2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Spandex: A Flexible Interface for Efficient Heterogeneous Coherence

Johnathan Alsop[1], Matthew D. Sinclair[1,2], and Sarita V. Adve[1]

[1]University of Illinois at Urbana-Champaign
[2]AMD Research, University of Wisconsin - Madison
[1]*hetero@cs.illinois.edu*

***Abstract*—Recent heterogeneous architectures have trended toward tighter integration and shared memory largely due to the efficient communication and programmability enabled by this shift. However, such integration is complex, because accelerators have widely disparate methods for accessing and keeping data coherent. Some processors use caches backed by hardware coherence protocols like MESI, while others prefer lightweight software coherence protocols or use specialized memories like scratchpads with differing state and communication granularities. Modern solutions tend to build interfaces that extend existing MESI-style CPU coherence protocols, often by adding hierarchical indirection through intermediate shared caches. Although functionally correct, these strategies lack flexibility and generally suffer from performance limitations that make them sub-optimal for some emerging accelerators and workloads.**

Instead, we need a flexible interface that can efficiently integrate existing and future devices – without requiring intrusive changes to their memory structure. We introduce Spandex, an improved coherence interface based on the simple and scalable DeNovo coherence protocol. Spandex (which takes its name from the flexible material commonly used in one-size-fits-all textiles) directly interfaces devices with diverse coherence properties and memory demands, enabling each device to communicate in a manner appropriate for its specific access properties. We demonstrate the importance of this flexibility by comparing this strategy against a more conventional MESI-based hierarchical solution for a diverse range of heterogeneous applications. On average for the applications studied, Spandex reduces execution time by 16% (max 29%) and network traffic by 27% (max 58%) relative to the MESI-based hierarchical solution.

I. Introduction

Architectures are increasingly relying on parallelism and hardware specialization to exceed the limits of single core performance. As GPUs, FPGAs, and other specialized devices are incorporated into systems ranging from mobile devices to supercomputers and data centers, programmability and inter-device communication have become bottlenecks for many applications. A unified coherent address space can greatly improve programmability and communication efficiency. As a result, many heterogeneous architectures are moving towards tighter host and device integration [1], [2], [3], [4].

Unfortunately, efficiently implementing coherence between multiple heterogeneous devices is hard because such devices may have widely varying memory demands. As emerging systems on chip (SoCs) try to ease the programming and communication bottlenecks of heterogeneous computation through tighter integration of multiple devices, efficiently interfacing the devices' diverse coherence strategies becomes an important challenge. The design of such an interface must take into account the device properties and data access patterns that motivate these different coherence strategies.

For example, CPU applications tend to prefer low latency memory accesses. As a result, they use coherence protocols like MESI which obtain persistent read and write permissions for data at line granularity. Although such protocols are complex and can incur high coherence overheads, they can also be very effective at exploiting cache reuse, which is important for applications sensitive to memory latency. On the other hand, conventional GPU applications are throughput-oriented and exhibit streaming access patterns. Thus, the complexity and overheads of MESI are a poor fit for these applications [5]. Instead, GPUs prefer a simple, lightweight coherence protocol which writes through dirty data and performs cache self-invalidation at synchronization points. Like prior literature, we refer to this protocol as *GPU coherence* [6], [7]. GPU coherence works well for streaming workloads with little data reuse or sharing, but is inefficient for emerging applications that have more frequent synchronization [8], [9], [10], [11], [12], [13]. Recent work has shown that the DeNovo coherence protocol improves scalability and efficiency for both CPU [14], [15], [16] and GPU [6], [7] applications. Section II discusses each of these approaches in more detail.

Current interfaces attempt to integrate these disparate needs by building on hardware coherence protocols designed for multicore CPUs. These schemes use a fixed granularity MESI-based coherence protocol (including variants such as MOESI and MESIF) to interface heterogeneous devices [4], [17], [18]. This approach is unsurprising because designing and verifying a new hardware coherence protocol is difficult, and existing MESI-based protocols have already been optimized for multicore CPUs. However, as systems start integrating increasing numbers of devices with increased memory throughput demands, the overheads of a MESI-based protocol become onerous.

In this work we introduce Spandex, a novel coherence interface for integrating heterogeneous devices. Spandex is designed to efficiently support memory requests from a broad spectrum of existing and future devices and workloads. Whether a device prefers to request data or propagate updates in a similar manner to a MESI cache, a GPU coherence cache, or using some novel hybrid strategy such as DeNovo, it is able to dynamically select its ideal coherence policy tradeoffs in a Spandex system. Unlike prior solutions, which build upon complex CPU cache coherence protocols, Spandex extends the hybrid DeNovo protocol to support a wide range of memory demands in a simple and scalable manner.

The rest of this paper is organized as follows. Section II discusses the trade-offs of existing CPU and GPU coherence strategies, classifies them based on three important coherence

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00031

Coherence Strategy	Stale invalidation	Write propagation	Granularity
MESI	writer-invalidation	ownership	line
GPU Coherence	self-invalidation	write-through	loads: line stores: word
DeNovo	self-invalidation	ownership	loads: flexible stores: word

TABLE I: Coherence strategy classification.

design dimensions, and highlights the limitations of existing strategies for integrating different protocols. Section III describes how the Spandex design implements coherence for devices with any combination of the aforementioned coherence design dimensions. Sections IV and V describe how Spandex is evaluated against a more conventional hierarchical MESI-based interface for a set of CPU-GPU applications with a variety of sharing patterns. We find that on average the best Spandex cache configuration reduces execution time and network traffic relative to the best configuration possible in a more conventional hierarchical MESI interface by 16% (max 29%) and 27% (max 58%), respectively for the applications studied.

II. Background

Devices in heterogeneous systems can have a wide range of memory demands which in turn motivate different cache coherence strategies. We identify three dominant design decisions: 1) how stale data is invalidated, 2) how written data is propagated, and 3) what granularity is used for state tracking and communication. Sections II-A–II-C describe three coherence protocols – MESI, GPU coherence, and DeNovo – that make different choices for the above decisions (summarized in Table I). Section II-D discusses existing solutions for integrating diverse coherence strategies in heterogeneous systems.

A. MESI Coherence

MESI-based protocols are designed to exploit as much locality as possible by using **writer-initiated invalidation**, **ownership**-based (write-back) caches, and **line granularity** state and communication. Writer-initiated invalidation means that every read miss triggers a request for read permission in the form of Shared state – this permission is revoked only when a future write to the line sends an explicit invalidation to the reader. Ownership-based caching means that every cache miss for a write or atomic read-modify-write (RMW) access triggers a request for exclusive write permission in the form of ownership, or Modified state. Both read and write permissions are requested for the full line in MESI. Once read permission (Shared state) or write permission (Modified or Exclusive state) is obtained, subsequent reads or writes to the same cache line may hit in the cache until a conflicting access from a remote core causes a downgrade or until the cache line is evicted. This strategy can offer high cache efficiency by exploiting temporal and spatial locality, but it comes at a cost.

The overhead incurred by writer-initiated invalidation imposes throughput and scalability limitations on the system. Each write miss must trigger invalidations in every core that may have read or write permissions for the target cache line. This can incur significant latency, storage, and communication bottlenecks as core counts and memory request throughput demands increase.[1] Tracking state at cache line granularity rather than word granularity helps reduce this overhead. However, it also increases the likelihood of false sharing, which causes wasteful communication, latency, and state downgrades when cores access different words in the same cache line.

In addition, MESI-based protocols also suffer from high complexity. MESI is a read-for-ownership (RfO) protocol, which means that servicing a request for the Modified state requires transferring ownership as well as providing up-to-date data. On a write miss, both the missing cache and the LLC must transition to a transient blocking state, typically delaying subsequent requests to the target address while all remote owners or sharers are downgraded and up-to-date data is retrieved. These transient states degrade performance, add complexity, and make extensions and optimizations to MESI-based protocols difficult to implement and verify.

MESI's limited scalability, high complexity, and ability to exploit locality make it a good fit for conventional multicore CPU or accelerator workloads with small core counts, high locality, memory latency sensitivity, and low memory bandwidth demands.

B. GPU coherence

While MESI is ideal when locality is high and memory throughput demands are low, traditional GPU applications often exhibit limited temporal locality and are more tolerant to memory latency because of their highly multi-threaded and parallel execution. Here the reuse benefits of MESI are less helpful, and the high core counts and high memory throughput demands of GPUs exacerbate the inefficiencies of MESI and make writer-initiated invalidation inefficient [5]. Instead, GPU coherence protocols are typically designed for high bandwidth and simplicity. Rather than obtaining ownership for writes, GPU L1 caches **write-through** dirty data to the backing cache. Similarly, atomic read-modify-write (RMW) operations bypass the L1 and are performed directly at the backing cache. Rather than sending invalidation messages to potential sharers on a write miss, GPU caches rely on software cues (typically synchronization or atomics) to **self-invalidate** potentially stale data in the local cache at appropriate points. Finally, request granularity is chosen to exploit spatial locality while minimizing coherence overheads: read requests are sent at **line granularity** while write-through and RMW requests are sent at the granularity of updates or **word granularity**. We refer to this coherence strategy as GPU coherence.

The primary advantage of GPU coherence lies in its simplicity. By using write-through caches and self invalidation, GPU L1 caches avoid the overheads of obtaining read and write permission, including sharer invalidation, indirection, and transient blocking states. Sending write-through and RMW requests at the granularity of updates avoids the latency and communication overheads of RfO caches. However, GPU coherence can still exploit locality by coalescing stores to the same line in the write buffer. Self-invalidation enables read

[1] Which of these overheads dominates depends on the sharing patterns of the target workload, and whether a snoopy or directory-based protocol is used.

data to be tracked and communicated at line granularity without the risk of wasteful downgrades due to false sharing. As a result, GPU coherence exploits the abundant spatial locality in GPU workloads while sustaining a higher memory request bandwidth than is possible in a MESI-based protocol.

However, the simplicity of GPU coherence comes with some costs when it comes to synchronization. GPU coherence typically relies on a data race-free (DRF) consistency model, which requires software to differentiate synchronization and data accesses so that invalidations and store buffer flushes can be performed only when necessary.[2] In GPU coherence, these synchronization accesses (e.g., kernel boundaries, atomic updates, reads and writes to a lock) invalidate the entire L1 cache, which can significantly limit cache efficiency if synchronization is frequent. Past work aims to limit this cost when synchronization is local (i.e., communication between threads that share a cache) or can be relaxed (i.e., eliding the flush or invalidate results in acceptable behavior) [6], [7], [19], [20]. Even with these optimizations, GPU coherence is still a poor fit for conventional CPU workloads which exhibit high locality and latency sensitivity. Instead, GPU coherence performs best for workloads where global synchronization is infrequent, temporal locality is absent or reuse distance is small, and memory latency is not a performance bottleneck.

C. DeNovo Coherence

The DeNovo protocol can be thought of as a sweet spot between the complexity and cache efficiency of MESI at one end and the simplicity and expensive synchronization actions of GPU coherence at the other. Like MESI, DeNovo coherence obtains **ownership** for stores and atomic accesses. However, like GPU coherence, DeNovo **self-invalidates** stale data at synchronization points. Both read and write requests are sent (and Owned state is tracked) at **word granularity**, although a read response may be sent at line granularity when more data in the requested line is available (Owned) at the responding core. By requesting and tracking ownership at modification granularity, DeNovo avoids false sharing. In addition, writes do not need to request up-to-date data and ownership can be transferred without transient blocking states delaying subsequent requests.

Like GPU coherence, DeNovo must self-invalidate potentially stale data at synchronization points. However, this operation is less expensive for DeNovo than for GPU coherence. This is because DeNovo obtains ownership for writes and atomic accesses instead of writing them through to the backing cache. Since Owned data is not invalidated at synchronization points, DeNovo caches are able to exploit reuse in this data even in the presence of frequent synchronization. Thus, DeNovo is able to achieve better cache efficiency than GPU coherence while avoiding much of the coherence overhead, false sharing, and transient states that come with writer-initiated invalidation and RfO protocols like MESI. This makes DeNovo a good fit for a wide range of CPU, GPU, and accelerator workloads.

Of course, DeNovo is not ideal for every workload. By using word granularity writes, DeNovo is less able to exploit spatial locality in written data (although writes to the same line can be coalesced into a single request in the DeNovo write buffer). Additionally, self-invalidation can still harm cache efficiency if there is locality in non-owned data. DeNovo proposes the use of regions to address this inefficiency by selectively invalidating only potentially stale data based on information from software. Even with this optimization, however, DeNovo performs best for programs with high temporal locality in written data and low locality or predictable sharing patterns in read data.

[2]DRF models are commonly used for CPUs as well, but synchronization does not trigger flush or invalidate operations in MESI caches.

D. Heterogeneous Coherence Solutions

There are many recent and ongoing efforts to implement coherent memory between devices with heterogeneous memory demands. This section describes key recent work that influenced the baseline system for our evaluations; Section VI provides a more comprehensive description of related work.

Most existing solutions rely on an assumption of limited inter-device communication demands. This motivates a MESI-based last level protocol, potentially with an intermediate cache level for filtering requests from devices such as GPU cores, which don't work well with MESI.

For example, the IBM Coherent Accelerator Processor Interface (CAPI) enables FPGAs and other accelerators to use a coherent MESI-based cache which interfaces with a snoopy MESI-based last level cache fabric [4]. The use of a MESI-based protocol means that sharing patterns unsuitable for MESI may incur excessive coherence overhead.

The AMD APU uses a hierarchical cache structure with a MESI-based directory to integrate its CPUs and GPUs [17]. This approach is best suited for hierarchical sharing patterns; intra-device communication is prioritized while communication between CPU and GPU cores suffers added latency and energy overhead due to hierarchical indirection and blocking transient states at the LLC. In our evaluation, the hierarchical MESI configuration is based on this design.

ARM ACE [18] specifies an interface for implementing coherence between CPUs and accelerators. Unlike CAPI, ACE defines "non-cached" read and write request types of varying granularity – these could potentially be used in a similar way as self-invalidated loads and write-through stores, although details on how such aspects would integrate with the rest of the system were not readily available. Unlike the AMD APU, ACE allows direct integration of CPUs and accelerators. However, ACE uses a centralized snoop-based interface, MOESI protocol, and line granularity state tracking, which potentially result in limited scalability, high protocol complexity, and several transient blocking states for inter-device communication.

Until recently, the assumption of limited inter-device sharing has been valid. Inter-device communication has historically been expensive, so conventional heterogeneous algorithms tend to prioritize local communication when possible. For high-throughput devices, intermediate shared caches can therefore be expected to efficiently filter and coalesce requests from cores within a device, limiting the request bandwidth to the last level protocol. However, as accelerators become more tightly coupled and heterogeneous applications become more capable and diverse, it is unclear that hierarchical approaches generalize efficiently to a broader range of access patterns.

III. SPANDEX DESIGN

The Spandex design can be divided into device side logic and integration logic. At both the device side and the integration side, Spandex defines a set of supported states, a request interface, and the request handling and state transition logic required to implement Spandex.

Section III-A begins by describing the supported device states and the request types used to interface a device with the Spandex LLC. At the integration side, Section III-B discusses the states, requests, and device transitions at the Spandex LLC. Section III-C specifies the logic required at the device side to handle forwarded requests and probes from the Spandex LLC, and Section III-D provides examples of how a thin per-device translation unit (TU) may be used to implement elements of this logic when it is not natively supported by device caches. Section III-E discusses the memory consistency assumptions of Spandex, and Section III-F discusses the overheads of Spandex relative to a line granularity MESI-based LLC.

Throughout, Figures 1a-1d are used to illustrate how Spandex handles some basic request types of varying granularity. In each figure a CPU, GPU, and custom accelerator interface directly with the Spandex LLC. Each device has a local cache tailored to the memory demands of that device that connects to the Spandex system through a custom translation unit.

A. Spandex Device: States and Request Types

Spandex supports four stable coherence states at an attached device memory: Invalid (I), Valid (V), Owned (O), and Shared (S). Although attached devices may use other protocol state names, any internal state should map to one of these supported states from the perspective of the rest of the system. Devices may track state at any granularity. However, requests for state and data may only occur at word or line granularity, and devices must be able to handle responses, forwarded requests, and probes at word granularity (discussed more in Section III-C). The device states and when it generates a Spandex request are:

- **I** indicates that the data is invalid. A read or write from the device must generate a Spandex request.
- **V** indicates that the data is up-to-date. A read hits without further action but a write initiates a Spandex request. A key attribute of **V** is that the device itself is responsible for self-invalidating valid data at appropriate points to ensure no stale data is read. The consistency model defines when data becomes stale (Section III-E).
- **S** is similar to **V** except that the device is not required to self-invalidate Shared state data. The system (Spandex LLC) is responsible for sending an invalidation message when Shared data becomes stale.
- **O** indicates an exclusive up-to-date copy of the data. A read, write, or atomic RMW hits without further action.

Spandex's flexibility arises from its ability to interface devices with widely varying memory demands. There are 7 request types that may be issued from a Spandex device. Requests carry granularity information (word or line),[3] and multiple word granularity requests to the same line may be coalesced into a single multi-word request with a bitmask indicating the targeted words within the cache line.

[3]Although other request granularities may be useful for some workloads, word and line granularity were sufficient for the devices we considered.

Device Type	Device Request	Spandex Request	Granularity
GPU Coherence	Read	ReqV	line
	Write	ReqWT	word
	RMW	ReqWT+data	word
DeNovo	Read	ReqV	flexible*
	Write	ReqO	word
	RMW	ReqO+data	word
	Owned Repl	ReqWB	word
MESI	Read	ReqS	line
	Write	ReqO+data	line
	RMW	ReqO+data	line
	Owned Repl	ReqWB	line

TABLE II: Type and granularity of requests generated for read misses, write misses, and replacements of owned data in GPU coherence, DeNovo, and MESI caches. *A DeNovo ReqV request is issued at word granularity, but the responding device may include any available up-to-date data in the line.

The following requests are sufficient to support devices that use write-through or ownership for updates, self-invalidating or writer-invalidated reads, and diverse request granularities. Table II provides a mapping from GPU coherence, DeNovo, and MESI requests to Spandex request types and granularity.

- **ReqV** is generated for a self-invalidated read miss; it simply requests the up-to-date data at the target address. A RspV response causes a transition to V state.
- **ReqS** is generated for a writer-invalidated read miss; it requests both up-to-date data and Shared state. A RspS response causes a transition to S state.
- **ReqWT** is generated for a write-through store miss. The granularity of this request type is the same as the granularity of modification, so up-to-date data is not needed to satisfy this request. If target data was previously in I state, a ReqWT operation causes a transition to V state at the time of update.
- **ReqO** is generated for an ownership-based store miss that is overwriting all requested data. Thus, it requests ownership but not the up-to-date data. If target data was in I, V, or S state, a ReqO operation causes a transition to O state at the time of update.
- **ReqWT+data** sends an update operation to be performed at the LLC. This is similar to a ReqWT, but the operation also requires up-to-date data. Unlike ReqWT, which simply overwrites the current data value, this request must specify the required update operation and may be used for an atomic read or RMW performed at the LLC.[4] A RspWT+data response carries the value of the data before the update was performed and triggers a downgrade at the device cache (since the response data is potentially stale).
- **ReqO+data** is generated from an ownership-based (write-back) cache for a request that needs both the up-to-date data *and* ownership. This may be used to request ownership for a locally performed RMW operation, or for a store in a line granularity ownership-based cache which does not overwrite all data in the line. A RspO+data

[4]For GPU coherence, all atomic accesses are performed at the LLC.

response causes a transition to O state.
- **ReqWB** writes back owned data to the LLC. This request is necessary whenever Owned data is downgraded as a result of local cache actions (e.g., a cache replacement). During a pending ReqWB, up-to-date data must be retained until the write-back has completed.

Every Spandex request (Req) type has an associated response (Rsp) type. Since Spandex tracks ownership at word granularity (see Section III-B), different words within a single multi-word request may be satisfied at different devices. Therefore, a device that can issue multi-word requests (including line granularity requests) must be able to handle multiple partial word granularity responses. The implications of this requirement for line-based protocols are discussed in Section III-D.

B. Spandex LLC

LLC States

There are four stable coherence states at the Spandex LLC: Invalid (I), Valid (V), Owned (O), and Shared (S). I, V, and O come directly from DeNovo while S is needed to support devices that use writer-initiated invalidation.

- **I** indicates that only the backing memory is guaranteed to have an up-to-date copy of the data.
- **V** indicates the data at the LLC is up-to-date and is not in Shared or Owned state in any attached device memory.
- **S** indicates the data at the LLC is up-to-date, but one or more sharer devices may require invalidations if the data is modified or replaced.
- **O** indicates the target data is Owned within an attached device memory.

Spandex tracks ownership at word granularity, and this helps avoid many of the inefficiencies present in MESI-based protocols. In protocols with line granularity ownership state (e.g., MESI), an ownership request must revoke ownership from the previous owner and wait until ownership and data for the full line have been transferred. This is especially wasteful in the case of false sharing, when the devices are accessing different words in the same line. With word granularity ownership state, devices can issue ReqO and ReqWT requests for exactly the words being updated in a line. These requests can be satisfied without obtaining up-to-date data (because the requested data is overwritten), or revoking ownership for the entire block.

Figure 1a demonstrates these benefits. The accelerator device issues a word granularity ownership request (①) which triggers an immediate transition to owned state and a data-less RspO for the target words (②). The GPU then issues a write-through request for disparate words in the same line (③), resulting in an immediate update of the LLC data and a data-less RspWT to the requestor (④). Due to the word granularity ownership tracking, false sharing is avoided and write-only requests proceed without requiring blocking states at the LLC or data responses.

To limit tag and state overhead, allocation occurs at line granularity. For each line, two bits indicate whether the line is Invalid, Valid or Shared. For each word within the cache line, a single bit tracks whether the word is Owned in a remote cache. For each Owned word, the data field itself stores the ID of the remote owner (this is similar to how DeNovo tracks owners).

Request Type	Next State	Fwd Msg
ReqV	–	ReqV
ReqS (1)	S	ReqS
ReqS (2)	–	ReqV
ReqS (3)	O	ReqO+data
ReqWT	V	ReqO
ReqO	O	ReqO
ReqWT+data	V	RvkO
ReqO+data	O	ReqO+data
ReqWB from owner	V	–
ReqWB from non-owner	–	–

TABLE III: The state transition triggered at the LLC by each request type (Next State) and the request type forwarded to the owning core in the event the data is in O state (Fwd Msg). An entry of – indicates no transition or forwarded request is necessary.

Implementing byte granularity ownership state at the LLC is also possible, but it would incur more overhead (a bit per byte vs. a bit per word). Based on the workloads we studied, byte granularity stores that cannot be coalesced with others into a full word store are expected to be rare, so the benefits of byte granularity state tracking does not appear to be worth the overheads. Spandex therefore requires byte granularity stores to use word granularity ReqWT+data or ReqO+data rather than ReqWT or ReqO requests to ensure non-modified data in the requested word remains up-to-date.

LLC Requests

In addition to coherence requests generated at the attached devices, the LLC itself may initiate requests when necessary for downgrading the state in remote owner or sharer devices.

- **RvkO** is used to revoke ownership from an owner device and trigger a write-back of the owned data, and has a corresponding response type of **RspRvkO**.
- **Inv** is used to invalidate shared data in a sharer device and has a corresponding response type of **Ack**.

LLC State Transitions

The Spandex LLC serves as the coherence point for all device caches and serializes all write requests (ReqWT[+data] and ReqO[+data]) to a given (coherent) data address. Table III describes, for each device request type, the next stable state at the LLC and, in case the initial state is O, the message the LLC must forward to the owner. ReqS requests can be handled in multiple ways (see discussion of Shared state below). For multi-word requests, each word is handled individually, potentially triggering different actions.

In the common case, request handling occurs immediately without any blocking states. For all requests, if the target data is in V state at the LLC, the LLC immediately satisfies the request, triggers a state transition, and responds to the requestor. If a read request (ReqV or ReqS) arrives for target data in S state, the LLC immediately responds to the requestor and, in the case of ReqS, updates the sharer list. If the target data is in O state in the LLC, the LLC immediately forwards the request to the owning core rather than responding to the requestor. All

requests other than ReqWT+data and ReqS (1) for data in O state trigger an immediate state transition.

For some state transitions, blocking states are needed to wait for downgrades and write-backs to complete. Specifically, if a write request (ReqWT[+data], ReqO[+data], ReqWB) arrives for target data in S state, the LLC must send Inv requests to any potential sharer devices and transition to a blocking state while waiting for Ack responses. If a ReqS (1) or ReqWT+data request arrives for target data in O state, the LLC transitions to a blocking state while it waits for the forwarded request to trigger a write-back from the owning core. In both cases, once all Acks have been collected or the write-back has completed, the data transitions to the next stable state specified in Table III.

Figure 1b illustrates how a word granularity ReqWT+data request (①) is handled when the target data is in O state at the LLC. Since a ReqWT+data requires both up-to-date data and write serialization, the LLC sends a RvkO request to the owning accelerator device and transitions to a transient state *tr* (②). After receiving the RvkO, the owner device responds with a RspRvkO request for the entire line (③). The LLC performs the requested update and responds to the GPU and MESI caches with a RspWT+data and RspWB, respectively (④).

Supporting Shared State

To efficiently integrate devices that use writer-invalidated reads (e.g., MESI caches), Spandex extends the DeNovo protocol with S state. However, a writer-invalidated read (ReqS) does not always need to trigger a transition to S state. Spandex can handle a ReqS request in multiple ways, as Table III illustrates, depending on the system's demands and constraints.

The LLC may implement writer-initiated invalidation and transition to S state, represented by option (1). Supporting writer-initiated invalidation can improve reuse for data that is concurrently read by multiple writer-invalidated device caches. However, it also incurs some complexity and overhead in the protocol. A ReqS (1) for data in O state triggers a transition to a blocking state while the current owner completes a writeback. A write request (ReqO[+data], ReqWT[+data]) for data in S state triggers a transition to a blocking state while potential sharers are invalidated. In addition, tracking and invalidating sharers incurs storage and network traffic overheads. These overheads are consistent with the overheads of writer-initiated invalidation in MESI.

Alternatively, a ReqS may be treated the same as a ReqV or ReqO+data request, represented by options (2) and (3) in Table III, respectively. Both options avoid the complexity and overheads of Shared state. However, option (2) requires the requesting cache to downgrade the target data to Invalid after the read operation is satisfied and therefore precludes any further reuse in the requestor cache. In contrast, option (3) requires the requesting cache to upgrade the target data to Owned state after the read operation is satisfied. This enables additional reuse in the requestor cache, but precludes multiple concurrent readers and can lead to wasteful ownership transfers if there is high contention for the target data.

In our evaluation, Spandex uses option (1) if the target data is in S state or owned in a MESI core. In all other situations (I, V, or owned at a non-MESI core), we use option (3). This is similar to MESI's response to a Shared request with Exclusive state if data is not present in another MESI cache.

Spandex Request	Expected State	Next State	Response
ReqV	O	O	RspV to requestor
ReqO	O	I	RspO to requestor
ReqO+data	O	I	RspO+data to requestor
RvkO	O	I	RspRvkO to LLC
Inv	S	I	Ack to LLC
ReqS	O	S	RspS to requestor RspRvkO to LLC

TABLE IV: The state transition and response message triggered at a device by each external Spandex request. If the target data is not in the expected state when a request arrives, different behavior may be required (discussed in Section III-C).

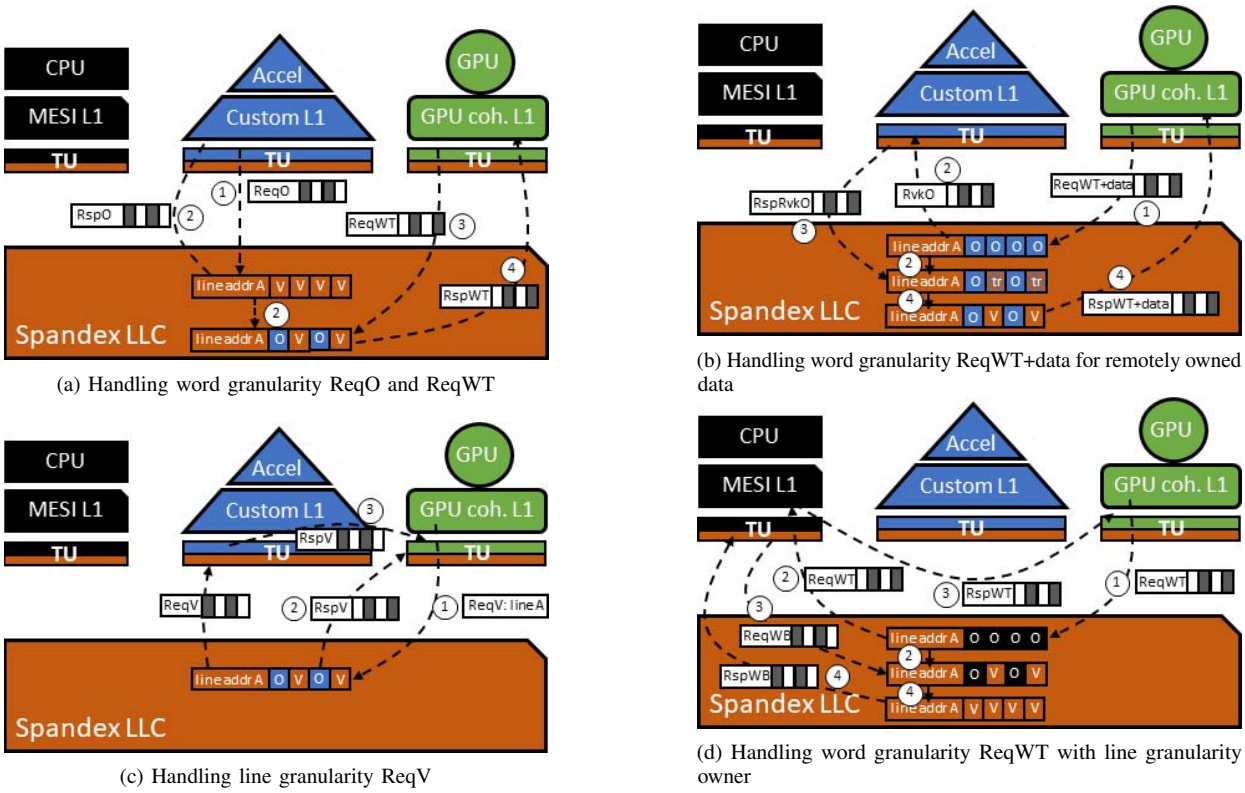
C. Spandex Device: Handling External Requests

We next describe how a Spandex device must handle external messages (forwarded requests and probes) it may receive from the Spandex system. Table IV summarizes, for each external request type, the expected device state of the target data, the next stable state, and the response sent for each external request type. A device must be able to handle an external request type if it supports the expected state for that request. Since Spandex tracks ownership at word granularity (see section III-B) and external requests may therefore arrive at word granularity, devices must be able to implement these transitions and responses at word granularity. Section III-D discusses how the per-device TU can be used to implement this requirement in line granularity caches.

It is possible that the device state will not match the expected state of the specified transition at the time the request arrives. This happens when data requests race with other requests or write-backs to the same data, and it can require different handling behavior. When an external request arrives, the target data may be (1) in a pending transition to the expected state, (2) in a pending transition away from the expected state, or (3) specifically for an Inv request or a forwarded ReqV request, in a stable state other than the expected state. The behavior required for cases (1) and (2) is consistent with how conventional protocols handle race conditions. Case (3), however, involves a race that is unique to Spandex and thus requires additional consideration.

1) Pending Transition to Expected State: If an external request requires a data response (ReqV, ReqS, ReqWT+data, ReqO+data, or RvkO request) and up-to-date data is not available but is pending (possible for a pending ReqO+data), the external data request must be delayed until the device's pending data request completes. In many cases, however, up-to-date data is either available (the pending request is a ReqO) or unneeded (the external request is a ReqO or Inv), and a response to the external request may be sent immediately.

2) Pending Transition from Expected State: The required device behavior depends on whether its pending transition is an upgrade or a downgrade. If it is an upgrade, then it must be a pending transition from S to O (since external requests expect data in S or O state, and no upgrade is possible from O)



(a) Handling word granularity ReqO and ReqWT

(b) Handling word granularity ReqWT+data for remotely owned data

(c) Handling line granularity ReqV

(d) Handling word granularity ReqWT with line granularity owner

Fig. 1: Handling basic request types at the Spandex LLC

with an external Inv request. In this case no state transition is necessary and the external request will not require up-to-date data; the device should respond immediately, and the pending upgrade may proceed as normal.

If the pending transition is a downgrade from the expected state, it must be a transition from O to I due to a pending ReqWB (all other downgrades from O or S happen immediately). Here the device may respond immediately, with data if necessary, to the forwarded request.[5] If the external request itself triggers a downgrade (ReqO[+data], RvkO, ReqS), it should also trigger completion of the pending ReqWB since the LLC no longer considers this device the owner.

3) Stable State other than Expected (Inv or ReqV): Inv requests differ from other request types because they expects a state which can be silently downgraded on the device side. If a device receives an Inv request for data in a stable state other than S, it may simply Ack the request without updating state.

ReqV requests differ from other request types in that they do not affect coherence state at the LLC or the owning core and they enforce no global ordering with other operations to the same target data. As a result, when a ReqV request is forwarded to an owning core, that core may completely transition away from Owned state before the forwarded ReqV arrives since neither the owning core nor the LLC has any way to know there is a ReqV request en route for the owned data. When this happens, Spandex requires that the incorrectly assumed owner Nack the failed ReqV request, and the requesting device retry the ReqV. This is the same strategy used by DeNovo.

However, Spandex faces a new challenge not encountered by MESI, GPU coherence, or DeNovo alone: ReqV starvation. Only Spandex allows line granularity ownership requests from MESI caches to race with ReqV requests from GPU coherence and DeNovo caches. When a ReqV request races with frequent ownership requests, a rapidly changing ownership state can lead to repeated failure and eventual starvation for the ReqV access.

To avoid ReqV starvation, after a finite number of failed ReqV requests (1 in our evaluation), a Spandex device must replace the failed ReqV request with a request type that will enforce ordering for racy accesses: a ReqWT+data or ReqO+data request. Both request types enforce a global ordering of operations to a given address in the presence of racy ownership requests and ensure forward progress.

D. Spandex Device: TU Responsibilities

Although much of the required functionality described in section III-C is implemented natively by attached device caches, the per-device translation unit (TU) is responsible for filling in any gaps. Here we describe how the TU helps to implement these requirements for GPU coherence, DeNovo, and MESI caches in a Spandex system.

The states used in GPU coherence (I, V) and DeNovo (I, V, O) map directly to Spandex states I, V, and O. MESI states I

[5]In the case of a RvkO or ReqS, the RspRvkO does not need to carry data since up-to-date data has already been sent in the pending ReqWB.

and S similarly map directly to Spandex states I and S, while M and E both map to O state.

Required functionality for GPU coherence TU: Since GPU coherence does not support O or S state, its caches do not need to handle forwarded requests or probes. However, GPU coherence alone does not support ReqV retries, so the TU must retry a Nack for a ReqV and retry the request as a ReqWT+data. In addition, GPU coherence can issue multi-word requests but may not handle partial word granularity responses. Thus, the TU must collect and coalesce responses from multiple sources before sending a response to the GPU cache.

In Figure 1c, the GPU coherence TU must coalesce word granularity responses for a line granularity request. First, the GPU sends a line granularity request for valid data to the LLC (①). The LLC immediately responds with valid data in the line and forwards a word granularity request for remotely owned words (②). No state transition is necessary. The owner responds directly to the requestor with valid data (③). The GPU's TU coalesces the responses and sends the GPU cache a line granularity response once all have arrived.

Required functionality for DeNovo TU: The word-based DeNovo protocol already handles partial responses for multi-word requests, as well as responding to forwarded requests for owned data at word granularity. The only added functionality needed at the TU is the ability to replace a Nacked ReqV request with a ReqWT+data or ReqO+data request after a finite number of tries (DeNovo alone does not need to do this).

Required functionality for MESI TU: MESI requires the most help from the TU of the three protocols discussed. Like GPU coherence, the TU must collect and coalesce responses from multiple sources before sending a response to the line-based MESI cache. MESI devices can natively handle ReqS and Inv requests. Since a MESI cache supports O state, it must also be able to handle external requests for owned data at word granularity, which is complicated by MESI's line granularity state. Depending on the state of the target data when the external request arrives, there are three cases discussed in detail below: (1) the data is in stable O state, (2) there is a pending request to bring the data in O state, and (3) there is a pending write-back request for the target data.

1) O state: Requests that require word granularity data or ownership downgrades are converted to line granularity and passed through to the MESI cache. If the request required ownership downgrade for only part of the line, the TU must trigger a ReqWB for any non-downgraded words. Figure 1d illustrates this case. First, the GPU sends a word granularity write-through request for data that is remotely owned at the MESI cache (①). The LLC immediately updates the ownership state and data of the written words (to V) and forwards the write-through request to the MESI owner (②). The MESI cache downgrades the requested data, responds directly to the requestor, and triggers a write-back for the words that were not requested in the downgraded line (③). Lastly, the LLC handles the ReqWB and responds (④).

2) Pending O request: The TU is responsible for delaying word granularity requests that require data (ReqV, ReqS, ReqO+data, RvkO) and responding immediately for for requests that only require ownership downgrade (ReqO). After ownership for the full line has been received, if any downgrade requests have been received, the TU must cause a transition to I state rather than O state for the target line in the MESI cache, and trigger a ReqWB for any words in those lines that had not received downgrade requests.

3) Pending write-back: The TU is responsible for responding to data and ownership downgrade requests for the data being written back. Ownership downgrade requests are handled as write-back responses by the TU.

As previously discussed a forwarded ReqV may arrive while the MESI device is not in any of the above states. In that case, the TU is required to Nack the request.

E. Spandex Consistency Requirements

The memory consistency model implementation requirements can be divided into those that are met within a device (e.g., preserving the program order of certain memory operations) and those that are met at the system level (e.g., ensuring a write appears atomic such that its value is not visible to a core while other cores may still see the stale value). The exact actions depend on the consistency model used and addressing them for general models is outside the scope of this work – Lustig et al. [21] offer a framework for determining exactly what ordering constraints need to be added when interfacing devices with different consistency models. Here we assume a sequential consistency for data race-free (SC-for-DRF) model [22], which is commonly used in CPUs (Java, C++) and, with an added semantics for scoped synchronization [19], in GPUs (HSA, CUDA).

In a DRF program, conflicting data accesses in different threads must by separated by an intervening chain of happens-before order inducing synchronization accesses. For such a program, a load must return the value of the last conflicting write ordered before it by happens-before.

In a Spandex system, the value returned by a load is determined by when ownership and write-through requests occur, when requests for data occur, and when self-invalidation of valid data is triggered. Since synchronization accesses indicate possible inter-device communication in a DRF program (through happens-before), SC-for-DRF may be implemented in a Spandex system by 1) restricting the reordering (or concurrent issue) of synchronization accesses with other accesses in program order, 2) completing write buffer (ownership or write-through) flushes at synchronization points, and 3) self-invalidating valid data at synchronization points. Techniques such as DeNovo regions [14], scoped synchronization [19], relaxed atomics [7], [23], and hLRC [20] may relax these consistency requirements for some synchronization accesses. A Spandex system where the devices obey the above requirements and the LLC ensures serialized and atomic writes can be shown to obey SC-for-DRF.

F. Spandex Overheads

The added request type flexibility offered by Spandex may increase the number of message identifier bits by at most 1 relative to a MESI-based protocol. Word granularity ownership results in the following overheads: an additional state bit per

word is needed in the LLC to indicate owned words, a bitmask is needed for multi-word requests to indicate target words in the line (although this may be offset by sub-line data transfer), more forwarded requests when words in a line are owned at different locations (although this may be offset by reduced false sharing), and a potentially more expensive state read or update since it requires accessing the data field.

Spandex also requires inclusivity for Owned data at the Spandex LLC. It is possible to implement a state-only Spandex LLC, however it would still need to track owner ID at word granularity. As a result, a state-only Spandex LLC cannot achieve the same storage efficiency as a state-only directory for a line granularity protocol.

Many states used in MESI-based protocols such as MESI Exclusive (E), MOESI Owned (MO), or MESIF Forward (F) state are not directly supported in Spandex; these must map to Spandex's O or S state. It is possible to add support for these states in Spandex, but with some added complexity that is mostly well understood in coherence protocol design. Our goal here instead is to show we can integrate flexibility in diverse protocol dimensions that are emerging as more fundamentally important for emerging workloads in heterogeneous systems.

The overhead incurred by the TU will be highly dependent on the functionality supported by the attached device, and it may require slight changes to device cache IP to enable probes or state updates from the TU. We do not provide area or power cost estimates for the TU, but we expect its cost to be comparable to an MSHR (if possible, TU overhead could be reduced by allowing it to interface directly with the device MSHR). We do, however, model TU queuing latency in our evaluation, assuming a single-cycle lookup.

Overall, these overheads are expected to be offset by the advantages of Spandex. By supporting significant request flexibility and tracking ownership at word granularity, Spandex avoids the need for a hierarchical cache structure and offers reduced complexity, blocking states, and false sharing in the protocol. This leads to fewer state transitions, less network traffic, and lower latency for a wide range of sharing patterns.

IV. Methodology

To evaluate Spandex we run a set of CPU-GPU applications on an integrated architectural simulator. We compare Spandex against a hierarchical cache structure, each with a variety of CPU and GPU coherence strategies. Our simulator uses Simics [24] to model the CPU, GEMS [25] to model the memory system, GPGPU-Sim [26] to model the GPU, and Garnet [27] to model the network.

A. Cache Configurations

Table V describes the 6 memory configurations we evaluate classified by LLC protocol, CPU cache protocol, and GPU cache protocol. Configurations with hierarchical MESI LLC (H-MESI) use a hierarchical cache structure in which GPU L1 caches interface with each other through a shared intermediate L2 cache, and CPU L1 caches interface with the GPU L2 cache through a shared MESI LLC. Configurations with a Spandex LLC directly interface CPU L1 caches and GPU L1 caches through the shared Spandex L2. System parameters for these configurations are given in Table VI. We do not evaluate a hierarchical Spandex configuration. Although such an organization is possible and may even be preferable for some workloads, our focus is on Spandex's ability to efficiently interface devices in configurations not possible for existing approaches.

Cache Config.	LLC Protocol	CPU L1 Protocol	GPU L1 Protocol
HMG	H-MESI	MESI	GPU coherence
HMD	H-MESI	MESI	DeNovo
SMG	Spandex	MESI	GPU coherence
SMD	Spandex	MESI	DeNovo
SDG	Spandex	DeNovo	GPU coherence
SDD	Spandex	DeNovo	DeNovo

TABLE V: Simulated cache configurations.

CPU Parameters		
Frequency	2 GHz	
Cores	8	
GPU Parameters		
Frequency	700 MHz	
CUs	16	
Memory Hierarchy Parameters		
Parameter	**Hierarchical**	**Spandex**
L1 Size (8 banks, 8-way assoc.)	32 KB	32 KB
L2 Size (16 banks, NUCA)	4 MB	8MB
L3 Size (16 banks, NUCA)	8 MB	–
Store Buffer Size	128 entries	
L1 MSHRs	128 entries	
L1 hit latency	1	
Remote L1 hit latency in cycles	35−83	
L2 hit latency in cycles	29−61	
L3 hit latency in cycles	CPU: 29−61 GPU: 52−100	–
Memory latency in cycles	CPU: 197−261 GPU: 222−306	197−261

TABLE VI: Simulated heterogeneous system parameters.

We vary the CPU and GPU L1 cache protocols based on what is feasible for each device type and what is supported by the cache hierarchy. CPU L1 caches use MESI or DeNovo, while GPU L1 caches use DeNovo or GPU coherence. The hierarchical MESI LLC only supports MESI CPU caches, but its intermediate GPU L2 can support either DeNovo or GPU coherence requests from GPU L1s. Spandex supports MESI, DeNovo, or GPU coherence requests at the LLC. For caches that use self-invalidation protocols, V data is invalidated in a single-cycle flash operation. In SDG, CPU caches depart slightly from the DeNovo protocol, performing atomic accesses at the L2 rather than obtaining ownership (ReqWT+data rather than ReqO+data requests). By matching the CPU atomic access strategy to the GPU strategy, this avoids blocking states that would otherwise arise from inter-device synchronization.

B. Benchmarks

1) Synthetic Microbenchmarks: Our synthetic microbenchmarks are a set of simple multithreaded CPU kernels and GPU kernels that share data between devices in a way that highlights the performance implications of using a hierarchical vs. flat cache structure, ownership vs. write-through requests for updates, and writer-invalidated reads vs. self-invalidated

reads. While existing applications are generally designed with a fixed coherence strategy in mind, these microbenchmarks highlight a range of sharing patterns that may benefit if systems are given more flexibility in how coherence is implemented.

Indirection: In this microbenchmark, the CPU and the GPU take turns transposing a matrix in a loop. CPU threads read tiles in matrix A and write tiles in matrix B, while GPU threads read tiles in matrix B and write tiles in matrix A. Accesses are strided to reduce spatial locality, and tile size is selected to ensure data is not reused from the L1 cache. Indirection primarily demonstrates the cost of hierarchical indirection.

ReuseO: In this microbenchmark, CPU threads sparsely read matrix A and densely read and write matrix B, and GPU thread blocks sparsely read matrix B and densely read and write matrix A. Tiles are sized to fit in the cache, and the process is repeated iteratively so that data written in one iteration is reused at the same core in the subsequent iteration. ReuseO highlights the benefits of using ownership for updates.

ReuseS: In this microbenchmark, CPU threads and GPU thread blocks take turns densely reading and sparsely writing a shared matrix. It highlights the benefits of writer-initiated invalidation because only Shared state can exploit reuse in read data across iterations; data in Valid state must be invalidated in case it has been updated by a remote compute unit.

2) Applications: We also select a diverse set of applications from Pannotia [9] and Chai [10]. These emerging applications use shared memory to collaboratively execute real-world functions in a CPU-GPU system. From Pannotia we use two iterative graph analytics algorithms: Betweenness Centrality (BC) and PageRank (PR). BC is a push-based algorithm that computes the centrality of each vertex in a graph based on the shortest path to every other node. Each thread updates the neighbors of its assigned nodes and the updates must use atomics since multiple threads may attempt to update the same neighbor. PageRank is a pull-based algorithm that iteratively computes a ranking for every node in a graph based on its neighbors rankings. Threads only update their assigned nodes and only read the values of neighboring nodes, so PageRank does not require atomic accesses. We modified both applications to partition vertices across CPU and GPU cores.

From Chai we use data partitioned and task partitioned applications. Input partitioned histogram (HSTI) is a data partitioned algorithm where CPUs and GPUs use fine-grained synchronization to pop image task blocks from a shared queue and atomically update histogram bins. In-place transposition (TRNS) is a data partitioned matrix transpose algorithm that uses fine-grained CPU-GPU synchronization to arbitrate between threads that are reading and writing conflicting matrix blocks. Random sample consensus (RSCT) is a fine-grained task partitioned algorithm that uses CPU-GPU synchronization to indicate that a sample parameter set produced on the CPU is ready to be consumed on the GPU. CPU cores sparsely read the input matrix and communicate a small amount of data to the GPU, while every GPU core densely reads the same input matrix. Thus, hierarchical sharing is significant. Task queue system histogram (TQH) is a task partitioned algorithm in which the CPU and the GPU use fine-grained synchronization to respectively push to and pop from a set of histogram task queues. Again, CPU cores communicate a relatively small amount of data to GPUs, while GPU cores densely access an input array. However, in TQH each GPU core accesses a different partition of the array, so hierarchical sharing is minimal. GPU cores also use atomic accesses to update a histogram.

Table VII summarizes the communication patterns and execution parameters of each application. Per the Chai categorization, collaborative applications may implement data partitioning or task partitioning. In addition, we classify synchronization between CPU and GPU cores as either fine-grain or coarse-grain, and sharing between CPU and GPU cores as either hierarchical (if a core is more likely to share data with other cores of the same type) or flat (if a core is equally likely to share data with both device types). Finally, we specify whether the application exhibits high or low locality in data or atomic/synchronization accesses.

V. Results

Figure 2 and 3 show the execution time and network traffic of each cache configuration, normalized to HMG, for the synthetic microbenchmarks and collaborative applications, respectively. Network traffic is broken down by request type, and each request category includes the corresponding responses. The Probe network message category represents Inv and RvkO messages. Hbest and Sbest represent the averages of the L1 cache configurations within Hierarchical and Spandex, respectively, that achieve the lowest execution time for each workload. Although existing fixed-protocol caches may be unable to achieve these best-case performance gains for every workload, SBest and HBest illustrate Spandex's ability to support the best possible cache design for a target workload, with an eye towards future caches that may dynamically adapt their coherence strategy.

A. Synthetic Microbenchmarks

Overall, each microbenchmark highlights the design tradeoffs described in Section IV-B1, though secondary effects also impact execution time and network traffic. For each microbenchmark, we compare the effects of using a hierarchical vs. flat cache structure, using MESI vs. DeNovo at the CPU, and using GPU coherence vs. DeNovo at the GPU.

Indirection exhibits very little locality and involves high-bandwidth CPU-GPU communication. As a result, hierarchical configurations (HMG, HMD) exhibit significantly higher execution time and network traffic (HMG, HMD) because all CPU-GPU communication is routed through both a shared LLC and an intermediate GPU L2 cache.

Configurations that use DeNovo rather than MESI at the CPU cache exhibit less network traffic because DeNovo only obtains ownership for updated words in a line. When owned data is replaced in the CPU cache or requested by a remote GPU core, DeNovo caches only need to transfer the owned words while MESI caches transfer the full line.

Finally, configurations that use GPU coherence rather than DeNovo for GPU caches (HMG, SMG, SDG) very slightly outperform the corresponding configurations that use DeNovo for GPU caches (HMD, SMD, SDD) because DeNovo obtains

Benchmark Suite	Application	Communication Pattern				Execution Parameters
		Partitioning	Synchronization	Sharing	Locality	
Pannotia [9]	BC	data	fine-grain	flat	high	graph: olesnik [28] vertices: 88,263 edges: 243,088 CTs: 8, TBs: 64
	PR	data	coarse-grain	flat	moderate	graph: wing [28] vertices: 62,032 edges: 402,623 CTs: 8, TBs: 8
Chai [10]	HSTI	data	fine-grain	flat	data: low atomic: high	input size: 1,572,864 CTs: 4, TBs: 16
	TRNS	data	fine-grain	flat	low	input size: 64x4,096 CTs: 8, TBs: 8
	RSCT	task	fine-grain	hierarchical	data: high atomic: low	input size: 2,000x5,922x4 CTs: 1, TBs: 16
	TQH	task	fine-grain	hierarchical	data: low atomic: high	input size: 80x133x176 CTs: 1, TBs: 32

TABLE VII: Collaborative applications communication patterns and execution parameters. CTs = CPU threads. TBs = GPU Thread Blocks.

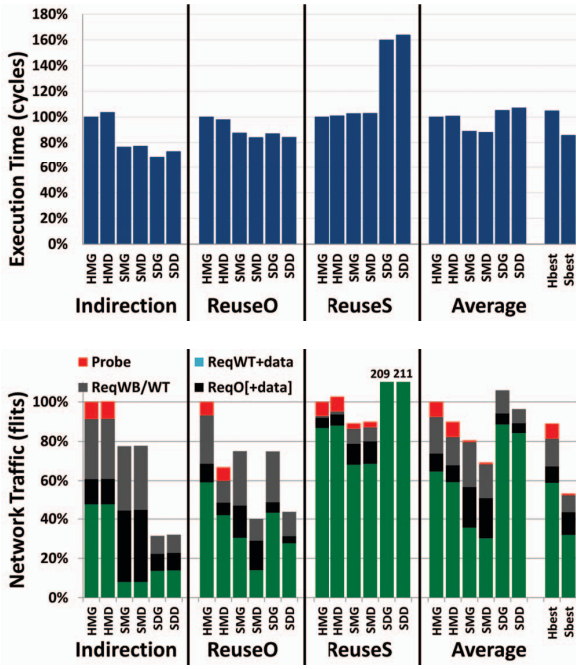


Fig. 2: Results for all synthetic microbenchmarks, normalized to HMG.

ownership for written data, adding latency when that data is accessed by a CPU core. However, most data produced by the GPU is evicted and written back to the LLC before it is next accessed by the CPU, which is why this has such a minimal effect on performance.

ReuseO exhibits more locality and less CPU-GPU communication than **Indirection**, but the costs of hierarchical indirection are still evident in hierarchical MESI configurations (HMG and HMD), which incur greater execution time and network traffic than Spandex configurations. Even though The choice of MESI or DeNovo at the CPU caches has a negligible effect on execution time and network traffic since both obtain ownership for the written data. However, **ReuseO** benefits from using DeNovo in GPU cores (HMD, SMD, SDD) because obtaining ownership for updates enables the GPU caches to exploit locality present in written data. The increased cache reuse leads to greatly reduced network traffic for configurations with DeNovo coherence at the GPU. It also slightly reduces in execution time, although this is minor because GPUs are relatively tolerant to memory latency.

ReuseS performance is largely unaffected by the indirection incurred by hierarchical configurations. Hierarchical MESI is able to avoid indirection overheads here because the intermediate L2 cache obtains Shared permission for reads and is therefore able to satisfy most GPU L1 misses at a similar cost to a flat Spandex LLC.

Performance for Reuse is much more dependent on the choice of protocol for the CPU cache. Configurations that use MESI at CPU caches (HMG, HMD, SMG, SMD) significantly reduce both execution time and network traffic relative to those that use DeNovo (SDG, SDD) due to the fact that DeNovo will self-invalidate the densely read data before it can be reused.

The choice of GPU coherence strategy has little effect on performance here; neither DeNovo nor GPU coherence support Shared state, so they are unable to exploit locality in the dense reads. In addition, the sparse data writes are not on the critical path and whether they obtain ownership or are written through has a negligible effect on subsequent read latency.

Across all microbenchmarks, the optimal Spandex configuration roughly matches or exceeds the optimal hierarchical configuration, reducing execution time and network traffic by an average 18% (max 31%) and 40% (max 69%), respectively. This demonstrates that Spandex can more flexibly adapt to diverse CPU-GPU sharing patterns than a hierarchical MESI-based configuration.

B. Collaborative Applications

The collaborative applications have a wide range of memory demands. The impact of each design dimension varies per workload, but the most consistent effect is that a flat Spandex LLC improves performance relative to hierarchical MESI.

BC and **PR** are data partitioned graph algorithms with a flat sharing pattern and irregular input-dependent accesses

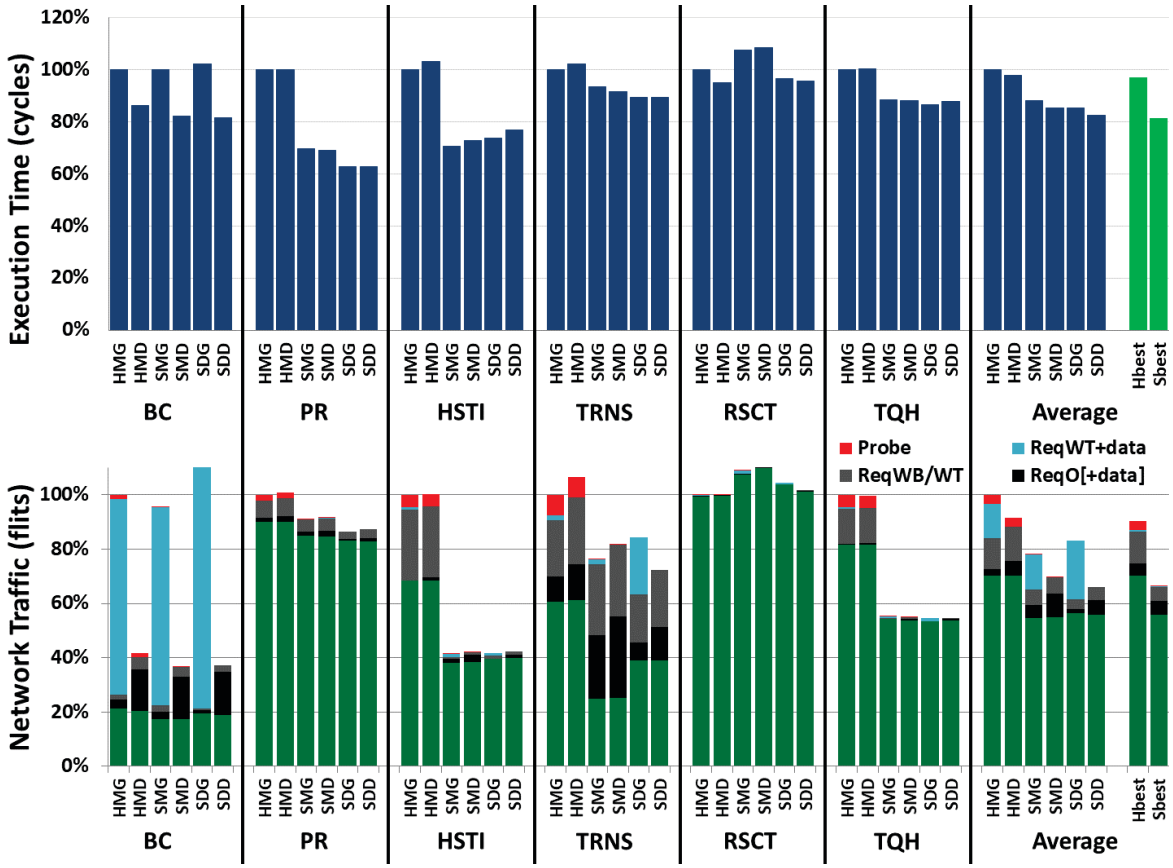


Fig. 3: Results for all applications, normalized to HMG.

to the graph array. **BC** accesses the graph structure using atomic updates, and these updates exhibit high temporal locality for the input studied. The most important design dimension for **BC** is whether GPU caches use DeNovo or GPU coherence. DeNovo's use of ownership enables GPU caches to exploit the high locality present in atomic accesses, drastically reducing execution time and network traffic relative to a similar configuration using GPU coherence. The performance impact of DeNovo GPU caches for **BC** is greater than in **ReuseO** because **BC** is able to exploit locality in atomic operations in an unbalanced workload, where they are more likely to be on the critical path. **BC**'s performance is not significantly affected by CPU or LLC protocol choice.

PR accesses the graph structure using data loads (rather than atomics), which exhibit moderate locality. Here the bottleneck is memory throughput, and the most important design dimension is whether flat Spandex or hierarchical MESI is used at the LLC. A Spandex LLC reduces latency and network traffic for GPU read accesses because it avoids an extra level of indirection and uses a simple last-level protocol. Unlike hierarchical MESI, Spandex can handle a GPU ReqV without allocating a cache line in an intermediate cache, without transitioning to a blocking state, without revoking ownership or triggering a write-back from the current owner, and without requiring sharer invalidation the next time that data is written. Similarly, configurations which use a DeNovo CPU perform slightly better than a MESI CPU for this workload because DeNovo reads do not incur blocking states or revoke ownership from the current owner.

HSTI and **TRNS** are data partitioned applications with limited data locality and frequent synchronization. The choice of LLC protocol has the most significant performance impact for these applications, and using a flat Spandex configuration reduces execution time and network traffic for both. This is largely due to the reduced indirection for data accesses, which have a relatively high miss rate for these workloads. In addition, frequent inter-device synchronization benefits from Spandex's non-blocking ownership transfer. **HSTI** atomics exhibit high spatial locality. As a result, using line granularity MESI rather than DeNovo at the CPU enables improved atomic reuse and slightly improved performance. **TRNS** atomics, on the other hand, exhibit low spatial locality, and word granularity DeNovo ownership is able to offer slightly better performance by avoiding false sharing.

RSCT and **TQH** are task partitioned algorithms which exhibit hierarchical sharing and fine-grain synchronization with low locality. In **RSCT**, all GPU cores access the same shared array. Hierarchical MESI configurations are able to exploit this sharing at the intermediate cache level, filtering GPU requests, reducing contention at the LLC, and improving

CPU performance. Additionally, **RSCT**'s low-locality atomic accesses benefit from DeNovo's word granularity ownership at the CPU and non-blocking ownership transfer at the Spandex LLC. Combining these effects, hierarchical MESI configurations (HMG, HMD) and Spandex configurations with DeNovo coherence at the CPU (HDG, HDD) both offer performance improvements relative to flat Spandex configurations with MESI CPU caches (SMG, SMD).

TQH exhibits some hierarchical sharing in atomic variables, but reads to disparate data sets dominate the access pattern, limiting available reuse or sharing. As with PR, HSTI, and TRNS, flat Spandex configurations reduce TQH execution time and network traffic relative to hierarchical MESI configurations by reducing indirection and blocking states for data accesses with limited locality.

Overall, the benefits of a flat Spandex configuration relative to hierarchical MESI tend to be the most prominent performance effect for the applications studied. Although applications like **RSCT** may benefit from a hierarchical cache structure due to its hierarchical sharing pattern, Spandex can also be made hierarchical to efficiently support such sharing patterns; the reverse is not true of a hierarchical MESI LLC and flat heterogeneous sharing patterns. On average, we find that the best Spandex configuration reduces execution time by 16% (max 29%) and network traffic by 27% (max 58%) relative to the best hierarchical MESI configuration.

VI. Related Work

Section II-D already described the most relevant technologies. Here we describe other related efforts to improve coherence efficiency and flexibility for emerging systems.

There are multiple ongoing efforts to define communication interfaces between devices in heterogeneous systems. However, at the time of writing, these specifications are either not publicly available (e.g., CCIX [29], Gen-Z [30]) or do not implement coherent caching for accelerators (e.g., OpenCAPI [31]).

Past work has implemented efficient CPU-GPU coherence, using both hardware and software techniques. QuickRelease (QR) [32], Heterogeneous System Coherence (HSC) [33], and the Fusion architecture [34] all use a clustered hierarchical cache structure and MESI-based last level directory to interface CPU and GPU devices. Software-managed page migration strategies that intelligently perform explicit page copies between CPU and GPU memories are effective for many dense, hierarchical CPU-GPU sharing patterns [35], [3], [36]. However, performance can suffer for new or irregular sharing patterns due to the high cost of inter-device communication.

Interfacing heterogeneous protocols can be avoided by using the same protocol at both CPU and GPU L1 caches. Both DeNovo for GPUs [6] and VIPs-G [37] adopt this approach. However, the design of an L1 interface is often tightly integrated with processor design, and requiring all cores to interface with a new cache protocol may be an unacceptable design burden.

Past work has also studied the benefits of coherence flexibility. Dynamic self invalidation (DSI) supports both writer-invalidated and self-invalidated reads at the LLC based on software annotations or hardware prediction [38]. Multiple studies have also examined the performance effects of state and communication granularity, and have proposed techniques for adapting these parameters to an executing workload [39], [40], [41], [42]. These works share Spandex's goal of adaptability but are less comprehensive, each focusing on only a single coherence design dimension in the context of multicore CPUs.

Crossing Guard offers a simple and stable coherence interface for accelerator caches [43]. However, Crossing Guard's primary goal is correctness and security in heterogeneous coherence, not improved performance. It also only interfaces with MESI and MOESI style caches. Thus it is unclear whether accelerator caches that prefer simpler protocols such as GPU coherence or DeNovo would be able to efficiently interface with the system.

Manager-client pairing (MCP) is a framework for defining hierarchical coherence protocols in large-scale systems [44]. Like Spandex, MCP addresses the complexity of hierarchical protocols and evaluates the costs of deeper cache organizations. However, MCP is focused on providing a generic framework for building hierarchical protocols for devices that request read and write permission. Spandex instead aims to avoid the need for hierarchical organization for a wide variety of heterogeneous devices, some of which use self-invalidations or write-through caches instead of requesting read/write permissions, and which can use variable request granularity.

Past work has also addressed the challenge of enforcing memory consistency in heterogeneous systems. ArMOR provides a framework for precisely defining the ordering requirements of different consistency models and ensures that devices with different memory models respect ordering constraints in a heterogeneous system [21]. COATCheck offers a framework for specifying and verifying memory consistency ordering constraints, considering both microarchitectural and operating system concerns in the design of a coherent device [45]. These techniques may be used in conjunction with Spandex when designing and integrating new device types.

VII. Conclusions and Future Work

Tightly coupled specialized hardware has become an important driver of performance in many compute domains. Existing strategies for heterogeneous coherence tend to deal with the diverse memory demands of accelerator workloads typically by adding additional cache layers and/or protocol complexity to an already complex MESI-based protocol. This work defines Spandex, a heterogeneous coherence interface designed to be flexible and simple. By directly interfacing devices that prefer self-invalidated or writer-invalidated loads, owned or write-through stores, and coarse or fine-grain state tracking and communication, Spandex enables CPU, GPU, and other accelerators to efficiently use whichever coherence strategy is most appropriate for the executed workload.

Furthermore, the flexibility and simplicity of the Spandex LLC protocol has performance implications beyond the integration of existing devices. Although not explored in this work, additional optimizations can be fairly easily implemented if device memories are designed with Spandex in mind. If hints about locality and contention can be provided by software or inferred in hardware, devices could dynamically choose different Spandex request types for different memory accesses in a single program, offering fine-grain coherence

specialization. The Spandex LLC could be extended to exploit dataflow information by forwarding ReqWT or ReqWT+data requests from a writer (producer) device to an owning reader (consumer) device, enabling in-place data updates without affecting LLC state. Furthermore, if the current owner can be predicted, Spandex requests that don't update LLC state may be speculatively sent directly to the expected owner of target data, avoiding the overheads of LLC lookups. Although we have discussed only word vs. line granularity, Spandex may operate at other software-driven data structure specific granularities as well that may be more or less than a typical cache line. While several of the above optimizations have been considered in the past (e.g., for DeNovo [14]), Spandex provides the opportunity to cleanly integrate them within a system with diverse devices.

To demonstrate the benefits of Spandex simplicity and flexibility for CPU-GPU coherence, we evaluated it against a less flexible and more complex hierarchical cache structure with a MESI LLC. We show that an ideally configured Spandex interface is able to reduce execution time and network traffic by on average 16% (max 29%) and 27% (max 58%), respectively, for a diverse range of collaborative CPU-GPU applications when compared with an ideally configured hierarchical MESI coherence interface. These benefits make Spandex a uniquely suitable coherence strategy for the exceedingly diverse workloads of existing, emerging, and future specialized devices.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants CCF 13-02641 and CCF 16-19245, and by the Center for Future Architectures Research (C-FAR) and the Applications Driving Architectures (ADA) center, Semiconductor Research Corporation programs sponsored by MARCO and DARPA. We also thank Weon Taek Na for his valuable contributions to the Spandex evaluation.

REFERENCES

[1] B. Munger, D. Akeson, *et al.*, "Carrizo: A high performance, energy efficient 28 nm apu," *JSSC*, vol. 51, no. 1, pp. 105–116, 2016.
[2] I. Bratt, "The ARM® Mali-T880 Mobile GPU," in *IEEE Hot Chips 27 Symposium*, pp. 1–27, 2015.
[3] N. Sakharnykh, "Beyond GPU Memory Limits with Unified Memory on Pascal." https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/, 2016.
[4] J. Stuecheli, B. Blaner, *et al.*, "CAPI: A Coherent Accelerator Processor Interface," *IBM JRD*, vol. 59, no. 1, pp. 7–1, 2015.
[5] I. Singh, A. Shriraman, *et al.*, "Cache Coherence for GPU Architectures," in *HPCA*, 2013.
[6] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models," in *MICRO*, pp. 647–659, 2015.
[7] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems," in *ISCA*, pp. 161–174, 2017.
[8] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, pp. 141–151, 2012.
[9] S. Che, B. Beckmann, *et al.*, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *IISWC*, 2013.
[10] J. Gómez-Luna, I. El Hajj, *et al.*, "Chai: Collaborative Heterogeneous Applications for Integrated-Architectures," in *ISPASS*, pp. 43–54, 2017.
[11] J. Y. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *MICRO*, pp. 75–87, 2014.
[12] M. S. Orr, S. Che, *et al.*, "Synchronization Using Remote-Scope Promotion," in *ASPLOS*, pp. 73–86, 2015.
[13] M. D. Sinclair, J. Alsop, and S. V. Adve, "HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs," in *IISWC*, 2017.
[14] B. Choi, R. Komuravelli, *et al.*, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *PACT*, pp. 155–166, 2011.
[15] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-determinism," in *ASPLOS*, 2013.
[16] H. Sung and S. V. Adve, "DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations," in *ASPLOS*, pp. 545–559, 2015.
[17] B. M. Beckmann and A. Gutierrez, "The AMD gem5 APU Simulator: Modeling Heterogeneous Systems in gem5," in *MICRO Tutorial*, 2015.
[18] ARM, "AMBA AXI and ACE protocol specification." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html, 2017.
[19] D. R. Hower, B. A. Hechtman, *et al.*, "Heterogeneous-Race-Free Memory Models," in *ASPLOS*, pp. 427–440, 2014.
[20] J. Alsop, M. S. Orr, *et al.*, "Lazy Release Consistency for GPUs," in *MICRO*, pp. 1–14, 2016.
[21] D. Lustig, C. Trippel, *et al.*, "ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures," in *ISCA*, pp. 388–400, 2015.
[22] S. Adve and M. Hill, "Weak Ordering – A New Definition," in *ISCA*, 1990.
[23] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models," *TACO*, vol. 12, pp. 7:1–7:26, April 2015.
[24] P. S. Magnusson, M. Christensson, *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.
[25] M. M. K. Martin, D. J. Sorin, *et al.*, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, 2005.
[26] A. Bakhoda, G. L. Yuan, *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, pp. 163–174, 2009.
[27] N. Agarwal, T. Krishna, *et al.*, "GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator," in *ISPASS*, pp. 33–42, 2009.
[28] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, pp. 1:1–1:25, Dec. 2011.
[29] "Cache Coherent Interconnect for Accelerators (CCIX)." http://www.ccixconsortium.com, 2017.
[30] "Welcome to The Gen-Z Consortium!." http://genzconsortium.org, 2017.
[31] "Welcome to OpenCAPI Consortium." http://www.opencapi.org, 2017.
[32] B. Hechtman, S. Che, *et al.*, "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs," in *HPCA*, 2014.
[33] J. Power, A. Basu, *et al.*, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *MICRO*, pp. 457–467, 2013.
[34] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *ISCA*, 2015.
[35] D. Negrut, R. Serban, *et al.*, "Unified Memory in CUDA 6.0: A Brief Overview of Related Data Access and Transfer Issues," tech. rep., University of Wisconsin-Madison, 2014.
[36] N. Agarwal, D. Nellans, *et al.*, "Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence," in *HPCA*, pp. 494–506, 2016.
[37] K. Koukos, A. Ros, *et al.*, "Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead," *TACO*, vol. 13, no. 1, 2016.
[38] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *ISCA*, pp. 48–59, 1995.
[39] J. Torrellas, H. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *TOCS*, vol. 43, no. 6, 1994.
[40] H. Zhao, A. Shriraman, *et al.*, "Protozoa: Adaptive Granularity Cache Coherence," in *ISCA*, pp. 547–558, 2013.
[41] J. F. Cantin, J. E. Smith, *et al.*, "Coarse-grain Coherence Tracking: RegionScout and Region Coherence Arrays," *IEEE Micro*, 2006.
[42] J. B. Rothman and A. J. Smith, "Sector Cache Design and Performance," in *ISMASCTS*, pp. 124–133, 2000.
[43] L. E. Olson, M. D. Hill, and D. A. Wood, "Crossing Guard: Mediating Host-Accelerator Coherence Interactions," in *ASPLOS*, 2017.
[44] J. G. Beu, M. C. Rosier, and T. M. Conte, "Manager-Client Pairing: A Framework for Implementing Coherence Hierarchies," in *MICRO*, pp. 226–236, 2011.
[45] D. Lustig, G. Sethi, *et al.*, "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface," in *ASPLOS*, pp. 233–247, 2016.