Asymmetric Resilience: Exploiting Task-level Idempotency for Transient Error Recovery in Accelerator-based Systems

Jingwen Leng[1], Alper Buyuktosunoglu[2], Ramon Bertran[2], Pradip Bose[2], Quan Chen[1], Minyi Guo[1], Vijay Janapa Reddi[3,4]
[1]*Shanghai Jiao Tong University,* [2]*IBM T. J. Watson Research Center,* [3]*Harvard University,* [4]*The University of Texas at Austin*

ABSTRACT

Accelerators make the task of building systems that are resilient against transient errors like voltage noise and soft errors hard. Architects integrate accelerators into the system as black box third-party IP components. So a fault in one or more accelerators may threaten the system's reliability if there are no established failure semantics for how an error propagates from the accelerator to the main CPU. Existing solutions that assure system reliability come at the cost of sacrificing accelerator generality, efficiency, and incur significant overhead, even in the absence of errors. To overcome these drawbacks, we examine reliability management of accelerator systems via hardware-software co-design, coupling an efficient architecture design with compiler and runtime support, to cope with transient errors. We introduce *asymmetric resilience* that architects reliability at the system level, centered around a hardened CPU, rather than at the accelerator level. At runtime, the system exploits *task-level idempotency* to contain accelerator errors and use memory protection instead of taking checkpoints to mitigate overheads. We also leverage the fact that errors rarely occur in systems, and exploit the trade-off between error recovery performance and improved error-free performance to enhance system efficiency. Using GPUs, which are at the forefront of accelerator systems, we demonstrate how our system architecture manages reliability in both integrated and discrete systems, under voltage-noise and soft-error related faults, leading to extremely low overhead (less than 1%) and substantial gains (20% energy savings on average).

1 Introduction

Domain specific architecture (DSA) [1] is the key enabler for emerging computation intensive applications such as deep learning [2, 3] and DNA sequencing [4]. The principle of DSAs is to tailor the processor design to a specific domain by identifying and accelerating the "hot" part of an application. As such, the DSAs are also called hardware accelerators [5, 6, 7, 8], which provide continued performance and power efficiency improvements beyond the general-purpose CPU that is stalled by the end of Dennard scaling and the diminishing returns from microarchitecture enhancements.

While heterogeneous accelerator systems provide good performance and power efficiency benefits, they can introduce reliability challenges. For example, prior work [9, 10] has shown that the GPU's MTBF (mean time between failures) is almost eight times lower than the CPU's in a large-scale system. As such, the reliability of a heterogeneous accelerator system may be compromised, which could seriously hinder their adoption in mission-critical domains such as autonomous driving. In fact, the recent accelerator system for Tesla's autonomous driving uses the dual module redundancy for protecting against transient hardware errors [11].

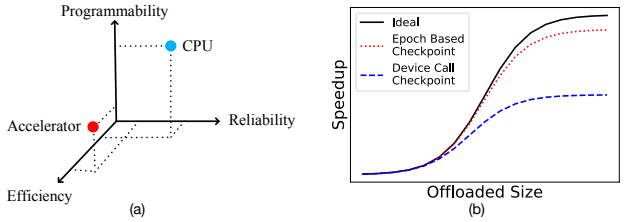


Fig. 1: (a) Efficiency, reliability, and programmability trade-off in accelerator-based systems. (b) Applying existing techniques can hurt the peak performance of an accelerator [15].

Therefore, a key challenge facing the development of future systems is the design and implementation of resiliency techniques that can maintain the overall system reliability in the face of increasing accelerator integration. But as Fig. 1(a) shows, optimizing a system comprised of different processing accelerators for overall performance, efficiency, and reliability is difficult because the "sweet spot" for each accelerator can be inherently different from one another [12, 13].

In this work, we focus on the effective recovery of accelerators from errors. More specifically, we focus on transient error recovery, which is important in the face of deep technology scaling [14]. The typical approach for recovering from transient errors in a heterogeneous system involves checkpoint and replay, which can significantly degrade the accelerator's performance. In Fig. 1(b), we use LogCA [15] (a recently proposed analytical performance model) to estimate the overall performance impact. For each accelerator task (i.e., a device call), if the system takes a checkpoint of the offloaded data [16], the speedup can be cut by half. Other approaches, such as epoch-based checkpointing, can reduce the cost [17]. However, this becomes complicated when the epoch includes mixed CPU and accelerator computation.

We present *asymmetric resilience*, a system-level architecture design for heterogeneous system reliability. In asymmetric resilience, we break the system down into domains. We deploy accelerators in the *weak resilient domain* and the CPU in the *strong resilient domain*. The idea is to ensure the system's reliability using the most resilient domain. The strong domain hosts the reliable CPU, and the weak domain hosts the error-prone accelerators. The two domains, strong and weak, are optimized for resiliency and performance/power, respectively. In doing so, we simplify system

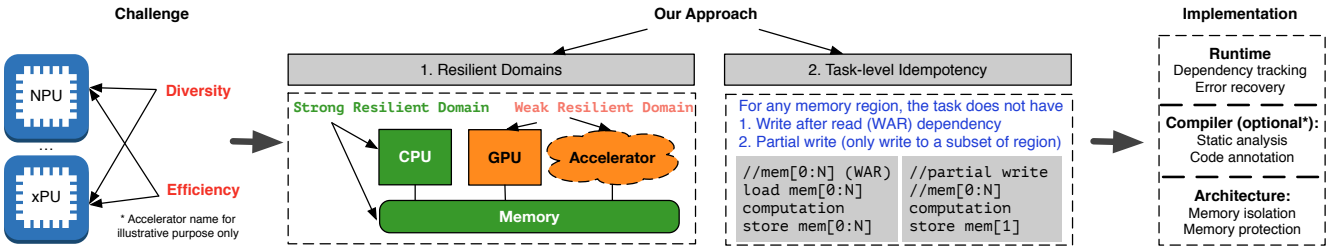


Fig. 2: Overview of asymmetric resilience. It requires synergy between the architecture, compiler, and runtime system. The accelerator only needs to detect the error. The CPU and its software components handle the error on its behalf. We use a GPU for real system demonstration, and an analytical model for other accelerators (due to the lack of real systems for accelerators).

reliability management into error detection (at the accelerator level) and error recovery (at the CPU level).

The next step is to understand how to orchestrate reliability management between the different accelerator and the CPU domains. To this end, we build upon the concept of idempotency [18]. We extend idempotency to the task-level for minimizing transient error recovery overhead. A task is idempotent if its multiple executions lead to the same result. *Task-level idempotency* suits accelerators because it is determined only by the accelerator's task interface, which is often well-defined with input/output information. This critical nugget of information opens up powerful hardware/software co-design opportunities. For example, in the event of an error, we can eliminate the checkpointing overhead for the input and output memory by using memory protection schemes and re-issuing/executing the accelerator task.

First, we implement and evaluate asymmetric resilience on a real system, using a GPU as an accelerator in the system. We demonstrate that asymmetric resilience is already feasible to apply to today's systems. *We identify the interaction between the architecture, compiler, and runtime system in achieving high system-level efficiency.* Fig. 2 shows an overview of our work. The compiler performs static analysis on a GPU kernel (i.e., task) and annotates the code with the derived input and output information. Based on the information, the runtime leverages the augmented memory system to grant write permission only to the output memory. During kernel execution, most memory regions are protected in the read-only mode. For the output memory, the runtime looks for opportunities to regenerate it by re-executing a single or multiple previously launched kernel rather than take a checkpoint. This optimization sacrifices recovery performance but results in a negligible error-free execution overhead.

Second, we evaluate the effects of asymmetric resilience on discrete versus integrated accelerator systems. We study the integrated CPU-GPU systems and examine two types of error event: voltage (or $L\frac{di}{dt}$) noise and soft errors. As GPUs are programmable, we use a set of representative programs to cover as large accelerator space as possible. For most kernels, our system does not need to take any checkpoint and can recover from transient errors from a kernel by simply relaunching the erroneous kernel. For the rest kernels, the runtime tracks dependencies and decides the minimum set of kernels to relaunch. Our system has negligible overhead (less than 1%) during error-free execution, and it incurs an error recovery penalty that is linear with error probability. Thanks to this low overhead, we show how using asymmetric resilience, we can safely unlock the benefits of voltage undervolting. We achieve 19.6% savings in energy, which is close to the ideal oracle-based energy savings of 21%.

Third, we study asymmetric resilience in the context of emerging hardware accelerators. A key challenge with a forward-looking study is the lack of real systems containing accelerators that are readily available to the public for experimentation. Therefore, as the best effort, we evaluate how asymmetric resilience works based on its first-order design principles and an analytical model. We consider accelerators from seven new domains and show that asymmetric resilience applies to all but one, and show the effectiveness.

In summary, we make the following contributions:

- We propose *asymmetric resilience* as an architectural design to ensure the reliability of heterogeneous systems in the presence of transient errors. Such a design relies on the CPU for error recovery and exempts GPUs and other accelerators from heavy resiliency optimizations (Sec. 3).
- We describe how the concept of *task-level idempotency* can be leveraged to efficiently implement asymmetric resilience with the right architectural- and system-level support at the CPU, GPU, and the memory subsystem (Sec. 4).
- We describe *hardware-software co-design* of the components, including the compiler and runtime that allows us to minimize the system overhead within 1% (Sec. 5).

We organize the rest of the paper as follows. Sec. 2 motivates the need for asymmetric resilience in heterogeneous systems. Sec. 3 gives its overview. Sec. 4 and Sec. 5 describes its architectural and software support respectively. Sec. 6 describes the experimental setup. Sec. 7 evaluates the asymmetric resilient design and its associated benefits in the CPU-GPU system. It also quantifies the overhead for applying the design to accelerators with an accelerator-agnostic analytical model. We compare with related work in Sec. 8. Finally, we conclude the paper in Sec. 9 with future thoughts.

2 Accelerator System Reliability Challenges

Our work focuses on the transient error recovery in the context of heterogeneous systems that are equipped with accelerators. In such a system with unprotected data exchange, an error in the accelerator (such as the GPU) can corrupt the CPU's states, leading to the program or even operating system crash [29]. In this section, we explain the fundamental

Category	Related Work	Generalization Ability	Efficiency
Epoch based checkpoint and restart (no idempotency)	CRUM [19], CheCUDA [20], NVCR [21], CheCL [22], HeteroCheckpoint [23], VOCL-FT [17], and others [16, 24, 25]	✓	✗
Instruction sequence idempotency	Encore [26], Clover [27], iGPU [28]	✗	✓
Task-level idempotency with resilient domains	Asymmetric resilience (this work)	✓	✓

Tbl. 1: Prior checkpoint and restart (CPR) work either does not work well for accelerators, or has low efficiency.

requirements for handling transient accelerator errors. However, none of the existing error handling technique satisfies all the fundamental requirements at the same time, which requires a dedicated study for the accelerator-rich system.

2.1 Need for Generality and Efficiency

Resilient accelerator-based computing systems require accelerators to detect and recover from transient errors. We focus more on the error recovery than on error detection, as we find that accelerators' unique features make the error recovery task more challenging. Sec. 4.2 discusses the error detection in greater details. We consider the commonly adopted checkpoint and restart (CPR) scheme for accelerator error recovery, which has two fundamental requirements: *generalization* and *efficiency*. We explain why the existing CPR schemes fail to achieve them simultaneously.

Generality The growing number of accelerator architectures requires a recovery technique that can work across a broad range of architecture implementations. Specialized accelerators target different computation domains (e.g., data base [30] and robotics [7]). As a consequence, they result in vastly different architecture implementations. Even within a given application domain, different accelerators can have different microarchitecture implementations. For instance, neural network accelerators can leverage different forms of parallelism (e.g., layer parallelism [5] versus neuron parallelism [31]). Such differences inevitably lead to diverse microarchitecture implementations. To make matters worse accelerators can use asynchronous and non-deterministic control logic [32] for improved efficiency. Consequently, traditional recovery schemes that rely on the assumption of a Von Neumann architecture cannot readily apply to accelerators, and in the case it does it is inefficient (as we demonstrate).

Efficiency The other challenge for accelerator error recovery is retaining their highly sought after performance efficiency. Computing systems almost always adopt accelerators because of their superior performance and efficiency. However, CPR based schemes unfortunately impact their efficiency. CPR incurs checkpoint overheads for checkpointing the system's relevant memory and the accelerators' internal states. Checkpointing the address space becomes costly relative to the short accelerator task duration, thereby dominating the total execution time. Accelerators also have large internal states that can incur severe checkpointing overheads. Accelerators use large on-chip SRAM caches and buffers and exploit massive parallelism [6, 32]. So, if we naively apply CPR, accelerator systems can incur significant overhead that defeats the purpose of using accelerators altogether.

2.2 Limitations of State-of-the-Art

We identify two state-of-the-art techniques for accelerator error recovery (as shown in Tbl. 1) and explain why neither satisfies the generalization and efficiency requirements at the same time. We do not consider the fine-grained techniques like instruction-level redundancy [33, 34] as accelerators often have little or even no instruction-level support.

Epoch-based Checkpointing A typical and straightforward approach for ensuring accelerator reliability is to take the checkpoint of an accelerator's state before using it and restoring the state when an error occurs [16]. However, that requires checkpointing regardless of whether the error event occurs. Our evaluation shows that this approach incurs a steady 10× overhead (Sec. 7 provides more details), which defeats the purpose of adopting accelerators in the first place.

Other researchers proposed more efficient solutions [17, 20], which periodically take checkpoints at the granularity of an epoch. This approach amortizes the checkpointing overhead by dividing a GPU program into epochs. However, the epoch can mix the CPU and accelerator (GPU) computation. That requires taking the checkpoint of the entire system data, which includes both the CPU and accelerator(s). Our evaluation shows that such a solution still incurs a significant overhead of around 10% - 100% (as we demonstrate later in Sec. 7), similar to the numbers reported by prior work.

Idempotency-based Checkpoint Prior work [26] has leveraged the idempotency property to reduce the checkpoint overhead. A code region is idempotent if it does not have write after read (WAR) dependency for all registers, which can be identified through dataflow and pointer analysis. The idempotency property guarantees the same result of multiple executions so that a re-execution can recover from the transient error. However, it is difficult to extend this property to non-Von Neumann architecture-based accelerators where programs are not expressed in the form of sequential instructions. Furthermore, prior work assumed a robust fault model where a transient error is detected before corrupting the memory state. Such a robust fault model is based on CPU's speculation feature and feasibility of control flow signature, but are likely to be absent in accelerators.

To summarize, the existing checkpoint-restart schemes do not simultaneously satisfy the two fundamental requirements for accelerator recovery, i.e., generality and efficiency. On the one hand, the epoch-based approach overlooks accelerator systems' unique opportunity and incurs significant overhead. On the other hand, the idempotency-based approach relies on CPU-specific architectural characteristics and a robust fault model regarding memory state corruption. These limitations motivate a dedicated study for better solutions.

3 Asymmetric Resilience

To overcome the challenges of generality and efficiency for recovering from transient errors in accelerators, we introduce a system-level design called *asymmetric resilience*. At the system level the architecture is comprised of two key abstractions – i.e., *resilient domains* and *task-level idempotency* (as shown in Fig. 2). We explain these concepts and describe how we leverage them to provide a general and efficient error recovery mechanism for transient errors in accelerator systems. The key benefit of our design is that it abstracts away the low-level microarchitecture details for error recovery at the isolated level of individual accelerators. Instead, it alleviates the issue to the global system level.

3.1 System-level Abstractions

Resilient Domains To provide a general solution, we propose to hide the implementation details of accelerators and introduce resilient domains as their system-level abstractions. Our design centers around the communication channel between different domains, which is the memory subsystem as Fig. 2 shows, without touching the internal microarchitecture of the accelerator. For example, with the proper memory access permission management for each domain, errors in a resilient domain are contained within the domain and cannot corrupt the states in the other resilient domain.

To further minimize any possible modification to the accelerator, we also separate its error detection and recovery capability. We propose to augment the accelerator only with error detection capability and move the error recovery capability into the CPU. As a result, the system can have asymmetric resilient domains, i.e. the *strong resilient domain* and a *weak resilient domain* in Fig. 2. The strong resilient domain requires both error detection 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. In this sense, the resilient domain differs from prior containment domain [35] that requires error detection and recovery within a domain.

Since our design relies on the CPU to handle the transient accelerator error, the accelerator error recovery now becomes a system-level reliability issue. We explain later on that it allows for more holistic, system-level optimizations, rather than siloed individual accelerator optimizations. Again these optimizations only assume a generic inter-domain interface to handle transient accelerator errors, without touching the accelerator's internal structure.

Task-level Idempotency We extend the fundamental concept of instruction-level idempotency [26] to the task, calling it *task-level idempotency*, which (as we show later) provides efficient accelerator error recovery. As Fig. 2 shows, there are two conditions for an accelerator task to qualify as having the task-level idempotency property. First, the task must not have any write after read (WAR) dependency for any memory regions. This condition guarantees multiple executions of the task have the same result. Second, the task does not have partial write for any memory region. We explain later that our design enforces access control for memory regions and assumes the weak (general) fault model: an error can corrupt the entire memory address space. In this case, the second condition guarantees the accelerator's output memory region can be restored via re-execution.

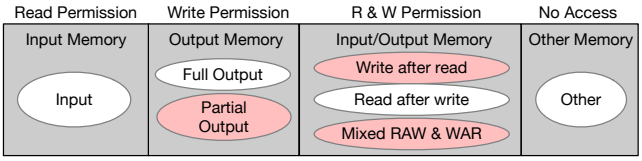


Fig. 3: Accelerator memory address space categorization.

Task-level resiliency is not restricted to the von Neumann architecture model and therefore provides another general abstraction for accelerators. It is only determined by the information of memory regions that the task reads from (input memory region) and writes to (output memory region). E.g., a convolutional layer accelerator [36] takes two input memory regions (for weight and input feature map) and stores the output feature map in another memory region.

3.2 Cross-layer Design

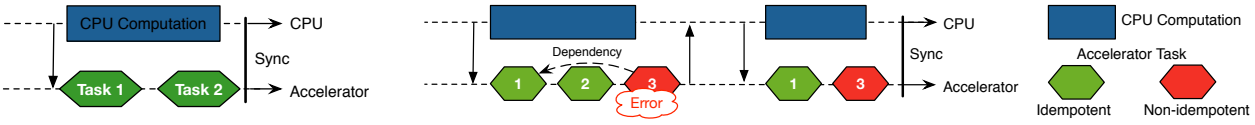
We propose a novel cross-layer design to exploit resilient domains and task-level idempotency. At the hardware-level, the memory subsystem provides underlying mechanisms for enforcing resilient domains. The runtime system leverages the task-level idempotency and tracks the task dependency to handle transient accelerator errors efficiently.

To maximize the generality and hide the accelerator details, we propose a general memory address space categorization method. As Fig. 3 shows, we divide the memory regions for an accelerator task into *input, output, and input/output memory*. The output memory can have full output and partial output memory. The latter means the task only modifies a subset of the memory, such as the histogram calculation where the bin updates are data-dependent. Input/output memory refers to the memory region used as both input and output by the task. Since a memory region has multiple bytes, there are three dependency types for input/output memory: read after write (RAW) for all bytes, write after read (WAR) for all bytes, and mixed RAW & WAR.

Our cross-layer design leverages the information of the accelerator's memory address space to perform checkpoint overhead optimizations. From hereon, for brevity, we abbreviate task-level idempotency simply as idempotency. The task is idempotent if it has only input memory, full output, and WAR output memory. In Fig. 4a, if the tasks are idempotent, the runtime does not need to take a checkpoint of any data. It leverages the augmented memory subsystem to protect input memory in the read-only mode and recovers the full output and WAR output memory through re-execution.

In contrast, only the non-idempotent accelerator task requires checkpoint, specifically the partial output, the WAR memory, and mixed RAW & WAR memory region as they violate the idempotency property. In the partial memory case, since we assume the weak fault model, an error may corrupt one byte that a correct re-execution does not write, which makes the re-execution cannot overwrite the error.

Our runtime system further leverages re-computation to mitigate the checkpointing overhead for those memory regions. As Fig. 4b shows, multiple accelerator tasks can form



(a) Idempotent task. (b) Non-idempotent task.

Fig. 4: Accelerator error recovery in asymmetric resilience. (a) Idempotent task does not require checkpoint. (b) Non-idempotent task requires checkpoint, which can be removed if there exists re-computation opportunity.

a dependency chain, where the first task has an output region used by the third task as the WAR output memory. Instead of taking the checkpoint, we can recover it by re-executing the first task. We will detail this optimization and how the runtime tracks dependencies among different tasks.

4 Architectural Support

We explain the architectural support in our cross-layer system. To demonstrate and evaluate the effectiveness of the hardware and software co-designed asymmetric resilience, we use the GPU as the exemplary accelerator. We choose the GPU kernel as the error detection and recovery unit as it is the smallest control unit by the CPU. As described in the previous section, our system ignores the kernel's internal computation and only cares about its memory access characteristics. In our design, we rely on a compiler analysis to extract and annotate that information for GPU kernels. It is straightforward to derive them for accelerators, and therefore extending this GPU-centric protection work to the broader scope of accelerator-rich systems would only incur minimal efforts. We present a detailed discussion and quantitive estimation on the required extensions in Sec. 7.4. Since the software layer provides all the necessary support for the CPU, our description focuses primarily on the architectural components, including the memory subsystem and the GPU.

4.1 Augmented Memory Subsystem

As we explained previously, the runtime system performs active read/write permission control to improve the efficiency of error recovery. As such, the memory subsystem needs to provide the permission management mechanisms for the runtime. We categorize today's memory subsystem in the heterogeneous processors into two types and discuss how to augment each memory type for providing such mechanisms.

Fig. 5a depicts the discrete memory, in which the CPU and the GPU have separate physical memory. In such a system, permission control requirements can be avoided thanks to the memory system's unique characteristics. We explain this using the example of a discrete GPU system. First, the system does not allow the GPU to directly access the CPU memory, which guarantees no CPU memory corruption even in the presence of GPU errors. Second, in its programming model, the CPU also needs to allocate and copy the input data to the GPU explicitly. It leads to the duplicated data on the CPU and the GPU, which can be used as a checkpoint and eliminates the need for access permission control.

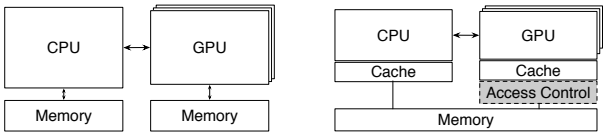
Fig. 5b shows the integrated memory subsystem, which will become more widely adopted due to its performance and programmability advantages. We identify a usable prior work Border Control [37] that allows read/write permission control for every GPU memory access that misses at the cache. We add the access control unit shown in Fig. 5b using the Border Control logic, whose majority part is a cache structure that stores the permission table. The added logic should reside in the strong resilient domain through the strong ECC protection for soft errors, or a higher voltage margin or separate voltage rail for voltage noise. Since GPU programs typically operate on the data size of several to a few hundred MBs, we set the access control granularity to be 8 KB, the same used by the original work. We show that this unit adds near-zero storage and performance overhead in our later evaluation. *It is worth noting that our work focuses first and foremost on the cross-layer design principals, i.e., how to leverage the access control knobs. Our solution is independent of the microarchitecture level details, which is an important and intentional feature of asymmetric resilience.*

4.2 Error Detection

In asymmetric resilience, accelerators only need to detect errors and delegate the recovery to the CPU. *The accelerator error detection and recovery are two orthogonal problems.* Hence, our work focuses on recovery because it is more complex and architecture-dependent. In contrast, accelerator error detection is relatively architecture-independent because of the same underlying physical causes for errors across architectures. Our cross-layer design also relaxes the detection requirement, allowing for simpler error detectors.

Circuit-level error detectors for CPUs are generally applicable for accelerators. For instance, current high-end GPUs already use error correction codes (ECC) for the soft error protection of register files and caches [9]. When uncorrectable errors are detected, the GPU deems the current context as corrupted and requires a full restart [38]. Regarding the voltage noise, voltage droop sensors or canary circuits [39, 40, 41, 42] are available. Besides the circuit-level solution, accelerators can also deploy microarchitecture-level error detection techniques, including modular redundancy [43, 44] and anomaly/exception detection [45, 46], which makes the solution slightly more architecture-dependent.

The detection latency is a critical metric for choosing the appropriate solution. Prior CPU-centric work [26] requires fast error detection such as before the error starts to cor-



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

Fig. 5: Taxonomy of the different memory subsystems.

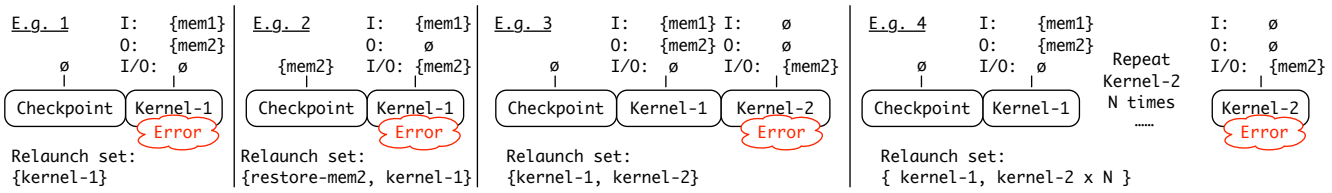


Fig. 6: Examples for illustrating error recovery process. The `input`, `output`, and `input/output` memory are noted as `I`, `O`, and `I/O`, respectively. The last three examples involve non-idempotent kernels that rely on kernel cohesion.

rupt the memory state. In contrast, our cross-layer design relaxes the latency requirement, where the error only needs to be caught before the CPU accesses the data. In asymmetric resilience, even if an error is detected late, it can only corrupt the kernel's memory region (with write permission). This matches our fault model assumption and is handled by the upper-level runtime. In summary, we can borrow the CPU's mature error detection techniques and apply them to the GPU/accelerator, and leave the more challenging error recovery tasks to the system-level.

5 Compiler and Runtime Support

We describe how the compiler and runtime system cooperate to support asymmetric resilience efficiently. We first propose an approximate but safe compiler analysis for identifying the task-level idempotency property of a kernel. Based on the compiler annotation pass, the runtime manages the checkpoint and performs optimizations to reduce the checkpoint overhead during error-free execution, which is the typical operation mode of the system as error events happen rarely. We also present how to handle discrete versus integrated accelerator systems, showing that the software component for asymmetric resilience is extensible.

5.1 Compiler Analysis and Annotation

As explained in Sec. 3, asymmetric resilience relies on the task-level idempotency property to achieve efficiency. Task-level idempotency is solely determined by the relationship between the memory regions that the task reads and writes. In our work, we use a compiler static analysis to automatically derive those characteristics for each kernel. Alternatively, the system could use more advanced tools such as a dynamic instrumentation tool [47] for deriving those information. However, the compiler analysis is optional in our solution as accelerator architects can manually determine and provide those characteristics as we discuss in Sec. 7.4.

We start off with a conservative (safe) compiler analysis pass because it is impossible to derive the exact memory information in Fig. 3 using only the static data flow analysis. The analysis is approximate because it does not distinguish the different types of input/output memory and treats them the same, assuming they violate idempotency. Therefore, the system may consider an idempotent task as non-idempotent. Of course, even with this inaccurate information, the system can recover from the task's error by taking the checkpoint, albeit of unnecessary data. Similarly, the compiler may treat a full output memory as partial output, i.e., a false alarm of idempotency violation. But such "mistakes" only cost performance overhead, not the reliability of the system.

Owing to the conservative approach, our LLVM [48] compiler only needs to determine each kernel's *input, full output, partial output, and input/output memory*. For each kernel, it extracts all the memory regions shared between CPU and GPU through its argument list. The compiler then performs *use-def* analysis [49] for each memory region. The region is used solely as the kernel's input (output) if it has only *use* (*def*) information. On the other hand, the region is used as input/output if it contains both *use* and *def* information.

To distinguish between the kernel's full or partial output, the compiler combines static symbolic execution with dynamic runtime checking. It first uses statically known variables (i.e., thread ID and thread block ID) to derive the size of the output region, and annotates this information for the runtime. The runtime compares the compiler-derived memory size with the allocated memory size (by tracking `cudaMalloc`). If the two match, it concludes the memory region is the kernel's full output. If the derived size and tracked size do not match, or if symbolic execution fails to derive the size of an output region, we attribute the output region as a partial output. As we explained previously, this analysis is safe for the system's reliability because it does not mistakenly identify a partial output region as a full output region. In our study, we find that our compiler analysis is simple but effective as it identifies all regions correctly.

5.2 Runtime Management and Optimization

In asymmetric resilience, the runtime system is responsible for managing the checkpoint and recovering from any detected errors. We explain how the runtime leverages the architectural support and compiler annotated information for eliminating most of the system overhead. We also present a dynamic optimization called *kernel cohesion* to handle the checkpoint process for the non-idempotent kernels.

Idempotent Kernel Based on the compiler-annotated memory region information, the runtime identifies the kernel as idempotent if it only has input memory and full output memory. The runtime uses augmented memory subsystem to enforce the read-only access permission for the input memory, which eliminates its checkpoint requirement. It is also unnecessary to checkpoint the output memory, which can be recovered by re-executing the kernel after a transient error occurs. In other words, an idempotent kernel does not require any checkpoint. Tbl. 2 shows that only 12 out of 81 kernels in our studied programs are non-idempotent, which

Kernel Type	Reason	Count
Idempotent	-	69
Non-idempotent	With input/output memory	9
	With partial-output memory	3

Tbl. 2: Idempotent and non-idempotent kernels count.

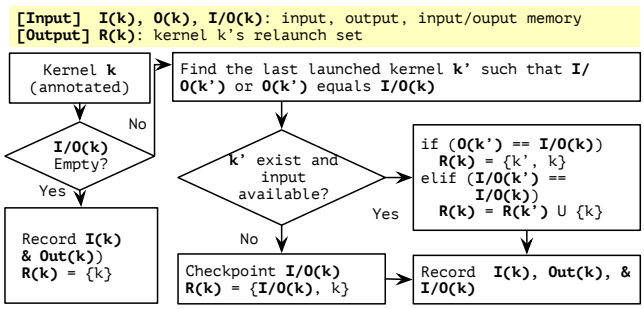


Fig. 7: The runtime flowchart for creating checkpoint and deciding relaunch set before executing a kernel.

lays the foundation of high efficiency of our system.

Kernel Cohesion In contrast, a non-idempotent kernel requires to take the checkpoint of relevant data. However, it is also unnecessary to take the checkpoint of all its memory regions. Instead, only the input/output (denoted as I/O from now on) and partial output memory require checkpointing as the runtime can handle other memory regions (input and full output) in the same way of idempotent kernels. Our system also performs another software optimization called *kernel cohesion* to eliminate the checkpoint requirement for those memory regions. Instead of taking the checkpoint, the runtime actively looks for opportunity to restore them through re-computation, which is often cheaper than taking and restoring checkpoint [50, 51]. Performing kernel cohesion requires the runtime to track the dependency among kernels. We use examples in Fig. 6 to explain the process and then summarize the full operational flow.

Running Examples The erroneous kernel in the first example is idempotent because it has no I/O memory. Note that we omitted the partial memory throughout the examples as it is treated the same as I/O memory. The runtime does not take any checkpoint and simply relaunches it for error recovery. In contrast, the erroneous kernel in the second example is non-idempotent, and the runtime needs to checkpoint the I/O memory before the execution. It needs to restore the checkpoint and relaunch the kernel for recovering its error.

The last two examples involve the kernel cohesion for the checkpoint elimination. In the third example, kernel-2 has I/O memory mem2, which is the output from kernel-1. There is no need to take checkpoint of mem2 as we can group the two kernels for error recovery. As a result, to recover from the kernel-2's error, the runtime needs to relaunch the kernel-2 and additional kernel-1. In the fourth example, the program invokes kernel-2 multiple times, and only the last invocation experiences an error. Similarly, kernel cohesion eliminates all checkpoint requirements for the non-idempotent kernel kernel-2. In the recovery process, the runtime relaunches kernel-1 and N instances of kernel-2.

Operational Flow For the kernel cohesion implementation, we maintain a *relaunch set* for each kernel, which is the set of kernels the runtime needs to relaunch for recovering the erroneous kernel. In its essence, the kernel cohesion optimization simply keeps track of the kernel dependency and does not actually fuse kernels as it requires re-compilation. Fig. 7 shows the full operational flow, including the checkpoint and relaunch set management. The relaunch set for an idempotent kernel is just itself. If input/output memory in the non-idempotent kernel k can be restored by re-executing kernel k', the runtime adds k' to the relaunch set of k. In addition, if kernel k' is non-idempotent, the runtime also needs to add the relaunch set of k' to the relaunch set of k. If the runtime does not find the kernel cohesion opportunity, it creates a checkpoint, which is very rare in our system.

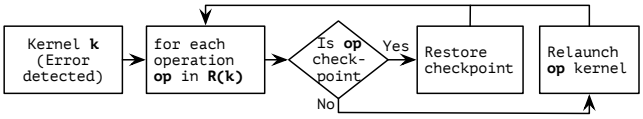


Fig. 8: Flowchart of the runtime handling GPU errors.

Fig. 8 illustrates the runtime's operation flow when recovering from an error in kernel k. The runtime directly goes through the relaunch set of kernel k, which may contain the operation of restoring the relevant checkpoint and relaunching a series of kernels. The runtime reissues these operations following their recorded order to recover from the error.

6 Experimental Methodology

We describe our methodology for prototyping an efficient asymmetric resilient CPU-GPU system. Later on, we generalize to other (non-GPU type) accelerators. We use both discrete and integrated system and systematically inject two transient error types: soft errors and voltage noise. Finally, we explain the testbed design for studying GPU errors.

6.1 Hardware, Software and Benchmarks

Hardware Setup We study the discrete GPU system with a real hardware setup in Fig. 9. All the programs execute on only one GPU card, and we use a redundant GPU card purely for error detection. 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. Note that this apparatus is used for prototyping purposes only. If we had access to prior work setup [9], we can skip this step and directly use the GPU's error detection capability. We use a GTX 680 and 780 card [52], and an Intel Core i5 CPU.

We use the gem5-GPU simulator [53] for evaluating the integrated CPU-GPU system. We augment it with the memory access control mechanism proposed by prior work [37] (Sec. 4.1). Also, we use a simulator setup similar to that used by prior work, which has one CPU core and eight GPU cores (SMs). The setup represents the AMD Kaveri APU [54] with increased memory throughput, and it runs CUDA programs.

Finally, we develop a simple analytical model to estimate how well asymmetric resilience works on (non-GPU) accelerators. We use an analytical model, rather than do simulations since there are no robust end-to-end simulation infrastructures that are readily available to the public for use.

Software Infrastructure For the GPU evaluations, we implement our runtime system at the CUDA runtime (version 7.0) level as Fig. 9 shows. The runtime API is the control interface between CPU and GPU [55]. We use the CUPTI library [56] to overcome the closed sourced CUDA APIs. This tool provides instrumentation features [57] for implementing

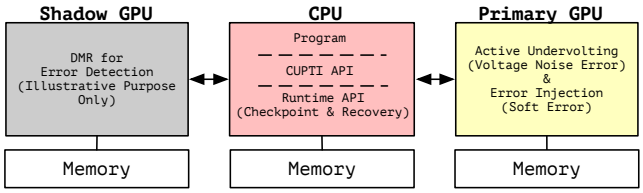


Fig. 9: The dual-GPU system experimental setup. *Note that our overhead does not include the shadow GPU based DMR.*

asymmetric resilience runtime. 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 functions, which are required for checkpointing and error recovery.

CUDA Benchmarks On the discrete GPU setup, we use 32 programs. These programs have diverse characteristics, which lets us make insightful observations and comprehensive evaluation. On the integrated setup, we use all the seven programs supported by the infrastruture [37]. We find that seven programs have similar characteristics of the other 32 programs used in the real hardware, and therefore, the simulation result is also representative. For all programs, we choose the largest possible dataset, which ranges from several to hundreds of MBs, to fully exercise our system.

Though we can run most programs, we cannot support all programs out in the field because some applications rely on special drivers. Our runtime only tracks runtime APIs, and so it cannot support programs that use CUDA driver APIs (e.g., the CUFFT [58] library uses driver APIs). Fixing it requires engineering effort and does not affect our design.

6.2 Error Injection

Soft Errors We use the CUPTI instrumentation tool [56] to inject errors into a kernel's various memory regions (e.g., input and output in Fig. 3) in both the discrete and integrated system. For a target error probability, we run the program 1, 000 times with randomly chosen single- and multi-bit flip. This gives good coverage of possible soft error failures points, thereby allowing us to test our runtime's efficacy because the failure point impacts the system behavior.

Voltage Noise Since voltage noise does not manifest any timing errors at the nominal voltage due to the excessive voltage guardband, we "inject" voltage noise errors [29] by operating the chip just below the V_{min}. To determine V_{min}, we decrease the GPU's operating voltage from its stock setting, progressively, one step at a time, until the GPU starts failing. The stock setting of the GTX 680 card is 1.09 V at 1.1 GHz. We use the MSI Afterburner [59] to lower the GPU chip's voltage with a granularity of 12 mV at a fixed frequency. With each 12 mV undervolting step, we run the kernel and check the kernel correctness by validating its output against the output from the redundant GPU that is always running at the nominal voltage. We perform a byte-level comparison using the utility `memcmp` [60]. A run passes if the two outputs are identical. We measure each program 1,000 times for measuring its V_{min} level, similar to prior work [29]. To inject the voltage noise error, we run each program one step below its measured V_{min} level. We also run the voltage noise error injection 1, 000 times for each program, which we observe has around 10% error probability.

6.3 DMR Error Detection Testbed Design

For the real system study, our runtime system leverages DMR for GPU error detection. We only use this solution for studying error recovery; we presented more lightweight solutions in Sec. 4.2 that are better in practice when the systems are available for production-level implementation.

Our runtime system can i) maintain an identical state of the first GPU in an extra shadow GPU; ii) launch an identical shadow kernel from the original GPU in the shadow GPU; iii) compare the results from the original kernel and shadow kernel to detect a possible execution error.

In practice, the runtime effectively creates an identical state in the shadow GPU (see Fig. 9). It intercepts the GPU's runtime functions and selectively repeats them in the shadow GPU. We categorize those functions into three kinds: memory allocation, copy, and binding. Whenever the program allocates memory or copies memory to the first GPU, we perform the same operation in the shadow GPU. This allows us to maintain an identical memory state. After the original and shadow kernels complete their execution, we copy their memory back to the CPU and perform a byte-level comparison. Such fine-grained memory checking lets us precisely investigate any errors during execution.

7 Evaluation

We perform a comprehensive evaluation of asymmetric resilience in different aspects. We first emphasize its performance cost during the error-free execution. Such cost should be as low as possible because it does not involve any error events. Our co-designed system can achieve almost zero cost. We also show that it can also recover from frequent errors with reasonable overhead in most cases, and evaluate the effectiveness of kernel cohesion optimization for handling non-idempotent kernels. Furthermore, we show the high efficiency of our design also enables active GPU undervolting for improving the GPU energy efficiency. Based on the GPU insights, we derive an analytical model to estimate the overhead for other types of accelerators. As such, we show our design is a generic and efficient solution for recovering from transient errors across accelerators.

7.1 Error-Free and Error-Recovery Overhead

Fig. 10 shows the information for the different programs we study, including the number of static kernels, memory regions, and their region sizes. The number of kernel definitions varies from 1 to 8 and their invocations vary from 1 to 512 (not shown). The number of regions ranges from 1 to 15. The sizes range from hundreds of KBs to MBs. Most programs do not have I/O memory and partial output memory. A possible explanation can be the well structured code and GPU's performance advantages over regular problems.

We first evaluate the runtime's overhead when no error occurs under four different checkpointing scenarios, as shown in Fig. 11. The scenarios 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 to the OS file system ("File") or

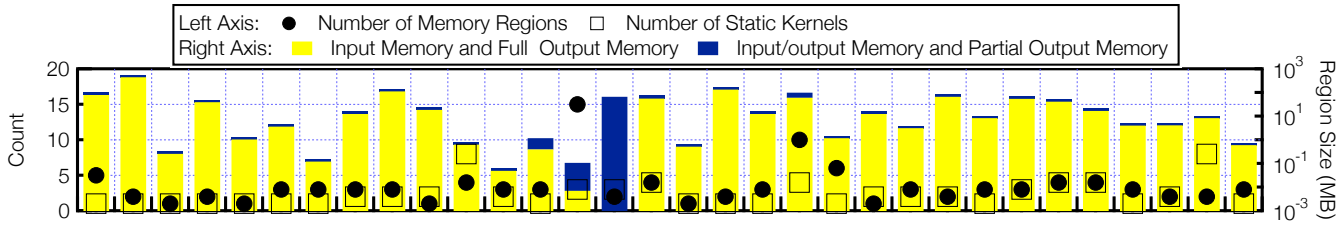


Fig. 10: Program characteristics. The static kernel means a kernel definition, which can be executed multiple times.

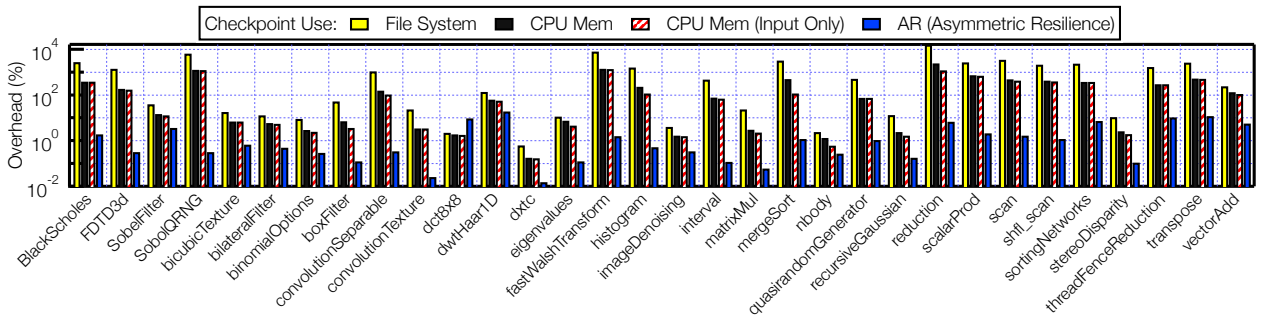


Fig. 11: The overhead (normalized to the original unprotected system) when no accelerator errors occur. The first three bars represent different checkpointing schemes while the last bar is our AR system with hardware memory protection support.

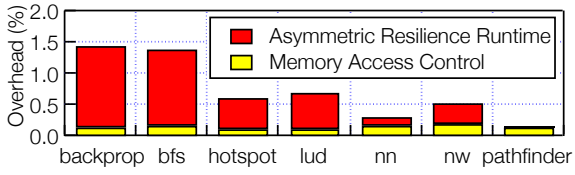


Fig. 12: Error free overhead for the integrated GPU.

CPU memory ("CPU Mem"). They represent different assumptions on whether the CPU and GPU memory are fully isolated. The first case assumes no isolation, i.e., the CPU memory is not safe when a GPU error occurs, used by prior work [17]. The second case makes the opposite assumption that the CPU memory is safe from GPU errors.

The "CPU Mem (Input Only)" scenario implements fine-grained driver level fault tolerance technique proposed by prior work [24]. Such a scenario takes the checkpoint for a kernel's input using CPU memory before the kernel's execution starts and restores that memory when a GPU error is detected. The last scenario "AR" corresponds to asymmetric resilience that uses co-design. The runtime uses the hardware's fine-grained memory access control (Sec. 4.1) to eliminate the checkpoint requirement for memory regions other than input/output and partial output memory. It further performs kernel cohesion (Sec. 5.2) to remove the checkpoint requirement for those two kinds of memory region.

Fig. 11 compares the different scenarios' checkpoint overhead, collected from our discrete GPU setup. The geometric means for the four scenarios are 162%, 38%, 30%, and 0.76%, respectively. In some cases, checkpointing the entire memory space to the file system can result in 100× overhead, which defeats the purpose of introducing accelerators. *The result highlights that asymmetric resilience achieves near-zero error-free execution overhead through co-design.* The main overhead in our runtime comes from the relaunch set calculation process illustrated in Fig. 7. Although we only show the results on the SDK benchmark suite, asymmetric resilience achieves similar efficiency on Rodinia [61] and ISPASS suite [62]. We show the latter results on the integrated GPU and omit the former owing to the space limit.

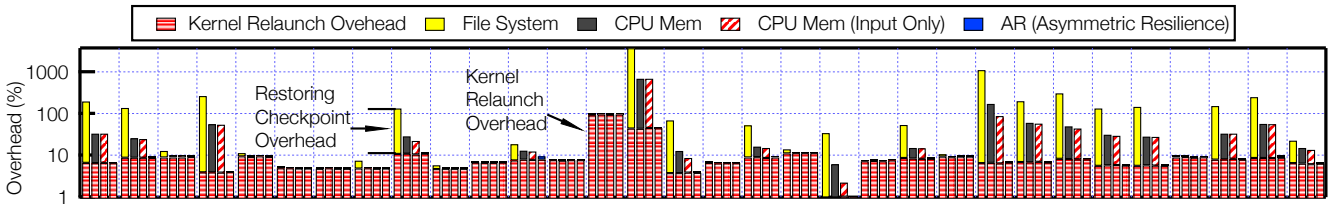
Fig. 12 shows the overhead of asymmetric resilience for the integrated GPU system. Note that here we show the results for all benchmarks currently supported by the simulation platform. Although such a system requires architectural modification to enforce memory access control, the incurred overhead is less than 0.2%. For backprop and bfs, the runtime needs to checkpoint input/output memory, and therefore incurs higher checkpoint overhead. But the overall overhead is 0.6% and therefore negligible.

We now evaluate the overhead of the asymmetric resilience runtime system to recover from errors. Recall that we have two mechanisms to induce errors—operating the kernel *below* its safe V_{min}, which is only used in the discrete GPU hardware setup, and directly corrupting the memory space during execution through fault injection (i.e., soft error). We use the same setup for the discrete and integrated GPUs.

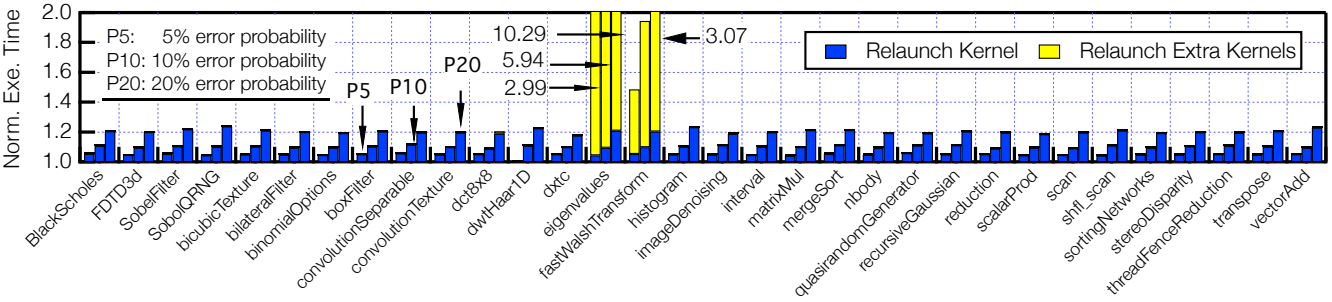
Voltage Noise Fig. 13a shows the voltage noise recovery overhead for the four scenarios. We only show the overhead of checkpoint restoration (top component), and kernel(s) re-execution (bottom). The checkpoint creation overhead is the same in Fig. 11 and thus not shown. Comparing the two parts, we find the checkpoint overhead dominates in all cases except AR, which replaces the overhead of making/restoring checkpoint with the negligible overhead of memory protection control and kernel cohesion optimization.

Soft Error We evaluate the soft error with different probabilities and only show asymmetric resilience results due to space limits. Other scenarios have similar behavior as the voltage noise case: the checkpoint overhead dominates. We focus on kernel relaunch overhead as the checkpoint overhead is negligible in asymmetric resilience.

In Fig. 13b (discrete GPU), the label "P5" means running



(a) Voltage noise error recovery overhead (normalized to the error free execution). Different bars represent different checkpoint schemes.



(b) Soft error recovery time (normalized to the error free execution) of different error injection probability ("P5" means 5% probability).

Fig. 13: Error recovery overhead for voltage noise error and soft error in different scenarios for the discrete GPU case.

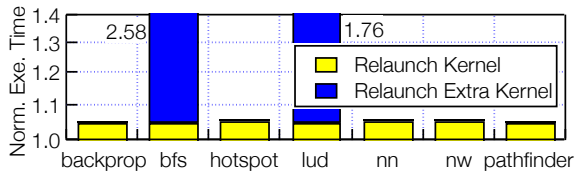


Fig. 14: Error recovery overhead for the integrated GPU.

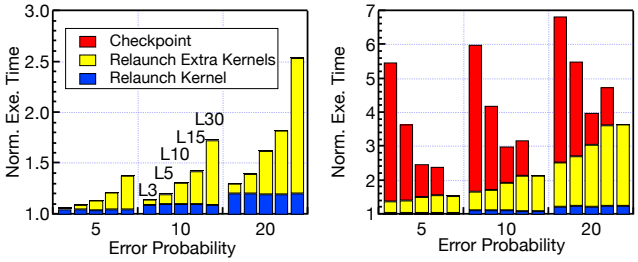
a program 100 times with 5% error probability. We also evaluate the 10% and 20% probability. Those rates are much higher than practical soft error rate, which is smaller than 10^{-6} but difficult to derive statistically sound results. We break down the cost into relaunching the erroneous kernel itself and relaunching extra kernels due to kernel cohesion.

Most programs in Fig. 13b do not need kernel cohesion, so *their recovery overhead increases linearly with the error rate.* Only dct8x8, eigenvalues, and fastWalshTransform activate the kernel cohesion and relaunch extra kernels. In kernel cohesion, the extra relaunched kernels essentially form a dependency chain (e.g. the fourth example of Fig. 6), 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 they suffer from the long dependency chain.

Fig. 14 shows a similar result in the integrated GPU system. Due to the simulation speed constraints, we only evaluate the 5% error probability case. Other than the two benchmarks bfs and lud that requires kernel cohesion, all other benchmarks suffer from very low error recovery overhead.

7.2 Error-Recovery Optimization

We evaluate optimizations to reduce the kernel relaunching cost. The runtime dynamically makes a checkpoint if the kernel dependency chain length exceeds a threshold, and therefore, avoids re-execution of multiple kernels. We study the optimal chain length through design space exploration. Fig. 15 shows the results for two benchmarks with the highest recovery cost. The label "L3" indicates a chain length



(a) eigenvalues. (b) fastWalshTransform.

Fig. 15: Execution time with varying dependency chain length thresholds. "L3" indicates a threshold value of 3.

threshold of 3. We vary the threshold value from 3 to 30.

Our results unveil an interesting observation that the optimal chain size depends on the program characteristics. In Fig. 15a, eigenvalues prefers short chain size because the checkpoint time is relatively small compared to the kernel execution time. In contrast, fastWalshTransform in Fig. 15b prefers a larger chain size because the checkpoint overhead is much higher. This observation suggests that *the runtime can use the ratio between kernel execution time and checkpoint time to decide the optimal chain length on-the-fly.*

7.3 Resilient Voltage Undervolting Benefits

We demonstrate that the extreme efficiency of asymmetric resilience opens up a greater reliability trade-off space for GPUs. To understand this trade-off, we perform an end-to-end system-level evaluation using undervolting as a case study. Undervolting is a power, performance and reliability trade-off optimization [13]. We operate the GPU at the measured V_{min} and measure different cases's energy savings.

We show that the extremely low overhead of our runtime is important for energy saving schemes such as voltage guardband optimization. The first row (Oracle) in Tbl. 3 shows the average measured energy savings when operating each ker-

Scenario	%	Scenario	%
Oracle	21.0	File System	-251.6
		CPU Mem	-61.2
Asymmetric Resilience	19.6	CPU Mem (Input Only)	-48.7

Tbl. 3: Undervolting net energy savings (%) comparison.

nel at its V_{min} on a GTX 680 card. In our case, *we find that up to 18.3% of the nominal voltage can be safely reduced, which translates to 21% energy savings on average.*

The actual realization of the oracular saving opportunity is tightly coupled to safeguard efficiency. Tbl. 3 also compares the net energy savings considering the performance cost of the four safeguard schemes when operating at the V_{min}. Our runtime incurs less than 1% overhead while the others incur 30% to 160% overhead (Fig. 11). *Therefore, only our co-designed system can achieve close to the 21% energy saving. All other schemes result in negative net savings.*

7.4 Accelerator Overhead Estimation

In this work, we cannot directly study accelerators owing to the lack of publicly available simulation infrastructure. Based on the results and insights of the representative kernels on GPUs, we derive an analytical model to quantify the overhead of applying our design to accelerators, and demonstrate its potential as a *generic and extremely efficient* solution for recovering from accelerator errors. We leave the more detailed study in the future work.

Analytical Model Eq. (1) illustrates the proposed model. The numerator is the checkpoint time: the size of idempotency-violating input/output memory (IO_{size}) divided by the checkpoint bandwidth (B_{width}). The denominator is the total task duration time with the kernel cohesion optimization: the average accelerator task duration (T_{task}) multiplied by the dependency chain length (L_{chain}, see the last example in Fig. 6).

$$Checkpoint\ Overhead = \frac{(\frac{IO_{size}}{B_{width}})}{(T_{task} \times L_{chain})} \quad (1)$$

We extract the above information for accelerators according to their high-level computation characteristics, without knowing their implementation details. Tbl. 4 lists our studied accelerators and Fig. 16 shows their characteristics.

Idempotent Accelerators We can identify the idempotency property of an accelerator by inspecting its targeted task. In Tbl. 4, Conv. Engine [63] and neural network inference accelerators (Diannao [5] and TPU [6]) have only input data

Domain	Accelerator	Idempotent?
Algebra	Convolution Engine [63]	Yes
Database	Q100 [30]	Yes
Neural network inference	Diannao [5]	Yes
	TPU [6]	Yes
Neural network training	TPU [6]	No
Robotics	Motion Planning [7]	Yes
DNA Alignment	Darwin [8]	Yes
Graph Analytics	Graphicionado [64]	No

Tbl. 4: Different accelerators' idempotent characteristics.

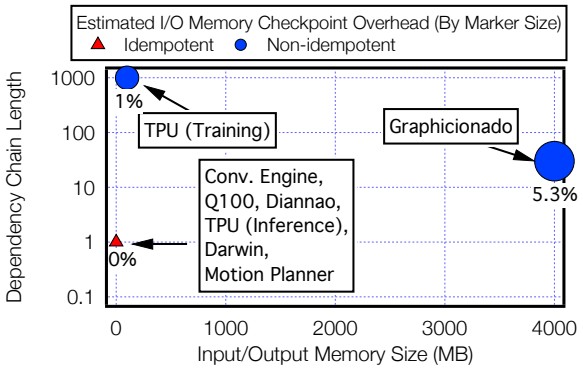


Fig. 16: Accelerator characteristics and estimated overhead.

(for input images and model weights) and output data (for model prediction results), and do not have any I/O memory. Similarly, Q100 [30] targets the SQL query of the database domain, and only supports the query statement that does not modify/update the SQL table[1]. As such, it has only input data (for SQL table and query statement) and output data (for query result). For the motion planner [7], its input is the obstacles voxels and starting/ending point. Its output is the collision-free path. For Darwin [8] that accelerates genome sequence alignment, its input is a query genome sequence and a reference genome sequence. Its output is the gap sequence that maximizes an alignment score. All these accelerators are idempotent because each output element is determined and updated by the input. As such, they do not incur any checkpoint overhead in our system as Fig. 16 shows.

Non-idempotent Accelerators In Fig. 16, only the TPU training and Graphicionado [64] are non-idempotent. The TPU training accumulates the model weights, i.e., I/O memory, whose size is several hundreds of MBs. The training usually takes thousands of iterations, which form a dependency chain for kernel cohesion. Take ResNet-50 [66] with 100 MB weights as an example, the TPU-v2 executes an iteration within 10 ms [67]. If we take the checkpoint every 100 iterations and assume a bandwidth of 10 GB/s, the estimated overhead is only 1%. Graphicionado [64] targets graph analytics that iteratively updates the vertex/edge data, which is the I/O memory. Its clock frequency is 1 GHz and can process four edges per cycle. The duration for processing a graph with 1 G edges (4 GB data) is 0.25 s. With 10 GB/s bandwidth and 30 iterations, the estimated overhead is 5.3%.

In summary, by inspecting the accelerator's high-level computation characteristics, we find most accelerators are idempotent, i.e., perfect fit for our design. For non-idempotent accelerators, there exists a system-level opportunity for kernel cohesion to minimize the checkpoint cost of our design, similarly observed in GPU programs.

8 Related Work

In this work, we propose a generic paradigm called asymmetric resilience for recovering accelerator errors (i.e., soft errors and voltage noise errors). We compare with prior

[1]SQL consists mainly three types of languages: data query language (DQL), data definition language (DDL), and data control language (DCL) [65]. Q100 only accelerates the DQL.

work on relevant topics in the CPU and GPU/accelerator.

Soft Error There are several soft error characterization efforts using the real CPU hardware. Prior work [68] performed application-level fault injection experiments to study the vulnerability of various architectural components in BlueGene/Q processors. Recent work [69] characterized the soft error impact on the DRAM in a supercomputer, while another work [70] studied the soft error for both DRAM and on-chip SRAM caches. There are similar efforts to study the impact of soft error on the GPU architecture [9, 10, 71], which shows that the GPU is more vulnerable than the CPU. Mukherjee et al. introduced the architectural vulnerability factor (AVF) based framework to reduce the error probability. There are subsequent efforts on improving the AVF through scheduling in the heterogeneous multicore processor [72] and data placement in the heterogeneous memory architectures [73]. Those efforts are orthogonal to our work that focuses on efficiently recovering from errors.

Voltage Noise There are also many efforts to study the voltage noise in both the CPU and GPU architectures. Prior work performed measurement-based analysis using various real CPU hardware platforms [74, 75, 76, 77] and GPU platforms [78, 29] to quantify the impact of voltage noise. Since the excessive voltage guardband leads to energy wastage, researchers propose voltage smoothing [74, 79, 80, 81, 82, 83] that mitigates the worst-case voltage droop magnitude for improving energy efficiency. Another line of work tries to actively adapt to the voltage noise fluctuation for energy saving [39, 40, 84] on the CPU architecture. Our work can be used as a safeguard solution to handle the possible voltage noise errors in a GPU with active guardband management.

Error Detection Our work assumes the error detection capability on the accelerators. The error detection techniques generally fall into the architecture independent and architecture dependent categories. The former includes voltage sensors [41, 42], error correction code [84], and modular redundancy [43, 85]. The latter techniques extract signatures from the architecture such as control flow [45], dataflow [86], and coherence token [87]. The architecture-independent error detection techniques are generally good candidates for accelerator error detection. Another work [88] builds on the existing hardware transactional memory. It leverages HTM's features for error detection and recovery, which has the potential for handling the partial output that violates the task-level idempotency in our work.

Error Recovery There are mainly two different types of error recovery work. The first category is checkpoint and restart with different levels of granularities [16, 24, 17, 89]. The other category leverages the idempotency property for reducing the checkpoint overhead. However, the idempotency based solutions [26, 27, 28] assume a strong fault model and are heavily coupled with the CPU microarchitecture. In contrast, our work extends the idempotency concept to the accelerator task and assumes a much more general fault model. The DeSyRe work [90] proposed a runtime system design for reliability optimization in the shared-memory SoCs: under the assumption of the strong fault model, it used multi-versioned task codes for avoiding permanent faults in an SoC component. Prior CPU-centric work [91] used compiler-assisted eager checkpointing to relax the strong fault model assumption. Our work also uses the compiler information for highly efficient hardware/software co-design.

Resilient Domain Prior work proposed containment domain, which is responsible for detecting and recovering from its own errors [35]. In contrast, our work introduces the asymmetric resilient domains, where the weak domain only detects the error, and the strong domain recovers the weak domain's error. Other work designed multicore architecture with different reliability levels for probabilistic applications [92]. The implementation of asymmetric resilient domains builds upon prior BorderControl [37] that prevent errors in one domain from affecting others. Our work focuses on how the system can leverage this hardware mechanism for efficient and generic error recovery.

9 Conclusion

Asymmetric resilience relies on the CPU for error recovery and exempts the accelerators from heavy resiliency optimizations. We show that asymmetric resilience handles transient errors with extreme efficiency, thanks to the coordinated hardware, compiler and runtime design. Our design maintains the accelerator's performance advantages, and also enables more aggressive energy efficiency optimizations such as accelerator undervolting. Asymmetric resilience can be a blueprint for how to scale-out SoC architectures without compromising system-level reliability.

Acknowledgement

We thank the anonymous reviewers for their constructive feedback that helped us improve the work. This work was supported by National Key R&D Program of China (2018YFB1305900), the National Natural Science Foundation of China (NSFC) grant 61702328, and Alibaba Collaborative Research Project. The corresponding author Minyi Guo is supported by NSFC grant 61832006. This research was also 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.

10 References

[1] J. L. Hennessy and D. A. Patterson, "A New Golden Age for Computer Architecture," *Commun. ACM*, 2019.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[3] Google, "Google Self-Driving Car Project," https://www.google.com/selfdrivingcar/.

[4] C. T. Brown, "How much compute power do you need for next-gen sequencing?" http://ivory.idyll.org/blog/how-much-compute-ngs.html.

[5] 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*, 2014.

[6] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[7] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016.

[8] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000x acceleration on long read

assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[9] 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 *International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015.

[10] 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 *44th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[11] E. Times, "Tesla's Kitchen-Sink Approach to AVs," https://www.eetimes.com/teslas-kitchen-sink-approach-to-avs/.

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

[13] K. Swaminathan, N. Chandramoorthy, C. Cher, R. Bertran, A. Buyuktosunoglu, and P. Bose, "Bravo: Balanced reliability-aware voltage optimization," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[14] "International Technology Roadmap For Semiconductor 2013 Edtion," http://www.itrs.net/Links/2013ITRS/\Summary2013.htm.

[15] M. S. B. Altaf and D. A. Wood, "Logca: A high-level performance model for hardware accelerators," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[16] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Tolerating hardware device failures in software," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[17] A. J. Peña, W. Bland, and P. Balaji, "Vocl-ft: introducing techniques for efficient soft error coprocessor recovery," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[18] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *International Symposium on Microarchitecture*.

[19] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "Crum: Checkpoint-restart support for cuda's unified memory," in *International Conference on Cluster Computing (CLUSTER)*, 2018.

[20] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "Checuda: A checkpoint/restart tool for cuda applications," in *International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009.

[21] A. Nukada, H. Takizawa, and S. Matsuoka, "Nvcr: A transparent checkpoint-restart library for nvidia cuda," in *International Symposium on Parallel and Distributed Processing*, 2011.

[22] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, "Checl: Transparent checkpointing and process migration of opencl applications," in *International Parallel & Distributed Processing Symposium*, 2011.

[23] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan, "Heterocheckpoint: Efficient checkpointing for accelerator-based systems," in *International Conference on Dependable Systems and Networks*, 2014.

[24] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[25] L. B. Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka, "Transparent low-overhead checkpoint for gpu-accelerated clusters," 2010.

[26] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: low-cost, fine-grained transient fault recovery," in *International Symposium on Microarchitecture*, 2011.

[27] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *ACM Sigplan Notices*, 2015.

[28] J. Menon, M. De Kruijf, and K. Sankaralingam, "iGPU: Exception support and speculative execution on gpus," in *International Symposium on Computer Architecture*, 2012.

[29] 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.

[30] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[31] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *International Symposium on Computer Architecture*, 2015.

[32] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Network," in *2017 IEEE 23rd International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[33] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer, "Elzar: Triple modular redundancy using intel avx (practical experience report)," in *International Conference on Dependable Systems and Networks (DSN)*, 2016.

[34] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE Micro*, 2007.

[35] 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 *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[36] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, 2017.

[37] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *International Symposium on Microarchitecture*, 2015.

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

[39] 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 *International Symposium on Microarchitecture*, 2003.

[40] 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 *International Symposium on Microarchitecture (MICRO)*, 2011.

[41] K. Bowman, J. Tschanz, S. Lu *et al.*, "A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance," *IEEE Journal of Solid-State Circuits*, 2011.

[42] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "A Self-tuning DVS Processor using Delay-error Detection and Correction," *IEEE Journal of Solid-State Circuits*, 2006.

[43] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. A. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," in *Int. Conf. on Dependable Systems and Networks*, 2007.

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

[45] R. Vemu and J. A. Abraham, "Ceda: Control-flow error detection through assertions," in *International Symposium on On-Line Testing*, 2006.

[46] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Trans. Dependable Secur. Comput.*, 2006.

[47] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in *International Symposium on Microarchitecture*, 2019.

[48] "The LLVM Compiler Infrastructure," https://llvm.org.

[49] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[50] I. Akturk and U. R. Karpuzcu, "Trading computation for communication: A taxonomy of data recomputation techniques," *IEEE Transactions on Emerging Topics in Computing*, 2019.

[51] K. Bergman, S. Borkar *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *DARPA Information Processing Techniques Office, Tech. Rep*, 2008.

[52] NVIDIA, "GTX 680 Kepler Whitepaper - GeForce," http://goo.gl/fyg2z1, 2012.

[53] J. Power, J. Hestness, M. Orr, M. Hill, and D. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *Computer Architecture Letters*, 2014.

[54] AMD, "CMOS AMD's most advanced APU ever," https://goo.gl/W3WwJQ, 2014.

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

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

[57] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[58] N. Corporation, "cuFFT Library," http://docs.nvidia.com/cuda/cufft/.

[59] "MSI Afterburner," http://event.msi.com/vga/afterburner.

[60] "memcmp," http://www.cplusplus.com/reference/cstring/memcmp/.

[61] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *International Symposium on Workload Characterization*, 2009.

[62] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[63] 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*, 2013.

[64] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016.

[65] H. C. Chan, H. J. Lu, and K. K. Wei, "A survey of sql language," *Journal of Database Management (JDM)*, vol. 4, 1993.

[66] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[67] Y. Wang, G. Wei, and D. Brooks, "Benchmarking tpu, gpu, and CPU platforms for deep learning," *CoRR*, vol. abs/1907.10701, 2019.

[68] C. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[69] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of dram raw error rate on a supercomputer," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[70] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory positional effects in dram and sram faults," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[71] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on gpus in the field," in *International Symposium on High Performance Computer Architecture*, 2016.

[72] A. Naithani, S. Eyerman, and L. Eeckhout, "Reliability-aware scheduling on heterogeneous multicore processors," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[73] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta, "Reliability-aware data placement for heterogeneous memory architecture," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[74] V. Reddi, S. Kanev, S. Campanoni, M. Smith, G. Wei, and D. Brooks, "Voltage Smoothing: Characterizing and Mitigating Voltage Noise in Production Processors Using Software-Guided Thread Scheduling," in *International Symposium on Microarchitecture*, 2010.

[75] 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 *International Symposium on Microarchitecture (MICRO)*, 2012.

[76] 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*, 2014.

[77] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore cpus," in *International Symposium on Microarchitecture*, 2017.

[78] J. Leng, Y. Zu, and V. J. Reddi, "Energy efficiency benefits of reducing the voltage guardband on the kepler gpu architecture," in *Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[79] T. Miller *et al.*, "VRSync: characterizing and eliminating synchronization induced voltage emergencies in manycore processors," in *International Symposium on Computer Architecture*, 2012.

[80] 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*, 2015.

[81] 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)*, March 2016.

[82] 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)*, April 2016.

[83] A. Zou, J. Leng, X. He, Y. Zu, C. D. Gill, V. J. Reddi, and X. Zhang, "Voltage-stacked gpus: A control theory driven cross-layer solution for practical voltage stacking in gpus," in *International Symposium on Microarchitecture*, 2018.

[84] A. Bacha and R. Teodorescu, "Dynamic reduction of voltage margins by leveraging on-chip ECC in itanium II processors," in *International Symposium on Computer Architecture*, 2013.

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

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

[87] A. Meixner and D. J. Sorin, "Error detection via online checking of cache coherence with token coherence signatures," in *International Symposium on High Performance Computer Architecture*, 2007.

[88] S. W. S. Do and M. Dubois, "Core reliability: Leveraging hardware transactional memory," *Computer Architecture Letters*, 2018.

[89] A. Rezaei, G. Coviello, C.-H. Li, S. Chakradhar, and F. Mueller, "Snapify: Capturing snapshots of offload applications on xeon phi manycore processors," in *International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[90] D. Pnevmatikatos, S. Tzilis, and I. Sourdis, "The DeSyRe Runtime Support for Fault-Tolerant Embedded MPSoCs," in *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2014.

[91] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-Directed Lightweight Checkpointing for Fine-Grained Guaranteed Soft Error Recovery," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[92] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra, "ERSA: Error Resilient System Architecture for probabilistic applications," in *Design, Automation Test in Europe Conference Exhibition*, 2010.