Extracting Useful Computation From Error-Prone Processors For Streaming Applications

Yavuz Yetim Margaret Martonosi Sharad Malik
Princeton University

***Abstract*—As semiconductor fabrics scale closer to fundamental physical limits, their reliability is decreasing due to process variation, noise margin effects, aging effects, and increased susceptibility to soft errors. Reliability can be regained through redundancy, error checking with recovery, voltage scaling and other means, but these techniques impose area/energy costs. Since some applications (e.g. media) can tolerate limited computation errors and still provide useful results, error-tolerant computation models have been explored, with both the application and computation fabric having stochastic characteristics. Stochastic computation has, however, largely focused on application-specific hardware solutions, and is not general enough to handle arbitrary bit errors that impact memory addressing or control in processors.**

In response, this paper addresses requirements for error-tolerant execution by proposing and evaluating techniques for running error-tolerant software on a general-purpose processor built from an unreliable fabric. We study the minimum error-protection required, from a microarchitecture perspective, to still produce useful results at the application output. Even with random errors as frequent as every 250μs, our proposed design allows JPEG and MP3 benchmarks to sustain good output quality—14dB and 7dB respectively. Overall, this work establishes the potential for error-tolerant single-threaded execution, and details its required hardware/system support.

I. Introduction

As transistor sizing and threshold voltages approach their limits, semiconductor fabrics are expected to experience increasing device failures and higher error rates [7]. These trends make it increasingly difficult to maintain an error-free hardware abstraction. Techniques such as modular redundancy, error checking with recovery, or voltage scaling may mitigate error frequency, but often at high area/energy costs. Razor [4] focused on checking and recovering from timing errors to reduce these overheads. Stochastic computing approaches relax the expectation of error-free hardware with specialized approaches for inherently error-tolerant applications, such as media processing [6]. Such stochastic computing, however, mostly assumes application-specific hardware solutions.

An important next step is to extend error-tolerant computation to software on programmable processors implemented on unreliable fabrics. Irrespective of application error-tolerance, corruptions in architectural state can corrupt program control flow or cause memory addressing errors or other exceptions that terminate the program. Our characterizations show that a streaming application whose runtime is only 50ms may infinitely loop or crash when bit errors occur roughly every 1ms. In previous work, Jolt [1] detects some infinite loops, and exits them by dynamic binary modification. A system with hardware errors must more aggressively detect infinite loops and cannot rely on software-only solutions. ERSA [9] deals with hardware errors by isolating the algorithmic flow on a highly reliable core, and the data computation on reduced reliability cores. This solution requires a reliable core, and also requires the application to be easily partitioned into a single control and multiple worker threads. EnerJ [13] and Flikker [10] are distinct in proposing type systems to annotate variables and partition the computation/data into error-tolerant and error-intolerant parts. They further partition the hardware as "error-prone/low power" and "error-free/high power". Error-prone components run error-tolerant instructions, store error-tolerant data, whereas error-free components run error-intolerant instructions and store error-intolerant data. This approach ensures reliability but significantly reduces potential gains, because such partitioning means less than half of the instructions are error-tolerant and less than half of the core is error-prone.

This paper studies the minimum error-protection required from the microarchitecture in order to support useful application-level results for streaming applications running on a general-purpose processor built on an error-prone fabric. Overall, we make the following contributions: i) We are the first to propose a solution to ensure software progress through errors without assuming any fully-reliable cores or a specific error model. Further, we characterize how hardware errors propagate through layers of the computation stack and change program behavior or output. We propose minimal protection schemes for each type of error. Our solution space includes components that guide control flow, memory addressing and I/O accesses with customizable granularities to handle these errors. ii) Even though errors quickly become program-critical under even modest error rates, we show that control flow and memory addressing do not have to be perfectly reliable to eliminate such outcomes. Our microarchitectural components bound and guide computation at a coarser grain to avoid these crashes and hangs with minimal hardware overhead. iii) We further show that modest information on the program structure of streaming applications allows further error mitigation. Our solutions do not require rewrite of the streaming applications; in fact the only code transformation we require can be automated in the compiler. iv) We show the results of different error types for 7 Streamit [15] benchmarks including two widely used multimedia benchmarks (JPEG and MP3 decoders). The resulting output quality is as good as the zero-error case for errors that occur as frequently as every 10^7 instructions ($<$ 10ms of runtime). For even more frequent errors (every 250 microseconds) the SNR value remains acceptable: 14dB (-32%) for JPEG and 7dB (-28%) for MP3.

II. Overview

Error Dependence Characterization: While applications may have some inherent error tolerance, that does not translate uniformly to instruction-level error-tolerance. Errors in critical

The first author can be reached at yyetim@princeton.edu. The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. In addition, this work was supported in part by National Science Foundation.

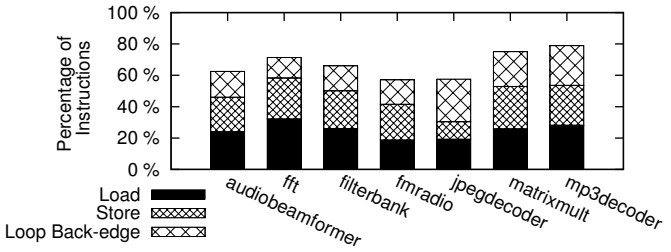


Fig. 1. Percentage of error-free instructions needed to avoid i) memory accesses that may cause crashes and ii) control flow operations that change looping behavior leading to hangs.

instructions may have a catastrophic effect on execution, even for error-tolerant applications. For example, memory access instructions cause segmentation faults if a corrupted address points to a disallowed location. Similarly if control flow is corrupted, a program may hang or loop indefinitely. A location/instruction is *error-intolerant* if its corruption could lead to a catastrophic failure such as a crash or a hang. For an application to progress without crashing due to segmentation faults or going into an unresponsive state, all memory addressing and loop back-edges are considered as error-intolerant. Transitively, all control and data dependencies for intolerant instructions must be considered error-intolerant as well.

Given the likelihood of these dependence chains of error-intolerance, we first characterized the error-intolerant instructions in StreamIt applications. Using LLVM [8], we marked the error-intolerant instructions and transitively their data and control dependencies. Figure 1 shows that as many as 65% of all instructions are error intolerant in these benchmarks and thus need some protection to avoid crashes and hangs. The predominance of error-intolerant instructions motivates our work to mitigate control and memory addressing errors to improve application success rates on error-prone fabrics. Further, our error-control mechanisms must work well and incur low overheads despite this predominance.

Minimal Requirements For Error-Tolerant Systems: Overall, in order to provide acceptable error-tolerant application operation on error-prone fabrics, there are four design requirements that a system must meet. First, the program should not hang or run indefinitely. Control flow errors that result in infinite loops or other failures to terminate must be addressed. Second, the application should only be allowed to access information that it is allowed to. Memory addressing errors that cause the program to access off-limits areas must be handled. Third, application input/output sequences must not cause external device corruption such as filesystem errors. This is related to the second requirement, but calls for proper I/O controls. Fourth and very important, the *accuracy* of the computation as viewed in terms of the end-result data values stored in the program outputs must be acceptable, i.e., errors should result in only acceptably-small changes to these output data. Acceptability is defined via an appropriate application-level metric, such as SNR [14]. Thus, errors that result in small changes in calculation data, or even small (within-range) memory addressing errors or control flow errors, are all acceptable, as long as the above four requirements are met.

III. Proposed Design

Figure 2 shows the key components of our proposed approach, with the five reliable modules in dashed boxes. The

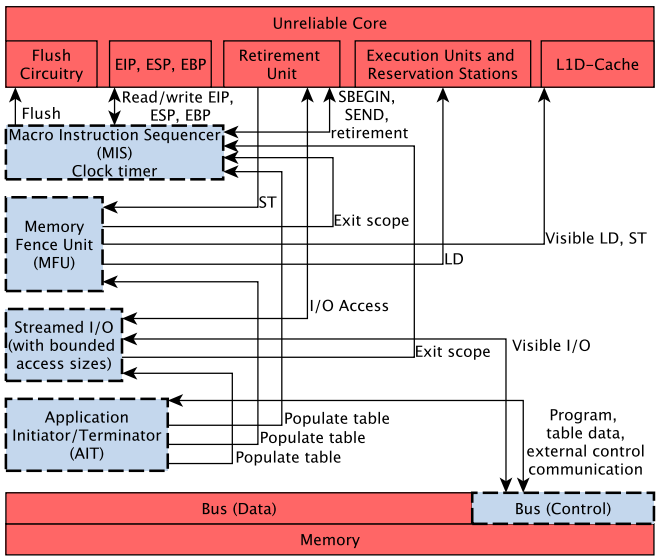


Fig. 2. Our low-overhead support for an application to eliminate crashes, hangs and device corruptions includes five reliable components (dashed outline, only conceptually separated from the core). The *macro instruction sequencer (MIS)* with a timer ensures forward progress. The *memory fence unit (MFU)* constrains memory accesses. *Streamed I/O* manages bounded data streams. *Application initiator/terminator (AIT)* communicates with the components and external devices/processors on application initiation or termination, and *Bus control* handles correct communication with the external devices/processors. The rest of the system (solid outline) can be unreliable with best-effort operation.

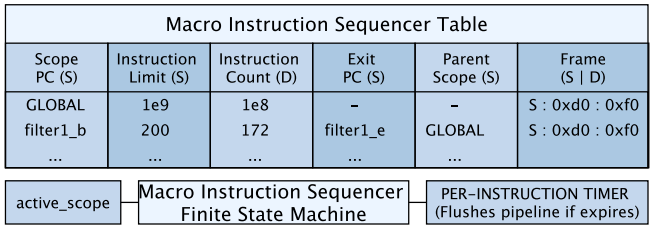


Fig. 3. *Macro instruction sequencer* guides the control flow of an application by enforcing bounds on nested scopes. The table stores information regarding the *scopes* of a program and the FSM uses and updates this information to handle application hangs or other exceptions.

three bounds modules—*MIS*, *MFU* and the *streamed I/O*—constrain execution based on application profile information, and are described in the following subsections. The *AIT* and *bus control* communicate with external devices/processors.

A. Macro Instruction Sequencer

The *Macro Instruction Sequencer (MIS)* has primary responsibility for constraining the control-flow behavior of the system. Central to the MIS design is the observation that a single-threaded streaming application can be viewed as a series of coarse-grained chunks of computation. Thus, we can constrain coarse-grained control flow by bounding the allowed operation count per chunk, and by retaining information about legal or likely series of chunks. Figure 3 shows the MIS implementation—primarily a state machine and a sequencer table. The sequencer table stores the needed application profile information, and the MIS state machine uses this information in the sequencer table to bound each chunk's run-time, and to restart chunk execution from known points when needed.

To guide control flow, we introduce the concept of a *scope* (i.e. the "chunks" referred to in the previous paragraph). A scope is a region of code denoted by S_BEGIN and S_END markers (placed by the compiler). Each scope has an associated bound on the operation count. During normal operation, the operation count for the active scope is incremented on each retiring instruction. If the current operation count exceeds the scope's allowed bound, the MIS causes this scope to be exited, according to the scope recovery process discussed below. Scopes have some of the attributes of procedure call interfaces—known clean-slate starting points for regions of code—but without the actual stack changes or non-sequential PC values. Normally, scopes enclose straight-line code that the PC sequences through instruction by instruction; other nesting and function call cases are handled in the paragraph below. Their purpose is to delineate chunks of execution whose execution time can be bounded. This is used to constrain the impact of error-prone fabrics in causing major program derailments as noted in Section II's requirement 1.

Figure 3 illustrates an MIS that achieves the described functionality. The sequencer table consists of six columns for every scope. The *scope pc* and the *exit pc* are program counter values for the S_BEGIN and S_END instructions, i.e., markers for a scope. The *instruction limit* is the per-scope bound on instruction count. The current count of instructions executed in this scope thus far is stored in *instruction count*. The *parent scope* is used to pass control back to the parent scope when the active scope terminates, and to check for a legal child when a new scope begins, as discussed below. The final column manages information for the application call frame, which is used when an execution reset must occur.

When an S_BEGIN is encountered, the MIS first checks if its PC corresponds to an allowed child of the active scope, and if the active scope has enough instructions to execute the child scope. This scope check first uses the PC to perform a sequencer table lookup to find the child scope entry. If the active scope is indeed a parent, then the instruction limits are checked. If allowed to proceed, the "active scope" is updated to start tracking instruction count and other key information of the new scope. If the current PC is not a legal child scope, then control flow must have erroneously jumped due to errors, eventually encountering this incorrect S_BEGIN. For such cases, a scope recovery is executed as discussed below.

When an S_END is encountered, the MIS similarly checks if the current PC corresponds to the correct *exit pc* of the active scope. If it does, then the active scope is updated to be the parent scope, and execution continues. As part of this scope transfer, the parent scope's instruction counter is updated and the child scope's is zeroed out. One can either update by the amount of instructions actually executed by the child scope (i.e. the counter value) or by the child's limit. These options have subtle tradeoffs in analyzability; for this paper's results, we use the limit value. If the scope check fails, the MIS executes a scope recovery as discussed below.

The final possible event is instruction retirement. Here it simply checks if the active scope has any instructions left in its limit. If so, it retires the instruction and increments. If not, it cancels the retirement and performs scope recovery. While naively this check adds latency on retiring instructions, we hide this latency by batch retirements. Alternatively, the check can be performed while instructions wait in the reorder buffer.

Profiling for Instruction Count Bounds: Our approach rests on having reasonable instruction count bounds for each scope. For streaming applications, this is fairly tractable because the control flow includes few dynamic conditions and the longest execution paths are easily seen at compile-time. When programs have high variability or even in some cases no finite overall bound (e.g. infinitely running data processing) the application or compiler can use blocking transformations to place bounds on groups of loop iterations without bounding the whole. We use static profiling to obtain useful bounds for our benchmarks, but many other dynamic or adaptive techniques are possible and there is prior work to draw from [16].

Scope Recovery Process: When scope recovery is needed, the MIS updates the PC to be the *exit pc* of the active scope. In the case of infinite loops, control may have remained within this scope, but exceeded the allowed instruction count. Forcing execution to the *exit pc* breaks this loop. In the case of bit errors that cause misdirected jumps, control may have transferred to an incorrect scope. In these cases, we "reset" execution by going to the *exit pc* and resuming from there. This may entail some number of incorrect instructions being executed, but again the goal is to reduce hardware overhead and extend generality by constraining the extent and side effects of such behavior, rather than disallowing it entirely.

Nesting, Function Calls and Other Scope Issues: Scope annotations in the program should be cleanly nested, meaning that scope regions can only intersect if they have a parent-child relationship. Further, S_BEGINs should dominate the closing S_ENDs and the S_ENDs should post-dominate these S_BEGINS (e.g., a loop can contain a scope and/or can be contained in a scope but not cross one). Function calls should be fully contained by S_BEGIN and S_END statements at matching scope depth. Since recovery from a scope violation involves jumping to the current scope's *exit pc*, further information is needed for scopes that include a function call. For these, we record the stack and base pointers (last column of the sequencer table) so that when the recovery routine jumps to the scope's *exit pc*, it also resets the call frame. The frame information is statically known if the function's call depth is statically known (as in our applications and many others), but one could also record the frame information dynamically at scope start, for possible use at scope recovery. Recursive calls should be enclosed at the outermost caller location and the recursive functions should not contain scopes in them.

Hardware Overheads: The MIS manages only one scope at a time to keep the hardware overhead low. The most common operation, bounds checking and increment, only requires one comparator and one adder. The S_END and S_BEGIN events are less frequent, and only require a lookup to the sequencer table, a condition check, and possibly an addition. At S_ENDs, a lookup in the sequencer table is necessary for the parent scope, whose index is stored with the active scope entry. For S_BEGIN *current pc* is used to get the information for the new scope and check if it is a valid child. Identifying a scope by the *pc* of its S_BEGIN instruction makes it possible to avoid storing the children of a scope in the sequencer table.

The sequencer table can be implemented either as a single table, or as a combination of the full table and a cache for some of the entries. For StreamIt programs, scope counts vary from 12 for jpegdecoder to 124 for mp3decoder; this number determines the size of the full table. If a cache is used instead, then three cached entries suffice, i.e., the parent scope, the current scope and the child scope. Having these entries ready

in the cache requires a prefetcher and since the scope tree is trivial for StreamIt applications (main scope with sequential children), implementation of the prefetcher would be trivial.

B. Memory Fence Unit

In addition to control errors, our four fundamental design requirements also require us to mitigate the effects of memory access errors. For full general usage, such access errors can cause segmentation faults which crash the program. Even in constrained no-Operating-System scenarios considered here, such errors can influence output accuracy. To mitigate error effects, we propose a *memory fencing unit (MFU)*. The MFU checks accesses against the legal address range allowed for a particular scope or instruction. The MFU can be implemented similarly to earlier segmentation schemes [2], [3] with the addition of designated error handlers for different error conditions. In this work, we partition memory to be execute&read and read&write regions, although further programming language support can be used to obtain finer grain ranges [5].

If a memory access is out-of-range, the MFU's response depends on what type of access it was. For read or write accesses (i.e. data references) we silence the fault either by skipping this memory instruction or by referencing a dummy physical location instead. While perhaps surprising, this response is simple to implement and abides by the four requirements in Section II—our goal is simply to prevent memory accesses to disallowed ranges. The third memory access type is *execute* which applies to fetches intended for instruction memory. *Execute failures* are illegal instruction fetches where the program counter points to a disallowed region of memory: either this region does not contain program code or perhaps the PC is pointing outside the application's allowed memory range entirely. Simply silencing the current instruction (as we do for reads and writes) does not work for execute failures, since advancing the program counter typically leads to yet another illegal execute access. Instead, for *execute failures*, the MFU signals the MIS to end the current scope and begin scope recovery from a known point in the code. For example, this would be the next filter in StreamIt applications. (Other exceptions are handled similarly.)

C. Streamed I/O Constraints

Finally, requirement 3 from Section II calls for I/O constraints. Real-world applications must read and/or write data from/to external devices; prior work has not substantively covered this issue. Even if the *use* of data may be error tolerant, its transfer should not cause crashes or corrupt the file system. To achieve this, we use streamed I/O which only performs fixed-sized, streamed read and write operations, and we limit the number of I/O operations allowed per file or scope. Constraining I/O to bounded sequential access gives an error-prone processor access to data, but not to arbitrary addresses or file system data structures. This approach works well for StreamIt and similar benchmarks. Future work can explore additional variants.

IV. Experimental Methodology

A. Simulation Infrastructure

To study how errors percolate from hardware through ISA to the application, we built a simulation infrastructure based on the detailed Virtutech Simics functional architecture simulator [11]. Our baseline system is the 32-bit Intel x86 architecture.

The MIS is simulated as a snooping device that observes the retiring instructions. It implements the scope table and the scope FSM shown in Figure 3. The MFU is implemented as a modified TLB module. Based on the access address and request type, it checks if the application has the required access permission for that memory region. For read/write access violations, this module returns a physical dummy location. For execute access failures, it instructs the MIS to initiate a scope termination and recovery. Finally, we simulate Streamed I/O by transforming the I/O calls in the StreamIt applications to certain x86 instructions that our simulator uses as markers for initiating emulated streamed I/O accesses.

Error Injection The simulator models hardware errors by flipping bits in the register file. Error events occur randomly following a uniform distribution for a given mean time between errors (MTBE). When an error occurs, the simulator randomly selects a bit in a randomly-selected register and flips it. The random bit-flips in random registers model the architectural visibility of underlying hardware faults [12]. The 32-bit x86 architecture has a small number of general purpose registers. Arithmetic registers are used for complex operations that use a larger state space compared to the operations on ESP, EBP and EIP registers, thus they should experience more errors. As a result, this module injects errors directly to the following six registers: E[A-D]X, ESI, and EDI. Other state, including registers like ESP, EBP or EIP, can become corrupted transitively. For example, since these registers are written to/from the stack at procedure calls, they are corrupted via memory addressing errors. Thus, our protection and system recovery mechanisms apply to ESP, EBP and EIP errors also.

B. Benchmarks

Our experiments use seven benchmarks from the StreamIt benchmark suite [15]. These are either multimedia processing applications or kernels for such applications. These benchmarks were primarily selected due to the suitability of multimedia applications for error-tolerant computation. In this work, we insert the S_BEGIN and S_END instructions around the location of each StreamIt filter function call. After profiling the applications in error-free runs to determine the scope execution bounds and other static scope information, we run them with our modules activated for error-prone operation.

The StreamIt compiler produces corresponding C++ code. An open-source MP3 encoder/decoder library (http://lame.sourceforge.net/) compresses a recorded signal and decodes it to the StreamIt C++ back-end's preferred format. The StreamIt java implementations provide the JPEG encoder/decoder; we use these implementations to encode a raw image and decode it to the preferred back-end format. We run the benchmarks with varying MTBEs to see how the corresponding application output changes and which error types are prominent for the acceptable ranges of the output quality. For each MTBE, we do 10 runs using different seeds for the random number generator of the error injector. The simulation runs to measure end-to-end error impact on program output use full length of application runs on the simulated machine, while other error characterizations use a truncated version (the first 50ms measured on the machine) due to long simulation times.

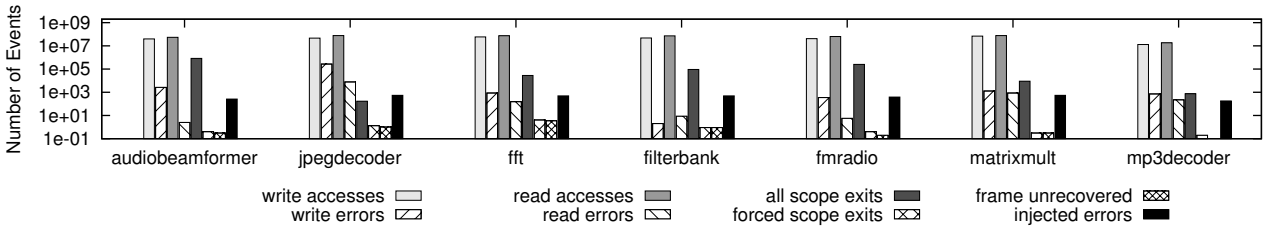
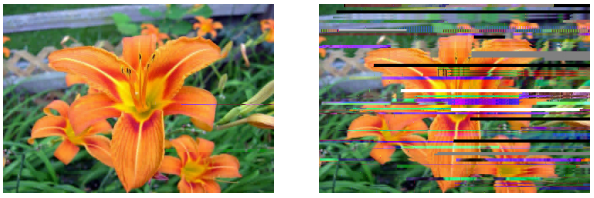


Fig. 4. At an MTBE of 256k instructions, the most common error types are memory address errors (read and write accesses to non-permitted memory regions) and, in some benchmarks, forced scope exits (due to execute access errors, processor exceptions, or exceeding an instruction count limit). Shaded bars show total event counts (e.g. memory accesses) and patterned bars indicate the number of events experiencing an error. The last bar per application shows the average number of bit flips injected to the architectural registers.



(a) 2048k instrs (18dB) (b) 32k instrs (7dB)

Fig. 5. Image outputs from the JPEG decoder benchmark at different MTBEs from the same seed for the random number generator.

V. Experimental Results

Here, we first characterize catastrophic errors in terms of their frequency and execution impact. Second, we study the efficacy of the proposed protection mechanisms on application output quality for JPEG and MP3.

A. Characterization of Catastrophic Errors

Prominent Error Types Handled By Protection Modules: Figure 4 shows how often different protection handlers are invoked for each type of error. An MTBE of 256k instrs is high enough to maintain acceptable output quality and low enough to analyze the effects of errors (see Section V-B). The figure shows that memory write and read failures are the most common failures experienced—up to an order of magnitude more frequent than architectural bit flips themselves (Consider for example if a bit flip occurs in a pointer used for several memory accesses). Write failures are more frequent than read failures, because an application usually has read permissions for address ranges that it can write to, but the converse is not true. For applications with more complicated control flow, forced scope exits are also prominent.

Effects of Illegal Instruction Fetch: As Section III discusses, an execute failure occurs due to an illegal instruction fetch. Without the MFU, trying to execute the corresponding memory location would likely fail and the next memory location would be illegal too. This would effectively be a program crash and Figure 6(a) shows results related to this issue. Only two of our applications ever experience execute access failures. The failures for *fft* start for MTBE $\approx 10^6$ instrs and for *mp3decoder* they start for MTBE $\approx 10^5$ instrs. In our system, the MFU detects such accesses and the MIS initiates scope recovery. Even though this may cause data errors for the computation on the current block of data, the streaming application can use scope recovery to withstand this error and continue execution with useful results (see Section V-B).

Errors Causing Application Hangs: While the MFU could enable some error-tolerant operation, it would be insufficient without the MIS. In particular, without the MIS's ability to keep control flow on track, errors not only affect program and address values, but they also cause the programs to hang and fail to reach the end of the computation. Figure 6(b) shows that *fft*, *jpegdecoder* and *mp3decoder* would hang (run-time higher than 20x) at an MTBE as infrequent as $\approx 10^6$ instrs. As Section V-B shows, our modules enable computation with acceptable outputs for even more frequent errors executing strictly fewer instructions than the given limit.

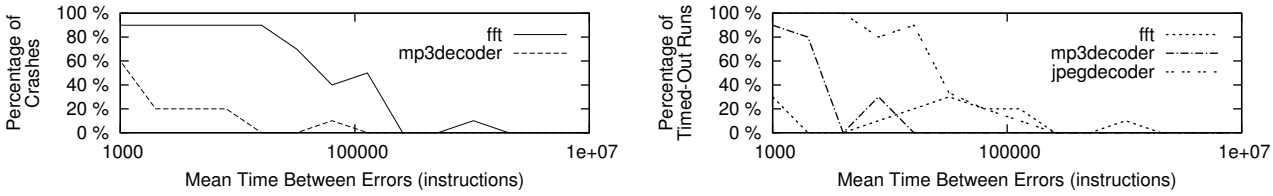
The MIS guarantees that the scopes do not exceed their instruction limits, however it cannot eliminate all performance overheads that may be caused by errors. For example, an error-free run may exercise shorter paths of the control flow, whereas an error-prone run may erroneously choose longer paths, hence causing a performance overhead. StreamIt applications do not exhibit this behavior in our experiments. In contrast, for high error rates, applications tend to exit loops early, causing the application to complete faster but with degraded output quality. This degradation is reflected in our results.

B. Effects of Errors on Application-level Quality Metrics

Our second set of experiments assess how low-level hardware errors affect application-level quality metrics with the protection modules in place. We focus on two important applications due to their wide adoption and ability to tolerate errors. JPEG is a widely used lossy image compression standard and MPEG-2 Audio Layer III (MP3) is a widely used lossy audio compression standard. For a given raw signal X, they define a compression algorithm to produce a smaller file Y and also a decompression algorithm to reproduce the output signal Z. However, due to information loss in the compression stage, the output Z is not the same as input X. SNR [14] is a common metric to quantify this difference. We use SNR to quantify the change lossy compression introduces, even assuming error-free hardware. Next, we run the decompression stage through our simulator and calculate the SNR_e of the output from error-prone hardware, Z_e. Comparing SNR with SNR_e provides a useful metric for the quality of the output of the error-prone run for a given MTBE. As before, we perform 10 runs to capture the statistical variation of SNR_e across runs.

Figure 7(a) shows JPEG benchmark results. With very frequent errors, SNR_e is close to 0dB, meaning that output error is as prominent as the signal itself. As errors become less frequent, however, SNR_e improves and reaches SNR.

Figure 5 presents images for two SNR_e values. With an MTBE of 2048k instrs on average, there is little visible error. In fact, SNR_e matches SNR even though the protection mechanism has actually handled 19k memory errors. When the MTBE reaches 32k the image is visibly corrupted, but still quite recognizable. At this point, the program has with-



(a) MFU disabled to characterize execute failures

(b) MIS disabled to characterize hangs

Fig. 6. (a) Applications experience hangs (infinite looping) with the MTBE as high as every $\approx 10^6$ instrs. Hangs are more common with increasing error rates. (b) Without the MFU, we define an illegal instruction fetch as a crash. *fft* and *mp3decoder* experience crashes, particularly for MTBEs of 128K instrs or less.

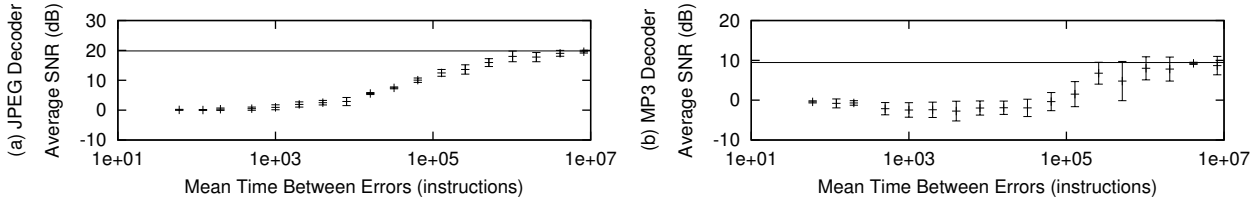


Fig. 7. Output quality of the (a) JPEG and (b) MP3 decoder algorithms running at different MTBEs. The flat line in each graph corresponds to the output quality when the decoder is run without any errors.

stood $\approx 10^6$ memory errors, 10 forced scope exits due to instruction limits, 1 illegal instruction fetch, and 2 processor exceptions. Note that due to the streaming nature of these applications the errors show up as lines in the image. Since a StreamIt application works on a block of data per iteration and crucial variables such as buffer pointers are re-initialized every iteration, a corruption burst is cleared in the following iteration. Interestingly, we did not alter the application to have this behavior; this "partial restore" of buffer indices is natural to StreamIt. These experiments highlight how our lightweight hardware additions enable programs to withstand and recover from error, in order to produce useful results.

Figure 7(b) for MP3 decoder shows similar trends. However, in contrast with JPEG, here the SNR can go to negative values. This is because of data representation. JPEG application efficiently uses the 8-bit each of Red-Green-Blue per output pixel, so even the highest error power is comparable to the original signal power. However, MP3 represents the audio output signal as pulse code modulated, so the bit utilization depends on sound volume. Since an error can make arbitrary output signal changes, our output can have higher power than the original, resulting in negative SNR. More frequent errors can improve the SNR value (to zero) because with lower MTBEs, the application produces a zero output signal, silence is better than noise. The reader can listen to different sounds for different error rates here: http://youtu.be/2XhZxbNz7Lk.

VI. Conclusions

This work has explored the design and utilization of future programmable processors that may be error-prone due to hardware faults expected in future technology generations. The abstraction of an error-free hardware-software interface comes at considerable area/energy cost; it important to consider cases where errors may cross this interface. We classify possible catastrophic errors as control flow errors, memory errors and I/O errors. We then propose and evaluate a microarchitecture with low-overhead protection mechanisms that mitigate error effects so execution can proceed.

Our experimental results characterize the frequency and type of catastrophic errors and the efficacy of the proposed protection mechanisms. Our output quality analysis on JPEG and MP3 shows that for MTBE less than $\approx 10^7$ instructions, the output SNR is on par with the quality effects seen by mildly-lossy compression. The output quality is 14dB and 7dB respectively if the mean is 256k instructions, and the example visual and aural datasets provide further subjective support. Overall, this study is the first to demonstrate that error-tolerant applications can be deployed on error-prone processors using relatively simple protection mechanisms.

References

[1] M. Carbin et al. Detecting and escaping infinite loops with Jolt. In *ECOOP*, 2011.
[2] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM*, 11(5):306–312, 1968.
[3] J. B. Dennis. Segmentation and the design of multiprogrammed computer systems. *J. ACM*, 12(4):589–602, 1965.
[4] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*, 2003.
[5] C. S. Gordon et al. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, 2012.
[6] R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *ISLPED*, 1999.
[7] ITRS. ITRS process integration, devices, and structures, 2011.
[8] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
[9] L. Leem et al. Error resilient system architecture (ERSA) for probabilistic applications. In *DATE*, 2010.
[10] S. Liu et al. Flikker: saving dram refresh-power through critical data partitioning. In *ASPLOS*, 2011.
[11] P. S. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35(2), 2002.
[12] S. S. Mukherjee et al. Measuring architectural vulnerability factors. IEEE Computer Society, 2003.
[13] A. Sampson et al. EnerJ: approximate data types for safe and general low-power computation. In *PLDI*, 2011.
[14] T. Stathaki. *Image Fusion: Algorithms and Applications*. Academic Press, 2008.
[15] W. Thies et al. StreamIt: A language for streaming applications. In *ICCC*, 2002.
[16] R. Wilhelm et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.