PPU: A Control Error-Tolerant Processor for Streaming Applications with Formal Guarantees

PAREESA AMENEH GOLNARI, YAVUZ YETIM, MARGARET MARTONOSI, YAKIR VIZEL, and SHARAD MALIK, Princeton University

With increasing technology scaling and design complexity there are increasing threats from device and circuit failures. This is expected to worsen with post-CMOS devices. Current error-resilient solutions ensure reliability of circuits through protection mechanisms such as redundancy, error correction, and recovery. However, the costs of these solutions may be high, rendering them impractical. In contrast, error-tolerant solutions allow errors in the computation and are positioned to be suitable for error-tolerant applications such as media applications. For such programmable error-tolerant processors, the Instruction-Set-Architecture (ISA) no longer serves as a specification since it is acceptable for the processor to allow for errors during the execution of instructions. In this work, we address this specification gap by defining the basic requirements needed for an error-tolerant processor to provide acceptable results. Furthermore, we formally define properties that capture these requirements. Based on this, we propose the Partially Protected Uniprocessor (PPU), an error-tolerant processor that aims to meet these requirements with low-cost microarchitectural support. These protection mechanisms convert potentially fatal control errors to potentially tolerable data errors instead of ensuring instruction-level or byte-level correctness. The protection mechanisms in PPU protect the system against crashes, unresponsiveness, and external device corruption. In addition, they also provide support for achieving acceptable result quality. Additionally, we provide a methodology that formally proves the specification properties on PPU using model checking. This methodology uses models for the hardware and software that are integrated with the fault and recovery models. Finally, we experimentally demonstrate the results of model checking and the application-level quality of results for PPU.

CCS Concepts: • **Hardware** → **Robustness**; **Fault tolerance**; **Error detection and error correction**; **Failure prediction**; **Failure recovery, maintenance and self-repair**; **System-level fault tolerance**; Hardware validation; Model checking; • **General and reference** → *Cross-computing tools and techniques*; *Reliability*

Additional Key Words and Phrases: Error-tolerant computing, streaming applications, reliability requirements, progress, control flow, verification

ACM Reference Format:
Pareesa Ameneh Golnari, Yavuz Yetim, Margaret Martonosi, Yakir Vizel, and Sharad Malik. 2016. PPU: A control error-tolerant processor for streaming applications with formal guarantees. J. Emerg. Technol. Comput. Syst. 13, 3, Article 43 (April 2016), 29 pages.
DOI: http://dx.doi.org/10.1145/2990502

This work was supported in part by Systems on Nanoscale Information fabriCs (SONIC) and Center for Future Architectures Research (C-FAR), two of the six SRC STARnet Centers, sponsored by MARCO and DARPA. The initial work on PPU was supported by the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program, a SRC program. In addition, this work was supported in part by National Science Foundation.
Authors' addresses: P. A. Golnari, Y. Vizel, and S. Malik, Electrical Engineering Department, Princeton University, Princeton, NJ, US; emails: pgolnari@gmail.com, {yvizel, sharad}@princeton.edu; Y. Yetim, (Current address) Google, 345 Spear St, San Francisco, CA 94105; email: yyetim@gmail.com; M. Martonosi, Computer Science Department, Princeton University, Princeton, NJ, US; email: mrm@princeton.edu.

DOI: http://dx.doi.org/10.1145/2990502

1. INTRODUCTION

As transistor sizing approaches its limits, semiconductor fabrics are expected to experience increasing device failures [ITRS 2011]. Hardware failure arise from various sources such as energized particle hits, increasing variability of device parameters, and device aging [Borkar 2005]. There has been significant recent work in computer architecture attempting to guard future processors against this host of resiliency threats. Generally, these processors deal with transient faults. The effect of these faults is that the underlying processor may crash or the program executes incorrectly, resulting in data or control flow errors.

Fully protecting processors against hardware threats is generally viewed as being too expensive [Narayanan et al. 2010]. Instead, architectural solutions seek to allow certain errors and then fully/partially correct them using lower cost architectural correction mechanisms. In the latter category are error-tolerant architectures such as ERSA [Cho et al. 2012], EnerJ [Sampson et al. 2011], and Argus [Meixner et al. 2007]. These processors are *error-tolerant* in that they allow errors in the architectural registers, in contrast to *error-resilient* processors, which correct all errors before they change the architectural registers [Austin 1999; Ernst et al. 2003].

An important consequence of the error-tolerant approach is that it is no longer guaranteed that the processor executes each instruction as per the ISA. Rather it provides a best-effort execution in the presence of faults while avoiding fatal errors. Thus, the ISA can no longer be used as a formal specification for error-tolerant processors. Rely is a programming language that permits quantitative reasoning about the results for an application implemented on such processors [Carbin et al. 2013]. However, in the absence of a specification, it is unclear what the requirements are for such processors or what is actually delivered in the final implementation.

In this article, we address this gap by first defining formal specifications that capture the necessary, but not sufficient, requirements on such error-tolerant processors to produce useful output (which is application-specific, user-acceptable output). These serve as a lower bound for what they need to implement.

Based on these requirements, we propose a Partially Protected Uniprocessor (PPU) that provides protection schemes to avoid fatal errors and allow the processor to deliver useful results [Yetim et al. 2013]. This processor includes components that guide control flow, memory addressing, and I/O accesses with customizable granularities to handle errors. In this work, we consider transient faults in the logic and memory. The effect of these faults is bit flips in the architectural state. The errors can propagate to the Program Counter (PC) and the Stack Pointer (SP).

We evaluate the quality of results of PPU by showing the results of different error rates for seven Streamit [Thies et al. 2002] 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.

Additionally, we show how reliability specifications can be checked on PPU using model checking. In the system model, we consider both the hardware (HW) and software (SW) models of PPU, as well as the effect of faults and PPU protection mechanisms. In these experiments, the formal specifications are provided using temporal logic [Clarke et al. 1999]. In each case that PPU fails to satisfy a property, the model checker provides a fatal scenario which is not corrected by the processor. This demonstrates the utility of such a methodology to verify future error-tolerant architectures to ensure that they meet at least the minimum specifications defined in this article.

Overall, this work makes the following contributions: (i) It identifies the basic required properties for error-tolerant processors and formally defines them using

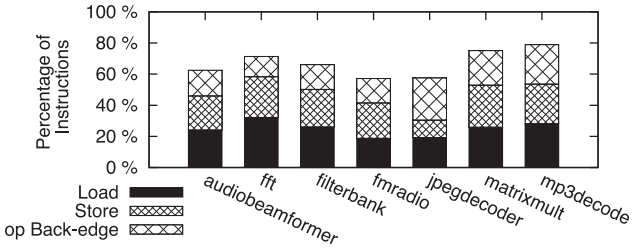


Fig. 1. Percentage of error-free instructions needed to avoid (i) memory accesses that may cause crashes and (ii) control flow operations that change looping behavior and leads to hangs.

temporal logic. (ii) It proposes PPU as a solution to ensure software progress through errors without assuming a fully reliable core. (iii) Even though errors quickly become program-critical under even modest error rates, PPU shows that control flow and memory addressing do not have to be perfectly reliable to get useful results. (iv) It shows how the error-tolerant processor, the faults, and the program are modeled as a complete system. (v) It formally verifies these properties on PPU and identifies aspects of the implementation which result in those properties failing.

2. CONTROL ERROR TOLERANCE

While applications may have some inherent error tolerance, this does not translate uniformly to instruction-level error tolerance. Errors in critical 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 [Lattner and Adve 2004], 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. The predominance of error-intolerant instructions shows that language support to separate the error-tolerant parts of the program from the error-intolerant ones is not enough. This motivates our work to protect processor control flow to improve application success rates on error-prone fabrics.

Thus far, error-tolerant processors provide only a weak best-effort guarantee on the quality of the results. In practice, the result quality is expected to be acceptable in terms of human perception. In order to accomplish this, PPU emphasizes the importance of avoiding fatal control errors and making progress, and also of limiting the effect of data errors such that their effects are short-lived or ephemeral [Yetim et al. 2015]. We now sharpen these goals by developing a minimum formal specification for such processors.

3. BASIC DESIRED RELIABILITY REQUIREMENTS

The preceding discussion provides some guidance on acceptable and unacceptable control and data errors.

Control Errors: We need to avoid fatal errors such as a program hang or a crash because this will preclude the processor from delivering any results into the future.

Thus, permanent control errors such as crashes and hangs are unacceptable. Transient control errors that result in control flow that executes less or more than the correct code may be acceptable.

Data Errors: The processor needs to provide useful results, so any errors that result in the output quality being useless need to be avoided. Note that this requirement on data errors is relatively weak because it does not assure that all results will be useful. However, it is nonetheless valuable because it precludes data errors that will result in the output quality being useless. We now see what this space of unacceptable and acceptable errors implies in terms of the basic desired properties. We briefly state what these properties capture, and, in Section 6, we provide their formal language specifications.

3.1. Progress

This property arises from the need for the processor to avoid fatal control errors such as hangs. As discussed, transient control errors may be acceptable as long as the processor continues to provide useful results into the future. This requirement needs to be captured in some notion of progress, where, at each point in time, even in the presence of faults, the processor is guaranteed to provide some useful results in the future.

3.2. Ephemeral Effect of Errors

Although it is hard to state in general what it means for the result quality to be acceptable for human perception, we note that there is one important application-level characteristic that these processors are exploiting. If the faults result in transient errors in the output, then they only impact some of the output, which can potentially be overlooked by human perception (e.g., an erroneous pixel or even a frame). Once the transient error has passed, the subsequent results have the potential to be error-free. Thus, the desired property here is that of the errors being transient (i.e., ephemeral) [Yetim et al. 2015]. For the errors to be ephemeral, there should always be a time in the future when the system state needed to compute future results can be error-free (i.e., all errors in such a system state are erased).

3.3. Executing Essential States

Because error-tolerant processors permit control errors to skip states and do less or more computation than in the error-free case, we need to assure that useful results can still be produced. This requires that control states essential to producing results are visited. For instance, assuming that the result of the computations in a program are written to the output in the k^{th} basic block, the purpose of the program is not fulfilled if basic block k is not executed. A model that satisfies this requirement makes sure that the k^{th} block is executed at some point during execution.

This set of properties does not guarantee that the processor will produce useful results. While not sufficient, they are necessary. However, if they are not satisfied, the processor does fail in its requirement of providing useful results. Thus, they serve as a lower bound that may be augmented with additional stronger properties for individual processors. In the following, we propose a set of control flow protection mechanisms that provide reliability to a general-purpose processor. We then evaluate the effectiveness of these mechanisms in terms of quality of results and providing the basic reliability requirements.

4. PPU DESIGN

Figure 2 shows the key components of our proposed approach, with the five reliable modules in dashed boxes. These modules are built on reliable hardware (error-free)

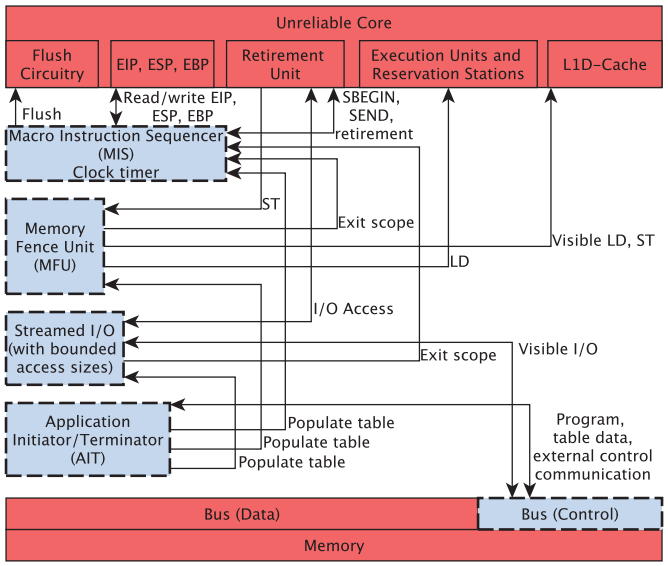


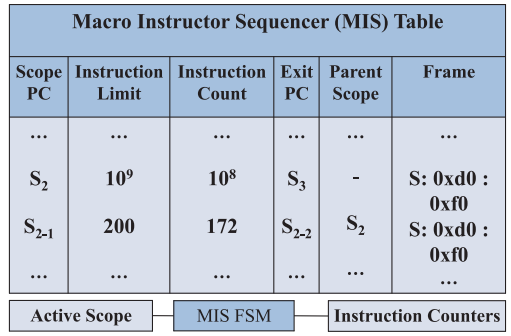
Fig. 2. Our low-overhead support for an application to eliminate crashes, hangs, and device corruptions includes five reliable components (dashed outline, only conceptually separated from the core). The *Macro Instruction Sequencer (MIS)* with a timer ensures forward progress. The *Memory Fence Unit (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 remaining system (solid outline) can be unreliable with best-effort operation.

and supervise the control flow. They provide control-error protections to mitigate fatal error effects. These supervising modules have the following goals:

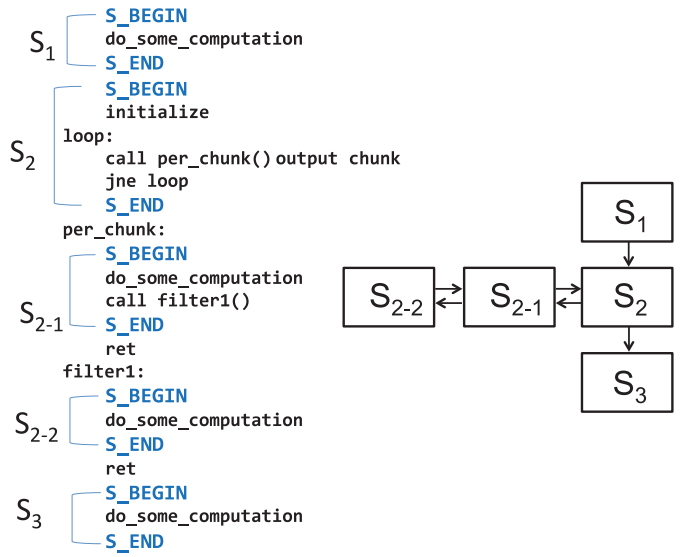
(1) The program should not hang or run indefinitely. Control flow errors that result in infinite loops or other failures to terminate must be addressed.
(2) The application should only be allowed to access information that it is allowed to. Memory addressing errors that cause the program to access off-limits areas must be handled.
(3) 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.
(4) 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 [Stathaki 2008].

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 preceding four goals are met. The first three goals guarantee *progress*, which is the first requirement to produce any result. The fourth goal corresponds to satisfying ephemeral effects of errors and executing essential states, which are required (but not sufficient) for providing results with acceptable accuracy.

To achieve these four goals, three bounds modules—*MIS*, *MFU,* and the *streamed I/O* in Figure 2—are designed to 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) MIS.



(b) An example of PPU target applications with three parent scopes and two child scopes.

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.

4.1. Macro Instruction Sequencer

Our implementation examples for program analyses are based on StreamIt [Thies et al. 2002], a programming language and a compiler for streaming applications. Central to the Macro Instruction Sequencer (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. The MIS has the primary responsibility of a control checker for constraining the control flow behavior of the system.

Figure 3(a) shows the MIS implementation, primarily as 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 runtime and to restart chunk execution from known points when needed.

Scope: 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). 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 nonsequential PC values. Normally, scopes enclose straight-line code that the PC sequences through instruction by instruction; however, unlike basic blocks, scopes can contain one or multiple loops. The loops boundaries cannot cross the scopes, and a scope cannot be enclosed in a loop.

Moreover, the nesting and function call cases are handled by nested scopes. Here, we refer to those scopes that are nested inside other scopes as "child" scopes. And "parent" scopes are those scopes that are not a child of any other scope. While a "child" scope can have other scopes nested inside it, a parent scope cannot be nested inside another scope.

The purpose of scopes 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 to address the mentioned four design requirements. 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 later). Figure 3 gives an example for a part of a program with three parent scopes and two child scopes nested to the parent scope S_2. The CFG is depicted in this figure, where each state represents a scope.

Figure 3(a) 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 later. 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 reaches zero. 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 article's results, we use the limit value. If the scope check fails, the MIS executes a scope recovery, as discussed later.

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 Wilhelm et al. [2008].

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. Furthermore, 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 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, implementation of the prefetcher would be trivial.

4.2. Memory Fence Unit

In addition to control errors, our four design goals also require us to mitigate the effects of memory access errors. For full general usage, such access errors can cause segmentation faults that crash the program. Even in the 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 [Daley and Dennis 1968; Dennis 1965] 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 grained ranges [Gordon et al. 2012].

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 design goals—our aim 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.)

4.3. Streamed I/O Constraints

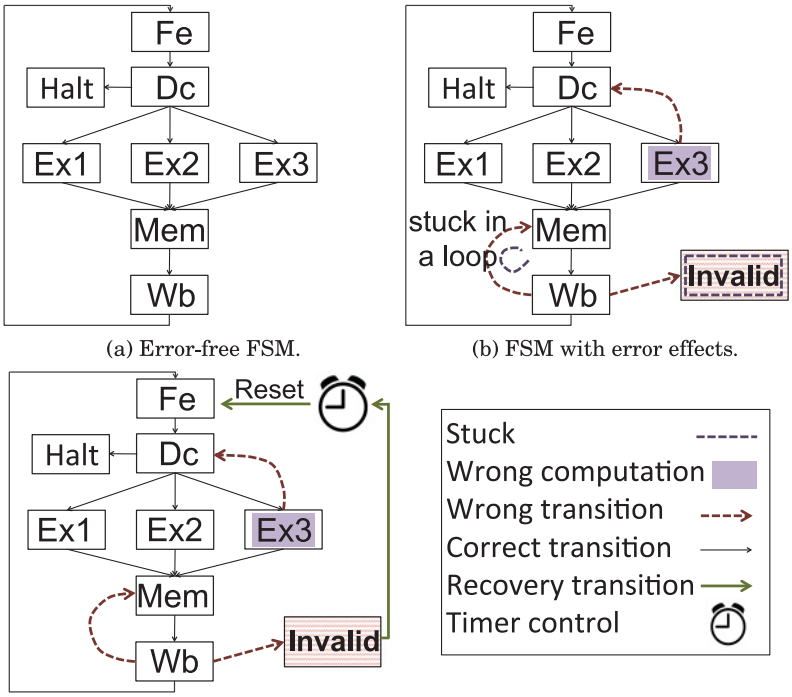
Finally, design goal 3 calls for I/O constraints. Real-world applications must read and/or write data from/to external devices. Even if the use of data may be error-tolerant, its transfer should not cause crashes or corrupt the file system. There are possible solutions to achieve this. For example, Lax is a device driver allowing for a tolerable amount of error in I/O [Stanley-Marbell and Rinard 2015], or a periodic check can detect and repair file system errors [Yang et al. 2006]. Also, a journaling file system can repair erroneous file operations before they become permanent [Prabhakaran et al. 2005].

In this work, 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. We assume that size and number of I/O operations for a given application are known. By constraining I/O to bounded sequential access, we achieve 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.

To evaluate the reliability of the PPU design, we model the control flow of a processor augmented with the discussed supervising modules as follows.

5. CONTROL FLOW MODELS AT HW AND SW LEVELS

Consider the case where a program gets in an infinite loop due to a software-level fault. The system as a whole does not make any progress since the program hangs due to this fatal control error. Note that, in this scenario, the processor hardware does make progress because it keeps on fetching instructions. This simple example shows that, in order to satisfy a reliability requirement such as progress, both hardware and software



(a) Error-free FSM.

(b) FSM with error effects.

(c) FSM with errors and protection mechanisms.

Fig. 4. Hardware-level FSM models.

need to be taken into account because together they form a system. Therefore, in our setting, we consider the hardware and software components as a whole system, and we define the requirements with respect to that system. We model the hardware as a Finite State Machine (FSM) and software as a Control Flow Graph (CFG).

In addition to modeling the HW/SW function, we also model the effect of faults and the mechanisms to protect against these faults at each level. Recall that the focus of PPU and other error-tolerant architectures is on transient faults and protecting against these faults. Thus, our modeling also retains this focus.

5.1. Modeling the Hardware

The PPU protection mechanisms are implemented on a general-purpose error-prone processor. Here, we assume M_{hw} (discussed later) as our baseline hardware model of the processor. M_{hw} considers a pipeline design of a single-issue, in-order MIPS architecture [Patterson and Hennessy 2012]. Figure 4(a) shows an example of an FSM for this architecture, where each instruction is fetched (Fe), decoded (Dc), executed (Ex_i), accesses the memory (Mem), and writes back to the register bank (Wb).

5.1.1. Effect of Transient Faults. In the FSM model, a fault is modeled as additional transitions. These transitions are nondeterministic; thus, they may or may not be taken in any specific execution, capturing the transient nature of the fault. A transient fault in the hardware can have either a transient or permanent effect. A fault is classified as having a permanent effect when a processor ends up in an unwanted control state for the rest of the program execution (e.g., when the FSM stays in an incorrect subset of states forever, causing the program to either crash or hang). On the other hand, a fault is classified as having a transient effect when it does not cause a fatal behavior (like

a hang or crash). Figure 4(b) shows examples of the effect of transient faults on the FSM. Shaded states and dashed transitions represent invalid states and transitions, respectively. To keep the figure readable, we do not show all the incorrect transitions. As an example, consider a fault resulting in the addition of an invalid transition from the state Dc to Ex3. This addition has a transient effect because program execution continues after Ex3. Note, however, that the erroneous state transition may have resulted in incorrect data computation inside state Ex3. On the other hand, the result of a fault with a permanent effect is an addition of an invalid state with transitions that lead to it (e.g. transitions to the invalid state in the figure).

5.1.2. Protection Mechanisms. Error-tolerant processors provide micro-architectural mechanisms to protect against hardware faults. Since our aim is to see if these mechanisms are sufficient to provide certain guarantees, we need to add these to our FSM, which thus far has captured the intended function and the fault effects. As we saw, a primary requirement is to avoid fatal errors (e.g., getting to an invalid state with no way to recover). In practice, this is often accomplished using timeout mechanisms. As an example, consider assigning a *timer* (referred to as *fetch timer*) to the Fe state in our model. This timer provides a transition to the Fe state on its timeout and is reset on entering the Fe state. This simple protection mechanism enables us to avoid the fatal errors of getting stuck in a valid/invalid state or a loop (Figure 4(c)). We assume that these protection mechanisms have resilient implementations and are fault-free.

5.2. Modeling the Software

As mentioned earlier, based on the StreamIt language, PPU divides a program into *scopes*, where a scope is a set of instructions with the *start* and *end* of scope forming a boundary. Figure 3 shows a CFG for part of a program with three parent scopes. This CFG depicts many features of PPU target applications. But since our goal is to verify reliability properties for *any* program that matches PPU, we now describe a general CFG that follows PPU's target application modeling.

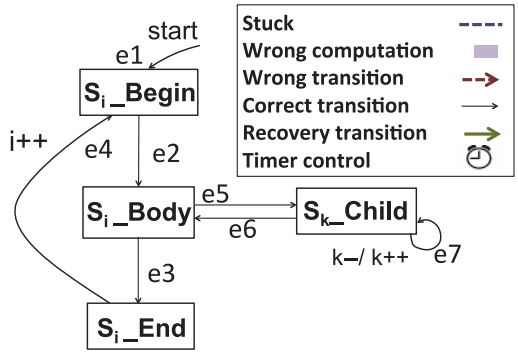
A template CFG model for PPU's target applications is shown in Figure 5(a). It represents error-free programs with a variable number of scopes. The states S_i_Begin, S_i_Body, and S_i_End represent the beginning of a scope, its body, and its exit point, respectively. If a scope S_i has child scopes (nested calls), the children are represented by a transition from S_i_Body to S_k_Child. When a child scope is started, it becomes the active scope. When the nested call is finished, its parent scope becomes the active scope. Note that since PPU does not allow back-edges or split-joint structures between scopes, they are not allowed by our model.

While this template represents many different CFGs, for simplicity, we refer to it as the *CFG*. Table I depicts the purpose of the transitions in this CFG. For example, the first row explains edge e_4, which represents an instantiation of a new scope.

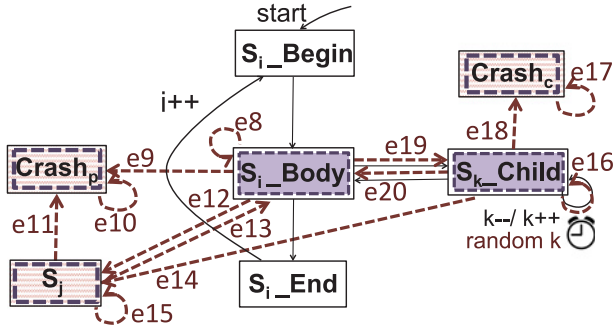
5.2.1. Effect of Errors. The effect of errors on the CFG are shown in Figure 5(b). Erroneous states and transitions are shown with dashed lines, and their descriptions appear in Table I. Note that S_i_Begin and S_i_End can have no errors since they are only marking the scope boundary and do not contain any executed code, which is limited to the scope body.

The two distinct crash states ($Crash_p$ and $Crash_c$) are added to the error-prone CFG because, when a crash happens, the recovery mechanisms depend on whether the active scope is a child scope or a parent scope.

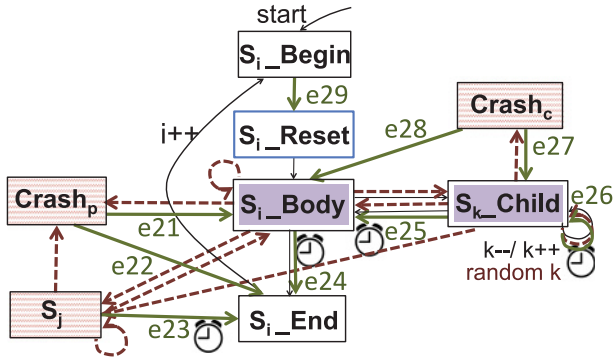
5.2.2. Protection Mechanisms and Their Modeling. Unlike DIVA [Austin 1999] or Argus [Meixner et al. 2007], PPU does not correct every error. The effects of fatal errors are



(a) Error-free CFG.



(b) CFG with error effects.



(c) Modified CFG with error effects and protection mechanisms.

Fig. 5. PPU software-level CFG models.

detected and resolved by added protection mechanisms that supervise the data and control flow.

Figure 5(c) shows a CFG that includes the effects of the MIS and MFU, denoted as M_{PPU}, and Table I provides a detailed description of the changes to the CFG. As an example, consider the transitions from a crash state to a valid state (e_{21} and e_{27}). This captures the effect of the ability of PPU to reset the PC when a crash occurs.

Table I. Transitions of the PPU Modified CFG

e_4	i++ indicates that S_i ends and S_{i+1} begins.
e_5, e_6	Model arbitrary number of nested scopes inside S_i.
e_7	Models arbitrary depth of scope nesting.
e_{10}, e_{17}	Without recovery, crash states are absorbing (sink).
e_{12}, e_{13}, e_{14}	Wrong transition to/from a parent scope.
e_{19}, e_{20}	Wrong transition to/from a child scope.
e_{21}, e_{27}	Wrong data-memory access is silenced or referenced to a dummy address.
e_{22}, e_{28}	Wrong I-memory access is silenced and S_i is terminated.
e_{23}	If PC reaches a wrong scope boundary, MIS terminates the S_i
$e_{23}, e_{24}, e_{25}, e_{26}$	Timers represent the MIS instruction counters, which prevent getting stuck in a scope.
e_{29}	SP and BP are refreshed at the beginning of S_i.

6. PROPERTY DEFINITIONS

Section 3 discussed the reliability concerns in terms of the high-level properties of *progress*, *ephemeral effect of errors*, and *executing essential states*. In this section, we take this a step further by showing how these properties can be defined formally. We then discuss PPU's additional properties and formally define them. The definitions we provide in this section are solely examples of many different forms to define these properties depending on the application.

In these formal definitions, we benefit Linear Temporal Logic (LTL) [Clarke et al. 1999]. LTL adds temporal modal operators to propositional logic. In this work, we use the *Globally* (**G**), *Finally* (**F**), and *Next* (**X**) operators. Given a formula ψ, $\mathbf{G}\psi$ indicates that ψ is always true, $\mathbf{F}\psi$ indicates that ψ is eventually true at some point in the future, and $\mathbf{X}\psi$ indicates that ψ is true at the next state.

6.1. Basic Desired Reliability Properties

6.1.1. Progress. Progress is defined with respect to to the FSM or CFG models. A reasonable notion of progress is that neither model is stuck in a set of states (i.e., a state cannot be visited infinitely often with the exception of states that are part of a valid infinite loop). From now on, we assume that our target CFG and FSM do not have a valid infinite loop and eventually halt.

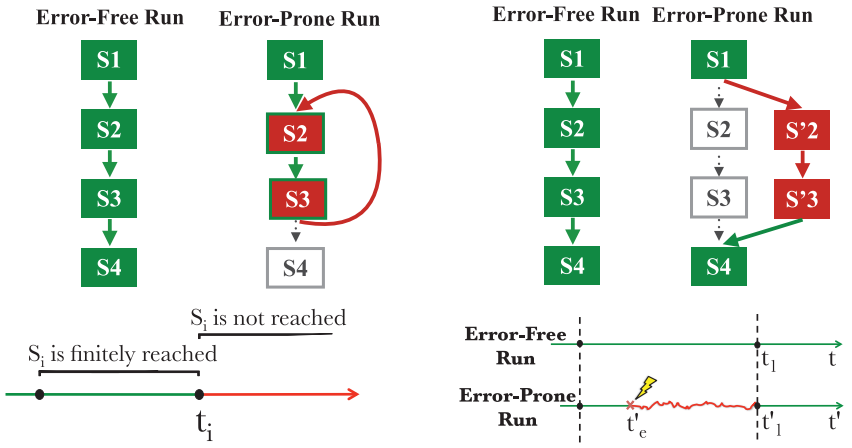
General Definition. If progress is satisfied, each state S_i that is executed should be executed in a finite time interval. Figure 6(a) compares an error-free run with an error-prone run that violates progress. In the error-prone run, progress is violated because control flow keeps visiting S_2 and S_3 infinitely. In other words, to satisfy progress, there should exists a *finite* time interval (including the empty interval) for each state S_i within which S_i is visited. After this time interval, state S_i is not visited again:

$$\forall S_i \;\; \exists t_i : \quad \forall t > t_i \quad S(t) \neq S_i, \tag{1}$$

where $S(t)$ is the state of the control flow at time t.

Formal Definition. A system makes progress when both the hardware FSM and the software CFG are making progress. It is important to note that if the hardware FSM does not make progress, then the CFG does not make progress either. However, the FSM can progress while the CFG is stuck (e.g., the software is stuck in an invalid infinite loop). This requires us to verify progress on both the FSM and the CFG.

Hardware Level: The FSM meets the control states Fe, Dc, Ex, Mem, and Wb in a loop that is repeated for each instruction. It comes to a halt when the "halt" command is decoded. Therefore, the progress property can be said to hold at the hardware level as long as instructions are being fetched. This is true even if the FSM states are visited



(a) Each state should be visited within a finite time duration.

(b) Error effects should fade within a finite time duration.

Fig. 6. Progress and ephemeral effect of errors properties.

in an incorrect order due to faults. Essentially, this weak notion of progress permits the instruction results to be erroneous as long as instructions are being fetched.

This can be checked in practice in a bounded way as follows: Consider a timer that is reset any time that Fe is visited. Then, this timer always shows the time interval since the last visit of Fe (T_{FE}). Now, we verify progress by checking if the FSM visits state Fe at least once in a T_N time interval:

$$\mathbf{G}(T_{FE} < T_N). \tag{2}$$

Software Level: The CFG does not make progress if either (i) it is stuck in a state, (ii) it is stuck in a loop, or (iii) it reaches an invalid state with no way to recover. In M_{PPU}, since the states S_i_Begin and S_i_End are not corrupted by errors, if M_{PPU} reaches S_i_End, either e_4 must be taken and thus S_{i+1}_Begin is reached, or the program terminates. We therefore check for progress by proving the following property:

$$\forall i : \mathbf{G}(S_i\_Begin \rightarrow \mathbf{F}\, S_i\_End). \tag{3}$$

6.1.2. Ephemeral Effect of Errors. The main purpose of this property is to avoid accumulation of data errors. Figure 6(b) compares an error-free run with an error-prone run, in which the error effects cause the control flow to deviate from its correct path at S_1. This control flow converges to its correct path at S_4 since the error effects fade by then.

General Definition. This property demands that the effect of a single error fades in a certain time duration (i.e., there is always some time t in the future where the data errors before t do not effect the results after t).

This is illustrated in Figure 6(b). This figure shows a time interval (t'_0 to t'_2) of the error-prone run of a program and the corresponding time interval of the error-free run (t_0 to t_2). We assume all previous error effects have been cleared by time t'_0 and the two error-free and error-prone runs have identical states at the beginning of this time interval. At time $t' = t'_e$, a transient error occurs in the error-prone run; hence, this run deviates from the error-free run. If the effects of the error totally disappear before t'_2, say at time t'_1, and no other error occurs, then the state of the error-prone run at t'_1 should match the error-free run state at some corresponding time t_1. The following

equation captures this:

$$\begin{aligned} &\exists t'_0,\ \exists t_0 : S_e(t'_0) = S_{e\_free}(t_0), \forall t' \in (t'_e,\ t'_2], t'_e > t'_0 : e(t') = false \\ \Rightarrow\ &\exists t'_1 \in (t'_0,\ t'_2],\ \exists t_1 : \forall \delta t \in (0,\ t'_2 - t'_1] : S_e(t'_1 + \delta t) = S_{e\_free}(t_1 + \delta t), \end{aligned} \tag{4}$$

where $e(t') = false$ means that no error has occurred at time t', $S_e(t')$ is the state of the error-prone run at time t', and $S_{e\_free}(t)$ stands for the state of the error-free run at time t.

Formal Definition. As previously mentioned, the ideal situation is that effects of transient errors totally disappear before the next time interval starts. However, this requirement may be too strong and, in addition, hard to satisfy. In this section, we give a practical definition by focusing on the effects of errors on *persistent variable states* at either the hardware or software levels. By persistent variables we mean those variables that are alive longer than some time interval (ΔT). A variable is alive between the time it is defined until it is used [Appel 1998].

For the FSM, the persistent variables include the processor's architectural states, such as the PC, Stack Pointer (SP), and Base Pointer (BP). For the CFG, the persistent variables can include loop counters and the variables that are alive inside a loop. These parameters are application-specific, and we assume that, for a given program, they can be determined automatically by static or dynamic methods.

In our definition of this property, we assign a timer to each of the persistent variables. The timer is reset whenever the variable state is known to be correctly updated to a value independent of its previous state.

Using this timer, Equation (5) captures the ephemeral effect of an error's property, where P_i is the i^{th} persistent variable and T_i is the timer value.

$$\forall P_i : \mathbf{G}(T_i < \Delta T). \tag{5}$$

6.1.3. Executing Essential States. Essential control states are those states that play an irreplaceable role in a program. Thus, if any of these states is missed, the corresponding run of the program cannot deliver useful results.

General Definition. There are many ways to define this property. Our definition indicates that the essential states (S_{ess}) should be met at least once during execution:

$$\forall S_{ess} \exists t : S(t) = S_{ess}. \tag{6}$$

Figure 7(a) compares an error-free run with an error-prone run that still satisfies the essential states property. In this run, the essential state (S_{ess}) is not skipped, while other state[s] (S_2) might be skipped because of error effects.

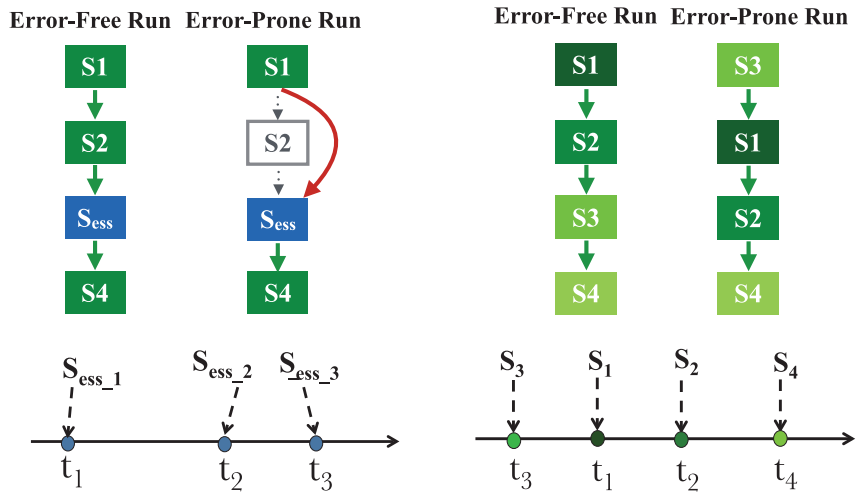
Formal Definition. Defining the essential states for both the FSM and CFG is generally application-based. In this work, we assume that the essential states are known for both the FSM and CFG. Then, we formally define this property based on Equation (6):

$$\forall S_{ess} :\ \mathbf{F}\,(S_{control} = S_{ess}), \tag{7}$$

where $S_{control}$ is a state of CFG or FSM. Comparing this definition with Equation (2), we can see that, in the FSM and for the essential state of Fe, executing the essential states is a special case of making progress with ΔT equal to the program run. However, in a conservative definition, all hardware states can be defined as essential.

6.2. PPU's Additional Properties

The PPU model satisfies additional properties that are not necessarily required but are helpful for providing useful results. We now discuss two of these properties, in-order control flow and availability of partial results. These properties are at the software level.



(a) All essential states should be visited some time during the program run.

(b) An error-prone run might fail to produce useful results because of a wrong order of computations.

Fig. 7. Executing the essential states and in-order control flow properties.

6.2.1. In-Order Control Flow. Figure 7(b) compares an error-free run with an error-prone run, where the latter run visits the same set of states but in an incorrect order. The error-prone control flow does not violate any of the basic desired reliability properties. This is not stuck in a state or a loop, and it also visits all the states including possible essential states. However, there is a low chance a processor with this control flow behavior produces useful results.

General Definition. The in-order control flow property indicates that the states of a control flow are visited in order. This property is defined as follow:

$$\forall S_i, S_j : \ \left(t_j^R > 0,\ j > i\right) \rightarrow t_j^R > t_i^R. \tag{8}$$

Here, the inequality $j > i$ denotes that S_i strictly precedes S_j in the error-free CFG. In this definition, t_j^R is the retirement time of S_j, which is also the last time S_j is visited. Note that t_j^R is zero when S_j is skipped. In that case, comparing the retirement times is meaningless. Effectively, this property allows S_j to be skipped but not executed out of order.

Formal Definition. To formally define this property for M_{PPU}, we define parameter I_{ret} as the index of the last retired scope at the time. For instance, if in a run of a program, S_i is the last retired state thus far, $I_{ret} = i$ at the time. Now, Equation (9) formally defines this property:

$$\mathbf{G}(\mathbf{X}I_{ret} \leq I_{ret}). \tag{9}$$

This means that the latest retired state is always either the same as the previous retired state or succeeds that state in the error-free CFG. In a stronger definition of this property, we can prohibit skipping the states:

$$\mathbf{G}(\mathbf{X}I_{ret} = I_{ret} - 1 \ \vee \ \mathbf{X}I_{ret} = I_{ret}). \tag{10}$$

Which means that the latest retired state is always either the same as the previous retired state or is the *immediate* successor of that state in the error-free CFG.

6.2.2. Availability of Partial Results. The in-order property restricts the order of the scopes being retired, but it does not reason about when the states are retired. After S_i is retired, the results produced by this state are ready. Thus, information about this time is helpful to know when to expect partial results instead of waiting for the whole program to finish.

General Definition. Given t_i^R as the partial result time for S_i, one can expect partial results produced by S_i any time after t_i^R. Thus, S_i should be retired and is not visited again by t_i^R:

$$\forall t > t_i^R, \;\; S(t) \neq S_i. \tag{11}$$

In a stronger definition, we might replace $S(t) \neq S_i$ with $S(t) > S_i$. The stronger definition indicates that partial results produced by S_i are expected after all states preceding S_i (in the error-free CFG) are retired.

Note that if a CFG satisfies the progress property in Equation (1), the program eventually finishes at some point (T_{tot}). It is obvious that $t_i^R \geq T_{tot}$ satisfies Equation (11). But the interesting case is when $t_i^R < T_{tot}$ satisfies Equation (11). Which means that partial results produced by state S_i are ready by some known time t_i^R and before the program finishes.

Formal Definition. The in-order property indicates that the scopes are retired in order. It means that scope S_i is retired only after all its preceding scopes are skipped or retired. However, considering M_{PPU} (Figure 5(c)), there are various possible paths in the CFG that satisfy the in-order requirement. For instance, the finishing part of some of the possible paths are:

... $S_i\_Begin \rightarrow S_i\_Body \rightarrow S_i\_End$, or
... $S_i\_Begin \rightarrow S_i\_Body \rightarrow S_k\_Child \rightarrow S_i\_Body \rightarrow S_i\_End$, or
... $S_i\_Begin \rightarrow S_i\_Body \rightarrow S_j \rightarrow S_j \rightarrow ... \rightarrow S_i\_End$, or *etc.*

However, MIS forces a strict timing schedule on all the possible paths. Based on this schedule, regardless of which path PC takes, partial results produced by scope S_i are ready any time after t_i^R. And since M_{PPU} retires the scopes in order, the retirement time of state S_i can be calculated as the sum of the time durations spent in S_i and the preceding states in the error-free CFG. As the stronger definition in Equation (10) indicates, this partial result time should satisfy the following equation:

$$\forall i \;\; \mathbf{G}(T_{timer} > t_i^R \rightarrow I_{ret} > i), \tag{12}$$

where, T_{timer} is the time passed since the beginning of a program and is measured by a timer.

Note that, to calculate t_i^R, we assumed PPU retires scopes in order. The assumption of in-order retirement, as well as other properties defined in this section, is verified on M_{PPU} next.

7. EVALUATION

In this section, we evaluate the reliability of the PPU design by formally verifying the properties discussed in Section 6. We then evaluate it in terms of quality of results.

7.1. Property Verification

We verify the reliability properties on the hardware and software levels of the PPU design's control flow. Figure 8 shows the flow of our experiments. In Section 5, we modeled the fault-free control flow of this design at both levels, then we added the

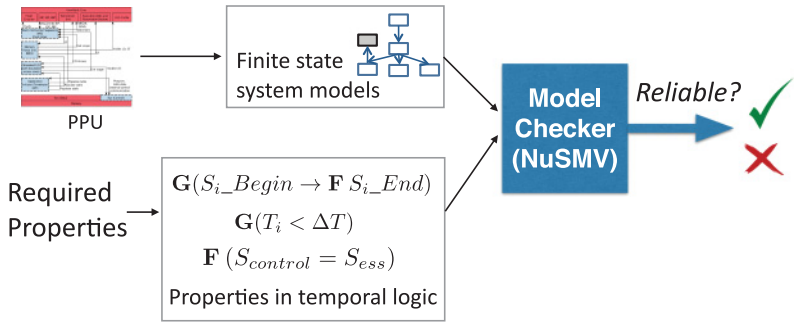


Fig. 8. Flow of the experiments for the formal verification of the properties.

Table II. Model Checking Results

Model	Property	Reachable States	Time (s)	Holds?
HW Level Single Core FSM	Progress	308	0.022	Y
	Ephemeral effect		-	N
	Essential states		0.022	Y
SW Level PPU CFG	Progress	2.06×10^{13}	2.97	Y
	Ephemeral effect		1.44	N
	Essential states		99.90	N
	In-order CF		1.61	Y
	Partial results availability		3.30	Y

effects of faults and protection mechanisms to our models. Now, we verify the reliability properties written in SMV using the NuSMV [Cimatti et al. 2000] model checker.

The experiments were run on a 1.7GHz Intel Core i7 running OS X 10.9.5. Table II reports the verification results, where the reachable states show the number of the states that the model checker could reach in the respective model. The CPU time shows the time that the model checker needs to verify the respective property on a model. The huge difference between the reachable states of the two models and also their model checking time is because the HW model FSM in Figure 4 is much simpler (with fewer states and transitions) than PPU's SW level CFG in Figure 5. A "-" in the column for the CPU time indicates that the property did not formally need to be checked, as discussed in the respective subsections.

7.1.1. Single Core Error-Prone Processor. We verify progress (Equation (2)), ephemeral effects of errors (Equation (5)), and meeting essential states (Equation (6)) on the M_{hw} model in Figure 4(c). Table II indicates the results of this experiment.

Progress: The progress property (defined in Equation (2)) is satisfied in M_{hw}. This is due to the added fetch timer that makes sure instructions are being fetched constantly.

Ephemeral effect of errors: To satisfy this property, persistent variable states should be updated frequently enough. Since in M_{hw} no variable state is refreshed, this property is not satisfied in M_{hw}.

Meeting essential states: We assume that the essential state in M_{hw} is Fe. This assumption implies that this property is satisfied in M_{hw} because of the added fetch timer. Note that, in order to satisfy this property given a different set of essential states, a separate timer must be assigned for each essential state.

7.1.2. PPU. At the software level, we verify PPU's reliability with respect to M_{PPU}. In our model, we select reasonable values of the parameters: The number of scopes is 20,

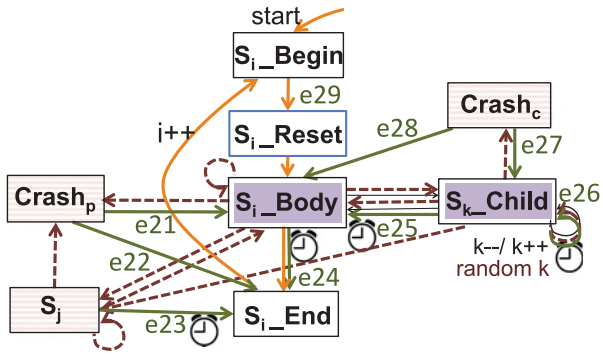


Fig. 9. PPU can keep skipping a child scope.

ΔT is 15, the time limit for the parent scopes (T_{ps}) is 10, and an instruction counter for the child scopes is not used.

Progress. As discussed in Section 6, for the application to make progress, the PC should not get stuck in a state or a loop. The results show that this property holds in M_{PPU} (Table II). Thus, the PC supervision by MIS and the recovery mechanisms in M_{PPU} are strong enough to ensure progress. Also, MIS counters for child scopes are not necessary for satisfying the progress property.

Ephemeral Effect of Errors. To satisfy this property, persistent variable states should be updated frequently enough. In PPU, the hardware-level persistent variables of PC, SP, and BP are updated any time a scope is started. Therefore, this property holds at the hardware level if and only if the maximum time that PC is allowed to spend in a parent scope (T_{ps}) is less than the maximum acceptable period for the persistent variables to update (ΔT). However, this property does not necessarily hold with respect to M_{PPU} because PPU does not update any program variable.

Possible Solution: Generally speaking, in contrast with many programs that carry the data for a while, stream applications work on a frame of data for a short while. Therefore, we do not expect many persistent variables in this type of programs. However, we noticed that each of the StreamIt applications targeted by PPU contains one main loop whose loop counter is alive as long as the program runs. Clearly, the loop counter is a persistent variable and thus should be protected.

Executing Essential States. We assume there is one essential state that can be any of the valid scopes. Our experimental results show that this property is satisfied in M_{PPU} when the essential state is a parent scope ($S_{ess} = S_i\_Begin$ for any i) and fails when it is a child scope ($S_{ess} = S_k$ for any k). One simple counter example occurs when the essential state is $S_{2.1}$ and the CFG of Figure 5(c) takes this series of transitions: [active scope $= S_1$] $e_1, e_{29}, e_2, e_3, e_4$ [active scope $= S_2$], e_{29}, e_3, e_4 [active scope $= S_3$], *etc.* This path is colored orange in Figure 9. As this example shows, the essential state of $S_{2.1}$ can simply be skipped.

Possible Solution: To prevent PC from skipping essential child scopes, we suggest adding a column to the MIS table and keeping track of visiting the essential child scopes. One should note that if one scope is essential, its parent scope (if any) should also be marked essential. Otherwise, skipping the parent scope would prevent visiting the essential child scope.

Assume that scope S_i (either parent scope or child scope) has some of child scopes, among which $m > 0$ child scopes are essential. Since S_i has essential child scopes, it would not be skipped. As PC enters S_i, MIS allows S_i to be retired only after all of

its m essential child scopes are visited. At any attempt of PC to exit S_i, MIS should reset the PC to the beginning of one of the unvisited essential child scopes. Therefore, S_i retirement would be rejected at most m times, and, since m is a finite number, this scheme would not interfere with the progress property.

In-order Control Flow. As the results show, M_{PPU} satisfies the in-order property for parent scopes. The reason is that MIS keeps track of the retired scopes and does not let a wrong scope be retired. Since MIS does not let the parent scopes be skipped (discussed in the essential states property), we also verified the strong version of this property (Equation (10)) on M_{PPU}, where I_{ret} in Equation (10) is the index of the last retired parent scope thus far. The result of this experiment is depicted in Table II.

Partial Results Availability. Since M_{PPU} satisfies the in-order property for the parent scopes, the path PC takes is deterministic in term of the scopes' order. However, the time duration that PC spends in each of the scopes is not deterministic. Thus, t_i^R is calculated assuming that PC spends maximum allowed time in all preceding scopes. Consequently, once a program run begins, one can expect partial results of the computations inside scope S_i in t_i^R amount of time.

Addressing Desired Properties in a Multicore Implementation. A subsequent implementation of PPU in a multicore context [Yetim et al. 2015] addresses the gaps in the desired properties as follows:

Ephemeral Effect of Errors: PPU satisfies this property by protecting the persistent variables in an error-free header. At the beginning of each chunk of computation (referred to as frame computation), a frame header is inserted to indicate the beginning of the frame computation. The persistent variables of a frame computation, including the main loop counter, are Error Correction Coding (ECC) protected inside the respective frame headers [Yetim et al. 2015].

Executing Essential States: PPU protects the main loop computation and assigns each of the child scopes to a core [Yetim et al. 2015]. Therefore, the assigned cores would execute the child scopes, and the possible essential child scopes would not be missed.

Overall, PPU meets most of its desired goals, but the results of the property checking point to failures in meeting some of the basic desired reliability properties. As discussed, these were fixed in a subsequent multicore implementation of PPU. In the following, we evaluate PPU in term of quality of results.

7.2. Quality of Results

We evaluate the performance of PPU by running StreamIt benchmarks and evaluating the accuracy of the results in the presence of faults.

7.2.1. 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 [Magnusson et al. 2002]. 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(a). 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). We model hardware errors by randomly flipping bits in the register file following a uniform distribution for a given MTBE. 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 into the following six registers: E[A-D]X, ESI, and EDI. Other states, 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. It is worth mentioning that our observation confirms that errors affect ESP, EBP, and EIP registers even without direct error injection. For instance, we observed that erroneous pointers in loops would often overwrite large sections of the stack.

7.2.2. Benchmarks. Our experiments use seven benchmarks from the StreamIt benchmark suite [Thies et al. 2002]. 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 application runs on the simulated machine, whereas other error characterizations use a truncated version (the first 50ms measured on the machine) due to long simulation times.

7.2.3. 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.

Characterization of Catastrophic Errors. Prominent Error Types Handled by Protection Modules: Figure 10 shows how often different protection handlers are invoked for each type of error. An MTBE of 256k instructions is high enough to maintain acceptable output quality and low enough to analyze the effects of errors (see Section 7.2.3). 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.

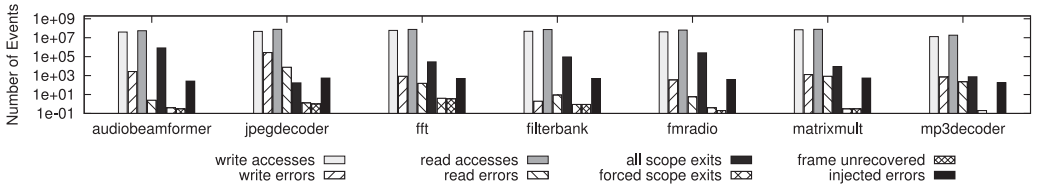


Fig. 10. At an MTBE of 256k instructions, the most common error types are memory address errors (read and write accesses to nonpermitted memory regions) and, in some benchmarks, forced scope exits (due to execute access errors, processor exceptions, or exceeding an instruction count limit). Shaded bars 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 into the architectural registers.

Effects of Illegal Instruction Fetch: As Section 4 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 12(a) shows results related to this issue. Only two of our applications ever experience execute access failures. The failures for *fft* start for MTBE $\approx 10^6$ instructions, and for *mp3decoder* they start for MTBE $\approx 10^5$ instructions. 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 7.2.3).

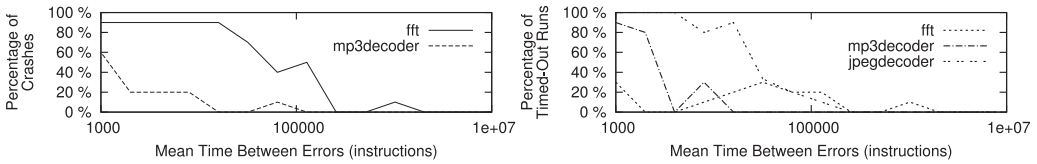
Errors Causing Application Hangs: While the MFU could enable some error-tolerant operation, it would be insufficient without the MIS. In particular, without the MIS's ability to keep control flow on track, errors not only affect program and address values, but they also cause the programs to hang and fail to reach the end of the computation. Figure 12(b) shows that *fft*, *jpegdecoder,* and *mp3decoder* would hang (runtime higher than 20×) at an MTBE as infrequent as $\approx 10^6$ instructions. As Section 7.2.3 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 exit loops early, causing the application to complete faster but with degraded output quality.

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 [Stathaki 2008] is a common metric to quantify this difference. We use SNR to quantify the change that 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) 2048k instrs (18dB) (b) 32k instrs (7dB)

Fig. 11. Image outputs from the JPEG decoder benchmark at different Mean Time Between Errors (MTBEs) from the same seed for the random number generator.



(a) MFU disabled to characterize execute failures (b) MIS disabled to characterize hangs

Fig. 12. (a) Applications experience hangs (infinite looping) with the Mean Time Between Error (MTBE) as high as every $\approx 10^6$ instructions. 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 instructions or less.

a given MTBE. As before, we perform 10 runs to capture the statistical variation of SNR_e across runs.

Figure 13(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 11 presents images for two SNR_e values. With an MTBE of 2048k instructions 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 withstood $\approx 10^6$ 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 reinitialized at 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 13(b) for the MP3 decoder shows similar trends. However, in contrast with JPEG, here the SNR can go to negative values. This is because of data representation. The 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, and silence is better

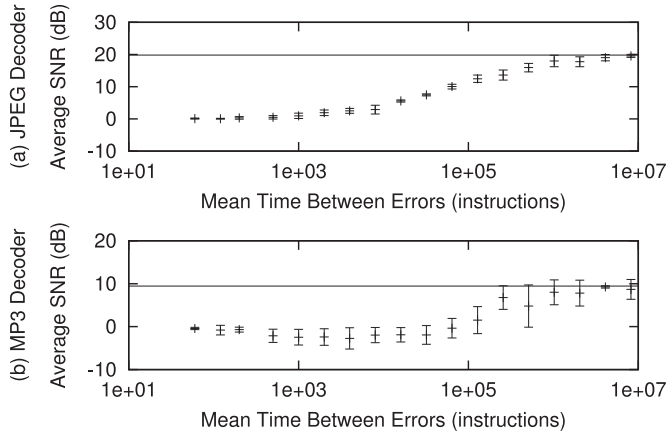


Fig. 13. Output quality of the (a) JPEG and (b) MP3 decoder algorithms running at different Mean Time Between Errors (MTBEs). The flat line in each graph corresponds to the output quality when the decoder is run without any errors.

Table III. Efficacy of Protection Mechanisms in Terms of Satisfying Reliability

	Error in Control Flow?	Progress	Ephemeral Effect of Errors	Executing Essential States
EnerJ	No	It benefits from error-free control flow.	It is possible to distinguish the persistent variables and define them as "precise."	Approximation methods should not cause essential states to be skipped.
PPU	Yes	MIS assigns a timer to each scope and prevents the PC from getting stuck.	PC, SP, and BP are refreshed periodically. But SW level variables are neglected.	All parent scopes are visited but child scopes might be skipped.
Argus	Yes	Timers are applied to basic blocks. But getting stuck in a loop is still possible	Signatures could be assigned to persistent variables to protect them. However, signatures are not fully reliable.	Proper signatures might be helpful to satisfy this property.
ERSA	Yes	SRC does not let RRCs to get stuck. But the system as a whole might get stuck.	Persistent variables in SRC are error-free. But this is not necessarily true for persistent variables in RRCs.	In case the system does not get stuck, application-based sanity checks are helpful for satisfying this property.

than noise. The reader can listen to different sounds for different error rates here: http://youtu.be/2XhZxbNz7Lk.

8. RELATED WORKS

We now discuss some of the related works in reliable system design using redundancy, error correction, recovery, and other methods. We then discuss the efficacy of the provided protection mechanisms in terms of satisfying the reliability properties of progress, ephemeral effect of errors, and executing essential states (Table III).

8.1. Error-Resilient Hardware

The error-resilient designs protect hardware against errors by either preventing the transient error from happening or by detecting and correcting all those errors. Storage units, such as SRAM, are protected by ECC [Alameldeen et al. 2011]. Logic units, on

the other hand, might be protected by using larger transistors or higher voltages or through error correction [Kuhn et al. 2011; Mukherjee 2011]. Razor detects timing errors in the critical paths by redundant shadow latches [Ernst et al. 2003].

DIVA: DIVA is an example of an error-resilient processor that uses error detection techniques. In this work, the whole processor is protected against errors by adding a reliable checker immediately preceding the commit stage [Austin 1999]. The checker is fed with the operands and the op-code of each instruction to replicate the computation. In case of inconsistency in the results, the failing instruction is repeated. In this way, all errors in the computation are caught and corrected.

Reliability Properties in DIVA: The main advantage of error-resilient computations is reliability. Designs that use error-resilient computations can target all types of applications regardless of the applications' error-tolerance. However, protecting against all errors may be expensive in terms of the area, power, and delay overhead.

8.2. Error-Tolerant Data Flow

The requirement of achieving perfectly correct results can be relaxed for inherently error-tolerant applications, such as media. Leveraging this feature, error-tolerant accelerators incur lower overhead cost while adequately handling computational errors. For instance, Algorithmic Noise Tolerant (ANT) accelerators mitigate data errors by adding estimator blocks to the error-prone computational units [Hegde and Shanbhag 1999]. Alternately, N-Modular Redundancy (NMR) diminishes error probability by leveraging redundancy. In this method, the output is the majority vote of N identical units processing the same input [Kim and Shanbhag 2012].

Additionally, some processors let errors propagate in their data flow while keeping the control flow error-resilient.

EnerJ: EnerJ is an example of a processor with error-tolerant data flow [Sampson et al. 2011]. EnerJ proposes a new programming language (EnerJ) that isolates the precise part of the computation from the approximate part. This allows the approximate part to be run on error-prone hardware. To avoid control errors, they do not let control flow elements, such as pointers, be approximate.

Reliability Properties in EnerJ: EnerJ satisfies the progress property by relying on error-free control flow. It can additionally satisfy the ephemeral effect of errors property by marking the persistent variables as precise. Since EnerJ runs the precise computations on reliable hardware, the persistent variables would be protected. EnerJ also satisfies the execution of essential states if the approximation mechanisms never include removing essential states. Otherwise, this property might be violated since EnerJ does not verify the accuracy of the approximate part of the data flow.

8.3. Error-Tolerant Data and Control Flow

Control flow is generally less error-tolerant than data flow since even a single error in the control flow can cause catastrophic effects. Control error-tolerant processors, like PPU, include architectural mechanisms that protect against fatal errors while permitting potentially tolerable errors.

Argus: In Argus, error is injected and propagated in both data and control flow [Meixner et al. 2007]. To protect the processor, errors are detected and corrected applying the following methods: (i) Argus prevents wrong transitions in the CFG by declaring the boundary of basic blocks before the runtime. (ii) Argus leverages signatures in computations, where signatures are reliable estimations of respective variables and are computed at compile time. (iii) Argus protects memory elements using parity bits.

Comparison with PPU: (i) PPU does not prevent wrong transitions in the CFG; however, it forces the right order of scope retirement. (ii) PPU also leverages compile-time computed static information in the MIS table (e.g., base pointer and stack pointer

values at the beginning of each scope) for supervising the control flow. (iii) PPU does not use parity checking.

Reliability Properties in Argus: The protection schemes in Argus do not fully protect essential elements of the control flow, such as PC or loop counters. Therefore, Argus might violate progress. On the other hand, the signature mechanism in this work is promising for satisfying essential states and ephemeral properties. The accuracy of persistent variables could be tracked by assigning signatures to those variables. Also, by using proper signatures, visiting essential states could be verified. However, it is not a fully reliable mechanism since signatures are built on the error-prone hardware.

ERSA: ERSA is another processor with error-tolerant control flow [Cho et al. 2012]. ERSA runs the main control flow on an error-free core (SRC), which assigns the computational tasks to cores with error-prone data and control flow (RRCs). (i) SRC supervises the RRCs and reboots them in case of hangs or crashes. (ii) SRC also verifies the accuracy of RRC computations by applying application-based sanity checks. In case of unacceptable results, SRC reschedules the corresponding task.

Comparison with PPU: (i) The MIS in PPU terminates the scopes in case of hangs or crashes. (ii) PPU supervises the control flow to achieve the best-effort results. It does not verify the accuracy of the results with application-based tests.

Reliability Properties in ERSA: Although SRC prevents RRCs from getting stuck by rebooting them, ERSA might still violate the progress property as a whole system. The counter example is when all RRCs keep failing in completing a task. In that case, SRC has to reschedule the remaining task infinitely, which violates both progress and the essential states properties. However, in addition to this case, ERSA makes sure of executing essential states by providing sanity checks before accepting the result of each RRC task. As for the ephemeral effect of errors, ERSA does not protect any persistent variable at the RRC level. However, since SRC is error-free, the persistent variables in that core are error-free. We formally verified the reliability properties on ERSA's programming model using a model checker [Golnari et al. 2015].

8.4. Approximate Computing

In approximate computation, the programmer decides which part of the program can be approximated and which parts should be precise. This type of processor usually consists of both error-prone (and efficient) units and reliable units. The critical regions of the computation are executed on the reliable hardware while the approximate part might be assigned to the error-prone part of the architecture. There are many works in the literature to help the programmer with defining the boundary, mapping the critical parts of the program to reliable hardware, and applying approximation to the rest of the computations [Esmaeilzadeh et al. 2012a; Liu et al. 2012; Sampson et al. 2011; Misailovic et al. 2014]. The EnerJ system provides an example of such a processor.

Applying approximation techniques, such as approximating functions at runtime [Baek and Chilimbi 2010], skipping loop iterations [Sidiroglou-Douskos et al. 2011], replacing deterministic units with a neural network [Esmaeilzadeh et al. 2012b], and many other approximations techniques [Mittal 2016; Xu et al. 2016], are beneficial in terms of energy performance. However, all these approximation techniques cause inaccuracy in the result.

Generally, in these processors, error in the computation arises from applying approximation and using error-prone hardware. The effects of errors caused by utilizing error-prone hardware are similar to hardware error effects in error-tolerant computing, and, therefore, the same protection mechanisms could be applied. However, in approximate computing, the programmer can decide which parts to run on the reliable part. Therefore, it is usual to keep the control flow elements error-free in these processors and let the error propagate in certain parts of the data flow.

9. CONCLUSION

Emerging research in reliable computer architecture is relaxing the requirement that processors strictly meet their ISA specification. However, they do not provide an alternate specification beyond best-effort. In this article, we suggest a set of basic desired properties for such processors to provide useful results. We show that these properties need to be met at both the hardware and software levels and provide for a formal system modeling and property specification for use with model checking. Based on these requirements, we propose PPU, a novel architecture that can survive in the presence of architecturally visible errors using coarse-grained management of application execution. The minimal microarchitectural support of PPU allows streaming applications to run on an unreliable processor with low protection overheads and still provide good output quality. 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 of 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.

Additionally, we show how to construct a system model that includes the fault-free behavior of PPU, the fault effects, and the protection and recovery mechanisms provided by PPU. Finally, we use model checking to verify the reliability properties of PPU. Furthermore, the property failures suggest ways of augmenting the design so as to overcome this limitation. These results point to the value of this methodology in the design of such future systems.

REFERENCES

Alaa R. Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu. 2011. Energy-efficient cache design using variable-strength error-correcting codes. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 461–471.

Andrew W. Appel. 1998. *Modern Compiler Implementation in ML*. Cambridge University Press.

Todd M. Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual IEEE International Symposium on Microarchitecture*. IEEE.

Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. *ACM Sigplan Notices* 45, 6 (2010).

Shekhar Y. Borkar. 2005. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (2005), 10–16.

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. *ACM SIGPLAN Notices* 48, 10 (2013), 33–52.

Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. 2012. ERSA: Error resilient system architecture for probabilistic applications. *IEEE Transactions on CAD of Integrated Circuits and Systems* 31, 4 (2012), 546–558.

Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. 2000. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 410–425.

Edmund Clarke Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking*. MIT Press.

Robert C. Daley and Jack B. Dennis. 1968. Virtual memory, processes, and sharing in multics. *Communications of the ACM* 11, 5 (1968), 306–312.

Jack B. Dennis. 1965. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM (JACM)* 12, 4 (1965), 589–602.

Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012a. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 301–312.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012b. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 449–460.

Ameneh Golnari, Yakir Vizel, and Sharad Malik. 2015. Error-tolerant processors: Formal specification and verification. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 286–293.

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 21–40.

Rajamohana Hegde and Naresh R. Shanbhag. 1999. Energy-efficient signal processing via algorithmic noise-tolerance. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*. ACM, 30–35.

ITRS. 2011. ITRS Process Integration, Devices, and Structures. Retrieved from http://www.itrs.net/Links/2005ITRS/Home2005.htm

Eric P. Kim and Naresh R. Shanbhag. 2012. Soft n-modular redundancy. *IEEE Transactions on Computers* 61, 3 (2012), 323–336.

Kelin J. Kuhn, Martin D. Giles, David Becher, Pramod Kolar, Avner Kornfeld, Roza Kotlyar, Sean T. Ma, Atul Maheshwari, and Sivakumar Mudanai. 2011. Process technology variation. *IEEE Transactions on Electron Devices* 58, 8 (2011), 2197–2208.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*. IEEE.

Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2012. Flikker: Saving DRAM refresh-power through critical data partitioning. *ACM SIGPLAN Notices* 47, 4 (2012), 213–224.

Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 210–222.

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 309–328.

Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 62.

Shubu Mukherjee. 2011. *Architecture Design for Soft Errors*. Morgan Kaufmann.

Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. 2010. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 335–338.

David A. Patterson and John L. Hennessy. 2012. *Computer Organization and Design: The Hardware/Software Interface*. Academic Press.

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Analysis and evolution of journaling file systems. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 105–120.

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 164–174.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the European Conference on Foundations of Software Engineering*. DOI:http://dx.doi.org/10.1145/2025113.2025133

Phillip Stanley-Marbell and Martin Rinard. 2015. Lax: Driver interfaces for approximate sensor device access. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

Tania Stathaki. 2008. *Image Fusion: Algorithms and Applications*. Academic Press.

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer Berlin Heidelberg, 179–196.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 36.

Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2016. Approximate computing: A survey. *IEEE Design & Test* 33, 1 (2016), 8–22.

Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)* 24, 4 (2006), 393–423.

Yavuz Yetim, Sharad Malik, and Margaret Martonosi. 2015. CommGuard: Mitigating communication errors in error-prone parallel execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

Yavuz Yetim, Margaret Martonosi, and Sharad Malik. 2013. Extracting useful computation from error-prone processors for streaming applications. In *Proceedings of the Design, Automation and Test in Europe*.

Received November 2015; revised May 2016; accepted August 2016