A Memory Coherence Technique for Online Transient Error Recovery of FPGA Configurations

Wei-Je Huang and Edward J. McCluskey
CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory, Department of Electrical Engineering
Stanford University, Stanford, California 94305-4055
{weije, ejm}@crc.stanford.edu

ABSTRACT

The partial reconfiguration feature of some of the current-generation Field Programmable Gate Arrays (FPGAs) can improve dependability by detecting and correcting errors in on-chip configuration data. Such an error recovery process can be executed online with minimal interference of user applications. However, because Look-up Tables (LUTs) in Configurable Logic Blocks (CLBs) of FPGAs can also implement memory modules for user applications, a memory coherence issue arises such that memory contents in user applications may be altered by the online configuration data recovery process. In this paper, we investigate this memory coherence problem and propose a memory coherence technique that does not impose extra constraints on the placement of memory-configured LUTs. Theoretical analyses and simulation results show that the proposed technique guarantees the memory coherence with a very small (on the order of 0.1%) execution time overhead in user applications.

Keywords

Fault Tolerance, FPGA, Memory Coherence, Error Recovery.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) based on SRAM cells are becoming very important components in various applications, including communication, networking, and storage systems. With the advance of process technology and device architecture, current-generation FPGAs have the capacity of more than 500K logic gates with the maximum clock rate up to several hundred-MHz [Altera 00][Xilinx 00]. In addition, because of the reconfigurable feature, FPGA-based systems can facilitate not only the prototyping of system-level designs but also the adaptive computing system. All of these characteristics allow FPGA-based systems to be optimized for different figures of merit such as performance, cost, and fault tolerance.

From the performance perspective, the abundance of Configurable Logic Blocks (CLBs) and routing resources in current FPGAs makes it possible to realize highly parallel architectures to boost the throughput of many applications [Huang 00a]. From the cost perspective, optimized designs of different applications can be multiplexed in time domain on the same FPGA hardware to achieve better efficiency [Trimberger 98][Chang 98].

The fault tolerance perspective of FPGAs can be explained in two aspects: tolerance for permanent faults and tolerance for transient faults. First, for permanent faults in FPGAs, once the locations of faulty CLBs or routing resources are diagnosed, the FPGA can be reconfigured to avoid these faulty parts in the mapped circuit. These faulty parts can be replaced with previously unused resources in the same FPGA hardware. In this way, the system can still operate in the presence of faulty parts, and dependability is improved with very small hardware redundancy [Saxena 98]. Extensive research has been done to achieve this goal. For instance, *Concurrent Error Detection* (CED) techniques for detecting the occurrence of errors during run-time [Saxena 00][Huang 00a][Mitra 00], fault location techniques in FPGA-based reconfigurable systems [Mitra 98][Das 99], and FPGA reconfiguration for fault tolerance [Hanchek 98][Emmert 98][Lach 99][Mahapatra 99][Lakamraju 00].

Second, for transient faults in FPGAs, the online partial reconfiguration mechanism can recover errors in the on-chip configuration memory cells that alter the logic functionality. A typical example of the cause of such errors is Single Event Upsets (SEUs) in space environment. The occurrence rate of SEUs for applications in the projected low earth orbital path is expected to be around one per hour across three Xilinx XCV1000 FPGA devices and is crucial for space applications [Carmichael 99]. Such upsets manifest themselves as permanent faults because of the change in functionality, and they cannot be recovered by traditional transient fault recovery techniques such as *rollback* or *roll-forward* [Pradhan 96]. However, the cause of the failure is actually transient.

The online partial reconfiguration feature makes it possible to read back and modify the data in configuration memory cells of FPGAs without stopping the circuit loaded in the chip [Kelem 00]. In this way, the configuration data read back can be processed for error detection and correction. Once an error is detected and corrected, the fault-free configuration data can be written back to the chip, and the system can be recovered from failures caused by SEUs that alter the on-chip configuration data.

To minimize the interference to the applications in FPGAs, the error detection and correction process of FPGA configuration data is executed concurrently with system-level user operations in the FPGA hardware. Here, the *system-level user operations* contain all processes in the FPGA except the configuration readback and writeback processes for error detection and correction. Typical system-level user operations include normal operations for user applications, CED operations for the applications, and traditional system-level error recovery schemes targeting transient faults that do not cause errors in FPGA configuration.

In FPGAs, Look-up Tables (LUTs) in the CLBs can be configured to implement either fixed logic functions or RAM modules in user applications. In this way, however, a memory coherence problem may arise when both the system-level user operations and the online configuration error recovery scheme contend in writing data to the same RAM modules that are implemented in LUTs. In this paper, we define the memory coherence problem for the online transient error recovery of FPGA configuration and propose a "*dirty-bit*" protocol as a solution.

The organization of this paper is as follows. In Sec. 2, we briefly describe the SRAM-based FPGA architectural model and the partial reconfiguration scheme used in this paper. In Sec. 3, we present the transient error recovery scheme for FPGA applications and define the memory coherence problem in this scheme. In Sec. 4, we propose the "*dirty-bit*" memory coherence protocol for the online error recovery scheme for FPGA configuration data. In Sec. 5, we analyze the performance overhead and compare the results with the schemes without the proposed memory coherence protocol. In Sec. 6, simulation results for different application models are demonstrated. Section 7 summarizes this paper.

2. Overview of FPGA Architecture

Figure 1 shows a simplified model of the programmable logic core of FPGAs used in this paper. In this model, the programmable logic core of FPGAs consists of an array of three basic elements: *configurable logic blocks* (CLBs), *connection boxes* (CBs), and *switch boxes* (SBs).

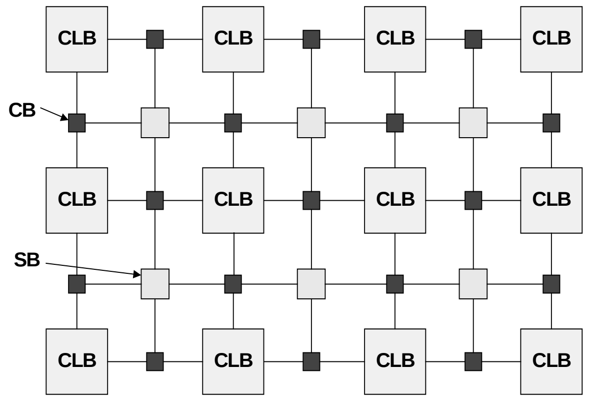


Figure 1: Architecture of the programmable logic core in FPGAs.

In SRAM-based FPGAs, a CLB generally contains SRAM lookup tables (LUTs) that store the truth tables of user-defined combinational logic functions. In this way, a combinational logic gate is realized by looking up the output value in the LUT that is addressed by the corresponding gate inputs.

LUTs can also be configured as *memory modules* in user applications. Here, the *memory modules* refer to all types of memory implemented as RAM modules, such as RAM and FIFO. Bistables such as flip-flops and latches, however, are excluded from memory modules implemented in LUTs because of area efficiency. Instead, they are implemented in addition to LUTs in each CLB to realize sequential logic. Each CLB may also contain multiplexers and other dedicated circuitry in order to enhance functional flexibility and to speed up certain paths with long delay, such as the carry chain in a ripple adder [Xilinx 00].

To complete a logic network, CLBs are connected through horizontal and vertical wiring channels located between two neighboring rows or columns. There are two types of routing devices, CBs and SBs, to direct the signal flows among CLBs and wiring channels. CBs serve as a local bridge between CLBs and the adjacent wiring channels. SBs are switch matrices that connect horizontal and vertical wiring channels. In SRAM-based FPGAs, the state of connections in these routing devices is controlled by SRAM cells, which are configured according to the desired functionality.

Logic functions in an FPGA are configured through the configuration bitstream, which contains a mixture of commands, configuration data, SRAM cell addresses for storing configuration data, and control overhead, such as the type of commands and parity checks. Configuration data contain values in LUTs and interconnect states. User memory contents are also included if they are stored in LUTs. Values of flip-flops and latches in the user applications are normally not included in configuration bitstream, unless the values are shifted to configuration memory locations.

In general, the entire configuration data of an FPGA can be partitioned vertically or horizontally into configuration data frames, which are the smallest amount of data that can be accessed with a single configuration command. Partitioning the entire configuration data into frames in an FPGA enables *concurrent partial reconfiguration* of the functional definition of the circuit. Because configuring an FPGA can be executed frame by frame, part of the FPGA configuration can be altered without stopping the operations running in the FPGA, as long as these operations do not interact with the frames under reconfiguration during the partial reconfiguration period.

Figure 2 shows an example of configuration frame partitioning in Xilinx Virtex-series FPGAs [Xilinx 00]. The entire configuration data is partitioned vertically along each column of blocks in the array. The number of frames in each column varies according to the type of configurable blocks in the column.

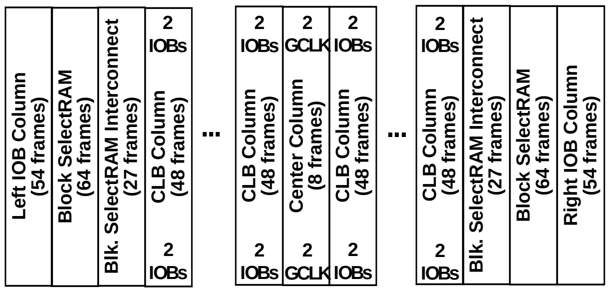

Left IOB Column (54 frames)
Block SelectRAM (64 frames)
Blk. SelectRAM Interconnect (27 frames)
2 IOBs
CLB Column (48 frames)
2 IOBs
...
2 IOBs
CLB Column (48 frames)
2 IOBs
2 GCLK
Center Column (8 frames)
2 GCLK
2 IOBs
CLB Column (48 frames)
2 IOBs
...
2 IOBs
CLB Column (48 frames)
2 IOBs
Blk. SelectRAM Interconnect (27 frames)
Block SelectRAM (64 frames)
Right IOB Column (54 frames)

Figure 2: Configuration frame partitioning in Xilinx Virtex-series FPGAs.

3. Transient Error Recovery of FPGAs

According to the effect of failures, transient faults in FPGA applications can be classified into two complementary categories: *system transients* that do not alter the FPGA configuration, and *configuration transients* that alter the FPGA configuration. Examples of system transients are SEUs that alter the user memory contents and electro-magnatic couplings that change the voltage level of a signal wire. Examples of configuration transients are SEUs that change the values in on-chip configuration memory cells that control the state of switch boxes or represent the truth table of user-defined logic functions.

For system transients, traditional transient error recovery techniques can be applied to correct the errors. Typical examples include the roll-forward and rollback recovery techniques [Huang 00b][Pradhan 96]. Basically, these approaches are designed in the system level and thus are general to recover both Application Specific Integrated Circuit (ASIC) and FPGA systems.

For configuration transients, configuration *readback* and *writeback* mechanisms can be applied to recover the system. The *readback* process of FPGA configuration refers to the operation of reading the on-chip configuration memory contents out of the chip through designated ports, and the *writeback* process refers to the operation of writing valid configuration data into the on-chip configuration memory cells. Both processes are executed in units of configuration frames.

The configuration data read back from FPGAs can be fed into a verification circuitry for error detection and correction. Once errors in the configuration data are detected and corrected, the correct configuration frames can be written back to the corresponding FPGA to complete the recovery. A typical example is illustrated in Fig. 3, which shows a dual-FPGA architecture for error recovery of FPGA configurations.

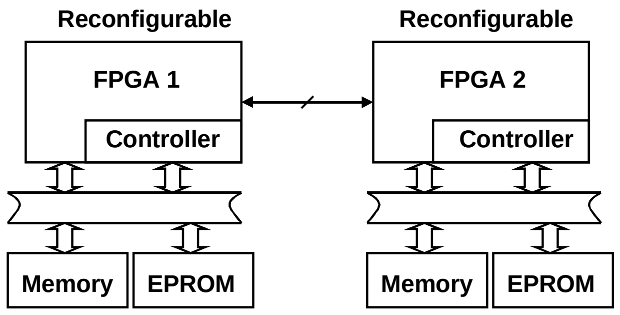


Figure 3: A dual-FPGA architecture for error recovery of FPGA configuration.

In Fig. 3, each FPGA is configured to run certain user applications with some system-level CED schemes. In addition, part of each FPGA is configured to implement the verification circuitry that performs error detection and correction of the configuration data read back from the other FPGA. In this scheme, the configuration frames or the error correcting codes (ECCs) of configuration data of both FPGAs are also stored in a safe memory space that is either replicated or protected by ECC to guarantee the correctness. Such correct configuration information is used as a verification and correction basis during the error detection and correction process of configuration data.

Note that both the traditional system-level error recovery techniques and the error detection and correction of configuration data should be used in order to recover an FPGA-based system from both system transients and configuration transients. The error detection and correction process of configuration data in each FPGA can either be scheduled periodically as a background scrubbing operation, or be applied together with the traditional system-level error recovery techniques after the CED scheme of user applications in FPGAs signals the occurrence of errors. In both cases, the error detection and correction process of FPGA configuration is executed simultaneously with system-level user operations in order to reduce the execution time overhead on user applications.

3.1 Memory Coherence Problem

The goal of configuration error detection and correction is to recover the system from configuration transients that cause errors in the part of configuration data that represents fixed logic functions or interconnect states. However, as explained in Sec. 2, some data in configuration frames may represent memory contents in user applications, and their values vary dynamically with the progress of user applications. These dynamic contents should be masked from the configuration error recovery process, and errors in these memory contents are corrected by the traditional system-level error recovery schemes.

When a writeback operation of a configuration frame is required to correct the errors in the configuration data that represents fixed logic functions, the dynamic contents in the corresponding frame are identical with those which are obtained during the most recent readback operation. In this case, a memory coherence problem will occur if system-level user operations update the dynamic contents of a configuration frame between the readback and writeback operations of the frame.

Figure 4 illustrates an example of this memory coherence problem. In this example, suppose that an 8-entry LUT in the CLB at the first row of the first column (R1C1) of the CLB array is configured as a truth table of hexadecimal value *0x04* to represent a 3-input logic function. Also, the LUT in the CLB at the second row of the same column (R2C1) is configured as a RAM module. We assume that the configuration data frames are partitioned vertically such that each frame contains data from different rows in the same column.

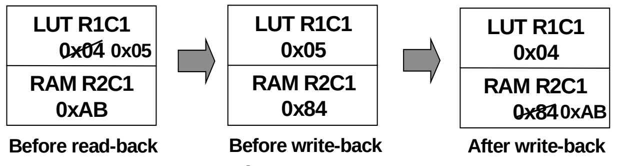


Figure 4: Example of the memory coherence problem.

Before the readback of the configuration data frames in the first column, we assume that a configuration transient occurs in the FPGA and changes the value stored in LUT R1C1 from *0x04* to *0x05*. Also, we assume that the RAM R2C1 stores a user memory content *0xAB* prior to the readback operation of the frame. These assumptions result in frames containing *0x05* in LUT R1C1 and *0xAB* in RAM R2C1 that are read back for configuration error detection and correction processes.

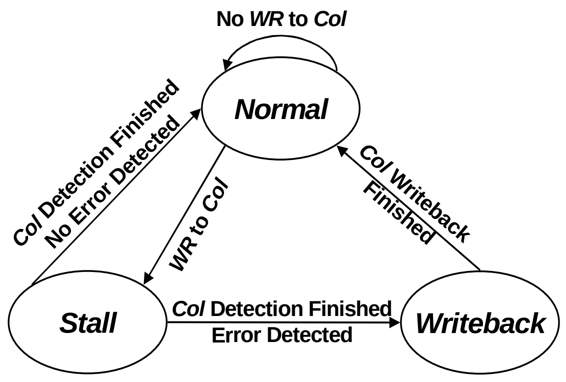
The error in LUT R1C1 (*0x05*) in the frames read back can be detected and corrected in the subsequent verification circuitry, and a writeback operation of the faulty frames is required to complete the error recovery. The values of RAM R2C1 (*0xAB*) in the frames read back are masked for the verification purpose, and they are used in constructing the frames for writeback. As a result, the values in the first column after the configuration writeback operation become *0x04* in LUT R1C1 and *0xAB* in RAM R2C1, regardless of the system-level user operations between readback and writeback.

To minimize the execution time overhead of the error recovery process, it is desired that the system-level user operations be only stalled during the writeback process of the frames. However, if the system-level user operations are not stalled between readback and writeback, the values in RAM R2C1 may change during this period. In the example in Fig. 4, suppose the system-level user operations update the values in RAM R2C1 as *0x84* before the writeback of the frames begins. If the writeback process is ignorant of the update, the old value *0xAB* in RAM R2C1 will be written back again. In this case, memory coherence of RAM R2C1 is not preserved in the system-level user operations.

[Kelem 00] explored this memory coherence problem in Xilinx Virtex-series FPGAs and suggested placing LUTs configured as memory modules in different frame slices from LUTs configured as fixed logic functions. If the frame slices that contain LUTs configured as memory modules do not contain any data that control the interconnect states, this method avoids the problem because the configuration error correction process can only change the frames that do not contain user memory modules. As a result, no write conflict to user memory modules between the configuration writeback for error recovery and system-level user operation processes can happen. However, such a method imposes an extra limitation on the placement and routing for FPGAs, which reduces the routing capability of FPGAs and increases path delays due to the separation of logic functions and local RAM or FIFO modules.

A system-level solution without adding constraints on the placement and routing is to stall the system-level user operations once a *write-enable* is signaled to the memory modules that are implemented in LUTs in the frame under configuration error detection and correction. Figure 5 shows the state diagram of this *stall-when-write* strategy for error detection and correction in configuration data. Initially, the system is in the "*normal*" state, which means that the configuration data readback operation runs without stalling system-level user operations. If there are no write operations (WR) to the column under configuration error detection and correction (Col), the system stays at the *normal* state.

When a write operation is issued to memory modules in LUTs in the column under configuration error detection and correction, the system enters a "*stall*" state and system-level user operations are paused. The *write-enable* signal of the user memory modules in LUTs in the column under configuration error detection and correction is also stalled with system-level user operations. In this way, memory contents in the column under configuration error detection and correction remain unchanged between the readback and writeback of the frames, thus ensuring the memory coherence during the partial reconfiguration.



No *WR* to *Col*
Normal
Col Detection Finished No Error Detected
WR to *Col*
Col Writeback Finished
Stall
Col Detection Finished Error Detected
Writeback

Figure 5: State diagram of the *stall-when-write* strategy.

The system leaves the *stall* state when the frames in the column under configuration error detection and correction are verified. If no error is detected, the system returns to the *normal* state, and system-level user operations continue. Otherwise, the system enters a "*writeback*" state, and correct configuration frames are written back to correct the errors in the on-chip configuration data. To avoid conflicts when system-level user operations interact with the column under reconfiguration during the writeback process, the system-level user operations are still stalled in this state. The system returns to the *normal* state after the writeback process of the frames in the column finishes.

The stall-when-write strategy is a conservative approach at the system level to preserve the memory coherence during the configuration error detection and correction. However, there are extra cycles stalled when the system-level user operations write to memory modules in an error-free column that is read back for error detection. In this case, the error-free column under configuration error detection is not required to be partially reconfigured, but the system-level user operations are stalled for memory coherence purposes. This would result in a large execution time overhead caused by configuration error detection and correction, especially when a large percentage of CLB columns contain memory modules, and the probability of memory writes in each cycle is high.

In the next section, we present a memory coherence technique to minimize the execution time overhead in system-level user operations in FPGA without extra constraints on the placement of memory blocks.

4. The Proposed Memory Coherence Scheme

To reduce the execution time overhead, it is desired to stall system-level user operations only when it is necessary. For concurrent error detection and correction of FPGA configuration, if we do not consider the memory coherence problem, the only necessary situation to stall system-level user operations is when configuration data is required to be written back. Therefore, one of the goals for designing the memory coherence scheme at the system level is to approximate the resulting execution time overhead to this lower bound.

Practically, the number of frames that require a writeback operation in one configuration error recovery period for the entire FPGA is small. This is because the SEU occurrence rate is in the order of one per hour or smaller, and the latency of configuring the entire FPGA is in the order of 10 to 100 ms in current technology [Carmichael 99]. Therefore, the first step towards minimizing the performance overhead in system-level user operations is to avoid stalling memory writes that are issued to error-free columns under configuration error detection.

To achieve this objective, one can always insert a second readback period of a faulty frame *before* it is written back, and the system-level user operations are only stalled in this second readback and the writeback periods. In this "*readback-again*" strategy, the user memory contents in the frame written back is always updated because of the second readback period, and system-level user operations are not stalled for memory writes to error-free columns that are read back for configuration error detection.

However, this "readback-again" approach still creates an unnecessary overhead by stalling the system-level user operations for the second readback of a faulty frame when the user memory contents in this frame are not updated since the first readback. This effect is more obvious when the number of memory writes per cycle in system-level user operations is small.

The extra overhead in the "readback-again" approach can be further reduced by noting that the memory coherence problem described here is analogous to that in a *shared memory multi-processor* system. In this system, multiple processors share the same memory banks, and each processor follows a protocol when a memory access is required. To guarantee the memory coherence while minimizing the performance overhead, several protocols for such a system have been proposed in the literature, such as *MESI* [Papamarcos 84] and *Firefly* [Archibald 86].

The basic concept of these protocols is to add extra flag bits in each memory line to explicitly indicate the current state of the line relative to each processor. When a processor accesses a memory line, it looks at the corresponding state bits and reacts accordingly. Also, the processor updates the status of the memory line when the access is completed. In this way, each processor has the same picture of the status of memory line to which it accesses, and potential write conflicts can thus be avoided.

A similar strategy can be deployed by adding a "*dirty bit*" to indicate the memory status in the FPGA transient error recovery scheme. In this case, there are two processes accessing the local memory blocks that are configured in CLB columns. One process is the system-level user operations, and the other is the error detection and correction process for the FPGA configuration. The dirty bit is set when the user memory contents in LUTs of the CLB column under configuration error detection are updated by system-level user operations since the most current readback.

By adding a dirty bit in each CLB column, both the system-level user operations and the configuration error recovery process can have the same knowledge of the memory status. The second readback stage for the faulty column is inserted only when the dirty bit is set, and the unnecessary overhead in the "readback-again" approach can thus be avoided.

The state diagram associated with each CLB column during the error recovery for the proposed *dirty-bit* memory coherence technique is shown in Fig. 6 and is explained as follows:

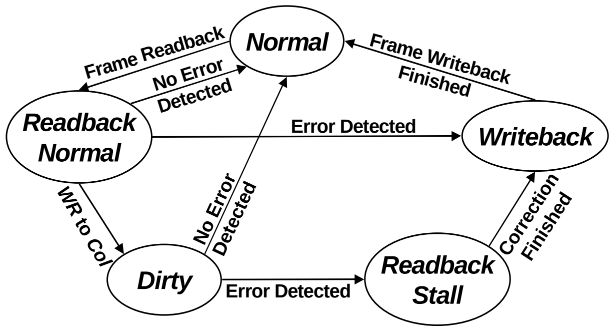


Figure 6: State diagram of the *dirty-bit* memory coherence technique.

(1) A "*dirty*" bit is associated with each CLB column to indicate that there is a memory write operation issued to that column in the system-level user operation. If the column does not contain LUTs that are configured as a memory module in user applications, this bit is always a zero.

(2) A CLB column enters the "*Readback Normal*" state when the configuration frames of that column are read back for error detection and correction. If no memory write instruction is issued from the system-level operation to this column, it stays at the same state until the error detection and correction process finishes for all configuration frames in the column.

(3) In the "*Readback Normal*" state, if no error is detected in the configuration data in the column, the column returns to the "*Normal*" state, and the configuration error recovery process continues to check the next CLB column. Otherwise, it enters the "*Writeback*" state where the correct configuration frames are written back. The system-level operations are stalled in the "*Writeback*" state, and the column returns to the "*Normal*" state after the correct configuration in the faulty frame is written back.

(4) In the "*Readback Normal*" state, if the system-level operation requests a write operation to the CLB column under configuration checking before the error detection of the column finishes, the dirty bit is asserted, and the column enters a "*Dirty*" state. The system-level operations are not stalled in the "*Dirty*" state.

(5) In the "*Dirty*" state, if no error is detected after all configuration frames in the column is examined, the column returns to the "*Normal*" state. The dirty bit is reset, and the configuration error recovery process continues to check the next CLB column. Otherwise, it enters a "*Readback Stall*" state and resets the dirty bit.

(6) A column enters the "*Readback Stall*" state only when it contains faulty configuration frames, and the RAM-configured LUTs in the column is modified by the system-level operations during the configuration error detection of the column. Since the memory contents are different from those in the most recent configuration frames read back, the frames to be written back need to be read back again so that the updated memory contents can be extracted. Also, system-level operations are

stalled because we would like to keep the memory contents coherent before the writeback begins.

(7) The updated memory contents can be extracted from the second readback and combined in the corrected configuration frame to be written back. When the correction finishes, the column enters the "*Writeback*" state, and the corrected configuration frames are written back. The system-level operations are stalled in the "*Writeback*" state, and the column returns to the "*Normal*" state after the correct configuration in the faulty frame is written back.

The implementation of the dirty-bit memory coherence technique in the system level using the dual-FPGA error recovery scheme in Sec. 3 is as follows. In Fig. 3, let us assume that the FPGA1 is the *checking device* and FPGA2 is the *device under check*. In this case, FPGA1 reads the configuration data frames from FPGA2 and start checking FPGA2. Also, FPGA1 monitors the memory addresses of write operations in the system-level user application processes of FPGA2 and checks if the addresses are in the column under configuration error detection and correction. In this way, because the checking device (FPGA1) has the information of memory activities of the device under check (FPGA2), a finite state machine can be set up in the checking device to control the memory coherence in the device under check according to the dirty-bit protocol.

5. Analyses of Performance Overhead

In this section, we analyze the total number of cycles stalled in system-level user operations during the entire FPGA configuration error detection and correction process for the "*stall-when-write*" and "*dirty-bit*" strategies described in Sec. 3 and Sec. 4. In the analyses, a cycle refers to that of the system-level user operations that are executed simultaneously with the configuration error detection and correction process.

To model applications with distributed local memory modules that are commonly implemented by LUTs, we assume that the RAM-configured LUTs are randomly distributed in the FPGA and are randomly accessed by the system-level user operations. Also, the following parameters and their notations are used throughout the analyses:

(1) Number of cycles for configuration readback, error detection and correction = *r* cycles per frame.

(2) Number of cycles for configuration writeback = *w* cycles per frame.

(3) Number of frames per CLB column in = *f*.

(4) Number of columns per FPGA = *col*.

(5) Number of memory writes per cycle in the system-level operation = m_w.

(6) Probability that a CLB column contains RAM-configured LUTs = p_r.

(7) Number of frames that contain errorneous configuration data = n_f.

5.1 Stall-when-write Strategy

The recovery latency T_{total} for a complete error detection and correction process of all FPGA configuration frames is

$$\begin{aligned} T_{total} &= Total\ (readback + writeback)\ cycles \\ &= col \cdot f \cdot r + n_f \cdot w\ . \end{aligned}$$

In each cycle, there are m_w memory write operations issued by the system-level process. Since there are ($p_r \bullet col$) CLB columns containing RAM-configured LUTs, the probability of memory writes per cycle in a CLB column that contains RAM-configured LUTs is

$$p_w = \frac{m_w}{Number\ of\ columns\ containing\ memory} = \frac{m_w}{p_r \cdot col}\ .$$

In the *stall-when-write* strategy, to avoid the potential write conflict in the memory, the system-level operations are stalled whenever a memory write is issued to the column under configuration error detection and correction. Since the latency of configuration readback and error detection and correction for each column is ($f \bullet r$) cycles, the number of cycles stalled in system-level operations during configuration readback and error detection and correction of a CLB column containing RAM-configured LUTs is

$$\begin{aligned} Stall_{col} &= \sum_{k=1}^{fr} [\,prob(stalls\ start\ at\ the\ k_{th}\ cycle) \bullet \\ &\qquad (latency\ of\ stalls\ from\ the\ k_{th}\ cycle)] \\ &= p_w \cdot f \cdot r + p_w(1-p_w)(fr-1) + p_w(1-p_w)^2(fr-2) \\ &\qquad + \cdots + p_w(1-p_w)^{fr-1} \cdot 1 \\ &= p_w \cdot \sum_{k=1}^{fr} k(1-p_w)^{f \cdot r - k} \\ &= f \cdot r + 1 - \frac{1}{p_w}\left(1-(1-p_w)^{fr+1}\right). \end{aligned}$$

Since there are n_f frames with faulty configuration, the total number of cycles stalled in the system-level process due to the configuration writeback is ($n_f \bullet w$). Therefore, for system-level operations, total number of cycles stalled during the entire configuration error detection and correction of FPGA is

$$\begin{aligned} Stall_{total} &= cycles\ stalled\ during\ configuration\ readback \\ &\qquad and\ error\ correction + cycles\ stalled\ for\ writeback \\ &= p_r \cdot col \cdot Stall_{col} + n_f \cdot w \\ &= p_r \cdot col \cdot \left(fr + 1 - \frac{1}{p_w}\left(1-(1-p_w)^{fr+1}\right)\right) + n_f \cdot w, \end{aligned}$$

and the percentage of cycles stalled during the entire configuration error detection and correction is

$$Per_{stall} = \frac{Stall_{total}}{T_{total}} \cdot 100\%$$

$$= \frac{p_r \cdot col \cdot \left(fr + 1 - \frac{1}{p_w}\left(1 - \left(1 - p_w\right)^{fr+1}\right)\right) + n_f \cdot w}{col \cdot fr + n_f \cdot w} \cdot 100\% .$$

5.2 The Proposed *"Dirty-bit"* Strategy

Under this scheme, system-level operations are only stalled when a configuration frame writeback is needed and the dirty bit is set during the configuration readback and error correction for the faulty frame. In this case, the corresponding frame is read back again for the error correction scheme to obtain the latest memory content in system-level operations before the writeback is executed. System-level operations are stalled during the second readback and the writeback periods.

To analyze the number of cycles stalled, we need to compute the probability of a dirty frame first. Since the frame is clean only when there is no memory write to it during the configuration readback and error correction period, the conditional probability of a dirty frame given that the column contains memory modules is

$$p_{dirty\,|\,column\ contains\ memory} = 1 - Prob(\,no\ memory\ write\ for\ the\ column\ in\ r\ cycles\,)$$
$$= 1 - \left(1 - p_w\right)^r .$$

Therefore, the probability that a frame to be written back is dirty is

$$p_{dirty\ frame} = prob(\,column\ contains\ memory\,) \cdot p_{dirty\,|\,column\ contains\ memory}$$
$$= p_r \cdot \left(1 - \left(1 - p_w\right)^r\right).$$

Then, we can calculate the number of cycles stalled for system-level operations during the configuration error recovery of entire FPGA as

$$Stall_{total} = n_f \cdot (p_{dirty\ frame} \cdot readback\ cycles\ for\ the\ dirty\ frame + writeback\ cycles)$$
$$= n_f \cdot \left(p_r \left(1 - \left(1 - \frac{m_w}{p_r \cdot col} \right)^r \right) \cdot r + w \right),$$

and the latency for the configuration error detection and correction is

$$T_{total} = readback\ cycles + readback\ penalty\ for\ the\ dirty\ writeback\ frames + writeback\ cycles$$
$$= col \cdot fr + Stall_{total}$$
$$= col \cdot fr + n_f \cdot \left(p_r \cdot r \left(1 - \left(1 - \frac{m_w}{p_r \cdot col} \right)^r \right) + w \right).$$

Therefore, the percentage of stalled cycles during configuration error detection and correction is

$$Per_{stall} = \frac{Stall_{total}}{T_{total}} \cdot 100\%$$

$$= \frac{n_f \cdot \left(p_r \cdot r \left(1 - \left(1 - \frac{m_w}{p_r \cdot col} \right)^r \right) + w \right)}{col \cdot fr + n_f \cdot \left(p_r \cdot r \left(1 - \left(1 - \frac{m_w}{p_r \cdot col} \right)^r \right) + w \right)} \cdot 100\% .$$

6. Simulation Results

In addition to mathematical analyses, we wrote a C-simulator to estimate the percentage of cycles stalled in the system-level user operations in the "stall-when-write" and "dirty-bit" memory coherence schemes. To be consistent with the average SEU occurrence rate measured in [Carmichael 99], we assume that only one faulty configuration frame (n_f = 1) is present in each simulation iteration. Also, we assume that the number of CLB frames per column is 48, which is identical with Xilinx Virtex-series FPGA architecture [Xilinx 00].

The configuration error recovery operations are simulated by a process that reads the entire configuration frame by frame starting from the leftmost CLB column. The read operation and the error detection and correction for each frame take *r* cycles, and a writeback operation for *w* cycles is inserted for a frame that is assumed to be faulty in that iteration. The simulation stops after the error recovery operation process is finished for all configuration frames in the FPGA.

Two different memory access and memory placement models for system-level user operations are used in the simulator: the *random access and random placement* model that simulates distributed local RAM modules (the *RAM* model), and the *sequential access and contiguous placement* model that simulates FIFO blocks (the *FIFO* model). The frequency of the memory write requests and all the other parameters described in Sec. 5 are changed in different simulation iterations to measure the performance overhead for system-level user processes under different operation environments.

6.1 RAM Model

In the RAM model, we use a random number generator to create a placement map of memory-configured LUTs in the CLB array. In each cycle of the simulation, if the system-level user operations are not stalled, another random number generator is used to generate a Bernoulli random variable with parameter m_w. This Bernoulli random variable is used to determine if a memory write operation is issued by system-level user operations in that cycle. If there is a memory write, a third random number generator is used to generate the column index of the location of the memory write. The simulator then determines the number of cycles required to be stalled for this memory write operation in system-level user operations according to the memory coherence protocol.

In Fig. 7, both the simulation and analytical results (obtained from the equations in Sec. 5) of the percentage of cycles stalled in system-level user operations are plotted in logarithmic scale. The values of *r* and *w* are assumed to be 30 cycles each, which are chosen under 3μs per frame configuration recovery latency in [Carmichael 99] and 10MHz system-level user operation clock rate environment. Also, 36 CLB columns are assumed in the FPGA, and 50% of the CLB columns are assumed to contain memory-configured LUTs in this example.

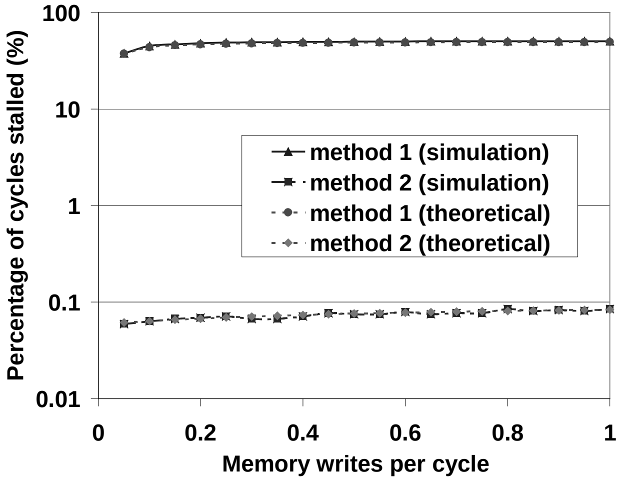


Figure 7: Percentage of cycles stalled when *r* = *w* = 30, *col* = 36, and *pr* = 0.5 for RAM model.

In Fig. 7, it is clear that the percentage of cycles stalled in the *dirty-bit* technique is under 0.1%, which is much better than the *stall-when-write* strategy. In addition, for both schemes, the simulation results and analytical estimations are very consistent.

Figure 8 (a) to (c) shows the simulation results of the percentage of cycles stalled for system-level user operations in the dirty-bit technique with respect to different environmental parameters. Besides the number of memory writes per cycle (m_w), the different environmental parameters in Fig. 8 (a) to (c) are the percentage of CLB columns with RAM-configured LUTs, the number of CLB columns in the FPGA, and the latency for the configuration error correction process per frame, respectively. The resulting percentages of stalls in system-level user operations are around 0.1%, and this overhead percentage changes little for different design environments.

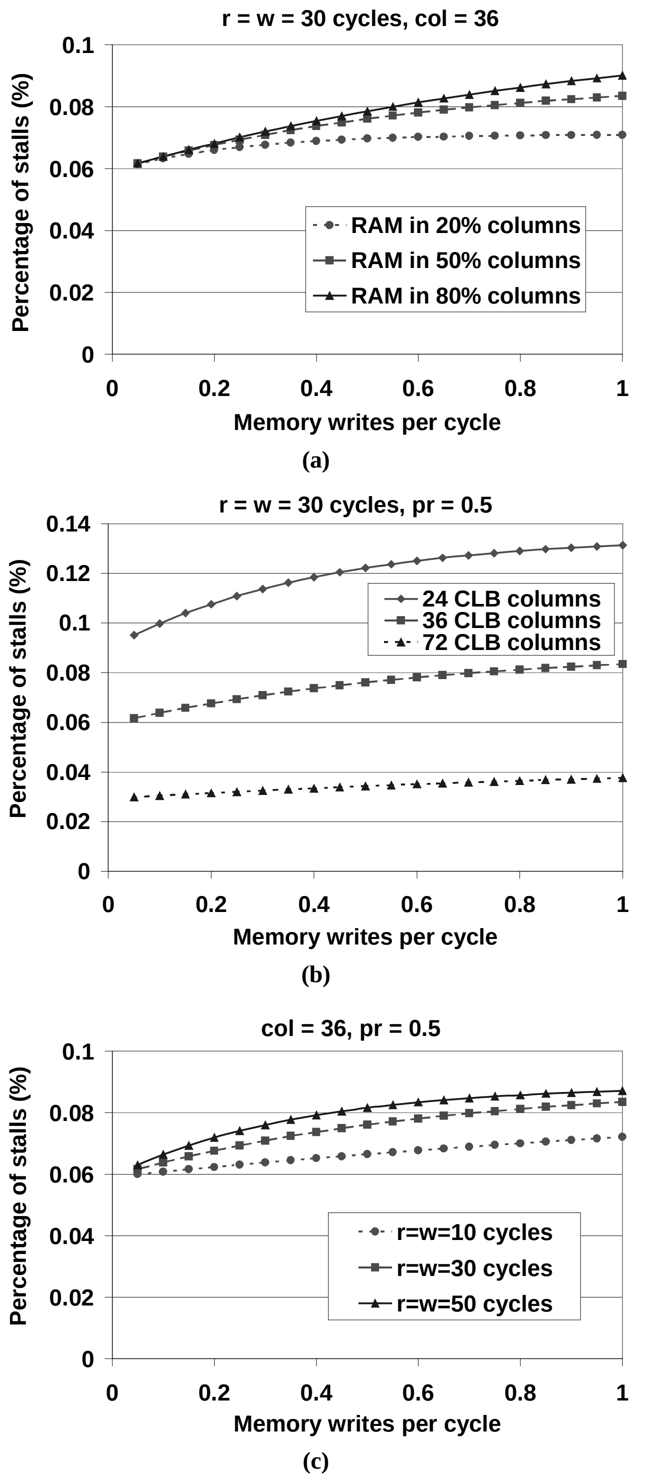


Figure 8: Percentage of cycles stalled for dirty-bit technique for RAM model.

6.2 FIFO Model

In addition to the RAM model, we also simulate a FIFO model for applications with a local FIFO that is placed in LUTs in contiguous CLB columns with sequential memory access. Before the simulation begins, we first choose the starting column of the FIFO randomly, and we assign the next ($pr \bullet col - 1$) columns to have memory-configured LUTs. The location of each memory write in system-level user operations is determined by one of two sequential orders. Those which start from the leftmost column and advance rightwards are called "*forward FIFO*", where the directions of sequential memory access are identical for both system-level user operations and the configuration readback process. Conversely, those which advance in the opposite direction are called "*reverse FIFO*". When the last column in the FIFO block is accessed, the next memory write operation wraps back to the first column.

Figure 9 (a) and (b) shows the simulation results of the FIFO model with two different access directions. Again, we assume that $r = w = 30$ cycles because of the same reason described in Sec. 6.1. We also assume that $col = 72$, and the memory write accesses from system-level user operations stay at the same CLB column for 16 times before advancing to the next CLB column. This assumption is to model the configuration of a 16-entry FIFO module implemented in each CLB column of the FIFO block.

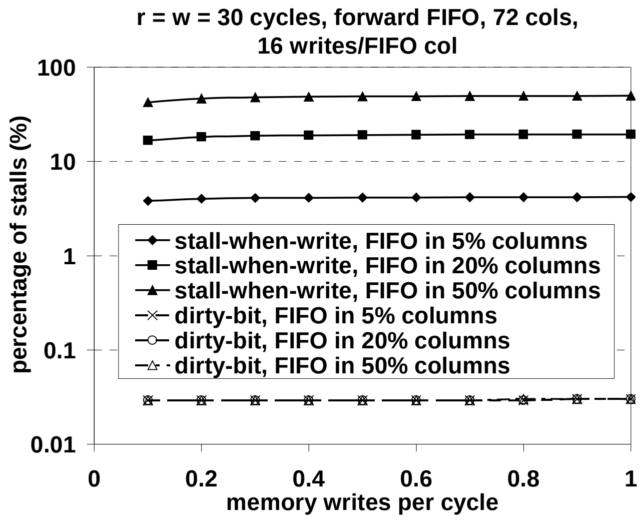
In Fig. 9, curves of the percentage of stalls in system-level user operations for different number of memory writes per cycle (m_w) are plotted in the logarithmic scale. It is clear that the "dirty-bit" technique still has a very small cycle count overhead (under 0.1%). Also, the percentages of cycles stalled in the system-level operations are relatively flat for the "dirty-bit" technique across different environmental parameters.

For the reverse FIFO model in Fig. 9(b), some of the curves for the "stall-when-write" strategy do not monotonically increase with m_w. This is because the direction of memory access in system-level user operations is opposite to that in the configuration error recovery process, and the location of memory access in system-level user operations wraps around at the end of FIFO. In this way, under certain values of m_w, both operations simultaneously access the same CLB columns for more than once during the entire error recovery period of FPGA configuration. Such cases do not occur in the forward FIFO model.

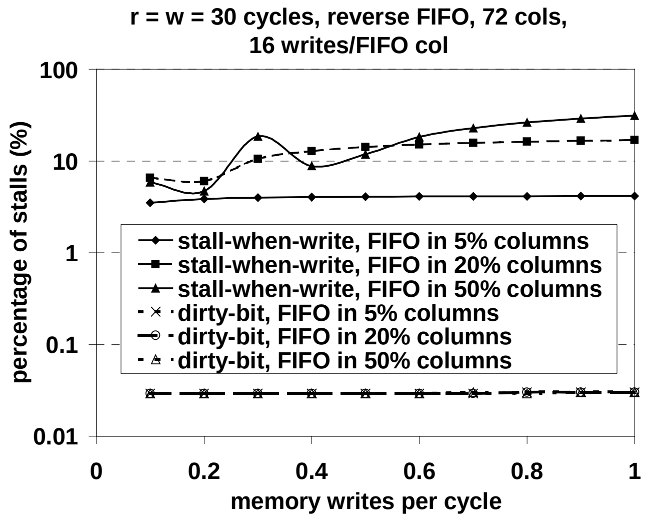
7. Summary

In this paper, we investigate a memory coherence problem for the transient error recovery of FPGA configuration. Transient errors in FPGA configuration can be recovered by configuration readback and writeback processes. Such processes are executed concurrently with system-level user operations to minimize the execution time overhead in user applications. The memory coherence problem in this scheme occurs because configuration data may contain user memory contents that are updated by system-level user operations between the configuration readback and writeback operations.

To solve the potential memory coherence problem, we propose a "dirty-bit" technique that guarantees the memory coherence during transient error recovery without adding constraints on the placement and route of the user-application memory blocks. Both mathematical analyses and simulation results show that the performance overhead in system-level user operations using the proposed memory coherence technique is very small compared to the memory coherence strategy without the additional dirty bit.



(a)



(b)

Figure 9: Percentage of cycles stalled for FIFO model.

8. Acknowledgements

The authors would like to thank Dr. Nirmal Saxena, Dr. Santiago Fernandez-Gomez, Dr. Subhasish Mitra, Philip Shirvani, Chao-Wen Tseng, Chien-Mo Li, and Shu-Yi Yu for their valuable feedback and suggestions. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

9. REFERENCES

[Altera 00] Altera Inc., *http://www.altera.com*, 2000.

[Archibald 86] Archibald, J., and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems*, pp. 273-298, Nov. 1986.

[Carmichael 99] Carmichael, C., E. Fuller, P. Blain, and M. Caffrey, “SEU Mitigation Techniques for Virtex FPGAs in Space Applications,” *MAPLD ‘99*, Sept. 1999.

[Chang 98] Chang, D., and M. Marek-Sadowska, “Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs,” *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 161-167, 1998.

[Das 99] Das, D., and N.A. Touba, “A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-Based Reconfigurable Systems,” *Proc. of IEEE International Conference on VLSI Design*, pp. 266-269, 1999.

[Emmert 98] Emmert, J. M., and D. Bhatia, “Incremental Routing in FPGAs,” *Proc. of 11th Annual IEEE International ASIC Conference*, pp. 217-221, 1998.

[Hanchek 98] Hanchek, F., and S. Dutt, “Methods for Tolerating Cell and Interconnect Faults in FPGAs,” *IEEE Trans. on Computers*, Vol. 47, No. 1, pp. 15-32, 1998.

[Huang 00a] Huang, W.-J., N. Saxena, and E. J. McCluskey, “A Reliable LZ Data Compressor on Reconfigurable Coprocessors,” to appear in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2000.

[Huang 00b] Huang, W.-J., and E. J. McCluskey, “Transient Errors and Rollback Recovery in LZ Compression,” to appear in *Proc. 2000 Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2000.

[Kelem 00] Kelem, S., “XAPP151: Virtex Configuration Architecture Advanced Users’ Guide,” Xilinx Application Note, *http://www.xilinx.com*, 2000.

[Lach 99] Lach, J., W. H. Mangione-Smith, and Miodrag Potkonjak, “Algorithms for Efficient Runtime Faulty Recovery on Diverse FPGA Architectures”, *DFT’99*, pp. 386-394, 1999.

[Lakamraju 00] Lakamraju, V., and R. Tessier, “Tolerating Operational Faults in Cluster-Based FPGAs,” *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 187-194, 2000.

[Mahapatra 99] Mahapatra, N. R., and S. Dutt, “Efficient Network-Flow Based Techniques for Dynamic Fault Reconfiguration in FPGAs”, *FTCS’99*, pp. 122-129, 1999.

[Mitra 98] Mitra, S., P. P. Shirvani, and E.J. McCluskey, “Fault Location in FPGA-Based Reconfigurable Systems,” *IEEE Intl. High Level Design Validation and Test Workshop*, 1998.

[Mitra 00] Mitra, S. and E.J. McCluskey, “Which Concurrent Error Detection Scheme to Choose?,” *Proc. International Test Conference*, 2000.

[Papamarcos 84] Papamarcos, M. and J. Patel, “A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories,” *Proc. of 11th Intl. Symposium on Computer Architecture*, pp. 348-354, 1984.

[Pradhan 96] Pradhan, D. K., *Fault-Tolerant Computer System Design*, Prentice Hall, 1996.

[Saxena 98] Saxena, N. R., and E. J. McCluskey, “Dependable Adaptive Computing Systems,” *IEEE Systems, Man, and Cybernetics Conf.*, pp. 2172-2177, Oct. 11-14, 1998.

[Saxena 00] Saxena, N. R., S. Fernandez-Gomez, W.-J. Huang, S. Mitra, S.-Y Yu, and E. J. McCluskey, “Dependable Computing and On-Line Testing in Adaptive and Configurable Systems,” *IEEE Design and Test*, Vol.17, No. 1, pp. 29-41, 2000.

[Trimberger 98] Trimberger, S., “Scheduling Designs into a Time-Multiplexed FPGA,” *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 153-160, 1998.

[Xilinx 00] Xilinx Inc., *http://www.xilinx.com*, 2000.