S2DB : A Novel Simulation-Based Debugger for Sensor Network Applications *

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

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

Selim Gurun
University of California, Santa Barbara
gurun@cs.ucsb.edu

ABSTRACT

Sensor network computing can be characterized as resource-constrained distributed computing using unreliable, low bandwidth communication. This combination of characteristics poses significant software development and maintenance challenges. Effective and efficient debugging tools for sensor network are thus critical. Existent development tools, such as TOSSIM, EmStar, ATEMU and Avrora, provide useful debugging support, but not with the fidelity, scale and functionality that we believe are sufficient to meet the needs of the next generation of applications.

In this paper, we propose a debugger, called S^{2}DB, based on a distributed full system sensor network simulator with high fidelity and scalable performance, DiSenS. By exploiting the potential of DiSenS as a scalable full system simulator, S^{2}DB extends conventional debugging methods by adding novel device level, program source level, group level, and network level debugging abstractions. The performance evaluation shows that all these debugging features introduce overhead that is generally less than 10% into the simulator and thus making S^{2}DB an efficient and effective debugging tool for sensor networks.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; I.6 [**Simulation and Modeling**]

General Terms

Experimentation, Performance

Keywords

Sensor Network, Debugging, Simulation

1. INTRODUCTION

Sensor networks, comprised of tiny resource-constrained devices connected by short range radios and powered by batteries, provide an innovative way to implement pervasive and non-intrusive environmental instrumentation and (potentially) actuation. The resource-constrained nature of sensor network devices poses significant software development and maintenance challenges. To prolong battery life and promote miniaturization, most devices have little memory, use low-power and unreliable radios, and run long duty cycles. In addition to these per-device constraints, by definition sensor networks are also distributed systems, with all of the concomitant synchronization and consistency concerns that distributed coordination implies.

For these reasons, effective debugging support is critical. A number of sensor network development systems [2, 18, 3, 17, 13, 6] provide debugging support for individual devices and/or the complete network. However, they all have their limitations. Some rely on hardware support, subject to the same resource constraints that as the programs on which they operate. Some only monitor the network radio traffic. And most importantly, as networks scale, these tools become difficult to apply to the details of collections of interacting sensor nodes.

In this paper, we present a new approach that is based on scalable full system sensor network simulation with enhanced debugging features. Our debugging tool is called S^{2}DB (where S^2 stands for Simulation and Sensor network). The goal of S^{2}DB is to adapt conventional debugging methods to sensor network applications so that we can have better control of hardware details and debug the complete sensor network in a coordinated way. Our approach relies upon four principle innovations in the area of debugging resource constrained devices.

- At the single device level, we introduce the concept of *debugging point* – a generalized notion of break point, watch point, and state interrogation – that permits state display from all sensor device subsystems (flash pages, buffers, etc.);
- Also at the device level, we introduce virtual registers within the simulator to support source level instrumentation and tracing. The access to these registers does not affect the correct functioning of other components;
- At the multi-device level, we introduce a coordinated break condition, which enables the coordinated execution control of multiple devices;
- Finally, at the network level, we provide a "time traveling" facility to use with network level trace analysis, so that error site can be rapidly restored for detailed inspection.

S^{2}DB is built upon DiSenS [25], a scalable distributed full system sensor network simulator DiSenS has a distributed simulation framework. Individual sensor devices are emulated in separated operating system threads. DiSenS then partitions and schedules these

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'06, October 22–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-542-8/06/0010 ...$5.00.

device emulations to the computer nodes of a cluster, and simulates inter-device communication at the radio level (i.e. below the communication protocol stack and radio hardware device interfaces). Sensor device emulations in DiSenS are cycle-accurate. Moreover, a plugin mechanism allows the insertion of power models and radio models with different fidelity levels. Thus DiSenS is capable of accurate, large-scale sensor network simulation where the application and operating system code can be executed, unmodified on native hardware.

DiSenS benefits our design and implementation in many aspects. Its simulator infrastructure gives us the full control of device states, which enables the design of *debugging point*s. Its high performance makes our debugger execute efficiently. Its scalability enables us to debug large-scale sensor networks. While the availability of a high-fidelity radio model for sensor network radio remains elusive (making many senor network implementors reluctant to embrace simulation and/or emulation), we believe the ability to debug sensor network programs at scale as a precursor to actual deployment will cut development time and reduce the amount of *in situ* debugging that will be required in an actual deployment.

We also wish to emphasize that in this paper we do not claim S^2DB adequately addresses many of the thorny difficulties associated with all debugging tools (e.g. the ability to debug optimized code). Rather our focus is on innovations that we believe are important to the development of large-scale senor network deployments and that also improve the current state-of-the-practice in sensor network debugging. In Section 2, we first give the background of sensor network debugging. In Section 3, we briefly introduce the features and details of DiSenS that are relevant to our debugging purpose. In Section 4, we introduce the *debugging point* and its use with break conditions. We also present the design of virtual hardware based source level instrumentation. In Section 5, we discuss how to control the execution of multiple devices in a coordinated way. We focus on the implementation detail in DiSenS infrastructure. In Section 6, we talk about the checkpoint implementations for fast time traveling. We evaluate the performance of our enhancing techniques in Section 7. And we conclude our work in Section 8.

2. RELATED WORK

Like most embedded devices, sensor network devices can be debugged with special hardware support. For motes (e.g. Mica2 and MicaZ), Atmel's AVR JTAG ICE (In-Circuit Emulator) [2] is one of the popular hardware-based debuggers. Atmel's AVR family of microcontrollers (that are currently used as the processing elements in many mote implementations) has built-in debugging support, called On-Chip Debugging (OCD). Developers can access the OCD functions via JTAG [10] hardware interface. With JTAG ICE, developers can set break points, step-execute program and query hardware resources. JTAG ICE can also be used with GUI interfaces or a GDB debugging console. Hardware-based approaches such as JTAG ICE typically have their limitations. For example, it is not possible to synchronize the states of program execution with I/O systems in debugging. This is because when the program execution is stopped in JTAG ICE, the I/O system continues to run at full speed [1]. Also since the debugging support is only provided with the processing unit (i.e. the microcontroller), it is not easy to interrogate the state of other on-board systems, like flash memory. In contrast, by working with the full system DiSenS simulations, S^2DB does not suffer from these limitations.

At network level, many monitoring and visualization tools like Sympathy [18, 19], SpyGlass [3], Surge Network Viewer [22] and Mote-VIEW [16] provide a way to trace, display and analyze network activities for a physical sensor network. These tools usually use a software data collecting module running on sensor nodes in the network. The collected data is transferred using flooding or multihop routing to the gateway node. The gateway node then forwards the data to a PC class machine for analysis or visualization. These tools are useful for displaying the network topology and and analyzing the dynamics of data flow, particularly with respect to specific inter-node communication events. Tools like Sympathy even specialize in detecting and localizing sensor network failures in data collection applications. However, these monitoring may be intrusive in that they share many of the scarce device resources they use with the applications they are intended to instrument. These tools may complement what we have with S^2DB . When a communication anomaly is detected, for example, often a program-level debugger may still be necessary to pinpoint the exact location of error in code.

More generally, while debugging on real hardware is the ultimate way to verify the correctness of sensor network applications, simulation based debuggers provide complementary advantages that have been successfully demonstrated by other projects. Many sensor network simulators, like TOSSIM [13], ATEMU [17], Avrora [23] and EmStar [6], provide significant debugging capabilities. TOSSIM is a discrete event simulator for TinyOS applications. It translates the TinyOS code into emulation code and links with the emulator itself. So debugging with TOSSIM is actually debugging the emulator. Developers have to keep in their mind the internal representation of device states. While discrete event simulators are useful for verifying functional correctness, they typically do not capture the precise timing characteristics of device hardware, and thus have limited capability in exposing errors in program logic. In contrast, full system simulators, such as ATEMU and Avrora, have much higher fidelity. ATEMU features a source level debugger XATDB, which has a graphic frontend for easy use. XATDB can debug multiple sensor devices, but can only focus on one at a time. Avrora provides rich built-in support for profiling and instrumentation. User code can be inserted at any program address, watches can be attached to memory locations, and specific events can be monitored. These facilities can be quite useful for debugging purposes. Indeed, we extend Avrora's probe and watch concepts in the development of S^2DB's debugging points (cf. Section 4). In addition to this support for simulator instrumentation, S^2DB also provides a source code level instrumentation facility, via virtual debugging registers, since it is easier to use for some debugging problems.

Time traveling for debugging is currently the subject of much research [11, 20] in the field of software system development and virtualization. Flashback [20] is a lightweight extension for rollback and replay for software debugging. Flashback uses shadow processes to take snapshots of the in-memory states of a running process and logs the process' I/O events with the underlying system to support deterministic rollback or replay. VMM (virtual machine monitor) level logging is used in [11] for replaying the system executing in a virtual machine. Checkpointing the state of a full system simulator is easier than that in a real OS or virtual machine monitor since all the hardware are simulated in software. Our results show that time traveling support in DiSenS has very low overhead due to the simpleness of sensor hardware it emulates.

3. THE DiSenS SIMULATOR

S^2DB is built upon DiSenS [25], a distributed sensor network simulator designed for high fidelity and scalable performance. DiSenS provides sensor network applications an execution environment as "close" to real deployment as possible. DiSenS is also able

to simulate a sensor network with hundreds of nodes in real time speed using computer clusters. In this section, we briefly introduce the design aspects of DiSenS that are relevant to the implementation of S^2DB . The complete discussion and evaluation of DiSenS are in papers [25, 24].

3.1 Full System Device Simulation

The building blocks of DiSenS are full system device simulators, supporting popular sensor network devices, including iPAQ [9], Stargate [21] and Mica2/MicaZ motes [15]. In this paper, we confine our description to the functionality necessary for debugging mote applications. However, the same functionality is implemented for more complex devices such as the iPAQ and Stargate. A more full examination of debugging for heterogeneous sensor devices is the subject of our future work.

The mote device simulator in DiSenS supports most of the Mica2 and MicaZ hardware features, including the AVR instruction set, the ATmega128L microcontroller (memories, UARTs, timers, SPI and ADC, etc.), the on-board Flash memory, CC1000 (Mica2) and CC2420 (MicaZ) radio chips and other miscellaneous components (like sensor board, LEDs, etc.).

The core of the device simulator is a cycle-accurate AVR instruction emulator. The instruction emulator interacts with other hardware simulation components via memory mapped I/O. When an application binary is executed in the simulator, each machine instruction is fed into the instruction emulator, shifting the internal representation of hardware states accordingly and faithfully. Asynchronous state change is modelled as events. Events are scheduled by hardware components and kept in an event queue. The instruction emulator checks the event queue for each instruction execution, triggering timed events. The collection of simulated hardware features is rich enough to boot and execute unmodified binaries of TinyOS [8] and most sensor network applications, including Surge, TinyDB [14] and Deluge [4]. By correctly simulating hardware components, the device simulator ensures the cycle accuracy, providing the basis of faithful simulation of a complete sensor network.

The full system device simulator in DiSenS also presents extension points or "hooks" for integrating power and radio models. This extensible architecture provides a way to support the development of new models and to trade simulation speed for level of accuracy. For debugging, this extensibility enables developers to test applications with different settings. For example, radio models representing different environments (like outdoor, indoor, etc.) can be plugged in to test applications under different circumstances.

In its default configuration, DiSenS incorporates an accurate power model from [12], a simple linear battery model, a basic lossless radio model, and a simple parameterized statistical model. The structure of the system, however, incorporates these models as modules that can be replaced with more sophisticated counterparts.

3.2 Scalable Distributed Simulation

DiSenS's ability to simulate hundreds of mote devices using distributed cluster computing resources is its most distinctive feature. This level of scalability makes it possible to experiment with large sensor network applications before they are actually deployed and to explore reconfiguration options "virtually" so that only the most promising need to be investigated *in situ*. As a debugging tool, DiSenS's scalability allows developers to identify and correct problems associated with scale. For example, a data sink application may work well in a network of dozens of nodes, but fails when the network size increases to hundreds, due to the problems such as insufficient queue or buffer size. Even for small scale network, the scalability is useful because it translates into simulation speed, and thus debugging efficiency.

DiSenS achieves its scalability by using a simple yet effective synchronization protocol for radio simulation and applying automatic node partition algorithms to spread the simulation/emulation workload across machines in a computer cluster. In DiSenS, sensor nodes are simulated in parallel, each running in its own operating system thread and keeping its own virtual clock. Sensor nodes interact with each other only in the radio transmission, during which radio packets are exchanged. The radio interaction of sensor nodes can be abstracted into two operations: read radio channel and write radio channel. The analysis [25] shows that only when a node reads radio channel, it needs to synchronize its clock with its neighbors (i.e., potential radio transmitters in its radio range). This ensures that each receiving node receives all the packets it is supposed to receive. A primitive called *wait_on_sync* is introduced to perform this synchronization, which forces the caller to wait for neighbor nodes to catch up with its current clock time. To implement this protocol, each node also has to keep its neighbors updated about its clock advance by periodically sending out its current clock time. A more detailed description and analysis of this protocol is in [25].

To utilize distributed computing resources, DiSenS partitions nodes into groups, each simulated on one machine within a cluster. Communication between sensor nodes assigned to the same machine is via a shared-memory communication channel. However, when motes assigned to distinct machines communicate, that communication and synchronization must be implemented via a message pass between machines. Due to the relatively large overhead of remote synchronization via message passing (caused by network latency), partitioning of simulated nodes to cluster machines plays an important role in making the ensemble simulation efficient.

To address this problem, graph-partitioning algorithms, originally developed for tightly-coupled data-parallel high-performance computing applications, are employed. DiSenS uses a popular partitioning package [7] to partition nodes nearly optimally.

Our S^2DB debugging tool is built upon DiSenS , whose design has huge impact on how the debugging facilities that we have implemented, including both advantages and limitations. In the next 3 sections, we'll discuss how DiSenS interacts with S^2DB to support both conventional and novel debugging techniques.

4. DEBUGGING INDIVIDUAL DEVICES

S^2DB was first built as a conventional distributed debugger on the DiSenS simulator. Each group of sensor nodes has a standalone debugging proxy waiting for incoming debugging commands. A debugger console thus can attach to each individual sensor node via this group proxy and perform debugging operations. The basic S^2DB includes most functions in a conventional debugger, like state (register and memory) checking, break points and step execution, etc.

In this section, we discuss how we exploit the potential of a simulation environment to devise novel techniques for debugging single sensor devices.

4.1 Debugging Point

Debugging is essentially a process of exposing program's internal states relevant to its abnormal behavior and pinpointing the cause. Visibility of execution states is a determining factor of how difficult the debugging task is. Building upon a full system simulator for each device gives S^2DB a great potential to expose time synchronized state.

Conventional debuggers essentially manipulate three states of a program: register, memory and program counter (PC). Simulators

	Component	Parameters	Value	Interrupt	Watchable	Overhead
PC (*pc*)	microcontroller	none	Int	No	Yes	Large
Register (*reg*)	microcontroller	address	Int	No	Yes	Large
Memory Read (*mem_rd*)	SRAM	address	Boolean	No	Yes	Small
Memory Write (*mem_wr*)	SRAM	address	Boolean	No	Yes	Small
Memory (*mem*)	SRAM	address	Int	No	Yes	Small
Flash Access (*flash_access*)	Flash	command, address	Boolean	No	Yes	Small
Flash (*flash*)	Flash	address	Int	No	Yes	Small
Power Change (*power*)	Power Model	none	Float	No	Yes	Small
Timer Match (*timer*)	Timers	none	Boolean	Yes	No	Small
Radio Data Ready (*spi*)	SPI (radio)	none	Boolean	Yes	No	Small
ADC Data Ready (*adc*)	ADC (radio/sensor)	none	Boolean	Yes	No	Small
Serial Data Received (*uart*)	UART	none	Boolean	Yes	No	Small
Clock (*clock*)	Virtual	none	Int	No	Yes	Minimal
Radio Packet Ready (*packet*)	Radio Chip	none	Packet	No	Yes	Small
Program Defined (*custom*)	Virtual Debugging Hardware	ID	Int	No	Yes	Program defined

Table 1: The current set of debugging points in S^2DB .

can provide much more abundant state information, which may enable or ease certain debugging tasks. For example, to debug a TinyOS module that manages on-board flash memory, it is important for the internal buffers and flash pages to be displayed directly. It is straightforward for DiSenS but rather difficult in a conventional debugger, which has to invoke complex code sequence to access the flash indirectly.

We carefully studied the device states in DiSenS and defined a series of *debugging points*. A *debugging point* is the access point to one of the internal states of the simulated device. The device state that is exposed by a *debugging point* can then be used by the debugger for displaying program status and controlling program execution, e.g., break and watch, as that in a conventional debugger. In this sense, *debugging point*s have extended our debugger's capability of program manipulation.

Table 1 lists the current set of *debugging point*s defined in S^2DB. It is not a complete list since we are still improving our implementation and discovering more meaningful *debugging point*s. In the table, the first column shows the *debugging point* name and the abbreviated notation (in parentheses) used by the debugger console. The corresponding hardware component that a *debugging point* belongs to is listed in the second column. The third and fourth columns specify the parameters and return value of a *debugging point*. For example, the "memory" point returns the byte content by the given memory address. The fifth column tells whether a *debugging point* has an interrupt associated. And the sixth column specifies whether a watch can be added to the point. The last column estimates the theoretical performance overhead of monitoring a particular *debugging point*.

As we see in the table, the common program states interrogated by conventional debuggers, i.e. register, memory and program counter, are also generalized as *debugging point*s in S^2DB , listed as *reg*, *mem* and *pc*. For memory, we also introduced two extra *debugging point*s, *mem_rd* and *mem_wr*, to monitor the access to memory in terms of direction. Notice that *debugging point*s have different time properties: some are persistent while others are transient. In the memory case, the memory content, *mem*, is persistent, while memory accesses, *mem_rd* and *mem_wr*, are transient. They are valid only when memory is read or written.

Similarly, the on-board flash has two defined *debugging point*s: one for the page content (*flash*) and the other (*flash_access*) for the flash access, including read, program and erase. The *power* debugging point is used to access the simulated power state of the device, which may be useful for debugging power-aware algorithms.

Four important hardware events are defined as *debugging point*s: timer match event (*timer*), radio (SPI) data ready (*spi*), ADC data ready (*adc*) and serial data ready (*uart*). They are all transient and all related to an interrupt. These *debugging point*s provide a natural and convenient way to debug sensor network programs since many of these programs are event-driven, such as TinyOS and its application suite. As an example, if we want to break the program execution at the occurrence of a timer match event, we can simply invoke the command:

```
> break when timer() == true
```

In a more conventional debugger, a breakpoint is typically set in the interrupt handling code, the name of which must be known to the programmer. Furthermore, breaking on these event-based *debugging point*s is much more efficient than breaking on a source code line (i.e., a specific program address). This is because matching program addresses requires a comparison after the execution of each instruction while matching event-based *debugging point*s only happens when the corresponding hardware events are triggered, which occur much less frequently. We will discuss how to use *debugging point*s to set break conditions and their overhead in later this subsection.

The *clock* debugging point provides a way for accurate timing control over program execution. It can be used to fast forward the execution to a certain point if we know that the bug of our interest will not occur until after a period of time. It would be rather difficult to implement this in a conventional debugger since there is no easy way to obtain accurate clock timing across device subsystems.

It is also possible to analyze the states and data in the simulator to extract useful high-level semantics and use them to build advanced *debugging point*s. An example is the recognition of radio packet. The Mica2 sensor device uses the CC1000 radio chip, which operates at the byte level. Thus an emulator can only see the byte stream transmitted from/to neighbor nodes and not packet boundaries. For application debugging, however, it is often necessary to break program execution when a complete packet has been transmitted or received. A typical debugging strategy is to set a breakpoint in the radio software stack at the the line of code line that finishes a packet reception. However, this process can be both tedious and unreliable (e.g. software stack may change when a new image is installed), especially during development or maintenance of the radio stack itself. Fortunately, in the current TinyOS radio stack implementation, the radio packet has a fixed format. We implemented a tiny radio packet recognizer in the radio chip simulation code. A "radio packet ready" (*packet*) debugging point is defined to signal the state

when a complete packet is received. These extracted high-level semantics are useful because we can debug applications without relying on the source code, especially when the application binary is optimized code and it is hard to associate exact program addresses with specific source code line. However, discovering these semantics using low-level data/states is challenging and non-obvious (at least, to us) and as such continues to be a focus of our on-going research in this area.

4.1.1 Break Conditions Using Debugging Points

*Debugging point*s are used in a functional form. For example, if we want to print a variable X, we can use:

```
> print mem(X)
```

To implement conditional break or watch points, they can be included in imperatives such as:

```
> break when flash_access(erase, 0x1)
```

which breaks the execution when the first page of the flash is erased. It is also possible to compose them:

```
> break when timer() && mem(Y) > 1
```

which breaks when a timer match event occurs and a state variable Y, like a counter, is larger than 1.

The basic algorithm for monitoring and evaluating break conditions is as follows. Each *debugging point* maintains a monitor queue. Whenever a break point is set, its condition is added to the queue of every *debugging point* that is used by the condition. Every time the state changes at a *debugging point*, the conditions in its queue is re-evaluated to check whether any of them is satisfied. If so, one of the break points is reached and the execution is suspended. Otherwise, the execution continues.

Note that the monitoring overhead varies for different *debugging point*s revealing the possibility for optimization of the basic condition evaluating algorithm. The monitoring overhead is determined by the frequency of state change at a *debugging point*. Obviously, *pc* has the largest overhead because it changes at each instruction execution. Event related debugging points have very low overhead since hardware events occur less frequently. For example, the timer event may be triggered for every hundreds of cycles. *Clock* logically has a large overhead since it changes every clock cycle. However, in simulation, clock time is checked anyway for event triggering. By implementing the clock monitoring itself as an event, we introduce no extra overhead for monitoring *clock* debugging point.

Thus we are able to optimize the implementation of condition evaluation. For example, considering the following break condition:

```
> break when pc() == foo && mem(Y) > 1
```

Using the basic algorithm, the overhead of monitoring the condition is the sum of *pc*'s overhead and *mem*'s overhead. However, since the condition is satisfied when both *debugging point*s match their expression, we could only track *mem* since it has smaller overhead than *pc*. When *mem* is satisfied, we then continue to check *pc*. In this way, the overall overhead reduces.

Now we present the general condition evaluation algorithm. Given a condition as a logic expression, C, it is first converted into canonical form using product of maxterms:

$$C = t_1 \wedge t_2 \wedge ... \wedge t_n \quad (1)$$

where t_i is a maxterm. The overhead function f_{ov} is defined as the total overhead to monitor all the *debugging point*s in a maxterm. Then we sort the maxterms by the value of $f_{ov}(t_i)$ in incremental order, say, t_{k_1}, ..., t_{k_n}. We start the monitoring of C first using maxterm t_{k_1} by adding C to all the *debugging point*s that belong to t_{k_1}. When t_{k_1} is satisfied, we re-evaluate C and stop if it is true. Otherwise, we remove C from t_{k_1}'s *debugging point*s and start monitoring t_{k_2}. If t_{k_n} is monitored and C is still not satisfied, we loop back to t_{k_1}. We repeat this process until C is satisfied. If C is unsatisfiable, this process never ends.

*Debugging point*s give us powerful capability to debug sensor network programs at a level between the hardware level and the source-code level. However, a direct instrumentation of the source code is sometimes easiest and most straight-forward debugging method. The typical methodology for implementing source-level instrumentation is to use print statements to dump states. Printing, however, can introduce considerable overhead that can mask the problem being tracked.

In S^2DB we include an instrumentation facility based on virtual registers that serves the same purpose with reduced overhead. We introduce our instrumentation facility in the next subsection.

4.2 Virtual Hardware Based Source Code Instrumentation

Sensor devices are usually resource-constrained, lacking the necessary facility for debugging in both hardware and software. On a Mica2 sensor device, the only I/O method that can be used for display internal status by the program is to flash the three LEDs, which is tedious and error-prone to decode. DiSenS faithfully simulates the sensor hardware, thus inheriting this limitation. Because we insist that DiSenS maintain binary transparency with the native hardware it emulates, the simulated sensor network program is not able to perform a simple "printf".

To solve this problem, we introduce three virtual registers as an I/O channel for the communication between application and simulator. Their I/O addresses are allocated in the reserved memory space of ATmega128L. Thus the access of these virtual registers will not affect the correct functioning of other components. Table 2 lists the three registers and their functions.

Address	Name	Functionality
0x75	VDBCMD	Command Register
0x76	VDBIN	Input Register
0x77	VDBOUT	Output Register

Table 2: Virtual registers for communication between application and simulator.

The operation of virtual registers is as follows: an application first issues a command in the command register, *VDBCMD*; then the output data is transferred via *VDBOUT* register and the input data is read from em VDBIN register. The simplest application of virtual registers is to print debugging messages by first sending a "PRINT" command and then continuously writing the ASCII characters in a string to the *VDBOUT* register until a new line is reached. On the simulator side, whenever a command is issued, it either reads from the *VDBOUT* register or sending data to *VDBIN*. In the print case, when the simulator gets all the characters (ended by a new line), it will print out on the host console of the simulating machine.

A more advanced use of virtual registers is to control a *debugging point*. We term this combination of virtual registers and debugging points a *program defined debugging point* (*custom*, as listed in the last line of Table 1). The state of a *custom* debugging point is generated by the instrumentation code in the program. To do so, the instrumentation code first sends a "DEBUG" command to the *VD-*

BCMD register, then outputs the debugging data on the *VDBOUT* register, in the form of a tuple, $< id, value >$. The id is used to identify the instrumentation point in the source code and the $value$ is any value generated by the instrumented code. If there is a break condition registered at this point, it will be checked against the tuple and execution will stop when it is matched. As an example, if we want to break at the $10th$ entry of a function, we can instrument the function and keep a counter of entries. Every time the counter changes, we output the counter value via virtual registers. The break condition will be satisfied when the value equals to 10.

To make it easy to use, we developed a small C library for accessing the virtual registers transparently. Developers can invoke accessing functions on these registers by simply calling the C APIs, for example, in a TinyOS program.

Instrumentation via the virtual registers has the minimal intrusiveness on application execution. When generating a *debugging point* event by sending a $< id, value >$ tuple, only three register accesses are needed if both values in the tuple are 8-bits each (one for the command and two for the data).

5. COORDINATED PARALLEL DEBUGGING OF MULTIPLE DEVICES

DiSenS's scalability and performance enables S^{2}DB to debug large cooperating ensembles of sensors as a simulated sensor network deployment. Like other debuggers, S^{2}DB permits its user to attach to and "focus" on a specific sensor while the other sensors in the ensemble execute independently. However, often, more systematic errors emerge from the interactions among sensor nodes even when individual devices and/or applications are functioning correctly. To reveal these kinds of errors, developers must be able to interrogate and control multiple sensor devices in a coordinated way.

Debugging a program normally involves displaying program status, breaking program execution at arbitrary points, step-executing, etc. By extending this concept to parallel debugging, we want to be able to:

1. Display the status of multiple devices in parallel;
2. Break the execution of multiple devices at certain common point;
3. Step-execute multiple devices at the same pace.

The first and third items in the above "wish list" are easy to implement in a simulation context. S^{2}DB can simply "multicast" its debugging commands to a batch of sensor nodes once their execution stop at a certain common point. As for the second item, since DiSenS is, in effect, executing multiple parallel simulations without a centralized clock, implementing a time-correlated and common breakpoint shares the same coordination challenges with in parallel debugging counterpart.

The simplest form of coordinated break is to pause the execution of a set of involved nodes at a specific virtual time, T:

```
> :break when clock() == T
```

where the colon before "break" indicates that it is a batch command and will be sent to all the nodes in a global batch list (maintained by other commands).

It is necessary to review DiSenS's synchronization mechanism first. We summarize the major rules as follows:

1. A node that receives or samples radio channel must wait for all its neighbors to catch up with its current clock time;

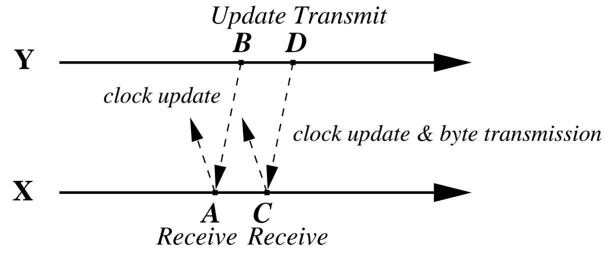

Figure 1: Illustration of synchronization between sensor nodes in DiSenS . Dashed arrows indicate the update and transmission messages. $A < B < C < D$**.**

2. All nodes must periodically broadcast their clock updates to neighbors;
3. Before any wait, a node must first send its clock update (to avoid loop waiting);
4. Radio byte is always sent with a clock update at the end of its last bit transmission.

Figure 1 illustrates the process. At time point A, node X receives. It first sends an update of its clock and wait for its neighbor Y (rule 3 & 1). Y runs to B and sends its clock update (rule 2), which wakes up X. X proceeds to C and receives again. Y starts a byte transmission at B. At D, the last bit transmitted and so the byte along with a clock update is sent to X (rule 4). X receives the byte, knowing Y passes its current time, and proceeds.

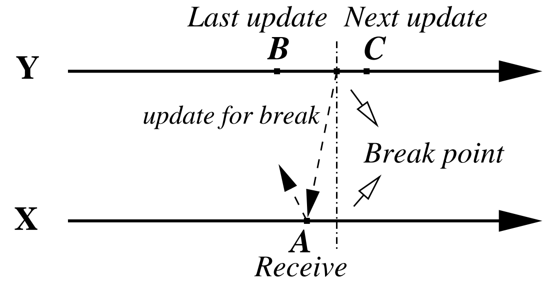

Figure 2: Break at a certain point of time. Dashed arrows indicate the update and transmission messages. $B < A < C$**.**

Now, let's see what happens when we ask multiple nodes to stop at the same time. Figure 2 shows one case of the situation. X receives at time A and sends an update and waits for Y. Y sends an update at B. Its next update time is C. But we want to break at a point before C but after A. Since Y breaks (thus waits), it sends an update (rule 3). X receives the update, wakes up, proceeds to the break point and stops. Now both X and Y are stopped at the same time point.

In Figure 3, the situation is similar to the case in Figure 2. The difference is that now the break point is in the middle of a byte transmission for Y. Y can not just send an update to X and let X proceeds to break point as in Figure 2. because if X gets the update from Y, it believes Y has no byte to send up to the break point and will continue its radio receiving logic. Thus the partial byte from Y is lost. This problem is caused by rule 4. We solve it by relaxing the rule: Whenever a node is stopped (thus it waits) in the middle of a byte transmission, the byte is *pre-transmitted* with the clock update. We can do this because mote radio always transmits in byte unit. Once a byte transmission starts, we already

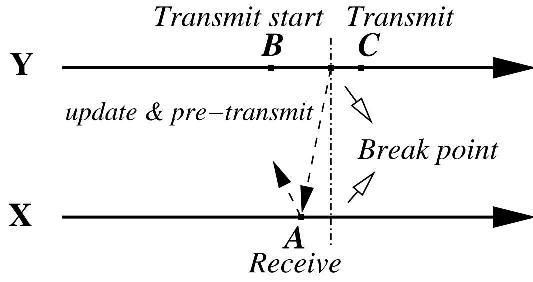

Figure 3: Extension to the synchronization protocol: *pre-transmission*. Dashed arrows indicate the update and transmission messages. $B < A < C$**.**

know its content. Also, in DiSenS , each byte received by a node is buffered with timestamp. It will be processed only when the time matches the local clock. With this relaxed rule, we are now able to stop multiple sensor nodes at the same virtual time.

The next question is how to perform a conditional break on multiple nodes. Notice that we cannot simply implement:

```
> :break when mem(X) > 3
```

because it asks the nodes to break independently. Whenever a node breaks at some point, other nodes with direct or indirect neighborhood relationship with it will wait at indeterminate points due to the synchronization requirement. Whether they all satisfy the condition is not clear. A reasonable version of this command is:

```
> :break when *.mem(X) > 3
```

or

```
> :break when node1.mem(X) > 3
          && ... && nodek.mem(X) > 3
```

which means "break when $X > 3$ for all the nodes". In the general form, we define a *coordinated break* as a break with condition $cond_1 \wedge cond_2 \wedge ... \wedge cond_k$, where $cond_i$ is a logic expression for node i.

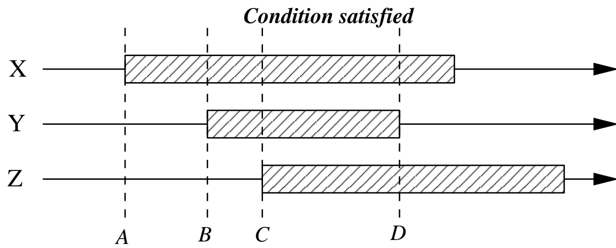

Figure 4: Coordinated break. The shaded boxes represent the time range during which a local condition is satisfied. Between C **and** D**, the global condition is satisfied.** $A < B < C < D$**.**

Figure 4 illustrates the meaning of this form of breakpoint. The shaded boxes are the time period during which the local condition for a node is satisfied. In Figure 4, the global condition, i.e. $cond_x \wedge cond_y \wedge cond_z$, is satisfied between time C and D. Time C is the exact point where we want to break.

Before we present the algorithm that implements *coordinated break*, we need to first introduce a new synchronization scheme. We call it *partially ordered synchronization*. By default DiSenS implements *peer synchronization*: all the nodes are running in arbitrary order except synchronized during receiving or sampling. The new scheme imposes a partial order. In this scheme, a node master is first specified. Then all the other nodes proceed by following the master node. That is, at any wall clock time t_{wall} (i.e., the real world time), for any $node_i$, $clock_i <= clock_{master}$.

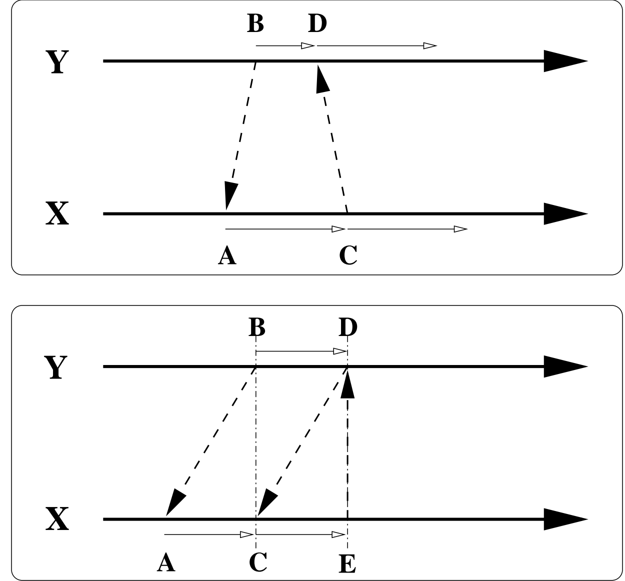

Figure 5: TOP: *peer synchronization* in DiSenS . $A < B < D < C$**. BOTTOM: *partially ordered synchronization* for S^{2}DB.** $A < C < E$**,** $B = C$**,** $D = E$**. Dashed arrows indicate the update and transmission messages. (Some update messages are omitted)**

Figure 5 illustrates the two synchronization schemes. The top part shows DiSenS's *peer synchronization* scheme. Node X waits at A. Y sends update at B and wakes X. Then Y waits at D, waken by X's update at C. X and Y proceed in parallel afterwards. The bottom part shows S^{2}DB's *partially ordered synchronization* scheme. Here Y is the master. X first waits at A. Y sends its update at B. X receives the update and runs to the updated point, which is C (=B). Then X waits again. When Y runs to D and sends update. X can proceed to E (=D). If Y needs to wait to receive, X will wake it up when X reaches E according to rule 3. Obviously, in this scheme, X always follows Y.

Now we can give our algorithm for *coordinated break*. Using Figure 4 as the example, we first designate X as the master. At point A, X's condition is satisfied. X stops at A. Since Y and Z follow X, they all stop at A. Then we choose the next node as the new master, whose condition is not satisfied yet. It is Y. X and Z follow Y until Y reaches B. Next, similarly, we choose Z as new master. At time C, we find $cond_x \wedge cond_y \wedge cond_z = true$. We break the execution and C is exactly our break point. In this algorithm, the aforementioned *pre-transmission* also plays an important role in that it enables us to stop all nodes at the same time point precisely.

Coordinated break, however, does not work with arbitrary conditions. Consider the case where the local conditions in Figure 4 are connected by injunction instead of conjunction. The break point now should be at time A. Since we are not able to predict which node will first satisfy its condition, it is not possible for us to stop all the nodes together at time A unless we synchronize all the nodes cycle by cycle, which would limit the scalability and the performance significantly. For the same reason, we can not set up multiple *coordinated break points*. We reiterate that these limita-

tions are a direct result of our desire to scale DiSenS and to use S^{2}DB on large-scale simulated networks. That is, we have sacrificed generality in favor of the performance gained through parallel and distributed-memory implementation.

Although the generality of *coordinated break* is limited, it is still useful in many situations. For example, for a data sink application, we may want to determine why data is lost when a surge of data flows to the sink node. In this case, we would break the execution of the sink node based on the condition that its neighbor nodes have sent data to it. Then we step-execute the program running on the sink node to determine why the data is being lost. To implement the condition of data sent on neighbor nodes we can simply use source code instrumentation exporting a *custom* debugging point. Thus this example also illustrates how the single-device debugging features discussed in the previous section can be integrated with the group debugging features.

6. FAST TIME TRAVELING FOR REPLAYABLE DEBUGGING

Even with the ability to perform coordinated breakpoints, the normal debugging cycle of break/step/print is still cumbersome when the complete sensor network is debugged, especially if the size of network is large. The high level nature of some systematic errors requires a global view of the interactions among sensor nodes. An alternative model for debugging sensor networks is:

- A simulation is conducted with tracing. Trace log is analyzed to pinpoint the anomaly.
- Quickly return to the point when the anomaly occurs to perform detailed source code level debugging.

To achieve this, we need to trace the simulation and restore the state of network at any point in the trace. The *debugging point*s and virtual hardware based instrumentation discussed in section 4 can be used to trace the simulation in a way similar to [23]. In this section, we present the S^{2}DB's design of fast time traveling, which enables the restoration of network states.

The basic mechanism required to implement time traveling is a periodic checkpoint. A checkpoint of a simulation is a complete copy of the state of the simulated sensor network. DiSenS is an object oriented framework in representing device components. When a checkpoint is initiated, the state saving function is invoked first at the highest level "machine" object. Recursively, the subcomponents in the "machine" invoke their own state saving functions. The saved state is comprised of registers, memories (SRAM, EEPROM, etc.) and auxiliary state variables in each component. It also includes some simulation related states. For example, we need to save the event queue content, the received radio byte queue in the radio model and the status of the power model, etc. The complete binary of the state is saved into a timestamped file. The result checkpoint file for DiSenS has a size of 4948 bytes, mostly comprised of SRAM (4KB) content.

Checkpoint for the on-board flash has to be handled differently. Motes have a 512KB flash chip used for sensor data logging and in-network programming. If flash content is saved as other components, the checkpoint file will be as large as over half megabyte, which is 128 times larger than the one without flash. So if flash is also saved in a snapshot way, it is both extremely space and time inefficient for a large scale sensor network. We solve this problem by saving flash operations in a log file. Since most sensor network applications use flash infrequently and flash content is updated in page unit, the overhead of saving log is much smaller than saving flash snapshots. Notice that the flash buffers have to be saved in the snapshot checkpoint file.

Once a simulation is finished, we have a set of snapshot checkpoints and a continuous flash log. Given an arbitrary time point T, to restore the state of system includes the following steps:

1. *Restore*: find the latest checkpoint CP that is prior to T and load the snapshot checkpoint file;
2. *Replay*: if flash is used, replay the flash operation log up to CP's time;
3. *Re-run*: start from CP, re-run the simulation until time T.

Checkpoints can also be initiated by methods other than the need to take a periodic snapshot. For example, under S^{2}DB a break point can be associated with a checkpoint so that once the execution breaks, a checkpoint is generated. Thus, a developer can move between the checkpoints to find the exact point when error occurs during a replayed simulation. Checkpoint can also be initiated by a *debugging point*, especially *custom* debugging points. By allowing checkpoint to be triggered in conjunction with *debugging points* S^{2}DB integrates the replay and state-saving capabilities needed to efficiently re-examine an error condition with the execution control over state changes.

7. EVALUATION

Since S^{2}DB is built upon DiSenS , its performance is highly dependent on DiSenS itself. We begin this section by focusing on the the performance of DiSenS simulation/emulation and then show the overhead introduced by various S^{2}DB debugging facilities. All experiments described in this section are conducted using a 16-node cluster in which each host has dual 3.2GHz Intel Xeon processors with 1GB memory. The hosts are connected via switched gigabit Ethernet. To make fair comparison, we use the same sensor network application *CntToRfm* for evaluation.

7.1 Performance of DiSenS

For brevity, we present only the typical simulation speed of DiSenS on the cluster. A more thorough examination of scalability and performance under different configurations can be found in paper [25].

Figure 6 shows the performance achieved by DiSenS when simulating various numbers of nodes on the cluster in both 1-D and 2-D topologies. In the figure, the X axis shows the total number of nodes simulated. The Y-axis is the normalized simulation speed (compared to real time speed on hardware). For the 1-D topology, all nodes are oriented on a straight line, 50 meters apart (assuming the maximal radio range is 60 meters). For the 2-D topology, nodes are arranged in a square grid. Again the distance between two nodes is 50 meters. Both performance curves are very close except in the middle part, where 2-D topology has slightly worse performance.

The simulation speed drops noticeably from 1 to 4 nodes but then the speed curve keeps flat until 128 nodes are simulated. After that, the speed decreases linearly. The transition from flat to linear decrement is because there is not enough computing resources within the cluster (16 hosts).

To summarize the results from [25], DiSenS is able to simulate one mote **9** times faster than real time speed, or **160** nodes at near real time speed, or **2048** nodes at nearly a tenth of real time speed.

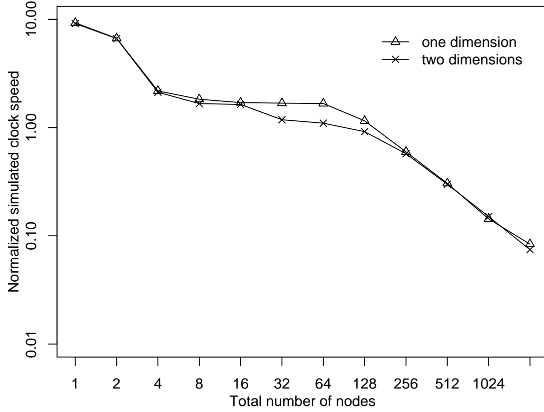

Figure 6: DiSenS simulation performance in 1-D and 2-D topologies. X-axis is total number of nodes simulated. Y-axis is normalized simulation speed (compared to execution speed on real device).

7.2 Performance of a Break Condition on a Single Device

We first evaluate the cost of monitoring debugging points in single-device debugging. Not all the listed (in Table 1) debugging points are evaluated since the overhead for some of them is application dependent.

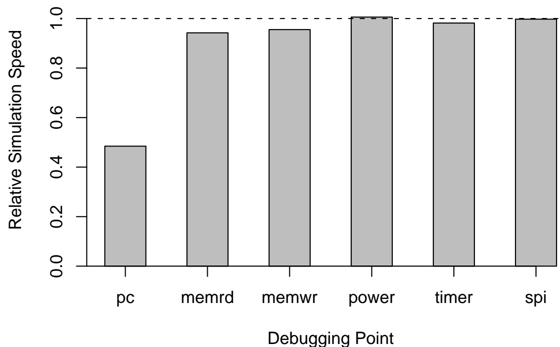

Figure 7: Relative simulation speed for various debugging points. X-axis shows the name of debugging points. Y-axis is the ratio to original simulation speed (without monitoring debugging points).

Figure 7 gives the relative simulation speed of evaluating various debugging points. For each one, we set a break condition using the debugging point and run the simulation. The result shows that *pc* has the largest overhead since the PC change occurs for every instruction execution. Memory related debugging points has less overhead. *Power* and event-based debugging points have the least overhead since their states change infrequently.

7.3 Performance of a Coordinated Break Condition with Multiple Devices

We evaluate the overhead of monitoring the coordinated break condition in this subsection. We run our experiments with a 2-D 4×4 grid of sensor nodes, distributed in 4 groups (hosts).

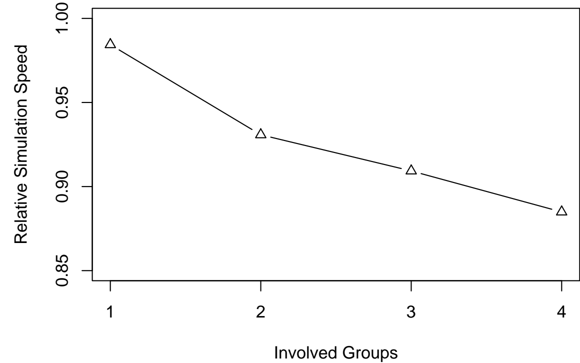

Figure 8: Relative simulation speed of monitoring a coordinated break condition for multiple devices. X-axis is the number of groups (hosts) involved. Y-axis is the ratio to original simulation speed (without condition monitoring).

Figure 8 shows the speed ratio between the simulation with monitoring and without. When the group number is 1, only nodes in one group are involved in the break condition. For group number 2, nodes in both groups are used in the break condition, and so on. The speed ratio curve drops when the number of groups increases.

The overhead of monitoring coordinated break condition is mostly due to the extra synchronization cost introduced by the new *partially ordered synchronization* scheme. Obviously, when more nodes (especially remote nodes) involved, the simulation overhead is higher.

7.4 Performance of Checkpointing for Time Traveling

We evaluate the overhead of checkpointing in four configurations: 1×1, 4×1, 16×1 and 4×4, where $x \times y$ means x nodes per group and y groups. For each one, we vary the checkpoint interval from $1/8$ up to 4 virtual seconds.

Figure 9 shows the relative simulation speed when checkpointing the system periodically. Naturally, the overhead increases when checkpointing more frequently. It is hard to distinguish the single-group curves since their differences are so small. In general, checkpointing in multi-group simulation seems to have larger overhead than single-group. However, the checkpoint overhead is relatively small. All four curves lie above 96% of original simulation speed, which translates to less than 4% of overhead. This result encourages us to use time-traveling extensively in debugging. Developers thus can always return to the last break point or a previous trace point with little cost.

To summarize, we find that most of the new debugging facilities we have introduced with S^{2}DB have small overhead (less than 10%). As a result, we are able to debug sensor network applications using tools that operate at different levels of abstraction while preserving the high performance and scalability provided by DiSenS.

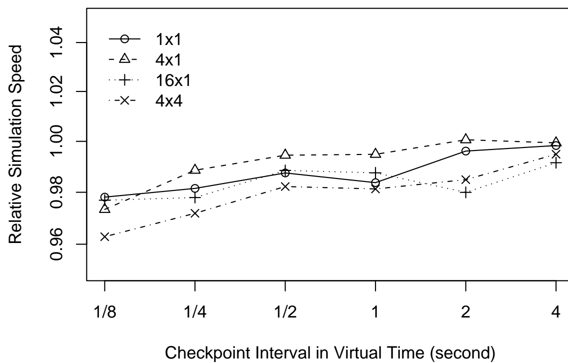

Figure 9: Relative simulation speed for checkpointing. X-axis is the interval between two checkpoints (in terms of virtual clock time of mote device). Y-axis is the ratio to original simulation speed (without checkpointing).

8. CONCLUSION

S^{2}DB is an efficient and effective sensor network debugger based on DiSenS, a scalable distributed sensor network simulator. S^{2}DB makes four innovations to the conventional debugging scheme at different levels of abstraction. For effective debugging of single sensor devices, *debugging point*s are introduced for the interrogation of all interested subsystem states in a sensor device. To facilitate source level tracing and instrumentation, we extend the simulated sensor device hardware with a set of virtual registers providing a way for the communication between simulator and simulated program. At the multi-device level, we discuss the implementation of coordinated break condition in the distributed framework. This new type of break condition enables coordinated parallel execution control of multiple sensor devices. A time traveling facility is introduced for the network level debugging, used for rapid error site restoration when working with sensor network trace analysis. Overall, these debugging features impose overhead of less than 10% (generally) to DiSenS, and thus enable efficient debugging of large scale sensor networks.

S^{2}DB is still an ongoing project that we think to make it a comprehensive debugging tool for sensor networks, there is still a lot of work to do. The most imperative task is to design and implement a graphic user interface for intuitive and productive debugging. We are planning to build a plugin in the famous Eclipse [5] development environment, which controls the debugging and simulation functions in S^{2}DB and DiSenS. We are also interested in incorporating the debugging needs according to people's experiences in sensor network development and discovering new debugging techniques, especially at the network level.

9. REFERENCES

[1] Atmel. AVR JTAG ICE User Guide. 2001. `http://www.atmel.com/dyn/resources/prod_documents/DOC2475.PDF`.

[2] Atmel's AVR JTAG ICE. `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2737`.

[3] C. Buschmann, D. Pfisterer, S. Fischer, S. P. Fekete, and A. Kröller. SpyGlass: taking a closer look at sensor networks. *In the Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 301–302, 2004. New York, NY, USA.

[4] A. Chlipala, J. W. Hui, and G. Tolle. Deluge: Dissemination Protocols for Network Reprogramming at Scale. *Fall 2003 UC Berkeley class project paper*, 2003.

[5] Eclipse: an extensible development platform and application frameworks for building software. `http://www.eclipse.org`.

[6] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks. *USENIX Technical Conference*, 2004.

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

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

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

[10] Boundary-Scan (JTAG) test and in-system programming solutions (IEEE 1149.1). `http://www.jtag.com/main.php`.

[11] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. *In the Proceedings of USENIX Annual Technical Conference 2005*, Apr. 2005. Anaheim, CA.

[12] O. Landsiedel, K. Wehrle, and S. Gtz. Accurate Prediction of Power Consumption in Sensor Networks. *In Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, May 2005. Sydney, Australia.

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

[14] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. *In Proceedings of SIGMOD 2003*, June 2003.

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

[16] MOTE-VIEW Monitoring Software. `http://www.xbow.com/Products/productsdetails.aspx?sid=88`.

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

[18] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the Sensor Network Debugger. *In the Proceedings of 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys '05)*, Nov. 2005. San Diego, California.

[19] N. Ramanathan, E. Kohler, and D. Estrin. Towards a debugging system for sensor networks. *International Journal of Network Management*, 15(4):223–234, 2005.

[20] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. *In the Proceedings of USENIX Annual Technical Conference 2004*, June 2004. Boston, MA.

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

[22] Surge Network Viewer. `http://xbow.com/Products/productsdetails.aspx?sid=86`.

[23] B. Titzer and J. Palsberg. Nonintrusive Precision Instrumentation of Microcontroller Software. *In the Proceedings of ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, June 2005. Chicago, Illinois.

[24] Y. Wen, S. Gurun, N. Chohan, R. Wolski, and C. Krintz. SimGate: Full-System, Cycle-Close Simulation of the Stargate Sensor Network Intermediate Node. *In Proceedings of International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (IC-SAMOS)*, 2006. Samos, Greece.

[25] Y. Wen, R. Wolski, and G. Moore. DiSenS: Scalable Distributed Sensor Network Simulation. Technical Report CS2005-30, University of California, Santa Barbara, 2005.