Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 139–150
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Interrupt Verification via Thread Verification [1]

John Regehr[2] Nathan Cooprider[3]

School of Computing
University of Utah

Abstract

Most of the research effort towards verification of concurrent software has focused on multithreaded code. On the other hand, concurrency in low-end embedded systems is predominantly based on interrupts. Low-end embedded systems are ubiquitous in safety-critical applications such as those supporting transportation and medical automation; their verification is important. Although interrupts are superficially similar to threads, there are subtle semantic differences between the two abstractions. This paper compares and contrasts threads and interrupts from the point of view of verifying the absence of race conditions. We identify a small set of extensions that permit thread verification tools to also verify interrupt-driven software, and we present examples of source-to-source transformations that turn interrupt-driven code into semantically equivalent thread-based code that can be checked by a thread verifier. Finally, we demonstrate a proof-of-concept program transformation tool that converts interrupt-driven sensor network applications into multithreaded code, and we use an existing tool to find race conditions in these applications.

Keywords: race conditions, interrupts, threads, embedded software

1 Introduction

For programs running on general-purpose operating systems on PC-class hardware, threads are the most important abstraction supporting concurrent programming. On the other hand, interrupts are the dominant method for expressing concurrency in low-end embedded systems, such as those based on microcontroller units (MCUs). All major MCU architectures support interrupts, and a large number of these chips are deployed in embedded systems: according to a Gartner report, 3.5 billion 8-bit MCUs and a billion 16-bit MCUs were shipped in 2003. The correctness of interrupt-driven software is important: a substantial number of these 4.5 billion MCUs were deployed in safety-critical applications such as vehicle control and medical automation. For example, some of the extremely serious Therac-25 bugs [17, App. A] were directly caused by race conditions involving interrupt handlers.

[1] This work is supported by National Science Foundation CAREER Award CNS-0448047.

[2] Email: regehr@cs.utah.edu

[3] Email: coop@cs.utah.edu

doi:10.1016/j.entcs.2007.04.002

Most work on verifying concurrent software has focused on thread-based and process-based concurrency; interrupts have received relatively little attention. The thesis of this paper is that:

> Verifying the absence of race conditions in interrupt-driven systems is important, but the technology for this is primitive. We must understand interrupts and their semantics in order to understand where thread verifiers can, and cannot, be applied to interrupt-driven systems. In particular, we want to identify a minimal set of extensions to verifiers for thread-based programs that permit them to also check interrupt-based programs. A secondary goal is to exploit the semantics of interrupts to make checking faster and more precise.

Of course, threads come in many flavors. In this paper we assume the common case of POSIX-style thread semantics [5]: preemptively scheduled blocking threads, scheduled either in the kernel or at user level. However, the source code examples in this paper use a simplified syntax rather than using the verbose POSIX calls.

2 Interrupts and Threads

This section outlines significant ways in which threads differ from interrupts. See [21] for a more detailed treatment of interrupts and their use.

2.1 Blocking

A key part of the programming model for threads is their ability to *block*: for execution to be transparently suspended by the operating system until some condition is met. Blocking is important for several reasons. First, it enables timeslicing. Second, it adds an important degree of asynchrony to the programming model: a developer need not check for the case where a resource to be used is busy—the OS transparently blocks the requesting thread until the resource becomes available.

Interrupts cannot block: they run to completion unless preempted by other interrupts. The inability to block is very inconvenient and makes it impossible for interrupts to use advanced operating system services. On the other hand, non-blocking execution has a few compelling advantages. First, all interrupts, in addition to the non-interrupt execution context, can share a single call stack. Threads, of course, require their own stacks, making them an unsuitable abstraction for low-end MCUs that typically have at most a few KB of RAM. Second, since interrupts never block, their internal states are invisible to non-interrupt code. In other words, interrupts execute atomically with respect to code running in the non-interrupt context. Third, non-blocking execution means that interrupts can have highly predictable timing. Finally, interrupt handlers are not subject to most forms of deadlock.

2.2 Preemption and scheduling

Threads typically have symmetrical preemption relations: for any given pair of threads, either one can preempt the other. In general, this is true even for threads that are scheduled using fixed priorities, since even a high-priority thread can become blocked voluntarily, due to making a blocking kernel or library call, or involuntarily, due to a page fault. In contrast, asymmetrical preemption relations are the norm for interrupts. First, all interrupts can preempt non-interrupt code, whereas non-interrupt code can never preempt any interrupt. Second, interrupts are often scheduled using fixed priorities, resulting in asymmetrical preemption relations among interrupt handlers.

Some hardware platforms, such as the programmable interrupt controller on a PC, enforce priority scheduling of interrupts. On other systems, prioritized interrupt scheduling must be implemented in software. For example, the Atmel ATmega128 [2], a popular MCU that is the basis for Mica2 sensor network nodes [7], performs priority scheduling only among interrupts that are concurrently pending. Once an interrupt begins to execute on an ATmega128, it can be preempted by an interrupt of any priority, if interrupts are enabled. To implement priority-based preemptive scheduling on this platform, software must manipulate the individual enable bits associated with various interrupts.

2.3 Concurrency control

A thread lock uses blocking to prevent a thread from passing a given program point until the lock resource becomes available. Since interrupts cannot be blocked once they begin to execute, concurrency control consists of preventing an interrupt from starting to execute in the first place. This is accomplished either by disabling all interrupts, or by selectively disabling only interrupts that might interfere with a particular computation. The former is cheaper while the latter avoids incidentally delaying unrelated code.

Since preemption in interrupt-based systems is asymmetrical, so are locking strategies. In other words, while the non-interrupt context must disable interrupts in order to execute atomically with respect to interrupts, code running in interrupt mode does not need to take any special action to run atomically with respect to non-interrupt code.

2.4 Reentrance

An interrupt is *reentrant* if there may be multiple concurrent invocations of the same handler. Reentrant interrupts come in two varieties: accidental and deliberate. Accidental reentrance occurs when a developer misunderstands the consequences of enabling interrupts inside an interrupt handler. We mention this type of reentrance because it usually leads to buggy systems, and because in our experience it is common in real embedded systems.

Deliberate reentrance is a sophisticated technique that can be used to reduce interrupt latency. It is useful when code close to the beginning of a long-running interrupt handler is subject to time constraints. By permitting subsequent invocations of the handler to fire before previous ones have finished executing, the average latency of the early part of the handler can be decreased. For example, the timer interrupt handler in the AvrX system [4] is reentrant in order to avoid missing ticks.

Reentrance has two unavoidable costs. First, later parts of the handler will have their average latency increased, rather than decreased, since total latency in a work-conserving system is a zero-sum game. Second, in practice it is difficult to create correct reentrant interrupt code, even for expert developers.

2.5 Invocation style

Some interrupts are *spontaneous*: they may fire at any time. For example, interrupts generated by a network interface are spontaneous. Other interrupts are *requested*: they only occur in response to an action taken by the running program. For example, timer interrupts are requested, as are analog to digital converter (ADC) completion interrupts. The causal relation between an interrupt request and a subsequent interrupt can only be seen by understanding the semantics of the underlying hardware.

2.6 Deferred calls

Interrupts are often subject to time constraints: they need to execute within a bounded time after they become pending. The best way to ensure that these constraints are met is to make interrupt handlers short. A common idiom is for an interrupt to perform minimal computation associated with the interrupt request, such as acknowledging a hardware condition and possibly moving data associated with the interrupt into a memory buffer. Then, the interrupt handler initiates a longer-running computation in a deferred context, such as a bottom-half handler or thread.

3 Verifying Interrupts

This section shows how to use source-to-source transformation to transform an interrupt-driven program into a semantically equivalent thread-based program so that a thread verification tool can be used to find bugs in the original system.

Consider a C program that contains two interrupt handler functions: `irq5()` and `irq7()`. We further assume that interrupts are scheduled using priorities: interrupt 5 can preempt interrupt 7, but not the other way around. A non-portable language extension "`INTERRUPT`" is used to tell the compiler to generate interrupt prologue and epilogue code for the interrupt functions instead of using the standard calling convention. Any functions called from an interrupt handler can use the standard calling convention.

The skeleton of this system's code is:

```
INTERRUPT void irq5 (void) {
  ...code...
}

INTERRUPT void irq7 (void) {
    ...code...
  }

  int main (void) {
    ...code...
  }
```

Of course this code cannot be checked as-is by a verification tool that understands threads; it needs to be rewritten by a source-to-source translation tool. To treat an interrupt as a thread, the verification tool must consider the interrupt as being invoked in an infinite loop:

```
  void irq5_thread (void) {
    while (1) irq5();
  }
```

Interrupt 7 is analogous. In addition, the interrupt threads must be spawned at boot time:

```
  int main (void) {
    ...initialization code: runs with interrupts disabled...
    create_thread (irq5_thread);
    create_thread (irq7_thread);
    ...main loop code: runs with interrupts enabled...
  }
```

The program point at which the threads emulating interrupt handlers are spawned should correspond to the point where interrupts are first enabled. A typical embedded system runs system initialization code with interrupts disabled, and then enables interrupts just before running the main program loop.

3.1 Blocking, preemption, and scheduling

In the expected case, two threads may each preempt each other at any instruction boundary where preemption is not suppressed by locks. Interrupts, on the other hand, are not typically subject to dynamic priority adjustments and must never block on a page fault or for any other reason. Consequently, the highest-priority runnable interrupt always executes: priorities are strictly enforced. Support for "strict priorities" should be added to a model checker in order to support analysis of interrupt-based code. The first benefit is that this will eliminate false positives: race conditions that cannot actually occur. The second benefit is that the size of the state space can be reduced by considering fewer preemption points. This insight was exploited by Hatcliff et al. [11] in the context of fixed-priority threading, but it is not clear how they dealt with the problem of priority inversion through page faults—the issue is not addressed in their paper.

The threads that are used to model interrupts must be spawned with priorities that the verification tool interprets as strict:

```
  int main (void) {
    ...initialization code: runs with interrupts disabled...
    create_thread (irq5_thread, HIGHEST_STRICT_PRIORITY);
    create_thread (irq7_thread, SECOND_HIGHEST_STRICT_PRIORITY);
    ...main loop code: runs with interrupts enabled...
  }
```

It is important that the priority range assigned to interrupt handlers does not overlap with the priority range that is available to threads.

3.2 Concurrency control

To emulate interrupt-style concurrency control, thread locks must be created to model the processor's interrupt enable bits. Functions for executing atomically in the original interrupt-based program look like these:

```
  void begin_critical_section (void)
  {
    // non-portable code for disabling interrupts
  }

  void end_critical_section (void)
  {
    // non-portable code for enabling interrupts
  }
```

In practice these functions are written in such a way that they operate properly when used recursively. Recursive locks are those that can be safely acquired multiple times. We ignore the issue of recursive locks here; their semantics are the same in threaded and interrupt-based code.

To create versions of these functions that a thread-checking tool can make use of, we translate them as follows:

```
  void begin_critical_section (void)
  {
    acquire_mutex (irq5_lock);
    acquire_mutex (irq7_lock);
  }

  void end_critical_section (void)
  {
    release_mutex (irq5_lock);
    release_mutex (irq7_lock);
  }
```

To complete the transformation supporting concurrency control, we need to turn asymmetrical interrupt-style synchronization into symmetrical thread-style synchronization by requiring that each "interrupt thread" always runs with the appropriate lock held:

```
void irq5_thread (void) {
  while (1) {
    acquire_mutex (irq5_lock);
    irq5();
    release_mutex (irq5_lock);
  }
}

void irq7_thread (void) {
  while (1) {
    acquire_mutex (irq7_lock);
    irq7();
    release_mutex (irq7_lock);
  }
}
```

It is necessary to have a lock for each interrupt handler, rather than a single global lock, to model the situation where interrupt handlers can preempt each other.

3.3 Reentrance

A reentrant interrupt is one that can have multiple invocations on the stack at the same time. Since the number of concurrent invocations cannot usually be bounded, only verifiers that can model an unbounded number of threads, such as Henzinger et al.'s CIRC extensions to Blast [12], are suitable for reasoning about reentrant interrupts.

3.4 Invocation style

The code presented so far models spontaneous interrupts that may fire at any moment; it can also be used to model requested interrupts but this loses information about the causal relation between the request and the subsequent interrupt. We know of no tool for analyzing interrupts that exploits causality between requests and interrupts, but we believe that this would be useful. To model interrupt requests, explicit interrupt request calls must be inserted either manually or automatically. An interrupt request can be exposed to a thread checking tool using a condition variable:

```
void request_irq5 (void) {
  cond_signal (irq5_request);
}
```

Then, the associated interrupt is only permitted to fire following a request:

```
void irq5_thread (void) {
  while (1) {
    cond_wait (irq5_request);
    acquire_mutex (irq5_lock);
```

```
      irq5();
      release_mutex (irq5_lock);
    }
  }
```

3.5 Deferred calls

As interrupt requests cause interrupts to fire, interrupts cause deferred code to run. Deferred thread-mode code requires no special support since the interrupt will signal a waiting thread using standard thread primitives. Bottom-half handlers in the Linux kernel, or DPCs (deferred procedure calls) in the Windows kernel [23], must be modeled as run-to-completion code running at a priority strictly higher than any real thread and strictly lower than any interrupt.

3.6 Summary

Thread verification tools can be used to check interrupt-driven software under three conditions:

(i) A source-to-source transformation tool is used to convert interrupt code into semantically equivalent thread code.

(ii) The thread checking tool supports a separate class of strict priorities that (1) are higher-priority than any real thread and (2) model the fact that interrupts run atomically with respect to lower-priority interrupts and non-interrupt code.

(iii) If the interrupt-driven code contains any reentrant interrupts, the thread-checking tool must be able to model multiple outstanding invocations of a single interrupt handler.

4 Preliminary Results

This section reports on the results from using a proof-of-concept implementation of the techniques in this paper.

4.1 Method

We wrote a plugin for CIL [19] that converts a TinyOS application [13] into a POSIX threads program that we then check for race conditions using Locksmith [20]. The nesC compiler [10] for TinyOS applications includes a race condition detector, but it is unsound due to not following pointers. The transformation closely follows the one outlined in Section 3. The transformation tool was not difficult to implement, taking one of the authors about five hours.

4.2 Results

Table 1 shows the results of this experiment, using a collection of TinyOS 1.0 applications targeting the Mica2 sensor network platform. The “LOC” column indicates

Application	LOC	Interrupts	Locksmith races	cXprop races
Blink	1920	1	1	0
CntToLedsAndRfm	8043	5	37	11
ECC	8851	5	39	11
GenericBase	6941	5	30	1
Oscilloscope	5587	4	16	20
SenseTask	3490	2	2	0
Surge	11022	5	46	13

Fig. 1. Race checking results for TinyOS applications

the number of lines of C code in the application after processing by the nesC compiler and the "Interrupts" column indicates the number of interrupt handlers, out of a maximum of 34 supported by the Atmel ATmega128 processor upon which the Mica2 node is based. The fourth column in Table 1 shows the number of racing variables detected by Locksmith, and the fifth column shows the number of races detected by cXprop [6], a tool that we developed that soundly analyzes TinyOS applications for race conditions.

4.3 Analysis

It is difficult to directly compare the number of races found by Locksmith and cXprop, as the tools operate very differently. Here we examine the major differences. First, cXprop makes the assumption that critical sections are lexically nested—this is a guarantee provided by the nesC compiler that implies, for example, that a TinyOS application cannot enter a function with the interrupt lock held, and then exit it with the lock released. This program property greatly simplifies concurrency analysis. When cXprop is used to analyze embedded programs other than TinyOS applications, the assumption of lexically nested critical sections must be manually verified. Second, Locksmith does not yet support recursive locks: those that may be safely acquired multiple times. TinyOS code makes significant use of nested locks, causing Locksmith to report race conditions that do not exist. This problem reflects a genuine impedance mismatch between TinyOS and Locksmith, that cannot be easily worked around using source-to-source transformation. Third, cXprop runs a fairly aggressive dataflow analysis in an attempt to find dead code and data, in order to reduce the number of race conditions reported.

Finally, note that many detected races are not errors. Other possibilities include:

- There is no race condition: a program property that is invisible to the analyzer ensures that the variable is not actually accessed concurrently.
- There is no race condition: a user-defined synchronization idiom ensures that data invariants are met without explicit locking.

- The racing variable holds non-critical information such as a performance counter, and developers do not care if it gets corrupted occasionally.

5 Future Work

5.1 Verifying the translation

Concurrency—whether in the form of threads or interrupts—is difficult to reason about. We would like to develop a formal argument that the translation presented in Section 3 preserves the semantics of an embedded system. An significant obstacle is the lack of a formal semantics for any interrupt-based platform that we are aware of. Making matters worse, there are significant variations in interrupt semantics across compilers and hardware platforms; see [21, §2]. On the other hand, the necessity of a semantics for multithreaded C and C++ is well-understood and efforts in that direction are underway [1,24].

5.2 Mixed concurrency models

Many systems support not only interrupts and threads, but also deferred calls, such as DPCs in Windows and tasklets in Linux, with their own semantics. Typically, various kinds of locks are available, each protecting against some subset of these concurrency abstractions. In large code bases such as the Windows or Linux kernels, the diversity of concurrency and synchronization abstractions becomes a barrier to creating correct code. A principled way to reason about heterogeneous concurrency models is needed. Our previous work includes TSL [22], an initial step in this direction.

5.3 Other interrupt bugs

Interrupts lend themselves to many kinds of bugs other than race conditions. These include stack overflows, livelock, blocking in interrupt context, erroneous use of non-reentrant functions, and erroneous reentrant interrupt handlers. Unfortunately, there is no obvious way to check for most of these types of bugs via translation to threaded code and use of an existing thread checking tool.

6 Related Work

This section presents a brief survey of the literature on the semantics of interrupts, and on verification of interrupt-driven programs.

6.1 Interrupt semantics

Most attempts to fit interrupts into a system's semantics focus on making interrupts look like threads. For example, Hills [14] provides perhaps the clearest existing description of the problems associated with interrupts, and proposes as a solution a model where the interrupt handler is simply an atomic stub that awakens some threads. These threads perform the processing that would normally run in interrupt context. Similarly, Leyva-del-Foyo and Mejia-Alvarez's work [8], the Nemesis OS [16], TimeSys Linux [25], and Solaris [15] all take the interrupts-as-threads approach.

6.2 Race checking for interrupt-driven systems

nesC [10], the programming language for TinyOS applications, checks for race conditions in interrupt-driven applications by looking for variables that are accessed by interrupt handlers and that are not protected by atomic execution. Henzinger et al. [12] refine nesC's race checking using an analysis that can show that some non-atomic variable accesses are safe. Mercer and Jones [18] exploit GDB's state saving and restoring capabilities to model check interrupt-driven embedded code. The SLAM [3] model checker and the RacerX [9] race condition detector are used to find bugs in kernel code, including interrupt handlers.

7 Conclusion

The contribution of this paper is a generic enabling technique for verification of interrupt-driven embedded software by exploiting existing verification tools for multithreaded code. This is useful because the vast majority of the research effort expended on checking concurrent code has focused on thread-based applications. Our technique consists of leaving an application largely unchanged while turning statements with interrupt semantics into statements with equivalent thread semantics. This is interesting—and potentially tricky—due to subtle differences between the two semantics. We have demonstrated the utility of our technique by converting interrupt-driven TinyOS applications into threaded code and then using an existing race condition detector to look for potential bugs.

References

[1] Alexandrescu, A., H. Boehm, K. Henney, D. Lea and B. Pugh, *Memory model for multithreaded C++* (2004), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf.

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

[3] Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani and A. Ustuner, *Thorough static analysis of device drivers*, in: *Proc. of the 1st EuroSys Conf.*, Leuven, Belgium, 2006.

[4] Barello, L., *The AvrX real time kernel* (2004), http://barello.net/avrx.

[5] Butenhof, D., "Programming with POSIX Threads," Addison Wesley, 1997.

[6] Cooprider, N. and J. Regehr, *Pluggable abstract domains for analyzing embedded software*, in: *Proc. of the 2006 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Ottawa, Canada, 2006, pp. 44–53.

[7] *Crossbow Technology, Inc.*, http://xbow.com.

[8] del Foyo, L. E. L. and P. Mejia-Alvarez, *Custom interrupt management for real-time and embedded system kernels*, in: *Proc. of the Workshop on Embedded and Real-Time Systems Implementation (ERTSI)*, Lisbon, Portugal, 2004.

[9] Engler, D. and K. Ashcraft, *RacerX: Effective, static detection of race conditions and deadlocks*, in: *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP)*, Bolton Landing, NY, 2003.

[10] Gay, D., P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler, *The nesC language: A holistic approach to networked embedded systems*, in: *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, CA, 2003, pp. 1–11.

[11] Hatcliff, J., X. Deng, M. B. Dwyer, G. Jung and V. P. Ranganath, *Cadena: an integrated development, analysis, and verification environment for component-based systems*, in: *Proc. of the 25th Intl. Conf. on Software Engineering (ICSE)*, Portland, OR, 2003, pp. 160–173.

[12] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre, *Software verification with Blast*, in: *Proc. of the 10th Intl. Workshop on Model Checking of Software (SPIN)*, Portland, OR, 2003, pp. 235–239.

[13] Hill, J., R. Szewczyk, A. Woo, S. Hollar, D. Culler and K. Pister, *System architecture directions for networked sensors*, in: *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, 2000, pp. 93–104.

[14] Hills, T., *Structured interrupts*, ACM SIGOPS Operating Systems Review **27** (1993), pp. 51–68.

[15] Kleiman, S. and J. Eykholt, *Interrupts as threads*, ACM SIGOPS Operating Systems Review **29** (1995), pp. 21–26.

[16] Leslie, I., D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns and E. Hyden, *The design and implementation of an operating system to support distributed multimedia applications*, IEEE Journal on Selected Areas in Communications **14** (1996), pp. 1280–1297.

[17] Leveson, N., "Safeware: System Safety and Computers," Addison-Wesley, 1995.

[18] Mercer, E. G. and M. D. Jones, *Model checking machine code with the GNU debugger*, in: *Proc. of the SPIN Workshop on Model Checking of Software*, San Francisco, CA, 2005.

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

[20] Pratikakis, P., J. S. Foster and M. Hicks, *Context-sensitive correlation analysis for detecting races*, in: *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation (PLDI)*, 2006.

[21] Regehr, J., *Safe and structured use of interrupts in real-time and embedded software*, in: I. Lee, J. Y.-T. Leung and S. Son, editors, *Handbook of Real-Time and Embedded Systems*, CRC Press, 2007 .

[22] Regehr, J., A. Reid, K. Webb, M. Parker and J. Lepreau, *Evolving real-time systems using hierarchical scheduling and concurrency analysis*, in: *Proc. of the 24th IEEE Real-Time Systems Symp. (RTSS)*, Cancun, Mexico, 2003.

[23] Solomon, D. A. and M. E. Russinovich, "Inside Microsoft Windows 2000," Microsoft Press, 2000, third edition, xxix + 903 pp.

[24] Sutter, H., *Prism: A principle-based sequential memory model for microsoft native code platforms* (2006), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2075.pdf .

[25] TimeSys Corporation, *TimeSys Linux*, http://timesys.com/.