FlashBox: A system for logging non-deterministic events in deployed embedded systems

Siddharth Choudhuri
Center for Embedded Computer Systems
University of California, Irvine
sid@ics.uci.edu

Tony Givargis
Center for Embedded Computer Systems
University of California, Irvine
givargis@uci.edu

ABSTRACT

The ability to postmortem failures in deployed systems due to non-deterministic events is useful in crash investigations. With this goal in mind, we propose *FlashBox* – a system that acts as a black box for embedded systems, recording non-deterministic events (interrupts). The FlashBox hardware consists of a microcontroller and flash memory. The FlashBox software is an extension to a compiler, enabling recording capabilities at various granularities. There are no source code modifications required to use FlashBox and no assumptions made on processor capabilities such as hardware counters. The FlashBox log can be used for faithful replay with a goal to isolate faults and reason about failure.

We present a prototype implementation of FlashBox that logs non-deterministic events on an AVR ATMega169 microcontroller. The FlashBox prototype consists of a 8051 microcontroller with flash memory. The `avr-gcc` compiler has been extended to log non-deterministic events. Based on our experimental results, FlashBox results in $10-23\%$ overhead while providing capability to log non-deterministic events at instruction level granularity. With decreasing cost of flash memories, FlashBox provides a low cost logging mechanism. The use of standard I/O communication protocols enhances portability, enabling ease of integration for different classes of embedded systems.

1. INTRODUCTION

Embedded systems software range from code running on a tiny microcontrollers to full blown operating systems running on high end embedded processors. One of the common threads underlying embedded systems software is to provide software reliability and robustness. This is due to the fact that embedded systems could be deployed in safety critical, real time and physical environments that are hard to reach. While utmost importance is given to ensure software reliability during design and testing phase of software development, bugs do creep up resulting in failures, sometimes catastrophic [7].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'09 March 8-12, 2009, Honolulu, Hawaii,U.S.A.

Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

Software failure in a runtime (deployed) system could be due to various reasons. One of the reasons that is difficult to catch during testing and design phase is due to *causality* i.e., bugs caused due to events that are interleaved in a way that is unforeseen or not taken care of during testing. Thus, the ability to postmortem a failed system and replay the exact events that led to a failure mode cannot be undermined. Such a capability could be crucial in embedded and real-time systems that are often part of a bigger system and a malfunction could lead to safety issues and revenue loss. For example, a critical piece of software running on BMW 745i had to be recalled, due to de-synchronization error, at millions of dollars in cost [30].

Traditionally, desktop class systems have facilities and mechanisms to log events that occur post deployment. The mechanisms provided could be hardware assisted, modifying software with logging support, installing special monitoring software or using a hybrid scheme involving one or more mechanisms [13, 15, 20]. On a failure, the logs are used to determine the cause of a failure, and in some cases reproduce the exact failure scenario. Such postmortem analysis of embedded systems, however, is non-trivial. Embedded systems, specifically post deployment, are hard to debug due to the following reasons. Most embedded systems have limited or no user interface making it difficult to interactively debug a deployed system. The amount of processing power in embedded systems is limited therefore, the operating system may not have the luxury of running logging or monitoring daemons. A given system could be deployed in a physically hostile environment. The amount of secondary storage in an embedded system might be limited and not available for logging events. Moreover, using an embedded system to log itself will consume main memory to store intermediate buffers, thereby affecting overall system performance.

In this paper, we propose *FlashBox* - a flash based system to record non-deterministic events (interrupts) in embedded systems that can aid in crash investigation and replay. FlashBox is a minimally invasive monitoring system composed of a flash memory and an off the shelf 8-bit microcontroller. FlashBox acts similar to a flight data recorder (blackbox) when integrated along with an embedded system. The software running on a system to be monitored logs non-deterministic events, interrupts in our case, to the FlashBox. There is no need for any source code modifications. FlashBox is a compiler assisted approach. Compiling applications with our modified compiler ensures that non-deterministic events are logged. We do not require that the system being monitored support any special hardware for

logging. The following are the contributions of this paper $--$ *(i)* A minimally invasive, low-cost system architecture to log non-deterministic events in deployed embedded systems with no assumptions on processor capabilities. *(ii)* Compiler techniques to aid logging non-deterministic events during run-time. *(ii)* A demonstrable system based on microcontroller, flash and extensions to `avr-gcc` compiler.

Software safety has long been an important issue in the embedded systems design [11]. Interrupts are an integral part of an embedded system software and also notorious for causing bugs that are hard to catch due to non-determinism. Eg: a malfunctioning Therac-25 medical system resulting in people receiving overdose of radiation has been well documented as one of the worst bugs in software history [12, 7]. The bug was non-deterministic (due to interrupts) and triggered by a way keys were pressed by an operator. The motivation behind this work is to record such non-deterministic events on deployed systems using a low cost, portable solution, thereby providing data that can be used to replay, reason and pinpoint the cause of a system crash.

Our methodology is motivated by the fact that embedded systems impose certain limitations that hinder the adoption of traditional debug replay schemes [9, 8, 17, 23, 13, 27] on deployed systems. For instance, an embedded processor may not have atomic test and set instructions that can avoid certain class of errors. Some of the existing techniques like watchdog timers simply reboot the system and provide little or no information on what caused the reboot. On board storage could be a serious deterrent to logging techniques. Formal verification could be hard in case of embedded software where often code consists of a mix of high level language and low level assembly routines. Moreover, formal verification techniques impose restrictions on coding style and model checking require that the code be represented in a form that is verifiable. Further, it cannot be assumed that the embedded system is networked thereby eliminating the logging over a network or providing a feedback path. A large class of solution has been based on modifying processor architecture/data-path, which might be cost prohibitive and based on the assumption that the processor IP is available.

The rest of the paper is organized as follows: Section 2 motivates the problem and addresses the related work followed by system architecture in Section 3. Section 4 consists of experimental results, analysis followed by conclusions.

2. RELATED WORK

One of the early approaches in dealing with non-determinism due to interrupts proposed in [23], was to convert the sources of non-determinism into messages that arrive at well defined points. However, this model was intended for distributed systems and does not fit in embedded systems where exact timing is desired. In [21], the authors propose a thread library that logs software interrupts (signals) for deterministic replay of threads. [1] The user level thread library is based on software counters [3]. Support for software counters may not be present in embedded processors. Besides, the proposed thread library is designed to log only signals. In our approach, we avoid using software counters due to potential overhead and the lack of software counter support in a large class of embedded processors. Our approach logs non-determinism due to interrupts and signals.

In [2], the goals of the authors is to statically check the stack size and latency analysis due to interrupts. One of the side effects of such an analysis is increased confidence in correctness of interrupt related code. While static analysis increases the confidence in code correctness, it does not eliminate timing related errors that are non-deterministic. Static analysis and model checking based approaches try to address the correctness of a given code thereby eliminating the need for testing execution paths or the need for logging software. However, such approaches are limited to what kind of code can be proven to be correct, and often, require that the programmer specify the source code in a form that can be verified [6]. In [18], the authors present a novel approach called RID - a restricted interrupt discipline whose goal is to design systems that makes the software robust by reducing the possibility of aberrant interrupts. Such an approach though useful, is not intended to log non-determnistic interrupts. RID can be used in conjunction with our approach.

In [17], the authors propose a system for high availability servers that maintains a state (checkpoints) and rolls back to a previous safe state on a software failure and starts re-executing. However, there are no guarantees that the re-execution will not lead to the same failure. This approach could eliminate non-deterministic bugs, but it is not suited for embedded and real time systems due to: *(i)* extensive checkpointing infrastructure built into the OS; and *(ii)* the fact that a real time system might be interacting with outside world (sensors, actuators) in which case reverting back to an earlier state is not a possibility. Further, approaches such as [17, 22] assume a full blown OS running on a system. This may not always hold true for embedded systems.

TTVM proposed by [9, 4] is an approach where a virtual machine is used to run an unmodified OS along with `gdb` in order to debug an operating system. Such an approach could be used in our work in order to perform a postmortem after a system crash. The advantage of a virtual machine is that it is a software solution and can provide mechanisms to freeze an operating system and also allows for the ability to feed the recorded interrupts as stimulus to the operating system being debugged. However, unlike desktops, running deployed systems on a virtualized environment may not be a feasible on embedded processors. A technique like TTVM can be used for replaying. `liblog`, a user space library is based on the idea of intercepting `libc` function calls and logging the function details along with the data [8]. Being aimed at distributed systems, `liblog` assumes the existence of spare resources, specifically processor, memory, network bandwidth and disk. `liblog` assumes that any activity outside of `libc` call is deterministic. Further, the granularity of replaying is at `libc` function call. Our approach is more fine grained and lightweight in terms of overhead. Jockey [19], like `liblog` is also a user space library and shares the limitations of user space replay debugging tool.

In [27], Thane et. al., share the similar goal of debug replay of real-time systems. The authors propose three kinds of logging mechanisms, two of which assume hardware support. The “type 2 - software recorders” based logging approach proposed in [27] is intended to serve as a blackbox that records events. This approach is closest to ours. However, the proposed software recorders are not minimally intrusive and do an in-system logging. The authors propose using a “marker” which is a tuple consisting of timestamp,

[1] We use *interrupt* to mean hardware interrupts and *signal* to mean software interrupts provided by a library

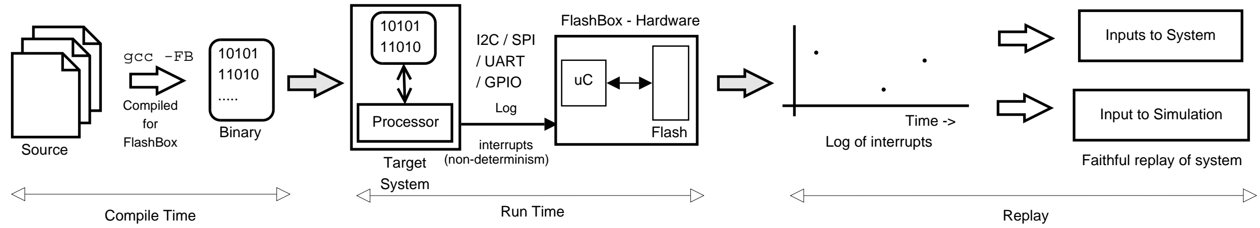

Figure 1: FlashBox Architecture

program counter, stack pointer and a checksum of the register set. Calculating and storing this tuple is more expensive compared to our approach due to *(i)* the amount of data stored at each marker and *(ii)* the checksum calculation that is done at each timestamp. Further, the replay method assumes access to instruction level simulators, JTAG or ICE. In the absence of of such hardware features, it is assumed that the underlying RTOS has the necessary hooks to record desired events. In our case, we do not make an assumption of such features. There is extensive work done by Thane et. al., [24, 25, 26] in the area of log and replay techniques for debugging real time systems, thus, based on special hardware support (JTAG, BDM etc). We are concerned with providing logging for in-field, deployed systems and thus do not make any assumptions on traditional hardware debugging facilities. A survey of rollback techniques can be found in [5]. There have been approaches (Eg: [29, 16]) that assume architectural changes.

3. SYSTEM ARCHITECTURE

FlashBox comprises of a hardware and a software component. The hardware component consists of a flash memory and a microcontroller. The software component is a modified compiler. Figure 1 depicts the FlashBox system architecture in the following phases: During *compile time*, application source code is compiled using an additional compiler flag, `-FB`. During *run time*, the FlashBox code sends (logs) occurrence of non-deterministic interrupts to the FlashBox hardware. The target system is interfaced with the FlashBox hardware using a standard protocol like UART, I^2C or GPIO pins. The log of interrupts can be used for faithfully reproducing the exact timeline of interrupts that occurred during runtime. Such an information can be used in conjunction with a simulator (Eg: [28]) or a virtual machine (Eg: [9]) to investigate (or recreate) a system crash.

3.1 FlashBox Hardware

A separate hardware approach was chosen so that the process of logging is minimally invasive. The flash could be directly interfaced with the target system. However, such an approach would impose overhead on the target system due to the flash controller software, the logging technique and main memory consumption. Moreover, making the FlashBox hardware a separate standalone entity makes it easier to be plugged into existing systems. The growing sizes and lowering cost is the primary motivation behind the choice of flash memory as a storage medium for FlashBox [10]. Besides, the newer high density flash technologies like the MLC large block NAND lend themselves very well to data that is written sequentially, such as log data in our case.

3.2 FlashBox Compiler

We have implemented the FlashBox compiler by adding code to existing gcc compiler (`avr-gcc`) for the Atmel AVR class of processors [1]. Code generation for the FlashBox is enabled by using `-FB` option along with `avr-gcc`. In order to capture this non-determinism due to interrupts, the FlashBox compiler checkpoints the instances in a process's execution path at which interrupts occur. Thus, whenever one of the desired interrupt triggers, the additional code due to the FlashBox compiler executes in the context of the interrupt service handler. This additional code logs the interrupt number and any additional data that might be required to uniquely identify the instance in time when the interrupt occurred. The process of logging implies an overhead on the target system. Therefore, depending on how accurately the non-deterministic events are logged, we provide two approaches to compiling an application for FlashBox that are a trade off between accuracy of logging versus overhead on the target system.

3.2.1 Approach 1

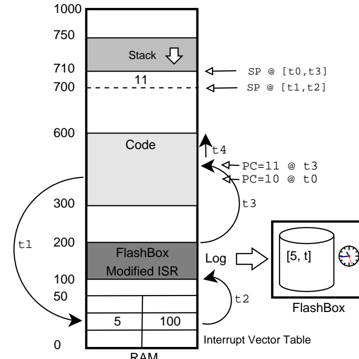

Figure 2: Approach 1

In this approach the overhead on the target system due to FlashBox is minimal. Each occurrence of a non-determinsitic interrupt is logged by the FlashBox. Figure 2 depicts a hypothetical situation. An interrupt occurs at time `t0` leading to the next PC value i.e., 11 getting stored on the stack. The control jumps to the IVT at time `t1` which redirects the PC to address 100 corresponding to the ISR for interrupt number 5. Starting `t2`, the ISR starts executing and sends log data to the FlashBox hardware. The FlashBox hardware uses its own clock as time stamp (i.e., `t`), and records the pair $\langle 5, \mathtt{t} \rangle$ uniquely identify the interrupt and the instance in time when the interrupt occurred. The advantage of this approach is that it is minimally invasive on the target system. This approach provides a way of logging non-deterministic interrupts with minimal overhead by delegating the job of time stamping non-deterministic interrupts to the FlashBox system. While this approach may not accurately pin point the exact times during replay, it might still be useful in narrowing down a range of time during which non-deterministic events occur. The disadvantage of this approach is potential lack of accuracy while replaying. Since the FlashBox hardware uses its own clock, the time stamp can only be as

accurate as the clock resolution of the FlashBox hardware. Note that the accuracy can be improved by using a common clock source or a phase locked oscillator.

3.2.2 Approach 2

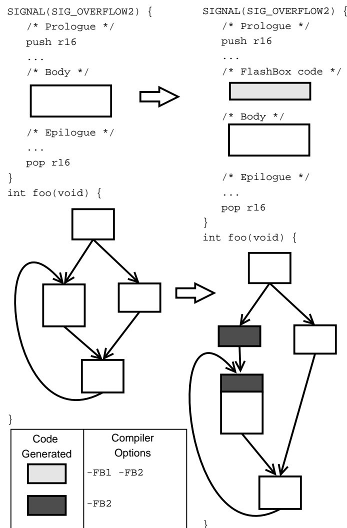

Figure 3: FlashBox code insertion

In this approach, the FlashBox compiler generates all the code generated by approach 1 (`-FB1` option). Additionally, this approach logs the value of PC when a non-deterministic interrupt occurs. This value of PC is obtained from the stack of the function that was interrupted. However, the PC cannot uniquely pin point the instance when a non-deterministic interrupt occurs due to the following two cases: *case (i)* during loop iterations, the execution of a program iterates over the same set of PC values. Thus, if there are more than one instance of a non-deterministic interrupt that occurs at different iterations of a loop, but at the same value of the PC, the combination of interrupt vector number and PC cannot distinguish between the multiple occurrence of the non-deterministic interrupts. *case (ii)* during the execution of a function call, the PC iterates over the same set of PC values. It is therefore possible that a given function `f()`, is called from two different places in a program, and a non-deterministic interrupt occurs at the same PC value during execution of `f()`. Such a case makes it difficult to distinguish between the two instances of the non-deterministic interrupts. This ambiguity can by eliminated by logging the PC value before invocation of the function `f()`. However, not all function calls need to be logged before their invocations. Note that, if a function call exists within a loop body, the loop entry is logged by the FlashBox. Therefore, a function call within a loop need not be logged before its invocation. Such an analysis can drastically reduce the number of functions that need to be logged before their invocation. The algorithm to determine a subset of function calls that need to be logged is presented in the next sub section.

Approach 2 logs the following: *(i)* the interrupt number and the PC value during the execution of an ISR, *(ii)* the PC value before the entry of a loop, *(iii)* the PC value before invocation of a function, (determined by algorithm 2), and *(iv)* a marker at every iteration of a loop that serves as a loop iteration counter. The advantage of this approach is that it provides precise log of non-deterministic events. Thus, it can be used to faithfully replay the exact instances of non-deterministic events in an application's runtime. However, the accuracy comes at a cost of runtime overhead. Figure 3 illustrates the additional code generated by approach 3 (i.e., `-FB2`) for a given signal handler and a function represented in terms of basic blocks.

3.2.3 Algorithm

There are three algorithms to generate code that logs data on the FlashBox hardware. The first algorithm, integrated into the compiler, generates logging code at every execution of a signal or an ISR. Algorithm 1 describes the process of generating logging code in the context of loops for the `-FB2` option. This algorithm is invoked post assembly code generation phase. The algorithm does two passes over every function. In the first pass (lines 6-15), the algorithm makes a list of all labels and back edges. In the second pass (lines 16-27), the log instructions are generated for each back edge. The code generation function returns the number of additional instructions added (denoted by variable `count`). This value is used by the linker at a later phase to reserve space in the instruction memory. The complexity of this algorithm is $O(N)$ per function given N instructions in a function.

Algorithm 2 determines a subset of functions in a program for which logging code needs to be generated. Its inputs are a program call tree and a list of loops. The *call tree* depicts the calling sequence with `main` as the root node (Figure 4) and is used to determine the list of functions that need logging. Such functions are determined using a recursive depth first traversal of the program call tree.

Algorithm 1 FlashBox code for loops

1: *Input:* Function `func`, Flag `flg`
2: *Output:* Function `func` with FlashBox code if required
3: List labels $L \leftarrow \phi$, List backedges $B \leftarrow \phi$
4: Instructions `count` $\leftarrow 0$
5: **if** `func` $\neq signal$ and `func` $\neq ISR$ **then**
6:     **for all** instruction $I \in$ `func` **do**
7:         **if** $I \in label_type$ **then**
8:             $L = L \cup \langle I, I.loc() \rangle$
9:         **else if** $I \in jump_type$ **then**
10:             $T \leftarrow I.target()$
11:             **if** $(L.lookup(T)) \in L$ **then**
12:                 $B = B \cup T$
13:             **end if**
14:         **end if**
15:     **end for**
16:     **for all** instruction $I \in$ `func` **do**
17:         **if** $I \in B$ **then**
18:             `count` $= generate_logcode()$;
19:             $generate_label()$;
20:             `count` $+= generate_logcode()$;
21:             $update_instr_count($`func`, `count`$)$;
22:         **end if**
23:     **end for**
24: **end if**

The functions that need FlashBox code before their invocations are determined in the following two cases: *(i)* if a function is outside of any loop and none of its caller functions (or ancestors) are part of a loop either (lines 4-6). *(ii)* if a function is part of the loop, however, the same function was called earlier in the same loop. (lines 7-9). Lines 10-13 follow the recursive algorithm for depth first traversal. The algorithm when applied to the code section shown in Figure 4 generates FlashBox code only for function `g()` before it is called by `main()`. Note that additional code is not generated for either `f()` or `g()` which are called inside the context of a loop. For recursive calls we place the code inside function.

Algorithm 2 FlashBox code for functions

1: gen_fb_func(T)
2: *Input:* Program Call Tree T, Loop List L
3: *Output:* Program annotated with FlashBox code
4: **if** $T \notin L$ and $ancestor(T) \notin L$ **then**
5: *generate_fb_code()*
6: **end if**
7: **if** $T \in L$ and $siblings(T) = T$ **then**
8: *generate_fb_code()*
9: **end if**
10: $T \leftarrow$ visited
11: **for all** vertices v adjacent to T not visited **do**
12: gen_fb_func(v)
13: **end for**

4. EXPERIMENTAL RESULTS

Figure 5 shows our prototype. The target system is an AVR butterfly board running Atmel ATmega169V (8MHz) 8-bit processor with 16KB of program memory and 512 bytes of data memory [1]. The butterfly board was chosen due to availability of gcc, glibc and support for multiple interrupts. The FlashBox hardware shown in Figure 5 consists of a 8051 microcontroller with external flash. We used NOR flash as a proof of concept. However, without loss of generality, this can be replaced with a NAND flash and additional software to manage flash.

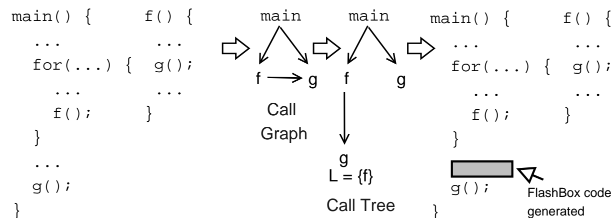

Figure 4: FlashBox code for function calls

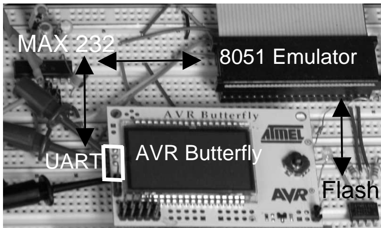

Figure 5: Experimental Setup

4.1 Benchmarks

We use two sets of benchmarks to evaluate our approach. The first set of benchmarks consists of two applications based on earlier work in the area of non-determinism. The first application is based on a bug described in [18]. This application called *ADC*, is part of TinyOS distribution for sensor networks. The bug occurs if an analog to digital convertor interrupt arrives at an unexpected period in time when the size of a circular buffer is being reset. The code was rewritten in C from its original implementation in the NesC language. The second application is based on an example from avionics provided in [6]. In this case if the interrupts do not arrive at a specific interval (10 Hz), the altitude meter in avionics system could lose its accuracy over an extended period of time. Our second set of results evaluate *(i)* the overhead due to loops and function calls, and *(ii)* the effect of hardware interface between the target system and the FlashBox hardware. We chose UART and GPIO for comparisons as these represent the worst and best case scenario in terms of lines of code required to log data (as per Table 1). We chose six applications from the powerstone benchmark suite [14].

Table 1: Interface Protocol Variations for AVR

Protocol	$UART$	I^2C	SPI	$GPIO$
Instructions	21	18	20	8
Pins	3	3	5	8

4.2 Results

Table 2 is intended to highlight the error due to using separate clocks on the target system and the FlashBox in case of approach 1. We ran the first set of benchmarks on the setup shown in Figure 5. There is no direct way of measuring clock cycles on the ATmega169 microcontroller. We therefore inserted loop counters in both the target system and the FlashBox. The conversion from loop counters to cycles were done on a simulator. The numbers shown in Table 2 was derived by converting loop counters into estimated clock cycles on the target system. Note that a given number of clock cycles on the FlashBox can be converted into an estimate of clock cycles on the target since we know the clock frequencies of both the target and the FlashBox. Further, we only show approach 1 since it results in the worst case error due to a non-PC based logging approach.

Although the percentage error of estimated cycles is negligible ($< 1\%$), this number is not sufficient to accurately log when an interrupt occurs. For example, in case of ADC, if the loop body is of the order of 1000 cycles, the FlashBox would estimate that 48359 iterations of loop elapsed between two successive occurrences of interrupts (as opposed to 48377 loop iterations on the target). Thus, while a non-PC based approach may not be good at pin pointing the exact interrupt instance, it can still be used to narrow down to a range of when an interrupt occurs. Further, the estimates in such cases can be improved by logging more frequently at some pre-identified points (finer granularity) in the application.

Table 2: Logging error in non-PC based approach

Application	FlashBox Estimate	Target Estimate
ADC	48359407	48377319
Avionics	10335144	10325304

We used the Avrora cycle accurate simulator [28] to run our second set of benchmarks. We used simulation due to the fact that the ATmega169 microcontroller does not provide mechanisms for program instrumentation (eg: evaluating the total cycle count) and also due to the existence of profiling tools (eg: `gcov`) in software. The powerstone benchmarks has a lightweight implementation of common glibc functions. Thus, for all our results, the overhead numbers presented take into account the overhead of FlashBox due to logging the application *and* the overhead due to logging the library functions used by the application.

Figure 6 depicts the overhead due to FlashBox. The benchmarks are compiled with `-FB2` option (approach 2) since this option is the most intrusive thereby providing a worst case scenario. Further, two kinds of interfaces UART and GPIO are considered. The first two plots depict the percentage increase in execution cycles due to FlashBox code. The benchmark entitled “average” is the arithmetic average

of all benchmarks. The next two plots depict the percentage increase in code size due to FlashBox. The increase in code size is calculated over the size of *hex* file generated post compilation and linking.

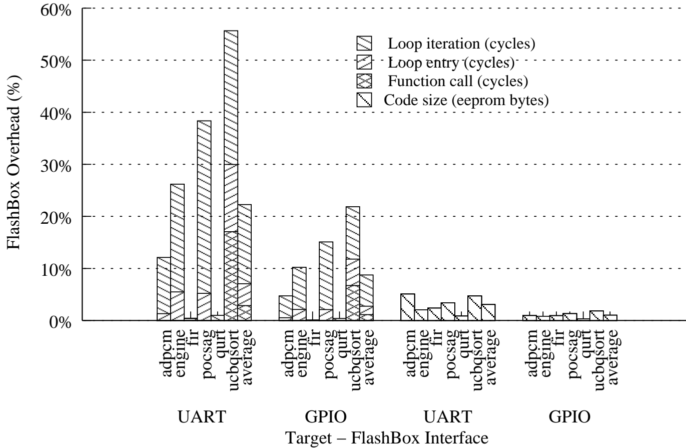

Figure 6: FlashBox execution, code size overheads

The following analysis is made based on Figure 6: *(i)* the interface used to communicate between the target system and the FlashBox hardware has a drastic impact on the overhead, both in terms of execution cycles and code size, more so on the execution cycles. Comparing UART and GPIO, it can be seen that a GPIO based interface results on an average close to 10% lesser overhead compared to a UART based interface. This is due to the fact that a GPIO interface is a parallel interface compared to a UART interface (serial). Moreover, data to be sent on UART requires a setup code. Similarly, it can be seen that using a GPIO interface results in lesser code size compared to a UART interface. *(ii)* the overhead due to logging function calls is negligible. This can be attributed to the fact that most of the functions are executed in the context of a loop as determined by Algorithm 2. *(iii)* the only application that shows a noticeable overhead due to logging function calls is *ucbqsort*, which is a recursive function. Since recursive functions log every call of the function, such functions impose considerable overhead in case of FlashBox. *(iv)* the runtime (execution cycle) overhead depends on the number of loops as well as the computational complexity of the program. For example in case of *pocsag*, the total number of loop iterations (3355) is lower compared to 19809 loop iterations in *engine*. However, the percentage overhead of *engine* is less than that of *pocsag*. This is due to the fact that *engine* is computationally much more expensive (i.e., runtime takes more cycles) compared to *pocsag*, thereby amortizing the overhead due to loops in the overall program runtime. *(v)* the contribution due to FlashBox code inside a loop entry dominates the total runtime overhead. This is obvious due to the fact that programs spend most of their execution cycles within loops. Also, note that if the "loop iteration" overhead is omitted from the histograms, the resulting histograms (loop entry + function call) would be the overhead due to approach 2. Approach 1 does not include any of the overheads presented in Figure 6.

Table 3 shows the log data generated (in bytes) by each run of the benchmark. Note that the amount of log data generated could depend on the program inputs. Therefore the numbers for data in Table 3 are not absolute. For example the log data generated for *ucbqsort* would vary depending on the input size of numbers to be sorted (since the number of recursive calls is proportional to numbers to be sorted). However, this is true for any benchmark, thus, the log data generation rate is application specific. The second row shows the number of functions that were logged over total number of function calls. For example, in case of *fir*, the total number of functions in call tree were 7 out of which 1 function was logged by FlashBox. Note that, comparing Table 3 and Figure 6, it can be seen that the overhead of logging function calls is negligible thereby showing the efficacy of Algorithm 1. The exception is the case of recursive function calls – *ucbqsort* in this case.

Table 3: Runtime statistics

	adpcm	engine	fir	pocsag	qurt	ucbqsort
Log Data	8418	75186	1605	11670	189	20520
Functions	2/12	1/6	1/7	1/17	2/6	2/6

```
1: __vector_5:
2: /* prologue: frame size=0 */
3:         push ...
4:         ...
5: /* prologue end (size=19) */
6: /* FlashBox code begin: -FB3 */
7:          in r28,__SP_L__   // load PC from Stack
8:          in r29,__SP_H__
9:          ld r22,-Y         // r22 = PC (High)
10:         ld r23,-Y         // r23 = PC (Low)
11:         ldi r25,lo8(5)    // r25 = Vect. Number
12:         ldi r30,lo8(192)  // Setup UART comm
13:         ldi r31,hi8(192)
14:         ldi r26,lo8(198)
15:         ldi r27,hi8(198)
16: .B30:   ld r24,Z          // Begin send UART
17:         sbrs r24,5
18:         rjmp .B30
19:         st X,r25          // Send Vect. Number
20: .B31:   ld r24,Z          // Begin send UART
21:         sbrs r24,5
22:         rjmp .B31
23:         st X,r23          // Send PC (Low)
24: .B32:   ld r24,Z          // Begin send UART
25:         sbrs r24,5
26:         rjmp .B32
27:         st X,r22          // Send PC (High)
28: /* FlashBox code: end */
29:         ...               // Handler code
30: /* epilogue: frame size=0 */
31:         pop ...
32:         ...
33:         reti
34: /* epilogue end (size=19) */
```

Figure 7: Code generated by FlashBox

4.3 Code Generation

Figure 7 shows the code generated inside a signal handler (`SIG_OVERFLOW2`) by FlashBox for a UART interface. The code generated does not save and restore registers used by the FlashBox code (lines 6-28) as the default signal handler saves (lines 2-5) and restores (lines 30-32) the registers. The code generated logs the interrupt vector number (lines 16-19), and the two byte PC (lines 20-27).

Note that *(i)* the number of instructions in case of GPIO would be much less (8 instructions) compared to the UART implementation. *(ii)* due to the architecture specifics of the ATmega169V microcontroller, the PC is not register accessible. Thus, in order to log the PC value for loops, multiple instructions are required. These instructions contribute to the overall runtime overhead. In case of an architecture like MIPS, where the PC is register accessible, the overhead

would reduce to a single instruction, thereby showing lesser runtime overhead due to FlashBox code.

Adding FlashBox code to loops that are used for timing would disrupt the purpose of such loops. Using approach 2 for infinite loops with minimal loop body can prove expensive due to the amount of data that would be generated. However, this problem can be solved by applying approach 2 for such loops. Also, the scope of faithful replay is limited by the size of flash memory. This, however, is a limitation with any kind of debug-replay mechanism. One of the ways to address this issue is to discard data based on age.

5. CONCLUSIONS AND FUTURE WORK

We presented FlashBox – a system to log non-determinism due to interrupts in embedded systems. FlashBox hardware is minimally intrusive making it easy to integrate with existing systems using well known I/O communication protocols. FlashBox software is a compiler assisted approach that can be integrated into existing compilers. Further, there are no source code modification necessary. Thus, for systems where the ability to postmortem or debug is critical, FlashBox can be integrated with a tolerable overhead.

For our future work, we would like to investigate ways to optimize the checkpointing code generated by FlashBox and evaluate the power consumption overhead. Adding FlashBox code to loops that are used for timing is disruptive. Using approach 2 for infinite loops with minimal loop body is expensive. The compiler can be extended to take hints from the programmer in order to optimize the two cases. Also, the scope of faithful replay is limited by the size of the flash memory. This, however, is a limitation with any debug-replay schemes. One of the ways to address this is to discard data based on age.

6. REFERENCES

[1] Atmel. AVR RISC processors. `http://www.atmel.com/products/avr/`, 2007.

[2] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. *ICSE*, 00:0047, 2001.

[3] T. A. Cargill and B. N. Locanthi. Cheap hardware support for software debugging and profiling. In *ASPLOS-II*, pages 82–83, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[6] C. Fidge and P. Cook. Model checking interrupt-dependent software. *ASPEC*, 0:51–58, 2005.

[7] S. Garfinkel. History's worst software bugs. *Byte Magazine*, November 2005.

[8] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications (awarded best paper). In *USENIX Annual Technical Conference, General Track*, pages 289–300, 2006.

[9] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 1–15, 2005.

[10] G. Lawton. Improved flash memory grows in popularity. *Computer*, 39(1):16–18, 2006.

[11] N. G. Leveson. Software safety in embedded computer systems. *Commun. ACM*, 34(2):34–46, 1991.

[12] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. volume 26, pages 18–41, Los Alamitos, CA, USA, 1993. IEEE Computer Society.

[13] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI 2003*, pages 141–154, New York, NY, USA, 2003. ACM Press.

[14] A. Malik, B. Moyer, and D. Cermak. A Lower Power Unified Cache Architecture Providing Power and Performance Flexibility. In *Internationl Symposium on Low Power Electronics and Design*, 2000.

[15] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

[16] B. Plattner. Real-time execution monitoring. pages 55–63, 1995.

[17] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP 2005*, pages 235–248, New York, NY, USA, 2005. ACM Press.

[18] J. Regehr. Random testing of interrupt-driven software. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 290–298, New York, NY, USA, 2005. ACM.

[19] Y. Saito. Jockey: a user-space library for record-replay debugging. In *ACM AADEBUG 2005*, pages 69–76, New York, NY, USA, 2005. ACM Press.

[20] J. Slye and E. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. *FTCS*, 00:250, 1996.

[21] J. H. Slye and E. Elnozahy. Support for software interrupts in log-based rollback-recovery. *IEEE Transactions on Computers*, 47(10):1113–1123, 1998.

[22] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX ATEC 2004*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[23] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[24] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, May 2000.

[25] H. Thane. Time machines and black box recorders for embedded systems software. *ERCIM News*, (52):32–33, January 2003.

[26] H. Thane and D. Sundmark. Debugging using time machines: replay your embedded system's history. In *Real-Time and Embedded Computing Conference*, page Kap 22, Milan, Italy, November 2001.

[27] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 288–295, Nice, France, April 2003. ACM.

[28] B. L. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[29] 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.

[30] USDoT. National highway traffic safety administration nhtsa 03v-240. `www.autotechdaily.com/pdfs/T02-05-03.pdf`, 2003.