table 2 provides the results. The performance for a 2-D space is somewhat worse than for the 1-D case when the number of nodes per host is below 32. However comparing Tables 1 and 2 for node-per-host counts above 32 shows surprising similarity. Again, as the number of simulated nodes increases, the number of available hosts becomes the scalability limiting factor – not the node count. In the 2-D case, however, performance equivocates between 16 and 32 nodes per host corresponding to a slowdown factor of between 0.6 and 0.3. That is, while it is possible for our system to achieve scalable 2-D simulation of the benchmark, it is not possible to do so and to run in faster-than-real time.

We present the "gold curves" in Figure 6 but combine the node count and Avrora comparison curves onto a single graph. Again,

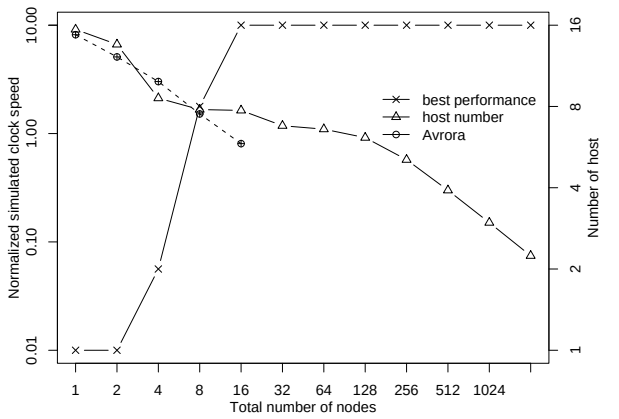

Figure 6. Gold curves for 2-D topology. X-axis is total number of nodes simulated. The left Y-axis is normalized performance and the right one is number of hosts. The decreasing curve is the fastest speed curve. The increasing curve gives the corresponding host number at each point. The dashed curve is Avrora's speed curve.

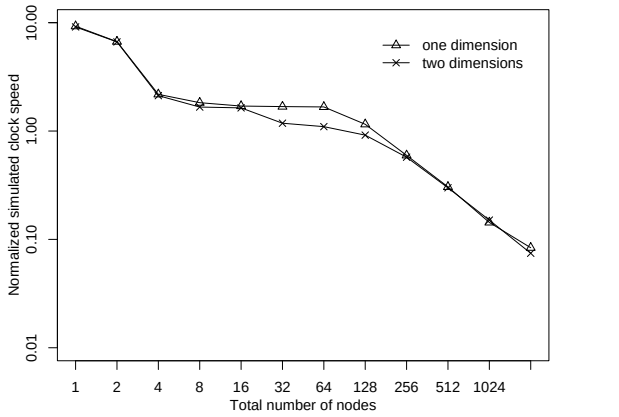

Figure 7. Best performance comparison of 1-D and 2-D topology. X-axis is total number of nodes simulated. The Y-axis is normalized performance.

our system performs similarly to Avrora (this time on the 2-D problem) but in this case, it requires more hosts to achieve the same results. For example, the simulator requires 8 hosts to duplicate Avrora's 8 node performance (using a single host). Surprisingly, however, the 2-D gold curve and the 1-D gold curve have similar shape. Figure 7 shows both on the same graph (note the log-log scale). Between 32 and 128 simulated nodes there is a reduction in speedup factor for the 2-D case, but apart from that region, the curves track almost exactly.

4.3.3 Worst Case: All-to-all Network

The previous scalability results we have presented rely on the limited neighborhood relationship imposed by radio range. For the worst case, we simulate an "all-to-all" complete graph configuration in which each simulated node must consider all of the other nodes to be in radio range making communication overhead maximal. Table 3 and Figure 8 shows the speedup factors and scalability curves respectively. In this worst case, communication overhead increases as the square of the node density. For small node-per-host

Nodes per host	Hosts number				
	1	2	4	8	16
1	9.28	2.36	1.66	1.60	1.36
2	6.68	1.41	1.07	0.81	0.66
4	2.04	0.94	0.75	0.62	0.42
8	1.22	0.65	0.54	0.43	0.29
16	0.62	0.44	0.32	0.23	0.14
32	0.29	0.20	0.14	0.08	0.04
64	0.12	0.08	0.04	0.02	0.01
128	0.05	0.02	0.01	0.002	0.0008

Table 3. Simulated clock speed for "all-to-all" complete graph. Each row has fixed number of nodes per host and each column has fixed number of hosts. All value is normalized to real time clock speed.

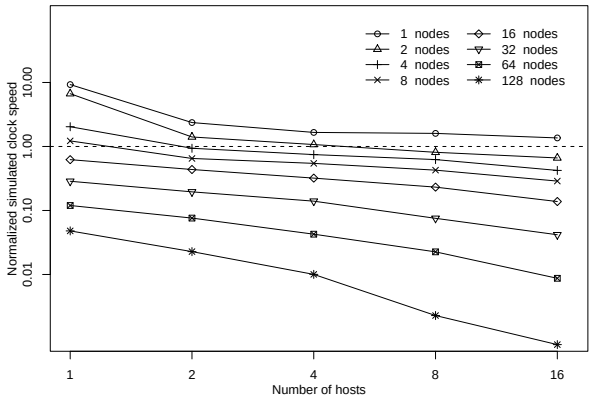

Figure 8. Scalability of "all-to-all" complete graph. X-axis is number of hosts and Y-axis is clock speed. Each curve represents the performance with a fixed number of nodes per host. Dashed line shows real time speed.

and host counts, the speedup factors are similar to the 1-D and 2-D grid cases, but as both are increased the speedup factor is continually reduced.

4.3.4 Larger Scale Experiment

Nodes per host	Hosts number						
	1	2	4	8	16	32	64
1	7.21	0.85	0.70	0.55	0.45	0.41	0.41
2	3.33	0.55	0.50	0.44	0.38	0.34	0.32
4	2.51	0.55	0.48	0.42	0.39	0.35	0.34
8	1.37	0.51	0.44	0.39	0.36	0.39	0.30
16	0.74	0.47	0.39	0.37	0.37	0.36	0.33
32	0.37	0.32	0.29	0.29	0.27	0.28	0.23
64	0.17	0.16	0.15	0.15	0.13	0.16	0.12
128	0.08	0.07	0.07	0.07	0.07	0.08	0.07

Table 4. Simulated clock speed for simulation of 2-D grid of Mica2 motes on CLUSTER2. Each row has fixed number of nodes per host and each column has fixed number of hosts. All values are normalized to real time clock speed.

To test our simulator in a larger scale, we perform the 2-D experiment on CLUSTER2, a 64-node cluster. Table 4 presents the results. Comparing Table 4 to Table 2 (which used CLUSTER1 for the same configuration) CLUSTER2 achieves lower speedup factors for the test cases they have in common (columns 1 through 5). Both a slower processor speed in CLUSTER2 and, somewhat curiously, higher network latency (even though CLUSTER1 and CLUSTER2 both use gigabit Ethernet as an interconnect) contribute to this lower performance. However, as the size of the sensor network being simulated scales, the overhead is once again amortized. For example, using 64 hosts of CLUSTER2 and 128 nodes/host our system can simulate **8192** nodes in total with a slowdown factor of 0.07 representing an almost **32** fold improvement over previously reported TOSSIM results.

5. Related Work

There have been numerous previously successful efforts to build sensor network simulation systems. Of these, Avrora [30] is the most similar to our work. Avrora is a full-system sensor network simulator supporting cycle-accurate emulation of the Mica2 Mote platform. Avrora uses a multithreaded structure in which each sensor node is simulated in a separate thread. A lock-step style synchronization scheme that is coordinated with the communication model is used to ensure the correctness of radio simulation. Avrora has a "Wait for Neighbors" operation that is similar to our *wait_on_recv* primitive, but it is only used to implement correct signal sampling operation. In addition, because it relies on clock-cycle synchronization according to a strict periodic schedule, Avrora can only scale using shared-memory multiprocessors using low-latency shared memory to facilitative the necessary communication. In contrast, by synchronizing on message transmission (and not through a lock-step mechanism) and by controlling message delivery to preserve time ordering, DiSenS executes efficiently on both shared-memory multiprocessors and distributed memory cluster platforms. DiSenS also includes an automatic partitioning system that optimizes simulated node to machine mapping based on simulated deployment topology. It is thus able to achieve greater scalability and overall performance than Avrora, and indeed many high-performance discrete-event simulators such as TOSSIM [12]. Finally, DiSenS also has extended hardware support, including even advanced platforms like Stargate (not discussed in this paper).

ATEMU [19] is another full-system sensor network simulator. It focuses on the detailed bit-level hardware simulation. It employs a very simple synchronization mechanism by executing one instruction a time for each sensor node. It is so simple that no extra facility and protocol is required to ensure correct radio simulation because nodes are already synchronized cycle-by-cycle using shared memory data structures. However, it can only utilize one process and thus does not scale to parallel computational resources.

Other simulators, include TOSSIM [12], SensorSim [18], GTSNetS [17], OLIMPO [2] and Shawn [11], explore the tradeoff between accuracy and performance by using discrete event simulation to elide the complexity associated with cycle-accurate hardware emulation. These systems typically report higher performance levels than simulators such as Avrora and ATEMU, but sacrifice execution transparency to do so. That is, these systems do not achieve the transparency of DiSenS in that application and operating code must either be translated to, or compiled for, their respective discrete-event environments.

TOSSIM is a popular event-driven simulator which models not only the wireless network but also the application behavior. TOSSIM is light weight and can simulate thousands of sensor nodes on one host. Nonetheless, we demonstrate how DiSenS is able to achieve improved performance levels over TOSSIM by leveraging distributed cluster resources while achieving transparency and cycle accuracy. That is, DiSenSachieves the cycle accuracy and execution transparency of systems such as Avrora, with performance levels that exceed those achieved by discrete-event simulators such as TOSSIM.

Similarly, SensorSim is a sensor network simulator based on NS-2 [16] which is a discrete event network simulator. It does not model application itself as TOSSIM does achieving even less

transparency. A sensor network simulator based on GTNetS [22] claims to be able to simulate a sensor network at a scale of hundreds of thousands of nodes. This scale exceeds what we have been able to test using the resources at our disposal, but to achieve this level of scalability, the operating system and application codes must be represented in a high-level, abstract way. Thus it is not possible to use this system to directly compare executions of sensor network software in simulation and on real hardware, as it is using DiSenS.

The general distributed simulation problem has been studied [6] for quite a long time. Compared to most existent discrete event simulation systems, the key feature of this work is that unmodified sensor network program binaries can run at scale, in or near real time in a fully instrumented simulated environment as if they were executing in real hardware. While there have been some recent virtualization efforts that can run native binaries using VMs like Xen, scaling these systems beyond what can be achieved in native mode has yet to be demonstrated. In contrast, DiSenS is able to emulate sensor networks that are far greater in scale than previous systems. Thus, by employing effective parallel techniques and combining them with high performance virtualization, DiSenS enables virtualized execution of program binaries at a scale that is larger than can be investigated through execution on native hardware. It is this capability that forms the novelty of the contribution.

6. Conclusion

DiSenS is a complete sensor network simulation framework providing high levels fidelity, extensibility and scalability. It addresses the conflict between accuracy and performance. Given enough computational resources, researchers do not have to trade simulation quality for simulation efficiency.

DiSenS also provides a complete and transparent simulation framework, including a cycle-accurate device emulator and replaceable plugin models. Users of DiSenS are able to employ customized models to explicitly control simulation quality. Internally, DiSenS uses a peer-to-peer simulation design for distributed clock synchronization. Individual node simulation threads are glued together by a simple and efficient synchronization protocol, which makes the complete simulation scalable to a large size of distributed computation resources. Using commodity cluster hardware, DiSenS can simulate one node approximately **9** times faster than real time speed, **160** nodes in real time speed using 16 dual-processor machines and **8192** nodes at nearly tenth of real time speed, which is **32** times of that reported previously [12].

We are actively improving DiSenS to make it a useful tool for sensor network research. A big challenge is to look for a dynamic node partition algorithm so that non-dedicated, heterogeneous distributed systems can be used for simulation.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 2002.

[2] J. Barbancho, F. Molina, C. Len, J. Ropero, and A. Barbancho. OLIMPO, An Ad-Hoc Wireless Sensor Network Simulator for Optimal SCADA-Applications. *Communication Systems and Networks (CSN 2004)*, 450, Sept. 2004.

[3] The Bochs Emulator. `http://bochs.sourceforge.net`.

[4] A. Cerpa, J. L. Wong, L. Kuang, M. Potkonjak, and D. Estrin. Statistical Model of Lossy Links in Wireless Sensor Networks. *In the ACM/IEEE Fourth International Conference on Information Processing in Sensor Networks*, Apr. 2005. Los Angeles, California.

[5] RF Receivers from Chipcon. `http://www.chipcon.com/index.cfm?kat_id=2`.

[6] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley and Sons, Inc., 2000.

[7] B. Hendrickson and R. Leland. The Chaco User's Guide: Version 2.0. Technical Report SAND94–2692, Sandia National Lab, 1994.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.

[9] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. *The 2nd ACM Conference on Embedded Networked Sensor Systems*, 2004.

[10] iPAQ devices. `http://welcome.hp.com/country/us/en/prodserv/handheld.html`.

[11] A. Kroeller, D. Pfisterer, C. Buschmann, S. P. Fekete, and S. Fischer. Shawn: A new approach to simulating wireless sensor networks. *eprint arXiv:cs/0502003*, Feb. 2005.

[12] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. *ACM Conference on Embedded Networked Sensor Systems*, Nov. 2003.

[13] Mica2 sensor board. `http://www.xbow.com/`.

[14] MicaZ sensor board. `http://www.xbow.com/`.

[15] Mote hardware platform. `http://www.tinyos.net/scoop/special/hardware`.

[16] NS-2 network simulator. `http://www.isi.edu/nsnam/ns/`.

[17] E. Ould-Ahmed-Vall, G. F. Riley, B. S. Heck, and D. Reddy. Simulation of Large-Scale Sensor Networks Using GTSNetS. *In Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, 2005.

[18] S. Park, A. Savvides, , and M. B. Srivastava. SensorSim: a simulation framework for sensor networks. *ACM International workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, 2000.

[19] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras. ATEMU: A Fine-grained Sensor Network Simulator. *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[20] A. Pothen. Graph partitioning algorithms with applications to scientific computing. *Parallel Numerical Algorithms*, pages 323–368, 1997. Kluwer.

[21] Wireless Propagation Bibliography. `http://w3.antd.nist.gov/wctg/manet/wirelesspropagation_bibliog.html`.

[22] G. F. Riley. Large-scale network simulations with GTNetS. *In Proceedings of the 2003 Winter Simulation Conference*, 2003.

[23] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, winter:34–43, 1995.

[24] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. *Draft to be included in CRPC Parallel Computing Handbook, Morgan Kaufmann*, Sept. 2000.

[25] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[26] Simulavr: A simulator for the Amtel AVR processor family. `http://www.nongnu.org/simulavr`.

[27] Stargate: a platform X project. `http://platformx.sourceforge.net/`.

[28] S. Sundresh, W. Kim, and G. Agha. SENS: A Sensor, Environment and Network Simulator. *The IEEE 37th Annual Simulation Symposium*, 2004.

[29] Moteiv Corporation. Telos Sensor Network Module. `http://www.moteiv.com/`.

[30] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. *The Fourth International*

Symposium on Information Processing in Sensor Networks, Apr. 2005.

[31] F. A. Tobagi and L. Kleinrock. Packet switching in radio channels: Part II-The hidden terminal problem in carrier sense multiple-access and the busy-tone solution. *IEEE Transactions on Communications*, COM-23:1417–1433, 1975.

[32] Y. Wen, S. Gurun, N. Chohan, R. Wolski, and C. Krintz. Toward Full-System, Cycle-Accurate Simulation of Sensor Networks. Technical Report CS2005-12, University of California, Santa Barbara, 2005.

[33] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. *In Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003.

[34] G. Zhou, T. He, S. Krishnamurthy, and J. A. Stankovic. Impact of radio irregularity on wireless sensor networks. *In Proceedings of the 2nd international conference on Mobile systems, applications, and services (MobiSYS'04)*, 2004.