Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 953593, 14 pages
http://dx.doi.org/10.1155/2013/953593

Research Article

Verification of Data Races in Concurrent Interrupt Handlers

Guy Martin Tchamgoue, Kyong Hoon Kim, and Yong-Kee Jun

Department of Informatics, Gyeongsang National University, 660-701 Jinju, Republic of Korea

Correspondence should be addressed to Yong-Kee Jun; jun@gnu.ac.kr

Received 22 May 2013; Accepted 11 September 2013

Academic Editor: Tai-hoon Kim

Data races are common in interrupt-driven programs and have already led to well-known real-world problems. Unfortunately, existing dynamic tools for reporting data races in interrupt-driven programs are not only unsound, but they also fail to verify the existence of data races in such programs. This paper presents an efficient and scalable on-the-fly technique that precisely detects, without false positives, apparent data races in interrupt-driven programs. The technique combines a tailored lightweight labeling scheme to maintain logical concurrency between the main program and every instance of its interrupt handlers with a precise detection protocol that analyzes conflicting shared memory accesses by storing at most two accesses for each shared variable. We implemented a prototype of this technique, called *i*Race, on top of the Avrora simulation framework. An empirical evaluation of *i*Race revealed the presence of data races in some existing TinyOS components and applications with a worst-case slowdown of only about 6 times on average and an increased average memory consumption of only about 20% in comparison with the original program execution. The evaluation also proved that the labeling scheme alone generates an average runtime overhead of only about 0.4x while consuming only about 12% more memory than the original program execution.

1. Introduction

Asynchronous interrupt handling is a common mechanism in various cyber-physical systems including sensor networks, avionics, and automotive systems, or various medical devices. However, such interrupt-driven applications are hard to be thoroughly tested and debugged since they have to deal with external asynchronous interrupts that exponentially increase the number of their execution paths at runtime. Moreover, asynchronous interrupts introduce fine-grained concurrency into interrupt-driven programs making them prone to data races [1, 2] which occur when two threads access a shared memory location without proper synchronization, and at least one of the accesses is a write.

Data races in interrupt-driven programs have already led to real-world problems like the well-known accident of the Therac-25 [3] where, as the result of a data race caused by the keyboard interrupt handler, many people received fatal radiation doses. Another example of data race is certainly the GE XA/21 bug [4] that, in August 2003, caused a total blackout for about 50 million people in the Northeast United States and Canada. It becomes therefore crucial to precisely detect data races ahead of the exploitation phase. Data races are often classified into *feasible data races* that are data races based on the possible behavior of a program and *apparent data races*, which are defined in the context of a specific program execution [2]. Apparent data races which are only based on the behavior of the explicit synchronization in a program, represent an approximation of the feasible data races that are the real data races that might happen in any specific execution of the program. Apparent data races are less accurate but easier to detect than feasible data races, whose detection has been proven NP-hard [2].

Few dynamic race detectors [5, 6] have recently been proposed for detecting data races in interrupt-driven programs. However, legacy race detectors for multithreaded programs [7–9] are very intrusive and cannot directly be applied to interrupt-driven due not only to differences between the programming models, but also to the high overhead they incur. Thus, Lee et al. [6] proposed to convert an interrupt-driven program into a corresponding multithreaded program and to use an existing dynamic race detection tool to detect data races into the newly generated program. This technique fails to capture the semantics of interrupts and incurs high space and time overheads due to the use of tools basically

designed for multithreaded programs. Erickson et al. [5] presented a technique that aims to detect data races in low-level kernel codes by randomly sampling parts of the program to be used as candidates for the race detection, and using data and code breakpoints to detect conflicting accesses to shared variables. Although some data races caused by interrupts may be reported, the tool still reports many false positives and fails to guarantee that programs are free from data races. Moreover, existing testing methods for interrupt-driven programs [11–14] are not sufficient for debugging since they can only reveal the presence of data races, but fail to adequately detect their root causes and, therefore, require important manual inspections for real data race detection. Thus, some data races still remain undetectable and make their way to the exploitation phase, leading the application into unpredictable executions sometimes with severe consequences.

Dynamic data race detection methods are generally classified into *on-the-fly* [5–9] and *postmortem* [15, 16] techniques. While postmortem techniques collect runtime data for post-execution analysis, on-the-fly techniques on the other hand will dynamically detect data races at runtime. On-the-fly techniques still require less space overhead than postmortem techniques due to the fact that unnecessary information is discarded at runtime as the detection progresses. On-the-fly techniques are generally based either on the happens-before analysis [7, 8, 17, 18], the lockset analysis [19], or the hybrid analysis [9, 20] that is a combination of the first two. However, on-the-fly techniques still suffer from the high runtime overhead they incur. Dynamic analysis methods can be a good complement to static analysis methods [21, 22] since they guarantee to isolate real data races in a particular execution instance of a program given an input from the potential data races reported by static analysis.

This paper presents an on-the-fly technique that efficiently detects apparent data races in interrupt-driven programs without false positives. The technique combines a tailored lightweight labeling scheme to maintain the logical concurrency between a program and every instance of its interrupt handlers with a precise detection protocol that analyzes conflicting accesses to shared memories by storing at most two accesses for each shared variable. The labeling scheme generates unique identifiers to track and analyze the happens-before relation between the main program and each of its interrupt handlers. The precise detection protocol eases the debugging process since it guarantees to verify the existence of data races in the monitored program by reporting at least one data race for each shared variable involved in a data race. Moreover, the detection protocol shows a space complexity of only $O(VT^2)$, while any pair of labels can be compared in time $O(\text{Log}T)$, where T designates the total number of interrupts supported by the underlying system and V the number of shared variables in the program.

We implemented a prototype of our technique, called *i*Race, on top of the Avrora simulation and analysis framework [23] that provides a time accurate simulator and an instrumentation engine [24] for programs written for the AVR microcontrollers and sensor networks. An empirical evaluation of *i*Race revealed the presence of data races in some existing TinyOS [25] components and applications with a worst-case slowdown of only about 6 times on average and an increased average memory consumption of only about 20% in comparison with the original program execution. Moreover, this evaluation also proved that the labeling scheme alone generates an average runtime overhead of only about 0.4x with an average memory consumption of only about 12% relatively to the original program execution.

The remainder of this paper is organized as follows. Section 2 presents some properties of interrupt-driven programs and introduces the notion of data races. Section 3 focuses on our lightweight labeling scheme for interrupt-driven programs. Section 4 presents our precise data race detection protocol. In Section 5, a theoretical analysis of the proposed race detection technique is given. Section 6 discusses the implementation challenges of the technique and presents our evaluation results. The conclusion and future work come in Section 7.

2. Background

This section presents the interrupt-driven programs with some important properties of interrupts and introduces the notion of data races together with some useful definitions.

2.1. Interrupt-Driven Programs. Interrupts are generally defined as *hardware signals* as they are generated by the hardware in response to an external operation or environment change. A system is then *interrupt driven* if a significant amount of its processing is initiated and influenced by external interrupts [13, 26]. For systems based on more powerful processors, interrupts are commonly managed inside the operating system kernel, hiding their complexity from programmers. However, for highly resource-constrained embedded systems with a thin kernel layer like TinyOS [25], applications have to provide and manage their own interrupt handlers [13, 26]. We will focus our attention on the latter since this kind of applications is likely to contain data races. Interrupt handling is a common mechanism in various cyber physical systems including embedded sensor networks, avionics and automotive systems, or various medical devices. However, since interrupt-driven programs come with different flavors, our description will be based on the nesC language [21] used for TinyOS applications.

An interrupt handler is an asynchronous callback routine attached to each interrupt to service it whenever it arrives in the system. When an interrupt is received, it remains pending until it is delivered or handled. Contrarily to threads, interrupt handlers cannot block: they run to completion except when preempted by another interrupt handler [13, 21, 26]. Interrupt handlers have an asymmetric preemption relation with the noninterrupt code or *task* in nesC: tasks can be preempted by interrupt handlers, but not the contrary. Tasks in nesC are not allowed to preempt themselves; they are ordered and run in a first-in first-out manner. Interrupts are nested when they preempt each other. Nested interrupts are used to allow time-sensitive operations to be handled with low latency. An interrupt is reentrant when it directly or indirectly preempts itself.

```
(1)   TOS_Msg msg; /* shared variable */
(2)   uint8_t packet Index; /* shared variable */
(3)   async event result_t ADC.dataReady (uint16_t data) {
(4)       struct OscopeMsg *pack;
(5)       pack = (struct OscopeMsg *) msg.data;
(6)       pack->data [packetIndex++] = data;
(7)       if (packetIndex == BUFFER_SIZE) {
(8)         post dataTask ( );
(9)       }
(10)  }
(11)  async event result_t PhotoSensor.dataReady (uint16_t data) {
(12)      post photoTask ( );
(13)  }
(14)  task void dataTask ( ) {
(15)      struct OscopeMsg *pack;
(16)      atomic {
(17)        pack = (struct OscopeMsg *)msg.data;
(18)        check (pack);
(19)        packetIndex = 0;
(20)       }
(21)      pack->channel = 1;
(22)      pack->sourceMoteID = TOS_LOCAL_ADDRESS;
(23)       send (&msg);
(24)  }
(25)  task void photoTask ( ) {/* do something; */}
```

Algorithm 1: An example nesC program.

In order to minimize the impact of interrupt handlers on tasks, all long-latency operations must run in split-phase. With this mechanism, interrupt handlers immediately return after servicing critical operations and *post* heavy computations for later execution as new tasks. The concurrency model of interrupt-driven programs allows at runtime an exponentially increasing number of execution paths making them difficult to reason about. Since the only way to share data with interrupt handlers is through global variables, interrupt-driven programs are prone to concurrency bugs like data races [1, 2, 13, 26]. To protect sensitive parts of the program, interrupts must be *disabled* before and *enabled* only after critical sections. Interrupts can be either asynchronous or synchronous. Asynchronous interrupts are signaled by external devices such as network interfaces and can fire at any time that the corresponding device is enabled [13]. Synchronous interrupts, on the other hand, are those that arrive in response to a specific action taken by the processor, such as setting a timer.

2.2. Data Races. In shared-memory concurrent programs, data races [1, 2, 27] represent one of the most notorious class of concurrency bugs. Data races occur when two threads access a shared memory location without proper synchronization, and at least one of the accesses is a write. It is important to detect data races because they may lead programs to nondeterministic executions with unpredictable results.

For illustration, we consider the example program of Algorithm 1 that is adapted from the Oscilloscope application distributed with TinyOS [10]. This application allows a node of a sensor network to periodically sample, cache, and send data from its sensors. We assume that interrupt handlers in Algorithm 1 are not reentrant and that they automatically disable themselves before they run. We define a partial order execution graph (POEG) [28] to represent an execution instance of an interrupt-driven program. Unless explicitly mentioned, we will use the term *thread* to refer to either a task or an interrupt handler. Therefore, we define a *thread segment* as a sequence of instructions executed by a single thread and delimited by either a disabling or an enabling instruction. Thus, a thread T_k is a finite set of n sequentially ordered thread segments; that is, $T_k = \{t_1, t_2, \ldots, t_n\}$, where n represents the number of disable/enable instructions plus one, and k a task identifier or an interrupt number. We therefore introduce the following definitions.

Definition 1 (POEG). A POEG is a directed acyclic graph of 2-tuple $G = (S, E)$, where S is a set of thread segments and E a set of synchronization edges. A synchronization edge is either a task posting edge for posting operations or a task ordering edge to represent the implicit order relation that exists between tasks.

Definition 2 (happens-before [29]). Given a program execution, if a thread segment t_i runs earlier than another thread segment t_j, then t_i *happens-before* t_j, denoted $t_i \xrightarrow{hb} t_j$. If the happens-before relation (i.e., $t_i \xrightarrow{hb} t_j$ or $t_j \xrightarrow{hb} t_i$) is not satisfied between t_i and t_j, the two thread segments are concurrent and denoted by $t_i \parallel t_j$.

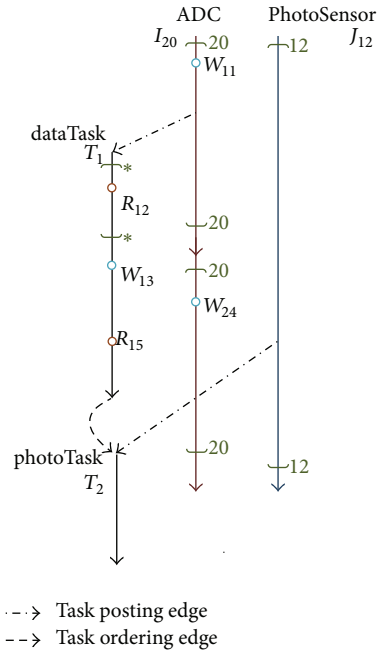

Figure 1: A POEG of an execution instance of the program in Algorithm 1. Only accesses to `msg` are represented.

In a POEG, two accesses are concurrent if and only if there is no path between them. Otherwise, they are ordered by the happens-before relation. An example POEG for the program in Algorithm 1 considering only accesses to the shared variable `msg` is represented in Figure 1. In this POEG, vertical lines symbolize threads while read and write accesses to shared variables are, respectively, represented by the letters R_{ij} and W_{ij}, where i is the thread instance number and j an observed execution order. We use the couple of symbols $\sqcap^n$ and $\sqcup^n$ to, respectively, represent the disabling and enabling of interrupts, where n is the corresponding interrupt number that will be replaced with the star symbol $(*)$ if all interrupts are concerned. The POEG of Figure 1 depicts an execution where a first instance of the ADC (Anolog-to-Digital Converter) interrupt handler collected new data and posted an instance of `dataTask`. Furthermore, another instance of the ADC interrupt arrives and preempts the execution of `dataTask`.

Although the first instance of the ADC interrupt handler posted `dataTask`, if the second ADC interrupt is received right before the message is sent at line 23 of Algorithm 1, data races will still occur between the write access in ADC and the accesses W_{13} and R_{15} in `dataTask`, making a corrupted message to be delivered. Those are the data races represented in Figure 1 by the pair of accesses W_{13}-W_{24} and R_{15}-W_{24}. If a task is posted by an interrupt handler, the task posting edge guarantees a happens-before relation only between the posted task and the actual instance of the interrupt handler that posted it, but not with future instances of the same interrupt handler. Consequently, we ignore the task posting edge when traversing the graph for analyzing the happens-before relation because we use a single arrow to represent all instances of the same interrupt. Thus, we can also report the data races W_{13}-W_{11} and R_{15}-W_{11} between T_1 and I_{20}. Therefore, in our POEG, we assume that two thread segments t_i and t_j, respectively, from two different threads T_i and T_j are always concurrent unless they are ordered by a task ordering edge or one of them is disabled by the other one. The nesC language [21] contains a static data race detector that however is still unsound since it cannot follow pointers [12, 21, 22]. Thus, the data races presented in Figure 1 cannot be reported by the native nesC data race detector.

3. Lightweight Labeling Scheme

For an effective dynamic race detection, it is essential to adequately determine the order of precedence between thread segments accessing shared memory locations. For this purpose, this section presents our lightweight labeling scheme used for capturing the logical concurrency between thread segments of interrupt-driven programs.

3.1. Generating Thread Labels. Happens-before based on-the-fly race detection techniques generally rely on a variation of Lamport's logical timestamp [29] in order to precisely identify concurrent thread segments of a program. The effectiveness of on-the-fly techniques is highly dependent on the cost and accuracy of this identification process. Our labeling scheme generates a unique identifier, called *label*, for each thread segment t_i during an execution instance of an interrupt-driven program. For the labeling scheme, we need to introduce the following definitions.

Definition 3 (critical section). A critical section with respect to a given interrupt k is a portion of program comprising between a *disable* and probably an *enable* instruction to interrupt k.

Definition 4 (interrupt set). The interrupt set or *iset* of a thread segment t_i represents the set of all interrupts k disabled by t_i. Thus, this guarantees a happens-before relation between t_i and every interrupt in its *iset*. Formally, for all $k \in t_i \cdot iset$, for all $t_j \in T_k, t_i \xrightarrow{hb} t_j \vee t_j \xrightarrow{hb} t_i$.

Definition 5 (critical section violation). A thread segment t_i of a thread T_k *violates* the critical section of another thread segment t_j, denoted by $t_i \overset{v}{\rightsquigarrow} t_j$, when t_j can be preempted by t_i.

The label of each thread segment t_i consists of two components: an identifier, referred to as *segment descriptor* SD_i, and an interrupt set $iset_i$. Put together, we use $SD_i\langle iset_i\rangle$ to designate the label of a thread segment t_i. The segment descriptor SD_i of a thread segment t_i is defined as a triplet $\langle tt_i, tid_i, cs_i\rangle$, where tt_i specifies whether tid_i is a task or an interrupt handler, tid_i contains the task identifier of a task or the interrupt number of an interrupt, and cs_i indicates whether the thread segment has disabled all interrupts in the system.

Creating labels for thread segments of an interrupt-driven program is straightforward with our labeling scheme as

suggested by the algorithms in Algorithm 2. A new labeled thread T_i is created when a new task starts its execution or whenever an interrupt handler is invoked. When a new task starts, a new label is generated for the first thread segment of the task by simply initializing the component of the label as in Algorithm 2(a). A new label is similarly created when an interrupt handler is invoked in Algorithm 2(b). Each thread then creates additional thread segments whenever it disables or enables interrupts.

When a thread segment t_i of a thread T_i disables an interrupt k, the label of the new thread segment t_j is generated by adding the interrupt number k into the interrupt set of the current label owned by t_i. However, if all interrupts in the system are disabled by t_i, the new label for t_j is obtained by setting cs_i and by emptying $iset_i$ from the current label owned by t_i as shown in Algorithm 2(c). Thus, as seen in Algorithm 2(d), if t_i enables an interrupt k, a new label for a new thread segment t_j is obtained by simply removing k from the interrupt set of the current label in t_i. If all interrupts are enabled at the same time, the flag cs_i and the interrupt set $iset_i$ are both reset from the current label of t_i to generate the new label for the new thread segment t_j. This helps maintain a reduced-size interrupt set for each thread segment since programmers generally disable and enable all interrupts rather than manipulating individual interrupts.

Moreover, the size of an interrupt set is bounded by the total number of interrupts supported by the hardware on which the program runs, which is generally a reasonable integer number. By reusing the same label for different instances of the same interrupt handler, our labeling scheme does not depend on the total number of interrupts received at runtime which can be very high.

Lemma 6. *Given two labeled thread segments t_i and t_j, with t_i belonging to an interrupt handler T_k, $t_i \overset{v}{\rightsquigarrow} t_j \Rightarrow k \notin iset_j \vee cs_j = false$.*

Proof. Let us consider two labeled thread segments t_i and t_j, and let us assume that t_i belongs to an interrupt handler T_k. We also assume that t_i violates the critical section of t_j. By definition, thread segments are delimited by either the disable or enable instructions. Therefore, if t_j was created after a the disable instruction targeting interrupt k, the interrupt number k will be in the interrupt set $iset_j$ of t_j according to the labeling algorithms of Algorithm 2. However, if t_j was created after the disabling of all interrupts, cs_j would then be true by the labeling algorithms of Algorithm 2 to signify that all interrupts including k are disabled. Thus, for all $t_i \in T_k$, t_i cannot violate t_j, which is contradictory since we assumed $t_i \overset{v}{\rightsquigarrow} t_j$. Therefore, t_j has necessarily enabled k; that is, for all $t_i \in T_k$, $t_i \overset{v}{\rightsquigarrow} t_j \Rightarrow k \notin iset_j \vee cs_j = false$. □

3.2. Concurrency Relation. Using our labeling scheme, it becomes easy to analyze the happens-before relation between any pair of thread segments, t_i and t_j, by comparing their labels.

```
[a. Task Starting]
a1:  tsk_start()
a2:      Label Lb
a3:      Lb.t = true
a4:      Lb.tid = getTaskid()
a5:      Lb.cs = false
a6:      Lb.iset = ∅
a7:      return Lb
[b. Interrupt Invocation]
b1:  int_invoke()
b2:      Label Lb
b3:      Lb.t = false
b4:      Lb.tid = getIntNumber()
b5:      Lb.cs = false
b6:      Lb.iset = ∅
b7:      return Lb
[c. Interrupt Disabling]
c1:  int_disable(Label Lb, Interrupt n)
c2:      if(n > 0) Lb.iset ∪ n
c3:      else
c4:        Lb.cs = true
c5:        Lb.iset = ∅
c6:      endif
c7:      return Lb
[d. Interrupt Enabling]
d1:  int_enable(Label Lb, Interrupt n)
d2:      if(n > 0) Lb.iset-n
d3:      else
d4:        Lb.cs = false
d5:        Lb.iset = ∅
d6:      endif
d7:      return Lb
```

Algorithm 2: Labeling algorithms.

Theorem 7. *A labeled thread segment t_i happens before another labeled thread segment t_j, denoted by $t_i \overset{hb}{\longrightarrow} t_j$, if and only if either that t_i and t_j belong to tasks, or that t_j does not violate t_i; that is, $\neg(t_j \overset{v}{\rightsquigarrow} t_i)$.*

Proof. Let us assume two labeled thread segments t_i and t_j, such that $t_i \overset{hb}{\longrightarrow} t_j$. Thus, according to the labeling algorithms of Algorithm 2, we should have $tt_i = tt_j = true$, meaning that t_i and t_j are either from the same or from different tasks. By definition, tasks are all ordered and cannot preempt each other. However, if t_j is a thread segment of an interrupt handler T_k, according to Definition 4, either $cs_i = true$ because t_i disabled all interrupts or the thread identifier tid_j of t_j that corresponds to k, the interrupt number of T_k, is in $iset_i$, the interrupt set of t_i. This means that t_i is a critical section thread segment that disables either all interrupts or particularly the interrupt number k. Thus, according to Lemma 6, t_j does not violate the critical section of t_i. Therefore, we can say that $t_i \overset{hb}{\longrightarrow} t_j$ implies either that t_i and t_j belong to tasks or that $\neg(t_j \overset{v}{\rightsquigarrow} t_i)$.

Now, let us assume that t_i and t_j belong to tasks. Naturally, t_i and t_j are ordered and the labeling algorithms of Algorithm 2 will set $tt_i = tt_j = true$, meaning that $t_i \xrightarrow{hb} t_j$. On the other hand, if we assume that $\neg(t_j \overset{v}{\rightsquigarrow} t_i)$, then t_j must be part of a given interrupt handler T_k and either t_i has disabled all interrupts to set $cs_i = true$ or the thread identifier $tid_j = k$ of t_j is in the interrupt set $iset_i$ of t_i, according to Definition 4 and Lemma 6. Therefore, we can conclude that $t_i \xrightarrow{hb} t_j$. □

Any two labeled thread segments t_i and t_j are therefore concurrent if they do not satisfy the happens-before relation as stated by Theorem 7, which can formally be rewritten as follows:

$$t_i \xrightarrow{hb} t_j \iff \begin{cases} tt_i = tt_j = true \\ \text{or} \\ \left(tt_j = false\right) \wedge \left(cs_i = true \vee tid_j \in iset_i\right). \end{cases} \tag{1}$$

By comparing, based on Theorem 7, the labels of the thread segments in the POEG represented by Figure 2, we can easily obtain that the thread segments of I_{20}, i_1, and i_2 are concurrent with other thread segments t_1, t_3, t_4, and j_1. Therefore, the write accesses in i_1 and i_2 conflict with the other accesses in t_3, leading to four data races W_{11}-W_{13}, W_{11}-R_{15}, W_{24}-W_{13}, and W_{24}-R_{15}. However, since the accesses W_{11} and W_{23}, respectively, from i_1 and i_2, represent the same write access, the set of data races to be reported for an effective debugging process should be reduced to only W_{11}-W_{13} and W_{11}-R_{15}. By disabling all interrupts, t_2 enforces the happens-before relation with other thread segments. Although T_1 is posted by I_{20}, they are still concurrent. Also, the thread segment j_1 is concurrent with i_1, i_2, t_1, t_3, and t_4.

4. Precise Detection Protocol

The race detection protocol monitors memory accesses by maintaining a special data structure, called *access history*, into which are stored the labels of prior accesses to each shared variable. Our detection protocol stores, in each entry of the access history of a shared variable, a unique label representing the thread segment that accessed the variable as described in Section 3. At each access to a shared variable, data races are reported by analyzing, based on Theorem 7, the happens-before relation between the current thread segment and the labels of prior thread segments stored in the access history. Therefore, the access history for each shared variable has to be efficiently organized in order to reduce the overhead generated by the detection process.

Our precise detection protocol maintains an access history consisting of two sets for each shared variable: a *Read* set and a *Write* set, to, respectively, store labels of thread segments that read and write the variable. We use $AH(x)$ to designate the access history of a shared variable x. Thus, $AH(x) \cdot Read$ and $AH(x) \cdot Write$ refer to the read set and the write set of $AH(x)$, respectively. The algorithms used by the detection protocol are presented in Algorithm 3. In these algorithms, we assume that the variable Lb always contains

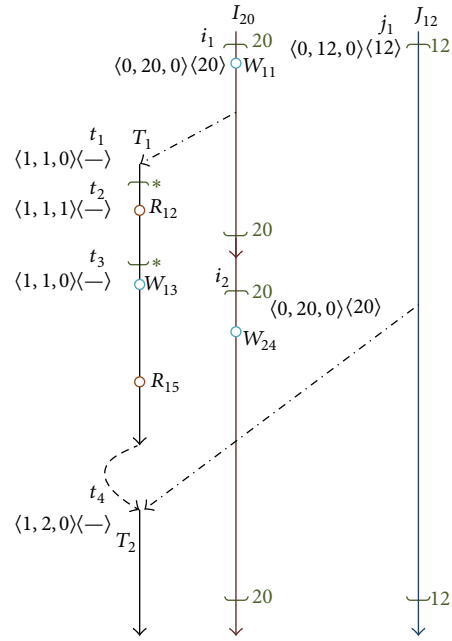

Figure 2: An example labeled POEG for an execution instance of the program in Algorithm 1.

the label of the current thread segment accessing a shared variable and that *pLb* represents any prior label stored in the access history.

When a read access occurs to a shared variable x, the detection protocol calls the **ah_read()** function of Algorithm 3(a) that then reports data races by analyzing the happens-before relation between the label of the accessing thread segment and the labels of prior accesses stored in $AH(x) \cdot Write$. Similarly, upon a write access to a shared variable, the monitoring function **ah_write()** of Algorithm 3(b) reports data races by looking into both $AH(x) \cdot Read$ and $AH(x) \cdot Write$. We assume the existence of a function **Ordered()** that, based on Theorem 7, decides whether a thread segment t_i happens before another thread segment t_j by comparing their labels.

In order to maintain a reduced size access history, accesses from critical sections that concern all interrupts within a task are not inserted in the access history since they will never run into a data race with other accesses from interrupt handlers. Also, only the first read and write accesses to a shared variable from the same thread segment are kept into the access history. Subsequent similar accesses to the same shared variable from the same thread segment are filtered out. This filtering operations are performed by the **isFiltered()** function of Algorithm 3(c) that is called by both **ah_read()** and **ah_write()** before any other action.

After data races have been reported with the current access, the function **maintain_ah()** of Algorithm 3(d) is used for keeping the size of the access history balanced. This function replaces some previous accesses in the access history with the current access. Thus, accesses from different tasks are considered from the same thread since tasks, by definition, are all ordered by happens-before. Similarly, accesses

```
(1)   ah_ read(Label Lb, Access_History AH, Variable x)
(2)     if (isFiltered(Lb, AH(x).Read)) then return;
(3)     for all labels pLb in AH(x).Write do
(4)       if (Not Ordered(Lb, pLb)) then
(5)         report_ race(x, pLb, Lb)
(6)     end for
(7)     maintain_ah(Lb, AH(x).Read)
    (a) Monitoring Protocol for Read Accesses
(1)   ah_write(Label Lb, Access_History AH,Variable x)
(2)     if (isFiltered(Lb, AH(x).Write)) then return;
(3)     for all labels pLb in
(4)       AH(x).Write ∪ AH(x).Read do
(5)       if (Not Ordered(Lb, pLb)) then
(6)         report_race(x, pLb, Lb)
(7)     end for
(8)     maintain_ah(Lb, AH(x).Write)
    (b) Monitoring Protocol for Write Accesses
(1)   isFiltered(Label Lb, Access_History AH)
(2)     if (Lb.cs = true) and (Lb.tt = true)
(3)       then return true;
(4)     if (any access with a label pLb exists in AH
(5)       such that pLb = Lb) then return true;
(6)     else
(7)       return false;
    (c) Filtering Redundant Accesses
(1)   maintain_ah(Label Lb, Access_History AH)
(2)     if (Lb.tt = true) then
(3)       delete all entries in AH with label pLb
(4)       such that pLb.tt = true
(5)     else
(6)       delete all entries in AH with label pLb such
(7)       that pLb.tt = Lb.tt and pLb.tid = Lb.tid
(8)     end if
(9)     add_ah(Lb, AH)
    (d) Maintaining Access History
```

Algorithm 3: Labeling algorithms.

from different instances of the same interrupt handler are considered from a single instance of the interrupt handler. This property is already enforced by the labeling scheme that reuses the same label for all instances of the same interrupt handler. Again, we assume the existence of another function, **add_ah()**, which is able to insert the label of the current thread segment into the appropriate set of the access history, given a shared variable.

Table 5 shows the different states of an access history for detecting data races with our protocol in the example program of Figure 2, where accesses to the shared variable `msg` appear in the first column. When the first access W_{11} occurs, the label $\langle 0, 20, 0\rangle\langle 20\rangle$ of the thread segment i_1 is simply added into the access history and no data race reported as there is no other access prior to W_{11}. The next access, R_{12}, from the thread segment t_2 is filtered out because t_2 that belongs to task T_1 has disabled all interrupts. The following access W_{13} occurs on t_3 with interrupts enabled. Thus, a data race W_{13}-W_{11} is reported by comparing the label of t_3, $\langle 1, 1, 0\rangle\langle -\rangle$, with the label of i_1 previously stored into the access history. Afterwards, the label of t_3 is inserted into the write set of the access history. The thread segment t_3 is then preempted by i_2, but the access W_{24} is filtered out, because i_2 which is a new instance of the interrupt I_{20}, also shares the same label, $\langle 0, 20, 0\rangle\langle 20\rangle$, with i_1 which previously performed a write and whose label is already in the access history. After the execution of i_2, t_3 resumes to perform the read access, R_{15}. Another data race R_{15}-W_{11} is reported and the label of t_3, $\langle 1, 1, 0\rangle\langle -\rangle$, stored into the read set of the access history. Although the program received two instances of the ADC interrupt, our protocol precisely reported only the set of two data races needed for effectively debug the program. This shows that our detection protocol does not depend on the number of interrupts received at runtime which can be very high.

Although it keeps only a reduced set of labels in the access history, our precise detection protocol even with a minimal number of interrupts, still guarantees to verify the existence of data races in a program. For this purpose, the protocol is designed to report, without false positives, at least one data race for each shared variable involved in a data race.

Theorem 8. *Given an interrupt-driven program P, the detection protocol reports a data race over a shared variable x, if and only if there exists a data race over x in an execution instance of P.*

Proof. We will first prove that if a data race is reported on a shared variable x by the detection protocol, then there also exists a data race in an execution instance of the program for the same shared variable x. For this, we assume a snapshot of the access history $AH(x)$ of the shared variable x at a given time and the current access event a_i to x from a thread segment t_i that owns a label we refer to as Lb_i in an execution instance of the program P. We can then distinguish two different cases.

Case A. $AH(x)$ is empty, and there is obviously no data race for x. In this case, the current access a_i is the only access to x from the program P and Lb_i will be added by the function **add_ah()** into $AH(x)$·*Read* if a_i is a read access or into $AH(x)$·*Write* if it is a write access.

Case B. $AH(x)$ is not empty. Data races are reported using the **ah_read()** function if a_i is a read access or the **ah_write()** in case of a write access. In both cases, the happens-before relation is analyzed between the label Lb_i of t_i and previous labels stored in $AH(x)$ according to Theorem 7. The outputs of these analyses can be classified into two categories.

(1) There is no data race between a_i and $AH(x)$: therefore, all previous accesses stored in $AH(x)$ happened before a_i meaning that no thread segment prior to t_i in P is concurrent with t_i; they are all ordered by happens-before. Thus, for all $a_j \in t_j \wedge Lb_j \in AH(x)$, $t_j \xrightarrow{hb} t_i$. In which case P also has no data race.

(2) There are data races between a_i and prior accesses in $AH(x)$: this means that there are some labels in $AH(x)$ belonging to some prior thread segments that are not ordered with t_i. Concretely, there are thread segments in P that are concurrent with t_i. Formally, $\exists t_j \in P, Lb_j \in AH(x), t_i \parallel t_j$. Therefore, the execution instance of P also contains data races over x.

Now, we show that if there exists a data race over a shared variable x in an execution instance of the program P, then our detection protocol also reports a data race over x. Using the contrapositive, we will prove that if the detection protocol does not report any data race, the program P also does not contain any for a shared variable x. We again consider a snapshot of the access history $AH(x)$ for a shared variable x at a given time. We also assume that the detection protocol reported no data races for the shared variable x. This means that $AH(x)$ is either empty and there is clearly no data race since there is no access to x from P yet, or contains only labels of thread segments that are all ordered by the happens-before relation, in which case there is no concurrent thread segment in P, which also implies that P has no data races over x. Therefore, we can formally write for all $t_i, t_j \in P \wedge Lb_i$, $Lb_j \in AH(x)$, $t_i \xrightarrow{hb} t_j \vee t_j \xrightarrow{hb} t_i$. Thus, P also contains no data race for x. □

Table 1: Efficiency of the precise detection protocol.

Space	Time		
	New label	comparison	Per access
$O(VT^2)$	$O(1)$	$O(\text{Log } T)$	$O(T\text{Log } T)$

5. Theoretical Analysis

On-the-fly data race detection techniques are known to have high runtime overhead that is generally provoked by the time spent monitoring accesses to shared variables and to compare thread labels. In this section, we theoretically analyze the efficiency of our technique which depends on: V, the number of shared variables, and T, the total number of interrupts in the underlying system plus one, the number of tasks. In general, T is very small since the number of interrupts handled by a system is also kept small. For example, the popular Atmel ATmega128 [30] processor supports only a maximum of 34 interrupts. Also, because of the implicit happens-before relation that exists between all tasks, they are viewed as a single task by the detection protocol.

Table 1 presents the space and time complexities of our labeling and detection techniques. The access history used by our technique stores from each thread segment, at most two accesses for each shared variable. Each label in the access history has a worst-case size of $O(T)$ that represents the size of its interrupt set. Therefore, the total labeling space requirement for each shared variable is $O(T^2)$. Thus, the total space complexity of our technique is $O(VT^2)$. However, because the number of interrupts is generally a reasonable integer number, the space complexity of our technique can further be reduced to $O(V)$.

A new label is always created in a constant amount of time by our labeling algorithms. Interrupts are generally given numbers in a sequence order. Therefore, a binary search technique is applied to locate values in interrupt sets which contain only interrupt numbers. Thus, the time complexity for comparing any two labels is $O(\text{Log } T)$ in the worst-case and $O(1)$ in most cases where the interrupt sets are empty. At each access to a shared variable, the total time complexity to report data races with the previous accesses stored in the access history is then $O(T\text{Log } T)$. The labels and the detection protocol do not depend on the total number of interrupts received at runtime making our technique highly scalable and practical.

6. Experimental Results

We empirically evaluated the preciseness and efficiency of our technique using a set of well-known programs provided with TinyOS [10]. This section details the implementation challenges and presents the analysis results of the experimentation.

6.1. Implementation. We implemented a prototype of our data race detection tool for interrupt-driven programs, called *i*Race, based on the concurrency model of TinyOS [21] and its nesC language [25]. We implemented *i*Race on top

Table 2: Characteristics of the benchmark applications from TinyOS 2.1.0 [10].

Applications	LOC	Int	SV	Brief description
6LowPanCli	19116	4	9	Implements a command line interface using the 6lowpan stack.
AnthiTheft	27421	5	4	Detects and reports a stolen mote by checking some sensors.
BaseStation	14905	6	13	Sends and receives simple active messages on the radio and serial links.
Blink	3688	1	0	Periodically blinks the three LEDs of a mote.
MultihopOscilloscope	24310	9	8	Periodically samples and broadcasts sensor data to a base station.
Oscilloscope	11686	5	5	Periodically samples sensor data and broadcasts a message every 10 times
Powerup	2352	0	0	Turns on the red LED when the mote powers up.
RadioCountToLeds	11165	4	3	Periodically samples sensors, and displays the last 3 bits on its LEDs.
RadioSenseToLeds	11472	5	2	Periodically update a counter, and displays the last 3 bits on its LEDs.
Sense	5743	2	0	Periodically takes sensor readings and displays the values on the LEDs.

of the Avrora [23] simulation and analysis framework that provides a time accurate simulator, different analysis tools known as *monitors*, and a binary instrumentation engine [24] for programs written for AVR microcontrollers and sensor networks. All these features make Avrora just suitable for our tool. Simulation is commonly used in the development cycle of embedded software, because target systems generally lack conventional input/output interfaces. With *i*Race, the functionalities of Avrora are extended with a precise data race detection tool that can easily be loaded as one of Avrora's monitors.

As shown in Figure 3, *i*Race consists of two main components: the Instrumenter, which exploits the Avrora instrumentation engine [24], and the core Race Detector itself. The Instrumenter monitors task and interrupts activities through its *i*Monitor component and provides entry points for the labeling routines. Instrumentation points are inserted at the beginning of each task, at the start and exit of each interrupt handler, and also before interrupts are either enabled or disabled. Similarly, accesses to shared memory locations are monitored using the AccessMonitor component that also calls routines of the detection protocol. The AccessMonitor instruments before read and write accesses to shared variables. The Race Detector is composed of labeling routines that generate and manage labels for each active thread segment at runtime, and detection routines for maintaining the access history and reporting data races on-the-fly as shared variables are manipulated.

In TinyOS [21, 25], interrupt handlers and tasks all run to completion unless preempted by other interrupt handlers. Thus, only a single thread runs at any point of time, giving us the opportunity to use only a single variable to maintain the active thread label at detection time. However, when preemption happens, the label of the preempted thread segment must be saved and restored again when the active interrupt handler exits. Otherwise, there will be no way to know about the concurrency information of the preempted thread segment when it gains back the execution control. In order to manage the preemption, we use a stack data structure where labels can be pushed and restored from since interrupts can be nested. This provides a simple but powerful way to handle preemption with negligible overhead.

Our labeling scheme needs to uniquely identify each interrupt handler whenever it is invoked and each task when it starts its execution. Each interrupt handler is easily identified by the interrupt number of the interrupt it serves. For task, we chose to generate a unique task identifier by simply maintaining a counter that increases at each time a task starts its execution. Thus, each task is attributed a unique identifier thanks to the fact that, once posted, TinyOS tasks run in a first-in first-out manner and cannot preempt each other. By manipulating only binary files, *i*Race does not require any source file nor source code annotation to effectively detect data races. However, compiling the program with debugging information may help *i*Race generate more detailed debugging information on data races it reports.

6.2. Environment. We implemented and evaluated our technique on a system that runs the kernel 3.0 of the Linux operating system using an Intel dual-core 3.20 GHz and 4 GB of main memory. On this machine was installed TinyOS 2.1.0 [10] along with the nesC compiler 1.3.1, the GNU C compiler 4.6, the Java Runtime Environment 1.7, and the latest Avrora framework version 1.7.115.

Table 2 presents the characteristics and a brief description of the benchmark applications selected for our evaluation from the TinyOS 2.1.0 repository [10]. These applications are freely distributed with each version of TinyOS. The column "LOC" indicates the number of line of C codes generated by the nesC compiler after compilation, while the column "Int" contains the number of interrupts handled by each application. The number of shared variables for each program appears in column "SV." It is worth mentioning that this number represents only the shared variables in the source code of each program. The exact number of variables shared by each application may finally depend on the number of components it shares with the operating system itself. Applications written in nesC share TinyOS components by wiring themselves to these components, increasing therefore their number of shared resources. We compiled each application for the Mica2 platform and measured the average runtime overhead over five executions. The Mica2 motes are sensor network nodes based on the ATmega128 [30] microcontrollers. We configured Avrora to simulate a sensor network during 10

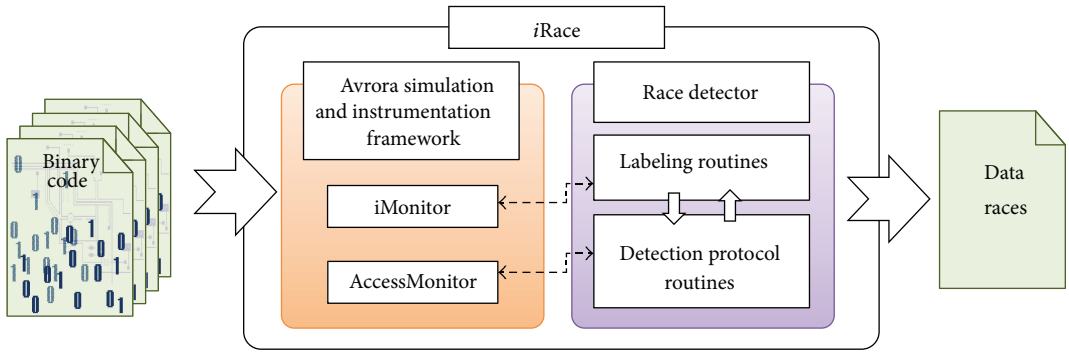

Figure 3: The Architecture of the Data Race Detection Tool.

Table 3: Reported data races on the benchmark applications.

Applications	Data races	Verif. races	Time (Sec.)		Memory (Kb)	
			Original	*i*Race	Original	*i*Race
6LowPanCli	0	0	2.69	3.57	19369.60	21681.60
AnthiTheft	**15**	15	2.92	4.57	19444.00	23200.00
BaseStation	0	0	2.79	3.76	19122.40	21482.40
Blink	0	0	1.61	1.80	18872.80	19496.00
MultihopOscilloscope	0	0	2.78	4.00	19396.00	21969.60
Oscilloscope	**13**	13	1.79	3.28	19093.60	22017.60
Powerup	0	0	1.51	1.86	18736.00	18799.20
RadioCountToLeds	**17**	17	1.96	12.55	19428.80	23263.20
RadioSenseToLeds	**15**	15	1.81	13.05	19444.80	22982.40
Sense	0	0	1.58	1.92	18664.00	20505.60

seconds allowing each application to run exactly for 73728000 instruction cycles. We set to five the number of nodes in the simulated sensor network.

6.3. Evaluation. For data race detection, we instructed Avrora to simulate a sensor network with five nodes while the benchmark applications are monitored by *i*Race. Table 3 represents the results of the evaluation of *i*Race on the benchmark applications in its column "Data Races," which contains the number of shared variables involved in a data race for each application. Our tool reported no data race for all applications except for four: `AnthiTheft` with 15 shared variables, `Oscilloscope` with 13 shared variables, `RadioCountToLeds` with 17 shared variables, and `RadioSenseToLeds` with 15 shared variables. Surprisingly, most of the reported data races reside inside some shared system components of TinyOS itself. The reported data races are caused by two different interrupts, namely, the serial peripheral interface (SPI) interrupt, and the timer interrupt. Also, we carefully analyzed the reported data races and inspected the source code of the applications for a manual verification process as shown in the column "Verified races" of Table 3.

In the `AnthiTheft` application, for example, which is used to detect and report a stolen mote by checking the level of the light or the acceleration, data races were reported over 15 shared variables. Most of the reported data races are however due to the fact that the SPI interrupt handler can preempt some functions or tasks owned by the underlying radio stack. This implies that if a message is being received or sent when a SPI interrupt occurs, shared variables like `radioState` or `count` in the component `CC1000SendReceiveP` might be corrupted. However, all shared variables involved in a data race do not directly belong to `AnthiTheft` but to functions of the `CC1000` radio stack in use in Mica2 motes. Similar data races are also reported in `RadioCountToLeds` and `RadioSenseToLeds` since they all access the radio stack.

Another data race reported both in `RadioCountToLeds` and `RadioSenseToLeds` and worthy to be commented concerns the shared variable `reqResId` inside the component `ArbiterP` (Recently, this data race has also been pointed out and a patch discussed in the TinyOS Forum). This data race occurs when the function `immediateRequest()`, called within the function `read` of the component `immediateRequest`, is preempted by a SPI interrupt handler. An excerpt of this function is presented in Algorithm 4. At line 2 of Algorithm 4, the handle of a resource is requested and eventually locked out if granted. However, if a SPI interrupt is received somewhere after line 2, a data race will occur and the resource handler stored in `reqResId` (line 6, Algorithm 4) may eventually be overwritten by the SPI interrupt handler. Also, other shared

```
(1)  async command error_t Resource.immediateRequest [uint8_t id] ( ) {
(2)      signal ResourceRequested. immediateRequested [resId] ( );
(3)      atomic {
(4)        if (state == RES_CONTROLLED) {
(5)            state = RES_IMM_GRANTING;
(6)            reqResId = id;
(7)      }
(8)      else return FAIL;
(9)      }
(10)      signal ResourceDefaultOwner. immediateRequested( );
(11)  ···;
(12)  }
```

Algorithm 4: An excerpt of the request function from the component `ArbiterP`.

```
(1)  command error_t Send. send[uint8_t clientId] (message_t * msg,
(2)                                                uint8_t len) {
(3)  ···;
(4)        queue [clientId]. msg = msg;
(5)        call Packet. setPayloadLength (msg, len);
(6)  ···;
(7)  }
```

Algorithm 5: An excerpt of the send function from the component `AMSend`.

variables in `ArbiterP` like `state` can be corrupted as well by the SPI interrupt handler.

In the case of `Oscilloscope`, which is a data collection application that periodically samples the default sensor of a mote and broadcasts a message over the radio every 10 readings, the reported data races are similar to those reported in `AnthiTheft`. Moreover, our results suggest that if a SPI interrupt arrives when a message is being transmitted via the `send()` function of the `AMSend` interface, the shared buffer `sendBuf`, used by `Oscilloscope` to store incoming messages, will be updated and a corrupted message delivered. The extract of code from the function `Send` presented in Algorithm 5 shows that the variable `msg`, which, in this case, is a pointer to the shared variable `sendBuf` in `Oscilloscope`, can be updated at any time if the function `send()` is preempted, because operations are not performed in a critical section. This data race is somehow similar to the one we described in the example program of Algorithm 1. We also find in `Oscilloscope` some of the data races reported in the other applications. The results in Table 3 show that our tool precisely detects data races on-the-fly in interrupt-driven programs without any false positives.

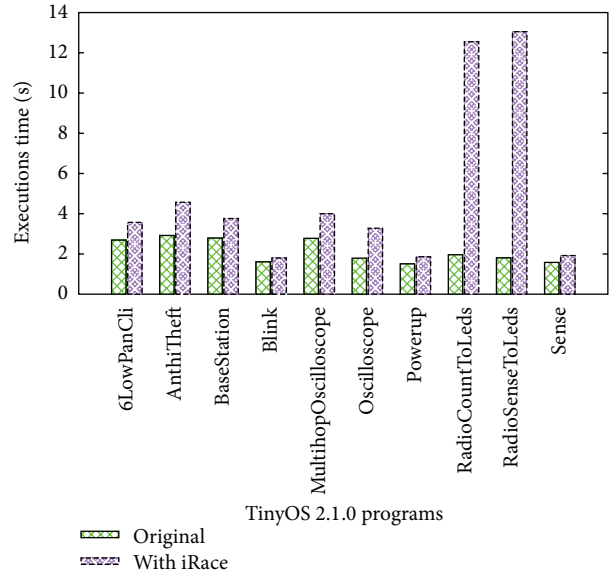

Figure 4: Average runtime overhead.

6.4. Detection Overhead. In order to evaluate the impact of our technique on the monitored programs, we configured Avrora to simulate a single node sensor network. We then measured the average runtime overhead and memory space required by each application of the benchmark of Table 2 to execute 73728000 instruction cycles. The total overhead of our technique mainly comes from actions of the instrumentation engine, the labeling routines, and the race detection engine which also has the responsibility to create and maintain the access histories. The average execution time and memory consumption collected for each application are presented in Table 3. The columns "Original" and "*i*Race" represent the execution time and memory consumption for each application without and with *i*Race, respectively.

Figure 4, which plots the data from the column "Time" of Table 3, compares the average execution times of each

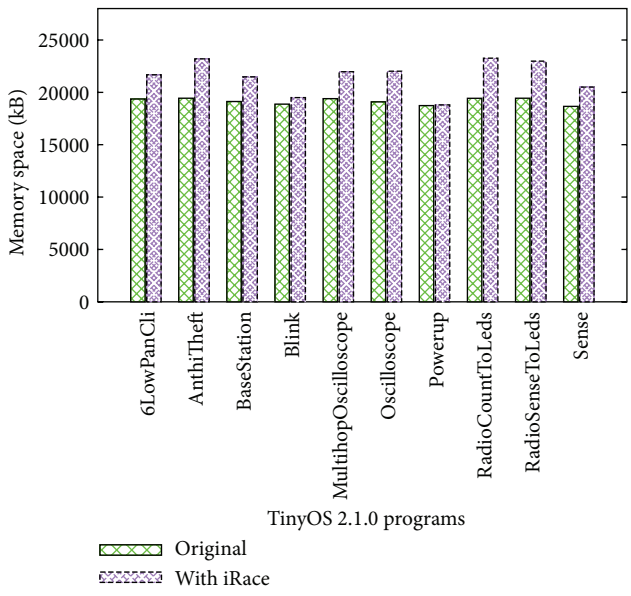

Figure 5: Average memory consumption.

application for five runs with and without *i*Race. The caption "Original" represents the native execution of the applications without *i*Race. The results of Figure 4 suggest that the average runtime overhead of our tool is only about 1x for all applications except for `RadioCountToLeds` and `RadioSenseToLeds` that reaches 6x the original execution time. These two applications sample and display data at high frequency rate (e.g., 4 Hz in our case) and, therefore, are likely to receive a higher number of interrupts at runtime. This low runtime overhead is achieved by exploiting the fact that shared variables are stored by Avrora at program loading time. Instead of monitoring all memory accesses to filter those targeting shared variables, *i*Race precisely instruments only accesses to the well-known set of shared memory locations obtained from the Avrora loader. This technique allows the tool to drastically cuts down its monitoring overhead and to increase its scalability.

Likewise, we measured the average memory consumption for each application during five runs. The collected data are represented in Figure 5. The overall average memory consumption of the tool is very small at just about 20% above the original memory requirement even for high frequency applications like `RadioCountToLeds` and `RadioSenseToLeds` thanks to our labeling scheme that reuses the same label for different instances of the same interrupt handler and the detection protocol that maintains only a reduced size access history. This result corroborates the theoretical analysis of our technique as summarized in Table 1. The low runtime overhead and memory consumption of our on-the-fly technique make it very scalable and suitable even for interrupt-driven programs that handle interrupts at high frequency rates.

6.5. Labeling Overhead. Since the race detection subcomponent of *i*Race is in charge of implementing the labeling and the detection routines, it was legitimate to analyze the impact of the labeling scheme alone on the monitored programs at runtime. For this purpose, we deactivated the detection engine of *i*Race before running it again on the benchmark applications. We collected the average execution time and memory space consumed by each application after five runs as shown in Table 4. The simulation was done using a single node network.

Table 4: Average labeling overhead.

Applications	Time (Sec.)	Memory (Kb)
6LowPanCli	3.13	21017.60
AnthiTheft	3.08	21488.80
BaseStation	3.09	20776.80
Blink	1.74	19646.40
MultihopOscilloscope	3.13	21419.20
Oscilloscope	2.49	20926.40
Powerup	1.76	18947.20
RadioCountToLeds	2.18	21674.40
RadioSenseToLeds	2.24	21400.80
Sense	1.84	20110.40

Table 5: The states of an access history for detecting data races with our protocol considering the example of Figure 2.

a_i	Read	Write	Races
W_{11}	⌀	$\langle 0,20,0\rangle$ $\langle 20\rangle$	
R_{12}	⌀	$\langle 0,20,0\rangle$ $\langle 20\rangle$	
W_{13}	⌀	$\langle 1,1,0\rangle$ $\langle —\rangle$, $\langle 0,20,0\rangle$ $\langle 20\rangle$	W_{13}-W_{11}
W_{24}	⌀	$\langle 1,1,0\rangle$ $\langle —\rangle$, $\langle 0,20,0\rangle$ $\langle 20\rangle$	
R_{15}	$\langle 1,1,0\rangle$ $\langle —\rangle$	$\langle 1,1,0\rangle$ $\langle —\rangle$, $\langle 0,20,0\rangle$ $\langle 20\rangle$	R_{15}-W_{11}

Figure 6 compares the original execution time of each program from Table 3 with the execution time of the labeling engine as presented in the column "Time" of Table 4. The comparison shows that the average runtime overhead of *i*Race with just the labeling engine activated is only about 0.4x in the worst-case. This represents only a small fraction of the average runtime overhead of 6x imposed by *i*Race as discussed in Section 6.4. The main reason of this performance can be found at source level. When browsing the source code of the benchmark applications, we came to understand that programmers generally disable and enable all the interrupts at once instead of focusing on particular interrupts. In this case, the size of each label generated by the labeling engine never approaches its worst-case of $O(T)$, meaning that the comparison overhead of labels is also reduced. This is very important, because as discussed in Section 6.4, although the size of labels does not depend on the total number of interrupts received at runtime, this number still has impact on the overall detection overhead. Thus, we can conclude that the labeling engine of *i*Race incurs a very low overhead at runtime, making it a highly scalable tool.

We also compared the memory consumption of the labeling engine with the memory requirement of the original

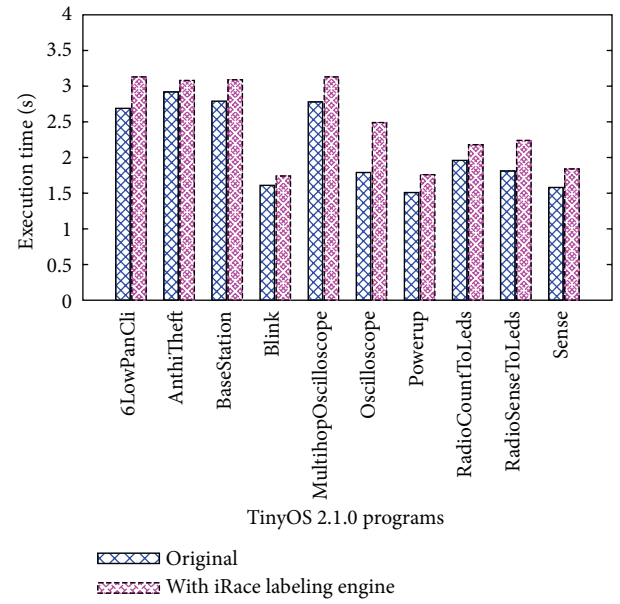

Figure 6: Average runtime overhead of the labeling engine.

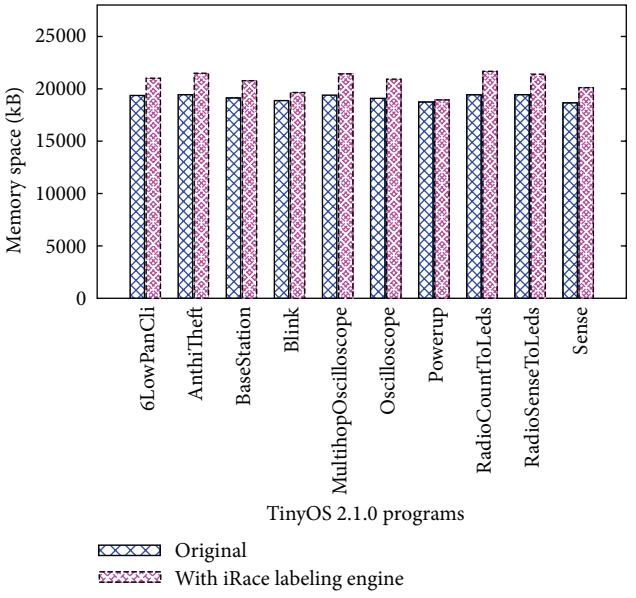

Figure 7: Average memory consumption by the labeling engine.

program execution presented in Table 3. The results plotted in Figure 7 show that when the labeling engine alone is running, *i*Race consumes only about 12% more memory on average. However, during the detection phase, the overall average memory consumption of *i*Race is only about 20%. These results also tell us that the detection engine does not increase that much the memory consumption of *i*Race. This is also reasonable because we only disabled the detection engine of *i*Race but did not remove the resources it used.

7. Conclusions

Asynchronous interrupt handling introduces fine-grained concurrency into interrupt-driven programs making them prone to data races. In this paper, we presented an efficient and scalable on-the-fly technique to precisely detect data races without false positives in such programs. Our technique introduced a lightweight labeling scheme for maintaining concurrency information for the main program and each of its interrupt handlers and a precise detection protocol that guarantees to verify the existence of data races in a program by reporting at least one data race for each shared variable involved in a data race.

We implemented a prototype of our technique as a race detection tool, called *i*Race, on top of the Avrora simulation framework. An evaluation of *i*Race revealed the existence of data races in some programs distributed with TinyOS and in some TinyOS components while incurring a slowdown of only about 6 times on average in the worst-case with an average memory consumption of only about 20% above the original memory requirements of the program. Moreover, this evaluation also proved that the labeling engine alone generates an average runtime overhead of only about 0.4x with an average memory consumption of only about 12% relatively to the original program execution. These results coupled with a theoretical analysis showed that the proposed technique is highly scalable and practical even for large interrupt-driven programs.

Detecting data races that occur first in a program is an important issue, since it may make other data races to disappear. It is therefore interesting to detect this class of data races in interrupt-driven programs to facilitate the debugging process. Also, we may be interested in extending our work to support multithreaded programs that handle interrupts and also to detect data races in low-level kernel codes.

Acknowledgment

This research was financially supported by the Ministry of Education (MOE) and National Research Foundation of Korea (NRF) through the Human Resource Training Project for Regional Innovation and BK21+ program.

References

[1] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "A theory of data race detection," in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD '06)*, pp. 69–78, July 2006.

[2] R. H. B. Netzer and B. P. Miller, "What are race conditions? Some issues and formalizations," *ACM letters on programming languages and systems*, vol. 1, no. 1, pp. 74–88, 1992.

[3] N. G. Leveson and C. S. Turner, "Investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[4] U.S.A., Canada. *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations.*; U.S.-Canada Power System Outage Task Force, 2004.

[5] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *Proceedings of*

the 9th Conference on Operating Systems Design and Implementation (USENIX '10), 2010.

[6] B.-K. Lee, M.-H. Kang, K. C. Park, J. S. Yi, S. W. Yang, and Y.-K. Jun, "Program conversion for detecting data races in concurrent interrupt handlers," *Communications in Computer and Information Science*, vol. 257, pp. 407–415, 2011.

[7] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H. J. Boehm, "IFRit: interference-free regions for dynamic data-race detection," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*, vol. 47, pp. 467–484, 2012.

[8] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*, vol. 44, pp. 121–133, 2009.

[9] X. Xie and J. Xue, "Acculock: accurate and efficient detection of data races," in *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO '11)*, pp. 201–212, April 2011.

[10] TinyOS Community Forum, TinyOS: An open source operating system for low-power wireless devices, 2008, http://www.tinyos.net.

[11] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An effective method to control interrupt handler for data race detection," in *International Workshop on Automation of Software Test (ACM '10)*, pp. 79–86, May 2010.

[12] Z. Lai, S. C. Cheung, and W. K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for nesC applications," in *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT '08)*, pp. 94–104, November 2008.

[13] J. Regehr, "Random testing of interrupt-driven software," in *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05)*, pp. 290–298, September 2005.

[14] Y. Zhou, X. Chen, M. R. Lyu, and J. Liu, "Sentomist: Unveiling transient sensor network bugs via symptom mining," in *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS '10)*, pp. 784–794, June 2010.

[15] G. Gracioli and S. Fischmeister, "Tracing interrupts in embedded software," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '09)*, vol. 44, pp. 137–146, 2009.

[16] V. Sundaram, P. Eugster, and X. Zhang, "Lightweight tracing for wireless sensor networks debugging," in *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (ACM '09)*, pp. 13–18, December 2009.

[17] O. K. Ha and Y. K. Jun, "Monitoring of programs with nested parallelism using efficient thread labeling," *International Journal Fo U- and E- Service, Science and Technology*, vol. 4, pp. 21–36, 2011.

[18] Y. C. Kim, S. S. Jun, and Y. K. Jun, "A tool for space-efficient on-the-fly race detection," *International Journal of Database Theory and Application*, vol. 4, pp. 25–38, 2011.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997.

[20] A. Jannesari and W. F. Tichy, "On-the-fly race detection in multi-threaded programs," in *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '08)*, July 2007.

[21] D. Gay, M. Welsh, P. Levis, E. Brewer, R. Von Behren, and D. Culler, "The nesC language: a holistic approach to networked embedded systems," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–11, June 2003.

[22] J. Regehr and N. Cooprider, "Interrupt verification via thread verification," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 9, pp. 139–150, 2007.

[23] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN '05)*, pp. 477–482, April 2005.

[24] B. L. Titzer and J. Palsberg, "Nonintrusive precision instrumentation of microcontroller software," in *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (Lctes '05)*, vol. 40, pp. 59–68, 2005.

[25] J. Hil, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, pp. 93–104, 2000.

[26] G. M. Tchamgoue, K. H. Kim, and Y. K. Jun, "Testing and debugging concurrency bugs in event-driven programs," *International Journal of Advanced Science and Technology*, vol. 40, pp. 55–68, 2012.

[27] G. M. Tchamgoue, O. K. Ha, K. H. Kim, and Y. K. Jun, "A framework for on-the-fly race healing in ARINC-653 applications," *International Journal of Hybrid Information Technology*, vol. 4, pp. 1–12, 2011.

[28] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '91)*, vol. 26, pp. 85–96, 1991.

[29] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[30] Atmel, inc. ATmega128 Datasheet. 2011, http://www.atmel.com/Images/doc2467.pdf.