Fault Detection Capabilities of an Enhanced Timing and Control Flow Checker for Hard Real-Time Systems

Julian Wolf, Bernhard Fechner and Theo Ungerer
University of Augsburg, Germany
Emails: {wolf, fechner, ungerer}@informatik.uni-augsburg.de

***Abstract*—Dependability and robustness are essential requirements of embedded systems. It is necessary to develop and integrate mechanisms for a reliable fault detection. Regarding the context of hard real-time computing, such a mechanism should also focus on the correct timing behavior. In this paper, we present results of the fault detection capabilities, i.e., the fault coverage and detection latencies, of a novel timing and control flow checker designed for hard real-time systems. An experimental evaluation shows that more than 65 % of injected faults uncaught by processor exceptions can be detected by our technique – at an average detection latency of only 22.1 processor cycles. Errors leading to endless loops can even be reduced by more than 90 %, while the check mechanism causes only very low overhead concerning additional memory usage (15.0 % on average) and execution time (12.2 % on average).**

Keywords-Control flow checking; timing correctness; reliability; embedded processors; hard real-time computing

I. Introduction

Most deployed systems in safety-critical areas, like the automotive and aerospace domains, are hard real-time computing systems. They must provide analyzable timing behavior, because missing a deadline potentially causes catastrophic consequences. The primary goal is not to optimize the average performance, but to provide analyzability and to determine timing guarantees [20]. If an embedded system is intended for safety-critical applications, designers must also guarantee that soft errors, e.g., caused by transient faults, have negligible impact on the execution behavior. It is necessary to integrate reliable error detection mechanisms with low detection latency enabling an immediate reaction to any misbehavior and possibly the execution of a fall-back solution within the required deadlines.

In this context, we focus on errors occurring in the control path of an embedded hard real-time processor, i.e., errors causing timing or logical divergence from the proper control flow. Fault injection studies show that up to 77 % [19] of errors occurring in a computer system are control flow errors. Regarding system errors caused by transient, non-reproducible faults, an on-line error detection mechanism is the only feasible solution to detect such errors.

In [23] and [24], Wolf et al. provide a detailed description of a novel timing and control flow check mechanism. The approach extends fine-grained on-line timing checks for hard real-time systems by a lightweight control flow monitoring technique. The instrumentation of application code at compile-time is combined with a small hardware check unit connected to the core verifying the correctness at run-time.

In this paper, we enhance this approach by additional checks of a lower timing bound. Moreover, we particularly focus on the fault detection capabilities of the check mechanism. A fault injection study based on automotive benchmarks provides additional results showing the main benefits of the approach, i.e., a wide coverage of possible soft errors resulting in a reduction of critical system failures, very low fault detection latencies, and low memory and execution time overhead.

This paper is organized as follows: Section II summarizes related work in the field of on-line checking techniques. Our proposed method for temporal and logical control flow monitoring is presented in Section III. Subsequently, Section IV shows details on implementation issues. The results of fault injection experiments are presented in Section V. Finally, Section VI concludes this paper and gives an outlook to future work.

II. Related Work

Several methods for control flow checking – neglecting timing correctness – have been proposed during the last decades, implemented in hardware or software. Accordingly, these approaches either introduce additional hardware, like a watchdog processor [12] performing reliability checks during run-time [11], [15], [19], [22], or they add supplementary code on software-level to perform monitoring operations [1], [8], [14], [18]. However, both alternatives have benefits and drawbacks as well: While hardware-based approaches usually provoke high complexity for the integration into a system, their advantage is a good average performance due to less overhead concerning memory usage and execution time. Moreover, most of these techniques do not require changes in the executed application. Software-based approaches on the other side are easy to integrate, but cause significant overhead. Also, it is needed to add redundant information to the application source code, given that it is available. A solution for this dilemma can be the usage of a *hybrid* detection technique [4], [17], combining benefits of both hardware- and software-based approaches.

 ISBN: 978-1-61208-233-2

Figure 1. Temporal instrumentation of basic blocks

On the other hand, Paolieri and Mariani [16] introduce a special hardware unit to support timing correctness at system level. The developed *timing-aware coverage monitor unit* is CPU-independent, but requires timing footprints of the running task. However, this approach focuses mainly on timing errors caused by a multi-threaded usage of commonly used resources, but not on transient faults. The intention of this technique is only to guarantee timing correctness while completely neglecting logical aberrations from the proper control flow.

III. Detection Mechanism

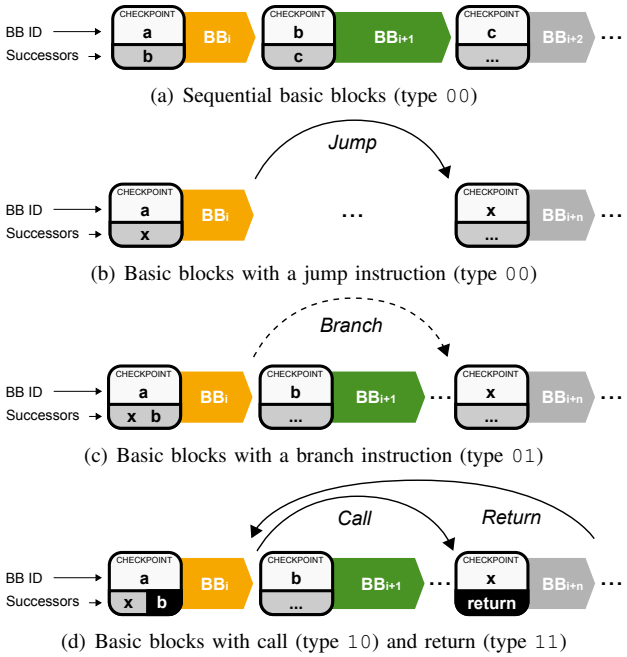
Our hybrid timing and control flow checking mechanism consists of two phases: In an *off-line* phase, the safety-critical application is split into basic blocks (BB), i.e., sequences of instructions in which the execution always begins at the first and terminates at the last instruction. These blocks are analyzed and hardened with instrumented checkpoints in the object code. In an *on-line* phase, a connected hardware check unit reacts to the inserted checkpoints during execution. If the program flow does not correspond to the instrumented information, an error is signaled.

We separate the description of our technique into two parts: Firstly, we explain the instrumentation and checking mechanism only for timing errors occurring in the control flow. Secondly, the additional part focusing on the detection of logical control flow errors is presented.

A. Temporal Control Flow Monitoring

After splitting the code into basic blocks, we add checkpoints at the beginning of each block containing information about its timing behaviour. In detail, this timing information consists of a lower bound symbolizing the *Best-Case Execution Time* (BCET) estimate and an upper bound, the *Worst-Case Execution Time* (WCET) estimate (see Fig. 1).

In the on-line phase, a specific hardware check unit transfers the timing values to defined registers, as soon as a checkpoint is reached. The register value symbolizing the WCET is decremented at each following processor cycle. When the next checkpoint is reached, the register is updated by the next WCET value. Therefore, we can assume a timing error, if the register is below zero. In this case, a basic block required more cycles than the WCET analysis had computed off-line. For checking the minimum execution time, a counter value is set to zero at the beginning of each basic block and is incremented at each following processor cycle. If the counter value is lower than the instrumented BCET bound when reaching the following checkpoint, a timing error occurred.



(a) Sequential basic blocks (type 00)

(b) Basic blocks with a jump instruction (type 00)

(c) Basic blocks with a branch instruction (type 01)

(d) Basic blocks with call (type 10) and return (type 11)

Figure 2. Logical instrumentation of basic blocks

B. Logical Control Flow Monitoring

If an application is split into basic blocks, their sequence during execution is analyzable. We annotate each basic block with a unique identifier (ID), which is added to the checkpoint containing the timing bounds. In order to enable a fast and easy check during run-time, we develop a technique to explicitly signalize successors: Along with each ID, we store the pre-calculated successor ID or two possible successor IDs of this basic block. So, the hardware checker compares in the on-line phase, if an actually executed basic block is an allowed successor. To give a better understanding, we regard each variation of the control flow, according to Fig. 2:

- In the *sequential* case (see Fig. 2(a)), we add to each checkpoint the ID of the basic block itself and the ID of its follower.
- In case of an unconditional (direct) *jump* instruction at the end of basic block BB_i in Fig. 2(b), the signalized successor of BB_i has to be updated accordingly.
- If a basic block ends with a *branch* or *loop* instruction, we cannot distinguish off-line which path will be taken during execution. So, basic block BB_i in Fig. 2(c) contains the IDs of both basic blocks BB_{i+1} and BB_{i+n} in a list of successors.
- Fig. 2(d) shows the instrumentation of *calls* and *returns*. In this example, BB_{i+n} is a function, which is called from BB_i. First, we add the ID of BB_{i+n} as the only allowed successor. Moreover, we append the basic block, which should be executed after the function's return (in this example BB_{i+1}). Within the function, it is sufficient just to signal the return. This instrumenta-

 ISBN: 978-1-61208-233-2

tion mechanism also works properly for nested function calls, if we introduce a stack memory to save multiple return IDs.

Since coding guidelines [5] for safety-critical hard real-time systems forbid the usage of indirect jumps and recursion due to problems concerning analyzability, we can neglect these issues in our context.

The hardware check unit, which becomes active as soon as a checkpoint is detected during run-time, is enhanced to interpret the instrumented values. The check unit has to verify, if the current ID corresponds to the signalized successor(s). Furthermore, it must save the current values for the checking progress when the application reaches the next checkpoint. We have to provide memory for storing at most two successors and a stack memory for function calls, which is dependent on the degree of function nesting.

IV. Implementation Issues

To evaluate strengths and weaknesses, we provide a tool for code instrumentation, which is integrated into the compilation process to enhance the assembly code of an application by checkpoints. This output is assembled and linked in order to get a binary that can be executed on a processor extended by a hardware check unit. The instrumentation could also be implemented on binary level. This might be useful, if application sources are not available, which can be neglected in our case.

A. Evaluation Platform

As a baseline for the execution we use the real-time capable multithreaded two-way superscalar CarCore processor [13]. The architecture of the CarCore is binary compatible to the Infineon TriCore [10], which is a commonly used microcontroller in safety-critical applications of the automotive industry. Up to two instructions per cycle can be assigned to its two pipelines (an *address* and a *data* pipeline) consisting of a *decode*, *execution* and *write back* stage. Both pipelines share the stages *instruction fetch* and *schedule* in the front part of the processor. The processor works in-order; instructions in both pipelines can be executed in parallel if an address instruction directly follows a data instruction.

Our simulations are executed on a cycle-accurate CarCore SystemC model, which exactly implements the timing behavior of the processor. This enables a measurement and comparison of realistic execution times of different applications.

B. Integration of Checkpoints

To handle the described methodology for timing and control flow checking, we need to enhance the application code by *BCET* and *WCET* estimates of a basic block, its *ID* along with two potential *Successor IDs*, and a field *Type* signalizing the required compare operation to the hardware check unit (see Fig. 2 for type values). However, we can avoid an explicit prediction of a second ID by a constraint on the assignment of basic block IDs: Each succeeding basic block in the assembler code should get an ID incremented by one (compared to its predecessor). By this, the second possible successor in a branch is always the ID of the block itself, incremented by one. Equally, we can implicitly define the return ID in case of a function call. The ID of the basic block of the return target is always the calling basic block's ID incremented by one.

Type	ID	Succ. ID	BCET	WCET
2 Bit	9 Bit	9 Bit	6 Bit	6 Bit

Figure 3. A 32 bit checkpoint

The timing bounds are computed with an analysis tool, which accumulates execution times of single instructions. The instrumentation tool can be enhanced by connecting a WCET tool providing less overestimation like the static WCET tool OTAWA [3], which also works on the baseline of basic blocks.

Focusing on low overhead, we choose an overall checkpoint bit width of 32 bit. This allows writing a checkpoint value to a 32 bit register of the CarCore processor, which has to be read by the hardware check unit. The bit mask displayed in Fig. 3 shows our implementation of a checkpoint: We need 2 bit for the declaration of the checkpoint type and 9 bit both for encoding the ID and the successor ID. The remaining 12 bits are used for the integration of the BCET and WCET values. This configuration allows a representation of 512 unique basic block IDs, which is sufficient regarding our evaluations.

To enable the check mechanism during run-time, it is necessary to add processor instructions, which trigger the check mechanism during execution. The CarCore processor provides special registers, called *Core Special Function Registers* (*CSFRs*) for hardware extensions. So, we implement the check unit to be triggered, as soon as an `MTCR` (move to core register) instruction on a specific checkpoint register is executed. For each checkpoint, we need three instructions: first, we write the checkpoint value into a data register (requires two instructions), then we call `MTCR` (one instruction) to copy the value to the special register. Since `MTCR` is an address instruction immediately following after a data instruction, the CarCore can execute two of these instructions in parallel, which minimizes execution time overhead.

C. Hardware Check Unit

To perform timing and control flow checks at run-time, we connect our hardware check unit directly to the processor pipeline. It needs two input signals: the processor clock and the decoded `MTCR` instructions including the checkpoint values. To estimate the hardware overhead of the integrated check unit, we transformed the SystemC code of the checker

 ISBN: 978-1-61208-233-2

to Very High Speed Integrated Circuit Hardware Description Language (VHDL) [21] and performed a synthesis for an Altera Stratix II Field Programmable Gate Array (FPGA) [2]. A critical point is the stack, which is needed for the call and return mechanism. Its size depends on the call depth of the program. If we store the stack in an on-chip RAM, which is cheaper than logic registers, the check unit requires only 163 Adaptive Look-Up Tables (ALUTs) (0.5 % compared to the overall CarCore processor) and 102 (1.0 %) logic registers, independent of the call depth. However, in this case we have an additional memory overhead of $(call\_depth * 9 \text{ Bit})/8$ Byte for the stack.

Currently, interrupts are neglected in our implementation. However, it is possible to extend the stack of the hardware check unit in order to support a kind of context change in case of an interrupt. But this will be part of our future work.

V. Experimental Evaluations

In this section, we focus on a detailed analysis of the implemented timing and control flow checking technique. In detail, we evaluate the detection coverage and latency using simulations with fault injections and we measure the overhead caused by the proposed mechanism. All evaluations are performed on the SystemC model of the CarCore processor, which was enhanced by the presented hardware check unit. Moreover, we integrated an extension enabling a systematic fault injection.

A. Benchmark Programs

We use different applications of the Embedded Microprocessor Benchmark Consortium (EEMBC) AutoBench 1.1 benchmark suite [7]. These programs are implemented in standard C and represent typical properties and requirements of automotive software for embedded systems. For the compilation, we use the *HighTec GNU C/C++ Compiler* for Infineon's TriCore (optimization level O2 enabled) [9].

B. Fault Model

Around 80 % - 90 % of hardware errors are induced by transient faults [6]. Therefore, in this context, we focus on transient faults in the form of Single Event Upsets (SEUs) during operation. These SEUs, usually appearing as bit flips, are presumed to occur in the instruction memory, since the consequences are very heterogeneous and challenging for a successful detection in such cases. As multiple bit faults at a time are extremely seldom, we assume only one single occurrence per program execution. For the fault injection studies, we modified the fetch stage of our simulated processor pipeline to inject bitflips. As the memory footprint of the EEMBC benchmarks is quite low, we can iteratively run simulations with a systematic injection of all potential bit flips in the instruction memory. Allover, we performed 143,673 simulation runs, each containing one bit flip.

C. Fault Coverage

As a first result of our evaluation studies, we observed that 67.0 % of injected faults cause an error, i.e., a deviation from the correct program functionality. In 33.0 % of all simulation runs, the injected faults showed no effects. This mainly results from the following causes:

- Since, according to our fault model, bit flips are injected in the fetch stage of the processor (always fetching 64 bit, i.e., up to four instructions at once), the faulty instructions are often not executed, e.g., in case of previously executed control flow instructions.
- The TriCore instruction set contains several unused bits in opcodes, where a bitflip will cause no erroneous behaviour, too.
- If a bitflip is injected inside an instruction representing a checkpoint, a shortening of the instrumented BCET value / an elongation of the WCET value will neither be detected nor lead to an error.

If we focus only on injected faults leading to errors, we can see that 28.4 % of these simulations abort due to an exception by the processor (19.5 % illegal opcode, 8.9 % wrong memory access). We can also neglect this part in our following considerations, since an additional error detection is not necessary in these cases.

In the first line (A) of Fig. 4 we can finally see the detection coverage of our proposed check mechanism – regarding errors, which are not caught by a processor exception. The results show that a total of 65.3 % can be detected: 41.1 % by logical control flow checks because of a wrong order of IDs, 19.4 % by timing checks due to an exceeding of the instrumented WCET estimates and 4.8 % by timing checks due to a deviation from the BCET values. On the other hand, 33.1 % of simulation runs terminate with wrong results. These are mostly pure data errors, which cannot be covered by our mechanism. Finally, we see 1.6 % of undetected endless loops; these loops comprise multiple basic blocks, since loops within one single basic block could be easily detected by our temporal check mechanism.

To compare the results to a system without our check mechanism, we also conduct a fault injection study using the EEMBC benchmarks without modifications. As these applications use less instruction memory due to the missing instrumentation, a less number of different bit flips can be injected (a total of 83,782 instead of 143,673). However, we can see a similar percentage of 38.7 % injected faults without any effects on the program behaviour. Regarding the remaining 61.3 % of simulation runs, there is a somewhat higher rate of 39.6 % of detections by processor exceptions. This increase is caused by the low detection latency of our check mechanism: Since several errors are detected very early, these errors can no longer raise a processor exception several cycles later. Focusing on errors, which are not caught by exceptions (see Fig. 4 (B)), there would be 83.2 % of

 ISBN: 978-1-61208-233-2

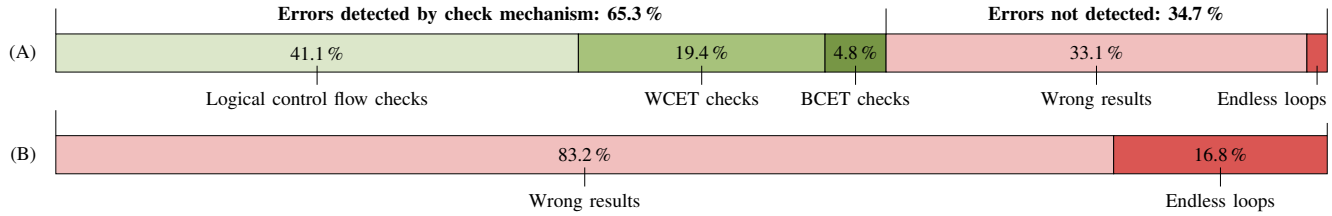


Figure 4. Behaviour of erroneous executions with (A) and without (B) integration of the proposed detection technique (results based on injected faults leading to errors, which are not caught by processor exceptions)

simulation runs terminating with wrong results and 16.8 % causing an endless loop.

Finally, we can conclude that the integration of our check mechanism reduces the number of errors leading to wrong results by more than 60 % (from 83.2 % to 33.1 %). The rate of errors leading to undetected endless loops even decreases by more than 90 % (from 16.8 % to 1.6 %).

D. Detection Latency

Beside the coverage, we evaluate the latency of our detection mechanism, i.e., the amount of processor cycles between fault injection and error detection. Focusing on latencies lower than 100 cycles (which make around 95 % of all executions), we receive a distribution shown in Fig. 5. Overall, the simulations show an average detection latency of 22.1 processor cycles. Values resulting from logical checks are generally lower (16.5 cycles on average) than those resulting from timing checks (25.4 cycles on average by BCET checks, 34.1 cycles by WCET checks). This difference is easy to explain: While an error is detected by logical checks directly after reaching a checkpoint with a wrong ID, an exceeding of the allowed execution time can only be detected when the estimated WCET bound of a basic block was overrun. The fact that several detections in Fig. 5 have a latency of around 70 cycles is a consequence of the call and return handling of the CarCore, which takes a high execution time compared to other architectures.

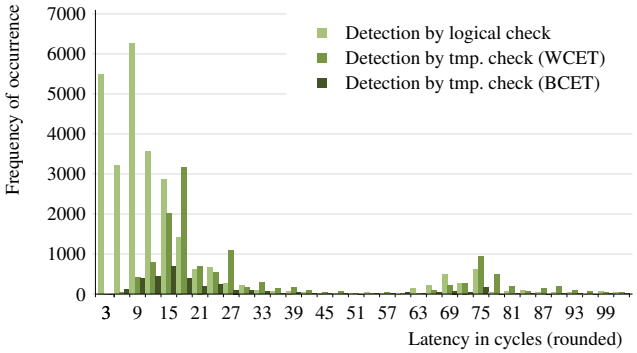


Figure 5. Distribution of detection latencies

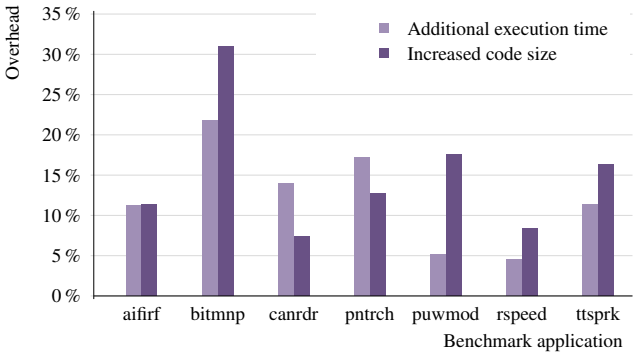


Figure 6. Overhead of EEMBC benchmarks

E. Overhead

The software instrumentation of our technique provides a higher level of reliability but causes overhead. We aim to find an optimal trade-off between execution time and memory overhead on the one hand and good results concerning error detection on the other.

As described in Section IV-B, our instrumentation technique needs three processor instructions for each checkpoint. Since the CarCore is able to execute two of these instructions in parallel, the execution of a checkpoint usually requires two processor cycles. Fig. 6 shows the results measured on the selected EEMBC benchmarks; as can be seen, the additional execution time is low, only 12.2 % in the average case. To determine the memory overhead we compare the number of instructions with the original benchmark program without instrumentation. Here, we can see an increased code size of 15.0 % in the average case. Regarding the benchmark *bitmnp*, we observe slightly higher results, since this application contains lots of very small basic blocks.

VI. Summary and Future Work

In this paper, we have presented the fault detection capabilities of our hybrid hardware-software technique for the on-line detection of control flow and timing errors. Our approach goes one step beyond related checking mechanisms:

 ISBN: 978-1-61208-233-2

Besides monitoring only logical correctness of the control flow, we additionally introduce a technique to guarantee temporal correctness, especially focusing on hard real-time systems.

We have implemented our error detection technique for the hard real-time capable CarCore processor. The hardware overhead of the check unit is very low, it requires only 0.5 % of ALUTs compared to the processor core. Fault injection experiments on automotive benchmarks prove the effectivity of our approach: More than 65 % of injected SEUs uncaught by processor exceptions can be detected. The number of simulation runs terminating with wrong results can be reduced by more than 60 %, the rate of endless loops even by 90 % using the proposed mechanism. Furthermore, the detection latency of our technique is very low: An error is detected after only 22.1 cycles in the average case. Moreover, we measured the instrumentation overhead for several benchmarks. In our evaluations, the mean additional execution time is only 12.2 %, while the increased code size is around 15.0 %.

In our future work, we will further optimize the ratio between coverage, latency and the occurring overhead: In case of a long basic block with a high WCET, a potential fault is currently detected with high latency. This problem can be avoided by splitting blocks and adding extra checkpoints in the middle. On the other side, very small basic block causing much overhead could be combined with neighboring blocks without suffering from detection quality.

REFERENCES

[1] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection," *IEEE Trans. Par. and Dist. Systems*, vol. 10, no. 6, pp. 627–641, 1999.

[2] "ALTERA Stratix II Device Handbook, Volume 1 (ver 4.5)," http://www.altera.com/literature/lit-stx2.jsp, 2011.

[3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," in *Proc.* 8^{th} *SEUS Workshop*, 2010, pp. 35–46.

[4] P. Bernardi, L. Bolzani, M. Rebaudengo, M. S. Reorda, F.L.Vargas, and M. Violante, "A New Hybrid Fault Detection Technique for Systems-on-a-Chip," *IEEE Trans. Comp.*, vol. 55, no. 2, pp. 185–198, 2006.

[5] A. Bonenfant, I. Broster, C. Ballabriga, G. Bernat, H. Cassé, M. Houston, N. Merriam, M. de Michiel, C. Rochange, and P. Sainrat, "Coding Guidelines for WCET Analysis Using Measurement-Based and Static Analysis Techniques," IRIT Toulouse, Tech. Rep., 2010.

[6] E. W. Czeck and D. P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," in *Proc.* 20^{th} *Int'l Symp. Fault-Tolerant Computing (FTCS)*, 1990, pp. 236–243.

[7] "EEMBC AutoBench 1.1," http://www.eembc.org/, 2011.

[8] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-Error Detection Using Control Flow Assertions," in *Proc.* 18^{th} *IEEE Int'l Symp. Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2003, pp. 581–588.

[9] HighTec EDV-Systeme GmbH, http://www.hightec-rt.com/.

[10] Infineon Technologies AG, *TriCore 1 User's Manual*, January 2008, v1.3.8.

[11] D. Lu, "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. Comp.*, vol. 31, no. 7, pp. 681–685, 1982.

[12] A. Mahmood and E. McCluskey, "Concurrent Error Detection Using Watchdog Processors–A Survey," *IEEE Trans. Comp.*, vol. 37, no. 2, pp. 160–174, 1988.

[13] J. Mische, I. Guliashvili, S. Uhrig, and T. Ungerer, "How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT," in *Proc.* 23^{rd} *Int'l Conf. Architecture of Computing Systems (ARCS)*, 2010, pp. 2–14.

[14] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow Checking by Software Signatures," *IEEE Trans. Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[15] J. Ohlsson and M. Rimen, "Implicit Signature Checking," in *Proc.* 25^{th} *Int'l Symp. Fault-Tolerant Computing (FTCS)*, 1995, pp. 218–227.

[16] M. Paolieri and R. Mariani, "Towards Functional-Safe Timing-Dependable Real-Time Architectures," in *Proc.* 17^{th} *Int'l On-Line Testing Symp. (IOLTS)*, 2011, pp. 31–36.

[17] R. Ragel and S. Parameswaran, "A Hybrid Hardware–Software Technique to Improve Reliability in Embedded Processors," *ACM Trans. Embedded Comp. Systems*, vol. 10, no. 3, pp. 36:1–36:16, 2011.

[18] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," in *Proc. Int'l Symp. Code Generation and Optimization (CGO)*, 2005, pp. 243–254.

[19] M. Schuette and J. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. Comp.*, vol. 36, no. 3, pp. 264–276, 1987.

[20] L. Thiele and R. Wilhelm, "Design for Timing Predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157–177, 2004.

[21] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[22] K. Wilken and J. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors," *IEEE Trans. Comp.-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 6, pp. 629–641, 1990.

[23] J. Wolf, B. Fechner, S. Uhrig, and T. Ungerer, "Fine-Grained Timing and Control Flow Error Checking for Hard Real-Time Task Execution," in *Proc.* 7^{th} *Int'l Symp. Industrial Embedded Systems (SIES)*, 2012, pp. 257–266.

[24] J. Wolf, B. Fechner, and T. Ungerer, "Fault Coverage of a Timing and Control Flow Checker for Hard Real-Time Systems," in *Proc.* 18^{th} *Int'l On-Line Testing Symp. (IOLTS)*, 2012, pp. 127–129.

 ISBN: 978-1-61208-233-2