CommGuard: Mitigating Communication Errors in Error-Prone Parallel Execution

Yavuz Yetim
Princeton University
yyetim@gmail.com

Sharad Malik
Princeton University
sharad@princeton.edu

Margaret Martonosi
Princeton University
mrm@princeton.edu

Abstract

As semiconductor technology scales towards ever-smaller transistor sizes, hardware fault rates are increasing. Since important application classes (e.g., multimedia, streaming workloads) are data-error-tolerant, recent research has proposed techniques that seek to save energy or improve yield by exploiting error tolerance at the architecture/microarchitecture level. Even seemingly error-tolerant applications, however, will crash or hang due to control-flow/memory addressing errors. In parallel computation, errors involving inter-thread communication can have equally catastrophic effects.

Our work explores techniques that mitigate the impact of potentially catastrophic errors in parallel computation, while still garnering power, cost, or yield benefits from data error tolerance. Our proposed CommGuard solution uses FSM-based checkers to pad and discard data in order to maintain semantic alignment between program control flow and the data communicated between processors. CommGuard techniques are low overhead and they exploit application information already provided by some parallel programming languages (e.g. StreamIt). By converting potentially catastrophic communication errors into potentially tolerable data errors, CommGuard allows important streaming applications like JPEG and MP3 decoding to execute without crashing and to sustain good output quality, even for errors as frequent as every $500\mu s$.

Categories and Subject Descriptors Computer Systems Organization [*Performance of Systems*]: Fault tolerance

Keywords Application-level error tolerance, high-level programming languages, parallel computing

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694354

1. Introduction

Achieving reliable, cost-effective computation with acceptable semiconductor yields becomes more challenging with each new technology generation. Resilience challenges arise due to extreme operating temperatures and currents, and due to increased susceptibility to soft errors [6, 12, 14, 17, 26]. Originally handled primarily at the circuit level or below, the challenges of semiconductor reliability have become pressing enough that architecture-level solutions must also be explored. Our goal is to find low-overhead methods to exploit application-level error tolerance in order to improve the cost or yield.

Not all errors are equally worrisome. In particular, while many applications may be tolerant of modest *data* error rates, other errors—in control flow, memory addressing, or parallel communication—are often catastrophic. Thus, in most prior work, programs are separated at design-time, compile-time, or run-time into the operations (calculations, loop control, etc.) that must be run on fully-reliable hardware versus those that can be run on potentially error-prone hardware. In some cases, the fully-reliable hardware represents the bulk of the machine (all instruction decode, sequencing, and control) while the error-prone hardware is limited to calculation units, a small fraction of processor die area [25]. In other prior work, a single fully-reliable processor manages loop-level control sequencing and timeouts for a set of potentially unreliable processors [18]. This allows more error-prone area (and thus increases potential benefits), but limits the types of applicable programs, e.g., do-all parallelism with idempotent loop bodies [18].

Our work proposes and evaluates techniques that support the data-error-tolerant execution of a fairly broad set of workloads, and that greatly decrease the amount of fully-reliable hardware required. In particular, we focus on parallel systems in which each component core is itself only partially reliable, and we explore methods to mitigate the catastrophic effects of error-prone interprocessor communication between such cores. *Our work enables low-cost error-tolerant execution by converting potentially catastrophic control flow or communication errors into potentially tolerable data errors.*

Our approach centers on CommGuard, a simple FSM-based technique for dynamically monitoring that the program abides by expectations about how many data communications will occur between different parallel entities. For example, in a producer-consumer interaction, resiliency hinges on the consumer receiving the correct number of data items; incorrect data counts lead to deadlock, livelock, permanent misalignments, or other fatal issues. CommGuard pads or discards communicated items when too many or too few have arrived. Discarded items are chosen arbitrarily and padding items fills data frames with arbitrary values to use. Such techniques succeed because data errors are generally more tolerable than the control-oriented errors that stem from incorrect item counts.

This paper shows how a CommGuard checker can be very effective, can be built at very low-overhead, and can exploit program-level information already frequently provided by languages like StreamIt [29] or Concurrent Collections [3]. CommGuard allows important streaming applications (e.g., jpeg, mp3 decode) to execute without crashing and to sustain good output quality, even for errors as frequent as every $500\mu s$. Furthermore, it requires low overhead, almost all of which is off the critical path of instruction execution.

2. Motivation

2.1 Tolerating Hardware Errors

Device scaling increases hardware fault rates for several reasons and ITRS has identified reliability as a major design challenge for newer technology nodes [16]. Hardware faults (e.g., transistor aging, process variation, energized particle strikes, and others) come in three types: permanent, intermittent or transient [12, 17, 26]. Intermittent and transient faults are particularly expected to increase for newer technologies [7]; this work focuses on them. Faults may or may not result in application-level errors depending on where and when they occur [23]. For this paper, in accordance with prior literature [24], we use the term *error* to refer to an architecture-level value change caused by a hardware-level *fault* .

With increasing fault rates, conventional redundancy techniques to detect and correct errors result in high overheads. In SRAM and other storage units, protection is achieved through error correction codes (ECC) but increasing fault rates require stronger codes with higher latencies and storage overheads [1]. Further, protecting data storage is only part of the issue; one must also address errors in control-flow, data-flow, computation and communication. Various forms of redundant execution [22] protect against these errors. Redundant execution can be temporal (e.g., executing instructions twice on the same core) or spatial (e.g., executing instructions in parallel on another core). Different choices trade-off hardware area, performance, and energy in different ways, but none effectively avoid the high overhead of redundancy.

Certain application classes, e.g. multimedia, have some degree of error tolerance. They are *data* error tolerant in that their output can be acceptable even without 100% error correction. They are still susceptible, however, to control flow and memory addressing errors, which can be catastrophic. Fully-reliable processors can avoid these errors completely [18, 19, 25] but will incur high performance and energy overheads and may be cost-prohibitive for future technologies.

Some control-flow and memory addressing errors may be acceptable as long as their effects on the execution are managed. For example, architectures proposed in [18, 32] handle control-flow and memory-addressing errors when they may cause crashes or hangs. The reliability analysis proposed in [4] allows cores that may have bounded control-flow and memory addressing errors in its execution model. Here, we refer to such processors as *partially protected uniprocessors (PPU)* and show that existing PPU microarchitectural techniques are either insufficient or incur high overheads for parallel computation. This paper addresses this with a low-overhead microarchitectural solution to manage communication errors in parallel systems built on PPU cores.

2.1.1 Requirements

Our design will not eliminate errors entirely, but rather seeks to limit their impact. In particular, our operational requirements are as follows. First, an error-tolerant execution needs to **progress** (i.e., not crash, hang, or corrupt I/O devices). Second, achieving long-term execution with acceptable output quality requires that the effects of errors must be **ephemeral**. That is, if errors occur, their effect on execution should diminish with time. The third design requirement is that the technique be **low-overhead**.

2.2 Coarse-Grain Error Management and High-Level Application Constructs

CommGuard exploits application-level constructs to identify coarse-grained linkages between control flow and communication operations. With appropriate knowledge of control-flow and communication relationships, small reliable hardware modules can detect and mitigate the error effects seen as violations of the operational requirements.

As a concrete example, this paper uses streaming applications. A streaming application consists of coarse-grain compute operations each of which inputs and outputs data streams. CommGuard uses static information, such as explicit producer/consumer connections, and monitors the communication and control-flow construct invocations at run-time. This allows CommGuard to identify groups of items in the data streams and establish a run-time link between control-flow and inter-core data-flow.

Our implementation examples are based on StreamIt [29], a programming language and a compiler for streaming applications. StreamIt offers constructs to support do-all, data, and pipeline (producer-consumer) parallelism, as well as streaming communication between threads. Figure

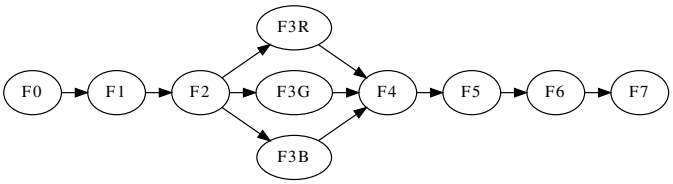


Figure 1: `jpeg` streaming computation graph with explicit communication between processing nodes.

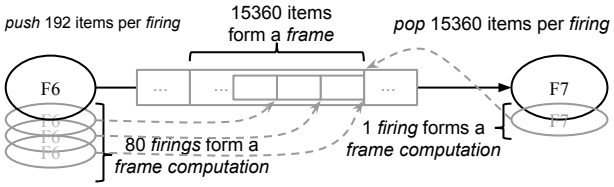
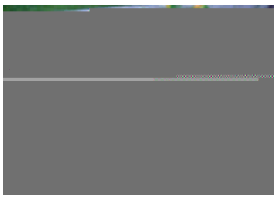
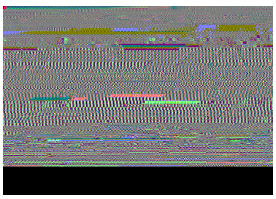


Figure 2: Example `jpeg` subgraph showing relations of items, firings, and frames

(a) Error-free processor cores



(b) Error-prone PPU cores



(c) Error-prone PPU cores with reliable data transmission

(d) CommGuard: Error-prone PPU cores with framed computation.

Figure 3: Output of `jpeg` running on 10 threads with varying protection mechanisms. Systems in Figures 3b-d had a mean time between errors of 1M instructions.

1 shows an example streaming graph for the `jpeg` application. The graph is composed of 10 parallel nodes connected with producer-consumer edges. The graph represents the producer-consumer pipelined execution flow of each image pixel, with the R, G, and B elements of a pixel being handled with data parallelism at one stage. For the rest of the paper, *node* refers to a chunk of computation and *edge* refers to communication streaming between two nodes. To execute this application, a node runs as a separate *thread* pinned to a processor and a *queue* implements the communication along an edge. StreamIt uses a concurrent queue data structure to implement a queue manager with head/tail pointers pointing to queue buffers.

As Figure 2 shows, StreamIt expresses communication through *push* and *pop* constructs. On each node *firing*, the node consumes items from its incoming edges and produces items to its outgoing edges. The amount of algorithm-specific communication determines the per-firing item counts. For example, the figure shows that `F6` produces 192 items, and this refer to an 8-pixel by 8-pixel image region, where each pixel consists of 3 items, the red, green and blue values.

From StreamIt constructs, one can link the control-flow and data-flow of communicating threads by matching node production and consumption rates. In Figure 2, a group of 192 items forms exactly one firing worth of output for `F6` but forms only a part of the input for `F7`. However, 15360 items correspond to exact multiples of firings in both filters. For this edge, 1 firing of `F7` depends on the output of the 80 most recent firings of `F6`. This analysis relates groups of producer firings to groups of items and transitively to groups of consumer firings, and thus links producer and consumer control-flows to the communication. In the following sections, we will use the terms *frames* to refer to the groups of items and *frame computations* to refer to the groups of firings that are linked through this analysis.

2.3 Example Scenario

As a motivational example, Figure 3 shows the result of running a 10-core execution of `jpeg` [31] with different error rates and error protection mechanisms. Figure 3a shows the output when the cores do not experience any errors. Next, Figure 3b shows the output when random bit errors in architectural registers occur on average every 1M instructions. Even with PPU cores, the effects of errors corrupt the queue data structure that manages the item transmissions between the nodes. As a result, execution proceeds less than 25 million instructions before the queue starts to transmit most of items incorrectly. With frequent communication being the norm (a communication event occurs as often as every 7 compute instructions on average in our benchmarks) it is clear that errors can quickly corrupt inter-thread communication.

A first recourse might be to error-protect the communication subsystem itself, but Figure 3c—obtained with fully-reliable queues interconnecting the PPU cores—shows that this too is insufficient. The reliable communication substrate is not enough to overcome alignment problems if the producer cores produce more or fewer data items than what downstream consumer cores expect. Much like Figure 3b, this scenario also quickly experiences significant error degradation; the running program loses or garbles many output pixels. *In addition to protecting the data being communicated, one must also be able to check and realign the number and type of data communications being sent and received.*

CommGuard provides low-overhead, coarse-grained communication checkers that detect when errors must be mitigated in order to avoid permanent corruptions for parallel streaming applications. Figure 3d shows the result of the same 10-core `jpeg` execution at the same error rate with the same PPU cores, but now with this paper's new CommGuard state machines to check and realign communications between cores. While errors still have mild data effects, CommGuard allows the parallel execution to complete with acceptable image quality.

3. How Errors Affect Communication

Data Transmission Errors (DTE): Bitflips can cause data to have the incorrect value when it is being stored or transmitted (e.g., in non-ECC memory or unreliable interconnect). If these errors only cause value changes in the items being transmitted but still ensure the transmission of the correct number of items then their effects are usually ephemeral as we focus on applications with data error tolerance running on PPU cores.

Queue Management Errors (QME): In some error scenarios, errors may corrupt the inter-thread communication management state, such as the head and tail pointers of a StreamIt queue. These corruptions degrade output quality significantly or cause catastrophic failure. In some cases, such errors cause deadlock, e.g, when cores have inconsistent views about the full/empty status of the queue. In other cases, corrupted head/tail pointers point to incorrect addresses and the values returned from the queue become incorrect for the rest of the computation. Therefore, either (i) these pointers and their operations must be reliable, or (ii) pointer corruptions need to be periodically repaired.

To avoid permanent errors due to data transmission, a reliable hardware queue may communicate items between producer-consumer pairs through error correction mechanisms. However, as shown in Figure 3c, an error-free queue is not enough. This is due to alignment errors discussed below.

Alignment Errors ($AE_{(I|F)(E|I)}$): Alignment errors occur when the number of communicated data items produced or consumed depends on program control flow that might itself be error-prone. For example, PPU cores allow small control-flow perturbations that change iteration counts, and that may partially construct a collection of items, possibly missing some. Operations on such a collection will execute incorrectly if they are written to expect properly-constructed collections with a particular item count.

We denote alignment errors with respect to the consumer using the notation $A_{(I|F)(E|L)}$ where the subscripts denote (i) misalignment granularity, i.e. if (I)tems (i.e., the smallest transmission units) in a frame or (F)rames are misaligned, and (ii) misalignment amount, i.e. if misalignment caused (E)xtra data or (L)ost data. Note that, we define errors in their effect, not by their causes. Misalignments can happen due to control-flow errors in the producer or consumer thread or due to queue management errors. A *realignment* (of a misalignment due to producer, consumer or the queue between them) operation ensures that every new frame, which is constructed by the frame computation of the producer, aligns with the start of a frame computation of the consumer.

For example misalignments, consider Figures 1 and 2. In Figure 2, an internal control-flow error in `F6` may cause it to push one extra pixel. `F7` would not be aware of the extra item and would *pop* the extra item as if it is the next item to be processed (A_{IE}). Then, `F7` compute operations would apply to incorrect items (e.g., a scalar vector multiplication of the incoming vectors, i.e. frames, with a constant vector). Frame-level misalignments may be benign for `F6` and `F7` but would still be catastrophic for split-joins. If `F3R` drops a full frame but `F3G` does not, then the following `F4` node will start merging different frames (A_{FI}). Both of these misalignment scenarios would shift items for the rest of the computation; without repair, they will never realign.

To summarize: a control-flow error in a producer thread can translate to a permanent control-flow corruption in a consumer thread. It is cost-prohibitive to prevent control-flow errors entirely and PPU cores prevent control flow errors from resulting in permanent hangs or crashes. As a result, multiprocessors built using such processors must be prepared for the control-flow error effects on interprocessor communication. To protect against such alignment errors, CommGuard draws inspiration from reliability solutions in data networking and uses headers and frame IDs to identify frames. Using these constructs, CommGuard handles alignment errors by discarding or padding items. This has the attribute of converting potentially catastrophic errors into potentially tolerable data errors, with much lower overhead than checkpointing or recomputation.

4. CommGuard: Communication Guards for Error-Tolerant Multiprocessor Execution

Figure 4 gives an overview of CommGuard's organization. To manage inter-processor communication effects, CommGuard includes three new modules per processor core; a *header inserter (HI)*, an *alignment manager (AM)*, and a *queue manager (QM)*. Figure 5 gives a more detailed logical view of how producer and consumer nodes interact with CommGuard modules. CommGuard interfaces with the normal program execution through *push* and *pop* operations. The *push* operation supplies the modules with a queue identifier (QID) and the data item, and the *pop* operation supplies only a queue identifier and receives data. (These can be implemented either in hardware as special instructions or in software as additions to the push/pop implementations.) The modules access the Queue Information Table (QIT) entries using QIDs.

The AM monitors the items being popped from the communication queue by the consumer thread. It uses the frame

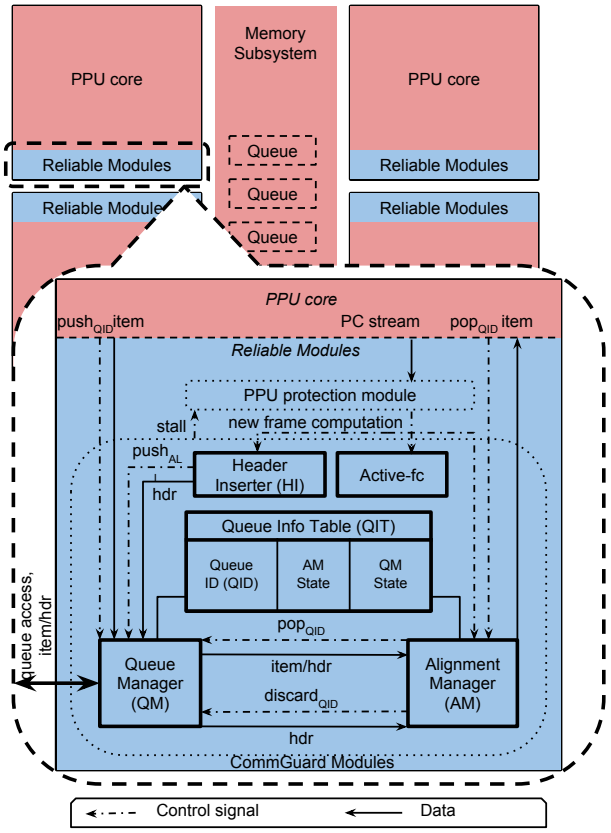


Figure 4: CommGuard modules attach to processor cores and the PPU protection module.

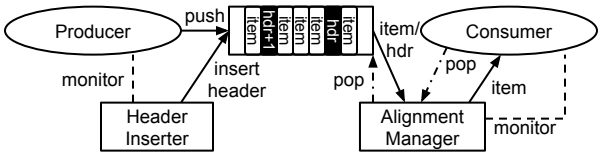


Figure 5: Logical placement of CommGuard modules between a producer and a consumer.

headers inserted by the HI to check and maintain alignment. The QM handles the data transmission to other threads. The PPU core increments the *active-fc* counter (i.e., active frame computation) for every new frame computation and this counter represents the frame progress of the thread. These modules must be built from error-free circuits. (As in prior work [13, 18, 19, 25], reliable hardware is considered feasible by increasing transistor sizes or voltage margins for that portion of the design.)

4.1 Header Inserter

On the producer side of a producer-consumer queue, at the start of every frame computation, the HI inserts a frame header with an ID into all outgoing queues to identify the frames as a sequence. This frame header is an alignment marker in the outgoing stream of data. Even if the count of data items or their specific bit values are incorrect in the stream that follows, the frame header gives downstream AMs specific points at which alignment can be restored. The HI inserts the value of active-fc as the frame ID. When active-fc reaches its limit, indicating the end of this thread's computation, a special frame ID indicating the end of computation is inserted to every outgoing queue. A thread is oblivious to the HI actions. Header reliability is ensured using ECC.

Table 1: Alignment manager FSM states and transitions. FSM receives events when local thread starts a new frame computation or issues a pop instruction. The FSM transitions to new states depending on the incoming header ID and the *active-fc*.

State	State Activity	Event	Next State
RcvCmp	Receiving and computing on items for active frame computation	New frame computation started	ExpHdr
		Received future header	Pdg
		Received past header	Disc
ExpHdr	New frame computation started and expecting header from queue	Received correct header	RcvCmp
		Received item or past header	DiscFr
		Received future header	Pdg
DiscFr	Discarding frames from queue (AE_{FE})	Received correct header	RcvCmp
		Received future header	Pdg
Disc	Discarding items and frames from queue (AE_{IE}, AE_{FE})	Received future header	Pdg
Pdg	Padding thread for lost data (AE_{IL}, AE_{FL})	New frame computation matched header	RcvCmp

4.2 Alignment Manager

The AM within a consumer core aligns incoming data with the execution of a consumer thread using the frame ID's in the communicated headers and the active-fc counter. The AM controls the data that is being transferred to the currently executing thread by discarding or padding items when necessary.

Table 1 summarizes the checking and control operations in AM in the form of a state transition table. The AM checks and responds to each *pop* instruction and to the start of a new frame computation. In response to a *pop* instruction, the QM is invoked to return the next item, which may be a regular item or a header. If a misalignment is detected, it may *pad* items to respond to the *pop* requests from the thread or *discard* items in the queue until the misalignment is resolved.

The normal consuming state of the thread is *RcvCmp*. The state *ExpHdr* denotes that the thread's control-flow has just rolled over to a new frame computation. That is, the*RcvCmp* state expects an item and *ExpHdr* expects a frame header from the queue. Failure to get what is expected indicates a misalignment, and puts the AM into one of the erroneous states— *Disc*, *DiscFr* or *Pdg*—to handle the specific alignment error. (These are shown in parentheses in Table 1.) The AM resolves erroneous states by advancing the local computation or communication to the correct frame boundaries. More specifically, either items are discarded to bring an incoming queue to the correct frame boundary (communication realignment) or items are padded to bring the local computation to the correct frame boundary (computation realignment).

4.3 Queue Manager

As shown in Figure 4, the QM interacts directly with cores and the CommGuard modules. Its responsibility is to transfer items/headers to the AM or to other threads when requested. The QM responds to *pop* and *discard* requests of the AM, and the *push* requests of the core and the HI. Hence, the QM performs these actions; (i) sending/receiving of items through the memory subsystem (ii) separating items and headers, and (iii) ECC checking of headers. Reliable queue management eliminates the errors belonging to the QME group from Section 3.

Here, we consider a hardware implementation for the data queue with ISA-visible push and pop operations. The queues (i.e., the items and their head and tail pointers) reside in a static location in memory and the QM implements the operations for a queue data structure. Section 5.1 elaborates on details of this design choice.

4.4 Guided Execution Management and Its Integration With CommGuard

We now describe CommGuard's integration with the PPU cores in [32]. The application is divided into coarse-grained (potentially nested) control-flow regions termed *scope*s. These are demarcated in the StreamIt benchmarks and are at the granularity of function calls or loop nests. The PPU cores ensure: (i) the thread running on a core sequences correctly from one scope to the next, and (ii) it does not loop indefinitely within some scope. This ensures acceptable coarse-grained forward progress for the application.

The PPU protection module monitors computation inside a thread and increments the active-fc, which the HI and AM use as explained in Sections 4.1 and 4.2. For the StreamIt programs as written, a scope encompasses each frame computation. Choosing coarser scopes or down-sampling the active-fc increment frequency through a saturating counter allows the CommGuard modules to monitor alignment at larger granularities. Lastly, to determine when the computation in a thread ends, the PPU protection module signals CommGuard when a thread's outermost global scope exits.

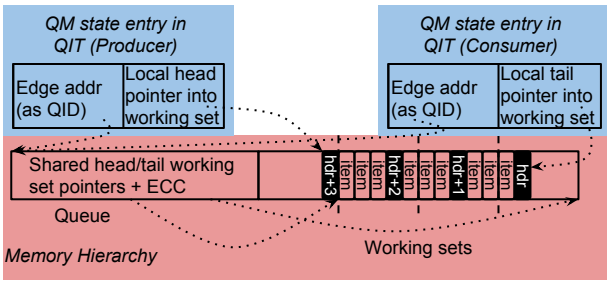


Figure 6: QM uses local pointers to access the local working set and shared pointers to share working sets across threads.

5. Hardware Design Considerations

5.1 Queue Manager

For this work we assume a hardware queue implementation as shown in Figure 6. The QM protects head/tail pointers through ECC and shares them with QM's in other cores. The QM follows the StreamIt implementation for a parallel queue; a 320KB memory region divided to 8 sub-regions to avoid per-item access to the head/tail pointers. Further, a blocked queue that spins on a condition does not repeatedly compute ECC since it is looking for a change in its value.

QM employs blocking operations when the queue is full or empty. PPU cores determine the queue state, e.g., number of items in the queue can overflow due to erroneous control-flow. Therefore, the QM needs timeout mechanisms to avoid indefinite blocking. A timeout may cause incorrect data to be transmitted but frame checking would still ensure alignment at the frame boundaries.[1]

5.2 Hardware Operations

Table 2 shows the hardware suboperations CommGuard executes in response to the CommGuard interface events. The *push* event adds the item to the local working set and gets a new working set if the working set becomes full. In response to a *pop* event, the FSM in Pdg state responds to the request with a 0. Otherwise, the next data unit is popped from the queue. Depending on if the data unit is a regular item or a header and the state of the FSM CommGuard may issue more pops and discard items (see Table 1 for different cases). *New frame computation* event triggers header insertions for all outgoing queues. Table 3 shows the overheads of the suboperations from Table 2.

5.3 Timing, Pipeline and Speculation

CommGuard's ECC and header bit pattern operations (Table 3), are off the critical path of execution in steady-state operation. Header bit patterns and ECC can be computed prior to insertion. At the receiver, the header-bit-pattern and ECC operations can be performed for the head of the queue before the item is needed.

[1] We did not observe any timeouts in any of our experiments.

Table 2: CommGuard Suboperations

Architectural Operation	CommGuard Suboperations
push	QM-push-local item if QM-local-full QM-get-new-workset
pop	if FSM-check not Pdg do QM-pop-local if QM-local-empty QM-get-new-workset if is-header then check-ECC for header FSM-check/update while FSM not DONE
new frame computation	prepare-header compute-ECC for header for out-queues: FSM-update QM-push-local header if QM-local-full QM-get-new-workset

Table 3: CommGuard Suboperations and Their Overheads

CommGuard Suboperation	Operational Overhead
prepare-header	Read then increment active-fc, set header-bit
is-header	Check header-bit
check/compute-ECC	Single-word ECC set/check
FSM-check/update	Checking/updating a 5-state FSM (see Table 1)
QM-push/pop-local, QM-local-full/empty	Additional memory events for header transmissions (No CommGuard overhead for regular item transmissions)
QM-get-new-workset	10 check/compute-ECC operations for shared pointer access through QM

Frame computation invocations are serializing operations for push/pop instructions, because CommGuard compares headers coming from the incoming streams with active-fc as set by frame computation invocations. While this dependency may incur stalls, the resulting performance hit is small since frame computations usually contain more instructions than typical pipeline depths. The number of instructions per frame computation in the median threads of our applications ranges from 72 and 33 for `audiobeamformer` and `complex-fir`, to thousands for the other applications. We evaluate the run-time overhead of the stalls in Section 7.2.2.

We propose three options regarding speculation for CommGuard's architecturally visible push and pop operations: (i) A processor may execute instructions speculatively until it encounters a push/pop. (ii) CommGuard QM stores speculative state for buffer pointers and committed push/pop instructions alter the visible queue state. This adds one copy in the QIT (Figure 4) for each speculative local pointer (Figure 6). (iii) QM operations use processor load/store operations to utilize the existing speculative hardware. Here, we consider option (ii) as the speculative storage overhead is small (Section 5.5).

5.4 Varying Frame Sizes Throughout an Application

Frame sizes are design knobs for adjusting CommGuard alignment granularity. The CommGuard implementation explained here targets a uniform frame definition across an application. That is, for all threads every frame received from the incoming queues matches a frame computation and every frame computation triggers a header insertion to the outgoing queues indicating a new frame. CommGuard can increase the application-wide frame definitions by downscaling the frame computation frequencies through one saturating counter for frame computation invocations. In addition, CommGuard can also support varying frame definitions across an application. This requires a redundant active-fc counter per frame domain. We do not detail this design here since our experiments showed that the default application-wide frame definitions provide sufficient output quality for our benchmarks.

5.5 Hardware Area

We do not provide circuit-level implementations for the logic of AM, HI and QM but we expect this to be small based on the storage needed. CommGuard modules need reliable storage for maintaining static and dynamic state and here we show that this storage is small enough to reside on core.

In our implementation, CommGuard modules store 2 counters and their limits; active-fc and a saturating counter to optionally down-scale frame computation frequency for larger frame sizes. Further, the modules need to store the following for each incoming queue; 3-bits and 1 word for header, queue ID, the local buffer pointer and its speculative copy in the QIT (Table 1, and Figure 6). The number of queues per thread varies and for our benchmarks this number was at most 4. Therefore, with 4 queues per core the total reliable storage would account to $4 \times 4B + 4 \times (3bits + 4B + 4B + 4B + 4B) \approx 82B$. Hence we assume this is completely cached on core.

6. Experimental Methodology

Simulation: To validate CommGuard, we implement the proposed hardware modules in a detailed Virtutech Simics functional architecture simulator [20] extended with error injection capabilities. Our baseline system is a 32-bit Intel x86 architecture simulating 10 processor cores. Our simulator models hardware errors through architectural error injection and implements the PPU cores described in [32].

Every core in our simulator implements an error injection module that randomly flips bits in the register file. Each error injector picks a random target cycle in the future following the mean error rate, and flips a random bit in the register file when the simulation reaches the target cycle. Register-based error injection has been used in prior work [18, 30].

We vary mean-time-between-errors (MTBE) to capture the trends in output quality changes, run-time and error handling statistics. Each core's error injection is independent and has its own random number generator. Therefore, the MTBE mentioned in our experiments represents the MTBE of each core, i.e. not the MTBE of the whole multi-core processor. (So errors in the full multi-core are more frequent.) For every MTBE, we ran the application 5 times using different random number generator seeds and observed the deviation across different runs.

To model realistic error rates, we use functional ISA simulation, which allows us to run applications fully at error rates of interest and compare output qualities. Our simulator does not use detailed microarchitectural models [2], which would provide accurate performance models but at the cost of simulation speed. CommGuard events are triggered by instructions modeled as ISA extensions. We implemented the new hardware modules, QM, HI, and AM as explained in Section 4. The hardware events of the CommGuard modules are implemented using the Simics API for hardware modules.

We implemented StreamIt's *push* and *pop* operations as ISA extensions that use x86 registers for hand-off for data and edge identifiers. Therefore, data transmissions are also error-prone due to our register-based error injector. This models uncore error sources. Headers are not error-prone because we assume they are end-to-end ECC protected and account for their overhead. Headers with their ECC are also word-sized items since header values in the order of thousands are enough to identify frames across a streaming graph.

Power Estimation: As a proxy for power estimates, we record processor event counts (i.e., all instructions committed, memory accesses), and CommGuard overhead event counts (Tables 2 and 3) during all simulation runs.
CommGuard's remaining hardware operations are simple comparisons and arithmetic operations. CommGuard suboperations are on par with or simpler than instruction processing pipestages. Aside from *push* and pop instructions, they are not implemented as software visible instructions, but to give some intuition regarding their relative frequency, we show hardware operation counts as a percentage of processor instruction counts.

Real-System Measurements: To estimate pipeline performance effects (Section 5.3) we use real hardware measurements of slightly-modified StreamIt applications running on an 8-core Xeon E3-1230 (3.30GHz) *without* Simics. We modify applications by inserting an *lfence* instruction at all frame boundaries. The lfence instruction in Intel x86 architecture is a serializing instruction, i.e. all instructions complete before lfence executes [15]. This serialization reflects the fact that our CommGuard implementation requires that pushes&pops that follow a new frame computation stall until the instruction for the new frame computation commits. lfence instructions at frame computation boundaries serialize all instructions and (conservatively) observe CommGuard dependencies.

Benchmarks: Our experiments use 6 benchmarks from the StreamIt benchmark suite: `audiobeamformer`, `channelvocoder`, `complex-fir`, `fft`, and widely-used multimedia applications `jpeg` and `mp3`. We used the cluster backend of StreamIt to automatically parallelize these applications for 10 cores using the shared-memory model. We modified the StreamIt compiler backend to use the new instructions for hardware-push and hardware-pop instead of library calls to the existing StreamIt software implementation of communication.

`mp3` and `jpeg` are lossy compression/decompression algorithms. Lossiness is commonly measured using signal-to-noise-ratio (SNR) for audio, and using peak-signal-to-noise-ratio (PSNR) for image [27]. The image quality obtained by standard lossy compression levels [28, 31] is a baseline in our experiments. To provide this quality reference we first encode the raw input file. We decode the raw data through an error-free decoder to establish the baseline SNR under lossy compression. We then decode the encoded file through the decoder running on error-prone hardware, and compare the result quality (both algorithmic and error-prone lossiness) with the baseline (only algorithmic). For the remaining four applications, we calculate SNR to compare the outputs of error-prone runs directly with the outputs of error-free runs.

7. Experimental Results

Our experiments show that (i) CommGuard avoids permanent corruptions due to communication and control-flow errors and has good output quality even at high error rates (Section 7.1) and (ii) it does this with low overhead (Section 7.2).

7.1 Quality of Results

Figure 7 shows an example run of the `jpeg` application on error-prone processors (MTBE of 512K instructions) with CommGuard modules. `jpeg` decoding the full image required 16 padding and discard operations. The frames at the `jpeg` output are rows 8-pixels high and pad/discard events are denoted on the right hand side of these rows.[2] Without CommGuard, Figure 3c showed that the output of `jpeg` is completely corrupted at this error rate even with a reliable queue and PPU cores. On the other hand, Fig-

[2] A misalignment may have happened in any thread. Because all threads share the frame definitions, we can annotate them visually at the application output.

Figure 7: Example `jpeg` output running with CommGuard (MTBE of 512k instructions) with PSNR of 20.2dB. Arrows for pad/discard actions are shown adjacent to the frames for which CommGuard detected misalignment.

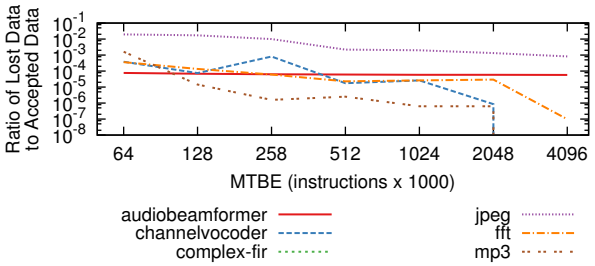


Figure 8: Realignment may result in data loss of correct items. We calculate data loss as the sum of padded and discarded bytes.

ure 7 shows only some 8-pixel-high erroneous lines. As CommGuard aligns computation and communication at every frame boundary, the frames following the corrupted frames start without the effects of misalignment. This is critical for the effects of control-flow errors to be ephemeral and not cumulative.

Figure 8 shows the amount of data loss (i.e. padded or discarded items) due to realignment across different error rates. Even at extreme error rates (MTBE of 64K instructions) the loss is less than 0.2% for five benchmarks. Jpeg loses more data than other benchmarks because it has the lowest frame/item ratio for default frame size (1 frame for every $7K$ items on average). However, the loss is still less than 0.2% at an MTBE of 512K instructions. This shows that data loss due to padding and discarding actions are small. The visible error clusters are largely due to intra-thread control-flow and memory addressing errors.

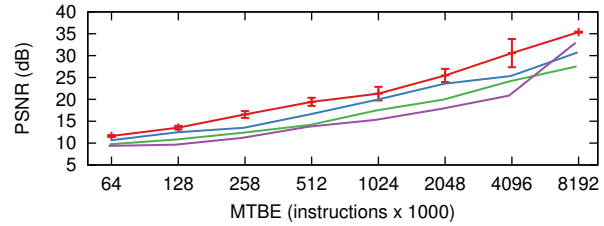
Figure 9 shows different outputs and PSNR values for the `jpeg` application at varying MTBE values. PSNRs of 20dB or more are quite acceptable, and for PSNRs larger than 30dB, the errors are hardly visible. When errors are quite frequent (MTBE=128k), the image is more corrupted, but the flower is still recognizable. Thus this technique is useful even at extreme error rates.

Figure 10 shows output quality averages and standard deviations for varying MTBEs for `jpeg` and `mp3`, respectively. For MTBEs as frequent as 512K instructions, output quality

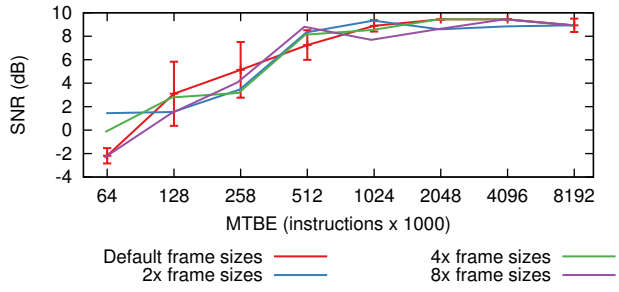
(a) MTBE=128k (PSNR: 14.7dB) (b) MTBE=512k (PSNR: 18.6dB)

(c) MTBE=2048k (PSNR: 28.6dB) (d) MTBE=8192k (PSNR: 35.6dB)

Figure 9: Visual results with PSNR values at varying MTBE values. Output quality reaches error-free PSNR, 35.6dB, at an MTBE of 8192k instructions.



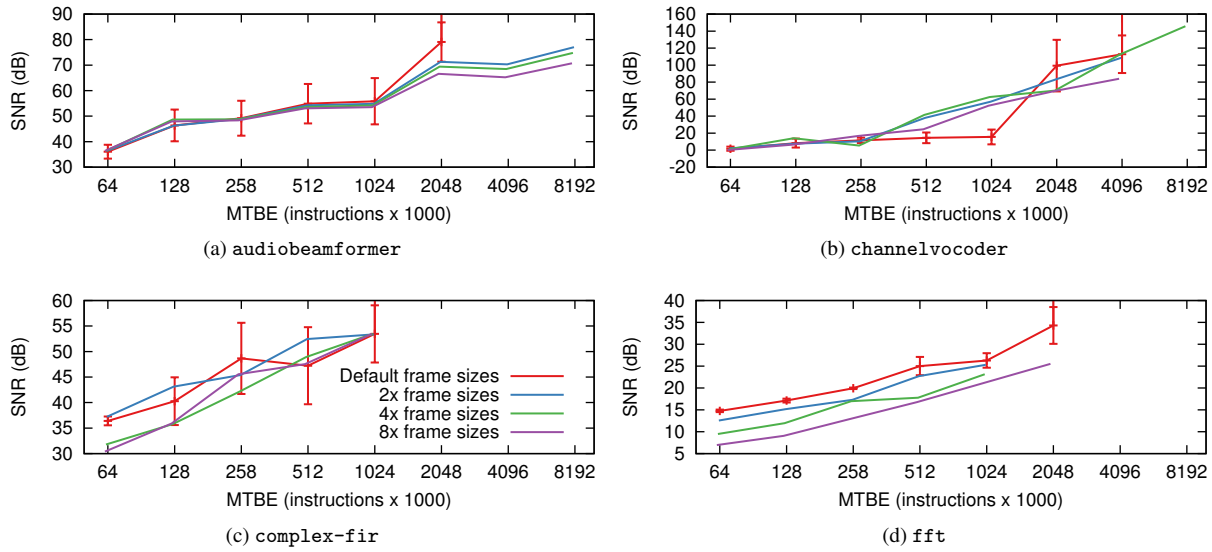
(a) `jpeg` (error-free PSNR: 35.6dB)



(b) `mp3` (error-free SNR: 9.4dB)

Figure 10: Comparison of quality loss due to "error-free lossy compression" and "error-prone lossy compression with CommGuard" at varying MTBEs and varying frame sizes. (Deviations are shown only for default frame sizes.)

is maintained at very good levels: 20dB (error free PSNR of 35.6dB) and 7.6dB (error free SNR of 9.4dB), respectively. Example audio outputs for different error rates for the `mp3` application may be heard at: http://youtu.be/k72hTg5zbik. With larger frame sizes, realignment operations less frequent. Therefore, run-time overheads decrease (as discussed



(a) `audiobeamformer` (b) `channelvocoder`

(c) `complex-fir` (d) `fft`

Figure 11: Output quality loss of error-prone runs with CommGuard with respect to the error-free execution (error-free SNR is infinity).

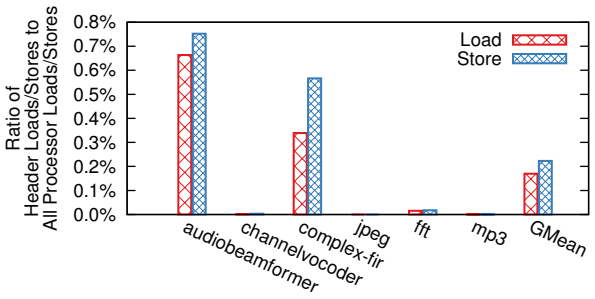


Figure 12: CommGuard overhead on memory events.

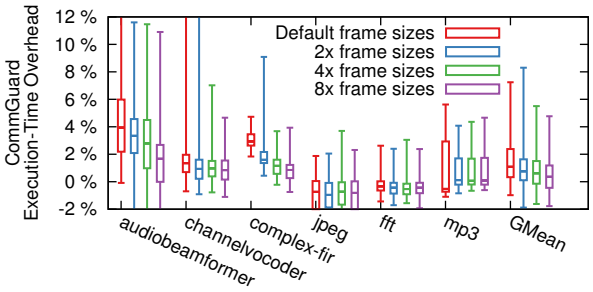


Figure 13: Runtime overhead of additional header pushes and pops, and pipeline serialization at frame boundaries. Measured on real hardware (100 runs) for varying frame sizes. Headers are additional item pushes&pops and pipeline stalls at frame boundaries are implemented by lfence instructions. Reference is execution without CommGuard.

in Section 7.2) but a misalignment is likely to corrupt more data.

Figure 11 shows SNR values for the remaining benchmarks. Among these benchmarks, `complex-fir` maintains error-free output quality for MTBEs as frequent as 1024K, while `channelvocoder` shows the same improvement at higher MTBEs (4096K). Similarly, `channelvocoder`, `fft`, and `mp3` approach 0dB quickly, whereas the remaining benchmarks are more resilient even at extreme error rates.

7.2 Hardware Overheads

CommGuard dynamically monitors execution and communication, induces additional memory events due to header pushes and pops and may affect the pipeline. Our evaluations show that coarse-grained protection keeps overheads low.

7.2.1 Extra Memory Events due to Headers:

CommGuard induces extra queue pushes and pops for the headers it inserts. The CommGuard QM implements the header pushes by storing the inserted headers in the software queue (just like data items in the queue). Since the queue is implemented as a software data structure in normal program memory, there is no extra storage overhead per se, but headers may lead to energy and performance overheads. As a proxy for power/energy influence in the memory hierarchy, Figure 12 shows the additional memory events due to header pushes and pops. (Performance effects are below.) Across benchmarks, our experiments show that the geometric mean increase in memory events is less than 0.2%. The maximum effect is for `audiobeamformer`, and even that is very small: Frame headers increase the number of data loads by only 0.66% and data stores by only 0.75%.

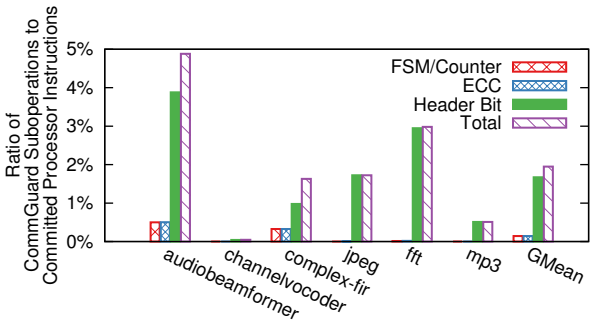


Figure 14: CommGuard operations normalized to processor instruction commits.

7.2.2 Execution Time Overheads:

Here, we measure CommGuard runtime overhead using the techniques described in Section 6, which account for extra pushes, pops, and pipeline serialization at frame boundaries. Figure 13 shows that extra CommGuard operations do not affect the performance for `jpeg`, `fft`, or `mp3`. `audiobeamformer` and `complex-fir` show the worst behaviors but their execution time increase is still less than 4%. The mean runtime overhead across all benchmarks is only 1%. Increasing frame sizes slightly reduces the already low execution time overheads, but since some applications, such as `jpeg`, have output quality sensitive to frame size (Figure 11) it is preferred to use the StreamIt-default frame size rather than anything larger.

7.2.3 Run-time Activity of CommGuard Modules

Tables 2 and 3 show the hardware operations that CommGuard may execute during run-time. CommGuard operations occur very infrequently relative to other processor activity. While some of our benchmarks (`audiobeamformer`, `channelvocoder`) contain threads that have a frame size of 1 item (i.e., one header overhead for every data item), the operations in the processor core still dominate all CommGuard hardware activities.

Figure 14 shows that the CommGuard operations are only as frequent as 2% of processor instruction commits (geometric mean). Even in the worst-case (`audiobeamformer`) all operations occur as frequently as 4.9% of processor instruction commits. The *header bit-pattern* operations (to indicate if a data unit is a regular item or a header) are the most frequent operations and are simple less-than-a-word size sets/checks. ECC computation for the headers (a relatively more complicated operation even though only for word-sized data) occurs only for 0.5% of instructions for `audiobeamformer`.

8. Applicability to Other Programming Models

In this work, CommGuard utilizes high-level information from StreamIt's programming model. However, CommGuard's principles of modular error-tolerance through coarse-grain control-flow/data identification apply more broadly to other programming models. The key attribute used by CommGuard is a frame structure linking shared data or communication to coarse-grained program control flow. Other StreamIt attributes, such as strict producer/consumer relationships and static definition of the size of data transfers are not needed by CommGuard.

Programming models that can express high-level control-flow constructs and how these control-flow constructs in different threads relate may easily implement CommGuard. For example, Concurrent Collections [3] expresses control-flow by *tagging* produced items of a thread and *steps* threads with a matching tag. Similarly, *keys* in MapReduce [8] programs identify a group of items and express the sequencing of parallel operations. CommGuard's headers are identifiers for data frames, and alignment manager modules use these identifiers for realignment. Other research such as DeNovo also proposes hardware-software constructs for coherence and communication that can similarly be employed [5].

9. Discussion and Related Work

Modularity and Self-Stabilization: CommGuard allows each local thread to make control-flow, memory addressing, and data errors without considering their effects on other threads. In addition to the design advantages of core-local management, this modular structure makes CommGuard easy to reason about and verify.

The concept of *Self-stabilization* [9] from distributed systems formalizes the *ephemeral* effects of errors. Self-stabilizing systems guarantee that a perturbation resulting in an erroneous state will eventually be corrected to bring the system to a valid state. Our design principles align with this goal. Errors may cause degradation of the output quality but the reliable protection modules repair the system to continue producing useful output. Existing techniques can rely on CommGuard's modularity to verify global self-stabilization properties using self-stabilizing local execution. Related work [10] shows verification of self-stabilizing Java programs. Work in [4] further shows how error-tolerant hardware models can be specified to verify such program-level properties.

Reliable Hardware: To address hardware errors, SRAM and other storage units use error correction codes [1], and other hardware units (i.e., logic) can use larger transistors or higher voltage [17, 24]. Razor [11] is a lower-level example of low-overhead error recovery. It detects timing errors using shadow latches and recovers from them by pipeline flush and re-execution. Razor provides low overhead error protection by targeting timing errors in critical paths. Redundant program execution [22] at different pipelines/hardware threads/processors helps recover from errors but incurs very high energy/performance overheads due to double redundancy for error detection and triple redundancy or re-execution for er-

ror recovery. These techniques are all directed towards maintaining an error-free hardware abstraction. In this work, we target applications that are tolerant to data errors and propose using these hardware reliability techniques only for small reliable modules. The rest of the micro-architecture permits errors that are potentially acceptable at the application level. Argus [21] detects errors separately for control-flow, data-flow, computation and memory. Like Argus, our work also distinguishes these concerns, and further includes inter-core communication errors. However, Argus targets error detection, while we go further with limited error correction to provide useful computation results.

Application-Level Error Tolerance: Some work has, like ours, leveraged application-level information while handling errors in computation. ANT (Algorithmic Noise Tolerance) [13] uses an estimator block to mitigate the effects of errors for circuits implementing signal processing algorithms. This is a hardware solution that does not need to deal with control-flow issues addressed in our work. Placing checkers using heuristics [30] strives to improve detection of control-flow and memory addressing errors. In contrast, CommGuard uses coarse-grain application information to detect the catastrophic errors and manage their effects.

Program-Level Specification and Analyses: EnerJ [25] partitions computation into error-tolerant and error-intolerant instructions to be implemented using "error-prone but low power" and "error-free but high power" components. EnerJ does not allow any errors on control-flow or memory addressing and requires a fully-reliable processor for error-intolerant instructions; this requires over half of the processor to be designed with reliable hardware. While EnerJ targets reducing energy consumption through using error-prone functional units on an otherwise-error-free processor, CommGuard more aggressively targets error-tolerant computation on unreliable processors that can only afford small reliable components.

Rely [4] lets programmers express application-level properties for error-tolerance and uses probabilistic hardware models to prove these properties. Further, Rely allows some control-flow and memory addressing errors. Here, we have shown that relaxed control-flow and memory addressing quickly cause catastrophic errors due to inter-core communication. Therefore, without CommGuard, Rely's reliability analysis would capture the misalignments and conclude that the application has virtually zero reliability. However, with CommGuard, the reliability analysis can capture that error effects do not propagate across frame boundaries. As a result, Rely's reliability analysis may compute the overall application reliability for streaming data. We experimentally verified CommGuard and have laid out this verification possibility using Rely-style analysis as future work.

Error-Tolerant Architectures ERSA [18] uses one reliable processor core to control algorithmic flow and many unreliable processors cores to execute error-tolerant worker tasks. ERSA requires application rewrite to fit this partitioned do-all execution model and the overhead of the reliable processor core can only be amortized when the application has many parallel threads. In contrast, CommGuard has fewer demands on the programming model, is fully automated in the StreamIt compiler and can also handle do-all parallelism which can be easily written in StreamIt. CommGuard could potentially enhance ERSA by enabling inter-core cooperation among the unreliable cores and thus help expand its programming model.

10. Conclusions

This work investigates the challenges of parallel program execution on error-prone hardware, and proposes CommGuard as a low-overhead communication protection mechanism. CommGuard uses small reliable hardware modules designed as simple FSMs to manage the coarse-grained computation progress of a core within which they reside, and the communication to/from that core. CommGuard uses application-level constructs that express how coarse-grain control-flow operations are related to data-flow operations. We use already existing constructs in StreamIt as a concrete example, however these constructs exist in other high-level programming languages as well.

Our results show that CommGuard can sustain correct operation and provide good output quality: 20dB for `jpeg` and 7.6dB for `mp3`, running on 10 error-prone cores, for errors occurring as frequently as every 500μs at each core. CommGuard avoids catastrophic degradation of output quality and only introduces mean overheads of 0.3% on the memory subsystem events, 2% as additional hardware operations relative to the committed instructions, and 1% on execution time.

11. Acknowledgements

The authors acknowledge the support of C-FAR and SONIC (under the grants HR0011-13-3-0002 and 2013-MA-2385), two of six SRC STARnet centers by MARCO and DARPA. In addition, this work was supported in part by the National Science Foundation (under the grant CCF-0916971).

References

[1] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-efficient cache design using variable-strength error-correcting codes," in *Proceedings of the Annual International Symposium on Computer Architecture*, 2011.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[3] Z. Budimlic, M. Burke, V. Cave, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach,

and S. Tasirlar, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[4] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

[5] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "Denovo: Rethinking the memory hierarchy for disciplined parallelism," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[6] 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.

[7] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[10] Y. h. Eom and B. Demsky, "Self-stabilizing java," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2012.

[11] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proceedings of the Annual International Symposium on Microarchitecture*, 2003.

[12] G. Gielen, P. De Wit, E. Maricau, J. Loeckx, J. Martín-Martínez, B. Kaczer, G. Groeseneken, R. Rodríguez, and M. Nafría, "Emerging yield and reliability challenges in nanometer cmos technologies," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2008.

[13] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1999.

[14] 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.

[15] Intel Corporation, vol. 3A, pp. 8–16, 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[16] ITRS, "ITRS process integration, devices, and structures," http://public.itrs.net/Links/2011ITRS/2011Chapters/2011PIDS.pdf, ITRS, 2011.

[17] 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.

[18] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010.

[19] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[21] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Proceedings of the Annual International Symposium on Microarchitecture*, 2007.

[22] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the Annual International Symposium on Computer Architecture*, 2002.

[23] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the Annual International Symposium on Microarchitecture*, 2003.

[24] S. Mukherjee, *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., 2008.

[25] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2011.

[26] 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.

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

[28] G. Stoll and K. Brandenburg, "The iso/mpeg-audio codec: A generic standard for coding of high quality digital audio," in *Audio Engineering Society Convention*, 1992.

[29] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the International Conference on Compiler Construction*, 2002.

[30] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *Proceedings of the Conference on Dependable Systems and Networks*, 2013.

[31] G. K. Wallace, "The JPEG still picture compression standard," *Commun. ACM*, vol. 34, no. 4, 1991.

[32] Y. Yetim, M. Martonosi, and S. Malik, "Extracting useful computation from error-prone processors for streaming applications," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013.