Pinpointing Interrupts in Embedded Real-Time Systems using Context Checksums

Daniel Sundmark and Henrik Thane
MRTC, Mälardalen University
Box 883, SE-721 23 Västerås, Sweden
{daniel.sundmark, henrik.thane}@mdh.se

Abstract

When trying to track down bugs using cyclic debugging, the ability to correctly reproduce executions is imperative. In sequential, deterministic, non-real-time software, this reproducibility is inherent. However, when the execution is affected by preemptive interrupts, this will have severe effects on the ability to reproduce program behaviors deterministically, since a reproduction requires the interrupts to hit the program at the exact same instructions. In previous methods, this problem has been solved using different kinds of instruction counters, that induce large execution time perturbations, demand for specialized hardware, or provide inexact results. This makes them highly unfit for resource-constrained embedded real-time systems.

In this paper, we propose an alternative method for pinpointing interrupts in embedded real-time systems using context checksums, which is not dependent on specific hardware features or special compilers - but which rather can be applied to any system. Although context checksums in some cases also prove inexact or ambiguous, we will show that they serve as a practical method for pinpointing and reproducing interrupts in embedded real-time systems. Furthermore, our method performs perfectly well with standard development tools and operating systems, requires no additional hardware support and, according to preliminary results, consumes merely a tenth of the execution time of existing software-based methods for pinpointing interrupts.

1 Introduction

Cyclic debugging is the process of examining the behavior of a faulty execution in an iterative fashion, thereby narrowing down the temporal and functional scope within which the system infection (the execution of a bug) might have taken place. By doing this, the bug can hopefully be found and safely removed.

1.1 Background

Reproducibility in cyclic debugging is trivial as long as the programs under investigation are deterministic (i.e. their execution behavior depends only on input parameters provided and controlled by the user). However, correct reproduction of execution behavior has always been a problem during cyclic debugging of non-deterministic programs. Since such programs exhibit different execution behavior over different executions, encountered failures are hard to reproduce and therefore hard to investigate in depth.

Interrupts are a major source of non-determinism in program executions. As an interrupt occurs, the ongoing CPU activity is halted, the state of the executing program is stored and the interrupt is handled by an interrupt service routine (ISR). To reproduce this scenario (e.g. for debugging), we need to be able to correctly reproduce the occurrence of the interrupt. In other words, we must make sure that the interrupt preempts the execution of the CPU activity at the exact same state during the reproduction as it did in the first execution. An intuitive solution might be to make use of the program counter value at the occurrence of the interrupt. If we make sure that the interrupt is forced upon the program at the same program counter value during debugging, the program will be interrupted at the correct instruction. However, program counter values might be revisited. A loop, e.g., could execute a number of instructions during each iteration. Using solely the program counter, we cannot distinguish an interrupt occurring at some program counter value during the third iteration of the loop from one occurring at the same instruction during the seventh iteration of the loop. Consequently, additional information is required. We need some sort of a *unique marker*.

1.2 Related Work

Traditionally, the simplest unique marker technique is the instruction counter. In its basic form, this is a mechanism that counts machine code instructions as they are exe-

cuted. When an interrupt occurs, the counter value is sampled along with the program counter in order to pinpoint the location of occurrence.

1.2.1 Instruction Counters

In order to be able to count each instruction as it is executed, there is a need for some kind of specialized hardware [5, 7]. The hardware required to count instructions is a simple counter incremented on each instruction execution cycle. Even though the hardware instruction counter technique solves the problem of uniquely pinpointing the location of occurence of interrupts, the method has drawbacks. One of the more significant problems is lack of the hardware needed in modern state-of-the-practice embedded microprocessors. Even though some of the larger processors available today have registers capable of performing instruction counting [1], this is no standard component in smaller or embedded processors. In addition, for many existing hardware platforms, there is reason to doubt the accuracy of the instruction counters [10].

Another problem is the sampling of the instruction counter. For applications using an Operating System (OS) or a Real-Time Operating System (RTOS), this may call for OS- or RTOS support. For example, consider an interrupt occurring at time t_{evt}. At time $t_{evt} + \delta$, the hardware instruction counter is sampled. Obviously, we will receive the instruction counter value of the latter, giving us a sampling error equal to the number of instructions executed during δ. This might be problematic, especially if δ varies from time to time, due to ISR- or kernel jitter.

1.2.2 Software Instruction Counters

In 1989, Mellor-Crummey and LeBlanc proposed the use of a software instruction counter (SIC) [9], suitable for systems running on top of platforms not equipped with the instruction counter hardware. However, since a software-based instruction counter performing the same task as a traditional hardware instruction counter would incur an intolerable overhead on the system, the software counterpart has to be much more restrictive when selecting upon which instructions to increment.

The SIC idea is based on the fact that only backward branches in a program can cause program counter values to be revisited. For instance, in a sequential program without backward branches, no instruction will be executed more than once. In such a system, the program counter is a unique marker, defining unique states in the execution. However, using structures such as loops, subroutines and recursive calls will require backward branches. Due to performance reasons, the implementation of the SIC not only increments on backward branches, but also on forward branches. In short, the SIC is implemented as a register-bound counter, requiring special compiler support. In addition, a platform-specific tool is used to instrument the machine code with incrementing instructions before each branch. According to the authors, the SIC incurs an execution overhead of approximately 10% in the instrumented programs.

The problem of getting the correct instruction counter value at sampling time $t_{evt} + \delta$ is not solved using software instruction counters, although we are only interested in the number of backward branches rather than the number of instructions during δ.

There has also been improvements to the SIC method, e.g., [8], which managed an 4 – 22% overhead reduction rate (i.e., a resulting CPU utilization overhead of about 9%) by analysing the machine code and separating deterministic scopes from non-deterministic scopes, and only instrumenting the non-deterministic scopes. Furthermore, [4], used an approximate SIC, based on entry- and exit points of ISRs, and was thus able to reproduce orderings of interrupts correctly, but the interrupts will not be reproduced at the correct instruction.

1.2.3 Trace Port Solutions

It should also be noted that, for many microcontrollers, there exist hardware-based solutions (e.g., JTAG [11] and BDM [6]), where a trace port is able to in detail track the execution of the system. However, even though these solutions are useful in a lab environment, they are not well suited for instrumentation of deployed systems in their intended environment. Therefore, they are not considered in the scope of this paper.

1.3 Problem Formulation

As discussed in the above section, several methods have been proposed for pinpointing and reproducing interrupts. However, due to variouos drawbacks, none of these methods has been fully accepted. All instruction counter methods require platform-dependent specialized development tools, such as specialized compilers, in order to work. When discussing embedded systems, additional drawbacks become significant. Very few embedded microcontrollers are equipped with sufficiently accurate hardware instruction counters. Turning to software instruction counters, these will require approximately 10% of the overall execution time, a hard to meet requirement in resource constrained systems. Consequently, a different approach, more adapted to the requirements of embedded systems and able to perform using standard development tools, is needed.

1.4 Contribution

In this paper, we present a novel method for pinpointing interrupts, suitable for embedded real-time systems. In our

method, we use an approximation of the state of the preempted program at the time of the interrupt as a marker. This approximation is represented in the form of the stack pointer and a checksum of the execution environment of the program, such as the registers or (part of) the program stack. According to preliminary results, our method imposes an execution time overhead of approximately 0.002 – 0.37 % (27 – 5000 times less than that of the SIC) and requires no additional hardware support to work. Our method primarily focuses on embedded multi-tasking real-time systems, running on top of a real-time operating system. However, the method also applies to simpler system models without operating system support, using interrupts to handle external events. Here the calculation of the checksums could be performed within the ISRs.

In our previous work, we have proposed a method, Deterministic Replay [15], for recording and reproducing (or replaying) the execution behavior of non-deterministic real-time systems. The technique presented and evaluated in this paper is incorporated into the Deterministic Replay method, allowing replay debugging of interrupt-driven embedded real-time systems.

1.5 Paper Outline

The remainder of this paper is organized as follows: In Section 2, our method is described in detail. In Section 3, we address the issue of the accuracy of our approximative method and in Section 4, we evaluate this accuracy through simulation. Here, we also evaluate the system perturbation of our method. In Section 6, we conclude the paper and Section 5 discusses future work.

2 Context Checksums

The basic idea of our method is to reproduce interrupts by recording the program counter values and unique markers of their occurrence. In order to reproduce these interrupts, a debugger breakpoint is set at each interrupted program counter address and the program is restarted in the debugger. As a breakpoint is hit, the unique marker of the current execution is compared to the recorded unique marker value. If these markers match, we consider this interrupt to be pinpointed and an interrupt is forced upon the system.

In this section, we will describe how our method uses the stack pointer together with approximations of the data in the execution context as unique markers. As an introduction, we will describe our concept of the execution context.

2.1 Execution Context

Looking back on the example formulated in Section 1.1, where a program counter value is indistinguishably revisited a number of times, a good solution in theory would be to make use of the loop counter together with the program counter value as a unique marker. Unfortunately, not all loops have loop counters. In a more general sense, it is very hard to determine exactly which parts of the program context that differentiate between specific loop iterations, subroutine calls or recursive calls. Ideally, we would base our unique marker on the *entire* content of the execution context in order to be able to differentiate between loop iterations. However, considering the amount of data used to represent this context, we face a practical problem when recording it during execution due to the massive perturbation to the system. Consequently, we need to derive a subset of the execution context suitable for unique marker use.

The program execution context is basically a set of data stored in registers, on the stack or on the heap. Since the processor registers are small and very fast, these hold the most current parts of the execution context. And, since they are small and very fast, they make exellent candidates as a basis for the execution context-based unique markers.

2.2 Register Checksum

One solution would be to store the contents of each processor register. However, in most embedded systems, memory resources are scarce. Storing all registers at each interrupt might incur an intolerable perturbation on the memory usage of the system. In our method, we handle this problem by separately storing the stack pointer, as it is invaluable for differentiating between recursive calls, and calculating and storing a checksum of the contents of the remaining registers. By doing this, we destroy information, but still preserve an approximative representation of the register contents from the time of the interrupt.

The register checksum operation is a simple addition of all processor registers. Overflow of the accumulated checksum is ignored. Hence, if the processor is equipped with eight general-purpose 16-bit registers ($R_0 ... R_7$), the register checksum C_R is calculated as follows:

$$C_R = (R_0 + R_1 + ... + R_7) \ \ mod \ \ 2^{16}$$

Due to the modest size and the ease of access of processor registers, the computational cost of calculating a register checksum is very small. However, since the register checksum is based solely on the processor registers, its main disadvantage is that it only covers a minor subset of the execution context. If an interrupt occurs within a loop and the actual parameters differentiating between iterations are not included in this subset, we will not be able to uniquely pinpoint the occurrence of the interrupt.

2.3 Stack Checksum

In order to capture the interrupt occurrences not successfully pinpointed by the register checksum, we must expand the execution context included in the context checksum. As we already used the registers, the remainder of the execution context is located on the stack and on the heap. In our method, we chose to work with the program stack contents. This has two reasons: First, implementing a stack checksum calculation in an instrumentation probe [12] is significantly easier than implementing a heap checksum in the same probe. The stack area is well defined, continuous and often easy to access from within the probe. Second, without having extensive proof of this, we assume that variables influencing the program control flow, such as loop counters, are often allocated on the stack rather than on the heap.

The checksum operation of the stack checksum is identical to the one performed in order to calculate the register checksum. Here, the subset of the execution context included in the checksum is bounded by the boundaries of the stack of the executing program. Hence, on a 16-bit architecture, the stack checksum C_S is calculated using the following formula:

$$C_S = (S_{SP} + S_{SP+1} + ... + S_{SB}) \ \ mod \ \ 2^{16}$$

In the above formula, S_X denotes the byte at stack address X. SP denotes the value of the stack pointer at the time of the interrupt and SB denotes the value of the stack base of the interrupted program.

The stack checksum should be viewed upon as a complement to the register checksum rather than a stand-alone solution for pinpointing interrupts. The reason for this is the fact that the execution overhead of the stack checksum exceeds the overhead of the register checksum to such an extent that the perturbation of the latter becomes negligible. In addition, discarding the option of using a register checksum when choosing a stack checksum solution will eliminate the possibility of detecting changes in register-bound variables over loop iterations.

2.3.1 Instrumentation Jitter

Besides the size issue, there is another property that separates the register checksum from the stack checksum. For the register checksum, we always use the same number of elements in order to calculate the checksum. If the processor has eight registers, the checksum will always calculate the register checksum by accumulating the values stored in these eigth registers. Hence, we can guarantee a constant execution time as far as number of instructions are concerned. Using a stack checksum, the situation is different. The stack base is constant whereas the stack pointer varies over time. This implies a variable size of the stack and thus a variable execution time of the stack checksum calculation, depending on the size of the stack at the time of the interrupt. In addition, the execution time of the stack checksum calculation will only be bounded by the stack limit.

Variations in execution time of software are usually referred to as *jitter*. In multi-tasking systems (such as most embedded real-time systems), designers try to keep the jitter to a minimum, since it comprises the testability and analyzability of the system [14]. Therefore, jitter introduced by instrumentation activities (such as the stack checksum calculation) may complicate testing of sensitive systems, even though the instrumentation was included in order to increase the analyzability.

2.4 Partial Stack Checksum

To reduce the execution time perturbation and the problem of instrumentation jitter when using the stack checksum technique, an option is not to include the entire program stack in the stack checksum. A partial stack checksum C_P would be calculated similarily to the original stack checksum (once again on a 16-bit platform):

$$C_P = (S_{SP} + S_{SP+1} + ... + S_{SX}) \ \ mod \ \ 2^{16}$$

However, the upper boundary SX of the stack interval to be included in the checksum is chosen such that:

$$SP \leq SX \leq SB$$

By using this formula, we once again reduce the percentage of the execution context included in the stack checksum, thereby reducing the accuracy of the unique marker approximation. In turn, we obtain the following benefits:

- **Eliminating instrumentation jitter**

  By defining SX in terms of a constant positive offset to SP (denoted x in Figure 1) such that the interval $[SP, SX]$ delimits a constant number of bytes on the stack, we make sure that the stack checksum will be calculated using a constant number of instructions. This will eliminate the instrumentation jitter of the checksum calculation (not considering cache effects or similar). In the case where the size of the fixed interval $[SP, SX]$ exceeds the size of the actual program stack (i.e. when $SX > SB$), this can be detected and the remaining instructions can be simulated using additions of zero to the checksum or similar.

- **Bounding and reducing instrumentation overhead**

  Intuitively, reducing the percentage of the stack included in the stack checksum will reduce the execution time of the stack checksum calculation. By selecting a tolerable limit (according to system specification), the

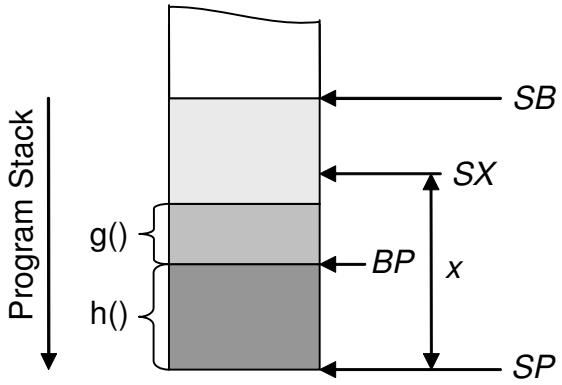

Figure 1. Different delimiter alternatives for the stack checksum.

instrumentation overhead can easily be reduced and bounded, while jitter is eliminated. On the other hand, if total elimination of instrumentation jitter is no major requirement, a reasonable candidate for SX may be the base of the stack for the current subroutine. Many processors are equipped with a dedicated register holding the value of this base pointer (BP in Figure 1) to the current scope of execution.

3 Approximation Accuracy

In the above section, we have proposed a set of approximations of unique markers designed to be able to pinpoint locations of occurrence of interrupts. As our method is inexact, this raises questions of how accurate these approximations are. In this section, we will discuss the ambiguity of our method, starting by describing our method in a more formal notion: Given a program P, we define S_P to be the set of states that can be reached in an arbitrary execution of P. Each element $s \in S_P$ is made up of a tuple $< pc_s, env_s >$, where pc_s is an executable address in the program code and env_s is a reachable program environment. Furthermore, an execution E_P of P is defined as an ordered set of visited states. Hence, in a more formal notion, given an execution E_P that is preempted by an interrupt at state s_i, our aim is to uniquely identify s_i.

If no information is extracted during the execution, we know nothing of when the interrupt preempted the program. The interrupt may have occurred anywhere during the execution. In other words, there is no element $s \in E_P$ that can be disqualified from being the potential state of the interrupt. By identifying the program counter of the state of the interrupt pc_i, we are able to eliminate a large subset of the set of visited states in the program execution. Those states for which $pc \neq pc_i$ can be discarded from further investigation. However, we are still left with a set of inseparable states, since we cannot differentiate between the various state environments. Adding the stack pointer and the register- and stack checksum will aid in further pruning the execution state set. Using these, we have access to an approximative representation of the program environment of s_i. Using these prunings, we will end up with a non-empty set E_i for which the following is true:

$$E_i = \{s : s \in E_P \wedge pc_i = pc_s \wedge env_i \approx env_s\}$$

Ideally, at this stage E_i will include exactly one element (the actual state of the interrupt). Unfortunately, we cannot guarantee that the environment approximation and the program counter value are not valid for other states in E_P. The reason for the inability of differentiating between s_i and other states in E_P is twofold:

- **Insufficient scope of execution context**

  Intuitively, including a smaller scope of execution context in the context checksum will increase the risk of leaving important variables out. Thus, a register checksum alone will provide less accuracy than a register checksum combined with a stack checksum. If we are dependent on variables located on the heap, which are addressed by memory direct machine code operations, typically possible in a CISC architecture, neither register- nor stack checksums will be of any use.

- **Checksum ambiguity**

  As the checksum by default is a non-reversible operation, two completely different stacks or sets of registers may give rise to the exact same checksum. For example, both $1 + 3 + 5$ and $7 + 2 + 0$ equals 9, even though they contain entirely different terms. In addition, since overflow is handled in no other way than a simple modulo operation, our method will not differ between a checksum of 1 and one of $2^n + 1$, where n is the number of bits in the checksum.

It should be noted that our method will never fail in pinpointing the correct state of the interrupt s_i. The problem is that it also might find *false positives*, i.e. it might pinpoint other states as well. Due to the fact that the interrupt-matching set E_i is ordered (states are ordered in the same sequence as in which they were visited in the original execution), in our current implementation, the method will choose the state that is reached first. Yet the question remains, how frequently do we find the correct state first?

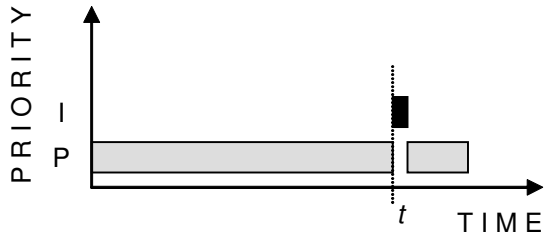

Figure 2. Execution of program P preempted at time t by interrupt I

4 Evaluation

In order to evaluate the approximation accuracy and the level of perturbation of our method, we have conducted a number of tests. In the accuracy tests, a tailor-made program P was written, executed and preempted by an interrupt I. Our test platform was the IAR Embedded Workbench (EW) [2], a commercial integrated development environment for embedded systems. We used the NEC V850 (a RISC architecture processor) version of EW and our tests were performed using the Deterministic Replay implementation on the EW V850 target simulator. Using the cycle counter of the simulator, we were able to simulate interrupts after a fixed number of clock cycles.

In our experiments, both P and I were implemented as real-time tasks running on top of the Asterix real-time operating system [13]. By varying the time t (or rather the number of clock cycles during t), we can cause I to preempt P at different states of its execution (see Figure 2). The accuracy tests and their results are described further in Section 4.1. As for the perturbation tests, these were performed on a 2.5 million LOC industrial robotics application [12] with approximately 70 tasks running on top of an Intel PII 400 MHz processor and the commercial VxWorks real-time operating system [3] in order to produce results relevant to industrial applications. These tests will be further discussed in Section 4.2.

There are several reasons for choosing different test platforms for different experiments. In a way, it would be desirable to use the full-scale industrial application for the accuracy tests as well. However, using a tailor-made program instead, it is possible to force execution scenarios upon the method in such a way that it is tested more thoroughly. In addition, due to the current implementations, applications running on top of Asterix are significantly more manageable with respect to interactive debugging. This is an invaluable property when examining and comparing program states during run-time.

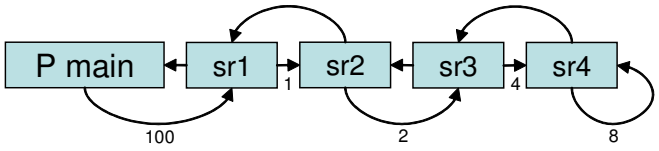

Figure 3. The structure of test program P.

4.1 Approximation Accuracy

To be able to test the accuracy of the context checksum methods under different circumstances, the program P was written such that its properties could be easily changed. Over all tests, however, P preserved its basic structure (depicted in Figure 3). In short, P consists of a main function and four subroutines $sr1..sr4$. In a loop with 100 iterations, P main calls $sr1$, which in turn performs some calculations and calls $sr2$. The $sr2$ subroutine, in turn, calls $sr3$ in two loop iterations. From $sr3$, $sr4$ is called four times. In its structure, $sr4$ has a loop that iterates eight times.

On the lowest subroutine level, this yields 6400 iterations ($100 * 1 * 2 * 4 * 8$) in the $sr4$ loop for each execution of P, meaning that each instruction in this loop is visited 6400 times. As a consequence, in an execution of P, every instruction of the $sr4$ loop will result in 6400 states in E_P. All of these states will have identical program counter values, but different environments. Hence, P is well suited for examination of how well our method will perform regarding differentiation between states with identical program counter values. In our tests, we used four different allocation schemes in order to investigate the performance of our method. These schemes were modeled such that the allocation of variables were placed in different parts of the execution context:

1. Stack Allocation 1

   In this scheme, all variables (loop counters and calculation variables) were allocated on the stack of each subroutine. No parameters were passed.

2. Stack Allocation 2

   This scheme allocated like Stack Allocation 1, but also passed parameters explicitly between subroutines.

3. Heap Allocation 1

   In this scheme, all variables were allocated globally. No parameters were passed.

4. Heap Allocation 2

   All variables were allocated globally and parameters were passed between subroutines.

	Stack allocation 1	Stack allocation 2	Heap allocation 1	Heap allocation 2
Register checksum	42%	45%*	54%	46%
Stack checksum	100%	100%	68%*	73%
Partial stack checksum	100%	93%	59%*	49%

Figure 4. Success rates for different unique marker techniques.

	Register checksum	Stack checksum
WCET	1.75 µs	0.11 ms
BCET	0.37 µs	0.40 µs
Overall perturbartion	0.002 - 0,007%	0.003 - 0.37%

Figure 5. Perturbation levels for stack- and register checksum.

Each of these schemes were tested using the register checksum, the stack checksum and the partial stack checksum. We used the scope of the current subroutine as a delimiter of the partial stack interval (corresponding to the $[SP, BP]$ interval in Figure 1). This yielded 12 different test scenarios, all tested by executing P during t time units. At time t, P was preempted by an interrupt, and the unique marker and the program counter were sampled. Then, P was deterministically re-executed until the program counter and the unique marker matched those that were sampled. At this point, we compared the current state of execution with the original interrupt state by comparing the values of a set of globally defined loop counters. If the states matched, we considered the test successful.

As stated in Section 4, by varying t, we can cause the interrupt to preempt the program at different states each time. In our tests, we started at a t value of 10000 clock cycles and in increments of 200, we raised it to 18000 cycles. This produced 41 test cases in each of the 12 test scenarios. The reason for the sparse number of tests is that they had to be performed by hand. Each test case yielded a binary outcome (either the interrupt is successfully pinpointed, or it is not). In Figure 4, the results of the accuracy simulations are shown. Naturally, since the partial stack checksum and the stack checksum are complementary techniques to the register checksum, these always exhibit a better accuracy. If all variables are allocated on the stack, the stack checksum techniques outperformed the register checksum techniques by far. However, if all variables were allocated on the heap, the difference was not that significant. It should also be noted that a checksum of parts of the stack in many cases performed nearly as well as a full stack checksum.

4.2 Perturbation

As the unique marker checksums need to be sampled during run-time at the occurrence of an interrupt, this imposes a perturbation to the execution of the context switch. However, contrary to the perturbation of the SIC, discussed in Section 1.2.2, the size of this perturbation is not proportional to the number of branches in the code, but to the number of interrupts in an execution.

In order to test the level of perturbation of our method, we measured the execution-time perturbation of the checksum calculations in a full-scale industrial robotics system [12]. While letting the system perform some simple robot arm movements, we sampled the unique markers at interrupt occurrences as well as the execution time of the unique marker code. The upper and lower boundaries of the instrumentation execution time is presented in Figure 5.

As we can see from the figure, the instrumentation jitter of the stack checksum is several magnitudes larger than that of the register checksum. The alterations in stack checksum execution time are mostly due to differences in the size of the stack at the time of the interrupt, whereas the register checksum execution time alterations are due to cache effects (since the registers are sampled from the task control blocks rather than the actual hardware registers). Regarding the level of perturbation, this should be compared with that of the software instruction counter [9], which requires approximately 10% of the overall CPU utilization.

5 Discussion and Future Work

When considering the results of the above evaluation, we would again like to stress that the program P used for this evaluation was designed to be malignant to our method. For example, both heap allocations assume that all variables affecting the program flow of control are allocated globally, a practice neither recommended nor common in real systems. However, as the checksum markers proposed in this paper for reproducing asynchronous events are approximate, and not always truly unique, we face a problem of handling the situations when the markers ambiguously pinpoint program locations. This basically occurs when the execution context of the location l_i of an interrupt i generates the same checksum as another location l_2, such that l_i and l_2 are on the same *PC* address and l_2 predates l_i in the execution. Luckily, this problem can be handled in a number of ways:

First, we believe that the precision of the checksum markers can be significantly improved by combining them with additional markers, such as *system call markers*, and

debugger cycle counters [2]. System call markers are per-task counters, incremented each time a potentially blocking system call is invoked by the task. If the call blocks the task, the value of the counter is recorded as a marker and the counter is reset. This way we may distinguish between in-between blocking intervals during reproduction. Debugger cycle counters increase marker accuracy by mapping the recorded event timestamps to small cycle counter intervals in which the event is feasible. As for now, inclusion of these markers is listed as future work. **Second**, if the ambiguity is discovered (by detecting dissimilarities between the initial and the replayed execution), the replay execution can be reset and restarted with an added skip count on l_2.

That being said, considering a full program execution, we might have to deal with pinpointing not only one, but two, four, or even hundreds of preemptive interrupts. Given an execution preempted by n_i uncorrelated interrupts and a P_i probability of pinpointing one interrupt correctly, the probability of reproducing the entire execution with all interrupts in place is $P_i^{n_i}$. With large numbers of interrupts, the probability of correctly reproducing the entire execution will be close to zero. Therefore, we will look further into ways of significantly raising the accuracy of our markers.

6 Conclusions

In this paper, we have presented an alternative to existing approaches for pinpointing interrupts in embedded real-time systems, based on checksums of parts of the execution context, e.g., registers or the stack. The need for non-standard tools of existing methods leads to problems when trying to port these methods to different platforms, processors, operating systems or compilers. Furthermore, these methods suffer from drawbacks such as insufficient hardware support [5, 7], inexact pinpointing of interrupts [4, 10] or large execution time perturbation [9].

Our method is approximate, but the accuracy depends on whether variables are allocated on the stack rather than on the heap. In the best cases presented in this paper, both the stack- and the partial stack checksum produce perfect results. This is achieved with an instrumentation perturbation at least ten times lower than that of the software instruction counter. Furthermore, our method only makes use of standard compilers, operating systems and platforms and requires no specialized hardware in order to work correctly.

References

[1] *Intel Architecture Software Developer's Manual 24319202*, 1999.

[2] www.iar.com, April 2008.

[3] www.windriver.com, April 2008.

[4] K. Audenaert and L. Levrouw. Interrupt Replay: A Debugging Method for Parallel Programs with Interrupts. *Journal of Microprocessors and Microsystems, Elsevier*, 18(10):601 – 612, December 1994.

[5] T. A. Cargill and B. N. Locanthi. Cheap Hardware Support for Software Debugging and Profiling. In *Proceedings of the 2nd International Conference on Architechtural Support for Programming Languages and Operating Systems*, pages 82 – 83, October 1987.

[6] S. Howard. A Background Debugging Mode Driver Package for Modular Microcontroller. Semiconductor Application Note AN1230/D, Motorola Inc., 1996.

[7] M. Johnson. Some Requirements for Architectural Support of Debugging. In *Proceedings of the Symposium on Arhcitectural Support for Programming Languages and Operating Systems*, pages 140 – 148. ACM, March 1982.

[8] D. Kim, Y.-H. Lee, D. Liu, and A. Lee. Enhanced Software Instruction Counter Method for Test Coverage Analysis of Real-Time Software. In *Proceedings of IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, March 2002.

[9] J. Mellor-Crummey and T. LeBlanc. A Software Instruction Counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78 – 86. ACM, April 1989.

[10] O. Oppitz. A Particular Bug Trap: Execution Replay Using Virtual Machines. In *M. Ronsse, K. De Bosschere (eds), proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG)*, pages 269 – 272. COmputer Research Repository (http://www.acm.org/corr/), September 2003.

[11] I. Std. IEEE Standard Test Access Port and Boundary-Scan Architecture. Technical Report 1532-2001, IEEE, 2001.

[12] D. Sundmark, H. Thane, J. Huselius, and A. Pettersson. Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies. In *Proceedings of the 5th International Workshop on Algorithmic and Automated Debugging (AADEBUG03)*, pages 211–222, Gent, Belgium, September 2003.

[13] H. Thane, A. Pettersson, and H. Hansson. Integration Testing of Fixed Priority Scheduled Real-Time Systems. In *Proceedings of Real-Time Embedded Systems Workshop*, London UK, December 3 2001.

[14] H. Thane and D. Sundmark. Debugging Using Time Machines: replay your embedded system's history. In *Proceedings of the Real-Time & Embedded Computing Conference*, page Kap 22, November 2001.

[15] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 288 – 295. ACM, April 2003.