Error detection based on execution-time monitoring

Dieter Steiner
Vienna, Austria
Email: dieter.steiner@spoilerhead.net

Peter Puschner
Department of Computer Engineering
Vienna University of Technology, Vienna, Austria
Email: peter@vmars.tuwien.ac.at

***Abstract*—This paper examines if monitoring of task-execution times can be used to detect errors, so that undesired system behavior and failures of real-time systems can be averted early. To this end, the paper investigates if respectively how the temporal behavior of algorithms changes in the presence of errors that have been caused by hardware faults.**

We used software-implemented fault injection on a number of benchmark programs to create errors and performed runtime measurements for the altered benchmarks to check if the observed execution times are below or above minimum respectively maximum execution-time bounds of the code.

Our results show that up to 70% of errors that are undetectable with other standard techniques can be detected with this simple execution-time monitoring method. The method thus provides an additional layer of protection against errors. It can be implemented with reasonable effort and overhead.

I. Introduction

With continuing miniaturization and fall in price, embedded systems have become ubiquitous. In every aspect of life, inconspicuous microcomputer systems are nowadays present. From smartphones to dishwashers, from sophisticated home automation systems to electric cars we are surrounded by computing systems, and we heavily rely on them.

Error detection should be an important asset in such applications, as many of them are safety-critical. Still the cost of implementing sophisticated detection methods (especially hardware based) sometimes exceed the already tight budget. We therefore investigate if execution-time monitoring is an effective, yet simple and cheap mechanism for runtime error detection. We use software-implemented fault injection to modify executable binaries of benchmark programs, execute the modified binaries, and observe their failure behaviour with a focus on easily observable "vicious" execution times, i.e., execution times outside the interval of "normal" execution times, delimited by the best-case execution time and the worst-case execution time.

A. Goals

In this paper we will be following several major goals:

- Show that computational errors have a measurable influence on execution times of algorithms, i.e., devise an *experimental* setup to demonstrate that cycle- and instruction counts of executions differ in the presence of errors. Figure 1 illustrates the basic error-detection concept. The upper half (green) shows a histogram of the processing times of an algorithm with varying input data. These data (or some WCET-analysis method) can be used to establish bounds for the possible (valid) execution times (vertical green lines). The lower half (red) shows how execution times might change in the presence of faults (e.g., by a bit flip in the code binary). Executions exhibiting a temporal behavior outside the previously established bounds are considered abnormal.

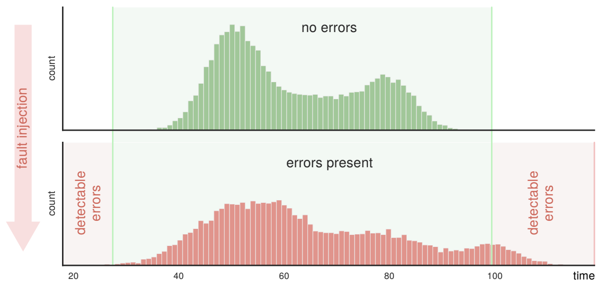

Fig. 1. Change in execution-time distribution in the presence of errors

- Show that execution-time monitoring can be used as an effective and cheap tool to detect errors that cause abnormal timing behaviors.
- Compare different types of algorithms resp. coding techniques to find out for which code types execution-time monitoring yields a higher error-detection rate.
- Evaluate the proposed method on both a simulator and on real hardware; compare and evaluate the results.

II. Scope and Hypothesis

As already mentioned, the working hypothesis used in this paper is that computational errors have a *measurable influence* on an algorithm's execution-time characteristics.

In the course of this paper the evaluated characteristics are:

- *Cycle counts* – This counter is proportional to the time passed during code execution. It accounts for variable-cycle-instructions, L1- and L2-cache delays, and load/store delays when accessing the main memory.
- *Instruction counts* – This value is the actual number of instructions executed. It disregards hardware dependent timing effects and only accounts for actual algorithmic differences – i.e., it will change only when an error leads to a different execution trace.

For our evaluation, faults were injected by modifying the instruction streams (text segments) of our benchmark programs (see Section III-A). We monitored how these modifications affected the mentioned performance counters.

A. Temporal Behavior

Program code typically exhibits different timing behaviors on different executions — programs execute diverging sequences of instructions (instruction traces) for different input data and hardware effects like variable memory access times in memory hierarchies or dynamic instruction scheduling give rise to variable instruction execution times.

These timing variations, meaning that a wide range of timing behaviors is possible and correct on correct executions (see Figure 1), make error detection by observing (monitoring) execution times non-straight-forward — any altered timing that is due to an error but falls into the range of allowed execution times cannot be detected as being erroneous. Only those errors that lead to timings outside the allowed range can be detected. These latter errors are exactly those errors that we want to assess and quantify in this work.

B. Assumptions

Experiments were run under the following conditions.

- Instruction cache is cold, i.e., our experiments assess the first run of the benchmarks. This is due to the nature of static fault injection — the possibility of a crash or side effects makes the successive runs unsuited for evaluation.
- Data caches are hot. As the benchmark fixture sets up the input data prior to running the actual algorithm, data are still cached during the benchmark execution.
- The CPU operates at a fixed frequency.
- Execution-time bounds are generated empirically.

III. Environment

To perform the measurements, the code was split into two parts, a common framework and the benchmark, the first consisting of a measurement library and a common benchmark fixture, the latter being the actual benchmark code under test and a specific fixture tying it to the shared benchmarking framework. This was done to increase modularity and reusability, keeping code duplication to an absolute minimum.

With this structure in place, the benchmarks were integrated into the measurement framework. This framework can be thought of as a *wrapper* around the whole compilation and evaluation process. It creates and mutates the benchmarks, runs them in gem5 and on the Raspberry Pi 2, collects the measurement results into a database, and then analyzes the collected data (see Figure 2).

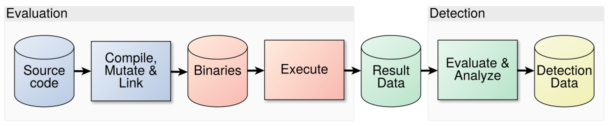

Fig. 2. Measurement framework concept

A. Fault model

Runtime-Injection Frameworks disturb the workload and can have an impact on the measured execution times [1]. To avoid these perturbations, we chose a compile-time- and pre-runtime injection approach, i.e., a program-under-test is modified before executing the binary.

The premise of this paper implies the use of algorithms with (mostly) data-independent execution times. Therefore the faults were injected into the benchmarks' instruction bitstream (`.text` segment [2]). The fault model was limited to *single-bit flips*, i.e., the inversion of a single bit in the program code, thereby simulating, e.g., a faulty memory cell, data corruption in a flash cell or a transient glitch on a bus (Figure 3). This model is similar to the work presented in [3] and a commonly used approach [4].

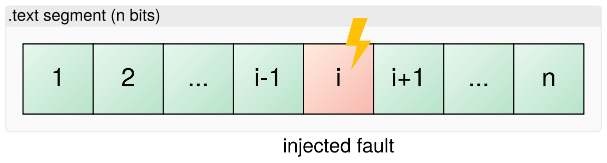

Fig. 3. Bitstream with fault injected at bit i

For each benchmark tested, *every bit* in the actual benchmark's program code was flipped once, and the resulting binary was measured – thereby giving exhaustive coverage of all possible (single-bit) mutations.

B. Used Platforms

All experiments were run on both a real target platform and a hardware simulator. A Raspberry Pi 2 Model B [5] single board computer with a quad-core Cortex-A7 processor and 1 GiB of LPDDR2 RAM served as the primary and real target platform — similar processor systems are used a lot in embedded applications, and the Raspberry Pi is an open system with almost all parts of the software available as open source. For the simulations, the *gem5* simulator [6] was used. Gem5 is a modular simulator, extensively used in system architecture and microarchitecture research, and endorsed by large companies, including AMD, ARM, IBM and Intel.

IV. Experimental Evaluation

Various microbenchmarks were selected with the restriction of them being purely computational, i.e., they are *simple tasks* [7] that are free of communication, synchronization or other blocking. The execution times of these tasks do not depend on other tasks and can be determined *in isolation*. Most tasks were taken from the *Mälardalen Benchmarks* [8]. Others resemble tasks commonly found in embedded systems.

A. Fault Injection Results

Analysis and execution results of all mutant benchmarks were recorded into a database. The results were analyzed in two rounds. First, we categorized the observed failure behaviours and compared how the real execution platform and the simulation environment would compare wrt. these behaviours. Second, we identified those categories where execution-time measurements would be considered helpful to detecting failures that would otherwise remain undetected and analyzed the results of these categories in more detail.

The failure behaviours were classified into four categories:

- CRASH: The benchmark crashes. This includes crashes due to illegal- and undefined instructions, segmentation faults, preliminary calls to `abort()`, etc. This type of fault is detectable by defensive programming methods and the evaluation of error codes, as the system was able to catch the error and report it.
- TIMEOUT: Indicates that the benchmark did not terminate. No result was produced. The benchmark (probably) got stuck in an infinite loop. To enforce termination in the experiment, a timeout was set. On gem5, the timeout was set to ten times the *original* binary's instruction count, on the Raspberry Pi it was set to 10 seconds. This class of faults can usually be detected by watchdog-timers.
- ERROR: The benchmark terminates, but the computed result is *not* as expected. This is the most interesting fault category, as these faults are *not* detected by classical watchdogs and can result in erroneous system behavior.
- OK: The microbenchmark terminates and the computed result is as expected. The timing might still be off, or there could be other unintended side effects, e.g. a temporary result not stored to the intended memory location. So silent data corruption cannot be ruled out.

Figure 4 shows how the recorded executions split into the four categories on the hardware platform resp. simulator. The first notable difference between gem5 and Raspberry is in the *TIMEOUT* class. Executions that run into an endless loop on gem5 crash when executed on the Raspberry. This is due to the looser memory protection of the simulator that allows memory accesses to complete that cause a crash on a real OS. There are also differences in handling ill-defined opcodes. Some of those are executed by the real CPU core, but lead to a crash on gem5. See, e.g., the results for *nop* and *long_nop*.

It is noteworthy that for every benchmark at least 30% of the injected faults did not lead to crashes (*CRASH*) or endless loops (*TIMEOUT*). In some cases this number was above 60%. Also note how large the share of the *OK* class is, and how widely distributed — ranging from about 5% for *aes* to 70% for *prime*. This distribution is similar to results reported in [3].

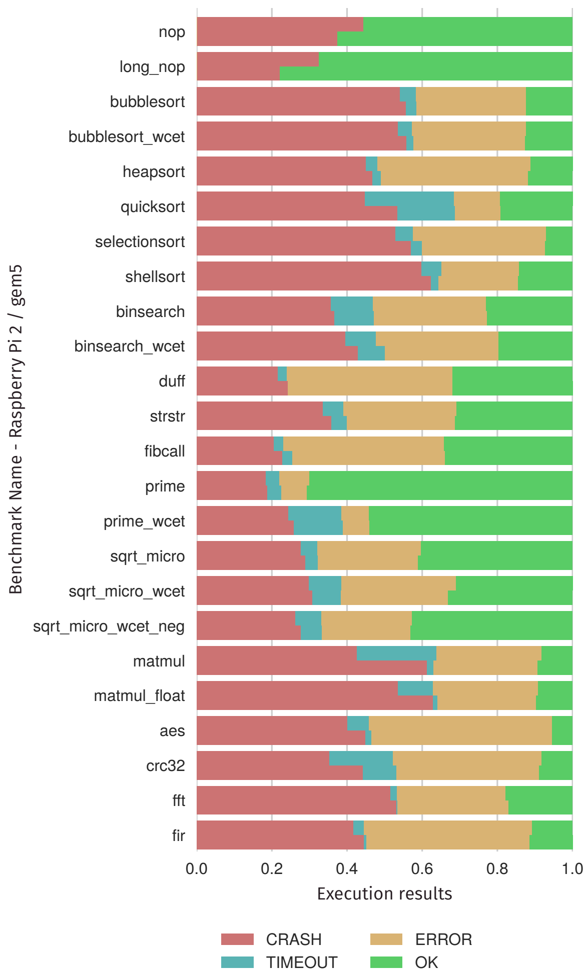

Fig. 4. Failure behaviors observed in experiments.

B. Detecting Abnormal Execution Times

The main goal of this work aimed at assessing if execution-time monitoring would be suitable for detecting errors that would otherwise remain undetected. The hypothesis was that some errors would affect the execution time of the code so that it would terminate either earlier than the fastest error-free execution or later than the slowest error-free execution.

To be able to assess timing errors by mutated code, we first had to determine best-case execution time (BCET) and worst-case execution time (WCET) estimates for the unmanipulated, correct version of each program under test. To this end we executed each benchmark with various input data (*Jitter* measurements), including the input data we would finally use in the fault-injection experiments (yielding the *Original* result). During the fault-injection experiments, the so-derived BCET/WCET estimates were used as thresholds for the detection of timing errors. Errors in mutated code were classified as *detected* if the observed execution time (cycle count) was outside the established bounds. Otherwise the error was classified as *undetected* (see Figure 5).

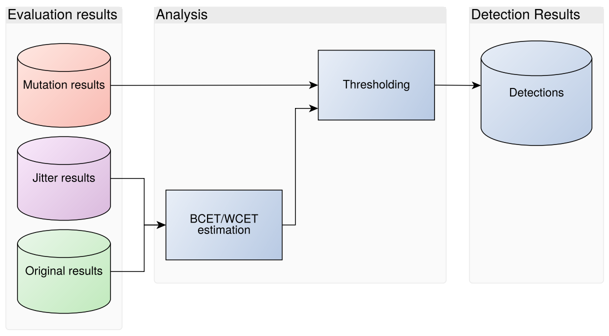

Fig. 5. Detection model

Tables I and II show the computed BCET/WCET thresholds used for the detection of timing errors on the Raspberry and gem5. The threshold values clearly show that a) the cycle values have a higher variation than the instruction counts — this is due to load/store latencies, caching effects and variable-cycles-instructions, and b) the WCET optimized algorithms [9], [10], i.e., algorithms that had been designed for small execution-time jitter by avoiding input-dependent branches whenever possible (*bubblesort_wcet*, *binsearch_wcet*, *prime_wcet*, *sqrt_micro_wcet*) have indeed a much lower variation than their non-optimized counterparts.

	$Cycles_{min}$	$Cycles_{max}$	$Instructions_{min}$	$Instructions_{max}$
nop	0	191	4	4
long_nop	580	921	103	103
bubblesort	7717	3762326	8008	4003004
bubblesort_wcet	3510792	3511609	3502504	3502508
heapsort	184419	206490	178668	200108
quicksort	104312	1063829	103646	1052975
selectionsort	3013509	3366114	4008011	4008015
shellsort	86507	190508	112554	177464
binsearch	921	1898	133	250
binsearch_wcet	1182	1993	398	398
duff	4743	5371	4620	4621
strstr	1479	105646	943	103411
fibcall	107	1260	21	1119
prime	43	601784	15	301222
prime_wcet	11699716	11732111	14188557	14188566
sqrt_micro	239	766	18	202
sqrt_micro_wcet	533	805	116	116
sqrt_micro_wcet_neg	106	1001	17	119
matmul	74148	86884	74145	74147
matmul_float	114104	114870	82542	82544
aes	24515	25498	19605	19606
crc32	76025	76684	110615	110617
fft	191958	197417	87680	87682
fir	185828	187157	159145	159147

TABLE I
COMPUTED TIMING BOUNDS – RASPBERRY PI 2

C. Detection Results

Figure 6 shows the results for the threshold-based detection of timing errors for all executions of mutants that did not lead to crashes. Those are the scenarios that could potentially benefit from error detection by execution-time monitoring — for those cases that run into timeouts (denoted by *Timeout*), execution-time monitoring based on the WCET could be used for *early* error detection (as soon as a WCET overrun is detected). For scenarios of the *ERROR* and *OK* classes of Section IV-A, any threshold violation allows us to detect errors that would otherwise remain undetected. Execution scenarios for which threshold violations were observed are marked by *Runtime*. Mutants of the *ERROR* or *OK* class that terminated within the established execution-time bounds are denoted by *Undetected (Error)* resp. *Undetected(Ok)*.

One can see that the error detection rates for execution-time monitoring strongly depend on the monitored algorithm — they range from less than 10% (*prime*) to over 60% (*heapsort*, *fir*). The cause of those variations can be found in the timing

	$Cycles_{min}$	$Cycles_{max}$	$Instructions_{min}$	$Instructions_{max}$
nop	0	276	0	26
long_nop	54	321	0	0
bubblesort	10255	5015489	7974	4002968
bubblesort_wcet	4515404	4515775	3502482	3502507
heapsort	251847	283675	178646	200086
quicksort	155915	1307556	103695	1053022
selectionsort	4018689	4019263	4007985	4008045
shellsort	117866	273377	112603	177513
binsearch	0	1283	182	299
binsearch_wcet	862	2948	341	361
duff	4646	4993	4583	4586
strstr	1516	138007	906	103374
fibcall	37	1288	0	1122
prime	0	370723	49	301254
prime_wcet	17596123	17596475	14188505	14188522
sqrt_micro	188	607	0	168
sqrt_micro_wcet	1024	1162	94	119
sqrt_micro_wcet_neg	37	1026	0	135
matmul	78228	78313	74088	74108
matmul_float	94706	94889	82505	82508
aes	25149	25717	19571	19621
crc32	112537	112732	110567	110649
fft	173955	178633	87646	89120
fir	217909	218084	159111	159146

TABLE II
COMPUTED TIMING BOUNDS – GEM5

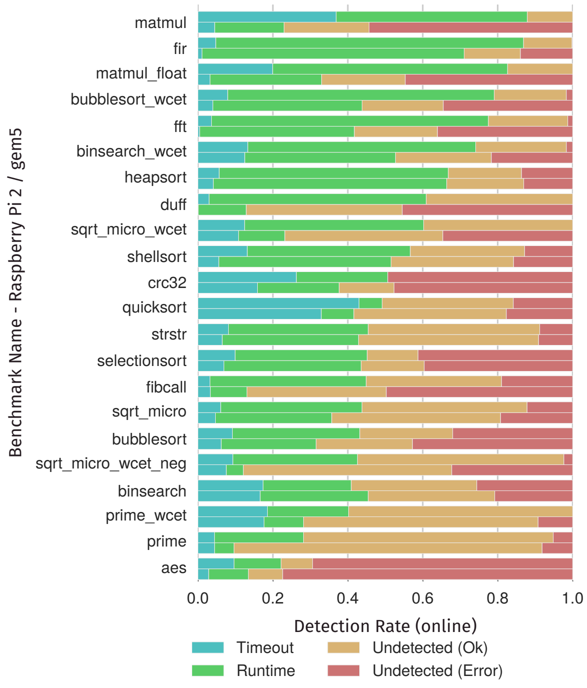

Fig. 6. Detection results excluding crashes — gem5 (top) vs Raspberry Pi 2 (bottom)

distributions and the derived detection thresholds (Tables I,II). *Heapsort* and *fir* show a low variation for both, cycles and instruction count, whereas *prime* exhibits an extremely input-data dependent runtime behavior that hides timing deviations.

Figure 6 also shows the impact of *timing variability* on the Raspberry — gem5's detection rate is up to four times higher (*matmul*). Timing variations on the Raspberry that, again, mask timing errors can be explained with caching effects, variable memory latencies and variable-cycle-instructions.

Of the sorting algorithms used, *quicksort* exhibited one of the lowest detection rates. This phenomenon can be explained by looking at the used detection thresholds, where the $O(n^2)$ worst-case complexity of the algorithm yields a wide BCET/WCET threshold window. Also, *quicksort* has the highest percentage of *TIMEOUT*s (infinite loops) of all tested algorithms, probably due to its fragile recursive structure and multiple loops. The WCET optimized bubblesort version *bubblesort_wcet* shows notably higher detection rates than the unoptimized counterpart, and while it is slower on average, its worst-case behavior is better.

AES also exhibits a low detection rate. This was not unexpected, as all loops have a constant number of iterations, i.e., there are no *data dependent branches* whose branching behavior can be disturbed by data-changing mutations.

In general, non-WCET optimized algorithms (with a large execution-time variability) tend to exhibit a lower detection rate (e.g. *bubblesort, prime*) than algorithms with stable timing. However the converse is not true – while some optimized benchmarks (*bubblesort_wcet, heapsort, matmul, fir,...*) show high detection rates – others (*aes, sqrt_micro_wcet*) exhibit average to low detection rates only. Also fixed-iteration loops (*fir,matmul,...*) seem to improve detection rates – presumably due to the amplification provided by the loop.

It is also noteworthy that WCET optimized algorithms are not necessarily better suited for the proposed detection approach. While the detection rate for the pairs *binsearch/binsearch/_wcet* and *prime/prime_wcet* shows distinct improvements — the opposite is true for *sqrt_micro/sqrt_micro_wcet*. This probably happens because *sqrt_micro_wcet* is simpler in structure and executes fewer data dependent instructions that can be disturbed by mutations.

V. Conclusion

We showed that fine-grained execution-time monitoring is possible and can be effectively used for the error detection at runtime. The method shows promising results: up to 70% of non-crash errors were detected by monitoring cycle-accurate execution times resp. instruction counts.

The achieved detection rate varies between different algorithms, with WCET-optimized algorithms yielding, on average, higher detection rates than non-optimized ones. Algorithms containing a high percentage of branches/loops are more sensitive to bit flips that can change their timing. On the other hand, the larger range of allowed execution times of the latter may cause some timing changes to remain undetected.

The implementation effort needed to add execution-time monitoring to an existing program is low, but one has to expect to spend a fair amount of time on identifying safe BCET/WCET bounds.

The comparison of the Raspberry Pi 2 computer and the gem5 simulator shows that platforms with temporally predictable instructions and memory access are beneficial for detecting erroneous behaviour by execution-time monitoring — in our experiment, gem5 allowed for a 20% higher error-detection rate than the Raspberry.

In future work, we expect to further increase the error-detection rate of our method, by breaking the BCET/WCET threshold interval down into error-detection intervals of finer time granularity.

Experimenting with gem5 showed that, while it is a reliable simulator for the ARM architecture, extensive fine tuning is needed to closely mimic the behavior of a given SoC/system-board. On the other hand, its open architecture and extensible API facilitate a consistent simulation and analysis of small differences in CPU design, thus providing helpful insights for choosing suitable processors for real-time workloads.

The tools and libraries needed to implement the proposed method are openly available, easy to use and integrate into existing systems running Linux — given that the SOC used provides a hardware PMU and support for the `perf_event` framework is enabled.

References

[1] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[2] Santa Cruz Operation, Inc., *System V Application Binary Interface*, 4th ed., The Santa Cruz Operation, Inc., 1997.

[3] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, *Computer Safety, Reliability, and Security: 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution, pp. 265–276.

[4] A. Johansson, N. Suri, and B. Murphy, "On the Selection of Error Model(s) for OS Robustness Evaluation," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, June 2007, pp. 502–511.

[5] R. P. Foundation, "Raspberry Pi 2 Model B," 2015, accessed: 2015-10-17. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[7] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[8] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146, the printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[9] P. P. Puschner, "Algorithms for Dependable Hard Real-Time Systems," in *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003. 8th IEEE International Workshop on*, Jan 2003, pp. 26–31.

[10] P. Puschner, "Experiments with WCET-Oriented Programming and the Single-Path Architecture," in *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, Feb 2005, pp. 205–210.