Crossing Guard: Mediating Host-Accelerator Coherence Interactions

Lena E. Olson Mark D. Hill David A. Wood
University of Wisconsin-Madison
Department of Computer Sciences

Summary of paper

Hardware accelerators are currently of great interest to academia and industry, because specialization is a path forward that does not rely on the continuation of Moore's Law or Dennard scaling. As a result, accelerators have been proposed for a wide variety of applications; for example, accelerators are enabling new advances in machine learning.

Some of these accelerators may benefit from sharing a unified address space with the host system. For many years, CPUs have optimized fine-grained, data-dependent memory accesses through the use of hardware cache coherence. Hardware coherence aids in both performance and programmability by freeing the programmer from explicitly managing where data is located in memory and the cache hierarchy. Some accelerators, such as GPGPUs, already support some degree of cache coherent memory accesses.

However, accelerators are not CPUs, and CPU approaches to coherence are not a perfect fit for unified accelerator-CPU systems for several reasons. First, while CPUs are general-purpose compute devices, accelerators are by definition specialized. There may therefore be benefits to optimizing the accelerator's caches based on expected data access patterns. In addition, groups of accelerator cores may work together on similar data, as in GPGPUs or Fusion [4]; these accelerators may see efficiency and performance improvements with hierarchical caches. Any coherence approach should therefore allow customized accelerator caches.

Second, accelerators differ from CPUs in that they may be designed by third parties, and then integrated into the host SoC. Unfortunately, it is not straight-forward to drop a third-party device into a host coherence protocol. Traditional CPU coherence protocols are generally proprietary; they also vary by host system. Coherence protocols are notoriously complex and difficult to implement correctly; accelerator designers may not have the expertise or time to implement and test complex traditional host protocols. The burden of implementing a new cache controller for every host system will limit the ability to design interoperable accelerators. A coherence approach should therefore export a single, standard coherence interface to the accelerator, and it should be as simple as possible while still allowing customization and high performance.

Crossing Guard: Mediating Host-Accelerator Coherence Interactions, Lena E. Olson, Mark D. Hill, David A. Wood. Published in ASPLOS, April 2017.

Finally, even CPU coherence implementations have been known to have bugs; accelerator controllers implemented by third parties may be prone to a higher rate of errata. An erroneous message sent from the accelerator to the host – or an omitted response – can deadlock the host coherence protocol as well, rendering the host system inoperable. A malicious accelerator could potentially even snoop on coherence traffic in a side-channel attack. For this reason, host system designers will want to ensure that the coherence interface is able to tolerate receiving any coherence message from the accelerator at any time, without coherence errors visible to the CPU caches. It should also limit the coherence activity the accelerator is able to observe.

These requirements for an interface – allowing customized accelerator caches, standardized messages across systems, and providing safety for the host – essentially describe the software concept of an Application Program Interface (API). An API provides clients (i.e., accelerators) with a public interface that 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 limits the potential error cases. Our work, *Crossing Guard*, aims to provide such an interface.

Related Work

There are several current industry approaches to providing safe, interoperable accelerator coherence. Several of them (IBM CAPI [6], ARM ACP [2]), provide interfaces to accelerators; however, they do not allow building a customized cache hierarchy at the accelerator. In addition, they are limited in what host systems they support.

To provide interoperable coherence between host system vendors, the CCIX consortium [1] intends to provide an industry standard. This may fulfill many of our requirements; however, at this time, all details about the standard are only available to member companies. It is therefore unclear how it handles incorrect accelerator messages, or whether it allows hierarchical caches at the accelerator.

Crossing Guard

Crossing Guard provides a coherence API through host-side hardware that translates between the two coherence domains. It exports a small set of coherence requests and responses to the accelerator; full details are in the paper. This interface allows the accelerator to request and receive cache blocks in shared or exclusive state; to evict blocks back to the host; and to respond to host requests with data or acknowledgments. We

	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.

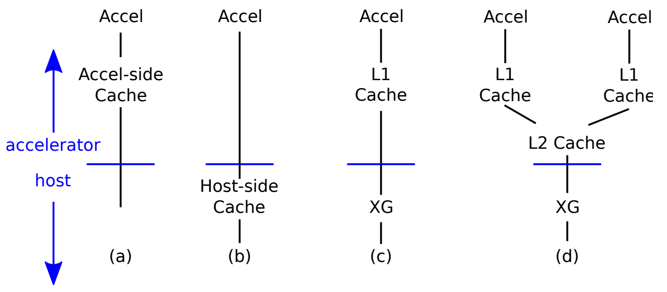

Figure 1: Cache organization options for accelerators.
(a) Accelerator-side cache (unsafe), using host protocol.
(b) Host-side cache, with no cache at the accelerator.
(c) Accelerator with Crossing Guard, with a single-level accelerator cache using an accelerator protocol.
(d) Multicore accelerator with Crossing Guard, with two-level cache using an accelerator protocol.

show that this set of operations is sufficient to enable both very simple and easily verified accelerator caches (as in Table 1), and complex, hierarchical caches. Once implemented, an accelerator cache using this interface can be connected to any host system with Crossing Guard hardware, regardless of host protocol implementation details.

From the host perspective, the Crossing Guard hardware appears similar to a CPU cache. We envision the host coherence designers implementing the Crossing Guard; they have the expertise and verification tools to ensure safety and correctness. The Crossing Guard hardware performs a translation between the accelerator and the host protocols. For example, in many CPU protocols, requesting a block can involve collecting acknowledgments from a number of CPU caches. Therefore, the Crossing Guard will gather these *Ack*s, and only send the data block to the accelerator once all have been collected. Similarly, when the accelerator sends a *Put* request, Crossing Guard will forward it to the host. It will also handle any further interactions for that block, such as sending *Unblock* messages or responding to racing host requests. Essentially, the accelerator sees a vastly simpler, more abstract, and less error-prone interaction than it would without Crossing Guard.

To protect the host from a malfunctioning accelerator, Crossing Guard stores information about the coherence state of blocks at the accelerator. We demonstrate two variants, differing in host protocol requirements and Crossing Guard storage needs. In the first variant, to work with any unmodified host protocol, Crossing Guard can keep an inclusive directory of tags for each block at the accelerator. For smaller storage, the second variant allows host protocols with certain characteristics to instead choose to keep only MSHRs for coherence transactions currently in flight. The accelerator developer need not know which type of Crossing Guard the host implements.

Crossing Guard uses this state both for translations, and also to check that messages from the accelerator are valid. For example, if the accelerator already has a block, it should not request it again. Similarly, if the accelerator is sent an invalidate request, it must respond with an *Ack* in a timely manner. If the accelerator malfunctions, Crossing Guard will block inappropriate messages, send appropriate ones, and generally recover the host protocol to a consistent state. It will also alert the OS, which can, e.g., terminate the process on the accelerator, and alert the user.

Results

We simulated several systems with Crossing Guard, to demonstrate its versatility (Figure 1). We selected two host protocols: an inclusive MESI protocol, and an exclusive MOESI protocol. To show that Crossing Guard's interface allows simple and easily understood accelerator protocols, we implemented the single level cache in Table 1. To show that it also allows complex and hierarchical accelerator protocols, we implemented a two-level protocol, with private per-accelerator L1s connected to a shared accelerator L2.

With Crossing Guard, the accelerator caches were drastically simpler to implement; the single-level accelerator cache had 5 states (20 transitions), while the AMD Hammer-like private cache had 24 states (128 transitions). We ran GPGPU workloads in gem5 with Ruby and confirmed that the simplicity of the accelerator caches did not come at a cost of reduced performance.

Because one of the main goals of Crossing Guard is to provide safety, even when the accelerator malfunctions, we extensively tested correctness with fuzz testing. To simulate a pathologically broken accelerator, we sent random requests and responses to the Crossing Guard.

Meanwhile, we issued load/store pairs from the host CPUs, to ensure that the host coherence system remained unaffected by deadlocks. Over billions of load/store pairs on the host, we verified that the host system tolerated the stream of incorrect messages injected by the accelerator. This suggests that Crossing Guard is able to provide safety to the host.

Case for Influence

The recent explosion in accelerators is changing the computing hardware landscape. Some accelerators, such as GPGPUs, are already widely deployed and are increasingly becoming first-class citizens in the memory system, with coherent access to (unified) host shared memory. Others, such as machine learning accelerators like the Tensor Processing Unit [3], are demonstrating new and exciting applications. For better or worse, accelerators are here.

Prior work on interfaces between accelerators and host memory has mostly focused on the I/O system. For example, the CCIX Consortium [1] recently announced a PCIe-based interface that shares many of the same goals as Crossing Guard, albeit with the higher latency and lower bandwidth of an I/O bus. CCIX specifies a simplified coherence interface, allowing third-party accelerator designs to be easily integrated across a range of host systems.

Crossing Guard goes beyond the current CCIX proposal in several ways. First, it addresses closely integrating third-party accelerators directly into the coherent memory system, rather than isolating them behind an IOMMU on the PCIe bus. Second, it directly demonstrates the importance of allowing customized memory hierarchies to efficiently exploit accelerator memory access behavior. Third, it proposes mechanisms that allow the host system to protect itself from buggy and malicious third-party accelerators. We have anecdotal evidence that Crossing Guard may impact future revisions of the CCIX standard; after a recent presentation of this work, a member of the CCIX committee in the audience asked "how do we get this into CCIX?" While only time will tell how CCIX evolves, we believe our work can provide insights for future iterations of such standards.

Beyond its direct applicability to accelerators, Crossing Guard also contributes to the body of knowledge on coherence in general. We know of no other work that describes hardware to translate between different coherence protocols. Such an interface allows a heterogeneous coherence system, without making the details of each protocol in the system visible to the others. Using an API to allow internal implementations to be treated as black boxes eases the testing and verification burden. For example, with Crossing Guard, a host protocol only needs to know that a block has been sent to the accelerator and not yet written back; it does not need to know where in the accelerator hierarchy the block resides or what state it is in. Having a smaller state space eases testing and verification. Coherence interfaces like Crossing Guard may therefore inspire further research into heterogeneous systems containing multiple coherence protocols.

Another aspect of Crossing Guard that may inspire future work is in fault-tolerant / secure coherence. Prior work on fault-tolerant coherence has assumed that errors are caused by faults, but that controllers are implemented correctly [5]; it also assumes non-Byzantine failures. Therefore, prior work uses approaches like rollbacks to guarantee data correctness. In contrast, Crossing Guard assumes some controllers are trusted, and others are potentially either incorrect or malicious. By guaranteeing only that the trusted controllers always see a consistent coherence state, Crossing Guard avoids much of the overhead of prior fault-tolerant coherence protocols. Future work may explore memory systems where different components are given different levels of trust.

Test of Time Citation

Crossing Guard demonstrated how complex coherence protocols could be abstracted to a simple high-level API, providing third-party accelerators with simple, safe, and efficient access to coherent shared memory.

References

[1] *Cache Coherent Interconnect for Accelerators (CCIX)*. [Online]. Available: http://www.ccixconsortium.com/

[2] J. Goodacre, "The evolution of the ARM architecture towards big data and the data-centre," Nov. 2013.

[3] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA '17*.

[4] S. Kumar, A. Shriraman, and N. Vedula, "Fusion : Design tradeoffs in coherent cache hierarchies for accelerators," in *ISCA '15*.

[5] 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 *ISCA '02*, pp. 123–134.

[6] 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, no. 1, pp. 7:1–7:7, Jan. 2015.