Asymmetric Resilience for Accelerator-Rich Systems

Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, Vijay Janapa Reddi

Abstract—Accelerators are becoming popular owing to their exceptional performance and power-efficiency. However, researchers are yet to pay close attention to their reliability—a key challenge as technology scaling makes building reliable systems challenging. A straightforward solution to make accelerators reliable is to design the accelerator from the ground-up to be reliable by itself. However, such a myopic view of the system, where each accelerator is designed in isolation, is unsustainable as the number of integrated accelerators continues to rise in SoCs. To address this challenge, we propose a paradigm called “asymmetric resilience” that avoids accelerator-specific reliability design. Instead, its core principle is to develop the reliable heterogeneous system around the CPU architecture. We explain the implications of architecting such a system and the modifications needed in a heterogeneous system to adopt such an approach. As an example, we demonstrate how to use asymmetric resilience to handle GPU execution errors using the CPU with minimal overhead. The general principles can be extended to include other accelerators.

Index Terms—Reliability, error recovery, soft errors, voltage noise, accelerator architecture, heterogenous system.

1 Introduction

CPU scaling can no longer sustain the demand for performance and power efficiency owing to the end of Dennard scaling and the diminishing returns from microarchitecture enhancements. Therefore, heterogeneous architectures, represented by widely adopted GPUs and custom hardware accelerators are deemed to be the solution. They provide continued performance and power efficiency improvements beyond the general purpose CPU.

Although heterogeneous architectures provide performance and power efficiency benefits, we must first and foremost address their reliability before integrating them into our system. Taking the GPU as an example, prior work [1] has shown that the its MTBF (mean time between failures) is almost eight times lower than the CPU’s in a large-scale system. So it is of crucial importance that we devise novel techniques to maintain system reliability in the new era of heterogeneous systems. Hence, simultaneously optimizing the performance, efficiency, and reliability is important.

It is difficult to directly extend prior error recovery work to accelerators because they either incur large overhead or rely on CPU-specific features. Meanwhile, accelerator architectures are often distinctive, which makes their reliability optimization even more difficult. Facing those challenges, we propose a paradigm called *asymmetric resilience* shown in Fig. 1 to ensure the reliability of heterogeneous systems in the presence of transient accelerator execution errors. Asymmetric resilience treats the accelerator task as a whole for reliability assurance. It extends the previously exploited idempotency property of a code region [2] to an accelerator task to maximize the efficiency. The extended property only uses the CPU-accelerator interface information and therefore generalizes well to different accelerators.

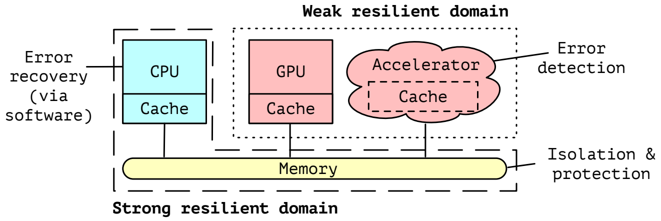


Figure 1: Asymmetric resilience overview. It ensures the system’s reliability using the most resilient component.

In this work, we demonstrate the efficiency of asymmetric resilience on the GPU as an accelerator example. Specifically, we leverage hardware/software co-design to implement the principles of asymmetric resilience. Through our study, we observe that most kernels have simple transaction-like control interface with well-defined input/output. As such, asymmetric resilience can eliminate the checkpoint overhead for the input and output memory by using memory protection mechanism and re-executing the kernel, respectively. We design a runtime system prototype that uses the CPU to recover from GPU voltage noise error and soft error, and demonstrate its negligible overhead.

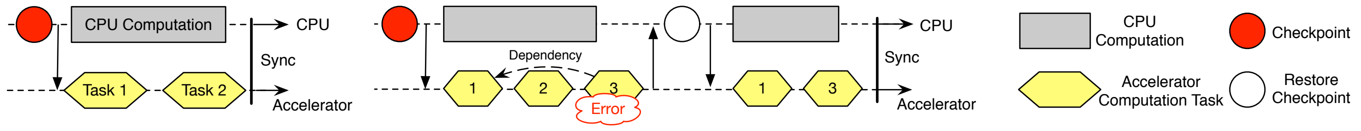
Finally, we extend the scope of the work to discuss how asymmetric resilience can be applied to different types of accelerators. We demosntrate that asymmetric resilience can relax the resiliency requirement of accelerators as it ensures the system reliability centering around the CPU architecture, which allows designers to focus on the accelerators’ performance or power efficiency optimizations.

2 Asymmetric Resilience

We start by explaining asymmetric resilience. In a traditional system, all components, i.e., the CPU and the accelerators, are expected to have the same resiliency level. They are all expected to detect and recover from an error, by themselves. The shortcoming of this approach is that it burdens both the CPU and accelerators to deal with reliability in isolation, not understanding the context of the system as a whole.

In contrast, the asymmetric resilience design paradigm introduces the notion of resiliency domains to separate error detection from error recovery. Resilient domains in our work

- *J. Leng and V. J. Reddi were with the University of Texas at Austin when conducting this work; They are now with Shanghai Jiao Tong University and Harvard University, respectively. E-mail: leng-jw@sjtu.edu.cn*
- *A. Buyuktosunoglu, R. Bertran, and P. Bose are with IBM T. J. Watson Research Center.*



(a) Error free operation.

(b) Error recovery process.

Figure 2: Accelerator error recovery using CPU. (a) Error-free execution and CPU checkpointing. (b) CPU performs error recovery when an error occurs inside an accelerator task, considering dependencies between the tasks.

also adopt the similar semantics as in prior containment domain [3]. Specifically, failures in a resilient domain are contained within the domain and cannot corrupt the states in the other resilient domain.

In addition, resilient domains can have a strong or weak resiliency level, which leads to a *strong resilient domain* and a *weak resilient domain* in Fig. 1. The strong resilient domain requires both error correction and recovery, while the weak resilient domain requires only error detection. It is natural to deploy the accelerators in the weak resilient domain and the CPU in the strong resilient domain. The resiliency separation and error containment allow for more holistic, big picture level optimizations, rather than siloed optimizations at the individual hardware accelerator level.

A method to recover from errors in the strong domain is to checkpoint and restart. So, we consider the commonly adopted checkpoint and restart scheme for implementing asymmetric resilience. The most straightforward implementation is to periodically create checkpoint for accelerator data [4], [5]. In the context of accelerator errors, it requires checkpointing for tasks even when no error occurs as shown in Fig. 2a. However, the resulting checkpoint overhead can be enormous (see Sec. 5), which defeats the purpose of adopting accelerators. Tbl. 1 summarizes why checkpoint and restart is not a good fit for accelerator-rich systems.

Prior work [2] has leveraged the idempotency to remove the checkpoint overhead. A code region is idempotent if its multiple executions lead to the same effect. As such, recovering from a transient error in an idempotent region simply requires re-execution. However, it is difficult to extend this line of work for accelerators. First, prior work assumed a strong fault model where a transient error is detected before corrupting the memory state. Such a strong fault model is based on CPU's speculation feature and feasibility of control flow signature, but are likely to be absent in accelerators. Second, most prior CPU-centric work relied on the dataflow or pointer analysis where programs are expressed in the form of sequential instructions. Applying them to non-Von Neumann architecture based accelerators is challenging.

Facing these obstacles, we extend the idempotency property to the entire task for overhead reduction. This property of a task, dubbed as *task-level idempotency*, is only determined by the information of memory region that it reads from (input memory region) and writes to (output memory region). We consider the example of a convolutional layer accelerator, which takes two input memory regions (for weight and input feature map) and stores the output feature map in another memory region. Recovering from transient error in the task requires no checkpoint as we can protect the input memory regions in the read-only mode and reissue the task to restore the output memory region. The extended property is not restricted to Von Neumann execution model and assumes a more general fault model: an error can corrupt the entire memory address space.

Category	Fault Model	CPU-specific?
Epoch-based checkpoint [4], [5]	Weak	No
Instruction sequence idempotency [2]	Strong	Yes
Task-level idempotency (asymmetric resilience)	Weak	No

Table 1: Checkpoint and restart (CPR) work comparison.

To facilitate the analysis process, we categorize the memory regions that are used by a task into *input, full output, partial output, and input/output memory*. Tbl. 2 gives the detailed description. For instance, a memory region for accumulation belongs to the input/output memory, which violates the task-level idempotency. The partial output means that the task only modifies a subset in the memory region. In this case, an error may corrupt one byte that a correct re-execution does not write (e.g., the histogram calculation where not all bins will be written). Both the input/output and partial output memory require checkpoint.

Further more, we leverage recomputation to reduce the extent of checkpoint requirement in asymmetric resilience, specifically for the input/output and partial output memory. We observe that accelerator tasks usually form a dependency chain. Consider an example of the three tasks in Fig. 2b. The first task has an output region used by the third task as the input/output memory. We can skip the checkpoint for the third task as we can recover it by re-executing the first task. In order to perform such optimizations, the CPU needs to track the dependency among different tasks.

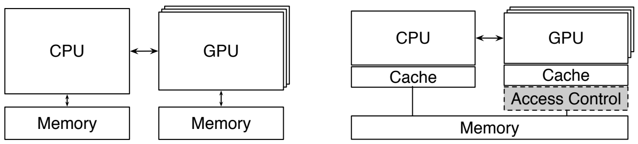
In the following sections, we introduce a set of architecture-runtime co-designed mechanisms to implement asymmetric resilience. At the architecture layer, the memory subsystem provides isolation and memory protection features. The runtime system running on the CPU uses the task-level idempotency with the proposed interface-level categorization framework for checkpoint management. In addition, the runtime can also track the dependency between kernels to optimize the checkpoint process. We demonstrate that those co-designed mechanisms can achieve reliability assurance with minimum overhead.

3 Architectural Support

Our work uses the GPU as the exemplary accelerator to implement and explore asymmetric resilience. We discuss other accelerators in Sec. 6. In this context, the architectural

Memory Type	Description	Permission
Input	Memory region used as input to the kernel	Read
Full output	Memory region that the kernel fully writes to	Read & Write
Partial output	Memory region that the kernel partially writes to	
Input/output	Used as input and output to the kernel	
Other legal	Allocated but unused by the kernel	Read
Illegal	Unallocated and unaccessible memory region	X

Table 2: Categorization of the memory address space.



(a) Discrete memory. **(b)** Integrated memory.

Figure 3: Taxonomy of the different memory subsystems.

components to implement asymmetric resilience include CPU, GPU, and memory subsystem. We focus on the last two components since all the required supports from the CPU are provided by the runtime software.

Augmented Memory Subsystem The memory subsystem provides the protection support, which allows the runtime to set appropriate read/write permission for different memory regions to minimize the checkpoint overhead. We categorize the memory subsystem in today's heterogeneous processors into two types, discrete memory in Fig. 3a and integrated memory in Fig. 3b. Our work focuses on the discrete memory, which requires almost no modification.

We explain how the discrete memory naturally provides the required support, using the example of NVIDIA's discrete CPU-GPU system. First, a GPU kernel cannot directly access the CPU memory (unless using special programming construct or system support) such that its errors cannot corrupt the CPU memory. Second, in the discrete system, the CPU allocates and manages the GPU memory. As such, some portion of GPU memory naturally has a copy on the CPU side. The system can use the copy as a checkpoint, which is equivalent to leveraging memory protection.

Although most high-end GPUs are discrete, the integrated paradigm in Fig. 3b has performance and programmability advantages, which we expect to become more widely adopted in the future. To augment it with the memory isolation and protection support, we need to enhance the memory management unit (MMU) to check read/write permission for every memory access from the accelerator. The permission control can be exposed to programmer via system call like mprotect. The access control unit sits between the accelerator and memory subsystem in Fig. 3b. One efficient design was provided by prior work Border Control [6], which we plan to adopt in the future work.

GPU Error Detection GPUs or other accelerators can borrow CPU-centric error detection techniques because of the identical underlying physical causes for errors. In fact, current high-end GPUs already deploy error correction codes (ECC) for detecting soft errors in register files and caches. However, current GPUs do not perform error recovery for uncorrectable errors. The GPU deems its current state as corrupted, destroys its context, and raises an exception to the CPU [7]. In contrast, we design a more efficient CPU based recovery solution. We choose the GPU kernel as the error detection and recovery unit as it is the smallest control unit by the CPU. Note that asymmetric resilience relaxes the requirement on error detection: the error just needs to be caught before the CPU accesses the data.

4 Runtime Management

Leveraging the architectural support, the runtime manages the checkpoint based on the task (i.e. kernel)-level idempotency property. It also tracks kernel dependency to minimize the overhead in the error-free situation, which is the typical operation mode as an error event rarely happens. Those optimizations significantly improve the system efficiency.

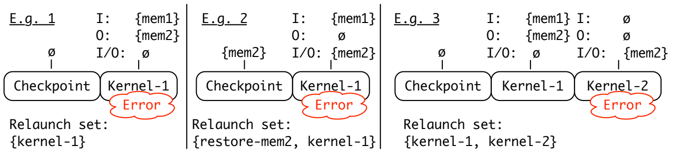


Figure 4: Checkpoint examples. The input, output, and input/output memory are noted as I, O, and I/O.

Checkpoint Management The runtime system manages the checkpoint for each kernel using its memory address information. As described in Tbl. 2, we categorize the memory regions of each kernel into different regions. In our current design, we manually annotate the source codes with each kernel's memory region. We are working on a feasible static compiler pass to automatically determine and annotate that information, with less complexity than prior work [2].

The runtime leverages the task-level idempotency to minimize the checkpoint overhead. It only grants write permission to the output memory (including full, partial, and input/output), leaving all other memory in the read-only mode. Since the full output memory can be restored by re-execution, only input/output and partial output memory require checkpoint. In our experiment, we find that kernels typically lack such memory regions, and therefore they do not require any checkpoint overhead.

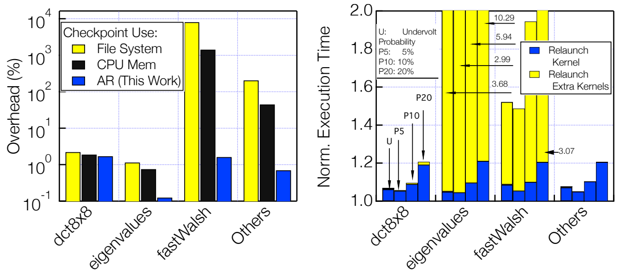
Dependency Tracking The runtime is also capable of tracking the dependency relationship between kernels so that it can eliminate the checkpoint via re-computation. Specifically, it maintains a *relaunch set* for each kernel, which it needs to relaunch for the kernel's error recovery. Examples in Fig. 4 summarize the key points. The erroneous kernel in the first example in Fig. 4 has no input/output memory (noted as I/O) as such its relaunch set is itself. The second example has I/O memory that requires checkpoint, so its relaunch set includes checkpoint restoration and itself. The third example activates the kernel cohesion technique that removes the checkpoint requirement of mem2. As a result, the runtime needs to relaunch the additional kernel-2.

5 Evaluation

We perform a comprehensive evaluation of asymmetric resilience in different aspects. We first emphasize its performance during the error-free execution. We then show that it can also recover from errors with reasonable overhead in most cases. For some cases with high error recovery cost, we evaluate the effectiveness of our mitigation optimizations.

Experimental Setup Because we do not have access to any error checking capability in the used GPUs, we rely on dual module redundancy (DMR) for GPU error detection. We prototype our design in a dual-GPU system where we use one GTX 780 as the primary GPU, and one GTX 680 as the shadow GPU for DMR. We use CUDA 7.0 and the CUPTI library to overcome the closed sourced CUDA APIs, which provides instrumentation features for implementing asymmetric resilience runtime. We study a set of 32 programs from the CUDA SDK. These programs have diverse performance characteristics, which help us make insightful observations and comprehensively evaluate our system.

Error Injection We study two types of GPU transient errors. The first type is the voltage noise errors [8] which



(a) Error free execution. **(b)** Error recovery execution.

Figure 5: Error free and recovery overhead comparison.

we "inject" by operating the chip just below the V_{min}. The second type is the soft error which we inject using the instrumentation tool. We inject errors to a kernel's input, output, input/output, and legal & illegal addresses memory, which gives good coverage of possible failures points, thereby allowing us to test our runtime's efficacy because the failure point impacts the system behavior.

Error Free We evaluate the error-free execution overhead under three different checkpointing scenarios, as shown in Fig. 5a. They differ by which memory requires checkpoint and where to store the checkpoint. The first two scenarios take a checkpoint of the entire GPU memory space, but the checkpoint is stored in the OS file system (File) or CPU memory (CPU Mem). The second (first) case assumes that the CPU memory is (not) safe from GPU errors.

Fig. 5a compares different scenarios' checkpoint overhead, collected from our discrete GPU setup. We show the names of programs dct8x8, eigenvalues, and fastWalshTransform that have kernels with I/O memory. Other regular programs with no I/O memory are grouped in the Others category. The geometric means for the four scenarios are 162%, 38%, 30%, and 0.76%, respectively. The significant checkpoint overhead defeats the purpose of introducing accelerators. ***The result highlights that asymmetric resilience achieves near-zero error-free execution overhead through co-design.***

Error Recovery We evaluate the runtime's error recovery overhead. We only show the cost of relaunching the erroneous kernel and extra dependent kernels in asymmetric resilience because the checkpoint overhead dominates in all other cases. The 'U' bar in Fig. 5b represents the results of undervolting while the following three bars represent the soft error results with different probabilities. The y-axis in the plot indicates the overall execution that is normalized to the error-free execution. Most programs do not rely on kernel cohesion, and therefore ***their recovery overhead increases linearly with the error rate.***

Only the three labeled programs on the x-axis activate the kernel cohesion and relaunch extra kernels. The relaunched extra kernels form a dependency chain (e.g., the third example of Fig. 4), whose length determines the recovery cost. The overhead in dct8x8 is small owing to its short dependency chain while the other two programs have very high recovery overhead because of the long dependency chain. We plan to apply further optimization, which makes a checkpoint if the kernel dependency chain length exceeds a threshold, to reduce the kernel relaunching cost.

6 Other Accelerators & Conclusion

Domain	Accelerator	Idempotency?
Algebra	Convolution Engine [9]	Yes
Neural network inference	Diannao [10]	Yes
	FlexFlow [11]	Yes
Robotics	Motion Planning [12]	Yes
Neural network training	PipeLayer [13]	No
Graph Analytics	Graphicionado [14]	No

Table 3: Different accelerator's relaunch characteristics.

We discuss how asymmetric resilience is a natural reliability optimization for emerging accelerators that extend beyond the GPU. Asymmetric resilience uses the idempotency property of the entire accelerator task to simplify the workflow. To understand this, we surveyed new accelerators, ranging from basic algebra to neural networks to graph analytics. Tbl. 3 shows that the vast majority of surveyed accelerators are idempotent, which we can naturally support. For the non-idempotent neural network training and graph analytics accelerators, the network weights and graph edges are used as both input and output. Thus, our insights from optimizing non-idempotent kernels can be directly applied. In addition, the dependency among those accelerator computations can also impact how to deploy asymmetric resilience efficiently. We leave this study for future work.

Asymmetric resilience is a generic design for coping with transient errors in accelerators. As the industry moves more toward more heterogeneous architectures, we must rethink how to achieve scalable, efficient error recovery mechanisms. We take a step in that direction and show how we can deliver efficient error recovery by centering around the relaunch characteristics of an accelerator.

Acknowledgement This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document is: Approved for Public Release, Distribution Unlimited.

References

[1] D. Tiwari *et al.*, "Understanding GPU errors on large-scale hpc systems and the implications for system design and operation," in *HPCA*, 2015.

[2] S. Feng *et al.*, "Encore: Low-cost, fine-grained transient fault recovery," in *MICRO*, 2011.

[3] J. Chung *et al.*, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *SC*, 2012.

[4] H. Takizawa *et al.*, "Checuda: A checkpoint/restart tool for cuda applications," in *PDCAT*, 2009.

[5] A. J. Pena *et al.*, "Vocl-ft: Introducing techniques for efficient soft error coprocessor recovery," in *SC*, 2015.

[6] L. E. Olson *et al.*, "Border control: Sandboxing accelerators," in *MICRO*, 2015.

[7] "CUDA Error Types," https://bit.ly/2Hvk3DK.

[8] J. Leng *et al.*, "Safe Limits on Voltage Reduction Efficiency in GPUs: a Direct Measurement Approach," in *MICRO*, 2015.

[9] W. Qadeer *et al.*, "Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing," in *ISCA*, 2013.

[10] T. Chen *et al.*, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *ASPLOS*, 2014.

[11] W. Lu *et al.*, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Network," in *HPCA*, 2017.

[12] S. Murray *et al.*, "The microarchitecture of a real-time robot motion planning accelerator," in *MICRO*, 2016.

[13] L. Song *et al.*, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *HPCA*, 2017.

[14] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, 2016.