Toward Full-System, Cycle-Accurate Simulation of Sensor Networks [*]

UCSB Computer Science Technical Report Number CS2005-12

Ye Wen, Selim Gurun, Navraj Chohan, Rich Wolski, and Chandra Krintz
Department of Computer Science
University of California, Santa Barbara

June 1, 2005

Abstract

We present an accuracy and performance analysis of SimGate – a full-system simulation of the Stargate intermediate-level sensor network device. We also examine ensemble simulations using SimGate and either one or two simulated Mica2 motes using the same criteria. We find that accurate functional behavior and cycle counts (at the full device level) are achievable using SimGate alone, and in conjunction with simulated Motes. Also, the slowdown compared to real-time for these simulations is modest with respect to previously published work.

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 so resource constrained, however, a sensor network must 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 a "gateway" between the network of sensor elements and more power-intensive network technologies.

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

Designing and investigating these ensemble systems, to date, has relied primarily on physical deployments and experimentation [8, 15, 16, 23, 38]. 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 [19, 25, 27, 32, 35, 36], 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 SimGate – a full-system simulation of the Intel Stargate device [34] (distributed by Crossbow Inc.) – that we have developed for use with sensor element simulations as part of a simulated ensemble. The Stargate device is intended to function as a general purpose processing, storage, and network gateway element in a sensor network deployment. These devices are battery powered, and are both fewer in number and larger in size than the sensing devices. The Stargate's more intrusive nature facilitates the use of large batteries that enable longer battery life and improved compute and storage capability.

Our goal is to provide both functional correctness and cycle-count accuracy at the device level, in a simulation of the Stargate that can be coupled with simulated sensors. The currently available tools for simulating more complicated, heavy-weight, intermediate sensor nodes (such as the Stargate) are limited. For example, there are tools for simulation of the Intel XScale processor [40] and its power consumption [4, 7] alone. However, to our knowledge, there are no simulation tools that simulate the complete Stargate device.

SimGate captures the behavior of the Stargate internal components including the processor, memory hierarchy, communications (serial and radio), and peripherals. In essence, SimGate is a *virtual device*, in that it boots and runs the Familiar Linux operating system and any program binary that executes over it, *without modification*. SimGate is also able to accurately estimate processor cycle counts. Moreover, this functionality can be toggled to trade off cycle-accuracy for simulation performance.

We are also able to couple SimGate with our own simulations of the Crossbow Mica2 sensor nodes (called SimMote) to produce a simulated sensor network *ensemble*. These simulations are also virtualized representations of the physical hardware (i.e. full-system emulations providing accurate cycle counts). SimMote provides similar functionality to that of existing Mote simulation and emulation systems [19, 25, 27, 32, 35, 36]. We make no claims as to its superiority over these systems – instead, we have used SimMote to expedite our investigation and empirical evaluation of heterogeneous device, ensemble simulation.

To empirically evaluate the efficacy of our system, we measure the accuracy (in terms of Stargate machine cycles) and real-time performance of SimGate using a range of stressmarks and community benchmarks. We also present results for similar experiments in which the SimGate and SimMote interoperate via a serial interface (simulated in both). Finally, we examine three-device ensemble consisting of a SimGate node, a serially-connected SimMote, and a third SimMote that communicates only via simulated radio. For these latter two cases, we run our own multi-device benchmark suite. Each experiment compares simulated results to measurements gathered empirically from physical Stargate and/or Mica2 devices. In all cases throughout this study, the actual hardware devices and simulations run the same operating system and benchmark binary, without modification. Thus, the results test the degree to which our simulations may be used in place of physical hardware in each experimental setting.

Our results indicate that we are able to accurately simulate the full system of an intermediate 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 $58X$ over real-time device execution and that a slowdown of $20X$ can be achieved if only a functional simulation (i.e. without accurate cycle counts) is required. 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 simulator. 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 SimGate Simulator

Simulation is a potentially an 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 [29, 20, 21], processor/power simulation [1, 4, 7], or on the sensing devices themselves [19, 25, 27, 32, 35, 36]. 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 co-simulation of different sensor network devices as part of an ensemble. 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 sensor nodes (which we refer to as Motes after the popular Berkeley Mote implementation [18]) and more powerful, wall-powered, computational environments. Intermediate nodes are commonly responsible for sensor device control and in-network processing [17] 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 [34].

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-accurate simulation of the Stargate's Intel XScale processor pipeline [41]. SimGate is completely transparent to the above software layers – i.e., the system boots and executes the popular embedded OS, Familiar Linux [9] and any program that executes over it, without modification. Moreover, SimGate eases sensor network program development by implementing a unified debugging interface. In this section, we present the design and implementation of the SimGate architecture.

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 400MHz Intel

XScale processor, an Intel SA1111 companion chip for I/O, Intel StrataFlash, SDRAM, PCMCIA/CF slots, and connector for a Mote [34]. 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, 32KB 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
- 64MB SDRAM chip
- 32MB 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.

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 [1]. 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.

To achieve the cycle accuracy of processor core simulation, we implemented a simulation component for the XScale CPU pipeline. The Intel XScale core employs a seven or eight stage (depending on the instruction flow), single-issue, super pipeline. There are actually three pipelines 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.

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 [7]. We used this implementation as a reference and extended and evolved it using benchmark measurements from a real Stargate device (since the Stargate implements a slightly different version of XScale processor that that implemented within XTREM). We implemented the MMU components (TLB, BTB, caches and buffers) within our pipeline simulator. Since these components are transparent to data correctness, we only perform fast symbolic simulation without the actual data movement. 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 alluded to above, we are able to toggle the type of simulation between cycle-accurate 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 [41]. 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.

To support pipeline simulation toggling, our pipeline simulation is trace-based. That is, after an instruction is executed using functional simulation, we feed it to the pipeline simulator to drive the clock. This may result in a delay between the execution and the clock advance. This delay is in the order of several cycles on average; as a result it has very little impact on the device level cycle accuracy (which we report in Section 4).

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 [14].

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 802.11b radio model used by the Stargate.

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.

2.2 Coupling SimGate with Other Sensor Network Simulators

To explore simulation of SimGate with that of other sensor network components, we has developed SimMote – a simulation of the Mica2 Mote [22]. We emphasize that SimMote is intended to provide similar functionality to other Mote simulators [36, 37, 27, 19, 25, 35] in this context and as such, we make no claim about its relative scientific value. Implementing SimMote simply has ensured, in the most expedient way, that the Mote simulation is interoperable with, and comparable to SimGate .

Mica2 features the 8MHz Atmel ATmega128 microcontroller (simple 16-bit RISC ISA), on-board Flash memory and a 900MHz radio. Compared to SimGate, the SimMote is much easier given the significantly simpler hardware and software design (it also has the added benefit of testing the flexibility of our simulation development framework).

SimMote currently supports the following features:

- AVR instruction set
- Most on-chip functions: program memory, IO registers, timer, UART, interrupt, SPI (Serial Peripheral Interface), and ADC (Analog/Digital Converter)
- 512KB on-board flash
- Serial ID chip
- CC1000 radio chip
- A very simple radio transmission model.

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.

To couple device simulators, we have also developed a multi-simulation manager. The manager is a multi-threaded software system that controls the life cycle of constituent simulators, e.g., it provides simulator services that include create, start, stop, join and leave. The manager forks a thread for each simulator invokes the start routine in each. The start routine initiates the OS boot process and uses a configuration file to invoke the benchmark or set of benchmarks of interest. The manager also implements a unified debugging interface (which we describe below) that dispatches debug commands to different simulators.

To achieve cycle-accurate, coordinated simulation of multiple simulators, the proportion of the rates of execution of simulated devices must be held to be roughly the same as that for real devices. This coordination is important for execution as well as for communication (e.g., for a radio or serial connection). To enable this coordination, we employ a simple, lock-step method that forces the clock within each simulator to synchronize periodically. This is similar to the synchronization mechanism in the Avrora mote simulator [36], however, we maintain separate, individual clocks per simulator as opposed to a single global clock.

To implement this synchronization, The multi-simulation manager inserts a synchronization event into the event queue of each simulator when it is first instantiated. The event repeatedly fires at a fixed interval. When the event fires, all of the threads of simulation meet at the same clock point before continuing execution. We set the synchronization interval based on the clock frequency of the communication technologies. Since the fastest technology is serial transmission between the Mote and the Stargate (at 57.6KB/second), we use the one byte serial transmission time as the synchronization period. This equals 128 Mote cycles.

We implemented a simple Mote radio model within SimMote based on that implemented in Avrora [36]. Our model is different in that we do not use a global clock across simulators or a centralized packet dispatching/assembly object. Instead, we distribute each transmitted packet to the receiving device simulator which assembles the packet locally, using its own clock. Since the simulators execute in lock step, our choice of a 128 Mote cycles synchronization period is sufficient to cover radio transmission (correct packet assembly) since radio transmission is 19.2KB/s.

Since the clock rate of a Stargate is 54 times that of a Mote, we must synchronize SimGate simulators with SimMote simulators. To enable this, we can use a SimGate synchronization interval that is 54 times that of the SimMote. An interesting side effect of this however, is that doing so forces us to simulate the Motes *as slow as* the Stargate. Since the machine on which we run our simulations is much faster than the real speed of the Mote, we can simulate up to 6 times faster than real Mote execution. However, in an ensemble system of heterogeneous device simulators, the fastest machine simulated is the performance bottleneck. As such, we must slow the SimMotes to match SimGate speed.

As stated previously, we have not yet simulated the 802.11b radio model used by the Stargate. As a result, we are only able to simulate communication between SimGates and other SimGates via the *Mote-NIC*, i.e., the attached Mote via the serial connection between Mote and Stargate. We are able to simulate communication between SimGates and SimMotes as is done for real sensor networks using this same interface (Mote-NIC) and between SimMotes and other SimMotes via our simple radio model.

2.2.1 Other Simulation Framework Features

Debugging is a key component in an ensemble system of sensor network devices simulators. To facilitate debugging, we implemented a unified debugging interface and dispatch within the multi-simulation manager that supports debugging of concurrently executing simulation systems.

The manager dynamically dispatches debug commands to the individual simulators. Since each simulator runs on a separate thread, the debugger can attach to any of the simulator threads to control its execution flow and to watch the change in the execution state. The functions we support in the simulators include step execution, the dump of memory and flash, and watching of internal state and break points.

Another useful function that we implemented is checkpointing. Our checkpointing mechanism within each simulator saves the current, full-system, simulation state including the snapshot of memory and flash file system. We provide mechanisms that facilitate the storage and loading of such images to enable fast forwarding and continuation of an executing system.

Benchmark	Executables	Description
DCacheReadHitDep	dcachehit_r	Data cache read 100% hit with data dependency
DCacheReadHit	dcachehit_nd_r	Data cache read 100% hit without 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 1. Stressmarks 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

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

3 Experimental Method

To evaluate and analyze the performance and accuracy of SimGate, 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 a 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).

In the following subsections, we first detail the benchmarks that we use for the SimGate alone, for the SimMote alone, and for our ensemble scenarios. We also describe the experimental apparatus that we use to collect our simulated and actual measurements.

3.1 Benchmarks

For stand-alone SimGate evaluation, we employed our hand-coded stressmarks and benchmarks from both the MiBench [13] and the Mediabench [5]. In Table 1 we present our stressmarks to measure the simulation performance.

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

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

Table 4. Benchmarks that we used to evaluate the components of SimMote . The third column shows the functional units that were evaluated during the test.

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 dependency. 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 (25MBytes). 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 standalone applications (i.e. the measurements were independent of SimGate 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

Benchmark	Description	Functional Unit
Ping	Echoes network packet back to sender	Network interface
Sense	Processes a sensor read query	Analog/Digital converter
APS [24]	Ad-hoc positioning system	Arithmetic/Logic unit
Log	Reads log from Flash	Secondary Flash & UART interface
Multi	Parallel APS computations on Mote and Stargate	Arithmetic/Logic unit

Table 5. Benchmarks that we used to evaluate the ensemble simulation of SimGate and SimMote. The third column shows the functional units that are exercised most heavily during benchmark execution.

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 Multi benchmark also tests concurrent computation by running parallel computations of ad-hoc positioning system (APS) [24] on both Mote and Stargate. This test is useful to evaluate the performance of simulating coordinated computation on Mote and Stargate.

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.

We ran our simulators on a dedicated Linux (kernel ver 2.6.8) machine. The machine has a 64bit AMD Opteron CPU running at 2.4GHz and 4GB 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

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

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

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` on the Mote. PORTC is directly connected 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 nanoseconds) 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.

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.

Table 6, Table 7 and Table 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

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

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

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

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

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 [10] with 19 degrees of freedom to model the difference of the averages (marked as $\pm$ confidence bound in the table).

Note that the error percentage and confidence interval also indicate whether the 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 the error percentage we observe for each benchmark is statistically significant at the 95% confidence level. There are 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 [7, 31], 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

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

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

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 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

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

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	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 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

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.

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 SimGate 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

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

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

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 $2.93X$ (37 to 80 times slower than real hardware). There is a higher variance in these numbers, e.g., `gsm` vs `jpeg`, 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 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 $4000X - 6000X$ [29] although the results are not completely comparable since we use different host machines and simulate different targets.

5 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.

5.1 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 [32], ATEMU [27], Mule [37] Avrora [36], TOSSIM [19], SensorSim [25] and SENS [35]. 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 similar to ours to synchronize simulation threads and enable accurate timing and correct communication. 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 the similar synchronization technique as in Avrora. However, we must deal with more complex situation in which coordination between devices happens between very different devices. To simulate only Motes (as is done in these prior works), we achieve slightly better performance than Avrora because of the use of C++ instead of Java.

There are also systems that employ *heterogeneous*, ensemble simulation. In particular, our design vision is similar to the work of [12]. The work in [12] is a comprehensive framework that supports the simulation, emulation, and deployment of heterogeneous sensor network systems and applications. This framework uses TOSSIM [19] to emulate Motes and EmStar [11] 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.

In SimGate, 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. This SimGate model 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 [19] can emulate a Mote 50 times faster that actual Mote execution using a 1.8GHz 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 [6, 39].

5.2 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 [29, 40, 39, 30, 20, 21, 2, 33, 28, 3, 26]. Once such, very popular, system is SimOS [29]. 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 [6, 39] for maximal simulation speed. The finest-grain model simulates the advanced pipeline structure 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 $10X$ for binary translation and $5000X$ for detailed pipeline simulation on a SGI 4-processor (150MHz) machine.

Skyeye [33] 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.

6 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 full-system, functional and cycle-accurate simulation of intermediate sensor nodes. Our system, called SimGate, implements the complete Intel Stargate device and executes the Linux operating system XScale applications transparently, without modification.

We investigate the accuracy and efficiency of SimGate in isolation as well as in concert with sensor device (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 SimGates with SimMotes(our Mote simulator) our system introduces accuracy error of less than 4% in all cases. On average, our system is $20X$ slower than a real device when using functional emulation and $58X$ 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. As part of future work, we plan to investigate techniques for accurate radio and battery modeling, optimization of simulation speed, the scalability of our multi-simulation system for large-scale sensor networks, and simulation of other devices and components.

References

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

[2] R. C. Bedichek. Efficient Memory Simulation in SimICS. *In Proceedings of ACM SIGMETRICS*, 1995.

[3] The Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net`.

[4] 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*, pages 83–94, 2000.

[5] 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)*, pages 330–335, 1997.

[6] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. *In Proceedings of ACM SIGMETRICS*, 1994.

[7] 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*, June 2004.

[8] Habitat Monitoring on Great Duck Island. http://www.greatduckisland.net/index.php.

[9] Familiar Web Page. http://www.handhelds.org.

[10] G. W. Hill. ACM Algorithm 395: Student's T-Distribution. *Communications of the ACM*, 13(10):617–619, Oct. 1970.

[11] 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.

[12] 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*, Nov. 2004.

[13] 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*, Dec. 2001. Austin, TX.

[14] Intel StrataFlash Chip Verilog Model. http://www.intel.com/design/flcomp/toolbrfs/298189.htm.

[15] Habitat Monitoring on James Reserve. http://www.jamesreserve.edu.

[16] Ohio State University,Kansei: Sensor Testbed for At-Scale Experiments. Poster, 2nd International TinyOS Technology Exchange, Berkeley, February 2005.

[17] 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*, Sept. 2003.

[18] 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*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[19] 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*, Nov. 2003.

[20] P. Magnusson and B. Werner. Efficient Memory Simulation in SimICS. *In Proceedings of the 28th Annual Simulation Symposium*, 1995.

[21] 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.

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

[23] Mirage: Microeconomic Resource Allocation for SensorNet Testbeds. https://mirage.berkeley.intel-research.net/.

[24] D. Niculescu and B. Nath. Ad Hoc Positioning System (APS). *In Proceedings of IEEE Global Communications Conference*, Nov. 2001.

[25] 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*, pages 104–111, 2000.

[26] PearPC: PowerPC Architecture Emulator. http://fabrice.bellard.free.fr/qemu/.

[27] 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 Scociety Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[28] QEMU: a generic and open source processor emulator. http://fabrice.bellard.free.fr/qemu/.

[29] 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.

[30] SimOS-PPC: ARL's Full System Simulation Project. `http://www.cs.utexas.edu/users/cart/simOS/index.htm`

[31] SimpleScalar for ARM. `http://www.simplescalar.com/v4test.html`.

[32] Simulavr: A simulator for the Atmel AVR family of microcontrollers. `http://www.nongnu.org/simulavr`.

[33] SKYEYE: an embedded simulation system. `http://www.skyeye.org`.

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

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

[36] 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.

[37] 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*, Aug. 2004.

[38] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A Wireless Sensor Network Testbed. In *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)*, April 2005.

[39] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):68–79, May 1996.

[40] Intel XScale Microarchitecture XDB Simulator 2.0. `http://www.intel.com/design/pca/prodbref/250424.htm`.

[41] Intel XScale Technology. `http://www.intel.com/design/intelxscale/`.