

Dependence-based Multi-level Tracing and Replay for Wireless Sensor Networks Debugging

Abstract

Due to resource constraints and unreliable communication, wireless sensor network (WSN) programming and debugging remain to be challenging tasks. Deterministic replay is an error diagnosis method which has long been proposed for distributed systems. However, one of the significant hurdles for applying deterministic replay on WSN is posed by the small program memory on typical sensor nodes. This paper proposes a dependence-based multi-level method for memory-efficient tracing and replay. In the interest of portability across different hardware platforms, the method is implemented as a source-level tracing and replaying tool. To further reduce the code size after tracing instrumentation, a cost model is used for making the decision on which functions to in-line. A prototype for the tool targets C programs is developed on top of the Open64 compiler and is tested using several TinyOS applications running on TelosB motes. Preliminary experimental results show that the test programs, which do not fit the program memory after straightforward instrumentation, can be successfully instrumented using the new method such that the injected errors can be found.

Categories and Subject Descriptors D.2.5 [Testing and Debugging]: Debugging aids

General Terms Algorithms, Reliability.

Keywords Wireless sensor network; program debugging; invariants; dependence analysis; resource constraints.

1. Introduction

Wireless sensor networks (WSN) are gaining increased attention for possible use in applications such as structural health monitoring, environmental surveillance, scientific observation, and others [27][28]. A wireless sensor network typically consists of a large number of unattended wireless sensor nodes. Despite the increasing efforts [6][20] made to ease the development and simulation of WSN applications, sensor network programming and debugging is still a difficult task in view of resource constraints and unreliable communications of wireless sensor nodes.

Deterministic replay (or record-replay) is an error diagnosis method which has long been proposed for distributed systems. Under this method, nondeterministic events are recorded throughout the system operation. When an error is reported, the program can be re-run, with the recorded events restaged to allow the programmer to inspect the executed statements and the state change they cause such that the source of the error, namely the incorrectly written statements or unexpected events causing the error, can be located. The replay method significantly reduces the amount of information to record at run time. Unfortunately, the relatively small program memory poses another significant hurdle to the adoption of record-replay on sensor nodes. The popular TelosB mote has only 48KB flash memory to store the program. This limits the extent to which one can add program statements to perform tracing. Lacking sufficient memory, most existing

schemes for run-time WSN logging record only coarse information which is far from sufficient for deterministic replay. As a result, it remains difficult to pin-point the source of the errors which are detected at run time.

The aim of this paper is to develop a dependence-based multi-level tracing and replay scheme to overcome the constraint posed by the limited memory on WSN motes.

Based on our scheme, we develop a source level tracing and replaying tool which is independent of the hardware platforms and the native compiler. The source-level tracing, compared to assembly-level tracing, offers high portability of the tool. It also enables the user to take advantage of many existing source-level debuggers, such as GNU's gdb, when replaying on a desktop machine.

We have implemented our tool based on the Open64 C compiler [13]. Given the error conditions to be detected at runtime, we use the compiler to compute the program slice [19], using the error conditions as the slicing criteria. We then partition the program slice into multiple levels such that initially lower-level tracing is performed, targeting only the functions close to where error conditions are checked. The tracing level increases as farther away functions must be traced in order to find the error source. Tracing operations are inserted in the given C program before the native compiler converts it to the machine code. In addition, our tool generates another C program for later replaying on a desktop machine.

We use TinyOS [12], which is written in nesC [20], as our current testing environment for the developed tool. TinyOS is one of the most popular operating systems for sensor network applications. It has been used by more than 100 research groups worldwide. Typically, a TinyOS application consists of a scheduler and a group of wired components. The nesC compiler first converts the application into a C program which is then compiled by the native compiler into machine code executable on the specific hardware. We use the same native compiler to compile the C program instrumented by our Open64-based tool before loading it on the sensor mote for normal execution with tracing. When an error is detected, we retrieve the trace and feed it to our replaying C program which is executed on a desktop machine where many well developed debugging tools, e.g. the GNU gdb, can be used to isolate the source of the error. To replay the interaction between different motes, we simply start multiple processes, one for each mote, feeding each with its own retrieved logging information. How to retrieve the logged information from a node is out of scope of this paper. We can use one of various existing methods [37] such as using a different radio on the same node, storing logged information on a nearby node, etc.

The rest of the paper is organized as follows. Section 2 defines the problem addressed by this paper and gives an overview of our solution. Section 3 discusses how to reduce instrumentation based on dependence information and proves its effectiveness under a number of assumptions. Section 4 discusses multi-level tracing in case such assumptions are not satisfied. Implementation and experimental results are presented in Section 5. After summarizing related works on WSN debugging and deterministic replay we conclude in Section 6.

2. An Overview

For a given program, we specify a set of correctness properties using predicates defined over a list of program variables under a certain system of logic, e.g. temporal logic [21][22]. (The exact system of logic used is not of particular concern for this paper.) The program is required to satisfy this set of predicates within a specified program scope, e.g. the entire program (as long as all variables in the predicate are global), individual functions, individual program segments, or any point between two specific program statements.

In general, a distributed system reacts to events whose timing is difficult to predict or specify at the time of program development. Also, the developers may not have verified the correctness of the program thoroughly under deployment conditions, which can be significantly different from the test conditions in the lab. Errors, therefore, often exist in WSN programs after deployment. However, by inserting assertions to the program, violations of certain predicates, i.e. errors, can be detected at run time. In order to fully check whether a predicate is satisfied, it must be reevaluated every time one of its variables gets updated.

Assuming that the predicates themselves are composed correctly, when a predicate gets violated, we know at least one of its variables has obtained an incorrect value through some point in the program where the variable was updated. That value may be the result of earlier operations using incorrect operands, and so on. Eventually the error must be traced back to its source through a chain of data dependences and control dependences. *For the purpose of this paper, we consider two possibilities: (a) one or more program statements are written incorrectly, and (b) certain unexpected events occur, e.g. messages are received with incorrect contents. The type (a) errors include the erroneous removal of program statements which were supposed to provide the correct values, causing a later operation to use values written by wrong statements.*

If the entire sequence of executed instructions and operands were recorded, then one could follow the dynamic use-def chain backward and inspect the program statements along the way until the origin of the error is found. The cost of such extensive recording is prohibitive in both time and space. Under the record-and-replay scheme, however, we only need to record all nondeterministic events on each mote, which includes all external messages, task scheduling decisions, and internal interrupts. On current WSNs, all these can be captured by inserting logging operations in interrupt handlers written in high-level languages such as C. This observation of ours is a key to enabling tracing for debugging in the resource-constrained WSNs.

Since the program on each node in a distributed system such as a WSN may run indefinitely, the length of the trace is unbounded. With limited storage for the trace, in general one retrieves only a tail of the full trace. Replay is therefore often partial in practice. In order to enable deterministic replay corresponding to the retrieved trace tail, we require the program being considered to satisfy the following assumptions:

- **Assumption 1:** The program contains no recursive calls.
- **Assumption 2:** The infinite running of the program is controlled by one or more infinite loops which can be recognized at compile time.

Under these two fundamental assumptions, we insert in each infinite loop an *anchor checkpoint* at which we record the values of all variables needed to enable replaying the program starting from this program point. In order for the replay tool to capture the source of the error, the following assumption must also be satisfied:

- **Assumption 3:** The trace storage is sufficiently large such that, when an error is detected, the stored trace will contain at least one anchor point prior to the source of the error.

If the above assumption is unsatisfied, then either the trace cannot be replayed (because of the lack of any anchor point) or the replay will not lead to the source of the error (because the error source falls off the trace). In such an unfortunate case, we will resort to multi-level tracing which instruments a subset of the functions but yet permit the trace to be replayed. One of the main goals of this paper is to reduce the storage overhead for tracing, thus increasing the chance of capturing the source of the error in the stored trace. This is in addition to the objectives to minimize the instrumented code size and the increased processing time.

The information logged must include the execution context for each invocation of an interrupt handler so that the replay program can restage the invocation of the interrupt handler accurately. The type of information to be recorded at run time will be discussed in the next section. Since interrupt handlers will be treated differently from the other functions, we identify interrupt handlers by annotations before applying our instrumentation tool. Furthermore, those interrupt routines which take external inputs such as radio communication messages are explicitly marked. The external inputs can then be recorded at run time, allowing the interrupt routine to be replayed.

Although our experiments are performed on TinyOS-based WSN applications, the proposed methodology and the developed tools can be applied to other distributed embedded platforms as long as *the program is single threaded and the assumptions made above are satisfied*. Our current toolset also relies on an intermediate C code generated from the original program.

In the next section, we first discuss how to use dependence information to exclude functions irrelevant to the invariants from the run-time logging so as to reduce the size of the instrumented program. This is followed by a presentation of the instrumentation algorithm with relevant guarantees.

3. Using Dependence Information to Reduce Runtime Logging

The benefit of reducing runtime logging is two-fold. Firstly, a longer execution history can be replayed with the same amount of data storage for the trace. The time to execute the annotated program that is being traced is reduced. Finally, the number of instrumented operations to perform tracing is reduced, which leads to a smaller code size.

If a function never has any effect on the kind of errors we monitor, i.e. on any of the variables appearing in the predicates (also called the *invariants*) which specify the correctness properties, then such a function does not need to be traced at runtime. To exclude such functions from tracing, we first compute the backward slice [19] using the given set of invariants as the slicing criteria. The result of this computation is a set of control/data dependence chains which include all operations (such as assignments, branching decisions and function calls) having an effect on the set of invariants. Each function which contains any of these operations will be instrumented to obtain the runtime

execution log. Obviously, the main function of the program is always instrumented.

This set of functions, however, does not yet include those interrupt handlers which may have an effect on the invariants. In microcontroller execution, interrupts are the basic source of non-determinism. For example, a TinyOS application is interrupt driven. It runs in two contexts, the task context and the async context. The transition from the task context to the async context can happen only as the result of an interrupt causing control to transfer to an interrupt handler, interrupting any currently running task. Conversely, the transition from async context to task context occurs when the interrupt handlers completes, at which time TinyOS takes one of the following actions: (i) process the next pending interrupt if any, (ii) continues the execution of the task that was interrupted, (iii) start the next task in the queue, and (iv) go idle.

Since it is infeasible to predict when a particular interrupt may happen, we instrument all those interrupt handlers whose execution may modify global variables on which the invariants depend.

3.1 What to log

After we determine the set of functions to instrument, we insert operations into the source code of these functions to record the following pieces of information. We shall prove in this section that this set of information is sufficient for accurate replay. (The necessity of the information is self evident.)

LOG type 1 (Function entry/return) -- A function always has a single entry but may have multiple returns. We use N_RET_i , where i is an integer, to indicate which return statement is executed. If this is a function entry, it marks whether it is an interrupt handler and, if so, the name of the function.

LOG type 2 (Global variable update count) – In order to prepare for replaying interrupt routines, when an interrupt routine is invoked at run time, a global-variable reference counter, denoted by $\#gv_reference$, is written to the log, after which the count is reset to zero. Immediately after the exit of the interrupt routine, $\#gv_reference$ is reset to zero again. For any other functions, $\#gv_reference$ is reset to zero both at the entry and at the exit. Every reference (read or write) to a global variable is followed by an increment of $\#gv_reference$. This count will be used during the replay to help determine where in the program to replay specific interrupt routines.

LOG type 3 (Task scheduling) – If task scheduling order is random, then we need to record the task that is scheduled to next. However, TinyOS uses a FIFO task queue. Hence, as long as the invocations of the interrupt routines are replayed accurately, this type of information does not need to be recorded.

LOG type 4 (Anchor points) – As discussed previously, at each anchor point, we record all variable values which are needed in order for the program to replay from here.

LOG type 5 (non-deterministic inputs) – It is necessary to record non-deterministic input for future replay. In TinyOS, the messages received from radio communication and the sensor data arriving from the bus belong to this type. Note that the interrupt handlers export such input by writing it to a global variable. Since the interrupt handlers which take external input are explicitly marked, we add operations in such handlers to save their global variables to the trace.

3.2 How to replay

Our replay program is written automatically by the compiler at the source code level. At the same time as the code is instrumented for tracing, the compiler stages record-handling in the replay

program. For each operation inserted to the instrumented mote program to write LOG type i to the trace, we insert a corresponding operation, $readLOG(type\ i)$, in the replay program. The replay program is essentially the original C program with these calls inserted.

The main program, however, looks quite different from the original main program. It starts by calling $readLOG(type\ 4)$, which looks through the recorded trace for the first anchor point and gives a pointer to the loop to be executed next. At the beginning of such an anchored loop, all variables needed to continue the execution are retrieved from the recorded trace. After this, the replay program simply executes the original C program statements until it meets the inserted $readLOG$ library calls. For each log type, the read library call executes according to the following description.

- $readLOG(type\ 1)$ – This is encountered either at the beginning of a program or right before a return. This routine looks ahead in the trace and checks to see whether the next log is for an interrupt handler's entry. If so, it remembers the $\#gv_reference$ at which the interrupt handler is invoked. The $readLOG$ routine returns, and the replay program, after resetting $\#gv_reference$ to zero, continues to execute until reaching the triggering $\#gv_reference$, at which time it calls the interrupt handler. If the next log is not for an interrupt handler, then the $readLOG(type\ 1)$ routine simply returns, letting the replay program continue execution until seeing the next $readLOG$.
- $readLOG(type\ 2)$ – see above.
- $readLOG(type\ 3)$ – If the tasks are scheduled randomly, the replay program reads LOG type 3 in order to determine which task to execute.
- $readLOG(type\ 4)$ – A flag indicates whether this is the first anchor point encountered. If so, according to the pre-determined format, this $readLOG$ routine reads in all variable values before starting to execute the first statement at the anchor point. If this is not the first anchor encountered by the replay program, all recorded variables at this point are skipped.
- $readLOG(type\ 5)$ – The replay must be at the entry of an interrupt handler which takes external input. This $readLOG$ routine reads in the saved global variables to allow the interrupt handler to be replayed.

We have two alternatives for handling hardware-dependent code in interrupt routines, the write operations to hardware registers by the interrupt handlers, to be specific. Our first option is to remove all hardware dependent code for replay. The impact of interrupts will be on the values of certain global variables. (Similar handling is performed in certain TinyOS simulators [6][24].) This however misses the opportunity to trace the error source further when a message containing wrong contents is received and saved to a hardware register by a low-level interrupt handler. Only when the second interrupt handler, posted by the first one, copies the wrong contents from a hardware register to a global variable will the error be located by backward tracking from a violated invariant. A remedy for this omission is to write a preprocessor customized for the hardware platform which converts references to hardware registers to global variables.

Statements which do not affect the invariants are deleted from the replayed program. The sliced code execution techniques [10] are utilized in this part. After these treatments, the resulting code for replay can be compiled and executed on an ordinary desktop machine.

Note that the bookkeeping on $\#gv_reference$ to enable source-level tracing and replay does not cost much more than the

operations to save the loop counts in the assembly code in order for the replay program to be able to continue correct execution after an interrupt handler exits. Recording the return address in the trace alone is insufficient. As a matter of fact, if the function contains irreducible cycles in its control flow graph, it is not obvious how to count loop iterations so the replay can continue correctly after returning from an interrupt handler.

Before we prove the correctness of the scheme presented in this section, we remind the reader that, for each violated invariant, the error eventually must be traced back to a wrong value propagated through a use-def chain to the invariant. If not for the nondeterministic events at run time such as interrupts, it would be a trivial matter to show that the use-def chains observed during replay is identical to that exhibited by the mote program. Our proof thus focuses on the impact of the nondeterministic events.

Theorem 3.1: Suppose an incorrect program statement causes an invariant to be violated at run time. Under the record-replay scheme described above, the same incorrect program statement will cause the same invariant to be violated in the replayed program.

Proof: The LOG type 3 ensures that the order in which tasks are scheduled from the task queue is exactly the same when executed by the replay program as by the mote program. We just need to prove that interrupts do not cause the programmer to observe incorrect use-def chains during replay.

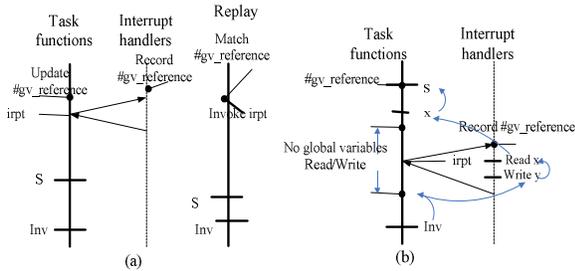


Fig 1. An illustration for Proof of Theorem 3.1

First, suppose the incorrect statement execution S and the invariant violation Inv are both outside any interrupt routine. As illustrated by Figure 1(a), the $\#gv_reference$ value at the time S must be the same in the mote program and the replay program. If no interrupts occur between these two at run time, then the replay program will find the last interrupt routine prior to Inv before it replays S .

Conversely, as illustrated by Figure 1(b), if an interrupt, $irpt$, occurs between S and Inv , then the programmer must pay attention to $irpt$ only if it is part of the use-def chains between S and Inv . This, however, is possible only if $irpt$ first reads a global variable, x , computed outside $irpt$ such that x depends on S and then writes to a global variable y on which Inv depends. (Both dependences are by transitivity, and x may be the same variable as y .) Consider two possibilities: (i) $\#gv_reference$ recorded by $irpt$ is greater than the value at the time of S . S will be replayed before $irpt$ in this case. (ii) $\#gv_reference$ is reset to zero due to other functions called between S and $irpt$. The replay program will replay S before these called functions and therefore before $irpt$. Furthermore, the $\#gv_reference$ match ensures that the replay program invokes $irpt$ between the correct pairs of consecutive

references to any global variables. In both cases, the correct use-def chains will be observed.

Next, consider three other possibilities: a) S is outside any interrupt routine but Inv is inside an interrupt routine $irpt$. There must be a global variable, x , which, by transitivity, depends on S , is read inside $irpt$ and eventually leads to the violation of Inv . b) S is within an interrupt routine $irpt$, and Inv is outside any interrupt routine. There must be a global variable, x , written between S and the end of $irpt$ such that x depends on S , by transitivity, and x is read after the exit from $irpt$ which eventually leads to the violation of Inv .

For both a) and b), by reasoning about $\#gv_reference$ and LOG type 1, we can prove that the order between S , write to x , Inv will be preserved in replay regardless whether the write to x is inside any interrupt routine or not. The order will also be preserved no matter whether the write to x happens to yet another interrupt routine. Details are omitted.

Finally, suppose S is in an interrupt routine $irpt1$ and Inv is in another interrupt routine $irpt2$. There must be a global variable, x , written in $irpt1$ and another, y , read in $irpt2$ such that the value of y depends on x by transitivity and x is depends on S by transitivity. (It is possible for x and y to be the same variable.) Again, by reasoning about $\#gv_reference$ and LOG type 1, it can be proven that the order between S , write to x , read of y , and Inv will be preserved during replay regardless whether other interrupt routines are invoked. Details are also omitted. \square

4. Multi-level Tracing

Theorem 3.1 uses Assumption 3 made in Section 2. If that assumption is not satisfied, then when an error is detected, we either cannot find an anchor point to replay the program or cannot find the error source during replay. This can happen if the storage for logging is small or the error happens a long time before it is detected (through the violation of a predicate). To enable replay under such a circumstance, we present multi-level tracing in this section. Rather than instrumenting the whole program, we divide the program functions into different levels based on how “far away” they are from the invariants being checked. Naturally, another benefit of multi-level tracing is the relaxed requirement on program memory size. Nonetheless, with multi-level tracing, we no longer have the guarantee that the error source will be found, but at least we have partial traces to narrow the search.

Multi-level tracing follows an iterative procedure described below.

4.1 An Iterative Tracing and Replay Procedure

For the purpose of defining the levels of tracing, we build a graph based on the dependence information computed previously. For convenience of implementation, we wrap each invariant-checking operation in an *invariant-checking function* and insert a call to this function everywhere the invariant must be checked.

Definition 4.1 Given a set of invariants, the *invariant-based Program Function Dependence Graph (PFDG)* for a program is a set of nodes, each representing a function whose execution directly or indirectly affects whether the invariants hold, and a set of edges of two kinds, namely the *calling edges* and the *dependence edges*. A calling edge $\langle f1, f2, C \rangle$ is drawn if $f1$ is directly called by $f2$. Dependence edges are drawn according the construction rule below.

Construction Rule for Dependence Edges:

Suppose operation u in function f_1 has direct control/data dependence on another operation d in function f_2 and this dependence is a link in a dependence chain originating from an invariant. We draw a directed dependence edge from f_1 to f_2 , denoted by (f_1, f_2, D) if one of the following is true:

- Function f_1 calls f_2 (u takes place when f_2 returns to f_1)
- Function f_2 calls f_1 (d takes place before f_1 is called)
- Functions f_1 and f_2 are both directly called by a third function g

However, if none of the above is true, then f_1 's dependence on f_2 is passed along through a number of function calls and returns. For the purpose of our tracing algorithm, we draw a chain of dependences to make it clear how this dependence is propagated through a call chain. This is described below.

If there is a call chain from g to f_1 and another from g to f_2 such that no other node belongs to both call chains, we say g is an *closest common ancestor* of f_1 and f_2 . We find all *closest common ancestors* of f_1 and f_2 in the call graph.

Next, for each closest common ancestor of f_1 and f_2 , say g , we find two of its callees, g_1 and g_2 , one in the path from g to f_1 the other in the path from g to f_2 . We draw a chain of dependence edges connecting f_1 all the way to g_1 along the first call chain. Next we draw another chain of dependence from g_2 to f_2 , opposite the direction of the other call chain. Finally, we connect these two chains of dependences by the edge (g_1, g_2, D) □

By following call edges and dependence edges, all dependences can be found in this graph by transitivity. Unless specified otherwise, functions mentioned in the rest of the paper refer to those in the invariant-based PFDG, and all variables mentioned will be those used in the invariants or those affecting the variables in the invariants.

Example:

```
f {
  f1(); //
  inv(); //use x
}

f1 {
  f2(); //define x
```

In the example above, $inv()$ is assumed to be an invariant-checking function. We have call edges (inv, f, C) , (f_2, f_1, C) and dependence edges (f_1, f_2, D) and (inv, f_1, D) .

Figure 2 shows another piece of program and its invariant-based PFDG. Here the function $Inv_fun()$ is an invariant-checking function and function $f_3()$ and $f_4()$ both modify some variables used in the invariants.

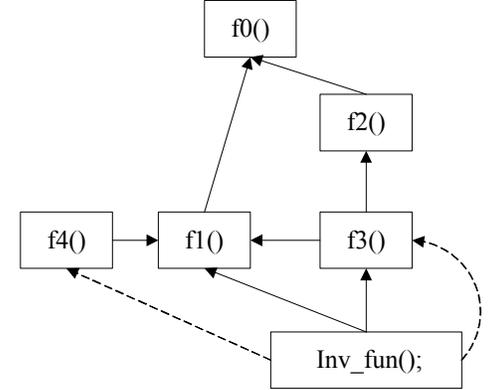
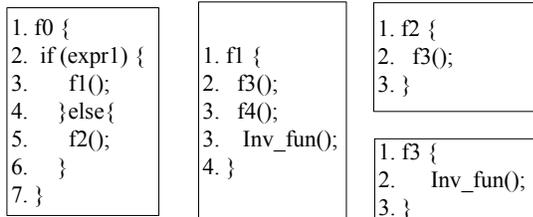


Fig 2. An example of invariant-based PFDG (Solid arcs represent call (C) edges and dotted arcs represent dependence (D) edges. In this example, an operation within $Inv_fun()$ uses a value passed from the caller $f_4()$ and another value passed from the caller $f_3()$.)

Definition 4.2 In an invariant-based PFDG, a sequence of connecting edges is called a *canonical path* if the sequence originates from an invariant-checking function inv and is composed by a prefix $\alpha = (inv, f_1, C), (f_1, f_2, C), \dots, (f_{m-1}, f_m, C)$, with calling edges only, and a postfix $\beta = (f_m, g_1, D), (g_1, g_2, D), \dots, (g_{n-1}, g_n, D)$, with dependence edges only. The prefix or the postfix may be empty, but not both.

Definition 4.3 In an invariant-based PFDG, a function f is said to be at the level n ($n \geq 1$) if, among all canonical paths ending with f , the shortest path has the length n .

The reason for us to require each canonical path to have clearly separated prefix and postfix is to have a clearly defined set of functions where we can record variable values for replaying. In order to make replay possible, in addition to the four types of logs discussed in the previous section, we need to record additional information for boundary functions defined below.

Definition 4.4 In an invariant-based PFDG, a function f is said to be a *boundary function* for level- n tracing if there exist an n -long canonical path ending with f which consists of call edges only.

In our iterative debugging procedure, what to be included in level- n tracing depends on the result of tracing and replay at the lower levels. Our iterative procedure can start with any level m , as long as all functions at levels m or lower are all included for instrumentation. Without loss of generality, we assume the procedure starts at level 1. The functions to be instrumented include all level-1 functions and all interrupt handlers which may modify any global variables used by any level-1 functions.

Obviously, for level-1 tracing, all immediate callers of an invariant-checking function are boundary functions. At the entry of each boundary function we record the entire calling context at run time, i.e. all global variable values and the arguments passed to the function. For all non-boundary level-1 functions, i.e. those non-interrupt functions connected by D edges from an invariant-checking function only, logs of types 1-3 are recorded but not the entire calling context. For all interrupt functions that receive external inputs, logs of type 5 are also recorded.

If an instrumented function calls a higher-level function g (which is not instrumented), g 's return value (if any) and the

global variables written by g when g returns are recorded. Nothing else in g is recorded no matter what non-instrumented routines are called within g . At replay, the program statements in g are not replayed, but its return value and modified global variables are used to continue the execution of g 's caller. This way, we limit the size of the instrumented code and the recorded trace.

Note that, during replay, the level-1 functions may be executed multiple times while the program statements belonging to higher-level functions are skipped in between.

Since the invariant-checking functions are always replayed, violation of invariants will always be detected. The programmer, using debugging tools such as GNU's `gdb`, can follow the program execution and produce a replayed execution trace. The statements along the trace leading to the error can be examined, which will have one of the two outcomes: the faulty statements (or the unexpected events) which cause the error are found, or such statements (or events) lie outside the level-1 trace. In the former case, debugging is done. In the latter case, the execution path extends beyond the level-1 trace. Mapping this non-ending path back to the invariant-based PDFG, we obtain a subset of canonical paths which are called *error-hiding paths* from level-1 tracing.

Next, we inductively assume level- $(n-1)$ tracing has not led to the discovery of the source of the error but has marked error-hiding paths from all level- m tracing ($m < n$). We present the following algorithm for level- n tracing.

Algorithm 4.1 Determine which functions should be instrumented for level- n tracing

Steps:

1. Let S be the set of functions to be instrumented.
2. Add all functions in the error-hiding paths from level- m tracing ($m < n$) to S .
3. Add every level- n function which is immediately reachable from any error-hiding path (i.e. can be connected by a single edge from a node in the path) to S .
4. Add all invariant-checking functions and to S .

□

Among all functions in S , we find the boundary functions for level- n tracing according to the invariant-based PDFG. We add recording operations in these functions to record the entire calling context. The rest of the instrumentation follows the same discussion in the case of level-1 tracing.

In practice, one can be flexible when using our iterative tracing procedure. If the original program size is too large for even level-1 tracing described above, one can a subset of level-1 functions as long as the side-effect of their callees are recorded to allow replay to continue. The invariant-checking functions must always be executed for tracing, so that the error can at least be detected. If the subset chosen for level-1 tracing does not lead to the discovery of the error source, another subset is chosen, and so on. On the other hand, if the size of the original program is small, one can start with level- m tracing, with $m > 1$. The relationship between the original code size, the available program memory and the choice of m is not explored further in this paper.

4.2 Termination of the Iterative Tracing Procedure

If the replay for the level- n tracing does not lead to the discovery of the error source and neither does it repeat any of the previous execution paths, then the execution paths used for the next level tracing will accumulate further. The tracing may also lead to the violation of another invariant, the level-1 tracing for the new violation will then be mixed with tracing for the previous

violations. All these may theoretically cause the instrumented code size to exceed the available program memory.

However, if we assume that the error-hiding path found in level- m tracing always repeats itself in level $m+1$ tracing, then, obviously, the iterative tracing and replay will eventually expose the error source by replay, as long as the instrumentation of all functions in the error-hiding paths always fit in the program memory. Note that the program memory required in this case will usually be significantly less than full instrumentation, because we instrument along a single path. Also note that, even though under nondeterministic external inputs the program may take different execution paths in each deployment or each tracing, the *function call/dependence paths* leading to the violation of the invariant, i.e. the error-hiding path, may still be the same. Our assumption here, therefore, accommodates nondeterministic behavior to a certain degree, even though it is not ideal.

4.3 Decision on Whether to Inline a Function

To further reduce the code size after tracing instrumentation, we notice that we can reduce the number of logs of type 1 if we inline function calls. Of course, interrupt handlers cannot be inlined. On the other hand, if a function is called in more than one place in the program, then inlining may increase the program size due to duplication of the function body. Fortunately, the inlining decisions for different functions are independent and the cost model is simple. For each function, let $S_{original}$ be the code size

before instrumentation and S_{Instr_func} be the increased code size due to inserted operations to write LOG types 2, 3 and 4. (Type 5 is recorded in interrupt handlers only, which are never in-lined.)

Further, let S_{call} the increased code size due to inserted operations to write LOG type 1. For inlining to be beneficial for the function under consideration, we must have

$$(S_{original} + S_{Instr_func})n < S_{call} \quad (1)$$

5. Implementation and Experiments

5.1 Implementation

We have implemented a preliminary version of the proposal tool targeting WSN applications based on TinyOS 2x executed on TelosB motes. A TelosB has 48KB program memory and 1MB external flash memory for data. The program analyses and transformations proposed in this paper are incorporated in an Open64 C compiler [13] which is a widely used compiler infrastructure that supports a rich set of code analysis and transformation features.

Figure 3 shows the framework of our dependence-based multi-level tracing and replaying tool. A TinyOS application written in the nesC language, with invariants specified for certain program scopes, is first preprocessed by a tool to automatically insert the invariant checking operations in the nesC program. The program is then compiled by a nesC [20] compiler (version 1.2.9) into a C program which is analyzed by a customized Open64 C compiler for dependence information before being transformed by the same compiler into two copies of programs. One is a C program instrumented with the LOG writing operations and the other is another C program for replay on desktop machines. The C program with LOG writing instrumentation is finally compiled by the native compiler to run on TelosB.

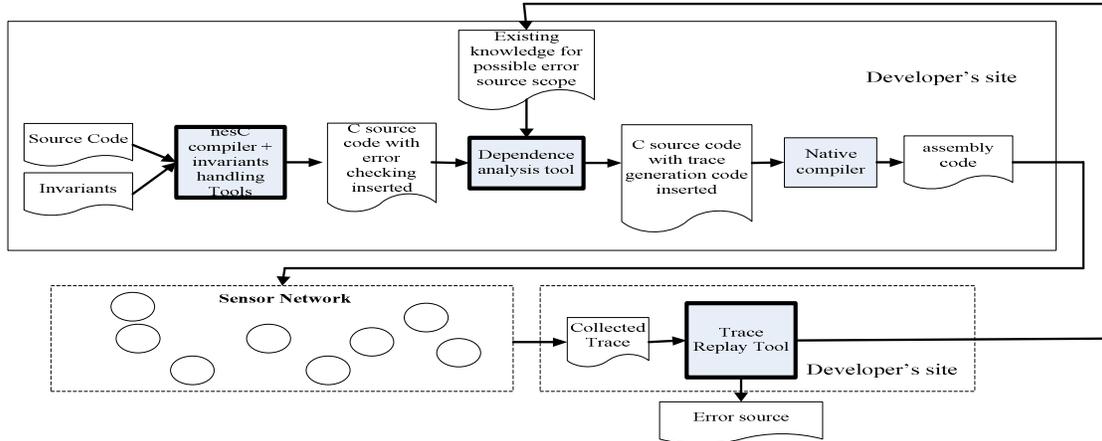


Fig 3 Framework of dependence-based multi-level tracing and replay for WSN debugging

Each invariant inserted in the WSN application specifies certain correctness property based on local information only. If a property concerns a global behavior, it is first decomposed into a set of “local” invariants before they are inserted in the nesC program. In this paper, we consider only those global properties that can be decomposed into a set of local ones. The issue of decomposing global properties into logical expressions over local properties has been discussed extensively in literature and will not be addressed in this paper.

Currently, we use a trace buffer of the size of 2KB in the RAM for LOG recording which is transferred to the external flash memory when the buffer is full.

5.2 Experiment

For experiments, we have used the following three test cases.

- TC1 (BlinkC) -- This is a published TinyOS 2x application. We modified it slightly by inserting a long running task which increases the latency of Timer.fire(). We insert an invariant which requires that the frequency of three LED’s blinking must follow a user specified pattern.
- TC2 (TestSerialCO2) -- This application monitors indoor CO2 data in multiple locations inside a building. We insert an invariant requiring that, from each mote, the base station must receive a new piece of CO2 reading every 10 seconds or less.
- TC3 (EasyCollectionC) -- This is a published code which implements the Collection Tree Protocol. We insert invariants require that the data must be sent in sequence.

To show the potential for dependence-based tracing, Table 1 compares the number of the functions traced using the dependence information with those without such information. The data indicate that, with a single invariant consisting of fewer than 3 variables, from 40% to 85% functions (not including interrupts handlers) can be skipped for tracing. However, the number of functions to be traced remains to be large for the test cases TC2 and TC3.

Figure 4 shows how many functions can be inlined to reduce the size of instrumented code for each test case. Based on the

result, over 70% of the functions are called only once and, based on the simple cost model, can be inlined. Table 2 lists the code size under different instrumentation schemes in comparison with the original code size, $S_{original}$. For the baseline code size, $S_{baseline}$, we include the inserted operations to record all types of log information without taking advantage of dependence information. The data show that the baseline size is too large for the program memory on TelosB motes. The column $S_{no-inlining}$ shows the remaining code size if we do not trace functions which have no effect on the invariants. It is much smaller than the baseline size, but still large. Take TC2 for example, the size of its $S_{no-inlining}$ is 46694, which is very close to the Telosb memory boundary size 48K. The column S_{inline} shows the code size after selective inlining. After inlining, the code size is decreased further. Of course, if many invariants are checked in the same program or some invariants involve many variables, then the use-def chains may cover more functions and the instrumented program size may increase. In the worst case, the code size may be too large to fit in the program memory, in which case multi-level tracing will be needed.

Table 1 Functions Instrumented Using Dependence Information as a Fraction of the Total Functions

Test case	# of traced functions	# of total functions	%
TC1	46	299	15.38
TC2	605	1499	40.36
TC3	604	1385	43.61

Table 2 Code Size (bytes)

	$S_{original}$	$S_{baseline}$	$S_{no-inlining}$	S_{inline}
TC1	2650	17918	10428	10201
TC2	26102	75566	46694	37376
TC3	18670	64840	39389	30148

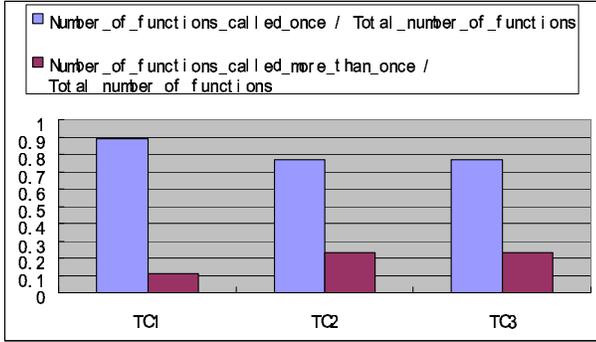


Figure 4 Inlined functions as a fraction of the total

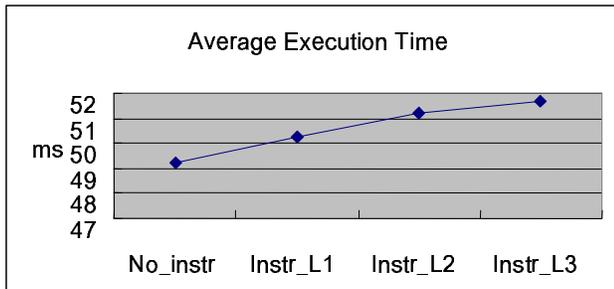


Figure 5 Comparing average execution time between two message send operations with and without Instrumentation for TC3

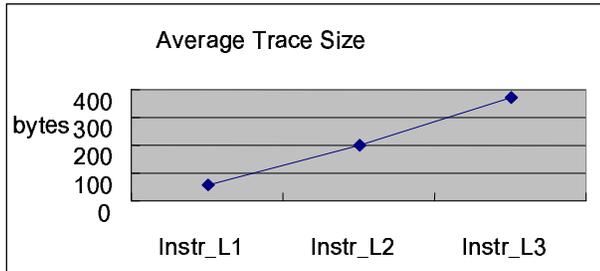


Figure 6 Comparing average trace size between two message send operations with different levels of instrumentation for TC3

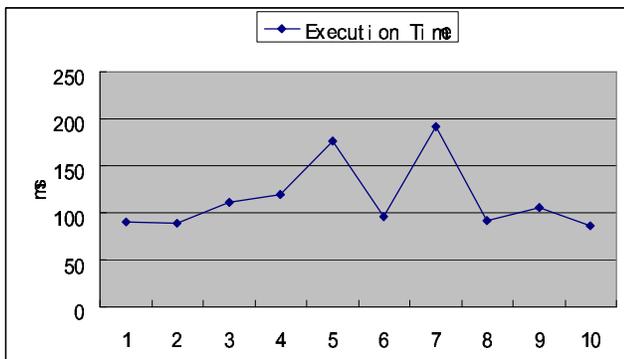


Figure 7 The execution time between two message send operations when errors occur in TC3

For each call to a task function which does not contain an anchor point, the storage used for log trace is 4 bytes (2 bytes for function entry and 2 bytes for function return). At each anchor, each saved variable requires a record of 5 bytes, including 2 bytes for the variable name, 1 byte for the variable type, and 2 bytes for the variable value. To save a nondeterministic input or the current #gv reference value, each variable also takes 5 bytes. The extra time spent to record the log for a task function call is under a microsecond on the TelosB, if the function contains no anchor point. An anchor point typically consumes 13.3 microseconds to record the variables in the case of TC3.

We used TC3 for the multi-level tracing and replay experiments. In this program, we send a piece of data (called a message) from the mote every 50ms. When a TinyOS program sends a message, it first checks to see whether the send-busy flag is raised (which indicates that the send buffer being full). If not, it is unsafe for the program to start sending a message. If the frequency of messages is low, failure to check the send-busy flag may not cause lost messages because the buffer is more likely to be free anyway. However, at a higher frequency, e.g. when the motes communicate frequently to form a cluster or to execute a security protocol, the chance for the buffer to become full increases, so do lost messages. We injected a programming error which lets the sender send the message without checking the send-busy flag first. We then load various versions of the instrumented code on the motes to execute, as separate experiments.

The run-time overhead due to instrumentation increases with the number of instrumented functions. For TC3, we measure the time interval between two Send.sendDone events. At each event a message in the send queue is sent. Fig 5 compares the average time interval for no instrumentation (No_Instr), level-1 instrumentation (Instr_L1), level-2 instrumentation (Instr_L2), and level-3 instrumentation (Instr_L3). Fig 6 shows the recorded trace size for each kind of instrumentation. From these figures, we observe that, until errors occur, the instrumented code does not incur much overhead in time and space.

Once an error occur but is not yet detected, the execution time is increased. Fig 7 collects the data from different time intervals leading to detected invariant violation. The code is traced at level 3. The time interval is lengthened due to the following reason. When the send-busy flag is raised, the sender is not supposed to send a message. With the injected error, the sender initiates a message send operation nonetheless. It then finds itself unable to post the sending task because the buffer is full. Eventually a queued message leaves the buffer, which triggers a Send.sendDone() event. As a result, the time interval between two Send.sendDone() is increased even without the instrumentation.

6. Related Work

Methods for error diagnosis and debugging for wireless sensor networks can be loosely classified into three categories, simulation/emulation, interactive debugging, and run-time logging.

A number of previous efforts [4][6][9][37] based on simulation or emulation support features ranging from parallel debugging to GUI-based debugging for programmers to simulate program execution on large-scales networks. Due to limited testing scope and rather significant difference between the simulation environment and the real operation environment, however, many unanticipated errors can still occur in systems deployed after simulation/emulation.

Interactive debugging [7][14] allows programmers to interact with sensor nodes by sending commands. The set of commands usually include those which set break points, watch points, and initiate step-by-step tracing. This methodology works particularly well if the programmer already knows what kind of errors will happen and where are the places to look. Otherwise, the step-by-step execution can be quite slow and tedious, with no guarantee that the anticipated error will surface in the debugging mode. In other circumstances, especially when the number of nodes to be debugged simultaneously is large, it seems much more convenient to have execution traces ready when an error is detected.

Run-time logging has gained increased importance recently. The critical questions encountered when adopting this approach include what kind of errors should be monitored, where and how to log information for later debugging, and how to analyze the logged information to find out the error cause. Among recent efforts, Sympathy [1] focuses on data-collection applications. The matrices generated by each node are sent to a data sink, and a decision tree is applied to the collected data to find the failures. Dustminer [2] is a tool for uncovering bugs in networked sensing applications due to nondeterministic and incorrect interactions between different nodes. This tool collects a sequence of events and uses data mining techniques to recognize abnormal behaviors. PAD [3] is a light-weight packet marking scheme for collecting necessary hints, and it uses a probabilistic inference model residing at the sink to capture unique features of the sensor networks. Passive Distributed Assertions [11] allows the programmer to define certain properties of a distributed system. The state information of each affected node is collected and analyzed through a separately-deployed sniffer network. PD2 [8] focuses on the data flows generated by an application. It relates poor application performance to significant data losses or latencies of certain data flows (called problematic data flows) as they go through the software modules on individual nodes and through the network.

Replay has long been widely used for bug reproduction. As mentioned in the introduction, this approach has mainly been used on resource-rich distributed and parallel systems. We briefly describe software-only deterministic replay techniques, given that our work is software-only. A typical and popular idea is to record all possible factors (referred to as non-determinism) that affect the program's execution before re-executing the program. The idea is straightforward, but potential overhead is large.

A significant number of prior efforts have focused on how to reduce the overhead in terms of space and execution time [32-34]. As an example, iDNA [35] developed by Microsoft logs memory instruction input values and maintains a copy of user-level memory, which is used to identify system-call side-effects. This tool is OS-independent, and it handles side-effects such as DMA transfers and direct-mapped I/O. Its log file is large. PinPlay [36] developed by Intel is a framework for deterministic capture and reproducible analysis of parallel programs. PinPlay is based on the Pin dynamic instrumentation. It has been used to identify the sources of nondeterminism in serial and parallel programs, and it employs several ways to control the non-determinism. It can also be integrated with other Pintools, many of which are used for selecting reproducible simulation points and for simulation and tracing on large parallel programs running on multiprocessors. Another use of these Pintools is to support repeatable debugging. PinPlay is OS-independent and quite high overhead.

PRES [30] is an attempt to significantly lower the production-run recording overhead by recording only partial replay information. Based on the recorded sketching, the tool navigates a non-deterministic execution space several times, trying to reproduce the errors. After several replay attempts, PRES can then reproduce the error with 100% probability on every subsequent replay for diagnostic purpose. ODR [31] addresses the output-failure replay problem by using output-determinism rather than value-determinism. That is, it generates a run that exhibits the same outputs as the original rather than an identical replica in order to achieve low-overhead recording of multiprocessor runs.

However, the replay approaches mentioned above cannot be used practically on WSNs due to resource limitation and the presence of interrupts, which motivates the work presented in this paper.

7. Conclusion and Future Work

In this paper, we have presented a multi-level tracing method based on dependence information. Our experiments show that the approach has made it possible to instrument several test programs on WSN under the stringent program memory constraint and find injected errors.

Although our current experiments are performed on TinyOS-based applications, the proposed methodology and tool can be applied to all embedded systems which satisfy the assumptions made in the introduction. It can also be extended to a broader range of embedded systems.

Several improvements are considered for future work. We plan to cover a larger set of realistic errors. More experiments are needed to apply the tool to additional WSN applications and other types of distributed embedded systems.

References

- [1] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. *In SenSys*, 2005.
- [2] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. *In Sensys*, 2008.
- [3] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong. Passive diagnosis for wireless sensor networks. *In Sensys*, 2008.
- [4] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. *I Proceedings of the 2004 USENIX Technical Conference*, 2004, pp. 283–296.
- [5] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. *In Sensys*, 2008.
- [6] P. Levis and N. Lee. Tossim: Accurate and scalable simulation of entire tinyos applications. *In SenSys*, 2003.
- [7] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. *In ACM SenSys*, 2007.
- [8] Z. Chen, K. G. Shin. Post-Deployment Performance Debugging in Wireless Sensor Networks. *In 30th IEEE Real-Time Systems Symposium*. 2009.
- [9] Y. Wen, R. Wolski. s2db: A novel simulation-based debugger for sensor network applications. *UCSB 2006*, 2006-01

- [10] D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*. 2(1-4):31-45, March-December 1993.
- [11] K. Römer, J. Ma. PDA: Passive distributed assertions for sensor networks. In *ACM IPSN*, 2009
- [12] <http://www.tinyos.net/community.html>
- [13] <http://www.open64.net/>
- [14] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN'06*, 2006.
- [15] V. Krunić, E. Trumpler, R. Han. NodeMD: Diagnosing Node-Level Faults in Remote Wireless Sensor Systems. In *MobiSys*, 2007.
- [16] M. Diaz, G. Juanole, and J. Courtiat. Observer-A Concept for Formal On-Line Validation of Distributed Systems. *IEEE Transactions on Software Engineering*, 20(12), 1994.
- [17] G. Khanna, P. Varadarajan, and S. Bagchi. Self Checking Network Protocols: A Monitor Based Approach. In *International Symposium on Reliable Distributed Systems*, pages 18–30, 2004.
- [18] M. Zulkernine and R. E. Seviora. A Compositional Approach to Monitoring Distributed Systems. In *International Conference on Dependable Systems and Networks*, 2002, pp. 763–772.
- [19] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. 1981.
- [20] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.
- [21] K. Sen, A. Vardhan, G. Agha and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *proceedings of 26th International Conference on Software Engineering*, 2004
- [22] L. Lamport, The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):8720923, 1994
- [23] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, 2009.
- [24] P. Li, J. Regehr. T-Check: Bug Finding for Sensor Networks. *IPSN'10*, 2010
- [25] Y. Pan, D. Pan, and M. Chen. Slicing component-based systems. *10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005
- [26] J. T. Lalchandani and R. Mall. Regression testing based on slicing of component-based software architectures. In *proceedings of the 1st India software engineering conference*. Pages: 67-76. 2008
- [27] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A Wireless Sensor Network For Structural Monitoring. In *Proc. of ACM SenSys*, 2004.
- [28] T. He, S. Krishnamurthy, J. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. Energy-Efficient Surveillance System using Wireless Sensor Networks. In *Proc. of ACM MobiSys*, 2004.
- [29] M. Zulkernine and R. E. Seviora. A Compositional Approach to Monitoring Distributed Systems. In *International Conference on Dependable Systems and Networks*, pages 763–772, 2002
- [30] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, October 11-14, 2009, Big Sky, Montana, USA.
- [31] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, October 11-14, 2009, Big Sky, Montana, USA.
- [32] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, 2007.
- [33] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *T. C. Bressoud and M. F. Kaashoek, editors, SOSP*, pages 131–144. ACM, 2007.
- [34] M. Olszewski, J. Ansel, and S. P. Amarasinghe. Kendo: efficient deterministic multithreading in software. In M. L. Soffa and M. J. Irwin, editors, *ASPLOS*, pages 97–108. ACM, 2009.
- [35] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihoćka, and J. Chau. Framework for instructionlevel tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE)*, pages 154–163, 2006.
- [36] H. Patil, C. Pereira, M. Stallcup, G. Lueck, J. Cownie: PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs, *CGO'10*, 2010
- [37] G. Tolle, D. Culler: Design of an Application-Cooperative Management System for Wireless Sensor Networks, *Proceedings of Second European Workshop on Wireless Sensor Networks*, pp. 121- 132, Jan. 31- Feb. 2. 2005