Fault Recovery Time Analysis for Coarse-Grained Reconfigurable Architectures

GANGHEE LEE, University of New South Wales
EDIZ CETIN, Macquarie University
OLIVER DIESSEL, University of New South Wales

Coarse-grained reconfigurable architectures (CGRAs) have drawn increasing attention due to their performance and flexibility advantages. Typically, CGRAs incorporate many processing elements in the form of an array, which is suitable for implementing spatial redundancy, as used in the design of fault-tolerant systems. This article introduces a recovery time model for transient faults in CGRAs. The proposed fault-tolerant CGRAs are based on triple modular redundancy and coding techniques for error detection and correction. To evaluate the model, several kernels from space computing are mapped onto the suggested architecture. We demonstrate the tradeoff between recovery time, performance, and area. In addition, the average execution time of an application including recovery time is evaluated using area-based error-rate estimates in harsh radiation environments. The results show that task partitioning is important for bounding the recovery time of applications that have long execution times. It is also shown that error-correcting code (ECC) is of limited practical value for tasks with long execution times in high radiation environments, or when the degree of task partitioning is high.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Failure recovery, maintenance and self-repair**;

Additional Key Words and Phrases: Coarse-grained reconfigurable architecture, triple modular redundancy

ACM Reference format:
Ganghee Lee, Ediz Cetin, and Oliver Diessel. 2017. Fault Recovery Time Analysis for Coarse-Grained Reconfigurable Architectures. *ACM Trans. Embed. Comput. Syst.* 17, 2, Article 42 (November 2017), 21 pages.
https://doi.org/10.1145/3140944

1 INTRODUCTION

With unprecedented growth in the variety of missions and applications in the harsh radiation environment of space, advanced space computing based on high-performance and low-energy architectures is becoming of paramount importance. The main challenge of designing space-based processing architectures is mitigating the effects of radiation-induced Single-Event Upsets (SEUs). An SEU occurs when a deposited charge causes a change of state in dynamic circuit elements, which results in calculation errors. Thus, the provision of tolerance of SEU-induced faults must

This research was supported in part by the Australian Research Council's Linkage (LP140100328) and Discovery (DP150103866) Projects funding schemes.
Authors' addresses: G. Lee and O. Diessel, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW, Australia 2052; emails: {ganghee.lee, o.diessel}@unsw.edu.au; E. Cetin, School of Engineering, Macquarie University, Sydney, NSW, Australia 2109; email: ediz.cetin@mq.edu.au.

https://doi.org/10.1145/3140944

be considered for space computing. However, creating radiation-hardened space processors is a lengthy, complex, and costly process. Hence, there is a large technological gap between commercial and space-based architectures. For example, recent state-of-the-art Commercial Off-the-Shelf (COTS) Field-Programmable Gate Arrays (FPGAs) (such as Virtex-6 and Virtex-7) are designed with highly advanced technologies, high density, and high performance. However, the SRAM-based configuration and data memories of these FPGAs are susceptible to radiation-induced errors. Xilinx manufactures FPGAs specifically developed for space (such as Virtex-5QV and Virtex-4QV) that are radiation hardened and radiation tolerant (Xilinx 2014). However, these devices suffer from limited capacity and performance and are relatively expensive (Glein et al. 2015).

A primary enabling technology for advanced space computing is applying spatial redundancy in the form of Triple Modular Redundancy (TMR) on COTS FPGAs. With this approach, the user design is triplicated, with each module operating in parallel and relying on a voting circuit to override a single erroneous output, provided that the other two modules agree. However, this approach may not satisfy the computing requirements of next-generation on-board space computing, which demands high complexity, high-resolution image processing, and high-compression computation (Cieslewski et al. 2008; Lovelly et al. 2014). Image processing, in particular, is becoming increasingly complex because of resolution enhancement, stereo vision, and detection and tracking of features across image frames. Currently, image processing relies on Application-Specific Integrated Circuits (ASICs), since multiprocessors or FPGAs cannot meet the performance demands at a sufficiently low power budget (Yoon et al. 2013).

Unlike FPGAs, which are fine-grained architectures with large routing area overheads and poor routability (Hartenstein 2001), Coarse-Grained Reconfigurable Architectures (CGRAs) offer word-level data paths resulting in improved performance, a reduction in place and route effort, and reduced routing area overheads. Research into CGRAs has shown that they achieve superior density per watt (Mathstar 2007; Williams et al. 2010) than FPGAs when using word-level integer operations. Although image processing has high complexity, most of the computation consists of word-level integer operations. For integer processing applications, CGRAs therefore help to satisfy the performance requirements of next-generation space applications.

While TMR-based FPGAs have been intensively researched for fault tolerance (Cetin et al. 2013, 2016; Cheatham et al. 2006; Ostler et al. 2009), there is far less research related to the fault tolerance of CGRAs. The approach in Azeem et al. (2011) presents an error recovery technique based on an instruction retry using temporal redundancy. Since the approaches of Rakosi et al. (2009) and Alnajjar et al. (2013) enhance the reliability of a CGRA by adding specialized hardware, they involve major modifications of the architecture and incur significant area penalties. The Mean Time to Failure (MTTF) due to SEUs is reported in Konoura et al. (2014). However, the authors do not address the recovery time. Han et al. (2014) implement a fault-tolerant CGRA without area penalty but do not provide a fault recovery model or a tradeoff analysis. The contributions of this article are:

- We introduce a fault-tolerant CGRA design that protects the Processing Elements (PEs), memories, and control logic. Our approach is based on a software approach that does not involve major modification of current CGRAs. The proposed design exploits spatial redundancy afforded by the PEs and uses coding techniques such as parity checks and Error-Correcting Codes (ECCs) to protect the configuration and data memories.
- While previous work has focused on architectural or software approaches to improving the reliability of CGRAs (Alnajjar et al. 2013; Han et al. 2014; Konoura et al. 2014; Rakosi et al. 2009), our work focuses on modeling the recovery time when standard CGRAs, whose architectures have not been modified, and that implement TMR in software, are subjected to radiation-induced errors.

- An application is spatially partitioned into several subtasks, or a task is temporally repeated. Irrespective of whether the application is spatially or temporally partitioned, voters are inserted at the partition boundaries to check the outputs of each partition. We examine the tradeoff between the number of partitions and recovery time, which includes the execution time of voters, while considering the area of the CGRA design and the number of SEUs that affect the recovery time.

The remainder of this article is organized as follows. Section 2 provides the background on fault-tolerant CGRAs. In Section 3, we introduce our reliable CGRA design. In Section 4, we present a novel recovery time analysis for the proposed CGRA, while in Section 5, we evaluate our recovery model for the CGRA with several kernels from space-computing applications. Finally, conclusions and future work are presented in Section 6.

2 BACKGROUND

2.1 Coarse-Grained Reconfigurable Architecture

Reconfigurable architectures are classified into two categories according to the level of reconfigurability: fine-grained architectures, such as FPGAs, which provide reconfigurability for bit-level operations, and CGRAs, which support word-level operations. Although CGRAs are less flexible when compared to FPGAs, they have higher performance-power-area efficiency for word-level operations. Furthermore, modern medium- to large-scale FPGAs are configured via a bit-stream of several tens of megabytes to program the millions of resources available on chip. In contrast, CGRAs are programmed like a processor via a very wide instruction stream. Each instruction performs a word-level operation on an individual PE of the CGRA. Thus, CGRAs provide shorter reconfiguration time due to the smaller number of functional configuration bits.

As can be observed from Figure 1, a typical CGRA consists of four parts: an array of PEs, configuration (instruction) memory, data memory, and control logic. The control logic includes an RISC processor, an external memory controller, a DMA controller, a configuration memory controller, a data memory controller, and an execution controller. Among them, the PE array, configuration memory, data memory, and their associated controllers constitute the main execution module, which is referred to as a Reconfigurable Computing Module (RCM). The RISC processor controls the DMA to copy the configurations and data for the RCM, and sets the RCM's control registers to start execution. The configuration memory is used to store the configuration context for the PE array, whereas the data memory is used to store the data consumed or produced by the PE array. Each PE in the PE array can independently perform different arithmetic operations. These operations in the PE array are configured cycle by cycle according to the context in the configuration memory.

2.2 Transient Faults in CGRAs

While fabrication defects cause permanent faults, soft errors or SEUs due to alpha particles or electromagnetic interference cause transient faults. SEUs typically perturb the logic circuits for a duration of a few hundred picoseconds (Anghel and Nicolaidis 2000), giving rise to bit flips. To ensure resilience to transient faults in memories, often an error-correcting memory is used together with circuitry to periodically read or scrub the memory of errors before the errors overwhelm the error correction circuitry. On the other hand, soft errors in the logic circuits are often detected and masked by redundancy. Duplicating or triplicating the execution units is referred to as spatial redundancy, and repeating the computation is referred to as temporal redundancy.

A system that executes two copies of an application's program is referred to as implementing DMR (Dual Modular Redundancy). Both replicas are expected to be in the same state at all times.

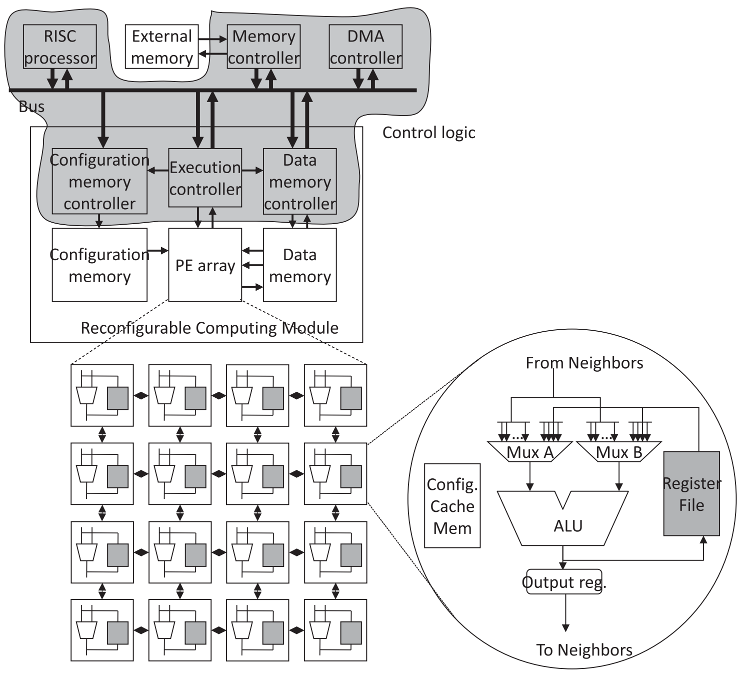


Fig. 1. Coarse-grained reconfigurable array architecture with a 4x4 PE array.

The same inputs are provided to each replica, and the same outputs are expected. The outputs of the replicas are compared using a voter that is required to detect any disagreement between them. While DMR affords error detection, it does not provide error correction since the correct output cannot be determined. A system that uses three replicas is referred to as implementing TMR. TMR incurs a larger area overhead than DMR for the benefit of determining which replica is in error when a two-to-one vote is observed. Under this condition, the voter can also output the correct result via a majority function.

2.3 Triple Modular Redundancy on CGRAs

Since a CGRA has an abundance of functional units in the PE array, it is preferable to adopt spatial redundancy for resilience against transient faults. TMR can be implemented on the CGRA by changing the configuration of the PEs using a software approach without incurring a fixed silicon area overhead (Han et al. 2014) but at the cost of additional processing cycles.

To execute an application on a CGRA, the configurations of the CGRA corresponding to the required functionality must be generated. The functionality of the application is generally expressed as a Dataflow Graph (DFG), with each node represented by a PE in the CGRA. The process of generating the configuration involves mapping an application represented by a DFG onto the PE array in the CGRA (Lee et al. 2011). Figure 2 shows an example of an Add Compare Select (ACS) operation of a Viterbi decoder mapped with TMR onto a CGRA with an 8x8 PE array. There are eight PEs in a column, and thus each TMR replica can be mapped to two PE rows as shown in Figure 2(b). These replicas are proceeded by the triplicated voters as shown in the shaded region of Figure 2(b). When TMR is applied to the PE array, the configuration memory is also triplicated for each PE replica to have its own instruction copy.

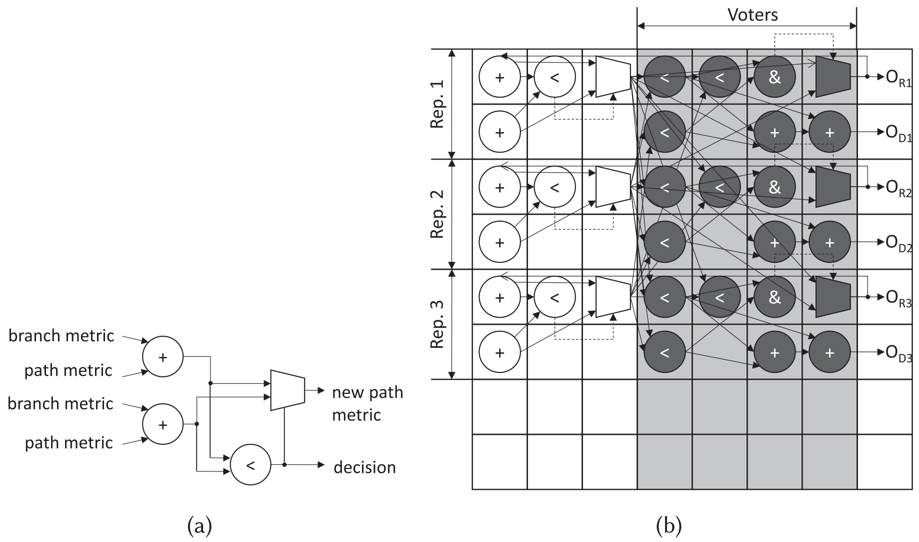


Fig. 2. An application mapping example. (a) Dataflow graph of ACS (Add-Compare-Select) operation in a Viterbi decoder. (b) Three replicas of a TMR implementation of ACS on an 8x8 PE array.

Table 1. Architecture Alternatives

Component	Fault-Tolerance Technique(s)
PE array	TMR
Configuration memory	TMR, Parity, ECC
Data memory	TMR, Parity, ECC
Controller	N/A

Since in CGRAs the PEs are mapped column-wise, the best size of the PE array for implementing TMR with high PE utilization is a multiple of three. However, in this article, we are interested in evaluating the performance of legacy CGRA architectures and implementations when TMR is applied. The selection of the best PE array size is therefore not further considered.

3 RELIABLE CGRA DESIGN

A fault occurring in the PE array can be detected and masked via TMR implementation. However, the system is also vulnerable to errors in the instruction and data memories and the supporting control circuitry. This brings about the need for techniques to detect and correct these errors. Table 1 shows some architectural alternatives for the design of a reliable CGRA with spatial redundancy. Like logic, memory modules can be protected using TMR. Furthermore, memory modules can be protected using parity or ECC. Parity indicates whether the number of ones in a memory word is even or odd. Parity allows detection of some memory errors, but cannot correct errors. By using a Hamming code (Hamming 1950) with additional parity bits, ECC can detect some errors and also correct a limited range of errors. However, implementing an ECC-protected memory incurs a larger area penalty in comparison with a parity implementation. The designer can choose to combine any of the fault-tolerance techniques referred to in Table 1 to design a reliable CGRA. For example, a PE array employing a TMR implementation and configuration (or data) memory with ECC is a possible design candidate.

When selecting a design candidate, care must be taken with the hardware overheads. For example, parity checks result in an area overhead but afford shorter recovery time as the parity helps to identify errors and narrow down the affected region in memory. In this article, we do not consider approaches that incur significant area overheads. We only consider approaches that can be easily adopted in legacy CGRAs. As a result, the insertion of a voter between the memory and the PE array is not considered as it incurs a large area penalty and requires major modification of the architecture (Alnajjar et al. 2013). For the same reason, the triplication of the controllers is also not considered. Instead, we use a software approach to implement TMR as in Han et al. (2014) for the PE array and incorporate coding techniques such as parity and ECC for the memories.

If errors originate in the controller, TMR voters implemented in the PE array may not detect any errors, since all voters may agree even though all replicas generate incorrect results. We may solve this problem by inserting detection logic into the controller, such as DMR. However, if the DMR comparator suffers an error, the system is unable to detect the error. In this article, we rely on indirect detection methods that are implemented using voters in the PE array using a software approach. The reliability enhancement of the controller requires major hardware modifications that are beyond the scope of this article.

A second consideration for designing a reliable CGRA is task partitioning for TMR. If we do not partition the application into several subtasks, we can only identify an error at the very end of execution, when the outputs are checked. Thus, we cannot identify the specific location of the error, and the longer period between checks increases the likelihood that multiple errors in multiple replicas overcome the protection afforded by TMR. In these cases, we may need to reload the entire memory contents of the CGRA and repeat execution to correct the errors. By inserting voters judiciously at the partition boundaries, the propagation of errors to the next partition is prevented, and thus the system can be made more reliable by reducing the time to isolate and repair faults. In the case of recovery, it is sufficient just to resume the task that is in error, or to update the configuration memory or data memory for that task. However, to resume the task, we need a checkpoint at each partition boundary. Checkpoints store the state, such as memory and register values, at convenient points, such as at the conclusion of each task. If we do not have a checkpoint, even though the application is partitioned into several subtasks, we cannot resume the task that is in error, but need to re-execute all the tasks of the application. Checkpointing reduces the recovery time compared with the complete re-execution of the application. However, we have to consider the number of partitions created since a large number of TMR partitions results in performance degradation as more time is spent on voting operations at the partition boundaries.

4 RECOVERY TIME ANALYSIS

For the rest of this article, we will assume that the PE array of the CGRA is protected from SEUs with TMR implemented in software; that is, all operations are triplicated and results of subtasks are voted upon. The checkpoint system used in this analytical model is based on a software approach controlled by a RISC processor in the CGRA as listed in Pseudocode 1.

As indicated in Pseudocode 1, the task in each partition, T_i, is programmed to copy the intermediate results from local memory (CGRA data memory) to external memory at each partition boundary. The checkpoint system increases the memory bandwidth and adds to the latency of an application. While the need for greater memory bandwidth cannot be avoided, the additional latency can be hidden by parallelizing T_i's communication (saving checkpoints) with T_{i+1}'s execution. When errors, which cannot be masked by voters, are detected via voters in task T_i, the erroneous task is re-executed until at least two out of three voters agree or the number of errors does not exceed the maximum allowed. If the errors are judged to stem from the local memory,

Pseudocode 1: Checkpoint System

```
foreach partitioned task, $T_i$ do
    ErrorCount = 0;
    do
        if ErrorCount is not zero and errors stem from the local memory then
            reload data for $T_i$ from the external memory;
        end
        execute $T_i$;
        if at least two out of three voters in $T_i$ agree then
            ErrorCount = 0;
            save voted results of $T_i$ to the external memory;
        else
            ErrorCount++;
            if ErrorCount exceeds a threshold then
                terminate $T_i$ and reset the system;
            end
        end
    while ErrorCount is not zero;
end
```

the task's source data needs to be reloaded before re-executing the task. When the error cannot be cleared, execution of the task is terminated, the system is reinitialized, and execution recommences with task T_0.

In this section, we introduce a mathematical recovery time model based on the proposed checkpoint system. Then, we evaluate the tradeoff between the recovery time and the application execution time.

4.1 Recovery Time Modeling

If an SEU affects the CGRA, then this could be observed as discrepancies in the output of the three TMR replicas. In this case, assuming only one replica is affected, we can continue processing after the voted results have been stored. This is referred to as an "error" in the CGRA. However, if SEUs affect two or more replicas, then this is observed as a disagreement between all voters and leads to a so-called failure of the CGRA. In this case, we need to re-execute the failed task after recovery processing. Recovery processing refers to the activities undertaken to recover the normal state of operations and to resume processing from the point of execution when an error was detected. In detail, SEUs may be observed in the (1) PE array, (2) configuration and data memory, or (3) control logic. (1) If the failure stems from errors in the PE array, it might be corrected by a simple re-execution without any recovery processing. When checkpoints are used, we can re-execute from the last checkpoint. Otherwise, we need to restart from the beginning of the application. (2) If the failure was caused by memory errors, we need to reload the corrupted memory region with the correct data before re-execution. (3) Finally, if the failure originated in the control logic, registers in the control logic need to be reloaded, which involves reinitializing the system.

A state diagram showing transitions between CGRA system states and their corresponding recovery times is depicted in Figure 3. As can be observed, there are four states that represent the status of the CGRA:

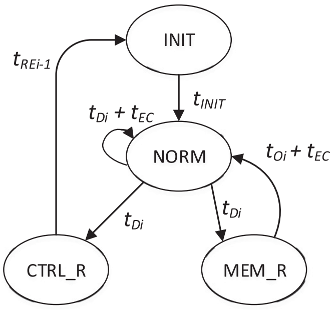


Fig. 3. State transitions between CGRA system states and their corresponding recovery times.

1) *INIT* state: The CGRA commences initialization. The time for initialization, denoted as t_{INIT}, can be expressed as

$$t_{INIT} = t_{OS} + t_{DMA} + \sum_{i=0}^{k-1} t_{CM_i} + \sum_{i=0}^{k-1} t_{DM_i} + t_{EC}, \tag{1}$$

where t_{OS} represents the boot time of the RISC processor, t_{DMA} the DMA setup time, t_{CM_i} the time to copy configurations for task T_i from the external memory to the configuration memory, t_{DM_i} the time to copy source data for task T_i from the external memory to the data memory, and t_{EC} the setup time for the execution controller, respectively, whereas k represents the number of tasks the application is partitioned into, namely, $\{T_0, T_1, \ldots, T_{k-1}\}$. Since booting the CGRA is a one-time operation, t_{OS} does not critically affect the recovery time of the CGRA. However, in the event that checkpoints are not used and the application's execution time is less than t_{OS}, it will dominate the recovery time, since the system must be reset. In the proposed CGRA, the DMA and execution controller registers are programmed by the RISC processor via memory-mapped IO. Therefore, t_{DMA} and t_{EC} are proportional to the number of internal registers of the hardware unit, and t_{CM_i} and t_{DM_i} are proportional to the amount of source data for the application.

2) *NORM* state: The CGRA operates in TMR mode executing each partitioned task. If there is an SEU, this is detected by the voters at the partition boundary. The time to detect an error in the partitioned task T_i, denoted as t_{D_i}, can be expressed as

$$t_{D_i} = MAX(t_{E_i}, t_{M_i}) \tag{2}$$

with

$$t_{E_i} = t_{R_i} + t_{V_i}, \tag{3}$$

where t_{E_i} represents the execution time of the partitioned task T_i, t_{R_i} represents the replicated module's execution time, and t_{V_i} is the voting time. t_{R_i} is determined by the number of partitions, and t_{V_i} is proportional to the number of data values passing between the partitioned tasks. t_{M_i} represents the error detection (and/or correction) time of the memory used for the partitioned task T_i. If we add additional hardware logic for error detection and correction to the memory, the error correction and detection operations can be undertaken concurrently with t_{E_i}. Then t_{D_i} is defined as the maximum of t_{E_i} and t_{M_i}. Note that t_{E_i} is only valid when voters, which are implemented in the PE array, detect errors. If a voter does not detect the error, for example, when the error occurs in the execution controller, t_{E_i} is not bounded by the execution time of the partitioned task. These errors cannot be detected or corrected by TMR and will therefore not be considered in this article.

If the detection is classified as an "error," we can continue processing due to the error masking provided by voters. However, if the detection is observed as a "failure," the failed task must be

Table 2. Memory Error Detection Results

Number of SEUs in Each Replica			Detection Result for 3 Different Memory Protection Methods		
Rep0	Rep1	Rep2	TMR	TMR+Parity	TMR+ECC
1bit	-	-	Error	Error	-
1bit	1bit	-	Failure	Failure	-
1bit	1bit	1bit	Failure	Failure	-
2bit	1bit	1bit	Failure	Failure	Error
2bit	2bit	1bit	Failure	Failure	Failure

re-executed. If there is a checkpoint at each boundary of the partitioned task T_i, we can resume execution after the checkpointed state has been reloaded. We classify the failure into three cases according to the source of the error, be it the PE array, memory, or control logic.

If a failure originated in the PE array, the recovery time for task T_i, denoted as $t_{R\_PE_i}$, represents the state transition from the *NORM* state to the *NORM* state in Figure 3 and can be expressed as

$$t_{R\_PE_i} = t_{D_i} + t_{EC}, \tag{4}$$

where t_{D_i} is the detection time of task T_i, and t_{EC} is the setup time for the execution controller, which represents the time to reprogram the execution controller to resume the failed task T_i.

Coding techniques, such as ECC, result in an area overhead in the memory block. However, with coding, we can often identify the specific SEU location and may even be able to correct it. If no SEU is found in the memory when the voter indicates a failure, we presume that the SEUs exist in the PE array. However, if we find SEUs in the configuration memory or the data memory, the memory contents must be corrected.

3) *MEM_R* state: If a failure originated in the configuration memory or the data memory, we need to overwrite the respective memory contents by reloading them from the external memory. The time for overwriting the memory for task T_i, denoted as t_{O_i}, can be expressed as

$$t_{O_i} = t_{DMA} + t_{CM_i} + t_{DM_i}, \tag{5}$$

where t_{CM_i} represents the reload time of the configuration memory for task T_i, t_{DM_i} represents the reload time of the data memory for task T_i, and t_{DMA} is the setup time of the DMA controller. In particular, if we have memory protected by parity, then t_{O_i} becomes the time to overwrite the number of words of the memory where the error has occurred. Alternatively, if we have ECC-protected memory, we do not need to overwrite the memory contents for recovery. However, when the number of errors exceeds the capability of the ECC, it may experience a failure, which then requires overwriting. Table 2 lists the ability of memory to detect errors according to the various number of SEUs and protection methods employed. ECC is assumed to have a 1-bit error correction capability per replica.

Recovery from a failure of the memory for task T_i follows the state transitions along the *NORM* − *MEM_R* − *NORM* path, which can be expressed as

$$t_{R\_MEM_i} = t_{D_i} + t_{O_i} + t_{EC}. \tag{6}$$

4) *CTRL_R* state: If the error persists after the memory is overwritten, we infer that the error must exist in the control logic or that errors are so severe that TMR or overwriting the memory cannot solve them. In this case, we need to clear errors by reinitializing the system.

Table 3. Recovery Time According to the Presumed Error Type

Recovery Scheme	Presumed Error Type	Recovery Time
Checkpoint	Error	-
	Failure in PE	$t_{D_i} + t_{EC}$
	Failure in memory	$t_{D_i} + t_{O_i} + t_{EC}$
	Failure in controller	$t_{D_i} + t_{RE_{i-1}} + t_{INIT}$
No checkpoint	Error	-
	Failure	$t_{D_i} + t_{RE_{i-1}} + t_{INIT}$

The recovery from a failure of the controller for task T_i follows the state transitions along the $NORM - CTRL\_R - INIT - NORM$ path, which can be expressed as

$$t_{R\_CTRL_i} = t_{D_i} + t_{RE_{i-1}} + t_{INIT} \tag{7}$$

with

$$t_{RE_{i-1}} = \sum_{j=0}^{i-1} t_{D_j}, \tag{8}$$

where $\sum_{j=0}^{i-1} t_{D_j}$ represents the summation of the re-execution time from the first partition, T_0, to the $(i-1)$-th partition, T_{i-1}, following the detection of a failure in the control logic during the execution of task T_i. Note that we add $t_{RE_{i-1}}$ to the recovery time of the controller, since all tasks that have been executed so far need to be re-executed. It should be noted that reinitializing the system involves reloading the memory for all tasks. This may appear conservative, but since we cannot categorically establish the origin of the error, it enhances the likelihood that further control errors can be avoided.

Table 3 summarizes the recovery time according to the presumed error type. Note that if we experience an error in the configuration memory, we do not recover the errors even though errors in the configuration memory can result in ongoing execution errors while the task is being processed. In Alnajjar et al. (2013), to prevent error propagation, configurations are triplicated and voted upon. The resulting output instruction is also then written back to the configuration memory on every clock cycle. However, this approach results in considerable area overhead. Our approach solves this problem in software by only reloading memory contents when a failure is detected.

4.2 Iterative Method to Detect the Presumed Error Type

As shown in Table 3, the only case when we need to classify the origin of an error is when we have checkpoints and when we have detected the presumed error as a failure. If we have a detection unit, such as parity in the memory, we can easily classify the presumed error type and determine which recovery to undertake. However, if there is no detection logic, we identify errors iteratively. Using an iterative approach, the time to detect a failure for task T_i, denoted as t_{F_i}, can be expressed as

$$t_{F_i} = h_{PE} \times t_{R\_PE_i} + h_{MEM} \times t_{R\_MEM_i}, \tag{9}$$

where h_{PE} and h_{MEM} represent threshold values for observing repeated failures in the PE and the memory, respectively. In the iterative approach, t_{F_i} is added to the total recovery time as shown in Table 4, which lists t_{F_i} according to the various protection schemes. For example, in the TMR case, when $h_{PE} = 2$ and $h_{MEM} = 1$, the system allows two consecutive recovery processes corresponding to the PE failures followed by one recovery process corresponding to a memory failure. If a failure is detected in task T_i, initially we determine it to be a PE failure, which has the smallest

Table 4. Time to Detect a Failure Using an Iterative Approach

Protection Scheme	Presumed Error Type	Time to Detect a Failure, t_{F_i}
TMR	PE	0
	Memory	$h_{PE} \times t_{R\_PE_i}$
	Controller	$h_{PE} \times t_{R\_PE_i} + h_{MEM} \times t_{R\_MEM_i}$
TMR+Parity	PE	0
	Memory	0
	Controller	$h_{MEM} \times t_{R\_MEM_i}$
TMR+ECC	PE	0
	Memory	0
	Controller	0

recovery time. However, if the system experiences a second and a third failure, after two consecutive PE recovery processes for task T_i, the third failure is classified as a memory failure. Having performed a memory recovery process for task T_i, if the system experiences a fourth failure of task T_i, it is finally classified as a controller failure.

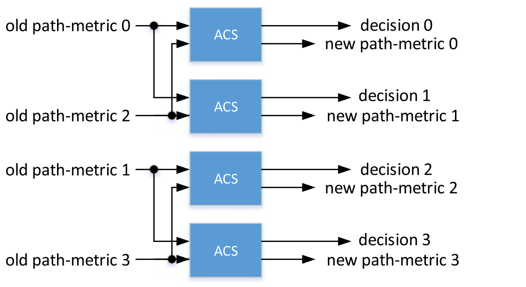
4.3 Considerations for Task Partitioning

The objective of a partitioning algorithm is to find a minimum cut with the following considerations: (1) Finding partitions that bound the maximum recovery time or minimize the average recovery time. Whether the objective is bounding the maximum time or minimizing the average time, the resulting recovery time is influenced by the application execution time. This relationship between the recovery time and the execution time will be discussed in detail in Section 5. (2) Minimizing the total number of edges between partitioned tasks. At the partition boundary, voters are inserted according to the number of outgoing edges and each inserted voter increases the total execution latency. If there is a dependency between the two partitioned tasks, we have to pay the cost of increasing the number of voter cycles and the cost of storing the values to the memory. The number of voter cycles increases since only three voters can be executed concurrently when the height of the array is restricted to eight rows but more than one output per module may need to be voted upon. The cost of storing the values also increases because space has to be provided for the interpartition results in local and external memories to store checkpointed values. (3) Maximizing the reliability of the system. A large number of partitions improves the reliability of the system, but it increases the total execution time.

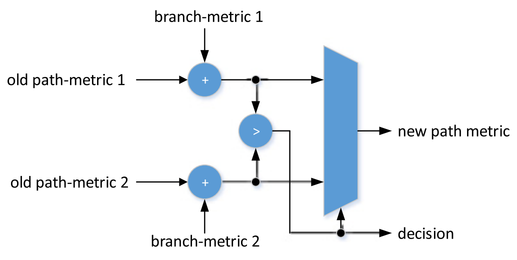
Apart from these considerations, there may be many other objectives to be solved. However, in this article, we focus on the first consideration, which concerns the relationship between the recovery and execution times.

5 EVALUATION

In order to evaluate the proposed recovery model, we have modeled a reliable CGRA that employs TMR in software and examined adding both parity and ECC logic for error detection and correction of the memory. The proposed analytical model supports arbitrary partition sizes. It is envisaged that task partitioning could be applied in a spatial manner, in which case voters are inserted between the subtasks of an application, or in a temporal manner, in which case voters are inserted between repeated tasks. While CGRAs with their limited configuration memory are usually designed to accelerate repeated kernels, partitioning is often implemented temporally. In this section, we evaluate the temporal partitioning of a repeated kernel using the ACS operations of a



(a) Butterfly operation between ACS operations (K=3)



(b) Detail of the ACS unit

Fig. 4. Block diagram of the ACS operation.

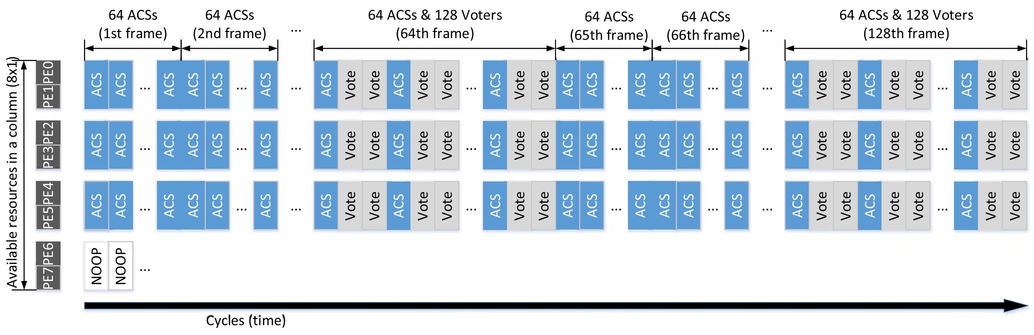


Fig. 5. An example of ACS implementation with TMR on an 8x8 PE CGRA with partitions, k = 2.

Viterbi decoder as an example. Following this, the effectiveness of our recovery model is further evaluated with practical space kernel examples.

5.1 Mapping ACS Operations onto the CGRA

Since the ACS function is one of the most time-consuming operations of a Viterbi decoder, it is commonly accelerated with special hardware like that afforded by CGRAs. Figure 4 shows an example block diagram of an ACS operation with constraint length $K = 3$. The total number of ACS operations is determined by the constraint length, K, and the frame length, F. Table 5 lists the application and architecture parameters that we used to map ACS operations onto the CGRA. With the constraint length $K = 7$, as has been used in the Voyager space probe program (Deutsch and Miller 1982), the application requires $y = 2^{K-1} = 64$ ACS operations in parallel for one frame, and the frame is repeated $L = F + K + 1 = 128$ times. However, due to the resource constraints of the proposed 8x8 PE array, only one ACS operation per replica can be executed at a time, as shown in Figure 2(b). Thus, we implement the application with temporal mapping (Lee et al. 2011), whereby configurations can be changed on each clock cycle. In temporal mapping, the CGRA schedules operations only with the first column (8x1 PEs), and the other seven columns execute the same operations as the first column, but in successive clock cycles, as instructions flow from the left side of the PE array to the right.

Figure 5 shows the result of scheduling 128 frames onto 8x1 PEs with TMR using $k = 2$ partitions. In the first frame, the execution controller in the CGRA fetches the ACS operation 64 times from the configuration memory. Each ACS operation reads input data (old path-metric and branch-metric) from the data memory, executes, and stores results (new path-metric and decision) to the

Table 5. Application and Architecture Parameters for the ACS Application

Symbol	Description	Value
K	Constraint length	7
F	Frame length	122
y	Number of ACS operations within 1 frame, $y = 2^{K-1}$	64
L	Loop count, $L = F + K + 1$	128
MxN	PE array size	8×8
k	Number of partitioned tasks	-
II	Initiation interval (given the resource constraints of 2 read and 1 write memory ports)	2
x	Number of execution cycles for 1 ACS operation	6
CF	Clock frequency	1GHz
t_f	Application execution time for 1 frame, $t_f = (x + II \times (N-1)) \times (y/N)/CF$	160ns
v	Voter cycles	4
a	Number of output data for 1 ACS operation	2
x_v	Number of execution cycles for 1 ACS operation with voter, $x_v = x + a \times v$	14
t_{fv}	Application execution time with voter for 1 frame, $t_{fv} = (x_v + II \times (N-1)) \times (y/N)/CF$	224ns
W_{CM}	Configuration memory bit width	128
W_B	Data bus bit width	32
t_{CW}	Transfer time for 1 configuration word, $t_{CW} = (W_{CM}/W_B)/CF$	4ns
b	Number of input data for 1 frame	64
W_{DM}	Data memory bit width	16
t_{DW}	Transfer time for 1 data word, $t_{DW} = (W_{DM}/W_B)/CF$	0.5ns
e_C	Number of words that are detected as having parity error in the configuration memory	-
e_D	Number of words that are detected as having parity error in the data memory	-
t_{DMA}	DMA setup time	36ns
t_{EC}	Execution controller setup time	47ns
t_{OS}	OS booting time	800ns
h_{PE}	Threshold value to detect memory failure	1
h_{MEM}	Threshold value to detect controller failure	1

data memory. The other frames are the same as the first frame except for the 64th and the 128th frames, which are located on the partition boundaries, and therefore include two voting operations (one for the path-metric and one for the decision results) along with the ACS operation. Since the size of the proposed PE array is 8x8, we have eight columns and they can therefore execute eight ACS operations in parallel. However, since the number of memory ports is restricted (we have two read and one write port per row of the PE array), we cannot have all eight columns executing the same instructions. The second column thus cannot start its read operations until the first column finishes its read operations. The difference between the start time of the current column and the start time of the next column is referred to as the Initiation Interval (II), which is two in this example. The total execution time of the application for one frame, denoted as t_f, can be calculated as

$$t_f = (x + II \times (N-1)) \times (y/N)/CF,$$

where x is the number of execution cycles for one ACS operation, N is the number of PEs in a row in the PE array, y is the number of total ACS operations within one frame, and CF is the clock frequency. In this example, the execution time for one frame, t_f, is $(6 + 2 \times (8 - 1)) \times (64/8)/109 = 160ns$. When we partition an application into several subtasks, voters are inserted at partition boundaries, as can be seen at the 64th and the 128th frames in Figure 5. The execution cycles including voter operations, denoted as x_v, can be calculated as $x_v = x + a \times v = 14$, where a is the number of output data for one ACS operation and v is the number of execution cycles for a voter. Then, the number of execution cycles per frame with voters, denoted as t_{fv}, is $(14 + 2 \times (8 - 1)) \times (64/8)/109 = 224ns$.

If we assume the memory detection time for task T_i, t_{M_i} is smaller than the execution time for task T_i, t_{E_i}, we can derive the detection time of task T_i, t_{D_i}, as

$$t_{D_i} = MAX(t_{M_i}, t_{E_i}) = t_{E_i} = (L/k - 1) \times t_f + t_{fv}.$$

The exact number of execution cycles for an application varies according to the characteristics of the CGRA. However, most advanced CGRAs use loop-level optimizations such as loop pipelining, which is also used in our analysis.

As shown in Figure 5, an identical ACS operation is used for all iterations. Thus, the overwrite time of the configuration memory for task T_i, t_{CM_i}, is fixed by the instruction count for one ACS operation, which is calculated as

$$t_{CM_i} = x \times (W_{CM}/W_B)/CF = x \times t_{CW},$$

where $t_{CW} = (W_{CM}/W_B)/CF$ is the transfer time of one configuration word from the external memory to the configuration memory. Note that the transfer time is bounded by the bandwidth of the system bus.

In contrast to the fixed overwrite time of the configuration memory, the overwrite time of the data memory for task T_i, t_{DM_i}, is directly related to the number of partitions, which is calculated as

$$t_{DM_i} = (b \times L/k) \times (W_{DM}/W_B)/CF = (b \times L/k) \times t_{DW},$$

where b is the number of input data to copy from the external memory to the data memory for one frame, and $t_{DW} = (W_{DM}/W_B)/CF$ is the transfer time of one data word from the external memory to the data memory. Note that t_{CM_i} and t_{DM_i} are the time to overwrite all the data for task T_i. If we have parity, we do not need to overwrite all the data. Only the words that are detected as having parity errors will be overwritten. Therefore, t_{O_i} should be calculated as follows for the three different protection schemes:

$$t_{O_i}(TMR) = t_{DMA} + t_{CM_i} + t_{DM_i}$$
$$t_{O_i}(TMR + Parity) = t_{DMA} + e_C \times t_{CW} + e_D \times t_{DW}$$
$$t_{O_i}(TMR + ECC) = t_{DMA} + e_C \times t_{CW} + e_D \times t_{DW},$$

where e_C is the number of words that are detected as having parity errors in the configuration memory and e_D is the number of words that are detected as having parity errors in the data memory.

5.2 Recovery Time Analysis for ACS

Based on Table 5, we have calculated the time for each of the state transitions in Figure 3. Table 6 reports the maximum recovery times of the ACS operation when checkpoints are available. As described in Table 3, when checkpoints are not available in the system, the recovery time for task T_i is simply given by the $t_{R\_CTRL_i}$ columns of Table 6 since re-executing a task is not possible.

Table 6. Maximum Recovery Time for ACS with Checkpointing

Number of Partitions, k	Application Execution Time (us)	Recovery from PE Failure, $t_{R\_PE_i}$ (us)	Recovery from Memory Failure, $t_{R\_MEM_i}$ (us)			Recovery from Controller Failure $t_{R\_CTRL_i}$ (us)		
			TMR	TMR+ Parity	TMR+ ECC	TMR	TMR+ Parity	TMR+ ECC
1	20.5	20.6	66.0	20.7	0	92.5	47.1	**<u>26.4</u>**
2	20.6	10.4	33.3	10.4	0	59.8	36.9	26.5
4	20.7	5.2	16.9	5.3	0	43.5	31.9	26.6
8	21.0	2.7	8.7	2.8	0	35.6	29.6	26.8
16	21.5	1.4	4.6	1.5	0	32.0	**<u>28.9</u>**	27.3
32	22.5	0.8	2.6	0.8	0	**<u>31.0</u>**	29.2	28.4
64	24.6	0.4	1.5	0.5	0	32.0	31.0	30.4
128	28.7	**<u>0.3</u>**	**<u>1.0</u>**	**<u>0.4</u>**	0	35.6	34.9	34.5

We assume that two SEUs are detected during the execution, and that these cause a failure of the CGRA, as shown in Table 2.

For example, in Table 6, when the number of partitions $k = 2$, recovery from PE failure for task T_i is given by Equation (4):

$$t_{R\_PE_i} = t_{D_i} + t_{EC} = (L/k - 1) \times t_f + t_{fv} + t_{EC} = 10.4us.$$

The recovery from a memory failure for task T_i, denoted as $t_{R\_MEM_i}$, is determined by Equation (6) and Equation (9):

$$\begin{aligned}
&t_{R\_MEM_i}(TMR) = t_{F_i} + t_{D_i} + t_{O_i} + t_{EC} = (h_{PE} + 1) \times t_{R\_PE_i} + t_{O_i}(TMR) = 33.3us \\
&t_{R\_MEM_i}(TMR + Parity) = t_{D_i} + t_{O_i} + t_{EC} = t_{R\_PE_i} + t_{O_i}(TMR + Parity) = 10.4us \\
&t_{R\_MEM_i}(TMR + ECC) = 0.
\end{aligned}$$

In particular for the $TMR + ECC$ case, the recovery time is zero, since the memory is protected by ECC, and thus the reload and re-execution of the operation is unnecessary. Finally, the recovery from a controller failure for task T_i is derived from Equation (7) and Equation (9):

$$\begin{aligned}
t_{R\_CTRL_i}(TMR) &= t_{F_i} + t_{D_i} + t_{RE_{i-1}} + t_{INIT} \\
&= h_{PE} \times t_{R\_PE_i} + h_{MEM} \times t_{R\_MEM_i} + t_{RE_i} + t_{INIT} = 59.8us \\
t_{R\_CTRL_i}(TMR + Parity) &= t_{F_i} + t_{D_i} + t_{RE_{i-1}} + t_{INIT} \\
&= h_{MEM} \times t_{R\_MEM_i} + t_{RE_i} + t_{INIT} = 36.9us \\
t_{R\_CTRL_i}(TMR + ECC) &= t_{D_i} + t_{RE_{i-1}} + t_{INIT} = 26.5us.
\end{aligned}$$

Since this is the maximum recovery time, when we calculate $t_{R\_CTRL_i}$, we set $t_{D_i} + t_{RE_{i-1}}$ to be equal to the total application execution time ($t_{D_i} + t_{RE_{i-1}} = t_{RE_i} = t_{E_0} + t_{E_1} = 20.6us$).

In Table 6, the smallest (which is the best) recovery time is highlighted in bold and underlined. The time to recover from a PE failure or a memory failure decreases dramatically in conjunction with an increase in the number of partitions, k, because of the impact of t_{D_i}, which decreases as k increases. However, the minimum recovery time from a controller failure has a distinct trough due to the effects of both t_{D_i} and t_{RE_i}, which increase with the increase in k.

Table 7. Area and Utilization of Each CGRA Component by Protection Scheme

CGRA Component	Protection Scheme	*Area	Utilization
Control logic	-	1.00	1
PE array	TMR	6.32	0.75
Configuration memory (128 bits wide)	TMR	1.44	0.1
	TMR+Parity	1.45	
	TMR+ECC	1.54	
Data memory (16 bits wide)	TMR	15.43	1
	TMR+Parity	17.43	
	TMR+ECC	21.28	

*Area is normalized to control logic.

Our analysis indicates that, in a CGRA, the degree of task partitioning plays a crucial role in determining the optimum recovery time. Although this evaluation is based on a specific CGRA, which has an 8x8 PE array and uses a temporal mapping scheme, the proposed mathematical models can be used to analyze any case by adapting the parameters given in Table 5 to the specific characteristics of a given CGRA and application.

5.3 Recovery Time Scaled by the Area Overhead

As shown in Table 6, since recovery from a controller failure usually takes longer than recovery from a PE or memory failure, we have to pay special attention to controller failure. To verify the general effect of a controller failure, we have analyzed the average recovery time, which is scaled by the area overhead. Table 7 compares the area of CGRA components with that of the control logic. From the baseline architecture in Kim and Mahapatra (2011), we increased the size of the data memory to store the complete source dataset needed for ACS operations. Since we use a software approach for the TMR implementation, area overhead only comes from the error detection and correction logic of the memory block. The area overhead of parity is calculated assuming that 1 bit of parity is added to each word in each replica. The area overhead of ECC, which has Single-Error-Correction-Double-Error-Detection (SEC-DED) capabilities, has been obtained from Naseer and Draper (2008) and calculated according to the memory bit width.

When we assume that the error rate is directly and only proportional to the area, we can obtain the average recovery time by scaling the recovery times by the area of each component in Table 7. Note that increasing the number of partitions does not affect the area of the computation, but the time spent checking results in voters does affect the time needed for execution. Figure 6 shows the average recovery time according to the number of partitions, which is scaled according to the respective areas of the affected components, denoted as $t_{R\_AVERAGE_i}$, and calculated as

$$t_{R\_AVERAGE_i} = \alpha \times t_{R\_PE_i} + \beta \times t_{R\_MEM_i} + \gamma \times t_{R\_CTRL_i}, \tag{10}$$

where α represents the proportion of the total CGRA area allocated to the PE array, β represents the proportion of CGRA area allocated to the memory, and γ represents the proportion of CGRA area allocated to the control logic. Table 7 also lists the resource utilization of the CGRA components. Since SEUs in unused (nonutilized) resources do not cause an error, we multiply the utilization of CGRA components by the area weights to account for the susceptibility to error of each CGRA

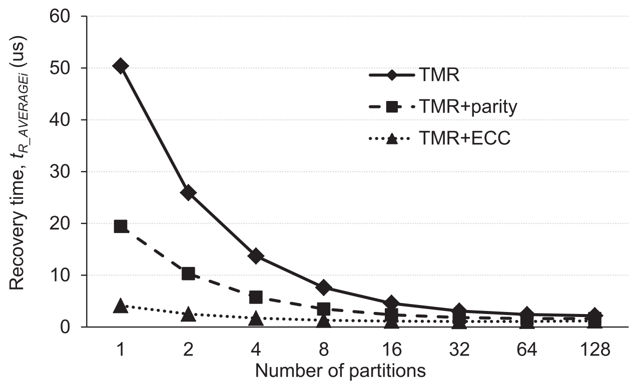


Fig. 6. Average recovery time considering the area and susceptibility to error of the various CGRA components.

component. For example, in the TMR case,

$$\alpha = (6.32 \times 0.75)/(6.32 + 1.44 + 15.43 + 1.00) = 0.20,$$
$$\beta = (1.44 \times 0.1 + 15.43 \times 1)/(6.32 + 1.44 + 15.43 + 1.00) = 0.67,$$
$$\gamma = (1.00 \times 1)/(6.32 + 1.44 + 15.43 + 1.00) = 0.04.$$

Even though the maximum recovery time of the control logic is high (92.5us at most, when device area is not considered), we can expect a rather smaller recovery time (50.4us at most) when the utilization and device area are included in the calculations.

The average recovery time is further reduced if we partition the application into several tasks or add error detection and correction logic to the memory. In this evaluation, the optimal result is achieved when ECC is used with TMR and the number of partitions, k, is equal to 32.

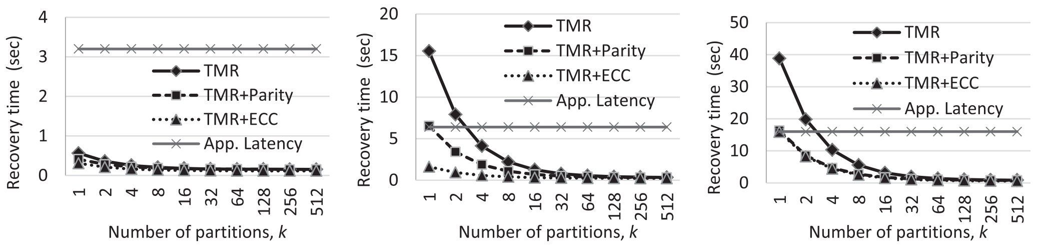
5.4 Recovery Time Estimation in Harsh Radiation Environments

In Geostationary Equatorial Orbit (GEO), the system is expected to experience a peak SEU rate of 3.29E-1 per second (Ostler et al. 2009). This result is applicable to a space-grade Xilinx Virtex-4 FPGA (XQR4VSX55), which has a similar gate count (1.3 million) (Xilinx 1997) with the proposed CGRA. Recent COTS FPGAs that are fabricated with modern sub-100nm process technologies are more sensitive to low-energy protons than space-grade FPGAs, which can increase the SEU rate by several orders of magnitude (Glein et al. 2015; Heidel et al. 2008). This fact suggests that the SEU rate of CGRAs that are not radiation hardened will be much higher than that given in Ostler et al. (2009). However, CGRAs share some similarities with both processors (subject to an order of magnitude lower SEU rates than FPGAs) and SRAMs (subject to an order of magnitude higher SEU rates than FPGAs) (Quinn and Graham 2005). Thus, it is difficult to estimate the SEU rate of CGRAs without performing radiation testing. We therefore chose the value stated previously and consider a range of SEU rates below and above this value.

When we increase the number of frames in the ACS operations, the chance of experiencing SEUs during the execution increases due to the increase in execution time. Table 8 shows the number of SEUs per application execution, when we assume a harsh radiation environment of 3.29E-1 SEUs per second. If we make the following two assumptions, (1) that the number in Table 8 is the maximum number of SEUs per execution and (2) that the system may experience the worst-case scenario, namely, that errors can occur in one partition at the same time, then the average

Table 8. Estimated SEUs During the Execution

Number of Frames	Application Latency (ms)	Number of SEUs per Execution
2.00E + 7	3.20E + 3	1
4.00E + 7	6.40E + 3	2
6.00E + 7	9.60E + 3	3
8.00E + 7	1.28E + 4	4
1.00E + 8	1.60E + 4	5



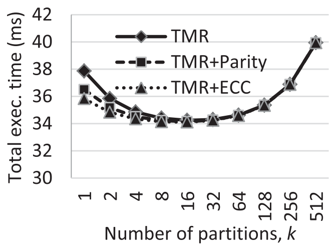
(a) SEUs per execution = 1 (b) SEUs per execution = 2 (c) SEUs per execution = 5

Fig. 7. Average recovery times in harsh radiation environments.

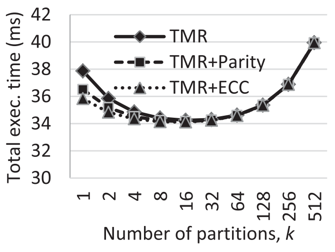
recovery times for one, two, and five SEUs are shown in Figure 7. When the number of SEUs per execution is one, as shown in Figure 7(a), recovery time is only incurred for control errors since all other errors are masked by TMR. Thus, the average recovery time is observed to be much smaller than the application execution time. If we increase the number of SEUs per execution to two, as shown in Figure 7(b), the recovery time becomes similar to the result of Figure 6. Finally, when we increase the number of SEUs per execution to five, as shown in Figure 7(c), we observe no difference between the TMR+Parity case and the TMR+ECC case since ECC is ineffective.

5.5 Total Execution Times for Some Practical Kernels

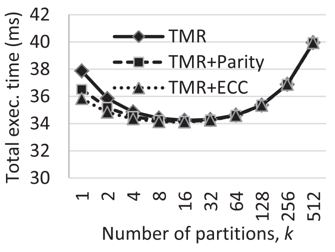
To confirm the effectiveness of the presented recovery time analysis model, we have evaluated various practical kernels, including vector convolution, matrix multiplication, and de-quantization, that are widely used in space applications (Lovelly et al. 2014). Figure 8 shows the total execution times, which include the application execution times of the CGRA and the expected recovery times assuming a radiation environment causing between 0.1 SEU/sec and 10 SEUs/sec, which we expect to be the range in maximum error rates CGRAs may experience in GEO. Note that the result in Figure 8 is not the complete execution time to generate an output of an application, but it is the expected application execution time including the recovery process. If errors occur during the recovery process or the re-execution process, the system may repeat this expected execution consecutively until at least two out of three voters agree with the majority result. It can be seen from Figure 8 that the experimental result is classified into three categories: (1) The number of SEUs per execution is less than two, where there is little difference between the protection schemes. This category includes (a), (b), (c), (d), (e), and (g) in Figure 8. (2) The number of SEUs per execution is between two and four, where there is a large difference between protection schemes, especially when the number of partitions, k, is small. In this category, a memory protection scheme such as parity helps to reduce the overall execution time, and the best overall execution time decreases along with an increase in the number of partitions, k. This category includes (f) and (h) in Figure 8. (3) The number of SEUs per execution is larger than four, where the graph



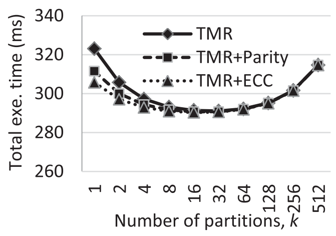
(a) Convolution at 0.1 SEU/sec (1 SEU/execution)



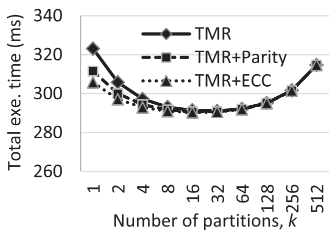
(b) Convolution at 1 SEU/sec (1 SEU/execution)



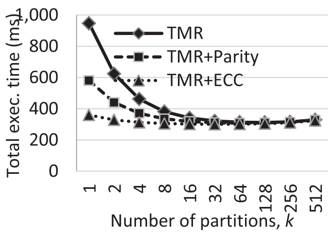
(c) Convolution at 10 SEUs/sec (1 SEU/execution)



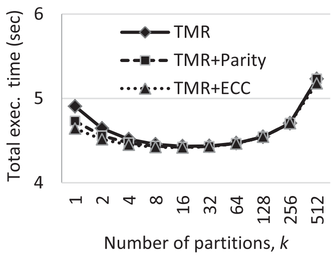
(d) Matrix mult. at 0.1 SEU/sec (1 SEU/execution)



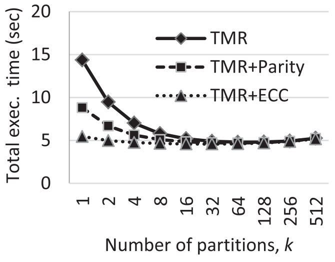
(e) Matrix mult. at 1 SEU/sec (1 SEU/execution)



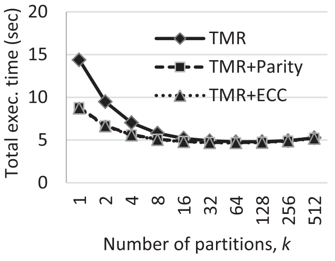
(f) Matrix mult. at 10 SEUs/sec (3 SEUs/execution)



(g) Dequantization at 0.1 SEU/sec (1 SEU/execution)



(h) Dequantization at 1 SEU/sec (4 SEUs/execution)



(i) Dequantization at 10 SEUs/sec (44 SEUs/execution)

Fig. 8. Total execution times for some practical kernels.

shows similar characteristics to the second category, but there is no difference between the TMR+Parity case and the TMR+ECC case. This category includes (i) in Figure 8. The results of practical kernel examples also support the observations we made in Figure 7.

6 CONCLUSIONS

We have presented a reliable CGRA and a recovery time model for transient faults in CGRAs. In the proposed architecture, triplicated modules and voters are implemented in software so as to avoid incurring a silicon area overhead. Hardware logic is included to enable error detection and correction for fault-tolerant memory.

In particular, we analyzed the maximum recovery time from errors, which are categorized as PE errors, memory errors, and control logic errors. We have also analyzed the average recovery time according to silicon area estimates. Finally, the overall execution time including the application execution time and the recovery time in harsh radiation environments was analyzed with some practical space kernel examples.

The results show that task partitioning is crucial for bounding the recovery time of applications that have long execution times. However, when an optimal number of partitions is reached, which yields the shortest average execution time, the benefits of using parity and ECC were negligible (under 1% on average) in terms of overall average execution time. The use of parity and ECC is only beneficial in circumstances where the number of partitions cannot be increased due to dependency issues, or for real-time applications where the worst-case execution time is important rather than the average execution time. In a very high-radiation environment, there was no advantage to using ECC rather than parity to protect configuration and data memory.

Future work envisages the implementation of a complete fault-tolerant CGRA architecture and an evaluation of better protection methods at the additional cost of area. We are also investigating the development of a software framework for CGRAs, which includes automated partitioning and roll-back mechanisms.

REFERENCES

Dawood Alnajjar, Hiroaki Konoura, Younghun Ko, Yukio Mitsuyama, Masanori Hashimoto, and Takao Onoye. 2013. Implementing flexible reliability in a coarse-grained reconfigurable architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 12 (2013), 2165–2178.

Lorena Anghel and Michael Nicolaidis. 2000. Cost reduction and evaluation of temporary faults detecting technique. In *Proceedings of the conference on Design, Automation and Test in Europe*. ACM, 591–598.

Muhammad Moazam Azeem, Stanislaw J. Piestrak, Olivier Sentieys, and Sebastien Pillement. 2011. Error recovery technique for coarse-grained reconfigurable architectures. In *2011 IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'11)*. IEEE, 441–446.

Ediz Cetin, Oliver Diessel, Lingkan Gong, and Victor Lai. 2013. Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration. In *2013 23rd International Conference on Field Programmable Logic and Applications (FPL'13)*. IEEE, 1–4.

Ediz Cetin, Oliver Diessel, Tuo Li, Jude A. Ambrose, Thomas Fisk, Sri Parameswaran, and Andrew G. Dempster. 2016. Overview and investigation of SEU detection and recovery approaches for FPGA-based heterogeneous systems. In *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 33–46.

Jason A. Cheatham, John M. Emmert, and Stan Baumgart. 2006. A survey of fault tolerant methodologies for FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 11, 2 (2006), 501–533.

G. Cieslewski, A. Jacobs, C. Conger, and A. George. 2008. Advanced space computing with system-level fault tolerance (invited talk). In *1st Workshop on Fault-Tolerant Spaceborne Computing-Employing New Technologies*.

L. J. Deutsch and R. L. Miller. 1982. The effects of Viterbi decoder node synchronization losses on the telemetry receiving system. *Jet Propulsion Lab., Pasadena, CA, TDA Progress Rep* 42 (1982), 68.

Robért Glein, Florian Rittner, Andreas Becher, Daniel Ziener, Jürgen Frickel, Jürgen Teich, and Albert Heuberger. 2015. Reliability of space-grade vs. COTS SRAM-based FPGA in N-modular redundancy. In *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS'15)*. IEEE, 1–8.

Richard W. Hamming. 1950. Error detecting and error correcting codes. *Bell Labs Technical Journal* 29, 2 (1950), 147–160.

Kyuseung Han, Ganghee Lee, and Kiyoung Choi. 2014. Software-level approaches for tolerating transient faults in a coarse-grained reconfigurable architecture. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (2014), 392–398.

Reiner Hartenstein. 2001. Coarse grain reconfigurable architectures. In *Proceedings of the Design Automation Conference, 2001 (ASP-DAC'01), Asia and South Pacific*. IEEE, 564–569.

David F. Heidel, Paul W. Marshall, Kenneth A. LaBel, James R. Schwank, Kenneth P. Rodbell, Mark C. Hakey, Melanie D. Berg, Paul E. Dodd, Mark R. Friendlich, Anthony D. Phan, Christina M. Seidleck, Marty R. Shaneyfelt, and Michael A. Xapsos. 2008. Low energy proton single-event-upset test results on 65 nm SOI SRAM. *IEEE Transactions on Nuclear Science* 55, 6 (2008), 3394–3400.

Yoonjin Kim and Rabi N. Mahapatra. 2011. *Design of Low-Power Coarse-Grained Reconfigurable Architectures*. CRC Press, Boca Raton.

Hiroaki Konoura, Dawood Alnajjar, Yukio Mitsuyama, Hajime Shimada, Kazutoshi Kobayashi, Hiroyuki Kanbara, Hiroyuki Ochi, Takashi Imagawa, Kazutoshi Wakabayashix, Masanori Hashimoto, Takao Onoye, and Hidetoshi Onoderas. 2014. Reliability-configurable mixed-grained reconfigurable array supporting c-based design and its irradiation testing. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 97, 12 (2014), 2518–2529.

Ganghee Lee, Kiyoung Choi, and Nikil D. Dutt. 2011. Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 5 (2011), 637–650.

Tyler M. Lovelly, Donavon Bryan, Kevin Cheng, Rachel Kreynin, Alan D. George, Ann Gordon-Ross, and Gabriel Mounce. 2014. A framework to analyze processor architectures for next-generation on-board space computing. In *2014 IEEE Aerospace Conference*. IEEE, 1–10.

Mathstar. 2007. Arrix Family FPOA architecture guide. Retrieved from http://www.mathstar.com.

Riaz Naseer and Jeff Draper. 2008. DEC ECC design to improve memory reliability in sub-100nm technologies. In *15th IEEE International Conference on Electronics, Circuits and Systems, 2008 (ICECS'08).* IEEE, 586–589.

Patrick S. Ostler, Michael P. Caffrey, Derrick S. Gibelyou, Paul S. Graham, Keith S. Morgan, Brian H. Pratt, Heather M. Quinn, and Michael J. Wirthlin. 2009. SRAM FPGA reliability analysis for harsh radiation environments. *IEEE Transactions on Nuclear Science* 56, 6 (2009), 3519–3526.

Heather Quinn and Paul Graham. 2005. Terrestrial-based radiation upsets: A cautionary tale. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2005 (FCCM'05).* IEEE, 193–202.

Zoltan E. Rakosi, Masayuki Hiromoto, Hiroyuki Ochi, and Yukihiro Nakamura. 2009. Hot-swapping architecture extension for mitigation of permanent functional unit faults. In *International Conference on Field Programmable Logic and Applications, 2009 (FPL'09).* IEEE, 578–581.

Jason Williams, Chris Massie, Alan D. George, Justin Richardson, Kunal Gosrani, and Herman Lam. 2010. Characterization of fixed and reconfigurable multi-core devices for application acceleration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3, 4 (2010), 19.

Xilinx. 1997. Gate count capacity metrics for FPGAs. Retrieved from https://www.xilinx.com/support/documentation/application_notes/xapp059.pdf.

Xilinx. 2014. Radiation-hardened, space-grade Virtex-5QV family overview. Retrieved from https://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf.

Jae-Sung Yoon, Choonseung Lee, Changsoo Park, Ganghee Lee, Kyungkoo Lee, Sungho Roh, Minsu Jeon, Youngbeom Jung, Jinhong Oh, and Jin-Aeon Lee. 2013. An H. 265/HEVC codec for UHD (3840× 2160) capturing and playback. In *2013 International SoC Design Conference (ISOCC'13).* IEEE, 218–220.

Received November 2016; revised June 2017; accepted September 2017