Parallel Streaming Computation on Error-Prone Processors

Yavuz Yetim Margaret Martonosi Sharad Malik
Princeton University

***Abstract*—Hardware fault rates are increasing due to decreasing transistor sizes in newer technology nodes. To ensure reliability, circuits need more redundancy and higher voltage margins, incurring high area/performance/energy overheads. Since some applications, such as multimedia processing and streaming applications, can tolerate some types of errors, approximate computing research has begun exploring scenarios in which the hardware that will run such applications may have errors visible to software. This research has thus far largely focused on managing data and control errors in single threaded processors. Parallel execution on error-prone hardware needs to deal with the added fragility of communication among the threads and is the focus of this work.**

We describe the design of CommGuard, a technique to enable error-tolerant communication for streaming applications. CommGuard uses the explicit communication information provided by languages such as StreamIt in order to provide coarse grain communication protection by detecting and handling data misalignments. With a small amount of reliable storage per processor core ($150B$) , even with reasonably high error rates (one per million instructions) acceptable output quality can be sustained throughout the computation for the JPEG Decoder application ($16dB$).

I. Introduction

Reliable computation on newer technology nodes is more difficult due to increasing process variation, reduced resilience to extreme operating temperatures and currents, and increased susceptibility to soft errors[15], [2], [7], [5], [3]. Redundancy techniques, such as error codes and redundant multi-threading, are increasingly used to lower the probability of errors so that software can continue to treat the hardware as error-free. However, with ever increasing error rates, the cost and area overhead of such techniques is becoming prohibitive [1], [12].

Approximate computing research aims to eliminate the strict assumption of hardware being error-free by utilizing application-level error-tolerance. For example, an application may tolerate data errors if the algorithm iteratively processes data and can mitigate intermediate errors (e.g., simulated annealing [8]), or if the output quality of the application can be measured and desired levels can be expressed (e.g., multimedia processing [4], [20]). However, while applications may be tolerant of *data errors*, they are much less likely to be tolerant of *control-flow* and *memory-addressing errors*; these usually quickly cause the application to crash or hang. Therefore, hardware errors affecting program control-flow or memory-addressing must either be explicitly disallowed [9], [14], [8], or managed by minimal protected hardware to avoid such catastrophic outcomes [20].

Moving from sequential to parallel execution, *communication errors* arise as a new challenge impeding error-tolerant execution on error-prone hardware. In this work, we propose and evaluate CommGuard, a parallel architecture with low-overhead protection applicable to error-tolerant parallel streaming applications. Streaming applications are easily parallelizable [17] but parallel execution on error-prone hardware proves to be challenging due to communication among the threads. Even though the form of communication for streaming applications is a well-structured producer-consumer relationship, we show that communication errors quickly corrupt intermediate structures. Furthermore, we show that even if the intermediate structures are protected, control flow errors in a producer thread can turn into irrecoverable control flow errors for consumer threads. Existing approximate computing mechanisms either disallow control-flow and memory-addressing errors or only work for single-threaded applications. CommGuard extends single-threaded protection mechanisms and provides error-tolerant parallel producer-consumer communication.

This work makes the following contributions:

- We build on the prior work for error-prone single-threaded computation and show that it fails to provide sustainable operation for applications running in parallel. We show that errors in communication among the parallel threads disrupt program flow, either causing crashes or hangs, or causing extreme degradation in the output quality. We further show that even if corruptions can be prevented in the communication medium itself, control-flow errors in a producer thread results in data misalignment for the consumer threads for the rest of the computation.
- We propose the CommGuard design as a response to communication errors in streaming programs. CommGuard uses coarse-grained enforcement of thread alignments to maintain acceptable program execution even in the face of high error rates.
- We show that good output quality can be maintained in practice using CommGuard. For JPEG Decoder running on 8 cores, CommGuard can sustain the output quality at $16dB$ even for errors as frequent as every million instructions. Our proposed implementation has very low hardware overhead of only around $150B$ of storage per processor core.

II. Background: Increasing Hardware Faults and Approximate Computing as a Response

Device scaling increases hardware fault rates for several reasons and ITRS has identified reliability as a major design challenge for newer technology nodes [6]. Some hardware faults are permanent and result in lower yield [7] whereas others, due to fluctuating voltage and temperature values and inherent randomness of cosmic rays, are transient and stochastic. This work focuses on transient faults. Transient faults may

or may not result in application-level errors depending on where and when they exhibit themselves [13]. For the rest of the paper, we use the terms *fault* to refer to a hardware-level bit-flip and and *error* to refer to an application-level software value change (in accordance with the existing literature [11]).

With increasing fault rates, conventional redundancy techniques to detect and correct random errors result in high overheads . In SRAM and other storage units, protection is achieved through error correction codes (ECC). Look-up latency incurred by a single-error correcting and double-error detecting code can be as low as a single cycle and the logic can be implemented with tens of thousands of gates per cache line. However, with increasing fault-rates, stronger codes are necessary and the look-up latency increases to tens of cycles and the implementation requires hundreds of thousands of gates [1]. Furthermore, protecting data storage is only part of the issue; one must also mitigate the effects of control flow and communication errors. In addition to ECC, redundant execution [12] has been proposed to protect against other hardware errors. Redundant execution can be temporal (e.g., executing the instruction twice on the same module) or spatial (e.g., executing the instruction in parallel on another core). Different choices trade-off hardware area, performance, and energy, but high overhead persists.

Approximate computing is an approach to reduce the protection overheads by exploiting that some applications do not require 100% error coverage. Full error coverage is not necessary if the application can mitigate errors in time (e.g., simulated annealing [8]) or the output quality can be measured and desired quality levels can be expressed (e.g., multimedia processing [4], [20]). The error-tolerant parts of these applications can be expressed through programming language constructs [9], [14] and different heuristics can improve an application's resiliency [18].

Approximate computing research has largely focused on data errors, but control flow and memory addressing errors remain and may cause the application to hang or crash. For example, a corrupted *write* access to memory may cause a segmentation fault, a corrupted *execute* access may irrecoverably corrupted the program counter, or a corrupted *branch condition* may cause the application to get stuck in an unresponsive state. Therefore, such errors have been either explicitly disallowed [14], [18] by prior work, or managed by a reliable thread [8].

Work in [20] proposed solutions for mitigating control-flow and memory addressing errors. These rely on coarse-grained guided execution of streaming applications, with suppression of illegal addressing or control flow operations. In this paper, we build on the minimal protection mechanisms in [20] particularly focusing on communication errors in parallel streaming applications.

A set of design requirements for acceptable error-prone programmable computation have been proposed in [20] and are applicable to this work. An error-prone execution needs to **progress** (i.e., not crash, hang, or corrupt devices) and have **acceptable output degradation**. The requirement of **acceptable output degradation** ensures that output quality of an application (e.g., SNR [16]) is above a desired level. To satisfy the design requirements, the hardware is extended with minimal reliable hardware modules to handle errors gracefully [20]. Lastly, it is desirable that the hardware error protection not have any output effect in the case of fault-free execution.

Our work on parallel error-tolerance has similar goals, and starts with similar protected modules as in [20]. A brief summary of the per-thread protection modules [20] and their connection to this work are explained in Section V. Our experiments show that he requirement for **acceptable output degradation** for long running applications using only per-thread protection modules. For a long running application, the error effects may accumulate and eventually result in permanently corrupted output. To support long-term operation, the effects of errors must be transient. The following definition helps sharpen this notion of transient effects of errors:

Definition 1. *A system is* suitably-error-tolerant *for a property P if it can recover the property in a bounded time R at any corrupted state if there is an interval R during which there are no errors that directly or indirectly affect P.*

The definition for suitable error-tolerance requires a finite recovery time for the properties. The recovery may assume that errors during this period do not affect the property (directly or indirectly). This arises because errors can corrupt any state any time, and hence there can never be a deterministic guarantee for full recovery. Therefore, the recovery process is allowed to assume error-free operation with respect to the property. The probabilistic nature of errors will occasionally allow such time intervals, during which the properties will be repaired.

The properties that are mentioned in Definition 1 can be general, such as the overall output quality, or more specific, such as consistency of head-tail pointers and size of a software queue implementation. We first determine the specific properties whose strengthening against errors results in sustained overall output quality in parallel computation. We then provide protection mechanisms so that the system is suitably-error-tolerant for these properties.

III. Permanent Communication Errors in Parallel Streaming Applications

Figure 1 shows the result of running the JPEG Decoder [19] application on 8 cores with varying levels of protection. Figure 1a shows the output when the cores do not experience any errors. When every thread is protected using the single core protection mechanisms in [20], the application fails to provide acceptable output quality due to communication errors as shown in Figure 1b. While protecting the communication subsystem itself from errors may seem sufficient, Figure 1c shows it is not. This experimental observation motivates the need to go beyond just the communication subsystem and led to the eventual design of CommGuard (Section IV) to avoid communication errors for parallel streaming applications.

A. Brief Overview of Streaming Applications

Figure 2 shows an example streaming graph for the `JPEG Decoder` application, after the code has been parallelized by the StreamIt compiler [17]. The graph is composed of 10 nodes (each running a separate thread) that handle the producer-consumer computation flow of each image pixel, with R, G, and B elements of a pixel being handled in parallel at one stage. The node-to-node communication is expressed explicitly

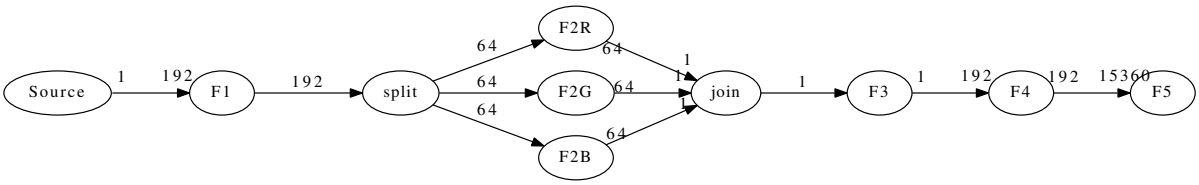
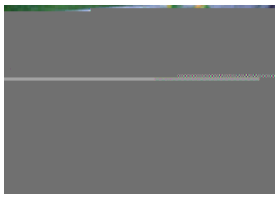
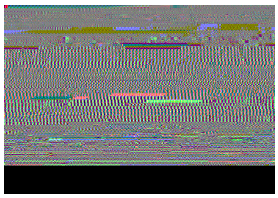


Fig. 2: JPEG Decoder streaming computation graph with explicit communication between processing nodes. The sequential execution of the graph expresses single-threaded coarse-grain control-flow.

(a) Error-free processors



(b) Error-prone processors with per-thread protection using [20]



(c) Error-prone processors with per-thread protection and error-free hardware queue

(d) CommGuard: Error-prone framed-computation with per-thread protection and error-free hardware queue

Fig. 1: Output of the JPEG decoder running on 8 threads with varying protection mechanisms. Error-prone systems in Figures 1b, 1c, and 1d experienced errors with a mean-time of 1M instructions

in StreamIt using *push* and *pop* directives, and the amount of communication (the number of data objects) per *node firing* is annotated on the arrows. Every node fires until its input is fully consumed.

In StreamIt, groups of communication are organized into *frames*. For producer nodes and their direct consumers, the frame size of an edge is the lowest common denominator of the indicated *push*es and *pop*s. As an example, `F3` *push*es 1 item per firing, and `F4` *pop*s 192 items per firing, so the *frame* size at this edge is 192 and `F3` needs to *fire* 192 times per *frame*.

B. How Errors Affect Communication

1) Data Transmission and Ordering: The choice of executing a streaming application sequentially or in parallel determines the complexity of the implementation of the producer-consumer communication. When the filters of a streaming application are executed sequentially, the amount of communication between a pair of nodes is known and is always the same for every iteration. Therefore, the communication can simply be through a stateless fixed-sized buffer. However, executing the nodes asynchronously and in parallel necessitates a data structure that retains state shared by the two ends of the communication. The retained state can be corrupted by errors and this corruption may quickly result in a permanent loss of communication.

The software-based parallel queue in StreamIt gets corrupted due to errors since it was not designed with error-resiliency in mind. The communication between a producer-consumer pair uses (i) local state belonging to the producer (ii) local state belonging to the consumer (iii) shared state accessed by both producer and consumer. These states are retained throughout the execution of the application and their consistency is key in communication. Under error-prone execution, any of these states may get corrupted, which in turn may halt the communication. When the communication between the filters stops, the filters process arbitrary data and the application overall ceases to produce useful output.

The best that can be done to resolve communication errors is to provide a reliable hardware queue between communicating streaming nodes. However, experimental observations indicate that an error-free queue is not enough to make parallel applications error-tolerant (Figure 1c). We now probe this deeper.

2) Alignment of Communication With Computation: An error-free queue cannot prevent misalignment due to control-flow errors in the producer and this results in a permanent output quality degradation. A minor control-flow error may cause a producer to *push* one extra item in the queue. The error-free queue would deliver these items to the receiving end of the edge. On the receiving end, the consumer node would not be aware that this item is extra and would *pop* the extra item as if it is the next item to be processed. This is an issue because items are applied to different operations with respect to their positions in the queue. Therefore, the operations applied to the items would shift by one for the rest of the computation. To provide a suitably-error-tolerant solution (see Definition 1), we can constrain the misalignment to *frames*. If we can make sure that the item produced at the start of a *firing* by the producer is consumed at the start of the *firing* by the consumer then the misalignment would be resolved. In Figure 2, consider the edge between the nodes `WEIGHTED_ROUND_ROBIN` and `F2R` . An extra *push* would result in 65 items in the queue for that specific *frame* instead of 64. If every time a new *frame* is started in `F2R` the first item of a frame can be searched in the queue then the 65th item would be dropped. Then, the remaining frames would not be affected by the error in this frame.

Another opportunity for permanent errors arises at the next-higher granularity: even in systems that ensure proper data alignment within a frame, a frame can be dropped as a whole. In a linear computation graph with no splits or merges, a streaming computation may be tolerant to dropping a whole

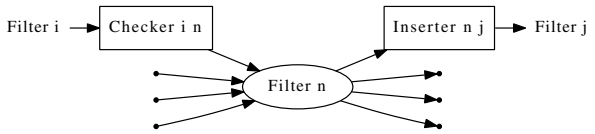


Fig. 3: Placement of CommGuard modules: At every edge, a *Frame Inserter* is placed at the producer end and a *Frame Checker* is placed at the consumer end

frame. On the other hand, where splits or joins occur, permanent misalignments may result from dropping a frame. For example, in Figure 2, if filter `F2R` drops a full frame but `F2G` does not, then the following `join` node will start merging different frames. Further, the mismatch in the merged frames will continue for the following frames. As before, a suitably-error-tolerant solution would avoid permanent misalignment by limiting the effects of dropping a frame to that frame only. This can happen by identifying the frames so that the consumer nodes can realign themselves upon the detection of a misalignment.

To summarize: a control-flow error in a producer thread can translate to a permanent control-flow corruption in a consumer thread. While [20] prevents control flow errors from resulting in permanent hangs or crashes, it would be cost-prohibitive to prevent them entirely. Thus, parallel error-tolerance approaches[1]must be able to tolerate such scenarios. The following section describes the design of CommGuard which provides a *suitably-error-tolerant* parallel communication.

IV. CommGuard: Communication Guards for Error-Tolerance

The principle behind CommGuard is to periodically refresh the alignment of communication (i.e., the items that are being transferred in the queue) with the computation (i.e., the control-flow of producers and consumers). To accomplish this, CommGuard draws inspiration from reliability solutions in data networking. In particular, CommGuard uses headers for chunks of data where the header includes the frame ID associated with this data. This frame ID can be used to keep data aligned for sequencing purposes.

To make use of frame IDs for sequencing, CommGuard includes two new protection modules; a *frame inserter* and a *frame checker*, as shown in Figure 3. The frame inserter monitors the execution of producer nodes and inserts frame headers in the queue. The frame checker monitors the items at the consumer end of a queue, monitors the execution of the consumer, detects erroneous conditions and repairs the misalignment. These two modules must be built from non-error-prone circuits. (Small amounts of reliable hardware are feasible by increasing transistor sizes or voltage margins for that portion of the design.)

[1]Note that, in contrast, the single-threaded execution of the application shown in Figure 2 sequences the *firing*s of the filters in a main loop that iterates for all *frames*. Due to the simple form of communication through a stateless fixed-sized buffer between communicating nodes, control-flow errors are strictly localized to every iteration. That is, even if there is a misalignment in the buffer shared by two nodes, the buffer will be completely overwritten for the next iteration, and the previous misalignment will be resolved.

TABLE I: Static and dynamic information per node

Value	Details	(S)tatic (D)ynamic
Firing per frame	How many times a node needs to fire before the computation starts for the next *common frame*	S
Frame limit	Number of total frames the application needs to process	S
Active frame	How many frames have been processed so far	D
Active firing	How many times the node has fired for the active frame	D

TABLE II: States of the frame checker

State	Details	(E)rroneous (N)ormal
Receiving items	Node is receiving items for the **active frame**	N
Expecting a header	Node has started new frame computationally hence the next item in the queue should be a header	N
Discarding	The computation in the node is ahead of the communication of the edge	E
Padding	The communication of the edge is ahead of the computation in the node	E
Discarding and padding	The computation and communication is not aligned but the progress is not comparable	E

To correctly attach and manage frame IDs, CommGuard modules need to monitor computation and communication and also need some static information from the program/compiler. Table I gives the static and dynamic states shared by all modules surrounding a node. The **active firing** is incremented every time the streaming node fires. This event is triggered by modifying the existing per-thread protection mechanisms [20] and the details are explained in Section V. When **active firing** reaches **firing per frame** , **active frame** is incremented and **active firing** is reset back to 0. Further details of the implementations of each module are provided below in Sections IV-A and IV-B.

A. Frame Inserter

Frame inserter monitors the **active firing** and inserts the frame header with the id of **active frame** to every outgoing edge each time **active firing** is reset. When **active frame** reaches **frame limit** a special id indicating the end of computation is inserted to every outgoing edge. Frame headers are prepended with a special bit pattern to distinguish them. (If a regular data item to be sent forms the same special bit pattern then it is encoded with an escape sequence.)

B. Frame Checker

The *Frame checker* monitors both the streaming computation of the node and the communication along an incoming edge. The events that trigger the state machine to check the misalignment conditions as explained below are: popping a header from the queue, popping a regular item from the queue, or the streaming node starting a new firing (i.e., **active frame** is reset to 0). Upon detection of a misalignment, it may *pad* items to respond the *pop* requests from the streaming node, *discard* items in the queue, or take both actions at the same time until the misalignment is resolved. The states in the finite state machine of the *frame checker* are given in Table II. Appendix B explains the transitions of finite state machine.

V. Integrating CommGuard With Guided Execution Management

In our envisioned error-tolerant parallel system, (i) a high-level programming language expresses coarse-grain control-flow, memory-addressing and communication requirements, (ii) every thread guarantees **progress** through single-threaded protection mechanisms and (iii) CommGuard manages coarse-grain communication between the threads. Below, we provide an overview for these separate parts and explain the connections between them.

We chose StreamIt [17] as our high-level language because many applications that can benefit from approximate computing can be expressed as a streaming computation. Data corruption in streaming applications is transient since new data is periodically streamed through the computation graph. A streaming application is usually a big loop of `while(data) {input-data; process-data; output-data}`. When errors occur during `process-data`, the effects of errors are very likely to disappear or decrease substantially for the following iteration.

Single-threaded protection can be achieved through low-overhead reliable modules as explained in [20]. Three modules that provide the protection are the MIS (macro instruction sequencer), the MFU (memory fence unit) and the streaming I/O. The application is divided into coarse-grain (potentially nested) control-flow regions termed *scope*s. The function of the MIS is (i) to ensure that the application follows the prescribed control flow between the scopes and (ii) that it does not get stuck within some scope. It does this by maintaining a table with the following static information for every scope of a program; start and exit PCs (for scope start and termination), instruction count limit (for bounding the time spent in a scope), parent scope (for detecting invalid jumps into the scope), and call frame (for function calls). The MIS monitors execution and instruction commits, and uses the static information in the table to guarantee that no scope exceeds it predetermined bound in run-time in terms of its total instruction count, and ensures correct scope sequencing to always enable the application to progress. The MFU uses conventional segmentation methods to detect memory access errors and gracefully drops invalid requests as long as application can proceed. On memory errors that cannot be recovered (such a fetch request with a corrupted program counter), the MFU informs the MIS to terminate the current scope. Streaming I/O is a limited form of input/output that eliminates risks of file system and device corruption.

The shared state explained in Section IV is managed by monitoring the computation inside a streaming node. More specifically, **active firing** is incremented every time a new *firing* starts. In order to signal a new firing, the MIS informs CommGuard every time the scope around the body of the main iteration of a streaming filter is activated. Similarly, to determine when the computation in a filter ends, the MIS signals CommGuard when the global scope of a filter exits.

VI. Experimental Methodology

Our simulation infrastructure is built on the detailed Virtutech Simics functional architecture simulator [10]. Our baseline system is a 32-bit Intel x86 architecture extended with error injection capabilities and single-threaded protection modules [20]. We implemented four new modules, the *hardware queue*, the *frame inserter*, the *frame checker* as explained in Section IV and a *context manager* to manage operational context for parallel threads. We further modified the error injector for parallel capabilities. To model the access to the hardware queue (i.e., *push* and *pop*), we used certain x86 instructions as markers and existing x86 registers for data hand-off. We modified the StreamIt compiler to use these new instructions instead of the library calls to the existing software implementation of the queue.

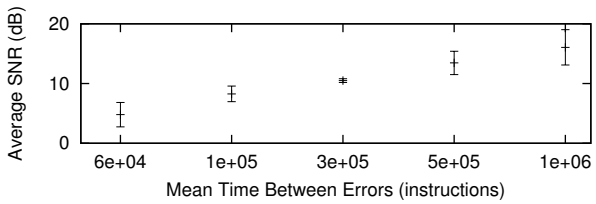


Fig. 4: Output quality of the JPEG Decoder at varying MTBEs (error-free SNR is 28dB).

Our experiments use a widely used multimedia application, the JPEG Decoder. We compressed a raw image to the JPEG format on an error-free hardware, and our error-prone runs decode this image back to the raw format. This allows us to compare the losses from the compression algorithm with the losses due to errors. We experimented with varying mean-times-between-error (MTBEs) to capture the trends in output quality changes, run-time and error handling statistics. For every MTBE, we ran the application 5 times and observed the standard deviation across different runs. Using a functional simulator enables fast simulation and allows us to compare output qualities of full runs of the application. Even though a precise run-time cannot be measured with a functional simulator, we use the total number of instructions executed as a proxy for run-time.

VII. Hardware Overheads and Design Improvements

A. Hardware Overheads

Our protection mechanisms require extra hardware components and these components are required to be reliable. *Frame inserter* and *frame checker* cost only approximately $256bits$ per processor core. Appendix A1 evaluates their implementation in more detail. A naive implementation of the *error-free hardware queue* may need around $2.5Mbits$ of storage per edge in a given streaming graph. However, thanks to the *frame inserter* and the *frame checker*, the error-free queue can be simplified and only $1kbits$ of storage is necessary per processor core. Appendix A2 provides a summary of the implementation and overhead calculation. In addition to the storage, the logic for the queue and the frame modules need to be implemented reliably, as well. The detailed evaluations for these structures are future work and not discussed here.

VIII. Experimental Results

A. Quality Degradation

Figure 4 shows SNR values at varying MTBEs and the output quality is maintained at acceptable levels (13dB) for

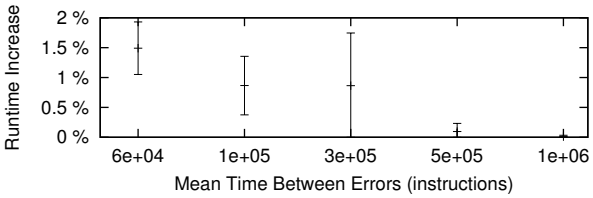


Fig. 5: Ratio of run-time on an error-prone processor with CommGuard to run-time on an error-free processor.

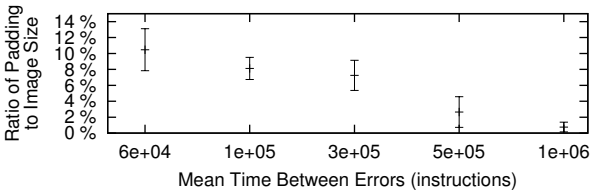


Fig. 6: Amount of padding done by the frame checker to realign the computation with the communication.

MTBEs as low as every 512k instructions. The output quality measurements are similar but slightly lower (e.g., 1dB lower at MTBE of 512k instructions) than single-threaded counterparts [20]. With lower MTBEs the quality decreases and eventually goes as low as 5dB for an MTBE of 64k instructions.

B. Effects of Errors on Data Loss and Run-time

Figure 5 compares the total run-time at varying MTBEs and the graph indicates a small but a consistent increase, up to 2%. The single-threaded execution experiments did not result in increased execution times, and in fact resulted in decreased run-time [20]. The reason for this slight increase is due to the realignment strategies for error-prone parallel execution. Realignment through the discarding state method explicitly blocks the consumer and the padding state indirectly may block the producer. On the other hand, the realignment for the single-threaded case is implicit where locations in the fixed sized buffer are simply skipped. However, the 2% run-time increase is observed when MTBEs are as low as every 64k instructions, i.e., where the output quality has already decreased to unacceptable levels.

Figure 6 shows the amount of padding as issued by the frame checker compared to the total size of the image. The ratio goes up to 13% for low values of MTBE. Even for acceptable values of MTBEs, such as 512k instructions, the ratio can be as high as 4%. Note that 4% padding more than justifies the protection overhead for misalignment, since even a single word misalignment causes permanent quality degradation at the application output.

IX. Conclusions

In this work, we investigate the challenges of parallel execution on error-prone hardware introduced due to communication between threads, and provide CommGuard as a low-overhead communication protection mechanism. Communicating threads of a parallel streaming application may fail due to the intermediate data structures getting corrupted or control-flow errors may cause permanent misalignment of the data with the computation. In order to realign the data with the computation; CommGuard modules encapsulate frames of data using headers, monitor the computation of a consumer node and the items received from the incoming edges, and detect and repair every misalignment. Our results show that CommGuard can sustain correct operation and provide output quality similar to existing single-threaded methodologies, 13dB for an MTBE of 512k instructions for a streaming JPEG decoder application running on 8 threads. These results show that while error-tolerant execution on almost-fully error-prone hardware is challenging, high-level application level information can be used to provide coarse-grain control mechanisms and guide applications through errors.

References

[1] A. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu. Energy-efficient cache design using variable-strength error-correcting codes. In *ISCA*, 2011.

[2] M. Clemens, B. Sierawski, K. Warren, M. Mendenhall, N. Dodds, R. Weller, R. Reed, P. Dodd, M. Shaneyfelt, J. Schwank, S. Wender, and R. Baumann. The effects of neutron energy and high-z materials on single event upsets and multiple cell upsets. *IEEE Transactions on Nuclear Science*, 2011.

[3] G. Gielen, P. De Wit, E. Maricau, J. Loeckx, J. Martin-Martinez, B. Kaczer, G. Groeseneken, R. Rodriguez, and M. Nafria. Emerging yield and reliability challenges in nanometer cmos technologies. In *DATE*, 2008.

[4] R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *ISLPED*, 1999.

[5] W. Huang, M. Stan, S. Gurumurthi, R. Ribando, and K. Skadron. Interaction of scaling trends in processor architecture and cooling. In *Semiconductor Thermal Measurement and Management Sym.*, 2010.

[6] ITRS. ITRS process integration, devices, and structures, 2011.

[7] K. Kuhn, M. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. Ma, A. Maheshwari, and S. Mudanai. Process technology variation. *IEEE Transactions on Electron Devices*, 2011.

[8] L. Leem et al. Error resilient system architecture (ERSA) for probabilistic applications. In *DATE*, 2010.

[9] S. Liu et al. Flikker: saving dram refresh-power through critical data partitioning. In *ASPLOS*, 2011.

[10] P. S. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35(2), 2002.

[11] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[12] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *ISCA*, 2002.

[13] S. S. Mukherjee et al. Measuring architectural vulnerability factors. IEEE Computer Society, 2003.

[14] A. Sampson et al. EnerJ: approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[15] B. Sierawski, R. Reed, M. Mendenhall, R. Weller, R. Schrimpf, S.-J. Wen, R. Wong, N. Tam, and R. Baumann. Effects of scaling on muon-induced soft errors. In *International Reliability Physics Symposium*, 2011.

[16] T. Stathaki. *Image Fusion: Algorithms and Applications*. Academic Press, 2008.

[17] W. Thies et al. StreamIt: A language for streaming applications. In *ICCC*, 2002.

[18] A. Thomas and K. Pattabiraman. Error detector placement for soft computation. In *Dependable Systems and Networks*, pages 1–12, 2013.

[19] G. K. Wallace. The JPEG still picture compression standard. *Commun. ACM*, 34(4), 1991.

[20] Y. Yetim, M. Martonosi, and S. Malik. Extracting useful computation from error-prone processors for streaming applications. In *DATE*, 2013.

APPENDIX

A. Design Overheads and Improvements

1) Frame Modules: Entries in Table I are stored for every streaming node and shared by the frame modules. Therefore, 4 words of data need to be stored for every node in the streaming graph. When StreamIt compiles a streaming graph for parallel execution, the number of streaming nodes are about the same as the number of available processor cores in the system. Therefore, for a system with C cores, total storage would amount to 4 C words. For the given system with 8 cores and a 32-bit word size, the total storage would be 1024bits. Note that the storage would actually be distributed to every core and the per-core storage overhead would only be 128 bits.

The *frame inserter* module does not need additional storage and the *frame checker* module needs 3 bits + 1 word per incoming edge. Because there are only 5 states in Table II and these can be represented using only 3 bits. An extra word also needs to be stored because the *frame checker* occasionally takes a shadow copy of the variable active frame (see Section IV-B). The example benchmark JPEG Decoder has at most 3 incoming edges for a given node. Therefore, this application would only require at most 105 bits per processor core to implement the *frame checker* .

2) Error-Free Hardware Queue: The default size of the software queue implementation in the StreamIt compiler is chosen as 80000 items per edge. Assuming a 32-bit size for every item, the total data size that can be stored on an edge between the nodes is approximately 2.5 Mbits. Therefore, a naive implementation for the hardware queue would be the bottleneck in terms of the protection overhead.

This overhead can be decreased substantially thanks to the frame modules. Because the misalignment issue is resolved through headers, the reliable communication requirements can be limited to transferring the tuples [header id, frame pointer] for every frame and storing the actual data in error-prone memory as pointed by the triplet. The reliable hardware queue would still manage the *push* and *pop* operations and handle requests from the frame modules but would not need to store the data reliably. For our application JPEG Decoder, the frame size is 15360 items, therefore there would only be a maximum of 6 tuples per edge. This would bring down the amount of storage per edge to 12 words. For a JPEG Decoder with maximum of 3 edges per node, this would be only 1152 bits per processor core.

B. Transitions for the Frame Checker Finite State Machine

The verbs in *italics* are the states from Table II.

As long as the computation and the communication are aligned, the *frame checker* is either *receiving items* or *expecting a header* (initially *expecting a header*). When the *frame checker* is *expecting a header* and the correct header is received at the incoming edge, the *frame checker* starts *receiving items* for the computation. When the streaming node starts a new firing the *frame checker* again starts *expecting a header* .

The *frame checker* starts *discarding and padding* when the node was *receiving items* but, instead of a regular item, the header of the current frame or and older frame is encountered. In this case, the items from the queue are discarded and the pop requests from the streaming node are answered with an arbitrary value, 0, since even if the correct frame is found the position in frame is not tracked. The realignment will happen at the point when the streaming node starts a new firing and the corresponding header is found in the incoming edge. Therefore, if a header that is newer than the **active frame** is found after discarding enough items then the *frame checker* only keeps *padding* items. Similarly, if the streaming node starts a new firing the *frame checker* only keeps *discarding* items from the queue since the search for the correct header is still in progress.

In addition to the transition from the *discarding and padding* state, the *frame checker* starts *discarding* items when there are extra items in the queue that are either old or of unexpected type. More specifically, the *frame checker* starts *discarding* when it was expecting header and either of these two conditions hold; a past header is received or a regular item instead of a header is received. The *frame checker* keeps discarding items from the incoming edge until the correct header is encountered and then starts *receiving items* . If another past header is found it keeps *discarding* . On the other hand, if a future header is found then it starts *padding* items. Note that, the *discarding* state is the only state that blocks the execution of the streaming node.

In addition to the transition from the *discarding and padding* state, the *frame checker* starts *padding* when a header is popped from the incoming edge and the header id indicates that this frame should not have been received yet. More specifically, all states can transition to *padding* when a header is received and the header id is larger than the **active frame** . Right after transitioning to this state, first, the header id is recorded as a shadow **active frame** . Then, until the streaming node catches up with the communication (i.e., **active frame** == **shadow active frame**) the streaming node is padded with an arbitrary item value, 0. When the alignment is established, the *frame checker* starts *receiving items* .