Thread Shadowing: Using Dynamic Redundancy on Hybrid Multi-cores for Error Detection

Sebastian Meisner and Marco Platzner

Computer Engineering Research Group, University of Paderborn, Germany
{sebastian.meisner, platzner}@upb.de

Abstract. Dynamic thread duplication is a known redundancy technique for multi-cores. The approach duplicates a thread under observation for some time period and compares the signatures of the two threads to detect errors. Hybrid multi-cores, typically implemented on platform FPGAs, enable the unique option of running the thread under observation and its copy in different modalities, i.e., software and hardware. We denote our dynamic redundancy technique on hybrid multi-cores as *thread shadowing*. In this paper we present the concept of thread shadowing and an implementation on a multi-threaded hybrid multi-core architecture. We report on experiments with a block-processing application and demonstrate the overheads, detection latencies and coverage for a range of thread shadowing modes. The results show that trans-modal thread shadowing, although bearing long detection latencies, offers attractive coverage at a low overhead.

1 Introduction

Hybrid multi-cores combine instruction set based cores that are software-programmable with cores that are implemented in reconfigurable logic. Much like heterogeneous multi-cores, which typically combine different CPU cores, hybrid multi-cores are of interest for applications that exhibit parallelism at the thread-level and can benefit from mapping the threads to different core types in order to improve performance or energy efficiency. A main challenge for hybrid multi-cores is to create a software architecture that allows for a convenient integration of reconfigurable logic cores with software-programmable cores. In our work we leverage ReconOS [1], a programming model and runtime environment that extends multithreading to reconfigurable logic cores. In ReconOS, cores mapped to reconfigurable logic are turned into so-called hardware threads that can call an operating system running on the system's main CPU much like a software thread. In general, a ReconOS system can comprise several CPUs and reconfigurable hardware cores and run dynamically created software and hardware threads on their respective cores.

This work focuses on detecting errors at the thread level in single-chip hybrid multi-cores. Such errors arise when faults at the physical level are not masked and thus propagate up to the level of application threads. There are several causes for faults. Continuously shrinking microelectronic device structures lead to an

increase in components per chip area. This increase in functional density comes at the cost of reduced reliability due to increasing variations in device behavior and device degradation, as described by Borkar [2]. One particularly important source of unreliability is heat. Thermal hot spots and extensive temperature swings should be avoided since these effects accelerate aging which in turn leads to degradation [3] and, eventually, to total chip failure. Another and external cause for faults are single-event upsets [4]. Especially FPGAs, which are currently the main implementation platform for hybrid multi-cores, are vulnerable to single-event upsets since they store their configuration data in SRAM cells.

One traditional approach to detect faults is dual modular redundancy (DMR). DMR duplicates elements of a system and compares their results. This approach can be applied at hardware and software levels. At hardware level, the simplest form is lockstep execution [5]. A number of improvements over simple DMR have been presented to balance the trade-off between area/energy consumption and error detection rate [6–8]. At software level, DMR can be applied to threads or processes. Compared to hardware, DMR at the software level offers higher flexibility, but comes with the challenge of encapsulation, as all interactions of the thread or process with the environment have to be observed and compared. Works like [9] and [10] present prototypes for thread and process level DMR under Linux and POSIX, respectively.

The main contribution of this paper is the presentation and evaluation of *thread shadowing*, a thread-level error detection technique for hybrid multi-cores. Thread shadowing is a dynamic redundancy technique that duplicates (shadows) a running software or hardware thread for some time period. During shadowing, we compare the signatures of the two threads and detect an error if they deviate. The novel option unique to hybrid multi-cores is *trans-modal error detection*, i.e., hardware threads can shadow software threads and vice versa.

Allowing for dynamic redundancy across the hardware-software boundary opens up new potential for optimizing efficiency and overheads, as well as opening new ways of design for reliable systems. Thread shadowing offers several advantages: it eliminates the need for dedicated redundant cores and can use idle cores of any modality. Additionally, error detection can be activated per thread, either permanently or on a spot sample basis, providing the means for application with mixed requirements.

2 Shadowing Prototype Implementation

2.1 ReconOS and Shadowing Extensions

Our work leverages ReconOS [1], a programming model and runtime environment that extends multithreading to reconfigurable logic cores. ReconOS builds on a host operating system such as Linux or eCos and distinguishes between hardware threads and software threads. Both thread types, denoted as thread modalities, can call operating system functions to interact with other threads and the operating system kernel using well known programming objects such

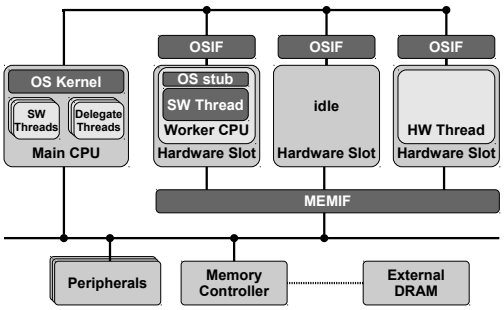

Fig. 1. Exemplary ReconOS architecture

Thread mgmt.:	`getinitdata(), yield(),exit()`
Semaphore:	`post(),wait()`
Mutex:	`lock(),unlock(), trylock()`
Cond. variable:	`wait(),signal(), broadcast()`
ReconOS queue:	`send(),receive()`
Mailbox:	`get(),put(), tryget(),tryput()`

Fig. 2. List of shadowed function calls.

as semaphores, message boxes and shared memory. Figure 1 depicts an exemplary ReconOS system architecture comprising a main CPU, three reconfigurable hardware slots, a memory controller and peripherals. Every hardware slot has two interfaces, an operating system interface (OSIF) for calling operating system functions and a memory interface (MEMIF) enabling direct access to the shared system memory. ReconOS uses a main CPU which runs an operating system kernel and user software threads. Hardware slots either accommodate hardware threads or worker CPUs that run additional software threads. Worker CPUs implement an operating system stub to embed their software thread into the multithreading environment. Hardware threads and software threads running on worker CPUs communicate with the operating system kernel by means of delegate threads. These delegates call operating system functions on behalf of their corresponding hardware threads or software threads on worker CPUs. Since ReconOS supports run-time reconfiguration, both thread types can be instantiated, loaded and started at run-time.

We have extended the ReconOS architecture and runtime system to monitor the calls listed in Figure 2 for thread synchronization, communication and management. We denote the original thread as *thread under observation* (TUO) and its duplicate as *shadow thread* (ST). The shadowing system is implemented in form of a user space library. The library substitutes each of the monitored functions with a version that wraps the original function and, in case shadowing is activated, implements function call tracing and comparison. At start-up, the runtime system creates all required STs and puts them to sleep state for later activation by the shadowing scheduler. While in our prototype this consumes some memory and, for hardware STs, a hardware slot, the ST activation time is greatly reduced. At runtime, the shadowing scheduler chooses a thread from the thread list to shadow.

2.2 Thread Signatures and Shadowing Schemes

As thread signature we use the sequence of OS-calls and their parameters. Since in ReconOS, all OS-calls issued by hardware threads and software threads on

worker CPUs are relayed by delegate threads, it is sufficient to monitor delegate threads and software threads on the main CPU. The data structure for one OS-call includes a pointer to the function name, the parameters, the return value and some meta data such as the lengths of data fields and a timestamp. For OS-calls that involve a pointer to a block of writable memory, the shadowing system creates a copy of the memory block for the ST. This way, the ST gets a pristine copy of the input data that can be modified independently of the TUO.

In this paper, we report on two different shadowing schemes. The first scheme shadows n TUOs by another n STs, with the characteristics that all TUOs are of the same modality, i.e., software or hardware, and all STs are of the same modality, i.e., software or hardware, as well. This shadowing scheme basically doubles the number of required cores but runs for each thread a permanent copy and thus fault detection covers the complete runtime, which makes it suitable for SEU detection. The second shadowing scheme shadows n TUOs of one modality with only one S in a round-robin fashion. This scheme performs error detection on a spot-sample basis, but requires only one additional core, which makes it preferable for permanent fault detection, when single errors are acceptable. We have selected these two schemes for presentation in this paper since they represent interesting corner cases. Obviously, there a many more schemes with arbitrary modalities for the single TUOs and STs.

3 Experimental Evaluation

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, which is equipped with a Virtex-6 LX240T FPGA. We have set up a static architecture with a MicroBlaze soft core as the main CPU, seven MicroBlaze worker CPUs for additional software threads and seven hardware slots for hardware threads. In the experiments shown in this section we use at most three worker cores and three hardware slots. The main CPU, the worker CPUs and all hardware slots have been clocked at 100 MHz. For testing, we have implemented a sorting application that sorts integers in 8 KiB blocks. The main application thread distributes the workload over several software and hardware sorting threads. A software sorting thread is able to sort data at a rate of 0.537 blocks/s; a hardware sorting thread sorts at a rate of 8.333 blocks/s. Therefore, using a hardware thread results in a speedup of 15.518. The number of software and hardware sorting threads used, as well as the number of blocks to be sorted are parameterized, but for the reported experiments we have fixed the number of blocks to 64. The sorting application communicates the data/results to/from the threads via ReconOS message queues which utilize only the OSIF (cmp. Figure 1). Since the shadowing system checks the OS-call names *and* parameters, all input and output, including the sorted data, is checked for consistency.

Figure 3 shows the simplified main loop of a sorting thread with its operating system interactions. The first `receive()` call returns the number of integers to be sorted, while the second `receive()` call provides the actual data to be sorted.

```
while (true)
{
 yield();
 receive(&recv_queue,
         &len, 4);
 if (len == UINT_MAX)
   { exit(); }
 receive(&recv_queue,
         buffer, len);
 /* sort buffer ... */
 send(&send_queue,
      buffer, len);
}
```

Fig. 3. Simplified main loop of the sorting thread.

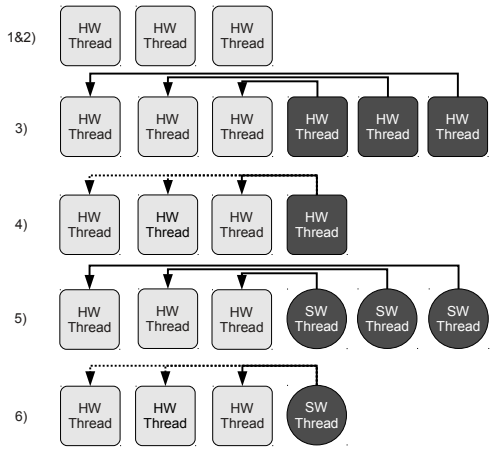

Fig. 4. Visualization of thread shadowing modes. Dark shapes are STs, light shapes are TUOs, and arrows indicate shadowing.

The sorted data is written back to the main thread via a `send()` call. Since the sorting application operates on blocks of data, there is actually no state between the processing of consecutive blocks. Such application models are widespread, especially in the signal processing domain. The shadowing scheduler can deactivate/activate a thread at the `yield()` call issued at the beginning of the thread's main loop. Based on the two shadowing schemes described in Section 2.2 we have experimented with the following six different shadowing modes:

1. *Original:* Reference sorting application on ReconOS without shadowing support for baseline comparison.
2. *Shadowing Off:* The shadowing system is in place, but no ST is activated. This mode measures the overhead of the shadowing system, i.e., the tracing of operating system calls.
3. *Shadowing On:* Intra-modal shadowing for all threads, i.e., n hardware STs shadow n hardware TUOs or n software STs shadow n software TUOs, respectively, with $n = 1 \ldots 3$.
4. *Shadowing Round-robin:* Intra-modal shadowing for all threads with one ST switched on every `yield()` call in a round-robin fashion, i.e., one hardware ST shadows n hardware TUOs or one software ST shadows n software TUOs, respectively, with $n = 1 \ldots 3$.
5. *Shadowing On Trans-modal:* Trans-modal shadowing for all threads, i.e., n hardware STs shadow n software TUOs or n software STs shadow n hardware TUOs, respectively, with $n = 1 \ldots 3$.
6. *Shadowing Round-robin Trans-modal:* Trans-modal shadowing for all threads with one ST switched on every `yield()` call in a round-robin fashion, i.e., one software ST shadows n hardware TUOs or one hardware ST shadows n software TUOs, respectively, with $n = 1 \ldots 3$.

Figure 4 visualizes these modes of operation for the case that the TUOs are hardware threads and $n = 3$. All modes are symmetrical for software threads as

TUOs. We have verified the correct functionality of our shadowing implementation by modifying the source code of one hardware and one software thread to include an error that leads to a different thread OS call signature. These erroneous threads have been used as TUOs and STs in varying configurations to successfully test the shadowing system. In the following, we report on three measured metrics: the slowdown of the TUOs inflicted by using thread shadowing, the time difference between identical OS-calls of TUO and ST, which we call the error detection latency, and the shadowing coverage. All data presented has been averaged over 10 runs of the application.

3.1 Runtimes

Thread Count	Software TUOs			Hardware TUOs		
	1	2	3	1	2	3
Original	1.00	1.00	1.00	1.00	1.00	1.00
Shadowing Off	1.00	1.00	1.00	1.02	1.02	1.01
Shadowing On	1.00	1.01	1.01	1.06	1.09	1.13
Shadowing Round-robin	1.00	1.00	1.00	1.07	1.03	1.02
Shadowing On Trans-modal	1.00	1.01	1.01	15.22	13.81	12.50
Shadowing RR Trans-modal	1.00	1.01	1.01	15.07	1.72	1.82

Table 1. Normalized runtimes of the sorting application under all shadowing modes over different numbers of software and hardware TUOs.

Table 1 shows the runtimes of the different shadowing modes for different system configurations, i.e., number of software and hardware threads. The data has been normalized to mode "Original" for the given number of TUOs. Using software TUOs, the absolute runtimes for sorting 64 data blocks in the "Original" mode decrease from 119.1 s, over 59.94 s to 41.46 s when going from one to three cores. Similarly, using hardware TUOs the absolute runtimes in the "Original" mode decrease from 7.73 s, over 4.24 s to 3.19 s. The data for software TUOs in Table 1 shows that the overhead posed by the shadowing scheme and the slowdowns for different shadowing modes are negligible with at most 1%. The data for hardware TUOs shows that the overhead posed by the shadowing system alone (Shadowing Off) is at most 2% and thus negligible. For intra-modal shadowing, i.e., hardware STs shadow hardware TUOs, the sort application is slowed down by 13% at maximum for three TUOs for mode "Shadowing On". In mode "Shadowing Round-robin" the slowdown reduces to 2% for three TUOs. As expected for the given application, trans-modal shadowing where software STs shadow hardware TUOs severely affects runtimes due to the speed difference between software and hardware threads, e.g., 1422% for shadowing one hardware thread by one software thread. In the trans-modal round-robin shadowing mode, the hardware TUOs are shadowed by one software thread only for a fraction of the overall runtime. Hence, the slowdown decreases with the number of hardware threads down to 82% for three TUOs.

3.2 Latencies and Coverage

To determine the error detection latency, we measure the time difference between two identical operating system calls of the TUO and the ST. Since our shadowing scheme is symmetrical with respect to TUO and ST roles, we report on positive values for the detection latency, where an ST lags behind the TUO, as well as negative values, where an ST actually called the operating system function earlier than its TUO. As our measurements show, the results differ significantly between intra-modal and trans-modal shadowing modes. While for intra-modal shadowing the latencies lie between $-0.35\,ms$ and $2.92\,ms$, trans-modal shadowing results in latencies increased by orders of magnitude, lying between $-578.24\,ms$ and $580\,ms$. These increased latencies are easily explained by the differences in execution speed of hardware and software implementations.

Thread Count	Software TUOs			Hardware TUOs		
	1	2	3	1	2	3
Shadowing Round-robin	100%	31.09%	17.81%	100%	31.09%	17.97%
Shadowing RR Trans-modal	100%	31.25%	17.66%	100%	4.69%	6.25%

Table 2. Average percentage of shadowed application cycles per thread in round-robin modes 4 and 6.

In the round-robin shadowing modes, the TUOs are not shadowed for the complete runtime. In order to quantify the coverage of shadowing, we measure the number of application cycles a TUO is actually shadowed and relate it to the overall number of the application cycles a TUO executes. In our sorting application, one application cycle consists of one iteration of the while-loop in Figure 3. Table 2 shows the percentage of shadowed application cycles. Obviously, if only one TUO is to be shadowed by one ST in a round-robin fashion the coverage is 100%. With an increasing number of TUOs shadowed by one ST in a round-robin fashion the coverage decreases. While one would expect that with n TUOs the coverage decreases to $\frac{1}{n}$, the measured coverage is lower since de-attaching and attaching STs to TUOs is always synchronized to the TUOs `yield()` operating system calls, thus adding a synchronization delay when changing the TUO. Another result is that hardware TUOs in trans-modal round-robin shadowing mode have a rather low coverage of around 5%. Since hardware TUOs are slowed down by their software STs, every other non-shadowed TUO is able to complete a lot of application cycles in this time period, thereby decreasing the number of potential application cycles for shadowing.

4 Conclusion and Future Work

In this paper we have presented thread shadowing, our thread-level dynamic redundancy technique for hybrid multi-cores that allows not only for intra-modal but also for the novel technique of trans-modal error detection. We have discussed its implementation on a ReconOS system. Our multi-core setup allows us to

systematically experiment with different shadowing schemes and determine their overhead, application slowdown, detection latency, and achieved coverage. In this paper we have studied several shadowing schemes for a sorting application and we have identified two interesting configurations: First, if one can accept the high cost for doubling the number of cores intra-modal shadowing for all threads results in with full coverage, minimal slow down and low error detection latency. A reasonable alternative that requires only one additional core is intra-modal round-robin shadowing at a somewhat reduced coverage. Second, the novel technique of trans-modal shadowing is an attractive option when hardware threads shadow software threads. Here, especially the trans-modal round-robin shadowing mode is very appealing since often a hardware thread implementation will be fast enough to shadow a number of software threads.

Future work will include experimenting with more applications, studying alternative thread signatures and setting up fault injection experiments to quantitatively characterize the effectivity of the different thread shadowing modes.

References

1. Lübbers, E., Platzner, M.: ReconOS: Multithreaded Programming for Reconfigurable Computers. ACM Transactions on Embedded Computing Systems (TECS) **9**(1) (October 2009)
2. Borkar, S.: Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. IEEE MICRO (November/December 2005) 10–16
3. Stott, E.A., Wong, J.S., Sedcole, P., Cheung, P.Y.: Degradation in FPGAs: measurement and modelling. In: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays. FPGA '10, New York, NY, USA, ACM (2010) 229–238
4. Lesea, A., Drimer, S., Fabula, J., Carmichael, C., Alfke, P.: The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. Device and Materials Reliability, IEEE Transactions on **5**(3) (sept. 2005) 317 – 328
5. IBM: PowerPC 750GX Lockstep Facility. Application Note (March 2008)
6. Austin, T.: DIVA: a reliable substrate for deep submicron microarchitecture design. In: Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on. (1999) 196 –207
7. Vadlamani, R., Zhao, J., Burleson, W., Tessier, R.: Multicore soft error rate stabilization using adaptive dual modular redundancy. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2010. (march 2010) 27 –32
8. Rodrigues, R., Koren, I., Kundu, S.: An Architecture to Enable Life Cycle Testing in CMPs. In: Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2011 IEEE International Symposium on. (oct. 2011) 341 –348
9. Mushtaq, H., Al-Ars, Z., Bertels, K.: A user-level library for fault tolerance on shared memory multicore systems. In: Design and Diagnostics of Electronic Circuits Systems (DDECS), 2012 IEEE 15th International Symposium on. (april 2012) 266 –269
10. Shye, A., Blomstedt, J., Moseley, T., Reddi, V., Connors, D.: PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures. Dependable and Secure Computing, IEEE Transactions on **6**(2) (april-june 2009) 135 –148