Comparison of Thread Signatures for Error Detection in Hybrid Multi-cores

Sebastian Meisner and Marco Platzner
University of Paderborn, Germany
{sebastian.meisner, platzner}@upb.de

Abstract — Dynamic thread duplication is a known redundancy technique for multi-cores. We have adopted this concept to reconfigurable hardware cores running hardware threads in a hybrid multi-core system. For hardware threads we can compare three different signatures: the sequence of operating system (OS) function calls, the sequence of OS function calls with their parameters, and additionally all memory accesses with their type and data. These signatures allow for an increasing error detection coverage, but also come with increasing performance overheads, which enables an application designer to trade-off the error detection coverage for performance. Our experiments with three benchmark applications show that the best error coverage can be achieved at a performance cost between 2% and 55%, depending on the benchmark, the signature used and the utilization of the CPU.

I. Introduction

Hybrid multi-cores are single-chip systems comprising software programmable cores with reconfigurable hardware cores. Hybrid multi-cores are particularly interesting for applications that can benefit from different core types and customized accelerators in terms of performance and energy-efficiency. With their architectures and logic capacities, todays platform FPGAs enable the implementation of hybrid multi-core systems with dozens and even hundreds of cores. Such hybrid multi-cores are ideal platforms for the multi-threaded programming model, which has been explored by several approaches, e.g., Hthreads [1] and ReconOS [2], [3]. These approaches introduce the concept of a hardware thread (HWT), which basically is a reconfigurable hardware module turned into a thread, i.e., extended to implement OS-like thread semantics. However, due to the continued scaling of nanoelectronic technology, hybrid multi-cores—as any other digital design—will experience a higher failure rate in future [4]. FPGAs already experience a variety of fault modes, for instance permanent faults such as wear over time [4] and transient faults caused, for example, by radiation [5]. Hence, techniques are required for detecting and subsequently dealing with errors in hybrid multi-cores. While at the thread level several approaches have been presented for software programmable cores, dealing with faults in reconfigurable hardware cores has received less attention so far. Typical countermeasures implemented at the hardware or/and the software level rely on redundancy to detect and, optionally, compensate for errors. Dual modular redundancy (DMR) and triple modular redundancy (TMR) are popular schemes that can be implemented at different system levels. For example, at the level of FPGA circuits, the BYU tool [6] takes EDIF netlists as input and applies TMR at look-up table level. R3TOS [7] focusses on redundancy on the module and system level and replicates complete CPUs and voters to detect and correct errors. At the software layer Dobel et al. [8] implement TMR for software threads, while PLR [9] addresses the process level. While software-based redundancy techniques require less implementation effort than hardware-based ones, they typically suffer from enormous slowdowns due to the need for intercepting memory accesses for all thread or process copies.

Consequently, we discuss in this paper an approach for detecting errors in HWTs using a dynamic redundancy technique. We select a HWT to be checked, the thread-under-observation (TUO), and temporarily run a copy of this HWT, the shadowing thread (ST), in the multi-core architecture. During shadowing, we compare the so-called signatures of both threads and interpret any deviation in the signatures as a failure in one of the threads. These signatures are subdivided into three levels and allow for an increasing error detection coverage with increasing level, but also come with increasing performance and/or area overheads. Our error detection framework provides flexibility and enables an application designer to trade error detection coverage for performance needed. To use our framework the HWTs do not have to be modified.

As a new contribution of this paper we present the concept and multi-core implementation of a dynamic error detection mechanism for HWTs that can use *three variants of signatures* for detecting errors as well as an *experimental comparison* of the variants in terms of slowdown, error detection latency and resource usage for three benchmarks with different characteristics. In contrast to related work using a similar approach [10], we introduce the novel memory access signature that is, in combination with other signature types, the best possible signature as it covers the complete behaviour of a thread towards the OS and system memory.

II. Signature Levels Concept and Implementation

We have extended ReconOS [3] with our error detection technique. ReconOS is an architecture, programming model, and execution environment for run-time reconfigurable hybrid multi-cores and it extends the multi-threading model from software to HWTs. Figure 1 shows the our used ReconOS architecture with a CPU, two hardware slots, peripherals and

978-1-4673-9091-0/15/$31.00

memory controller mapped onto a platform FPGA. The CPU runs the host OS, in particular its kernel, and the application's software threads. A hardware slot is a partially reconfigurable region that can accommodate a HWT. The OS calls of a HWT are communicated via its operating system interface (OSIF) and executed by a delegate thread. HWTs can also access main memory through the memory interface (MEMIF).

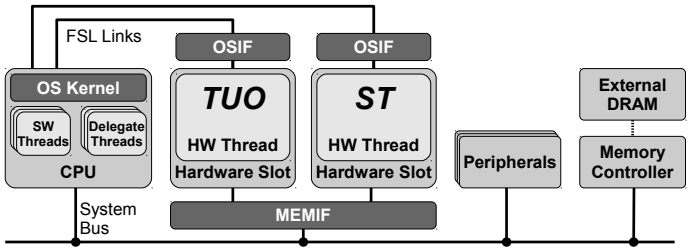

Fig. 1. ReconOS architecture with one CPU and two hardware slots accommodating a thread under observation (TUO) and a shadowing thread (ST).

Looking at the ReconOS architecture, we can split the interaction of a HWT into two categories. Calling an OS function results in communication over the OSIF, which includes for each call the components function name or identifier, respectively, the parameters and return values. Reading from and writing to memory results in communication over the MEMIF, which includes for each access as components the type, address and data to be read or written. We define as *signature level* a set of components that can be observed to gain information about the behavior of a thread. The *hardware thread signature* is then the actual sequence of observed interactions of a thread.

TABLE I
OVERVIEW OF SIGNATURE LEVELS FOR HARDWARE THREAD SHADOWING.

Signature Level	OS function calls (OSIF)			Memory accesses (MEMIF)		
	Name	Params	Ret. Val.	R/W	Addr.	Data
#1	✓					
#2	✓	✓	✓			
#3	✓	✓	✓	✓	✓	✓

Table I summarizes the components included in the three signature levels. The three levels provide a trade-off between error coverage and performance, i.e., higher levels are able to catch more errors but slow down a HWT and require more resources to implement the shadowing technique. *Signature level #1* checks whether the threads issue the same sequence of OS functions. Signatures of level #1 are light-weight and allow us to recognize mainly failures in the control flow. *Signature level #2*, when compared to level #1, provides a higher error coverage and allows us to check for wrong results in many HWT's registers. *Signature level #3* involves the OSIF and the MEMIF and provides the best possible error coverage by checking all communication a HWT can have toward the OS and memory.

Our shadowing system has to intercept HWT communication over both the OSIF and the MEMIF for implementing the three signature levels and to replay the captured data properly to the ST. The following paragraphs detail the technical realization of this interception mechanisms.

For intercepting and comparing OS function calls we adopted the work from [10] and extended it to include the signature levels. In short, the shadowing runtime system running on the CPU intercepts all OS function calls and maintains a FIFO buffer to store those calls. The TUO writes to the FIFO and the ST reads from it and compares its contents to its own function calls. ST function calls are not propagated to the OS kernel, but instead answered with the help of the data stored in the FIFO entry. This mechanism is independent from the signature level and ensures that the ST is always provided with the same return values as the TUO. What changes with the signature level is the amount of data stored and checked. For signature level #1 the shadowing system stores and compares the function names, for signature level #2 additionally the parameters and return values. If these values differ, the shadowing system issues an error message or, alternatively, calls a user defined function. The FIFO decouples the TUO and the ST and ensures that both threads can run independently until the FIFO fills up. The FIFO depth is configurable and allows for trading between performance, i.e., the TUO is not immediately slowed down by a slower ST, and error detection latency. In our experiments, we have used a shadowing system implementation with a 512-entry FIFO.

For intercepting and comparing memory accesses of the TUO and the ST we had to extend the original MEMIF of ReconOS. The ReconOS MEMIF uses a packet-based protocol that allows for read and write requests to main memory. Each read and write request packet comprises a header with the type of the request, length and address and, for writes, the data. The MEMIF of the default ReconOS system supports up to 16 HWTs and implements a round robin protocol to maintain a fair sharing of bus accesses.

We had to extend the MEMIF to guarantee that the ST always reads the same data from memory as the TUO and that ST writes do not reach main memory, but are compared against the written data of TUO instead. To fulfill these requirements we have implemented two new MEMIF arbiters.

The first arbiter implementation is denoted as *arbiter_dlat* since it minimizes the error detection latency. TUO and ST are synchronized on every memory access. Errors in the MEMIF packet header are immediately detected by comparing with the ST's MEMIF packet header and are not propagated to main memory. Data errors are detected while on flight to main memory. Even if a data error is detected, the rest of the write request is completed. In either case an error message is sent to the CPU detailing the kind of error and the position in the data stream.

The second arbiter implementation is called *arbiter_perf* since it aims at maximizing the TUO performance by not slowing it down too much. To that end, TUO and ST are decoupled by a FIFO buffer. Similar to the shadowing system on the CPU, the FIFO allows the TUO to advance further than the ST at the cost of a higher error detection latency. The TUO writes all memory access requests, including all read or

TABLE II
HARDWARE THREAD CHARACTERISTICS FOR OUR BENCHMARK APPLICATIONS.

Benchmark	# OS function calls (OSIF)	Memory accesses (MEMIF)				Total computation cycles	Average computation cycles per second
		Reads	Writes	Data read	Data written		
matrixmul	3	132	128	128 KiB	64 KiB	49	9.16
sort	4	1	1	8 KiB	8 KiB	512	8.33
gsm	3	4	2	716 to 1144 Byte	674 to 888 Byte	28976	357.72

written data, into the FIFO. When the ST issues a memory access request, the packet is compared to the packet in the FIFO. On read, the data stored in the FIFO is sent to the ST and on write, the data written by the TUO is compared to the ST's data. In our implementation we have chosen a FIFO buffer size of 32 KB. In general, small buffer sizes lead to a TUO slowdown and larger buffers consume more resources.

III. EXPERIMENTS

We have conducted experiments on a ReconOS implementation extended for thread shadowing running under Linux kernel version 2.6.37 on the Xilinx ML605 Evaluation Kit The CPU and the two hardware slots have been clocked at 100 MHz. We have conducted experiments with three benchmarks: *matrixmul*, *sort* and *gsm*. All benchmark applications instantiate a HWT for processing and run several computation cycles until all input data is processed. The benchmarks have been chosen to be quite diverse in their needs for calling OS functions and accessing memory. Table II lists the HWT characteristics for the benchmarks. The number of OS function calls and memory accesses are given per application cycle and the the average computation cycles per second are measured in a system without shadowing. The *matrixmul* benchmark multiplies matrices using the Strassen algorithm [11] to split the multiplication of a bigger matrix into several multiplications of smaller matrices. The HWT works on integer matrices of 128 columns $\times$ 128 rows, resulting in 64 KiB per matrix. The *sort* benchmark sorts an array of integers by splitting it in blocks of 8 KiB, sorting the blocks with the HWTs, and joining the results via a merge sort on the CPU. The *gsm* benchmark is part of the well-known embedded MiBench benchmark [12]. We have ported the original benchmark code to ReconOS and turned it into a software/hardware implementation, where it computes the short term synthesis filtering of a GSM audio stream for noise reduction with the help of a HWT. We have conducted a set of experiments to quantify the effects of our error detection technique with different HWT signature levels on the runtime of the TUO, the error detection latency, and the required logic resources. We have run each benchmark in each shadowing configuration for 10 times for averaging the results.

Figure 2 shows the slowdowns (formally, the speedups) for our benchmarks under different shadowing system configurations. The figure includes error bars showing the standard deviation, which ranges from 5.1×10^{-4} to 8.2×10^{-3}. The *baseline_run* measurements have been conducted with a non-shadowed version of the benchmarks and the ReconOS default arbiter and act as a reference point. The *baseline_sh_off_run* measurements have been done with a shadowing implementation that only intercepts OS function calls via the OSIF but not the memory accesses via the MEMIF. There is no ST running and there is no comparison of signatures. This variant basically measures the overhead of OS function call interception and handling. The next two measurements, *baseline_sh_on_lvl1_run* and *baseline_sh_on_lvl2_run* are for shadowing configurations that run an ST and perform comparisons of signatures of level #1 and #2, respectively, but still without intercepting memory accesses. The next six measurements are for full shadowing configurations with interception of OSIF and MEMIF communication and signature comparisons at all three signature levels. Measurements *dlat_* are for the MEMIF arbiter that optimizes error detection latency, and measurements *perf_* for the MEMIF arbiter that minimizes TUO slowdown.

Generally, the experimental results underline that increasing the signature level decreases the TUO performance. However, the slowdown depends more on the benchmark than on the signature level. While the *matrixmul* and *sort* benchmarks are slowed down by 2% at worst by shadowing, the *gsm* benchmark is slowed down up to 55%. This is because it performs a much higher number of OS function calls during its runtime, which can also be seen in Table II (OS calls $\times$ computation cycles). Hence, it uses the CPU up to capacity already at the *baseline_run* measurement, and the additional load of shadowing directly affects its performance. Comparing the two MEMIF arbiters, *arbiter_dlat* is always slower or as fast as the arbiter *arbiter_perf*, although the performance difference is rather small for all our benchmarks.

Figure 3 shows the minimum, average and maximum error detection latencies of all benchmarks in microseconds. The error bars indicate a standard deviation from 83.24 us to 9483.84 us. We define the error detection latency as the time difference between the ST and TUO function calls. The detection latency can be positive, when the ST follows the TUO, or negative in case the ST is blocked waiting for the TUO. The earliest possible point in time to detect an error is when the TUO issues an OS function call or accesses the memory. In this sense our error detection latency measures the additional delay in excess to this time. The results resemble the trends in Figure 2: While *matrixmul* and *sort* experience small detection latencies in the range of roughly 0.5 to 13 ms, the *gsm* benchmark experiences latencies of up to 67.8 ms. Again, this increased latency is caused by the high load of the system CPU.

Table III shows the logic resource overheads of the three arbiter implementations. Most of the resource increases comes from the error detection and handling. Additionally, the *arbiter_perf* implements a 32 KB FIFO buffer in distributed

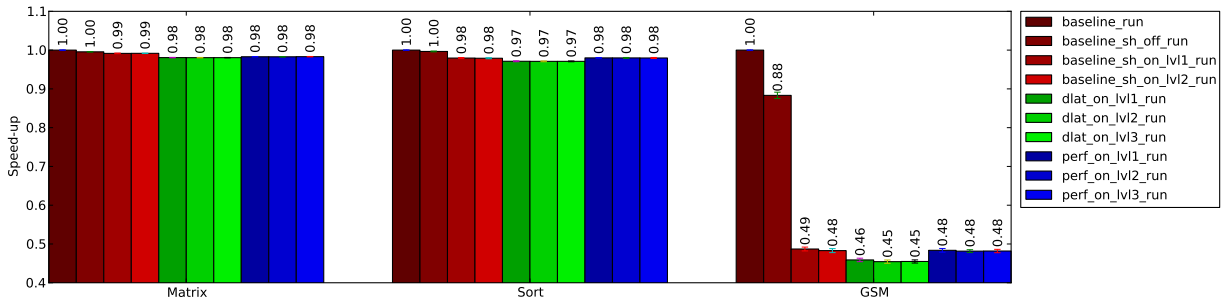

Fig. 2. TUO slowdown for our error detection technique, depending on the shadowing configuration, signature level and benchmark.

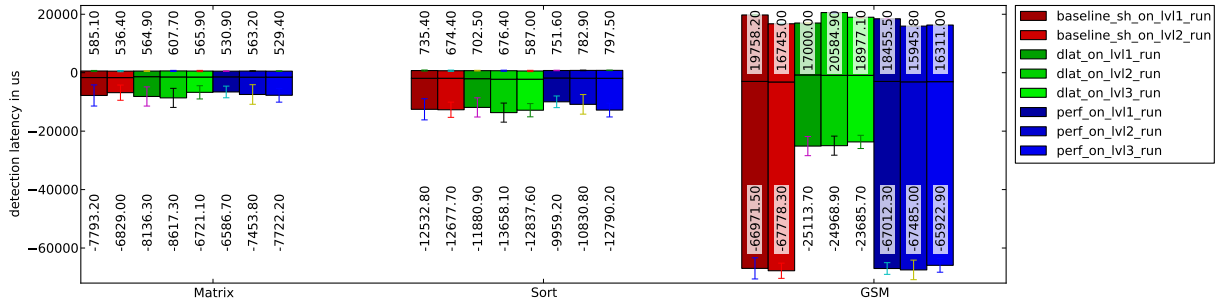

Fig. 3. Detection latency for our error detection technique, depending on the shadowing configuration, signature level and benchmark.

TABLE III
RESOURCE CONSUMPTION OF THE MEMIF ARBITER IMPLEMENTATIONS.

MEMIF Abiter	LUTs	% FPGA	Registers	% FPGA
default ReconOS	211	0.13 %	133	0.04 %
arbiter_dlat	2089	1.39 %	531	0.18 %
arbiter_perf	9075	6.02 %	661	0.22 %

memory which leads to a high LUT count. Comparing the resource requirements to the total available resources on our FPGA, the increase in resource usage seems acceptable.

IV. CONCLUSION

In this paper we have presented our concept and implementation for dynamic error detection in hardware threads of hybrid multi-cores. We have introduced three different signature levels for checking OS functions calls and accesses to system memory. We have experimented with three benchmarks and discussed the slowdown for the thread under observation, error detection latency and resource usage for using our error detection technique. There are two main insights. First, in strong contrast to software-based systems, our implementation for intercepting and comparing memory accesses of hardware threads is very efficient and has only small impact on the performance of the thread under observation. Second, the load on the system CPU is a critical factor. Benchmarks that frequently call OS functions increase CPU load which, in turn, can severely slow down the hardware thread under observation. Future work could therefore focus on spending a separate CPU or even a hardware module for supporting the error detection mechanism.

REFERENCES

[1] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A computational model for reconfigurable devices," in *International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2006, pp. 1–4.

[2] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "Reconos: An operating system approach for reconfigurable computing," *Micro, IEEE*, vol. 34, no. 1, pp. 60–71, Jan 2014.

[3] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 1, October 2009.

[4] E. A. Stott, J. S. Wong, P. Sedcole, and P. Y. Cheung, "Degradation in FPGAs: measurement and modelling," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*. New York, NY, USA: ACM, 2010, pp. 229–238.

[5] T. Karnik and P. Hazucha, "Characterization of soft errors caused by single event upsets in cmos processes," in *IEEE Transactions on Dependable and secure Computing*, vol. 1, no. 2, april-june 2004, pp. 128 – 143.

[6] J. M. Johnson and M. J. Wirthlin, "Voter Insertion Algorithms for FPGA Designs Using Triple Modular Redundancy," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. New York, NY, USA: ACM, 2010, pp. 249–258.

[7] C. Hong, K. Benkrid, X. Iturbe, and A. Ebrahim, "Design and implementation of fault-tolerant soft processors on FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 683–686.

[8] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT)*. New York, NY, USA: ACM, 2012, pp. 83–92.

[9] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135 –148, april-june 2009.

[10] S. Meisner and M. Platzner, "Thread shadowing: Using dynamic redundancy on hybrid multi-cores for error detection," in *Proceedings of the International Symposium on Applied Reconfigurable Computing*, ser. ARC, April 2014.

[11] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE International Workshop on Proceedings of the Workload Characterization (WWC)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.