iGPU: Exception Support and Speculative Execution on GPUs

Jaikrishnan Menon, Marc de Kruijf, Karthikeyan Sankaralingam
Department of Computer Sciences
University of Wisconsin-Madison
{menon, dekruijf, karu}@cs.wisc.edu

Abstract

Since the introduction of fully programmable vertex shader hardware, GPU computing has made tremendous advances. Exception support and speculative execution are the next steps to expand the scope and improve the usability of GPUs. However, traditional mechanisms to support exceptions and speculative execution are highly intrusive to GPU hardware design. This paper builds on two related insights to provide a unified lightweight mechanism for supporting exceptions and speculation on GPUs.

First, we observe that GPU programs can be broken into code regions that contain little or no live register state at their entry point. We then also recognize that it is simple to generate these regions in such a way that they are idempotent, allowing their entry points to function as program recovery points and enabling support for exception handling, fast context switches, and speculation, all with very low overhead. We call the architecture of GPUs executing these idempotent regions the iGPU *architecture. The hardware extensions required are minimal and the construction of idempotent code regions is fully transparent under the typical dynamic compilation framework of GPUs. We demonstrate how iGPU exception support enables virtual memory paging with very low overhead (1% to 4%), and how speculation support enables circuit-speculation techniques that can provide over 25% reduction in energy.*

1 Introduction

Since the introduction of fully programmable vertex shader hardware [23], GPU computing has made enormous strides. Modern GPUs incorporate sophisticated architecture and microarchitecture techniques such as predication, caching, and prefetching, while abstracting the details away from programmers through their software stack and dynamic compilation approach. To improve the effectiveness of GPUs as general-purpose computing devices, GPU programming models and architectures continue to evolve, and we foresee exception support and speculative execution as the next key steps in their evolution. Below, we reflect on the evolution of traditional CPUs to illuminate why this progression appears natural and imminent.

Just as CPU programmers were forced to explicitly manage CPU memories in the days before virtual memory, for almost a decade, GPU programmers directly and explicitly managed the GPU memory hierarchy. The recent release of NVIDIA's Fermi architecture and AMD's Fusion architecture, however, has brought GPUs to an inflection point: both architectures implement a unified address space that eliminates the need for explicit memory movement to and from GPU memory structures. Yet, without demand paging, something taken for granted in the CPU space, programmers must still explicitly reason about available memory. The drawbacks of exposing physical memory size to programmers are well known. Other issues like debugging and supporting arithmetic exceptions are likely to emerge as problems for future GPUs as well. Exception support is a fundamental pillar of modern CPUs and is used to provide all of the above features. To make the leap to becoming a truly general-purpose programming platform, we believe future GPUs will require robust exception support to enable virtual memory, and will significantly benefit from this support in other areas as well.

Modern GPUs are also positioned to benefit from speculation support in the near future. Shortly after the development of exception support in CPUs, speculation was developed as a mechanism to transparently handle "difficult" code, and a recent study claims that GPUs must similarly begin incorporating techniques like speculation to expand the domains they can target [6]. Speculation support has also increasingly been proposed for handling recovery from hardware reliability problems in CPUs [4, 17, 30]. Such problems, which include variability, noise, and excessive guard-banding, are also emerging problems for GPUs [7]. However, recent work on GPU solutions to overcome these problems still has at least 40% overheads [35]. As with CPUs, efficient speculation support in GPUs can serve as a fundamental primitive that enables support for a more diverse range of application programs and handling of hardware reliability issues.

1.1 Key Challenges

Exception support and speculative execution can expand the scope and improve the usability of GPUs. However, implementing them efficiently on GPUs presents key challenges. Below, we discuss exception and speculation support in CPUs and why CPU mechanisms are problematic to apply directly to GPUs. The three key challenges we identify are: *consistent exception state*, *efficient context switching*, and *speculative writes*.

For CPUs, the problem of exception support was solved at a relatively early stage [36, 38]. This support was a key enabler to their success, and instrumental in this success was the definition of *precise exception handling*, where an exception is handled precisely if, with respect to the excepting instruction, the exception is handled and the process resumed at a point consistent with the sequential architectural model [36]. With support for precise exceptions, all types of exceptions could be handled using a universal mechanism such as the re-order buffer. However, precise exception support has historically been difficult to implement for architectures that execute parallel SIMD or vector instructions, where precise state with respect to an individual instruction is not natural to the hardware. High fan-out control signals to maintain sequential ordering in a vector pipeline are challenging to implement, and while buffering and register renaming approaches have been proposed [14, 36], they are costly in terms of power, area, and/or performance. Hence, a key challenge is supporting *consistent exception state*: exposing sequentially-ordered program state to an exception handler and also enabling program restart from a self-consistent point in the program.

A second reason for the widespread adoption of precise exception support in CPUs was that it enabled support for demand paging in virtual memory systems: to overlap processor execution with the long latency of paging I/O, the state of a faulting process could be cleanly saved away and another process restored in its place. Simply borrowing techniques from the CPU space to implement context switching on GPUs, however, is difficult. In particular, saving GPU state and then context switching to another process while a page fault is handled imposes a monumental undertaking: while on a conventional CPU core a context switch requires little more than saving and restoring a few tens of registers, for a GPU it can require saving and restoring *hundreds of thousands* of registers. Thus, a second key challenge is supporting *efficient context switching*: minimizing the amount of state that must be saved and restored to switch among running processes.

Finally, speculation support in GPUs faces similar obstacles to exception support. In fact, CPUs often implement speculation recovery using the same mechanisms as for exception recovery. However, speculation has the additional property that it generates state that may be incorrect with respect to the program's execution. On CPUs, this problem is handled simply by incorporating large hardware checkpointing or buffering structures to manage speculative state. The MIPS R10K for example, implements checkpointing by maintaining four copies of the register rename table [41]. However, the amount of register state on GPUs is simply too vast to consider this option. Hence, a third key challenge is supporting *speculative writes*: finding a way to manage large amounts of speculative program state.

1.2 Paper Overview

In this paper, we develop a low-overhead technique to support exceptions and speculative execution on GPUs. Fundamentally, we observe that the three key challenges of enabling precise exception and speculation recovery on GPUs ultimately distill down to just two core problems: (i) minimizing the amount of program state that needs to be preserved and (ii) enabling restart from a consistent program state. While previous work has explored optimizations to each of these pieces individually, the iGPU architecture developed in this work synergistically enables both.

In terms of preserving minimal program state, we observe that preserving live state alone is sufficient. Others have made this observation as well [28, 29, 34]. However, they have assumed either a checkpoint was available, or restarting from the same program state as at the site of a mis-speculation or exception was necessary. The architectural state on CPUs is also typically small (tens of registers) and hence the optimization of furthermore *minimizing* this live state has historically been relatively insignificant. For GPUs, however, minimizing the amount of state that must be managed to handle context switching is valuable.

Second, in terms of restarting the program from a consistent state, we observe that it is not always necessary to restart the program from the site of an exception or mis-speculation, even without checkpoints, and that restarting from consistent live state, as opposed to architectural state, is in most cases sufficient. Again, others have made this observation as well [9, 18, 22]. However, they largely ignore live-state minimization and/or do not provide general exception and speculation support. Other shortcomings that preclude their use for GPUs are discussed in Section 6.

This paper builds upon previous work and delivers a simple, elegant, and efficient solution to the problems of exception and speculation on GPUs. *The iGPU architecture leverages multiple synergistic properties of GPUs and their workloads to preserve minimal program state and allow restart from effectively arbitrary points in the program at very low cost.*

Figure 1 illustrates our approach using a stylized example code sequence. First, as shown in Figure 1(a), we observe that at each point in a program's execution there are differing amounts of live state. We build upon the observation

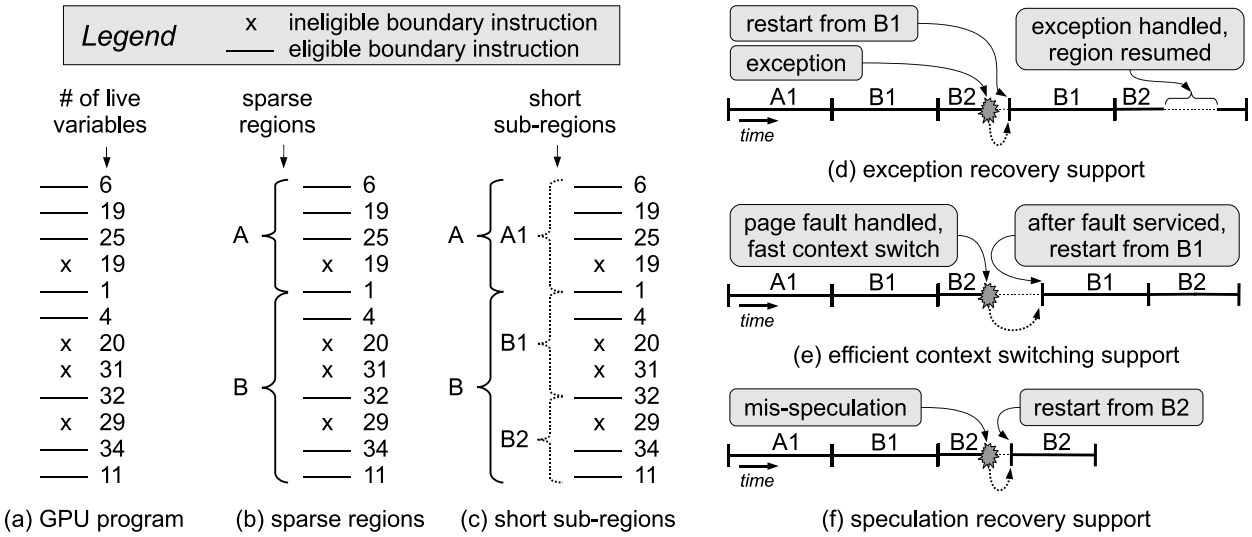


Figure 1: iGPU support for context switching, precise exceptions, and speculation recovery.

previously made that programs fully decompose into *idempotent* (re-executable) code sequences [9], and partition GPU kernels into idempotent "regions" as shown in Figure 1(b), with the key property that the boundaries between regions fall at locations that contain relatively little amounts of live state. By their very nature, GPU application programs tend to have large regions of code that are idempotent and hence these regions can be very large (see Section 5). We call these regions *sparse* idempotent regions because they are sparse both in terms of the amount of live input (live-in) state, and in terms of their occurrence. They can be used to service exceptions that require a context switch, such as page faults, very rapidly due to the low amounts of live-in state, and to service other infrequently occurring exceptions as well. For mis-speculations, however, these sparse idempotent regions tend to be unsuitably large. Hence, as shown in Figure 1(c), we sub-divide these regions into *short* idempotent sub-regions. These short idempotent regions are small and can be re-executed quickly for recovery. However, they are unsuitable for servicing exceptions that require a context switch, because they may have large amounts of live-state at their entry point.

Figure 1(d) shows how the sparse idempotent regions can be used to recover from general exceptions. Suppose that an arithmetic exception occurs executing region B2 and that the architectural state at the point where it is detected is not sequentially consistent with respect to the excepting instruction. The GPU recovers by re-executing the sparse region precisely to the point of the exception, handles the exception, and then recovers by resuming execution from the immediately following instruction. The precise re-execution allows the exception handler to see a consistent live program state with respect to the point of the exception, with forward progress ensured when it is augmented with some support for avoiding live-lock (see Section 4.1).

Figure 1(e) shows how the sparse regions can also be used to efficiently recover from exception conditions requiring a context switch, such as a page fault. Suppose that in the midst of executing region B2 a page fault occurs. Suppose also that the running program pushes and pops live registers from the program stack at the boundary points between sparse idempotent regions. Then, the page fault can be serviced and a context switch can occur effectively instantaneously. After the fault has been serviced, the original process can be switched back in at a convenient time, restarting from region B1. In this scenario, the exception handling need not be "precise" with respect to the faulting instruction, and hence we can both handle it and perform the context switch immediately without concern for the program's state at the point of the fault.

Finally, Figure 1(f) shows how the short idempotent subregions can be used for speculation recovery. Speculative execution writes speculative state directly to the architectural state. When a mis-speculation is detected, recovery occurs simply by re-executing from the start of the containing sub-region, which will regenerate a consistent program state. Depending on the nature of the speculation, the side-effects of a mis-speculation must be appropriately contained to guarantee successful recovery by re-execution.

1.3 Paper Contributions

The key contribution of this paper is the development of the iGPU architecture that leverages the property of idempotence to implement exception and speculation support for GPUs. We present the design and implementation of iGPU hardware and software mechanisms, and demonstrate how sparse and short idempotent regions can be used to support demand paged virtual memory and circuit-level techniques

such as voltage speculation and timing speculation. Our results show that the iGPU architecture can provide demand paging support with less than 4% performance overhead, and that circuit-speculation techniques provide up to 30% energy benefits.

The remainder of this paper is organized as follows: Section 2 presents background on GPUs, Section 3 discusses the iGPU architecture, Section 4 discusses how to support exceptions, context switching and speculation on the iGPU architecture, Section 5 presents our evaluation, Section 6 presents related work, and Section 7 concludes.

2 GPU Background

Before presenting the iGPU architecture, we first give background on the memory and register architecture and the state of exception support in current-generation GPUs.

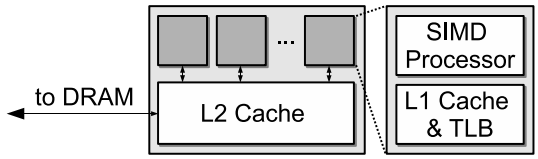
Terminology. Table 1 shows the terms we use as well as the equivalent NVIDIA and OpenCL terms. A GPU consists of a number of *SIMD processors* that execute *SIMD instructions* sequentially ordered into *SIMD threads*. A vertical cut of a SIMD thread, which corresponds with one element of a SIMD lane, we call a *SIMD thread lane*. Finally, identical SIMD threads that run on the same processor form a *SIMD thread group*.

GPU Memory and Register Architecture. Figure 2 shows the high-level architecture of a modern GPU that supports virtual address translation. Each SIMD processor has a hardware-managed L1 cache, we assume each processor has a TLB, and all processors share an L2 cache. NVIDIA's most recent GPU architecture, Fermi, and AMD's recent Llano Fusion architecture both resemble this description [1, 2, 26]. The size characteristics for an integrated AMD GPU part and a discrete NVIDIA GPU part are shown in a table. The table shows that GPU register state is more than both the L1 and L2 cache state combined (often much more).

GPU Exception Support. Current GPUs do not implement general exception support, although Fermi supports timer interrupts used for application time-slicing [26]. While both Fermi and Llano support virtual addressing on some level [1, 26], neither supports all the features of virtual memory, such as demand paging and complete support for execution of processes only partially resident in memory.

Term we use	NVIDIA term	OpenCL term
SIMD processor	Streaming MP	Compute Unit
SIMD instruction	PTX instruction	FSAIL (AMD)
SIMD thread	Warp	Wavefront
SIMD thread lane	Thread	Work item
SIMD thread group	Thread block	Work group

Table 1: GPU terms used in this paper (adapted from Hennessy and Patterson [19]).



GPU architecture	Memory and register state
AMD HD 6550D	128KB L2, 40KB L1, 1.28MB register
NVIDIA GTX 580	768KB L2, 1MB L1, 1.92MB register

Figure 2: The memory organization of a modern GPU that supports virtual address translation and the size characteristics of two commodity GPUs.

3 iGPU Architecture

In this section, we develop the iGPU architecture and organization. First, we define idempotence over a region of code and present a code example demonstrating how idempotence manifests in GPU workloads. We then describe the iGPU compiler, ISA, and hardware support.

3.1 Idempotence

A region of code, which we define as a subset of the program control flow graph, is idempotent if it can execute once or multiple times with the same effect. De Kruijf *et al.* identify idempotence through the absence of *clobber antidependences*, where a clobber antidependence is defined as an antidependence (WAR dependence) with no prior flow dependence (RAW dependence) on the same variable [10]. In other words, an idempotent region can contain an antidependence as long as the antidependent variable is defined before it is used. For example, a {*read*, *write*} sequence over a variable x is not idempotent due to a clobber antidependence on x. However, a {*write*, *read*, *write*} sequence *is* idempotent due to the initial "protecting" write of x.

Figure 3 shows a simple GPU kernel written in C for CUDA that is representative of the types of workloads typically run on GPUs—workloads that have a high degree of data parallelism and have regular streaming memory interactions. A common byproduct of these characteristics is distinct read and write data sets, which implies a lack of antidependences, which leads to the property of idempotence.

The kernel of Figure 3 computes the matrix multiplication of matrices `A` and `B` and accumulates the result onto matrix `C`. It is the unoptimized version of the matrix multiplication kernel presented in the CUDA Programming Guide [27] with the accumulation onto `C` added to make it more interesting. The accumulation forms a clobber antidependence (across lines $4 \rightarrow 7$) in the kernel and hence the kernel is not idempotent. In the next section we show how the iGPU compiler can sub-divide this kernel into two idempotent sub-regions by "cutting" the clobber antidependence and placing a region boundary at the site of cut.

CUDA source code	Live variables[a]
1 `__global__ void MatrixMultiplyAccumulate(Matrix A, Matrix B, Matrix C) {`	
2 `  int row = blockIdx.y * blockDim.y + threadIdx.y;`	0
3 `  int col = blockIdx.x * blockDim.x + threadIdx.x;`	1
4 `  float Cvalue = C.data[row * C.width + col];`	2
5 `  for (int i = 0; i < A.width; ++i)`	3
6 `    Cvalue += A.data[row * A.width + i] * B.data[i * B.width + col];`	8
7 `  C.data[row * C.width + col] = Cvalue;`	5
8 `}`	

[a] Although the `A`, `B`, and `C` matrix objects (with member variables `width` and `data`) are live on entry, along with built-in variables `blockIdx`, `blockDim`, and `threadIdx` (derived from the kernel launch configuration), these are read-only kernel input variables that are backed in memory. They are assumed to be loaded on first use inside a region and hence do not need to be saved on a context switch.

Figure 3: A simple matrix multiplication CUDA kernel annotated with live register information.

3.2 iGPU Compiler, ISA, and Hardware

Figure 5 shows the modifications the iGPU architecture makes to a conventional GPU architecture. The compiler, ISA, and hardware extensions are described in this section and are marked in the figure using black boxes.

Compiler. The iGPU compiler makes modifications to the device code generator of a traditional GPU as shown at the top of Figure 5. The code generator generates device code from an intermediate representation (IR), and the iGPU compiler identifies the sparse and short idempotent regions in the IR and compiles them in such a way that they remain idempotent through the code generation process. We assume an SSA-like IR (e.g. NVIDIA's PTX) with infinite registers so that IR antidependences occur only among non-local variables or across loop iterations. The compiler identifies may-alias clobber antidependences and constructs sparse idempotent regions by "cutting" them in the manner described by De Kruijf *et al.* [10], with the key difference that it prefers to place cuts before instructions with the minimum amount of live state. Placing sparse boundary instructions at these cuts forms the sparse idempotent regions. The compiler then linearly scans each sparse region to partition it into short idempotent sub-regions, scanning up to the point where some path through the sub-region would contain more than the short region instruction limit (e.g. 32), at which point it places a short idempotence boundary. This process of placing idempotent region boundaries is the *region formation* phase of the iGPU compiler and is shown on the left side of the Device Code Generator box of Figure 5.

After the regions are formed, the compiler enters a second phase, the *state preservation* phase shown next to the region formation phase in Figure 5. During this phase, the compiler prevents new clobber antidependences from arising during register and stack memory allocation by allocating local variables in such a way that those variables live at idempotence boundaries are not overwritten. Figure 4 illustrates how this is done for the sparse idempotent regions formed for the example kernel from Figure 3 using a stylized device code representation. Recall that the kernel contains one clobber antidependence. For sparse region formation, the clobber antidependence is cut at the point with the least live state, which occurs immediately before the loop entry at the initialization of loop variable `i` in register `r2`. The idempotence boundary is placed as shown in the lower half of Figure 4. To prevent overwriting, the compiler then logically inserts a move instruction from register `r1` to a freshly allocated register, `r3`, after the boundary instruction. From that point on, it accumulates the `CValue` variable onto `r3` instead of `r1`, preserving the value in the live register `r1` at the expense of some additional register pressure on the kernel. The other live variables at the idempotence boundary are not subsequently overwritten and hence require no action to be preserved. Although the example shows sparse region state preservation only, short region state is preserved similarly. Mechanisms utilizing the iGPU architecture may further transform and optimize the code by spilling registers, coalescing registers, etc.

Device code before:

```
 ...   ($r0 holds (row * C.width + col))
 add.u32 $r0, param[__C_data], $r0;
 mov.f32 $r1, global[$r0];
 mov.u32 $r2, 0x00000000;
LOOP:
 ...
```

Device code after:

```
 ...   ($r0 holds (row * C.width + col))
 add.u32 $r0, param[__C_data], $r0;
 mov.f32 $r1, global[$r0];
 idem.sparse.boundary;
 mov.f32 $r3, $r1;
 mov.u32 $r2, 0x00000000;
LOOP:
 ...   (all uses of $r1 replaced by $r3)
```

Figure 4: Idempotent code generation.

ISA. We extend the ISA with a special instruction to mark the boundaries between idempotent regions as in Figure 4, This instruction holds a single bit which specifies whether it

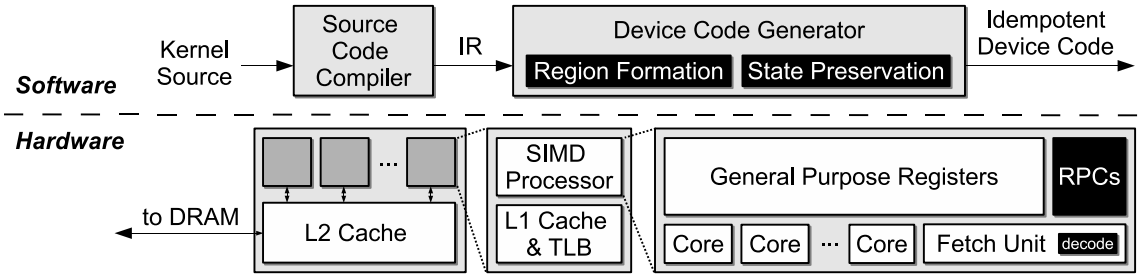


Figure 5: iGPU architecture with GPU modifications in black (not to scale).

starts a new sparse or a new short idempotent region. When the boundary instruction is executed, the PC value associated with the immediately following instruction is saved away in a special RPC (restart PC) register—either the sparse-RPC or the short-RPC depending on the bit setting. The boundary instruction also acts as an instruction barrier such that a SIMD thread's in-flight instructions must retire before proceeding. This ensures that a region remains recoverable while an exception or a mis-speculation remains undetected for the region's in-flight instructions.

Hardware. To support the ISA extension and the execution model in the hardware, we add two RPC registers per SIMD thread and some decode logic to process boundary instructions. These two changes are illustrated on the right side of Figure 5. NVIDIA's Fermi architecture allows a maximum of 48 SIMD threads per SIMD processor, so for this case the RPC state would amount to a 386-byte register file (assuming 4-byte RPC values) physically removed from the hardware critical path. To support the possibility of thread divergence causing thread lanes of the same thread to enter different idempotent regions (for which multiple RPC values would be required to maintain correct execution), we assume thread splitting techniques such as the one proposed by Meng *et al.* are employed when needed to maintain a single RPC per thread [24]. This allows divergent paths to be treated as separate logical SIMD threads, each maintaining its own RPC value. Reconvergence of divergent SIMD threads to saturate the available SIMD width is allowed as well, with the additional restriction that thread reconvergence may occur only *after* encountering the first boundary instruction following the path reconvergence point.

Overall, the hardware changes are small, especially considering the many thousands of registers and tens of functional units already resident on the SIMD processors of modern GPUs. Alternative CPU-like mechanisms to achieve both exception and speculation support would require much more hardware. Additionally, at the circuit level, the timing of exception and mis-speculation control signals can be relaxed compared to traditional hardware pipeline-based approaches.

4 iGPU Exception and Speculation Support

The iGPU compiler, ISA, and hardware changes described in the previous section are minor and are transparent underneath the dynamic compilation environment of modern GPUs. In this section, we develop non-invasive mechanisms that build upon the iGPU architecture to support exceptions, efficient context switching, and speculation.

4.1 General Exception Support

General exception support on iGPUs requires exposing consistent exception state to an exception handler and mechanisms to prevent exception live-lock. The problem of supporting consistent state is that on a GPU multiple exceptions can occur executing a single SIMD instruction, and determining in an exception handler which lanes of the instruction experience an exception can be difficult. The problem of exception live-lock is that multiple recurring exception conditions inside a single idempotent region can lead to live-lock. Below, we describe software and hardware solutions to both problems.

Consistent exception state. To service SIMD exceptions, an exception handler must determine which lanes in a SIMD instruction experience an exception. We initially propose to achieve this without hardware modification as follows. First, prior to re-execution, a software routine patches the excepting instruction with a trap to an emulation routine, similarly to the way a debugger inserts a breakpoint into a running program. Upon re-execution, the emulation routine then emulates the instruction in its entirety, servicing all exception conditions. It then "unpatches" the instruction and resumes execution at the immediately following instruction, as before. Other solutions that require hardware support, such as adding exception status bits to record the excepting lanes of an instruction and the associated circuitry, are also possible. With this solution, the exception handler would require that an excepting instruction execute all lanes of the instruction to completion, updating all result registers except those whose lanes experience an exception.

Exception live-lock. A potential live-lock condition is indicated if, during re-execution, an instruction experiences an exception and its PC value does not match the PC of

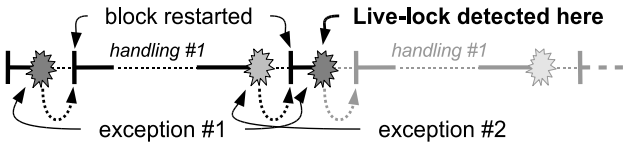


Figure 6: Live-lock under multiple recurring exceptions.

the instruction that initiated the re-execution. Consider, for instance, a region that experiences two arithmetic exceptions as shown in Figure 6. After the first exception is handled, the second exception is encountered shortly afterwards. However, upon subsequent re-execution, the first exception is encountered again, leading to live-lock.

One way to detect live-lock is to save the PC value of an excepting instruction to a dedicated register for comparison during re-execution. Live-lock is then detected as soon as an exception recurs a second time, as shown in Figure 6.

To resolve the live-lock condition after it has been detected, previous work has proposed single-stepped re-execution for CPUs [9]. However, GPU single-stepped execution must occur at the granularity of individual SIMD lanes to allow precise servicing of individual exceptions. GPUs do not natively support this type of execution and adding it requires non-trivial modification to the hardware. Instead, an alternative option is to leverage the virtualized nature of modern GPU instruction sets and employ dynamic re-compilation to prevent live-lock. Rather than the hardware entering a single-stepping mode upon live-lock detection, a dynamic compiler instead recompiles the code such that the two excepting instructions causing the live-lock condition are placed in separate idempotent regions. Afterwards, idempotence-based re-execution can be retried, and the re-compilation effort ensures forward progress. Alternatively, single-stepped lane execution can be implemented in the hardware. Both solutions would be slow to execute; however, it is important to note that potential live-lock should arise only in rare circumstances for the vast majority of possible exception conditions.

4.2 Efficient Context Switching Support

The iGPU architecture can substantially reduce the overheads of context switching on GPUs. This is particularly valuable to the implementation of demand paging, for which a context switch would traditionally require saving and restoring vast amounts of register state. Our key insight is that, utilizing idempotence, context switch state need only be saved and restored with respect to the boundaries between sparse idempotent regions, and that the locations of these boundaries is moreover configurable. Hence, boundary locations can be chosen such that this state is minimized, enabling much more efficient context switching.

To minimize live state on context switches, we simply construct sparse idempotent regions by cutting antidependences at instructions where there is the least amount of live state, as demonstrated in Section 3.2. To illustrate how this helps, suppose that a page fault occurs executing the statement on line 6 of the kernel in Figure 3. Using traditional state-minimization techniques [28, 34], a minimum of 8 live registers would need to be saved to memory. However, restarting from our idempotence boundary at the head of the loop, only 3 live registers need to be saved. This is a greater than two-fold reduction in context switch state.

To perform the actual context saving and restoring, we propose two mechanisms. For the first mechanism, which we call *continuous spilling*, the compiler simply pushes and pops live registers to and from the program stack at each sparse idempotent region boundary. This enables very fast context switching at the expense of paying a small save and restore penalty at each region boundary. For the second mechanism, which we call *just-in-time spilling*, only when an exception occurs does an exception handling routine save and restore the live state at the containing sparse idempotent region entry point. The advantage is that the save and restore cost is only paid when it is needed. The disadvantage, however, is that it is necessary for the compiler to communicate the live state at the region boundary points to the hardware. One way to do this would be for the region boundary instruction to encode as a mask which registers are live. This information is then saved away in a special-purpose hardware register for later recall by an exception handler.

4.3 Speculation Support

In traditional CPUs, speculation recovery support is used for multiple purposes ranging from out-of-order execution [38] and branch prediction [42], to hardware fault recovery [30] and circuit-level speculation [17]. However, modern GPUs implement simple, in-order cores to maximize throughput, and hence have little or no hardware support for speculative execution. Hence these techniques cannot be utilized in GPU architectures.

To bring speculation to GPUs, we observe that, similar to how the re-order buffer handles multiple responsibilities synergistically in traditional CPUs, idempotence can do the same with less hardware overhead for simpler processor designs. While idempotence has potentially higher recovery cost than a re-order buffer, we partition the sparse idempotent regions into the smaller short idempotent regions to keep the re-execution penalty relatively low.

As a concrete example, this section focuses specifically on the iGPU mechanisms to support speculative circuit-level techniques, which have received much attention in recent years [3, 8, 12, 16, 17, 31]. These circuit-level techniques speculatively assume common-case timing and voltage conditions, with occasional recovery employed under worst-case conditions. Timing and voltage speculation are two broad classifications of such techniques.

Timing speculation typically operates only in the processor pipeline, augmenting critical path latches with special shadow latches to detect timing errors [3, 8, 12]. Detection of mis-speculations is fast, and for a CPU, microarchitectural replay and flush support using a re-order buffer is often proposed for recovery. This is overly complex for GPUs, however, and idempotence-based re-execution provides an efficient alternative way to support timing speculation.

In contrast to timing speculation, voltage speculation typically operates at a coarser scale. It allows for coarser and simpler detection mechanisms, but requires more sophisticated recovery support due to longer detection latencies. We base the remainder of our discussion on a previous proposal by Gupta *et al.* for CPUs [17]. They divide processors into *rollback protected* and *timing-margin protected* regions, using both a re-order buffer and store queue to recover from voltage mis-speculation only in the rollback protected regions (pipeline regions). Since the timing-margin protected regions—which include the register file, the PC logic, the L1 write port, and the L2 cache—are less timing-sensitive than the rollback protected regions, they argue that this partitioning approach works well, although extra hardware support, in the form of the store buffer, is still required.

Voltage speculation can be implemented for GPUs by adapting their design to use iGPU mechanisms for efficient recovery. Like Gupta *et al.*, we propose to divide processors into rollback protected and timing-margin protected regions and propose a store queue to buffer stores to speculatively-produced addresses, since these addresses must be verified before a store can commit. However, this structure can be relatively simpler on the GPU. In particular, not all stores must use speculatively-produced addresses, and the GPU's device-independent IR enables a compiler to easily annotate non-speculative stores and communicate them to the hardware. Stores relative to the stack pointer (SP), for instance, typically target a constant offset applied to the SP. If the SP is not updated inside a region (as is typical) and SP-relative address computations are timing-margin protected, then these relatively common types of stores (register spills) are non-speculative and need not be buffered.

5 Evaluation

Our experimental evaluation reports quantitative data on idempotent region construction and on the efficiency of iGPU exception and speculation support. We first present the sparse idempotent region characteristics of region size and the number of live registers at region entry points—two fundamental characteristics motivating this work. We then quantify the overheads of forming the sparse idempotent regions, the overheads of spilling and reloading live register state at the boundaries between these regions for context switching, and the overheads of partitioning sparse regions into short sub-regions for speculation recovery. Finally, we show the performance of virtual memory support and the energy benefits of voltage and timing speculation.

5.1 Methodology

The benchmarks we evaluate are those distributed with GPGPU-Sim 2.1 and are described in detail by Bakhoda *et al.* [5]. We assume a GPU configuration with 4 SIMD processors, with each processor having a SIMD width of 32, pipeline width of 8, 16K registers, and a 256KB L1 cache.

To evaluate the iGPU compiler overheads, we implemented the region formation phase described in Section 3.2 using the Ocelot dynamic compilation framework [11], which operates on NVIDIA PTX. For short idempotent region sizes, we performed a sensitivity study with 16-, 32- and 64-instruction regions, and found that short regions sized at 32 instructions allowed for fast recovery at the cost of only a modest increase in execution time due to additional state preservation overheads.

To measure the overheads of the state preservation performed by the compiler for both sparse and short regions, we model a constrained register file in GPGPU-Sim and preserve live state across idempotent region boundary instructions at runtime. We do this at runtime and not at compile-time out of necessity, as there is no publicly available specification for actual GPU device code, and while attempts have been made to reverse-engineer device code executed by NVIDIA GPUs [40], the resulting tools and information are only useful for performance analysis and not for binary analysis and modification. Hence, we carefully account for the register spills that would be caused from resource constraints in our simulator. In particular, GPGPU-Sim uses the information supplied by the PTX assembler to schedule collections of thread blocks based on register availability constraints. We extend GPGPU-Sim to model detailed register state during execution, and account for the runtime cost of additional register moves, spills, and loads required to avoid register clobber antidependences.

5.2 Region Sizes and Live Register State

Figure 7 shows the median sparse idempotent region sizes *dynamically observed* across all benchmarks, with the 25th to 75th percentile range shown using vertical error bars. The figure shows that the median region size ranges from roughly 200 instructions to over 3,500 instructions. In contrast, for CPU benchmarks De Kruijf and Sankaralingam observed typical idempotent region sizes of only 20 to 40 instructions [9]. Figure 8 additionally shows the median number of registers (per SIMD thread lane) live at sparse idempotent region boundaries. A typical median value for the number of live registers is only 2 and no sparse region has more than 4 live-in registers. These results corroborate the observation from Section 3.1 that SIMD kernels naturally decompose into large idempotent regions that depend on little live state.

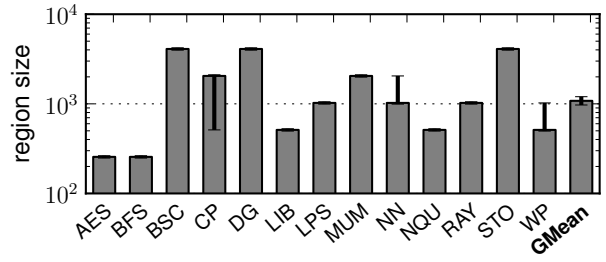


Figure 7: Median sparse region size (y-axis is log-scale). Error bars show 25th (lower) and 75th (upper) percentile.

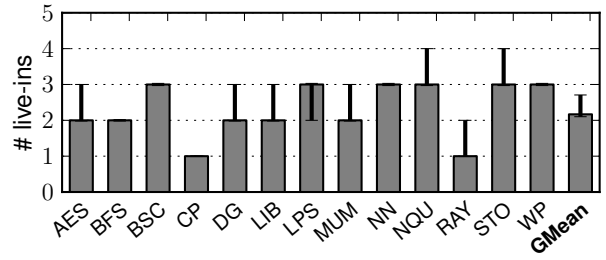


Figure 8: Median live-ins at sparse region boundaries. Error bars show 25th (lower) and 75th (upper) percentile.

5.3 Compiler-Induced Runtime Overheads

Figure 9 shows the compiler-induced iGPU runtime overheads relative to a conventional GPU using a stacked bar graph, where overhead is measured as the percentage additional cycles executed (which corresponds almost directly with the percentage additional instructions introduced by the compiler). The bottom stack shows the overhead introduced by the register spills, reloads, and moves associated with state preservation for sparse idempotent regions[1]. The middle stack shows the additional overhead to spill and reload the live variables at the sparse region boundaries using the continuous spilling approach described in Section 4.2. Finally, the top stack shows the additional overheads from the added register pressure introduced by the short sub-region state preservation. Overall, the total runtime overheads are in all cases less than 4%. Typical overheads are just 2% to 3%.

5.4 Virtual Memory Paging Support

The accumulation of the bottom two stacks in Figure 9 represents the runtime overhead of the sparse region state preservation and the continuous live-in spilling to support virtual memory page faults. Across all benchmarks, the geometric mean overhead is roughly 2.5%. The only additional overheads associated with full paging support are the page fault handling and re-execution overheads incurred to service a page fault. However, we argue that even with a page fault every 1 million instructions (as is common for CPU workloads [32]) this overhead should be negligible (less than 1%). Thus, we claim that the virtual memory paging overheads are effectively limited to just the compiler-related runtime overheads, which range from 1% to 4%.

[1]The compiler region formation is an IR-level analysis that does not directly modify generated device code and hence incurs no overhead.

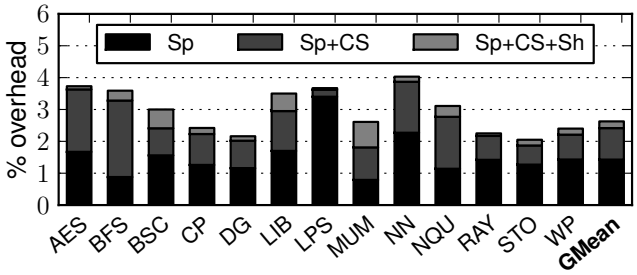


Figure 9: Runtime overheads introduced by the iGPU compiler. (Sp = sparse region state preservation, CS = continuous spilling, and Sh = short region state preservation.)

Speculation Technique	Error Rate	V_{dd} Reduction
DeCoR [17]	1%	10%
Emergency Prediction [31]	0.01%	10%
Razor [12]	1%	15%
Razor II [8]	0.1%	15%

Table 2: Circuit speculation techniques and their corresponding error rate and V_{dd} reduction (approximate).

5.5 Voltage and Timing Speculation Support

To evaluate iGPU speculation support using the short idempotent regions, we evaluate voltage and timing speculation considering a range of possible error rates—1%, 0.1%, and 0.01%—and two possible voltage reduction settings—10% and 15%—executing at those error rates. The different combinations of error rate and voltage improvement can be matched with previously proposed voltage and timing speculation techniques. Table 2 shows the error rates and corresponding V_{dd} reduction for four previously proposed techniques with differing levels of aggressiveness and complexity.

A detailed exploration of the interaction of the misspeculation detection mechanism with the iGPU architecture is part of future work. In the evaluation presented here, we assume first-order principles with the expectation that stalls at idempotence boundaries to support detection delays of the order of a few cycles can be hidden through multithreading on the SIMD processor.

Figure 10 shows the short region re-execution overheads considering the error rates of 1%, 0.1%, and 0.01%. This overhead includes the cumulative compiler-induced runtime overheads from Figure 9. The figure shows that the total runtime overhead ranges from 4% to 12% with a geometric mean of 8% when tolerating a 1% error rate.

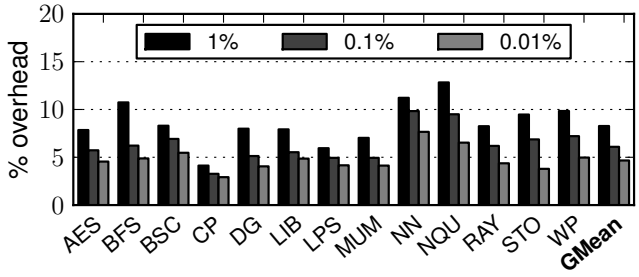


Figure 10: Re-execution overheads for short idempotent regions at three different error rates.

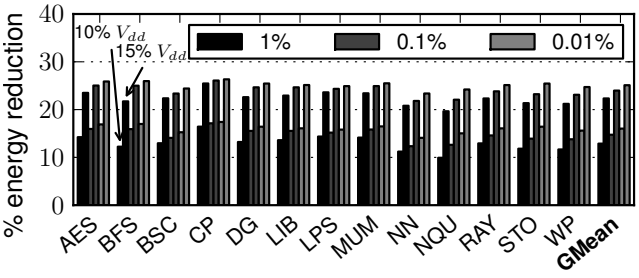


Figure 11: Energy reduction at various error rates and levels of V_{dd} reduction. Each pair of vertical bars evaluates a given benchmark at a given error rate for the two different voltage reduction points indicated by the arrows and labels on the left of the graph.

Figure 11 shows energy reductions based on possible V_{dd} reductions of 10% and 15% at each error rate. We consider dynamic power alone and assume no savings in static power. Dynamic power is proportional to the square of supply voltage and is further multiplied by the per-benchmark execution time to obtain the projected energy consumption. Percentage reductions are calculated with respect to a baseline GPU architecture executing conventional code. All benchmarks show significant energy reductions when tolerating 0.1% and 0.01% error rates. For the 1% error rate, energy reduction is more modest, but still over 10%. Overall, we observe significant possible energy reductions of over 25% when V_{dd} reductions of 15% and beyond are possible.

5.6 Summary

Our results show that the iGPU architecture provides a practical and low overhead way to implement exception support in GPUs. Additionally, idempotence in combination with continuous spilling at region boundaries for context switch state reduction supports virtual memory paging with less than 4% runtime overhead. The iGPU architecture is also able to provide significant energy reductions with efficient support for timing and voltage speculation.

Approach	Weakness
Restart markers [18]	Loop analysis; not generalizable
Idempotent processors [9]	No liveness or SIMD analysis
Fast switch points [37, 43]	External interrupts only
Compiler liveness bits [34]	Forces exact context switch state
Hardware in-use bits [28]	Forces exact context switch state
Buffering (ROB, etc.) [36]	Duplicate vector/SIMD registers
Register renaming [14]	H/W and performance overheads
State snapshotting [28]	Exposes microarchitecture details

Table 3: Prior work on exception and speculation recovery.

6 Related Work

Both idempotence and liveness analysis have been previously proposed as techniques to support efficient recovery. However, the two techniques have never before been combined as they are in this work. Table 3 classifies prior work into three sub-categories: idempotence-based exception recovery, context switch overhead reduction, and classical approaches to exception and speculation recovery.

Idempotence-based recovery. Hampton and Asanović explore idempotence to implement virtual memory for CPUs with tightly-coupled vector co-processors [18]. They explore only simple compute kernels executed by a vector co-processor, consider only loop regions, and do not develop mechanisms to tolerate multiple recurring exceptions. Their technique also does not support exceptions that require visibility to precise program state. Additionally, De Kruijf and Sankaralingam explore idempotence-based exception support in the context of general-purpose processors [9]. They develop single-stepped re-execution as a mechanism to tolerate multiple recurring exceptions and describe how applications can be fully decomposed into idempotent regions. However, they focus on the idempotence property alone, irrespective of live state, and do not address any SIMD-related complications. Compared to our work, neither work ties the benefits of idempotence-based exception recovery with the benefits of the large idempotent regions and the reduced live register state at region boundaries on GPU architectures. Our work also demonstrates how idempotence provides a unified mechanism for speculation and exception support.

Context switch overheads. Others have developed techniques to reduce context switch overheads but have failed to synergistically exploit the property of idempotence. Snyder *et al.* describe a compiler technique where each instruction is accompanied by a bit that indicates whether that instruction is a "fast context switch point" [37]. Zhou and Petrov also demonstrate how to pick low-overhead context switch points where there are few live registers [43]. Although both approaches work well for servicing external interrupts, nei-

ther approach can be used to service exception conditions associated with a specific instruction such as page faults.

Saulsbury and Rice propose compiler annotations to mark whether registers are live or dead [34], and the the IBM System/370 vector facility similarly maintains "in-use" and "changed" bits for vector registers to avoid unnecessary saves and restores on a context switch [28]. While these two approaches can be used to reduce switching overheads for page faults, the swapped state must still be sequentially precise with respect to the faulting instruction. Idempotence allows us to side-step this requirement.

Classical recovery approaches. For general-purpose processors, Smith and Pleszkun describe the re-order buffer, history buffer, and future file approaches for achieving precise exceptions [36]. They also discuss the complications of precise exception support in vector processing, and suggest duplicated vector register state to achieve it. They argue that such duplicated state is preferable to the performance complications arising from instead buffering completed results. For a processor implementing short-width SIMD processing capability, duplicated register state may be practical, but for a GPU it is not.

Other classical approaches to exception support include the use of a unified register file structure to hold speculative state alongside register renaming hardware [25, 38]. This approach has been proposed for vector processors as well [13, 14, 21]. However, compared to general-purpose processors, the benefits to vector processors are less clear as out-of-order execution (a side benefit of register renaming) is typically not necessary to achieve high performance. Hence, the additional hardware complexity, additional register pressure, and resulting performance loss only to support precise exceptions are hard to justify for GPUs.

Finally, "imprecise" exception support mechanisms that involve snapshotting some amount of microarchitectural state have also been proposed. These include the "length counter" technique of the IBM System/370 [28], the "invisible exchange package" of the CDC STAR-100 [36], the "instruction window" approach of Torng and Day [39], and the "replay buffer" approach of Rudd [33]. While these mechanisms provide full exception support, they expose microarchitectural details and are likely too complex for GPUs.

In closing, while no prior work has shown an effective means to bring exception support to GPUs, the need has definitely been recognized. In particular, id Software has demonstrated a need to support multi-gigabyte textures larger than can be resident on GPU physical memory, and develop a technique called *virtual texturing* to stream texture pages from disk [20]. Unfortunately, the technique requires careful scheduling effort on the part of the programmer. GPU virtual memory paging would simplify its implementation and the implementation of similar techniques. Gelado *et al.* recognize the GPU productivity issue and explore asymmetric distributed shared memory (ADSM) for heterogeneous computing architectures. ADSM allows CPUs direct access to objects in GPU physical memory but not vice-versa [15]. This improves programmability at low hardware cost, but falls short of supporting a complete virtual memory abstraction on the GPU.

7 Conclusion

Exception support and speculative execution are crucial next steps in the evolution of the GPU as a general-purpose computing platform. Unfortunately, traditional CPU mechanisms to support exceptions and speculative execution are intrusive to GPU hardware design. In this paper, we proposed the iGPU architecture and supporting mechanisms to enable precise exceptions and speculative execution in a simple manner, with very low overhead, and with efficient context switching. Our technique requires only modest compiler, ISA, and hardware modifications and can be fully automated underneath the traditional software stack of the GPU and its dynamic compilation environment. It leverages the property that GPU kernel programs in particular have large regions of code that are idempotent, and that these large idempotent regions can be constructed to contain very little live register state at their entry point. The property of idempotence is used to enable straightforward exception and speculation recovery, while the minimal live state provides straightforward low-overhead context swapping support. Our results demonstrate that the iGPU architecture provides virtual memory support with less than 4% overhead, and that circuit-speculation techniques can additionally provide up to 30% energy benefits.

Given recent trends—with NVIDIA's Fermi architecture and, more recently, AMD's Fusion architecture both providing a unified GPU address space abstraction—GPUs are on the cusp of implementing full virtual memory support. The iGPU architecture provides an efficient way for demand paged virtual memory in GPUs by providing both exception and fast context switch support. With this exception support, GPUs could moreover support arithmetic exceptions, debugger breakpoints, and other general exception conditions as well. Additionally, by providing short code regions that provide fast recovery, idempotence becomes a general mechanism for speculation recovery. In addition to the use cases we showed, this speculation support can allow GPUs to expand the application domains they can target and provide a means to recover from a variety of hardware reliability problems. Hence, the iGPU architecture is a compelling platform for the continued evolution of the GPU as a general-purpose computing device.

Acknowledgments

We thank the anonymous reviewers, the Vertical group, and Mark Gebhart for comments. Many thanks to Steve

Reinhardt and Brad Beckmann, Guri Sohi, and Mark Hill for several discussions that helped refine this work. Support for this research was provided by NSF grants CCF-0845751, CCF-0917238, and CNS-0917213, and by a Google U.S./Canada PhD Fellowship.

References

[1] AMD. *Memory System on Fusion APUs*. http://goo.gl/r72cp.

[2] AMD. *AMD Accelerated Parallel Processing OpenCL Programming Guide, Rev. 1.3f*. 2011.

[3] L. Anghel and M. Nicolaidis. Cost reduction and evaluation of a temporary faults detecting technique. In *DATE '00*.

[4] T. Austin. DIVA: A Reliable Substrate for Deep Submicron MicroarchitectureDesign. In *MICRO '99*.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS '09*.

[6] E. Blem, M. Sinclair, and K. Sankaralingam. Challenge benchmarks that must be conquered to sustain the GPU revolution. In *Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture*, 2011.

[7] J. Chen. GPU technology trends and future requirements. In *IEDM '09*.

[8] S. Das, C. Tokunaga, S. Pant, W. H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw. RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):32–48.

[9] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *MICRO '11*.

[10] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *PLDI '12*.

[11] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT '10*.

[12] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO '03*.

[13] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the alpha architecture. In *ISCA '02*.

[14] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *MICRO '97*.

[15] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ASPLOS '10*.

[16] B. Greskamp, L. Wan, U. Karpuzcu, J. Cook, J. Torrellas, D. Chen, and C. Zilles. Blueshift: Designing processors for timing speculation from the ground up. In *HPCA '09*.

[17] M. Gupta, K. Rangan, M. Smith, G.-Y. Wei, and D. Brooks. Decor: A delayed commit and rollback mechanism for handling inductive noise in processors. In *HPCA '08*.

[18] M. Hampton and K. Asanović. Implementing virtual memory in a vector processor with software restart markers. In *ICS '06*.

[19] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 5th edition, 2011.

[20] id. id tech 5 challenges: From texture virtualization to massive parallelization. In *SIGGRAPH '09*.

[21] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *ISCA '03*.

[22] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu. Compiler-based multiple instruction retry. *IEEE Transactions on Computers*, 44(1):35–46, 1995.

[23] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *SIGGRAPH '01*.

[24] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA '10*.

[25] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *MICRO '93*.

[26] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Ver. 1.1*. 2009.

[27] NVIDIA. *NVIDIA CUDA C Programming Guide, Ver. 3.1.1*. 2010.

[28] A. Padegs, B. Moore, R. Smith, and W. Buchholz. The IBM System/370 vector architecture: design considerations. *Computers, IEEE Transactions on*, 37(5):509–520, May 1988.

[29] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software – Practice & Experience*, 29(2):125–142, 1999.

[30] J. Ray, J. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *MICRO '01*.

[31] V. J. Reddi, M. S. Gupta, G. H. Holloway, G.-Y. Wei, M. D. Smith, and D. Brooks. Voltage emergency prediction: Using signatures to reduce operating margins. In *HPCA '09*.

[32] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *SOSP '95*.

[33] K. W. Rudd. Efficient exception handling techniques for high-performance processor architectures. Departments of Electrical Engineering and Computer Science, Stanford University, Technical Report CSL-TR-97-732, August 1997.

[34] A. Saulsbury and D. Rice. Microprocessor with reduced context switching and overhead and corresponding method. United States Patent 6,314,510, November 2001.

[35] J. W. Sheaffer, D. P. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *EUROGRAPHICS '07*.

[36] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37:562–573, May 1988.

[37] J. S. Snyder, D. B. Whalley, and T. P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, 1995.

[38] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *ISCA '87*.

[39] H. Torng and M. Day. Interrupt handling for out-of-order execution processors. *Computers, IEEE Transactions on*, 42(1), 1993.

[40] W. J. van der Laan. Decuda SM 1.1 (G80) disassembler. https://github.com/laanwj/decuda.

[41] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[42] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO '91*.

[43] X. Zhou and P. Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *DAC '06*.