In Proc. of the ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), Dublin, Ireland, June 2009.
Revised version.

Tracing Interrupts in Embedded Software

Giovani Gracioli
Department of Electrical and Computer Engineering
University of Waterloo
and
Laboratory for Software and Hardware Integration
Federal University of Santa Catarina
giovani@lisha.ufsc.br

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo
sfischme@uwaterloo.ca

Abstract

During the system development, developers often must correct wrong behavior in the software—an activity colloquially called program debugging. Debugging is a complex activity, especially in real-time embedded systems because such systems interact with the physical world and make heavy use of interrupts for timing and driving I/O devices.

Debugging interrupts is difficult, because they cause non-linear control flow in programs which is hard to reproduce in software. Record/replay mechanisms have proven their use to debugging embedded systems, because they provide means to recreate control flows offline where they can be debugged.

In this work, we present the data tracing part of the record/replay mechanism that is specifically targeted to record interrupt behavior. To tune our tracing mechanism, we use the observed principle of return address clustering and a formal model for quantitative reasoning about the tracing mechanism. The presented heuristic and mechanisms show surprisingly good results—up to an 800 percent speedup on the selector function and a 300 percent reduction on duplicates for non-optimal selector functions—considering the leanness of the approach.

Categories and Subject Descriptors D.2.5 Software Engineering [*Testing and Debugging*]: Tracing

General Terms Design, Experimentation

Keywords Tracing, Debugging Real-Time Systems, Embedded Software

1. Introduction

Debugging represents a key activity in the development of an embedded system. Debugging comes after testing in that the developer already detected an unspecified behavior—the program has a bug—and now wants to correct it. Since embedded and especially real-time embedded systems usually interact with the physical world, embedded software makes heavy use of interrupts for timing and I/O devices such as sensors and actuators.

Debugging interrupt-driven software is hard, and it is one reason why software testing and debugging take up between 30 to 50 percent of the total development cost in embedded systems (Bouyssounouse and J.Sifakis 2005; Gallaher and Kropp 2002). Traditional debugging techniques such as single stepping are unsuited for this type of system. Even with conditional breakpoints, any I/O heavy system will alter its behavior too drastically as the developer stops the program to step through the code. For example, when the software drives a motor, stopping the program will cause the motor to lose its torque as the software discontinues the duty cycle.

System tracing can help debugging control flow problems in interrupt-driven software. Tracing records information at run time, and the developer can use it to determine the application's control flow offline. Tracing techniques themselves have been around since the early programming stages (Baginski and Seiffert 1974; Barnes and Wear 1974), and they come in different flavors: using special hardware counters (Cargill and Locanthi 1987; Tsai et al. 1990), software-based monitoring (Mellor-Crummey and LeBlanc 1989; Dodd and Ravishankar 1995; Thane 2000; Kim et al. 2004), and hardware tracing (JTA 2001; NEX 2003; Moore and Moya 2003; Omre 2008).

The utility of the trace increases with a replay mechanism (Ronsse and De Bosschere 1999; Ronsse et al. 2003; Sundmark et al. 2003). The replay system allows the developer to step forward and backward in the execution trace of the application. The replay system loads the trace and simulates the program execution using the stored trace generated at run time. The developer can stop the simulation any time and examine the execution context that led to the particular control flow path.

Interrupts pose a particular problem to tracing applications. An interrupt can occur at any time and causes a break in the control flow as the program execution stops at the current instruction and continues at the interrupt handling routine. A good control-flow trace must also include these routines, however, if the application contains loops, then determining the exact control flow as it happened at run time becomes difficult.

Recent work (Sundmark and Thane 2008) suggested taking a snapshot of the execution context when an interrupt occurs. The presented results are encouraging, but the approach lacks accuracy. In this work, we present a better heuristic and contribute the following to the state of the art:

- We formalize the problem and allow quantitative reasoning about the individual aspects involved in the tracing mechanism.
- We present and discuss the principle of return address clustering.
- We propose two heuristics—frequency-based selection and round-robin hashing—and evaluate their its efficiency with our

metrics. Both heuristics incur little overhead and work surprisingly well.

- We investigate the trade-off in allocating bits to the return address and the hash portion of the fingerprint.
- We discuss a number of observations and insights that will help the community to advance tracing mechanisms.

The remainder of the paper is structured as follows: Section 2 provides an overview of the system model. Section 3 describes our finding of the clustering of return addresses. Section 4 shows our approach and algorithms for the tracing mechanism. Section 5 provides measurements on the quality of our approach. Section 6 discusses additional findings during the evaluation. And finally Section 7 closes the paper with our summary and conclusions.

2. System Model

In the system model, we provide an overview of the two core elements of a capture/replay mechanism: tracing and replay. We also discuss possible errors that can occur in such a system.

2.1 Tracing Mechanism

Figure 1 provides an overview of the tracing mechanism. The *execution context* s represents the current state of the system affecting the program's execution including for example all memory data, process registers, I/O status bits. A selector function g uses the execution context and selects data elements such as I/O registers or stack pointers as raw input components. The selector function also includes a reduction step. For example, the selector function can select only parts of the stack as input component. A hash function h calculates a *raw fingerprint* f^* from the given input components. Another element of the fingerprint is the reduced return address ra^* which is obtained by extracting the return address from the execution context and applying a reduction function r. The final fingerprint f of an execution environment comprises the reduced return address ra^* as prefix and the raw fingerprint f^*—expressed formally as $f(s) := \langle r(s.ra), h(g(s))\rangle = \langle r(s.ra), (h \cdot g)(s)\rangle$. The *fingerprint width* is the number of bits required to represent the fingerprint. Note that fingerprints may contain no return-address prefix with a reduction function r that filters out the complete return address.

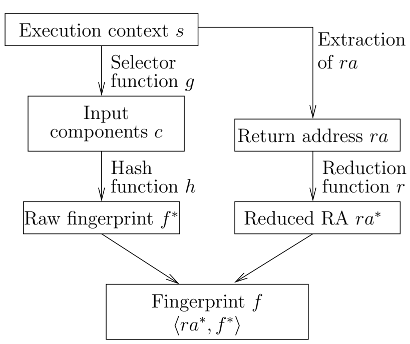

Figure 1. Computing a fingerprint.

Consider a simple example in which the processor state consists of memory and processor registers. A selector function picks registers R0 to R4 and memory addresses 0x00 to 0x3F as input components—for our purpose called $\{x_0, \ldots, x_n\}$. A simple hash function (Sundmark and Thane 2008) is $h_1(c) = (\sum x_i) \bmod 16$. The reduction function r picks the four least significant bits from the return address. The fingerprint width for this example is eight bits with four bits for ra^* and four bits for f^*. Assume that at the time the interrupt occurred, the processor state and memory and registers are filled with zeros. If an interrupt happens at address 0x16, then the resulting fingerprint will be $\langle b0110, b0000\rangle$.

At the time of recording, the tracing system computes a fingerprint each time an interrupt happened and adds it to the *fingerprint database* D.

Two fingerprints f_1 and f_2 will be equal, if their return address portion is equal ($f_1.ra^* = f_2.ra^*$) and their raw fingerprint is equal ($f_1.f^* = f_2.f^*$).

2.2 Replay Mechanism

In the replay phase, a simulator executes the application. To consider interrupts, after each step, the simulator checks whether to continue with the next instruction or with the interrupt handling routine. To perform this check, the simulator computes a fingerprint after each instruction. If the fingerprint database contains the computed fingerprint, then the system will continue with the interrupt handling routine; otherwise, it will continue with the next instruction.

2.3 Caveats

The following problems can occur in such a tracing system with replay: A *false input duplicate* occurs if two different states have equivalent input components after applying the selector function. Formally, this will occur, if $s \neq s'$ but $g(s) = g(s')$. False positives generally occur because of information reduction caused by the selector function and the subsequent hashing. A *false loop positive* will occur, if the replay mechanism wrongly continues at the interrupt handling routine although no interrupt happened at this point in time at this address. For example,the application contains a loop and the replay mechanism wrongly continues at the interrupt handling routine at an iteration count of three instead of seven. Formally, $s \neq s'$ and $s.ra = s'.ra$ but also $(h \cdot g)(s) = (h \cdot g)(s')$. A *false RA positive* will occur, if the replay mechanism wrongly continues with the interrupt handling routine although no interrupt ever happened at this return address. These errors occur as the reduction function r restricts the return address portion of the finger print. Formally, $s \neq s'$ and $s.ra \neq s'.ra$ but $r(s.ra) = r(s'.ra)$ and $(h \cdot g)(s) = (h \cdot g)(s')$

For example, assume the hash function h_1 from the previous example and exactly one interrupt occurs at run time at program address 0x8c. Further assume, that the resulting input components are $c_1 = g(s) = \{200, 40, 7\}$. The corresponding fingerprint is $fp_1 = \langle 0b1100, 0b0111\rangle$ and the fingerprint database is $D = \{199\}$. During the replay, a false loop positive will appear if the input components for the check are, for instance, $c_2 = \{200, 26, 5\}$ at program address 0x8c, because $h_1(c_2) = h_1(c_1) = 7$. A false RA positive will occur, if the input components are also c_1 but the return address is 0x0c, because $r(200) = r(192) = 0b1100$.

These caveats show that it is crucial to select good functions for the tracing method, otherwise the replay mechanism will be inaccurate.

3. Return Address Clustering

During our empirical evaluations, we observed an interesting phenomenon regarding return addresses: interrupts typically cluster around a few return addresses. The number of clusters is relative to the program size and the interrupt frequency, yet still return addresses spread non-uniformly across the address space.

Figure 2 demonstrates this observation with an example that initially tipped us off. Section 5 explains this program more closely with the name FuncsHeap. The x-axis shows the return address at which the interrupt occurred. The y-axis shows the frequency how often we observed that particular return address; note that the y-axis has logarithmic scale. The program runs on an AVR ATmega16 microcontroller and performs a periodic analog to digital conversion and processes this value. The program consists of about 7,000 instructions and we recorded more than 8,500 interrupts by inserting breakpoints and extracting data through the JTAG interface (JTA 2001). In the original program, the interrupts cluster around the return addresses 200 to 350 and 6650 to 6750. Restructuring the program did not substantially change the interrupt distribution. Figure 2 also shows the data for the modified version, and return addresses still cluster around few addresses.

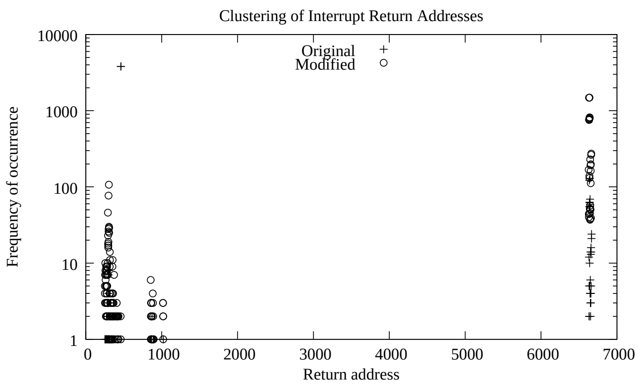

Figure 2. Example of the principle.

Factors contributing to the clustering include:

- Tight loops present in busy waiting routines, iterative calculations (e.g. division) and data processing, and array/matrix operations.
- Blocking operations present in time-costly instructions, non-preemptive peripheral access (e.g., EEPROM writes with several milliseconds), and waiting for user input.
- Concurrency control structures when the developer disables the interrupts to prevent interleaved execution of programs.

All these elements prevent return addresses from being uniformly distributed across the address space, because they change the likelihood of an interrupt occurring at particular addresses. For example, concurrency control structure often prevent interrupts from happening inside critical sections; so interrupts are more likely to be triggered immediately after the critical section completes.

As a concrete example in the Atmel ATmega processors, when an interrupt occurs, the Global Interrupt Enable (GIE) I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a "RETurn from Interrupt' instruction—called RETI—is executed. If an interrupt condition occurs while the corresponding interrupt enable bit is cleared, the Interrupt Flag (IF) will be set and *remembered* until the interrupt is enabled, or the flag is cleared by software. Similarly, if any interrupt conditions occur while the GIE bit is cleared, the corresponding Interrupt Flag(s) will be set and *remembered* until the GIE bit is set, and will be executed by order of priority. When the processor returns from an interrupt, it will always return to the main program and execute another instruction before any pending interrupt is served.

This principle allows us to better understand the system and to create a better tracing mechanism than tracing mechanisms that assume normally distributed return addresses. Pointing forward, the insights of RA clustering lead to our design a round-robin mechanism with markers in the code specifying which has function should be used. It also lead to an approach that failed and we describe our experience in Section 6.

4. Approach and Method Overview

The introduction already stated the motivation and the problem. We now revisit the problem definition with the additional knowledge of the systems model and the observed return address clustering.

The developer wants to trace control flow in applications with interrupts without using special hardware. A suggested approach is to create fingerprints of the system state at run time and then use these fingerprints to determine during the replay whether an interrupt happened at a particular location. While the replay mechanism is straightforward, the tracing mechanism has several design parameters:

- What is a good selector function?
- What is a good hash function?
- What is a good ratio between the return address and raw fingerprint portion?

In the following we present our approach to each of these design parameters and evaluate the quality through experimental data in the followup section.

4.1 Frequency-based Selection

The execution context of the application comprises a large amount of data. Key elements include control and status registers, which are essential to instruction execution, as well as memory-mapped peripherals and I/O controls—see (Tanenbaum 2001) for details on these concepts. These elements for example include the program status word (PSW), process control blocks, thread control blocks, and integrated peripherals. The PSW usually contains the instruction address, processor condition codes and status information including interrupt settings, and the kernel/user mode bit. The process control block comprises the PSW but also further process specific information such as identifiers, state, memory pointers, and accounting information. The thread control blocks—only present in multithreaded processing—contain further thread-specific execution information. Microcontrollers and microcomputers integrate computation units and peripheral devices into a single chip. This increases the execution context and adds elements such as peripheral status (for example the ADC conversion state), data buffers (for example memory-mapped CAN message buffers), and hardware state (e.g., timer values). Growing chip complexity—think system-on-chip or network-on-chip—enriches execution context even further.

The tracing mechanism must only use a subset of the execution context to guarantee short execution time and utility. Reading processor registers, memory, and peripherals' data requires execution time. Tracing should be transparent to the application, therefore it should have minimal overhead in terms of execution time, memory, and program code. Using the whole execution context contradicts this design goal. For example the ATmega16 microcontroller has 32 general-purpose registers, 64 I/O registers, 1KB of SRAM, 512 KB of EEPROM, and three timers. Reading all these data on the 4MHZ variant requires at least $280\mu s$ without the EEPROM and several milliseconds when including the EEPROM.

Using the whole execution context not only takes too long to read but also to process. Besides execution time, not all data of the execution context are useful for tracing, because they change too infrequently. For example, microcontrollers contain a status register that stores the reason of the last chip reset—brown-out, watchdog, or external reset. This register holds the same value in between resets and thus will always have the same value for all calculated fingerprints. Including this or other data with similar behavior provides no utility but increases execution time. The selector function picks a subset from the execution context; this subset becomes the input components for the hash function.

We propose a frequency-based selector function that picks the elements which occur most frequently in the application source code. The insight is that frequently-used elements will frequently change and thus make good input components. Change can occur in two cases: either the program changes the execution environment through instructions that store values or flip output pins, or peripherals induce change by modifying status registers or delivering new data. The selection process consists of the following steps:

1. **Generate assembly code:** In this step, we compile the application sources into assembly code. Most compilers allow the developer to stop before the assembly stage. The compiler then delivers assembly source code instead of machine code.
2. **Extract instructions:** In this step, we collect all lines of code from the assembly that affect the execution environment. Several instructions alter the execution context and they differ between computer architectures. In our experiments, we used the AVR ATmega architecture.
3. **Count suitable components:** In this step, we count how often suitable components appear in the extracted lines of code. This step scales in its complexity depending on the processor architecture and the thoroughness of the approach. For example, some processor architectures such as the AVR address peripherals by their memory location instead of direct text references in the assembly. GCC for AVR produces assembly that directly addresses registers—e.g., "`ldi r22, 0x40`" stores the value `0x40` in the register 22—but it addresses hardware peripherals such as timers by memory location—e.g., "`in r22, 0x3c`" loads the value of the Timer/Counter0 Output Compare Register into Register 22. If such components are suitable components, then the developer will need to enumerate their memory locations, so our algorithm can use them.

   Indirect addressing and high-level components also complicate this step. Data elements that are transparent for the processor but have significance in the software layer such as thread accounting information have not specific name in the assembly code and to use them, the developer must use the memory map files from the compilation process and apply the program analysis to identify which lines of code access specific memory variables.
4. **Order by frequency:** In this step, we order the suitable components by frequency and pick the most frequent ones as our input components. The number of used input components depends on how much execution time overhead the application permits for the tracing mechanism.

We execute these steps separately for the general-purpose and the I/O registers; otherwise, the I/O registers will always lose against the general-purpose registers, because they are less frequently used. The experimental data shown in see Section 5 confirms this decision.

4.2 Round-Robin Hashing

We cannot store the raw input components in the fingerprint, because this would require large fingerprint widths. We therefore use hash functions to turn all input components into a fixed width number. So for example, if the fingerprint width is at most 16 bit and we use 8 bit for the return address part (r^*), we will use a hash function that returns 8 bit numbers for the f^* part.

A good hash function has no funnels and characteristics (Jenkins 1997), meaning that the hash function will work well, even if only a few bits change on the input. This is an important property for our tracing mechanism, because if we trace a system with a high interrupt frequency and many loops, then chances are that only one or two input components changed in between two fingerprints.

Related work showed that the hash function (Sundmark et al. 2003) can be one of the primary weaknesses in the tracing mechanism. Observations during our own experiments lead us to the idea of round-robin hashing.

Interrupts happen at almost arbitrary locations in the code, yet they cluster around certain addresses. Round-robin hashing uses different hash functions in the code. A marker—essentially a variable assignment and an indicator in the execution environment—defines which hash function the tracing mechanism should use. The results in Section 5.2 show this to be more effective than any single hash function. Furthermore, Section 6 shows plots that explain why round-robin hashing works and is superior. Our current algorithm arbitrarily sets these markers based on the cluster information, static and run-time analysis of the program will improve these results.

4.3 RA/H Ratio

A fingerprint consists of a partial return address (ra^*) and a hash value of the input components (f^*). These two stands in an important relationship to each other that affects the count of false positives in the replay mechanism.

First let us consider the hash portion of the fingerprint and let us assume that we chose a good hash function. A good hash function causes as few hash collisions as possible for a given set of inputs. In fact, if we know all inputs, then we will be able to use perfect hashing preventing collisions (Fox et al. 1992).

If the f^* has a small width, then we will observe hash collisions at run time and false loop positives at replay time. A small width of f^* provides only a small space for hash values, so more inputs are mapped to the same hash value. This increases chances of two interrupts having an execution environment that gets mapped to the same value of f^*. At replay time, a small width of f^* increases the rate of false loop positives. At replay time, we stop after each instruction and check whether an interrupt occurred at run time; i.e., we compute a fingerprint from the simulated execution context and check whether this fingerprint exists in the fingerprint database—we call this a hit. If we use a small width for f^* and, for example, one forth of the possible fingerprint values were recorded in the database, then chances are that we will have a hit—and possibly a false loop positive—every fourth time the program reaches the same address, because of the nature of hash functions; therefore, the bigger the width of f^* the better.

If ra^* has a small width, then we will observe fewer hash collisions at run time but more false RA positives at replay time. A small width of ra^* leaves room for a larger width of f^*—this is a good thing as we have shown before; however, the ra^* acts as prefix to prevent false RA positives. If we use just one bit for the ra^* and, for example, one thousandth of the possible fingerprint values were recorded in the database, then chances are that we will observe a false RA positive every 2,000 executed instructions.

The developer must find a trade-off between these two types of false positives, as exemplified in Figure 3. Our experimental evaluations show the significance of this trade-off.

5. Evaluation

We used EPOS—the Embedded Parallel Operating system—for implementing our three test applications. EPOS is a component-based framework for the generations of dedicated runtime support environments. The EPOS system framework allows programmers to develop platform-independent applications for microcontroller-based systems. Analysis tools allow components to be automati-

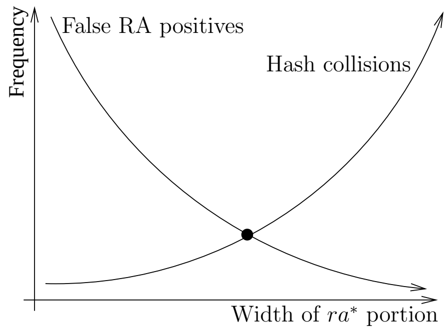

Figure 3. Relationship between false RA positives and hash collisions.

cally selected to fulfill the requirements of these particular applications. One instance of the system aggregates all the necessary support for its dedicated application, and nothing else. Furthermore, the system provides an active, opportunistic power manager, a system resource monitor, a real-time scheduler, and a set of several system components that range from thread management to UART, SPI, communications, and ADC handing.

We implemented three different programs on top of EPOS to evaluate our work: ADCApp, FuncsHeap, and FuncsHeapWithoutLoop. The ADCApp represents a typical embedded application that connects, for example, a sensor to a workstation through the serial line. The program continually reads data from an analog device, communicates the read value via the serial interface to a control station, and also stores results in the EEPROM in a circular buffer. The program first initializes the ADC, UART, and the EEPROM. Then, it spins in an infinite loop reading and writing data. The application uses 13,886 bytes of program memory which is about 85 percent of the available memory.

The FuncsHeap application is specifically designed to test the efficiency of our algorithm with loops and nested function calls. The program uses eight global variables and five functions (main, and func_a to func_d). The main function has an infinite loop. In it, it toggles some LEDs, contains a delay loop—from 0 to 0xffff simulating arbitrary computation—and calls func_a. func_a makes some calculations using global variables and calls func_b. func_b calls func_c twice inside a loop and passes two integers as arguments. func_c dynamically allocates an array of five integers and calls func_d four times inside a loop passing the array as argument. func_d loops from 0 to eight doing some calculations on global variables. The application uses 14,008 bytes of program memory—that is about 86 percent.

The FuncsHeapWithoutLoop application is similar to the FuncsHeap application in its design, however, we removed the delay loop to emulate simple background/foreground systems as often found in control applications. The application uses about 14KB of program memory which is about 86 percent of the available memory.

We used the ATMEL AVR computing platform to run our experiments. We ran the applications on the STK500 with an ATmega16 processor—note that in comparison to related work, we collected the data using real hardware with external inputs. This processor provides a good execution context of which we considered 32 general-purpose registers, 64 I/O registers, 1KB of SRAM, and three timers. Using the JTAG interface and AVR Studio, we put a breakpoint in the OS interrupt handler. When an interrupt occurs, we save the execution context (GP registers, I/O registers and SRAM memory) in files. We automated this process using the QuickMacros tool to speed up the collection process. Saving the context leaves the application and OS untouched, because AVR Studio keeps the system state including timers. Our method would only cause problems, if we had used a real-time clock, which we had not.

5.1 Selector function

We can measure the quality of the selector function by counting the number of false input duplicates. The optimal selector functions satisfies the property $\nexists s \neq s' : g(s) = g(s')$, that is, for all input exists an unique output. Unfortunately, since the selector function always picks a subset of the execution context, the optimal selector function always selects the whole execution context. Another criterion is the number of input components selected by the selector function. A good selector function should pick few input components as possible to produce as little as possible false input components. We include this in our criterion to minimize the execution time of the selector function. A good selector function should have as short an execution time as possible. For example, reading an I/O register requires one clock cycle while reading stack values requires at least two clock cycles—one to receive the stack pointer and more to read data. Minimizing the execution time will minimize the tracing mechanism's effect on the application and additional power consumption.

In the first step, we compute the frequency of use of registers and other potential input components in the source code. In the following we discuss registers—general purpose and I/O—however, the principle also applies to other data elements. We scanned the source code for these instructions:

- Arithmetic and Logic Instructions: ADD, ADC, ADIW, SUB, SUBI, SBC, SBCI, SBIW, AND, ANDI, OR, ORI, EOR, COM, NEG, SBR, CBR, INC, DEC, TST, CLR, SER.
- Data Transfer Instructions: MOV, MOVW, LDI, LD, LDD, LDS, LPM, IN, SBI, CBI.
- Bit and Bit-Test Instructions: LSL, LSR, ROL, ROR, ASR, SWAP.

Figure 4 shows the result of our frequency analysis for the general purpose registers and the I/O registers in the ADCApp. Because of space concerns, the figure only shows registers with at last 1.2 percent frequency in the scanned instructions. Register 24 appears most often in the source followed by registers 25, 30, 31, 28 etc. The microcontroller accesses I/O registers via memory addresses. The most frequently used ones are the status register at location 0x3f, the SP low at location 0x3d followed by SP high 0x3e. Using our proposed algorithm for the selector function, we will consider general purpose and I/O registers our input components in this order. However, does this affect the number of false input duplicates and how effective is it?

Figure 5 shows the effect of including more most-frequently used general purpose registers in the selector function for all three applications. The x-axis shows how many registers are included in the selector function. The y-axis shows the ratio of false input duplicates to the total measure number of interrupts. A value of y=5 means that using the present configuration of the selector function, 5 percent of the recorded interrupts resulted in false input duplicates.

The initial selector function includes the three most frequently used I/O registers—SP low, SP high, and SREG—and the return address. In the figures at value x=0, the selector function picks only these three data elements as input components. At value x=1, the selector function picks the original three data elements and the most-frequently used register. In the case of the ADCApp, the selector function includes at value x=1 the SP, SREG, RA, and R24. At value x=2, this function then also includes R25.

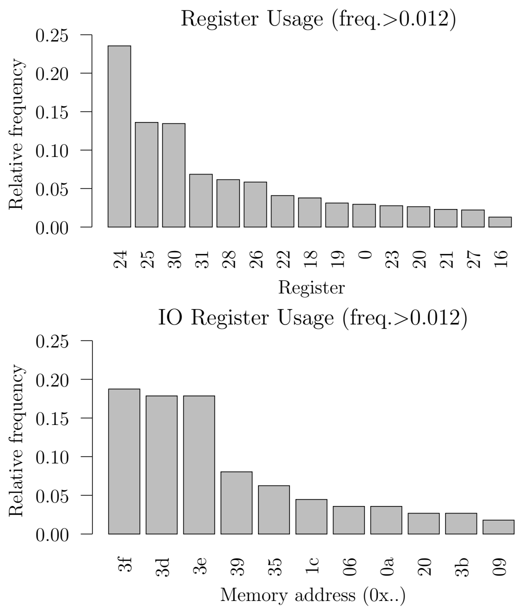

Figure 4. Frequently used registers in the ADCApp.

The figures show a dramatic increase in the quality of the selector function with only a few registers. In ADCApp with only three more registers—namely R24, R25, and R30—we cut the number of false input duplicates essentially to zero. In FuncsHeap we only need one more register to reduce the number of false input duplicates to a few percent.

Yet, including general-purpose registers alone does not make a good selector function. The results for FuncsHeapWithoutLoop in Figure 5 show that register selection can have limited impact in programs. The drop in the number of false input duplicates is still significant—after all it's 50 percent with only using half the register set—yet, it is not as dramatic as in the other two applications.

The solution for this is to include more I/O registers. If we include the I/O registers such as the TIFR, PINB, TCNT1L, and TCNT1H to the selector function in FuncsHeapWithoutLoop, then the number of false input duplicates will decrease to zero.

5.2 Hash function

Our algorithm allows developers to use any hashing function without funnels and characteristics. Yet, we were curious how different hash algorithms perform in our evaluation. We ran the experiments with the following hash functions and compared it to our proposed round robin hashing:

- **Modular Hashing:** This hash function simply applies the modulo operation in the key.
- **Two Hash Function based on Modulo:** This hash function combines two hash functions to produce the final fingerprint. It is a variation from the algorithm presented in (London 1999).
- **Bob Jenkin's Hash Function:** An alternative to modular hashing is the hash function adapted from the byte-by-byte Bob Jenkis' implementation (Jenkins 1997).

  If you need less than 32 bits, use a bitmask. For example, if you need only 16 bits, do `h = (h & hashmask(16))`. In which case, the hash table should have hashsize(16) elements.
- **Multiplicative Hashing:** An alternative of multiplicative hashing originally proposed by *Knuth* (Knuth 1997) was used.

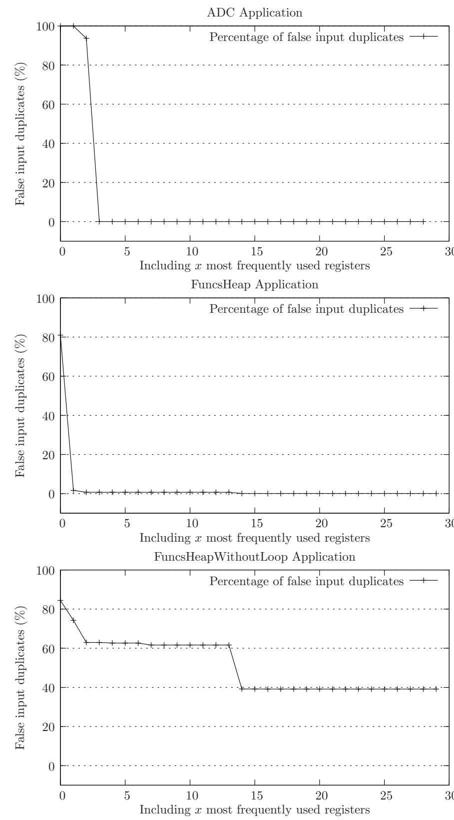

Figure 5. Contribution of most-frequently used general-purpose register to reducing false input duplicates in ADCapp, FuncsHeap, and FuncsHeapWithoutLoop.

We also consider our form of round-robin hashing;

- **Round Robin:** The round-robin combined hash function uses all four hash functions mentioned before in a round-robin fashion. Markers in the code together with markers in the execution context specify which hash function to be used.

We recorded more than 10,000 interrupts for each application and computed the number of fingerprint duplicates—$fp.ra^* = fp'.ra^* \wedge fp.f^* = fp'.f^*$ for two separate interrupts. The hypothesis is that there is no significant difference between different hash functions as long as each hash function has good properties. Figure 6 shows the results of this comparison. The x-axis shows the width of the hashing portion f^*. The y-axis shows the median number of collisions per return address—technically the median of the medians of the hash set with each RA prefix. We compared two scenarios: (a) using an optimal selector function that always creates unique input components and (b) a non-optimal selector function. We can observe the following from this data:

- **Individual hash functions differ little.** In both cases, the optimal selector function and the non-optimal one, single hash function perform about equally well. The two bundles of lines—one for each case—stand out in Figure 6. It therefore virtually makes no difference which of the single hash function is used for a system.
- **Combined hash functions work very well.** The outstanding result in the figure is our form of round-robin hashing. When using a non-optimal selector function, this type of hashing per-

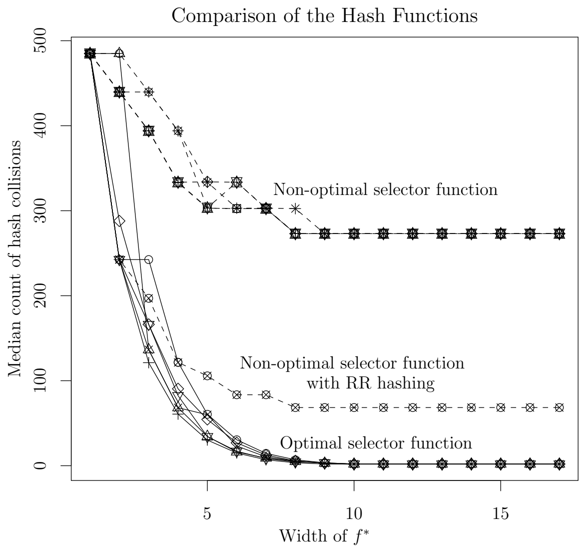

Figure 6. Comparison of effectivness of hashing algorithms.

forms three times better than single hash functions. We describe in Section 6 the reason for this.

- **Combined hash functions can compensate for poor selector functions.** Using an optimal selector function yields excellent results. The number of collisions reaches almost zero with already a low bit width, and it surpasses the non-optimal one by two orders of magnitude. But, round-robin hashing can compensate for non-optimal selector functions. Figure 6 shows that round-robin hashing with the non-optimal selector function comes close to the results with the optimal one—especially with low bit widths.

5.3 RA/H Ratio

In Section 4.3, we speculated on the effects of the ratio between the width of the ra^* and the f^* portion and their relation. We now examine the observed data and see whether our model works correctly.

We used the Avrora simulator (Titzer 2005) for replaying the execution and computing the data on the RA/H Ratio. In each program instruction the algorithm shown below was executed. First, the fingerprint of the actual instruction is calculated and the counters' value are saved. If the fingerprint is in the Database, the algorithm will look for all hash values with the same ra^* prefix; that is, all data that have the same fingerprint. Then, if the actual instruction's ra^* and f^* are equal to the ra^* and f^* saved in the DB, a fingerprint hit will be detected. If the complete RA is the same we found a false loop positive, otherwise we found a false RA positive. If the ra^* and f^* differ, then the algorithm will compare the complete execution context (all general-purpose registers, I/O registers and SRAM memory) and if it matches, it will count as a correct fingerprint hit and the original counter values are restored to the previous values. Otherwise, a false input duplicate or hash collision has been detected.

We first experimented with different widths of ra^*. The width of ra^* influences how many addresses of the address space map use the same postfix of r^*. It therefore should directly influence the number of false RA positives observed at replay time. Figure 7 shows the results and clearly supports our hypothesis. The x-axis shows the number of bits of a 16 bit fingerprint devoted to the return address. For example, a value of 4 means, that the first four bits of the fingerprint store the four least significant bits of the return address. The y-axis shows the number of false RA positives as percent of the worst value achieved. So a value of 0.4 means that the number of false RA positives has been reduced by 60 percent compared to the worst possible case. Note that each instruction has two bytes, so the program counter address always increase by two. The results therefore show no difference between using zero or one bit for ra^*. Already using a few instructionssignificantly decrease the number of false RA positives as seems to follow a trend of $\frac{1}{x}$.

```
fingerprint = calculate_fingerprint();

save_original_counters();

if(fingerprint is in DB) {
    for(all data which have the same fingerprint) {
        if(RA* == saved.RA* && f* == saved.f*) {
            counter_fp_hit++;
            if(FULL_RA == saved.FULL_RA) {
                counter_false_loop_positives++;
            } else {
                counter_false_RA_positives++;
            }
        } else {
            if(context executions are different) {
                if(selector function is the same) {
                    counter_false_input_duplicate++;
                } else {
                    counter_hash_collisions++;
                }
            } else {
                counter_fp_hit_correct++;
                restore_original_counters();
            }
        }
    }
}
```

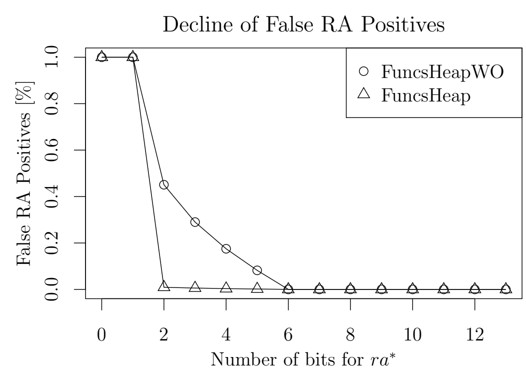

Figure 7. As the available bits for ra^* increases, the false RA positives decrease.

We then experimented with different widths of f^*. The width of f^* influences the available space for different recorded interrupts that map to the same ra^* portion. The smaller the width of f^*, the more hash collisions should occur at run time. Figure 8 shows the results and clearly corroborates our hypothesis. ADCApp was left out due to computational complexity problems in the simulation of the ADC capturing loop and the data aggregation. The x-axis shows the number of bits of a 16bit fingerprint devoted to the hash portion of the fingerprint. A value of 10 means, that the hashing function h returns a 10 bit value. The y-axis shows the median of the number of stored fingerprints per hash value as percent of the worst-case number. As the number of bits of f^* increases, the number of entries per hash value decreases non-linearly following

$\frac{1}{x}$. Note that Bob Jenkin's hashing produces consistently 0 or 1 for low has bit widths, so we started with a width of three bits.

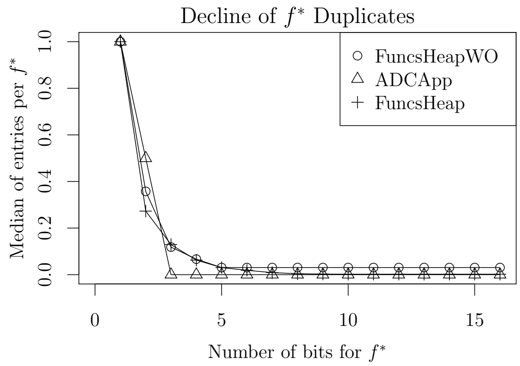

Figure 8. As the available bits for f^* increases, the duplicates decrease.

The observed clear and consistent decrease is surprising. The reason is that, the larger the width for f^*, the smaller the available width for ra^*. More fingerprints will have the same prefix and therefore more fingerprints will be stored in the same hash space of f^*. For example, thinking intuitively, if one records 10,000 fingerprints, with a ra^* width of three, then each the three hash spaces of f^* will have to store about 3,333 fingerprints each. If one duplicates the ra^* portion, then the hash spaces will only have to store about 1,600 fingerprints.

One cannot intuitively reason about this situation, because RA clustering breaks the assumption that return addresses are evenly distributed across recorded interrupts. This explains the surprise. The space increase for hash values achieved by growing widths of f^* offsets the described partitioning.

Figure 9 shows the design space for the fingerprint in the FuncsHeapWithoutLoop application. The other applications have a similar design space. The valley describes the optimal partitioning between the ra^* and f^* portion of the fingerprint. The differences among the individual configurations in the middle is negligible, so as a rule of thumb, a developer should divide the available bits evenly between ra^* and f^*.

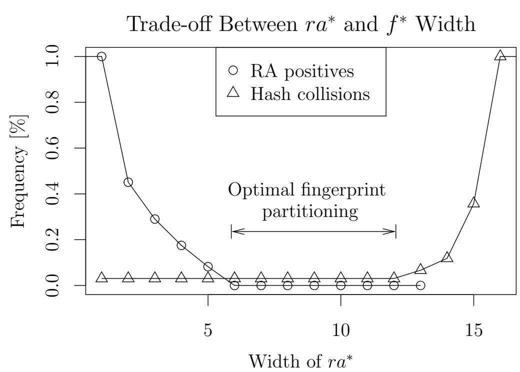

Figure 9. As the available bits for ra^* increases, the false RA positives decrease.

6. Discussion

During our work, we observed a number of interesting facts. The clustering of return addresses as described in Section 3 has profound impact on how well the tracing mechanism works. Here we describe other observations:

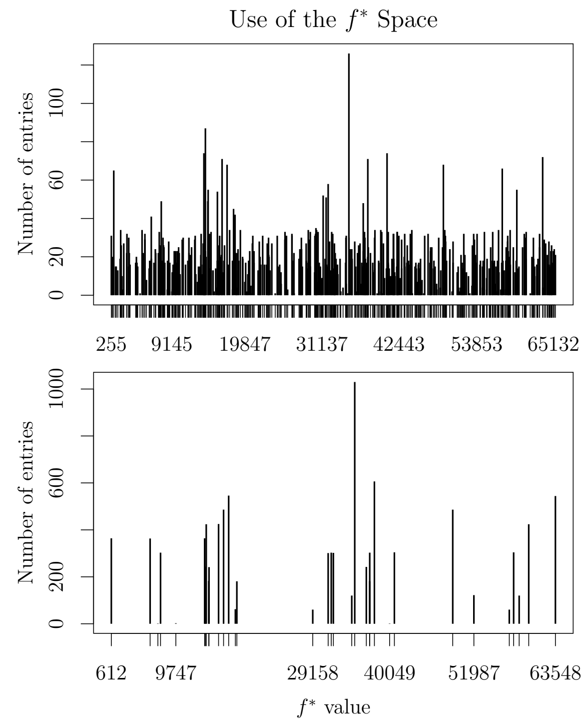

Figure 10. Poor use of the hashing space of f^*.

- **Uneven use of hash space.** Hash functions without funnels and characteristics still unevenly use the hash space. Figure 10 shows this fact. The x-axis lists hash values for 16 bit hashing. The y-axis shows the number of hash collisions for each hash value. The upper part shows the data for the funcsHeapWithLoopFP application with Bob Jenkins hashing. It shows a good even distribution except a few outliers. Still it uses only 0.7 percent of the available hash space after recording 10.000 fingerprints. The lower part shows the same result for the funcsHeapWithoutLoop application. It clearly shows that only very few hash values are used—only 0.05 percent—and a high number of collisions.
- **Round robin hashing works.** Since each hash function only uses a very limited portion of the hash space, using multiple hash function, as round robin hashing does, permits better use of the available space with minimal collisions among the hash functions. Yet the developer has to consider the program memory and execution time overhead (one conditional branch) of using multiple hash functions. Table 1 shows the overhead of the individual hash functions. Applying round robin then adds the overhead of the individual ones plus one additional switch statement.

***Hash* Function**	**Section**		
	.text	*.data*	*.bss*
Modular	4	0	0
2 Modular	92	0	0
Bob Jenkis	216	0	0
Multiplicative	126	0	0

Table 1. Hash functions memory consumption (in bytes).

- **Cancellation among input components.** Related and also our work sum up all input components before applying the hash function. Although the frequency-based selection algorithm minimizes false input duplicates, the hashing mechanism partially cancels this. This results in an increased number of hash collisions although a good selector function. Using message digests might eliminate this, however, their run-time overhead is large for microcontroller and deeply embedded applications.

- **Input components restrict hash function choice.** The developer must be careful with simple hash functions such as modular hashing. Even if the hash space is large, the maximum value achievable by summing the input components might be too low. For example when using twenty 8-bit values as input components can maximally add up to 5,120. Modular hashing with a bit width of 16 bits is unsuited for this selector function.
- **Linear increase of hash collisions.** Collisions in the hash portion grow linearly with the number of recorded fingerprints. For example in the ADCApp, we captured the complete system state of 10,000 interrupts and calculated a function describing collisions in h^*. Surprisingly, this relationship is nearly linear and we can describe it by the line $y = 0.75x - 1.51$ with negligible error. This means that the ratio of hash collisions to the total number of recorded fingerprints stays the same regardless whether we record for a short or a long time.

Our heuristic and approach works specifically in the context of microcontroller-based systems with a good number of processor registers. These types of systems provide a rich execution context with many register—general purpose and I/O ones—and usually have tight resource constraints in memory and computation. The general principle of the presented work also applies to other systems, however, some of the arguments might not be as strong as they are in our case. For example, tracing interrupts on a platform with a small execution context might provide enough computational resources to capture the whole context. However, even systems such as the typical Microchip 8-bit PIC microcontroller which has essentially only one general-purpose register still benefit from our approach, because they still have a large number of I/O registers.

The tested applications are large programs as they fill the program memory of the microcontrollers and are sufficiently complex as our analysis also includes the operating systems on top of which the programs run. However, applications running off external memory can be several magnitudes larger than the tested ones. The presented heuristic may not work well for these systems, because large systems can include much code that is not executed frequently. In this case, the frequency-based selection may choose input components which are in the worst case invariant in the most-frequently executed paths. We already indicated that we suggest using static program analysis or profiling information, if the developer thinks that the current application is a notorious outlier or has large portions of code that are executed infrequently.

The insight of return address clustering also lead us the idea of using a different hashing algorithm each time we record an interrupt. This fights the problem of the non-uniform distribution of return addresses, however, it is impossible to implement a round-robin system like this. While the recording system is straightforward, the replay mechanism will fail. In the replay phase, we need to check each line during the simulation whether an interrupt has occurred. Assuming that this approach chooses a new hashing algorithm each time we record an interrupt, we cannot tell in the general case which hashing algorithm would be used at the current program address if it recorded an interrupt. We could tell which hashing algorithm is used, if we were to correctly recreate all interrupts without a single false positive of false negative. Since we cannot tell which hashing algorithms was used, the replay algorithm will use all of them. This will increase the number of false positives and false negatives to the point where the approach performs worse than using a single hashing algorithm.

7. Conclusions

Debugging interrupt-driven software is hard and takes a considerable amount of time during the software development process. Traditional methods are often inapplicable, because interrupts occur independent of the control flow. System tracing is a possible option for debugging the control flow with a number of advantages.

In this work, we investigated the recording part of a capture/replay mechanism and proposed heuristics for its steps. Frequency-based selection for input components proved to be an effective but simple to implement mechanism to lower the number of false input duplicates. In some cases the algorithm delivers similar results as picking all registers with only a few registers which translates into a speedup of $\frac{n}{32}$ for our hardware platform with n as the number of selected registers, and about an 800 percent increase compared to related work. Round-robin hashing proved to be superior to regular hashing algorithms with non-optimal selector functions. In the measured case, it was about three times more efficient. And finally, our experiments with the RA/H ratio assist in allocating bits to the return address portion and the hash portion of the fingerprint.

With these encouraging results, there is still lots of room for improvement. Our frequency-based selection algorithm ignore control flow and we expect that program analysis can enhance our algorithm. The use of message digests might solve the problem of uneven use of hash spaces, however, one has to be careful as they incur considerable overhead too high for deeply embedded systems. Yet, work on message digests with short bit width and short execution time might improve the results.

8. Acknowledgements

This research was partly sponsored by the Graduate Students' Exchange Program (GSEP) funded by Foreign Affairs and International Trade Canada (DFAIT), Ontario Research Fund Research Excellence (ORF-RE) program, and by CIMIT under U.S. Army Medical Research Acquisition Activity Cooperative Agreement W81XWH-07-2-0011. The information contained herein does not necessarily reflect the position or policy of the US Government, and no official endorsement should be inferred.

References

Standard Test Access Port and Boundary-Scan Architecture. IEEE 1149.1 standard, 2001.

Standard for a Global Embedded Processor Debug Interface, 2003.

J. Baginski and H. Seiffert. Interaktives Trace- und Debugging- System ALGOL KIEL X 8. In *3. Fachtagung über Programmiersprachen, Gesellschaft für Informatik*, pages 173–188, London, UK, 1974. Springer-Verlag. ISBN 3-540-06666-7.

D. H. Barnes and L. L. Wear. Instruction Tracing via Microprogramming. In *Conference record of the 7th Annual Workshop on Microprogramming (MICRO 7)*, pages 25–27, New York, NY, USA, 1974. ACM. doi: http://doi.acm.org/10.1145/800118.803836.

B. Bouyssounouse and J.Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *LNCS*. Springer, first edition, May 2005.

T. A. Cargill and B. N. Locanthi. Cheap hardware support for software debugging and profiling. In *Proc. of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 82–83, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-8186-0805-6. doi: http://doi.acm.org/10.1145/36206.36187.

P.S. Dodd and C.V. Ravishankar. *Monitoring and debugging distributed real-time programs*, chapter Monitoring and debugging distributed real-time programs, pages 143–157. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995. ISBN 0-8186-6537-8.

E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud. Practical Minimal Perfect Hash Functions for Large Databases. *Communications of the ACM*, 35(1):105–121, 1992. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/129617.129623.

A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

M.P. Gallaher and B.M. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards & Technologg Planning Report 02–03, May 2002.

R. Jenkins. Hash Functions. *Dr. Dobb's*, September 1997.

M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Form. Methods Syst. Des.*, 24(2):129–155, 2004. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/B:FORM.0000017719.43755.7c.

D. E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, November 1997. ISBN 0201896842. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0201896842`.

K. London. *Mastering algorithms with C*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999. ISBN 1-56592-453-3.

J. M. Mellor-Crummey and T. J. LeBlanc. A Software Instruction Counter. In *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 78–86, New York, NY, USA, 1989. ACM. ISBN 0-89791-300-0. doi: http://doi.acm.org/10.1145/70082.68189.

L. J. Moore and A. R. Moya. Non-Intrusive Debug Technique for Embedded Programming. In *Proc. of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, page 375, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2007-3.

W. Omre. Debug and Trace for Multicore SoCs. Technical report, ARM, September 2008.

M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions Computer Systems*, 17(2):133–152, 1999. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/312203.312214.

M. Ronsse, K. De Bosschere, M. Christiaens, J.C. de Kergommeaux, and D. Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, 2003. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/903893.903895.

D. Sundmark and H. Thane. Pinpointing Interrupts in Embedded Real-Time Systemsusing Context Checksums. In *Proc. of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Hamburg, Germany, 2008.

D. Sundmark, H. Thane, J. Huselius, and A. Pettersson. Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies. In *Proc. of the 5th International Workshop on Algorithmic and Automated Debugging (AADEBUG)*, pages 211–222, Gent, Belgium, September 2003. URL `http://www.mrtc.mdh.se/index.php?choice=publications&id=0573`.

A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Department of Computer Science and Electronics, Mälardalens University, May 2000. URL `http://www.mrtc.mdh.se/index.php?choice=publications&id=0242`.

B. L. Titzer. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, pages 477–482, 2005.

J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi. A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. *IEEE Trans. Softw. Eng.*, 16(8):897–916, 1990. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.57626.