Thread Shadowing: On the Effectiveness of Error Detection at the Hardware Thread Level

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. Recent research applied this concept to hybrid multi-cores for error detection and introduced thread shadowing that runs hardware threads in the reconfigurable cores and compares their outputs for deviation at configurable signature levels. Previously published work evaluated this concept in terms of performance, error detection latency and resource consumption.

In this paper we report on the error detection capabilities of thread shadowing by presenting an extensive fault injection campaign. We employ the Xilinx Soft Error Mitigation Controller for fault injection and the Xilinx Essential Bit facility to limit the fault injections to relevant bits in the configuration bitstream. Our findings from fault injection experiments with a sorting benchmark are threefold: First, up to 98% of all errors are detected by the operating system of the hybrid multi-core supported by thread shadowing. Second, thread shadowing's signature levels provide a useful trade-off between detected errors and effort needed, with around 5% of all errors detected in calls to operating system functions and around 52% of errors detected in memory accesses of the hardware thread. Third, essential bit testing is effective and cuts down the amount of bits to be tested by a factor of 14.48 compared to the total amount of bits available in the configuration address space.

I. Introduction

Hybrid multi-cores are effective at leveraging the vast amounts of reconfigurable area available in todays Field Programmable Gate Arrays (FPGA). In contrast to heterogeneous multi-cores, which combine CPU cores using different instruction sets or, sometimes, cores using different implementations of the same instruction set, hybrid multi-cores combine instruction set based cores, such as CPUs, with cores implemented in reconfigurable logic. When combined with a suitable programming model, hybrid multi-cores yield ease of programming and high compute power at the same time. In particular, the multi-threading programming model for hybrid multi-cores has been investigated by several projects, e.g., Hthreads [1], ReconOS [2] and SPREAD[3] and has shown its feasibility.

However, shrinking transistor sizes combined with an increasing number of transistors per chip lead to an increased number of faults during production and run-time of hybrid multi-cores. According to S. Borkar [4] run-time monitoring of errors will be needed to keep future systems perform over a long period of time. Aside from production variations, run-time effects such as aging [5], radiation [6][7] and violations of operating parameters [8][9] introduce faults.

To this end, thread shadowing introduced error detection in hybrid multi-cores [10][11]. It uses dynamic dual modular redundancy (DMR) by attaching a *shadow thread (ST)* to a *thread-under-observation (TUO)*. All inputs to the TUO are mirrored to the ST and all outputs of TUO and ST are compared for deviations. The dynamic nature of thread shadowing allows to attach and deattach an ST to any TUO at run-time. This enables the error detection to be tailored to the current run-time needs of the system. Thread shadowing's speciality is that the ST may lag behind the TUO to avoid slowdowns of the TUO through synchronization. This may lead to an increased error detection latency and lower error detection coverage.

Our previous research focussed on performance, detection latency and resource implications of thread shadowing. The novelty of this paper is that we evaluate the *error detection capabilities* of thread shadowing. We are interested in learning how many of the possible errors in the FPGA configuration can be caught by either the operating system itself or by our thread shadowing system, also in dependency of the signature level chosen for thread comparison. To answer these questions, we perform an extensive fault injection campaign. The specific contributions of this paper are as follows:

- We show the effectiveness of thread shadowing towards detecting errors and the resulting distribution of errors across different error classes.
- Regarding the error numbers and classes found, we discuss the effectiveness of thread signatures for trading performance versus error detection coverage and their usefulness to adapt to different classes of applications.
- In contrast to most of the related work, we use a full production level system, comprising a full operating system on top of the test hardware. Therefore, we can monitor the behaviour of such a complex system under fault injection.

The remainder of the paper is structured as follows: Section II presents related work, while Section III gives a short introduction into thread shadowing concepts and implementation. Section IV then elaborates on our fault injection methodology. Section V finally presents and discusses the fault injection results. Section VI concludes the paper.

II. Background and Related Work

This section presents related work in the fields of error detection and mitigation methods and the corresponding test and evaluation techniques.

A. Redundancy for Error Detection and Correction

There are numerous error detection and mitigation methods available at both hardware and software level.

At hardware level we can distinguish between the granularity at which the methods are applied. In the FPGA context, Look-Up-Tables (LUT) and registers are the smallest elements at which dual (DMR) or triple modular redundancy (TMR) can be applied. DMR uses one replica and a comparator to detect errors, while TMR uses two replicas and a voter to correct a single error via a majority vote. The BYU tool [12] operates at this level: it takes EDIF netlists as input and applies TMR at look-up table level. The tool writes EDIF netlists, which can be fed back to the FPGA synthesis flow. At the next higher level complete (sub-)modules are replicated. For example R3TOS [13] and Re2DA [14] focus on redundancy on the module and system level and replicate complete CPUs and voters to detect and correct errors. An FPGA specific technique is scrubbing [15][16][17]. Scrubbing repeatedly updates, i.e., reconfigures, an FPGA's configuration data with the original one. Bit flips in the configuration plane are thus corrected. There exist scrubbing variants that read back the configuration data to check for errors before re-configuring. Scrubbing, however, is only useful against accumulation of errors in the bitstream. One scrubbing cycle may take hundreds of milliseconds, which is enough time for an error to propagate into the system.

At software level redundancy is typically applied at thread or process level, where a process can consist of several threads. Dobel et al. [18] implement TMR at the thread level, while several other works [19][20][21] address 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. Besides pure hardware and software implementations for error detection and recovery, there are also hybrid approaches. For example, Campagna and Violante [22] combine a hypervisor-based thread-level time redundancy with a hardware module that implements a data checker and a watchdog.

Thread shadowing [10][11] is also a hybrid approach since part of the replication and comparison is handled in software and another part is handled in hardware (see Section III for details). Thread shadowing is especially suitable for hybrid multi-cores. It is as dynamic as the underlying reconfigurable logic: as new hardware threads can be placed into or removed from reconfigurable logic, so can shadow threads be attached and removed from threads-under-observation. Additionally, thread shadowing does not suffer from the overheads of memory access interception like software-only approaches, since comparison of memory accesses is handled in hardware.

B. Reliability Test and Evaluation Methods

For test and evaluation of reliability measures such as redundancy or hardened designs, we need methods to inject faults into the FPGAs. Literature [23] offers numerous methods for fault injection. Beam testing [24][25] puts the target FPGA into the beam of strong ionizing radiation and provokes charge generation inside the silicon. The necessary equipment is big, expensive and requires extensive knowledge, experience and safety measures to operate. Atmosphere testing is used in the Rosetta project of Xilinx Inc. [6]. It uses arrays of FPGAs to measure the hit rate of high energy neutrons, which are generated in the upper atmosphere by high energy radiation. This method needs a high number of FPGAs and long time spans to generate statistically significant data. Additionally, the radiation strength is dependent on altitude and geographic longitude and latitude and therefore requires several locations to gain general insights. Other methods require less effort to implement. Accelerated ageing [5] is implemented by exceeding the specified maximum temperature and voltage specifications of an FPGA for an extended period of time. By the increased thermal and electrical stress the silicon structures age in weeks instead of years and show an increased error rate. An even stronger violation of operating parameters can also directly lead to faults. Overclocking of a design [8] or overheating [9] are examples of this approach. Finally, bitstream manipulation is an FPGA specific fault injection method in widespread use [26][27]. It employs the run-time reconfiguration feature of modern FPGAs to alter the FPGAs configuration and thereby alter the actual hardware implemented in the reconfigurable logic. Depending on the reconfiguration facility, faults may be injected into the hardware portion only or even into the user memory elements. This paper uses bitstream manipulation for fault injection for three reasons: First, it allows for reproducible results. Second, it gives direct control over the exact position of the fault injection. Third, it is much easier to use compared to the more involved methods mentioned above.

III. Thread Shadowing Concept and Implementation

In this section, we introduce the fault model and nomenclature to distinguish between the different stages of an error, and then present the details of thread shadowing and how it is implemented on top of ReconOS.

A. Fault Model and Nomenclature

Before introducing thread shadowing, we define the fault model and some terms in accordance with the resilience articulation point (RAP) model [28].

Physical sources are the origins of faults, errors and failures. They encompass all physical imperfections and effects that can disturb the operation of the circuit. Among others, radiation, electromagnetic interferences, operating parameter violations and ageing related mechanisms count as physical sources. If a physical source is strong enough it leads to a *fault*: a fault is a deviation of desired operation at transistor level. Examples are deviations from specified voltage and current levels or

physical changes in the silicon. A fault can lead to an *error*, which is a deviation from correct operation in the register-transfer-level view of a hardware architecture. The resilience articulation point (RAP) model ascribes all errors to bit-flips in a corresponding register or memory cell. For example, in an FPGA a change in the configuration memory could cause an error and by changing the configuration bitstream of the FPGA we can simulate the error. Finally, a *failure* is what can be seen at application level: aborted programs or corrupted results.

Masking may happen between adjacent levels: Physical sources may not develop into faults if the specific part of the silicon is not used. Faults may not propagate into an error if a current surge is so short that a register does not store it. And an error might not become a failure, for example if the register containing the error is overwritten before it is read.

Thread shadowing detects errors at the register-transfer-level. Since in the RAP model any fault articulates itself at this level, thread shadowing is able to detect all kinds of faults turning into errors: radiation induced single-event upsets (SEU) and even multiple-event upsets (MEU), aging related stuck-at-faults (SAF) and operating parameter violations induced errors. As long as the behaviour of the thread differs from its replica, even multiple errors appearing at the same time are detectable. The limitations of thread shadowing are the same as for any other DMR system: If an error leads to the same behaviour in the original thread as well as in its replica, it is undetectable.

B. Concept of Thread Shadowing

Thread shadowing is an error detection technique and framework for hybrid multi-cores. Since thread shadowing is based on ReconOS, hardware accelerators, for which we want to detect errors, are viewed and implemented analogously to software threads: the accelerators are active in the sense that they call operating system functions and access memory on their own. They also present themselves to the application developer like software threads. We have designed thread shadowing to impose only minimal work for the application developer and no additional work for the hardware developer.

The main elements in the concept of thread shadowing are the thread under observation (TUO) and the shadow thread (ST). The TUO is the target whose operation is checked by dynamically attaching a ST to it. All inputs to the TUO are mirrored to the ST and all outputs of TUO and ST are compared for deviations. A feature of thread shadowing is the loose coupling of TUO and ST. Any free thread with the same function as the TUO can serve as a ST. This makes it possible to reuse idling threads for error detection or assign a single ST to several TUOs in a round-robin fashion.

Another feature of thread shadowing are signature levels. Depending on application requirements, comparison of outputs can be set to three levels with increasing error coverage: Level 1 compares only names of function calls to the operating system, level 2 compares function call names and function parameters and, finally, level 3 adds comparison of memory accesses. This allows for balancing performance versus error detection coverage and for adapting to applications where only errors in the control path are critical.

It is the choice of the application designer to determine what actions are taken when thread shadowing reports an error. While some applications might require a 100% error free operation, others such as multi-media applications, might tolerate single errors in the output data. However, statistics have to be gathered and system reconfiguration may happen when application specific error levels are exceeded.

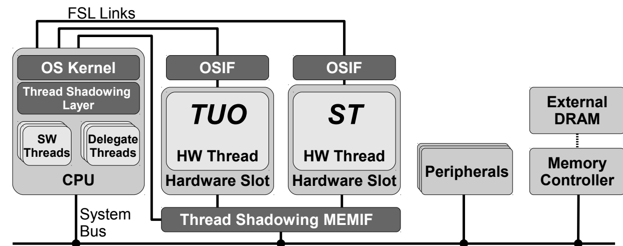

Fig. 1. ReconOS architecture extended to support thread shadowing.

C. Implementation of Thread Shadowing

ReconOS [2] serves as the implementation platform for thread shadowing. It is an architecture, programming model, and execution environment for run-time reconfigurable hybrid multi-cores. ReconOS builds upon widely used host operating systems such as Linux and Xilkernel and extends the multi-threading model from software to hardware threads. From an application programmer's perspective, a hardware thread looks like any other software thread and can use well-known operating system calls to communicate and synchronize with other threads and access shared system memory.

Thread shadowing builds upon the ReconOS infrastructure and extends it with dual modular redundancy. Figure 1 shows the architecture of a typical thread shadowing system. It depicts the main system CPU which runs the operating system (OS), the ReconOS delegate threads, which call OS functions on behalf of the hardware and the thread shadowing layer.

The hardware threads (HWT) run in hardware slots, which are preassigned areas in the FPGA's reconfigurable logic. HWTs communicate via two interfaces: the operating system interface (OSIF) and the memory interface (MEMIF). The OSIF enables the HWTs to call OS functions for synchronization via, e.g., semaphores and message boxes. The MEMIF enables direct access to shared system memory without involvement of the system CPU.

Thread shadowing extends the ReconOS architecture in two places: on the main system CPU and in the MEMIF. On the main system CPU, the newly added thread shadowing layer has several tasks:

- It starts STs and (de-)attaches them to/from TUOs.
- It schedules the ST in round-robin mode.
- It intercepts OS function calls from the delegate threads, compares them and relays them to the OS.

- It controls the signature levels.
- It configures the thread shadowing MEMIF and receives error information from it.
- It forwards error information to the application.

In the MEMIF we added additional hardware, which mirrors and compares memory accesses between TUO and ST. An additional Fast Simplex Link (FSL) between CPU and MEMIF allows for configuration and error reporting. This extended MEMIF avoids costly performance slowdowns found in pure software thread based systems [19][18].

IV. Fault injection setup

Running a fault injection campaign requires extensive preparations: Means to manipulate the configuration bitstream have to be integrated into the design, address lists of where to inject faults have to be created, a careful floorplan to limit the effects of fault injection has to be designed, and a robust test harness, which injects faults and records the results, has to be programmed. For our work, we leverage the Xilinx Soft Error Mitigation Controller and the Xilinx Essential Bits facility. In this section, we present and discuss our design choices for each step of the preparation.

A. Fault Injection via Xilinx Soft Error Mitigation Controller

Our fault injection is based on the Soft Error Mitigation Controller (SEM) from Xilinx [17]. Originally a module to implement scrubbing in the reconfigurable area of the FPGA, it also offers a simple interface for flipping a bit in the configuration bitstream of the FPGA. We have wrapped a Processor-Local-Bus (PLB) interface around the SEM and implemented a simple register based interface to its functionality. This way, we can attach the SEM easily to the system bus in the ReconOS architecture and extend the system with control over the fault injection.

B. Addressing

The SEM accepts only addresses in the physical address format. Physical addresses for the Xilinx Virtex 6 architecture [29] consist of 35 bits and contain the following fields from most significant bit to least significant bit: a 1 bit field which is always zero, 2 bit block type, 1 bit top or bottom half of FPGA, 5 bit row address, 8 bit column address, 7 bit minor address, 7 bit word address and finally a 5 bit wide bit address. Down to the level of a column address the physical position is documented via the Xilinx datasheets. Starting with the minor address, the position or functionality of configuration bits is Xilinx's proprietary knowledge.

However, when constraining (sub-)modules in the Xilinx user constraints file, one has to use slice, BRAM and DSP addresses which use different address spaces. Conversion between address spaces requires knowledge of the FPGA layout. We have implemented an address translation function using positional data extracted from PlanAhead as a reference.

C. Essential Bits

A result of the addressing subsection is that every column in a bitstream may have up to 524'288 bits where a fault might be injected. If one takes the durations needed to test every bit (see Section IV-D) into account, a fault injection campaign may take between 12 days for a run with no errors and 1092(!) days in case all faults result into errors. This is a prohibitively long time. Therefore, we have used the *essential bit file* facility provided by the Xilinx tools. The documentation [30] defines:

> Essential bits are defined [...] as those bits associated with the circuitry of the design, and are a subset of the device configuration bits.

By testing only the essential bits, we avoid testing configuration bits which are actually unused by our design. With an additional parameter to the Xilinx command line tool *bitgen*, it generates an **.ebd* file which matches the configuration bits and contains a '1' where a configuration bit is essential and a '0' when it is not. With knowledge of the FPGA layout and a visualization of the essential bit data, we have reverse-engineered the format of the file. Thus, we were able to extract the physical addresses of essential bits. Results are shown in Section V.

D. Test Setup

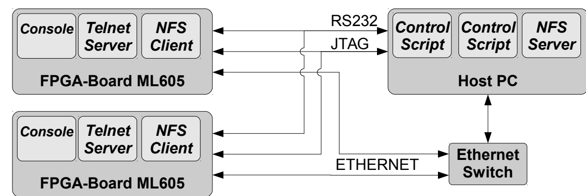

Fig. 2. Fault injection setup with one controlling Host PC and two FPGAs boards to speed up fault injection.

Figure 2 shows our fault injection setup. It consists of a host PC and two Xilinx ML605 FPGA boards. We are using two FPGA boards to split the set of fault injection addresses across the boards and thus reduce total testing time. Host and FPGAs are connected over several interfaces: JTAG for bitstream and Linux kernel download, Ethernet for network file system (NFS) access and telnet for remote shell access. The RS232 serial connection is not used by the control script, but remains as means for manual monitoring of the FPGA boards. For every board there is an instance of the control script. It steers download of the bitstream and Linux kernel and starts fault injection and the test program. Its main task is to record the error codes generated by the test program and to reset the FPGA board after an error is registered. Resetting the FPGA board includes re-downloading the bitstream and Linux kernel. With a duration of over 3 minutes this is a time-consuming operation, compared to an error-less fault injection and test program run of approximately 2 to 3 seconds.

Figure 3 shows the flowchart of our control script. The control script starts in the lower left with a list of fault

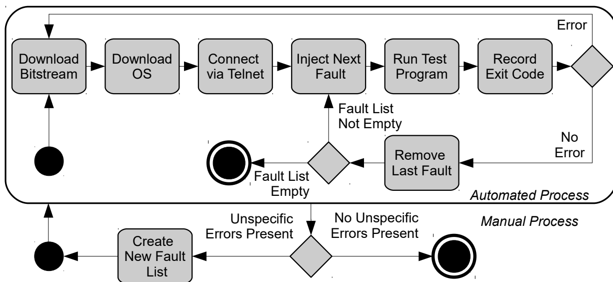

Fig. 3. Flowchart of our fault injection methodology.

addresses. Then, for the first address, it downloads the FPGA bitstream, operating system, and after the system has boot up, connects via telnet to it. Now, it injects a fault into the FPGA bitstream and then runs the test program. After the test program finished or a preset timeout, the control script records the exit code of the test program. Here, we use an optimization: In case no error was recorded, the script removes the last fault, injects the next one and re-runs the test program. Only after an error was recorded, we reset the whole FPGA. This reduces the fault injection time to 2 or 3 seconds in most cases, instead of 3 minutes for a whole reset. This automated process stops after every fault address on the list has been tested.

After the automated process we manually inspect the results and re-run the tests for fault addresses that resulted in unspecific error codes to differentiate between random errors of the test harness and actual errors due to fault injection. These unspecific error codes have their root cause in network instability or erroneous boot-up.

V. Experiments

After laying the groundwork, this section describes the benchmark application used, the FPGA layout chosen and the corresponding essential bits. Finally, it presents and discusses the results of our fault injection campaign.

A. Application Description and Placement

We have implemented the *sort* benchmark used in thread shadowing publications [10][11] as our test program. It 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 benchmark runs under Linux kernel version 2.6.37. For the fault injection campaign we have configured the system to use two hardware threads, one for the TUO role and one for the ST. The sort program was parameterized to sort only one 8 KiB block. Thread shadowing was configured to signature level 3, which includes comparison of OS function names, parameters and all memory accesses and to performance mode. In particular, this means that the ST can lag behind the TUO. As soon as the sort benchmark encountered an error, it quit operation and returned a unique error code for every error condition encountered during fault injection.

At the end of the program, the sort benchmark performs an additional sort purely in software on the system's main CPU. Thus, we are capable of comparing the results to the ones from the HWT and of identifying cases of silent data corruption (SDC) that might have not been detected by thread shadowing.

TABLE I
Resource utilization of the sort_demo HWT, constrained to three columns.

Resource Type	Available	Used	Utilization
LUT	960	769	81%
FD_LD	1920	412	22%
SLICEL	120	97	81%
SLICEM	120	97	81%
RAMBFIFO36E1	2	2	100%

To reduce the duration of the fault injection campaign we have constrained the sort hardware thread to use the smallest possible FPGA area. The resulting HWT fits into three columns and reaches a utilization of 81% of the available LUTs; see Table I for full details. A high utilization of resources per area is important to achieve a high error rate, since faults injected into unused FPGA resources will unlikely lead to an error.

B. Essential Bits

Figures 4 and 5 show the layout and the essential bits on the Virtex-6 LX240T FPGA we have used. Figure 4 shows the placement from PlanAhead. We have separated the base system and the hardware threads to make sure that injected faults result in errors in the HWTs and not in the base system. In the top left of Figure 4 we see the TUO over the ST, and the bottom half is reserved for the base system. The straight lines in the middle of the figure connect the base system to the external input/output blocks.

Figure 5 shows our interpretation of the essential bit file corresponding to our design. Every pixel of the image corresponds to one word (32 bits) of configuration data and the color corresponds to the amount of essential bits in that word: bright red means 32 essential bits, black means 1 essential bit and white means no essential bit. The layout correlates to the one in Figure 4 very well, except for red horizontal bands with unknown function. The top left shows the two hardware threads, the bottom is filled with the base system and in between we see patches of black which represent configuration bits for the reconfigurable wiring between base system and HWTs and base system and input/output blocks.

Figures 6 and 7 show the TUO in a magnified view. The TUO occupies, from left to right, two block-RAMs and three columns. Please note that, since we have no information about the actual meaning and physical position of a configuration bit below the level of a column, we choose our own layout: Every line in a column consists of 81 words, and every column is 36 minors high. To fit the aspect ratio of the original PlanAhead layout, we vertically stretched our essential bit visualization. Figure 8 shows the error density found by our fault injection

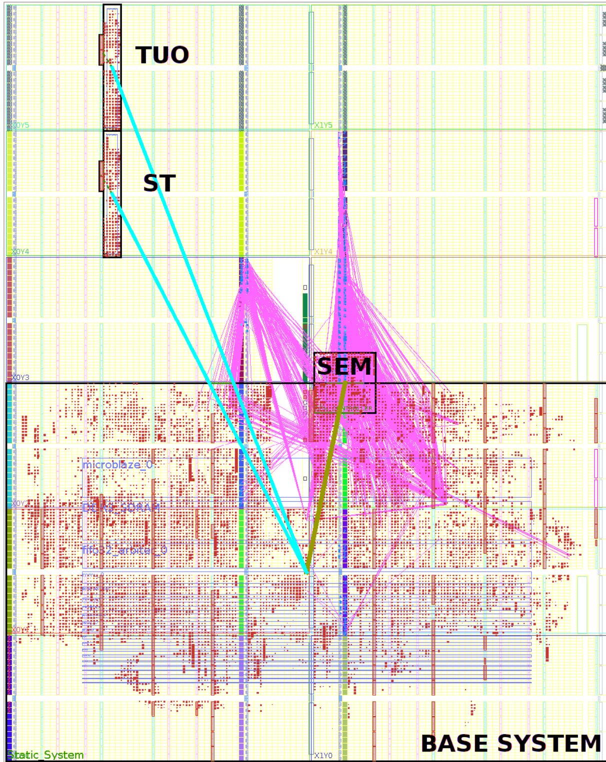

Fig. 4. Visualization of placement from PlanAhead for the whole FPGA.

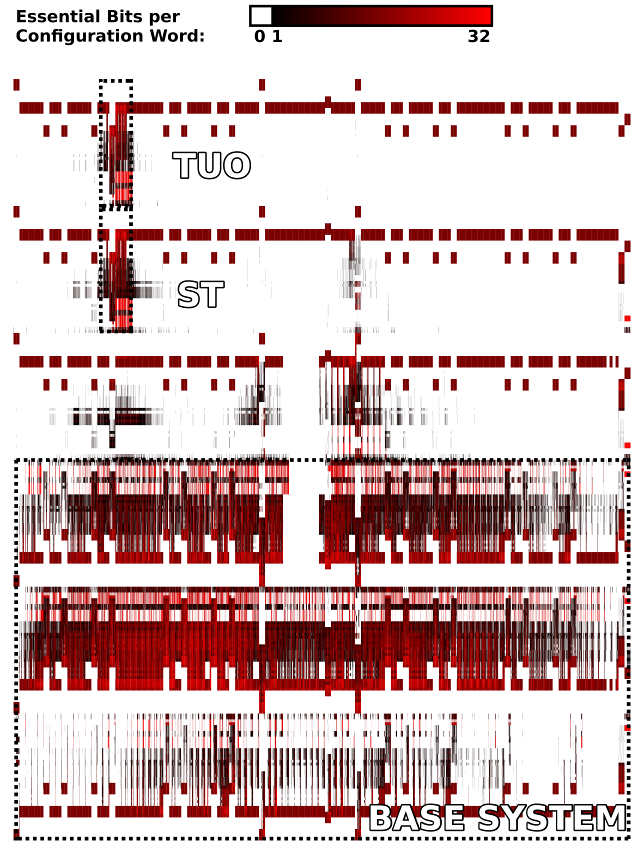

Fig. 5. Visualization of the essential bit data file for the whole FPGA.

TABLE II
FAULT INJECTION RESULTS: ERROR TYPES AND COUNTS

Error Classes	Column 1	Column 2	Column 3	Total	Total Share
Silent Data Corruption	100	33	3	136	1.79 %
OS Detected Errors	340	306	140	786	10.37 %
Timeouts	692	1337	299	2328	30.70 %
Thread Shadowing detected errors	1769	1506	1057	4332	57.14 %
Function call errors	61	239	87	387	5.10 %
Memory access errors	1708	1267	970	3945	52.03 %
Total Error Counts	2901	3182	1499	7582	
# Essential Bits and Faults Injected	43519	40874	24238	108631	
% Errors of Essential Bits	6.67	7.78	6.18	6.98	
Duration of fault injection campaign	9 days	7 days	6 days	22 days	

experiments (see next subsection for numerical results). It shows, that not every fault in an essential bit turned into an error, but error density correlates with essential bit density.

C. Fault Injection Results

We have conducted fault injection campaigns based on essential bits for every of the three columns of the TUO. The results are shown in Table II. We sum up all encountered errors in four categories, as follows:

- **Silent Data Corruption:** An actual error was not detected until the test program compared the HWT's results to known good results. The reason for these deviations lies in our shadowing mechanism that allows the ST lag behind the TUO for performance reasons. Since in our fault injection campaign we only sort one block of data, the sort benchmark observes the completion of the TUO and exits before the ST can report the error that it, in fact, had caught. While for longer running programs this should be less of an issue, we plan to introduce a barrier mechanism which blocks the main thread until all STs have completed as future work. Such a barrier mechanism will eliminate silent data corruption errors.

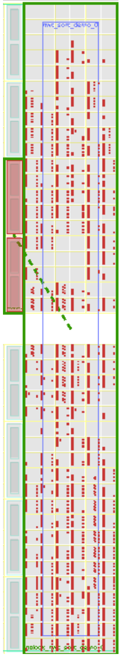

Fig. 6. Visualization of placement from PlanAhead for the TUO.

Fig. 7. Visualization of essential bits of the TUO.

Fig. 8. Visualization of error bits in the TUO (block-RAM column removed).

- **OS Detected Errors:** Operating system facilities detected illegal behaviour: this category encompasses mainly signals from the OS kernel to the test program indicating memory access violations and illegal instructions. Such errors do not even require our thread shadowing system and can be easily resolved by using appropriate signal handlers in the host OS.
- **Timeouts:** The program did not finish within a 30 seconds timeout period, which is approximately 15 times longer than the test program needs to finish. Such timeouts are caused by crashed threads and can in any operating system easily be checked by watchdog mechanisms.
- **Thread Shadowing Detected Errors:** The thread shadowing system reported the error. This includes errors in the OS function calls (signature levels 1 and 2) as well as memory access errors (signature level 3).

Table II lists the absolute error counts as well as relative numbers. We can see that the host OS was able to catch about 40% of all errors, including 30.7% timeouts and 10.37 % OS kernel signals raised due to errors. Our thread shadowing system detected further 57.14% of the errors, importantly these are errors that the host OS would not be able to recognize because they do not lead to a thread crash or access violation in the overall system. The remaining 1.79% of errors are silent data corruptions that appear in our current shadowing system but could easily be avoided.

Table II details the errors detected by thread shadowing into function call errors (signature levels 1 and 2) and memory errors (signature level 3). Memory errors outnumber function call errors by a factor of only 10. This is noteworthy since the HWT writes around 256 times more bytes to memory than needed for calling functions over the OSIF. However, these numbers will change with different HWTs that have different characteristics in function calls and memory accesses. The table also shows that, across all columns, less than 7% of all faults injected lead to an error. We have expected this value to be higher, since we have injected faults into the essential bits, which were calculated by the FPGA vendors tools.

A notable observation is the correlation between memory related errors, such as memory access errors and silent data corruption, and the distance to the block-RAM, which on the FPGA fabric is placed left of column 1 of the HWT. The closer the column is to the block-RAM, the more errors we see.

Testing every bit in the address space would have resulted in 1’572’864 fault addresses. Usage of essential bits reduced the amount of fault addresses to 108’631. This is a reduction by a factor of 14.48. Regarding the duration of the fault injection campaigns, we see that injecting faults into the essential bits saved a lot of time. If we assume 3 seconds per error free fault injection, full injection would have added 1’464’233 fault addresses, and thus additional 51 days to the already spent 22 days.

Experimentation times of 22 days may seem prohibitively long. On one hand we have to note that the fault injection process can be optimized for speed, in particular by performing experiments on several boards in parallel, to reduce validation time. This is part of our ongoing work. On the other hand, considering industry product development times, e.g., 7.5 years on average for a satellite [31], spending a few weeks for design validation seems to be feasible. Additionally, we believe that it is not necessary to conduct such fault injections experiments for all possible system configurations. Rather, these experiments help understand the effectiveness of our thread shadowing approach and classify the different errors caught into classes.

VI. Conclusion

In this paper we have prepared and conducted fault injection experiments to test the effectiveness of the previously presented thread shadowing mechanism for error detection at the hardware thread level. We have used the Xilinx Soft Error Mitigation Controller to inject errors and Xilinx facilities to extract essential bits from the configuration bitstream. Then, we have injected faults only into these essential bits to eliminate testing of unused configuration bits and thus speed up fault injection experiments.

Our measurements demonstrate the effectiveness of thread shadowing. The hybrid multi-core’s host operating system together with our thread shadowing layer were able to detect over 98% of the overall errors. The remaining 2% of errors result in silent data corruptions and are caused by our thread shadowing system that allows the shadowing threads to lag behind the threads-under-observation for performance reasons. As we have shown in previous work [11], letting the shadow thread lag behind results in slowdowns of only

2% to 3% compared to a system without shadowing. We can easily catch all errors by closer synchronizing the shadow thread and the thread under observation, but this comes with a performance penalty. Additionally, our experiments have shown that different signature levels are feasible and allow the application programmer to distinguish between control path and data path errors, where the former mostly manifest themselves in corrupted sequences and parameters of OS calls and the latter in wrong data written to memory.

Future work will extend our thread shadowing with a barrier mechanism that allows the main application thread to synchronize with the completion of all shadowing threads to avoid the error class of silent data corruption. Moreover, we will reduce the time needed for fault injection experiments by optimizing the test procedures and running experiments on several boards in parallel. Reduced validation times will allow us to evaluate the effectiveness of thread shadowing for a larger number of benchmark applications.

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] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "Spread: A streaming-based partially reconfigurable architecture and programming model," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2013.

[4] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10 – 16, nov.-dec. 2005.

[5] 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.

[6] A. Lesea, S. Drimer, J. Fabula, C. Carmichael, and P. Alfke, "The rosetta experiment: atmospheric soft error rate testing in differing technology fpgas," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 317 – 328, sept. 2005.

[7] M. Caffrey, K. Morgan, D. Roussel-Dupre, S. Robinson, A. Nelson, A. Salazar, M. Wirthlin, W. Howes, and D. Richins, "On-orbit flight results from the reconfigurable cibola flight experiment satellite (cfesat)," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, April 2009, pp. 3–10.

[8] K. Shi, D. Boland, and G. Constantinides, "Accuracy-performance tradeoffs on an fpga through overclocking," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, 2013, pp. 29–36.

[9] M. Happe, H. Hangmann, A. Agne, and C. Plessl, "Eight ways to put your fpga on fire - a systematic study of heat generators," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, 2012, pp. 1–6.

[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, pp. 283–290.

[11] ——, "Comparison of thread signatures for error detection in hybrid multi-cores," in *Field Programmable Technology (FPT), 2015 International Conference on*, ser. FPT, Dec 2015, pp. 212–215.

[12] 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.

[13] C. Hong, K. Benkrid, X. Iturbe, and A. Ebrahim, "Design and implementation of fault-tolerant soft processors on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 683–686.

[14] H.-M. Pham, L. Devaux, and P. Sbastien, "Reda: Reliable and reconfigurable dynamic architecture," in *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, ser. Reconfigurable Communication-centric Systems-on-Chip, L. S. Indrusiak, D. Goehringer, G. Sassatelli, and G. M. Almeida, Eds., 2011.

[15] F. Brosser, E. Milh, V. Geijer, and P. Larsson-Edefors, "Assessing scrubbing techniques for xilinx sram-based fpgas in space applications," in *Field-Programmable Technology (FPT), 2014 International Conference on*, Dec 2014, pp. 296–299.

[16] U. Legat, A. Biasizzo, and N. Francisco, "Self-reparable system on fpga for single event upset recovery," in *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, ser. Reconfigurable Communication-centric Systems-on-Chip, L. S. Indrusiak, D. Goehringer, G. Sassatelli, and G. M. Almeida, Eds., 2011.

[17] *LogiCORE IP Soft Error Mitigation Controller v4.1*, Xilinx Inc., 2014.

[18] 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.

[19] 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.

[20] C. Bolchini, A. Miele, and D. Sciuto, "An adaptive approach for online fault management in many-core architectures," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, ser. DATE '12, march 2012, pp. 1429 –1432.

[21] J. Pool, I. Sin, K. Wong, and D. Lie, "Relaxed determinism: Making redundant execution on multiprocessors practical," in *In Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS*, 2007.

[22] S. Campagna and M. Violante, "An hybrid architecture to detect transient faults in microprocessors: An experimental validation," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, ser. DATE '12, march 2012, pp. 1433 –1438.

[23] M.-C. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75 –82, apr 1997.

[24] 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.

[25] A. H. M. J. Wirthlin, H. Takai, "Soft error rate estimations of the kintex-7 fpga within the atlas liquid argon (lar) calorimeter," *Journal of Instrumentation*, vol. 9, 2014. [Online]. Available: http://iopscience.iop.org/1748-0221/9/01/C01025

[26] U. Kretzschmar, A. Astarloa, J. Jimenez, M. Garay, and J. Ser, "Fast and accurate single bit error injection into sram based fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 675–678.

[27] G. G. Cieslewski, A. D. George, and A. M. Jacobs, "Acceleration of fpga fault injection through multi-bit testing," in *Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, ser. ERSA, July 2010. [Online]. Available: http://www.chrec.ufl.edu/pubs/ERSA10_F6.pdf

[28] A. Herkersdorf, H. Aliee, M. Engel, M. Gla, C. Gimmler-Dumont, J. Henkel, V. B. Kleeberger, M. A. Kochte, J. M. Khn, D. Mueller-Gritschneder, S. R. Nassif, H. Rauchfuss, W. Rosenstiel, U. Schlichtmann, M. Shafique, M. B. Tahoori, J. Teich, N. Wehn, C. Weis, and H.-J. Wunderlich, "Resilience articulation point (rap): Cross-layer dependability modeling for nanometer system-on-chip resilience," *Microelectronics Reliability*, vol. 54, no. 67, pp. 1066 – 1074, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026271413004630

[29] *Virtex-6 FPGA Configuration User Guide (UG360)*, Xilinx Inc., 2015.

[30] *Soft Error Mitigation Using Prioritized Essential Bits*, Xilinx Inc., 2012.

[31] L. A. Davis and L. Filip, "How long does it take to develop and launch government satellite systems?" International Cost Estimating and Analysis Association" AEROSPACE REPORT, March 2015. [Online]. Available: http://www.iceaaonline.com/ready/wp-content/uploads/2014/03/Davis-Satellite-ICEAASoCal-090915.pdf