B. Juurlink, W.Karl (Hrsg.)
Proceedings 28th PARS Workshop

Symptom-based Fault Detection in Modern Computer Systems

Thomas Becker [1], Nico Rudolf[2], Dai Yang[3], Wolfgang Karl [4]

Abstract: Miniaturization and the increasing number of components, which get steadily more complex, lead to a rising failure rate in modern computer systems. Especially soft hardware errors are a major problem because they are usually temporary and therefore hard to detect. As classical fault-tolerance methods are very costly and reduce system efficiency, light-weight methods are needed to increase system reliability. A method that copes with this requirement is symptom-based fault detection. In this work, we evaluate the ability to detect different faults with symptom-based fault detection by using hardware performance counters. As the knowledge of a fault occurrence is usually not enough, we also evaluate the possibility to make conclusions about which fault occurred. For the evaluation, we used the fault-injection library FINJ and manually manipulated loops. The results show that symptom-based fault detection enables the system to detect faulty application behavior, however fine-grained conclusions about the causing fault are hardly possible.

Keywords: System Reliability; Fault Detection; Fault Analysis

1 Motivation

To satisfy the demand of higher computing power, increasing miniaturization of components with increasing complexity and a growing number of components is deployed. This leads to the constant rising of the failure rate of today's computing systems. Especially soft errors in hardware that are random and of temporary nature occur more often, as they are caused by lowering the system voltage in the creation of energy-efficient products [SS02]. These faults are particularly hard to detect as they do not always lead to wrong results and may not be reproducible.

Lightweight methods that increase system reliability are needed as classical methods, like redundancy and checkpoints, increase the cost and significantly reduce the system efficiency [Be08]. One example of a lightweight method is symptom-based fault detection. Symptom-based fault detection identifies faults in the system by comparing values of runtime metrics called symptoms with a database of correct behavior and assuming differing behavior is caused by a fault.

[1] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, Germany thomas.becker@kit.edu
[2] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, Germany nico.rudolf@student.kit.edu
[3] Technische Universität München, Boltzmanstraße 3, 85748 Garching, Germany d.yang@tum.de
[4] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, Germany wolfgang.karl@kit.edu

In this work, we evaluate the ability to detect different faults caused by hardware and system interferences. As the sole knowledge of the occurrence of a fault is not enough to choose an adequate recovery method, we also want to examine if the resulting symptoms allow to make conclusions about the root cause. In summary, we make the following contributions:

- We evaluate the concept of symptom-based fault detection with different faults and system interferences, e.g. created by the fault-injection library FINJ, on a state-of-the-art computing system.
- We analyze if the resulting symptoms make conclusions about the causing fault possible.

The remainder of this paper is structured as follows: Section 2 explains the necessary fundamentals for this work. We describe our method of symptom-based fault detection and how to evaluate it in Section 3. The following Section 4 presents our experimental setup including the fault-injection methods used and our results. We then describe related work in Section 5 and wrap up with conclusions and future work in Section 6.

2 Fundamentals

To define **faults**, **errors** and **failures**, we use the work of Salfner et al. [SLM10]:

- A **failure** refers to misbehavior that can be observed by the user. This means there may be something wrong inside the system, but as long this does not result in incorrect output there is no failure.
- An **error** is defined as the deviation of the system state from the correct state. Hence, an error may lead to the service failure of an system, but also can stay unnoticed.
- **Faults** are then the hypothesized cause of an error. This means that errors are manifestations of faults.

Symptom-based fault detection is based on the following hypothesis: Systems exhibit steady-state performance behavior with few variations in the non-faulty case. However, a fault manifests itself as increasingly unstable performance-related behavior before escalating into a failure [WPN07]. This means that a symptom for occurring faults manifests itsel as a variation of performance-related behavior, which can be monitored by performance counters. To sum up, the basic concept is to monitor performance counters and assume the occurrence of faults if they vary significantly compared to a baseline.

Performance Application Programming Interface (PAPI), developed by the University of Tennessee [Te10], is a user-level library that grants easy access to performance counters. PAPI provides two interfaces for application developers. The high level interface is simple

to use and allows fast access to standard events that are present in most architectures. In contrast, the low level interface allows a more detailed control and the access to so-called native events that are specific to the underlying architecture. In this work, we use the PAPI low level interface.

Fault injection is the deliberate triggering of faults with the objective to observe the resulting behavior and to test error handling code. Fault injection can be done directly in hardware or by using specific software tools. In this work, we focus on software implemented fault injection. As fault injection is an important technique in proving the correctness and robustness of a system or software, many tools and libraries exist. For this work, we chose FINJ [Ne18], a fault injection tool for HPC systems. FINJ is implemented in Python and based upon tasks. Thereby, a task can represent a benchmark or a fault-triggering program. The execution of a workload of tasks is controlled by a specific controller that schedules and starts the tasks on an engine.

3 Method

To evaluate symptom-based fault detection, the first step is to create a database, which stores the performance behavior of correct executions. As a wide range of performance-related metrics are available, relevant metrics have to be filtered out. We define a metric as relevant, if their values do not vary significantly during repeated runs without faults and show significant variance in the presence of faults. Relevant metrics can be found via profiling runs. If the profiling runs only include executions without faults, the set of possibly relevant metrics can be at least reduced to metrics that are stable during repeated executions. Additionally, a lower and an upper threshold for the values of the selected metrics have to be chosen. These thresholds are used to detect anomalies later. If a monitored value lies outside of these thresholds, the occurrence of a fault is suspected.

For a selected number of benchmarks, we execute runs with injected faults. For each run, only one specific fault is used. If there are monitored metrics whose values lie outside of the chosen thresholds, the injected fault is assumed to be detected.

After all experiments are conducted, an analysis step follows. The injected faults are classified. Then, the monitored results are checked if faults belonging to the same class show similar changes in the runtime metrics. We also check if we can differentiate faults belonging to different calsses by observing the monitored runtime behavior. In the current state, this is done by hand, but the process should be automated in the future by using machine learning algorithms.

4 Evaluation

This section presents our fault injection methods, the experimental setup and the conducted experiments. We used three different benchmarks, i.e. matrix multiplications with 300×300

and 500×500 floating-point matrices, as well as Hotspot3D and SRAD of the Rodinia Benchmark Suite. We added PAPI instrumentation code to each benchmark in order to monitor selected performance counters. All experiments are executed ten times with and without fault injection. The results show the average $\varnothing$ and the standard variation s of the ten executions with and without the injected fault. Additionally, we computed an occurrence ratio, that shows how often the value of the performance counter varied significantly from the non-faulty case. This means if a value increases significantly in 8 out of 10 executions, the occurrence ratio would be 80 %.

The experiments are conducted on a server with two Intel Xeon E5-2650 v4 CPUs a 12 cores each and 128 GB with 2400MHz DDR4 SDRAM DIMM (PC4-19200). The software environment includes Ubuntu 18.04.1, the Linux 4.15.0-43-generic kernel and glibc 2.27.

4.1 Fault Injection

In this work, we analyse three types of faults: the alteration of loop index variables to create random memory accesses, the reduction of loop iterations, and interferences created by the FINJ library. The intereferences are used to mimic anomalies in real-life systems by stressing single components, emulating interference or malfunction in that component.

The alteration of loop index variables is done by overwriting the current value of the index variable in the pages of the process in main memory via opening */proc/$procid/mem/* and then jumping to the address of the variable. An example can be seen in Listing 1.

```
FILE *mem = fopen("/proc/$procid/mem/", "w");
fseek(mem, (uintptr_t) &i, SEEK_CUR);
fwrite(&manipulation, sizeof(i), 1, mem);
fclose(mem);
```

List. 1: Altering an index variable

The process id and and the variable adress can be obtained by calling `popen("pid of $processname", "r")` and writing out the adress to a file that can be read by the alteration process, respectively. As we only want to create random accesses, we made sure that the altered index value always lies within the given range and that the correct number of iterations is executed. In the same way, the number of iterations can be altered by overwriting the current iteration bound.

From the FINJ library we used five interference applications: *copy*, *ddot*, *dial*, *leak* ,and *memeater* that are inspired by Tuncer et al. [Tu17].

4.2 Benchmarks

As a first benchmark, we implemented a floating-point matrix multiplication (**mmult**) with 300×300 and 500×500 matrices initialized with random floating-point numbers. Other benchmarks we used are:

Hotspot3D iteratively computes the heat distribution of a 3d chip represented by a grid. In every iteration, a new temperature value depending on the last value, the surrounding values, and a power value is computed for each element. For the evaluation, we used a $512 \times 512 \times 8$ grid with the start values for temperature and power included in the benchmark suite, and a total of 1000 iterations.

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations used to remove noise without destroying important image features. The benchmark consists of image extraction, continuous iterations over the image, and image compression. As input, we used the 502×458 image provided by the benchmark suite with 100 iterations and $\lambda = 0.5$.

4.2.1 Experiments

The Alteration of the Loop Index Variable creates random accesses into the used data structure. This changes the data cache behavior of the benchmark increasing misses as the random accesses violate the locality principle. The results of the mMult benchmark (s. Table 1) show these changes. Misses in the data translation lookaside buffer (TLB DM)

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI TLB DM	27.8	13.8	1882	258.7	100 %
PAPI PRF DM	88898.7	290.1	503561.3	794	100 %
PAPI L2 DCM	15733227	1125.2	16188659.3	4214.8	100 %
PAPI L3 DCA	15733439.3	312.3	16185326.6	1193.8	100 %

Tab. 1: Results of the combination of mmult and the manipulation of the loop index

and in the L2 data cache (L2 DCM), data prefetch misses (PRF DM), and L3 data caches accesses (L3 DCA) increase significantly.

The Reduction of the Number of Iterations effectively leads to a reduction of issued and executed instructions. In general, the instruction performance counters are very precise, e.g. the counters for the executed floating-point operations always match the actually executed operations with a deviation of 0. Therefore, a reduction of the executed instructions is easily recognizable using the provided instruction counters as can be seen exemplary in the results of the mMult benchmark in Table 2. Here, the total floating-point operations, the floating-point additions and floating-point multiplications scale according to the number of

executed loops. As these results are pretty straightforward, we omitted the results of the other two benchmarks.

Symptom	mMult w/o faults	mMult w faults		
	N = 300	N = 200	N = 100	N = 50
PAPI FP OPS	$54 \cdot 10^6$	$36 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$
PAPI FML OPS	$27 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$	$4.5 \cdot 10^6$
PAPI FADD OPS	$27 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$	$4.5 \cdot 10^6$

Tab. 2: Results of the iteration number reduction for the matrix multiplication benchmark

Copy constantly executes file in- and output executions, thereby creating hard drive interferences (I/O overhead). Expected results are variations in the low-level cache structure and the TLB. These expectations are confirmed by the results of the SRAD benchmark shown in Table 3. In 9 out of 10 runs, large variations can be noted for the L3 total cache misses (L3 TCM) and TLB DM counters. Additionally, the total number of stalls increases on average about 3 % and the number of L2 instruction cache misses (ICM) by about 150 % on average. However, these two symptoms occurre less often and in the case of the L2 ICMs vary significantly between runs, which aggravates a detection. The results for mMult are

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurrence ratio
PAPI L3 TCM	1.8	1.87	44.44	54.81	90 %
PAPI TLB DM	11899.7	1080.84	19816.7	3421.12	90 %
STALLS TOTAL	79803752.8	155218.56	82099161.3	1197866.62	80 %
PAPI L2 ICM	2088	154.6	3188.3	1366.39	60 %

Tab. 3: Results of the combination of SRAD and copy

presented in Table 4. There is a similarity to the results of SRAD where the PRF DMs and total stalls increase, but the biggest variations are seen in the instruction caches. Hotspot3D has similar results with an increase in L2 and L3 cache accesses as well as symptoms such as an increase in L3 data cache writes (DCW) and in cycles stalled waiting for memory writes (PAPI MEM WCY).

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICA	92.7	8.06	751.3	29.04	100 %
PAPI L3 ICA	110.1	26.76	227	20.33	100 %
PAPI PRF DM	258091.5	2350.31	270598.5	1349.06	100 %
STALLS TOTAL	33806710.3	448643.96	35368628.1	1226479.31	80 %

Tab. 4: Results of the combination of the mMult and copy

Leak creates a controlled memory leak by constantly allocating new arrays and copying data into them using `memcpy()`. This leads to additional data cache and TLB misses. The results of SRAD in Table 5 show that the number of L3 TCMs and the number of total

cycles increase significantly throughout all test runs. Additional symptoms are TLB DMs and L3 instruction caches accesses (L3 ICA), which are visible in 80 % of the conducted executions. Similar to the copy benchmark, variations for Hotspot3D are mostly visible in

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occ. rat.
PAPI L3 TCM	1.8	1.87	11061.6	17445.73	100 %
PAPI REF CYC	911683861	4293565.15	1042523457.8	18936885.44	100 %
PAPI TLB DM	11899.7	1080.84	16347.5	4986.21	80 %
PAPI L3 ICA	1950.6	101.63	2439.8	262.8	80 %

Tab. 5: Results of the combination of SRAD and leak

the instruction caches (s. Table 6). In this case however, there is no single symptom that is present in all 10 executions. mMult also shows only two symptoms in combination with leak, increases in total stalls and TLB IMs, which are visible in 80 % of the test runs.

Symptom	Hotspot3D w/o faults		Hotspot3D w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICM	1360	269.32	1870.1	252.18	80 %
PAPI L1 ICM	1641.3	251.18	2079.1	358.2	70 %

Tab. 6: Results of the combination of Hotspot3D and leak

Memeater, like leak, creates a controlled memory leak. Additionally, memeater also executes additions, which in total create misses in the instructions caches additional to the symptoms visible for leak. Mirroring the results of leak, the number of total cache misses and the total cycle number increase significantly in all of the test runs for SRAD. TLB data misses are also significantly augmented again and observable in 8 out of the 10 executions. Furthermore, the additional instructions executed then lead to an increase in L2 ICMs. The results for the combination of Hotspot3D and memeater are identical to the combination of Hotspot3D and leak with increases in L1 and L2 ICMs and MEM WCYs. Even the occurrence ratio is identical for all three symptoms. For mMult, the results resemble the results of the copy benchmark instead of leak, as the visible symptoms are increases in L1 and L2 ICMs, total stalls, PRF DMs, and additionally an increase in cycles with maximum instruction issue (FUL ICY).

Dial uses several floating-point math instructions, like `pow()` and `sqrt`, to create interferences in the ALU. As not much additional data is used while performing these operations, mostly variations in the instruction caches are expected. For SRAD, the results are displayed in Table 7. As expected, significant increases in the L2 ICMs are measured. These correlate with the decrease in instructions cache hits (ICH) and an increase in L3 instruction cache accesses (ICA) (and reads (ICR)). All those symptoms are observable in 100 % of the execution runs. The additional data used for the computations lead to an increase in TLB DMs and a decrease in L2 DCAs. Symptoms for mMult are also manifested in the increase in ICMs and correlating increases in L2 and L3 ICAs. Furthermore, the number of prefetch

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurr. ratio
PAPI L2 ICM	2088	154.6	2915.9	162.82	100 %
PAPI L2 ICH	20175.3	512.32	18633	169.35	100 %
PAPI L3 ICA	1950.6	101.63	2956.6	92.42	100 %
PAPI TLB DM	11899.7	1080.84	15885.1	2255.44	100 %
PAPI L2 DCA	19876232.64	2957162.47	10991900.1	14624.57	90 %

Tab. 7: Results of the combination of SRAD and dial

data misses decreases in every run. Hotspot3D also showed significant increases (up to

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI PRF DM	258091.5	2350.3	236171	15761.98	100 %
PAPI L1 ICM	109.8	16.7	170.5	25.02	90 %
PAPI L2 ICM	130.5	19.34	155.9	14.92	70 %
PAPI L3 ICA	93.4	6.6	130.4	21.98	70 %

Tab. 8: Results of the combination of mMult and dial

400 %) in instruction cache misses. Surprisingly, we measured a decrease in TLB DMs and MEM WCYs.

Ddot is also used to create ALU interferences executing float-point operations. The benchmark allocates and initializes matrices and then executes a floating-point matrix multiplication. Compared to dial, this means that more additional data is used. The results for SRAD (s. Table 9) show the effects of the additional instructions executed on the instruction caches. Again, the ICMs on the L2 level increase, which correlates with the increase of L3 ICAs (and ICRs) and decrease of L2 ICHs. The data usage is not really visible in the monitored values, as the only visible variation was a decrease in L2 data cache accesses, as also seen for SRAD with dial. Similar to the results for dial, we observe

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICM	2088	154.6	2784.3	114.36	100 %
PAPI L3 ICA	1950.6	101.63	2830.9	67.07	100 %
PAPI L2 DCA	19517962.78	3226987.33	11026045.4	66239.26	90 %
PAPI L2 ICH	20175.3	512.32	18740.4	445.68	90 %

Tab. 9: Results of the combination of SRAD and ddot

a significant increase in L1 and L2 ICMs for Hotspot3D and also noticed an increase in misses in the instruction TLB (ITLB). Again, the number of TLB data misses decrease compared to the execution without interferences. The mMult benchmark also shows similar results to dial. Increases in ICMs, TLB DMs and a decrease in PRF DMs are again detected. Additionally, we measured increases in ITLB misses and total stalls.

Symptom	Hotspot3D w/o faults		Hotspot3D w faults		
	∅	s	∅	s	occurrence ratio
PAPI L1 ICM	1277.4	208.14	5060.8	286.99	100 %
PAPI L2 ICM	1416.9	249.7	5228.9	268.77	100 %
PAPI TLB DM	513562.6	137417.52	213266.5	4514.72	90 %
ITLB MISS	587.4	257.26	958.2	312.8	70 %

Tab. 10: Results of the combination of Hotspot3D and ddot

4.2.2 Statistical Analysis

To test the statistical significance of our results, we compute the Welch's t-test [WE47], a statistical test that is used to test the hypothesis that two means belong to the same population. If that is the case, the occurring symptom originates from correct behavior and not a fault. Exemplary, the results of three tests are shown here. Tables 11, 12 and 13 show the results for the combination of SRAD and dial, Hotspot3D and leak, and mMult and copy. For all symptoms monitored in these benchmarks, the deviation that occured in

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICM	17.95	-11.66	0.001	-3.922	$8.26 \cdot 10^{-10}$
PAPI L2 ICH	10.94	9.04	0.001	4.437	$2.09 \cdot 10^{-6}$
PAPI L3 ICA	17.84	-23.16	0.001	-3.922	$1 \cdot 10^{-14}$
PAPI TLB DM	12.93	-5.04	0.001	-4.221	$2.31 \cdot 10^{-4}$
PAPI L2 DCA	9.00	9.50	0.001	4.781	$5.47 \cdot 10^{-6}$

Tab. 11: Welch's t-test results for the combination of SRAD and dial

the fault-injection runs has a probabilty to occur in normal runs of less than 1 % and in most cases even less than 0.1 %. This means that it is almost definite that the monitored symptoms are not from the distribution observed in the fault-free runs. Therefore, it is reasonable to say that the injected faults changed the application behavior. For all conducted

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICM	17.92	-4.37	0.001	-3.922	$3.71 \cdot 10^{-4}$
PAPI L2 ICM	16.13	-3.16	0.01	-2.898	0.006

Tab. 12: Welch's t-test results for the combination of Hotspot3D and leak

benchmarks, the maximum probability for a symptom occuring in a fault-free run is 3.6 %. For 18 of 30 symptoms examined, the test resulted in a probability of less than 0.1 %. So in summary, the Welch's t-tests show the statistical significance of the monitored symptoms for all benchmarks in our experiments.

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICA	10.38	-69.1	0.001	-4.437	0
PAPI L3 ICA	16.79	-11	0.001	-3.965	$4.3 \cdot 10^{-9}$
PAPI PRF DM	14.35	-14.59.16	0.001	-4.073	$5.24 \cdot 10^{-10}$
STALLS TOTAL	11.37	-3.78	0.01	-3.012	0.0029

Tab. 13: Welch's t-test results for the combination of mMult and copy

5 Related Work

Symptom-based fault detection has be done before in the literature. Arulaj et al. [Ar13] use performance counters as symptoms to detect concurrency bugs in production-run systems. They access the performance counters via Linux perf.

Williams et al. [WPN07] also use different performance counters to detect anomalous behavior that should form patterns leading up to failures. With an anomaly detector they create a time series that serves as input for a failure predictor. The predictor checks if there is a pattern that indicates escalating instability which then signals an impending failure.

Instead of considering all performance metrics, Narayanasamy et al. [NCC07] focus on the branch predictor, the store set predictor and L2 cache accesses. They assume that a faulty execution leads to an increase in *undesirable outcomes*, e.g. a misprediction by a branch predictor.

The ReStore architecture by Wang et al. [WP05] uses symptom-based fault detection combined with a checkpointing mechanism. If the fault detection signals the occurrence of a fault, the architectural state of an earlier checkpoint is restored. Exceptions, branch mispredictions coupled with a confidence predictor for the branch and event logs that store events, like control instruction outcomes, are used as symptoms.

mSWAT [Ha09] is a fault detection and fault diagnosis framework for multicore architectures. The framework uses a fatal-traps detector, a hang detector checking branch frequencies, a high-OS detector that monitors OS invocations, and a kernel panic detector as symptoms. If a symptom occurs, a diagnosis mechanism is invoked that decides whether the fault is just a software bug or a hardware fault, whether it is a transient or permanent fault and which core is faulty. This is done by tracing and replaying execution.

6 Conclusion

In this work, we evaluated the concept of symptom-based fault detection with three different benchmarks: a matrix multiplication, Hotspot3D and SRAD of the Rodinia Benchmark Suite, combined with seven ways to generate faults and interferences. We generated faults and interferences that affect different parts of a computing system, thereby creating a

classification for the injected faults. We then analyzed the results, if the monitored symptoms allow to conclude which fault originally occurred.

In general, we have found minimally two symptoms for every benchmark fault combination. In the worst case there was at least one symptom with an occurrence ratio of 80 % and for most cases at least one symptom with a ratio of 100 %. We have also shown the statistical significance of the monitored symptoms by computing Welch's t-test. Therefore, it is fair to say that we are able to detect all faults in every benchmark using symptom-based fault detection.

Faults that alter the instruction number are easily detectable as these counters are very precise. A distinction from the other fault classes is also easy, as they do not alter the number of instructions.

Considering each benchmark on their own, the different instances of the interference classes (memory-bound and ALU-bound interferences) have very similar behavior. E.g. SRAD showed significant variations in total L3 cache misses, the number of cycles needed and TLB DMs for all three memory bound benchmarks. However, there is no single set of symptoms that is relevant for every instance of a class over all benchmarks. Only significant variations of instruction cache accesses and misses were visible for each instance of the interference classes over all benchmarks. A possible distinction could be the degree to which the values increase. ALU-bound interferences create larger increases compared to memory-bound ones. Additionally, the memory-bound interferences create more variations in data related counters in most cases. Differentiating between the interferences and the loop index manipulation is possible as the loop index manipulation does not affect the instruction caches. However, to be able to make a differentiation the complete set of symptoms has to occur simultaneously. For five symptoms with an occurence ratio of 80 % each, the probabilty that all symptoms occur together is only around 33 %. Without the whole set of symptoms however, a useful differentiation is not possible. As a summary, we conclude that symptom-based fault detection is very useful to detect faulty application behavior. Coarse-grained conclusions about the causing fault are generally possible, but finer distinctions need additional tool support.

In the future, we plan to extend the evaluation to GPUs. Additionally, to make our approach practically usable, we will integrate it into a library-based runtime system designed for user support of heterogeneous architectures, thereby automating the instrumentation process, the search for relevant performance metrics and the analysis of the results.

References

[Ar13] Arulraj, J.; Chang, P.-C.; Jin, G.; Lu, S.: Production-run Software Failure Diagnosis via Hardware Performance Counters. SIGARCH Comput. Archit. News 41/1, pp. 101–112, Mar. 2013, ISSN: 0163-5964, URL: http://doi.acm.org/10.1145/2490301.2451128.

[Be08] Bergman, K.; Borkar, S.; Campbell, D.; Carlson, W.; Dally, W.; Denneau, M.; Franzon, P.; Harrod, W.; Hiller, J.; Karp, S.; Keckler S. an d Klein, D.; Lucas, R.; Richards, M.; Scarpelli, A.; Scott, S.; Snavely, A.; Sterling, T.; Williams, R. S.; Yelick, K.; Kogge, P.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead, 2008.

[Ha09] Hari, S. K. S.; Li, M.; Ramachandran, P.; Choi, B.; Adve, S. V.: mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Pp. 122–132, Dec. 2009.

[NCC07] Narayanasamy, S.; Coskun, A. K.; Calder, B.: Transient Fault Prediction Based on Anomalies in Processor Events. In: 2007 Design, Automation Test in Europe Conference Exhibition. Pp. 1–6, Apr. 2007.

[Ne18] Netti, A.; Kiziltan, Z.; Babaoglu, Ö.; Sîrbu, A.; Bartolini, A.; Borghesi, A.: FINJ: A Fault Injection Tool for HPC Systems. CoRR abs/1807.10056/, 2018, arXiv: 1807.10056, URL: http://arxiv.org/abs/1807.10056.

[SLM10] Salfner, F.; Lenk, M.; Malek, M.: A Survey of Online Failure Prediction Methods. ACM Comput. Surv. 42/3, 10:1–10:42, Mar. 2010, ISSN: 0360-0300, URL: http://doi.acm.org/10.1145/1670679.1670680.

[SS02] Schiffmann, W.; Schmitz, R.: Technische Informatik 2. Springer Berlin Heidelberg, 2002.

[Te10] Terpstra, D.; Jagode, H.; You, H.; Dongarra, J.: Collecting Performance Data with PAPI-C. In (Müller, M. S.; Resch, M. M.; Schulz, A.; Nagel, W. E., eds.): Tools for High Performance Computing 2009. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 157–173, 2010, ISBN: 978-3-642-11261-4.

[Tu17] Tuncer, O.; Ates, E.; Zhang, Y.; Turk, A.; Brandt, J.; Leung, V. J.; Egele, M.; Coskun, A. K.: Diagnosing Performance Variations in HPC Applications Using Machine Learning. In (Kunkel, J. M.; Yokota, R.; Balaji, P.; Keyes, D., eds.): High Performance Computing. Springer International Publishing, Cham, pp. 355–373, 2017, ISBN: 978-3-319-58667-0.

[WE47] WELCH, B. L.: THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED. Biometrika 34/1-2, pp. 28–35, Jan. 1947, ISSN: 0006-3444, eprint: http://oup.prod.sis.lan/biomet/article-pdf/34/1-2/28/553093/34-1-2-28.pdf, URL: https://doi.org/10.1093/biomet/34.1-2.28.

[WP05] Wang, N. J.; Patel, S. J.: ReStore: symptom based soft error detection in microprocessors. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). Pp. 30–39, June 2005.

[WPN07] Williams, A. W.; Pertet, S. M.; Narasimhan, P.: Tiresias: Black-Box Failure Prediction in Distributed Systems. In: 2007 IEEE International Parallel and Distributed Processing Symposium. Pp. 1–8, Mar. 2007.