Troubleshooting interactive complexity bugs in wireless sensor networks using data mining techniques

Mohammad Maifi Hasan Khan,Hieu Khac Le,Hossein Ahmadi,
Tarek F. Abdelzaher
and
Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign

This article presents a tool for uncovering bugs due to interactive complexity in networked sensing applications. Such bugs are not localized to one component that is faulty, but rather result from complex and unexpected interactions between multiple often individually non-faulty components. Moreover, the manifestations of these bugs are often not repeatable, making them particularly hard to find, as the particular sequence of events that invokes the bug may not be easy to reconstruct. Because of the distributed nature of failure scenarios, our tool looks for *sequences* of events that may be responsible for faulty behavior, as opposed to localized bugs such as a bad pointer in a module. We identified several challenges in applying discriminative sequence mining for root cause analysis when the system fails to perform as expected and presented our solutions to those challenges. We also presented two alternatives schemes, namely, two stage mining and the progressive discriminative sequence mining to address the scalability challenge. An extensible framework is developed where a front-end collects runtime data logs of the system being debugged and an offline back-end uses frequent discriminative pattern mining to uncover likely causes of failure. We provided three case studies where we applied our tool successfully to troubleshoot the cause of the problem. We uncovered a kernel-level race condition bug in the LiteOS operating system and a protocol design bug in the directed diffusion protocol. We also presented a case study of debugging a multichannel MAC protocol that was found to exhibit corner cases of poor performance (worse than single channel MAC). The tool helped uncover event sequences that lead to a highly degraded mode of operation. Fixing the problem significantly improved the performance of the protocol. Finally, we provided a detailed analysis of tool overhead in terms of memory requirements and impact on the running application.

Key Words: Protocol debugging, Distributed automated debugging, Wireless sensor networks

1. INTRODUCTION

DustMiner [Khan et al. 2008b] is a diagnostic tool that leverages an extensible framework for uncovering root causes of failures and performance anomalies in wireless sensor network applications in an automated way. This paper presents the design and implementation of *Dustminer* along with three case studies of real life failure diagnosis scenarios. The goal of this work is to further contribute to automating the process of debugging, instead of relying only on manual efforts, and hence reduce the development time and effort significantly.

Developing wireless sensor network applications still remains a significant challenge and a time consuming task. To make the development of wireless sensor network applications easier, much of the previous work focused on programming abstractions [Madden et al. 2005; Cheong et al. 2003; Luo et al. 2006; Levis and Culler 2002; Gay et al. 2003]. Most wireless sensor network application developers would agree, however, to the fact that most of the development time is spent on debugging and troubleshooting the current code, which greatly reduces productivity.

Early debugging and troubleshooting support revolved around testbeds [Ertin et al. 2006; Werner-Allen et al. 2005], simulation [Levis et al. 2003; Wen et al. 2007; Wen and Wolski] and emulation environments [Polley et al. 2004; Girod et al. 2004]. Source level debugging tools [Yang et al. 2007; Whitehouse et al. 2006] have greatly contributed to the convenience of the troubleshooting process. They make it easier to zoom in on sources of errors by offering more visibility into run-time state. Unfortunately wireless sensor network applications often fail not because of a single node coding error but as a result of improper interaction between components. Such interaction may be due to some protocol design flaw (e.g., missed corner cases that the protocol does not handle correctly) or unexpected artifacts of component integration. Interaction errors are often non-reproducible since repeating the experiment might not lead to the same corner-case again. Hence, in contrast to previous debugging tools, in this paper, we focus on finding (generally non-deterministically occurring) bugs that arise from interactions among seemingly individually sound components.

The main approach of *Dustminer* is to log many different types of events in the sensor network and then analyze the logs in an automated fashion to extract the sequences of events that lead to failure. These sequences shed light on what caused the bug to manifest making it easier to understand and fix the root cause of the problem.

Our work extends a somewhat sparse chain of prior attempts at diagnostic debugging in sensor networks. Sympathy [Ramanathan et al. 2005] is one of the earliest tools in the wireless sensor networks domain that addresses the issue of diagnosing failures in deployed systems in an automated fashion. Specifically, it infers the identities of failed nodes or links based on reduced throughput at the base station. SNTS [Khan et al. 2007] provides a more general diagnostic tool that extracts conditions on current network state that are correlated with failure, in hopes that they may include the root cause. A diagnostic simulator (an extension to TOSSIM) is described in one of our earlier work [Khan et al. 2008a]. Being a simulator extension, it is not intended to troubleshoot deployed systems. It is based on frequent pattern mining (to find event patterns that occur frequently when the bugs manifest). Unfortunately, the cause of a problem is often an infrequent pattern; a single "bad" chain of events that leads to many problems.

Our prior work [Khan et al. 2008b] identified several limitations in the Apriori algorithm and addressed those limitations. More specifically, our prior work addressed limitations such as preventing generation of false patterns using dynamic search window, elimination of redundant subsequences and presented two stage mining for scalability enhancements. In our prior work, we extended the diagnostic capability by implementing the two stage mining [Khan et al. 2008b], an automated discrim-

inative sequence analysis technique that relies on two separate stages to find root causes of the problems or performance anomalies in sensor networks; the first stage identifies frequent patterns correlated to failure. The second stage focuses on those patterns, correlating them with (infrequent) events that may have caused them, hence uncovering the true root cause of the problem. We apply this technique to identify the sequences of events that cause manifestations of interaction bugs. In turn, identifying these sequences helps the developer to understand the nature of the bug.

In this work, we extended our prior work [Khan et al. 2008b] as follows.

—Firstly, our prior approach does not consider the time differences between consecutive events as a feature while mining for discriminative sequences of events. This may cause the algorithm to fail to identify certain patterns that require to look at not only event types but also the timing of the events. We extended our algorithm so that it considers not only temporal order but also timing information while mining for discriminative patterns.

—Secondly, although our prior work can easily identify pair wise causality relationship (e.g., one message sent is followed by one acknowledgement), it fails to identify causality relationship when this is not true. For example, in many file systems it is often common practice to buffer data before actually writing to disk to minimize the number of disk accesses and to save energy [Yang et al. 2004]. In a correctly functioning system, it can be expected that multiple buffering operations will be followed by a single disk write operation. In such cases, our prior algorithm would fail to identify cases when the file system fails due to a missing disk write operation. We extended our prior work to handle such scenarios and explained further details in section 4.4.

—Finally, to address the scalability issue, we presented another alternative scheme, the progressive discriminative sequence mining (which is different than the earlier two stage mining) and is elaborated in section 5.2. To enhance the scalability, we exploited the idea of discriminative sequence mining explored in the data mining literature [Cheng et al. 2008; Cheng et al. 2007]. Earlier work in data mining [Cheng et al. 2007] showed that the upper bound for information gain of an event (i.e., the potential discriminative power of en event) increases monotonically with the support of that event. This implies that the events with lower support has less discriminative power in general. DDPMine algorithm [Cheng et al. 2008] showed how the upper bound estimation [Cheng et al. 2007] can be exploited during the frequent item set mining process for effective classification. Our presented progressive discriminative sequence mining algorithm takes a similar approach as DDPMine where it iteratively searches for discriminative patterns while looking for longer patterns at each stage. But the mechanism is different.

The DDPMine algorithm [Cheng et al. 2008] differs from our work in several ways. Firstly, they proposed an algorithm for item set mining which is a simpler problem than subsequence mining where the relative ordering of the events is critical. Secondly, in DDPMine [Cheng et al. 2008], the goal is to identify a set of item sets

that can collectively classify all the training instances. Intuitively, this approach searches for a combination of features that collectively can classify the input space. In contrary, our goal is to identify discriminative sequences of events (often more than one) where each discriminative sequence individually can distinguish between successful and failed executions.

Our tool is based on a front-end that collects data from the run-time system and a back-end that analyzes it. Both are plug-and-play modules that can be chosen from multiple alternatives. We identify the architectural requirements of building such an extensible framework and present a modular software architecture that addresses these requirements.

We provide three case studies of real life debugging using the new tool. The first case study shows how our tool isolates a kernel-level bug in the radio communication stack in the LiteOS [Cao et al. 2008] operating system for sensor networks that offers a UNIX-like remote file system abstraction. The second case study shows how the tool is used to debug a performance problem in a multichannel Media Access Control (MAC) protocol [Le et al. 2008]. For both case studies we used MicaZ motes as target hardware. In the third case study, we applied our tool to diagnose a protocol design bug [Khan et al. 2008a] in the directed diffusion protocol [Inatanagonwiwat et al. 2000].

It is important to stress what our tool is not. We specialize in uncovering problems that result from component interactions. Specifically, the proposed tool is not intended to look for local errors (*e.g.*, code errors that occur and manifest themselves on one node). Examples of the later type of errors include infinite loops, dereferencing invalid pointers, or running out of memory. Current debugging tools are, for the most part, well-equipped to help find such errors. An integrated development environment can use previous tools for that purpose.

The rest of the paper is organized as follows. In Section 2, we describe recent work on debugging sensor network applications. In Section 3, we describe the main idea of our tool. Section 4 elaborates several challenges that need to be addressed before we can apply discriminative frequent pattern mining for debugging along with our proposed solutions. Section 5 proposes two alternative methods, the two stage mining and the progressive discriminative sequence mining for scalability enhancements. Applicability of these two alternatives depends on the specific bug characteristics. Section 6 describes the system architecture of *Dustminer*. Section 7 describes the implementation details of the data collection front-end and data analysis back-end that are used in this paper along with their system overhead. To show the effectiveness of our tool and compare different algorithms, in Section 8, we provided three case studies. In the first, we uncover a kernel-level bug in LiteOS. In the second, we use our tool to identify a protocol design bug in a multichannel MAC protocol [Le et al. 2008]. When the bug was fixed, the throughput of the system improved by nearly 50%. In the third, we applied our tool to diagnose a protocol design bug [Khan et al. 2008a] in the directed diffusion protocol [Inatanagonwiwat et al. 2000]. Section 9 concludes the paper.

2. RELATED WORK

Most of the troubleshooting support for sensor networks (i) favors reproducible errors, and (ii) is generally geared towards finding *local* bugs such as an incorrectly written line of code, an erroneous pointer reference, or an infinite loop. Existing tools revolve around testing, measurements, or stepping through instruction execution. Marionette [Whitehouse et al. 2006] and Clairvoyant [Yang et al. 2007] are examples of source debugging systems that allow the programmer to interact with the sensor network using breakpoints, watches, and line-by-line tracing. Source level debugger is more suitable to identify programming errors which is contained in a single node. It is difficult to find distributed bugs using source level debugger due to the fact that source level debugging interferes heavily with the normal operation of the code and may prevent the excitation of distributed bugs in the first place. It also involves manual checking of system states which is not scalable. SNMS [Tolle and Culler 2005] presents a sensor network measurement service that collects performance statistics such as packet loss and radio energy consumption. Testing-based systems include laboratory testbeds such as Motelab [Werner-Allen et al. 2005], Kansei [Ertin et al. 2006], Emstar [Girod et al. 2004] etc. These systems are good at exposing manifestations of errors, but leave it to the programmer's skill to guess the cause of the problem.

Simulation and emulation based systems include TOSSIM [Levis et al. 2003], DiSenS [Wen et al. 2007], S^2DB [Wen and Wolski], Atemu [Polley et al. 2004] etc. Atemu provides XATDB which is a GUI based debugger that provides interface to debug code at line level. S^2DB is a simulation based debugger that provides debugging abstractions at different levels such as the node level and network level. It provides the concept of parallel debugging where a developer can set breakpoints across multiple devices to access internal system state. This remains a manual process, and it is very hard to debug a large system manually for design bugs. Moreover the simulated environment prevents the system from exciting bugs which arise due to peculiar characteristics of real hardware, and deployment scenarios such as clock skew, radio irregularities, and sensing failures, to name a few.

For offline parameter tuning and performance analysis, *record and replay* is a popular technique which is implemented by Envirolog [Luo et al. 2006] for sensor network applications. Envirolog stores module outputs (e.g., outputs of sensing modules) locally and replays them to reproduce the original execution. It is good at testing performance of high-level protocols subjected to a given recorded set of environmental conditions. While this can also help with debugging such protocols, no automated diagnosis support is offered.

In another work [Szewczyk et al. 2004], the authors pointed out that sensor data, such as humidity and temperature, may be used to predict network and node failures, although they do not propose an automated technique for correlating failures with sensor data. In contrast, we are focused on automating the troubleshooting of general protocol bugs which may not necessarily be revealed by analyzing sensor measurements.

PDA [Romer and Ma 2009] presents a passive assertion checking approach where the user can use several predefined commands to upload or store values of interest (e.g., variables). They described several approaches for collecting traces such as

packet sniffing network, logging, wired testbed etc. The assertions can be specified over distributed node states using a declarative language. Although this approach can identify the assertion violations but may not reveal the cause of the violation. FIND [Guo et al. 2009] describes a novel approach for faulty node detection in wireless sensor network. The algorithm used in FIND is based on the assumption that the monitored event fades in intensity with increasing distance from the source (e.g., sound, temperature). Based on this assumption, FIND tries to predict the most likely node sequence for an observed event and compares it with the reported node sequences to identify the faulty node (if any). Find attacks a different problem than ours. The goal is to identify the faulty node rather than the cause of the failure. The concept of declarative trace points [Cao et al. 2008] is proposed for efficient collection of runtime logs. But it does not automate the process of debugging. From that perspective, data collection using declarative tracepoint [Cao et al. 2008] can be thought of as a data collection front end. PAD is a passive troubleshooting framework for root cause analysis [Liu et al. 2008]. It uses a probabilistic inference model for determining dependencies among multiple network elements. It generates the network topology based on partial information collected from different nodes using a packet marking scheme. It diagnoses problems in real time and categorizes faults in several categories such as physical damage, software crashes, network congestion, environmental interference and application flaws. But it is not geared to troubleshoot arbitrary protocol bugs.

Sympathy [Ramanathan et al. 2005] presents an early step towards sensor network self-diagnosis. It specializes in attributing reduced communication throughput at a base-station to the failure of a node or link in the sensor network. Another example of automated diagnostic tools is SNTS [Khan et al. 2007] which analyzes passively logged radio communication messages using a classification algorithm [Frank and Witten 1998] to identify states correlated with the occurrence of bugs. The diagnostic capability of SNTS [Khan et al. 2007] is constrained by its inability to identify event *sequences* that precipitate an interaction-related bug. The tool also does not offer an interface to the debugged system that allows logging internal events. The diagnostic simulation effort [Khan et al. 2008a] presented automated diagnosis of the problem by analyzing simulation output. Dustminer [Khan et al. 2008b] extended the diagnostic capability by implementing an actual system (as opposed to using simulation) and presenting a better log analysis algorithm that is able to uncover *infrequent* sequences that lead to failures. The idea of symbolic sequence mining [Khan et al. 2009] tried to address the challenge of sequence mining based on absolute values. This work [Khan et al. 2009] identified that mining for patterns based on absolute attribute values may not be able to identify certain bugs where the system fails because of a hidden relationship such as hop distance, neighborhood etc. Moreover, they showed that symbolizing patterns increases the support count and hence improves the chance of subtle patterns to be ranked higher.

Discriminative pattern mining received a lot of attention from the researcher in the data mining community. Earlier work in data mining [Cheng et al. 2007] showed that the upper bound for information gain of an event (i.e., the potential discriminative power of en event) increases monotonically with the support of that event. This implies that the events with lower support has less discriminative power in

general. DDPMine algorithm [Cheng et al. 2008] showed how the upper bound estimation [Cheng et al. 2007] can be exploited during the frequent item set mining process for effective classification. One of the more recent work is NDPMine [Kim et al. 2010] that formulates the discriminative pattern mining problem as an optimization problem. NDPMine maps the given datasets to a high-dimensional space and learn the hyperplane that can correctly classify the input space. Intuitively, both DDPMine and NDPMine search for a combination of features that collectively can classify the input space. In contrary, our goal is to identify discriminative sequences of events(often more than one) where each discriminative sequence individually can distinguish between successful and failed executions.

Machine learning techniques have previously been applied to failure diagnosis in other systems [Bodk et al. 2005; Aguilera et al. 2003; Liu et al. 2005]. One prior effort tried to localize bug [Liu et al. 2006] by leveraging difference in the distribution between buggy and non-buggy execution. Software behavior graph analysis [Liu et al. 2005] were also used to identify the cause of the problem. Much of these efforts attempted to identify the cause of the problem at the code level (i.e., the responsible line or function in the code).

Formal methods [Volgyesi et al. 2005; Hanna et al. 2008; Ballarini and Miller 2006; Olveczky and Thorvaldsen 2006] offer a different alternative based on verifying component correctness or, conversely, identifying which properties are violated. The approach is challenging to apply to large systems due to the high degree of concurrency and non-determinism in complex wireless sensor networks, which leads to an exponential state space explosion. Unreliable wireless communication and sensing pose additional challenges. Moreover, even a verified system can suffer poor performance due to a design flaw. Our tool automatically answers questions that help to identify the cause of the failure that manifests during run time.

3. DUSTMINER OVERVIEW

Most previous debugging approaches for sensor networks are geared at finding localized errors in code (with preference to those that are reproducible). In contrast, we focus on non-deterministically occurring errors that arise because of unexpected or adverse distributed interactions between multiple seemingly individually-correct components. The non-localized, hard-to-reproduce nature of such errors makes them especially hard to find.

Dustminer is based on the idea that, in a distributed wireless sensor network, nodes interact with each other in a manner defined by their distributed protocols to perform cooperative tasks. Unexpected sequences of events, subtle interactions between modules, or unintended design flaws in protocols may occasionally lead to an undesirable or invalid state, causing the system to fail or exhibit poor performance. Hence, in principle, if we log different types of events in the network, we may be able to capture the unexpected sequence that leads to failure (along with thousands of other sequences of events). The challenge for the diagnostic tool is to automatically identify this culprit sequence. Our approach *exploits* both (i) non-determinism and (ii) interactive complexity to *improve* ability to diagnose distributed interaction bugs. This point is elaborated below:

—*Exploiting non-reproducible behavior:* We adapt data mining approaches that use examples of both "good" and "bad" system behavior to be able to classify the conditions correlated with good and bad. In particular, note that conditions that cause a problem to occur are correlated (by causality) with the resulting bad behavior. Root causes of non-reproducible bugs are thus inherently suited for discovery using such data mining approaches; the lack of reproducibility itself and the inherent system non-determinism improve the odds of occurrence of sufficiently diverse behavior examples to train the troubleshooting system to understand the relevant correlations and identify causes of problems.

—*Exploiting interactive complexity:* Interactive complexity describes a system where scale and complexity cause components to interact in unexpected ways. A failure that occurs due to such unexpected interactions is typically hard to "blame" on any single component. This fundamentally changes the objective of a troubleshooting tool from aiding in stepping through code (which is more suitable for finding a localized error in some line, such as an incorrect pointer reference), to aiding with diagnosing a *sequence of events* (component interactions) that leads to a failure state.

At a high level, our tool first uses a data collection front-end to collect runtime events for post-mortem analysis. Once the log of runtime events is available, the tool separates the collected sequence of events into two piles - a "good" pile, which contains the parts of the log when the system performs as expected, and a "bad" pile, which contains the parts of the log when the system fails or exhibits poor performance. This data separation phase is done based on a predicate that defines "good" versus "bad" behavior, provided by the application developer. For example the predicate, applied offline to logged data, might state that a sequence of more than 10 consecutive lost messages between a sender and receiver is bad behavior (hence return "bad" in this case). To increase diagnostic accuracy, experiments can be run multiple times before data analysis.

A discriminative frequent pattern mining algorithm then looks for patterns (sequences of events) that exist with very different frequencies in the two piles. These patterns are called *discriminative*. Later, such patterns are analyzed for correlations with preceding events in the logs, if any, that occur less frequently. Hence, it is possible to catch anomalies that cause problems as well as common roots of multiple error manifestations.

A well-known algorithm for finding frequent patterns in data is the Apriori algorithm [Agrawal and Srikant 1994]. This algorithm was used in previous work on sensor network debugging [Khan et al. 2008a] to address a problem similar to ours where Apriori algorithm is used to determine the event sequences that lead to problems in sensor networks. We show that the approach has serious limitations and extend this algorithm to suite purposes of sensor network debugging. The original Apriori algorithm, used in the aforementioned work, is an iterative algorithm that proceeds as follows. At the first iteration, it counts the number of occurrences (called *support*) of each distinct event in the data set (i.e., in the "good" or "bad" pile). Next, it discards all events that are infrequent (their support is less than some parameter *minSup*). The remaining events are frequent patterns of length 1. Assume the set of frequent patterns of length 1 is S_1. At the next iteration, the

algorithm generates all the candidate patterns of length 2 which is $S_1 \times S_1$. Here '$\times$' represents the Cartesian product. It then computes the frequency of occurrence of each pattern in $S_1 \times S_1$ and discards those with support less than *minSup* again. The remaining patterns are the frequent patterns of length 2. Let us call them set S_2. Similarly, the algorithm will generate all the candidate patterns of length 3 which is $S_2 \times S_1$ and discard infrequent patterns (with support less than *minSup*) to generate S_3 and so on. It continues this process until it cannot generate any more frequent patterns.
We show in this paper, how the previous work is extended for purposes of diagnosis. The problems with the algorithm and our proposed solutions are described in section 4 and section 5.

4. ADAPTATION OF SEQUENCE MINING FOR DEBUGGING

Performing discriminative frequent pattern mining based on frequent patterns generated by the Apriori algorithm poses several challenges that need to be addressed before we can apply discriminative frequent pattern mining for debugging. For the purposes of the discussion below, let as define an *event* to be the basic element in the log that is analyzed for failure diagnosis. The structure of an event in our log is as follows:
$< NodeId, EventType, attribute_1, attribute_2, ...attribute_n, Timestamp >$
$NodeId$ is used to identify the node that records the event. $EventType$ is used to identify the event type (*e.g.*, message dropped, flash write finished, etc). Based on the event type, it is possible to interpret the rest of the record (the list of attributes). The set of distinct $EventType$s is often called the *alphabet* in an analogy with strings. In other words, if events were letters in an alphabet, we are looking for strings that cause errors to occur. These strings represent event *sequences* (ordered lists of events). The generated log can be thought of as a single sequence of logged events. For example, $S_1 = (< a >, < b >, < b >, < c >, < d >, < b >, < b >, < a >, < c >)$ is an event sequence. Elements $< a >$, $< b >$, ..., are events. A discriminative pattern between two data sets is a subsequence of (not necessarily contiguous) events that occurs with a different count in the two sets. The larger the difference, the better the discrimination. With the above terminology in mind, we present how the algorithm is tailored to apply to debugging.

4.1 Challenge-I: Preventing False Frequent Patterns

The Apriori algorithm generates all possible combinations of frequent subsequences of the original sequence. As a result, it generates subsequences combining events that are "too far" apart to be causally correlated with high probability and thus reduces the chance of finding the "culprit sequence" that actually caused the failure. This strategy could negatively impact the ability to identify discriminative patterns in two ways; (i) it could lead to the generation of discriminative patterns that are not causally related, and (ii) it could eliminate discriminative patterns by generating false patterns. Consider the following example.
Suppose we have the following two sequences:

$S_1 = (< a >, < b >, < c >, < d >, < a >, < b >, < c >, < d >)$
$S_2 = (< a >, < b >, < c >, < d >, < a >, < c >, < b >, < d >$

Suppose the system fails when $< a >$ is followed by $< c >$ before $< b >$. As this condition is violated in sequence S_2, ideally, we would like our algorithm to be able to detect $(< a >, < c >, < b >)$ as a discriminative pattern that distinguishes these two sequences.
Now, if we apply the Apriori technique, it will generate $(< a >, < c >, < b >)$ as an equally likely pattern for both S_1, and S_2. As in both S_1 and S_2, it will combine the first occurrence of $< a >$ and the first occurrence of $< c >$ with the second occurrence of $< b >$. So it will get canceled out at the differential analysis phase.
To address this issue, the key observation here is that the first occurrence of $< a >$ should not be allowed to combine with the second occurrence of $< b >$ as there is another event $< a >$ after the first occurrence of $< a >$ but before the second occurrence of $< b >$ and the second occurrence of $< b >$ is correlated with second occurrence of $< a >$ with higher probability.
To prevent such erroneous combinations, we use a *dynamic search window* scheme where the first item of any candidate sequence is used to determine the search window. In this case, for any pattern starting with $< a >$, the search window is $[1, 4]$ and $[5, 8]$ in S_1 and S_2. With this search window, the algorithm will search for pattern $(< a >, < c >, < b >)$ in window $[1, 4]$ and $[5, 8]$ and will fail to find it in S_1 but will find it in sequence S_2 only. As a result, the algorithm will be able to report pattern $(< a >, < c >, < b >)$ as a discriminative pattern.
This *dynamic search window* scheme also speeds up the search significantly. In this scheme, the original pattern (of size 8 events) was reduced to windows of size 4 making the search for patterns in those windows more efficient.

4.2 Challenge-II: Suppressing Redundant Subsequences

At the frequent pattern generation stage, if two patterns, S_i and S_j, have *support* $\geq$ *minSup*, the Apriori algorithm keeps both sequences as frequent patterns even if one is a subsequence of the other and both have equal support. This makes perfect sense in data mining but not in debugging. For example, when mining the "good" data set, the above strategy assumes that any subset of a "good" pattern is also a good pattern. In real-life, this is not true. Forgetting a step in a multi-step procedure may well cause failure. Hence, subsequences of good sequences are not necessarily good. Keeping these subsequences as examples of "good" behavior leads to a major problem at the differential analysis stage when discriminative patterns are generated since they may incorrectly cancel out similar subsequences found frequent in the other (i.e., "bad" behavior) data pile. For example, consider two sequences below:

$S_1 = (< a >, < b >, < c >, < d >, < a >, < b >, < c >, < d >)$
$S_2 = (< a >, < b >, < c >, < d >, < a >, < b >, < d >, < c >)$

Suppose, for correct operation of the protocol, event $< a >$ has to be followed by event $< c >$ before event $< d >$ can happen. In sequence S_2 this condition is violated. Ideally, we would like our algorithm to report the following sequence S_3 as the "culprit" sequence:
$S_3 = (< a >, < b >, < d >)$
However, if we apply Apriori algorithm, it will fail to catch this sequence. This is because it will generate S_3 as a frequent pattern both for S_1 and S_2 with support 2 and will get canceled out at the differential analysis phase. As expected, S_3 will

never show up as a "discriminative pattern". Note that with the *dynamic search window* scheme alone, we cannot prevent this.
To illustrate, suppose a successful message transmission involves the following sequence of events:

$(< enableRadio >, < messageSent >, < ackReceived >, < disableRadio >)$

Now although sequence:

$(< enableRadio >, < messageSent >, < disableRadio >)$

is a subsequence of the original "good" sequence, it does not represent a successful scenario as it disables radio before receiving the "ACK" message.
To solve this problem, we need an extra step (which we call *sequenceCompression*) before we perform differential analysis to identify discriminative patterns. At this step, we remove the sequence S_i if it is a subsequence of S_j with the same *support*[1]. This will remove all the redundant subsequences from the frequent pattern list. Subsequences with a (sufficiently) different support, will be retained and will show up after discriminative pattern mining.
In the above example, pattern $(< a >, < b >, < c >, < d >)$ has *support* 2 in S_1 and *support* 1 in S_2. Pattern $(< a >, < b >, < d >)$ has *support* 2 in both S_1 and S_2. Fortunately, at the *sequenceCompression* step, pattern $(< a >, < b >, < d >)$ will be removed from the frequent pattern list generated for S_1 because it is a subsequence of a larger frequent pattern of the same support. It will therefore remain only on the frequent pattern list generated for S_2 and will show up as a discriminative pattern.

4.3 Challenge-III: Capturing the Timing Effect

Although all the events in the log are temporally ordered, sequence mining algorithm does not consider the time difference between consecutive events as a feature while mining for discriminative sequences of events. This may cause the algorithm to fail to identify certain patterns that involve the timing relations. For example, due to hardware limitations, sampling the sensors "too" frequently may cause the sensor reading to become faulty due to capacitance effect. Assume that the temperature sensing operation is represented by the symbol S. Say, for the sensor reading to be correct, the timing difference between two consecutive sampling must be at least 2 ms. A correctly functioning system would generate sequence of events that would look like as follows:

$S_{good} = (< S(1) >, < S(3) >, < S(5) >)$

Values in the parentheses record the timestamp in millisecond since the system starts (e.g., S(3) means event S happened at 3 milliseconds after the system reboot). Note that this timing information in parentheses is used to serialize the log but not used in the sequence mining algorithm as absolute timestamp is not relevant for debugging.

Now, a failed operation might generate sequence of events that may look like as follows:

$S_{bad} = (< S(1) >, < S(3) >, < S(4) >)$

To the sequence mining algorithm, both of the sequences are identical (both are $(< S >, < S >, < S >)$). But they are different if the timing information is taken

[1]This mechanism can be extended to remove subsequences of a similar but not identical support.

into account. In the failed case, the third sample was taken just after 1 ms of the second sample and is responsible for the corrupted reading. To address this issue, we incorporated the timing information by introducing fake event 't' in the log which represents a single clock tick. Using this transformation, the good log would be as follows:

$S_{good} = (< S >, < t >, < t >, < S >, < t >, < t >, < S >)$

Whereas the bad log would be as follows:

$S_{bad} = (< S >, < t >, < t >, < S >, < t >, < S >)$

Now, with this input transformation, our algorithm would be able to identify that $(< S >, < t >, < S >)$ is a discriminative pattern that occurs only in the bad log.

4.4 Challenge-IV: Identifying the Accumulative Effect

Pair wise causality relationship (e.g., one message sent is followed by one acknowledgement) is fairly common in the program execution. But there are cases when this is not true. For example, in many file systems it is often common practice to buffer data before actually writing to disk to minimize disk access and to save energy [Yang et al. 2004]. In a correctly functioning system it can be expected that multiple buffer operations will be followed by a single disk write operation. Assume that the data buffering and buffer flush operations are represented by symbols B and F respectively. A normal operation would generate sequence of events that may looks like as follows:

$S_{good} = (< B >, < B >, < F >, < B >, < F >, < B >, < B >, < B >, < F >)$

Now, a failed opeartion might generate sequence of events as follows:

$S_{bad} = (< B >, < B >, < F >, < B >, < F >, < B >, < B >, < F >, < B >)$

Note that in S_{bad}, the last buffer operation($< B >$) is not followed by a flush operation($< F >$). Unfortunately, sequence mining algorithm would identify the pattern $(< B >, < F >)$ as a common pattern in both good and bad logs. It would also identify $< B >$ as a common event with support 6 in both cases. As $< B >$ has a different support than $(< B >, < F >)$ in both good and bad logs, our subsequence elimination rule would fail to eliminate $< B >$ in either of the logs and eventually would cancel out in the discriminative analysis.

We observed that a very minor modification to our previously proposed definition of dynamic search window can solve this shortcoming. According to our earlier definition, any subsequence that starts with $< F >$ can combine with any other events that happened before the next occurrence of $< F >$. We changed the definition to include the next occurrence of $< F >$ in the search window. With the new search window, the algorithm can easily identify that $(< F >, < B >, < F >)$ is a common pattern in both good and bad logs whereas $(< F >, < B >)$ occurred only in bad log. Note that $(< F >, < B >)$ will be generated in good log as well but will be eliminated as it is a subsequence of $(< F >, < B >, < F >)$ with same support(i.e., support 2). But in the bad log, $(< F >, < B >, < F >)$ has support 2 and $(< F >, < B >)$ has support 3. Hence $(< F >, < B >)$ will be retained and will be reported as a discriminative pattern which clearly indicates that in one case $< B >$ is not followed by $< F >$. Without this extension, this pattern can not be identified.

4.5 Other Challenges

Several other changes need to be made to standard data mining techniques. For example, the amount of logged events and the corresponding frequency of patterns can be different from run to run depending on factors such as length of execution and system load. A higher sampling rate at sensors, for example, may generate more messages and cause more events to be logged. Many logged event patterns in this case will appear to be more frequent. This is problematic when it is desired to compare the frequency of patterns found in "good" and "bad" data piles for purposes of identifying those correlated with bad behavior. To address this issue, we need to normalize the frequency count of events in the log. In the case of single events (i.e., patterns of length 1), we use the ratio of occurrence of the event instead of absolute counts. In other words, the *support* of any particular event, $< e >$ in the event log is divided by the total number of events logged, yielding in essence the probability of finding that event in the log, $P(e)$. For patterns of length more than 1, we extend the scheme to compute the probability of the pattern given recursively by $P(e1).P(e2|e1).P(e3|e1.e2), ...$ The individual terms above are easy to compute. For example, $P(e2|e1)$ is obtained by dividing the support of the pattern $(< e1 >, < e2 >)$ by the total support of patterns starting with $< e1 >$.
Finally, there are issues with handling event parameters. Logged events may have parameters (e.g., identity of the receiver for a message transmission event). Since event parameter lists may be different, calling each variation a different event will cause a combinatorial explosion of the alphabet. For example, an event with 10 parameters, each of 10 possible values will generate a space of 10^{10} possible combinations. To address the problem, continuous or fine-grained parameters need to be discretized into a smaller number of ranges. Multi-parameter events need to be converted into sequences of single-parameter events each listing one parameter at a time. Hence, the exponential explosion is reduced to linear growth in the alphabet, proportional to the number of discrete categories a single parameter can take and the average number of parameters per event. Techniques for dealing with event parameter lists were introduced in our earlier work [Khan et al. 2008a] and are not discussed further in this paper.

5. SCALABILITY ENHANCEMENTS

5.1 Two Stage Mining

In debugging, sometimes less frequent patterns could be more indicative of the cause of failure than the most frequent patterns. A single mistake can cause a damaging sequence of events. For example, a single node reboot event can cause a large number of message losses. In such cases, if frequent patterns are generated that are commonly found in failure cases, the most frequent patterns may not include the real cause of the problem. For example, in case of node reboot, manifestation of the bug (message loss event) will be reported as the most frequent pattern and the real cause of the problem (the node reboot event) may be overlooked.
Fortunately, in the case of sensor network debugging, a solution may be inspired by the nature of the problem domain. The fundamental issue to observe is that much computation in sensor networks is *recurrent*. Code repeatedly visits the same states (perhaps not strictly periodically), repeating the same actions over time. Hence, a

single problem, such as a node reboot or a race condition that pollutes a data structure, often results in multiple manifestations of the same unusual symptom (like multiple subsequent message losses or multiple subsequent false alarms). Catching these recurrent symptoms by an algorithm such as Apriori is much easier due to their larger frequency. With such symptoms identified, the search space can be narrowed and it becomes easier to correlate them with other less frequent preceding event occurrences. To address this challenge, we developed a two stage pattern mining scheme.

At the first stage, the Apriori algorithm generates the usual frequent discriminative patterns that have support larger than *minSup*. For the first stage, *minSup* is set larger than 1. It is expected that the patterns involving manifestations of bugs will survive at the end of this stage but infrequent events like a node reboot will be dropped due to their low support.

At the second stage, at first, the algorithm splits the log into fixed width segments (default width is 50 events in our implementation). Next, the algorithm counts the number of discriminative frequent patterns found in each segment and ranks each segment of the log based on the count (the higher the number of discriminative patterns in a segment, the higher the rank). If discriminative patterns occurred consecutively in multiple segments, those segments are merged into a larger segment. Next, the algorithm generates frequent patterns with *minSup* reduced to 1 on the K highest-ranked segments separately (default K is 5 in our implementation) and extracts the patterns that are common in these regions. Note that the initial value of K is set conservatively. The optimum value of K depends on the application. If with the initial value of K, the tool failed to catch the real cause, the value of K is increased iteratively. In this scheme, we have a higher chance of reporting single events such as race conditions that cause multiple problematic symptoms. Observe that the algorithm is applied on data that is the total logs from several experimental runs. The race condition may have occurred once at different points of some of these runs.

This scheme has a significant impact on the performance of the frequent pattern mining algorithm. Scalability is one of the biggest challenges in applying discriminative frequent pattern analysis to debugging. For example, if the total number of logged events is of the order of thousands (more than 40000 in one of our later examples), it is computationally impossible to generate frequent patterns of non-trivial length for this whole sequence. Using two stage mining, we can dramatically reduce the search space and make it feasible to mine for longer frequent patterns which are more indicative of the cause of failure than shorter sequences.

5.2 Progressive Discriminative Sequence Mining

Scalability is one of the biggest challenges in using sequence mining for analyzing logs to identify the latent patterns that are potentially correlated to failure. Although two stage mining addresses the scalability issue to some extent, the algorithm still suffers from the following problems. *Firstly,* due to exponential number of combinations, the number of candidate patterns grows very quickly. This exponential growth rate of the size of the candidate patterns is dependent on the size of the base patterns that is used to generate the candidate set rather than the size of the log. *Secondly,* the number of patterns returned is typically in the order of

several thousands. Although the "culprit" patterns are expected to be at the top of the list, this number of final patterns is still daunting.

Our prior algorithm [Khan et al. 2008b] generates all the frequent patterns of length up to "n" that are common across all the bad logs and common across all the good logs before performing the discriminative analysis to identify the "culprit" sequences of events. In this approach, the algorithm generates a lot of patterns that are eventually going to be dropped at the last stage.

Inspired by the work presented in DDPMine [Cheng et al. 2008], we developed the progressive discriminative analysis which tries to prune patterns as early as possible without risking the possibility of dropping the "culprit" sequence. Instead of performing discriminative analysis at the last stage, we perform discriminative analysis at each stage k after generating frequent patterns of length k. Our approach has several differences with the work presented in DDPMine [Cheng et al. 2008]. Firstly, DDPMine is for frequent unordered item set mining. In contrast, our algorithm is for ordered sequence mining. Secondly, our pruning strategy for early elimination of candidate patterns is different than DDPMine. DDPMine exploited the idea that the information gain upper bound is lower for less frequent items/events to prune items. Using this approach is not suitable for debugging. We took a different approach as explained below.

Suppose, we have I number of good files and J number of bad files. Now, assume that GP_k is the set of good patterns of length k and each pattern g_i in GP_k has support θ_{gi} where $\theta_{gi}/|I| > \delta$. If δ =0.8, the above condition implies that each pattern in set GP_k occurred in at least 80% of the good files. Similarly, assume that BP_k is the set of bad patterns of length k and each pattern b_i in BP_k has support θ_{bi} where $\theta_{bi}/|J| > \delta$. If δ =0.8, the above condition implies that each pattern in set BP_k occurred in at least 80% of the bad files.

Now, before generating patterns of length (k+1) we do the following. We calculate three sets of patterns. $FinalGP_k$=$FindDiscriminative(GP_k,BP_k)$, $FinalBP_k$= $FindDiscriminative(BP_k,GP_k)$, $CommonP_k$=$FindCommon(GP_k, BP_k)$. Here the function FindDiscriminative and function FindCommon are defined in Table I.

At the next step, we use the patterns in set $CommonP_k$ to generate patterns of length (k+1). We stop using patterns in set $FinalGP_k$ and $FinalBP_k$ to generate longer patterns. Because any pattern in $FinalGP_k$ is already discriminative and by making them longer is not going to make them any more discriminative. Rather it can decrease their potential for being discriminative.

This scheme has the following three advantages. *Firstly,* As we prune at each stage, it reduces the size of the candidate patterns that needs to be checked substantially and speed up the overall process by huge factor. *Secondly,* it reduces the size of the final patterns returned as the discriminative patterns. In one example it returned just five patterns instead of thousands. *Thirdly,* in this scheme the user can now decide to stop the analysis as soon as the set of discriminative good and bad patterns become nonempty. Earlier we have to specify the parameter "n" which is the maximum length of the pattern that the user wishes to generate. The optimum value of "n" can be hard to guess apriori. If it is "too short" or "too long", the algorithm may fail to identify the "culprit" pattern. The algorithm is presented in Table I. We evaluated progressive discriminative sequence mining in section 8.

Algorithm: Progressive Discriminative Sequence Mining

Input: Set of Good Logs (GL), Set of Bad Logs(BL)
Output: Set of discriminative sequences of events
1. S_{common}=ϵ, SG=ϵ,SB=ϵ,K=1
2. while(SG == ϵ or SB == ϵ)
   2.1 S_{good}=GenerateFrequentSubSequences(GL,K,S_{common})
   2.2 S_{bad}=GenerateFrequentSubSequences(BL,K,S_{common})
   2.3 If (SG == ϵ) SG = SG U FindDiscriminative(S_{good}, S_{bad})
   2.4 If (SB == ϵ) SB=SB U FindDiscriminative(S_{bad}, S_{good})
   2.5 S_{common}=FindCommon(SG,SB)
   2.6 k=k+1

Function: FindDiscriminative
Input: Set of Frequent SubSequences(A),Set of Frequent SubSequences(B)
Return: Set of discriminative SubSequences that distinguishes A from B
Assumption: Each sequence p_i in A or B has two supports, $sup_{i_{inFile}}$ and $sup_{i_{acrossFile}}$.
$sup_{i_{inFile}}$ records the average number of occurrence of p_i within a file.
$sup_{i_{acrossFile}}$ records the probability of occurrence of p_i in a file.
1. $S_{discriminative}$=ϵ
2. for each sequence p_i in A
   2.1 for each sequence q_j in B
      2.1.1 if(p_i == q_j) then
      if($sup_{i_{acrossFile}}/sup_{j_{acrossFile}}<\theta$) then $S_{discriminative}$=$S_{discriminative}$ U p_i
      else if($sup_{i_{inFile}}/sup_{j_{inFile}}<\delta$) then $S_{discriminative}$=$S_{discriminative}$ U p_i
3. Return $S_{discriminative}$

Function: GenerateFrequentSubSequences
Input: Set of Logs(L),sequenceLength(K), baseSet(S_{common})
Return: Set of frequent SubSequences of length K
1. Use the Apriori algorithm to generate frequent subsequences of length K using baseSet
2. Return frequent subsequences of length K generated at step 1

Table I. Progressive Discriminative Sequence Mining

6. DUSTMINER ARCHITECTURE

We realize that the types of debugging algorithms needed are different for different applications, and are going to evolve over time with the evolution of hardware and software platforms. Hence, we aim to develop a modular tool architecture that facilitates evolution and reuse. Keeping that in mind, we developed a software architecture that provides the necessary functionality and flexibility for future development. The goal of our architecture is to facilitate easy use and experimentation with different debugging techniques and foster future development. As there are numerous different types of hardware, programming abstractions, and operating systems in use for wireless sensor networks, the architecture must be able to accommodate different combinations of hardware and software. Different ways of data collection should not affect the way the data analysis layer works. Similarly we realize that for different types of bugs, we may need different types of techniques to identify the bug and we want to provide a flexible framework to experiment with different data analysis algorithms. Based on the above requirements, we designed a layered, modular architecture as shown in Figure 1. We separate the whole system into three subsystems; (i) a data collection front-end, (ii) data preprocessing middleware and (iii) a data analysis back-end.

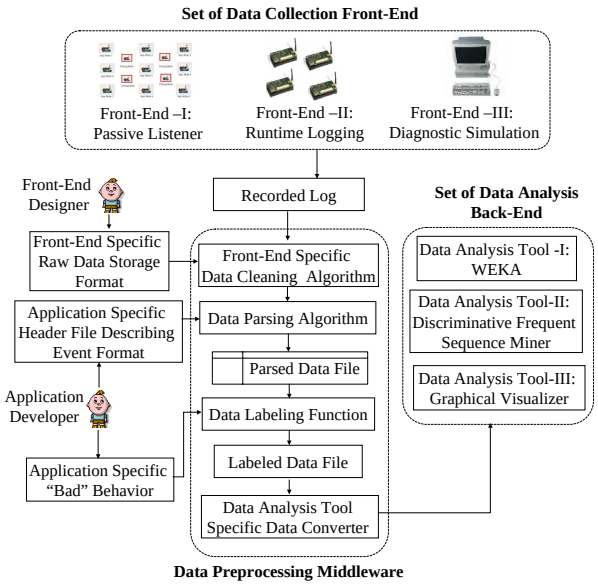

Fig. 1. Debugging framework

6.1 Data Collection Front-End

The role of data collection front-end is to provide the debug information (*i.e.*, log files) that can be analyzed for diagnosing failures. The source of this debug log is irrelevant to the data analysis subsystem. As shown in Figure 1, the developer may choose to analyze the recorded radio communication messages obtained using a passive listening tool, or the execution traces obtained from simulation runs, or the run-time sequences of events obtained by logging on actual application motes and so on. With this separation of concerns, the front-end developer could design and implement the data collection subsystem more efficiently and independently. The data collection front-end developer merely needs to provide the format of the recorded data. These data are used by the data preprocessing middleware to parse the raw recorded byte streams.

6.2 Data Preprocessing Middleware

This middleware that sits between the data collection front-end and the data analysis back-end provides the necessary functionality to change or modify one subsystem without affecting the other. The interface between the data collection front-end and the data analysis back-end is further divided into the following layers:

—*Data cleaning layer:* This layer is front-end specific. Each supported front-end will have one instance of it. The layer is the interface between the particular data collection front-end and the data preprocessing middleware. It ensures that the recorded events are compliant with format requirements.

—*Data parsing layer:* This layer is provided by our framework and is responsible for extracting meaningful records from the recorded raw byte stream. To parse the recorded byte stream, this layer requires a header file describing the recorded message format. This information is provided by the application developer (*i.e.*, the user of the data collection front-end).

—*Data labeling layer:* To be able to identify the probable causes of failure, the data analysis subsystem needs samples of logged events representing both "good" and "bad" behavior. As "good" or "bad" behavior semantics are an application specific criterion, the application developer needs to implement a predicate (a small module) whose interface is already provided by us in the framework. The predicate, presented with an ordered event log, decides whether behavior is good or bad.

—*Data conversion layer:* This layer provides the interface between the data preprocessing middleware and the data analysis subsystem. One instance of this layer exists for each different analysis back-end. This layer is responsible for converting the labeled data into appropriate format for the data analysis algorithm. The interface of this data conversion layer is provided by the framework. As different data analysis algorithms and techniques can be used for analysis, each may have different input format requirements. This layer provides the necessary functionality to accommodate supported data analysis techniques.

6.3 Data Analysis Back-End

At present, we implement the data analysis algorithm and its modifications presented earlier in Section 4 and Section 5. It is responsible for identifying the causes of failures. The approach is extensible. As newer analysis algorithms are developed that catch more or different types of bugs, they can be easily incorporated into the tool as alternative back-ends. Such algorithms can be applied in parallel to analyze the same set of logs to find different problems with them.

7. DUSTMINER IMPLEMENTATION

In this section, we describe the implementation of the data collection front-end and the data analysis back-end that are used for failure diagnosis in this paper. We used three different data collection front-ends for three different case studies. The front-end used for the first case study was a built-in logging support functionality provided by the LiteOS operating system for MicaZ motes. For the second case study, the front-end is implemented by us and used for real time logging of user defined events on flash memory in MicaZ motes. For the last case study, we used the logging support provided by TOSSIM [Levis et al. 2003] for logging different runtime events in simulation. At the data analysis back-end, we used discriminative frequent pattern analysis for failure diagnosis. We describe the implementation of each of these next.

7.1 The Front-End: Acquiring System State

We used two different data collection front-ends to collect data: (i) event logging system implemented for MicaZ platform in TinyOS 2.0 and (ii) kernel event logger for MicaZ platform provided by LiteOS. The format of the event logged by the

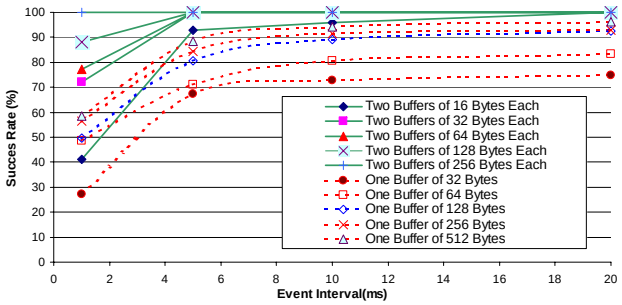

Fig. 2. Impact of buffer size and event rate on logging performance

two subsystems are completely different. We were able to use our framework to easily integrate the two different front-ends and use the same back-end to analyze the cause of failures, which shows modularity. We briefly describe each of these front-ends below.

7.1.1 *Data Collection Front-End for TinyOS.*
The event logger for MicaZ hardware is implemented using the TinyOS 2.0 BlockRead and BlockWrite interfaces to perform read and write operations respectively on flash. BlockRead and BlockWrite interfaces allow accessing the flash memory at a larger granularity which minimizes the recording time to flash.
To minimize the number of flash accesses we used a global buffer to accumulate events temporarily before writing to flash. Two identical buffers (buffer A and B) are used alternately to minimize the interference between event buffering and writing to flash. When buffer A gets filled up, buffer B is used for temporary buffering and buffer A is written to flash and vice versa. In Figure 2 we show the effect of buffer size on logging performance for single buffer and double buffer respectively. Using two buffers increases the logging performance substantially. As shown in figure, for event rate of 1000 events/second, using one buffer of 512 bytes has a success ratio (measured as the ratio of successfully logged events to the total number of generated events) of only 60% whereas using two buffers of 256 bytes each (512 bytes in total) can give almost 100% success ratio. For a rate of 200 events/second, two buffers of 32 bytes each is enough for 100% success ratio.
The sizes of these buffers are configurable as different applications need different amounts of runtime memory. It is to be noted that if the system crashes while some data are still in the RAM buffer, those events will be lost. The flash space layout is given in Figure 3.
A separate MicaZ mote (*LogManager*) is used to communicate with the logging subsystem to start and stop logging. Until the logging subsystem receives the "StartLogging" command, it will not log anything and after receiving "StopLogging" command it will flush the remaining data that is in the buffer to flash and stop logging. This gives the user the flexibility to start and stop logging whenever they want. It also lets the user to run their application without enabling logging, when needed, to avoid the runtime overhead of logging functionality without recompiling the code.

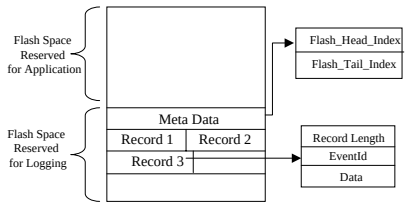

Fig. 3. Flash space layout

We realize that occasional event reordering can occur due to preemption, interrupts, or task scheduling delays. An occasional invalid log entry is not a problem. An occasional incorrect logging sequence is fine too as long as the same occasional wrong sequence does not occur consistently. This is because common sequences do not have to occur every time, but only often enough to be noticed. Hence, they can be occasionally mis-logged without affecting the diagnostic accuracy.

Time Synchronization:
We need to timestamp the recorded events so that events recorded on different nodes can be serialized later during offline analysis. To avoid the overhead of running a time synchronization protocol on the application mote, we used an offline time synchronization scheme. A separate node (*TimeManager*) is used to broadcast its local clock periodically. The event logging component will receive the message and log it in flash with a local timestamp. From this information we can calculate the clock skew on different nodes in reference to *TimeManager* node, adjust the timestamp of the logged events and serialize the logs. We realize that the serialized log may not be exact but it is good enough for pattern mining.

System Overhead:
The event logging support requires 14670 bytes of program memory (this includes the code size for BlockRead and BlockWrite interface provided by TinyOS 2.0) and 830 bytes of data memory when 400 bytes are used for buffering (two buffers of 200 bytes each) data before writing to flash. User can choose to use less buffer space if the expected event rate is low. To instrument code, the program size increase is minimal. To log an event with no attributes, it needs a single line of code. To log an event with n attributes, it takes $n + 1$ lines of code, n lines are to initialize the record and 1 line to call the $log()$ function.

API for Logging in TinyOS:
The only part of the data collection front-end that is exposed to the user is the interface for logging user defined events. Our design goal was to have an easy-to-use interface and efficient implementation to reduce the runtime overhead as much as possible. One critical issue with distributed logging was to timestamp the recorded events so that events on different nodes can be serialized later during offline analysis. To make event logging functionality simpler, we defined the interface to the logging

component as follows:

```
log(EventId,(void *)buffer,unit8_t size)
```

$log(EventId, (void*)buffer, unit8\_tsize)$ is the key interface between application developers and the logging subsystem. To log an event, the user has to call the log() function with appropriate parameters. For example, if the user wants to log the event that a radio message was sent and also wants to log the receiverId along with the event, he/she needs to define the appropriate record structure in a header file (this file will also be used to parse the data) with these fields, initialize the record with appropriate values and call the *log* function with that record as the parameter. This simple function call will log the event. The rest is taken care of by the logging system underneath. The logging system will pad the timestamp with the recorded event and log as a single event. Note that $NodeId$ is not recorded during logging. This information is added when data is uploaded to PC for offline analysis.

7.1.2 *Data Collection Front-End for LiteOS.*

LiteOS [Cao et al. 2008] provides the required functionality to log kernel events on MicaZ platforms. Specifically, the kernel logs events including system calls, radio activities, context switches and so on. An event log entry is a single 8-bit code without attributes. We used an experimental set up of a debugging testbed with all motes connected to a PC via serial interfaces. In pre-deployment testing on our indoor testbed, logs can thus be transmitted in real-time through a programming board via serial communication with a base-station. When a system call is invoked or a radio packet is received on a node, the corresponding code for that specific event is transmitted through the serial port to the base station (PC). The base station collects event codes from the serial port and records it in a globally ordered file.

Logging in LiteOS:

The logging support provided by LiteOS lets user to log different system calls. Several C macros are defined in the kernel source code. There is a specific macro for each system call that enables the logging for that specific call. The user has to call the specific Macro for that system call to enable logging that particular system call. The resulting log contains only the unique id of the system call that is sent to the serial port if that system call is invoked during the execution. This is done by calling a function *"addTrace(systemCallId)"* in each invocation of the system call if the corresponding macro value is set. The user requires to recompile the kernel to enable logging.

7.2 The Data Analysis Back-End

At the back-end, we implement the data preprocessing and discriminative frequent pattern mining algorithm. To integrate the data collection front-end with the data preprocessing middleware, we provide a simple text file interface that describes the storage format of the raw byte streams collected for each of the front-ends. This file is used to parse the recorded events. Once the data is parsed, the user can either manually label the data files as "good" or "bad", or the user can supply different predicates as a Java function that can be used to annotate data automatically. The

rest of the system is a collection of data analysis algorithms such as discriminative frequent pattern mining, or any other tool such as Weka [Weka]. Our algorithm automatically generates discriminative patterns and report it to the user.

8. EVALUATION

To test the effectiveness of the tool, we applied our tool to troubleshoot three real life applications. The first was a kernel level bug in the LiteOS operating system. The second was to debug a multichannel Media Access Control(MAC) protocol [Le et al. 2008] implemented in TinyOS 2.0 for MicaZ platform with only one half-duplex radio interface. In the third, we applied our tool to diagnose a protocol design bug [Khan et al. 2008a] in the directed diffusion protocol [Inatanagonwiwat et al. 2000].
We analyzed the logs using the following algorithms depending on the applicability: (i) the Apriori sequence mining algorithm [Agrawal and Srikant 1994] used in our earlier work [Khan et al. 2008a] that does not incorporate the extensions described in section 4(we call this the "Basic Apriori Algorithm"), (ii) sequence mining algorithm that incorporates the extensions described in section 4(we call this the "Extended Apriori Algorithm"),(iii) sequence mining algorithm that incorporates the extensions described in section 4 and applies the two stage mining technique described in section 5.1(we call this the "Extended Apriori Algorithm with Two Stage Mining"), (iv) sequence mining algorithm that incorporates the extensions described in section 4 and progressive discriminative analysis described in section 5.2(we call this the "Extended Apriori Algorithm with Progressive Discriminative Analysis").

8.1 Case Study - I: LiteOS Bug

In this case study, we troubleshoot a simple data collection application where several sensors monitor light and report it to a sink node. The communication is performed in a single-hop environment. In this scenario, sensors transmit packets to the receiver, and the receiver records received packets and sends an "ACK" back. The sending rate that sensors use is variable and depends on the variations in their readings. After receiving each message, depending on its sequence number, the receiver decides to record the value or not. If the sequence number is older than the last sequence number it has received, the packet is dropped.
This application is implemented using MicaZ motes on the LiteOS operating system and is tested on an experimental testbed. Each of the nodes is connected to a desktop computer via an MIB520 programming board and a serial cable. The PC acts as the base station. In this experiment, there was one receiver (the base node) and a set of 5 senders (monitoring sensors). This experiment illustrates a typical experimental debugging set up. Prior to deployment, programmers would typically test the protocol on target hardware in the lab. This is how such a test might proceed.

8.1.1 *Failure Scenario.*

When this simple application was stress tested, some of the nodes would crash occasionally and non-deterministically. Each time different nodes would crash and at different times. Perplexed by the situation, the developer (a first-year graduate

RecordedEvents	*AttributeList*
Context_Switch_To_User_Thread	*Null*
Get_Current_Thread_Index	*Null*
Get_Current_Radio_Info_Address	*Null*
Get_Current_Radio_Handle_Address	*Null*
Post_Thread_Task	*Null*
Get_Serial_Mutex	*Null*
Get_Current_Serial_Info_Address	*Null*
Get_Serial_Send_Function	*Null*
Disable_Radio_State	*Null*
Packet_Received	*Null*
Packet_Sent	*Null*
Yield_To_System_Thread	*Null*
Get_Current_Thread_Address	*Null*
Get_Radio_Mutex	*Null*
Get_Radio_Send_Function	*Null*
Mutex_Unlock_Function	*Null*
Get_Current_Radio_Handle	*Null*

Table II. Logged events for diagnosing LiteOS application bug

student with no prior experience with sensor networks) decided to log different types of events using LiteOS support and use our debugging tool. These were mostly kernel-level events along with a few application-level events. The built-in logging functionality provided by LiteOS was used to log the events. A subset of the different types of logged events are listed in Table II.

Before presenting the results obtained by our algorithms, we will briefly describe the way a received packet is handled in the LiteOS and the real cause of the problem. In the application, receiver always registers for receiving packets, then waits until a packet arrives. At that time, the kernel switches back to the user thread with appropriate packet information. The packet is then processed in the application. However, at very high data rates, another packet can come when the processing of the previous packet has not yet been done. In that case, LiteOS kernel overwrites the radio receive buffer with new information even if the user is still using the old packet data to process the previous packet. Indeed, for correct operation, $< Packet\_Received >$ event always has to be preceded by $< Get\_Current\_Radio\_Handle >$ event. Otherwise it crashes the system. Overwriting a receive buffer for some reason is a very typical bug in sensor networks.

After running the experiment, "good" logs were collected from the nodes that did not crash during the experiment and "bad" logs were collected from nodes that crashed at some point in time. We subsequently analyzed the logs as follows.

8.1.2 *Basic Apriori Algorithm.*

We implemented the Apriori algorithm used in [Khan et al. 2008a] without incorporating the extensions described in section 4 and applied it to generate frequent patterns and perform differential analysis to extract discriminative patterns. For this case study, when we applied the Apriori algorithm to the "good" log and the "bad" log, the list of discriminative patterns missed the $< Packet\_Received >$ event completely and failed to identify the fact that the problem was correlated with the timing of packet reception. Moreover, when we applied the Apriori al-

$< Packet\_Received >, < Packet\_Sent >, < Get\_Current\_Radio\_Handle >$
$< Packet\_Received >, < Get\_Current\_Radio\_Handle\_Address >, < Get\_Current\_Radio\_Handle >$
$Packet\_Received >, < Mutex\_Unlock\_Function >, < Get\_Current\_Radio\_Handle >$
$< Packet\_Received >, < Disabale\_Radio\_State >, < Get\_Current\_Radio\_Handle >$
$< Packet\_Received >, < Post\_Thread\_Task >, < Get\_Current\_Radio\_Handle >$

Table III. Discriminative frequent patterns found only in "good" log for LiteOS bug

$< Context\_Switch\_to\_User\_Thread >, < Get\_Current\_Thread\_Address >, < Get\_Serial\_Send\_Function >$
$< Packet\_Received >, < Context\_Switch\_to\_User\_Thread >, < Get\_Serial\_Send\_Function >$
$< Packet\_Received >, < Post\_Thread\_Task >, < Get\_Serial\_Send\_Function >$
$< Packet\_Received >, < Get\_Current\_Thread\_Index >, < Get\_Serial\_Send\_Function >$
$< Packet\_Received >, < Get\_Current\_Thread\_Address >, < Get\_Serial\_Send\_Function >$

Table IV. Discriminative frequent patterns found only in "bad" log for LiteOS bug

gorithm to multiple instances of "good" logs and "bad" logs together, the list of discriminative patterns returned was empty. All the frequent patterns generated by Apriori algorithm were canceled at the differential phase. This result highlights the weakness of the Apriori algorithm when applied for debugging and emphasize the necessity of our extensions as described in section 4.

8.1.3 *Extedned Apriori Algorithm.*
After applying our discriminative frequent pattern mining algorithm that incorporates the extensions described in section 4 to the logs, we provided two sets of patterns to the developer, one set includes the highest ranked discriminative patterns that are found only in "good" logs as shown in Table III, and the other set includes the highest ranked discriminative patterns that are found only in "bad" logs as shown in Table IV.
Based on the discriminative frequent pattern, it is clear that in "good" pile, $< Packet\_Received >$ event is highly correlated with the $< Get\_Current\_Radio\_Handle >$ event. On the other hand, in the "bad" pile, though $< Packet\_Received >$ event is present, the other event is missing. In the "bad" pile, $< Packet\_Received >$ is highly correlated with $< Get\_serial\_Send\_Function >$ event. From these observations, it is clear that proceeding with a $< Get\_serial\_Send\_Function >$ when $< Get\_Current\_Radio\_Handle >$ is missing is the most likely cause of failure.

8.1.4 *Extended Apriori Algorithm with Two Stage Mining.*
As two stage mining is more suitable for bugs that have frequent manifestations such as high number of message losses, we did not apply the two stage mining for this case study as the manifestation of the problem (system crash) was infrequent in this case study. Two stage mining is applied for the case study presented in section 8.2 where the manifestation (message loss) of the bug is frequent.

8.1.5 *Extended Apriori Algorithm with Progressive Discriminative Analysis.*
To evaluate the performance improvement due to applying the progressive discriminative sequence mining scheme, we applied the progressive discriminative sequence mining algorithm on the same set of logs. It comes up with twenty six sequences of events as "culprit" sequences of events along with following one:

$< Get\_Current\_Radio\_Handle >, < Packet\_Received >, < Packet\_Received >$

Algorithm Used	*Runtime*	*Comments*
Basic Apriori	N/A	Need the extensions presented in Section 4
Extended Apriori	248 sec	
Extended Apriori with Two Stage Mining	N/A	Effect of the problem is infrequent. Requires manual parameter tuning.
Extended Apriori with Progressive Discriminative Analysis	127 sec	

Table V. Comparison of different schemes for LiteOs bug

This sequence explains the bug in one step. From this sequence it is obvious that if two consecutive messages are received following a single $< Get\_Current\_Radio\_Handle >$ event, the system crashes. Indeed, $< Get\_Current\_Radio\_Handle >$ event represents the required handle registration process and it must precede a message receive event.

One thing to note is that our earlier algorithm dropped it due to a particular setting of a threshold parameter that was used to measure the discriminative power of a particular sequence. As earlier we only used the support of a sequence within a file, due to a normalization factor this crucial sequence was dropped mistakenly. Missing this pattern in [Khan et al. 2008b] highlights the difficulty of parameter tuning that can affect the accuracy of the algorithm. Although with parameter tuning it is possible to capture this sequence, in many cases it is hard to guess the right values apriori.

One of the main contributions of progressive discriminative mining is the enhancement in the scalability. To mine for the discriminative patterns, it took 127 seconds with progressive mining where as the earlier algorithm took 248 seconds which is an improvement of almost 95%.

We compare the effectiveness and performance of different schemes in Table V.

8.2 Case Study - II: Multichannel MAC Protocol

In this case study we debug a multichannel MAC protocol [Le et al. 2008]. The objective of the protocol used in our study is to assign a home channel to each node in the network dynamically in such a way that the throughput is maximized. The design of the protocol exploits the fact that in most wireless sensor networks, the communication rate among different nodes is not uniform (*e.g.*, in a data aggregation network). Hence, the problem was formulated in such a way that nodes communicating frequently are clustered together and assigned the same home channel whereas nodes that communicate less frequently are clustered into different channels. This minimizes overhead of channel switching when nodes need to communicate.

During experimentation with the protocol, it was noticed that when data rates between different internally closely-communicating clusters is low, the multi-channel protocol outperforms a single channel MAC protocol comfortably as it should. However, when the data rate between clusters was increased, while the throughput near the base station still outperformed a single channel MAC significantly, nodes further from the base station were performing worse than in the single channel MAC.

This should not have happened in a well-designed protocol as the multichannel MAC protocol should utilize the communication spectrum better than a single channel MAC. The author of the protocol initially concluded that the performance degradation was due to the overhead associated with communication across clusters assigned to different channels. Such communication entails frequent channel switching as the sender node, according to the protocol, must switch the frequency of the receiver before transmission, then return to its home channel. This incurs overhead that increases with the transmission rate across clusters. We decided to verify this conjecture.

As a stress test of our tool, we instrumented the protocol to log events related to the MAC layer (such as message transmission and reception as well as channel switching) and used our tool to determine the discriminative patterns generated from different runs with different message rates, some of which performing better than others. For better understanding of the failure scenario detected, we briefly describe the operation of the multichannel MAC protocol below.

8.2.1 *Multichannel MAC Protocol Overview.*

In the multichannel MAC protocol, each node initially starts at channel 0 as its home channel. To communicate with others, every node maintains a data structure called "neighbor table" that stores the neighbor home channel for each of its neighboring nodes. Channels are organized as a ladder, numbered from lowest (0) to highest (12). When a node decides to change its home channel, it sends out a "Bye" message in its current home channel which includes its new home channel number. Receiving a "Bye" message, each other node updates its neighbor table to reflect the new home channel number for the sender of the "Bye" message. After changing its home channel, a node sends out a "Hello" message in the new home channel which includes its nodeID. All neighboring nodes on that channel add this node as a new neighbor and update their neighbor tables accordingly.

To increase robustness to message loss, the protocol also includes a mechanism for discovering the home channel of a neighbor when its current entry in the neighbor table becomes stale. When a node sends a message to a receiver on that receiver's home channel (as listed in the neighbor table) but does not receive an "ACK' after 'n' (n is set to 5) tries, it assumes that the destination node is not on its home channel. The reason may be that the destination node has changed its home channel permanently but the notification was lost. Instead of wasting more time on retransmissions on the same channel, the sender starts scanning all channels, asking if the receiver is there. The purpose is to find the receiver's new home channel and update the neighbor table accordingly. The destination node will eventually hear this data message and reply when it is on its home channel.

Since the above mechanism is expensive, as an optimization, overhearing is used to reduce staleness of the neighbor table. Namely, a node updates the home channel of a neighbor in its neighbor table when the node overhears an acknowledgement ("ACK") from that neighbor sent on that channel. Since the "ACK"s are used as a mechanism to infer home channel information, whenever a node switches channels temporarily (e.g., to send to a different node on the home channel of the latter), it delays sending out "ACK" messages until it comes back to its home channel in order to prevent incorrect updates of neighbor tables by recipients of such ACKs.

$RecordedEvents$	$AttributeList$
$Ack\_Received$	$Null$
$Home\_Channel\_Changed$	$oldChannel, newChannel$
$TimeSyncMsg$	$referenceTime, localTime$
$Channel\_Update\_Msg\_Sent$	$homeChannel$
$Data\_Msg\_Sent\_On\_Same\_Channel$	$destId, homeChannel$
$Data\_Msg\_Sent\_On\_Different\_Channel$	$destId, homeChannel, destChannel$
$Channel\_Update\_Msg\_Received$	$homeChannel, neighborId, neighborChannel$
$Retry\_Transmission$	$oldChannelTried, nextChannelToTry$
$No\_Ack\_Received$	$Null$

Table VI. Logged events for diagnosing multichannel MAC protocol

Finally, to estimate channel conditions, each node periodically broadcasts a "channelUpdate" message which contains the information about successfully received and sent messages during the last measurement period (where the period is set at compile time). Based on that information, each node calculates the channel quality (*i.e.*, probability of successfully accessing the medium), and uses that measure to probabilistically decide whether to change its home channel or not. Nodes that sink a lot of traffic (e.g., aggregation hubs or cluster heads) switch first. Others that communicate heavily with them follow. This typically results into a natural separation of node clusters into different frequencies so they do not interfere.

8.2.2 *Performance Problem.*
This protocol was executed on 16 MicaZ motes implementing an aggregation tree where several aggregation cluster-heads filter data received from their children, significantly reducing the amount forwarded, then send that reduced data to a basestation. When the data rate across clusters was low, the protocol outperformed the single channel MAC. However, when the data rate among clusters was increased, the performance of the protocol deteriorated significantly, performing worse than a single channel MAC in some cases. The developer of the protocol assumed that this was due to the overhead associated with the channel change mechanism which is incurred when communication happens among different clusters heavily. Much debugging effort was spent on that direction with no result. To diagnose the cause of the performance problem, we logged different types of MAC events as listed in Table VI.
The question posed to our tool was "Why is the performance bad at higher data rate?". To answer this question, we first executed the protocol at low data rates (when the performance is better than single channel MAC) to collect logs representing "good" behavior. We then again executed the protocol with a high data rate (when the performance is worse than single channel MAC) to collect logs representing "bad" behavior. We subsequently analyzed the logs as follows.

8.2.3 *Basic Apriori Algorithm.*
Using the basic Apriori algorithm , to generate frequent patterns of length 2 for 40000 events in the "good" log, it took 1683.02 seconds (28 minutes) and to finish the whole computation including differential analysis it took 4323 seconds (72 minutes). We tried to generate frequent patterns of length 3 with the approach in [Khan et al. 2008a] but terminated the process after one day of computation

$< No\_Ack\_Received >, < Retry\_Transmission >$
$< Retry\_Transmission >, < No\_Ack\_Received >$
$< Data\_Msg\_Sent\_On\_Same\_Channel : homechannel : 0 >,$ $< No\_Ack\_Received >,$ $< Retry\_Transmission >,$ $< Retry\_Transmission : nextchanneltotry : 1 >,$ $< Retry\_Transmission >,$ $< Retry\_Transmission : oldchanneltried : 1 >,$ $< No\_Ack\_Received >$
$< Data\_Msg\_Sent\_On\_Same\_Channel : homechannel : 0 >,$ $< No\_Ack\_Received >,$ $< Retry\_Transmission >,$ $< Retry\_Transmission : nextchanneltotry : 1 >,$ $< Retry\_Transmission : nextchanneltotry : 2 >$
$< Data\_Msg\_Sent\_On\_Same\_Channel : homechannel : 0 >,$ $< No\_Ack\_Received >,$ $< Retry\_Transmission >,$ $< Retry\_Transmission : nextchanneltotry : 1 >,$ $< Retry\_Transmission : oldchanneltried : 2 >,$ $< Retry\_Transmission : nextchanneltotry : 3 >,$ $< No\_Ack\_Received >,$ $< Retry\_Transmission : oldchanneltried : 3 >$

Table VII. Discriminative frequent patterns for multichannel MAC protocol

that remained in progress. We used a machine of 2.53 GHz speed and 512 MB RAM. This highlights the scalability problem.

8.2.4 *Extended Apriori Algorithm with Two Stage Mining.*
With our two-stage mining scheme, it took 5.547 seconds to finish the first stage and finishing the whole computation including differential analysis took 332.924 seconds (6 minutes). After performing discriminative pattern analysis, the list of top 5 discriminative patterns that were produced by our tool is shown in Table VII. The sequences indicate that, in all cases, there seems to be a problem with not receiving acknowledgements. Lack of acknowledgements causes a channel scanning pattern to unfold. This is shown as the $< Retry\_Transmission >$ event on different channels, as a result of not receiving acknowledgements. Hence, the problem does not lie in the frequent overhead of senders changing their channel to that of their receiver in order to send a message across clusters. The problem lied in the frequent lack of response (an ACK) from a receiver. At the first stage of frequent pattern mining $< No\_Ack\_Received >$ is identified as the most frequent event. At the second stage, the algorithm searched for frequent patterns in top K (e.g., top 5) segments of the logs where $< No\_Ack\_Received >$ event occurred with highest frequency. The second stage of the log analysis (correlating frequent events to preceding ones) then uncovered that the lack of an ACK from the receiver is preceded by a temporary channel change. This gave away the bug. As we described earlier, whenever a node changes its channel temporarily, it disables "ACK"s until it comes back to its home channel. In a high intercluster communication scenario, disabling the "ACK" is a bad decision for a node that spends a significant amount of time communicating with other clusters on channels other than its own home channel. As a side effect, nodes which are trying to communicate with it fail to

Algorithm Used	*Runtime*	*Comments*
Basic Apriori	N/A	Too Slow
Extended Apriori	N/A	Too Slow
Extended Apriori with Two Stage Mining	333 sec	Effect of the problem (msg loss)is frequent
Extended Apriori with Progressive Discriminative Analysis	14 sec	

Table VIII. Comparison of different schemes for multichannel MAC Protocol bug

receive an "ACK" for a long time and start scanning channels frequently looking for the missing receiver. Another interesting aspect of the problem that was discovered is the cascading effect of the problem. When we look at generated discriminative patterns across multiple nodes, we found that the scanning patterns revealed in the logs shown in fact cascades. Channel scanning at the destination node often triggers channel scanning at the sender node and this interesting cascaded effect was also captured by our tool.

8.2.5 *Extended Apriori Algorithm with Progressive Discriminative Analysis.* Progressive discriminative mining returned in 14 seconds and returned the 59 single events as highly correlated to poor performance. The top events were the followings:
$< No\_Ack\_Received >, < Retry\_Transmission >, < Data\_Msg\_Sent\_On\_Different\_Channel >,$
$< Retry\_Transmission : oldchanneltried : 0 >, < Retry\_Transmission : nextchanneltotry : 1 >,$
$< Retry\_Transmission : oldchanneltried : 1 >, < Retry\_Transmission : nextchanneltotry : 2 >,$
$< Retry\_Transmission : oldchanneltried : 2 >, < Retry\_Transmission : nextchanneltotry : 3 >,$
$< Retry\_Transmission : oldchanneltried : 3 >, < Retry\_Transmission : nextchanneltotry : 4 >,$
$< Retry\_Transmission : oldchanneltried : 4 >, < Retry\_Transmission : nextchanneltotry : 5 >,$
$< Retry\_Transmission : oldchanneltried : 5 >, Retry\_Transmission : nextchanneltotry : 6 >,$
$< Retry\_Transmission : oldchanneltried : 6 >, < Retry\_Transmission : nextchanneltotry : 7 >,$
$< Retry\_Transmission : oldchanneltried : 7 >, < Retry\_Transmission : nextchanneltotry : 8 >,$
$< Retry\_Transmission : oldchanneltried : 8 >, < Retry\_Transmission : nextchanneltotry : 9 >,$
$< Retry\_Transmission : oldchanneltried : 9 >, < Retry\_Transmission : nextchanneltotry : 10 >,$
$< Retry\_Transmission : oldchanneltried : 10 >, < Retry\_Transmission : nextchanneltotry : 11 >$
Althogh these were all single events, in this case study it would have been adequate to provide the necessary insights to the real problem. The designer of the protocol can readily understand the channel scanning phenomenon that was happening at high intercluster data rate.

We compare the effectiveness and performance of different schemes in Table VIII.

8.3 Case Study -III : Directed Diffusion Protocol Bug

We have reported this bug in our earlier work [Khan et al. 2007; Khan et al. 2009]. Directed diffusion [Inatanagonwiwat et al. 2000] is a widely popular data centric communication protocol in wireless sensor network. For completeness, we briefly describe the design of the protocol below. In directed diffusion, any node that is interested in a particular type of data (e.g., detected vehicle in a particular region in a surveillance network) would first need to broadcast its "interest" in the network. This interest message includes the type of the data, geographic coordinates

of interest and the duration of the interest. Any node receiving the interest message would store that in its cache memory. Later when any node receives a data, it checks its interest cache to verify whether it is on the path and whether it is supposed to forward that data message to the designated path or not. If the interest cache has no matching entry, it would drop the data silently assuming that it is not in the data forwarding path. The problem is if a node gets rebooted for some reason, it erases the interest cache completely and would result in a broken path if there is a single path from the source node to the sink node and the rebooted node is on that critical path. Due to this design flaw, there would be a large number of consecutive message losses following a reboot. To evaluate the scalability, we collected three good logs (when no node was rebooted) and three bad logs (when a node was rebooted).

8.3.1 *Basic Apriori Algorithm.*
We applied the basic Apriori algorithm used in our earlier work [Khan et al. 2008a] on six logs (three good logs and three bad logs). We configured the algorithm to generate frequent patterns of length up-to 3. The algorithm failed to finish after six hours of computation.

8.3.2 *Extended Apriori Algorithm with Two Stage Mining.*
Next, we applied the extended Apriori algorithm used in our prior work [Khan et al. 2008b] with two stage mining and configured the algorithm to generate frequent patterns of length up-to 3. The algorithm finished in about 2.5 hours. Unfortunately, it returned several thousands of patterns. Moreover, although the algorithm was able to identify the "Reboot" event as correlated to failure, it was ranked at the very end of the list due to low support and increased the chance of being overlooked by the developer as unimportant pattern.

8.3.3 *Extended Apriori Algorithm with Progressive Discriminative Analysis.*
In comparison, progressive discriminative sequence mining finished in about 5 seconds and it returned only seven individual events as correlated to failure. Four of these seven events are listed below:
$< BOOT\_EVENT : (NodeId : X) >$,
$< interestCacheEmpty : (NodeId : X) >$,
$< dataCacheEmpty : (NodeId : X) >$, and
$< msgDropped : (ReasonToDrop : dataWithNoMatchingInterest >$.
The reason for such drastic improvement is that the progressive mining strategy reduced the search space significantly by applying the discriminative analysis at each stage. Another important characteristic is that it reduced the number of final patterns returned from several thousands to only a few which enhances the usability of the tool significantly.

We compare the effectiveness and performance of different schemes in Table IX.

8.4 Debugging Overhead

To test the impact of logging on application behavior, we ran the multichannel MAC protocol with logging enabled and without logging enabled with both moderate data rate and high data rate. The network was set as a data aggregation network.
For moderate data rate experiment, the source nodes (node that only sends mes-

Algorithm Used	*Runtime*	*Comments*
Basic Apriori	N/A	Too Slow
Extended Apriori	N/A	Too Slow
Extended Apriori with Two Stage Mining	2.5 hr	Failed to rank the Reboot event at the top
Extended Apriori with Progressive Discriminative Analysis	5 sec	

Table IX. Comparison of different schemes for Directed diffusion protocol bug

sages) were set to transmit data at a rate of 10 messages/sec, the intermediate nodes were set to transmit data at a rate of 2 messages/sec and one node was acting as the base station (which only receives messages). We tested this on a 8 nodes network with 5 source nodes, 2 intermediate nodes and one base station. Over multiple runs, after we take the average to get a reliable estimate, average number of successfully transmitted messages was increased by 9.57% and average number of successfully received messages was increased by 2.32%. The most likely reason for this minor improvement is writing to flash was creating a randomization effect which probably helped to reduce interference at the MAC layer.

At high data rate, source nodes were set to transmit data at a rate of 100 messages/sec and intermediate nodes were set to transmit data at a rate of 20 messages/sec. Over multiple runs, after we take the average to get a reliable estimate, average number of successfully transmitted messages was reduced by 1.09% and average number of successfully received messages was dropped by 1.62%. The most likely reason is the overhead of writing to flash kicked in at a such high data rate and eventually reduced the advantage experienced at a low data rate.

The performance improvement of the multichannel MAC protocol reported in this paper is obtained by running the protocol at the high data rate to prevent over estimation.

We realize that this effect on application may change the behavior of the original application slightly, but that effect seems to be negligible from our experience and did not affect the diagnostic capability of the discriminative pattern mining algorithm which is inherently robust against minor statistical variance.

As multichannel MAC protocol did not use flash memory to store any data, we were able to use the whole flash for logging events. To test the relation between quality of generated discriminative patterns and the logging space used, we used 100KB, 200KB and 400KB of flash space in three different experiments. The generated discriminative patterns were similar. We realize that different application has different amount of flash space requirements and the amount of logging space may affect the diagnostic capability. To help in severe space constraints, we provide the radio interface so users can choose to log at different times instead of logging continuously. User can also choose to log events at different resolutions (*e.g.*, instead of logging every message transmitted, log only every 50^{th} message transmitted).

For LiteOS case study, we did not use flash space at all as the events were transmitted to basestation (PC) directly using serial connection and eliminate the flash space overhead completely which makes our tool easily usable for testbeds which often provides serial connections.

8.5 Discussion

From the above evaluation we can draw the following conclusions. *Firstly,* the changes as described in section 4 that were made to the apriori sequence mining algorithm to adapt it for debugging is critical for effective diagnosis. *Secondly,* progressive discriminative analysis is extremely fast. Although, in some cases it may return single events as correlated to failure, these events can be used as the clues to begin with and can be further explored if the user of the tool desires. *Thirdly,* progressive mining has an automatic way of identifying when to stop the mining process. For example, in the LiteOS bug case study, it stopped after generating patterns of length 3 when the set of discriminative patterns became non empty and in case of the MAC protocol bug it stopped right after mining patterns of length 1. Earlier we had to guess and set the pattern length conservatively which often wastes a lot of time for mining longer patterns and returns too many patterns. *Fourthly,* the two stage mining is not suitable in cases where the manifestation of the problem is not frequent(e.g., the bug found in the LiteOS operating system has infrequent manifestation and cause, a single reordering of events followed by system crash).

9. CONCLUSION

In this paper, we presented a sensor network troubleshooting tool that helps the developer diagnose root causes of errors. The tool is geared towards finding interaction bugs. Very successful examples of debugging tools that hunt for localized errors in code have been produced in previous literature. The point of departure in this paper lies in focusing on errors that are not localized (such as a bad pointer or an incorrect assignment statement) but rather arise because of adverse interactions among multiple components each of which appears to be correctly designed. The cascading channel-scanning example that occurred due to disabling acknowledgements in the MAC Protocol illustrates the subtlety of interaction problems in sensor networks. With increased distribution and resource constraints, the interactive complexity of sensor networks applications will remain high, motivating tools such as the one we described. Future development of *Dustminer* will focus on scalability and user interface to reduce the time and effort needed to understand and use the new tool.

Acknowledgment

This work was supported in part by NSF grants DNS 05-54759,CNS 06-26342, CNS 06-13665. Any opinions and findings are those of the authors and not necessarily those of the funding agencies.

REFERENCES

http://www.cs.waikato.ac.nz/ml/weka/.

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proceedings of the Twentieth International Conference on Very Large Data Bases (VLDB'94)*. 487–499.

Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. 2003. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth*

ACM symposium on Operating systems principles (SOSP'03). 74–89. Bolton Landing, NY, USA.

Ballarini, P. and Miller, A. 2006. Model checking medium access control for sensor networks. In *Proceedings of the 2nd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'06)*. Paphos, Cyprus, 255–262.

Bodk, P., Friedman, G., Biewald, L., Levine, H., Candea, G., Patel, K., Tolle, G., Hui, J., Fox, A., Jordan, M. I., and Patterson, D. 2005. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the 2nd International Conference on Autonomic Computing(ICAC'05)*.

Cao, Q., Abdelzaher, T., Stankovic, J., and He, T. 2008. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *Proceedings of the Seventh International Conference on Information Processing in Sensor Networks (IPSN'08)*.

Cao, Q., Abdelzaher, T., Stankovic, J., Whitehouse, K., and Luo, L. 2008. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*. Raleigh, NC, USA.

Cheng, H., Yan, X., Han, J., and Hsu, C. 2007. Discriminative frequent pattern analysis for effective classification. In *In Proceedings of ICDE*. 716–725.

Cheng, H., Yan, X., Han, J., and Yu, P. S. 2008. Direct discriminative pattern mining for effective classification. In *Proc. of Int. Conf. on Data Engineering (ICDE'08)*.

Cheong, E., Liebman, J., Liu, J., and Zhao, F. 2003. Tinygals: a programming model for event-driven embedded systems. In *Proceedings of the 2003 ACM symposium on Applied computing (SAC'03)*. 698–704. Melbourne, Florida.

Ertin, E., Arora, A., Ramnath, R., and Nesterenko, M. 2006. Kansei: A testbed for sensing at scale. In *Proceedings of the 4th Symposium on Information Processing in Sensor Networks (IPSN/SPOTS track)*.

Frank, E. and Witten, I. H. 1998. Generating accurate rule sets without global optimization. In *Proceedings of the Fifteenth International Conference on Machine Learning (ICML'98)*. 144–151.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI'03)*. 1–11.

Girod, L., Elson, J., Cerpa, A., Stathopoulos, T., Ramanathan, N., and Estrin, D. 2004. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC'04)*. Boston, MA, 24–24.

Guo, S., Zhong, Z., and He, T. 2009. Find: faulty node detection for wireless sensor networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. Berkeley, California, USA, 253–266.

Hanna, Y., Rajan, H., and Zhang, W. 2008. Slede: Lightweight specification and formal verification of sensor networks protocols. In *Proceedings of the First ACM Conference on Wireless Network Security (WiSec)*. Alexandria, VA.

Inatanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Mobicom*. Boston, MA, USA.

Khan, M. M. H., Abdelzaher, T., and Gupta, K. K. 2008a. Towards diagnostic simulation in sensor networks. In *Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS)*. Greece.

Khan, M. M. H., Abdelzaher, T., Han, J., and Ahmadi, H. 2009. Finding symbolic bug patterns in sensor networks. In *Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS)*. California, USA.

Khan, M. M. H., Le, H. K., Ahmadi, H., Abdelzaher, T. F., and Han, J. 2008b. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, New York, NY, USA, 99–112.

Khan, M. M. H., Luo, L., Huang, C., and Abdelzaher, T. 2007. Snts: Sensor network troubleshooting suite. In *Proceedings of International Conference on Distributed Computing in Sensor Systems (DCOSS)*. Santa Fe, New Mexico, USA.

Kim, H. S., Kim, S., Weninger, T., Han, J., and Abdelzaher, T. 2010. Ndpmine: Efficiently mining discriminative numerical features for pattern-based classification. In *Proceedings of 2010 European Conf. on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD'10)*. Barcelona, Spain.

Le, H. K., Henriksson, D., and Abdelzaher, T. 2008. A practical multi-channel medium access control protocol for wireless sensor networks. In *Proceedings of International Conference on Information Processing in Sensor Networks (IPSN'08)*. St. Louis, Missouri.

Levis, P. and Culler, D. 2002. Mate: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. San Jose, California.

Levis, P., Lee, N., Welsh, M., and Culler, D. 2003. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys'03)*. Los Angeles, California, USA, 126–137.

Liu, C., Fei, L., Yan, X., Han, J., and Midkiff, S. P. 2006. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering 32*, 831–848.

Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. 2005. Sober: statistical model-based bug localization. In *Proceedings of the 13th ACM SIGSOFT international symposium on Foundations of software engineering (FSE-13)*. Lisbon, Portugal.

Liu, K., Li, M., Liu, Y., Li, M., Guo, Z., and Hong, F. 2008. Pad: Passive diagnosis for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*. Raleigh, NC, USA.

Luo, L., Abdelzaher, T. F., He, T., and Stankovic, J. A. 2006. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems 5,* 3, 543–576.

Luo, L., He, T., Zhou, G., Gu, L., Abdelzaher, T., and Stankovic, J. 2006. Achieving Repeatability of Asynchronous Events in Wireless Sensor Networks with EnviroLog. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM'06)*. 1–14.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2005. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems 30,* 1, 122–173.

Olveczky, P. and Thorvaldsen, S. 2006. Formal modeling and analysis of wireless sensor network algorithms in real-time maude. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. Rhodes Island, Greece.

Polley, J., Blazakis, D., McGee, J., Rusk, D., and Baras, J. S. 2004. Atemu: A fine-grained sensor network simulator. In *Proceedings of the First International Conference on Sensor and Ad Hoc Communications and Networks (SECON'04)*. Santa Clara, CA, 145–152.

Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., and Estrin, D. 2005. Sympathy for the sensor network debugger. In *Proceedings of the 3rd international conference on Embedded networked sensor systems (SenSys'05)*. 255–267.

Romer, K. and Ma, J. 2009. Pda: Passive distributed assertions for sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*. 337–348.

Szewczyk, R., Polastre, J., Mainwaring, A., and Culler, D. 2004. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*.

Tolle, G. and Culler, D. 2005. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN'05)*. Istanbul, Turkey, 121–132.

Volgyesi, P., Maroti, M., Dora, S., Osses, E., and Ledeczi, A. 2005. Software composition and verification for sensor networks. *Science of Computer Programming 56,* 1-2, 191–210.

Wen, Y. and Wolski, R. s^2db: A novel simulation-based debugger for sensor network applications. UCSB 2006. 2006-01.

Wen, Y., Wolski, R., and Moore, G. 2007. Disens: scalable distributed sensor network simulation. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'07)*. 24–34. San Jose, California, USA.

Werner-Allen, G., Swieskowski, P., and Welsh, M. 2005. 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)*. 483–488.

Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., and Culler, D. 2006. Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks: Special Track on Sensor Platform, Tools, and Design Methods for Network Embedded Systems (IPSN/SPOTS)*. Nashville, TN, 416–423.

Yang, J., Soffa, M. L., Selavo, L., and Whitehouse, K. 2007. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems (SenSys'07)*. 189–203.

Yang, J., Twohey, P., Engler, D., and Musuvathi, M. 2004. Using model checking to find serious file system errors. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 19–19.