# Random Testing of Interrupt-Driven Software

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

## ABSTRACT

Interrupt-driven embedded software is hard to thoroughly test since it usually contains a very large number of executable paths. Developers can test more of these paths using *random interrupt testing*—firing random interrupt handlers at random times. Unfortunately, naïve application of random testing to interrupt-driven software does not work: some randomly generated interrupt schedules violate system semantics, causing spurious failures. The contribution of this paper is the design, implementation, and experimental evaluation of RID, a restricted interrupt discipline that hardens embedded software with respect to unexpected interrupts, making it possible to perform random interrupt testing and also protecting it from spurious interrupts after deployment. We evaluate RID by implementing it in TinyOS and then using random interrupt testing to find bugs and also to drive applications toward their worst-case stack depths.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging—*Testing tools*; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

## General Terms

Design, reliability

## Keywords

Random testing, interrupt-driven software, embedded systems, sensor networks

## 1. INTRODUCTION

Despite advances in model checking, static analysis, and automatic code generation, testing is still the primary way to create reliable embedded software [3]. *Random testing* [11, 18] provides a way to create a large number of uncorrelated test cases automatically. These can be used to drive a system into interesting states, with the goal of eliciting failure modes that cannot be found using other testing methods or static analysis. This paper explores the application of random testing techniques to *interrupt-driven* software, where processing is initiated when external devices signal the CPU.

Interrupts are problematic because they add fine-grained concurrency to embedded software. The race conditions that typically result are difficult to eliminate before a system is shipped. Since the number of executable paths through a system can grow exponentially with the number of interrupt sources, interrupt-driven software is hard to reason about and it is hard to test adequately.

Interrupts have led to well-known problems in safety-critical embedded software. For example, a number of people received fatal radiation overdoses in the Therac-25 incidents [16, App. A]. One of the problematic bugs was a race condition in which a keyboard interrupt handler participated. The race could be triggered only by a particularly fast typist. This is the kind of bug that we believe can be exposed using random interrupt testing.

To randomly test an interrupt-driven system, an *interrupt schedule*—a sequence of interrupts firing at specified times—is generated. Next, the system is executed with interrupts arriving according to the schedule, and monitored for signs of malfunction. In this paper we explore both pure random testing and also directed random testing where a genetic algorithm is used to evolve desirable schedules using feedback from previous tests.

This project was motivated by our previous work on analytically determining the worst-case stack depth (WCSD) [22] of TinyOS [12] applications. We observed the stack memory usage of TinyOS applications both in simulation and on real processors, in an attempt to narrow or eliminate the gap between the observed lower bound and the analytic upper bound on WCSD. We found that the applications we studied did not closely approach their worst-case stack depths because many clearly feasible paths involving nested interrupt handlers were not being executed during testing. Furthermore, the obvious random testing strategy for exploring these paths—firing interrupts at random times—did not work because random interrupt schedules sometimes contain *aberrant interrupts.* An aberrant interrupt is simply one that fires at a time when the system cannot handle it properly. A random interrupt schedule that contains aberrant interrupts can cause an embedded system to fail in spurious ways. This undermines the main benefit of testing, which is to find genuine software errors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*EMSOFT'05,* September 19–22, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-091-4/05/0009 ...$5.00.

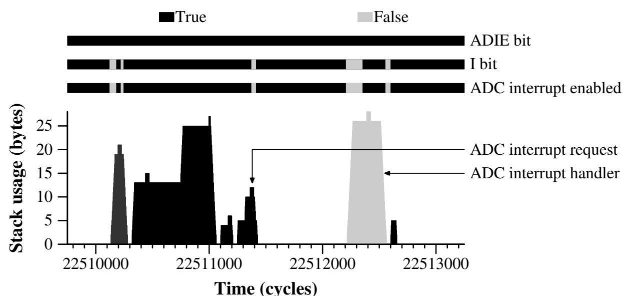


**Figure 1: Visualization of about 0.5 ms of the execution of the original TinyOS Oscilloscope application. The analog to digital converter (ADC) interrupt is enabled most of the time, including before it is requested and during execution of the handler. In this and in subsequent figures, stack memory used by the main computation is shown in black; memory used by interrupt handlers is shown in shades of gray.**

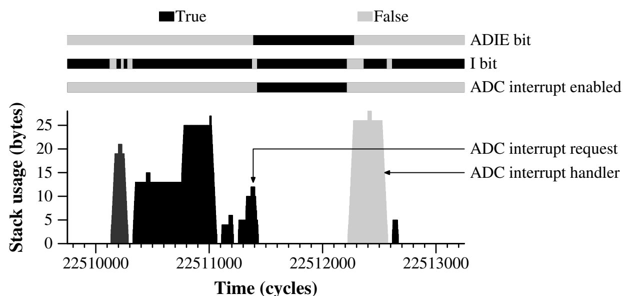


**Figure 2: Visualization of about 0.5 ms of the execution of the TinyOS Oscilloscope application with RID. The ADC interrupt is enabled only after it is requested, and disabled before the first instruction of the handler executes.**

We developed RID, a restricted interrupt discipline for interrupt-driven embedded systems, to solve the problem of aberrant interrupts. RID is a modification to embedded software that uses existing hardware-supported interrupt mask bits to harden a system with respect to aberrant interrupts. This is accomplished through a partially automated program transformation typically requiring two manual changes to the source code per interrupt vector. RID makes it possible to test embedded software using random interrupt schedules. It also has other benefits such as discouraging reentrant interrupt handlers and preventing system failures caused by aberrant interrupts in deployed systems. In a deployed system aberrant interrupts can be caused by software bugs (for example, a stray write to a register that requests an interrupt) or by hardware faults (for example, static discharge, a loose connection, or a damaged wire).

Figures 1 and 2 show the difference between a typical embedded system and one that implements RID. The bars at the top of each figure depict the status of the ADIE bit (the interrupt-enable bit for the ADC, or analog to digital converter), the I bit (the CPU's master interrupt enable bit), and the overall status of the ADC interrupt, which is enabled only when the ADIE and I bits are both set. In the original system, shown in Figure 1, the ADC interrupt is enabled almost all of the time, including when the ADC interrupt itself is running. In the system implementing RID, shown in Figure 2, the ADC interrupt handler is enabled only between the time that the interrupt is requested and the time at which the handler begins to run. If the ADC interrupt is signaled at a time when it is not enabled, the interrupt is not seen by software—it remains pending until the system is ready to handle it.

## 2. INTERRUPTS AND INTERRUPT-DRIVEN SOFTWARE

*Interrupt-driven software.* A system is *interrupt-driven* when a significant amount of its processing is initiated by interrupts. Embedded systems based on powerful processors tend to manage interrupts inside the RTOS, insulating application developers from the problematic aspects of interrupts. On the other hand, highly resource constrained embedded platforms such as pacemakers and sensor network nodes have a relatively thin layer of OS code, and it is common for application code to run inside of interrupt handlers. It is this second kind of system that we focus on: since it is more likely to contain buggy interrupt code, it stands to benefit most from random interrupt testing.

*Handling interrupts.* There is variation in the details of how microprocessors handle interrupts. Here we describe the behavior of Atmel AVR processors as it is typical and these are the processors that we use to evaluate our work in Section 5. Each interrupt has an associated *pending* bit that becomes set when the interrupt's firing condition is met. Typically, a pending bit stays set until the handler runs or until the bit is explicitly cleared. Each interrupt also has an associated *enable* bit. The interrupt is enabled when its enable bit and the processor's global interrupt enable bit are both set. All interrupts that are both pending and enabled compete for execution; of these, the processor selects the lowest numbered and executes it. To execute an interrupt the processor atomically clears the global interrupt enable bit, clears the interrupt's pending bit, pushes the program counter, and jumps to the first instruction of the interrupt's handler.

*Preemption, nesting, and reentrancy.* There are two main ways in which the execution model for interrupts differs from that of threads. First, interrupts cannot block: they run to completion except when preempted. Second, interrupts have an asymmetric preemption relation with the processor's non-interrupt context: interrupts can preempt non-interrupt activity but the reverse is not true. Whether interrupts can preempt each other is determined by the way that an embedded system manipulates its interrupt masks. *Nested interrupts* are those that preempt each other; they are used to permit time-sensitive interrupts to run with low latency. *Reentrant interrupts* are those that directly or indirectly preempt themselves. In other words, a reentrant interrupt may have multiple invocations on the stack at the same time. Although they are occasionally useful, reentrant interrupts are more often an unintended consequence of setting the processor's master interrupt enable bit during execution of an interrupt handler without first clearing the interrupt's own

```
adc_interrupt_handler ()
{
  read_data (adc_buffer_ptr);
  adc_complete = true;
}

...
adc_buffer_ptr =
  xmalloc (sizeof (adc_buffer));
begin_adc_conversion();
...
if (adc_complete) {
  process_data (adc_buffer_ptr);
  free (adc_buffer_ptr);
}
...
```

**Figure 3: Embedded code that is correct, but will malfunction if an aberrant analog to digital converter (ADC) interrupt arrives. `adc_buffer_ptr` is a valid pointer only when the system is expecting an ADC interrupt.**

enable bit. Unintentionally reentrant interrupt handlers are highly unlikely to be correct.

*Requested vs. spontaneous interrupts.* Interrupts can be categorized as *requested* or *spontaneous*. Spontaneous interrupts are signaled by external devices such as network interfaces and can arrive at any time that the corresponding device is enabled. Requested interrupts, on the other hand, are those that arrive in response to a specific action taken by the processor, such as setting a timer or initiating an analog to digital conversion (ADC). For example, typical use of an ADC is as follows. First, the system sets up the ADC device using configuration registers. Subsequently, the program requests a conversion when data is needed. Because ADC devices are slow, an ADC completion interrupt signals the system when the conversion has finished.

The AVR ATmega128 [2] supports 34 interrupt sources. Of these, 18 are requested, 8 are spontaneous, and 8 have no predefined semantics—they are external interrupt lines that could be either spontaneous or requested, depending on the devices they are connected to. Other microcontrollers support a similar mix.

*Aberrant interrupts.* Interrupts are not allowed to arrive at arbitrary times. For example, a requested interrupt that arrives at a time when it has not been requested is considered *aberrant*. Many embedded systems are not robust with respect to the arrival of aberrant interrupts; they may crash if one arrives. Thus, it is undesirable for random interrupt schedules to contain aberrant interrupts. Consider the example code in Figure 3, which has the invariant that `adc_buffer_ptr` is a valid pointer only when an ADC interrupt has been requested. If an ADC interrupt arrives at a time when the system is not expecting such an interrupt, the fault lies with the interrupt schedule, not with the system software. Developers would probably be irritated if random interrupt testing reported this failure.

Spontaneous interrupts may also be aberrant, if they overload a system by violating the minimum interarrival time for a given interrupt source. We have addressed the problem of

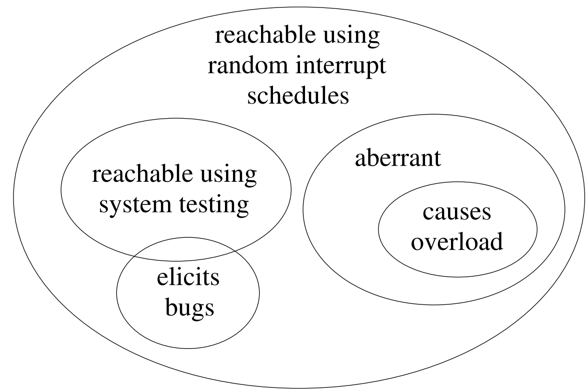


**Figure 4: Regions in the space of interrupt schedules**

preventing interrupt overload in previous work [21], and we do not address it further here.

*The space of interrupt schedules.* Interrupt schedules can be seen as occupying points in a high-dimensional space. Figure 4 shows some regions within this space. Obviously, the goal of random interrupt testing is to find schedules that elicit bugs. Next, consider the area that is reachable through system testing, where an entire embedded system is tested in a realistic environment. A significant advantage of random interrupt testing is that it can be used to explore a larger part of the space of interrupt schedules than can be explored using system testing. For example, consider a wheeled robot that uses an optical encoder to interrupt the CPU at a rate proportional to the angular velocity of the robot's drive shaft, in order to estimate ground speed. System testing is likely to generate interrupts up to the rate corresponding to the robot's maximum speed. However, this neglects many corner cases that can occur in practice: a dirty encoder wheel, a damaged wire, a loose connection, or a speeding robot could easily result in dense bursts of interrupts that are handled incorrectly by the control software. Random testing can help find these corner-case bugs that lie in parts of the space of interrupt schedules not explored during system testing. However, a basic problem with random interrupt testing is that it is difficult to avoid generating interrupt schedules that contain aberrant interrupts. The next section presents a solution to this problem.

## 3. A RESTRICTED INTERRUPT DISCIPLINE

In the previous section we argued that it is unacceptable to naïvely apply random testing to interrupt-driven systems, because many resulting failures are spurious ones caused by aberrant interrupts. One solution would be to avoid generating random interrupt schedules that contain aberrant interrupts. However, based on our previous experience with interrupt-driven systems [22, 23] we believe this to be difficult. First, there is no specification of which interrupts TinyOS applications are expecting in which states. Any such specification would have to be mined from source code by looking at its manipulation of interrupt control registers. It is not clear that mining of precise specifications is even possible because interrupt requests often depend on complex

relationships between multiple control registers. Second, the path merging that occurs during static analysis would lead to uncertainty about system state, forcing the analysis to conservatively assume that a particular interrupt might not be expected at many program points. This would preclude interrupts from being scheduled at these program points, effectively tying the hands of the interrupt schedule generator.

We believe that a simpler and superior solution is to harden embedded software with respect to aberrant interrupts using RID, our restricted interrupt discipline. Under RID, aberrant interrupts are ignored by software and remain pending until they are not aberrant. This results in systems that:

1. Can be tested using random interrupt schedules.
2. Are robust with respect to aberrant interrupts that occur in deployed systems, for example due to software bugs or hardware faults.

RID is runtime code that uses a system's hardware-based interrupt enable bits to prevent each interrupt vector from being serviced at all times at which the system is not ready to handle that interrupt. Implementing RID in embedded software is fairly straightforward. The first step is manual and the second step can be largely automated.

First, developers need to ensure that device initialization code leaves requested interrupts disabled. This requires manual effort per interrupt source. Spontaneous interrupts, on the other hand, can be enabled as soon as the system is ready to handle them—usually this is when the corresponding device subsystem is completely initialized.

Second, requested interrupts should be disabled except between the time at which they are requested and the time at which the handler starts to run. Spontaneous interrupts should be disabled while the handler is running, but can remain enabled at all other program points. The added manipulation of interrupt enable bits adds little overhead. This step can be largely automated, provided that a developer annotates each piece of code that requests an interrupt. We use a custom CIL [19] extension to replace the annotations with code that performs the appropriate manipulation of the interrupt mask. Our CIL code also augments the prologue and epilogue of each interrupt handler to perform the appropriate masking operations. The results of this program transformation can be seen by contrasting the status of the ADC interrupt over time in Figure 1 with its status in Figure 2.

Surprising bugs can occur when an interrupt preempts the execution of a region of code that a developer had not considered. Making interrupt requests explicit has the additional benefit of providing developers with documentation of interrupt disciplines. We believe this makes it easier to create correct interrupt-driven code.

# 4. IMPLEMENTING RID IN TINYOS

This section introduces TinyOS and describes our implementation of RID in that system.

## 4.1 TinyOS and nesC

TinyOS [12] is system-level software for sensor network nodes, primarily the Berkeley motes based on Atmel's AVR family of microcontrollers. TinyOS applications are written in nesC [10], a dialect of C that has specialized constructs

```
async command result_t ADC.samplePort(uint8_t port)
{
  atomic {
    outp((TOSH_adc_portmap[port] & 0x1F), ADMUX);
  }
  sbi(ADCSR, ADEN);

  // this is the actual interrupt request: a write
  // to the ADSC (ADC start conversion) bit
  sbi(ADCSR, ADSC);

  // this is the annotation
  RID_request_SIG_ADC();

  return SUCCESS;
}
```

**Figure 5: TinyOS ADC driver code annotated to make the act of requesting an interrupt explicit**

for dealing with software components and with concurrency. We used TinyOS version 1.1.13.

Concurrency in TinyOS follows a two-level structure. First, interrupt handlers run at high priority, preempting each other freely when this is permitted by the interrupt masks. Second, *tasks* run at low priority. Tasks are scheduled non-preemptively and in FIFO order. Interrupt context is reserved for quick computations such as grabbing data from devices, with the expectation that long computations will be run in tasks.

## 4.2 Modifying TinyOS

The TinyOS applications that we examined use five interrupt handlers: a timer, the analog-to-digital converter (ADC) completion interrupt, the transmit and receive interrupts for a serial port, and the serial peripheral interface (SPI) interrupt. We implemented RID for all five of these interrupt handlers. In each case, only a single file—the bottom-level TinyOS driver for the device—needed to be modified. This is advantageous: low-level drivers are seldom modified and they are shared among many applications. As a result, the vast majority of TinyOS application developers can take advantage of RID without any additional effort.

The ADC driver is representative. First the driver code was modified to leave the ADC completion interrupt disabled at the end of initialization, requiring the addition of one line of code. Second, the `samplePort` function that initiates a conversion is annotated as shown in Figure 5. Finally, the TinyOS application is post-processed using CIL as described in the previous section in order to alter the prologue and epilogue of each interrupt handler.

# 5. EXPERIENCE AND EVALUATION

The section describes our experience in applying random interrupt testing to some TinyOS applications.

## 5.1 Avrora

All of our experiments were run in Avrora [26, 27], a cycle-accurate simulator for networks of Mica2 motes. Since Avrora includes models of the off-chip devices on the motes, it is a suitable platform for work on interrupt scheduling. We modified Avrora to read an interrupt schedule and generate corresponding simulator events. When an interrupt event is

processed by the simulator, it uses an Avrora primitive to force the interrupt to fire. No further changes to Avrora were required. We have passed the interrupt scheduling logic to the Avrora maintainers; it has been included in the sources and should be available in the 1.6 release.

## 5.2 Generating interrupt schedules

It is easy to generate an interrupt schedule, which is just a list of interrupt requests. Each request is represented by an interrupt vector and a firing time. Our current interrupt schedule generator takes as input a list of interrupt vectors to schedule, a duration of the random interrupt schedule, and then for each vector it takes a density.

The choice of interrupt density is important. If the schedule is too sparse, then preemption among interrupts does not happen often enough. If the schedule is too dense, then there are always multiple pending interrupts, a condition that permits the processor's interrupt arbitration logic to deterministically pick the lowest-numbered pending interrupt to run, subverting the randomness of the interrupt schedule. Furthermore, if the processor is always in interrupt mode, then background work cannot make progress, limiting the states that will be visited during testing. Currently we chose interrupt densities empirically. For example, a reasonable initial guess for interrupt density would be one that forces the system to spend 50% of its cycles handling interrupts.

When a fitness function is available to measure the quality of an interrupt schedule, it becomes possible to perform directed random testing rather than pure random testing. We do this using a genetic algorithm as described in Section 5.5.

## 5.3 Test methodology

It is difficult to create precise testing oracles for embedded systems since these systems have a narrow interface to the external world and often have no specification, formal or otherwise. Rather, like Koopman and DeVale [13], we take a weak view of correctness: the system is assumed to be operating correctly if it does not fail. We infer failure in two ways:

1. Using existing interfaces to the simulator. Avrora is capable of detecting access to illegal memory locations and execution of illegal instructions.

2. Using existing interfaces to the embedded system. Many TinyOS kernels report information to a connected PC using the serial port, or they respond to incoming radio packets in a predictable way.

We assume that a bug has been found if either Avrora reports a problem or if the system fails to return to normal operation after the interrupt schedule has finished. Since simulators and interrupt schedules are deterministic, all bugs that we find are trivially reproducible.

## 5.4 Case study 1: A buggy application

This section describes a bug that we found in the TinyOS Oscilloscope application using random interrupt testing. Oscilloscope reads sensor values using the analog-to-digital converter and forwards them to a PC using the mote's serial interface. The buggy code, shown in Figure 6, is found in the ADC completion interrupt handler, which stores incoming ADC data in an array. Each time this interrupt fires, it increments an array index. When the index reaches the size of the array, the interrupt handler posts a TinyOS task to perform further processing on the data. The task also resets the index to zero.

```
async event result_t ADC.dataReady(uint16_t data)
{
  struct OscopeMsg *pack;
  atomic {
    pack =
      (struct OscopeMsg *)msg[currentMsg].data;

    // this line can store out-of-bounds,
    // corrupting memory
    pack->data[packetReadingNumber++] = data;

    readingNumber++;
    if (packetReadingNumber == BUFFER_SIZE) {
      post dataTask();
    }
  }
  ...
}
```

**Figure 6: nesC code that contains a concurrency error. The index variable `packetReadingNumber` is incremented on interrupt and it is also used as an array index. The code assumes that when `BUFFER_SIZE` is reached, the posted task `dataTask` will run, clearing the index variable, before the interrupt is next signaled. However, no interlock exists to ensure that this actually happens. Under certain random interrupt schedules out-of-bounds array accesses lead to memory corruption, starting a chain of events that ends in a crash.**

The stage is now set for a classic synchronization error: the ADC interrupt handler is written under the assumption that the posted task will run before the interrupt handler next fires. However, since interrupts take priority over tasks, there is no guarantee that the task will complete before the interrupt fires again. Under some interrupt schedules, the task does not complete in time, causing an array overrun in the ADC interrupt handler. This has the unfortunate effect of corrupting the TinyOS task queue—the list of tasks that are waiting to be executed—which happens to be located in nearby RAM. Corrupting this queue causes the TinyOS task scheduler to jump to an essentially random address, which happens to be in the middle of a function, with predictably disastrous results.

This bug can be fixed by bounds-checking the ADC buffer or by disabling the ADC interrupt as soon as the buffer becomes full. Although fixing this bug was easy, finding it was difficult as it manifested through multiple levels of memory corruption. In fact, it would have taken a long time to track down the error without two tools. First, we used the Delta debugging algorithm [30] to minimize failure-inducing interrupt schedules. For example, while the original random interrupt schedule that caused the Oscilloscope application to fail contained nearly 300,000 interrupts, the Delta algorithm was able to find a 75-interrupt schedule that caused the failure, greatly simplifying the search for a root cause. Second, Avrora is a highly extensible simulator that makes it straightforward to execute user-defined code in response to simulated events such as interrupts, accesses to particular memory locations, etc.

nesC's built-in race-condition detection, which looks for non-atomic accesses to shared variables, is incapable of detecting errors of the type reported here, since the problem is one of sequencing, not atomicity. On the other hand, type-safe language technology would have trapped the out-of-bounds reference, making the error much easier to find and also preventing the store from being corrupted.

Since the Oscilloscope application has existed for several years, it is reasonable to ask why this bug was not found and fixed earlier. The answer is that Oscilloscope has a *low duty cycle*: it leaves the processor idle much of the time. In practice, the TinyOS task that resets the array index completes before the ADC interrupt next fires. However, it is worth finding even latent bugs: TinyOS is designed as a collection of reusable components, and the next application in which the Oscilloscope component is used may have a higher duty cycle, causing the bug to manifest.

## 5.5 Case study 2: Approximating the worst-case stack depth

In addition to using random interrupt schedules to improve software robustness, we were interested in trying to drive TinyOS applications to their worst-case stack depths. It is important to understand the stack memory usage of embedded systems because it is critically important for the stack not to overflow into memory used for other purposes. Our previous work on static analysis of the stack depths of TinyOS applications [22] produced a tool that attempts to provide as tight a bound on stack depth as possible. Ideally this tool would be sound; in practice we compromised soundness in several ways to provide tighter bounds. Each such compromise resulted in an additional assumption that system developers have to verify through other means, for example by inspection of source code or object code.

While analysis usually overestimates the true maximum stack depth of a system, testing usually underestimates. The goal of both testing and analysis is to make the gap between the two bounds as narrow as possible. In our previous work we found that it was impossible to drive any nontrivial TinyOS application even close to its predicted worst-case stack depth. This left an open question: was the analysis pessimistic, or was the testing optimistic? Our belief was that the static analysis was fairly accurate but testing was failing to execute feasible paths though the code that would lead to large stack depths. Our present work verifies this hunch.

Consider again the TinyOS Oscilloscope application. Running Oscilloscope in the unmodified simulator produces a worst observed stack depth of 28 bytes, as shown in Figure 7. On the other hand, using the version of Avrora enhanced with interrupt scheduling, we were able to drive the Oscilloscope application to use 112 bytes of stack space as shown in Figure 8. This closely approaches the static upper bound of 118 bytes. The remaining 6-byte gap appears to be caused by feasible code paths that are not actually traversed during simulated execution (our stack tool uses an aggressive dataflow analysis to avoid following infeasible paths).

We implemented a genetic algorithm (GA) to direct the search for interrupt schedules that cause a system to consume a lot of stack memory. For some applications like Oscilloscope that have four or fewer interrupt handlers, the GA appeared to be overkill: after only a few generations

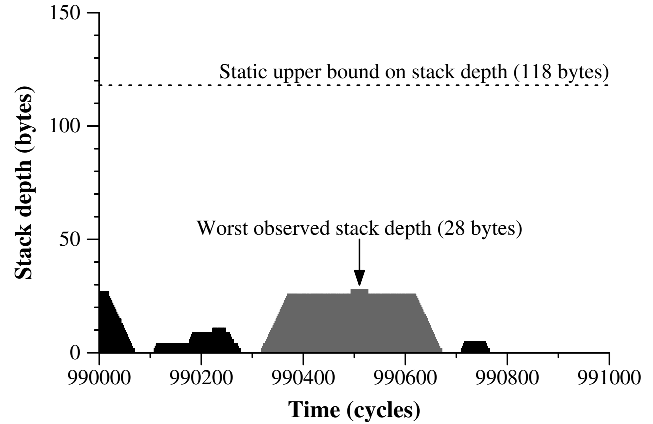


**Figure 7: Worst observed stack depth of the Oscilloscope application without random interrupts**

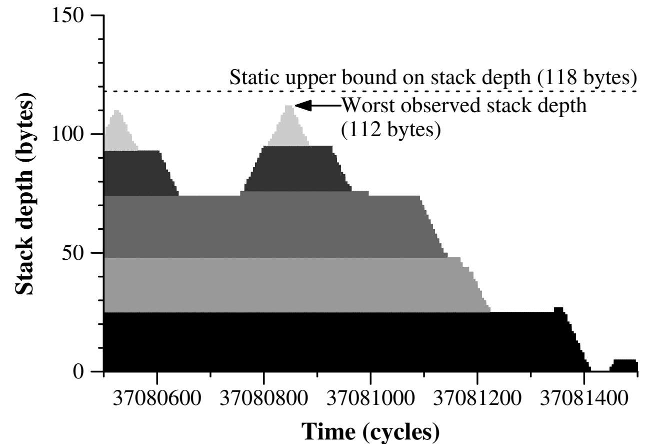


**Figure 8: Worst observed stack depth of the Oscilloscope application under a random interrupt schedule**

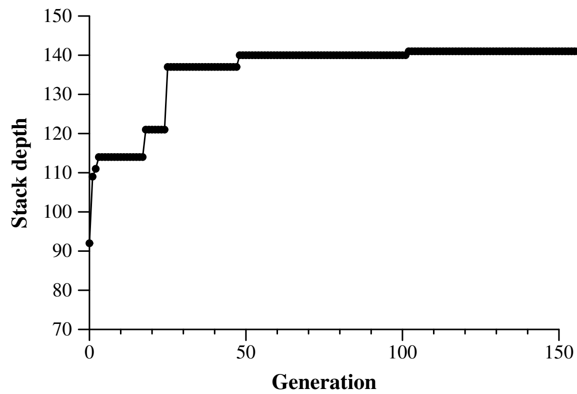


**Figure 9: Results of using a genetic algorithm to search for interrupt schedules that maximize CntToLedsAndRfm's use of stack memory. Although the figure stops at 150 generations, the GA was run to 500 generations without seeing further improvements.**

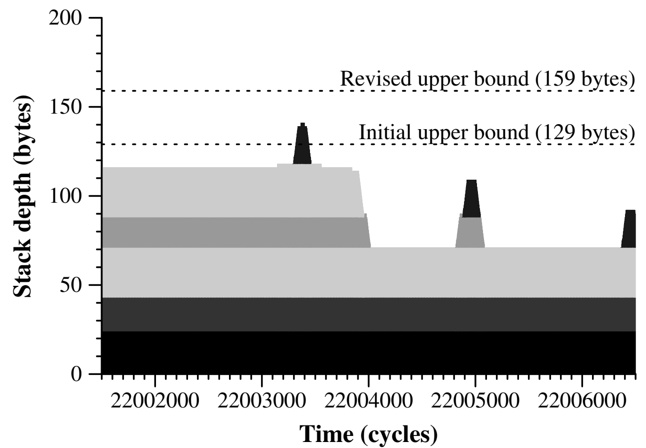


**Figure 10: Initial and revised stack depth bounds for the CntToLedsAndRfm application. Notice that the ADC interrupt handler (the lightest colored) is on the stack twice.**

it reached a maximum that could not be improved upon. However, for other applications such as CntToLedsAndRfm (discussed in the next section), the search space was large enough to justify using the GA. For example, it took more than 100 generations to find a schedule that drove this application to its apparent maximum stack depth, corresponding to about three hours of CPU time on a fast PC. Essentially all of this time was spent in Avrora. Figure 9 illustrates the stack depth of the fittest individual produced by each generation of a run of the GA. The alternative to using a GA—undirected random search of the space of interrupt schedules—was not able to find any interrupt schedules closely approximating the best found by the GA, even in an overnight run.

While the Delta algorithm was helpful in reducing the size of interesting interrupt schedules, it was disappointing that it could not make these schedules even smaller. For example, the minimal failure-inducing schedule for the buggy Oscilloscope application contains 75 interrupts, although it is likely that a human who understands the problem could construct a schedule containing about a dozen interrupts. Similarly, the interrupt schedule that induces the worst-known stack depth for Oscilloscope contains 77 interrupts, which also seems too large. The likely explanation for Delta's failure is that interrupt schedules are fragile since they fire events at fixed times. A line-oriented Delta implementation can only delete interrupts rather than moving them around to fire at different times. This second capability would be required to create truly minimal offending interrupt schedules.

## 5.6 Case study 3: Finding a mistaken assumption

In this section we describe a case where random interrupt testing was able to expose an error in our stack depth analysis. Static analysis of the CntToLedsAndRfm application indicated that its worst-case stack depth is 129 bytes. However, using the genetic algorithm to search for a stack-depth-maximizing interrupt schedule turned up a schedule that caused the application to use 141 bytes. Clearly there is a major problem when the observed depth exceeds the static bound.

On investigation we found that the mistaken assumption behind the erroneous bound was that each interrupt is assumed to be on the stack at most once. While developing our stack tool we found that most embedded systems that we looked at (including almost all TinyOS applications) fail to protect against reentrant interrupts. Rather than doing the "right thing," which would be to fail to return a stack bound for these systems, we made a pragmatic compromise by permitting the stack tool user to specify a maximum reentrancy count for each interrupt, with the default value being one. A benefit of RID is that it prevents reentrant interrupt handlers—but only if interrupt handlers themselves do not request additional interrupts. The SPI interrupt handler in the TinyOS radio stack does exactly this. It requests an ADC interrupt and this admits the possibility that the ADC interrupt handler will be preempted by the SPI handler, which requests another ADC interrupt, which then reenters by preempting the SPI interrupt. Figure 10 illustrates the problem. After understanding the problem it was trivial to notify the stack tool that there may be two outstanding instances of the ADC interrupt handler, which increases the stack bound to a conservative 159 bytes.

Interactions between interrupt handlers are often complicated and can have unforeseen consequences. In this case the reentrant ADC interrupt handler should be considered to be a bug: inspection of the code reveals that it is not designed to support reentrancy. The best fix for this error would probably be to prevent the SPI interrupt handler from preempting the ADC interrupt handler in the first place. Our experience is that developers seldom take global interrupt preemption relations into account when creating embedded software. This is particularly the case for TinyOS, where the component system encourages black-box reuse of existing modules.

## 6. RELATED WORK

The contribution of this paper is to enable random testing of interrupt-driven software. Although there have been many previous applications of random testing to embedded software, we know of no previous identification of, or solution to, the problem of aberrant interrupts in random interrupt schedules. The most closely related work is by Brylow et al. [4], who used a genetic algorithm to find an interrupt schedule that caused an embedded system to approach or reach its worst case stack depth. It appears that the systems that they studied (small control programs written in Z86 assembly) were simple enough that neither interrupt overload or aberrant interrupts were issues. Wegener and Mueller [29] used a genetic algorithm to search for inputs that cause a program to run for as long as possible, in order to approximate the worst-case execution time. Similarly, Alander et al. [1] tested software response times using shared memory and network inputs that were computed using a genetic algorithm. Both of these projects use random input data, rather than forcing interrupts to fire at random times. It would be interesting and probably useful to combine approaches such as these with the work presented in this paper, in order to perform whole-system testing of task response times.

The part of our work that uses the Delta algorithm to minimize interrupt schedules that elicit bugs is closely analogous to work done by Choi and Zeller [6]. Their contribution was a method to find two thread schedules that are as alike as possible while still differing in some important way; i.e., one exposes a bug and the other does not. Our work applies to interrupts instead of threads, and also we minimize the size of single schedule, rather than minimizing the difference between two schedules.

There is a large body of literature on model-based and specification-based testing of embedded systems [9, 15, 20, 24, 25, 28]. These techniques are able to create precise testing oracles, and they are useful for conformance testing. Robustness testing [7, 8, 14], on the other hand, amounts to attempting to cause a program or system to fail, often using random inputs. The primary advantage of robustness testing is that since it uses very simple oracles it can be easily applied to the many embedded systems—such as the hundreds of existing TinyOS applications—for which no specifications exist.

Random testing is a well-established sub-discipline of software testing [11]. Theoretical properties of random testing have been extensively studied, for example by Mankefors et al. [17] and Chen et al. [5]. The main advantages of random testing appear to be its relative ease of implementation and the fact that it is more amenable to mathematical analysis than are other types of testing.

## 7. FUTURE WORK

***Language support for RID.*** Although nesC makes a distinction between *asynchronous* code that is reachable from interrupt mode and *synchronous* code that is not, it has no real semantics for interrupts or interrupt requests. We would like to make the act of requesting an interrupt explicit at the level of nesC. This would permit better static error checking, it would permit the nesC compiler to control preemption relations among interrupt handlers, and it would enable the elimination of dead interrupts, and all transitively called dead code, leading to more effective use of resources on the motes.

***Integrating RID with interrupt overload protection.*** RID is about preventing aberrant interrupts from firing. Our work on interrupt schedulers [21] is about preventing interrupts from firing when their execution would endanger timely execution of other code running on a node. An embedded system using both forms of interrupt protection should be able function correctly even under a truly arbitrary interrupt workload, making it easy to perform testing that should help increase developers' confidence in the robustness of a system.

## 8. CONCLUSION

The growing popularity of sensor networks is exposing a large number of software developers to low-level microcontroller programming. Creating robust software for sensor network nodes and other resource-constrained embedded systems is difficult because there is only a thin layer of OS code, forcing application developers to write code that runs in interrupt mode. One way to help these developers create more robust systems is to support random interrupt testing: firing random interrupts at random times in order to stress-test a system.

This paper introduces RID: a restricted interrupt discipline that hardens embedded software with respect to aberrant interrupts. RID adds no significant overhead because embedded platforms provide hardware support for disabling individual interrupt sources. The implementation of RID can be substantially automated. Also, since RID requires modifying only low-level device drivers, the vast majority of application developers can take advantage of it without taking any special action.

RID permits embedded software to be tested under random interrupt loads without encountering false positive errors due to the arrival of aberrant interrupts. RID also makes systems robust with respect to aberrant interrupts that can occur in deployed systems due to software bugs or hardware faults. It discourages reentrant interrupt handlers, which can lead to stack overflow and are difficult to implement correctly. Basically, RID is simply a good defensive programming practice.

We have applied RID to several TinyOS kernels and then tested them using random interrupt schedules, finding real bugs and driving applications to near their worst-case stack depths. RID is what made this random testing approach feasible.

## 9. ACKNOWLEDGMENTS

The author would like to thank: Nathan Cooprider for writing CIL code supporting this work; Nathan Cooprider, David Coppit, Usit Duongsaa, Eric Eide, David Gay, and Alastair Reid for their helpful comments and advice; and Ben Titzer for his help with Avrora. This material is based upon work supported by the National Science Foundation under Grant Nos. 0209185 and 0448047.

## 10. REFERENCES

[1] Jarmo T. Alander, Timo Mantere, and Ghodrat Moghadampour. Testing software response times using a genetic algorithm. In *Proc. of the 3rd Nordic*

*Workshop on Genetic Algoritms and their Applications (3NWGA)*, pages 293–298, 1997.

[2] Atmel, Inc. ATmega128 datasheet, 2002. http://www.atmel.com/atmel/acrobat/doc2467.pdf.

[3] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison-Wesley, 2002.

[4] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pages 47–56, Toronto, Canada, May 2001.

[5] Tsong Yueh Chen, Fei-Ching Kuo, and Robert G. Merkel. On the statistical properties of the F-measure. In *Proc. of the 4th International Conference on Quality Software (QSIC)*, pages 146–153, Braunschweig, Germany, September 2004.

[6] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 210–220, Rome, Italy, July 2002.

[7] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *Proc. of the ACM SIGPLAN 2002 Haskell Workshop*, Pittsburgh, PA, October 2002.

[8] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice and Experience*, 34(11):1025–1050, 2004.

[9] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, and Nicolas Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *Proc. of the 1999 Intl. Conf. on Software Engineering (ICSE)*, pages 267–276, Los Angeles, CA, 1999.

[10] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[11] Richard Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, second edition, 2001.

[12] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, November 2000.

[13] Philip Koopman and John DeVale. Comparing the robustness of POSIX operating systems. In *Proc. of the 29th Fault Tolerant Computing Symp.*, Madison, WI, June 1999.

[14] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proc. of the Fault Tolerant Computing Symp.*, Munich, Germany, June 1998.

[15] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using UPPAAL. In *Proc. of the 4th Intl. Workshop on Formal Approaches to Testing of Software*, Linz, Austria, September 2004.

[16] Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[17] S. Mankefors, R. Torkar, and A. Boklund. New quality estimations in random testing. In *Proc. of the 14th IEEE Intl. Symp. on Software Reliability Engineering (ISSRE)*, Denver, CO, November 2003.

[18] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[19] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 213–228, Grenoble, France, April 2002.

[20] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model based testing for real. *Software Tools for Technology Transfer*, 5(2–3):140–157, March 2004.

[21] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.

[22] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.

[23] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. of the 24th IEEE Real-Time Systems Symp. (RTSS)*, Cancun, Mexico, December 2003.

[24] Antoine Rollet. Testing robustness of real-time embedded systems. In *Proc. of the Workshop on Testing Real-Time and Embedded Systems (WTRTES)*, Pisa, Italy, September 2003.

[25] Li Tan, Jesung Kim, and Insup Lee. Testing and monitoring model-based generated program. In *Proc. of the Runtime Verification Workshop*, Boulder, Colorado, July 2003.

[26] Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, Los Angeles, CA, April 2005.

[27] Ben L. Titzer and Jens Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Chicago, IL, June 2005.

[28] Aki Watanabe and Ken Sakamura. A specification-based adaptive test case generation strategy for open operating system standards. In *Proc. of the 18th Intl. Conf. on Software Engineering (ICSE)*, pages 81–89, Berlin, Germany, March 1996.

[29] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

[30] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.