Journal of Signal Processing Systems 50, 115–136, 2008
© 2007 Springer Science + Business Media, LLC. Manufactured in The United States.
DOI: 10.1007/s11265-007-0113-2

Accurate and Scalable Simulation of Network of Heterogeneous Sensor Devices

YE WEN, SELIM GURUN, NAVRAJ CHOHAN, RICH WOLSKI AND CHANDRA KRINTZ
University of California, Santa Barbara, CA 93106, USA

Received: 14 February 2007; Revised: 30 May 2007; Accepted: 13 June 2007

Abstract. Simulation is an important tool to study and analyze sensor networks. Prior work in sensor network simulation focuses on homogeneous devices. In this paper, we present a system that performs scalable and accurate simulation of a network of heterogeneous sensor devices, including both Stargate intermediate level devices and mote devices. We study accuracy, performance, and scalability of our system. The results show that we can achieve accurate functional behavior for both standalone Stargate simulation and ensemble simulation of a Stargate and motes. For motes, we have less than 4.06% cycle count error for all benchmarks and for Stargate, we have less than 10% error for most benchmarks, and less than 12.5% error for all benchmarks. We also achieve less than 3.6% error for all benchmarks when simulating an ensemble of Stargate and motes. Our system is also more scalable than prior work. We can simulate 160 sensor nodes in real time speed and 2,048 sensor nodes with ten times slowdown on a 16-node cluster.

Keywords: simulation, sensor network, intermediate sensor node, scalability

1. Introduction

Sensor networks have emerged as a technology for transparently interconnecting our physical world with more powerful computational environments, and ultimately, global information systems. In a typical sensor network, computationally simple, low-power sensor elements take physical readings and may perform some processing of these readings before ultimately relaying them to more powerful computational elements. The need for non-intrusiveness motivates sensor design toward small, inexpensive, low-power sensor implementations that can be deployed in large numbers throughout the environment to be sensed. Because the sensor elements themselves are resource constrained, a sensor network may include a smaller number of more complex and general purpose computational elements that are capable of substantial in-network processing, contain greater storage capacity, and can act as intermediate "gateway" nodes between the network of sensor elements and more power-intensive network technologies. Such heterogeneous designs have been proven to be able to make a sensor network more computationally powerful and energy efficient [1]. It has seen its application in many existent deployments [2–4], and is an important research direction for future sensor network architecture [5].

Designing and investigating these ensemble systems, to date, has relied primarily on physical deployments and experimentation [6–10]. While the quality of the results from such efforts is excellent, the need to work with the physical systems directly imposes a substantial research impediment. The labor cost, equipment cost, space requirements, debugging complexity, etc., that characterize such an engineering-based approach, all limit the scope of the research that can be performed, and the number of researchers who can perform it.

One obvious possibility for widening the scope of what can be investigated is to employ simulation as a complement to experimentation with deployed systems. While several simulation efforts have focused on the sensing elements themselves [11–16], an approach that combines sensor simulation with simulations of the other "heavier" devices as an ensemble—and does so with an acceptable level of accuracy—is necessary to make simulation a viable option.

In this paper, we investigate a system that simulates a complete sensor network that includes both simple (base level) sensor devices and more powerful intermediate sensor devices. Our system has two major components, SimMote and SimGate, for simulating the two classes of devices, respectively. The former, SimGate, performs cycle-accurate full-system simulation of mote sensors [17]. SimMote is similar to previous simulators [13, 14, 16] that simulate simple sensor devices, yet it enables better performance and more extensive hardware support. The distinctive feature of our system is SimGate, a full-system simulator for the Intel Stargate device [18] (distributed by Crossbow Inc.). The Stargate is intended to function as a general purpose processing, storage, and network gateway element in a sensor network deployment. Our goal is to provide both functional correctness and accurate cycle estimation. We refer to the latter as *cycle-close* [19].

SimGate is unique in that it supports cycle-close, functional simulation of the entire Stargate device as opposed the processor [20] or power consumption [21, 22] alone. SimGate captures the behavior of most Stargate components including the processor, memory hierarchy, communications (serial and radio), and peripherals. As the result, SimGate boots and runs the Familiar Linux operating system and any program binary that executes over it, *without modification*.

Our system is also unique in that it is able to simulate the ensemble of base level sensor devices and intermediate sensor devices, i.e. a complete deployment of typical sensor network. We implement a multi-simulation framework that coordinates the simulation of both SimMote and Sim Gate, and emulates the communications, including serial and radio, among the simulated devices. To make the ensemble simulation scalable, the multi-simulation framework is also able to distribute individual device simulations to distributed memory systems, such as a computing cluster.

We evaluate both the accuracy and the performance of our system. For the accuracy of each simulator, we compare the simulated clock cycles to measured clock cycles using a wide range of stressmarks and community benchmarks. We also present results for similar experiments in which the Stargate and mote inter-operate via a serial interface. Finally, we examine a multi-device ensemble consisting of a Stargate node, a serially connected mote, and a third mote that communicates only via simulated radio. We compare our simulated results to measurements that we gather from physical Stargate and motes. Our results indicate that we are able to accurately simulate the full system of a Stargate node with a *maximum error* of 12.4% across all benchmarks we test. We also find that, on average, simulation at this level of accuracy imposes a slowdown of 58× over real-time device execution and that a slowdown of 20× can be achieved if only a functional simulation (i.e. without accurate cycle counts) is required. For motes, the simulation accuracy is within 5% error. And the best simulation performance is approximately nine times that of real time speed.

We also study the scalability of simulating a complete sensor network that contains multiple Stargates and a large number of motes. Due to the large performance gap between SimGate and SimMote, SimGate is always the performance bottleneck. To investigate the scalability of SimMote, we evaluate the ensemble simulation of a multi-mote network. We show that our system can simulate 2048 motes on a 16-node cluster at one tenth the speed of real time. As a result, we believe this work demonstrates the potential of multi-device sensor network simulation as a research-enabling technology.

In the next section, we overview the design and implementation of our simulation system. In Section 3, we describe our experimental setup and measurement methodology. We then detail the accuracy and performance of our system in Section 4. In Sections 5 and 6, we present related work and conclude with some observations and our plans for future work, respectively.

2. Simulation System Design and Implementation

Simulation is a potentially important tool for sensor network system and application development. The focus of most prior work in system simulation has been on high-end, general-purpose, wall-powered devices [23–25], processor/power simulation [21, 22, 26], or on the base level sensor devices themselves [11–16]. However, to our knowledge, no extant approach to sensor network simulation enables full-system simulation of a key sensor network component: the intermediate "gateway" node. Moreover, no simulation system facilitates ensemble simulation of heterogeneous sensor devices. The goal of our work is to investigate, implement, and evaluate such mechanisms.

Intermediate nodes are resource-constrained, battery-powered, devices that provide a bridge between base level sensor nodes (which we refer to as motes after the popular Berkeley Mote implementation [27]) and more powerful, wall-powered, computational environments. Intermediate nodes are commonly responsible for sensor device control and in-network processing [1] of sensor data: receiving, processing, assimilating, forwarding, etc. These nodes reduce the power consumption of the system by reducing the communication distance from the motes to a powered device, and by coalescing and compressing the data that is forwarded to higher levels of the hierarchy. Intermediate nodes commonly have longer battery life and significantly more powerful computation and communication capabilities than the motes. A popular example of an intermediate node implementation is the Intel Stargate [18].

To simulate intermediate nodes, we developed a software system, called SimGate, that virtualizes the Stargate device. SimGate emulates the complete functionality of the Stargate and provides cycle-close simulation of the Stargate's Intel XScale processor pipeline [28]. SimGate is completely transparent to the above software layers—i.e., the system boots and executes the popular embedded OS, Familiar Linux [29] and any program that executes over it, without modification.

We also developed SimMote to simulate the base level sensor devices, i.e. motes. As other mote simulators [13, 14, 16], SimMote performs cycle-accurate full-system simulation, except that SimMote provides better simulation performance and more extended hardware support. Unmodified TinyOS [30] binaries can be executed directly in SimMote. We use SimMote to couple with SimGate for the ensemble simulation of a network of heterogeneous sensor devices, which includes both motes and Stargates. To do that, we designed a multi-simulation framework that coordinates the individual simulation of motes or Stargates and emulates the radio or serial communication among them.

Scalability becomes a serious issue when the size of sensor network to be simulated grows. Simulating a few Stargates and hundreds of motes on one machine is definitely unacceptable since the simulation speed will crawl. To attack this problem, the multi-simulation framework is designed to distribute device simulations among networked computers so that an unbounded computing resource can be used to scale the ensemble simulation.

We next introduce the design and implementation of SimGate. We then discuss the multi-simulation framework that combines SimGate and SimMote to enable scalable sensor network simulation.

2.1. SimGate Design and Implementation

SimGate provides full-system simulation of the Stargate intermediate sensor node. The Stargate is a single-board, embedded system (designed by Intel Research) that comprises a 400 MHz Intel XScale processor, an Intel SA1111 companion chip for I/O, Intel StrataFlash, SDRAM, PCMCIA/CF slots, and connector for a mote [18]. In *situ*, it communicates with motes in a sensor network via a mote that is physically connected to it via this connector.

The goal of our design and implementation of SimGate is to effectively trade-off simulator overhead for accuracy while enabling transparent, full-system simulation. To this end, we combine a number of different approaches to performance estimation of device components within a single system, including cycle-level simulation (which can be disabled when only functional simulation is needed) of some components and benchmark-based timing. Using cycle-level simulation, as we will show, we are able to achieve accurate system-level cycle counts as compared to a real device. By turning cycle-level simulation off, we can reduce simulation time and yet enable correct functional device behavior. In both cases, the same OS installation and application code runs without change.

We simulate the following features of the Stargate device:

- ARM v5TE instruction set without Thumb support and with XScale DSP instructions
- XScale pipeline simulation, including the 32-entry TLBs, 128-entry BTB, 32 KB caches and 8-entry fill/write buffers
- PXA255 processor, including MMU (co-processor), GPIO, interrupt controller, real time clock, OS timer, and memory controller
- Serial device (UART) that communicates with the attached mote
- SA1111 StrongARM companion chip
- 64 MB SDRAM chip
- 32 MB Intel StrataFlash chip
- Orinoco wireless LAN PC card including the PCMCIA interface

We found that simulation of this set of devices was sufficient to enable us to successfully boot the Linux kernel 2.4.19 and to execute a wide range of benchmarks.

2.1.1. Instruction Interpretation To implement the instruction set, we use a simple interpreter to execute the instruction flow using a large switch statement as is done in SimpleScalar [26]. The most complex part of the CPU core simulation is the memory management unit (MMU). The MMU is used constantly during program execution since each memory access requires an address translation. When cycle-level simulation is not required, we turn off simulation of the individual MMU components including the TLB, BTB, I/D caches, and fill/write buffers, to improve functional simulation performance. The cycle-level simulation of these components do not affect the correctness of functional program execution but they do, however, impose a large simulation cost. To further improve the address translation speed, we implemented an address lookup cache (soft TLB) for both instruction and data addresses. This soft TLB increases functional simulation time by 10% on average.

2.1.2. Pipeline Simulation To estimate cycle counts accurately, we simulate the Intel XScale pipeline. The Intel XScale core employs a seven or eight stage (depending on the instruction flow), single-issue pipeline. There are actually three pipeline branches that execute in parallel after the execution stage. As a result multiplication and memory access can happen concurrently and results may be written back to memory out of order. Due to the complexity and proprietary design of Intel XScale pipeline, it is hard to achieve strict cycle accuracy, which is also not necessary for most sensor network simulations. We approximate the internal functions of the pipeline and manage to achieve cycle-closeness [19] that is accurate enough for common simulation use. Since we were unable to obtain publically available documentation from Intel on the pipeline logic, we based our implementation on that from the XScale pipeline simulation implemented in XTREM power simulator [22]. We used this implementation as a reference and extended and evolved it using benchmark measurements from a real Stargate device (since the Stargate uses a slightly different version of XScale processor that that implemented within XTREM). Most MMU components (TLB, BTB, caches and buffers) are also implemented as part of the pipeline simulation. To account for cache and TLB miss penalties, the simulator uses estimates that we obtained via measurements from hand-coded benchmark execution on a real device.

As we mentioned above, we are able to toggle the type of simulation between cycle-close and functional. By doing so, we trade off the ability to collect cycle-level behavior with simulation speed; both simulations however, are functionally correct. We implemented a mechanism with which we can turn on/off pipeline simulation dynamically. As a result, we can also combine functional simulation with pipeline simulation to improve simulator startup time. For example, we turn off pipeline simulation during boot of the operating system and to fast-forward the simulator to a point of interest (at which we wish to investigate more accurate, cycle-level behavior).

We toggle cycle-level (pipeline) simulation through the use of a special virtual hardware interface that we integrated into the XScale hardware performance monitor (HPM) interface [28]. When any software activates and terminates HPMs, the simulator turns pipeline simulation on and off, respectively. We selected this implementation since it enables us to use the same interface to drive experimentation and measurement of programs executed with either unsimulated (real device) or simulated configurations easily.

2.1.3. Peripheral and I/O Simulation The most important peripheral and I/O devices we simulated are the Flash chip and the Orinoco PCMCIA wireless card. The Flash chip is controlled by memory mapped I/O registers. The simulator sends and receives the commands and data through these registers. In the Flash chip, a state machine controls the sequence of operations. We simulate both the interface and the internal state machine according to a Verilog model of the Flash chip from Intel [31].

The simulation of the wireless card consists of two parts: the PCMCIA interface and the wireless card interface. We have implemented the publically available PCMCIA interface in our simulator. However, we have been unable to obtain similar documentation on the interface and internals of the wireless card. To overcome this limitation, we simulate the card by mimicking card interface exposed in its Linux driver source code and using the parameters dumped from the real card. As a result, we can connect the card simulator to a Linux TunTap interface so that our simulator successfully builds a TCP/IP connection between a program executing on a real device and one that we are simulating. However, we have not yet simulated the 802.11 b radio model used by the Stargate since it is seldom used in a typical sensor network deployment.

We do not maintain cycle accuracy of the I/O devices (whether cycle-accurate simulation is turned on or off) due to the device-specific complexities and widely ranging functionality. Instead we employ a similar benchmarking approach to the one discussed previously to estimate the performance of I/O devices. That is, we collect the timing behavior using a range of hand-coded benchmark experiments, and use this data to advance the clock within the simulator. As an example, consider the buffer programming of Flash chip. The data is first written into an on-chip buffer. Then the program command is issued to the chip. The chip starts to program the memory after setting the status bit to busy. When the job is done, the status bit is cleared. To model this process, we collected access times for a range of buffer sizes and embedded the values in the simulator. During simulation, when the programming command is issued, the simulator dispatches an asynchronous event with the corresponding time interval. When the event is reached, the status bit is cleared just as it is for a real device.

2.2. SimMote Design and Implementation

We developed SimMote to couple it with SimGate for the ensemble simulation of a network of both Stargates and motes. SimMote simulates the Mica2/MicaZ motes with cycle-accuracy. Mica2/MicaZ features the 8 MHz Atmel ATmega128 microcontroller (simple 16-bit RISC ISA), on-board Flash memory and a 900 MHz radio. Compared to SimGate, the SimMote is much easier given the significantly simpler hardware and software design. SimMote currently supports the following features:

– AVR instruction set
– Most on-chip functions: program memory, RAM, EEPROM, timers, UARTs, SPI (Serial Peripheral Interface), ADC (Analog/Digital Converter), Watch Dog Timer and fuse bits (for boot loader and self-programming)
– 512 KB on-board flash
– Serial ID chip
– CC1000 (for Mica2) and CC2420 (for MicaZ) radio chip
– LEDs and sensor boards

We are able to achieve cycle accuracy of AVR ISA for most instructions since the instruction set specifies fixed cycle numbers. We use these timings within SimMote to forward the CPU clock. In a way analogous to SimGate, SimMote is able to boot the TinyOS mote operating system and to execute existing mote programs.

2.2.1. Radio Model The 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 [32]. 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 channel 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 [33]. 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 on a statistical description of channel loss, often derived from measurement trace data [34–36]. 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. [34] 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 [34]. 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.

2.2.2. Power and Battery Models At present, perhaps the most active area of sensor network simulation research focuses on modeling power dissipation. Sensor network simulators are required to provide accurate energy consumption estimation for any reasonable study based on simulation. A number of power models for sensor network devices have been proposed and investigated in the literature [37–39]. These models are typically based on the measurements obtained by using benchmarks to exercise the sensor device in various modes yielding different levels of fidelity. In this work, we incorporate one such model [37] in our simulator.

We also provide a simple linear battery model. Several battery models have been proposed in the literature [40–42]. Linear model is the simplest, again representing the ideal case in debugging and "back-of-the-envelope" settings. Moreover, in fast, lower-fidelity simulations of "steady-state", a linear model is often preferred since the middle of the discharge curve is often close to linear [43].

Note that, we use a similar approach to provide power and battery models for SimGate.

2.3. Scalable Ensemble Simulation

A typical sensor network contains a few Stargate devices as "gateway" nodes for relaying aggregated information or "super" nodes for efficient in-network processing [1], and massive motes as base level sensing devices. The Stargate device is normally not able to communicate directly with mote devices. Instead, a mote is connected to the Stargate via serial port to be used as the proxy or the network interface to other motes.

As an analogy of the physical sensor network, an ensemble simulation is composed by a set of individual device simulations, including both SimGate and SimMote, each running in a separate operating system thread to exploit the parallelism of the underlying hardware. Due to the resource limitation of a single computer, it is not feasible to execute all the simulation threads on one machine. As we will see in the experiment result, simulating one Stargate device exclusively on one machine already has a slowdown of 20 to 58×, not to mention simulating a few Stargates and massive number of motes. The scalability of the ensemble simulation thus determines its usability. To attack this problem, we design our multi-simulation framework to distribute the device simulations among multiple network connected computers, such as a computing cluster. In this way, a potentially unbounded computing resource can be used to scale the ensemble simulation.

Individual device simulations are not independent of each other. The communication, either by radio or by serial, builds time causality among the devices. As an example, a radio packet sent at cycle 1,000 on the sender has to be received at cycle 1,000 on the receiver. Thus the simulation threads have to be synchronized. Prior works [13, 16] use lock-step synchronization, in which all threads are stopped periodically to correctly transmit a radio byte. In a distributed computing environment, relatively large and variable network latencies make it infeasible. In a typical Ethernet, the network latency is measured in milliseconds while a desktop PC can emulate one instruction in much less than one microsecond. If lock-step synchronization is used, the ensemble simulation speed is determined by the all-to-all network communication latency.

2.3.1. Communication Model Derived from our work in [44], our approach is based on an abstraction of communication and a simple yet efficient synchronization protocol. There are two types of communication: the serial communication between a Stargate and its mote interface and the radio communication among motes. The serial communication is abstracted as a duplex FIFO channel shared by two devices and corresponding read/write operations. The write operation enqueues a time-stamped data byte in the channel. The channel buffers the sent byte. Whenever a read operation is performed by the other end, the channel is checked and the earliest data byte is dequeued. The radio communication is abstracted as follows. For each mote, a radio media is shared among itself and its radio neighbors (i.e. nodes within its maximal radio range). There are also a read operation and a write operation for the media. The read operation represents the radio receiving and channel sampling (RSSI [45]). The write operation represents the radio sending. A mote sends by "writing" a time-stamped data byte to each neighbor's radio media. Whenever a mote performs a read operation, its own media is checked and a data byte is assembled (according to the radio model) from the data bytes coincident with current time.

2.3.2. Synchronization Protocol Based on this abstraction, we designed a synchronization protocol that regulates communications among devices. Each device simulation thread maintains a local clock to keep the simulation progress. Obviously, for any two threads, if there is no communication, their executions are independent and there is no need to synchronize the clocks. Whenever a communication occurs, the causal relationship that exists between the two threads has to be rectified so that the data bytes are received in order. Since a sent data byte is time-stamped and buffered, the sender takes no responsibility to maintain causality and can advance freely. It is the receiver to keep track of all potential sendings, adjust its own progress accordingly and extract ordered data correctly. The protocol then can be described as follows:

1. A node that reads must wait for all its neighbors (or the other side of the channel) to catch up with its current clock time (to make sure it will receive all the data it potentially should receive);
2. All nodes must periodically broadcast their clock updates to neighbors (or the other side of the channel; to notify others of their progresses);
3. Before any wait, a node must first send its clock update (to avoid loop waiting);
4. Data byte is always sent with a clock update at the end of its last bit transmission (to avoid broken bytes).

This protocol only involves periodical message passing for clock updates and data bytes. It is easy to implement in a distributed memory environment with much lower cost than lock-step synchronization. For threads that share memory (e.g. running on the same PC), this can be implemented as a shared variable protected by thread locks. For threads that have separate memory (e.g. remotely connected through network), standard message passing mechanism, such as TCP, can be employed.

2.3.3. Node Partition Even with this relatively low cost synchronization protocol, message passing through network still incurs a considerable overhead compared to the simulation speed. To achieve maximal simulation performance, the remote network synchronization has to be minimized. Furthermore, the computation load of simulations have to be balanced to eliminate performance bottleneck. How to partition the ensemble simulation according to the available computing resources becomes the determining factor of performance optimization.

The simulation partitioning problem can be modeled as a "classic" graph partition problem that is well studied in parallel computing area [46–48]. 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) [46]. In our problem, the simulation threads are the graph vertices and the serial or radio communication relationships are the edges. The computation cost of simulation is the vertex weight and the communication cost is the edge weight.

A general graph partition problem is hard to solve. At this stage, we simplify the problem by considering several factors. First, the typical distributed parallel computing resource is a cluster, which is composed by identical machines with 1–2 processors on each. Second, the huge simulation performance difference between SimGate and SimMote implies that in most situations, we have to allocate a single machine to simulate a Stargate exclusively. Finally, because serial communication is much faster than radio communication and the serial hardware constantly reads shared channel, a Stargate and its mote interface are more tightly coupled than motes do. Thus it is preferable to simulate them with shared memory, i.e. on the same machine. The partition problem is then reduced to evenly distribute a set of motes to a set of identical machines with the same communication cost. We employ a famous graph partitioning package, Chaco [49], to solve the problem. We will explore the general partition problem in our future work.

3. Experimental Method

To evaluate and analyze the performance and accuracy of our system, we performed a number of experiments using the SimGate and SimMote alone as well as with SimGate-SimMote ensembles. To evaluate the latter, we implemented two scenarios: (1) A Mote attached to a Stargate through the serial expansion bus (2) A secondary Mote communicating with the first via simulated radio. In scenario (2), we located the motes such that their antennas are in physical contact to minimize errors caused by interference over the radio channel. At present, we do not model interference as part of the simple radio model that we implement, however are currently working on robust and accurate radio models as part of future work.

Scenario (1) represents the use of the Stargate as a gateway. Currently, the Stargate design does not include a radio interface that is compatible with Motes. Instead, the Stargate implements an expansion bus that allows a Mote to be physically attached to it. The communication between the attached Mote and Stargate uses one of the four UART channels; in other words, even though a Stargate gateway functions as 2a single machine, it is in fact two completely independent processors that are connected through a serial link. Thus scenario (2) represents a sensor network that has one Mote and one gateway (a Stargate with a Mote attached).

We also evaluate the scalability of our system under simplified situation as mentioned in the previous section where coupled Stargate and mote pair is simulated exclusively on a single machine and other motes are distributed among remaining machines in a cluster. In such situation, we actually only care about how performance scales with different number of motes and machines. As a result, we do not run Stargate simulation in scalability experiments.

3.1. Benchmarks

For stand-alone SimGate evaluation, we employed our hand-coded stressmarks and benchmarks from both the MiBench [50] and the Mediabench [51]. In Table 1 we present our stressmarks to measure the simulation performance.

The stressmarks is hand-coded to test the specific feature of the processor. The *DCacheReadHitDep* has a data working set that fits in the cache and the LD instructions have data dependency. The *DCacheReadHit* is similar but without data depen-

Table 1. Stressmarks that we used in the evaluation of SimGate.

Benchmark	Executables	Description
DCacheReadHitDep	dcachehit_r	Data cache read 100% hit w/ data dependency
DCacheReadHit	dcachehit_nd_r	Data cache read 100% hit w/o data dependency
DCacheReadMiss	dcachemiss_r	Data cache read 100% miss
DCacheWrite	dcache_w	Data cache write
BTB	btb	BTB test program
LUDecomp	ludcmp_heap	LU Decomposition algorithm

Table 2. MiBench benchmarks that we used in the evaluation of SimGate.

Benchmark	Executables	Description
BitCount	bitcnts	Bit manipulation of the processor
Dijkstra	dijkstra	An $O(n^2)$ algorithm to find shortest path in a graph
FFT	fft	Fast Fourier Transformation
SHA	sha	Secure hash program
StringSearch	search	A text search program
Mesa	mipmap	3D rendering program

dency. The *DCacheReadMiss* has a larger data set than cache size and produces 100% cache misses. For cache write, since the Linux running on the Stargate set the MMU to apply "write-through" policy, there is no difference between cache write hit and cache write miss. So we use a single *DCacheWrite* to test data cache write. We also have a *BTB* stressmark to exercise the BTB simulation. The *LUDecomp* is a stressmark to test the overall processor simulation.

In Table 2, we give the description of the benchmarks we choose from MiBench. These benchmarks cover the operations from simple bit manipulation to complex 3D rendering and to heavy floating point computation. For even more complex and realistic benchmarks, we use the Mediabench.

Mediabench includes a rich set of programs that are heavily used in multimedia and office type of applications. In Table 3, we describe the benchmarks that we use. We eliminate three benchmarks due to the constraints of the underlying platform: *Epic* does not run on the real Stargate platform (due to memory constraints), *MPEG2* requires too many hours to execute due to the execution of floating point operations, and *Ghostscript* does not fit in the available Stargate Flash memory (25 MBytes). We execute all remaining benchmarks from the RAM drive.

To evaluate the accuracy of SimMote simulator components, we choose a set of five benchmarks. Each benchmark contains data that is measured only during the execution of one particular unit. We describe the benchmarks and the components in Table 4. These benchmarks are stand-alone applications (i.e. the measurements were independent of Simmote simulator). Note that the floating points test evaluates the software implementation of floating point arithmetic.

To evaluate ensemble simulation, we employ open-source applications as well as hand-coded programs. We describe the applications in Table 5. Column 3 shows the functional units of the Motes that are heavily utilized during the execution of various benchmarks. In choosing benchmarks, we attempt to exercise the full device, and cover the major functions of a Mote: communication, sensing and logging. The difference between this and the previous set of benchmarks (the ones given in Table 4) is that these benchmarks show the behavior of the application as perceived by the SimGate (we will detail measurement methodology shortly) and the previous benchmarks show the behavior of that particular unit only (compared using external test equipment).

Each ensemble benchmark has a *Long* and *Short* form. The Short benchmarks exercise only the Stargate and serially-attached Mote communicating via the UART interface. The Long benchmarks exercise Stargate and the attached Mote, operating as a gateway or controller, and a remote Mote communicating via radio. Moreover, each of these applications takes the form of a remote procedure call (RPC). When the program on the Stargate sends a query to the Mote, it blocks until the receiver completes the appropriate execution and returns. The

Table 3. MediaBench benchmarks that we used in the evaluation of SimGate.

Benchmark	Executables	Description
adpcm	adpcmdecode/adpcmencode	Adaptive differential pulse code modulation for audio coding
g721	g721decode/g721encode	CCITT voice compression
gsm	gsmencode/gsmdecode	European standard for speed coding
jpeg	jpegencode/jpegdecode	Lossy compression for still images

Table 4. Benchmarks that we used to evaluate the components of SimMote.

Benchmark	Description	Functional Unit
ALU unit	Computation and Logic operations	Arithmetic/Logic Unit
Radio	Network Packet Transfer time	Radio Model
Floating PTS	Floating point operations	Arithmetic/Logic unit
Flash Read	Reads log from Flash	Secondary Flash
Flash Write	Write data to Flash	Secondary Flash

The third column shows the functional units that were evaluated during the test.

multi benchmark also tests concurrent computation by running parallel computations of ad-hoc positioning system (APS) [52] on both Mote and Stargate. This test is useful to evaluate the performance of simulating coordinated computation on Mote and Stargate.

For our scalability experiment, we use sensor network application *CntToRfm*, which is used as the touchstone in previous scalability studies [11, 16]. *CntToRfm* periodically sends out radio packets and keeps the radio channel busy. Although it does not receive packets, the radio chip still switches to receiving mode when it is not transmitting. So it does in effect exercise all radio activities.

3.2. Experimental Apparatus

We execute TinyOS v1.1 on the Motes (and SimMote) and a variation of Familiar Linux v0.5.1 on the Stargate (and SimGate). For the stand-alone Stargate applications (i.e. Mediabench), we measured the CPU clock cycles and instruction count using the XScale hardware performance monitors (HPM). The HPM system can monitor 3 events (CPU clock cycles and two events) concurrently. We read the performance monitors using a kernel module that we developed.

For performance and accuracy experiments, we ran our simulators on a dedicated Linux (kernel ver 2.6.8) machine. The machine has a 64 bit AMD Opteron CPU running at 2.4 GHz and 4 GB of memory. To measure wall clock execution time of each benchmark, we modified the simulator. Each time the performance monitoring registers of the simulated machine (i.e. Stargate) are accessed, the simulator reads the real (wall-clock) time from the host system (which is synchronized using NTP), and computes and logs the delta (time since previous access).

We *wrapped* each simulated application using a small program: The wrapper reads the HPMs immediately before and after the execution of simulated program. This enables us to collect both wall clock time and simulator statistics (number of instructions executed, number of clock cycles, and many other system events supported by XScale architecture).

We found measuring real Mote hardware challenging since the Atmel CPU on the Mote does not provide any mechanisms for performance monitoring features. To enable our measurements (and hence validation of the correctness, accuracy, and performance of SimMote), we measured the CPU clock cycles of the Mote and its executing software using high-precision external instrument. To collect data accurately, we used the CPU output register PORTC

Table 5. Benchmarks that we used to evaluate the ensemble simulation of SimGate and SimMote.

Benchmark	Description	Functional Unit
Ping	Echoes network packet back to sender	Network interface
Sense	Processes a sensor read query	Analog/Digital converter
APS [52]	Ad hoc positioning system	Arithmetic/Logic unit
Log	Reads log from Flash	Secondary Flash & UART
Multi	Parallel computations on both devices	Arithmetic/Logic unit

The third column shows the functional units that are exercised most heavily during benchmark execution.

on the Mote. PORTC is directly connected to pin 51 on the expansion bus. When we wanted to initiate a measurement, we raised the voltage on the pin by writing a 1 to this register. When we wanted to stop measurement we disabled pin by writing a 0. The overhead of accessing this register is one clock cycle.

To time Mote execution, we connected an Agilent 54621A Oscilloscope (accurate up to 10 ns) to the output pin. We configured the oscilloscope to monitor the pulse width (i.e. the time between raising and lowering a signal), and recorded the measurements. We then converted timing measurements to clock cycles by multiplying it by the Mote clock speed (7.3728 MHz). We were not able to collect the instruction count, as there is no way of accessing this information through the expansion bus.

To evaluate and compare the SimMote simulator with our timing and instruction cycle measurements (which we described in previous paragraph), we instrumented the implementation of Mote's PORTC register in the simulator. Writing a 1 to this register enables an internal instruction cycle counter at the simulator. By comparing the two sets of numbers that we collected from the simulator and the oscilloscope, we were able to determine the accuracy of the simulator with a very high confidence.

As for the scalability experiment, we perform distributed ensemble simulation on a 16-node dual-processor 3.2 GHz Intel Xeon cluster with gigabit Ethernet.

4. Results

We detail the accuracy of SimGate by comparing it to the Stargate in terms of the number of cycles required to execute the benchmarks described in the previous section. In the first set of comparisons, we make 20 identical runs of each benchmark on both SimGate and Stargate and compare the average number of cycles required per benchmark.

Tables 6, 7 and 8 give the cycle accuracy result of the stressmarks, MiBench and Mediabench, respectively, for SimGate. All tables use the following format. The first column shows the name of the benchmark, the second column (μ_{meas}) shows the average number of cycles measured on the Stargate hardware, the third column ($\mu_{simulated}$) presents the cycles reported by SimGate and the fourth column shows the difference. In the fifth column, we report the error percentage ($(\mu_{meas} - \mu_{simulated})/\mu_{meas}$) which is difference between the average of the measured cycle counts and the average of those generated by the simulator. We also compute the 95% confidence interval for the error percentage using a Student *t* distribution [53] with 19 degrees of freedom to model the difference of the averages (marked as ± confidence bound in the table).

Note that the error percentage and confidence interval also indicate whether we should reject the null hypothesis of equivalence in a two-sided hypothesis test at 95% confidence. If the "margin for error" (confidence interval) spans *0%* (i.e. the margin is greater than the error percentage itself), we fail to reject the null hypothesis of equivalence and hence cannot determine whether the observed difference in averages is due to random variation or not. In this experiment, however, the confidence intervals are all quite narrow indicating the error percentage we observe for each benchmark is statistically significant at the 95% confidence level. There are

Table 6. Average cycle counts for measurements and simulations of MediaBench benchmarks, 95% confidence interval on the difference of the means, fraction of average measurement that interval constitutes.

Benchmark	μ_{meas}	$\mu_{simulated}$	μ_{meas} - $\mu_{simulated}$	% error ± 95% conf. bound
adpcmdecode	3.367E + 07	3.069E + 07	2.980E + 06	8.9 ± 0.28
adpcmencode	3.068E + 07	2.766E + 07	3.014E + 06	9.8 ± 0.36
g721decode	6.272E + 08	5.735E + 08	5.368E + 07	8.6 ± 0.17
g721encode	6.527E + 08	6.006E + 08	5.213E + 07	7.9 ± 0.44
gsmdecode	1.526E + 08	1.420E + 08	1.061E + 07	7.0 ± 0.57
gsmencode	4.335E + 08	3.995E + 08	3.401E + 07	7.8 ± 0.09
jpegdecode	2.554E + 07	2.235E + 07	3.191E + 06	12.5 ± 1.16
jpegencode	5.412E + 07	4.731E + 07	6.813E + 06	12.5 ± 0.41

Table 7. Average cycle counts for measurements and simulations of stressmarks, 95% confidence interval on the difference of the means, fraction of average measurement that interval constitutes.

Benchmark	μ_{meas}	$\mu_{simulated}$	μ_{meas} - $\mu_{simulated}$	% error ± 95% conf. bound
DCacheReadHitDep	1.606E + 07	1.604E + 07	2.263E + 04	0.14 ± 0.10
DCacheReadHit	5.852E + 06	5.863E + 06	−1.118E + 04	0.19 ± 0.22
DCacheReadMiss	1.680E + 09	1.694E + 09	−1.419E + 07	0.84 ± 0.01
DCacheWrite	1.606E + 07	1.605E + 07	1.312E + 04	0.08 ± 0.10
BTB	6.142E + 07	6.547E + 07	−4.044E + 06	6.58 ± 0.07
LUDecomp	1.207E + 08	1.196E + 08	1.044E + 06	0.87 ± 0.09

three benchmarks whose confidence interval spans 0%. However, the difference is so close that with 95% confidence you can't determine if it is a true difference or random noise.

We observe that the accuracy of SimGate for this set of benchmarks is acceptable as a full-system simulation. While error percentages below 5% have been achieved for individual system components [22, 54], because we simulate the full device (including all parts of the memory hierarchy and the interrupt structure) and run both an operating system and application on it, we expect to introduce additional error. That the maximum error is no more than 13.5% (with 95% confidence) and most of the errors are below 10%, is surprising and is an indication that the simulation is of high quality.

Table 9 gives the cycle accuracy result of timing benchmarks for SimMote. The format of the table is same as Table 6. The data indicates that the accuracy of our Mote simulator is similar to that of SimGate. For floating point programs as well as the radio and flash write benchmarks, the error rate is insignificant since the error margin is greater than error percentage itself. For the ALU and flash read benchmarks, the confidence intervals are quite narrow and the error rate is very small.

4.1. Coupled SimGate and SimMote Simulations

To gauge how well SimGate will work in a simulation of a heterogeneous sensor network, we examine its cycle-count accuracy when it is used in conjunction with one or two SimMotes (as described in Section 3). Table 10 shows the cycle count results for the benchmarks that exercise the Stargate device and the Mote that is connected to it via a serial interface (scenario 1). As noted previously, the Stargate device does not support a radio device capable of communicating directly with Motes in a sensor network. Instead, it uses Mote directly connected to it via a serial interface as a network interface peripheral. These benchmarks are intended to exercise this interaction in a representative way.

The format of Table 10 is the same as that described for Table 6 in the previous subsection. Again, the sample size used to calculate each average is 20 and we compute a 95% confidence interval on the error percentage using a t distribution with 19 degrees of freedom.

Again, the accuracy of the coupled simulation is reasonable for two communicating independent full-device simulations. Note that while the error percentages appear significantly lower than for the SimGate simulation alone, the confidence intervals are also

Table 8. Average cycle counts for measurements and simulations of MiBench benchmarks, 95% confidence interval on the difference of the means, fraction of average measurement that interval constitutes.

Benchmark	μ_{meas}	$\mu_{simulated}$	μ_{meas} - $\mu_{simulated}$	% error ± 95% conf. bound
BitCount	3.648E + 07	3.589E + 07	5.959E + 05	1.63 ± 0.09
Dijkstra	1.867E + 08	1.731E + 08	1.360E + 07	7.29 ± 0.07
FFT	9.955E + 07	9.332E + 07	6.225E + 06	6.25 ± 2.39
Mesa	2.105E + 08	1.932E + 08	1.730E + 07	8.22 ± 4.79
SHA	6.391E + 07	6.208E + 07	1.834E + 06	2.87 ± 4.90
StringSearch	3.249E + 08	3.315E + 08	−6.574E + 06	2.02 ± 0.13

Table 9. Average cycle counts for measurements and simulations of Mote benchmarks, 95% confidence interval on the difference of the means, fraction of average measurement that interval constitutes.

Benchmark	μ_{meas}	$\mu_{simulated}$	μ_{meas} - $\mu_{simulated}$	% error ± 95% conf. bound
ALU unit	2.884E + 06	2.954E + 06	−7.031E + 04	2.44 ± 0.12
Radio	4.672E + 05	4.862E + 05	−1.898E + 04	4.06 ± 26.50
Floating Pts	3.498E + 06	3.499E + 06	−6.357E + 02	0.02 ± 0.10
Flash Read	2.884E + 06	2.954E + 06	−7.031E + 04	2.44 ± 0.12
Flash Write	2.521E + 03	2.434E + 03	8.688E + 01	3.44 ± 27.90

significantly wider. Thus, based on error percentage alone it may appear that the coupled simulations are more accurate. However, there is more relative variation (as we might expect) in the coupled case. As a result, it is the error range, and not the specific error value, that is significant in this case.

For example, consider the results for the *PingShort* benchmark shown in row 1 of Table 10. From the data, it is not possible to determine that the difference between the measured average and simulated average is statistically significant at the 95% confidence level (since the error range spans *0%*). However, there is enough variation in both measurements and simulation to make the difference indistinguishable from random variation across an interval that is ±6.6% centered on the observed average.

The *PingShort* benchmark exhibits the widest variation, as indicated by the error range. For the *SenseShort* benchmark the difference in observed average is, once again, statistically undetectable with 95% confidence, but the error range is smaller. In the remaining three cases, there is a statistically significant difference, but both the error percentages and the confidence bounds on those percentages are remarkably small. From this data, we conclude that cycle-counts taken from SimGate when coupled to SimMote via a serial interface, while introducing additional variation, are still reasonably accurate.

The final set of accuracy results we present is for benchmarks that couple SimGate with a SimMote via its serial interface that is then used to communicate with a second SimMote via the radio interface (scenario 2). As described previously, we do not yet know of a Mote radio communication simulation that is accurate enough not to overshadow the accuracy (or lack thereof) of SimGate. Thus, these experiments reflect a configuration in which the antenna of the two Motes are in physical contact. It is our experience that this configuration eliminates much of the variation resulting from radio communication.

Table 11 depicts these results using the same format as the in the previous two tables. Similar to the results for *PingShort* and *SenseShort* in Table 10, the additional variation introduced by the second Mote and the radio communication makes the difference between observed and simulated averages indistinguishable from random variation at a 95% confidence level. However, the 95% confidence intervals on the error percentage are, once again, similar in magnitude to the error percentages in Tables 6 and 10 for the cases where the averages are significantly different.

Table 10. Average cycle counts for measurements and simulations of benchmarks coupling SimGate with simulated Mote via serial link, error percentage, 95% confidence range for error.

Benchmark	μ_{meas}	$\mu_{simulated}$	μ_{meas} - $\mu_{simulated}$	% error ± 95% conf. bound
PingShort	9.414299E + 07	9.592680E + 07	−1.783813E + 06	1.9 ± 6.6
SenseShort	2.040608E + 08	2.051871E + 08	−1.126267E + 06	0.6 ± 1.1
APSShort	1.997744E + 08	1.966910E + 08	3.083427E + 06	1.5 ± 0.07
MultiShort	2.128019E + 08	2.080650E + 08	4.736897E + 06	2.2 ± 1.5
LogShort	1.637669E + 08	1.695956E + 08	−5.828771E + 06	3.6 ± 1.25

Table 11. Average cycle counts for measurements and simulations of benchmarks coupling SimGate with simulated Mote via serial link communicating with a Mote via the radio, error percentage, 95% confidence range for error.

Benchmark	μ_{meas}	$\mu_{simulated}$	μ_{meas} - $\mu_{simulated}$	% error ± 95% conf. bound
PingLong	3.228130E+08	3.116003E+08	1.121275E+07	3.5 ± 2.9
SenseLong	2.267467E+08	2.254300E+08	1.316726E+06	0.58 ± 2.1
APSLong	2.273877E+08	2.212660E+08	6.121661E+06	2.7 ± 6.3
MultiLong	2.362925E+08	2.285356E+08	7.756869E+06	3.3 ± 3.3
LogLong	1.891255E+08	1.915953E+08	−2.469811E+06	1.3 ± 2.4

From all three tables, then, we conclude that SimGate achieves a similar level of accuracy both when it is used as a single device simulation, and when it is part of a multi-device simulation in which the devices are communicating. Because the software, including the operating system, run by the physical hardware in each of these three experiments is precisely the same as that executed by the simulated devices, we believe that SimGate can be used as an effective tool for estimating Stargate cycle counts in heterogeneous sensor network configurations.

4.2. SimGate Execution Performance

Since our ultimate goal is to provide a complete sensor network simulation capability that can be used to complement current deployment-based research strategies, the real-time slowdown of SimGate versus the physical hardware is an important consideration. Table 12 and Table 13 compare wall-clock timings of the Stargate device to SimGate (t_{cycle}) and to SimaGate with the cycle-accuracy features disabled ($t_{nocycle}$). For cases where cycle accuracy is desired, we can enable the parts of SimGate that are necessary to make cycle count estimates internally. Comparing the performance of the resulting functional simulator to the full SimGate simulation gives the cost of achieving the accuracy levels described previously.

The simulator is 10 to 27 times slower than the real hardware when cycle accuracy is not required. This factor is smallest for *DCacheReadMiss*. There is no slowdown and instead the simulation is faster than actual hardware. The reason is that on real hardware, the cost of cache miss is so large that our simulation on a fast, high-end machine can catch up with its speed. Cycle accurate simulation (r_{cycle}) increases the cost by 2.93× (37 to 80 times slower than real hardware). There is a higher variance in these numbers, e.g., *gsmvsjpeg*, than for functional simulation ($r_{nocycle}$). One reason for this is cycle-accurate cache simulation. The *time required to simulate a cache miss and a cache hit is the same*—although the simulator adjusts the simulated clock and cycle counts appropriately for each. On a

Table 12. Average execution time (in seconds) of measurements and simulations (cycle accuracy disabled and enabled versions) of MediaBench benchmarks, slowdown rate for simulation when cycle accuracy disabled and slowdown rate for simulation when cycle accuracy enabled.

Benchmark	t_{meas}	$t_{nocycle}$	t_{cycle}	$r_{nocycle}$	r_{cycle}
adpcmdecode	7.60E−2	1.05E+00	3.23E+00	13.84	42.50
adpcmencode	8.40E−2	1.21E−02	3.61E+00	14.34	42.97
g721decode	1.57E+00	3.70E+01	1.13E+02	23.51	71.73
g721encode	1.64E+00	4.19E+01	1.19E+02	25.45	72.39
gsmdecode	3.82E−01	1.06E+01	2.86E+01	27.65	74.79
gsmencode	1.09E+00	3.01E+01	8.54E+01	27.67	78.64
jpegdecode	6.31E−02	7.28E−01	2.37E+00	11.54	37.61
jpegencode	1.35E−01	2.02E+00	6.18E+00	14.95	45.85

Table 13. Average execution time (in seconds) of measurements and simulations (cycle accuracy disabled and enabled versions) of stressmarks and MiBench benchmarks, slowdown rate for simulation when cycle accuracy disabled and slowdown rate for simulation when cycle accuracy enabled.

Benchmark	t_{meas}	$t_{nocycle}$	t_{cycle}	$r_{nocycle}$	r_{cycle}
DCacheReadHitDep	6.20E−02	6.25E−01	2.07E+00	10.08	33.37
DCacheReadHit	3.60E−02	5.80E−01	1.63E+00	16.11	45.14
DCacheReadMiss	4.24E+00	1.61E+00	7.56E+01	0.38	17.83
DCacheWrite	6.20E−02	6.60E−01	2.14E+00	10.65	34.47
BTB	1.76E−01	4.38E+00	1.27E+01	24.91	71.93
LUDecomp	3.28E−01	6.73E+00	2.05E+01	20.53	62.48
BitCount	1.15E−01	2.18E+00	6.73E+00	18.95	58.53
Dijkstra	5.63E−01	1.00E+01	3.18E+01	17.80	56.55
Mesa	6.54E−01	1.18E+01	4.09E+01	17.98	62.55
StringSearch	8.42E−01	2.30E+01	6.77E+01	27.34	80.44
SHA	2.37E−01	4.89E+00	1.41E+01	20.65	59.49
FFT	2.87E−01	5.14E+00	1.57E+01	17.89	54.68

real device a cache hit is much faster than a cache miss. Thus, application memory access patterns can have a large effect on the relative slow down of simulation. We are encouraged by these results since other full system, cycle-accurate, simulations of advanced computer systems executing an OS and application, e.g., SimOS, report slowdowns of 4,000–6,000× [23] although the results are not completely comparable since we use different host machines and simulate different targets.

5. Scalability of Multi-Simulation Framework

For our scalability experiment, we do not run Stargate simulation with mote simulation since in our simplified situation, Stargate simulation is always the bottleneck. To eliminate this interference, we thus experiment with motes only.

For each scalability experiment, we vary two experimental parameters independently: the number of sensor nodes simulated on each host of the cluster; and the number of hosts used for each experiment. For each node-count-host-count pair, we run *CntToRfm* for 60 s and record the average simulated clock speed.

In the first experiment, we simulate a one dimensional topology of sensor network. All the nodes are laid on a straight line, 50 m apart. We assumes the maximal radio range is 60 m.

Table 14 presents the results. Each cell of the table shows the ratio of the simulated average clock speed to the real time clock speed, of 7,372,800 cycles per second. To compute the average simulated clock speed, the simulator records the number of clock cycles each mote executed during the 60-s 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).

Table 14. Simulated clock speed for *1*-D topology.

	Hosts number				
Nodes per host	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

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.

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 *eight* nodes per host (the fourth row in 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 1 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.

By way of comparison to previous work, in this best case scenario 2,048 nodes can be simulated at nearly a tenth of the real time speed (in a speed comparable to Stargate simulation) using 16 hosts (lower righthand corner of Table 14), which is almost *8* times better than results reported for TOSSIM [11]. 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 Fig. 2 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 14. 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 the cluster) and the total node count is increased beyond 64. Thus, in this best case example, scalability is limited by host availability through 2,048 simulated nodes.

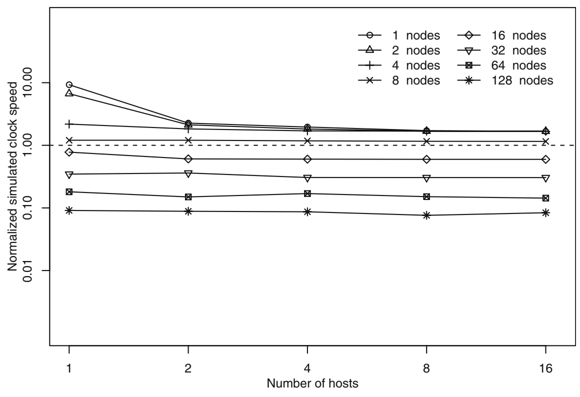

Figure 1. Scalability of 1-D topology. *X*-axis is number of hosts and *Y*-axis is clock speed. Each *curve* represents the performance with a fixed number of nodes per host. *Dashed line* shows real time speed.

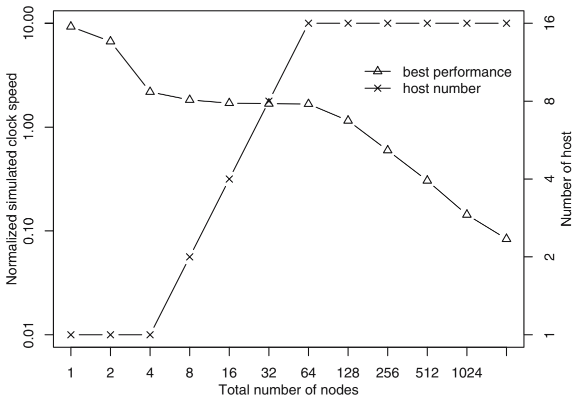

Figure 2. *Gold curves* for 1-D topology. *X*-axis is total number of nodes simulated. The left *Y*-axis is normalized performance and the right one is number of hosts. The decreasing curve is the fastest speed curve. The increasing curve gives the corresponding host number at each point.

We also perform similar experiment with more realistic two dimensional network topology, in which nodes are still 50 m apart and fill a grid whose shape is as close to a square as possible.

We show the data in Table 15. We also compare the "gold curves" of both topologies in Fig. 3. Although in 2-D case, the simulation performance is slightly worse, the shape of its curve is strikingly similar as that of 1-D topology.

To further illustrate the relationship between performance and node density, 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 16 and Fig. 4 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 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.

Table 15. Simulated clock speed for 2-D topology.

	Hosts number				
Nodes per host	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

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.

6. Related Work

There is a large body of research on simulation systems. In this section, we identify techniques that are most similar to our work. In particular, we describe and contrast frameworks for ensemble simulation for devices relevant to a sensor network and for tools for full system emulation.

6.1. *Frameworks for Ensemble Sensor Network Simulation*

There have been a number of significant efforts to simulate and emulate sensor network devices. Most of this prior work has focused on the sensing devices and in particular mote devices. These projects include Simulavr [14], ATEMU [13], Mule [55] Avrora [16], TOSSIM [11], SensorSim [12] and SENS [15]. Although, we implemented mote simulation as part of this project, we did so only to investigate ensemble simulation system for SimGate. We could have alternatively coupled current approaches with SimGate but decided instead to implement our own mote simulator to expedite the coupling process.

The ATEMU and Avrora mote simulation platforms are most similar to our system. Both provide full-system multi-simulation of mote devices. However, the multi-simulation enabled by these systems is *homogeneous*—only simulation of mote devices are coupled and no other sensor network devices, e.g., intermediate nodes, are supported. Both systems use a lock-step method. ATEMU synchronizes at each cycle and Avrora loosens the synchronization period to thousands mote cycles. Both ATEMU and Avrora can simulate motes in real time. Since Avrora is written in Java, its performance is highly dependent on JVM implementation. In our work, we use a complete different synchronization technique that is able to scale to the distributed environment, such that an unbounded computing resource can be used to improve the ensemble simulation performance. Our multi-simulation framework is also able to easily integrate heterogeneous sensor devices under a unified scheme.

There are also systems that employ *heterogeneous*, ensemble simulation. In particular, our design vision is similar to the work of [56]. The work in [56] is a

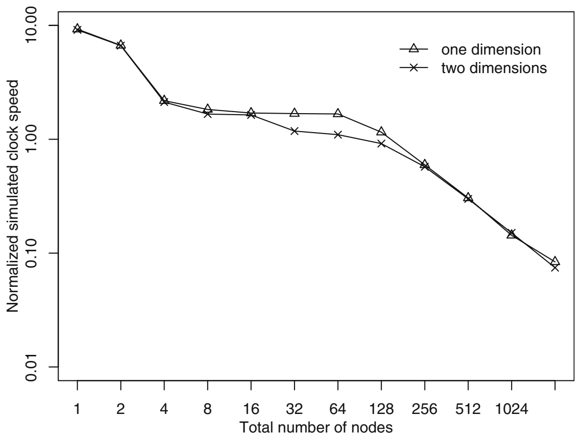

Figure 3. Best performance comparison of 1-D and 2-D topology. *X*-axis is total number of nodes simulated. The *Y*-axis is normalized performance.

Table 16. Simulated clock speed for “all-to-all” complete graph.

	Hosts number				
Nodes per host	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

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.

comprehensive framework that supports the simulation, emulation, and deployment of heterogeneous sensor network systems and applications. This framework uses TOSSIM [11] to emulate motes and EmStar [57] to emulate “microservers” (a general term for platforms like Stargate). The authors employ a wrapper library to glue the two simulation systems together. All applications must be re-compiled and linked to the EmStar library if they are to be emulated by the system.

Our goal is to enable the study, verification, debugging, and analysis of sensor network applications using a simulation platform that does not require any modification to the binaries of the applications or operating system on which they run. This enables increased flexibility for researchers and ensures that the simulation execution environment is the same as that on the real devices. Full-system simulation also enables us to easily obtain important application characteristics (e.g. accurate cycle estimation and interrupt properties) that is more difficult to collect in a purely emulative environment. Pure emulation systems do have a speed advantage however. For example, TOSSIM [11] can emulate a mote 50 times faster that actual mote execution using a 1.8 GHz Pentium IV machine. EmStar can execute re-compiled, microserver code at native speed. In SimGate, we enable users to toggle functional and cycle-accurate simulation to reduce the overhead of the latter. Moreover, we are currently investigating other optimization techniques to improve simulation speed while maintaining cycle accuracy, like dynamic binary translation [58, 59].

6.2. Full System Simulation

From the perspective of full system simulation and emulation, there are number of software systems that support a wide range of devices [20, 23–25, 59–65]. One such, very popular, system is SimOS [23]. SimOS is a full system simulator containing simulation models for most common hardware components, e.g., processor, memory, disk, network interfaces, etc. SimOS features a range of advanced processor models that trade-off accuracy for simulation speed. The fastest model applies dynamic binary translation [58, 59] for maximal simulation speed. The finest-grain model simulates the advanced pipeline struc-

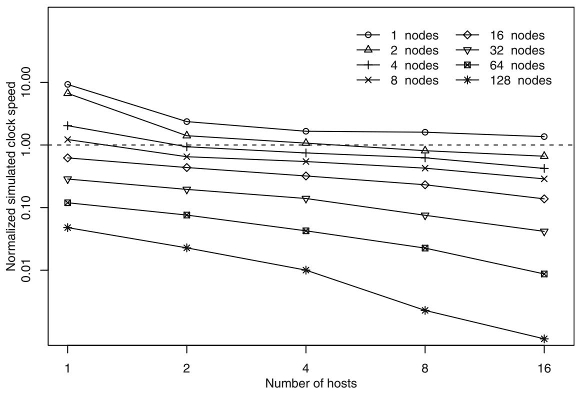

Figure 4. Scalability of “all-to-all” complete graph. *X*-axis is number of hosts and *Y*-axis is clock speed. Each *curve* represents the performance with a fixed number of nodes per host. *Dashed line* shows real time speed.

ture to provide accurate cycle-level behavior. SimOS is able to simulate the MIPS R4000 processor on a machine with the same architecture, with a slowdown of about 10× for binary translation and 5,000× for detailed pipeline simulation on a SGI 4-processor (150 MHz) machine.

Skyeye [62] is a similar project that simulates a number of ARM-based processors and development boards. Skyeye also emulates a number of peripherals, including LCD and the Ethernet interface. Skyeye is based on the GDB ARM emulator which naturally enables the use of gdb as a debugging interface—in much the same way that we do. Although some of the techniques employed in these projects are complementary and useful to our endeavor, these systems are not intended or used for sensor network research. The focus of our work is on a toolset for full-system emulation combined with cycle-accurate simulation of heterogeneous sensor network devices.

7. Conclusion

In an effort to make sensor network research more widely accessible to ease sensor software development and evolution, we have developed a system for scalable and accurate full-system simulation of a sensor network that contains both intermediate sensor nodes and base level sensor nodes. A significant part of our system, called SimGate, implements the complete Intel Stargate device and executes the Linux operating system XScale applications transparently, without modification. Our system is also capable of simulating ensemble of Stargates and motes using distributed computing resources, such that system performance scales to the growth of network size.

We investigate the accuracy and efficiency of SimGate in isolation as well as in concert with mote simulation. Our results indicate that SimGate is functionally correct and enables cycle accuracy (if desired) within 9% on average for the benchmarks that we evaluated. When we co-simulate SimGate with SimMotes (our Mote simulator) our system introduces accuracy error of less than 4% in all cases. On average, our system is 20× slower than a real device when using functional emulation and 58× slower when using cycle-accurate pipeline simulation. We believe that these results indicate that SimGate can be used as an effective tool for accurately simulating Stargate intermediate nodes in heterogeneous sensor network configurations.

We also study the scalability of our system in a simplified situation. The result shows that our multi-simulation framework is able to simulate as many as 2,048 motes at a 10× slowdown on a 16-node cluster, which is much better than prior results [11, 16].

As part of future work, we plan to investigate techniques for accurate radio and battery modeling, optimization of simulation speed, general simulation partition problem and simulation of other devices and components.

Acknowledgements

This work was supported by grants from Intel/ UCMicro, Microsoft, and the National Science Foundation (No. EHS-0209195, No. CNF-0423336 and No. CNS-0627183).

References

1. R. Kumar, V. Tsiatsis and M.B. Srivastava. "Computation Hierarchy for In-network Processing," in *Proceedings of Second ACM International Workshop on Wireless Sensor Networks and Applications*, September 2003.
2. L. Girod, V. Bychkovskiy, J. Elson and D. Estrin. "Locating Tiny Sensors in Time and Space: A Case Study," in *Proceedings of ICCD 2002,* vol. 00, 2002, p. 214.
3. A. Farina, G. Golino, A. Capponi and C. Pilotto. "Surveillance by Means of a Random Sensor Network: A Heterogeneous Sensor Approach," in *Proceedings of IEEE 8th International Conference on Information Fusion,* 2, 2005 (July).
4. A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. ACM: New York, NY, USA, 2002, pp. 88–97.
5. Heterogeneous Sensor Networks at Intel Research. http://www.intel.com/research/exploratory/heterogeneous.htm.
6. Habitat Monitoring on Great Duck Island. http://www.greatduckisland.net/index.php.
7. Habitat Monitoring on James Reserve. http://www.jamesreserve.edu.
8. Ohio State University, Kansei: Sensor Testbed for At-Scale Experiments. Poster, 2nd International TinyOS Technology Exchange, Berkeley, February 2005.
9. Mirage: Microeconomic Resource Allocation for SensorNet Testbeds. https://mirage.berkeley.intel-research.net/.
10. G. Werner-Allen, P. Swieskowski and M. Welsh. "MoteLab: A Wireless Sensor Network Testbed," *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, 2005 (April).
11. P. Levis, N. Lee, M. Welsh and D. Culler. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," in *Proceedings of ACM Conference on Embedded Networked Sensor Systems*, 2003 (November).

12. S. Park, A. Savvides and M.B. Srivastava. "SensorSim: A Simulation Framework For Sensor Networks." *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems,* 2000, pp. 104–111.
13. J. Polley, D. Blazakis, J. McGee, D. Rusk and J.S. Baras. "ATEMU: A Fine-Grained Sensor Network Simulator," in *Proceedings of First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks,* 2004.
14. Simulavr: A simulator for the Atmel AVR family of microcontrollers. http://www.nongnu.org/simulavr.
15. S. Sundresh, W. Kim and G. Agha. "SENS: A Sensor, Environment and Network Simulator." *The IEEE 37th Annual Simulation Symposium*, 2004.
16. 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*, 2005 (April).
17. Mote hardware platform. http://www.tinyos.net/scoop/special/hardware.
18. Stargate: A Platform X Project. http://platformx.sourceforge.net/.
19. C. Pereira, J. Lau, B. Calder and R.K. Gupta. "Dynamic Phase Analysis for Cycle-Close Trace Generation," in *the Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODE-S+ISSS 2005*, 2005. Jersey City, NJ (September).
20. Intel XScale Microarchitecture XDB Simulator 2.0. http://www.intel.com/design/pca/prodbref/250424.htm.
21. D. Brooks, V. Tiwari and M. Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *Proceedings of the 27th annual international symposium on Computer architecture,* 2000, pp. 83–94.
22. G. Contreras, M. Martonosi, J. Peng, R. Ju and G.-Y. Lueh. "XTREM: A Power Simulator for the Intel XScale Core," in *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2004 (June).
23. M. Rosenblum, S.A. Herrod, E. Witchel and A. Gupta. "Complete Computer System Simulation: The SimOS Approach." *IEEE Parallel and Distributed Technology,* 1995, winter:34–43.
24. P. Magnusson and B. Werner. "Efficient Memory Simulation in SimICS," in *Proceedings of the 28th Annual Simulation Symposium*, 1995.
25. P.S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrm and B. Werner. "SimICS/sun4m: A Virtual Workstation," in *Proceedings of the 1998 USENIX Annual Technical Conference*, 1998.
26. T. Austin, E. Larson and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, 2002.
27. P. Levis and D. Culler. "Mate: A Tiny Virtual Machine for Sensor Networks,"in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems,* ACM: New York, NY, USA, 2002, pp. 85–95
28. Intel XScale Technology. http://www.intel.com/design/intelxscale/.
29. Familiar linux. http://familiar.handhelds.org.
30. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler and K. Pister. "System Architecture Directions for Network Sensors," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000 (October).
31. Intel StrataFlash Chip Verilog Model. http://www.intel.com/design/flcomp/toolbrfs/298189.htm.
32. 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,* vol. 23, 1975, pp. 1417–1433.
33. Wireless Propagation Bibliography. http://w3.antd.nist.gov/wctg/manet/wirelesspropagation_bibliog.html.
34. 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 (IPSN'05),* Los Angeles, California, 2005 (April).
35. 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.
36. 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 (SenSys'03)*, 2003.
37. O. Landsiedel, K. Wehrle and S. Götz. "Accurate Prediction of Power Consumption in Sensor Networks," in *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, Sydney, Australia, 2005 (May).
38. V. Shnayder, M. Hempstead, B. rong Chen, G. Werner-Allen and M. Welsh. "Simulating the Power Consumption of Large-Scale Sensor Network Applications," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Baltimore, MD, 2004 (November).
39. V. Shnayder, M. Hempstead, B. rong Chen and M. Welsh. "PowerTOSSIM: Efficient Power Simulation for TinyOS Applications," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Baltimore, MD, 2004 (November).
40. K.C. Syracuse and W. Clark. "A Statistical Approach to Domain Performance Modeling for Oxyhalide Primary Lithium Batteries," in *Proceedings of Annual Battery Conference on Applications and Advances*, 1997 (January).
41. L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino and R. Scarsi. "A Discrete-time Battery Model for High-Level Power Estimation," in *Proceedings of Design, Automation and Test in Europe*, 2000.
42. D. Rakhmatov and S. Vrudhula. "Time-to-Failure Estimation for Batteries in Portable Electronic Systems," in *Proceedings*

of the International Symposium on Low Power Electronics and Design, 2001 (August).
43. D. Linden and T. B. Reddy. Handbook of Batteries (3rd edition). McGraw-Hill, 2002.
44. Y. Wen, R. Wolski and G. Moore. “DiSenS: Scalable Distributed Sensor Network Simulation,” in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* San Jose, CA, 2007 (March).
45. Chipcon CC1000 Brochure. http://www.chipcon.com/files/CC1000_Brochure.pdf.
46. 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,* 2000 (September).
47. H.D. Simon. “Partitioning of Unstructured Problems for Parallel Processing,” *Computing Systems in Engineering,* vol. 2, 1991, pp. 135–148.
48. A. Pothen. “Graph Partitioning Algorithms with Applications to Scientific Computing,” *Parallel Numerical Algorithms*, Kluwer, 1997, pp. 323–368.
49. B. Hendrickson and R. Leland. The Chaco User’s Guide: Version 2.0. Technical Report SAND94–2692, Sandia National Lab, 1994.
50. M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown. “Mibench: A Free, Commercially Representative Embedded Benchmark Suite,” *IEEE 4th Annual Workshop on Workload Characterization,* Austin, TX, 2001 (December).
51. C. Lee, M. Potkonjak and W. Mangione-Smith. “Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,” in *International Symposium on Microarchitecture (Micro-30),* 1997, pp. 330–335.
52. D. Niculescu and B. Nath. “Ad Hoc Positioning System (APS),” in *Proceedings of IEEE Global Communications Conference,* 2001 (November).
53. G.W. Hill. “ACM Algorithm 395: Student’s T-Distribution.” *Communications of the ACM,* vol. 13, no. 10, 1970, pp. 617–619 (October).
54. SimpleScalar for ARM. http://www.simplescalar.com/v4test.html.
55. D. Watson and M. Nesterenko. “Mule: Hybrid Simulator for Testing and Debugging Wireless Sensor Networks,” in *Workshop on Sensor and Actor Network Protocols and Applications,* 2004 (August).
56. L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil and T. Schoellhammer. “A System for Simulation, Emulation, and Deployment of Heterogeneous Sensor Networks,” in *Proceedings of ACM Conference on Embedded Networked Sensor Systems,* 2004 (November).
57. L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan and D. Estrin. “EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks,” in *Proceedings of USENIX Tech. Conf.,* 2004.
58. R.F. Cmelik and D. Keppel. “Shade: A Fast Instruction Set Simulator for Execution Profiling,” in *Proceedings of ACM SIGMETRICS*, 1994.
59. E. Witchel and M. Rosenblum. “Embra: Fast and Flexible Machine Simulation,” *ACM SIGMETRICS Performance Evaluation Review,* vol. 24, no. 1, 1996, pp. 68–79 (May).
60. SimOS-PPC: ARL’s Full System Simulation Project. http://www.cs.utexas.edu/users/cart/simOS/index.html.
61. R.C. Bedichek. “Efficient Memory Simulation in SimICS,” in *Proceedings of ACM SIGMETRICS,* 1995.
62. SKYEYE: An Embedded Simulation System. http://www.skyeye.org.
63. QEMU: A Generic and Open Source Processor Emulator. http://fabrice.bellard.free.fr/qemu/.
64. The Bochs IA-32 Emulator Project. http://bochs.sourceforge.net.
65. PearPC: PowerPC Architecture Emulator. http://pearpc.sourceforge.net/.

Ye Wen received his B.S. in Computer Science from the Zhejiang University, China and his M.S. and Ph.D. in Computer Science at the University of California, Santa Barbara. His interests include sensor networks, embedded system, mobile and wireless system, and parallel and distributed computing.

Selim Gurun received his B.S. degree in Computer Engineering from ODTU, Turkey in 1997 and his M.S. and the Ph.D. degrees in Computer Science from Rensselaer Polytechnic Institute, and University of California, Santa Barbara, USA in 1999 and 2007, respectively. He worked at Ericsson, Inc. from 1999 to 2002. He is a Post Doctoral Researcher in the Deparment of Computer Science, University of California Santa Barbara. His research interests are in the areas of embedded and distibuted computing systems.

Navraj Chohan received his B.S. in Computer Engineering from the University of California, Santa Barbara. He worked as a Software Engineer at Applied Signal Technologies shortly after graduating. He is currently pursuing his M.S. and Ph.D. at UCSB under his advisor, Chandra Krintz. His interests include sensor networks, distributed systems, networking, and VLSI.

Rich Wolski is an Associate Professor in Computer Science at the University of California, Santa Barbara (UCSB). Having received his M.S. and Ph.D. degrees from the University of California at Davis (while he held a full-time research position at Lawrence Livermore National Laboratory) he has also held positions at the University of California, San Diego and the University of Tennessee. He is currently also a Strategic Advisor to the San Diego Supercomputer Center and an Adjunct Faculty Member at the Lawrence Berkeley National Laboratory. Dr. Wolski heads the Middleware and Applications Yielding Heterogeneous Environments for Metacomputing (MAYHEM) Laboratory which is responsible for several national scale research efforts in the area of high-performance distributed computing and grid computing. These efforts have generated nationally supported production-quality software tools such as the Network Weather Service (currently distributed as part of the NSF Middleware Initiative's baseline software distribution) and QBETS (currently in production as a batch queue prediction service for the TeraGrid) resulting in an international user-community in addition to an extensive scholarly corpus. His most recent efforts have focused on both the implementation of nationally distributable tightly-coupled programs as well as the development of statistical predictive techniques for resource-constrained power usage and resource failure prediction.

Chandra Krintz is an Associate Professor at the University of California, Santa Barbara (UCSB). She joined the UCSB faculty in 2001 after receiving her M.S. and Ph.D. degrees in Computer Science from the University of California, San Diego (UCSD) under the advisement of Dr. Brad Calder. Chandra's research interests include automatic and adaptive compiler, virtual runtime, and operating system techniques that improve performance (for high-end systems) and increase battery life (for mobile, resource-constrained devices). In particular, her work focuses on exploiting repeating patterns in the time-varying behavior of underlying resources, applications, and workloads to guide dynamic optimization and specialization of program and system components.