RC25684 (WAT1807-017) July 30, 2018
Computer Science

IBM Research Report

Asymmetric Resilience: Rethinking Reliability for Accelerator-Rich Systems

Jingwen Leng[1,2], Alper Buyuktosunoglu[2], Ramon Bertran[2], Pradip Bose[2], Vijay Janapa Reddi[1]

[1]Department of Electrical and Computer Engineering
The University of Texas at Austin
2501 Speedway
Austin, TX 78712

[2]IBM T.J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598

Research Division
Almaden – Austin – Beijing – Cambridge – Dublin - Haifa – India – Melbourne - T.J. Watson – Tokyo - Zurich

Asymmetric Resilience: Rethinking Reliability for Accelerator-Rich Systems

Jingwen Leng[1,2]*, Alper Buyuktosunoglu[2], Ramon Bertran[2],
Pradip Bose[2], Vijay Janapa Reddi[1]
[1]*Dept. of Electrical and Computer Engineering, The University of Texas at Austin*
[2] *IBM T. J. Watson Research Center*
[1] leng-jw@sjtu.edu.cn, [2]{alperb, rbertra, pbose}@us.ibm.com, [1]vj@ece.utexas.edu

Abstract

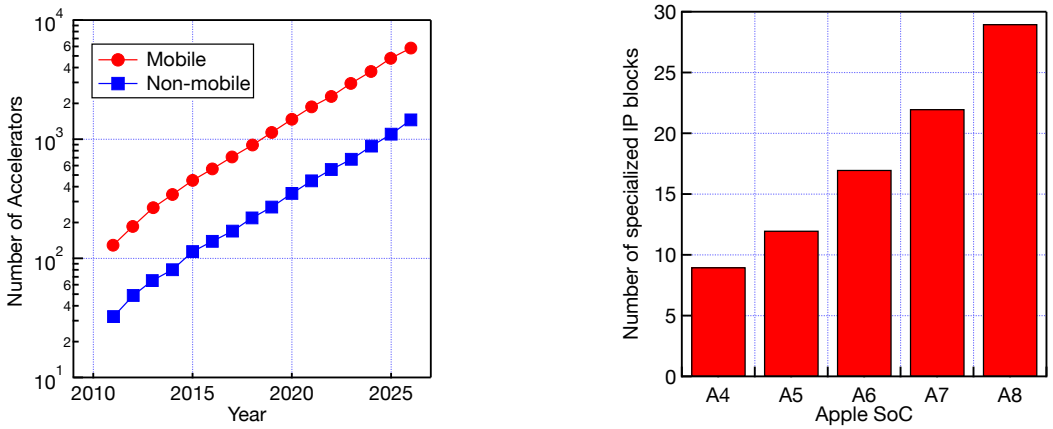
We have already entered the heterogeneous computing era when computing systems harness computational horsepower from not only general purpose CPUs but also other processors such as graphics processing unit (GPU) and hardware accelerators. Performance, power-efficiency, and reliability are three most critical aspects of processors, and there usually exists a tradeoff among them. Accelerators are heavily optimized for performance and power-efficiency rather than reliability. However, it is equally important to ensure overall reliability while introducing accelerators to computing systems.

In this paper, we focus on optimizing accelerator's reliability without adopting the "whac-a-mole" paradigm which develops accelerator-specific reliability optimization. Instead, we advocate maintaining the reliability at the system level, and propose the design paradigm called "asymmetric resilience," whose principle is to develop the reliable heterogeneous system centering around the CPU architecture. This generic design paradigm eases accelerators away from reliability optimization. We present the design principles and practices for the heterogeneous system that adopt such design paradigm. Following the principles of asymmetric resilience, we demonstrate how to use CPU architecture to handle GPU execution errors, which allows GPU focus on typical case operation for better energy efficiency. We explore the design space and show that the average overhead is only 1% for error-free execution and the overhead increases linearly with error probability.

1 Introduction

Performance, power efficiency, and reliability are three most crucial aspects of computing systems. However, the CPU scaling can no longer sustain our increasing demand for per-

*Jingwen is currently a tenure-track assistant professor at Dept. of Computer Science and Engineering, Shanghai Jiao Tong University. This work was completed while the first author was in the United States, working part-time at IBM T. J. Watson Research Center, Yorktown Heights, NY and part-time at U of Texas at Austin as a graduate assistant.



(a) ITRS projection [8]. **(b)** Apple SoC [9].

Figure 1: Trend for number of accelerators.

formance and power efficiency because of the end of Dennard scaling and the diminishing return of microarchitecture enhancement. As such, hardware accelerators [1, 2, 3, 4] are deemed to be the solution to provide continued performance and power efficiency improvements beyond the general purpose CPUs.

Introducing hardware accelerators into computing system can increase the system's error vulnerability. Accelerators are optimized for performance and power efficiency. This usually comes at the price of reduced reliability because of the inherent trade-off across those three metrics [5]. For example, prior works [6, 7] studying GPU errors in large-scale systems have found that the GPU's MTBF (mean time between failures) is almost 8x lower than that of the CPU. As such, it is crucial to maintain computing systems' reliability while introducing accelerators.

As shown in Figure 1, the number of accelerators is increasing at an unprecedented rate. Accelerators specialize in different computation patterns and thus have significantly different architectures. The rich diversity and high count of accelerators make the reliability challenge progressively steeper. Traditional CPU-centric reliability mechanisms would not work because they would defeat the original goal of efficiency that is targeted by the use of accelerators.

To this end, we propose a design paradigm called *asymmetric resilience* to ensure the reliability of heterogeneous systems in the presence of transient accelerator execution errors. The fundamental idea is to relax the resiliency requirement of the accelerator architecture and ensure the system reliability centering around the CPU architecture. In asymmetric resilience, an accelerator only needs to detect the error and report to the CPU, and the system relies on the CPU to recover from the detected error. As such, we can avoid the accelerator-specific reliability optimization, and let designers continue to focus on the accelerator performance or power efficiency optimizations.

We demonstrate the case of applying the asymmetric resilience principles to make a trade-off between energy efficiency and reliability in a CPU-GPU system. Specifically, we optimize the GPU's voltage guardband. Prior works on CPUs showed more than 20% energy saving potential [10, 11] because the nominal voltage is over-provisioned for

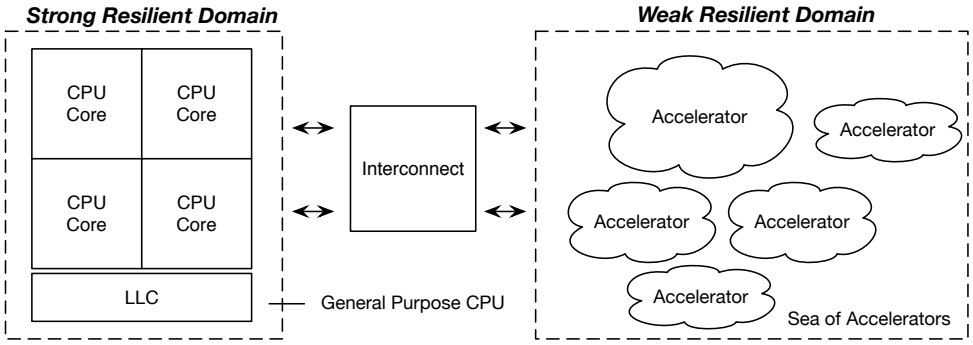


Figure 2: Overview of asymmetric resilience. The fundamental idea is to ensure the system's reliability using the most resilient component. Our studied paradigm has two (strong and weak) resilient domains, which host the CPU and accelerators, respectively. The former is optimized for resiliency and ensures system-level reliability and the latter is optimized for performance/power.

typical case operation. We propose a new design paradigm in which the CPU handles GPU's worst-case condition, and lets the GPU focus on the typical-case operation for better energy efficiency.

We develop a runtime system that uses the CPU to recover from GPU error caused by worst-case voltage conditions. We study the important implications and explore the design space of asymmetric resilience on the basis of a developed testbed system. By studying the error characteristics of such errors and how those errors propagate from the GPU to the CPU, we find that the CPU can recover from the GPU error by simply relaunching the kernel in most cases. We find that 69 out of 81 kernels in studied CUDA programs only require a single relaunch to recover from the detected error. For the remaining 12 kernels, our runtime can automatically decide the minimum set of kernels to relaunch, with a lightweight source-code level annotation. Our runtime system incurs negligible performance overhead in the error-free execution, and we propose several optimizations that can minimize the error recovery overhead.

We make the following contributions in this work:

- We propose the concept of asymmetric resilience, a generic design paradigm that ensures reliability of accelerator-rich systems in the presence of transient accelerator errors. Such a design paradigm relies on the CPU and exempts accelerators from heavy resiliency optimizations (Section 2).
- We demonstrate the case of applying the asymmetric resilience principles to make a trade-off between energy efficiency and reliability in a CPU-GPU system. We show that it is possible to use the CPU to handle GPU errors caused by worst case conditions, which lets the GPU operate at the typical case condition for better energy efficiency (Section 3).
- We show how to use the CPU to recover from GPU errors by implementing an asymmetric resilience runtime system. Our runtime has near zero overhead when no error occurs and we study optimizations to minimize its error recovery overhead (Section 4).

We organize the paper as follows. Section 2 details the principles behind asymmetric

resilience. Section 3 motivates a case of using asymmetric resilience to make a trade-off between energy efficiency and reliability. Section 4 describes our asymmetric resilience runtime system. Section 5 shows our experimental setup for our prototype, and Section 6 evaluates our system. Section 7 discusses current limitation of our system and possible extension. Section 8 compares against related works, and Section 9 concludes the paper.

2 Asymmetric Resilience

In this section, we propose the design paradigm called *asymmetric resilience* for ensuring the reliability of heterogeneous system against accelerator errors. We first describe the fundamental principles of asymmetric resilience, which divide the system to domains with different levels of resiliency and ensure the system reliability using the most resilient domain. We then show practices of applying those principles to build a reliable heterogeneous system. This design paradigm relaxes the resiliency requirement of the accelerator by relying on the CPU to handle the error during accelerator computation.

2.1 Design Principles

In this subsection, we describe the principles of our proposed asymmetric resilience design paradigm for building a reliable heterogeneous system against accelerator execution errors. We first explain the meaning of the term "resiliency" and the term "reliability," respectively. In our terminology, the term reliability means the system is capable of consistently performing computation according to its specifications and generating the expected outcome.

The term resiliency refers to a computing system's capability of recovering from errors. In other words, computing systems need to adopt different resilient techniques for different possible types of errors to achieve the reliability. For example, a computing system can deploy ECC (error correction code) enabled memory for protecting against soft errors, and can leverage RAID (redundant array of inexpensive disks) for protecting against disk failures [12].

Our work aims for the accelerator-independent mechanism to handle accelerator's error given the fact that the number of accelerators is exploding (Figure 1). In contrast, we can build a reliable heterogeneous system by developing resilient accelerators that can detect and recover from errors themselves, which is not desirable because it requires accelerator-specific optimization and can incur significant overhead. For example, recently studied accelerators [1, 2, 3, 4] all have moderate complexity and it is non-trivial to transform them to be resilient against errors. Moreover, not all CPU-centric error recovery mechanisms can apply to accelerators. For example, CPU can treat a transient error in the same way as mis-speculation [10, 13, 14] while emerging parallel architectures like GPU do not support speculation.

Recognizing the trend of increasing number of accelerators and the need for an architecture-independent solution, we propose a generic and low overhead system architecture for designing a reliable computing system in the presence of accelerator errors. The fundamental insight is to use the most resilient computing component to handle

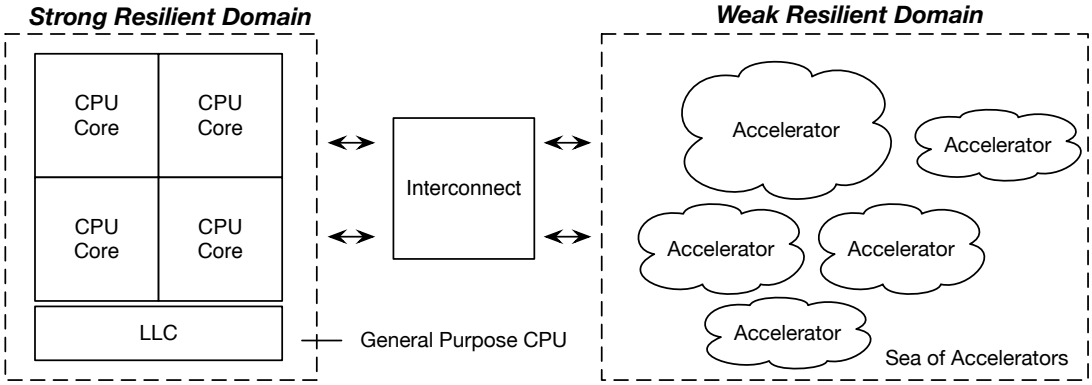


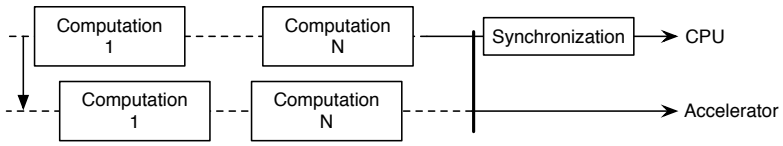
Figure 3: Overview of asymmetric resilience. The fundamental idea is to ensure the system's reliability using the most resilient component. Our studied paradigm has two (strong and weak) resilient domains, which host the CPU and accelerators, respectively. The former is optimized for resiliency and ensures system-level reliability and the latter is optimized for performance/power.

errors from other less resilient components, rather than make all components handle errors by themselves. As a result, the system has multiple domains with different levels of resiliency. As such, we call the proposed system architecture asymmetric resilience.

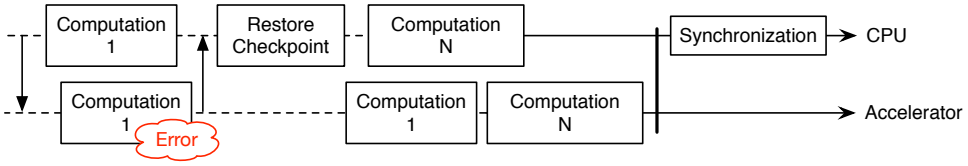
As the first effort for studying asymmetric resilience, we consider a system with two domains shown in Figure 3: the *strong resilient domain* and the *weak resilient domain*. The former domain has a strong resilience requirement: it can detect targeted errors and recover from them. In contrast, the latter domain has a relaxed resilience requirement: it only needs to detect the error. In asymmetric resilience, the system relies on the strong resilient domain to guarantee the system-level reliability in the presence of errors from the weak resilient domain. In other words, the strong resilient domain is the fail-safe mechanism for the entire system.

In the system with asymmetric resilience, it is natural to deploy the accelerators in the weak resilient domain, and the CPUs in the strong resilient domain, as shown in Figure 3. The design paradigm is best applicable to the loosely coupled heterogeneous system. In such system, the accelerator sits outside of the CPU core's pipeline, and they communicate through an interconnect or shared memory. We do not discuss the memory subsystem here because it does not affect the principles of asymmetric resilience. But we will explain how the memory subsystem determines the implementation in the next subsection. In contrast, an accelerator can also be tightly integrated as a special functional unit inside the CPU core pipeline. It is relatively easy to handle the accelerator execution error in the tightly coupled architecture. Because the accelerator is essentially a special functional unit in the CPU pipeline [15], its execution error can be treated in a similar way of handling the pipeline exception.

Our proposed design paradigm can be used to protect the system against transient errors originating from accelerators. External events such as capacitive cross-talk, power supply noise, cosmic rays and alpha particles can cause transient errors, and these constitute a major concern. The required effort of using the CPU in the strong resilient domain is relatively small because the reliability of general purpose CPU architecture



(a) Error free operation with overlapped CPU and accelerator computation.



(b) When the accelerator detects the execution error, the CPU pauses its computation, restores to the previous checkpoint, and re-compute in the accelerator.

Figure 4: The role of CPU in resilience design paradigm. (a) shows an error-free execution, and (b) shows an error recovery process.

in the presence of those transient errors is a well-understood challenge that already has sophisticated solutions [10, 13, 14, 16].

Adopting asymmetric resilience incurs relatively small overhead in the accelerator because the only extra required features are capabilities of error detection and reporting. The error detection is relatively architecture-independent because of the same underlying causes for the errors. The proposed design paradigm avoids the architecture-dependent error recovery procedure by "offloading" such job to the CPU because the CPU serves as the safety-net for the entire system. This allows accelerator designers to keep focusing on their performance or power efficiency optimizations instead of the resiliency.

2.2 Design Practice

In this subsection, we describe how to build a practically reliable computing system following the principles of asymmetric resilience. Specifically, we discuss the role of the accelerator, CPU, and memory subsystem in asymmetric resilience.

The Role of Accelerator Asymmetric resilience only requires accelerator to have the error detection capability. The specific solution depends on the kinds of errors against which the system targets to protect. For example, an accelerator can deploy error correction code (ECC) for protecting against the threat of soft errors in an SRAM array cell or latch [17, 18, 19]. The accelerator can report errors that cannot be recovered by the deployed ECC to the CPU for error recovery. In this work, we target on errors caused by transient supply voltage fluctuation, which requires voltage sensor circuits to detect. We will discuss the details in Section 4.2. But in general, the error detection is a relatively architecture independent process because of the same underlying causes for errors.

The Role of CPU To understand the role of CPU in the asymmetric resilience, we first explain our assumed execution model in the heterogeneous system. We make the assumption that the acceleration computation is asynchronous relative to the CPU computation in the loosely coupled architecture because it results in higher utilization rate and improves system performance. Figure 4a illustrates the execution model where acceleration computation can overlap with CPU computation. Because the accelerator computation is asynchronous, the CPU needs to issue the synchronization command when it wants to use results from the accelerator computation.

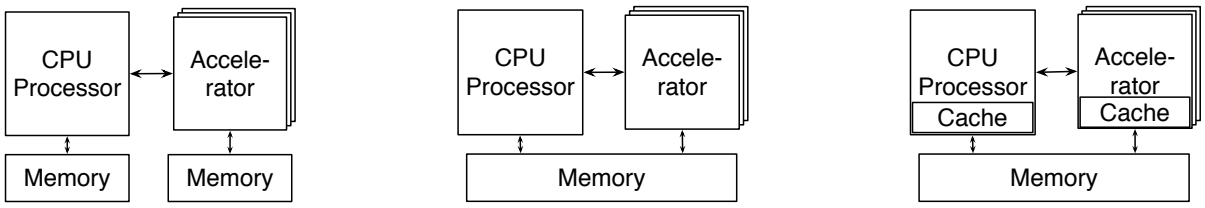
The CPU has two main roles in the asymmetric resilience: *checkpointing* and *error recovery*. First, the CPU needs to make the checkpoint of necessary data to recover from the detected error. Since the asymmetric resilience targets errors that happen in the accelerators, we only need to make the checkpoint of memory that can be written by the accelerator. However, the kind of memory subsystem affects the memory that can be accessed by the accelerator, and hence affects how the CPU needs to make the checkpoint. We will discuss this in detail in the next paragraphs.

Second, the CPU needs to perform the error recovery once the error is detected. Figure 4b illustrates the error recovery process initiated by the CPU upon the detected accelerator error. The CPU first pauses its own computation and restores the memory related to the accelerator computation to the previous checkpoint. After restoring the checkpoint, it then replays all the issued accelerator computations between the previous checkpoint and error detection to recover from the error. The program resumes the CPU computation and the rest of accelerator computation until the synchronization command.

Note that it is not necessary for the program to replay any CPU computation because we assume that the error during accelerator computation would not corrupt the CPU computation before the synchronization point. This assumption requires modification to the memory management unit in the CPU depending on the type of memory subsystem, which we discuss later. With such assumption, the program would have known whether there was an error during the accelerator computation before the synchronization point. Since the CPU computation only uses the results from the accelerator computation after the synchronization point, the CPU computation will always use the error-free accelerator computation result.

The Role of Memory Subsystem The memory subsystem plays a major role in the asymmetric resilience design paradigm. Because the asymmetric resilience does not require accelerators in the weak resilient domains to recover from errors, an accelerator error can corrupt the CPU's data and result in a catastrophic failure. As such, the asymmetric resilience design paradigm necessitates the memory subsystem's error isolation capability that contains an accelerator's execution error in the memory region that it has access to.

We categorize the memory subsystem in the heterogeneous system as shown in Figure 5, and discuss how to augment each memory subsystem kind with the error isolation feature. Figure 5 shows three kinds of heterogeneous system's memory: the discrete memory, the unified memory, and the coherent memory. The CPU and accelerator have



(a) Discrete memory. **(b)** Unified memory. **(c)** Coherent memory.

Figure 5: Heterogenous systems' memory subsystem taxonomy.

separate physical memory in the discrete memory subsystem. In contrast, they share the same memory in the unified memory subsystem. In the coherent memory subsystem, the CPU and accelerator share the same memory subsystem, and each has a coherent cache.

Current discrete CPU-GPU computing system adopts the discrete memory subsystem design. Although the physical memory is separate in such memory subsystem, all the memory can be in the same virtual address space [20]. In this kind of memory subsystem, the program running on the CPU needs to allocate and copy the input data to the accelerator memory, and the accelerator can only access its own memory while performing its computation. As such, this type of memory subsystem naturally ensures that the error on the accelerator does not propagate to the CPU memory.

We observe that the other two kinds of the memory subsystem (unified memory and coherent memory) are also used due to performance and programmability advantages. We require certain modification to achieve the error isolation effect for those two kinds of memory subsystems. The fundamental of error isolation is to prevent the accelerator from accessing an arbitrary memory address. One possible way of achieving that desired functionality is to modify the page table to distinguish the memory access of CPU and accelerator. We will discuss how to extend our work for these two memory subsystems in the discussion section.

3 A Case for Energy and Reliability Trade-Off

In this section, we study a case where we use the asymmetric resilience to make a trade-off between energy efficiency and reliability. Specifically, we demonstrate that following asymmetric resilience, the CPU architecture handles the worst case condition on the GPU and allows the GPU to focus on typical case situation for better energy efficiency. We first demonstrate that adapting the GPU's supply voltage to the typical case condition can significantly improve its energy efficiency compared to the current worst-case guardbanding GPU design. We then show the feasibility of using the CPU to handle the worst case condition in the GPU architecture to address the challenge of ensuring the reliability.

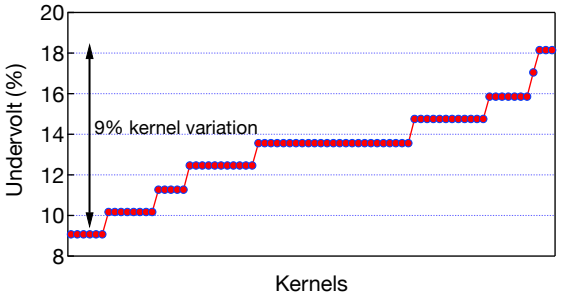


Figure 6: Each marker indicates a kernel's undervolt percent at its V_{min} point. The max. percent is 18% and the min. is 9%.

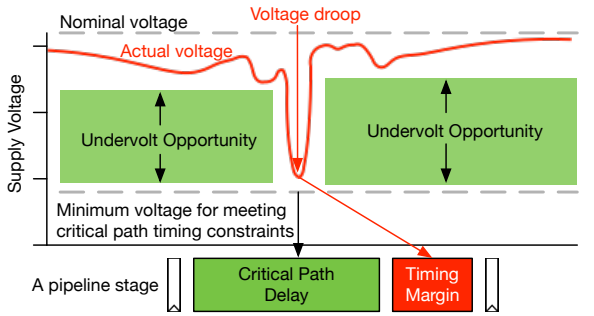


Figure 7: Processors rely on excessive supply voltage margin to protect against voltage droops.

3.1 GPU Typical-Case Voltage Guardband Optimization

We first demonstrate that there is a significant opportunity of adapting GPU's supply voltage to the typical case situation by voltage guardband optimization. Specifically, we push down the supply voltage (i.e. undervolting at the nominal frequency) to the limit without impacting the correctness level assumed by an application developer. The voltage guardband exists to protect against worst case process, voltage, and temperature variation as well as aging. Because worst case condition rarely occurs, we can operate the GPU with a lower voltage most of the time and achieve significant energy savings.

We quantify the voltage guardband optimization opportunity by measuring the V_{min} point of a set of representative GPU programs. V_{min} is the minimum voltage level at which the program executes correctly but fails if the voltage is reduced further below. In other words, V_{min} indicates the minimum working voltage at a fixed frequency. The margin between the nominal voltage and V_{min} level is the optimization potential.

We measure the V_{min} point for each individual kernel in the program. We gradually increase the GPU processor undervolt percent level when running each kernel and stop the experiment when the kernel starts to experience an error. We conduct such experiment in a testbed developed in this work with error checking capability, the details of which will be explained in Section 4. Briefly, we compare the GPU's allocated memory state with a "golden" state reference after each kernel execution. We consider the kernel executed correctly if memory states from the undervolt execution and golden reference are identical at the byte-level. We run all programs 1000 times at their measured V_{min} points to ensure statistically sound results.

Our measurement indicates that the current GPU voltage is significantly over-provisioned due to the worst-case guardband methodology. Figure 6 shows the margin between the nominal voltage and kernel V_{min} point on a GTX 680 card with Kepler architecture. The voltage stock setting of studied GTX 680 is 1.09 V at a frequency of 1.1 GHz. We find that up to 18.3% of the nominal voltage can be reduced without affecting the kernel's functional correctness.

We also observe a large variability in the studied kernel's V_{min} values. In other words, a kernel's V_{min} value strongly depends on its characteristics. As indicated in Figure 6, the

maximum undervolt percent for a kernel (18%) is twice of the minimum value (9%). Such strong kernel dependent guardband requirement necessitates a kernel characteristics aware guardband management. Otherwise, half of the optimization opportunity would be wasted: our measured system-level energy saving with 18% undervolt is 25%, but only 12% with 9% undervolt. Note that the system-level energy includes DRAM and peripherals, which are not undervolted.

3.2 CPU Worst-Case Voltage Guardband Handling

In this subsection, we present our motivational results and analysis for the design paradigm that GPU adapts to the typical case condition while the CPU handles its worst case condition following asymmetric resilience principles. Prior work on the GPU guardband study has shown how to adapt to a kernel's V_{min} by using performance counters based V_{min} prediction model [21], but has not studied how to handle the worst case condition. Our work studies how to use the CPU architecture to handle the worst case condition in the GPU architecture.

We first analyze the characteristics of GPU error caused by the worst case event to study how to handle it. We specifically study errors caused by voltage noise because prior work [21] shows that a significant portion of the voltage guardband is allocated to protect against voltage noise and it also determines a kernel's V_{min} value. Voltage noise refers to the constantly varying supply voltage as shown in Figure 7. It is the interaction result between the processor's non-zero impedance power delivery network and continuously varying current consumption [22]. For example, a sudden current surge after a pipeline stall caused by cache miss can cause the voltage droop below its nominal value. This also explains why the V_{min} is program dependent. Designers must add a large enough guardband to cover the worst-case voltage droop magnitude because it can slow down the circuit and cause a timing error.

We perform a comprehensive study on the error characteristics and how the GPU error affects the CPU by operating each kernel below its V_{min} point. Since we compare the entire allocated memory spaces in the GPU using our developed testbed, we can study how a timing error corrupts the memory spaces in a very fine-grained level. Figure 8 shows the aggregated kernel execution results when operating below its V_{min} point. We first observe that the kernel still has a probability of executing correctly even though operating below its V_{min} point, which was also observed by prior study [21].

We observe two major kinds of error events, which are memory output corruption and illegal memory access, as shown in Figure 8. The memory output corruption means that the kernel runs to completion but the output is different from the golden reference. When this occurs, the program finishes execution without any warning but produces an incorrect end result, which is commonly referred to silent data corruption (SDC) [23]. The illegal memory access means that the kernel execution reads or writes to an illegal memory address due to the error caused by large voltage droop. The illegal memory access will result in an explicitly reported CUDA runtime error. We summarize two important implications:

Implication 1: We observe that the error caused by voltage droop only corrupts the kernel's output memory, and does not corrupt the kernel's input memory. The possible

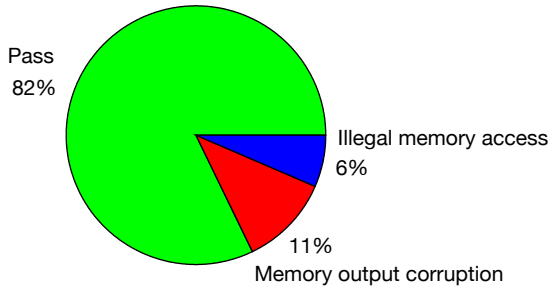


Figure 8: The distribution of runs that result in pass, memory output corruption, and illegal memory access when operating below the V_{min} point.

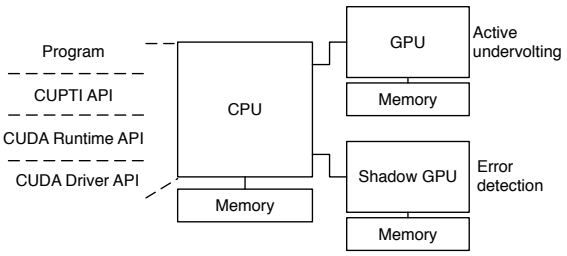


Figure 9: The multi-GPU system experimental setup used for prototyping the asymmetric resilience.

explanation is as follows. An input memory corruption means that an error at the address calculation results in the input memory address space, which is much smaller compared to the entire memory space. As such, an address calculation error most likely leads to an illegal memory address. The GPU itself (possibly its memory management unit) can catch the illegal memory access and raise a CUDA runtime error.

Implication 2: We find that a GPU error does not affect the CPU's states until it accesses GPU's erroneous memory. Although we find that the GPU error can cause an incorrect end-result, the CPU's state can only be corrupted when the CPU needs to access the GPU results. The error in the GPU could also possibly lead to an illegal access to the CPU memory, which will be caught as an illegal memory access error.

In summary, we observe that the memory output corruption and illegal memory accesses are most dominant error events when the kernel operates below the V_{min} point. We make the fundamental observation that the GPU error does not affect the CPU's states if the error is detected promptly. Such observation motivates us to use the CPU to recover from the GPU error using the principles of asymmetric resilience. In our experiment, we observe that an OS crash might occur when operating below the V_{min}. But it does not occur until an additional 4-5% undervolt percent below the V_{min} point, which leaves us enough error margin for avoiding the OS crash.

4 Asymmetric Resilience Runtime Design

Having demonstrated the possibility of using the CPU to recover from the GPU execution error, we present our runtime system design following the asymmetric resilience principles. We first present an overview of a testbed system on which we implement our runtime system. The testbed implements an intelligent and automatic redundant kernel execution that mimics GPU's error checking capability. We then present the details of our runtime system running on the CPU that can recover from GPU errors by automatic relaunch for most of the kernels. For kernels that we cannot automatically relaunch, we propose a solution to find the minimum set of kernels to relaunch with the annotation of the kernel's input and output.

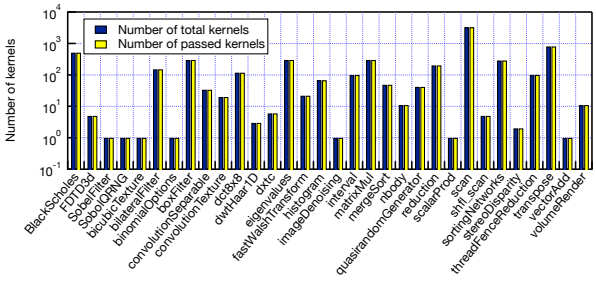


Figure 10: All programs are deterministic because they pass the byte-level comparison.

Category	CUDA runtime API
Allocation	cudaMalloc, cudaFree, cudaMallocPitch, cudaMallocArray, cudaMalloc3DArray
Copy	cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray cudaMemcpy2DToArray cudaMemcpyToSymbol, cudaMemset
Binding	cudaBindTexture cudaBindTexture2D, cudaBindTextureToArray cudaUnbindTexture

Table 1: Supported CUDA runtime functions.

4.1 Testbed Design

In this subsection, we describe the details of a testbed system that we developed for studying the GPU execution error protection and recovery mechanism. The asymmetric resilience requires the error detection capability in the GPU (Section 2.2). Because we do not have access to any error checking capability in the studied GPUs, we choose to use the dual module redundancy (DMR) to detect the possible GPU execution error. Note that this apparatus is used for illustrative purposes only. If we had access, we would rely on specially designed sensors for error detection inside GPU.

In our testbed, we develop a DMR runtime system which is capable of i) maintaining an identical state of the original GPU in an extra shadow GPU; ii) launching an identical shadow kernel from the original GPU in the shadow GPU; iii) comparing the results from the original kernel and shadow kernel to detect the possible execution error. Figure 9 shows the overview of the developed testbed, for which we use a multi-GPU system. The program runs on the original GPU and our runtime implements the DMR using the shadow GPU.

We implement our runtime system at the CUDA runtime API level as shown in Figure 9, which is the programming interface between CPU and GPU [24]. For example, developers use the CUDA runtime API to allocate and copy memory in the GPU. The CUDA driver API shown in Figure 9 is the other alternative of implementing our testbed [25]. We choose the runtime API because it provides a higher-level abstraction.

We use the software library CUPTI (CUDA profiling tools interface [26]) as shown in Figure 9. The CUPTI library provides two important instrumentation features for implementing the DMR based error detection in the closed sourced CUDA runtime APIs. First, it allows user-defined callbacks at the entry and exit point of each CUDA runtime function, which we use to monitor and control the original program's execution flow. Second, each callback also provides the original arguments of the CUDA runtime function, which are required for maintaining the shadow state and performing shadow computation for the DMR based error detection.

In order to implement the DMR, we first need to maintain a shadow memory state in the shadow GPU that must be identical to the original GPU card. This is required because the memory allocated through CUDA runtime functions belong to the pre-selected device, and a GPU card cannot access the memory allocated to the other GPU card. In

other words, the shadow GPU must have its identical copy of memory to launch a kernel to perform the redundant computation.

We develop an efficient approach for creating an identical state in the shadow GPU. Our approach intercepts CUDA runtime functions in the original GPU execution, and repeats certain functions in the shadow GPU. This allows us to maintain an identical memory state because a program can only modify GPU's memory states via CUDA runtime functions. We categorize the runtime functions into three kinds: memory allocation/deallocation, memory copy, and memory binding.

When the program executes the runtime function in any of the three categories, our runtime system intercepts the runtime function, reads the original arguments, and repeat the same operation in the shadow GPU with the CUPTI's callback based instrumentation feature. For example, whenever the program allocates the memory in the original GPU or copies memory from the CPU to the original GPU, we perform the same operation in the redundant GPU. We implement all the runtime APIs in Table 1 that are sufficient for our studied programs.

Our runtime system directly launches the same kernel to the redundant GPU because it can maintain the identical memory state. This allows us to perform the DMR at the kernel level. A kernel launch in CUDA comprises of a series of runtime functions. The first is cudaConfigureCall, which specifies the grid and block dimensions for the kernel launch. The second is cudaSetupArgument, which pushes an argument with specified bytes onto the top of GPU execution stack. Since it only pushes an argument a time, a kernel with multiple argument needs to call this function multiple times. The third is cudaLaunch, which launches the kernel function on the GPU. For the first and third runtime function, we execute them in the redundant GPU with the same argument as in the original GPU. We substitute the arguments of cudaSetupArgument that point to the allocated memory address in the original GPU with the memory address in the shadow GPU.

Our developed DMR runtime system allows us to perform the error checking at the end of each kernel's execution. After both the original and shadow kernel complete the execution, we copy their memory back to the CPU and perform a byte-level comparison using the utility memcmp [27]. The byte-level comparison is relatively strict for detecting the SDC error because the program may use floating points or be non-deterministic. We verify if this is the case by executing the original kernel and shadow kernel at the nominal voltage and comparing the number of total kernels and number of kernel that passes the byte-level comparison. Figure 10 shows such comparison for our programs running 100 times. As it shows, all kernels in all programs pass the byte-level result comparison, suggesting all studied kernels are deterministic.

4.2 Error Recovery Using the CPU

In this subsection, we describe how we apply the principles of the asymmetric resilience to use the CPU architecture recover from the GPU execution error. Our proposed mechanism can be mostly generalized to recover from errors in accelerators. Specifically, there are three primary functions required in the asymmetric resilience: the error detection, checkpointing, and error recovery. In our prototype, we implement all the three functions in our developed testbed using the CUDA runtime APIs. Our current imple-

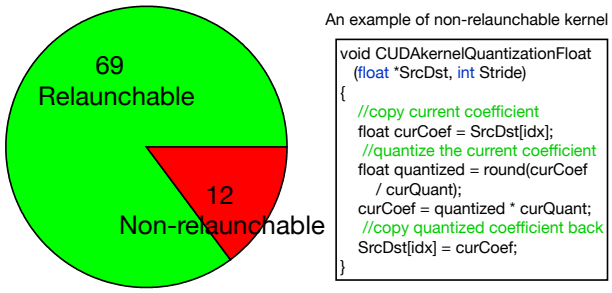


Figure 11: Comparing number of kernels that can and cannot recover from execution errors by relaunching.

mentation of error detection incurs a large overhead due to the lack of error checking capabilities in our studied GPUs. The checkpointing and error recovery are generically applicable to other types of accelerators.

We first explain the targeted error in our system. We then show that in the most commonly seen cases, we can simply relaunch the kernel using the CUDA runtime running on the CPU to recover from the GPU kernel execution error. In the case where the kernel relaunch cannot be recovered, our proposed mechanism requires a lightweight annotation of each kernel's input and output, and leverages the existence of *implicit checkpoint* in the CPU memory to recover from the GPU error. We describe those details in the following paragraphs.

Targeted Error In this work, we target on the error caused by transient voltage droops, which we detect through the kernel-level DMR. However, the ideal detection mechanism is to use voltage sensors. The voltage sensors that can be used for droop detection include the skitter circuit [28, 29], critical path monitor [30, 31], and tune replica circuit [32]. Alternatively, the voltage droop can also be detected by detecting the timing error such as Razor shadow flip-flops [10]. Those sensors are much more lightweight than kernel-level DMR.

Our runtime system aims to recover from memory output corruption error when the kernel operates below the V_{min} point. Although our error analysis in Section 3.2 showed that illegal memory accesses can also happen, such error causes CUDA runtime error. Current CUDA error handling mechanism makes the error recovery unnecessarily difficult because the runtime mandates a GPU reset: the error during the previous kernel execution persists for all following kernel execution. But a GPU reset destroys all previously allocated memory and all JIT-compiled codes, which basically requires a program restart to recover from the error. Our error recovery mechanism can be extended to handle CUDA runtime errors if future software allows better error handling. Otherwise, sensors based error detection mechanism can prevent the error from happening by setting a "soft" threshold [33].

Relaunchable Kernels In our prototype, we use the CPU to recover the GPU execution error at the kernel level because the kernel is smallest control granularity at the CPU

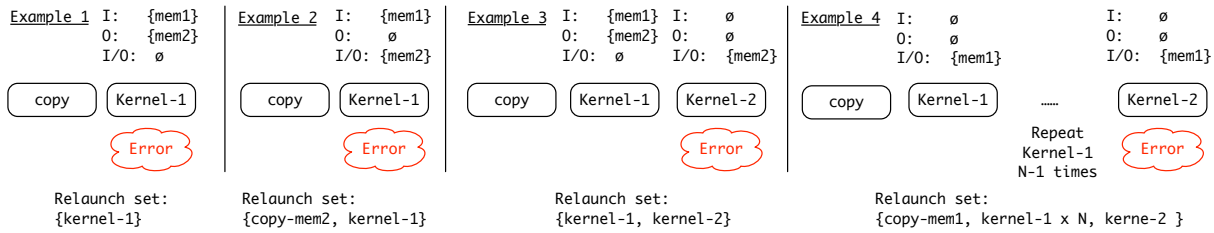


Figure 12: Examples of determining which kernels to relauch for recovering errors. The input, output, input/output memory are noted as I, O, and I/O, respectively. The kernel in Example 1 is relaunchable because it has no I/O memory. The rest three examples involve non-relaunchable kernels, for which runtime needs to copy CPU data and relaunch a set of dependable kernels.

side. Ensuring the reliability at the kernel level also greatly simplifies the checkpointing process, which usually incurs a large runtime and storage overhead. In our system, we only need to make the checkpoint for arguments for relaunching the previous kernel(s). We avoid making checkpoint of architectural states during kernel execution which otherwise could incur signifiant overhead owing to the thousands of concurrently running threads in the GPU.

We find that recovering from kernel execution errors can be achieved by relaunching the kernel computation in most cases. We define such kernel as *relaunchable* because its error can be recovered by relaunching. We observe that most kernels have well-defined input and output memory addresses. Our error characteristics analysis (Section 3.2) showed that the error caused by a transient voltage droop does not corrupt the input address space. As such, the CPU can re-issue the kernel computation to the GPU to recover from the detected error.

Non-Relaunchable Kernels We also find that there exists kernels that cannot recover from errors by relaunching, which we define as *non-relaunchable*. Figure 11 shows the comparison between number of relaunchable kernels and number of non-relaunchable kernels in studied programs. We find that only 12 out of 81 kernels are non-relaunchable, which can be further categorized into the following two scenarios.

In the first scenario, the kernel is not idempotent, which means a re-execution of the kernel will lead to a different result [34]. For example, a kernel has an argument pointing to a memory region that is used as both input and output, and performs increment operation for the memory region. Simply relaunching the kernel will cause an incorrect result. Figure 11 also shows an example of such non-relaunchable kernels. In the second scenario, the kernel is idempotent but the correct execution only updates a portion of its memory output region. For example, a kernel calculates the histogram for an array stored in an input memory region and writes the results to an output memory region. An execution error can cause the kernel to update the wrong bin in the histogram. Relaunching this kernel cannot recover the error because the correct execution cannot overwrite the error in the incorrect bin.

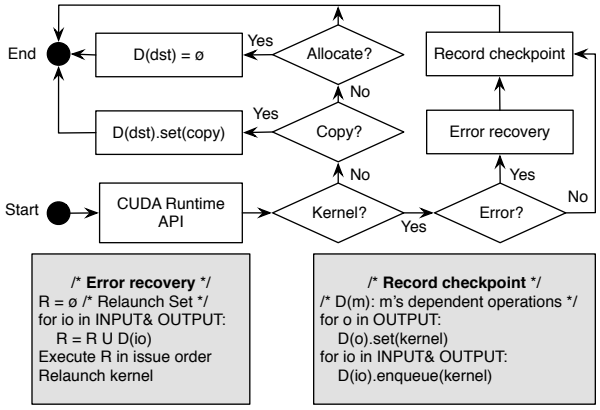


Figure 13: Flow chart of runtime system for handling GPU error.

Implicit Checkpoint Based Recovery We can leverage the *implicit checkpoint* in the CPU memory to handle errors in non-relaunchable kernels. The implicit checkpoint exists because the CPU-GPU heterogenous system adopts the discrete memory subsystem. In such system, the CPU explicitly manages the GPU's memory: the CPU allocates the memory in the GPU, and copy data from its own memory to the GPU for computation. Owing to the natural error isolation effect of discrete memory in which the error from the GPU cannot propagate to the CPU (Section 2.2), the CPU memory that the GPU copies data from can be used as a checkpoint for the corresponding memory region in the GPU. We call such checkpoint as implicit checkpoint because it is not explicitly made by our runtime system.

Besides using the implicit checkpoint, our runtime system also needs to track the dependency between kernels. For example, some programs launch multiple kernels after copying data from CPU to GPU, where each kernel uses the data for both input and output, forming a dependency chain. When a kernel in the middle of the dependency chain encounters an error, its input is the intermediate results from its previous kernel. In this case, we need to relaunch a set of kernels before the dependency chain to restore the input memory.

Figure 12 illustrates the process of determining the minimum set of kernels to relaunch. The kernel in the first example is relaunchable because it has no input/output memory (noted as I/O), as such the relaunch set is itself when it experiences error. In the second example, the kernel has an I/O memory region. To recover from the kernel's error, the runtime needs to first restore the implicit checkpoint by re-copying the CPU data and relaunch the kernel. In the third example, *kernel-2* has an I/O memory, which is the output from *kernel-1*. The runtime needs to relaunch both kernels to recover from the error of *kernel-2*. In the fourth example, there is a series of kernels that have I/O memory, and the runtime needs to relaunch all those kernels for recovery. In the last case, there is trade-off between using the implicit checkpoint and explicit checkpoint depending on the error probability and the length of the dependency chain. We explore such design space.

Because current GPU programming language does not allow programmers to specify

the input, output, and input & output memory regions, we manually annotate all the 12 non-relaunchable kernels for their input, output, and input & output regions. Such process can be automated with a compiler analysis pass, which we leave as future work. Using these annotated information, our runtime system can dynamically track the dependency between kernels and relaunch the smallest set of kernels to recover from error. Figure 13 summarizes the overall flow of our runtime that can recover from GPU errors.

5 Experimental Setup

We discuss the hardware setup and software infrastructure used in this work for prototyping the asymmetric resilience.

Hardware Setup We build our testbed using a multi-GPU system as previously shown in Figure 9. All our studied programs only use one GPU card, and we use a redundant GPU card for error detection. Table 2 lists the key microarchitectural specifications of the two studied GPU cards [35]. The CPU used in this study is an Intel Core i5 processor.

Software Infrastructure We use CUDA 7.0 and the CUPTI version 7.0. We use the MSI Afterburner [36] to control the GPU chip's voltage at a fixed frequency. The granularity for controlling the voltage is 12 mV. We do not modify the memory frequency and voltage.

CUDA Programs We study a set of 32 programs from the CUDA SDK benchmark suite version 7.0 [37]. The list of programs can be found in Figure 14. These programs have diverse performance characteristics, which help us make insightful observations and comprehensively evaluate our system.

6 Evaluation

In this section, we evaluate the overhead of asymmetric resilience runtime system, and demonstrate that it incurs negligible overhead when no error occurs. We also show that our runtime system can recover from high probability errors with reasonable overhead in most cases. We then explore the design space of combining explicit checkpoint in our system to further minimize the recovery overhead.

We first evaluate the overhead of asymmetric resilience runtime when no error occurs. Figure 14a first shows the original program's execution time between kernel execution

GPU	Architecture	Core Counts	Core Clock (MHz)	Memory Clock (MHz)	Register Per Core (KB)	L1 (KB)	L2 KB
GTX680	Kepler	8	1100	3004	256	48	512
GTX780		12					1536

Table 2: GPU cards' microarchitectural specifications.

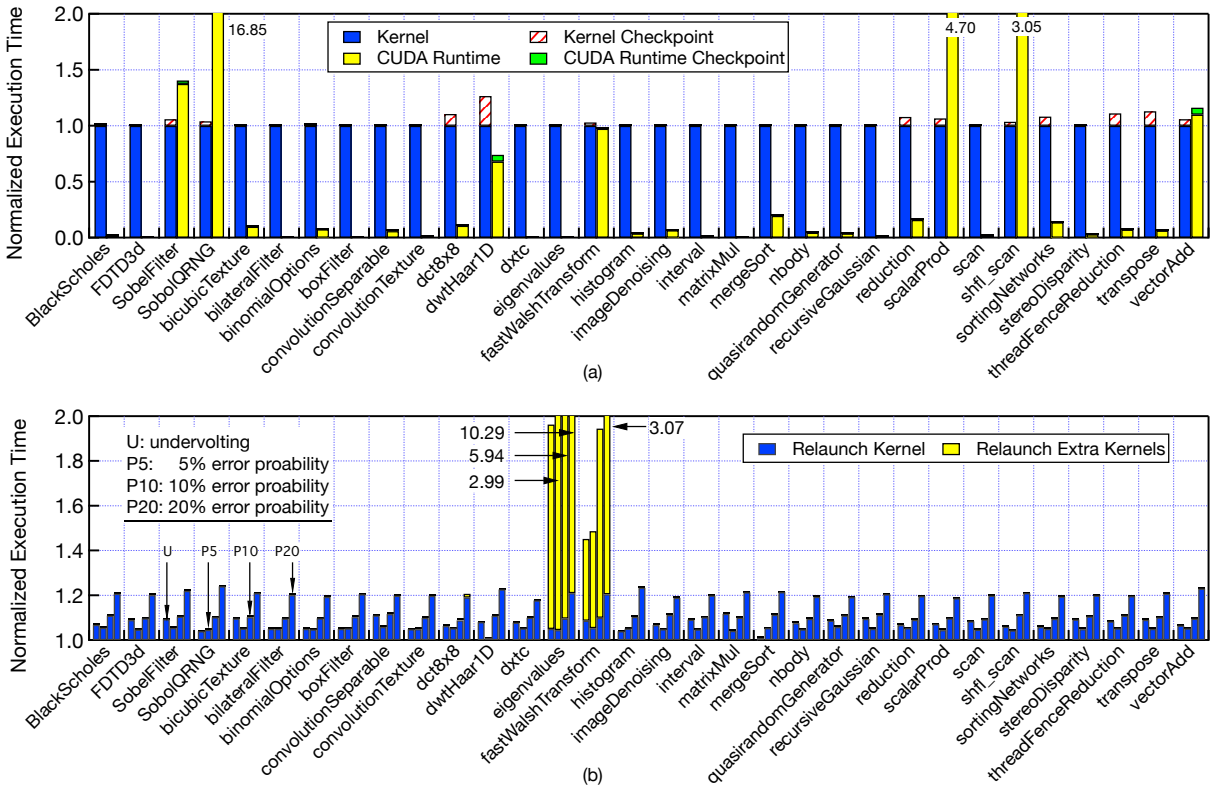
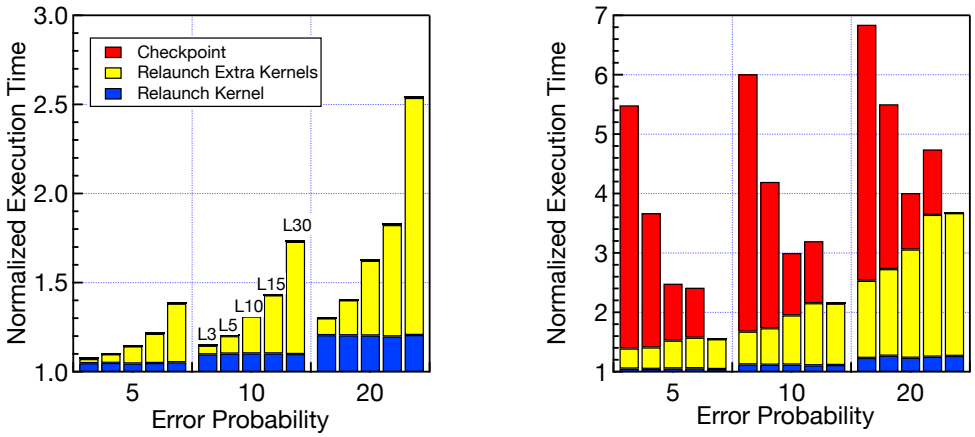


Figure 14: (a) Overhead when no error occurs. (b) Error recovery overhead of our runtime system in different error scenarios: 'U' represents operating below V_{min} value, and "P5" represents 5% output error injection probability.

and CUDA runtime functions, which are both normalized to the total kernel execution time. Most programs spend their majority of time in the kernel execution while some programs spend most of their time in CUDA runtime functions. For example, SobolQRNG spends 16.85× kernel execution time in CUDA runtime functions. Figure 14a then shows the overhead for both the kernel execution and CUDA runtime functions which our runtime keeps track of. On average, the kernel execution overhead is only about 1%. The average CUDA runtime functions overhead is 0.6%. Programs such as dwtHaar1D has more than 10% kernel execution overhead because its kernel execution time is much shorter compared to our runtime overhead. In summary, our runtime system incurs negligible performance overhead because we only keep track of operations such as kernel execution and data copying but rather than the actual memory content.

We then evaluate the overhead of asymmetric resilience runtime system to recover from errors. We have two error injection mechanisms for evaluating the robustness of our prototype. We intentionally operate the kernel below its V_{min} (i.e. undervolt) level as one of the two error injection mechanisms. The undervolt is a realistic cause of GPU execution errors but it can cause illegal memory access and therefore CUDA runtime errors, which requires a restart of the program (Section 4.2). As such, we use a second error injection approach that directly corrupts the kernel execution output to fully test the error recovery ability of our system.



(a) eigenvalues. **(b)** fastWalshTransform.

Figure 15: Execution time when bounding dependency chain length. "L3" means when the chain length becomes larger than 3, we make an explicit checkpoint in the CPU.

Figure 14b shows the normalized execution overhead under different error scenarios. The first bar indicated by 'U' represents the scenario when operating 12 mV below the V_{min}. The next three bars ("P5", "P10', "P20") represent the memory output error injection with probability of 5%, 10%, and 20%, respectively. We categorize the recovery overhead into two kinds: relaunching the erroneous kernel itself and relaunching extra kernels that the erroneous kernel is dependent on. We find that most programs do not need to launch extra kernels, and their recovery overhead increases linearly with the error probability. The only three programs that need to relaunch extra kernels are dct8x8, eigenvalues, and fastWalshTransform. The recovery overhead for dct8x8 is still small because it has only a non-relaunchable kernel whose execution time is small too. However, the overhead for eigenvalues and fastWalshTransform are very high, because they have mostly non-relaunchable kernels which form a long dependency chain.

We find that explicit checkpoint can be used to minimize the recovery overhead for programs with long kernel dependency chain. By making explicit checkpoint, the kernel dependency chain becomes a single memory copy instead of a series of kernel computation. We explore the design space of bounding the kernel dependency chain with different sizes, and show the results for eigenvalues and fastWalshTransform in Figure 15. The digit after letter 'L' indicates the length of the dependency chain. For example, in the "L3" case, the length of the dependency chain is bounded to be 3: when the chain size becomes larger than 3, we make an explicit checkpoint by copying data back to the CPU. We vary the size of chain from 3 to 30.

Our results unveil an interesting observation that the optimum chain size depends on the program characteristics. As shown in Figure 15a, eigenvalues prefers short chain size because its time of copying the checkpoint relative to its kernel execution time is very small. In contrast, fastWalshTransform in Figure 15b prefers a large chain size because of high checkpoint copy overhead. This observation suggests that the runtime can use the ratio between past kernel execution time and data copying time to decide the optimal chain length.

7 Discussion

In this section, we discuss current limitation of our system. We believe that our work is the first step towards making asymmetric resilience more practical and our work can be extended to a much broader scope.

GPU Architecture Our system currently only tracks CUDA runtime function, but can be extended to support CUDA driver functions. Such extension will allow our system to support more programs. We find that some programs use CUDA libraries such as CUFFT [38]. Those libraries internally use driver functions, which our current system does not support.

Our system does not support CUDA unified memory [20] and dynamic parallelism [39]. We believe that it is possible and straightforward for extending our work to support those features. CUDA unified memory allows automatic data copy between CPU and GPU. Our work requires the capability to intercept the data copy operation to support this feature. CUDA dynamic parallelism allows a thread in a kernel to launch another kernel. We can use a tree structure instead of the queue used in our system for maintaining kernel dependence and deciding the minimum set of kernels to relaunch.

Extending to Accelerators Although our work focuses on the CPU and GPU system with discrete memory, many insights can be extended to CPU and accelerator system with unified/coherent memory (Section 2.2). One of the most needed feature is the memory access isolation mechanism such as preventing the accelerator from accessing the CPU memory, and preventing the accelerator from corrupting read-only input memory. Prior works on security or shared memory access control for better performance can be extended for such purpose [40, 41, 42, 43], which we leave as future work. Although the implicit checkpoint does not exist in the unified/coherence memory, the trade-off of making explicit checkpoint revealed in our study is still applicable.

8 Related Work

In this work, we propose a generic design paradigm called asymmetric resilience for recovering accelerator errors using the CPU architecture. We also demonstrate how to follow its principles by using CPU architecture to recover from GPU errors caused by transient voltage droops. We compare and contrast our work with prior works on the error recovery and guardband optimization in the CPU and GPU/accelerator.

Error Recovery Prior work [44] proposed containment domain which aims to prevent errors in one part of the system from affecting others. [45] proposed asymmetric reliability for designing multi-core architecture with different reliability levels for probabilistic applications. Our work targets heterogeneous systems with CPU and accelerators and focuses on using CPU to recover from accelerator errors.

There have been works targeting soft error detection and recovery in CPUs [17, 46, 16, 47, 48, 49, 50] as well as GPUs [51, 52]. The CPU-centric works can be used to harden

CPUs in asymmetric resilience, and our work can be extended for handling on-chip soft errors. Prior works [53, 54, 55] have studied detection and recovery for other error types in GPUs. The error detection part applies to our work, and our error recovery is fundamentally different, for which we rely on the CPU instead of the GPU itself. Prior works [56] have used the concept of asymmetry for performance or energy optimization, but our work applies it to reliability optimization.

Guardband Optimization Prior works [11, 10, 57, 58, 59, 60, 61, 62, 63, 64] studied guardband management in the CPU. There have also been prior works [65, 66, 21, 67, 68] that emphasized the worst-case mitigation or typical case adaptation on GPU itself. Our work studies how to use the CPU to handle the worst-case GPU condition.

9 Conclusion

In this work, we demonstrate a runtime system design that uses the CPU to recover from GPU errors caused by transient voltage droops. Our runtime has near zero overhead when no error occurs due to our insightful observation on the error characteristics and how the GPU error propagates to the CPU. We also study optimizations to minimize its error recovery overhead. By using the CPU handling the GPU's worst case condition, the GPU can operate at typical condition and improve its energy efficiency significantly. We generalize such design to the concept of asymmetric resilience, a generic design paradigm that ensures reliability of accelerator-rich systems in the presence of transient accelerator errors. We believe that asymmetric resilience is a promising direction.

Acknowledgement

This research was developed in part 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] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency and flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[3] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[4] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[5] D. Kerbyson, A. Vishnu, K. Barker, and A. Hoisie, "Codesign challenges for exascale systems: Performance, power, and reliability," *Computer*, vol. 44, pp. 37–43, Nov 2011.

[6] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 331–342, Feb 2015.

[7] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 610–621, June 2014.

[8] International Technology Roadmap for Semiconductors, "System Drivers, 2011 Edition." `https://goo.gl/2SArhv`.

[9] Y. S. Shao, B. Reagen, G. Y. Wei, and D. Brooks, "The aladdin approach to accelerator design and modeling," *IEEE Micro*, vol. 35, pp. 58–70, May 2015.

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

[11] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, and J. B. Carter, "Active Management of Timing Guardband to Save Energy in POWER7," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[12] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, (New York, NY, USA), pp. 109–116, ACM, 1988.

[13] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, 1999.

[14] M. S. Gupta, J. A. Rivers, P. Bose, G. Y. Wei, and D. Brooks, "Tribeca: Design for pvt variations with local recovery and fine-grained adaptation," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 435–446, Dec 2009.

[15] Y. Zhu and V. J. Reddi, "Webcore: Architectural support for mobile web browsing," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 541–552, June 2014.

[16] S. Kim, "Reducing area overhead for error-protecting large l2/l3 caches," *IEEE Transactions on Computers*, vol. 58, pp. 300–310, March 2009.

[17] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, pp. 243–247, Feb 2005.

[18] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Dependable Systems and Networks, 2004 International Conference on*, pp. 61–70, June 2004.

[19] D. H. Yoon and M. Erez, "Flexible cache error protection using an ecc fifo," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, Nov 2009.

[20] M. Harris, "Unified Memory in CUDA 6." `devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/`.

[21] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe Limits on Voltage Reduction Efficiency in GPUs: a Direct Measurement Approach," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015.

[22] D. Ayers, "Microarchitectural Simulation and Control of Di/Dt-induced Power Supply Voltage Variation," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2002.

[23] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak, "Silent Data Corruption: Myth or reality?," in *Proc. of the International Conference on Dependable Systems and Networks With FTCS and DCC*, 2008.

[24] NVIDIA, "CUDA Runtime API." http://goo.gl/G27upA, 2015.

[25] NVIDIA, "CUDA Driver API." http://docs.nvidia.com/cuda/cuda-driver-api, 2016.

[26] NVIDIA, "CUDA Profiling Tools Interface." http://goo.gl/nbAVCf, 2015.

[27] "memcmp." `http://www.cplusplus.com/reference/cstring/memcmp/`.

[28] R. L. Franch, P. Restle, J. K. Norman, W. V. Huott, J. Friedrich, R. Dixon, S. Weitzel, K. van Goor, and G. Salem, "On-chip timing uncertainty measurements on IBM microprocessors," in *International Test Conference*, pp. 1–7, 2008.

[29] P. Restle, R. Franch, N. James, W. Huott, T. Skergan, S. Wilson, N. Schwartz, and J. Clabes, "Timing uncertainty measurements on the power5 microprocessor," in *International Solid-State Circuits Conference*, pp. 354–355 Vol.1, Feb 2004.

[30] A. Drake, R. Senger, H. Deogun, G. Carpenter, S. Ghiasi, T. Nguyen, N. James, M. Floyd, and V. Pokala, "A Distributed Critical-Path Timing Monitor for a 65nm High-Performance Microprocessor," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2007.

[31] A. Drake, M. Floyd, R. Willaman, D. Hathaway, J. Hernandez, C. Soja, M. Tiner, G. Carpenter, and R. Senger, "Single-cycle, pulse-shaped critical path monitor in the POWER7 microprocessor," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.

[32] Bowman, K.A. and Tschanz, J.W. and Lu, S.L. and Aseron, P.A. and Khellah, M.M. and Raychowdhury, A. and Geuskens, B.M. and Tokunaga, C. and Wilkerson, C.B. and Karnik, T. and De, V.K., "A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance," *IEEE Journal of Solid-State Circuits*, 2011.

[33] V. J. Reddi, M. S. Gupta, G. H. Holloway, G. Wei, M. D. Smith, and D. M. Brooks, "Voltage emergency prediction: Using signatures to reduce operating margins," in *International Conference on High-Performance Computer Architecture*, pp. 18–29, 2009.

[34] J. Gunawardena, "An Introduction to Idempotency." `web.maths.unsw.edu.au/~peterdel-moral/HPL-BRIMS-96-24.pdf`.

[35] NVIDIA Corporation, "NVIDIA CUDA Programming Guide," 2011.

[36] "MSI Afterburner." `http://event.msi.com/vga/afterburner`, 2015.

[37] NVIDIA Corporation, "CUDA C/C++ SDK CODE Samples," 2011.

[38] T. . c. NVIDIA Corporation, Howpublished = `http://docs.nvidia.com/cuda/cufft/`.

[39] A. Adinetz, "Adaptive Parallel Computation with CUDA Dynamic Parallelism." `https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/`.

[40] H. Foundation, "Memory Segment Isolation." `http://www.hsafoundation.com/html/Content/PRM/Topics/02_ProgModel/memory_segment_isolation.htm`.

[41] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, (Washington, DC, USA), pp. 299–308, IEEE Computer Society, 2012.

[42] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, (Washington, DC, USA), pp. 2–13, IEEE Computer Society, 2005.

[43] S. Jin and J. Huh, "Secure mmu: Architectural support for memory isolation among virtual machines," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 217–222, June 2011.

[44] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 58:1–58:11, IEEE Computer Society Press, 2012.

[45] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 1560–1565, March 2010.

[46] C. Oehmen and N. Multari, "Report on the First Meeting on Asymmetry in Resilience for Complex Cyber Systems," in *Asymmetry in Resilience*, September 2014.

[47] R. Fernandez-Pascual, J. M. Garcia, M. E. Acacio, and J. Duato, "A low overhead fault tolerant coherence protocol for cmp architectures," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 157–168, Feb 2007.

[48] N. Madan and R. Balasubramonian, "Power efficient approaches to redundant multithreading," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 1066–1079, Aug 2007.

[49] L. Chen and Z. Zhang, "Memguard: A low cost and energy efficient design to support and enhance memory system reliability," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 49–60, June 2014.

[50] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 319–330, Dec 2014.

[51] A. J. Peña, W. Bland, and P. Balaji, "Vocl-ft: Introducing techniques for efficient soft error coprocessor recovery," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 71:1–71:12, ACM, 2015.

[52] A. Rezaei, G. Coviello, C.-H. Li, S. Chakradhar, and F. Mueller, "Snapify: Capturing snapshots of offload applications on xeon phi manycore processors," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, (New York, NY, USA), pp. 1–12, ACM, 2014.

[53] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 140–151, ACM, 2011.

[54] E. Krimer, P. Chiang, and M. Erez, "Lane decoupling for improving the timing-error resiliency of wide-simd architectures," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 237–248, June 2012.

[55] H. Jeon and M. Annavaram, "Warped-dmr: Light-weight error detection for gpgpu," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 37–47, Dec 2012.

[56] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, (New York, NY, USA), pp. 253–264, ACM, 2009.

[57] M. D. Powell and T. N. Vijaykumar, "Pipeline damping: a microarchitectural technique to reduce inductive noise in supply voltage," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 72–83, June 2003.

[58] M. D. Powell and T. N. Vijaykumar, "Pipeline Muffling and a Priori Current Ramping: Architectural Techniques to Reduce High-frequency Inductive Noise," in *Proc. of ISLPED*, 2003.

[59] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[60] T. N. Miller, R. Thomas, X. Pan, and R. Teodorescu, "VRSync: Characterizing and eliminating synchronization-induced voltage emergencies in many-core processors," in *International Symposium on Computer Architecture*, pp. 249–260, 2012.

[61] M. S. Gupta, J. L. Oatley, R. Joseph, G. Wei, and D. M. Brooks, "Understanding voltage variations in chip multiprocessors using a distributed power-delivery network," in *Design, Automation Test in Europe Conference*, pp. 624–629, 2007.

[62] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G. Wei, and D. M. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *International Symposium on Microarchitecture*, pp. 77–88, 2010.

[63] R. Bertran, A. Buyuktosunoglu, P. Bose, T. J. Slegel, G. Salem, S. M. Carey, R. F. Rizzolo, and T. Strach, "Voltage noise in multi-core processors: Empirical characterization and optimization opportunities," in *International Symposium on Microarchitecture*, pp. 368–380, 2014.

[64] Y. Kim, L. K. John, S. Pant, S. Manne, M. Schulte, W. L. Bircher, and M. S. S. Govindan, "Audit: Stress testing the automatic way," in *Proc. of the International Symposium on Microarchitecture (MICRO)*, 2012.

[65] J. Leng, Y. Zu, M. Rhu, M. Gupta, and V. J. Reddi, "GPUVolt: Modeling and Characterizing Voltage Noise in GPU Architectures," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2014.

[66] J. Leng, Y. Zu, and V. J. Reddi, "GPU voltage noise: Characterization and hierarchical smoothing of spatial and temporal voltage noise interference in GPU architectures," in *International Symposium on High Performance Computer Architecture*, pp. 161–173, 2015.

[67] R. Thomas, K. Barber, N. Sedaghati, L. Zhou, and R. Teodorescu, "Core tunneling: Variation-aware voltage noise mitigation in GPUs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 151–162, March 2016.

[68] R. Thomas, N. Sedaghati, and R. Teodorescu, "EmerGPU: Understanding and mitigating resonance-induced voltage noise in GPU architectures," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 79 – 89, April 2016.