Fast and Low-Cost Instruction-Aware Fault Injection

Andreas Heinig, Ingo Korb, Florian Schmoll, Peter Marwedel and Michael Engel

Computer Science 12
TU Dortmund
D-44221 Dortmund, Germany
{firstname.lastname}@tu-dortmund.de

Abstract

In order to assess the robustness of software-based fault-tolerance methods, extensive tests have to be performed that inject faults, such as bit flips, into hardware components of a running system. Fault injection commonly uses either system simulations, resulting in execution times orders of magnitude longer than on real systems, or exposes a real system to error sources like radiation. This can take place in real time, but it enables only a very coarse-grained control over the affected system component.

A solution combining the best characteristics from both approaches should achieve precise fault injection in real hardware systems. The approach presented in this paper uses the JTAG background debug facility of a CPU to inject faults into main memory and registers of a running system. Compared to similar earlier approaches, our solution is able to achieve rapid fault injection using a low-cost microcontroller instead of a complex FPGA. Consequently, our injection software is much more flexible. It allows to restrict error injection to the execution of a set of predefined components, resulting in a more precise control of the injection, and also emulates error reporting, which enables the evaluation of different error detection approaches in addition to robustness evaluation.

1 Introduction

When designing embedded systems, resource and cost limitations often prohibit the use of hardware-based fault-tolerance mechanisms like ECC-protected memories and logic. However, according to forecasts of semiconductor experts [Int09], the trend towards lowered supply voltages and smaller structure sizes will reduce the inherent fault resilience of semiconductor devices. Thus, future semiconductor-based systems are expected to be affected by faults even under normal operation conditions. As a consequence, some form of error detection and correction (EDAC)

will be indispensable for future embedded systems. Previous research [RCV+05] has shown that software-based approaches are effective in reducing the overhead required for error correction. The overhead for error *detection* in software, however, is still comparatively high. Thus, many embedded systems will use a combination of hardware-based error detection and software-based error correction.

In order to create effective software-based fault-tolerance methods, the system designer requires methods to assess their effectiveness already in the design phase. This evaluation is commonly performed by exploring the effects of faults on the combined hardware/software system. In order to cover most of the possible faults, a large number of fault injection experiments has to be performed.

Often, a system simulator is used to perform fault injection. This approach has the advantage that faults can be injected precisely and the effects can be easily discovered by, e.g., log file analysis. However, simulations are commonly several orders of magnitude slower than execution on a real hardware system. As a result, fault injection campaigns using simulations commonly require large amounts of computing time.

Another approach uses physical effects, such as radiation or electro-magnetic interferences, to inject errors into the target system's hardware. The unforeseeable distribution of injected faults using this method has the disadvantage that neither the injection of faults nor the precise evaluation of their effects is feasible.

An approach that combines the best properties of both of these methods would be able to perform precise fault injection into live hardware systems. Several approaches to this have been discussed in previous publications. However, existing approaches either require complex hardware like FPGAs or modify the software running on the system. FPGA-based injection is inflexible, since any change to the injection properties requires a re-synthesis of the FPGA configuration. Modifying software, e.g., by inserting fault-injecting instructions into program code, has the disadvantage that the configuration used for testing the fault tolerance is different to the one deployed on the final system.

In this paper, we present a fault injection method that allows to perform fault injection into memory and registers of a running system using software running on an external, low-cost microcontroller without requiring software changes on the target system (often called system under test, SUT). In addition to fault injection, our system also supports the simulation of fault detection, which allows a system designer to jointly evaluate detection and correction approaches.

Our approach is based on the background debug facilities or On-Chip Debugger (OCD) available on a wide range of modern embedded processors. One widespread approach to communicate with the OCD is the JTAG (Joint Test Access Group) boundary scan interface, which allows to change the internal state of a system at runtime. Using JTAG, no hardware modifications are necessary. Hence, our approach is minimally intrusive[1] and easy to implement. The challenge in JTAG-

[1] Except for the intended intrusion of injecting the fault and the time required to inject the fault.

based error injection is to insert faults quickly, in order not to disturb the system execution for too long, and precisely, i.e., injecting faults in a given context.

The approach presented in this paper is able to inject faults into memory and registers of a running system while requiring only a small overhead. Even though error injection via JTAG will frequently stop the target system for a few cycles, our method is considerably fast and increases the runtime of the SUT by only about 2%. By using software to perform fault injection, injection properties are easily configurable with short turn-around times. The use of a small stand-alone microcontroller for fault injection also allows the use in mobile systems such as robots.

We specifically implemented the fault injection for an ARM926-based target system using a Marvell Kirkwood 88F6281 SoC. The system is running the RTEMS operating system and a H.264 video decoder as fault injection targets. The system performing the fault injection is an NXP LPC1768 ARM Cortex-M3 SoC. However, the fault injection methods and software are sufficiently general that adaptations to different injection hosts and targets should be possible with reasonable effort.

To summarize, the main contribution of this paper are as follows:

1. We present a microcontroller-based fast and flexible solution to inject faults into running embedded systems.
2. Our approach neither requires hardware modifications nor expensive test equipment.
3. We introduce instruction-aware fault injection to model real-world error reporting and to increase the precision of fault injection.

The rest of this paper is organized as follows. Section 2 gives an overview of related work. In Section 3 and Section 4, the fault model and the implemented injection routine are presented. A short overview of JTAG is given in Section 5. An evaluation of our fault injection approach is contained in Section 6. Finally, we conclude the paper in Section 7.

2 Related Work

Many approaches to fault injection have been described in the literature so far. Here, we give an overview of typical injection approaches as well as a more concise discussion of the methods closely related to the approach presented in this paper.

Hsueh et al. developed a classification of different fault injection techniques in [HTI97]. A global distinction is made between hardware and software fault injection techniques.

2.1 Hardware-Based Fault Injection

Hsueh et. al define hardware fault injection as a technique that uses additional hardware to inject faults into the target system. The injection can be with direct physical contact, e.g., using pin-level probes and sockets [ACL89][KF95], or without contact by exposing the circuit to a particle beam [KF95][VKC+92][ELDF92], using lasers [PLF03] or to electromagnetic inferences [KF95][VCG+05]. Especially the contactless methods have the ability to inject faults in a realistic way. An alpha particle emitted by the particle beam, for example, will hit the circuit in a random location, which reflects the stochastic nature of faults quite accurately.

Hardware-based solutions have the advantage that no extra CPU time on the system under test is required to inject the fault. Hence, its real-time behavior is not influenced by injection.

However, hardware-based injection brings along a number of disadvantages. Either the hardware of the target system has to be modified or special expensive equipment, e.g., a radiation source or an ion-beam injector, is required. The statistical nature of the injection results in a low precision of fault injection, since neither the exact location nor the exact point in time at which the injection should take place can be controlled.

2.2 Software-Based Fault Injection

Software-based fault injection methods resolve some of the problems discussed above. However, depending on the specific approach used, different problems may arise, which we discuss below.

Different ways to inject faults exist. In an off-line approach, selected instructions are modified at compile time. This kind of injection generates a software image with hard-coded permanent faults. Due to these specific modifications, faults are only activated when this particular code location is executed. An approach very similar to code modification is code insertion. Here, additional instructions are added to the target program that allow fault injection to occur before a particular instruction. A further disadvantage is that the configuration used for testing the fault tolerance is different to the one deployed on the final system.

A very simple online approach to inject faults into a process is to use a second thread randomly injecting memory faults [HESM10]. The main drawback of this approach is the limitation to the running process. It is not possible to inject faults into the operating system or into memory not owned by the process.

In order to randomly inject faults into any process of the target application, an interrupt-based method can be used [VRE00]. One of the simplest ways to do so is to use a timer circuit which can generate external interrupts based on a specified time interval. A similar approach is based on exceptions or traps which can be

triggered by software. When the trap is activated, control is transfered to the fault inject code. A slightly different approach is used by Xception [CMS98] or FlexFi [BRR99], which use the debugging features of modern processors. The basic idea is to insert a breakpoint to stop the processor when accessing a specific memory area. On access, the target processor is stopped and the host debugger can insert a fault. However, those methods have the drawback that the operating system has to be modified. Either fault injection code has to be linked to the interrupt handler vector, or extra debug code has to be inserted.

A further approach which also uses debug capability is FIMBUL [FSK98]. Here, fault injection is used to test error correction and detection of the Thor CPU. When a breakpoint condition is fulfilled, a fault will be injected. FIMBUL has the drawback that no error reporting is implemented and that only a single bit flip is performed per run.

2.3 Hybrid Fault Injection Approaches

Hybrid fault injection approaches comprise methods of software- and hardware-based fault injection. One candidate in this area is the usage of the On-Chip Debugger (OCD) unit via a JTAG link. Like Xception, this approach also uses the debugging capabilities of the target processor. In contrast, no modification of the target software is required, since the OCD is now controlled externally via JTAG. A possible implementation of this JTAG approach was presented in [PGLOGVE07]. The authors injected faults into an ARM7TDMI processor with a JTAG injection system running on an FPGA. With 2 ms fault injection time the proposed solution is very fast. The main drawbacks of the FPGA solution are the low flexibility and low adaptability. Due to the complexity of FPGA programming, it is very difficult to implement complex fault modes or to exchange the fault model. Furthermore, error reporting is not implemented. To simulate real-world error reporting, an instruction decoder running on the FPGA would be required to keep track of currently used hardware components. Implementing such a decoder for the FPGA is generally possible. However, due to the high complexity of FPGA programming, such a solution will be very expensive.

Nevertheless, using JTAG for fault injection seems to be a very promising approach since it is possible to achieve short injection times while not being required to modify the target system's hardware or software. To compensate for the drawbacks of the FPGA-based solution discussed above, we decided to use a microcontroller to implement the JTAG host controlling the OCD.

3 Fault Model

In this paper we focus on *transient faults* in main memory as well as in the processor's general purpose registers. Transient faults are single-shot phenomena. They affect a component of a system at an unpredictable moment in time and only persist until a status change of the affected component. For example, when an affected memory cell gets overwritten, a transient fault that affected this cell will cease to exist. In contrast, a *permanent fault* would persist.

The memory ranges and registers in which faults are injected are freely configurable by the designer. We call the hardware components with enabled fault injection *low reliable silicon* and the hardware components without fault injection *high reliable silicon*. With a suitable mapping, the developer can now divide the software stack into two parts. The part which has to be tested under fault injection is mapped to low reliable components and the rest is mapped to high reliable hardware components. This allows, e.g., to simulate ECC-protected reliable memory ranges without requiring the related ECC hardware.

Since our main research focus is on error correction, the error detection is modeled within our fault model. The error reporting shall be as realistic as possible. Typically, real-world hardware with error reporting capabilities, like memory protected by parity bits, will report an error only if the affected cell is used. Only on access, the memory controller computes the checksum of the stored data and checks the stored parity information. When a fault is detected, the processor will be signaled. Most likely, the memory controller cannot determine the exact location of the faulty bit. Otherwise the correction will be very easy – just flip the bit again. Depending on the implemented granularity of the memory controller's check routine, the main processor is notified of an error in a memory range, e.g., with the granularity of a cache line. Within this range, one or more bits may be affected by transient errors.

In our fault model we intend to implement this error reporting behavior exactly. Therefore, we simulate an error reporting device which raises an interrupt when a fault is detected. This device provides three memory mapped registers for reporting details of the fault. The first register describes the fault type (either register or memory fault). In the case of a register fault, only the second memory mapped register is used and contains an identifier for the affected register. When a memory fault is detected, the second memory mapped register contains the base address of the memory range and the third register holds the size of the memory range. Within this range, one or more faults may exist. To reflect the previously mentioned granularity issue, we define the minimum range to be one word. Hence, even byte fetches will be reported as a fault in a whole word.

In the following chapter, we provide an overview of the implementation of the fault injection based on the fault model.

4 Fault Injection Procedure

The basic idea when using a debug unit – whether on-line or not – is to periodically interrupt the normal operation of the target processor to inject faults. The most challenging part, however, is the implementation of error reporting, since we intend to simulate the behavior of real world hardware. As mentioned earlier, hardware typically will report an error if the requested resource is affected. This results in two challenges: (1) we have to know which components are used, (2) we have to know which components are affected.

To resolve the first problem, we decode the instruction executed on the target processor when interrupted by the OCD. Taking the content of the registers into account, it is also possible to compute the source/target address of a load/store operation, respectively.

The solution of the second problem could be a map which records all injected faults. Such a map would allow to check whether the target system accesses a faulty component or not. In the first case, a fault is reported. Using such a map, however, would have a huge disadvantage: we would need to decode and check every executed instruction of the target system. On each write access we would have to clear fault markings, since overwriting would eliminate all transient faults in the written range. Whereas on each read access we would have to check for error markings. Decoding and checking every instruction, however, means running the target system in single step mode. Hence, the execution speed of instructions of the target system is defined by the execution speed of JTAG, the OCD, and the algorithms executed on the injection hardware.

To deal with this problem, we turn this order around. Instead of reporting previously injected faults when they are accessed, we inject faults when they can be reported. In other words: *We inject faults aware of the instruction currently executed.* Hence, after decoding, we immediately inject faults directly into low reliable components used by the instruction.

The transient fault injection procedure is depicted in Fig. 1. All operations involving communication via JTAG are shaded dark. The whole injection procedure is repeated periodically at each timer interrupt of the JTAG host. The time between two interrupts is freely configurable.

The first step is to stop the SUT by sending a debug request via JTAG. At this point, the OCD takes control over the CPU. Hence, no instructions of the SUT's application are executed anymore until in the last step target resume is called. To keep jitter as low as possible, it is very important that all steps between target halt and target resume are executed as fast as possible.

Directly after interrupting the target system, we fetch the register contents to completely decode the interrupted instruction. Based on the components used and decisions of the fault model, we inject a memory or register fault, or no fault at all.

To inject a memory fault, the decoded address is used as base. Depending on the

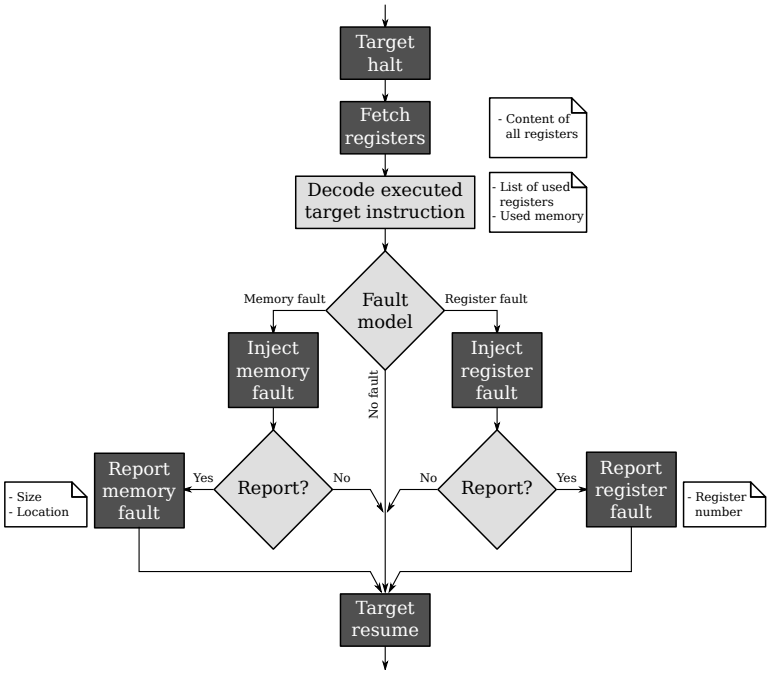

Figure 1: Fault Injection Procedure

memory access granularity of the target instruction, the range in which bits[2] are flipped is specified. In our model, the minimum range is the word size. In cases of half word or byte loads, the memory address will be aligned to the closest lower possible word aligned address and the size will be set to word length.

A register fault is injected by flipping bits in one of the used registers of the target instruction. Register bit flips are always performed on the full register, regardless of the instruction's access pattern.

Depending on the fault model the injected fault can be reported to the SUT via interrupt or not. In the case of an injected memory fault, the address and the injection range are reported. In the register fault case, the affected register is reported.

5 JTAG

The IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture [JTAG], more commonly known as JTAG - from the Joint Test Action Group that

[2]One or multiple bits

created the standard - is a four-wire serial interface. It was originally designed to facilitate automated connectivity tests of populated PCBs, but thanks to its extensible design it is now also commonly used as a hardware interface for in-circuit programming and debugging of code running on microcontrollers and SoCs.

Typically, JTAG consists of the following components: (1) the Test Access Port (TAP), also called JTAG port, (2) the TAP controller, and (3) several shift registers. The TAP controller is a state machine controlling the test logic. It is controlled by the clock (TCK) and mode select (TMS) signals. By moving through the states, the JTAG host can select one of several shift registers. For reading or writing of the registers, the data out (TDO) or data input (TDI) signals are used. The shift registers are also clocked by TCK, but only when the state machine is in one of two specific states. The first state selects the instruction register used to address a data register in the second state. Within those two states the state machine is only sensitive to changes on TMS. Hence, bits can be shifted at the frequency of the clock signal without any delays imposed by additional state changes. However, this creates a challenge: TMS has to signal a change of state while the last bit is moved into the shift register.

The shift registers can be used to build a system that can arbitrarily access chip internals. The JTAG standard itself specifies only an interface used for boundary scans - direct access to the chip pins to check for short and open circuits on a PCB - but proprietary extensions can be implemented as well. The bit width of the registers is not specified by the standard. In our case it varies from 4 to 67 bits.

Our first approach to inject faults was to use OpenOCD [OpenOCD] and a USB-JTAG adapter to interface a PC and the SUT. OpenOCD is an open source JTAG debugger software supporting a huge variety of different architecture and OCDs. Scripting OpenOCD is supported via Tcl. We created injection scripts which regularly stop the target CPU to modify a value in memory. Due to the overhead imposed by USB transfers, those operations required 38.90 milliseconds every time the SUT was accessed. This time is far too long to meet our requirements. Hence, we decided to develop our own microcontroller-based JTAG implementation.

6 Evaluation

6.1 Experimental Setup

Our experimental setup is depicted in Figure 2. The system under test is a TK71 Development Board [TK71]. The attached Marvell Kirkwood 88F6281 Processor [Marvell] is clocked at 1.2 GHz. It is an implementation of an ARM926EJ-S processor compliant with the v5TE architecture, as published in the ARM Architecture Manual [ARM]. Our system contains 256 MiB of DDR2 SDRAM and 128 MiB of NAND Flash.

The fault injection is implemented on an NXP mbed LPC1768 rapid prototyping

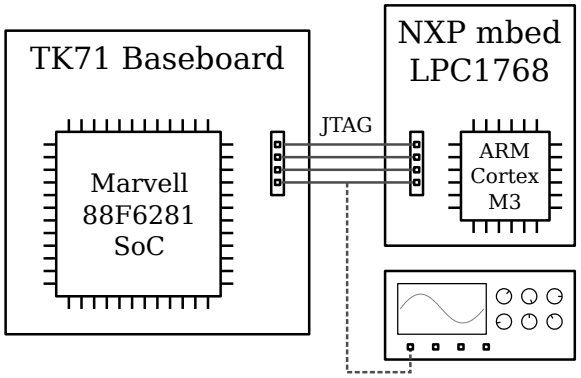

Figure 2: Experimental setup

board [mbed]. The microcontroller is clocked at 96 MHz and uses an ARM Cortex M3 Core. 32 kiB RAM and 512 kiB Flash are available for programming. We refer to this board as injection board or JTAG host.

Both boards are connected to each other via JTAG. To measure the time for fault injection, an oscilloscope is attached to the JTAG clock line.

We implemented the fault injection as described in the previous chapters. Additionally, we implemented a console interface for our JTAG host to control the memory fault injection interval and the register fault injection interval during runtime. Error reporting can be switched on/off.

As software load we execute an H.264 video decoder on the SUT as described in [HESM10].

6.2 Performance

To evaluate the performance of our approach we measured every path of the fault injection procedure. The measured times are shown in Table 1.

The fastest path, obviously, is when no fault is injected. This can be the case if the interrupted instruction only uses high reliable silicon or the desired low reliable component was not used.

Compared to injecting a register fault, injecting a memory fault requires additional read and write operations via JTAG. The read fetches the memory range in which bits have to be flipped. After flipping, the modified values have to be written back.

As mentioned earlier, OpenOCD requires 38.90 ms to perform a single fault injection with reporting. However, decoding the instruction was not implemented there. Compared to OpenOCD, our solution is more than 8.5 times faster.

Used path	Reporting?	Time [ms]
No fault	–	1.55
Memory fault	yes	4.48
Memory fault	no	2.64
Register fault	yes	3.82
Register fault	no	1.94

Table 1: JTAG Process Times

Far more interesting is the comparison with the FPGA-based JTAG implementation of [PGLOGVE07]. For injecting one memory bit flip the authors needed 2 ms. To be comparable, we turned off error reporting and instruction decoding of our solution. Since instruction decoding is turned off, we injected a fault in a random location inside the memory. With an injection time of 2.24 ms we nearly reached the speed of the FPGA-based solution.

What does the injection speed imply for our application? Let us assume that we intend to inject one memory fault per second with enabled error reporting. Due to our requirement to inject faults only at points in time when the hardware will report them, we need to find a memory operation using low reliable silicon. Our experiments have shown that in the case of our H.264 decoder application we have to check ten instructions in the worst case until we find a suitable spot. Hence, we need nine checks à 1.55 ms and one injecting step à 4.48 ms. In total, we require 18.430 ms to inject one fault per second. In other words, we use about 2% of the target CPU time to inject the fault.

6.3 Limitations

Using the OCD unit limits the possible components in which faults can be injected. In fact, injection is possible on every component writable by the CPU using normal machine instructions. This includes the register file and the memory. Theoretically, peripheral devices can also be accessed if they are memory mapped. However, the JTAG host then has to know the semantics of the memory mapped device registers to avoid rendering the system useless. Flipping the wrong bit in the power controller, for example, can turn off the system.

With a JTAG-based approach, injection of permanent faults is either impossible or incurs a high overhead. Every instruction has to be tracked and decoded to check whether an access to a permanent faulty bit is performed. This is equivalent to single-stepping the processor. Hence, the performance requirement of the target system cannot be achieved anymore. If injection of permanent faults is needed, a system simulator will be the better choice.

A further limitation is the fault coverage. Due to our instruction-aware injection,

we cannot give any guarantees whether we cover the whole memory and register space or not. In the one extreme, a program which idles most of the time is likely to never experience a fault injection. A `while(1)` loop in the idle body neither requires a register nor a memory transaction. The other extreme is a program permanently using the same memory range. With a very high probability a huge amount of memory faults are injected in this range only.

In any case, we can cope with all the limitations very well, since we are only interested in transient faults in memory and the register file. Idle times of our H.264 decoder are short and the decoder uses a wide range of memory.

7 Conclusions

In this paper we presented a very fast and highly flexible method to inject faults into a target system supporting on-chip debugging over JTAG. Our method does not require physical changes of the target system. Since standard C is used as programming language, the fault injection model can be easily adjusted and extended.

Our technique injects only faults matching the currently executed target instruction. Therefore, we completely decode the instruction to retrieve all used registers and memory cells. We inject faults only in the components used by the instruction. Together with enabled error reporting, this perfectly simulates hardware which reports faults on access.

Acknowledgments

This work was supported by the German Research Foundation (DFG) Priority Programme SPP1500 under grant no. MA-943/10.

Availability

To enable other researchers to employ and evaluate our fault injection, the source code is available online at `http://ls12-www.cs.tu-dortmund.de/daes/forschung/dependable-embedded-real-time-systems` or `http://www.andreasheinig.de/permalink/09131`

References

[ACL89] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 348–355, June 1989.

[ARM] ARM Architecture Reference Manual. *Online:* `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406b/`, visited 2013-04-19.

[BRR99] Alfredo Benso, Maurizio Rebaudengo, and MatteoSonza Reorda. FlexFi: A Flexible Fault Injection Environment for Microprocessor-Based Systems. In Massimo Felici and Karama Kanoun, editors, *Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 323–335. Springer Berlin Heidelberg, 1999.

[CMS98] J. Carreira, H. Madeira, and J.G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *Software Engineering, IEEE Transactions on*, 24(2):125–136, February 1998.

[ELDF92] R. Ecoffet, M. Labrunee, S. Duzellier, and D. Falguere. Heavy ion test results on memories. In *Radiation Effects Data Workshop, IEEE*, pages 27–33, July 1992.

[FSK98] P. Folkesson, S. Svensson, and J. Karlsson. A Comparison of Simulation Based and Scan Chain Implemented Fault Injection. In *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, FTCS '98, Washington, DC, USA, 1998. IEEE Computer Society.

[HESM10] Andreas Heinig, Michael Engel, Florian Schmoll, and Peter Marwedel. Improving Transient Memory Fault Resilience of an H.264 Decoder. In *Proceedings of the Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia 2010)*, Scottsdale, AZ, USA, October 2010. IEEE Computer Society Press.

[HTI97] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[Int09] International Technology Roadmap for Semiconductors (ITRS). International Technology Roadmap for Semiconductors, 2009 Edition, Executive Summary, `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf`, 2009.

[JTAG] IEEE Std. 1149.1 - Standard Test Access Port and Boundary-Scan Architecture. *Online:* `http://grouper.ieee.org/groups/1149/1/`, visited 2013-04-19.

[KF95] Johan Karlsson and Peter Folkesson. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *International Working Conf. on Dependable Computing for Critical Applications*, pages 267–287, USA, September 1995. IEEE Computer Society Press.

[Marvell] Marvell Kirkwood series. *Online:* `http://www.marvell.com/embedded-processors/kirkwood`, visited 2013-04-19.

[mbed] NXP mbed LPC1768 rapid prototyping board. *Online:* `http://mbed.org/handbook/mbed-NXP-LPC1768`, visited 2013-04-19.

[OpenOCD] Open On-Chip Debugger. *Online:* `http://openocd.sourceforge.net`, visited 2013-04-19.

[PGLOGVE07] M. Portela-Garcia, Celia Lopez-Ongil, M. Garcia-Valderas, and L. Entrena. A Rapid Fault Injection Approach for Measuring SEU Sensitivity in Complex Processors. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 101–106, July 2007.

[PLF03] V. Pouget, D. Lewis, and P. Fouillat. Time-resolved scanning of integrated circuits with a pulsed laser: application to transient fault injection in an ADC. In *Instrumentation and Measurement Technology Conference, 2003. IMTC '03. Proceedings of the 20th IEEE*, volume 2, pages 1376–1380, May 2003.

[RCV+05] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243–254, 2005.

[TK71] TK71 Development Board. *Online:* `http://www.karo-electronics.com/tk71.html`, visited 2013-04-19.

[VCG+05] F. Vargas, D.L. Cavalcante, E. Gatti, D. Prestes, and D. Lupi. On the proposition of an EMI-based fault injection approach. In *On-Line Testing Symposium, 2005. IOLTS 2005. 11th IEEE International*, pages 207–208, July 2005.

[VKC+92] R. Velazco, S. Karoui, T. Chapuis, D. Benezech, and L.H. Rosier. Heavy ion test results for the 68020 microprocessor and the 68882 coprocessor. *Nuclear Science, IEEE Transactions on*, 39(3):436–440, June 1992.

[VRE00] R. Velazco, S. Rezgui, and R. Ecoffet. Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection. *Nuclear Science, IEEE Transactions on*, 47(6):2405–2411, December 2000.