Kobold: Simplified Cache Coherence for Cache-Attached Accelerators

Jennifer Brana, Brian C. Schwedock, Yatin A. Manerkar, Nathan Beckmann

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 accelerators 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 the integration of cache-attached accelerators, we present Kobold, a new coherence protocol and implementation which restricts the added complexity of an accelerator to its local tile. Kobold introduces a new directory structure within the L2 cache to track the accelerator's private cache and maintain coherence between the core and accelerator. A minor modification to the LLC protocol also enables accelerators to improve performance by bypassing the local L2. We verified Kobold's stable-state coherence protocols using the Murphi model checker and estimated area overhead using Cacti 7. Kobold simplifies integration of cache-attached accelerators, adds only 0.09% area over the baseline caches, and provides clear performance advantages vs. naïve extensions of existing directory coherence protocols.

Index Terms—Cache coherence, near-data computing, data-centric

1 Introduction

Computer systems are increasingly bottlenecked by the rising cost of data movement. To combat this trend, near-data computing (NDC) designs propose accelerators that move compute closer to data. *Cache-attached accelerators* are a promising direction for NDC that enables fine-grain collaboration between cores and accelerators by offloading work to within the CPU cache hierarchy.

Fig. 1 shows täkō [12], a representative recent system with cache-attached accelerators. täkō augments a baseline, cache-coherent multicore with an *engine* (i.e., accelerator) on each tile, granting the engine efficient access to data in the tile's L2 and last-level cache (LLC) banks. Each engine also has its own private 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.

To complicate matters further, accelerators desire access to different levels of the cache hierarchy, depending on the application [8, 12]. Applications with frequent accelerator-to-core communication (e.g., irregular prefetchers) want the accelerator to sit beneath the core's private L2, but applications operating over large datasets (e.g., graph search) want the accelerator to access the LLC directly, without polluting or waiting for the L2. Systems should therefore provide cache-attached accelerators with efficient access to *both* the L2 and LLC, but this is not supported by existing coherence protocols.

• *Jennifer Brana is at the University of Portland and Carnegie Mellon University. Brian C. Schwedock and Nathan Beckmann are at Carnegie Mellon University. Yatin A. Manerkar is at the University of Michigan.*

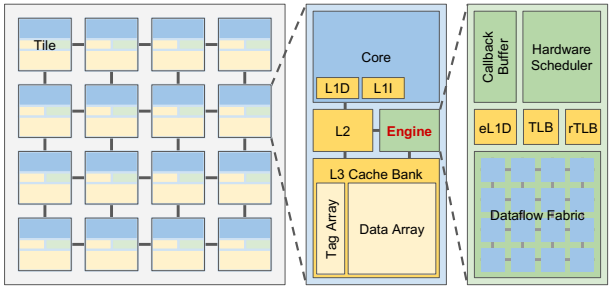
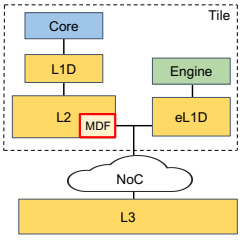


Fig. 1: täkō [12] adds a reconfigurable engine (i.e., accelerator) to each tile of a CMP. Engines accelerate tasks for data that resides in the L2 or LLC bank on that tile. Each engine has a coherent eL1D cache.

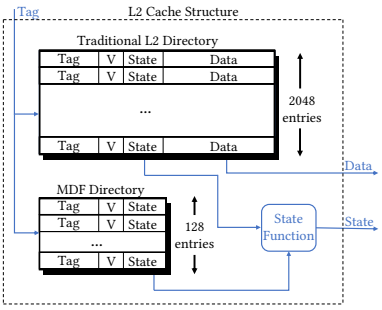
Insights: The complexity added to the shared LLC protocols caused by cache-attached accelerators can be mitigated if the accelerator's eL1D and its local L2 bank look like a single, unified cache to the LLC. This is achievable by adding extra state within each tile of the chip-multiprocessor (CMP) to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

However, just making the eL1D a child of the L2, as in traditional hierarchical coherence designs, can harm performance for systems in which the accelerator would rather sit beneath the LLC. L2 pollution can be mitigated by replacement policies [12], but going through the L2 still incurs unnecessary latency [8]. Consequently, we propose a design in which data accessed only by the accelerator bypasses the L2.

Approach: Our goal is to design a coherence protocol which ***(i)*** restricts the complexity of cache-attached accelerators to *within each tile* of a CMP and ***(ii)*** befits accelerators independent of which cache level they want to access. Our solution, Kobold, adds a directory-like structure to each tile, called the *mis-direction filter* (MDF) that tracks the state of the accelerator's eL1D. The MDF augments the L2 (see Fig. 2a) and allows the processor and accelerator to safely share data and transfer ownership within the tile, *with minimal 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 reduce unnecessary messages.



(a) Kobold's cache hierarchy design. A *mis-direction filter* (MDF) augments the L2 in each tile.



(b) Microarchitecture of the L2 and MDF.

Fig. 2: Architecture of the Kobold system.

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

Summary of results: We evaluate Kobold in the context of the täkō [12] architecture by modifying a baseline MESI coherence protocol. Our design significantly reduces communication to the LLC,

compared to a näive directory protocol. We verify Kobold's stable-state coherence protocol in Murphi [5]. We also estimate Kobold's area overhead (from the MDF) at just 0.09% of the baseline caches.

2 Background and Motivation

2.1 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 [6]. Others propose "near-data accelerators" (NDAs) which place co-processors off-chip close to main memory [2]. NDAs benefit streaming applications, but are inappropriate for applications with data reuse or fine-grained sharing.

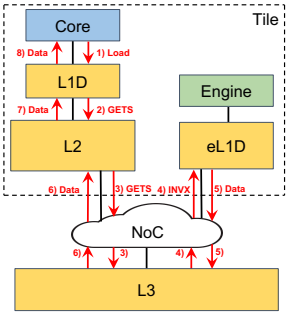
For applications with significant locality or frequent data sharing between core and accelerator, others propose integrating accelerators within the cache hierarchy, allowing CPUs to offload work to caches [8, 12]. Cache-attached accelerators share a unified address space with the host cores, eliminating the need to control low-level data movement in software. However, accessing the shared memory of the host core requires accelerators to maintain coherence.

2.2 Coherence and Consistency

2.2.1 Directory-Based Coherence Protocols

Directory-based protocols use a directory structure to track child caches. For coherence requests, the directory determines the actions required based on the current location(s) and state of the block.

However, *naïvely extending directory-based coherence to support cache-attached accelerators does not work well.* Fig. 3 shows an architecture where the accelerators' eL1Ds are additional sharers under the LLC. In this example, all transactions between the core and accelerator pass through the LLC, and data is written back to the LLC when transferred. Alternatively, the eL1D and L2 caches could directly forward data to each other when prompted by the LLC. This optimization eliminates data transfer through the LLC, but the request must still go through the LLC to update the directory. Unfortunately, *any* communication with the LLC is quite wasteful because the core and accelerator reside on the same tile, while the LLC directory can be across the chip. This naïve design also doubles LLC directory state to track twice as many sharers. To alleviate these issues, other types of coherence have been proposed.



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 accelerators' 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.

2.2.2 Hierarchical Coherence Protocols

Directory-based protocols face scaling challenges due to the storage required to track all caches and on-chip network traffic. To improve scalability, multicore chips can be organized into hierarchies of caches with multiple levels of directories. Intermediate levels of the hierarchy serve as directories for the lower levels, reducing storage overhead in the LLC. Additionally, locality enables the majority of transactions to be performed within a cluster, reducing traffic to the LLC.

The DASH [7] architecture improves scalability by mitigating the bottlenecks of directory protocols. To maintain coherence, DASH utilizes two coherence protocols: a snooping-based intra-cluster protocol and a directory-based inter-cluster protocol. Private data references are localized to the cluster, reducing accesses to the directory.

Kobold is also a hierarchical coherence protocol that, like DASH, uses a combination of local snooping and directories to improve scalability and limit coherence traffic. However, unlike DASH, Kobold does not add an intermediate cache on the critical path to arbitrate remote accesses. Instead, Kobold uses peer-to-peer communication to maintain intra-tile coherence via the MDF, and the accelerator can speculatively bypass the L2 to access the LLC directly.

2.2.3 Cache Inclusion

A key design choice when building a multi-level cache hierarchy is whether to enforce inclusion. Inclusive caches benefit from snoop filtering. However, inclusion leads to data duplication, reducing effective cache size. Additionally, data brought in by the eL1D can remain in the L2 long after it is evicted from the eL1D.

Kobold starts from a baseline inclusive protocol. However, Kobold implements the L2 as *non-inclusive* of the eL1D and integrates an additional directory (MDF) within the L2 to enable snoop filtering. 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. Finally, the MDF holds a copy of the eL1D tags but tracks the overall state of the tile, as discussed below (see Fig. 4).

2.2.4 Coherence and Consistency for Heterogeneous Systems

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 accelerator to share a single, coherent address space greatly improves inter-device communication and simplifies programming. However, ensuring that shared memory remains coherent is a major challenge due to the diverse memory demands and coherence properties of accelerators.

Recent coherence protocols target discrete co-processors located near memory, where communication is expensive between cores and accelerators. CoNDA [4] is a recent coherence mechanism that allows NDAs to optimistically execute kernels to gather information on memory accesses. It uses this information to avoid unnecessary coherence requests. Spandex [1] 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.

Overall, we find that *prior protocols for heterogeneous systems do not work well for cache-attached accelerators because they assume infrequent communication between the core and accelerator.* This assumption does not hold for the fine-grain communication commonly exhibited by cache-attached accelerators.

2.2.5 Formal Verification of Coherence Protocols

Modern CMPs employ coherence protocols that ensure high performance at the cost of significant verification complexity. To eliminate bugs from protocols, an exhaustive search of the protocols' state space is required. Verification overheads typically grow very fast with respect to protocol complexity, so it is desirable to limit the additional coherence-related complexity of the accelerators' caches to their respective tiles.

3 Kobold Design and Implementation

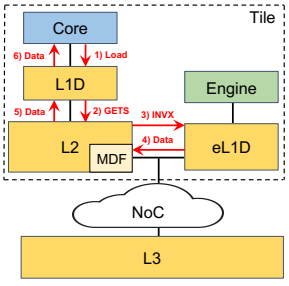
We consider a chip-multiprocessor (CMP) where each tile contains a core, private L1D/L1I, private L2, shared LLC bank, and cache-attached accelerator with its own private eL1D (see Fig. 1). To avoid

adding state and coherence complexity to the LLC, Kobold confines nearly all modifications to within a tile. 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, enabling the accelerator to access either the L2 or LLC "directly".

3.1 Overview

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



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.

Mis-direction filter: The MDF tracks the contents of the eL1D. It is a metadata-only array that maintains only the tags and coherence state for data in the eL1D. (The MDF tracks coherence state for the entire tile, which may diverge from the state in the eL1D, as in Fig. 4.)

Fig. 2b shows the microarchitecture of the L2 in Kobold. Ignoring the MDF, the operation of the main L2 tag and data arrays is unchanged from a baseline CPU cache hierarchy: e.g., data is inserted into the L2 tag and data arrays upon a L1D miss. However, to ensure coherence between the L2 and eL1D, the MDF is accessed in parallel with the main L2 tags to determine the 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 is accessed in parallel.) 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.

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 $>4\times$ in some cases [12]. 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, its tag 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. 6 below). Kobold thus eliminates L2 pollution by design.

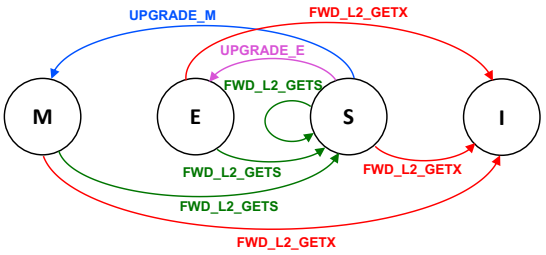


Fig. 5: Finite-state machine for eL1D. Only shows new intra-tile messages in Kobold that enable coherence between the eL1D and L2.

3.2 Kobold's Cache Coherence Protocol

Kobold introduces peer-to-peer communication between the eL1D and L2 that allows the caches to maintain coherence and data within a tile and coordinate responses to LLC requests. Fig. 5 highlights the new MESI state transitions at the eL1D, which are triggered by messages from the L2. The L2 MESI finite state machine is similarly modified in Kobold, but we omit the figure due to space constraints.

New requests fall into two categories: data requests and upgrades. *(i)* A GET request is forwarded from the L2 to eL1D when the core requests data which is located in the eL1D but not L2. The eL1D replies directly with the data and downgrades state if necessary; the LLC is not involved. *(ii)* Upgrades are sent when the L2 evicts data that is also in the eL1D, but the data is tracked with higher permissions in the MDF than eL1D. For example, if the data is tracked as modified (M) in the MDF but tracked as shared (S) in both the L2 and eL1D, when the L2 evicts the data a `FWD_L2_GETX` is sent to the eL1D to upgrade its state to M. This eliminates redundant coherence traffic to the LLC by allowing the L2 and eL1D to change states while the *tile*'s state remains unchanged from the LLC's perspective.

3.2.1 *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 uses 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 services requests when it can. However, when the eL1D holds the only copy of data, the eL1D responds. To ensure only one cache writes back at a time, the L2 cache prompts the eL1D to issue the response itself.

3.2.2 *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. 2b).

If the L2 cannot service the request but the MDF holds the line with the requested permissions, the request is forwarded to the eL1D. 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 in E or M when S is requested, the request is satisfied accordingly. 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 involvement of the LLC when a core and its local accelerator access the same data.

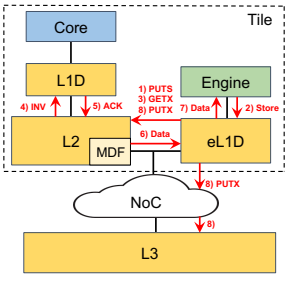
If the line is found in the MDF in state S when M is requested, concurrent requests are sent to the eL1D and the LLC. The eL1D

supplies the data to the L2 and transfers its permissions to the L2. An LLC request is sent in parallel to obtain exclusive permission, and, after receiving an acknowledgement from the LLC, the L2 finally upgrades to M and can satisfy the request.

If the line is not 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*.

3.2.3 Handling accelerator requests

When an accelerator-issued request misses in the eL1D, the request is first forwarded to the L2. In the case that the L2 can service the request fully, data is transferred, the L2 is downgraded if necessary, and the MDF is updated to reflect the new eL1D state (see steps 2-7 in Fig. 6).



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

If the L2 holds the block in E or M when S is requested, the L2 downgrades and sets the MDF to its old state. The L2 and eL1D caches share the data but the MDF and LLC track the data as E or M.

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

"Direct" LLC access via speculative L2 bypass: So far, Kobold adds L2 latency to the critical path of eL1D misses. This is to prevent requests arriving at the LLC while the L2 has a valid copy of the data. However, for applications which desire direct access to the LLC, the additional L2 latency on every eL1D miss can significantly harm performance [8].

Instead, in Kobold, the eL1D can speculatively issue an LLC request in parallel with an L2 request to hide L2 latency. To enable these speculative requests, the LLC needs to handle two additional scenarios: requests for data that the child has *(a)* in a shared state, and *(b)* in an exclusive state. Scenario *(a)* is common in prior protocols which allow the L2 to silently drop clean data. However, scenario *(b)* would not occur in a silent-drop protocol because dirty data cannot be dropped silently. Accordingly, Kobold requires a minor modification to the LLC protocol to ignore redundant requests to data owned by the requesting tile in an exclusive state, since the L2 will handle the request and no response from the LLC is required.

Speculative L2 bypass improves performance at the cost of unnecessary LLC accesses on misspeculation (i.e., when the data is actually in the L2). We observe that bypass can be predicted accurately [11], particularly since cache-attached accelerators often have strong predictors of data's location (i.e., at which level tasks were scheduled to execute [8, 12]).

3.2.4 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 as in E or M while the L2 holds the data in S, 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. 6). However, when the eL1D replaces a block in the S 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 a 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. 6).

4 Evaluation

Verification: We used the Murphi model checker [5] to formally verify Kobold's stable-state protocols. We made the model transaction-atomic based on the method in [9]. Our Murphi model verified Kobold's protocols against the single-writer, multiple-reader invariant and proved deadlock-freedom. During verification, Murphi explored 12,534 states.

Area: We used Cacti 7 [3] to evaluate the area requirements of the MDF. Like täkō [12], we evaluate 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%.

5 Future Work & Conclusion

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 coherence 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. Specifically, we plan to modify the HieraGen tool [10] to support Kobold's non-inclusive hierarchy design.

References

[1] J. Alsop *et al.*, "Spandex: A flexible interface for efficient heterogeneous coherence," in *ISCA*, 2018.
[2] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, 2014.
[3] R. Balasubramonian *et al.*, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM TACO*, 2017.
[4] A. Boroumand *et al.*, "Conda: Efficient cache coherence support for near-data accelerators," in *ISCA*, 2019.
[5] D. Dill *et al.*, "Protocol verification as a hardware design aid," in *VLSI*, 1992.
[6] C. E. Kozyrakis *et al.*, "Scalable processors in the billion-transistor era: IRAM," *IEEE Computer*, 1997.
[7] D. Lenoski *et al.*, "The directory-based cache coherence protocol for the dash multiprocessor," in *ISCA*, 1990.
[8] E. Lockerman *et al.*, "Livia: Data-centric computing throughout the memory hierarchy," in *ASPLOS*, 2020.
[9] T. Olausson, "Towards the automatic synthesis of cache coherence protocols," Ph.D. dissertation, 2020.
[10] N. Oswald *et al.*, "HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols," in *ISCA*, 2020.
[11] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-offs in Architecting DRAM Caches," in *Proc. of the 45th annual IEEE/ACM intl. symp. on Microarchitecture*, 2012.
[12] B. C. Schwedock *et al.*, "täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement," in *ISCA*, 2022.