On the Effects of Errors During Boot*

Mário Zenha-Rela[1], João Carlos Cunha[2],
Carlos Bruno Silva[1], and Luís Ferreira da Silva[2]

[1] University of Coimbra, 3030-290 Coimbra, Portugal
{mzrela,cbsilva}@dei.uc.pt
[2] DEIS/ISEC, 3030-199 Coimbra, Portugal
{jcunha,lmferrao}@isec.pt

Abstract. We present the results of injecting errors during the boot phase of an embedded real-time system based on the ERC32 space processor. In this phase the hardware is initialized, and the processor executes the boot loader followed by kernel initialization. For this reason most system support is not yet available and traditional fault-injection techniques such as SWIFI cannot be used. Thus our study was based in the processor's IEEE 1149.1 (boundary-scan) infrastructure through which we injected about 5000 double bit-flip errors. The observations show that such system will either crash(25%) or execute correctly(75%), since only 2 errors eventually lead to the output of wrong results. However about 10% of faults originated latent errors dormant in memory. We also provide some suggestions on what can be done to increase robustness during this system state, in which most fault-tolerance techniques are not yet setup.
Keywords: dependability evaluation, embedded systems, fault-tolerance, fault-injection, boundary-scan.

1 Introduction

Reset is the most common error-recovery mechanism present in embedded computer systems. When some non-permanent error is detected, a simple hardware or software module triggers a reset that has the ability to bring the system from an erroneous state into an error-free state. This last-resort technique is used from the smallest embedded device (e.g. smart cards, mobile phones) to complex computer control systems provided with high degrees of redundancy to detect and tolerate a large class of errors [1].

After a reset the system is assumed to be clean from errors (both detected and latent) and may resume execution, rolling back to a previous state or jumping forward into a new one. However, bringing a system from a hard reset into a fully operational state implies a long series of complex and sensitive operations, where the occurrence of a fault can lead to immediate drastic effects or stay latent until much later, with a high potential to induce a system failure.

* This work was partially supported by the R&D Unit 326/94 (Center for Informatics and Systems, CISUC), and the Portuguese Agency for Innovation (AdI) through project BSCAN4FI.

This point is exacerbated by the fact that the dependability issues of most critical systems are based on a single failure model or that the mean time between failures (MTBF) is much longer than the recovery process. This requirement is mandated both by economic reasons and to make the dependability issues tractable.

However in many circumstances faults occur in 'bursts', i.e. they are clustered in the time domain: while the MTBF may be large the occurrence of successive faults (the phenomenological cause that originates errors [2]) may be very close followed by long periods of inactivity. This problem is particularly acute in space, where cosmological events such as solar flares may affect an on-board satellite computer during a recovery [3]. In most situations this problem can be acceptably managed as a single 'long' fault. In such cases the system restarts operating as soon as the disturbance intensity goes down a specified threshold. However, in many situations the fluctuations of the disturbance can lead to successive nearby non-overlapping faults affecting the computer system.

The problem of dealing with multiple errors can be handled very satisfactorily by resetting the system: after a hardware reset the system is considered to be in an error-free state so the potentially multiple errors are simply wiped out. However, if an error generated during boot manages to pass undetected, it may not be sufficient to simply reboot the system until all tests pass (boot is not an atomic activity). Then, if another error occurs in operation, the consequences may be dramatic as the system may not be able to handle a multiple error situation.

To the best of our knowledge this problem has never been addressed in the literature, most probably due to the lack of proper tools. During boot the system kernel is not loaded, device handlers and monitoring software are not yet operational. Thus, the use of dedicated hardware monitoring tools is mandatory, but the complexity of modern processors makes this approach unfeasible or extremely complex. A recently proposed fault-injection approach based on the IEEE 1149.1 (boundary-scan) standard [4] associated with on-chip debugging facilities seems to overcome this problem since it is orthogonal to the chip functionalities, and is permanently available whenever the chip is powered [5, 6].

This paper presents the undergoing research aiming at clarifying the effects of transient faults that occur during a system boot. Permanent faults are not considered in this study, since they can be more easily detected by diagnostics hardware.

The remainder of the paper is organized as follows: in the next section we preset the methodology used in this study, namely the testbed, the workload, the faultload and the measurements that were performed. In section 3 the activities performed during the boot sequence that eventually lead to the application launch are described. Section 4 contains the experimental results and in the following section possible ways to avoid failure are discussed. Section 6 closes the paper.

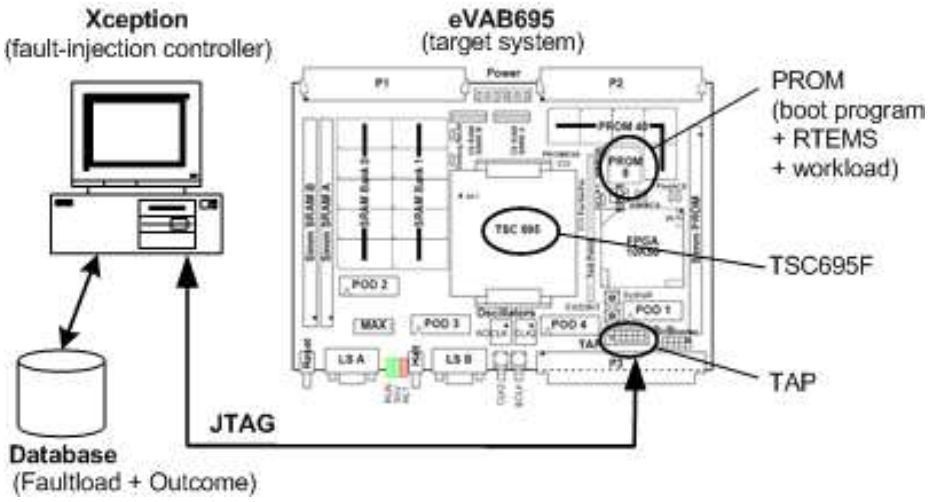

Fig. 1. The experimental testbed.

2 Experimental Methodology

Figure 1 depicts the main entities involved: the target system with the workload burned into its ROM, the fault-injection controller, the access to the target system boundary-scan port through its JTAG interface, and the database with the faultload and the outcomes.

The target hardware is the single board computer eVAB695 [7], built around the TSC695F (ERC32) 32-bit RISC space processor implementing the SPARC V7 architecture specification. This board includes 512K of radiation-hardened Flash ROM and 4096K of parity protected SRAM.

The TSC695F processor includes an integer unit (IU), a floating point unit (FPU), a memory controller (MEC) and a direct memory access (DMA) arbitrer. It also includes a watchog, two timers, an interrupt controller, one parallel and two serial interfaces. It supports on-chip debugging and boundary-scan testing accessible through the JTAG interface connected to a test access port (TAP) in the eVAB695. The TSC695F development was supported by the European Space Agency (ESA) aiming the space environment. This processor is currently on board the International Space Station and has also been adopted by the Brazilian and Chinese Space Agencies.

The application `gravity_v1.2`, is a program that calculates the trajectory of a mass (e.g. a satellite) attracted by a bigger mass (e.g. a planet) using Newton's gravity law. This workload, used extensively by ESA for testing purposes, outputs the successive satellite positions, in x-y coordinates.

The experimental workload `gravity` runs on top of the `RTEMS v4.6` kernel [8] ported to this specific hardware. RTEMS is an open source kernel designed to support applications with real-time requirements while being compatible with

open standards, namely the POSIX 1003.1b API and TCP/IP. The development of the board support package for the eVAB695 has also been supported by ESA and is available for download at the ESA web site [9].

The TSC695F was designed to stand the harsh space environment, so a number of error detection capabilities are built into the hardware, namely parity in the internal registers and data buses [10]. Thus, if we used the common single-bit fault model, this would generate easily detectable parity errors. Instead we adopted an adjacent double-bit flip error model to emulate SEU (*single event upset*) transients generated by cosmic radiation. In practice, this means that we are emulating bit-flips induced by the more energetic SEUs capable of flipping two adjacent bits.

The fault trigger is temporal: faults are injected following a uniform distribution during the boot phase, i.e. from the first clock cycle after reset to the moment the scheduler starts executing the first user application instruction. During this time frame we disturbed only processor registers (IU, FPU, control and status) since these could also emulate memory faults (e.g. erroneous values copied from ROM to RAM or into a wrong memory address). Moreover, the memory and buses are parity protected.

The injection of faults during system startup is not easily achieved with traditional techniques. Processor pin-level fault injection is currently an unfeasible option due to the ever-growing pin-hidden operations (e.g. prefetching and internal caches), as well as high clock frequencies. Radiation induced techniques, although applicable during system startup, pose known limitations of location and time control. Software induced fault injection (SWIFI) offers a level of control and efficiency hardly achievable by other techniques targeting processors. However, its dependence on specific routines that use the operating system resources while reacting to programmed breakpoints makes it unusable during system startup, the time frame focused in this study. In addition, practical SWIFI implementations have injection cores and operation (e.g. setup fault, collect data) highly dependent on the operating system design turning virtually impossible the development of an independent and pre-operating system solution. Moreover, the presence of potentially dangerous instrumentation code inside the target makes this kind of approaches less interesting for aeronautic and space applications where it is mandatory to '*test what you fly, fly what you test*'.

In recent years the boundary-scan infrastructure and its successors have been successfully used for fault injection [5, 6, 11] providing standard low level access without giving up from the flexibility recognized to software fault injection tools. Through this standard test interface port, the target processor offers an access path to its internals, allowing injecting faults even in state elements not accessible to the instruction set, like parity bits or pipeline registers. Moreover, the control of breakpoint resources and running status, both mapped to test registers, enable to program and perform fault injection and observations from the very first machine instruction executed, i.e. at any instant, an approach that is completely operating system independent.

The fault injection campaigns presented in this study were performed using an improved version of Xception™ [12, 13]. This is an automated fault injection environment designed to accommodate a variety of fault injection techniques namely the target processor on-chip debugging facilities available through the standard boundary-scan infrastructure.

The metrics collected were devised to provide a meaningful view of the target system robustness in face of faults injected during the boot phase:

Crash - the processor halted or was trapped in an endless loop. A hardware reset was needed to resume the experiments, so the watchdog timer could reboot the system.

OK/Clean - the system terminated correctly the boot sequence, the application was launched, and no errors were observed neither in the kernel nor in the application outputs. This involved a full scan of the target system memory segments (text, data, heap and stack) and processor registers after the boot and when the application terminated. We also collected the boot execution time in clock cycles.

OK/Latent - the boot sequence finished, the application was launched and no errors were observed in the outputs produced. However, internal (latent) errors which did not come to light during the experiment duration were present in the memory effectively used (errors in unused memory areas were not considered).

Wrong - the boot sequence finished and the application was launched but terminated with errors in the outputs produced.

The experiments were much simplified because the workload was being run in a controlled environment so the system state was deterministic. Through the boundary-scan port we could freeze the processor to modify and collect system data. Nevertheless, due to the low bandwidth of the IEEE 1149.1 interface each single injection run lasted more than 5 minutes, which meant that the experiments have taken several weeks running unattended.

3 The Boot Sequence

When a computer system is powered-on a long series of sensitive events occur before the target applications starts executing. These events aim at checking if hardware components are functional, configuring them, and loading software from a non-volatile storage media (ROM, flash RAM or hard disk) into main memory. Non-trivial embedded systems normally make use of a kernel which, after being loaded, must run through a complex initialization process. While this sequence of events is system specific, it usually follows these major steps (Fig. 2):

1. Power-on self test (POST) — when the processor is turned on the hardware performs a built-in self-test and some registers are initialized to a default value, namely the program counter (PC). Its default value points to a fixed

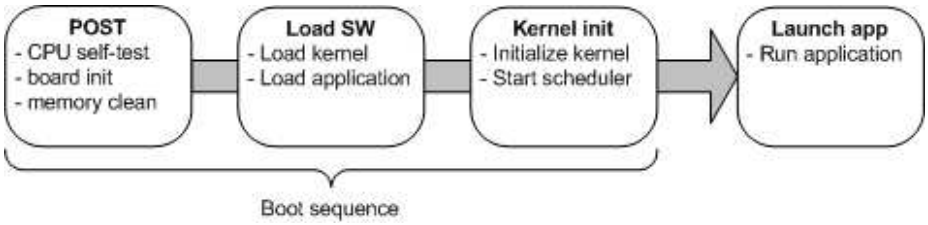

Fig. 2. Boot sequence.

memory address in ROM where is located the very first machine instruction executed by the processor. The subsequent tasks are performed under software control: configuration and status registers are initialized, interrupts are disabled and common registers (IU and FPU) are cleared. Board specific code detects the hardware configuration (e.g. the number of serial ports and the top of memory to initialize the stack), performs diagnostics to check if the basic components are in perfect condition (e.g. test and clear RAM memory), and initializes some hardware components (e.g. I/O ports).

2. Kernel and application load — as happens in most embedded systems, the kernel is loaded as an application library and the different segments (text and initialized data) of the kernel and workload are copied from ROM into the RAM areas. Usually the ROM images are compressed so the copy also involves decompressing those segments. If this software resides on disk, a loader application is first copied from ROM and then loads the kernel and workload.
3. Initialize and start kernel — the kernel data structures are initialized, the most complex parts of the hardware are configured (e.g. co-processor, if present) and the device drivers are installed. Finally, the kernel scheduler starts executing by enabling interrupts.
4. Launch application — the '`main`' routine in the user application code is entered and starts executing.

In figure 3 we present the *time*×*space* execution profile of the boot sequence for the target workload used in our experiments with *time* (clock cycles) in the horizontal axis and the *address* of instructions executed in the vertical axis.

Instead of being spread all over the address space, the memory accesses seem almost continuous (dark) horizontal bars. This indicates a tight access locality of the machine instructions executed, i.e. most of the system activity is centred in very few lines of code that are either clearing the memory or decompressing the application segments from ROM into RAM. We can see that the boot program starts executing in the ROM (lower addresses) followed by a long (about 4 million clock cycles) clearing of RAM memory. Then, the application and kernel code are copied into RAM (another 3 million clock cycles) followed by the initialized and uninitialized data areas (the 'uninitialized' data areas are effectively initialized to zero). This code fragment is loaded into RAM

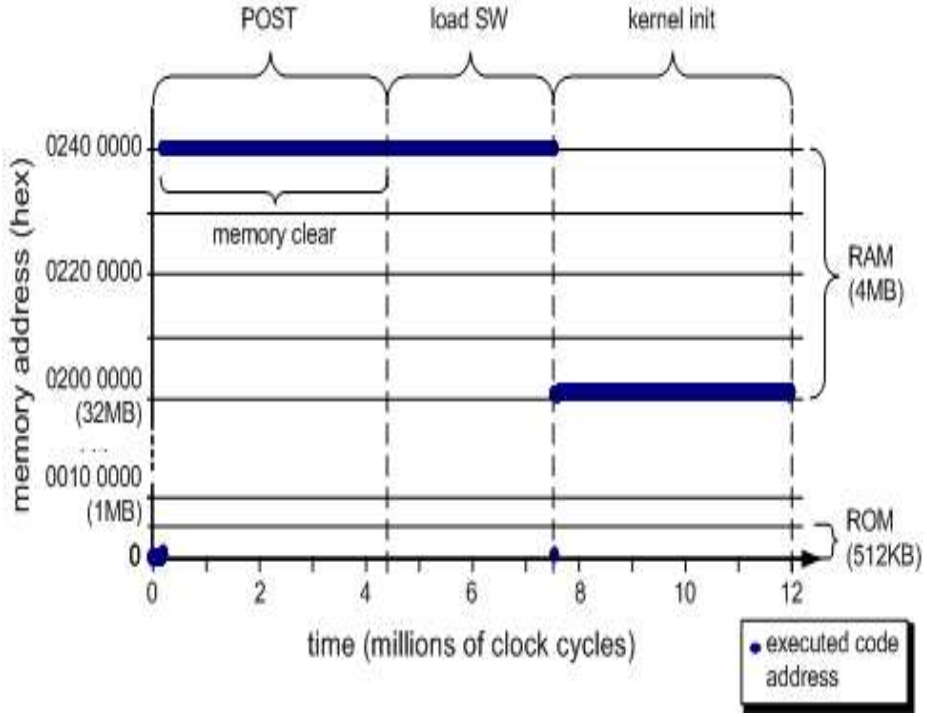

Fig. 3. Trace of instruction addresses accessed during boot.

because fetching this code directly from ROM would be much slower. This temporary area is located near the RAM top (Fig. 4) in the future stack area prior to the stack initialization, thus not conflicting with the boot operations being performed. Finally, the different kernel initialization routines are called to prepare the application launch by the scheduler (Fig. 3, 'kernel init').

The boot duration depends on the application size: the larger it is, the longer it takes to reboot. Moreover, if a large number of libraries were used, the longer it would be. In such embedded systems, services which are not required by the application are not even linked into the ROM image (e.g. if the workload did not use real arithmetic the floating point libraries would not be loaded).

In the presented testbed the kernel initialization code starts execution around the $7.500.000^{th}$ clock cycle. This means that for about 70% of the boot time the hardware is performing extremely tight code (cycles of about 10 machine instructions). As we shall see later this has a direct impact on the system's behaviour under faults.

Finally, around the $12.000.000^{th}$ clock cycle after reset, the application is started. At 20MHz clock frequency this means that 0.6 seconds are required for boot. While this seems a negligible fraction of time in missions lasting for decades, an error occurring during a reboot can have a dramatic impact on

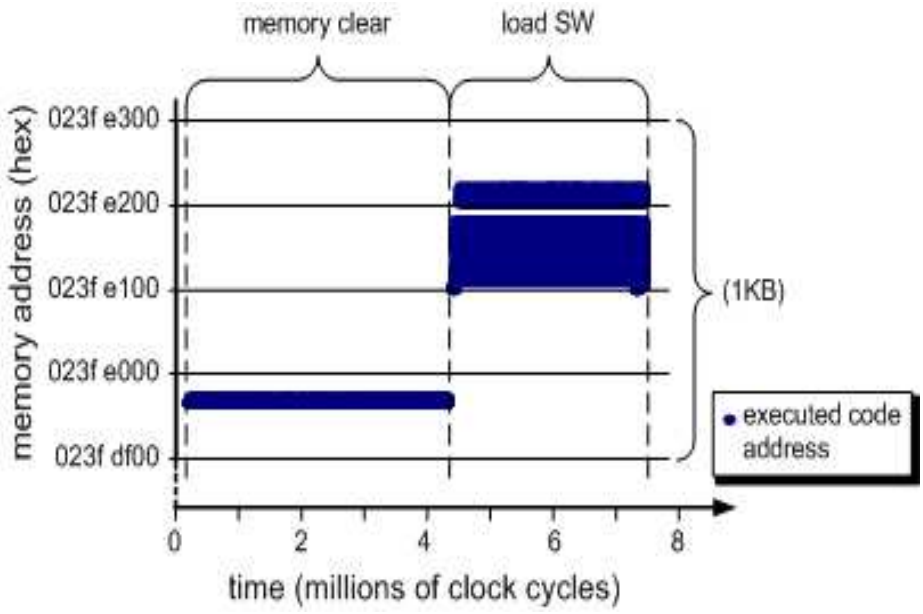

Fig. 4. The initial boot code is executed near the RAM top

dependability, since it is manipulating extremely sensitive parts of the system, such as memory (code and data segments), pointers, kernel data structures, device handlers, hardware configurations, etc. Furthermore, all this functions are executed in privileged mode, and for most of this time no error handlers are active and hardware based fault-tolerance support may not be configured yet.

4 Experimental Observations and Discussion

Table 1 presents an overview of the target system behaviour after the injection campaign involving 4997 effectively injected faults. It depicts the final system behaviour (columns) versus the system state observed when boot terminated (rows).

The most significant observation is its resilience to failure: the system either produces correct outputs (OK/75%) or no output is generated at all due to crash(25%). A residual 2 faults lead to the production of wrong outputs. It must be stressed that this behaviour is clearly distinct from what we have observed in previous research dealing with faults injected during steady-state operation in similar embedded systems (figures were around Clean(50%), Crash(48%), Wrong(2%) [14, 15]).

As would be expected the observations show that every OK/Clean outcome arises from a clean boot environment. The large number of samples where the system was unaffected (65%) indicates the presence of a significant intrinsic

Table 1. Overview of the target system behaviour.

State after boot		Final System Behaviour			
	$\sum$	**Crash**	**OK/Clean**	**OK/Latent**	**Wrong**
Clean	3250	0	3250	0	0
Corrupted	490	1	0	487	2
Crash	1257	1257	0	0	0
Totals	4997	1258 (25%)	3250 (65%)	487 (10%)	2 (0.04%)

hardware redundancy. In fact, as was referred in section 3, during about 70% of the boot time the processor is performing extremely tight code (cycles of about 10 machine instructions, using only 5 of its 256 windowed registers), checking and clearing memory (POST) and moving the data from ROM into the RAM space. This means that most of the processor resources are effectively idle and thus unused, which explains its resiliency to failure. Whenever a resourceful state machine (such as a processor) uses very few of its resources, the probability of a disturbance affecting the active functional units is reduced. This observation agrees with previous observations on the correlation of system load and the occurrence of errors [16].

Crashes are dominant from errors occurring during the transfer of the application image from ROM into RAM (Fig. 5, 'load SW'). Note that the decompression of the RAM image is parity protected, but since we are injecting adjacent double bit flip-faults, this mechanism is not enough to prevent the corruption of the memory image. It was observed that only one fault eventually leading to crash managed to reach the application entry point. This fault corrupted a global register (used as frame pointer) during the kernel initialization. The remaining 1257 faults crashed the system when the corrupted kernel code was executed, so the boot phase never terminated.

About 10% of all faults (490) lead to a corrupted system after boot termination and to the presence of (487) latent errors despite the production of correct outputs. The characterization of these errors show a prevalence of faults injected during the final phases of the boot, i.e. during the kernel initialization. These errors were resident in kernel structures which have not been used.

A most undesirable behaviour of any computer system is the production of wrong outputs without being detected by any error detection mechanism being rather preferable a crash (no outputs produced). This is known as the fail silent model [17], an assumption under which most dependable systems are designed. The fail-silent behaviour is usually associated with the evaluation of dedicated error detection mechanisms [14].

By tracing the executions that generated wrong outputs we observed that these two faults corrupted fixed data areas (static data) during the memory initialization phases. This behaviour agrees with the research performed on [15, 18] on the resiliency of errors and checkpoint corruption.

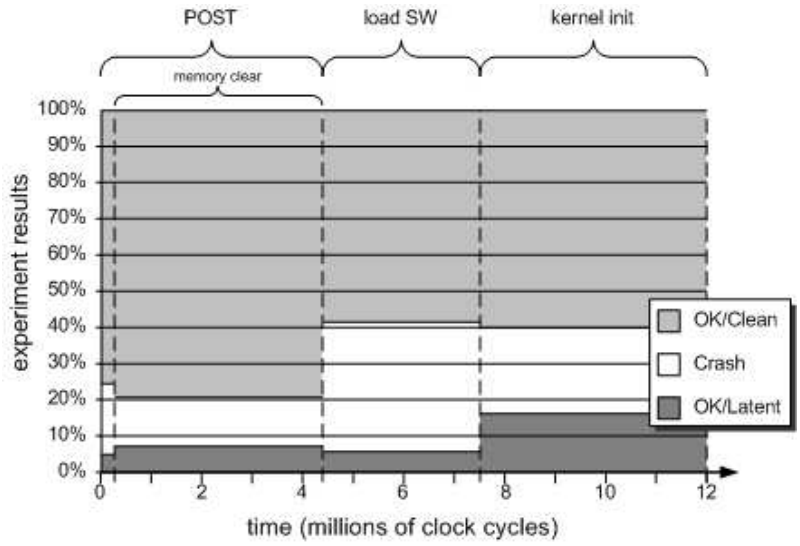

Fig. 5. Distribution of fault outcomes along the boot phases.

The analysis of the error impact versus fault profile showed a slightly higher correlation between the trigger address, i.e. what the processor was doing at the injection instant, rather than a specific target register. Obviously corrupting the PC lead to crash and the most sensitive address was at the long segment transfer routines, but beyond these exceptional cases no particular dependency was observed.

5 Tolerating Boot Errors

Based on the previous observations we shall now suggest possible ways to tolerate errors that may arise during boot.

5.1 Detection of Timing Deviations

The initial boot phase is a deterministic sequence that is not interrupted or subject to different execution threads. In fact, since interrupts are disabled, there are no external events that could cause diverse control flow sequences. The processor itself does not have any indeterministic characteristic, such as speculative execution. Interrupts and thus the scheduler are only enabled at the very end of boot. This means that we can know precisely the boot sequence duration, be it in real time or clock cycles. Since a clean environment after boot implies a correct timing, then the boot duration may be used for error detection. Effectively the observations show that about 13% of latent errors are associated with incorrect boot timing.

These observations have immediate applicability for error detection purposes: if a watchdog timer is set to the boot duration (plus a minor margin, since it is asynchronous relative to the processor clock), and a small routine at the end of boot checks the watchdog counter for an early boot termination, they will detect every crash and 13% of boot corruptions leading to latent errors (despite the fact that these would seem like false alarms since the outputs would be correct). In the current target, up to 62 out of 487 latent errors would have been detected. The two samples leading to the production of wrong outputs would not be detected since they showed a correct timing.

5.2 Detection of State Corruption

Since the boot sequence is fully deterministic, we can collect the system state in advance. Later in operation, during boot, the full system state (memory and relevant hardware configuration registers) is checked for corruption using (e.g.) a resident CRC check, and if any deviation is found a reboot is forced. The point is that in systems where there is no memory protection (as is the case) the check itself can corrupt memory areas which have already fed the CRC calculation. This suggests that –despite the additional power required and the time execution overhead– the use of ROM for the code segments and fixed data areas should be considered. Alternatively there could be some form of blocking the write accesses to such 'fixed content' areas.

It is mandatory to complement this approach with the watchdog timer referred above, since the CRC check itself may be bypassed by a control flow error. If, for some reason, this check were not executed, then the watchdog timer would expire and force a reset.

6 Conclusion

In this paper we presented an experimental study on the behaviour of an embedded real-time system under the occurrence of faults during boot. During this time frame operating system support is not yet available and traditional fault-injection techniques cannot be used. Thus, our research was based in a fault-injection approach based in the IEEE 1149.1 (boundary-scan) standard, since this infrastructure is orthogonal to the chip functionalities, and is permanently available whenever the chip is powered.

The activities performed during boot show that the system either produces correct outputs (75%) or no output is generated at all due to crash (25%). However, about 10% of the faults caused latent errors in the system despite the production of correct outputs. Only 2 faults lead to the production of wrong outputs.

The insights achieved from this study provide clues on what can be done to increase robustness during this system state in which most fault-tolerance techniques are not yet setup. The determinism of the boot, both in time and in the actions performed, indicate that the boot resiliency to failure can be significantly

increased by i) a watchdog timer finely tuned to the boot duration, ii) preventing writes to addresses with fixed contents and iii) associating a watchdog timer to a CRC check of the system memory and relevant hardware configuration registers.

Acknowledgements

We acknowledge Professor Algirdas Avižienis as the source of inspiration for this work as he remarked during the EDCC4 conference that no SWIFI tool could be used to inject faults immediately after a reset.

References

[1] J. Cunha, A. Correia, J. Henriques, M.Z.-Rela, J. Silva, *Reset-Driven Fault Tolerance*, 4th European Dependable Computing Conference (EDCC-4), Toulouse-France, October 23-25 2002, LNCS 2485, A. Bondavalli, P. Thevenod-Fosse (Eds.), Springer-Verlag Heidelberg 2002, pp. 102 - 120.

[2] J.-C. Laprie, A. Avižienis, H. Kopetz (Eds.), *Dependability: Basic Concepts and Terminology*, Springer-Verlag, ISBN:0-3878229-6-8, 268 pages, New York 1992.

[3] S. Potteck, *La conception de systèmes spatiaux*, Éditions du Schèmectif, Juillet 2001, ISBN 2-9513724-0-X (2 Tomes).

[4] IEEE Std 1149.1-2001, *IEEE Standard Test Access Port and Boundary-Scan Architecture*, ISBN: 0738129445, New York, 2001.

[5] P. Folkesson, S. Svensson, J. Karlsson, *A comparison of simulation based and scan chain implemented fault injection*, In Proc. of 28th Symposium on Fault Tolerant Computer Systems (FTCS-28), Munich, Germany, IEEE Computer Society 1998, pp. 284-293.

[6] L. Santos, M.Z.-Rela, *Constraints on the use of boundary-scan for fault injection*, in Proc. First Latin-American Dependable Computing Symposium, S. Paulo, Brazil, Oct. 2003, Lecture Notes in Computer Science, LNCS 2847, Springer-Verlag Heidelberg 2003.

[7] TSC695 Evaluation Board User Guide Manual, Rev.C 01/00, ATMEL Corp.2000 `http://www.estec.esa.nl /microelectronics /presentation/ERC32.pdf`

[8] RTEMS: *Real-Time Executive for Multiprocessor Systems* `http://www.rtems.com/`

[9] `http://www.estec.esa.nl/wsmwww/erc32/freesoft.html`

[10] J. Gaisler, *Evaluation of a 32-bit Microprocessor with Built-In Concurrent Error-Detection*, in Proc. FTCS-27, June 25-27, IEEE Computer Society 1997, pp. 42-46.

[11] P. Yuste, J.-C. Ruiz, L. Lemus, P. Gil, *Non-intrusive Software-Implemented Fault Injection in Embedded Systems*, in Proc. First Latin-American Dependable Computing Symposium, S. Paulo, Brazil, Oct. 2003, LNCS 2847, Springer-Verlag 2003, pp. 23 - 38.

[12] Xception^{TM}-*Enhanced Automated Fault-Injection Environment*, 2002, `http://www.xception.org`.

[13] J. Carreira, H. Madeira, J.G. Silva, *Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers*, IEEE Trans. on Software Engineering, February 1998.

[14] H. Madeira, J.G.Silva, *Experimental Evaluation of the Fail-silent behaviour in Computers without Error Masking*, In Proc. FTCS-24, Austin-USA, IEEE Computer Society 1994, pp. 350-359.

[15] J. Cunha, R. Maia, M. Z.-Rela, J.G. Silva, *A Study of Failure Models in Feedback Control Systems*, in Proc. DSN'2001, July 1-4, 2001, Göteborg-Sweden, IEEE Computer Society 2001.

[16] R. K. Iyer, D. Tang, *Experimental Analysis of Computer System Dependability*, Chap. 5 in Fault-Tolerant Computer System Design (ed. D.K. Pradhan), ISBN 0-13-057887-8, Prentice Hall 1996, pp. 282-392.

[17] D. Powell, G. Bonn, D. Seaton, P. Verissimo, *et. al, The Delta-4 approach to dependability in open distributed computing systems*, in Proc. FTCS18, Japan, June 1988.

[18] J. Vinter, A. Johansson, P. Folkesson, J. Karlsson, *On the Design of Robust Integrators for Fail-Bounded Control Systems*, DSN2003, ISBN 0-7695-1952-0, IEEE Computer Society 2003, pp. 415-424.