Border Control: Sandboxing Accelerators

Lena E. Olson Jason Power Mark D. Hill David A. Wood
University of Wisconsin-Madison Computer Sciences Department
{lena, powerjg, markhill, david}@cs.wisc.edu

ABSTRACT

As hardware accelerators proliferate, there is a desire to logically integrate them more tightly with CPUs through interfaces such as shared virtual memory. Although this integration has programmability and performance benefits, it may also have serious security and fault isolation implications, especially when accelerators are designed by third parties. Unchecked, accelerators could make incorrect memory accesses, causing information leaks, data corruption, or crashes not only for processes running on the accelerator, but for the rest of the system as well. Unfortunately, current security solutions are insufficient for providing memory protection from tightly integrated untrusted accelerators.

We propose *Border Control*, a sandboxing mechanism which guarantees that the memory access permissions in the page table are respected by accelerators, regardless of design errors or malicious intent. Our hardware implementation of Border Control provides safety against improper memory accesses with a space overhead of only 0.006% of system physical memory per accelerator. We show that when used with a current highly demanding accelerator, this initial Border Control implementation has on average a 0.15% runtime overhead relative to the unsafe baseline.

Categories

•**Security and privacy → Hardware attacks and countermeasures; •Computer systems organization →** *Processors and memory architectures;* Reliability;

Keywords

accelerators, memory protection, hardware sandboxing

MICRO-48 December 05 - 09, 2015, Waikiki, HI, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4034-2/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830819.

1. INTRODUCTION

'[B]order control' means the activity carried out at a border ... in response exclusively to an intention to cross or the act of crossing that border, regardless of any other consideration, consisting of border checks and border surveillance.

Article 2, Section 9 of the Schengen Borders Code [1]

With the breakdown of Dennard scaling, specialized hardware accelerators are being hailed as one way to improve performance and reduce power. Computations can be offloaded from the general-purpose CPU to specialized accelerators, which provide higher performance, lower power, or both. Consequently there is a proliferation of proposals for accelerators. Examples include cryptographic accelerators, GPUs, GPGPUs, accelerators for image processing, neural and approximate computing, database accelerators, and user reconfigurable hardware [2–7].

Some of this plethora of accelerators may be custom-designed by third parties, and purchased as soft or firm intellectual property (IP). For example, third-party designers such as Imagination Technologies provide accelerator IP (e.g., for GPUs) to companies such as Apple for inclusion in their devices. 3D die stacking allows dies fabricated in different manufacturing processes to be packaged together, increasing the opportunity for tightly integrated third-party accelerators [8]. Consumer electronic manufacturers may not have introspection capabilities into the third-party IP. For instance, with soft IP the netlist may be encrypted by the hardware IP designers to protect their research investment, which complicates verification and increases the likelihood of bugs and other errors. However, the consumer-facing company still must guarantee performance, stability, and consumer privacy. We expect that SoC designers will increasingly need methods to sandbox on-die and on-package third-party accelerators.

Accelerators increasingly resemble first-class processors in the way they access memory. Models such as Heterogeneous System Architecture (HSA) [9] adopt a shared virtual address space and cache coherence between accelerators and the CPU. This has a variety of benefits including the elimination of manual data copies, "pointer-is-a-pointer" semantics, increased performance of fine-grained sharing, and behavior familiar to programmers. Recently, FUSION [10] investigated the performance and energy improvements possible by using coherent accelerator caches rather than DMA transfers. Many systems today have tightly integrated accelerators which share a coherent unified virtual address space with CPU cores, including Microsoft's Xbox One and AMD Kaveri [11–14]. Much of the current support for tightly integrated accelerators is in first-party designs, but there is some initial support for third-party accelerators: for example, ARM Accelerator Coherency Port (ACP) [15] and Coherent Accelerator Processor Interface (CAPI) from IBM [12].

Allowing accelerators access to system memory has security and fault isolation implications that have yet to be fully explored, especially with third-party accelerators. In particular, designs where the accelerator is able to make memory requests directly by physical address introduce security concerns. There has been speculation about the existence of malicious hardware [16], and even unintentional hardware bugs can be exploited by an attacker (e.g., a bug in the MIPS R4000 allowed a privilege escalation attack [17]). A malicious or compromised accelerator that is allowed to make arbitrary memory requests could read sensitive information such as buffers storing encryption keys. It could then exfiltrate this data by writing it to any location in memory.

A buggy accelerator could cause a system crash by erroneously corrupting OS data structures. Buggy accelerators may be more common in the future with user-programmable hardware on package, such as the Xilinx Zync [7, 18]. In addition, more complex accelerators will be harder to perfectly verify, as evidenced by the errata released by even trusted CPU and GPU manufacturers. Design errors such as the AMD Phenom TLB bug and other errata relating to TLBs show that even first-party designs may incorrectly translate memory addresses [19–21].

A number of approaches have been used by industry to make accelerators more programmable and safer. However, these approaches have focused either on safety or high performance, but not both. For example, approaches such as the I/O Memory Management Unit

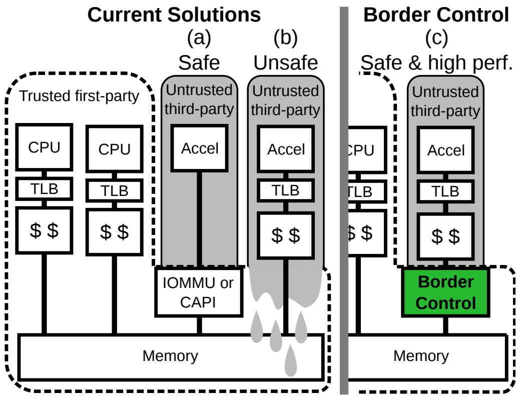


Figure 1: An example system with untrusted third-party accelerators interacting with the system in three ways: a safe but slow IOMMU (a), a fast but unsafe direct access (b), and Border Control, which is safe and high performance (c).

(IOMMU) [22–24] and CAPI [12, 25] provide safety by requiring that every memory request be translated and checked by a trusted hardware interface (Figure 1a). However, they prevent the accelerator from implementing TLBs or coherence-friendly physically addressed caches. This may be acceptable for accelerators with regular memory access patterns or low performance requirements, but some accelerators require higher performance.

In contrast, other approaches prioritize high performance over safety (Figure 1b). Some current high-performance GPU configurations use the IOMMU for translation, but allow the accelerator to store physical addresses in an accelerator TLB and caches [22–24]. The accelerator is then able to bypass the IOMMU and make requests directly to memory by physical address, and there are no checks of whether the address is legitimate. There are also orthogonal security techniques, such as ARM TrustZone, which protects sensitive data but provides little protection between processes.

We propose *Border Control*, which allows accelerators to use performance optimizations such as TLBs and physical caches, while guaranteeing that memory access permissions are respected by accelerators, regardless of design errors or malicious intent. To enforce this security property, we check the physical address of each memory request as it crosses the *trusted-untrusted border* from the accelerator to the memory system (Figure 1c). If the accelerator attempts an improper memory access, the access is blocked and the OS is notified. We build upon the existing process abstraction, using the permissions set by the OS as stored in the page table, rather than inventing a new means of determining access permissions.

We present a low-overhead implementation of Border Control, which provides security via a per-accelerator flat table stored in physical memory called the *Protection Table*, and a small cache of this table called the *Border Control Cache*. The Protection Table holds correct, up-to-date permissions for any physical address legitimately requested by the accelerator. We minimize storage overhead with the insight that translation and permission checking can be decoupled. For a current high-performance accelerator, the general-purpose GPU, this design has negligible performance overhead (on average 0.15% with an 8KB Border Control cache). This shows we can provide security with a simple design and low overheads for performance and storage.

This paper proactively addresses a class of accelerator security threats (rather than react after systems are designed and deployed), making the following contributions:

- We describe an emerging security threat to systems which use third-party accelerators.
- We propose *Border Control*: a comprehensive means of sandboxing and protecting system memory from misbehaving accelerators.
- We present a hardware design of Border Control and quantitatively evaluate its performance and space overheads, showing that they are minimal for even a high-performance accelerator.

2. GOALS AND CURRENT COMMERCIAL APPROACHES

We first define the threat model Border Control addresses and the scope of the threat model, and then discuss current commercial approaches.

2.1 Threat Model

The goal of Border Control is to provide *sandboxing*, a technique first proposed to provide isolation and protection from third-party software modules [26]. Rather than protecting against incorrect software modules, Border Control applies to incorrect hardware accelerators.

The threat model we address is an untrusted accelerator violating the confidentiality and integrity of the trusted host system physical memory via a read/write request to a physical memory location in violation of the permissions set by the trusted OS. More formally,
The **threat vector** is:

An untrusted accelerator, which may be buggy or malicious, that contains arbitrary logic and has direct access to physical memory.

The **addressed threats** are:

Violation of ***confidentiality*** *of host memory, if the untrusted accelerator* ***reads*** *a host physical address to which it is not currently granted* ***read*** *permission by the trusted OS.*

Violation of ***integrity*** *of host memory, if the untrusted accelerator* ***writes*** *a host physical address to which it is not currently granted* ***write*** *permission by the trusted OS.*

There are several cases where accelerators might violate confidentiality or integrity of the host memory by making incorrect memory accesses. Non-malicious accelerators may send memory requests to incorrect addresses due to design flaws. For example, in an accelerator with a TLB, an incorrect implementation of TLB shootdown could result in memory requests made with stale translations. A malicious programmer could exploit these flaws (as in the MIPS R4000 [17]) to read and write addresses to which they do not have access permissions. An accelerator that contains malicious hardware—for example, one that contains a hardware trojan—poses an even greater threat, because it can send arbitrary memory requests.

Incorrect memory accesses can cause a number of security and reliability problems for the system. In non-malicious cases, wild writes can corrupt data including OS structures, resulting in information loss, unreliability, and crashes. Allowing arbitrary reads to host memory can compromise sensitive information stored in memory, such as data stored in keyboard or network buffers, private keys, etc. The attacker can then exfiltrate this data through a variety of means, including writing it to locations belonging to other processes or enqueueing network packets to send to the Internet. In essence, the ability to make arbitrary memory requests provides full control of the system.

Note that these risks apply to *any* accelerator that can make physical memory accesses, *even accelerators designed for non-critical tasks*. For example, a video decoding accelerator may seem harmless from a security perspective if the videos it decodes are neither sensitive nor critical. However, the ability to access system memory means that these seemingly unimportant accelerators can have major implications for system security.

The Principle of Least Privilege [27] states that any part of a system should "operate using the least amount of privilege necessary to complete the job." Border Control enforces this by leveraging the fine-grained memory access permissions of virtual memory. Virtual memory has provided isolation and safety between processes for over 40 years, and Border Control extends this support to the burgeoning accelerator ecosystem by providing the OS with a mechanism to enforce permissions for untrusted accelerators. Current systems use the *process* construct as their container for isolation and protection, and we believe it is important to extend this isolation to computational engines other than the CPU. Whether a process (or an accelerator kernel invoked by the process) is running on the CPU or an accelerator, it should be subject to the same process memory permissions. Border Control's memory sandboxing allows system designers to tightly incorporate untrusted third-party or user-programmed accelerators into their devices while keeping the safety and isolation provided by virtual memory.

2.2 Threat Model Scope

There are a number of threats that are outside the scope of this threat model including the following:

- incorrect behavior internal to the accelerator.
- the accelerator violating the confidentiality or integrity of data to which it has been granted access by the OS.
- the OS providing incorrect permissions.

In particular, Border Control treats the accelerator as a black box, and therefore cannot control what it does internally. That is, a process that uses an accelerator assumes the same level of trust as a process that invokes a third-party software module. When a program calls a library, the OS does not ensure that the library is performing the functionality it advertises; it is possible that the library is buggy or malicious. However, it does prevent the library from accessing other processes' memory or other users' files. Similarly, Border Control does not monitor the operations the accelerator is performing for correctness or place limits on its access to the process's memory. It instead allows the OS to enforce memory access permissions, preventing the accelerator from accessing memory unassociated with the process it is executing. In analogy to the function of border control between countries in the real world, a country can inspect and block traffic entering at its border, but cannot interfere with or observe activity in neighboring countries.

2.3 Existing Commercial Approaches

There have been a number of previous approaches to preventing hardware from making bad memory requests. We discuss existing commercial approaches here (summarized in Table 1) and defer discussion of other

	Provides Protection		Direct Access
	For OS	Between Processes	to physical memory
ATS-only IOMMU	✗	✗	✓
Full IOMMU	✓	✓	✗
IBM CAPI	✓	✓	✗
ARM TrustZone	✓	✗	✓
Border Control	✓	✓	✓

Table 1: Comparison of Border Control with other approaches.

approaches and conceptually similar work to Section 6.

Unlike CPUs, accelerators cannot perform page table walks, and rely on the Address Translation Service (ATS), often provided by the IOMMU [22–24]. The ATS takes a virtual address, walks the page table on behalf of the accelerator, and returns the physical address.

High-performance accelerators may cache the physical addresses returned by the ATS in accelerator TLBs. This lets them maintain physical caches to reduce access latency and more easily handle synonyms, homonyms, and coherence requests. In this design, the accelerator makes memory requests by physical address, either directly to the memory or by sending them through the IOMMU as "pre-translated" requests, which are not checked for correctness [22].

Other accelerators use only virtual addresses, and send requests via the IOMMU, which translates them and passes them on to memory. In this case, the IOMMU provides full protection in an analogous manner to how the CPU TLB provides protection between software programs. Translating every request at the IOMMU provides safety but degrades performance. It also prevents the accelerator from maintaining physical caches.

A similar approach to the IOMMU is IBM CAPI [12, 25], which allows FPGA accelerators to access system memory coherently. The CAPI interface implements TLBs and coherent, physical caches in the trusted hardware. This provides complete memory safety from incorrect accelerators. However, accelerator designers are unable to optimize the caches and TLB for the needs of their accelerator, and the loose coupling may result in longer TLB and cache access times.

An orthogonal approach to providing protection from untrusted hardware components is ARM TrustZone [28]. This approach divides hardware and software into the Secure world and the Normal world. Although this approach could be used to prevent untrusted accelerators from accessing host memory belonging to the OS, it is coarse-grained and cannot enforce protection between Normal world processes. For example, with TrustZone a misbehaving accelerator could access memory belonging to other users. In addition, TrustZone involves large hardware and OS modifications.

3. BORDER CONTROL

Border Control sandboxes accelerators by checking the access permissions for every memory request crossing the untrusted-to-trusted border (Figure 1). If the accelerator makes a read request (e.g., on a cache miss), Border Control checks that the process running on the accelerator has read permission for that physical page. If the accelerator makes a write request (e.g., on an accelerator cache writeback), the accelerator process must have write permission for that page. If the accelerator attempts to access a page for which it does not have sufficient permission, the access is not allowed to proceed and the OS is notified. In a correct accelerator design, these incorrect accesses should never occur. Detecting one therefore indicates that there is a problem with the accelerator hardware.

We first give an overview of the structures used for Border Control, then describe the actions that occur on various accelerator events. We then discuss how Border Control deals with multiprocess accelerators, as well as some accelerator design considerations.

3.1 Border Control Structures

Border Control sits between the physical caches implemented on the accelerator and the rest of the memory hierarchy—at the border between the untrusted accelerator and the trusted system (left side of Figure 2). Border Control checks coherence requests sent from the accelerator to the memory system. It is important for our approach to be light-weight and able to support high request rates.

Our design consists of two pieces: the *Protection Table* and the *Border Control Cache (BCC)* (right side of Figure 2). The Protection Table contains all relevant permission information and is sufficient for a correct design. The BCC is simply a cache of the Protection Table to reduce Border Control latency, memory traffic, and energy by caching recently used permission information.

3.1.1 Protection Table Design

The Protection Table contains all information about physical page permissions needed to provide security. We rely upon the insight that it is not necessary to know the mapping from physical to virtual page number to determine page permissions; instead, we simply need to know whether there exists a mapping from a physical page to a virtual page with sufficient permissions in the process's address space. This drastically reduces the storage overhead per page—from 64 bits in the process page table entry to 2 bits in the Protection Table.

We implement the Protection Table as a flat table in physical memory, with a read permission bit and write permission bit for each physical page number (PPN) (Figure 2). The Protection Table is physically indexed (in contrast to the process page table, which is virtually indexed), because lookups are done by physical address. There is one Protection Table per active accelerator; details for accelerators executing multiple processes concurrently are in Section 3.3. The Protection Table does not store execute permissions, because once a block is within the accelerator, Border Control cannot monitor or enforce whether the block is read as data or executed as instructions (see Section 2.2).

Because the minimum page size is 4KB, this implies a memory storage overhead of approximately 0.006% of

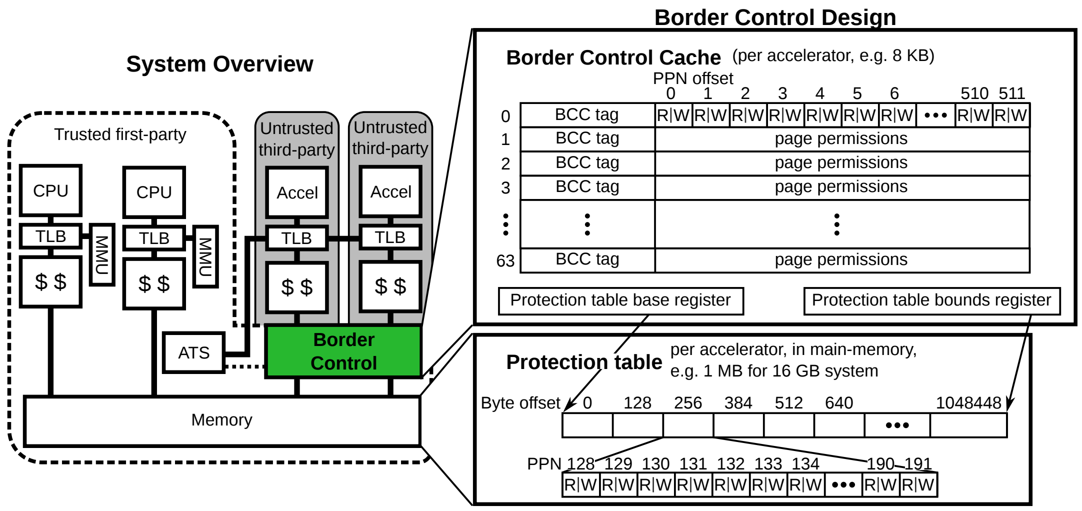

System Overview
Trusted first-party
CPU
TLB
$ $
MMU
CPU
TLB
$ $
MMU
Untrusted third-party
Accel
TLB
$ $
Untrusted third-party
Accel
TLB
$ $
ATS
Border Control
Memory
Border Control Design
Border Control Cache (per accelerator, e.g. 8 KB)
PPN offset
BCC tag
page permissions
R|W
Protection table base register
Protection table bounds register
Protection table per accelerator, in main-memory, e.g. 1 MB for 16 GB system
Byte offset 0 128 256 384 512 640 1048448
PPN 128 129 130 131 132 133 134 190 191

Figure 2: Implementation of Border Control. Left shows Border Control in the system. Right shows the Border Control design with a Protection Table in memory and a Border Control Cache between the accelerator caches and the shared memory system.

the physical address space per active accelerator. Thus, a system with 16GB of physical memory would require only 1MB for each accelerator's Protection Table. The flat layout guarantees that all permission lookups can be completed with a single memory access, which can proceed in parallel with read requests. Section 3.4.4 discusses handling large/huge pages.

The Protection Table is initialized to zero, indicating no access permissions. The Protection Table is updated on each accelerator request to the ATS (Address Translation Service). This occurs whether or not the accelerator caches the translation in a local TLB. The Protection Table is thus guaranteed to contain up-to-date permissions for any legitimate memory request from the accelerator—that is, any request where the physical address was originally supplied by the ATS.

When the accelerator makes a request for a physical address not first obtained from the ATS, we consider the behavior undefined and seek only to provide security. Since the accelerator is misbehaving, we do not attempt to provide the "true" page permissions as stored in the process page table. Thus, in some cases Border Control may report an insufficient permissions error although the process page table contains a mapping with sufficient permissions. However, safety is always preserved, and incorrect permissions are seen only when the accelerator misbehaves.

We expect the Protection Table will often be sparsely populated and an alternate structure could be more spatially efficient (e.g., a tree), or it could be stored in system virtual memory and allocated upon demand. However, the flat layout has small enough overhead that we do not evaluate alternate layouts.

3.1.2 *Border Control Cache Design*

While the Protection Table provides a correct implementation of Border Control, it requires a memory read for every request from the accelerator to memory, wasting memory bandwidth and energy. In addition, Border Control is on the critical path for accelerator cache misses. We therefore add a Border Control Cache (BCC) to store recently accessed page permissions.

The BCC caches the Protection Table and is tagged by PPN. To take advantage of spatial locality across physical pages [29], we store information for multiple physical pages in the same entry, similar to a subblock TLB [30]. This reduces the tag overhead per entry. We fetch an entire block at a time from memory; in our memory system the block size is 128 bytes, which corresponds to read/write permission for 512 pages per block. This gives our BCC a large reach—a 64-entry BCC contains permissions for 32K 4KB pages, for a total reach of 128MB. This cache is explicitly managed by the Border Control hardware and does not require hardware cache coherence, simplifying its design.

3.2 Border Control Operation

In this section, we describe the various actions that occur on several important events. The information is summarized in Figure 3. We discuss actions taken by Border Control when (a) a process starts executing on the accelerator, (b) the ATS completes a translation, (c) a memory request reaches the Border Control, (d) the page table changes, and (e) the process releases the accelerator upon termination.

3.2.1 *Process Initialization (Figure 3a)*

When a process starts executing on the accelerator, the OS sets up the ATS for the process by pointing it to the appropriate page table. If the accelerator was previously idle, the OS sets up the Protection Table by providing Border Control with a base register pointing to a zeroed region of physical memory. The OS also sets a bounds register to indicate the size of physical memory. If the accelerator was already running another process, the table will have already been allocated. Rather than setting the permissions for all pages at process initialization, Border Control initially assumes no permissions,

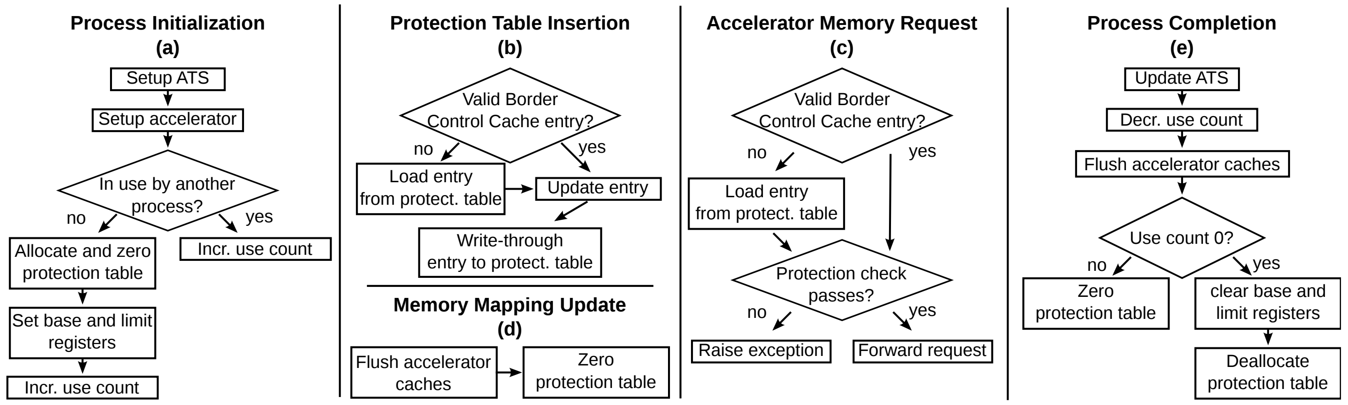

Process Initialization
(a)
Setup ATS
Setup accelerator
In use by another process?
no
yes
Allocate and zero protection table
Incr. use count
Set base and limit registers
Incr. use count
Protection Table Insertion
(b)
Valid Border Control Cache entry?
no
yes
Load entry from protect. table
Update entry
Write-through entry to protect. table
Memory Mapping Update
(d)
Flush accelerator caches
Zero protection table
Accelerator Memory Request
(c)
Valid Border Control Cache entry?
no
yes
Load entry from protect. table
Protection check passes?
no
yes
Raise exception
Forward request
Process Completion
(e)
Update ATS
Decr. use count
Flush accelerator caches
Use count 0?
no
yes
Zero protection table
clear base and limit registers
Deallocate protection table

Figure 3: Actions taken by Border Control on different events. Follows Section 3.2.

then lazily updates the table. This avoids the overhead of having to populate the entire large table at startup. However, the Protection Table always satisfies the invariant that no page ever has read or write permission in the Protection Table if it does not have it according to the process page table.

3.2.2 Protection Table Insertion (Figure 3b)

When the accelerator has a TLB miss, it requests an address translation from the ATS, which performs the translation and sends the result to both the accelerator TLB and Border Control. The ATS checks that the address space ID provided by the accelerator corresponds to a process running on the accelerator.

If there is an entry for this page in the BCC and it has the correct permissions, no action is taken. If there is an entry with fewer permissions (most commonly, because the entry contains information about many pages and this particular page has never been seen before), the entry is updated in the BCC, and the change is written back to the Protection Table. Otherwise, on a BCC miss, a new entry is allocated, and the BCC fetches the corresponding bits from the Protection Table and updates the permissions for the inserted page.

Updating the permissions stored at Border Control upon address translation reduces the complexity and overhead of our implementation. In a correctly functioning accelerator, the accelerator always receives a VPN-to-PPN translation from the ATS before making a memory request to that PPN. Thus, assuming zero permissions until a Protection Table insertion results in correct behavior without the overhead of determining and storing permissions for all entries in the process page table upon process initialization.

3.2.3 Accelerator Memory Request (Figure 3c)

Every accelerator memory request is checked by Border Control before it reaches the host memory system. Border Control first looks up the PPN in the BCC. On a BCC hit for the PPN, the permission bits are retrieved from the BCC entry. On a BCC miss, Border Control allocates an entry for the page, and fills it with the corresponding bits from the Protection Table. The Protection Table is only checked after ensuring that the physical address is within the bounds register.

If the request requires permissions not indicated in the BCC or Protection Table, Border Control notifies the OS. The OS can act accordingly by terminating the process or disabling the accelerator. In addition, any requested read data from the page is not returned to the accelerator, and any write request does not proceed.

3.2.4 Memory Mapping Update (Figure 3d)

During process execution, the page table may be updated for several reasons. If a new translation from virtual to physical page number is added, the Border Control takes no action (similar to the TLB). This case occurs frequently, because the OS lazily allocates physical pages to virtual pages.

When an existing virtual-to-physical mapping changes, a TLB shootdown occurs, and Border Control also needs to take action. In a TLB shootdown, the TLB either has the offending entry invalidated, or the entire TLB is flushed. In either case, no further action is needed on a CPU or trusted accelerator. The TLB will obtain a new entry from the page walk, and any dirty blocks in the caches are stored by physical address and can be correctly written back to memory upon eviction.

With Border Control, any dirty cache blocks from the affected page must be written back to memory before the Protection Table and BCC are updated on permission downgrades. Otherwise, if the accelerator is caching dirty copies of data from that page—even if it was legal for the accelerator to write to those addresses at the time—Border Control will report an error and block the subsequent writeback.

If a mapping changes and the page had read-only permission, the Protection Table and BCC entry can simply be updated, because no cached lines from the page can be dirty. If the page has a writable permission bit set in the Protection Table, then it has previously been accessed by the accelerator and the accelerator's caches may contain dirty blocks from the page. The accelerator can flush all blocks in its caches, or, as an optimization, selectively flush only blocks from the affected page.

Once the cache flush is complete, the corresponding entry in the BCC and the Protection Table should be updated. Alternately, if the entire accelerator cache is flushed, the Protection Table can be zeroed and the

BCC and accelerator TLB can be invalidated; these are equivalent in terms of correctness.

Mapping updates resulting in TLB shootdowns commonly occur for a few reasons: context switches, process termination, swapping, memory compaction, and copy-on-write. Context switches, process termination, memory compaction, and swapping are expected to be infrequent and slow operations. For copy-on-write pages, the mapping change will not incur an accelerator cache flush, because the page was previously read-only and thus could not have been dirty. Copy-on-write thus incurs no extra overhead over the trusted accelerator case. In our workloads, we never encountered a permission downgrade. We evaluate permission downgrade overheads in 5.2.4.

Even if the accelerator ignores the request to flush its caches, there is no security vulnerability. If the cache contains dirty blocks that are not flushed, they will be caught later when the accelerator cache attempts to write them back to memory. This will raise a permission error, and the writeback will be blocked.

3.2.5 Process Completion (Figure 3e)

When a process finishes executing on an accelerator, the accelerator caches should be flushed, all entries in the BCC and accelerator TLB invalidated, and the Protection Table zeroed. This ensures that all dirty data is written back to memory, and access permissions for that process are revoked from the accelerator. If the accelerator is not running any other processes, the OS can reclaim the memory from the Protection Table.

3.3 Multiprocess Accelerators

Border Control allows accelerators to safely use physically tagged caches, making it straightforward to run multiple processes simultaneously. The overhead of Border Control is per-accelerator and thus is not higher for multiprocess accelerators. When an accelerator is running multiple processes simultaneously, Border Control checks whether *at least one process* is able to access a page when determining permissions; the permissions we use are the union of those for all processes currently running on the accelerator.

Although this design choice may seem to weaken the safety properties of Border Control, we argue that this is not the case. The purpose of Border Control is to protect the rest of the system from incorrect accelerator memory accesses. Just as Border Control does not inspect the computations internal to the black box of the accelerator, it cannot prevent a buggy accelerator from allowing incorrect accesses to data stored in accelerator caches by other co-scheduled processes. For example, if a process exposes sensitive data to the accelerator, the accelerator can immediately leak the data to another process address space to which it has write access, or store it internally and leak it later. The accelerator is sandboxed from the rest of the system, but interference between processes within the sandbox is possible in an incorrectly implemented accelerator.

The OS can deal with security threats internal to the accelerator by simply not running sensitive or critical processes on untrusted accelerators. Dealing with misbehavior internal to the black box of the accelerator is outside the scope of our threat model.

3.4 Border Control Design Considerations

We briefly discuss several considerations with our Border Control implementation, including other permission sources than the page table, constraints on cache organization, and dealing with different page sizes.

3.4.1 Alternate Permission Sources

The access permissions stored in the process page tables are generally a good source of permission information for Border Control. However, in some cases, alternate sources may provide more fine-grained control. In particular, the OS might run an accelerator kernel directly. Because the OS has access to every page in the system, this would eliminate the memory protection from the accelerator. A simple way to handle this case is for the OS to provide an alternate (shadow) page table for the accelerator. This can already be done in implementations of the ATS as part of the IOMMU and requires no changes to the Border Control hardware.

Border Control can also be extended to work with alternate permission systems, provided that permissions correspond to physical addresses. For example, in Mondriaan Memory Protection [31], the accelerator keeps a Protection Lookaside Buffer (PLB) rather than a TLB. On a PLB miss, Border Control can update the Protection Table, just as it would on a TLB miss.

Capabilities are another access control method [32]. The accelerator cannot be allowed to directly access capability metadata, or it could forge capabilities. However, if the capabilities correspond to physical addresses and can be cached in a structure at the accelerator, the Protection Table and BCC can store and verify these permissions. For permissions at finer granularities than 4KB pages, an alternate format for Border Control's Protection Table and BCC may be more appropriate, to reduce storage overhead.

3.4.2 Virtualization

We have defined Border Control to operate with an unvirtualized trusted OS. Border Control can also operate with a trusted Virtual Machine Monitor (VMM) below guest OSes. In this case, the VMM allocates the Protection Table in (host physical) memory that is inaccessible to guest OSes. The present implementation works unchanged because table indexing uses "bare-metal" physical addresses: unvirtualized or host.

3.4.3 Cache Organization Requirements

Our implementation of Border Control places minor constraints on accelerator cache organizations. One invariant Border Control requires of cache coherence is that an untrusted cache should never provide data for a block for which it does not have write permission. This is simple to enforce in an inclusive or non-inclusive M(O)(E)SI cache: the ownership of non-writable blocks should always remain with the directory or the trusted cache hierarchy—for example, by not allowing a read-only request to be answered with an owned E state.

For exclusive caches, this invariant requires exclusive cache behavior to be slightly modified. Specifically, when a block is sent from a shared cache to the private accelerator cache, it is normally invalidated at the shared cache, so that there is at most one level of the cache hierarchy holding the block. If the block was dirty when sent to the accelerator, the copy in memory will be stale and the accelerator will need to write back data for a block for which it does not have write permission, violating the above invariant. A solution is to require that any dirty block requested by the accelerator with read-only permissions first be written back to memory.

3.4.4 Page Size

We assumed above that page size is 4KB, which is the minimum page size on most systems. However, some workloads see significant performance benefits when using larger page sizes. Our implementation of Border Control works with larger page sizes. When inserting a new translation for a large page, we can update the Protection Table and BCC entries for every 4KB page covered by the large page. Thus, for a 2MB page, Border Control updates the entries for 512 entries. This is the size of a single BCC entry or one memory block in our system with 128 byte blocks, so using 2MB pages does not cause any difficulties. More complex and efficient mechanisms may be required if future accelerators commonly use large pages.

4. BORDER CONTROL FAQ

We seek to anticipate and answer questions readers may have regarding Border Control and its alternatives.

Why not perfectly verify accelerator hardware?
Perfect verification of accelerator hardware would eliminate the problems associated with malicious or buggy hardware design. However, accelerators that wish to access shared memory are likely to be too complex to easily validate, especially in the presence of die stacking and/or malicious hardware which is intended to be difficult to detect [33, 34]. Even first-party hardware has bugs, as shown by the errata released by companies such as Intel and AMD [17,19–21,35]. Border Control makes perfect verification unnecessary by sandboxing memory errors, so that they can only affect processes running on the imperfect accelerator.

Why not eschew third-party accelerators?
Using only trusted (first-party) hardware could prevent the accelerator from generating incorrect memory requests, as all generation and propagation of physical addresses would occur in trusted hardware. However, we hypothesize that with the increasing prevalence of accelerators, there will be economic benefits to being able to purchase IP from third parties.

Why not forbid use of physical addresses?
Instead of disallowing all caches, Border Control can also be implemented by preventing the accelerator from using physical addresses, but letting it use caches with virtual addresses. Then the virtual-to-physical translation occurs at the border, giving the accelerator no way to access physical pages belonging to other processes.

There are a number of known drawbacks to virtual caches [36, 37], including problems with synonyms and homonyms. In particular, implementing coherence is significantly more difficult, especially because current CPU caches use physical addresses only. Recently proposed designs with virtual accelerator caches require modifications to the CPU coherence protocol (VIPS-M [38]), or do not allow homonyms or synonyms in the accelerator caches (FUSION [10]). Border Control allows standard cache coherence using physical addresses, including handling homonyms and synonyms.

Why not limit to "safe" memory ranges?
The system could use static regions and mark ranges of physical addresses which the accelerator is allowed to access, requiring applications to (either explicitly or implicitly) copy data into the accelerator address space. However, copying data between address spaces incurs a performance penalty, and it may be difficult to determine a priori which memory the accelerator will use. Also, limiting data to a contiguous range may also require rearranging the layout of data and updating pointers accordingly. Border Control allows full shared virtual memory between the accelerator and CPU.

Why not allow caches and TLBs and do address translation again at the border to verify?
Border Control could be implemented by doing a backwards translation from physical to virtual addresses and checking permissions with the process page table. Both Linux and Windows contain OS structures to perform reverse translation for swapping. However, these structures are OS-specific, and thus not good candidates for being accessed by hardware. Instead, this would require frequent traps into the OS, which may have performance impacts both on the accelerator process and host processes. Our implementation of Border Control relies on the insight that performing a reverse translation not required to determine access permissions.

5. EVALUATION

5.1 Methodology

Border Control aims to provide safety without significant performance or storage overheads. We quantitatively evaluate Border Control and several other safety approaches on the GPGPU, a high-performance accelerator which is capable of high memory traffic rates and irregular memory reference patterns. A GPGPU is a stress-test for memory safety mechanisms.

We summarize the configurations for each approach in Table 2 and describe them in detail below. We evaluate five approaches to memory safety. We use the unsafe *ATS-only IOMMU* as a baseline, where the IOMMU serves only to perform initial address translation but the GPU uses physical addresses in its TLB and caches. This enables the GPU to use all performance optimiza-

	Safe?	L1 $	L1 TLB	L2 $	BCC
ATS-only IOMMU	✗	✓	✓	✓	N/A
Full IOMMU	✓	—	—	—	N/A
CAPI-like	✓	—	—	✓	N/A
Border Control-noBCC	✓	✓	✓	✓	—
Border Control-BCC	✓	✓	✓	✓	✓

Table 2: Comparison of configurations under study.

tions, and is the norm in coherent integrated GPUs today, but does not satisfy our security goal.

The *full IOMMU* configuration is a simple approach to safety, appropriate for low-performance accelerators. For the IOMMU to enforce safety, the accelerator must issue every memory request as a virtual address to the IOMMU, which performs translation and permissions checking. We therefore remove the accelerator caches and accelerator TLB, but leave the L2 TLB because the IOMMU caches translations. Although this configuration is unrealistic for high-performance, high-bandwidth accelerators like GPUs, we include it to illustrate the challenges of providing safety using existing mechanisms.

The *CAPI-like* configuration is modeled on the philosophy of IBM CAPI, where the accelerator caches and TLB are implemented in the trusted system. Thus, they are more distant and less tightly integrated with the accelerator. We model the longer latency to the trusted accelerator cache by including only a shared L2 cache.

We evaluate two Border Control configurations. First, *Border Control-noBCC* includes the Protection Table for safety but does not include the BCC, to show the impact of caching the permission information. Border Control allows standard GPU performance optimizations: accelerator L1 and L2 caches and accelerator TLBs. The *Border Control-BCC* configuration adds in the BCC for higher performance.

To explore the varying pressures different accelerators put on Border Control, we evaluate two GPU configurations, a highly threaded GPU and a moderately threaded GPU. The *highly threaded GPU* is similar to a modern integrated GPU (e.g., AMD Kaveri [14]) with eight compute units, each with many execution contexts. This configuration is a proxy for a high-performance, latency-tolerant accelerator. The *moderately threaded GPU* has a single compute unit and can execute a single workgroup (thread block) at a time, but it executes multiple wavefronts (execution contexts). This configuration is a proxy for a more latency-sensitive accelerator.

We use a variety of workloads to explore a diverse set of high-performance accelerator behaviors. We evaluate Border Control with workloads from the Rodinia benchmark suite [39], including machine-learning, bioinformatics, graph, and scientific workloads. These workloads range from regular memory access patterns (e.g., `lud`) to irregular, data-dependent accesses (e.g., `bfs`). They use a unified address space between the CPU and GPU rather than explicit memory copies.

CPU	
CPU Cores	1
CPU Caches	64KB L1, 2MB L2
CPU Frequency	3 GHz
GPU	
Cores (highly threaded) Cores (moderately threaded)	8 1
Caches (highly threaded) Caches (moderately threaded)	16KB L1, shared 256KB L2 16KB L1, shared 64KB L2
L1 TLB	64 entries
Shared L2 TLB (trusted)	512 entries
GPU Frequency	700 MHz
Memory System	
Peak Memory Bandwidth	180 GB/s
Border Control	
BCC Size	8KB
BCC Access Latency	10 cycles
Protection Table Size	196KB
Protection Table Access Latency	100 cycles

Table 3: Simulation configuration details.

To simulate our system, we use the open-source gem5-gpu simulator [40], which combines gem5 [41] and GPGPU-Sim [42]. We integrated our Border Control implementation into the coherence protocol. Table 3 contains the details of the system we simulated. We use a MOESI cache coherence protocol with a null directory for coherence between the CPU and the GPU. Within the GPU, we use a simple write-through coherence protocol. This system is similar to current systems with an integrated GPU (e.g., AMD Kaveri [14]), and has increased memory bandwidth to simulate future systems.

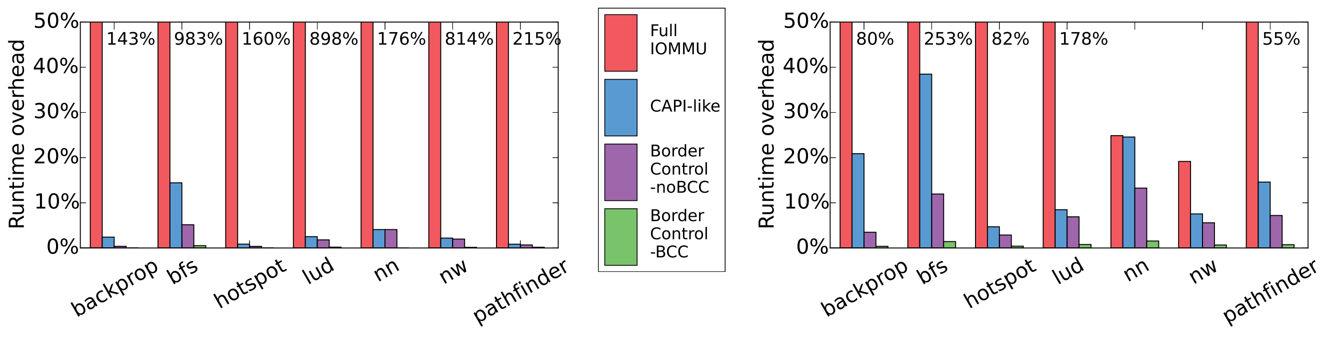
5.2 Results

To better understand the tradeoff between performance and safety, we evaluate the runtime overheads of Border Control and the other approaches to safety compared to the unsafe baseline (ATS-only IOMMU). Figure 4 shows the runtime overhead of the approaches we evaluate, for both GPU configurations.

ATS-only IOMMU The ATS-only IOMMU has no overhead, and all other configurations are normalized to it. However, it does not provide any safety guarantees.

Full IOMMU The first bar (red) in Figure 4 shows the impact of the full IOMMU: geometric mean of 374% runtime overhead for the highly threaded case and 85% for moderately threaded. The overhead is higher in the highly threaded case because the 8 core GPU is capable of issuing a high bandwidth of requests, and without the L2 to filter some of this bandwidth, the DRAM is overwhelmed and performance suffers. The moderately threaded case does not saturate the DRAM bandwidth and sees less performance degradation, but runtime overhead is still impractically high.

CAPI-Like (second, blue bar of Figure 4). The average runtime overhead for the CAPI-like configuration is only 3.81% in the highly threaded case. There are a few benchmarks (e.g., pathfinder, hotspot) that show almost no performance degradation. This is not surprising, as the GPU is designed to tolerate high memory latency. However, the moderately threaded case has higher overhead, at 16.5% on average. This is because



(a) ***Highly threaded GPU*** (b) ***Moderately threaded GPU***

Figure 4: Runtime overhead compared to ATS-only IOMMU (which has 0% overhead).

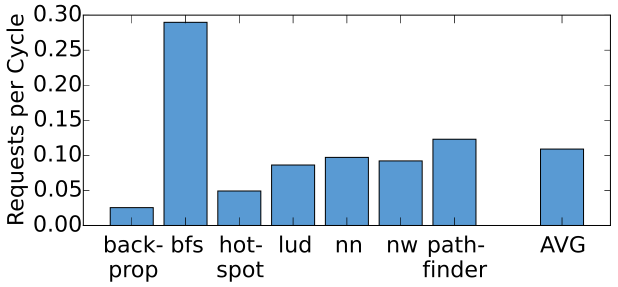


Figure 5: Number of requests per cycle checked by Border Control for the highly threaded GPU.

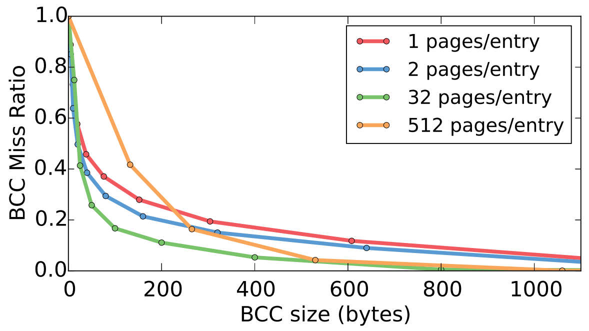


Figure 6: BCC miss ratio when varying BCC size and number of pages per entry.

the moderately threaded GPU does not have enough execution contexts to hide the increased memory latency. The CAPI-like configuration has lower overhead than the full IOMMU, but may still cause significant performance degradation, especially for more latency-sensitive accelerators.

Border Control-noBCC (third, purple bar of Figure 4). Border Control without a BCC has on average 2.04% runtime overhead for highly threaded and 7.26% for moderately threaded. Similarly to the CAPI-like configuration, the extra latency and bandwidth from accessing the protection table affects the the moderately threaded GPU more than the highly threaded GPU. Even without a BCC, Border Control reduces execution overhead compared to the IOMMU and CAPI-like configurations.

Border Control-BCC (fourth, green bar of Figure 4). Border Control with a BCC has an average runtime overhead for the highly threaded GPU of just 0.15%, and 0.84% for the moderately threaded case. The BCC eliminates the additional memory access on each accelerator memory access, improving performance.

In summary, Border Control with a BCC provides safety with almost no overhead over the unsafe baseline. For the highly threaded case, Border Control attains an average of 4.74× speedup over the full IOMMU and 1.04× over the CAPI-like configuration. For the moderately threaded case, Border Control is on average 1.83× faster than the full IOMMU and 1.16× faster than the CAPI-like configuration.

5.2.1 *Border Control Requests*

Figure 5 shows the number of requests per cycle checked by Border Control for the highly threaded GPU. On average, we saw approximately 0.11 requests per cycle, but there is significant variability: from 0.025 for `backprop` to 0.29 for `bfs`. From this data, we conclude that bandwidth at Border Control is not a bottleneck. This is not surprising since the private accelerator caches provide good bandwidth filtering for the rest of the system.

5.2.2 *BCC Sensitivity Analysis*

We show the results of a sensitivity analysis of BCC size in Figure 6, which gives the miss ratio at the BCC as size increases. Each line represents a different BCC entry size, from 1 page/entry (2 bits) up to 512 pages/entry (1024 bits) with the addition of a 36-bit tag per entry. The miss ratio is averaged over the benchmarks.

For the workloads we evaluate, storing multiple pages per entry shows a large benefit, especially when storing 512 pages (a reach of 2MB) per entry. Since there is spatial locality, larger entries reduce the per-page tag overhead. For a 1KB BCC with 512 pages/entry, the average BCC miss rate is below 0.1% for our workloads. However, we conservatively use an 8KB BCC, as it is still a small area overhead and may be more appropriate for larger, future workloads.

5.2.3 *Area and Memory Storage Overheads*

Our Border Control mechanism has minimal space overhead. The Protection Table uses 0.006% of physical memory capacity per active accelerator. The BCC was effective for our workloads with 64 entries of 128 bytes each, for a total of 8KB.

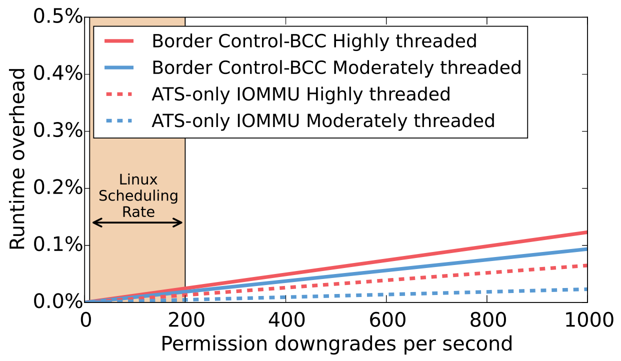


Figure 7: Runtime overhead with varying frequencies of page permission downgrades.

5.2.4 *Sensitivity to Memory Mapping Updates*

When memory mappings are updated and page permissions are downgraded, any accelerator must finish all outstanding requests (where most of the downgrade time is spent), invalidate its TLB entries, and the ATS must flush its caches. These actions occur even with trusted accelerators. Our implementation of Border Control maintains safety and correctness by also flushing the accelerator caches, the Protection Table, and the BCC, which incurs additional overhead (details discussed in Section 3.2.4). Figure 7 shows that the overhead for permission downgrades is negligible (approximately 0.02%) in the most common case of downgrades today, context switches (10–200 downgrades per second). Additionally, the overhead for permission downgrades continues to be small even when the rate of downgrades is much higher, as may be common in future systems. Figure 7 also shows that Border Control incurs roughly twice the overhead of the unsafe ATS-only IOMMU. This increased overhead is mostly due to writing back dirty data in the accelerator L2 cache (highly threaded) or compulsory cache misses (moderately threaded).

6. RELATED WORK

Some previous work is similar to Border Control in intent or mechanism. Current commercial approaches to protection against bad memory accesses were discussed in Section 2.3. We briefly discuss other work below.

rIOMMU [43] decreases IOMMU overheads for high-bandwidth I/O devices such as NICs and PCIe SSD drives. It drastically reduces IOMMU overheads while ensuring safety, but relies on the device using ring buffers, which are accessed in a predictable fashion. rIOMMU is not intended for devices that make unpredictable fine-grained memory accesses.

Hardware Information Flow Tracking (IFT) can provide comprehensive system safety guarantees though tracking untrusted data at the gate level [44]. However, this protection incurs large overheads relative to unsafe systems. Border Control focuses on one specific safety guarantee, but works with existing systems with very low overhead.

Hive [45] is an operating system designed for the Stanford FLASH [46] to provide fault containment. It protects against wild writes to remote memory by using permission vectors, to simplify recovery in the presence of hardware and software faults. Hive assumes that the hardware is correctly implemented but may have (non-Byzantine) software or hardware faults.

Intel's noDMA table and AMD's Device Exclusive Vector (DEV) were designed to protect against buggy or misconfigured devices by allowing the system to designate some addresses as not accessible by devices, and have been subsumed by the IOMMU.

Some previous work decouples translation and protection for all memory accesses. HP's PA-RISC and the single address space operating system [47,48] and Mondriaan memory protection [31] have some similarities to our implementation of Border Control, but with different assumptions and goals. The single address space OS attempts to simplify sharing and increase cache access speed. Mondriaan provides fine-grained (word or even byte-level) permissions, enabling fine-grained sharing and zero-copy networking. These approaches are replacements for the current page-based memory protection system, and rely on correct hardware. Border Control can use permissions from these approaches to provide memory protection.

Some approaches use metadata to enforce safety guarantees. The IBM System/360 and successors associate protection keys with each physical page [49] and check them before writes for software (but not hardware) security. The SPARC M7 provides Realtime Application Data Integrity (ADI) to help prevent buffer overflows and stale memory references [5], requiring pointers to have correct version numbers. However, version numbers may be guessable by the accelerator.

Intel Trusted Execution Technology is a hardware extension that protects against attacks by malicious software and firmware [50]. It protects the launch stack and boot process from threats, but not against malicious or buggy hardware components in the system.

7. CONCLUSION

Specialized hardware accelerators, including ones that directly access host system memory, are becoming more prevalent and more powerful, leading to new security and reliability challenges. In particular, incorrect memory accesses from accelerators can cause leaks, corruption, or crashes even for processes not running on the accelerator. Border Control provides full protection against incorrect memory accesses while maintaining high performance, and with low storage overhead.

8. ACKNOWLEDGEMENTS

We would like to thank Arka Basu, Tony Nowatzki, Marc Orr, Guri Sohi, Mike Swift, and Hongil Yoon for feedback on this work. This work is supported in part by the National Science Foundation (CCF-1218323, CNS-1302260, CCF-1438992, CCF-1533885), Cisco Systems Distinguished Graduate Fellowship, John P. Morgridge Chair, Google, and the University of Wisconsin–Madison (Named Professorship). Hill and Wood have a significant financial interest in AMD and Google.

9. REFERENCES

[1] "Regulation (EC) No 562/2006 of the European Parliament and of the Council of 15 March 2006 establishing a Community Code on the rules governing the movement of persons across borders (Schengen Borders Code)," 2006.

[2] W.-C. Park, H.-J. Shin, B. Lee, H. Yoon, and T.-D. Han, "RayChip: Real-time ray-tracing chip for embedded applications," in *Hot Chips 26*, 2014.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO-45*, 2012.

[4] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *MICRO-46*, 2013.

[5] S. Phillips, "M7: Next generation SPARC," in *Hot Chips 26*, 2014.

[6] K. Atasu, R. Polig, C. Hagleitner, and F. R. Reiss, "Hardware-accelerated regular expression matching for high-throughput text analytics," in *FPL 23*, 2013.

[7] V. Rajagopalan, "All programmable devices: Not just an FPGA anymore," MICRO-45, 2013. Keynote presentation.

[8] B. Black, "Die stacking is happening!." MICRO-45, 2013. Keynote presentation., Dec. 2013.

[9] P. Rogers, "Heterogeneous system architecture overview," in *Hot Chips 25*, 2013.

[10] S. Kumar, A. Shriraman, and N. Vedula, "Fusion : Design tradeoffs in coherent cache hierarchies for accelerators," in *ISCA 42*, 2015.

[11] J. Sell and P. O'Connor, "The Xbox One system on a chip and Kinect sensor," *IEEE Micro*, vol. 34, Mar. 2014.

[12] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, pp. 7:1–7:7, Jan. 2015.

[13] P. Hammarlund, "4th generation Intel core processor, codenamed haswell," in *Hot Chips 26*, 2014.

[14] AMD, "AMD's most advanced APU ever." `http://www.amd.com/us/products/desktop/processors/a-series/Pages/nextgenapu.aspx`.

[15] J. Goodacre, "The evolution of the ARM architecture towards big data and the data-centre." http://virtical.upv.es/pub/sc13.pdf, Nov. 2013.

[16] US Department of Defense, "Defense science board task force on high performance microchip supply," 2005.

[17] M. T. Inc, *MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0*. May 1994.

[18] "Zynq-7000 all programmable SoC." `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`, 2014.

[19] A. L. Shimpi, "AMD's B3 stepping Phenom previewed, TLB hardware fix tested," Mar. 2008.

[20] I. Corporation, "Intel Core i7-900 desktop processor extreme edition series and Intel Core i7-900 desktop processor series specification update." http://download.intel.com/design/processor/specupdt/320836.pdf, May 2011.

[21] *Intel Xeon Processor E5 Family: Specification Update*, Jan. 2014.

[22] *AMD I/O Virtualization Technology (IOMMU) Specification, Revision 2.00*, Mar. 2011.

[23] *Intel Virtualization Technology for Directed I/O, Revision 2.3*, Oct. 2014.

[24] *ARM System Memory Management Unit Architecture Specification, SMMU architecture version 2.0*, 2012-2013.

[25] J. Stuecheli, "Power8," in *Hot Chips 25*, 2013.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *SOSP 14*, 1993.

[27] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proc. of the IEEE*, vol. 63, Sept 1975.

[28] *ARM Security Technology: Building a Secure System using TrustZone Technology.*

[29] M. Gorman, "Understanding the Linux virtual memory manager," 2004.

[30] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," in *ASPLOS VI*, 1994.

[31] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *ASPLOS X*, 2002.

[32] H. M. Levy, *Capability-Based Computer Systems*. Digital Press, 1984.

[33] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *SP*, 2011.

[34] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating UCI: Building stealthy and malicious hardware," in *SP*, 2011.

[35] D. Price, "Pentium FDIV flaw-lessons learned," *Micro, IEEE*, vol. 15, pp. 86–88, Apr. 1995.

[36] M. Cekleov and M. Dubois, "Virtual-address caches part 1: Problems and solutions in uniprocessors," *IEEE Micro*, vol. 17, Sept 1997.

[37] M. Cekleov and M. Dubois, "Virtual-address caches, part 2: Multiprocessor issues," *IEEE Micro*, vol. 17, Nov 1997.

[38] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *ISCA 40*, 2013.

[39] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[40] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *Computer Architecture Letters*, vol. 13, no. 1.

[41] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *CAN*, 2011.

[42] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[43] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers," in *ASPLOS 20*, 2015.

[44] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levil, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and I/O security system with strict and provable information flow security," in *ISCA 38*, 2011.

[45] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," in *SOSP 15*, 1995.

[46] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH multiprocessor," in *ISCA 21*, 1994.

[47] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architectural support for single address space operating systems," in *ASPLOS V*, 1992.

[48] J. Wilkes and B. Sears, "A comparison of protection lookaside buffers and the PA-RISC protection architecture," Tech. Rep. HPL-92-55, Hewlett Packard Labs, 1992.

[49] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development*, vol. 8, pp. 87–101, Apr. 1964.

[50] J. Greene, "Intel trusted execution technology," *Intel Technology Whitepaper*, 2012.