INTEGRATION, the VLSI journal 46 (2013) 265–279

Contents lists available at SciVerse ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

A column parity based fault detection mechanism for FIFO buffers

Isidoros Sideris *, Kiamal Pekmestzi

School of Electrical and Computer Engineering, National Technical University of Athens, 9 Heroon Polytechneiou, Athens 15780, Greece

ARTICLE INFO

Article history:
Received 27 September 2011
Received in revised form
18 January 2012
Accepted 28 March 2012
Available online 10 April 2012

Keywords:
FIFO
Reliability
Fault detection
Column parity
Dynamic verification

ABSTRACT

This paper presents a low cost fault detection mechanism for FIFO buffers. The scheme is based on column parity maintenance in a single register, which is updated by monitoring the values written to and read from the FIFO memory array. A non-zero column parity when the FIFO is empty, constitutes an indication of fault, and this property is exploited for fault detection. The technique has gains in area, power and critical path delay, at the expense of (1) greater detection latency, due to the need for the FIFO to become empty in order to assert a violation and (2) worse Silent Data Corruption (SDC) rate.

1. Introduction

First In First Out (FIFO) memories are used for buffering and flow control and are indispensable parts of almost every system. They are widely used in on chip communication fabric, high speed communication protocol implementations, image/video accelerators [1], multiple channel DMA controllers and coprocessor interfaces (e.g. Xilinx FSL [2]). What is more, they are extensively used between different clock domains for synchronization. Modern multicore processors require different clock domains controlled by Dynamic Voltage and Frequency Scaling (DVFS) mechanisms to meet the power requirements (maximum Thermal Design Power—TDP) and communication between them is ensured using FIFOs [3]. Globally Asynchronous Locally Synchronous (GALS) systems [4] extensively use FIFOs. The 48 core Intel IA-32 chip consists of a lot of FIFOs for communication [5].

Since FIFOs intervene in many system operations, they should be protected properly to ensure reliable operation. This becomes more imperative in current and future technology nodes, in which system failures are becoming more and more dominant. Static and dynamic variations [6,7] result in unreliable operation. Moreover, aging mechanisms [8] such as NBTI [9], electromigration, time dependent dielectric breakdown degrade devices and wires during system's lifetime and cause faults in the field. Furthermore, soft errors [10] and other types of transient faults affect reliability significantly. Thus, dependability in vital system operations is of utmost importance.

Operation in lower voltage, which is more than required to meet the power constraints, exacerbates devices reliability and process variation related problems appear more intense. SRAM cells seem to be much more vulnerable than logic and flip flops [11], since they are more dense, and their stability greatly depends on the asymmetry of the threshold voltages of their transistors. In [11] has been reported that in 12 nm one every few thousands SRAM cells will be faulty (due to Random Dopant Fluctuations and aging). In [12] the dependence of SRAM cell probability of failure (p_{fail}) on voltage is shown for 32 nm technology nodes.

Results show that even in small memory arrays, faults due to process variation/aging will occur. On the other hand, soft error resiliency should be ensured. Results also show that combinational logic is less vulnerable to process variation/aging induced faults (unless the time constraints are very tight). Thus, in this paper we more focus on array related faults.

All these effects result from scaling. Thus it would be wise to spend as less resources as possible for protection, otherwise we could result in canceling the effects of scaling. Information redundancy is a viable solution for protection in FIFOs, but it comes at the expense of state overhead and combinational circuit path for encoding/decoding. Even byte parity, which is the simplest fault detection technique, imposes a 12.5% state overhead, which is not negligible. What is more, it imposes critical path (8–9 input XORs in encoding/decoding) and energy overheads.

In this paper we propose a low cost fault detection technique for FIFO buffers. It is based on the update of a global parity register, which stores global parity in a column basis, requiring only a flip flop and two XOR gates per column. The fault detection

* Corresponding author. Tel.: +30 2107723653.
E-mail addresses: isidoros@microlab.ntua.gr (I. Sideris), pekmes@microlab.ntua.gr (K. Pekmestzi).

0167-9260/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.vlsi.2012.03.004

is based on the fact that when the FIFO becomes empty, the accumulated parity of the data read will be the same with that of the written and this register should be zero. A non-zero value is an indication of fault, and this property is exploited for fault detection. All faults encountered between two empty states accumulate and are effectively indicated in the column parity register.

The requirement to wait for the FIFO to become empty, however, increases the detection latency. Thus, we trade off detection latency for area, power and critical path overhead reduction.

Due to the unbounded detection latency of the mechanism, it can be more easily applied in systems with backward error recovery (BER) schemes or just as a health monitoring mechanism. The technique is easily applicable also in hardware peripherals which take input data in communication bursts and cannot accept new data unless they have processed them.

Besides the errors in memory arrays, this scheme can detect even addressing problems and metastability related problems in dual clock domain FIFOs.

In particular, the contributions of this paper are the following:

- A low cost fault detection mechanism for FIFOs is proposed. It allows for low cost error detection, at the expense of greater latency. It eliminates the state overhead of standard horizontal parity codes, while keeping combinational area low and decreasing critical path overheads and power consumption.
- The mechanism was modeled and synthesized using Verilog HDL and its area, power and delay overheads were evaluated thoroughly, and compared to standard parity protection schemes. Single clock and dual clock domain FIFOs were considered.
- The mechanism's fault coverage is analytically estimated.

The remainder of the paper is organized as follows. Section 2 introduces the proposed mechanism and mentions its capabilities and the involved overheads. Section 3 presents experimental results regarding the implementation of the proposed technique in 90 nm ASIC technology, as well as detection latency results in some example systems. Section 4 presents results regarding the fault coverage of the mechanism. Finally, Section 5 lists the related work and Section 6 concludes the paper.

2. Protection mechanism

The protection technique is evaluated both for single clock and dual clock domain FIFOs. First we describe how the FIFO operates (in each case) and then we present how the proposed protection mechanism is incorporated and how it differs from other protection schemes.

2.1. Single clock domain FIFO

2.1.1. Operation

The FIFO memories are implemented using a dual port memory and two pointers delimiting a sliding window, in which the stored entries are kept. In Fig. 1 the pointers *push_addr* and *pop_addr* point to the next location to write and the next location to read, respectively. Upon each access, the respective pointer is incremented by 1 (modulo N for N entries FIFO). When both pointers point to the same location the FIFO is empty, while if *push_addr*+1 equals *pop_addr*, then the FIFO is full.

Fig. 1 shows the internals of a single clock domain FIFO and its operation. It has two interfaces, one for the producer (*push_req_n*, *data_in*) and one for the consumer (*pop_req_n*, *data_out*). The producer asserts (low) the *push_req_n* signal and at the next positive edge of the clock (*clk*) the data is written into the FIFO. *The push_addr* pointer is incremented by 1, to point to the next location to write. Fig. 1(b) shows a scenario where the FIFO was empty and after the push the empty signal was deasserted. Similarly, the consumer asserts (low) the *pop_req_n* signal and at the next positive edge of the clock it can latch the data (assuming an asynchronous read port interface—in case we use synchronous SRAM for the read port, then the output is available at the next positive edge of the clock). The *pop_addr* pointer is incremented in the next positive clock edge. Fig. 1(c) shows a scenario where the FIFO was full and after the pop the *full* signal was deasserted.

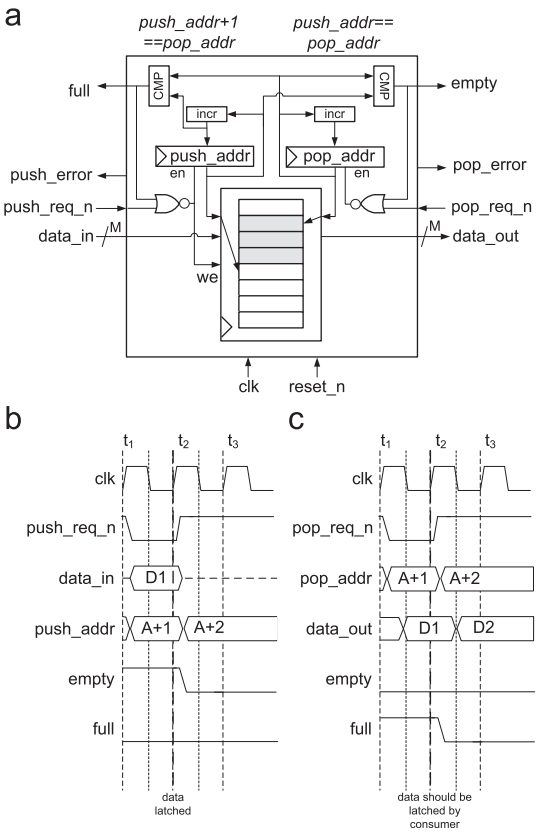

Fig. 1. FIFO operation (single clock).

The error outputs *push_error* and *pop_error* (shown in Fig. 1) are raised when a push is requested and the FIFO is full, or a pop is requested and the FIFO is empty, respectively.

2.1.2. Protection mechanism

Fig. 2 shows the FIFO of Fig. 1 augmented with the proposed protection mechanism. The mechanism maintains one single parity register for the whole memory array, effectively storing the parity of each column. Normally such parity would need a read before every write, in order to be updated, which could be a significant overhead. However, we make use of the FIFO operation and we update the parity register as follows: At every write we perform a xor of the value to be written with the parity register value and we store the new parity (*parity_reg*=*data_in xor parity_reg*). At every read we perform a xor of the read value with value of the *parity_register* (*parity_reg*=*data_out xor parity_reg*). Thus, we effectively store the parity of the active window. Whenever this window is empty (the FIFO is empty), the parity register should be zero, otherwise an error has occurred in some array bits. Fig. 2(a) shows how this algorithm is incorporated in

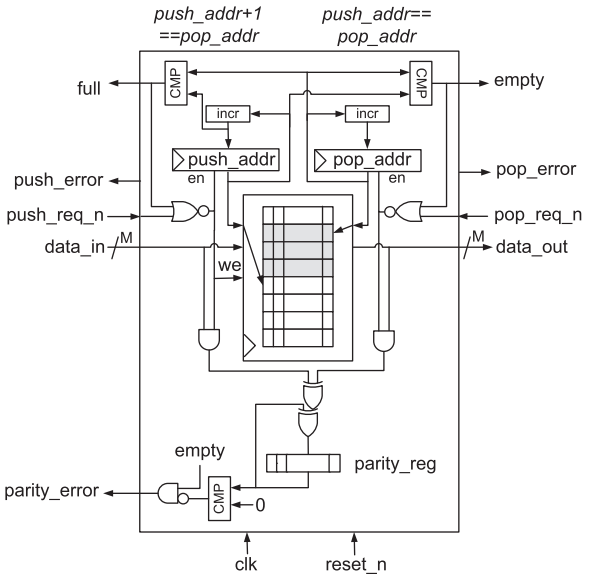

Fig. 2. Column parity based protection mechanism for single clock FIFOs.

the FIFO. The two XOR gates account for the case when read and write happen in parallel. Notice also that the parity update takes place only when a read or write (or both) takes place and not in any other case.

In the way the parity is updated faults accumulate in the parity register, and when the FIFO becomes empty they manifest their existence. Both permanent and soft errors can be caught, but both when the producer ceases and the FIFO becomes empty. Although there can be several time points that the FIFO becomes empty, periodic producer throttling can be applied to put an upper bound to the detection latency.

Fig. 3 shows a scenario of use of the FIFO and how a fault is indicated when the FIFO becomes empty. Three pops and one push take place. At the end of the third pop, *pop_addr* becomes equal to *push_addr* and the FIFO becomes empty (see signal *empty*). A possible parity error is indicated after some propagation time. The parity register is updated every cycle there is a read or write (or both). The new parity value latches at the next positive edge of the clock. Note that the fault may have occurred in any of the accessed entries, however it is indicated when the FIFO becomes empty.

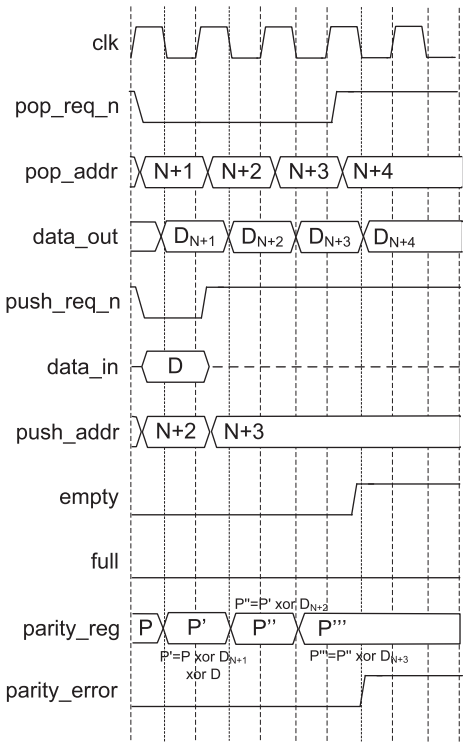

Fig. 3. Scenario illustrating the operation of the proposed protection mechanism (in single clock FIFOs).

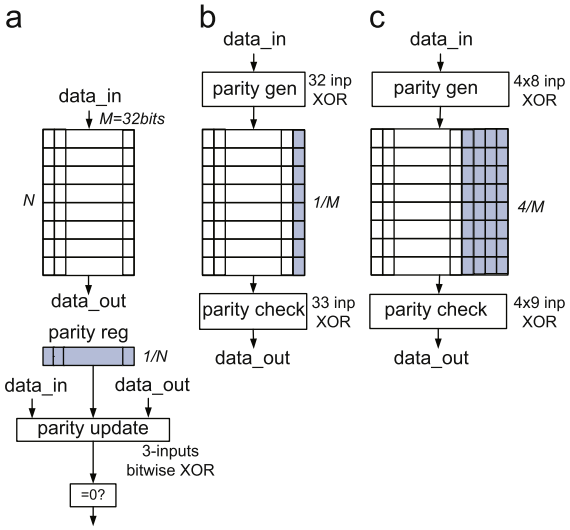

Fig. 4. Comparison of the overheads incurred in the (a) proposed mechanism vs. (b) 32-bit parity and (c) byte parity in a $N \times 32$ bits FIFO.

2.1.3. Comparison

Fig. 4 compares the proposed mechanism with 32-bit parity and byte parity (rather qualitatively—for detailed quantitative results see Section 3). The FIFO array has a size of $N \times M$ (with $M=32$) in all three cases. The proposed mechanism (a) has state overhead of $1/N$ (that is approximately 0.2% for $N=512$—or about 2% if the memory used is SRAM and the parity register is built with D flip flops, due to the mismatch in their size). What is more, no overhead is induced in the memory access, while the parity update needs two 2-bit XORs and one AND and it is off the critical path. On the other hand, 32-bit horizontal parity (b) imposes a $1/M$ state overhead (e.g. 3.125% for $M=32$), a 32-bit XOR tree for parity generation (on the critical path—unless if split by register) and a 33-bit XOR tree for parity check (again on the critical path). In the (horizontal) byte parity case (c), the state overhead is 4/32 (e.g. 12.5%), while the combinational circuit is composed of four 8-bit XOR trees for parity generation (on the critical path) and four 9-bit XOR trees for parity checks.

2.1.4. Fault coverage

The proposed mechanism can detect errors in the array (stuck-at 1, stuck-at 0 faults). A single error per column can be detected in its basic form (see variants for EDCs with more capabilities—Section 2.1.5).

In the way the mechanism works, it detects both permanent and soft errors. For every entry written to the memory array (pushed), the parity register is updated by xoring the data to be written with its value. When this entry is popped some time later, its value updates the parity register, thus removing this data from the parity register. If an error happens, either permanent (stuck-at) or a soft error (bit flip), this will contribute to the parity register, that is the effect of pushing/popping this data value will

not have zero contribution (as it would be expected in correct operation). When the FIFO becomes empty, all the values written will have been read, and if the parity register is not zero, then an error (permanent or bit flip) must have happened.

Besides the errors in memory arrays, the mechanism can detect errors in addressing, sense amplifier etc, since the detection is applied at the interface of the memory and not in the memory itself. For example, a whole column problem will be indicated in most cases, since the value written to the array will differ from the value read. Similarly, errors in addressing may be caught if different addresses map to the same location due to faults in the decoder.

Although the fault coverage of 1bit parity per column is a single error per column, the effective fault coverage may differ in the proposed mechanism, because the parity register covers all the entries having been accessed between two empty states. If the FIFO becomes empty fast, having transferred k values, then the fault detection capability is $1/k$. Ideally, if every value pushed is read before another one is written, then the mechanism can detect every fault. On the other side if the FIFO gets empty slow the detection capability is decreased. Thus, in cases of high producer rates may be periodic throttling should take place, or more complex codes (instead of parity) should be used. However, the traffic usually consists of bursts, directly consumed by the consumer, thus the expected number of writes (k) between empty states will be low.

2.1.5. Variants

The protection mechanism can be applied in some variations, to either lower the cost or increase its detection capabilities.

The first variant is to maintain the parity for more than one column in a parity bit, thus decreasing the state overhead of the parity register. This would require an extra k-input XOR for k columns parity maintenance. The state overhead reduction comes with an increase in the critical path and a decrease in the detection capability from a single error per column to a single error per k columns. This scheme makes sense for FIFOs of few entries.

Another variant is to partition the column in many segments based on the pointer address and maintain a parity register for each one. This would require k parity registers for k segments. This scheme comes at a k times state overhead increase, but the detection capabilities are increased from 1 per column to k per column. If the partitioning is performed in an interleaved form, multibit clustered errors can be caught. The path for updating the parity registers is increased by k bits address decoding, but since this is not the critical path, the frequency of operation is not affected.

2.1.6. Use

The FIFO can be used between two subsystems A (producer) and B (consumer) as follows. The producer passes data and if they are stored in a faulty FIFO entry, a parity error will be indicated at the end of the transfer. The parity error flag can be used by the producer as a negative acknowledge signal, which indicates that it has to resend the last packet (frame, etc.). At the side of the consumer, the parity error signal can be used to flush the data. The parity register can be reset when the producer (and consumer) has been notified. The parity error signal can drive an interrupt pin if some of the subsystems happen to be microprocessors, to allow for easier use.

Such kind of communication is very common in on chip communication protocols (supported by a higher level transport protocol). It is also common in coprocessor interfaces, as in Xilinx FSL where the microprocessor communicates with the coprocessor using FIFOs [2].

The proposed error detection mechanism can be easily adopted in systems which apply global backward error recovery (BER) schemes like checkpointing or logging. In such schemes, a possible detected violation (without knowing the specific data that caused it) would result in restoring the previous checkpoint or traversing backwards the log. The mechanism could be also used just for system health monitoring. Furthermore, if the data transferred happen to be image data (without control information) or other type of data which are not that critical and can tolerate few errors, then the parity error can be used just to estimate the quality of service (QoS) and react whenever is not met. Additionally, in some hardware peripherals a burst of input data is pushed to an internal FIFO (from the processor) and no new data are pushed before they are all processed. In such cases the FIFO gets empty often and allows for prompt error detection.

In other kinds of systems and applications, special care should be taken, since the error detection latency of the mechanism is not bounded and possibly some data may progress deeper in the pipeline before their fault is detected. To enable correct execution in the presence of faults, the communication should be handled by a higher level management logic, like a transport protocol. A message transmitted from the producer to the consumer can be followed by a throttle to force the FIFO to get empty and perform detection. Then, an acknowledge can be sent for the group of words which constitute this message. Doing that for many words would amortize the performance overhead and possibly would hide it by the idleness between transmitted messages. Something similar could be done with frames of data (e.g. image frames). At the end of the frame transmission, a stall could be performed to enable identification of the faulty frame.

Depending on the nature of the producer, this may be done in different ways. For example, if it is a microprocessor, then it can send specific control packets that stall the FIFOs (e.g. at the end of each message). If it is simple hardware (e.g. the previous pipeline stage in an accelerator), then counters can count frames and stall the FIFO, to enable restart of execution from a correct point.

The consumer, on its side, whenever an error is detected, it will know the group of words that the error belongs to and it will do actions to cancel its effect (if it is possible). If the consumer has some kind of speculation in its pipeline, the results can be committed based on the FIFO error detection outcome (in deep pipelines the error detection latency may be hidden). There is always the case that the effect of the consumer operation can be canceled by the producer (through end to end transfer protocol, or by keeping identifiers for the messages—groups of data at a higher level protocol). More sophisticated schemes can be devised depending on the type of the consumer.

This high level protocol can also have two modes, one without stalls and one with stalls. The second could be used when lowering the voltage for power savings, thus reducing reliability (when lowering the voltage, performance is not the primary goal, thus we can suffer some more performance loss).

2.2. Dual clock domain FIFO

2.2.1. Operation

Fig. 5 shows a FIFO with its two interfaces (push and pop) in separate clock domains (clocks *clk_w*, *clk_r*) [13,14]. The organization is similar to the single clock FIFO of Fig. 1. The push and pop interfaces (signals on the left and right, respectively) are also the same and are used in the same way, except that they are synchronized to a different clock. The main difference is that now the FIFO pointers *push_addr* and *pop_addr* are in separate clock domains and the full and empty signals cannot be obtained

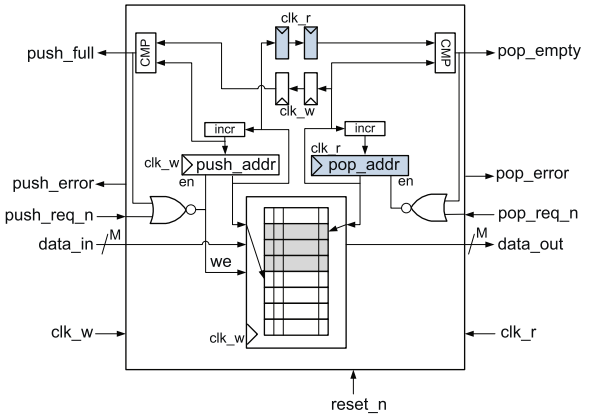

Fig. 5. FIFO crossing two different clock domains (the white registers use clock *clk_w*, while the filled ones use clock *clk_r*).

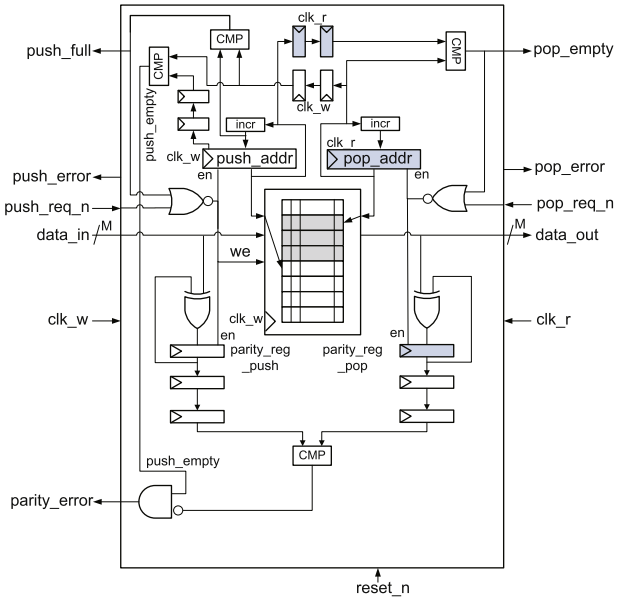

Fig. 6. Column parity based protection mechanism for dual clock FIFOs (the white registers use clock *clk_w*, while the filled ones use clock *clk_r*).

immediately. Instead, two registers of the opposite clock domain intervene between each of the two pointers and the comparator that generates the *pop_empty* (or *push_full*) signal (see Fig. 5). These registers (along with gray code incrementing logic for the pointers) serve to avoid metastability issues [15].

The organization shown in the scheme assumes a memory with a synchronous write port (for the push interface) and an asynchronous read port (for the pop interface). This is the case when flip flop based memories are used. When using dual clock synchronous SRAM, the other interface should be clocked at *clk_r* and the read data will be available in the next positive edge of clock *clk_r*.

2.2.2. Protection mechanism

The incorporation of the proposed protection mechanism in the dual clock domain FIFO differs slightly from its single clock counterpart (see Fig. 6). In particular, the parity generation is split in two parts, one at the push interface and one at the write. Thus, two parity registers are used (*push_parity_reg* and *pop_parity_reg*) both of which are updated when a push or pop take place, respectively. What is more, these parity values should be synchronized with the empty signal generation. Thus, two extra registers (of the opposite interface's clock: *clk_w*) intervene in the pop parity register (to avoid metastability). Two registers are used also in the push parity register (of the same clock in that case). Notice that the parities comparison outcome in the scheme is synchronized with the *push_empty* signal, both of which are needed to notify for a possible accumulated error.

The need for synchronizer registers increases the overhead of the mechanism (now six parity registers are needed instead of one), but still can result in area, delay and power gains as shown in Section 3.1.2.

Again the scheme shown assumes DFF based memories. When dual clock synchronous SRAM are used, slight modifications are needed. The read port address should latch the value of the incrementer (not the output of the *pop_addr* register) and the data will be available at the next positive edge of clock *clk_r* (and can be latched by the slave the edge after it asserts low the *pop_req_n* signal). This helps eliminating the need for more synchronizer registers. Schemes for both DFF based memories and dual clock synchronous memories have been implemented and evaluated (see Section 3).

3. Experimental results

3.1. Implementation results

3.1.1. Experimental methodology

Both single clock and dual clock FIFOs were evaluated. The designs were implemented in Verilog HDL, verified functionally in Modelsim and synthesized using Synopsys tools in 90 nm ASIC technology. Power results were also obtained using Synopsys PrimePower. Results for FIFOs with 32-bit parity, byte parity and without protection are also shown for comparison.

Two versions of each FIFO were evaluated, one using D flip flop (DFF) based Designware memories and one using dual port synchronous SRAM memories. Various memory sizes were used: 256×32 DFF based, 512×32, $1k \times 32$, $2k \times 32$ SRAMs.

3.1.2. Results

Single clock FIFOs. Fig. 7 shows the area, delay and power results of the single clock FIFO configurations, while Table 1 shows detailed results for the 256×32 DFF based memory FIFO. Fig. 8 compares the area overheads of the proposed mechanism with the two horizontal parity schemes (32-bit parity and byte parity). The proposed technique has minimal overhead, in comparison with those using horizontal parity, both in DFF based and SRAM based FIFOs. However, in DFF based FIFOs the differences are more intense, since in SRAM memories the overhead of the parity register is greater (due to the much larger size of the DFF). In small sizes, SRAM based FIFOs have no gains in area. But in such cases, DFF based memories would be more preferred.

In a 256×32 flip flop based FIFO we have reported an area overhead of 0.57% (and 2.33% power overhead), which is much less than the respective overheads of 32-bit horizontal parity (3.21% area, 4.33% power) and byte parity (12.21% area, 13.36% power), while the critical path is not affected at all. For greater reductions we could employ a scheme with columns grouping and result in an area overhead of 0.24% (see Table 1).

It is noteworthy that in SRAM based memories only up to 32-bit memories could be generated (due to restrictions of the memory compiler). Thus, in order to use parity (1-bit) two memories were generated (16 and 17-bit word size). For byte parity, two memories with 18-bit word size were generated. This is true even in FPGAs, when one should use typical dimensions, or otherwise he may need to occupy another memory (blockRAM).

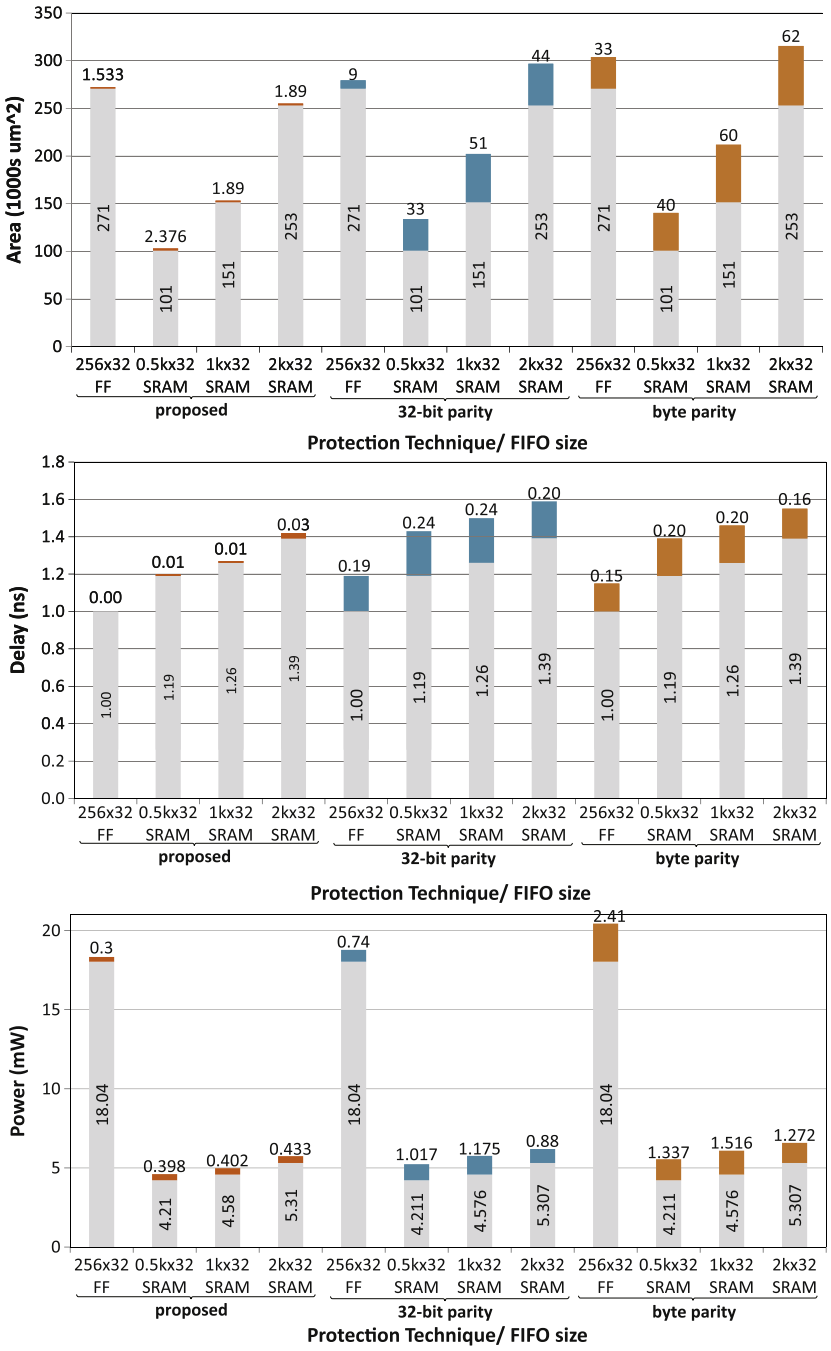

Fig. 7. Area, delay and power results for single clock FIFO protection techniques (the two parts of each column correspond to the baseline and the extra overhead for error detection).

Thus parity appears to be a much larger overhead than it may seem. However, to make the comparison more fair, we have included area results assuming we had 33 and 36-bit memories (indicated as SRAM * in the graphs). Again the benefits are clear.

In the second graph of Fig. 7 the critical paths of the FIFOs are shown. Notice that in the proposed mechanisms the critical path is not increased at all, while in parity and byte parity we have increases of 14.4–20.2% and 15–16.8%, respectively.

Power results are shown in the third graph of Fig. 7, while the corresponding overhead percentages are shown better in Fig. 9. The results were obtained assuming a specific simulation scenario (VCD files were extracted from functional simulation using the synthesis netlists and PrimePower was used to evaluate the power consumption). The scenario consists of continuous transfer of (random) data to the FIFO at the maximum rate and slightly delayed consumption of the data by the consumer.

The results show that the proposed scheme is the most power effective in all configurations. This is due to the more power consumed in the XOR trees of parity generation/check and on the larger size of the arrays. Another conclusion is that the DFF based memories consume much more power than their SRAM counterparts due to the large capacitance of the clock tree needed.

Dual clock FIFOs. Fig. 10 shows the area, delay and power results for the proposed and horizontal parity schemes for the dual clock FIFO case. The bottom part of Table 1 shows detailed results for DFF based memory. Figs. 11 and 12 compares the area and power overheads, respectively. The gains in area of the proposed mechanism appear decreased here, due to the larger

Table 1
Timing, area and power comparison of various protection techniques for 256 × 32 DFF based FIFOs.

	Area (μm^2)	incr (%)	**Power (mW) @ 200 MHz**				**Delay (ns)**	**Best delay (ns)**	
			Dynamic	Leakage	Total	incr (%)	(unconstr.)		incr
Single clock FIFO									
Reference (no protection)	270 748		17.75	0.29	18.04		2.77	1.00	
Proposed	272 281	0.57	18.05	0.29	18.34	1.66	2.92	1.00	0.00
Proposed with 4-bit/32 columns	271 405	0.24	18.33	0.29	18.62	3.22	2.96	1.00	0.00
Proposed with 1-bit/32 columns	271 221	0.17	17.79	0.29	18.08	0.22	3.42	1.03	0.03
Horizontal 32-bit parity	279 440	3.21	18.48	0.30	18.78	4.10	3.82	1.19	0.19
Horizontal byte parity	303 807	12.21	20.12	0.33	20.45	13.36	3.20	1.15	0.15
Dual clock FIFO									
Reference (no protection)	271 359		17.79	0.29	18.08		3.59	1.00	
Proposed	280 591	3.40	19.42	0.32	19.74	9.18	3.65	1.00	0.00
Proposed with 4-bit/32 columns	272 746	0.51	17.94	0.30	18.24	0.88	3.85	1.00	0.00
Proposed with 1-bit/32 columns	272 291	0.92	17.87	0.29	18.16	0.44	4.08	1.03	0.03
Horizontal 32-bit parity	279 999	3.18	18.70	0.30	19.00	5.09	3.54	1.19	0.19
Horizontal byte parity	304 436	12.19	20.78	0.34	21.12	16.81	3.34	1.14	0.14

The results have been taken using a 90 nm ASIC standard cell library.

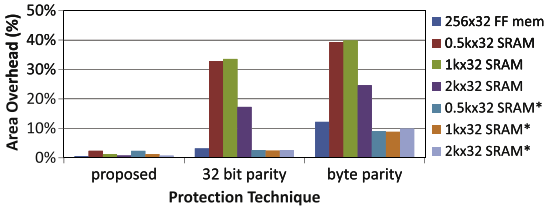

Fig. 8. Area overheads comparison of the various protection mechanisms for single clock FIFOs.

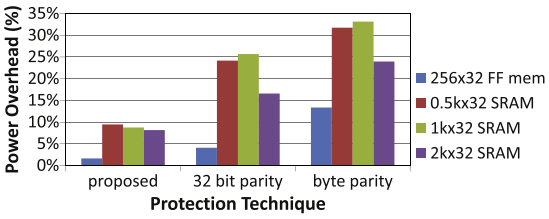

Fig. 9. Power overheads comparison of the various protection mechanisms for single clock FIFOs.

overhead for synchronizing the global parity registers (six parity registers instead of one in single clock FIFO). However in DFF based memories and in large SRAMs the gains are still significant. Power and delay results appear better than parity and byte parity FIFOs in most cases. Furthermore, when the circuit is constrained for higher frequency of operation, horizontal parity schemes tend to consume much more power (than the proposed mechanism), since large XOR gates are used for the encoding/decoding XOR tree. For the power consumption estimation, the same scenario used in the single clock domain FIFOs was used. For a scheme with reduced overhead, we can group many columns together (folding). The eight columns folding comes with a 0.51% area overhead, while it can be operated in up to 1 GHz.

Small FIFOs. Table 2 shows area and power results for a small DFF based FIFO (32 entries). The horizontal 32-bit parity scheme appears to have smaller area and power consumption than the proposed scheme (especially in the dual clock FIFOs), since there are no state overhead reductions in that FIFO size. Since the access time of the memory array is much smaller at that case, all designs met the 1 ns constraint. Thus, the horizontal 32-bit parity scheme seems to be the most viable choice for that configuration. The proposed technique would still have benefits if its variant which keeps one parity bit for all the columns was applied (see the configuration *prop.*1-*bit*/32 *columns* in the table).

3.2. Detection latency

Horizontal parity schemes perform fault detection immediately (when the data are read). However, the proposed mechanism needs some cycles, due to the requirement for the FIFO to become (at least) instantaneously empty.

The detection latency depends on the burst size and the average consumption rate. Even if the FIFO does not get empty often, periodic throttling of the producer can set an upper bound. Assuming a consumption rate of r (pops per cycle), this would induce a performance penalty of N/r cycles at most (where N the FIFO size).

The detection latency of the proposed mechanism depends on the implementation and varies with different systems or applications. To provide evidence about its applicability, we present results about some case studies. First, we study embedded FPGA SoCs which use the FIFO based FSL communication protocol [2] to communicate with hardware accelerators. Then, we estimate how often FIFOs get empty in Network-on-Chips (NoCs). Last, we show results about how often the queues of a DRAM controller get empty.

3.2.1. FPGA SoC with an edge detection coprocessor

First we examine the case of a Microblaze based SoC for edge detection in images [16]. In this system, the processor communicates with an edge detection accelerator using the FIFO based FSL communication protocol. The Microblaze processor (*mb*) transfers (32-bit) data through a 16 entries FIFO (*mb2periph*) to the accelerator and it receives the processed image by reading from another FIFO (*periph2mb*). The software loop (running on *mb*) that transfers the data is unrolled to achieve greater transferring speed. The left column (SoC1) of Table 3 shows monitoring information about the two FIFOs. The results were obtained through hardware–software cosimulation using Xilinx EDK 8.1 and Modelsim. The statistics include the number of cycles between two consecutive empty states for each FIFO (referred to as interval), the average error detection latency (obtained by counting the number of cycles between the push and the next empty state of the FIFO), the number of writes in each such interval and the number of intervals in the simulation. The results show that the detection is quite immediate (two and three cycles on average in *mb2periph* and *periph2mb* FIFOs, respectively).

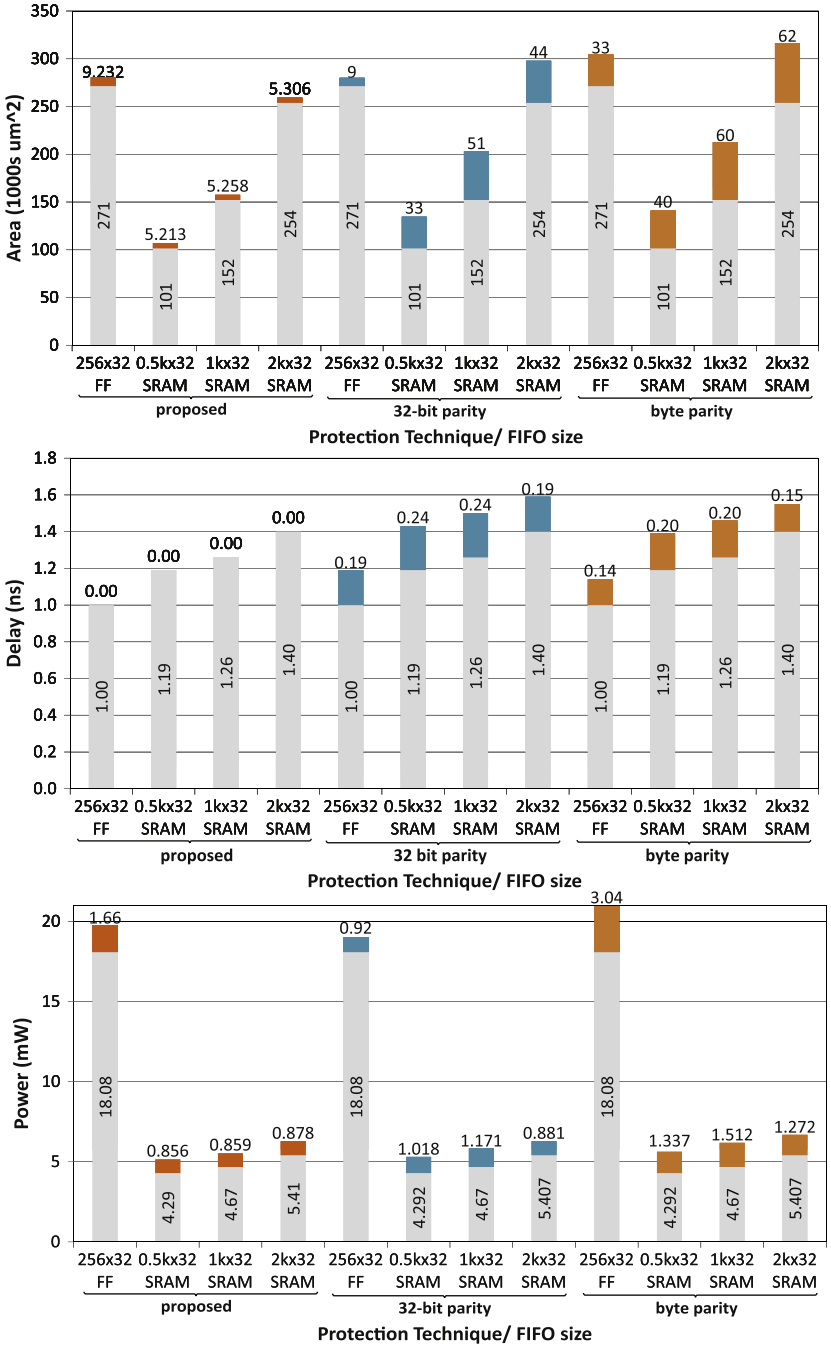

Fig. 10. Area, delay and power results for dual clock FIFO protection techniques (the two parts of each column correspond to the baseline and the extra overhead for error detection).

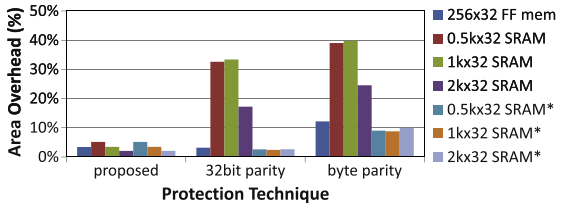

Fig. 11. Area overheads comparison of the various protection mechanisms for dual clock FIFOs.

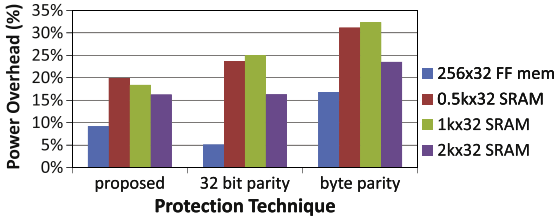

Fig. 12. Power overheads comparison of the various protection mechanisms for dual clock FIFOs.

3.2.2. *FPGA SoC with a long latency and non-pipelined coprocessor*

To make the error detection less efficient, we replaced the accelerator of the previous system with a long latency (15 cycles) non-pipelined execution unit which takes two 32-bit words from the input FIFO (*mb2periph*—256 entries now) and pushes the 32-bit result in the output FIFO (*periph2mb*—256 entries now).

The processor software sends 64 pairs of input data (e.g. 128 words) and then receives the results (64 words). The right column of Table 3 shows the respective results. The detection latency for the processor to peripheral FIFO (*mb2periph*) appears quite increased (about 902 cycles on average), since it gets empty only after all the 128 words of data have been processed. The other FIFO has about 277 cycles average detection latency although the average interval between two consecutive zero states is only 38.4 cycles. This is due to the fact that many results are pushed to the output FIFO while the processor is sending the input data and only after it starts receiving the output data, this FIFO gets empty (thus the maximum interval of 802 cycles). After that, the FIFO gets empty regularly.

3.2.3. NoC

In this study we evaluate how often FIFOs get empty in Network-on-Chips (NoCs). We have used the Noxim NoC simulator [17] and modeled a 4×4 mesh NoC with *xy* routing algorithm and random selection strategy. The routers have FIFOs of four entries per channel. We have assumed traffic of packet injection rate p which obeys (memoryless) Poisson time distribution and random spatial distribution. The packet size varied between 2 and 10 flits. The simulations were performed for 1 million cycles. Fig. 13 shows the number of cycles it takes for each FIFO of the NoC (16 routers times 5 FIFOs per router) to become empty, assuming a packet injection rate of $p=0.01$ (per cycle per processor element—each packet consists of 2–10 flits). The average number of cycles between two consecutive empty states varies between 1.07 and 1.30. The maximum number of cycles a FIFO remained non-empty was 145 cycles. Fig. 14 shows the number of pushes in each such interval. The average is low (1.02–1.09), while the maximum is 67 pushes. The results show that in such traffic the proposed detection mechanism performs error detection at low latency.

If we assume heavy traffic (rate $p=0.1$ in all nodes, which is quite unrealistic) we result in an average interval length of 1.69–2681 cycles and an average number of writes between 1.24 and 659. Even in that case the mechanism can perform detection (at higher latency).

3.2.4. DRAM controller

In this paragraph we present profiling information on the utilization of two queues of a DRAM controller. Each memory transaction coming from the system enters a queue called transaction queue (which is organized as a FIFO) and at its exit it is translated to DRAM commands which feed another queue, the command queue. The commands popped from the command queue are sent to the DRAM chips for execution. We have used the cycle accurate DRAM simulator DRAMSim2 [18] to model the DRAM controller. A single rank of 4 GB 667 MHz DDR3 memory is assumed for the experiments, while open page was selected for the row buffer policy. Both queues have 512 entries capacity.

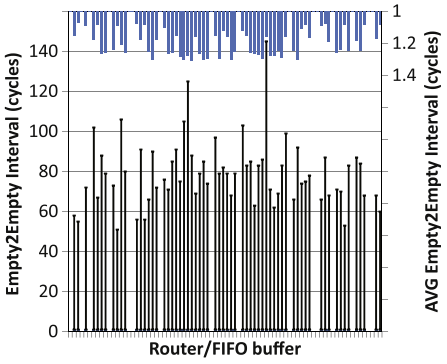

Fig. 13. Number of cycles between two consecutive empty states for each FIFO buffer of a 4×4 mesh NoC with packet injection rate of 0.01 per cycle (the black bars indicate the range, while the reverse bars show the average values).

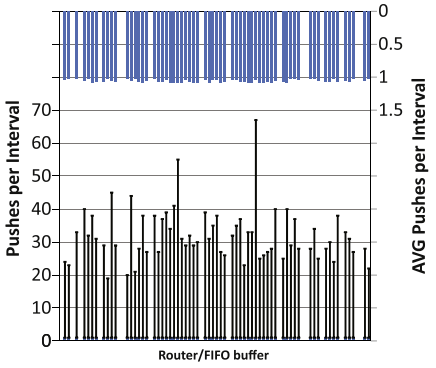

Fig. 14. Number of writes (pushes) between two consecutive empty states for each FIFO buffer of a 4×4 mesh NoC with packet injection rate of 0.01 per cycle (the black bars indicate the range, while the reverse bars show the average values).

Table 2
Area and power results for a small DFF based FIFO (32×32).

Scheme	Single clock		Dual clock	
	Area (μm^2)	Power (mW)	Area (μm^2)	Power (mW)
Unprotected	34 018	2.543	34 371	2.572
Proposed	35 644	2.809	39 384	3.388
Prop. 1-bit/32 cols	34 589	2.598	35 220	2.660
Hor. 32-bit parity	35 537	2.680	35 933	2.705
Hor. byte parity	38 526	2.960	38 922	2.978

The results assume 90 nm ASIC library. All FIFOs met the delay constraint of 1 ns. The power results assume 200 MHz frequency.

Table 3
Detection latency results in Microblaze based SoCs with FSL communication.

	SoC1 (edge detection)		SoC2 (long latency periph.)	
	mb2periph	periph2mb	mb2periph	periph2mb
FIFO size (entries)	16	16	256	256
Detection latency (cycles)[a]	2.04 (2–4)	3 (3–3)	902 (2–1195)	277 (3–803)
Interval[b] (cycles)[a]	1.04 (1–3)	2 (2–2)	399 (1–1194)	38.4 (2–802)
Writes per interval[a]	1 (1–1)	1 (1–1)	42.3 (1–125)	3 (1–43)
Number of intervals	1748	216	3	22

[a] AVG (MIN – MAX).
[b] Interval between two consecutive FIFO empty states.

First we gather statistics from full system simulation. We have used a modified version of the Marss × 86 simulator [19] coupled with the DRAMSim2 simulator. A single core 64-bit × 86 processor (running at 2 GHz and having a 4 MB eight way set associative L2 cache with eight cycles latency) was assumed. The processor was running Ubuntu nine operating system. We selected the benchmark *canneal*, which is the most memory demanding benchmark of the ParseC benchmark suite [20] (it is a place and route application). We simulated 100 million instructions in the region of interest (ROI), using ParseC ROI hooks. Tables 4 and 5 show the obtained results (left column) for the transactions queue and the command queue, respectively. The average number of cycles between two empty states in the transactions queue is 1.87 DRAM cycles (or 5.61 CPU cycles), while the maximum such interval is only 10 DRAM cycles. The command queue gets empty after 58.57 DRAM cycles on average. The results show that the proposed mechanism can be applied, since the detection latency will be few cycles.

To see how performance can be affected if a queue does not get empty at all and we are forced to throttle it regularly to perform detection, we now feed the DRAM simulator with back to back transactions (one transaction per cycle) and we throttle the queue after 10 pushes, until it gets empty. The simulation is performed using stored traces of transactions which feed the simulator (one push every cycle). In the transactions queue case, no performance penalty is reported, since it can get empty at a high rate (1 pop per cycle) feeding the command queue, which has always many commands to process. In the case of the command queue, we have reported a performance overhead of 5.25%. This is due to the slower rate of consumption of the command queue data (due to the DRAM commands execution latency). Although the command queue is not a pure FIFO (in contrast to the transactions queue), since some commands can take precedence, the conclusions drawn would be similar in the pure FIFO case.

Table 4
DRAM controller transactions queue monitoring results.

		back2back	
	Canneal	Nothrottle	Throttle[a]
FIFO size (entries)	512	512	512
Interval[b] (cycles)[c]	1.87 (1–10)	–	97.44 (3–307)
Pushes per interval[c]	1.87 (1–10)	–	10 (1–10)
Number of intervals	1 955 339	–	69 553
Cycles	321.51 M	6.849 M[d]	6.849 M[d]

[a] After 10 pushes.
[b] Interval between two consecutive FIFO empty states.
[c] AVG (MIN – MAX).
[d] DRAM cycles.

Table 5
DRAM controller command queue monitoring results.

		Back2back	
	Canneal	Nothrottle	Throttle[a]
Queue size (entries)	512	512	512
Interval[b] (cycles)[c]	58.57 (1–1301)	-	51.82 (20–244)
Pushes per interval[c]	4.00 (2–98)	–	10 (10–10)
Number of intervals	1 830 816	–	139 104
Cycles	321.67 M	6.849 M[d]	7.209 M[d]

[a] After 10 pushes.
[b] Interval between two consecutive queue empty states.
[c] AVG (MIN – MAX).
[d] DRAM cycles.

3.2.5. Discussion

The detection latency can become less immediate in some cases. However, the proposed mechanism can be always employed as a lightweight detection mechanism, along with a checkpointing scheme. That is, in a multicore system which employs checkpointing at regular time intervals, just before the end of each interval the FIFOs can be checked for possible violations. If an error is found, then the system will revert back to the last correct checkpoint, otherwise a new correct checkpoint will be committed. The performance overhead of throttling the FIFOs on a checkpoint interval basis would be very low.

Even if the detection latency does not allow for immediate detection, we can still detect much earlier (in most cases) than the error turns into symptom and we can prevent from system crashes or react faster. Thus, in FIFOs in tight critical paths in which standard horizontal parity may violate timing constraints, the proposed technique can be a viable solution.

The mechanism can also serve for transparent monitoring of FIFOs operation, without having to perform BIST (by writing and reading test data, thus disrupting the execution) while on execution. This may be useful in dynamic environments with DVFS schemes, where the probability of failure of the system components dynamically changes.

3.3. Fault map experiments

To test the correctness of the mechanism, faults that the mechanism could theoretically detect (e.g. up to one fault per column) were injected in functional simulation (Modelsim) and the functionality of the proposed detection mechanism was verified.

4. Analytical fault coverage estimation

As we saw earlier, the proposed mechanism's fault coverage depends not only on the physical column but on the access pattern as well. In this section, we estimate the probability of having a Silent Data Corruption (SDC), that is an error that escapes detection. We discern two cases: permanent faults and soft errors. For comparison, we estimate the respective probabilities for horizontal parity schemes and for the unprotected case.

4.1. Permanent faults

4.1.1. Assumptions

Fig. 15 shows the probabilities of having more than zero, one or two faults in a column for different column sizes, assuming random permanent faults with a cell p_{fail} value of 10^{-8}, consistent with the p_{fail} characterizations obtained in [12] for 32 nm technology node. In

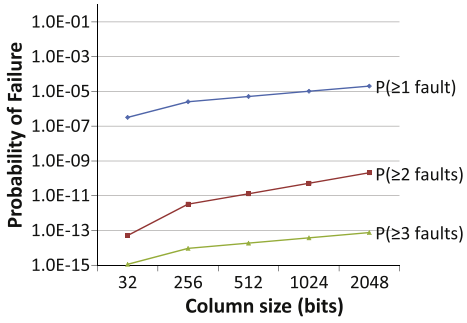

Fig. 15. Probabilities of failure for different column sizes assuming a cell p_{fail} of 10^{-8}.

particular, the probability that a column of N bits will have k faults is given by $q(N,k,p_{fail}) = \binom{N}{k} p_{fail}^k (1-p_{fail})^{N-k}$. The probability of at least one fault is $1-(1-p_{fail})^N$, while the probabilities of more than one and more than two are $1-q(N,0,p_{fail})-q(N,1,p_{fail})$ and $1-q(N,0,p_{fail})-q(N,1,p_{fail})-q(N,2,p_{fail})$, respectively. The results show that we will have one fault per column at most. In the remainder we will do this assumption to simplify the analysis. Furthermore, since the probability of a DFF failure is much smaller ($\sim 10^{-44}$ in 32 nm [11]) than in the SRAM cells, it is ignored in some calculations.

4.1.2. Proposed mechanism

To estimate the probability of SDC in the proposed scheme, we first estimate the probability of SDC in a column. Assume that we have a single stuck-at fault in one column. In order for an error to occur, the faulty bit should be written with the opposite value (otherwise there is no error). If the FIFO does not get empty before the faulty bit is accessed again, then a second write of the opposite value can cause the error to escape detection. Generally, there should be odd writes of the faulty bit with the opposite (to the stuck-at) value, in order to be able to detect an error. Assume now that we have λ accesses of the faulty bit before the FIFO becomes empty, that is the window slides circularly and revisits the same entry λ times and then the FIFO gets empty (note that λ accesses of the faulty bit means that the total accesses are from λN to $(\lambda+1)N$, that is λ full traversals).

Let $P_{SDC_{col},\lambda}$ denote the probability of a silent data corruption in the column throughout λ full FIFO traversals. Then, for $\lambda > 1$

$$P_{SDC_{col},\lambda} = (1-P_\lambda(D|A))P_{flt1}P_\lambda(A) \tag{1}$$

where P_{flt1} the probability of having one or more faults per column, $P_\lambda(A)$ the probability of accessing (writing) the faulty bit with the opposite value at least once throughout the λ visits, $P_\lambda(D)$ the probability of being able to perform detection when accessing the faulty bit exactly λ times (before the FIFO gets empty), and $P_\lambda(D|A)$ the probability to perform detection provided that there is at least 1 access of the opposite value to the faulty bit (in λ visits). P_{flt1} is given by

$$P_{flt1} = 1-(1-p_{fail})^N \tag{2}$$

where N the column size.

$P(D|A)$ can be derived using the equation

$$P_\lambda(D|A) = \frac{P_\lambda(A \cap D)}{P_\lambda(A)} \tag{3}$$

Assuming that the write of 1 and 0 are of equal probability, we can obtain the probability of having more than one access of the opposite value $P_\lambda(A)$ as

$$P_\lambda(A) = \sum_{k=1}^{\lambda} \binom{\lambda}{k} \left(\frac{1}{2}\right)^\lambda = (2^\lambda - 1)\left(\frac{1}{2}\right)^\lambda \tag{4}$$

The probability $P_\lambda(A \cap D)$ can be given by the sum

$$P_\lambda(A \cap D) = \sum_{k=0}^{\lfloor(\lambda-1)/2\rfloor} \binom{\lambda}{2k+1} \left(\frac{1}{2}\right)^\lambda = 2^{\lambda-1}\left(\frac{1}{2}\right)^\lambda \tag{5}$$

This equation is derived as follows. The probability of writing the opposite value in at least one access and perform detection is the same with the probability of writing the opposite value in odd number of accesses greater than or equal to 1. The probability of writing the opposite value in k accesses is $(1/2)^\lambda \binom{\lambda}{k}$, and the probability of writing the opposite value in k accesses and being able to detect is $(1/2)^\lambda \binom{\lambda}{k}$ if k is odd or 0 if k is even. By summing over all different number of odd accesses (less than or equal to λ) we obtain (5).

Using (1), (3)–(5) we obtain

$$P_{SDC_{col},\lambda} = \left(\frac{1}{2} - 2^{-\lambda}\right) P_{flt1} \tag{6}$$

for $\lambda > 1$.

Note that $P_{SDC_{col},\lambda}$ converges to $P_{flt1}/2$ after few full FIFO traversals (e.g. some value of λ) and the protection mechanism loses potential in detecting faults.

The probability of a silent data corruption in the FIFO array is given by the equation

$$P_{SDC_{arr},\lambda} = 1-(1-P_{SDC_{col},\lambda})^M \tag{7}$$

where M the size of the row (e.g. 32 bits).

The probability of a silent data corruption in a specific push into the array $P_{SDC_{entry},\lambda}$ can be obtained using the equation

$$P_{SDC_{arr},\lambda} = 1-(1-P_{SDC_{entry},\lambda})^{N'} \tag{8}$$

where N' the number of pushes between two empty states. Since the faulty bit can be in any of the N positions with the same probability, the average write accesses in λ visits of it are $\lambda N + N/2$. Thus N' can take the value $\lambda N + N/2$. Furthermore, since $(1-f)^n \approx 1-nf$ for $f \ll 1$ we obtain

$$P_{SDC_{arr},\lambda} \approx N' P_{SDC_{entry},\lambda} \tag{9}$$

Thus

$$P_{SDC_{entry},\lambda} \approx \frac{P_{SDC_{arr},\lambda}}{N'} = \frac{2P_{SDC_{arr},\lambda}}{(2\lambda+1)N} \tag{10}$$

For $\lambda \le 1$, since no faulty bit is revisited, the probability of silent data corruption (in a column) can be estimated by the probability of having even number of opposite accesses to stuck-at faults, that is

$$P_{SDC_{col},n} = \sum_{k=1}^{\lfloor n/2 \rfloor} \binom{n}{2k} (p_{fail}/2)^{2k} (1-p_{fail}/2)^{n-2k}, \quad n \le N \tag{11}$$

where n the number of total pushes between two empty states. The probability of silent data corruption in a specific push is given using (7) and (8):

$$P_{SDC_{entry},n} \approx \frac{MP_{SDC_{col},n}}{n}, \quad n \le N \tag{12}$$

4.1.3. Horizontal parity

For horizontal parity schemes the probability of silent data corruption in a specific push is straightforwardly obtained as the probability of having even number of opposite accesses to the stuck-at faults, that is

$$P_{SDC_{entry}}(M) = \sum_{k=1}^{\lfloor M/2 \rfloor} \binom{M}{2k} (p_{fail}/2)^{2k} (1-p_{fail}/2)^{M-2k} \tag{13}$$

where M the number of bits protected by 1-bit parity (including it). For 32-bit entry size and horizontal 32-bit parity, $P_{SDC_{entry}} = P_{SDC_{entry}}$ (33), while for byte parity $P_{SDC_{entry}} = 1-(1-P_{SDC_{entry}}(9))^4$.

4.1.4. Unprotected scheme

The probability of silent data corruption in a specific push when no protection is employed is estimated as the probability of having at least one stuck-at fault being written with the opposite value, that is: $P_{SDC_{entry}} = 1-(1-p_{fail}/2)^M$, where M the entry size in bits.

4.1.5. Comparison

Fig. 16 shows the SDC probability (per access) of the different schemes for different number of traversals between empty states

(different λ values) and column sizes. The proposed scheme (continuous lines) loses potential after $\lambda = 2$ ($2N$ pushes), but still is much lower than the unprotected case. The horizontal parity schemes have constant SDC rate, lower than all the other schemes. Fig. 17 shows the respective probabilities for $n \leq N$ pushes (between empty stages). You can see that the proposed mechanism performs very efficiently if the FIFO gets empty no later than after N pushes, where N its size. Indeed, for small values of n the proposed mechanism performs better than the horizontal parity schemes.

Although the increase of SDC due to permanent errors after the second traversal, the fault will be detected in some access and can be repaired or deconfigured (e.g. by index bypassing, as in [21]). However, repair is beyond the scope of this paper.

Table 6 compares the SDC rates of the proposed mechanism with horizontal parity schemes and the unprotected case for various random probabilities of cell failures (p_{fail}) and accesses. The left part of the table shows SDC rates when the FIFO gets empty before revisiting the first entry (accesses less than the column size, as in Fig. 17), while the right corresponds to full traversals of the FIFO (as in Fig. 16). Results are shown for four different column sizes. For p_{fail}, three different values were used: 10^{-8} (used in the graphs) and the increased values 10^{-7} and 10^{-6}, which may be used for lower voltages or cell sizes. The conclusions

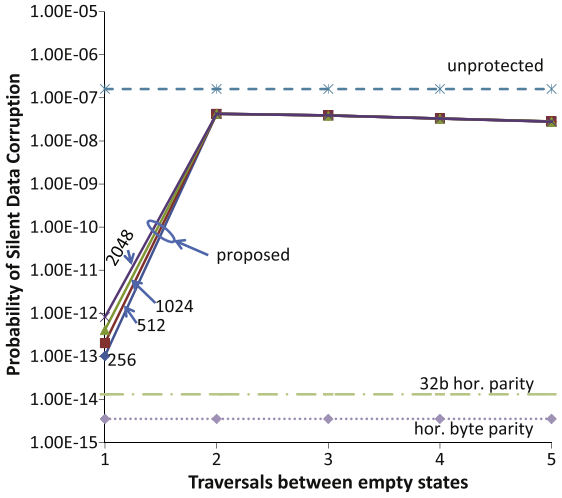

Fig. 16. Silent Data Corruption Rate due to permanent faults vs. number of FIFO traversals (λ) between empty states for various column sizes (assumed $p_{fail} = 10^{-8}$).

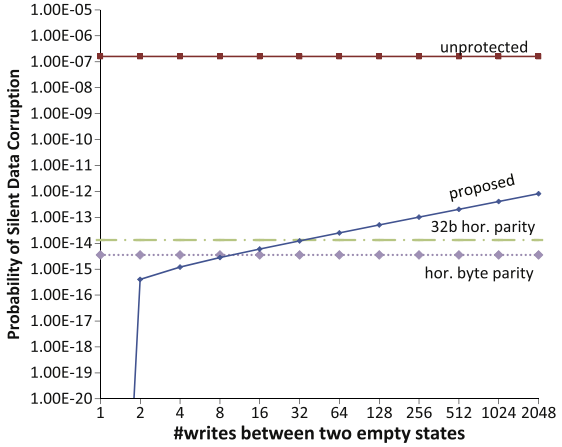

Fig. 17. Silent Data Corruption Rate due to permanent faults vs. number of writes ($n \leq N$) between empty states (assumed $p_{fail} = 10^{-8}$).

are similar with those drawn previously. However, in higher p_{fail} values, long traversals may compromise the fault detection capability.

Table 7 summarizes the accesses numbers of the case studies presented in Section 3.2, to judge about the expected SDC rates. You can see that in most cases the FIFOs get empty before revisiting the same entry. Thus, the proposed mechanism performs quite well. What is more, many FIFOs get empty after one push, which means that the fault detection capability is much better than in the horizontal parity case, since every written bit is protected by a D flip flop bit (which is also much more reliable than an SRAM cell).

4.2. Soft errors

For soft errors, the probability of an SDC in a column can be estimated as the SDC of a fixed size column of size $N' = \lambda N$, since there is no probability the same bit is flipped. That is

$$P_{SDC_{col}} = \sum_{k=1}^{\lfloor N'/2 \rfloor} \binom{N'}{2k} p'^{2k}_{fail} (1 - p'_{fail})^{N'-2k} \tag{14}$$

where $N' = \lambda N$ the dynamic column size (until the FIFO becomes empty) and $p'_{fail} = SER \cdot N/2r$, where r the rate of the FIFO data consumption. The p'_{fail} is based on the fact that after the first traversal the entry is rewritten and on the fact that on average a cell is vulnerable for $N/2r$ time. Assuming a soft error rate SER of $0.3-1 \times 10^{-8}$/h as it is assumed in [22], we can expect up to one flip between two empty states, thus the soft error resiliency is kept high. The probability of SDC in a specific push is obtained using (7) and (8)

$$P_{SDC_{entry}} \approx \frac{MP_{SDC_{col}}}{N'} \tag{15}$$

The SDC probabilities for unprotected and horizontal parity schemes are obtained using the same equations used for permanent faults by substituting $p_{fail}/2$ with p'_{fail} (see (13)). Fig. 18 compares the SDC rate (probability of SDC in a push) of the proposed scheme (continuous lines), with the unprotected and the horizontal parity schemes. You can see that the mechanism does not lose potential with the many traversals and can replace horizontal parity schemes (32-bit and byte parities—dotted lines on the bottom), while it decreased SDC by as much as 10^{15} times than in the unprotected case (dashed lines on the top). Thus, the soft error resiliency of the mechanism is sufficient.

4.3. Summary

Table 8 summarizes the key features of the proposed and horizontal parity schemes. The proposed mechanism requires less area and critical path at the expense of fault detection latency and higher SDC rates. An added advantage of the proposed mechanism however is its capability to detect addressing problems and other logic related problems (e.g. metastability, etc.). If the size of the FIFO is such that the SDC rate is acceptable, the proposed mechanism can reduce area, delay (or power) overheads.

Table 9 summarizes the comparisons of the proposed scheme with the horizontal parity schemes (for single clock FIFOs). Area, power and delay overheads, as well SDC rates are shown together for comparison. We can see that the proposed mechanism has minimal area, power and delay overheads and efficient protection against soft errors. However, it can be more vulnerable to permanent errors. The detection latency appears increased as well.

Table 6
SDC rates due to permanent faults for various random probabilities of cell failure p_{fail} and accesses.

Scheme	Column size (N)	p_{fail}	SDC rates for accesses $n \le N(\lambda \le 1)$							SDC for full traversals $\lambda > 1$		
			1	2	32	256	512	1024	2048	$\lambda=2$	$\lambda=3$	$\lambda=4$
Proposed	256	10^{-8}	1.6×10^{-43}	1.2×10^{-15}	1.2×10^{-14}	10^{-13}				4.3×10^{-8}	3.9×10^{-8}	3.3×10^{-8}
		10^{-7}	1.6×10^{-43}	1.2×10^{-13}	1.2×10^{-12}	10^{-11}				4.3×10^{-7}	3.9×10^{-7}	3.3×10^{-7}
		10^{-6}	1.6×10^{-43}	1.2×10^{-11}	1.2×10^{-10}	10^{-9}				4.3×10^{-6}	3.9×10^{-6}	3.3×10^{-6}
	512	10^{-8}	1.6×10^{-43}	1.2×10^{-15}	1.2×10^{-14}	10^{-13}	2×10^{-13}			4.3×10^{-8}	3.9×10^{-8}	3.3×10^{-8}
		10^{-7}	1.6×10^{-43}	1.2×10^{-13}	1.2×10^{-12}	10^{-11}	2×10^{-11}			4.3×10^{-7}	3.9×10^{-7}	3.3×10^{-7}
		10^{-6}	1.6×10^{-43}	1.2×10^{-11}	1.2×10^{-10}	10^{-9}	2×10^{-9}			4.3×10^{-6}	3.9×10^{-6}	3.3×10^{-6}
	1024	10^{-8}	1.6×10^{-43}	1.2×10^{-15}	1.2×10^{-14}	10^{-13}	2×10^{-13}	4.1×10^{-13}		4.3×10^{-8}	3.9×10^{-8}	3.3×10^{-8}
		10^{-7}	1.6×10^{-43}	1.2×10^{-13}	1.2×10^{-12}	10^{-11}	2×10^{-11}	4.1×10^{-11}		4.3×10^{-7}	3.9×10^{-7}	3.3×10^{-7}
		10^{-6}	1.6×10^{-43}	1.2×10^{-11}	1.2×10^{-10}	10^{-9}	2×10^{-9}	4.1×10^{-9}		4.2×10^{-6}	3.9×10^{-6}	3.3×10^{-6}
	2048	10^{-8}	1.6×10^{-43}	1.2×10^{-15}	1.2×10^{-14}	10^{-13}	2×10^{-13}	4.1×10^{-13}	8.2×10^{-13}	4.3×10^{-8}	3.9×10^{-8}	3.3×10^{-8}
		10^{-7}	1.6×10^{-43}	1.2×10^{-13}	1.2×10^{-12}	10^{-11}	2×10^{-11}	4.1×10^{-11}	8.2×10^{-11}	4.3×10^{-7}	3.9×10^{-7}	3.3×10^{-7}
		10^{-6}	1.6×10^{-43}	1.2×10^{-11}	1.2×10^{-10}	10^{-9}	2×10^{-9}	4.1×10^{-9}	8.2×10^{-9}	4.3×10^{-6}	3.9×10^{-6}	3.3×10^{-6}
32-bit horizontal parity	Any	10^{-8}	1.3×10^{-14}									
		10^{-7}	1.3×10^{-12}									
		10^{-6}	1.3×10^{-10}									
Horizontal byte parity	Any	10^{-8}	3.6×10^{-15}									
		10^{-7}	3.6×10^{-13}									
		10^{-6}	3.6×10^{-11}									
Unprotected	All	10^{-8}	1.6×10^{-7}									
		10^{-7}	1.6×10^{-6}									
		10^{-6}	1.6×10^{-5}									

Table 7
Number of pushes between consecutive empty states for the case studies of Section 3.2.

Case study	FIFO size	Avg. pushes		Max pushes	
		No.	λ	No.	λ
SoC1 mb2periph	16	1	≤ 1	1	≤ 1
SoC1 periph2mb	16	1	≤ 1	1	≤ 1
SoC2 mb2periph	256	42.3	≤ 1	125	≤ 1
SoC2 periph2mb	256	3	≤ 1	43	≤ 1
NoC	32	1.02–1.09	≤ 1	67	3
DRAM tran. queue	512	1.87	≤ 1	10	≤ 1
DRAM cmd. queue	512	4	≤ 1	98	≤ 1

5. Related work

Standard error detection codes (EDCs) [23,24] can be applied in the memory array of the FIFO. These codes can detect HD-1 faults and correct (HD-1)/2 faults, where HD the hamming distance between every two valid codewords [25].

In 2D parity product codes [26] horizontal and vertical parity bits are used to detect and correct single bit errors within a word. In [27] product codes for the entire RAM are applied.

In [28] the authors maintain column parity in conjunction with standard horizontal protection schemes (EDC/ECC), to allow for multibit error detection/correction in cache memories. Our scheme uses the column parity for low cost error detection in FIFO memories. It also avoids extra reads to update column parity, using the port addresses as markers for the buffer size.

In [29] they propose a general counter based scheme, which uses counters to keep track of how many ones exist in each column. The mechanism comes with the overhead of an extra read for each write, which however can be eliminated in some balanced accessed structures (in which for every write there is a read of the same location). Our technique uses low cost parity, instead of counters and specializes in FIFO memories. The counters could impose significant overheads in combinational

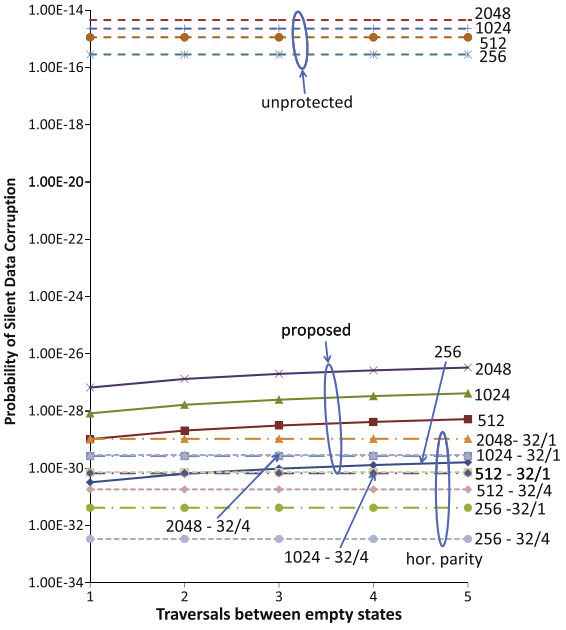

Fig. 18. Silent Data Corruption Rate due to soft errors for various number of FIFO traversals (between empty states) assuming a SER of 10^{-8}/h and a consumption rate of 1 per 25 ns.

delay, power and state and would not be compensated in small structures like FIFOs.

In [21] they propose a general technique that uses an auxiliary register to the buffer structure and writes both the register and the buffer, to test if they agree. However, this mechanism catches only permanent faults and it comes with some overheads (two reads and a write—although not paid in every write access). Our mechanism detects soft errors as well and is not intrusive to the FIFO operation.

Last, in [30] techniques for FIFO manufacturing testing are proposed. Our work focuses on online fault detection.

Table 8
Qualitative comparison.

Mechanism	Fault coverage	Detection latency	State overhead	Critical path overhead
Proposed	Soft ($1/n$), hard ($1/n$ for $n \leq N$, else worse), addressing and logic	When FIFO gets empty	$1/N$	None
32-bit parity	1/32 bits	When read	$1/M$	33-inp. XOR
Byte parity	1/8 bits	When read	$4/M$	9-inp. XOR

n: write accesses between two empty states, N: column size, M: row size.

Table 9
Comparison of protection schemes (single clock FIFOs).

Scheme	Area overhead (%)	Power overhead (%)	Critical path increase	SDC rate (per access)		Detection latency
				Permanent (approx.)	Soft (approx.)	
Proposed	0.57–2.36	1.66–9.45	Up to 2%	10^{-16}–10^{-8}	10^{-30}–10^{-27}	When empty
32-bit hor. parity	2.44–33.56	4.10–25.68	Up to 19%	10^{-14}	10^{-29}	When read
Hor. byte parity	8.83–39.95	13.36–33.13	Up to 15%	10^{-15}	10^{-30}	When read

Permanent errors assume a p_{fail} of 10^{-8}, while soft errors assume a SER of 10^{-8}/h and a consumption rate of 1 per 25 ns.

6. Conclusions

A low cost fault detection mechanism has been proposed for FIFO buffers. The mechanism trades off detection latency to gain reductions in area, delay and power overheads without compromising soft error resiliency. In a 256×32 DFF based FIFO, we have reported reductions of 5.63 times in area and 4.09 times in power overheads in comparison with standard horizontal 32-bit parity, while the critical path delay was not affected at all. The area overheads of the proposed protection technique varied between 0.57% and 2.36% in all single clock configuration examined (2.09–5.14% in dual clock), which can be further reduced to 0.17% with grouping of many columns. The respective overhead ranges of horizontal 32-bit parity and byte parity were 2.44–33.56% and 8.83–39.95%, respectively. No critical path overhead was imposed in any of the proposed configurations, in contrast with horizontal parity schemes (up to 19%). Beyond array cell errors, the mechanism can detect other types of errors (e.g. in addressing), which makes it particularly useful. The simplicity of the mechanism makes it easy to adopt and can be a good candidate in designs with many and/or large FIFOs, especially when area, power and frequency constraints cannot sustain expensive FIFO protection schemes.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments which helped them in improving the quality of this paper. Isidoros Sideris would like to thank Prof. Yiannakis Sazeides for the training in the area of reliability he offered him during his postdoctoral fellowship at the University of Cyprus, which greatly helped him to perform this study.

References

[1] B.-G. Nam, J. Lee, K. Kim, S. J. Lee, H.-J. Yoo, A 52.4 mW 3D graphics processor with 141 Mvertices/s vertex shader and 3 power domains of dynamic voltage and frequency scaling, in: Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International, 2009, pp. 278–603.
[2] Fast simplex link channel (FSL), product specification 〈http://www.xilinx.com〉, 2004.
[3] E. Rotem, A. Mendelson, R. Ginosar, U. Weiser, Multiple clock and voltage domains for chip multi processors, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, 2009, pp. 459–468.
[4] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, B. Baas, AsAP: an asynchronous array of simple processors, IEEE Journal of Solid-State Circuits 43 (2008) 695–705.
[5] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling, IEEE Journal of Solid-State Circuits 46 (2011) 173–183.
[6] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, V. De, Parameter variations and impact on circuits and microarchitecture, in: Proceedings of the 40th Annual Design Automation Conference, DAC '03, 2003, pp. 338–342.
[7] O.S. Unsal, J.W. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, O. Ergin, Impact of parameter variations on circuits and microarchitecture, IEEE Micro 26 (2006) 30–39.
[8] J. Srinivasan, S.V. Adve, P. Bose, J.A. Rivers, Lifetime reliability: toward an architectural solution, IEEE Micro 25 (2005) 70–80.
[9] J. Abella, X. Vera, A. Gonzalez, Penelope: the NBTI-aware processor, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, 2007, pp. 85–96.
[10] J.F. Ziegler, H.P. Muhlfeld, C.J. Montrose, H.W. Curtis, T.J. O'Gorman, J.M. Ross, Accelerated testing for cosmic soft-error rate, IBM Journal of Research and Development 40 (1996) 51–72.
[11] S.R. Nassif, N. Mehta, Y. Cao, A resilience roadmap, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10, 2010, pp. 1011–1016.
[12] S.-T. Zhou, S. Katariya, H. Ghasemi, S. Draper, N.S. Kim, Minimizing total area of low-voltage SRAM arrays through joint optimization of cell size, redundancy, and ECC, in: IEEE International Conference on Computer Design (ICCD), 2010, pp. 112–117.
[13] R. Apperson, Z. Yu, M. Meeuwsen, T. Mohsenin, B. Baas, A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains, IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 15 (2007) 1125–1134.
[14] C.E. Cummings, P. Alfke, Simulation and synthesis techniques for asynchronous FIFO design with asynchronous pointer comparisons, in: Synopsys Users Group, San Jose, CA, 2002.
[15] C. Portmann, H. Meng, Metastability in CMOS library elements in reduced supply and technology scaled applications, IEEE Journal of Solid-State Circuits 30 (1995) 39–46.
[16] N. Anastasiadis, I. Sideris, K. Pekmestzi, A fast multiplier-less edge detection accelerator for FPGAs, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, 2010, pp. 510–515.
[17] M. Pallesi, D. Patti, F. Fazzino, Network-on-Chip simulator 〈http://noxim.sourceforge.net〉, 2010.
[18] P. Rosenfeld, E. Cooper-Balis, B. Jacob, Dramsim2: a cycle accurate memory system simulator, IEEE Computer Architecture Letters 10 (2011) 16–19.
[19] A. Patel, F. Afram, S. Chen, K. Ghose, MARSS: a full system simulator for multicore $\times$86 CPUs, in: Proceedings of the 48th Design Automation Conference, DAC '11, 2011, pp. 1050–1055.
[20] C. Bienia, Benchmarking Modern Multiprocessors, Ph.D. Thesis, Princeton University, 2011.
[21] F.A. Bower, P.G. Shealy, S. Ozev, D.J. Sorin, Tolerating hard faults in microprocessor array structures, in: Proceedings of the 2004 International Conference on Dependable Systems and Networks, pp. 51–60.
[22] Z. Chishti, A.R. Alameldeen, C. Wilkerson, W. Wu, S.-L. Lu, Improving cache lifetime reliability at ultra-low voltages, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, 2009, pp. 89–99.

[23] I. Koren, C.M. Krishna, Fault Tolerant Systems, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
[24] D.J. Sorin, Fault Tolerant Computer Architecture, Synthesis Lectures on Computer Architecture, Morgan and Claypool Publishers, 2009.
[25] R.W. Hamming, Error detecting and error correcting codes, The Bell System Technical Journal 26 (1950) 147–160.
[26] P. Calingaert, Two-dimensional parity checking, Journal of the ACM 8 (1961) 186–200.
[27] R.M. Tanner, Fault-tolerant 256k memory designs, IEEE Transactions on Computers 33 (1984) 314–322.
[28] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, J. Hoe, Multi-bit error tolerant caches using two-dimensional error coding, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, 2007, pp. 197–209.
[29] Y. Sazeides, B. Ahsan, I. Sideris, L. Ndreu, S. Idgunji, E. Ozer, CBFD: a count-based fault detection scheme for memory arrays, in: 7th IEEE Workshop on Silicon Errors in Logic—System Effects, SELSE 2011.
[30] Y. Zorian, A.J. Van de Goor, I. Schanstra, An effective BIST scheme for ring-address type FIFOs, in: Proceedings of the 1994 International Conference on Test, ITC'94, IEEE Computer Society, 1994, pp. 378–387.

Isidoros Sideris received his diploma and Ph.D. Degree in Electrical and Computer Engineering from the National Technical University of Athens, Greece in 2004 and 2008, respectively. From 2010 to early 2011 he was with the Computer Architecture Research Laboratory of the University of Cyprus. He is currently a research associate in the Microprocessors and Digital Systems Laboratory of the National Technical University of Athens. His research interests include microarchitecture, low power design, fault tolerant—dependable systems and application specific hardware accelerators.

Kiamal Pekmestzi received his Diploma in Electrical Engineering from the National Technical University of Athens (1975). From 1975 to 1981 he was a research fellow in the Electronics Department of the Nuclear Research Center "Demokritos". He received his Ph.D. in Electrical Engineering from the University of Patras (1981). From 1983 to 1985 he was a Professor at Higher School of Electronics in Athens. Since 1985 he has been with the National Technical University of Athens, where he is currently a Professor in the Department of Electrical and Computer Engineering. His research interests include efficient implementation of arithmetic circuits, microprocessor based systems, reconfigurable computing and VLSI architectures for fast digital signal processing.