A Case for Fine-grain Coherence Specialization in Heterogeneous Systems

JOHNATHAN ALSOP, AMD Research, USA
WEON TAEK NA, MIT, USA
MATTHEW D. SINCLAIR, University of Wisconsin - Madison, USA
SAMUEL GRAYSON and SARITA ADVE, University of Illinois at Urbana-Champaign, USA

Hardware specialization is becoming a key enabler of energy-efficient performance. Future systems will be increasingly heterogeneous, integrating multiple specialized and programmable accelerators, each with different memory demands. Traditionally, communication between accelerators has been inefficient, typically orchestrated through explicit DMA transfers between different address spaces. More recently, industry has proposed unified coherent memory which enables implicit data movement and more data reuse, but often these interfaces limit the coherence flexibility available to heterogeneous systems.

This paper demonstrates the benefits of fine-grained coherence specialization for heterogeneous systems. We propose an architecture that enables low-complexity independent specialization of each individual coherence request in heterogeneous workloads by building upon a simple and flexible baseline coherence interface, Spandex. We then describe how to optimize individual memory requests to improve cache reuse and performance-critical memory latency in emerging heterogeneous workloads. Collectively, our techniques enable significant gains, reducing execution time by up to 61% or network traffic by up to 99% while adding minimal complexity to the Spandex protocol.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Heterogeneous (hybrid) systems**; • **Hardware** → **Emerging architectures**; **Emerging interfaces;**

Additional Key Words and Phrases: Shared memory systems, coherence, GPUs, caches

ACM Reference format:
Johnathan Alsop, Weon Taek Na, Matthew D. Sinclair, Samuel Grayson, and Sarita Adve. 2022. A Case for Fine-grain Coherence Specialization in Heterogeneous Systems. *ACM Trans. Archit. Code Optim.* 19, 3, Article 41 (August 2022), 26 pages.
https://doi.org/10.1145/3530819

This work is supported in part by the National Science Foundation under grant CCF 16-19245, by DARPA through the Domain-Specific System on Chip (DSSoC) program, a Google Faculty Research award, and by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Authors' addresses: J. Alsop, AMD Research 2002 156th Ave NE #300 Bellevue, WA 98007; email: johnathan.alsop@amd.com; W. T. Na, MIT CSAIL 32 Vassar Street, 32-G786 Cambridge, MA 02139; email: weontaek@mit.edu; M. D. Sinclair, University of Wisconsin-Madison Computer Sciences Department 1210 West Dayton Street Madison, WI 53706; email: sinclair@cs.wisc.edu; S. Grayson and S. Adve, Siebel Center for Computer Science University of Illinois at Urbana-Champaign 201 North Goodwin Avenue Urbana, IL 61801-2302; emails: {grayson5, sadve}@illinois.edu.

1544-3566/2022/08-ART41
https://doi.org/10.1145/3530819

1 INTRODUCTION

In nearly all compute domains, hardware specialization and parallelism have become important drivers of compute performance. GPUs, FPGAs, and other specialized devices are being incorporated into systems ranging from mobile and IoT devices to supercomputers and datacenters. However, the conventional SoC accelerator integration process is inefficient and complex. Communication with a new accelerator often uses explicit copies to and from DRAM. This enables simple integration and the use of specialized memories for each accelerator, but it prevents implicit reuse, limits programmability, and makes inter-device communication inefficient. As accelerators become a more important part of emerging applications and the memory wall increases the importance of efficient cache reuse, coherence specialization is becoming just as important to performance as compute specialization.

In order to efficiently adapt to emerging workloads and systems, mechanisms for coherence optimization must be both flexible and simple. There have been many coherence extensions proposed over the years (discussed further in Section 2), but these generally build upon conventional hardware protocols originally designed for CPUs such as MESI. Such protocols are effective for a wide range of CPU workloads, but these complex coherence strategies often incur unacceptable overheads for accelerators such as GPUs [35, 36, 78]. In addition, the complexity of MESI-based protocols makes validating protocol changes expensive, requiring that the cost of any coherence extension be amortized over a broad range of general-purpose applications. A simpler baseline protocol would make it practical to validate coherence extensions for an individual application domain or class of accelerators.

Recently, the Spandex protocol was proposed as a simple and flexible alternative to MESI-based coherence for heterogeneous systems [9]. Spandex is based on the DeNovo protocol [26], and is designed to simply and flexibly interface a diverse set of devices (CPU cores, GPUs, and accelerators), each with different static coherence strategies (e.g., MESI, GPU, or DeNovo coherence protocols). Spandex achieves this by identifying an interface consisting of a set of coherence request types that represent the diversity of the above protocols. Spandex avoids many of the pitfalls of a complex MESI-based approach, but it misses out on an entire dimension of coherence flexibility: the ability to mix and match specialized coherence request types from a device based on the properties of each individual access of that device.

In this work we advocate for fine-grain coherence flexibility in heterogeneous memory systems. By choosing a Spandex request type used for every access individually rather than at device granularity, we can motivate new coherence specializations that increase cache reuse, reduce data movement, and favor critical accesses over non-critical accesses. This flexibility motivates two coherence optimizations: update propagation and owner prediction. We show that these can be added to the simple Spandex protocol while avoiding much of the complexity required to extend conventional MESI-based protocols. These specialized coherence policies and coherence optimizations have different tradeoffs, and identifying which policy is best for each individual access is not trivial. We therefore propose a trace-based technique for automatically determining the best specialized request type to be used for each memory access, and we show that this can improve cache efficiency relative to device granularity coherence specialization for common access patterns.

Overall, we make the following contributions:

- We propose specializing coherence request types at the granularity of an individual access, rather than at the granularity of a device. In our initial fine-grain coherence specialized system, we enable, within a device, all request types supported by Spandex.
- We describe two opportunities for further coherence specialization (update forwarding and owner prediction) that can be used to improve cache performance for certain access patterns when fine-grained coherence specialization is enabled.

- We quantitatively show that adding the new specializations on top of Spandex is 2× less complex than adding them to a more conventional MESI-based protocol.
- To efficiently leverage our fine-grained coherence specializations, we describe a heuristic algorithm for offline request type selection that aims to maximize cache reuse in common heterogeneous applications.
- We evaluate fine-grain specialized coherence on a range of heterogeneous workloads with diverse access patterns and show that it can reduce execution time by up to 61% and network traffic by up to 99%, while also motivating programming patterns for heterogeneous systems that would otherwise be inefficient.

With fine-grain coherence specialization, we show that it is possible to significantly increase cache reuse, reduce data movement, and improve performance in heterogeneous applications, while keeping design complexity under control.

2 RELATED WORK

There are multiple active efforts to design coherence for heterogeneous systems [2, 12, 22, 63, 67, 84]. Many of these offer a high degree of flexibility, enabling "non-snoopable" regions and request types (similar to self-invalidated loads and write-through stores). Some interfaces also offer mechanisms for pushing writes directly to a designated node (e.g., cache stashing in ARM CHI), similar to write-through forwarding optimizations described in Section 4. However, in many cases these properties are fixed for a given address region, they incur added complexity in the form of transient states, they require indirection through the LLC, and they are generally not focused on request type optimization at the level of individual accesses, so they are more complex and less flexible than the coherence specialization described here.

Past work has studied the benefits of adaptive cache policies [49, 50, 66] and of flexible cache state or coherence granularities [20, 26, 30, 47, 74, 86, 87, 95]. Protocol extensions have also been proposed to detect and predict specific sharing patterns, optimizing requests to reduce wasteful traffic and latency for those patterns [3, 11, 19, 25, 29, 38, 40, 41, 46, 48, 58, 59, 62, 70, 72, 73, 79, 82]. All of the above innovations help to motivate and demonstrate the benefits of dynamic adaptability, and the methods they propose for detecting and predicting sharing patterns will be increasingly relevant to specialized memory systems. In fact, the two optimizations described in Section 4.2 (update forwarding and owner prediction) are very similar to prior optimizations [5]. However, all of these past protocol innovations focus on optimizing communication patterns in the context of multicore CPU workloads rather than heterogeneous workloads, and the benefits they provide often come at the cost of additional overhead and complexity on top of an already complex MESI-based protocol.

A wide range of past work explores allowing software to more explicitly control data movement and data locality ([16, 33, 34, 37, 39, 53, 93]). These data movement optimizations tend to involve simple hardware changes which minimally affect the coherence protocol, but they do not provide the dynamic flexibility enabled by coherence specialization at the granularity of individual accesses.

Many studies also have proposed ways to improve memory access efficiency through adaptive cache management (e.g., replacement, prefetching, and bypassing) in both CPUs and GPUs based on hints from the software, compiler, or runtime profilers [13, 17, 24, 45, 54, 68, 76, 85, 90, 91]. These techniques allow caches to adapt to different access patterns and improve cache efficiency, and many of the insights regarding cache policy prediction can be leveraged to help select request types in a system with fine-grain coherence specialization. However, their flexibility is constrained to cache management policies rather than coherence protocol flexibility.

Recent work proposes communicating high level properties of a program's memory access pattern to the underlying hardware to guide architectural optimizations such as cache policy specialization, intelligent data mapping in DRAM and intelligent GPU work distribution [88, 89]. These techniques are compatible with a Spandex system, and could take advantage of fine-grain coherence specialization by guiding request type selection.

Recent work has improved utilization, load balance, and data reuse by optimizing the scheduling and mapping computation tasks to compute devices in dataflow workloads. These efforts show significant gains for a range of CPU [94] and GPU/accelerator [8, 55, 81, 96] applications. These software frameworks rely on an awareness of reuse potential, parallelism, data dependencies, and producer-consumer relationships to enable more data-efficient compute. This awareness makes them well suited to exploit fine-grain coherence flexibility to further improve reuse or reduce wasteful communication overheads in emerging workloads.

Some recent work identifies the optimal cache coherence policy for different combinations of composable accelerators, easing the process of heterogeneous system design [18, 57]. However, flexibility is limited to a single coherence decision for each kernel, limiting the types of possible optimizations.

Overall, the adaptivity innovations of past work complement the insights of this work, and can be used to guide request type decisions and coherence extensions for future heterogeneous systems and workloads.

3 BACKGROUND: HETEROGENEOUS COHERENCE

Designing coherence protocols for heterogeneous systems is complicated by the diverse memory demands of different devices. Any coherence strategy involves tradeoffs in how it invalidates stale data, propagates updates, and manages state in order to exploit different types of cache locality while enforcing coherence. We next describe these tradeoffs in more detail for three coherence strategies, and we discuss how the Spandex protocol can be used to interface devices that use each request type. Although the following discussion considers a system with a shared **last level cache (LLC)**, the same tradeoffs apply to a stateless LLC.

3.1 MESI-based Coherence

MESI-based protocols are great at exploiting cache reuse, but they require coherence overheads which can limit performance for workloads with low locality. Because of this they are a good fit for CPUs, which tend to exhibit high sensitivity to memory latency but have low memory throughput demands. For simplicity, we refer to MESI-based protocols as "MESI" going forward, but the below tradeoffs apply to MOESI, MESIF, and most other common MESI variants.

Stale data invalidation: MESI uses writer invalidation, which means that reads obtain sharer permissions and writes must revoke these permissions from any sharer caches. Writer-invalidation allow readers to cache data until it is known to be stale. However, it also requires storage and network overheads related to tracking and invalidating sharer state.

Update propagation: MESI caches obtain exclusive ownership (i.e., writer permission) for writes and atomic RMW operations. This enables the writer cache to exploit locality in written data and avoid transfer of dirty data until absolutely necessary (the data is evicted, or it is read by another device). However, it also adds indirection latency for subsequent requests from remote devices because the ID of the owning device must be queried at the LLC and the request must then be forwarded to the owner.

Request granularity: MESI tracks state and transfers data at line granularity in order to exploit spatial locality and reduce the overheads involved with tracking sharer and writer permissions. The disadvantages of this policy include increased false sharing (accesses to different addresses in the same line may cause unnecessary state/data transfer) and a read-for-ownership requirement (writes that only update part of a cache line must also read the unmodified data before the line can be considered valid).

3.2 GPU Coherence

Relative to latency-sensitive CPUs, GPU workloads are generally more sensitive to memory throughput. To avoid the overheads of MESI, GPUs tend to use a simpler, more scalable coherence protocol, which we refer to as *GPU coherence* [77]. GPU coherence is not as good as MESI at exploiting cache reuse, but it can allow for much better performance when available locality is low and throughput demand is high.

Stale data invalidation: GPU coherence uses self-invalidation to prevent software from reading stale data. Often this is implemented via bulk cache invalidates that are triggered at synchronization points (e.g., a kernel boundary, or an operation with acquire semantics). Since in a DRF system conflicting accesses from different devices must be separated by synchronization, this is sufficient to avoid stale reads, and it avoids the overheads of sharer tracking, sharer invalidation, and false sharing. However, such a conservative strategy can also invalidate data that is not stale, leading to poor performance if synchronization is frequent.

Update propagation: To ensure updated data is visible to other devices, GPU coherence uses write-through caches for writes and atomics. This means, rather than requesting write permission, dirty data is written back to the LLC (or performed at the LLC in the case of atomics) before it is accessed by a remote device. Similar to bulk invalidates, non-synchronizing writes are often written to the LLC lazily via a bulk flush of dirty cached data, and in a DRF system this only needs to happen at synchronization points (kernel boundaries or operations with release semantics). This strategy avoids the overheads of obtaining and tracking exclusive owner permissions, but it also can lead to excessive data movement and reduced reuse in written data if dirty flushes are frequent.

Request granularity: GPU coherence uses line granularity for reads and word (or byte) granularity for writes and atomics. This allows it to exploit spatial locality for reads, avoid false sharing, and avoid read-for-ownership overhead for writes. It also requires a bitmask in store requests to indicate which words are being written, as well as in cache state (for write-combining caches) to indicate which words are dirty (state overhead is minimal for most cache sizes).

3.3 DeNovo Coherence

The DeNovo coherence protocol combines elements of both MESI and GPU protocols to offer an efficient hybrid protocol appropriate for CPUs [26] and GPUs [77].

Stale data invalidation: Like GPU coherence, DeNovo uses self-invalidation to prevent software from reading stale data. This allows it to avoid false sharing, as well as sharer tracking and invalidation overheads.

Update propagation: Like MESI, DeNovo obtains ownership for writes and atomics. Although this incurs some storage and indirection overheads, it allows caches to exploit locality in written data.

Request granularity: DeNovo uses word granularity for all request types, requiring bitmasks in requests and caches. However, a load response will supply all data in the target cache line that is owned by the satisfying device/cache level, which allows DeNovo to take advantage of spatial locality in reads when it is convenient. Ownership is tracked at word granularity, and the ID of the owning cache for each word is stored in the data line at the LLC. Word granularity ownership avoids the transient states required by MESI because ownership requests do not need up-to-date

Table 1. Coherence Request Type Classification and What They are Used for in a Static DeNovo, GPU Coherence (GPUc), or MESI Protocol

Coherence Dimension	**Options**	**Request Type**	**Used For**
Stale Data Invalidation	Self-inval.	ReqV	DeNovo (LD) GPUc (LD)
		ReqVo	**FCS(LD)**
	Writer-inval.	ReqS	MESI (LD)
Update Propagation	Ownership	ReqO	DeNovo (ST)
		ReqO+data	MESI (ST,RMW) DeNovo (RMW) **FCS(LD)**
	Write-through	ReqWT**[fwd]**	GPUc (ST) **FCS(ST)**
		ReqWTo	**FCS(ST)**
		ReqWT**[fwd]**+data	GPUc (RMW) **FCS(RMW)**
		ReqWTo+data	**FCS (RMW)**
Request Granularity	Word	all	DeNovo (all) GPUc (ST,RMW)
	Line	all	MESI (all) GPUc (LD)

Bolded text represents coherence flexibility added by fine-grain coherence specialization (FCS).

data, but it also requires additional cache controller logic for handling per-word state transitions. However, these vectorized operations incur relatively less complexity and performance overhead than the transient states it avoids.

3.4 The Spandex Interface

Spandex was proposed as a way to flexibly interface devices with disparate coherence protocols (such as those described above) by providing request type flexibility in stale data invalidation, update propagation, and request granularity [9]. It allows for each device to use writer invalidation or self invalidation, ownership for writes or write-through caches, and word or line granularity requests based on the properties of the device, and it handles the data movement and state transitions necessary when communicating with devices that prefer different coherence strategies.

4 FINE-GRAIN COHERENCE SPECIALIZATION

4.1 Mixing and Matching Coherence Request Types

While Spandex is an efficient interface for enabling coherence specialization at device granularity, in order to further optimize data movement in emerging systems we propose specializing coherence at the granularity of an individual request. Table 1 associates the coherence dimensions discussed in Section 3 with the corresponding Spandex request types and highlights which requests are used by MESI, GPU coherence, and DeNovo devices. As described in Section 3, DeNovo and GPU coherence use self invalidation for reads (ReqV), MESI uses writer invalidation for reads (ReqS), DeNovo and MESI obtain ownership for writes and atomics (ReqO[+data]), and GPU coherence uses write-through for writes and atomics (ReqWT[+data]). The "+data" notation associated with ReqO and ReqWT operations indicates that up-to-date data is needed as well (e.g., for read-for-ownership or atomics that operate on the prior value). Table 1 also includes new request types that are motivated by fine-grain coherence specialization, highlighted in bold. Section 4.2 describes these new request types. Section 4.3 demonstrates how these optimizations can be implemented with minimal complexity overhead as long as a simple protocol baseline (e.g., Spandex) is selected. Section 4.4 presents a method for selecting request types at the granularity of a single static instruction for individual requests in a parallel heterogeneous workload. Sections 6–7 demonstrate how these optimizations can improve performance for a range of workloads and access patterns.

4.2 New Coherence Request Types

Two coherence optimizations that become feasible with FCS are forwarded update propagation and destination owner prediction. We apply these optimizations on top of the baseline Spandex protocol (new request types shown in bold in Table 1). In designing these coherence optimizations, we require that new request types do not affect the consistency model (we assume DRF) because using a different type of request type should only impact performance, never functionality.

4.2.1 Forwarded Update Propagation (ReqWTfwd[+data]). The first coherence optimization adds a new type of update propagation method: forwarded write-through stores and RMWs (ReqWTfwd, ReqWTfwd+data). When such a request arrives at the LLC for data that is owned by a different core, it will be forwarded to the current owner and its update operation will be performed without affecting coherence state at the LLC or the current owner. If the target data is not remotely owned, this request is treated the same as a normal ReqWT/ReqWT+data request.

Write-through forwarding is useful when the consumer of a data update is expected to currently own it and the writer is not expected to exhibit more reuse than the consumer. This optimization motivates a new fine-grain specialized access pattern. By using ReqO+data for the consumer access (load or RMW) in a producer-consumer pair and ReqWTfwd[+data] for the producer access (store or RMW), the producer update will be forwarded to the owner and enable the performance-critical consumer access to exploit reuse that was previously infeasible. Similarly, if the consumer and producer are atomic operations (e.g., an acquire and a release), they can use ReqO+data and ReqWTfwd+data, respectively. This is an insight that has been exploited in the past, often in the context of update-based coherence. However, this optimization avoids the complex write atomicity challenges that plague the design of such protocols [80]. This is because write-through forwarding only sends an update to a single exclusive owner core rather than all sharer cores, ensuring a single point of modification which serializes subsequent requests.

4.2.2 Destination Owner Prediction (ReqVo, ReqWTo). The second optimization extends the interface for self-invalidated reads (ReqV) and forwarded write-through and RMW requests (ReqWTfwd[+data]) to allow a device to predict the current owner of target data and send a request directly to that core. This effectively adds three new request types to the coherence interface: ReqVo, ReqWTo, and ReqWTo+data. Once issued, these requests are handled the same as their root request types, except that they are sent directly to the expected owner core rather than the LLC (determined by an implementation-specific prediction mechanism).[1] This is a simple extension in Spandex because, unlike read and write requests in most protocols, self-invalidated reads and forwarded write-through requests do not affect the LLC's coherence state. Further, in the same way that a forwarded ReqV or ReqWTfwd request may fail and trigger a retry in Spandex, an incorrect owner prediction will simply trigger a retry with a non-forwarded (ReqWT or ReqO+data) request to the LLC. Therefore, mispredicting an owner may hurt performance, but it does not affect protocol correctness.

[1]A similar ownership prediction mechanism was considered as an extension to the DeNovo protocol [43], although the details were not discussed and the mechanism was not evaluated.

If the underlying hardware supports owner prediction and the current owner can be accurately predicted, then using the owner prediction request types to directly forward read or write requests avoids the latency, traffic, and energy overheads of looking up the owner at the LLC. Unlike write-through forwarding, this optimization requires an effective prediction mechanism to be worthwhile. In this work we find that a simple hardware prediction mechanism is sufficient to improve performance for some common workloads. Our evaluation assumes a small prediction table is stored at each core which tracks the core ID of the most recent responder, indexed by target address and request type. This table is queried for every ReqVo and ReqWTo request and updated for every ReqVo and ReqWTo response. As programming languages become more expressive and hardware-software co-design becomes more prevalent, software-provided ownership hints and finely-tuned hardware prediction techniques may be used to improve upon this strategy.

4.2.3 Correctness Considerations. Although the described optimizations affect both the cache coherence protocol and the memory request interface, the memory consistency model is unchanged. In particular, previous work [9] has described how Spandex is directly compatible with the **data-race-free (DRF)** class of models which give **sequential consistency (SC)** to DRF programs [6]. DRF is the foundation of most programming languages models (e.g., C, C++, and Java) and hardware models (e.g., CPU models such as from Arm and RISC V and GPU models such as from AMD and NVIDIA). For a Spandex ReqV request, the core issuing it must take responsibility to invalidate any loaded data in its private cache hierarchy before the data becomes stale (due to a remote write) and is accessed again – with the DRF class of models, this is typically achieved by invalidating the cache at an acquire operation. Analogously, for a Spandex ReqWT request, the new data must be written through to the next shared cache level before it is accessed by a remote core that shares that cache – with the DRF class of models, this is typically achieved by a dirty flush triggered on a release operation. If the above invalidation and flush operations are not supported by a private cache, then that core cannot use ReqV or ReqWT access types. These requirements are similar to modern GPU actions on acquires and releases. Our optimizations of adding forwarding and destination prediction do not fundamentally change these requirements.

4.3 Complexity Analysis

Coherence extensions (e.g., Section 4) are uncommon in practice primarily because they are difficult to design and validate when added to conventional protocols like MESI. This is largely due to the rapidly growing state space that occurs when adding new transitions and request types to an already complex protocol [44].

By building upon the simple and flexible DeNovo protocol, Spandex is easier to extend and optimize. DeNovo relies on word granularity state and a DRF memory model to avoid transient blocking states [44]. Murϕ is a model checking tool that takes a coherence protocol description and a set of system constraints and enumerates all reachable state vectors [31]. The state vector includes the coherence state at a single controller, the state of data in each cache, and any pending or in-flight coherence requests. Although the number of reachable state vectors in Murϕ is not a perfect measure of protocol complexity, we use it because it captures the state space increase of handling transient states in a multi-cache environment. The state space size reflects the number of corner cases that must be tested when validating a coherence protocol change.

We explore the Spandex protocols state space in Murϕ and compare it against a MESI protocol [60] extended to implement a simplified version of the non-snoopable request types similar to those of the ARM CHI [12] interface (discussed more in Section 2). We use a flat cache organization instead of a hierarchical one [15] because exhaustively exploring interactions between all devices in a multi-level protocol requires a prohibitively large state space. Additionally, although

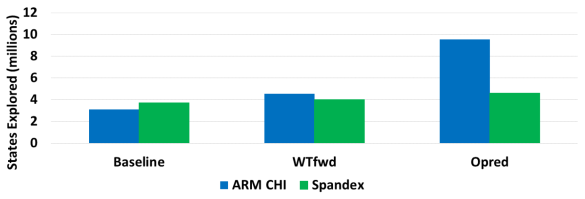


Fig. 1. Comparing Spandex and ARM CHI's state spaces as optimizations are added.

Spandex supports variable request granularity, we do not explicitly model this in Murϕ and instead assume a fixed word granularity. Spandex is expected to handle each word in a multi-word request individually based on its state. Therefore, we contend that adding support for coarse-grained or multi-word request types does not add any fundamental complexity to the protocol and would unfairly increase the state space in a Murϕ model. Each of these protocols is then further extended with the write-through forwarding (ReqWTfwd) and owner prediction (ReqVo, ReqWTo) optimizations discussed in Section 4.2.

Figure 1 plots the number of states explored by Murϕ for each protocol. Interestingly, the baseline Spandex protocol has slightly more states than the ARM CHI model because Spandex offers more flexibility. Where CHI has two types of load and store requests (snoopable and non-snoopable), Spandex allows the programmer to specify whether or not a store requires up-to-date data as well, and thus requires considering two more L1-initiated request types that may be pending at the L1 or LLC at any time. This added flexibility allows a device to avoid false sharing and transient states that are difficult to avoid in fixed granularity protocols.

However, as optimizations are added, the complexity of the MESI-based CHI coherence increases. Compared with Spandex, which applies optimizations to non-blocking request types (ReqV, ReqWT), a MESI-based protocol must consider many corner case transient states for racy accesses. Thus, Murϕ explores 1.1× more states in ARM CHI with write-through forwarding added to both protocols, and 2.1× more states with owner prediction added. This additional complexity stems from CHI being based on a line-granularity read-for-ownership protocol, and this prioritization is reflected in the state transitions. Every cache miss causes a transition to a transient state that blocks subsequent requests to the same cache block until some satisfying set of messages is received to transition back to a stable state. In contrast, Spandex builds upon DeNovo (which has no transient states) and avoids blocking states as much as possible. Thus, Spandex avoids much of the state space explosion that occurs when adding optimizations to a protocol with excessive transient states.

4.4 Request Type Selection: Overview

In a system with fine-grain specialization, request type selection could be implemented in hardware (typically at the granularity of a dynamic instruction), software (typically at the granularity of a static instruction), or a combination of both. In this work we assume software determines (and encodes) a request type for each static instruction at compile time. This requires an extension to the ISA. For example, if RISC-V is assumed as a baseline, each of the base load/store/RMW instructions can be mapped to a default request type, and a custom opcode may be allocated for each alternate type that may be used. Additional optimizations (forwarding, prediction) could be specified via bits in the added instructions, and masked RISC-V vector instructions could be used for word-granularity requests.[2]

[2]Other work has in fact implemented these request types in RISC-V (https://github.com/zzhu35/spandex-caches).

We use (offline) analysis on an execution trace to develop (and implement) a heuristic algorithm for determining appropriate request types for each static memory instruction.[3] We focus on offline trace-based request selection in this work because it gives us global visibility of cache locality and reuse potential. This works well for regular workloads with static dataflow. For irregular workloads, it may be preferable to dynamically choose request types at runtime based on software and/or hardware hints. We leave such methods for future work.

We start by generating a memory trace of a **sequentially consistent (SC)** dynamic execution of the program.[4] We apply Algorithms 1–3 to each individual word-granularity memory access in the trace.[5] For instructions that access multiple words, request selection treats each word as a separate access, and these word accesses vote on a request type for the full dynamic access. Similarly, dynamic accesses vote on the request type to use for their associated static memory instruction.

Algorithm 1 chooses between using a **ReqV**, **ReqS**, or **ReqO+data** for load accesses based on whether obtaining ownership is expected to improve global cache reuse, obtaining shared state will improve global cache reuse, or neither will improve reuse (motivating valid state). Algorithm 2 chooses between using a **ReqO** or **ReqWTfwd** for store accesses based on whether obtaining ownership is expected to increase global cache reuse. Algorithm 3 chooses **ReqO+data** or **ReqWTfwd+data** for an atomic RMW access based on whether obtaining ownership is expected to increase global cache reuse.

Although the selection process is designed to be as hardware-agnostic as possible, we assume the following information is available to the selection mechanism:

- Cache capacity (for determining long-term reuse potential) and cache block size (for spatial reuse potential).
- Whether or not the hardware supports the optimizations described in Section 4.2.
- Whether each device[6] is more likely to be sensitive to memory latency or bandwidth (i.e., whether it is a CPU or GPU).

If the above information is not available at selection time, multiple versions may be generated and a runtime can dynamically select from them when the information becomes available.

ALGORITHM 1: Load request type selection (uses Algorithms 5–7)

```
1: if OwnershipBeneficial($X$) then
2:     $type$ = ReqO+data
3: else if SharedStateBeneficial($X$) then
4:     $type$ = ReqS
5: else if OwnerPredBeneficial($X$) then
6:     $type$ = ReqVo
7: else
8:     $type$ = ReqV
```

ALGORITHM 2: Store request type selection (uses Algorithms 5 and 7)

```
1: if OwnershipBeneficial($X$) then
2:     $type$ = ReqO
3: else if OwnerPredBeneficial($X$) then
4:     $type$ = ReqWTo
5: else
6:     $type$ = ReqWTfwd
```

[3]This trace generation and analysis can be integrated into a compiler or profiler-guided optimization flow, but we leave this for future work.
[4]This SC trace generation is exclusively used for request type selection; as mentioned in Section 4.2.3 the underlying hardware is still free to use (and we evaluate) a relaxed memory model.
[5]This methodology can also use byte-level access granularity, but this was not necessary for the workloads studied.
[6]"Device" is used to refer to any computing element which has a single local private cache, including individual CPU and GPU cores.

ALGORITHM 3: RMW request type selection (uses Algorithms 5 and 7)

```
if OwnershipBeneficial($X$) then
    $type$ = ReqO+data
else if OwnerPredBeneficial($X$) then
    $type$ = ReqWTo+data
else
    $type$ = ReqWTfwd+data
```

ALGORITHM 4: Request granularity selection for access X of type $type$ (uses function definitions in Section 4.5)

```
if $type$ == ReqV then
    $mask$ = IntraSynchLoadReuse($X$)
else if $type$ == ReqS then
    $mask$ = FullBlockMask($X$)
else if $type$ == ReqWT OR $type$ == ReqWT+data then
    $mask$ = RequestedWordsOnly($X$)
else if $type$ == ReqO OR $type$ == ReqO+data then
    $mask$ = InterSynchStoreReuse($X$)
    if $mask$ != RequestedWordsOnly($X$) then
        $type$ = ReqO+data
```

ALGORITHM 5: OwnershipBeneficial(X) (uses function definitions in Section 4.5)

```
$Phase = 5, Xscore = 0, Y = X, Yprev = X, prevList = [X]$
while $Y$ = NextConflict($Y$) do
  if diffCores($Yprev, Y$) or SyncSep($Yprev, Y$) then
    $Phase = Phase - 1$
    if $Phase < 0$ or (sameCore($X, Y$) and not reusePossible($X, Y$)) then break
    if sameCore($Y$, prevList) then $Yval = 2 * $ Criticality($Y$)
    else $Yval = 0.5*$ Criticality($Y$)
    if sameCore($X, Y$) then $Xscore = Xscore + Yval$
    else $Xscore = Xscore - Yval$ ; $prevList.push(Y)$
  $Yprev = Y$
if $Xscore > 0$ then return $true$ else return $false$
```

ALGORITHM 6: SharedStateBeneficial(X) (uses function definitions in Section 4.5)

```
if fromGPU($X$) then
    return false
$Y = X, Yprev = X$
while $Y$ = NextBlockConflict($Y$) do
    if diffCores($Y, Yprev$) or syncSep($Y, Yprev$) then
        if isLoad($Y$) and sameCore($X, Y$) then
            return $true$
        if isStore($Y$) and diffCores($X, Y$) then
            return $false$
        $Yprev = Y$
return $false$
```

ALGORITHM 7: OwnerPredBeneficial(X) (uses function definitions in Section 4.5)

```
$Xscore = 0$
$Phase = 4$, $Xprev$ = PrevConf($X$)
$Y = X$
while $Y$ = PrevAcc($Y$) do
    $Yprev$ = PrevConf($Y$)
    if sameCore($X, Y$) and sameType($X, Y$) then
        $Phase = Phase - 1$
        if $Phase < 0$ then
            break
        if sameCore($Yprev, Xprev$) then
            $Xscore = Xscore + 1$
        else
            $Xscore = Xscore - 1$
if $Xscore > 0$ then
    return $true$
else
    return $false$
```

4.5 Basic Request Selection Functions

Some function definitions are straightforward. **isLoad**(X), **isStore**(X), **fromCPU**(X), and **fromGPU**(X) return true if X is a load, X is a store, X is issued from a CPU, or X is issued from a GPU, respectively. **sameCore**(X, Y) and **diffCores**(X, Y) return true if X and Y are issued from the same core or different cores, respectively. **sameInst**(X, Y) returns true if X and Y are dynamic instances of the same static instruction.

ReusePossible(X, Y) approximates whether the data accessed by X will still be in the cache when access Y is executed, and it returns true only if the reuse distance (defined here as the number of unique bytes accessed between X and Y from the issuing core in the dynamic trace) is less than N, where N is set to be 75% of the cache capacity.

NextConflict(X) returns the next access to the same address as X following X in the dynamic trace. It is used to determine whether subsequent accesses to a given variable will be issued from the same core (potentially enabling cache reuse) or from a different core (potentially interfering with reuse). **NextBlockConflict**(X) returns the next access to any address in the cache block accessed by X following X in the trace.

PrevAcc(X) returns the most recent access preceding X in the dynamic trace. **PrevConf**(X) returns the most recent access to the same address as X preceding X in the trace.

SyncSep(X, Y) is true if X and Y are issued from the same core and there exists a synchronization operation S between X and Y in program order such that either 1) X or Y is an atomic access, 2) X is a load and S is an acquire operation (e.g., the start of a new kernel), or 3) X is a store and S is a release operation (e.g., the completion of a kernel). Since synchronization accesses can invalidate and flush the cache of Valid data and atomic accesses can only hit on owned state, this helps determine whether obtaining Owned or Shared permission is required to exploit available reuse.

Criticality(X) returns a value that represents the estimated performance criticality of X. We use a very simple heuristic for our evaluation of CPU-GPU workloads. Loads and atomic RMW accesses without release semantics[7] tend to be more performance critical than stores and release atomics, and CPU accesses tend to be more sensitive to access latency than GPU loads. Therefore, we assign CPU loads and non-release atomic RMW accesses a criticality weight of 6 and GPU loads and non-release atomic RMW accesses a criticality weight of 2. All other access types have a criticality weight of 1.

IntraSynchLoadReuse(X) returns a word mask based on whether requesting valid state for additional words in a cache block is expected to improve reuse. For each word in the cache block accessed by X, this function sets the corresponding mask bit if there is a subsequent load Y to that word address which will not be invalidated before it is accessed (i.e., is not separated by a synchronization action). Specifically, this requires that Y satisfies the following properties: (1) **sameCore**(X, Y), (2) **reusePossible**(X, Y), and (3) **!syncSep**(X, Y).

InterSynchStoreReuse(X) returns a word mask based on whether requesting ownership for additional words in a cache block is expected to improve reuse. For each word in the cache block accessed by X, this function sets the corresponding mask bit if there is a subsequent store Y to that word address which cannot be coalesced in a write combining buffer (i.e., is separated by a synchronization action). Specifically, this requires that Y satisfies the properties: (1) **sameCore**(X, Y), (2) **reusePossible**(X, Y), and (3) **syncSep**(X, Y).

4.6 Heuristic Request Selection Algorithms

OwnershipBeneficial(X) is a heuristic that returns true if ownership for X is likely to improve overall performance and is defined in Algorithm 5. At a high level, ownership is beneficial for an access if the issuing core is the most common or most critical requestor in subsequent accesses to the same variable. To approximate whether this is the case, each access Y following X and to the same address as X in the SC execution is assigned a value based on whether X obtaining ownership will help or hurt performance for Y, and improved performance is expected if the sum

[7]Synchronization accesses with release semeantics generally don't use the return value and are therefore less critical.

of these is greater than zero. An access Y is ignored if the previously considered access in the SC memory trace was to the same core and the two are not sync-separated (this is because the lack of intervening synchronization means that reuse is possible for this access regardless of ownership state). Otherwise, Y is assigned a value based on whether the issuing core has accessed the data recently (indicating reuse potential for Y) and the criticality of Y (it is more important to exploit reuse for reads and CPU accesses). If Y is issued from the same core as X, then ownership would help and its value is added to the running sum; if Y is issued from a different core, then ownership for X would hurt performance for Y and its value is subtracted from the running sum. This process ends once a fixed number of subsequent Y accesses are evaluated (5 in our algorithm).

SharedStateBeneficial(X) is a heuristic that returns true if obtaining shared state for X would improve performance based on expected reuse effects, and is defined in Algorithm 6. We assume that obtaining shared state is beneficial if the issuing core is a CPU (it is more sensitive to cache miss overheads than sharer tracking overheads) and it can lead to at least one future cache hit. This is true if there is at least one load to the same cache block from the same core, it is sync-separated with X (otherwise reuse is possible with or without shared state), and there is no intervening remote store to that cache block (this would invalidate the shared state).

OwnerPredBeneficial(X) is a heuristic that returns true if using owner prediction is likely to be successful for the given prediction mechanism based on execution history, and is defined in Algorithm 7. In the evaluated system, each device uses a simple prediction mechanism which will succeed if accesses of the same type to the same address obtain data from the same location. To approximate this, we work backwards from X, considering previous accesses to the same address, from the same core, and of the same type (load, store, or RMW) as X. Similar to the ownership benefit heuristic, a score is calculated to determine potential benefit. If the immediate predecessor of an access Y in the SC execution is issued by a remote core and that core is the same core that issues the immediate predecessor to X, then accurate prediction is more likely and that access's value is added to the score. If not, then the value is subtracted from the score. Again, magnitude depends on proximity to X (more recent accesses affect the predictor more), and evaluation stops after a fixed number of prior accesses (five in our algorithm) have been considered.

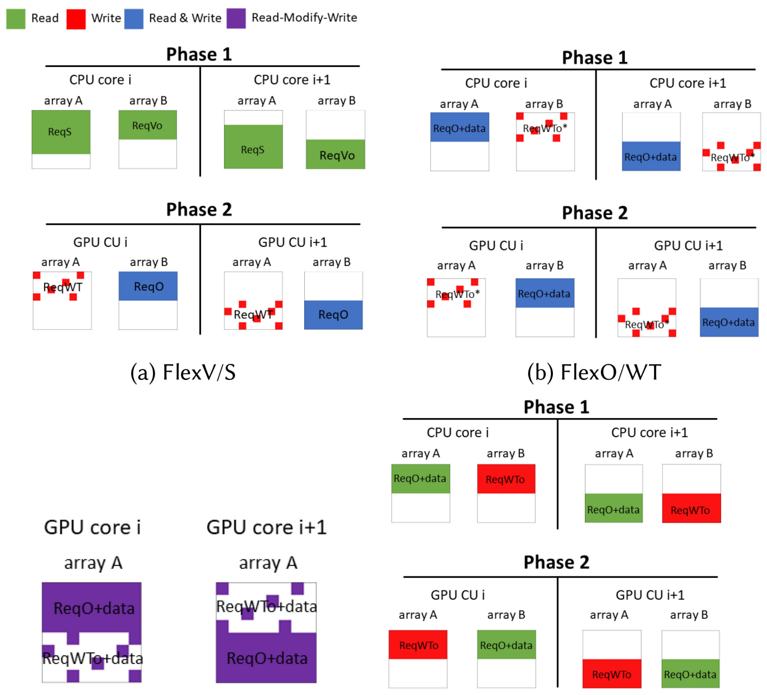
4.7 Incomplete Request Type Support

Some systems may not support all of the optimized request types described in the above algorithms. If owner prediction is not supported, then OwnerPredBeneficial always returns false. If write-through forwarding is not supported by a core, then producer-consumer forwarding is no longer possible. Thus, Criticality(X) should return the same value for loads and stores (consumers should not be preferred for ownership), and selection should then proceed as described above. After selection has completed, forwarded requests must be converted to supported request types. Stores assigned ReqWTfwd should use ReqWT instead. Atomic RMW operations assigned ReqWT-fwd+data should use ReqO+data if both the prior and subsequent accesses in the dynamic trace use ownership, and ReqWT+data otherwise.[8] If word granularity state is not supported by a core (state is stored at cache line granularity), then that core can use full block mask for each request, and any ReqO requests must be converted to ReqO+data requests.

5 TARGET WORKLOADS

Since current heterogeneous systems are not designed for fine-grain coherence specialization, there are no standard benchmarks. Instead, we identify two sets of workloads which exhibit more

[8]Stores and RMWs differ because, unlike a ReqWT, a ReqWT+data can cause an ownership revoke and excessive data transfer, so obtaining ownership may be preferred even if no reuse is possible.



(a) FlexV/S (b) FlexO/WT (c) FlexOa/WTa (d) Prod-Cons

Fig. 2. Access patterns and coherence request types used by fine-grain coherence specialization for microbenchmarks. All accesses except those marked with * are to the same partition in each subsequent iteration of the algorithm.

request diversity than is common in conventional heterogeneous applications, which we refer to as microbenchmarks and applications. We use the request selection algorithms from Section 4.4 for all benchmarks.

5.1 Microbenchmarks

We use four microbenchmarks to isolate the benefits of fine-grain coherence specialization. Each microbenchmark has one or more phases in which CPU and/or GPU cores concurrently access shared data, with cores synchronizing between each execution phase. These phases are repeatedly executed.

5.1.1 FlexV/S. This microbenchmark illustrates the benefits of choosing between self-invalidated reads (ReqV) and writer-invalidated reads (ReqS) at a per-access granularity. As shown in Figure 2(a), CPU cores densely read array A in phase 1, and these accesses exhibit both sharing (multiple CPUs read the same data) and sync-separated reuse (each core accesses the same partition in successive CPU phases). CPU cores also densely read array B in phase 1 with no sharing and no reuse (each core accesses different partitions in successive CPU phases). In phase 2, GPU cores sparsely write array A with no reuse, and densely read and write array B with high reuse. This workload is representative of any workload which must read from two data structures: one which exhibits inter-phase reuse and sharing and one which exhibits no reuse. For example, a weight matrix in a DNN layer will exhibit reuse and sharing because the same weight values will be accessed by all cores processing the layer in every iteration, while the feature inputs will read different data in each iteration (e.g., if a queue or double-buffering is used for inputs).

CPU reads to array A exhibit reuse across phases and are unlikely to be separated by a remote write, but multiple CPUs access the same data concurrently. Thus, shared state is beneficial and ownership is not beneficial, resulting in the use of ReqS. CPU reads to array B exhibit no reuse, but the remote core is likely to be predicted based on the last responder (the partition of B accessed by a CPU core in a given phase will have been produced by the same GPU core). Therefore, ReqVo is used for these accesses. GPU stores to A exhibit low inter-phase reuse, so ReqWTfwd is used. GPU stores to B exhibit high inter-phase reuse and are not accessed more frequently by any remote core, so ownership is beneficial and ReqO+data is used for these accesses.

5.1.2 FlexO/WT. This microbenchmark illustrates the benefits of choosing between ownership-based data stores (ReqO) and write-through data stores (ReqWT) at a per-access granularity. As illustrated in Figure 2(b), CPU cores densely read and write the same partition of array A and sparsely write array B in phase 1. Then, GPU cores densely read and write array B and sparsely write array A in phase 2. The dense CPU and GPU accesses are to the same partition in successive iterations, while the sparse CPU and GPU accesses are to different partitions in successive iterations. This represents workloads which update different data structures with different reuse levels in a non-racy manner (e.g., machine learning models with both sparse and dense layers like the Deep Learning Recommendation Model [65]).

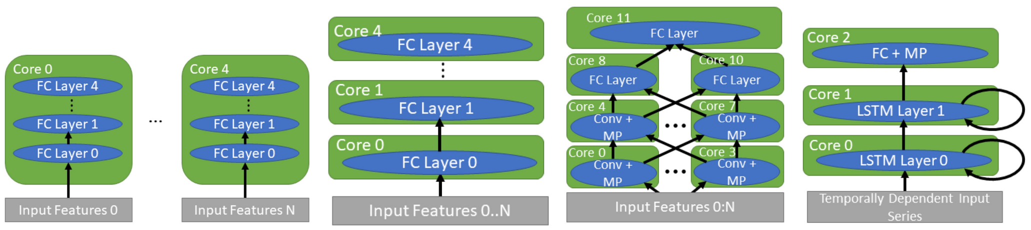
The dense CPU and GPU accesses exhibit sync-separated reuse, and ownership is beneficial because each element is unlikely to be accessed by a remote core between the reuse accesses. Therefore, these accesses request ownership (ReqO+data for reads, ReqO for writes). The sparse CPU and GPU writes exhibit low reuse potential, and owner prediction is feasible because the same remote core will be the most recent owner of all data accessed by a core in a given phase. Therefore, ReqWTo is used for these accesses.

5.1.3 FlexOa/WTa. This microbenchmark illustrates the benefits of choosing between ownership-based atomic accesses (ReqO+data) and write-through atomic accesses (ReqWT+data) at a per-access granularity. As illustrated in Figure 2(c), a single phase and single core type is sufficient for this purpose since atomics enable racy accesses and communication.[9] In each iteration, each core densely reads and writes its local partition and sparsely reads and writes a remote partition. This is representative of any workload which distributes a data structure across multiple cores and allows each core to sparsely access remote partitions concurrently with remote cores (e.g., racy updates to a hybrid dense+sparse machine learning model, or push-based graph workloads with densely connected components).

CPU accesses to A's local partition exhibit high sync-separated reuse and intervening remote accesses are rare, so ownership is beneficial and these RMWs use ReqO+data requests. Sparse accesses to remote partitions exhibit low sync-separated reuse, and the remote owner can be predicted within a phase. Therefore, these accesses use ReqWTo+data.

5.1.4 Prod-Cons. This microbenchmark illustrates the benefits of write-through forwarding and owner prediction. In each iteration, CPUs read a partition of array A and write a partition of array B in phase 1. GPUs then read a partition of array B and write a partition of array A in phase 2. This is representative of streaming workloads, data transform kernels, or applications that transfer input and output data to and from a specialized accelerator (e.g., the EP algorithm discussed in Section 5.2.4).

[9]GPU-only is studied, but a CPU-only version would exhibit the same request types and tradeoffs.



(a) Data Parallel FCNN (b) Pipelined FCNN (c) Pipelined LeNet (d) Pipelined LSTM

Fig. 3. Comparison of compute to device mappings for data parallel FCNN (a) and pipelined FCNN (b), LeNet (c), and LSTM (d) neural networks.

Each core reads the same partition and writes the same partition in each iteration, so there is sync-separated reuse for all accesses. When write-through forwarding is supported, the criticality and reuse of consumer reads will outweigh the locality benefits for producer writes such that ReqO+data will be used for reads, and ReqWTo will be used for writes (owner prediction is feasible due to the regular access pattern). If write-through forwarding is not supported, then reads are not rated more highly in the evaluation of ownership benefit, so ReqVo is used for reads and ReqO is used for writes.

5.2 Applications

In addition to the above microbenchmarks, we evaluate four full applications which can benefit from fine-grain coherence specialization: a **fully connected neural network (FCNN)**, a convolutional neural network (LeNet), a recurrent neural network (LSTM), and an **evolutionary programming algorithm (EP)**. FCNN, LeNet, and LSTM are GPU-only neural network workloads, all of which assume a batch size of 1.[10] EP is a CPU+GPU application which demonstrates how fine-grain coherence specialization can benefit applications that offload only some tasks to a specialized accelerator (the GPU).

5.2.1 Fully Connected Neural Network (FCNN). Fully connected networks have proven useful for processing inputs that are spatially and temporally unrelated such as medical [75], real estate [27], and recommendations [28].

We compare data parallel and pipelined implementations of a neural network consisting of five uniform, fully connected layers. Figures 3(a) and 3(b) illustrate how each type of parallelism maps layers to devices. In a data parallel implementation, each device independently processes all layers for a distinct subset of the input features, avoiding the need for inter-device communication but preventing reuse for weight data. In a pipelined implementation, each device executes a single stage of the computation for every input. Although a pipelined implementation requires inter-device communication along producer-consumer edges, it is able to exploit reuse in weight accesses because each device only needs to load a single weight matrix for all inputs.

Both versions use five GPU CUs to process 60 independent input features. In each iteration, each layer multiplies an input vector with a weight matrix and performs a ReLU operation. In the pipelined implementation, inputs and outputs are double-buffered to enable concurrent pipelined execution, and atomics are used to synchronize between adjacent layers. Specialized coherence is enabled only for the pipelined implementation (since data parallel does not benefit

[10]Larger batch sizes can improve cache reuse, but this comes at the cost of end-to-end latency, which is often unacceptable for latency sensitive inference workloads running on resource-constrained systems [32, 61].

from specialization). Ownership is beneficial for the weight matrix and input buffer loads (which use ReqO+data), while output buffers stores use write-through forwarding and owner prediction (ReqWTo). Atomic release accesses use write-through forwarding (ReqWTo+data), while acquire accesses use owned atomics (ReqO+data). This constitutes producer-consumer forwarding for the synchronizing atomics.

5.2.2 Convolutional Neural Network (LeNet). To classify spatially related data such as images, some of the most common neural networks in deep learning are primarily composed of convolutional layers with a few fully connected layers at the end. One such example is LeNet, which has been used to determine the numerical values of handwritten digits [52].

LeNet consists of two convolutional layers, two maxpool layers, and two fully connected layers [51]. Again, we evaluate both a data parallel version and a pipelined parallel version. Both pipelined and data parallel versions use 12 GPU CUs to process 80 independent input features. In the data parallel implementation, each CU independently processes all of LeNet's layers for a unique subset of the inputs, similar to data parallel FCNN. In the pipelined version, each CU processes a subset of the network for all inputs in a pipelined manner. Figure 3(c) illustrates the mapping of computation to devices for the pipelined implementation. However, the variable work required at each layer in LeNet complicates the pipelined implementation; load imbalance between nodes in the pipeline can lead to performance bottlenecks, since inputs can only be processed as fast as the slowest layer. To improve load balance, long-running layers have been split among multiple cores where possible. Like FCNN, loads to the weight matrices benefit from ownership and use ReqO+data when fine-grain coherence specialization is enabled, and atomic accesses use producer-consumer forwarding. However, feature data reads do not benefit from ownership, since the feature data produced at each layer is concurrently read by multiple consumers. Therefore, producer-consumer forwarding is not possible for LeNet features.

5.2.3 Recurrent Neural Network (LSTM). Recurrent neural networks can be used to classify sequentially associated data series such as audio [10], video [92], DNA [69], ECG data [23], and aircraft sensor data [64]. We evaluate a **long short-term memory (LSTM)** RNN model proposed for network anomaly detection [71]. The model uses two stacked LSTM layers, a single dense layer activation, and a single fully connected softmax output layer, with a ten-token input sequence to predict the subsequent (eleventh) token. This dataflow is illustrated in Figure 3(d).

Since the inputs have serial dependencies, the only way to parallelize a single input sequence is with pipelined parallelism. Each LSTM layer is composed of 50 hidden cells, 50 memory cells, a 50-entry input vector, a 50-entry output vector, and four 50×100 entry weight matrices. The layers are distributed among CUs; each LSTM layer is assigned to a CU, while the dense layer and softmax layer are fused and assigned to one CU. As with pipelined FCNN and LeNet, double-buffering and atomic accesses are used to synchronize between adjacent layers. With fine-grain coherence specialization, loads to the weight matrices and input buffers use ReqO+data, stores to output buffers use ReqWTo. Synchronizing atomic accesses use producer-consumer forwarding.

5.2.4 Evolutionary Programming (EP). Evolutionary Programming mimics the natural selection process in a simulated population to minimize a cost function. It consists of repeated execution of five consecutive stages: reproduction, evaluation, selection, crossover, and mutation [83]. We use parameters used by Capraro et al. for radar waveform selection [21] where population is 330 and mutation and crossover rates are set to 0.5. For the mutation stage, we perform a **centre inverse mutation (CIM)** where each chromosome is divided into two sections and all genes in each section are copied and then inversely placed in the same section of a child [4]. For task-partitioning, we use HeteroMark's implementation which offloads the *evaluation* and *mutation* stages to the GPU, and

Table 2. Simulated Heterogeneous System Parameters

CPU/GPU Parameters	
Frequency	2 GHz/700 MHz
Cores	16/16
Memory Hierarchy Parameters	
L1 Size (32-way assoc.)	128 KB
L2 Size (16 banks, 32-way assoc.)	8MB
L1 MSHRs, Write Buffer, and Store Buffer Sizes	128 entries each
L1 hit latency in cycles	1
Remote L1 hit latency in cycles	135−183
L2 hit latency in cycles	129−161
Memory latency in cycles	297−361

executes the remaining stages on the CPU [83]. With fine-grain coherence specialization, we find that much of the data transferred between CPU and GPU experiences high reuse on both devices. Since CPU reuse is weighted higher than GPU reuse in the selection algorithms (Section 4.4), our specialized coherence implementation obtains ownership for CPU reads (ReqO+data) and forwards GPU writes to the CPU to improve CPU reuse (ReqWTo), at the cost of GPU reuse.

6 METHODOLOGY

6.1 Coherence Configurations

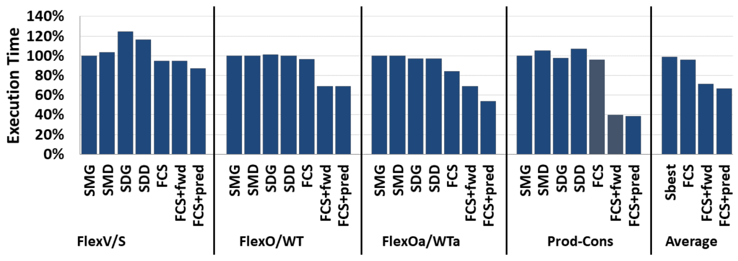
We compare seven different configurations of device cache architectures. The first four configurations use a static request type selection at each device cache corresponding to the MESI, DeNovo, or GPU coherence protocols. All L1 caches interface directly with the Spandex LLC.

SMG: Static - MESI CPU caches, GPU coh. GPU caches
SMD: Static - MESI CPU caches, DeNovo GPU caches
SDG: Static - DeNovo CPU caches, GPU coh. GPU caches
SDD: Static - DeNovo CPU caches, DeNovo GPU caches

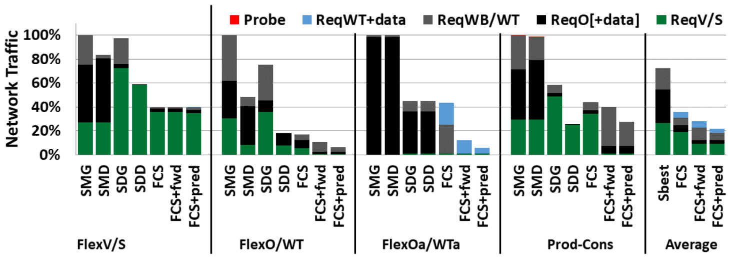
The static protocol configurations are chosen based on the suitability of each protocol for CPU and GPU memory demands. The final three configurations use fine-grain coherence specialization with neither write-through forwarding and owner prediction (**FCS**), with only write forwarding (**FCS+fwd**), and both (**FCS+pred**). We choose the memory request type according to Section 4.4, and owner prediction is based on a per-core prediction table which stores the last responder's ID for each Spandex request type.

6.2 Simulation Environment and Parameters

We execute the above workloads and configurations using an integrated CPU-GPU architecture simulator. GPGPU-Sim [14] models the GPU CUs (we use an architecture similar to NVIDIA GTX480). Simics [56] models the CPU cores, Wisconsin GEMS [60] models the memory timing, and Garnet [7] models the network. There are 16 CPU cores and 16 GPU CUs, connected through a 4x4 mesh (a CPU core and GPU CU at each node). Each CPU core and GPU CU has its own L1 cache, and these caches interface directly with a Spandex LLC with a latency consistent with GPU L2s [1, 42]. The modeled system is relatively small compared to the state of the art, but it is the latest GPGPU-Sim model we are aware of that has been validated. It can be considered representative of smaller embedded or edge GPU devices, and we believe it is sufficient for demonstrating the potential of FCS. With larger high performance GPUs, the benefits can be expected to improve further, since larger GPUs will be even more limited by memory efficiency. Table 2 lists the detailed simulation parameters.



(a) Execution time normalized to SMG.



(b) Network traffic normalized to SMG.

Fig. 4. Microbenchmark's execution time and network traffic.

7 EVALUATION

7.1 Microbenchmark Results

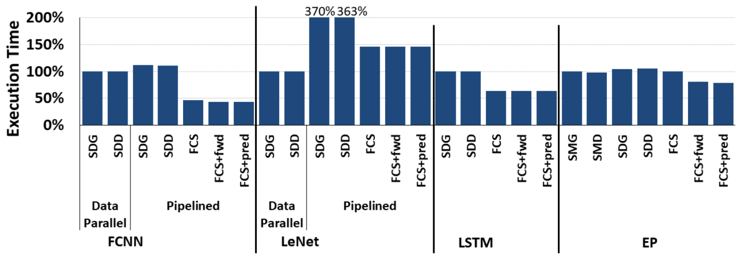
Figure 4 shows the execution time and network traffic for the microbenchmarks. Sbest represents the best performing (lowest execution time) static coherence configurations for each microbenchmark averaged over all microbenchmarks.

FlexV/S: Obtaining Shared state for the read accesses to array A improves CPU cache hit rates, reducing latency and traffic for these CPU read accesses. Obtaining Shared state for the CPU read accesses to array B with no inter-kernel reuse is wasteful and can require revoking ownership from a remote GPU core, preventing reuse there.

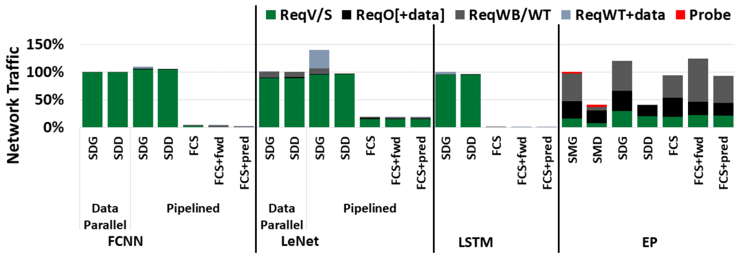
FCS exploits inter-phase CPU reuse for array A while also avoiding wasteful Sharer overheads (e.g., revoking ownership from the GPU) for array B. This improved reuse reduces network traffic by 60% relative to the fastest static configuration (SMG), although execution time is only reduced by 5% due to the GPU's ability to tolerate memory latency. FlexV/S does not benefit from write-through forwarding because written data is never remotely owned, but owner prediction is helpful for CPU reads to A, reducing execution time by a further 8%.

FlexO/WT: FCS reduces execution time and network traffic by 7% each relative to the best static configuration (SDD) by using ReqWT for sparse writes. This revokes ownership from dense owners, but subsequent dense accesses incur only 2-hop misses (versus 3-hops if the sparse access obtained ownership). With FCS+fwd, execution time and network traffic are reduced a further 13% and 38%, respectively, because sparse writes do not revoke ownership from dense owners. Ownership prediction avoids LLC lookups for forwarded stores (ReqWTo), reducing network traffic a further 19%.

FlexOa/WTa: As with FlexO/WT, FCS reduces execution time by 14% and network traffic by 3% relative to the fastest static configuration (SDD) because the write-through sparse accesses



(a) Execution time normalized to non-pipelined SDG



(b) Network traffic normalized to non-pipelined SDG

Fig. 5. Execution time and network traffic for applications.

reduce indirection for future dense accesses. When write-through forwarding is enabled, sparse atomics are forwarded to the core that owns the target partition, improving cache hit rates for dense accesses and reducing execution time and network traffic by a further 18% and 72%, respectively. Owner prediction further reduces indirection for sparse accesses, reducing execution time and network traffic by a further 22% and 50%, respectively.

Prod-Cons: FCS is unable to improve performance beyond SDG for Prod-Cons, but with forwarding enabled (FCS+fwd) it can avoid revoking ownership from latency-sensitive consumers. By forwarding writes to the consumer, FCS+fwd moves the overhead of data movement from the reader to the writer and reduces execution time by 58% relative to the fastest static configuration (SDG). However, since forwarded write-through data must travel two hops rather than one, FCS+fwd increases network traffic by 35% relative to SDD. This overhead can be avoided by owner prediction, and FCS+pred reduces network traffic by 19% relative to SDD.

7.2 Application Results

Figure 5 compares the execution time and network traffic for FCNN, LeNet, LSTM, and EP. FCNN, LeNet, and LSTM are executed on the GPU, and the choice of CPU coherence strategy has no impact on results. We therefore consider only the SDG and SDD static coherence configurations. For FCNN and LeNet, we compare data-parallel and pipeline-parallel implementations (FCS is only evaluated with the pipelined versions because data parallel implementations don't benefit from fine-grain coherence specialization). EP runs on both CPU and GPU cores, so all static coherence configurations are compared (similar to the microbenchmarks).

7.2.1 FCNN. The data parallel configurations exploit reuse in feature vectors and require no synchronization, but they are unable to exploit reuse in weight data since this will be evicted before it is reused. Static coherence configurations that use pipelining increase execution time by

12% relative to the data parallel implementations because they are unable to exploit reuse in feature values or weight values.

When fine-grain coherence specialization is enabled (FCS, FCS+fwd, FCS+pred), pipelined FCNN obtains ownership for reads to layer-specific weight values and is able to exploit cache reuse in this data, reducing execution time by 54% and network traffic by 96% relative to data parallel SDG. Write-through Forwarding (FCS+fwd) additionally reduces execution time by 7% and network traffic by 14% relative to FCS by enabling producers of feature data and synchronization variables to forward them to the consumer. Owner Prediction (FCS+pred) further reduces network traffic by 53% relative to FCS+fwd by avoiding LLC lookups for the minimal remaining inter-device communication required in FCS+fwd (as with FlexO/WT and Prod-Cons, execution time is minimally affected because GPU stores can tolerate latency).

7.2.2 LeNet. As with FCNN, the weight data for the convolutional and fully connected layers will be evicted before they can be reused in a data parallel LeNet implementation. LeNet's pipelined implementation faces one significant challenge that did not exist for FCNN: pipeline imbalance. This combined with an inability to reuse weight data across synchronization causes pipelined SDG and SDD to increase execution time by over 3x relative to data parallel SDG. By exploiting reuse for weight matrix reads, FCS eliminates much of the execution time overhead of the static pipelined implementations and reduces network traffic by 82% relative to data parallel. Relative to the fastest data parallel implementation, however, pipelined FCS+pred increases execution time by 46% due to pipeline imbalance, which FCS is unable to address. Even so, a pipelined FCS implementation reduces the end-to-end latency of a single iteration by 41% relative data parallel (which uses only a single device to execute each input), which may be important for latency-sensitive workloads.

7.2.3 LSTM. Compared to FCNN, weight matrices make up an even larger proportion of total data accesses in an LSTM layer. By obtaining ownership for this data, FCS almost entirely eliminates cache misses for LSTM, reducing network traffic by 99% and execution time by 36% relative to SDD. Write-through forwarding (FCS+fwd) additionally reduces the network traffic by 13% relative to FCS, and owner prediction (FCS+pred) further reduces the network traffic by 30% relative to FCS+fwd by avoiding LLC lookups for the minimal remaining inter-device communication required in FCS+fwd.

7.2.4 EP. In EP, during GPU phases, static coherence configurations which use DeNovo for GPU (SMD, SDD) exploit inter-kernel reuse of written data and exhibit lower GPU execution time than GPU coherence for GPU (SMG, SDG). However, this also results in reduced CPU performance for SMD and SDD, as the latency sensitive CPU must either read data or revoke ownership from a remote L1. With FCS, the CPU gains ownership of the heavily accessed data, prioritizing latency sensitive CPU accesses, while the GPU uses write-through stores. Without forwarding, FCS sees no benefit or improved reuse, since GPU writes must revoke the CPU's ownership. However, FCS+fwd forwards GPU writes to the CPU owner and reuse increases for latency-sensitive CPU accesses, resulting in 17% lower execution time versus the fastest static configuration, albeit at the cost of a 207% increase in network traffic due to the additional GPU misses and the indirection incurred by write-through forwarding. Using ownership prediction reduces this overhead, enabling a 20% reduction in execution time with a 130% increase in traffic relative to the fastest static configuration.

8 CONCLUSION

As hardware adapts to address the growing memory bottleneck, efficient coherent data movement is increasingly important. Existing heterogeneous hardware and software miss out on many opportunities for communication optimization because they are designed assuming static coherence protocols which are often complex and difficult to extend.

Fine-grain coherence specialization enables more flexible and efficient hardware-software co-design, allowing future heterogeneous software to leverage data movement optimizations in hardware. We present a methodology that empowers heterogeneous devices to mix and match specialized coherence request types based on individual access properties, including two new coherence specializations for data forwarding. We show that the new specializations can be implemented without adding significant complexity to the underlying protocol as long as a simple and flexible protocol like Spandex is chosen as a baseline. We then demonstrate the performance gains these optimizations can deliver for a set of microbenchmarks and applications. By giving software more control over data movement, we greatly improve cache efficiency, reducing execution time by up to 61% and network traffic by up to 99%. Looking forward, this focus on flexible coherent data movement can improve cache efficiency, motivate new cross-stack coherence optimizations, and enable more efficient programming patterns for emerging heterogeneous workloads.

REFERENCES

[1] Accel-Sim GPGPU-Sim configurations. https://github.com/accel-sim/gpgpu-sim_distribution/tree/dev/configs/tested-cfgs.

[2] 2019. Compute Express Link: Breakthrough CPU-to-Device Interconnect. http://www.computeexpresslink.org. (2019).

[3] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram S. Adve. 1997. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *High-Performance Computer Architecture, 1997., Third International Symposium on*. IEEE, 204–215.

[4] Otman Abdoun, Jaafar Abouchabaka, and Chakir Tajani. 2012. Analyzing the performance of mutation operators to solve the travelling salesman problem. *CoRR* abs/1203.3099 (2012). arXiv:1203.3099 http://arxiv.org/abs/1203.3099.

[5] Manuel E. Acacio, José González, José M. García, and José Duato. 2002. Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1–12.

[6] Sarita V. Adve. 1993. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. Dissertation. University of Wisconsin, Madison.

[7] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *ISPASS*. 33–42.

[8] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics* (2009).

[9] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *ISCA*. IEEE, 172–182.

[10] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu. 2016. Deep speech 2: End-to-end speech recognition in English and Mandarin. In *International Conference on Machine Learning (ICML'16)*. 173–182.

[11] Craig Anderson and Anna R. Karlin. 1996. Two adaptive hybrid cache coherency protocols. In *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*. IEEE, 303–313.

[12] ARM. 2018. AMBA 5 CHI Architecture Specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0050c/index.html. (2018).

[13] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.

[14] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*. 163–174.

[15] Bradford M. Beckmann and Anthony Gutierrez. 2015. The AMD gem5 APU simulator: Modeling heterogeneous systems in gem5. In *Tutorial at the 48th Annual IEEE/ACM International Symposium on Microarchitecture*.

[16] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 213–224.

[17] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250.

[18] Kshitij Bhardwaj, Marton Havasi, Yuan Yao, David M. Brooks, José Miguel Hernández-Lobato, and Gu-Yeon Wei. 2020. A comprehensive methodology to determine optimal coherence interfaces for many-accelerator SoCs. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 145–150.

[19] E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 1999. Multicast snooping: A new coherence method using a multicast address network. In *ISCA (ISCA'99)*. IEEE Computer Society, Washington, DC, USA, 294–304. https://doi.org/10.1145/300979.301004

[20] Jason F. Cantin, James E. Smith, Mikko H. Lipasti, Andreas Moshovos, and Babak Falsafi. 2006. Coarse-grain coherence tracking: RegionScout and region coherence arrays. *IEEE Micro* 26, 1 (2006), 70–79.

[21] C. T. Capraro, I. Bradaric, G. T. Capraro, and Tsu Kong Lue. 2008. Using genetic algorithms for radar waveform selection. In *2008 IEEE Radar Conference*. 1–6.

[22] CCIX. 2018. Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com. (2018).

[23] Sucheta Chauhan and Lovekesh Vig. 2015. Anomaly detection in ECG time signals via deep long short-term memory networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA'15)*. IEEE, 1–7.

[24] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 343–355.

[25] Liqun Cheng, John B. Carter, and Donglai Dai. 2007. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *High Performance Computer Architecture, (HPCA'07). IEEE 13th International Symposium on*. IEEE, 328–339.

[26] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 155–166.

[27] Sumit Chopra, Trivikraman Thampy, John Leahy, Andrew Caplin, and Yann LeCun. 2007. Discovering the hidden structure of house prices with a non-parametric latent manifold model. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 173–182.

[28] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for YouTube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 191–198.

[29] Alan L. Cox and Robert J. Fowler. 1993. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 98–108. https://doi.org/10.1109/ISCA.1993.698549

[30] Mahdad Davari, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2015. The effects of granularity and adaptivity on private/shared classification for coherence. *ACM Transactions on Architecture and Code Optimization (TACO'15)* 12, 3 (2015), 26.

[31] David L. Dill. 1996. The Mur ϕ verification system. In *International Conference on Computer Aided Verification*. Springer, 390–393.

[32] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 1–14.

[33] David Gelernter. 1989. Multiple tuple spaces in Linda. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 20–27.

[34] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. 184–195.

[35] B. A. Hechtman and D. J. Sorin. 2013. Evaluating cache coherent shared virtual memory for heterogeneous multicore chips. In *ISPASS*. https://doi.org/10.1109/ISPASS.2013.6557152

[36] Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *HPCA*.

[37] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. 2010. Remote store programming. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 3–17.

[38] Hemayet Hossain, Sandhya Dwarkadas, and Michael C. Huang. 2008. Improving support for locality and fine-grain sharing in chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 155–165.

[39] Hemayet Hossain, Sandhya Dwarkadas, and Michael C. Huang. 2011. POPS: Coherence protocol optimization for both private and shared data. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 45–55.

[40] Stefanos Kaxiras and James R. Goodman. 1999. Improving CC-NUMA performance using instruction-based prediction. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium on*. IEEE, 161–170.

[41] Stefanos Kaxiras and Georgios Keramidas. 2010. SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro* 30, 5 (2010), 54–65.

[42] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*. IEEE.

[43] Rakesh Komuravelli. 2015. *Exploiting Software Information for an Efficient Memory Hierarchy*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[44] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. 2014. Revisiting the complexity of hardware cache coherence and some implications. *ACM Transactions on Architecture and Code Optimization* 11, 4, Article 37 (Dec. 2014), 22 pages. https://doi.org/10.1145/2663345

[45] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access pattern-aware cache management for improving data utilization in GPUs. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 307–319.

[46] David A. Koufaty, Xiangfeng Chen, David K. Poulsen, and Josep Torrellas. 1996. Data forwarding in scalable shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 7, 12 (1996), 1250–1264.

[47] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. 2012. Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 376–388.

[48] George Kurian, Omer Khan, and Srinivas Devadas. 2013. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, USA, 523–534. https://doi.org/10.1145/2485922.2485967

[49] An-Chow Lai and Babak Falsafi. 2000. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of 27th International Symposium on Computer Architecture (ISCA)*. 139–148. https://doi.org/10.1109/ISCA.2000.854385

[50] Alvin R. Lebeck and David A. Wood. 1995. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *ISCA*. 48–59.

[51] Yann LeCun. 2015. LeNet-5, convolutional neural networks. 20 (2015), http://yann.lecun.com/exdb/lenet.

[52] Yann LeCun, L. D. Jackel, Leon Bottou, A. Brunot, Corinna Cortes, J. S. Denker, Harris Drucker, I. Guyon, U. A. Muller, Eduard Sackinger, Patrice Simard, and Vladimir Vapnik. 1995. Comparison of learning algorithms for handwritten digit recognition. In *International Conference on Artificial Neural Networks*, Vol. 60. Perth, Australia, 53–60.

[53] Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. 2011. An OpenCL framework for homogeneous manycores with no hardware cache coherence. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 56–67.

[54] Yun Liang, Xiaolong Xie, Guangyu Sun, and Deming Chen. 2015. An efficient compiler framework for cache bypassing on GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (2015), 1677–1690.

[55] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 553–564.

[56] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[57] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC development with open ESP. *arXiv preprint arXiv:2009.01178* (2020).

[58] Milo M. K. Martin, Mark D. Hill, and David A. Wood. 2003. Token coherence: Decoupling performance and correctness. In *ACM SIGARCH Computer Architecture News*, Vol. 31. ACM, 182–193.

[59] Milo M. K. Martin, Pacia J. Harper, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2003. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, New York, NY, USA, 206–217. https://doi.org/10.1145/859618.859642

[60] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News* (2005).

[61] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. 2018. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials* 20, 4 (2018), 2923–2960.

[62] Shubhendu S. Mukherjee and Mark D. Hill. 1998. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*. IEEE Computer Society, Washington, DC, USA, 179–190. https://doi.org/10.1145/279358.279386

[63] Benjamin Munger, David Akeson, Srikanth Arekapudi, Tom Burd, Harry R. Fair, Jim Farrell, Dave Johnson, Guhan Krishnan, Hugh McIntyre, Edward McLellan, Samuel Naffziger, Russell Schreiber, Sriram Sundaram, Jonathan White, and Kathryn Wilcox. 2016. Carrizo: A high performance, energy efficient 28 nm APU. *JSSC* 51, 1 (2016), 105–116.

[64] Anvardh Nanduri and Lance Sherry. 2016. Anomaly detection in aircraft data using Recurrent Neural Networks (RNN). In *2016 Integrated Communications Navigation and Surveillance (ICNS'16)*. IEEE, 5C2–1.

[65] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[66] Håkan Nilsson and Per Stenström. 1994. An adaptive update-based cache coherence protocol for reduction of miss rate and traffic. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 363–374.

[67] OpenCAPI. 2017. Welcome to OpenCAPI Consortium. http://www.opencapi.org. (2017).

[68] Vassilis Papaefstathiou, Manolis G. H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. 2013. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th International ACM Conference on Supercomputing*. ACM, 325–334.

[69] Seunghyun Park, Seonwoo Min, Hyun-Soo Choi, and Sungroh Yoon. 2017. Deep recurrent neural network-based identification of precursor microRNAs. In *Advances in Neural Information Processing Systems*. 2891–2900.

[70] David K. Poulsen and Pen-Chung Yew Pen-Chung Yew. 1994. Data prefetching and data forwarding in shared memory multiprocessors. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, Vol. 2. IEEE, 280–280.

[71] Benjamin J. Radford, Leonardo M. Apolonio, Antonio J. Trias, and Jim A. Simpson. 2018. Network traffic anomaly detection using recurrent neural networks. *arXiv preprint arXiv:1803.10769* (2018).

[72] Umakishore Ramachandran, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. 1995. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. IEEE, 62–62.

[73] Alberto Ros, Manuel E. Acacio, and José M. García. 2008. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[74] Jeffrey B. Rothman and Alan Jay Smith. 2000. Sector cache design and performance. In *ISMASCTS*. 124–133.

[75] Muhammad Akmal Sapon, Khadijah Ismail, and Suehazlyn Zainudin. 2011. Prediction of diabetes by using artificial neural network. In *Proceedings of the 2011 International Conference on Circuits, System and Simulation, Singapore*, Vol. 2829. 299303.

[76] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Press.

[77] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *MICRO*.

[78] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache coherence for GPU architectures. In *HPCA*. https://doi.org/10.1109/HPCA.2013.6522351

[79] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal memory streaming. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 69–80.

[80] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture* 6, 3 (2011), 1–212.

[81] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Transactions on Graphics (TOG'14)* 33, 6 (2014), 228.

[82] Per Stenström, Mats Brorsson, and Lars Sandberg. 1993. An adaptive cache coherence protocol optimized for migratory sharing. *ACM SIGARCH Computer Architecture News* 21, 2 (1993), 109–118.

[83] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, A benchmark suite for CPU-GPU collaborative computing. In *IEEE International Symposium on Workload Characterization (IISWC'16)*.

[84] The Gen-Z Consortium. 2017. Welcome to The Gen-Z Consortium! http://genzconsortium.org. (2017).

[85] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUS*. ACM, 25–35.

[86] Josep Torrellas, H. S. Lam, and John L. Hennessy. 1994. False sharing and spatial locality in multiprocessor caches. *TOCS* 43, 6 (1994).

[87] Pedro Trancoso and Josep Torrellas. 1996. The impact of speeding up critical sections with data prefetching and forwarding. In *Supercomputing*. ACM/IEEE.

[88] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu. 2018. The locality descriptor: A holistic cross-layer abstraction to express data locality in GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 829–842. https://doi.org/10.1109/ISCA.2018.00074

[89] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B. Gibbons, and Onur Mutlu. 2018. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE.

[90] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*. IEEE, 199–208.

[91] Hongbo Yang, Ramaswamy Govindarajan, Guang R. Gao, and Ziang Hu. 2005. Improving power efficiency with compiler-assisted cache replacement. *Journal of Embedded Computing* 1, 4 (2005), 487–499.

[92] Li Yao, Atousa Torabi, Kyunghyun Cho, Nicolas Ballas, Christopher Pal, Hugo Larochelle, and Aaron Courville. 2015. Describing videos by exploiting temporal structure. In *Proceedings of the IEEE International Conference on Computer Vision*. 4507–4515.

[93] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. ACM, 24–32.

[94] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based deep learning models 10x faster. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 951–965.

[95] Hongzhou Zhao, Arrvindh Shriraman, Snehasish Kumar, and Sandhya Dwarkadas. 2013. Protozoa: Adaptive granularity cache coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, USA, 547–558.

[96] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. VersaPipe: A versatile programming framework for pipelined computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 587–599.

Received August 2021; revised April 2022; accepted April 2022