Kobold: Simplified Cache Coherence for Cache-Attached Accelerators

Jennifer Brana[⋆†] Brian C. Schwedock[†] Yatin A. Manerkar[‡] Nathan Beckmann[†]
[⋆] University of Portland [†] Carnegie Mellon University [‡] University of Michigan
brana23@up.edu bschwedo@andrew.cmu.edu manerkar@umich.edu beckmann@cs.cmu.edu

***Abstract*—The ever-increasing cost of data movement in computer systems is driving a new era of data-centric computing. One of the most common data-centric paradigms is near-data computing (NDC), where accelerator resources are placed inside the memory hierarchy to avoid the costly transfer of data to the core. NDC systems show immense potential to improve performance and energy efficiency. Unfortunately, adding accelerators into the memory hierarchy incurs significant complexity for system integration because accelerators often require cache-coherent access to memory. The complex coherence protocols required to handle both cores and cache-attached accelerators result in significantly higher verification costs as well as an increase in directory state and on-chip network traffic. Furthermore, these mechanisms can cause cache pollution and worsen baseline processor performance.**

To simplify system integration for cache-attached accelerators, we present Kobold, a new coherence protocol and implementation. Kobold restricts the added complexity of the accelerator to its local tile and leaves the rest of the system unmodified. Kobold introduces a new directory structure within the L2 cache to track the accelerator's private cache and maintain local coherence between the core and accelerator, leaving the LLC protocol unchanged. Optionally, Kobold implements a minor modification to the LLC protocol to provide significant performance improvements. We verified Kobold's stable-state coherence protocols using the Murphi model checker and estimated area overhead using Cacti 7. Kobold adds only 0.09% area over the baseline caches. Kobold simplifies integration of cache-attached accelerators with minimal area and performance overhead.

I. Introduction

Computer systems are increasingly bottlenecked by the rising cost of data movement [14, 19, 20, 25]. To combat this trend, near-data computing (NDC) designs propose to add accelerator resources within memory hierarchies to, e.g., move compute closer to data instead of transfering data to compute. *Cache-attached accelerators* are a promising direction for NDC that enables fine-grain collaboration between cores and accelerators by offloading work to within the existing CPU cache hierarchy [12, 13, 28, 33, 52, 53].

Fig. 1 shows täkō [46], a representative recent system with cache-attached accelerators. täkō augments a baseline, cache-coherent multicore with an *engine* (i.e., accelerator[1]) on each tile, granting the engine efficient access to data in the tile's L2 and LLC banks. Each engine also contains its own private, cache-coherent data cache (eL1D).

Challenges: Cache-attached accelerators must maintain coherence with the core's caches to usefully access shared memory. However, introducing accelerators into the coherence protocol increases verification costs, directory state, and network traffic [34]. Additionally, cache-attached accelerators can disrupt baseline processor performance by, e.g., polluting processor caches.

[1]We use the terms "accelerator" and "engine" interchangeably.

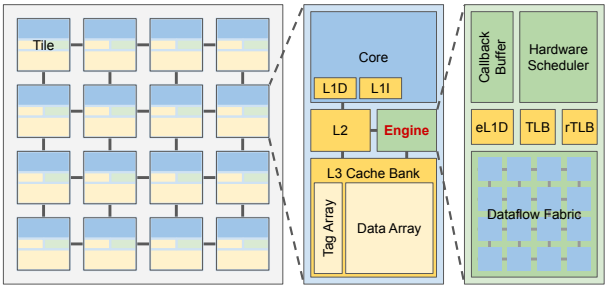


Fig. 1: täkō [46] adds a reconfigurable engine to each tile of a CMP. Engines accelerate tasks for data that resides in the L2 or L3 bank on that tile. Each engine has a coherent eL1D cache.

Insights: The complexity and extra state added to the shared last-level cache (LLC) protocols caused by integrating cache-attached accelerators can be eliminated if the engine's eL1D and its local L2 bank look like a single, unified cache to the LLC. This is achievable by adding extra state within each tile of the chip-multiprocessor (CMP) to track coherence between the core and engine. Keeping coherence between the core and engine local to the tile reduces the necessary directory state and on-chip network traffic, while also ensuring that the LLC coherence protocol remains unchanged.

However, just making the eL1D an inclusive child cache of the L2, as in traditional hiearchical coherence designs, can result in the engine polluting the L2 with data the core does not need. Consequently, we propose a design where the L2 is not inclusive of the eL1D, with simple policies to ensure that data accessed by the eL1D does not evict other useful data in the L2. Implementing coherence without inclusion requires tracking a small amount of additional state in the L2.

Approach: Our goal is to restrict the complexity of cache-attached accelerators to *within each tile* of a CMP, so that the rest of the system and coherence protocol are unmodified. Kobold adds a directory-like structure to each tile, called the *misdirection filter* (MDF) that tracks the state of the accelerator's eL1D. The MDF augments the L2 (see Fig. 2) and allows the processor and accelerator to safely share data and transfer ownership within the tile, *without any modification to the baseline directory coherence protocol* at the LLC. The L2 and eL1D maintain coherence between themselves and coordinate responses to LLC requests, leveraging the MDF to eliminate unnecessary coherence messages.

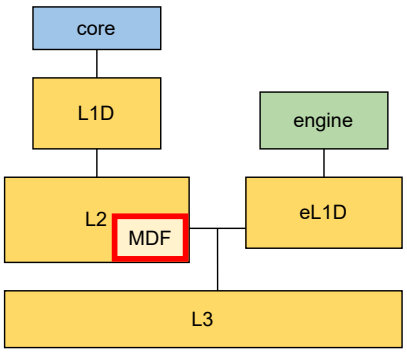


Fig. 2: Proposed cache hierarchy design with the mis-direction filter (MDF) augmenting the L2 cache.

In brief, Kobold is a new type of hierarchical coherence. The main difference from prior hierarchical coherence protocols is that Kobold requires negligible additional state and is non-inclusive to prevent the engine from polluting its local L2 bank. Moreover, by leveraging fast, local communication within a tile, Kobold avoids unnecessary coherence traffic to the LLC and minimizes the latency of both hits and misses in the eL1D and L2. However, to maintain coherence between the core and engine, the L2 must track the contents of the eL1D — this is the role of the MDF. The resulting design is a new twist on hierarchical coherence.

Summary of results: We evaluate Kobold in the context of the täkō [46] architecture by modifying a baseline MESI coherence protocol. The resulting design eliminates abundant communication to the LLC compared to a näive directory-based protocol. We implement Kobold's coherence protocol in Murphi, without changing the baseline LLC, to verify correctness. And we estimate Kobold's area overhead (from the MDF) at just 0.00076mm^2, or 0.09% of the baseline caches.

II. Background and Motivation

A. Near-Data Computing

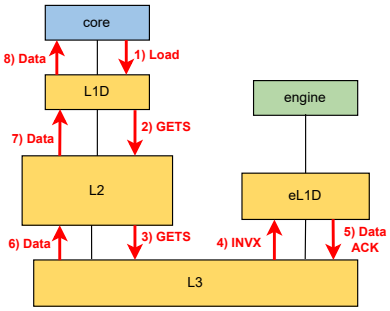
To minimize data movement, many architectures propose moving processing logic closer to data, rather than moving data to compute. Some designs propose "processing in-memory" architectures that place compute logic in memory [11, 18, 22, 23, 24, 27, 35, 37, 39, 47, 49]. Other designs propose "near-data accelerators" which place co-processors off-chip close to main memory [3, 5, 7, 8, 16, 17, 21, 42, 55, 56]. Co-processors that are integrated near main memory provide benefits for streaming applications, but they are inappropriate for applications with data reuse or fine-grained communication [2, 21, 29, 51, 55].

For applications with significant locality or frequent data sharing between core and accelerator, others propose integrating accelerators within the cache hierarchy, allowing CPUs to offload work to caches [1, 2, 29, 40, 46, 50, 54]. Cache-attached accelerators benefit from sharing a unified address space with the host cores, eliminating the need to control low-level data movement in software [48]. However, accessing the shared memory of the host core requires these accelerators to maintain coherence with the rest of the system.

B. Coherence and Consistency

1) Directory-Based Coherence Protocols: Directory-based protocols use a directory structure to track which caches hold a block and in which state [34]. For any coherence request, the directory determines the actions to be taken based on the current location(s) and state of the block. Directory protocols are popular for modern CMPs where the shared last-level cache (LLC) tracks coherence across the private caches of each core.

Naïvely extending directory-based coherence to support cache-attached accelerators does not work well. Fig. 3 shows an architecture where the engines' eL1Ds are treated as additional sharers under the LLC.



Step	L1D	eL1D	L2	L3
Init)	I	M	I	M
1)	I	M	I	M
2)	I	M	I	M
3)	I	M	I	M
4)	I	M	I	M
5)	I	S	I	S
6)	I	S	S	S
7)	S	S	S	S
8)	S	S	S	S

Fig. 3: Naïve architecture where the engines' eL1Ds are treated as additional sharers under the LLC. Example transaction for core read request when the eL1D holds the data in the modified (M) state. 'I' represents Invalid, and 'S' represents Shared.

This example shows a common access pattern where a core reads data produced by an engine. In Fig. 3, the modified cache line is stored in state M (modified) in the engine's eL1D. When the core issues a load to the line, the request must propagate down to the LLC and back up to the eL1D in order to revoke modified permission from the eL1D, and then send the data back through the LLC to the requesting core. This is quite wasteful because the core and engine reside on the same tile, and the LLC directory can be across the chip. It also increases directory overheads in the LLC by doubling the number of sharers. To alleviate these issues, other types of coherence protocols have been proposed.

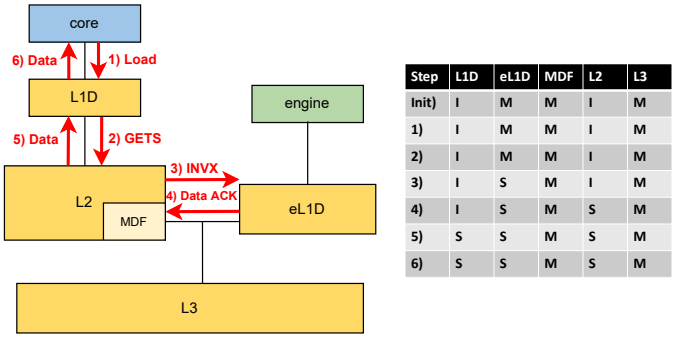
2) Hierarchical Coherence Protocols: Directory-based protocols face scaling challenges due to the storage required to track all caches and additional on-chip network traffic [32]. To improve scalability, multicore chips can be organized into hierarchies of shared buses and caches with multiple levels of on-chip directories. Intermediate levels of the hierarchy serve as directories for the lower levels, tracking their state and filtering traffic to the lower levels. The use of intermediate directories reduces the storage overhead by only requiring tracking of clusters, rather than tracking of each individual node [31]. Additionally, locality can enable the majority of transactions to be performed within a cluster, reducing network congestion by limiting the number of requests to the last-level cache.

The DASH [26] architecture enables scalability in CMPs by mitigating the bottlenecks of directory-based systems. They propose distributing the main memory and directory among

clusters throughout the system. To maintain coherence, DASH utilizes two coherence protocols: a snooping-based intra-cluster protocol and a directory-based inter-cluster protocol. All private data references can be localized to the cluster, reducing the need to access the on-chip network or directory. While more scalable than a flat protocol, the mixed coherence policy adds significant verification complexity [10].

Ros et al. [44] introduce the *Hierarchical Private/Shared Classification* method that enables data blocks to be shared entirely within a cluster but appear as private from outside the cluster. To eliminate complex recursive functions in the hierarchy, they implement simple coherence mechanisms such as self-invalidation and write-through that operate entirely within the sharing cluster. To implement the protocol, classification of clusters is performed dynamically using the page tables to detect and track the sharing level.

Kobold is also a hierarchical cache-coherence protocol that, like DASH, uses a combination of local snooping and directories to improve scalability and limit coherence traffic. Fig. 4 revisits the earlier example from Fig. 3, but using Kobold's hierarchical design.



Step	L1D	eL1D	MDF	L2	L3
Init)	I	M	M	I	M
1)	I	M	M	I	M
2)	I	M	M	I	M
3)	I	S	M	I	M
4)	I	S	M	S	M
5)	S	S	M	S	M
6)	S	S	M	S	M

Fig. 4: Kobold architecture where the L2 tracks eL1D state with an MDF. Example transaction for core read request when the eL1D holds the data in the modified (M) state.

As mentioned earlier, Kobold augments the L2 cache with a mis-direction filter (MDF) that tracks the state of the eL1D. Since the MDF knows that the local eL1D holds the block in state M, the core's load request can be handled *without involving the LLC*. The eL1D downgrades and responds to the L2 *locally*, leaving both in state S (shared), while the MDF continues tracking the tile as M, delaying write-back of dirty data, since the line is still dirty within the tile.

3) Cache Inclusion: One of the key design choices when building a multi-level cache hierarchy is whether to enforce inclusion. Inclusive caches benefit from snoop filtering (e.g., in an L2/LLC cache system, coherence requests do not require an L2 lookup if a miss occurs in LLC). However, since the LLC is inclusive of the L2, data is duplicated in both caches, reducing effective cache size, and data brought in by the eL1D can remain in the L2 long after it is evicted from the engine. For example, täkō [46] observed a $>4\times$ slowdown from L2 cache pollution on some benchmarks. Cache bypassing has been demonstrated to also significantly improve the miss rate for workloads with transient data [43].

NCID [57] proposes a non-inclusive cache, inclusive-directory design. To increase the space efficiency of the cache, data in the LLC is non-inclusive of the L2. However, the LLC retains tag inclusion in the directory to support complete snoop filtering. An independent tag/directory structure is maintained in the LLC cache to track additional addresses with no data and is exclusive of the main directory.

Kobold takes a similar approach as NCID by implementing the L2 as non-inclusive of the eL1D and integrating an additional directory structure (MDF) within the L2 to enable snoop filtering for requests coming from the LLC. However, Kobold differs in several areas. First, the MDF is not exclusive of the L2; tags can exist in both the MDF and the L2 at the same time. Furthermore, the MDF is used to determine coherence messages for requests originating from both the LLC *and* the core. In cases where both the L2 and eL1D share data that is modified but has not yet been written back (see Fig. 4), the MDF reflects the overall state of the tile rather than the eL1D.

4) Coherence and Consistency for Heterogeneous Systems: The trend toward heterogeneous architectures has led to tighter integration of accelerators and devices with shared memory. Inter-device communication in heterogeneous architectures is a major bottleneck that has motivated the adoption of a unified coherent address space. Allowing the host and devices to share a single, coherent address space greatly improves inter-device communication and simplifies programming [45]. Additionally, making accelerators coherent with the rest of the system allows accelerators to efficiently share data with the system. However, ensuring that shared memory remains coherent is a major challenge due to the diverse memory demands and coherence properties of accelerators.

Direct memory access (DMA) engines have become a popular option to maintain coherence between accelerators and host cache hierarchies [30]. DMA engines transfer data directly between the LLC and the accelerator caches. While efficient for accelerators with little data sharing, the DMA-based approach is undesirable for accelerators that share memory with the host or other accelerators due to excessive data transfers.

Maintaining coherence for near-data accelerators (NDA) that reside off-chip close to main memory is a major challenge due to communication costs with the CPU and high levels of required off-chip data movement. CoNDA [9] is a recent coherence mechanism that allows NDAs to optimistically execute kernels to gather information on memory accesses and uses this information to avoid performing unnecessary coherence requests. Spandex [4] is a coherence interface that efficiently supports integrating a variety of devices with divergent memory access patterns and diverse coherence properties into a single address space. Essentially, Spandex allows devices with much different coherence protocols to efficiently share a single unified address space. These designs target discrete co-processors with expensive communication between cores and accelerators.

Overall, we find that prior protocols for heterogeneous systems do not work well for cache-attached accelerators because they assume minimal communication between the

core and accelerator. This assumption motivates designs which perform well for coarse-grain communication, but not the fine-grain communication commonly exhibited by cache-attached accelerators.

5) Formal Verification: Modern CMPs employ coherence protocols that ensure high performance at the cost of significant verification complexity [34]. To eliminate bugs from these protocols, an exhaustive search of the protocols' state space is required. The exhaustive nature of coherence protocol verification dictates that the overhead costs, memory required, and verification time grow very fast with respect to the number of processors and the complexity of the protocol [41].

Murphi [15] is a commonly used verification tool that utilizes enumerative state checking. The Murphi model checker verifies the specified system by enumerating all possible states and checking them against a set of invariants while ensuring the protocol never causes system deadlock. We implement Kobold's coherence protocol in Murphi to verify correctness.

III. Kobold Design and Implementation

We consider a chip-multiprocessor (CMP) where each tile contains a core, private L1D/L1I, private L2, shared LLC bank, and cache-attached accelerator with its own private eL1D (see Fig. 1). To avoid adding state and coherence complexity to the LLC, all modifications to support the accelerator's eL1D are confined within its tile.[2] Similar to prior hierarchical coherence protocols, the eL1D is logically a child of the L2, alongside the core's L1D/L1I, and the L2 is responsible for maintaining coherence between the core and accelerator. But unlike prior protocols, the L2 is not inclusive of the eL1D, and the L2 and eL1D operate as peers via snooping to handle many coherence transactions.

A. Kobold Cache Hierarchy

In Kobold, additional coherence complexity and state is restricted to the L2 and eL1D. The L2's responsibilities are to *(i)* maintain coherence between its local L1D and eL1D banks, and *(ii)* prevent the accelerator from polluting the L2 bank. Kobold's design enforces these requirements with minimal overheads by augmenting the L2 with a small directory structure called the mis-direction filter (MDF).

Fig. 2 shows Kobold's cache hierarchy. The eL1D and the core's L1D (and L1I, not shown) are logically children of the L2 cache, as far as coherence is concerned. However, in many ways the eL1D operates as a peer cache of the L2 to, e.g., avoid polluting the L2 with data accessed by the engine. Interaction between the L2 and eL1D is mediated by the MDF.

Mis-direction filter: The MDF tracks the contents of the eL1D. It is a metadata-only array that maintains the tags and coherence state of data in the eL1D, but does not track the data itself. (The MDF tracks coherence state for the entire tile, which may diverge from the state in the eL1D, as in Fig. 4.)

Fig. 5 shows the detailed microarchitecture of the L2 in Kobold. Ignoring the MDF, the operation of the main L2 tag and data arrays is unchanged from a baseline, inclusive, unified L2 cache in a traditional CPU cache hierarchy: e.g., data is inserted into the L2 tag and data arrays upon a L1D miss (i.e., from a CPU request). However, to ensure coherence between the L2 and eL1D, the MDF is accessed in parallel with the main L2 tag array to determine the appropriate coherence action. Using the MDF to track the eL1D tags in the L2 enables Kobold to perform snooping-like logic on-demand with no performance overhead (the MDF is much smaller than the L2 tags and lower latency). If a line is cached in the eL1D, metadata for the line will be tracked in both the eL1D tags and MDF, and the state in the MDF will determine whether a memory transaction can be handled locally within the tile or if the LLC must be contacted to, e.g., upgrade permissions.

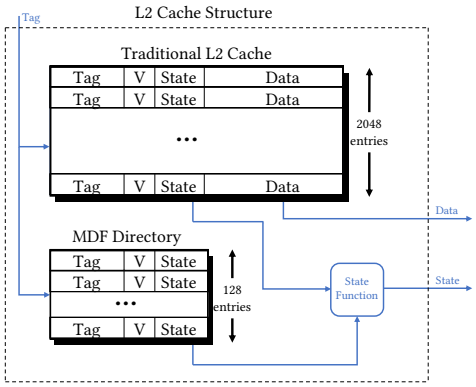


Fig. 5: Microarchitecture diagram of L2 cache. The L2 determines coherence actions by concurrently checking the L2 directory and MDF.

Avoiding L2 pollution: Finally, the MDF is key to enabling coherence for cache-attached accelerators without disrupting core performance. Prior work has demonstrated that, with a conventional inclusive cache hierarchy, cache-attached accelerators can cause severe cache pollution by streaming data into the L2 that evicts the core's working set, slowing down cores by $>4\times$ [46]. The MDF achieves a similar objective without modifying the L2 replacement policy or inserting data into the L2 at all: Kobold tracks the eL1D contents in the MDF and never inserts data into the L2 unless it is accessed directly by a core. When data is evicted from the eL1D, it is simultaneously evicted from the MDF, and the L2 contents are unaffected (though permissions may be upgraded, depending on the state in the MDF; see Fig. 8 below). Kobold thus eliminates L2 pollution by design.

B. Cache Coherence Protocols

Kobold's coherence protocols for the eL1D and L2 are designed for the unique structure of a cache-attached accelerator's hierarchy. Fig. 6 and Fig. 7 highlight the differences between the finite state machines of the baseline MESI protocol and the Kobold protocol. We omit the L2 finite state machine due to its size.

The Kobold protocol introduces peer-to-peer communication between the eL1D and L2 that allow the caches to maintain coherence between themselves and coordinate responses to LLC

[2]Sec. III-B3 discusses how to improve performance with minor changes to the LLC protocol.

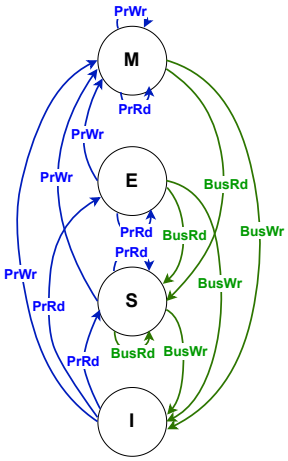


Fig. 6: Finite state machine of baseline MESI coherence protocol.

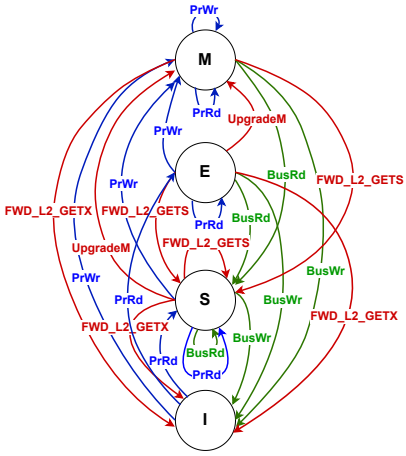


Fig. 7: Finite state machine of Kobold eL1D cache controller. Additional intra-tile messages (red arrows) are necessary to allow the eL1D and L2 to maintain coherence between themselves.

requests. Peer-to-peer communication allows the tile caches to transfer or share data between themselves.

For example, `FWD_L2_GETX` messages in Fig. 7 are sent from the L2 to the eL1D when the L2 requests exclusive access to data in the eL1D. The eL1D responds with data and changes its coherence state if necessary. Furthermore, peer-to-peer communication allows the tile caches to transfer ownership. For example, `UpgradeM` messages in Fig. 7 are sent from the L2 to the eL1D when the L2 evicts data that both caches share but is tracked as modified by the MDF. This eliminates redundant coherence traffic to the LLC by allowing the L2 and eL1D to change states while the overall *tile* state remains unchanged from the LLC's perspective.

1) Handling LLC requests: Requests from the LLC (i.e., downgrades or invalidations) to the tile are broadcast to both the eL1D and L2 caches. We ensure that only one of the caches, usually the L2, responds to LLC coherence requests. To enable this, the L2 protocol utilizes the MDF to determine when it must wait for the eL1D to complete an LLC request before responding to the LLC. Upon completing the LLC request, the eL1D sends an acknowledgement to the L2 cache. Following this acknowledgement, the L2 cache can respond to the LLC if needed.

In transactions requiring a data response or writeback, the L2 cache services requests when it can. However, when the eL1D holds the only copy of data, the eL1D will respond. To ensure there is only one cache writing back at a time, the L2 cache prompts the eL1D to issue the response itself.

2) Handling core requests: Each time a core-issued request reaches the L2, the L2 and MDF are searched in parallel. The L2 cache controller uses both results to determine how to proceed (see Fig. 5).

If the L2 cannot service the request but the MDF holds the line with the requested permissions, the request is forwarded to the eL1D cache which supplies the data. The eL1D responds to the request and downgrades its state if necessary. Upon receiving a response from the eL1D, the L2 updates its local state as well as the MDF to reflect any changes to the eL1D.

As demonstrated in Fig. 4, in the case that the line is not found in the L2 and the MDF holds the line with higher permissions (M or E) than what is requested (S), the request is satisfied in a similar manner. However, during the transaction the eL1D downgrades, leaving both the eL1D and L2 in the same state (S) while the MDF maintains its original state. The state of the MDF now reflects the overall state of the tile (M), rather than the state of the eL1D. This mechanism avoids any involvement of the LLC when a core and its local engine access the same data.

If the line is found in the MDF but the request requires higher permissions (e.g., MDF holds the line in shared state but the request requires it in modified state), concurrent requests are sent to the eL1D and the LLC. The eL1D supplies the data to the L2 and transfers its permissions to the L2. An LLC request is sent in parallel to obtain the permissions required to satisfy the initial request. After receiving an acknowledgement from the LLC, the L2 finally upgrades to the required permission level and can satisfy the request.

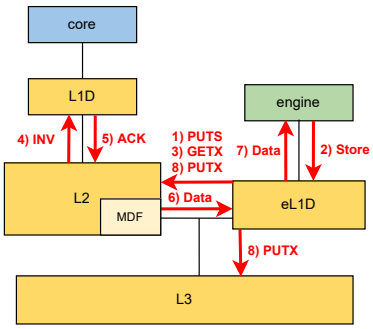
If the line is not found in the L2 or MDF, the request is sent directly to the LLC. Using the MDF to determine that the eL1D does not have the data ensures the L2 does not send an unnecessary request to the eL1D. Rather, the L2 immediately sends a request for the data to the LLC, ensuring the critical path is *the same as an L2 miss in a baseline CMP*.

3) Handling engine requests: When an engine-issued request misses in the eL1D, the request is first forwarded to the L2 cache. In the case that the L2 cache can service the request fully, data is transferred, the L2 is downgraded if necessary, and the MDF state is updated to reflect the new eL1D state (see steps 2-7 in Fig. 8).

If the L2 holds the block in a higher state than requested, the L2 downgrades and sets the MDF to its original state. The L2 and eL1D caches share the data but the MDF and LLC track the data as exclusive.

If the L2 cannot service the request, it informs the eL1D and the eL1D sends the request to the LLC. When the LLC responds, a state change is sent to the L2 cache to update the state of the MDF before the eL1D completes the request.

Optional Optimization – eL1D Speculative Loads: So far,



Step	L1D	eL1D	MDF	L2	L3
Init)	S	S	M	S	M
1)	S	I	I	M	M
2)	S	I	I	M	M
3)	S	I	I	M	M
4)	I	I	I	M	M
5)	I	I	M	I	M
6)	I	M	M	I	M
7)	I	M	M	I	M
8)	I	I	I	I	M

Fig. 8: Continuation of the example from Fig. 4 where the data is evicted from the eL1D (step 1), stored by the engine (steps 2-7), and finally evicted from the eL1D again (step 8).

Kobold adds L2 latency to the critical path of eL1D misses. This is to prevent requests arriving at the LLC directory while the L2 already has a valid copy of the data. However, for applications in which engines and cores share little data (a common case), the additional L2 latency on every eL1D miss can severely harm performance.

Instead, the eL1D can speculatively forward a request to the LLC in parallel with the L2 request to hide the L2 latency. In some coherence protocols, the LLC assumes that a tile will not request data if the LLC thinks the tile already has the data. Thus, if the eL1D forwards a speculative request to the LLC when the L2 already has the data, this can result in unexpected behavior at the LLC.

Protocols with silent drops allow the LLC to receive a request for data that it thinks the requester already has. In such designs, private caches drop clean data on eviction without notifying the LLC, so the LLC doesn't know whether a tile really has the data. These protocols allow the LLC to gracefully handle speculative eL1D requests if the data is clean in the L2.

However, if the data is dirty in the L2, the tile cannot have silently dropped it, since dirty data must be written back to the LLC on eviction. Kobold thus modifies the LLC protocol to ignore redundant requests to data already shared by the requesting tile, assuming that the L2 on that tile will handle the request.

4) Handling evictions: When the L2 replaces a block, it first checks the state in the L2 directory and the MDF. If only the L2 cache holds the line, the L2 issues a PUT request to the LLC. However, if the MDF also holds the tag (i.e., the eL1D has the data), the L2 *silently drops* the data. If the MDF tracks the data in an exclusive state while the L2 holds the data in shared, the eL1D state is upgraded to that of the MDF and the L2 drops its copy.

If the eL1D replaces a block in a private state, it concurrently issues a PUT request to the LLC and informs the MDF that it replaced the line (see step 8 in Fig. 8). However, when the eL1D replaces a block in the shared state, more indirection is required. First, the eL1D checks if the L2 cache holds the line. If the L2 does not hold the data, the MDF is invalidated and the L2 triggers the eL1D to issue PUT request to the LLC. However, if the L2 holds the data, the eL1D silently drops the data, the MDF is invalidated, and the L2 is upgraded to reflect the previous state of the MDF if necessary (see step 1 in Fig. 8).

IV. Evaluation

Verification: We used the Murphi model checker [15] to formally verify Kobold's stable-state protocols. We made the model transaction-atomic based on the methodology described in [36]. Our Murphi model verified Kobold's protocols against the single-writer, multiple-reader invariant and proved deadlock-freedom. During verification, Murphi explored a total of 12,534 states.

Area: We used Cacti 7 to evaluate the area requirements of the MDF [6]. We base our evaluation on parameters used in täkō [46]. We evaluate a system with a 128KB L2, 8KB eL1D, and 512KB LLC per tile. In 22nm, Cacti estimates the L2 size as 0.2706mm^2, the LLC bank as 0.5963mm^2, and the MDF size as 0.00076mm^2. Compared against the baseline area of the L2 cache and LLC bank, the MDF adds an area overhead of only 0.09%.

V. Conclusion & Future Work

In this era of memory-hierarchy specialization and heterogeneous architectures, ease of integration is vital for incorporating specialized hardware like cache-attached accelerators. Even in homogenous systems, cache coherence is a challenging mechanism to correctly implement and verify. To integrate cache-attached accelerators with minimal impact on coherence complexity and system overhead, we introduced the Kobold coherent protocol. By keeping additional coherence actions local to a single CMP tile, Kobold significantly simplifies accelerator integration, minimizes on-chip network traffic, and avoids impacting baseline processor performance.

Moving forward, we plan to generate the fully concurrent protocols, i.e. the transient states and transitions required for concurrency. We plan to use the HieraGen tool [38] which generates fully concurrent protocols for hierarchical cache systems based on input of correct stable state protocols for each level of the cache hierarchy.

References

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proc. of the 23rd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-23)*, 2017.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*, 2015.

[3] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked dram," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 131–143.

[4] J. Alsop, M. Sinclair, and S. Adve, "Spandex: A flexible interface for efficient heterogeneous coherence," in *Proc. of the 45th annual Intl. Symp. on Computer Architecture (Proc. ISCA-45)*, 2018.

[5] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[6] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, 2017.

[7] B. Black, "Die Stacking is Happening!" in *MICRO-46 Keynote*, 2013.

[8] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII)*, 2018.

[9] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, R. Ausavarungnirun, K. Hsieh, N. Hajinazar, K. T. Malladi, H. Zheng, and O. Mutlu, "Conda: Efficient cache coherence support for near-data accelerators," in *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46)*, 2019.

[10] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou, "Hierarchical cache coherence protocol verification one level at a time through assume guarantee," in *2007 IEEE International High Level Design Validation and Test Workshop*, 2007.

[11] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[12] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.

[13] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, 2012.

[14] W. J. Dally, "GPU Computing: To Exascale and Beyond," in *Supercomputing '10, Plenary Talk*, 2010.

[15] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol verification as a hardware design aid," in *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, 1992.

[16] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. of the 24th Intl. Conf. on Parallel Architectures and Compilation Techniques (Proc. PACT-24)*, 2015.

[17] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *Proc. of the 22nd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-22)*, 2016.

[18] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, 1995.

[19] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *Turing Award Lecture*, 2018.

[20] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[21] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[22] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, , and J. Torrellas, "FlexRAM: Towards an intelligent memory system," in *Proc. of the 17th Intl. Conf. on Computer Design (Proc. ICCD)*, 1999.

[23] P. M. Kogge, "EXECUBE-A new architecture for scaleable MPPs," in *Proc. of the intl conf. on Parallel Processing (ICPP)*, 1994.

[24] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *Computer*, vol. 30, no. 9, 1997.

[25] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, 2020.

[26] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," 1990.

[27] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 173.

[28] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013.

[29] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXV)*, 2020.

[30] S. Ma, L. Huang, Y. Lei, Y. Guo, and Z. Wang, "An efficient direct memory access (dma) controller for scientific computing accelerators," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019.

[31] Y.-C. Maa, D. K. Pradhan, and D. Thiebaut, "Two economical directory schemes for large-scale cache coherent multiprocessors," *SIGARCH Comput. Archit. News*, 1991.

[32] M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, 2012.

[33] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "Snnap: Approximate computing on programmable socs via neural acceleration," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[34] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on memory consistency and cache coherence*, 2020.

[35] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-machine multicomputer: an architectural evaluation," in *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993.

[36] T. Olausson, "Towards the automatic synthesis of cache coherence protocols," Ph.D. dissertation, 2020.

[37] M. Oskin, F. Chong, and T. Sherwood, "Active pages: A model of computation for intelligent memory," in *Proc. of the 25th annual Intl. Symp. on Computer Architecture (Proc. ISCA-25)*, 1998.

[38] N. Oswald, V. Nagarajan, and D. J. Sorin, "Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[39] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[40] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic computing in gpu architectures," in *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46)*, 2019.

[41] F. Pong and M. Dubois, "Verification techniques for cache coherence protocols," *ACM Comput. Surv.*, 1997.

[42] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, and V. Srinivasan, "NDC: Analyzing the Impact of 3D-Stacked Memory + Logic Devices on MapReduce Workloads," in *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[43] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of the 34th annual Intl. Symp. on Computer Architecture (Proc. ISCA-34)*, 2007.

[44] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[45] N. Sakharnykh, "Beyond gpu memory limits with unified memory on pascal," https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/, 2016.

[46] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement," in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (Proc. ISCA-49)*, 2022.

[47] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*, 2016.

[48] Y. S. Shao and D. Brooks, *Research infrastructures for hardware accelerators*, 2016.

[49] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-24)*, 2018.

[50] A. Subramaniyan, J. Wang, E. R. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proc. of the 50th*

annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50), 2017.

[51] P.-A. Tsai, C. Chen, and D. Sanchez, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-51)*, 2018.

[52] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, 2010.

[53] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *Proc. of the 44th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-44)*, 2011.

[54] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream floating: Enabling proactive and decentralized cache optimizations," in *Proc. of the 27th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-27)*, 2021.

[55] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "Top-pim: Throughput-oriented programmable processing in memory," in *Proc. HPDC*, 2014.

[56] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-24)*, 2018.

[57] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, "Ncid: A non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, 2010.