Kiwi: Dynamic Coherence in Heterogeneous Architecture

Samuel Cheung
skcheun2@illinois.edu

Samuel Lo
glo2@illinois.edu

Shivam Bharuka
bharuka2@illinois.edu

ABSTRACT

Recent trends in heterogeneous architectures have seen a rise in tighter integration between its compute cores. These cores are made up of different compute units such as CPU, GPU and specialized accelerators which exhibit disparate strategy for shared data access. This creates a need for an interface which performs the integration between these different data access strategies. Current solutions are implemented to perform these mappings but lack the flexibility to utilize a more diverse set of coherence strategies inside a single compute unit. In this paper, we introduce Kiwi, a proof-of-concept dynamic coherence framework that adapts to the different memory access patterns in a single program using different coherence strategies. Kiwi uses coherence strategies present in GPU systems on CPU cores depending on the input workload. We evaluate Kiwi and compare it against baseline CPU MSI protocol to show a significant improvement in execution time and network traffic.

KEYWORDS

shared memory, cache coherence, heterogeneous system

ACM Reference Format:
Samuel Cheung, Samuel Lo, and Shivam Bharuka. 2019. Kiwi: Dynamic Coherence in Heterogeneous Architecture. In *Proceedings of ECE511.* 7 pages.

1 INTRODUCTION

Heterogeneous architectures have gained a lot of traction in domains such as mobile devices, IoT and data centers. These architectures are composed of several compute units, such as CPUs, GPUs, FPGAs and specialized accelerators. Coherent shared memory is used for efficient communication between these compute units. Since these underlying devices use different cache coherence strategies to support different types of memory demands, an efficient interface is needed to map requests with different coherence properties. Current solutions include hierarchical coherence structure such as MESI-based protocols [5] which introduces an intermediate cache level for handling incompatibilities and the Spandex protocol [1] which directly interacts with these devices but uses a static mapping between request types. Due to the presence of different coherence strategies in these compute units, there is room to develop a dynamic interface which sends specialized coherence requests based on the data access pattern of a workload.

ECE511, Spring 2019, Champaign, IL
2019.

Kiwi introduces a framework that analyses the data access pattern of a program and finds the best coherence strategy for the specific workload. Currently, it is developed to provide a fine-grain coherence specialization unit for the CPU cores running in heterogeneous architectures. The framework contains (i) a memory access pattern recorder and predictor for different memory access in a single program, (ii) specialized coherence protocols which implement different GPU coherence strategies in CPU, and (iii) a translation unit which switches between these protocols based on the prediction.

In summary, this paper makes the following contributions:

- Trade-offs between ownership vs write through for writes, and writer invalidation vs self invalidation for reads.
- A proof-of-concept demonstrating translation between different coherence strategies in a heterogeneous architecture.

2 BACKGROUND

2.1 Cache Coherence

CPUs and GPUs are designed to handle different types of workloads and thus use different strategies to perform reads and writes for shared memory. CPUs perform writer-initiated invalidations for reads and ownership-based writes whereas GPUs use self-invalidation for reads and write-throughs dirty data to backing cache on writes. Inherently, CPU coherence protocols are designed for workloads where the data exhibits high temporal locality but the demand for memory bandwidth is low. On the other hand, GPU coherence protocols are a good fit for workloads where data exhibits high spatial locality but synchronization is infrequent.

Writer-Invalidation vs Self-Invalidation for Reads:
During writer-initiated invalidation coherence strategy, reads are performed by obtaining shared access which is revoked when an explicit invalidation is sent to the reader. This imposes high latency when the writer demand request increases. On the other hand, self-invalidation doesn't support a shared state and waits for a software hint to invalidate stale data in the local cache.

Ownership vs Write-Through for Writes:
During ownership-based coherence strategy, writes are performed by obtaining exclusive access which requires invalidating all the sharers. This imposes high communication overhead when the reader demand requests are high. On the other hand, write-through doesn't obtain ownership and instead writes the dirty data directly to the backing cache.

2.2 Related Work

Spandex is a flexible interface for cache coherence in heterogeneous architectures. This interface maps different cache coherence state and messages together to allow efficient data movement. However, it performs a static mapping which does not leverage the data access pattern for efficient communication between devices.

In the past, coherence predictors have been designed to reduce the overhead of communication due to shared memory synchronization. Perceptron-based coherence predictor [3] extends a directory-based writer-invalidate protocol to speculate when to push updates to remote nodes. Mukherjee et al. [4] developed a framework which predicts the next coherence message based on the incoming messages for a cache line. Lebeck et al. [2] proposed a system to reduce invalidations sent to directories using version numbers and additional states to determine tear-off copies. However, these works are limited to a homogeneous CPU architectures and cannot be efficiently used to leverage the coherence strategies present in a heterogeneous architecture.

3 SYSTEM DESIGN

This section presents Kiwi, a framework which offers fine-grain coherence specialization by dynamically choosing between different specialized coherence protocols for different memory accesses in a single program.

In this section, we describe the design of Kiwi in further detail. The main components include:

(1) In 3.1, we describe the Kiwi system model which serves as the base of our work.
(2) In 3.2, we describe how Kiwi identifies and predicts the memory access pattern for a workload.
(3) In 3.3, we present MSI-RH and MSI-WH, variants of the MSI coherence protocol that adapts to the read-heavy and write-heavy workloads respectively.
(4) In 3.4, we describe how Kiwi uses the workload analysis to perform translation between different specialized coherence protocols.

3.1 System Model

Kiwi uses a system with L1 cache per processor and a directory controller which interfaces with the main memory. It is developed on top of the directory-based MSI coherence protocol but can

Table 1: Data Access Pattern

Core A	Core B	Optimization
Dense Read	Dense Read	No optimization required if there is a shared state in the coherence protocol
Dense Read	Dense Write	-
Dense Read	Sparse Read	No optimization required if there is a shared state in the coherence protocol
Dense Read	Sparse Write	Use writer-invalidation for reads and write-through for writes
Dense Write	Dense Write	-
Dense Write	Sparse Read	Use self-invalidation for reads and ownership for writes
Dense Write	Sparse Write	Use ownership for dense writes and write-through for sparse writes
Sparse Read	Sparse Read	No optimization required if there is a shared state in the coherence protocol
Sparse Read	Sparse Write	-
Sparse Write	Sparse Write	-

be extended to use other coherence protocols such as MESI or MOESI.

We identify the different data access patterns to propose the coherence design strategy which can be used to optimize a specific workload. Table 1 shows the optimization which can be performed when two cores (CPU or GPU) are accessing a cache line to perform a dense or a sparse read/write operation on it. As highlighted in the table, Kiwi focuses on optimizing the (a) Dense Read, Sparse Write and (b) Dense Write, Sparse Read data access pattern for a CPU core. It uses specialized coherence protocols such as *MSI-RH* and *MSI-WH* as highlighted in section 3.3.2 and 3.3.1 to perform these optimizations.

3.2 Memory Access Pattern Predictor

Kiwi uses a simple memory access pattern history table which records the number of reads and writes per cache line. Each entry in the table also maintains the last access cycle time which is used to evict entries when the table is full and invalidate old entries. A fixed threshold is used to identify the data access pattern and classify a cache line as either a dense or a sparse read/write heavy. After a cache line is identified to satisfy an access pattern, a WH (Write-Heavy) or a RH (Read-Heavy) type message is sent to the directory controller which performs the translation between MSI and specialized coherence protocol. Section 3.4 describes the translation unit which performs the translation between these protocols based on an incoming message from the predictor.

3.3 Specialized Coherence Protocols

3.3.1 MSI-WH

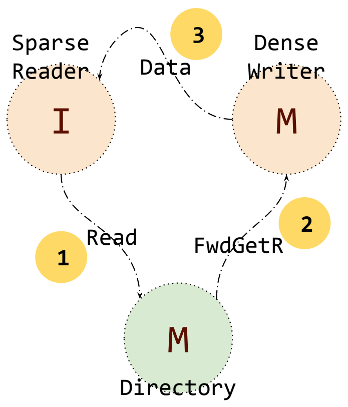

Figure 1: MSI-WH

MSI-WH is a variant of the MSI coherence protocol designed for write-heavy workloads. The MSI protocol transitions a modified state to a shared state during a read access which requires invalidations being sent to the sharers for a subsequent write operation. In a write-heavy workload, since the number of reads is much smaller than the number of reads, the writers need to invalidate all the sharers frequently. GPUs use a stale-data invalidation approach triggered by software hints to address this access pattern but this is not available in a CPU workload. Thus the goal of MSI-WH is to reduce these unnecessary invalidations by forwarding the cache line from the writer to the reader at every read operation. As shown in Figure 1, a read operation does not store the cache line in its local cache and instead forwards the data received from the writer to the CPU directly. This decreases the operation latency of a subsequent write operation on a core which already owns the data.

3.3.2 *MSI-RH*

MSI-RH is a variant of the MSI coherence protocol specifically designed for read-heavy workloads. The MSI protocol sends an invalidation to all the sharers of a cache line when it is transitioning from a shared state to a modified state during a write access. In a read-heavy workload, since the number of writes is much smaller than the number of reads, these sharer invalidations are frequent and unnecessary. The goal of MSI-RH is to reduce these unnecessary invalidations by performing a write-through when a processor is performing a store operation. As shown in Figure 2, during a store operation, MSI-RH ensures the cache line in both memory and sharers caches are updated instead of granting private access to the writer. This decreases the operation latency of a reader who already has the line in its local cache. A write-though operation increases the network traffic and operation latency for a store operation but since the workload is a read-heavy workload, it does not contribute to the overall execution time of the workload.

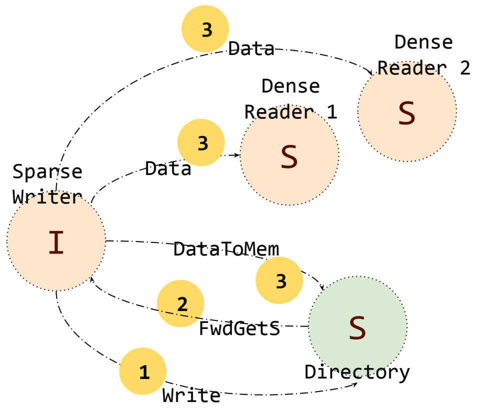

Figure 2: MSI-RH

3.3.3 *Correctness Requirements*

Kiwi aims to ensure a sequential consistency model with synchronization accesses which is commonly used in CPU programs. Since verifying the correctness of a coherence protocol is a challenging task due to its sheer non-determinism, we wrote a few test cases where multiple threads are reading and writing from shared memory locations to verify data correctness.

3.4 Translation Unit

Kiwi uses a translation unit to perform the switch between the MSI and the specialized coherence protocol at a fine-granularity. The directory controller subscribes to a queue which receives the messages sent from the memory access pattern predictor. If the corresponding cache line is present in a stable state, the directory controller performs a transition to the specialized coherence protocol. This transition involves moving from a stable state in MSI protocol to a stable state in specialized protocol. Since there is no shared state and modified state in MSI-WH and MSI-RH respectively, an invalidation is sent to transition the corresponding MSI state to an invalid state. The incoming predictor messages are stalled if the cache line is currently in a transient state. Currently, Kiwi only performs a one-way transition from MSI to a specialized coherence protocol but can be easily extended to support the vice-versa.

4 IMPLEMENTATION

We implemented Kiwi using Gem5 in approximately 4000 lines of code. We leveraged the ruby memory model for the underlying coherence protocols. We developed the specialized coherence protocols on top of the MSI coherence protocol and tested them individually for correctness. The memory access pattern recorder is also integrated with the MSI protocol and the static thresholds are determined manually using predetermined benchmarks. The translation unit is currently developed for MSI-WH to perform the switch from MSI when the write-heavy benchmark is running on the system. We leave MSI-RH integration with the MSI and MSI-WH unit for future work.

5 EVALUATION

In our evaluation of Kiwi, we try to answer the following questions:

- What is the improvement in execution time when we employ the Kiwi system?
- How much does Kiwi reduce the overall network traffic?

In all the experiments, we deal with tunable parameters of the Kiwi system as described in Table 2.

Table 2: Kiwi Evaluation Setup

Parameter	Configuration
`CPU Cores`	8
`Kernel`	x86 Linux
`CPU Frequency`	2 GHz
`L1 Size`	64 KB, 2-way assoc.
`Interconnect`	Crossbar

5.1 Benchmarks

WriteH: In this benchmark, CPU threads densely write and sparsely read shared memory locations. This highlights the benefits of using the MSI-WH coherence protocol over MSI and the advantage of stale-data invalidation over writer-invalidation.

ReadH: In this benchmark, CPU threads densely read and sparsely write shared memory locations. This highlights the benefits of using the MSI-RH coherence protocol over MSI and the advantage of write-through for writes instead of obtaining ownership.

5.2 Execution Time

The main motivation behind creating Kiwi is to demonstrate a measurable improvement in the execution time of a program using a specialized coherence strategy.

Figure 3 shows the execution time for the WriteH benchmark using the MSI, MSI-WH and MSI + MSI-WH (integrated using the fine-grain translation unit) coherence protocols. As shown in the figure, the execution time decreases by 9% and 2% with the specialized coherence protocol and its translation unit respectively.

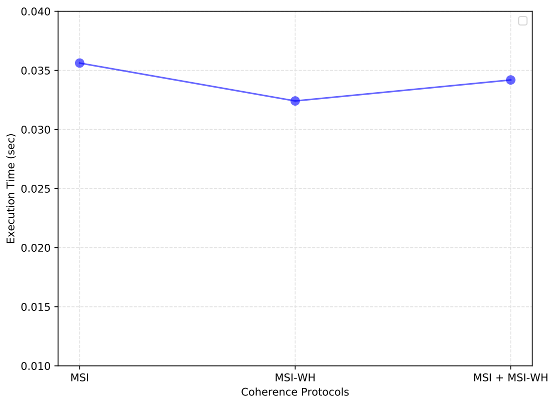

Figure 3: Execution Time (WriteH)

Figure 4 shows the execution time for the ReadH benchmark using the MSI and MSI-RH coherence protocols. From the figure, the execution time for both MSI-RH and MSI protocol is similar which shows that our design achieves the baseline performance in terms of execution time.

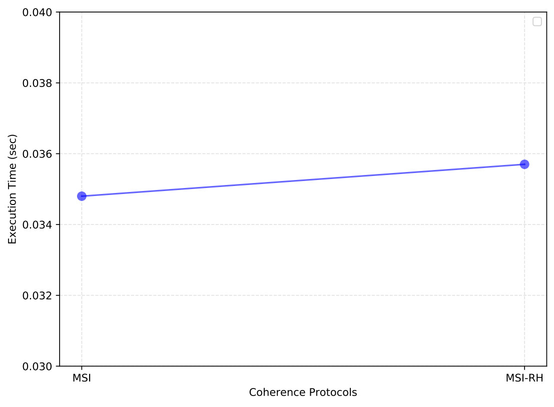

Figure 4: Execution Time (ReadH)

5.3 Network Traffic

The second factor in a coherence protocol is the network traffic in the interconnect due to the exchange of coherence messages during the execution of a multi-threaded program.

Figure 5 shows the network traffic for the WriteH benchmark using the MSI, MSI-WH and MSI + MSI-WH (integrated using the fine-grain translation unit) coherence protocols. As shown in the figure, the network traffic is significantly improved since the number of invalidations sent during ownership is removed.

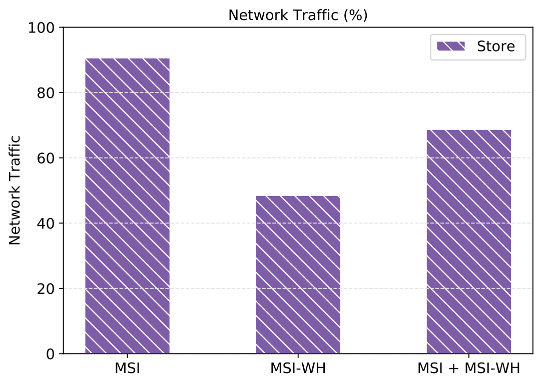

Figure 5: Network Traffic (WriteH)

Figure 6 shows the network traffic for the ReadH benchmark using the MSI and MSI-RH coherence protocols. The memory traffic for MSI-RH is 6 times less than memory traffic for MSI. This achieves the goal for our design which is to reduce the memory traffic by eliminating unnecessary invalidations. The design is evaluated through a benchmark with densely read and sparsely write operations which is not common in average scenario. However, it does show our design gives performance improvements.

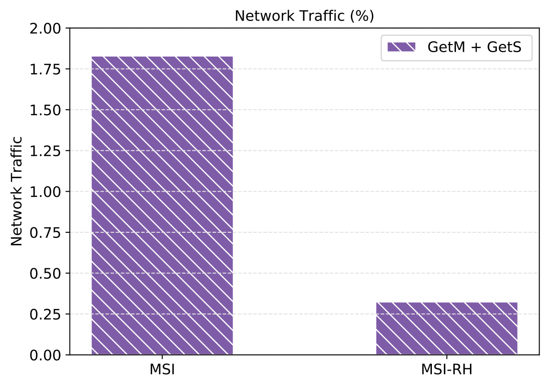

Figure 6: Network Traffic (ReadH)

6 FUTURE WORK

One of the main components of Kiwi that we could improve is the memory access pattern identifier and predictor. Currently, the thresholds for prediction are determined statically through manual analysis of the workload. We can extend Kiwi to dynamically determine this threshold based on the available history and counters.

As mentioned before, the translation unit is also currently setup to just perform a one-way translation from MSI to a specialized coherence protocol. In future, Kiwi can be extended to support two-way translation by making changes to the predictor. This will provide better performance when a cache line in a single program exhibit different access patterns during its entire life-cycle.

Currently, our work is limited to optimizing the CPU cores in a heterogeneous architecture. Kiwi can be extended to design specialized design strategies for GPUs and other accelerators as well.

7 CONCLUSION

In this paper we proposed Kiwi, a proof-of-concept of a coherence specialization framework that adapts to the different memory access patterns in a single program. We developed coherence optimizations for CPU cores based on the strategies used in a GPU memory architecture. Kiwi designs a data access predictor and uses write-through and stale-data invalidation strategies to optimize the read-heavy and write-heavy workload respectively.

ACKNOWLEDGMENTS

We would like to thank Professor Rakesh Kumar and TA Ahmed Abulila for their valuable advice throughout the project.

REFERENCES

[1] Johnathan Alsop, Matthew D Sinclair, and Sarita V Adve. 2018. Spandex: a flexible interface for efficient heterogeneous coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 261–274.

[2] David A. Wood Alvin R. Lebeck. 1995. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory MMultiprocesssors. In *ACM SIGARCH Computer Architecture News*, Vol. 23. ACM, 48–59.

[3] Devyani Ghosh, John B Carter, and Hal Daume III. 2008. Perceptron-based Coherence Predictors. In *2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (ICSA)*. IEEE Press.

[4] Shubhendu S. Mukherjee and Mark D. Hill. 1998. Using Prediction to Accelerate Coherence Protocols. ISCA.

[5] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development* 59, 1 (2015), 7–1.