Crossing Guard: Mediating Host-Accelerator Coherence Interactions

Lena E. Olson Mark D. Hill David A. Wood
{lena, markhill, david}@cs.wisc.edu

ABSTRACT

Specialized hardware accelerators have performance and energy-efficiency advantages over general-purpose processors. To fully realize these benefits and aid programmability, accelerators may share a physical and virtual address space and full cache coherence with the host system. However, allowing accelerators – particularly those designed by third parties – to directly communicate with host coherence protocols poses several problems. Host coherence protocols are complex, vary between companies, and may be proprietary, increasing burden on accelerator designers. Bugs in the accelerator implementation may cause crashes and other serious consequences to the host system.

We propose *Crossing Guard*, a coherence interface between the host coherence system and accelerators. The Crossing Guard interface provides the accelerator designer with a standardized set of coherence messages that are simple enough to aid in design of bug-free coherent caches. At the same time, they are sufficiently complex to allow customized and optimized accelerator caches with performance comparable to using the host protocol. The Crossing Guard hardware is implemented as part of the trusted host, and provides complete safety to the host coherence system, even in the presence of a pathologically buggy accelerator cache.

1. INTRODUCTION

Specialized hardware accelerators have recently been proposed for a wide range of applications [1, 2, 3, 4, 5, 6, 7, 8]. Some complex accelerators can benefit from the ability to share a unified address space with the host system. In particular, future accelerators may wish to share data with the host at a fine granularity, where the particular data to be accessed is not known a priori.

For many years, CPUs have optimized fine-grained, data-dependent accesses through the use of hardware cache coherence. Hardware coherence aids in both performance and programmability by avoiding the need for the programmer to explicitly manage where data is located in memory and the cache hierarchy. There is an increasing recognition that accelerators can benefit from hardware coherence as well [9, 10, 11].

There are several major differences between coherence protocols designed for CPUs and those designed for accelerators. First, some accelerators may benefit from specific accelerator cache organizations. Unlike CPUs, which are general-purpose and may exhibit many access patterns, accelerators may access data in very specific ways. For example, a video decoder may be block-based, while a graph processing accelerator may make many data-dependent accesses.

The coherence and cache organization design space is large and provides many opportunities for optimizations: caches may be inclusive, exclusive, or non-inclusive; write-back or write-through; write allocate or write no-allocate; prefetching or not. A simple coherence protocol may include only Invalid (I) and Valid (V) states; most coherence protocols also include states such as M (Modified), S (Shared), E (Exclusive), O (Owned) and/or F (Forward) [12]. Accelerator designers may choose to allocate cache resources based on their knowledge of likely data access patterns. For example, an accelerator that performs mostly streaming accesses may benefit from prefetching more than an accelerator that performs mostly highly data-dependent, unpredictable accesses. A GPGPU might have caches designed specifically for write-coalescing. Similarly, the FUSION accelerator coherence protocol [10] is designed to migrate data between a set of cooperating accelerators working on the same application. **Giving accelerator designers extensive flexibility will lead to better accelerator performance.**

Second, any particular accelerator may be integrated with a range of host systems. These host systems will be designed by different companies and will have different coherence protocols. For example, Intel uses an inclusive cache hierarchy with a MESI(F) protocol, ARM uses a MESI-like protocol and inclusive or exclusive caches, and AMD uses an exclusive MOESI protocol. In addition, the specific messages that these host systems use may be proprietary and unknown to the accelerator designer. Even if the protocol is known, it is a substantial burden for the accelerator designer to design and implement different caches for each host system. **Giving accelerator designers a single standard coherence interface will ease accelerator development.**

Third, accelerator designers may not have the same expertise in developing complex coherence protocols as host designers. Coherence protocols are notoriously difficult to design and implement correctly [13]. For performance reasons, protocols may allow messages between components of the coherence system to race. Direct communication between caches at the same level of the hierarchy, such as between L1s, also improves performance at the cost of complexity. Much of the difficulty of implementing coherence protocols comes from handling these cases. **Giving accelerator designers a simple coherence interface will make it easier for them to implement and verify correct coherent accelerator caches.**

Finally, accelerators may not share the same presumption of correctness and trustworthiness as CPUs. If the accelerator cache incorrectly discards a request from the host rather than responding to it, the host system may wait indefinitely. Over time, this bug could consume host resources until the host system is rendered unusable – including for processes that *never use the accelerator*. Similarly, unexpected messages from the accelerator may cause cache controller errors. It is even possible that an accelerator could contain malicious hardware that could be triggered to perform incorrect coherence actions as a denial of service attack. **Giving host designers a standardized coherence interface will allow them to limit the incorrect actions an accelerator can perform, reducing the attack surface and improving reliability and security.**

Industry recognizes the emerging importance of these issues and is developing initial interfaces between third-party (FPGA) accelerators and host systems. IBM's Coherent Accelerator Processor Interface (CAPI) [14, 15] provides a simple memory access interface for the accelerator developer and safety for the host with the Coherent Accelerator Processor Proxy (CAPP) on the CPU and the Power Service Layer (PSL) on the FPGA. The PSL supplies the accelerator with a cache, and the CAPP acts as a directory of lines at the accelerator, forwarding probes and keeping the PSL coherent. Virtual-to-physical address translation is also handled by the PSL. CAPI therefore provides the accelerator with an interface to which it can make loads and stores by virtual address, but does not allow it to build customized coherent caches on top of the PSL cache. In addition, CAPI works with the IBM POWER8, but is not a general interface for hosts designed by different companies.

ARM also provides a coherence interface in the form of the ARM Accelerator Coherency Port (ACP) [16], which allows accelerators to make coherent requests by physical address. However, it does not pass invalidations to the accelerator, preventing an accelerator designer from implementing fully coherent accelerator-side caches.

To provide a more interoperable interface, a group of seven companies–including IBM and ARM–recently formed the Cache Coherent Interconnect for Accelerators (CCIX) Consortium [17]. CCIX promises interoperability across host systems with different ISAs, as well as higher bandwidth and improved latency compared to current interfaces. While specific details of the interface and how accelerators will interact with it have not been released, the formation of the CCIX consortium indicates that industry values interoperable coherent third-party accelerators. In the best case, Crossing Guard can influence future versions of industrial approaches like CCIX.

This paper explores the design space of safe and standardized accelerator-host coherence interfaces, including both interfaces that allow customized accelerator caches and those that do not. We propose *Crossing Guard*: host hardware that forms an interface between the host system and the accelerator and which mediates coherence interactions between the host and the accelerator. Crossing Guard is similar to the software concept of an Application Program Interface (API), which provides a public interface which is simple and stable. Meanwhile, the back-end implementation can be treated as a black box and can change without notifying users of the API. The narrow set of interactions that are allowed by the API (or Crossing Guard) limits the potential error cases.

Crossing Guard provides benefits to both the host and accelerator designers. For the host designer, Crossing Guard provides a set of safety guarantees even in the presence of a buggy or malicious accelerator. Crossing Guard:

- Allows accelerators to use customized caches
- Provides a simple, standardized coherence interface
- Provides safety and reliability for the host system

We define a standardized set of coherence requests and responses for accelerators, and implement two accelerator coherence protocols that use the interface. The first is a single-level high-performance MESI cache, which requires only 4 stable states and single transient state. The second is a two-level accelerator protocol, with an L2 shared between multiple accelerator cores running the same application.

We implement Crossing Guard in a heterogeneous system where CPUs and GPGPUs coherently access shared memory. To demonstrate that it allows freedom for the host designer, we show that the CPUs can use an exclusive, AMD Hammer-like protocol or an inclusive MESI protocol. We stress-test the accelerator protocol by sending a stream of random requests from each accelerator core, and find that Crossing Guard's interface provides correct coherence operation. We then bombard the Crossing Guard with a stream of random coherence messages to random addresses, and find that Crossing Guard provides safety even when the accelerator is behaving badly: this fuzz testing never leads to a crash or deadlock. In addition, Crossing Guard performs similarly to the unsafe, hard-to-design accelerator-side cache and better than a safe but high-latency host-side cache.

2. CROSSING GUARD

Crossing Guard consists of hardware that implements a simple, standardized interface for coherence and translates requests between the host and the accelerator. It also ensures that an accelerator cannot affect other pro-

	Accelerator Events			XG Requests	XG Responses			
States	Load	Store	Replacement	Invalidate	DataM	DataE	DataS	WB Ack
M	hit	hit	issue PutM / B	send Dirty WB / I	-	-	-	-
E	hit	hit / M	issue PutE / B	send Clean WB / I	-	-	-	-
S	hit	issue GetM / B	issue PutS / B	send InvAck / I	-	-	-	-
I	issue GetS / B	issue GetM / B	-	send InvAck	-	-	-	-
B	*stall*	*stall*	*stall*	send InvAck	/ M	/ E	/ S	/ I

Table 1: Sample accelerator L1 cache implementing Crossing Guard's (XG) interface. Entries are of the form `action / next_state`, and - indicates an impossible transition.

cesses running on the CPU, no matter how pathologically bad the accelerator's behavior is; in particular, it protects against crashing or deadlocks. We focus on systems where the accelerator maintains a TLB to allow it to use physical addresses.

Crossing Guard provides translation between protocols: the accelerator caches can make requests to Crossing Guard with a standardized set of coherence messages, and the host can use the normal host protocol. To the host coherence protocol, Crossing Guard will appear as simply another coherent cache. Any complex interactions with the other caches in the system, such as counting acknowledgment messages or handling races, will be taken care of by Crossing Guard.

There is one instance of Crossing Guard per accelerator in the system. It is implemented by the host designer, who has full knowledge of the coherence protocol running on the host. Once implemented, the Crossing Guard hardware works with any accelerator making use of the interface; it only needs to be updated when the host protocol changes.

2.1 Accelerator Coherence Interface

We define a set of requests and responses that form the coherence interface. The messages chosen determine optimizations the accelerator coherence protocol can make. For example, if the only way for the accelerator to request a block is with a generic *Get* message, the accelerator will only be able to implement a VI protocol.

In our design, the accelerator can make a total of five requests to the host and receive one of four responses from the Crossing Guard interface; the host can make one request to the accelerator and receive one of three responses. Every request always results in exactly one response.

The accelerator may request a block from the host with *GetS* (shared and read-only) or *GetM* (exclusive and read-write). In return, it can receive *DataS* (shared+clean), *DataE* (exclusive+clean), or *DataM* (exclusive+modified). The accelerator may receive *DataE* or *DataM* on either a *GetS* or *GetM* request.

When the accelerator cache replaces a block, it must send an appropriate request: *PutM*, *PutE*, or *PutS*. The *PutM* and *PutE* messages carry data to avoid the complexity of a multi-phase commit. In response to any *Put* request, the cache will receive a *WritebackAck*. E and M are both owned states; when the host sends an *Invalidate* request, the accelerator must respond with *Clean Writeback* or *Dirty Writeback*, respectively. Otherwise, the accelerator must respond with an *InvAck* (Invalidation Acknowledgment).

As long as the accelerator cache follows these rules, the interface guarantees that normal coherence invariants will hold (e.g., single-writer / multiple-reader). These messages are sufficient to implement a fully coherent cache with similar performance characteristics to more complex traditional protocols such as AMD Hammer or Intel MESI(F), and which can communicate with a variety of host cache designs. Not all of these messages are necessary for integration with all host protocols; e.g., when connecting to a host protocol that allows silent eviction of blocks in S state, the Crossing Guard does not pass the *PutS* message to the host. However, the goal of the interface is to work with a wide variety of traditional coherence protocols.

To demonstrate the use of this interface, Table 1 shows an example transition matrix in the style of Sorin et al. [12] for an accelerator L1 cache. In this case, a single private L1 cache is connected to each Crossing Guard. This example cache implements the MESI stable states, and has only a single transient state, B (Busy). In comparison, the private L1 in our baseline implementation of an inclusive MESI host protocol has six transient states, some of which include extra information such as a dirty bit or counters for pending acks, and can receive four host requests and seven host responses. These extra states and messages are required to handle races and count pending acks. We require that the network between Crossing Guard and the accelerator be ordered, which avoids all races between the accelerator except between an accelerator *Put* and a host *Invalidate* request. All other races and complexity are hidden by Crossing Guard.

To demonstrate that the Crossing Guard interface allows a range of accelerator protocol designs, we also implement a hierarchical two-level accelerator cache design: private per-core L1s connected to a shared L2, which communicates with Crossing Guard. Blocks can be shared between accelerator L1s even if the block is in M state at the accelerator L2; the L2 coordinates sharing. The accelerator cache hierarchy and protocol are completely independent of the host design, and connect to the host using the same Crossing Guard hardware as the single-level protocol.

If an accelerator benefits more from simplicity than from being able to implement a full MESI protocol, the accelerator cache can reduce complexity by treating sev-

Guarantees to Host Protocol for Coherence Messages on Behalf of Accelerator

0. **Accelerator requests must respect page permissions.**
   (a) The accelerator cannot make any request for a block for which it has no page access permissions.
   (b) The accelerator cannot make an exclusive (write) request or respond with exclusive data for a block for which it does not have page write permission.
1. **Accelerator requests must be correct with respect to the host protocol.**
   (a) The accelerator cannot make a request inconsistent with the *stable* state of the block at the accelerator.
   (b) The accelerator cannot make a request inconsistent with the *transient* state of the block at the accelerator.
2. **Accelerator responses must be correct with respect to the host protocol.**
   (a) The accelerator cannot make a response inconsistent with the *stable* state of the block at the accelerator.
   (b) The accelerator cannot make a response inconsistent with the *transient* state of the block at the accelerator.
   (c) The accelerator must respond to any request from the host within a reasonable time.

Figure 1: List of guarantees provided to the host system by Crossing Guard.

eral messages identically. For example, an accelerator cache can implement a VI design by sending only *GetM* requests. An MSI design is possible by treating *DataE* as *DataM* (and sending only *Dirty Writeback*s).

Finally, although we focus on fully coherent caches, it is also possible to implement an accelerator protocol with weaker coherence guarantees than the host system. For example, an accelerator may have multiple private L1s and a shared L2, and a programming model that requires an explicit flush before data from one core is guaranteed visible at other accelerator L1s. Crossing Guard places no restrictions on coherence behavior within the accelerator protocol.

2.2 Crossing Guard Guarantees to Host

Crossing Guard ensures that coherence messages to the host protocol on behalf of the accelerator are consistent with three guarantees, shown in Figure 1. We elaborate on these guarantees and how they are enforced below.

If the accelerator message (or lack of message) to Crossing Guard would violate one of these guarantees, Crossing Guard does not forward the message to the host (or will send an appropriate message on a timeout), and reports an error to the OS. The OS can then use an appropriate policy to handle the error; for example, it can terminate the process running on the accelerator, disable the accelerator to prevent it from making further accesses, and/or not schedule further processes on the accelerator and alert the user. In practice, we believe that the simpler coherence interface associated with Crossing Guard will limit the number of accelerator protocol errors.

At a high level, Crossing Guard enforces these guarantees as follows:

Guarantee 0: This guarantee ensures that an accelerator cannot read information to which it does not have read permissions or corrupt data for which it does not have write permissions. Crossing Guard enforces it by checking page permissions as described in prior work such as Border Control [18], and blocking incorrect accesses.

Guarantee 1: This guarantee protects the host from receiving unexpected requests from the accelerator, such as a *PutM* when the accelerator does not own the block, or a second *GetS* request for the same block before the first receives a response.

For **1a**, Crossing Guard tracks the state relative to the host of all blocks in the accelerator cache. If the request type is incorrect given this state (e.g., writing back a block that it does not have), Crossing Guard will block the request and report an error to the OS.

For **1b**, Crossing Guard tracks all pending accelerator coherence requests. If there is already a pending accelerator request for an address, it will block all subsequent accelerator requests to the same address and report an error.

Guarantee 2: This guarantee protects the host from receiving unexpected responses from the accelerator, such as an *InvAck* without a corresponding request; it also ensures that the host always receives an appropriate response within some constraint. Crossing Guard ensures that the response *message type* is correct, but cannot ensure correctness of *data* in the presence of an accelerator protocol error – the accelerator caches are malfunctioning, so data they provide is not trustworthy either. Thus, Crossing Guard may thus send stale or zero data in response to a host request, but will always alert the OS so that it can take proper action.

For **2a**, Crossing Guard again checks the response type against its record of the state of the block at the accelerator. If the response type is incorrect based on this information, it corrects the response type and reports an error. For example, if the accelerator owns a block but responds to an *Invalidate* with an *InvAck*, Crossing Guard will send a *Writeback* of a zero block instead.

For **2b**, Crossing Guard keeps state indicating whether there is a pending request from the host for the block. If not, it blocks the response and reports an error.

For **2c**, Crossing Guard detects whether the accelerator responds to a request within a timeout interval. If the accelerator does not respond to the request within a timeout, Crossing Guard will report an error and respond on its behalf with an appropriate message.

To enforce these guarantees, Crossing Guard must assume that an incorrect accelerator might send any message at any time, and respond accordingly.

2.2.1 Guarantees Not Provided by Crossing Guard

Crossing Guard does not and cannot protect against an accelerator writing incorrect data to a block for which the accelerator has write permission. This is because Crossing Guard does not have access to internal accelerator logic. It is the responsibility of the accelerator designer to ensure that the accelerator operates correctly. The goal of Crossing Guard is simply to protect the host system from incorrect accelerator coherence messages and provide host stability.

2.3 Types of Crossing Guard

There are several possible implementations of the Crossing Guard hardware. They provide the same interface to the accelerator, but have different benefits depending on specifics of the host protocol and the importance of minimizing the storage needed by Crossing Guard. We discuss two types of Crossing Guard here, and evaluate their performance and implementation tradeoffs in Section 4. First, the *Full State Crossing Guard* places few restrictions on the characteristics of the host protocol, allowing it to be used with no modifications to the host protocol. However, it needs to store information about every block currently held by the accelerator. Second, the *Transactional Crossing Guard* alternative only tracks open coherence transactions: cases where the host or the accelerator has made a request but not yet received all responses. It has much lower storage requirements, but requires that the host coherence protocol have certain properties. We find that existing coherence protocols (such as an AMD Hammer-like protocol and an inclusive MESI protocol) require only minor changes to work with Transactional Crossing Guard.

2.3.1 Full State Crossing Guard

Full State Crossing Guard tracks the state of every block present at the accelerator. In effect, this design adds a trusted inclusive directory between the host protocol and the accelerator. Because the accelerator sends *PutS* requests, Crossing Guard can store exactly the blocks in the accelerator caches.

Because this directory contains a trusted copy of all data and metadata for each block, it allows Crossing Guard to work with a wide variety of host coherence protocols. Whenever Crossing Guard enters some stable state for a block, it can forward it on to the accelerator, no matter the details of how the block was obtained (e.g., via a data message + acks or by gathering tokens). When Crossing Guard loses access to a block (e.g., due to an invalidation or forward request or timeout), it can notify the accelerator to invalidate its copy. Thus, Full State Crossing Guard works with a variety of unchanged host coherence protocols.

Full State Crossing Guard must store all data and metadata that could lead to a host coherence error if it were improperly modified. At minimum, it must store tags for all blocks currently held by the accelerator. For a 256kB accelerator cache with 64B blocks, this storage is around 16kB. In addition, it would need storage for blocks in MSHRs or buffers.

Crossing Guard may also need to store data for some blocks. For example, if the accelerator is allowed to own a block to which it does not have Read-Write permission, Crossing Guard must store a copy of the data to avoid violating **Guarantee 0b**. This can happen for host protocols which respond to a *GetS* request with *DataE* or *DataM* if no other cache has the block. This allows silent upgrades to M state later, and optimizes for a common case of blocks that are read, then written. However, the accelerator may later be expected to provide data to other caches for this block – which is prohibited for a read-only block by **Guarantee 0b** because it would allow an incorrect accelerator to modify a block for which it does not have write permission.. Furthermore, the performance optimization is not helpful because the block will not be written by the accelerator. In practice, many protocols provide a means of making non-upgradable *GetS* requests for blocks that are expected to be read-only, such as instructions.

Due to the storage overhead for Full State Crossing Guard, it may be most appropriate for accelerators with small caches or where the host protocol maintains metadata with all blocks.

2.3.2 Transactional Crossing Guard

Transactional Crossing Guard only tracks open coherence transactions, greatly reducing the storage required. It may also ease time-sharing of the Crossing Guard hardware between accelerators, because storage will not need to be sized for a specific accelerator.

Most of Crossing Guard's guarantees to the host can be easily enforced without Crossing Guard knowing the state of the block at the accelerator. **Guarantee 0** requires only knowledge of each coherence request/response type and address, along with the corresponding page permissions. **Guarantees 1b & 2b** require only knowledge about transient states to determine allowed message type(s), while **Guarantee 2c** requires only knowing how long a block has been in a transient state at the Crossing Guard.

However, **Guarantees 1a & 2a** require Crossing Guard to determine whether a message is consistent with the state of the block at the accelerator. Full State Crossing Guard handles them by storing information on the stable state of blocks held by the accelerator.

An alternate approach, which we use for Transactional Crossing Guard, is to instead require that the host protocol have certain properties that make *any* message that is consistent with the transient accelerator state also consistent with stable accelerator state. We require that the host maintain normal coherence invariants for any accelerator message reflecting the true state of the block at the accelerator. For all other messages consistent with the transient accelerator state, we require only that the host protocol *tolerate* the message. For example, if the accelerator holds a block in S but sends a *PutM* request, the host may handle this by discarding the data or by updating its copy of the block. If it updates the block, this may result in multiple values of the block in the host coherence system, in violation of normal coherence invariants. However, as the accelerator is already behaving incorrectly, the data is not trustworthy even if the message is discarded. We focus instead on ensuring that the host caches do not experience deadlock or livelock or enter undefined states. We simply require that the host system eventually converge on a single value for each physical address in the system.

For **Guarantee 1a**, we require that the host protocol be able to handle any request from an accelerator for a block, regardless of "true" stable state of that block at the accelerator, as long as the request does not violate any other guarantee. For example, the host protocol must be able to to tolerate a *Put* message for a block the accelerator does not hold, or a *PutM* message for a block in S. In some cases, the directory maintains owner information, which allows the host to determine if a *Put* is erroneous and report an error (and recover). Similarly, a protocol that uses *PutS* is likely to do exact sharer tracking, allowing it to determine if a *PutS* message is correct.

In some protocols, non-owner *PutM* requests can occur due to a non-erroneous race condition [12]. However, if the host protocol generates *Nack*s or takes other action on the *Put* (besides simply sinking it), it is necessary to ensure that these actions do not cause harm. For example, if an improper accelerator *Put* can result in a spurious *Nack* later being sent to a CPU cache, it is necessary that the CPU cache can detect this case and safely handle the message.

For **Guarantee 2a**, we require that the host protocol be able to tolerate any response from an accelerator that is appropriate to the originating request, regardless of stable state at the accelerator. As an example, consider a protocol that does not track sharers and thus broadcasts a *Forward* message to all caches, whether or not they have the block. The correct response thus depends on the state of the block at the accelerator: a *Data* message if the accelerator is the owner, otherwise an *InvAck*. If the accelerator responds incorrectly, Crossing Guard will forward the incorrect response to the host. This can result in problems in the accelerator host protocol, where a cache might receive zero or multiple copies of data instead of the exactly one copy it expects.

One way for a host protocol to maintain safety is to ensure that there is a one-to-one correspondence between requests and responses in the host protocol. The host can exactly track owner and sharer information for each block, and send different type of *Forward* messages based on this state. An alternate approach that requires less storage is to modify the host protocol to handle the different responses interchangeably; rather than counting acks, the requestor can count the number of responses received. Then the protocol may receive zero or multiple copies of data for the block, resulting in corrupted *data*, but without disrupting the host protocol.

Finally, Transactional Crossing Guard is inappropriate for protocols that require non-modifiable metadata to be stored along with accelerator stable states. In general, any data or metadata given to the accelerator in a stable state must be considered unsafe and potentially modified; this means that, for example, a Token protocol where each block is associated with virtual tokens, would require extensive changes to work with Transactional Crossing Guard.

In practice, we find that our baseline host protocols require only minor changes to work correctly (Section 3.2). Since both Crossing Guard and the host coherence protocol are implemented in the host, it may be reasonable to optimize Crossing Guard storage by making host protocol changes, especially if the changes do not require extensive redesign. The trade-offs between Full State Crossing Guard and Transactional Crossing Guard may weigh differently for different host systems. Since both use the same interface with the accelerator, the decision can be made independently of any accelerator design decisions.

2.4 Comparison with Traditional Protocols

There are several differences between our interface and existing host protocols like our baselines, an AMD Hammer-like exclusive MOESI protocol and Intel-like inclusive MESI. First, Crossing Guard prevents direct communication between an accelerator cache and other sibling caches in the system. In contrast, many protocols allow a cache to forward data to other caches at the same level. Crossing Guard's interface allows the accelerator cache to communicate only with a single controller (Crossing Guard), which in turn communicates with the other caches in the system.

This design choice has benefits and drawbacks. The principle drawbacks of disallowing cache-to-cache communication are that some transitions will require more hops, and there will be more traffic through the directory. The main benefit is avoidance of the complexity associated with sending data or acknowledgements between sibling caches. In the Hammer-like protocol, a request for a block will frequently result in a response from every other cache; in the MESI inclusive protocol, the requestor is told by the L2 how many responses it should receive. In both cases, the requestor must count acks, determine the type (e.g., *DataS*, *DataE*, or *DataM*) of data received, and determine whether to send an unblock message to the parent. This adds complexity and thus implementation, debugging, and verification difficulty. We shift this complexity to Crossing Guard, which only needs to be designed once per host protocol. In return, the accelerator protocol can be drastically simplified.

A second difference is Crossing Guard filters out race conditions. In some traditional protocols, if a cache requests a block in S, it may receive an invalidation before the data, in which case it must go into a transient state such as ISI [12], since the cache does not know whether the data or the invalidation was sent first. Handling race conditions correctly is difficult and adds complexity. Crossing Guard hides these races from the accelerator cache. The accelerator designer then does not need to consider them; if the block is not in a stable state, the accelerator cache should always return an *InvAck* on an *Invalidate* request and take no further action.

Third, the protocol does not constrain cache design in terms of inclusivity / non-inclusivity / exclusivity or number of levels. In the two-level accelerator protocol, the accelerators can use an entirely different cache organization than the host. Our accelerator L2 is inclusive and shared between a number of L1s, all running the same application, which allows data to be transferred between the L1s without going through Crossing Guard and the host directory. It can be integrated with the host protocol, regardless of host inclusivity and the number of levels in the host protocol.

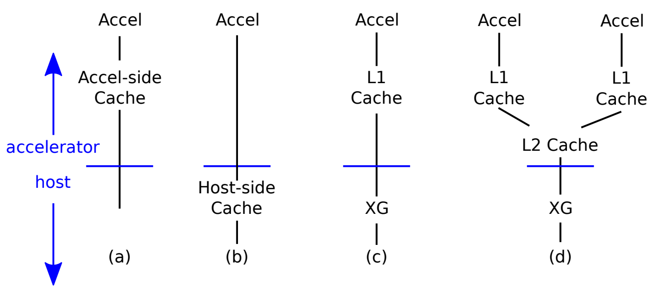

Figure 2: Cache organization options for accelerators.
(a) Accelerator-side cache (unsafe), using the host protocol.
(b) Host-side cache, with no cache at the accelerator.
(c) Accelerator with Crossing Guard, with a single-level accelerator cache using an accelerator protocol.
(d) Multicore accelerator with Crossing Guard, with two-level cache using an accelerator protocol.

2.5 Other Benefits of Crossing Guard

Besides the safety guarantees already discussed, Crossing Guard can protect against denial of service attacks where a misbehaving accelerator sends legitimate messages at a very high rate, consuming bandwidth, directory entries, or other resources shared with the host. Unchecked, this could cause significant performance degradation for processes being run on the host system. To avoid this situation, Crossing Guard can limit the rate at which an accelerator can send requests (responses should always be sent immediately). This rate limiting could be controlled by registers set by OS policies. Using a configurable limit would also allow correct accelerators to be throttled when they are impacting performance of other components of the system, while allowing them more resources when available.

Crossing Guard, along with translating between coherence protocols, may also translate between coherence block sizes. If the accelerator uses a larger block size than the host, Crossing Guard can merge requests and responses. On an accelerator request, it can request all needed host blocks, and once they arrive, it can forward the merged block to the accelerator. On a writeback, it can split the single accelerator block back into component blocks. (However, if the application developer does not realize that the block sizes are different, performance could be very bad due to an increase in expensive false sharing.)

If the accelerator uses a smaller block size than the host, Crossing Guard can request blocks at host granularity and store them, forwarding only pieces of them to the accelerator. Although this allows for correct operation, it increases necessary storage at the Crossing Guard. We believe that it is unlikely that an accelerator would use smaller granularity blocks than the host, since host block size is already relatively small (64kB).

3. CROSSING GUARD IMPLEMENTATION

To better understand how our proposed interface works with various host protocols, we design Crossing Guard hardware to connect our example one- and two-level accelerator protocols with two baseline host protocols: an AMD Hammer-like exclusive protocol and an inclusive MESI protocol. For our baseline protocols, we use the versions implemented in gem5 as the protocol description. We integrate Crossing Guard with preexisting protocol implementations to avoid making design choices in the host protocols that might inadvertently make it easier for us to integrate Crossing Guard.

Crossing Guard appears to the host protocol as an ordinary cache: for Hammer-like as a private L2, and for MESI inclusive as a private L1. Because the accelerator caches are behind Crossing Guard, the host protocol is completely agnostic to their design.

We show some of the design alternatives in Figure 2. First, in (a), the accelerator may implement a cache using the host protocol; this is unsafe and requires the host protocol to be exposed to the accelerator, but may be suitable if the accelerator is designed by the same company as the host. Second, in (b), the host provides a host-side cache to which the accelerator directly makes load and store requests by virtual address. While safe, this may increase access latency as well as disallowing the accelerator freedom to customize its cache design. Finally, in (c) and (d), the host exports the Crossing Guard interface, while the accelerator implements an L1 cache customized to its needs (c) or an L1 and shared L2 using its own protocol (d). In this case, Crossing Guard provides safety and an interface with the host, while letting the accelerator use customized caches.

In total, we evaluate 8 possible configurations using Crossing Guard: 2 host protocols, 2 Crossing Guard variants, and 2 accelerator protocols. In addition, we evaluate 4 configurations without Crossing Guard: an accelerator-side and a host-side cache for each host protocol.

3.1 Permission Information

Crossing Guard ensures that page access permissions are respected (**Guarantee 0**), as in prior works [18]. For each accelerator or host request where there is not already a pending transaction, Crossing Guard first obtains the page permissions (Read-Write, Read, or None), as in Border Control. It then stores the page permission information with any buffer entries or cache lines for the address. These permissions are used to provide protection; e.g., if a block has read-only permission, the accelerator should not be able to send any messages for that block containing data.

3.2 Integration with Host Protocols

We integrate Full State and Transactional Crossing Guard with two existing coherence protocols in gem5-gpu [19], with the general-purpose GPU as a proxy for a general high-performing accelerator. We discuss our experiences with each and the necessary host protocol modifications.

Although a different Crossing Guard implementation is needed for each host protocol, there were significant similarities. We discuss the general operation of Crossing Guard, then address host protocol-specific considerations.

First, the accelerator may make a *Get* request. If the request violates its guarantees, Crossing Guard reports an error and does not forward it to the host. Otherwise, Crossing Guard forwards the request and then waits for data and/or metadata such as acks from the host system. When Crossing Guard determines the request has been satisfied (i.e., it has received a copy of the data and the requisite number of acks), it passes the data along to the accelerator with the appropriate message (*DataS*, *DataE*, *DataM*). If necessary, it also sends the appropriate unblock message to the host directory.

Second, the accelerator may make a *Put* request. If the request violates the Crossing Guard guarantees, Crossing Guard again reports an error. Otherwise, Crossing Guard forwards the request and acks the accelerator, then waits for a writeback ack from the host. In a two-part writeback as in the Hammer-like protocol, Crossing Guard then sends the data.

Third, the host may make an *Invalidate* or *Forward* request. If Full State Crossing Guard shows that the accelerator does not have the block in a conflicting state (e.g., block is in S on a forwarded *GetS*), or if Transactional Crossing Guard can deduce block state from open transactions or page permissions, Crossing Guard can simply send an *Ack* on the accelerator's behalf. Otherwise, Crossing Guard forwards an *Invalidate* to the accelerator, then waits for its response so it can translate it and send it to the host (i.e., directory, sibling cache, or both) with appropriate message type and number of acks, etc. Crossing Guard also enforces a timeout for these transactions, so that if the accelerator takes too long to reply, Crossing Guard can recover by sending a zero data block or an ack to the requestor and alerting the OS that an error has occurred.

Even though Transactional Crossing Guard does not store full accelerator cache state information, it can respond to snoops for blocks where the accelerator has no permissions without consulting the accelerator. This has two benefits: first, it avoids needless messages and associated latency. Second, if the accelerator is malicious, coherence traffic could be a side-channel; preventing the accelerator from observing coherence traffic for blocks it cannot access closes the channel.

Finally, there may be a race between an accelerator and host request, resulting in a transaction that is waiting for both the host and the accelerator. For example, the accelerator may make a *Put* request at the same time as the host makes an *Invalidate* request for the same block. In this case, Crossing Guard must ensure that both the host and accelerator receive appropriate responses.

3.2.1 Integration with Hammer-Like Protocol

The `MOESI_hammer` protocol, as implemented in gem5, combines the private L1I, L1D, and L2 into a single cache controller, with a directory in a separate controller. We integrate Crossing Guard to act like a private L1/L2 cache. We discuss some of the nuances of implementing Crossing Guard for this protocol.

The Hammer-like protocol includes a non-exclusive owned state (O), which gives the owning cache a shared copy of the block but requires it to respond to requests from other caches. The Crossing Guard accelerator interface does not have a way of communicating this state to the accelerator with the messages in its interface. Therefore, Transactional Crossing Guard ensures that the host never considers the accelerator in this state. The O state is reached when there are multiple *GetS* requests from different cores in the Hammer-like protocol to a block in an exclusive owned state (M). The directory then forwards a *merged GetS* request to the owner. The expected response is for the owner to downgrade to the non-exclusive owned state O and also provide the data to the requestors. We handle this case by having Crossing Guard request an invalidation from the accelerator cache, then forward the data in the resulting writeback to the requestors. Crossing Guard also sends the data to the directory with a *Put* message, relinquishing ownership.

Three host protocol changes are required to support Transactional Crossing Guard. First, the Hammer-like protocol we build on does not include a non-upgradable *GetS* message; a cache in modified M state always responds with exclusive data. The accelerator cannot own a block with read-only permission without violating **Guarantee 0b**, so we add a non-upgradable *GetS_only* request. Similarly, we add a *Fwd_GetS_only* message to the host L1/L2. We believe that commercially implemented protocols generally already implement such a request for use by instruction caches.

Second, to ensure that any request from the accelerator is valid regardless of stable block state (**Guarantee 1a**), the protocol must be modified to handle unexpected accelerator *Put* requests. The baseline directory already handles *Put* requests when it believes that the directory already owns the block, because this can happen in a correctly implemented protocol if the *Put* request races with a *Get* from another cache. The directory then responds with a *Nack*. However, if the accelerator sends an incorrect *Put*, it can lead to a host cache receiving an unexpected *Nack* at a later time. We thus modify the host L1/L2 caches to sink unexpected *Nack*s and generate an error.

Third, to ensure that any response corresponding to a request from the host is always valid regardless of accelerator stable state (**Guarantee 2a**), we must modify the host L1/L2 logic for counting *Ack*s on a *Get* request. Rather than counting *Ack*s, the L1/L2 instead counts the number of responses. This allows it to receive zero or multiple copies of the data without a protocol disruption.

These changes require no additional states; in total, we added one new event to the directory and one to the L1/L2.

3.2.2 Integration with MESI Two-Level

The `MESI_Two_Level` protocol has a shared L2 that is inclusive of private L1s (where the L1I and L1D share a controller). Implementing Full State Crossing Guard is straightforward. The only change to the baseline protocol required for Transactional Crossing Guard is treating *Ack*s and *Data* as equivalent responses to a *Forward* request (**Guarantee 2a**). This is mostly simple, except in the case where Crossing Guard receives an invalidation due to a *GetM* request from a host L1 and the accelerator is not the owner. If the accelerator is buggy and responds with a *Writeback* rather than an *InvAck*, Crossing Guard will forward the data to the L2 rather than acking the requesting L1. Therefore, it is necessary for the L2 to respond to this unexpected event by acking the requestor on behalf of the accelerator.

The host protocol can handle requests from the accelerator at any time (**Guarantee 1a**) with no changes. Thus, the MESI inclusive protocol works with Transactional Crossing Guard without additional states or events in the host controllers.

4. RESULTS

To explore the effects of accelerator cache organization and the performance and error tolerance properties of Crossing Guard, we evaluate a range of cache organizations in gem5-gpu [19]. We evaluate Crossing Guard for three properties: correctness given a correct accelerator cache; safety with an incorrect accelerator cache; and performance.

4.1 Protocol Stress Test

We first evaluate the correctness of our implementation of the accelerator protocols and the interface with the host protocols. We stress-test the protocols by running a random testing framework which makes rapid loads and stores to random addresses and checks correctness of the data [20]. To increase contention, only a small number of addresses are used, and the cache sizes are correspondingly decreased so that replacements are frequent; message latencies are chosen randomly to model the case where messages are delayed in the network. We expect that if the tester is run with varying random seeds and cache sizes, all possible transitions will eventually be seen. We assume for this test that all blocks have full read-write permission at the accelerator.

We choose this approach over formal verification (e.g., model checking with Murϕ, [21]) due to practical limitations of existing verification methods. These approaches typically can verify a protocol for only a single address in the system, and rely on symmetry between cores that our heterogeneous system lacks. These methods require making simplifications for tractability and do not model all aspects of the protocol. An industrial implementation of Crossing Guard would likely include formal verification to complement stress testing.

We ran the random tester for at least 240 million load/check pairs per configuration, with up to 82 billion for configurations that did not have complete coverage. Across configurations, the tests ran for a sum of 22 compute years. To understand the coverage of the tester, we counted the state/event pairs that the random tester visited at each cache controller and compared it with the number that we believe are possible, based on transitions defined by the baseline protocols. In the case of a missing transition or coherence bug that caused incorrect data, the tester would produce an error; we did not observe any errors or missing transitions in the final versions of the protocols. We manually inspected transitions that were never visited and determined that some of them were not actually reachable (e.g., *Replacement* event for a block not in the cache); we therefore removed them from the total. We show the totals in Table 2. These coverage totals do not include transitions only reachable due to accelerator misbehavior.

The higher the percentage of transitions visited, the more likely it is that the tester would have uncovered any latent bugs. Although the percentage of visited transitions at each controller does not directly indicate the percentage of visited *global* transitions, it provides a lower bound on coverage.

We observed that over 99% of transitions were reached for each configuration for both host protocols. The remaining transitions tended to be reachable only by multiple races between host invalidations and accelerator replacements, making them very infrequent.

4.2 Accelerator Protocol Fuzz Testing

To test whether the host is protected from the accelerator when the accelerator caches are incorrect, we employed fuzz testing [22], where we generated random input to Crossing Guard from the accelerator side. The fuzz tester takes the place of the accelerator cache(s). It repeatedly randomly selects one message type and an address, and makes a request to the host (via Crossing

Protocol		**Full State XG**			**Transactional XG**		
Host	**Accel**	**Visited**	**Possible**	**%**	**Visited**	**Possible**	**%**
Hammer-like	**1-level**	60+148+70+19	60+148+73+19	**99%**	60+148+67+20	60+148+69+20	**99.3%**
Hammer-like	**2-level**	60+148+92+85+18	60+148+92+85+19	**99.8%**	60+148+68+87+18	60+148+69+87+18	**99.5%**
MESI	**1-level**	60+121+45+19	60+121+45+19	**100%**	60+121+43+20	60+121+43+20	**100%**
MESI	**2-level**	60+121+63+85+18	60+121+64+85+19	**99.4%**	60+121+43+87+18	60+121+43+87+19	**99.7%**

Table 2: Coverage with random tester. Hammer-like totals are for Directory+Host L1/L2 Cache+Crossing Guard+Accelerator L1 (+ Accelerator L2). MESI totals are for Host L2+Host L1+Crossing Guard+Accelerator L1 (+ Accelerator L2).

Host Protocol	**Full State XG**			**Transactional XG**		
	Visited	**Possible**	**%**	**Visited**	**Possible**	**%**
Hammer-like	148+57+211	148+60+211	**99.3%**	172+61+154	172+61+153	**99.7%**
MESI	60+121+148	60+121+149	**99.7%**	60+127+93	60+127+93	**100%**

Table 3: Coverage with fuzz tester. Hammer-like totals are for Directory+Host L1/L2 Cache+Crossing Guard. MESI totals are for Host L2+Host L1+Crossing Guard.

Guard), without considering the cache state of the address. This generates a stream of requests unlike what would be seen in a correctly functioning cache: for example, multiple *GetM* messages without receiving or relinquishing data, *PutM* messages when not the owner, and unsolicited *Writebacks* or *InvAcks*. It also can lead to missing responses, normally resulting in deadlock. The host side continues to run the previous random stress tester, but the check for data correctness is removed.

Table 3 shows that coverage with the fuzz tester was over 99%. This method quickly uncovered numerous errors in our initial implementation; however, in the final implementation we find no invalid transitions or deadlocks. These results suggest that Full State Crossing Guard fully protects an unmodified host protocol. In addition, Transactional Crossing Guard can, in concert with a slightly modified host protocol, hide accelerator misbehavior from the rest of the system.

4.3 Performance

Although safety is Crossing Guard's primary goal, our protection mechanism must not significantly degrade performance. We therefore evaluate performance on one type of accelerator, a general-purpose GPU. GPGPUs are complex, high-performance accelerators that can benefit from sharing a coherent address space with the host system.

Tables 4 and 5 give details of our simulator configuration. We evaluate our approach using shared-memory versions of workloads from the Rodinia benchmark suite [23]. For obtaining page permissions, we implement Border Control [18], which uses a table updated on accelerator TLB misses to determine permissions; we assume a 2 cycle latency to access an 8kB Border Control Cache. We hold the total system cache capacity constant across all configurations; thus, in the two-level accelerator cache design, the L1s will be smaller than in the single-level accelerator cache, as shown in Table 5.

We compare Crossing Guard with two alternate approaches. First, the *accelerator-side cache* gives each accelerator a unified L1 similar to the CPU caches. This

CPU		
CPU Cores	1	
CPU Frequency	3 GHz	
Host Caches		
	Hammer-like	**MESI Inclusive**
L1I & L1D	32kB each	32kB each
L2	128kB private	512kB shared w/ GPGPU

Table 4: CPU simulation configuration details.

cache uses the host protocol and is unsafe, but may be appropriate when the accelerator is designed by a trusted party. Second, the *host-side cache* provides the accelerator with an interface to which it can make loads and stores by virtual address. However, the cache has long latency on hits. The host-side cache is similar to CAPI's approach, but without allowing a trusted cache on the accelerator side.

We assume that the accelerator is not as closely integrated with the host L2/Directory as the rest of the caches, but rather has 20x the latency. High latency might result if the accelerator and host directory are in different clock domains. In addition, the high latency is consistent with performance results for integrated GPGPUs. Finally, the GPGPU and its workloads are fairly latency-insensitive and the high latency link emphasizes the difference in configurations, which might be more apparent for latency-sensitive accelerators. We place the host-side cache and Crossing Guard on the host side of the connection, and any accelerator caches at the other side.

Figure 3 shows performance for the Hammer-like case and Figure 4 for the inclusive MESI case, each normalized to the accelerator running an unchanged host protocol.

For the Hammer-like exclusive host protocol, we find that allowing the accelerator a cache is important for performance. The host-side cache degrades performance by an average of 27% compared to the unsafe accelerator-side cache, due to the long latency to access the host-side cache. Interestingly, in some cases the host-side cache performs better than the accelerator cache; this is because the Hammer-like protocol does not track shar-

GPGPU			
Cores			4
GPU Frequency			700 MHz
GPGPU Caches (Hammer-like)			
	Accel-side	**Host-side / 1-level**	**2-level**
L1	16kB I and D	160kB	32kB
L2	128kB private	-	512kB shared
GPGPU Caches (MESI Inclusive)			
	Accel-side	**Host-side / 1-level**	**2-Level**
L1	32kB I and D	64kB	16kB
L2	-	-	192kB shared

Cache-to-Cache Latency	
Accelerator L1 to L2	10 cycles
Accelerator L2 to XG	200 cycles
XG to Directory/Shared L2	10 cycles
Accelerator to Host-side Cache	210 cycles

Table 5: GPGPU simulation configuration details.

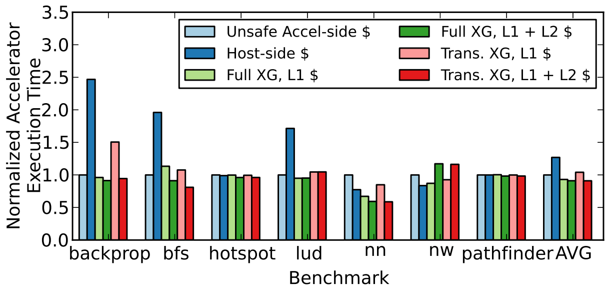

Figure 3: Performance with Hammer-like host protocol, normalized to an unsafe accelerator-side cache.

ers, and *Forward* requests thus must traverse the long-latency hop to the accelerator. This is particularly harmful when the accelerator does not have the block, and accounts for much of the slowdown of the Transactional Crossing Guard. The effect is stronger for the single-level accelerator cache, because the two-level cache allows accelerators to share blocks without traversing the link to the host. The combination of Crossing Guard and an accelerator cache provides the fast cache access of the accelerator-side cache plus the fast *Forwards* of the host-side cache, with performance results compared to the accelerator-side cache ranging from a 9% average improvement to a 4% overhead depending on configuration.

For the MESI inclusive host protocol, we find that the host-side cache has an average performance overhead of 36% compared to the unsafe accelerator-side cache. The two-level accelerator cache with Crossing Guard suffers from a combination of limited capacity and expensive demand misses. However, the single-level accelerator cache performs indistinguishably to the unsafe case while providing safety.

Together, these results suggest that Crossing Guard can provide safety to host while maintaining high performance.

5. RELATED WORK

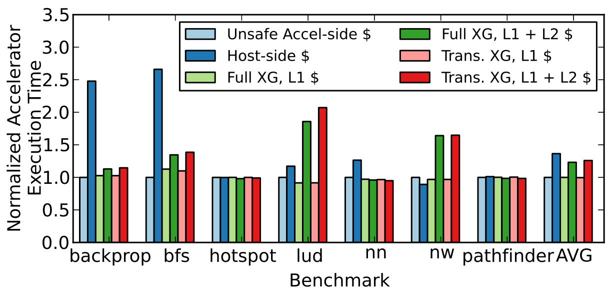

Figure 4: Performance with MESI inclusive host protocol, normalized to an unsafe accelerator-side cache.

Crossing Guard targets coherent accelerators that are high-performance, potentially buggy, and possibly designed by third parties. For accelerators not falling into these categories, other approaches may be viable. We discuss several alternatives to Crossing Guard which trade off host modification, safety, and performance, and summarize them in Table 6.

First, the accelerator could directly use the unchanged host protocol. This requires the host protocol to be known to the accelerator designer, and for the host designer to trust or verify that the accelerator is correctly implemented. The benefit of this approach is that there is no need to add hardware or modify existing cache coherence protocols, and performance will be good. It may be suitable for accelerators designed by the same company as the host system. There is considerable work on testing and verifying [24, 25, 26, 21, 27] cache coherence of systems without accelerators, some of which may be applied to heterogeneous systems.

Second, several companies have introduced interfaces for coherent memory accesses for third-party accelerators. IBM's Coherent Accelerator Processor Interface (CAPI) [14] allows accelerators to make coherent load and store requests to host memory, but precludes accelerator designers from implementing high-performance customized cache hierarchies. This model is well-suited for FPGAs, but may not be for complex, high-performance accelerators that benefit from customized and optimized accelerator cache organizations.

The ARM Accelerator Coherency Port (ACP) [16] also allows an accelerator to make coherent accesses to the host: an ACP access will return the most up-to-date copy of the block, even if it resides in a CPU cache. However, it does not provide full coherence: a request from the host will not return data modified by the accelerator until the accelerator has flushed its caches. This may be a reasonable approach for some classes of accelerators; in particular, those using a bulk-synchronous model where caches are flushed at the end of kernel execution. However, it is less well-suited when the accelerator, workload, or programming model cannot determine when the accelerator is done with a block.

CAPI and ACP are limited to a single host; the CCIX Consortium [17] promises a standardized coherence in-

Approach	Accelerator-side	Host-side	Robust	Full State XG	Transactional XG
Avoid Host Changes	✓	✓	✗	✓	Minimal
Standardized for Accel.	✗	✓	✗	✓	✓
Safe	✗	✓	✓	✓	✓
High Performance	✓	✗	✗	✓	✓
Low Storage Overheads	✓	✗	✗	✗	✓
Allows Cache Customization	✓	✗	✓	✓	✓

Table 6: Summary of related approaches to accelerator-host memory interactions.

terface. Unfortunately, detailed information has yet to be released.

Third, if data access patterns are coarse-grained or known a priori, coherence may be unnecessary and the accelerator can instead use scratchpads or other software-managed storage. Even if coherence is desirable from a programmability or performance standpoint, knowledge of specific coherence patterns may allow the accelerator designer to emulate hardware coherence at the cost of lower performance [11].

Finally, host coherence protocols could be redesigned to tolerate all bad coherence messages from accelerators. This approach has the benefit of providing protection both against bugs in the host caches and the accelerators. However, it requires a complete redesign of host coherence protocols, which is expensive. In addition, prior work degrades performance or is limited to certain error conditions (e.g., missing messages but not incorrect messages) [28, 29]. This is because these robust coherence protocols generally address a different fault model: they guarantee correctness of *data* as well as correctness of *coherence messages*.

Prior work has considered incorrect accelerator behavior. Border Control [18] enforces virtual memory permissions for accelerators, even if they make memory accesses by physical addresses. However, it does not prevent against incorrect types of coherence messages.

There have been several proposals for hardware that converts between coherence protocols; Stanford FLASH [30] mentions the possibility, but does not explore its implementation. Hagersten et al. [31] propose a coherence transformer. To our knowledge, no previous work discusses translating between coherence protocols with protection against incorrect coherence behavior.

6. CONCLUSION

Some future hardware accelerators will benefit from hardware cache coherence. However, accelerator designers face complex and proprietary host protocols, while host designers may be reluctant to allow potentially buggy accelerator caches to participate in host coherence. Crossing Guard addresses both challenges by providing a safe and standardized coherence interface, while maintaining high performance.

7. REFERENCES

[1] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, D. Donohoe, A. Lupas, S. Mitchell, D. Nicholls, and V. Toma, "Myriad 2: Eye of the computational vision storm," in *Hot Chips 26*, 2014.

[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 *Proc. of the 45th Annual IEEE/ACM International Symp. on Microarchitecture*, pp. 449–460, Dec. 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 *Proc. of the 46th Annual IEEE/ACM International Symp. on Microarchitecture*, pp. 468–479, Dec. 2013.

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

[6] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson, "XML accelerator engine," in *In The First International Workshop on High Performance XML Processing*, ACM, 2004.

[7] K. Atasu, R. Polig, C. Hagleitner, and F. R. Reiss, "Hardware-accelerated regular expression matching for high-throughput text analytics," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pp. 1–7, Sept. 2013.

[8] V. Rajagopalan, "All programmable devices: Not just an FPGA anymore," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2013. Keynote presentation.

[9] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated cpu-gpu systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 457–467, ACM, 2013.

[10] S. Kumar, A. Shriraman, and N. Vedula, "Fusion : Design tradeoffs in coherent cache hierarchies for accelerators," in *Proc. of the 42nd Annual Intnl. Symp. on Computer Architecture*, June 2015.

[11] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *Proc. of the 22nd IEEE Symp. on High-Performance Computer Architecture*, Mar. 2016.

[12] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures in Computer Architecture, 2011.

[13] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pp. 348–355, Oct 2008.

[14] IBM, *Coherent Accelerator Processor Interface User's Manual*, 2014.

[15] 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.

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

[17] *Cache Coherent Interconnect for Accelerators (CCIX)*.

[18] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *Proc. of the 48th Annual IEEE/ACM International Symp. on Microarchitecture*, pp. 470–481, Dec. 2015.

[19] 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.

[20] D. A. Wood, G. A. Gibson, and R. H. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design and Test of Computers*, pp. 13–25, Aug. 1990.

[21] D. L. Dill, "The mur ϕ verification system," in *Computer Aided Verification*, pp. 390–393, Springer, 1996.

[22] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, pp. 32–44, Dec. 1990.

[23] 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 *Proceedings of the International Symposium on Workload Characterization*, pp. 44–54, October 2009.

[24] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness, "Verification of the Futurebus+ cache coherence protocol.," in *CHDL*, vol. 93, pp. 15–30, Citeseer, 1993.

[25] D. Abts, D. J. Lilja, and S. Scott, "So many states, so little time: Verifying memory coherence in the cray x1," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2003.

[26] S. Park and D. L. Dill, "Verification of FLASH cache coherence protocol by aggregation of distributed transactions," in *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures*, pp. 288–296, June 1996.

[27] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "Ccicheck: using μhb graphs to verify the coherence-consistency interface," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 26–37, ACM, 2015.

[28] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, pp. 123–134, IEEE, May 2002.

[29] R. Fernandez-Pascual, J. M. Garcia, M. E. Acacio, and J. Duato, "A low overhead fault tolerant coherence protocol for cmp architectures," in *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2007.

[30] 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 *Proc. of the 21st Annual Intnl. Symp. on Computer Architecture*, pp. 302–313, Apr. 1994.

[31] E. E. Hagersten, M. D. Hill, and D. A. Wood, "Methods and apparatus for a coherence transformer for connecting computer system coherence domains," Jan. 12 1999. US Patent 5,860,109.