Mechanisms for Efficient Memory Access in Near-Cache Accelerators

Bas Yoovidhya

August 2024

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Nathan Beckmann, Chair
Phillip Gibbons

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science.

Keywords: Stuff, More Stuff

For my dog

Abstract

In traditional computer systems, data has to move through the memory hierarchy for computation to take place within the core.This data movement cost has been dominating computer systems' performance, and will only get worse over time. Many proposals address this problem by introducing architectures that move compute closer to data.

Like some of these proposals, our approach to this places engines within the cache hierarchy, allowing the core to offload work to the caches. When the engine experiences a cache miss, the requested data could be residing at two different levels within the memory hierarchy. Sending a request to only one of the two locations could result in a miss, increasing the miss latency of the engine. However, sending a request to both locations at once also leads to higher energy consumption.

In this thesis, we introduce a novel Memory Access Predictor to the system that assists the engine in sending requests that minimizes energy usage, while retaining high performance. We evaluate the predictor on various micro-benchmarks, showing that it is able to improve the performance by 21%, and reduces additional energy consumption by 83% in other applications.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Background and Motivation

To combat the increasing data movement costs, there have been many proposals for architectures that move compute closer to memory. In this chapter, we discuss one popular approach called "Near-Cache Computing". We will briefly look through some prior near-cache computing proposals, understanding why data movement costs can be minimized through this design. We then introduce our near-cache computing system, Kobold, and outline some optimizations made to our system. Finally, we motivate a mechanism that can significantly improve performance while keeping energy consumption low.

2.1 Near-Cache Computing

In near-cache computing architectures, accelerators are integrated within the cache hierarchy, allowing the core to offload tasks to cache. There have been many recent proposals that follow this design. Near-Stream Computing proposes a paradigm that utilizes a compiler, CPU ISA extension, and a novel microarchitecture to identify opportunities and offload computation to shared caches [8]. Minnow augments a chip multiprocessor core with a programmable engine that supports worklist scheduling and performs worklist-directed prefetching [10]. Caches in täkō can trigger software callbacks that run in near-cache engines in response to miss, evictions, and writebacks [5].

A representative system, Livia, shows how these engines can be utilized to minimize data movement. In a baseline system (Fig. 2.1) executing a tree lookup application, data is always being moved into the core because code only executes on the core. With Livia, engines are placed throughout the memory hierarchy, allowing for tasks to be scheduled and executed at locations that minimizes data movement. This can be seen in Fig. 2.2, where minimal data movement is needed to execute the same tree lookup application.

2.2 Coherence in Near-Cache Accelerators

When the core and the engine work in parallel, data inconsistencies can happen. For example, if the core's L2 and the engine's eL1D both hold a copy to the same data, it is possible for the core to modify its copy of the data, making the engine's copy outdated. This is a problem because

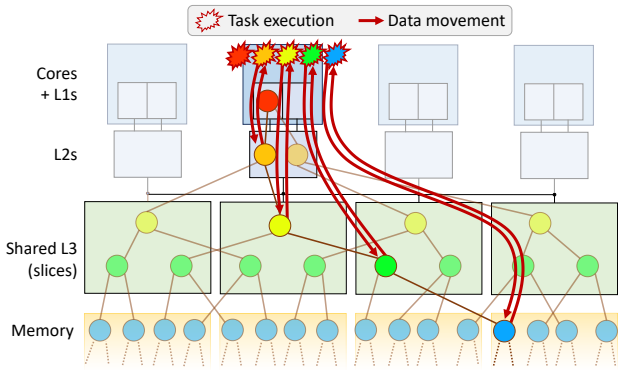

Figure 2.1: Baseline compute-centric systems end up moving data over long distances.

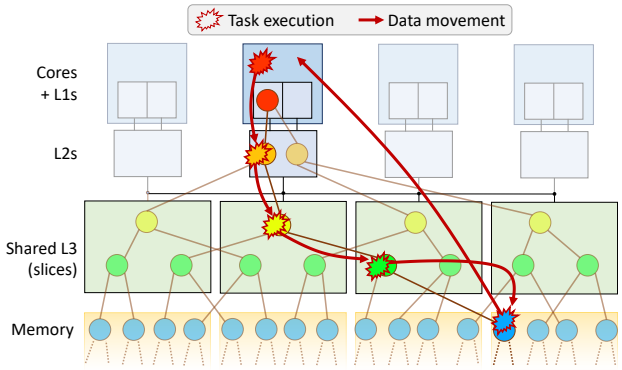

Figure 2.2: Livia executes tree lookup at locations that minimize data movement.

working with outdated data can lead to correctness issues. Maintaining cache coherence is about ensuring changes in the value of the data is correctly propagated through the system. In our case, we want to maintain coherence between the engine and the core. This is a difficult problem; we do not want to introduce much additional complexity to maintain coherence as it may negatively impact the performance of the system. Prior proposals for near-cache accelerators [2, 5] made some assumptions about coherence to help ease implementation efforts. In this section, we will discuss possible protocols that could be used, and explain their shortcomings in relation to our proposed protocol and implementation, Kobold.

2.2.1 Naive Coherence

In directory-based protocols, a directory structure is used to track which cache holds a block, and which state it is in [3]. A coherence request is handled by the directory, which determines what action to take based on the location and state of the block. To increase scalability of directory-based protocols, many systems use a hierarchical coherence protocol, which desginate intermediate levels of the hierarchy to serve as directories for the lower levels. Naively extending these hierarchical protocols by treating the eL1D as a sharer under the LLC can be detrimental to the system's performance. If the eL1D is treated as a sharer under the LLC, for a core to send coherence requests to the eL1D, the request has to be propagated down to the LLC before making its

way to the eL1D and vice versa. This is wasteful as the core and the engine reside on the same tile, and the LLC directory could be much further away. Additionally, directory overhead is also higher as the number of sharers has increased.

2.2.2 Kobold

Some of the aforementioned problems can be avoided by making it so the core's L2 and the eL1D appear like a single, unified cache to the LLC. In this situation, the eL1D would logically be a child of the L2, and coherence is maintained between themselves. This approach would eliminate much on-chip network traffic and additional directory states that would otherwise be required to maintain coherence between core and the engine. To achieve this, Kobold introduces a directory-like structure called the "Mis-direction Filter" (MDF) that tracks the state of the eL1D. As seen in Fig. 2.3, the MDF augments the L2. The MDF allows the core and the engine to safely share data and transfer ownership without involving the LLC. This greatly simplifies coherence between the engine and the core, only adding negligible additional state, while not requiring any changes to the baseline directory coherence protocol at the LLC.

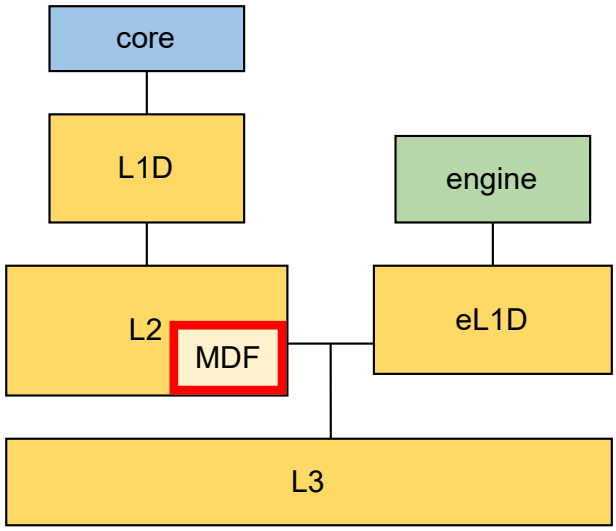

Figure 2.3: Cache hierarchy design with the mis-direction filter (MDF) augmenting the L2 cache.

However, this design can still harm the overall performance for applications where the eL1D is better suited to be the child of the LLC, Ideally, the eL1D would have direct access to both the L2 and the LLC. This is difficult to implement in practice because in certain protocols, the LLC assumes that data will not be requested if the tile already holds the data. This means that unexpected issues can arise if we allow the eL1D to send a request to the LLC when a valid copy already exists in the L2. For this reason, we made it so the L2 is always on the critical path of an eL1D miss, even if the requested data resides within the LLC.

In order to hide the L2 latency from the critical path, an additional optimization is made to allow the eL1D to speculatively forward a request to the LLC when a miss occurs. For correctness purposes, modifications were made to the protocol to handle additional scenarios that come

with the optimization. Fig. 2.5 shows a scenario where the LLC receives the eL1D's request for data that is already shared by the requesting tile. In this scenario, the LLC protocol is modified to ignore such requests, and the L2 will also have to wait for an acknowledgement from the LLC before being able to service the request. This means that in those scenarios, the critical path of an eL1D miss will be slightly longer than if the eL1D did not speculate. Nevertheless, many applications still benefit from this optimization, and Fig. 2.4 demonstrates how successful speculative requests are at hiding the L2 latency from the critical path in an application that relies on accesses to the LLC. In this application, always speculating is able to achieve a 24% speedup over Kobold without the speculation optimization in place.

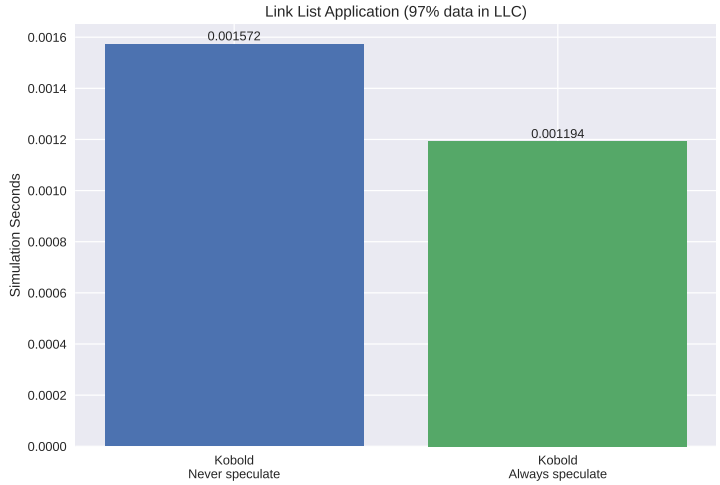

Figure 2.4: An application that mostly accesses data residing in the LLC benefits a lot from speculative accesses.

2.3 Memory Access Predictor

Although speculation is able to hide the L2 latency from the critical path of eL1D misses well, it comes at cost of having to send additional messages to the LLC that always result in tag lookups. If we were to always speculatively send eL1D requests to the LLC, there could be a lot of wasted energy for requests that only needed to be sent to the L2. The increased energy consumption is especially apparent for applications that mostly rely on accesses to the L2. We can see this in Fig. 2.6, where an application with most of its data residing in the L2 ends up consuming 1.19× more energy by speculatively sending requests every eL1D miss. On top of that, since this application mostly relies on accesses to the L2, the increased eL1D miss critical path (due to having to wait for the LLC's response) is especially apparent here as well. Fig. 2.7 shows us that for an application such as this, it can result in around a 6% slowdown. In this case, never speculating is better than always speculating as it saves more energy and improves the performance.

In an ideal scenario, we want the eL1D to send speculative requests only when the requested data is in the LLC, and never send speculative requests otherwise. We can get close to this

Figure 2.5: When the eL1D requests the LLC for data that is already in the L2, this request is ignored by the LLC.

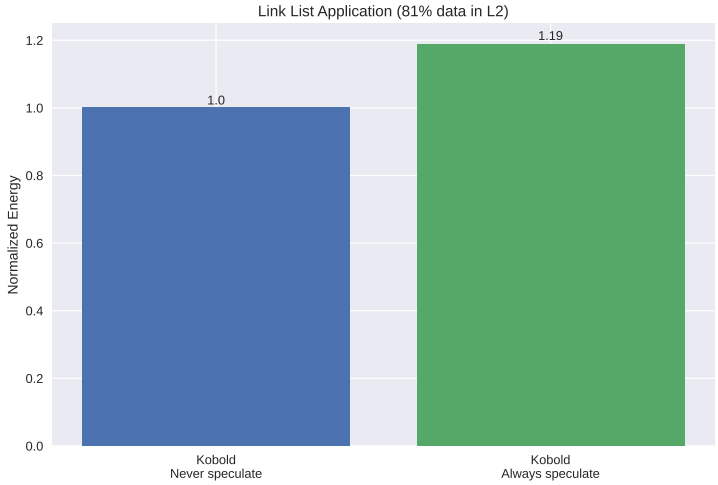

Figure 2.6: In a simple linked list traversal application, a lot of energy is wasted to speculating when most of the data is in the L2.

ideal scenario by using a *Memory Access Predictor* (MAP). We drew inspiration from various places [4, 6], which uilizes predictors for similar scenarios in their system (also cite branch predictors). For Kobold, this predictor would decide whether or not to speculatively send a request to the LLC. If the predictor has a high enough accuracy, it would be able to achieve good performance while keeping additional energy consumption low.

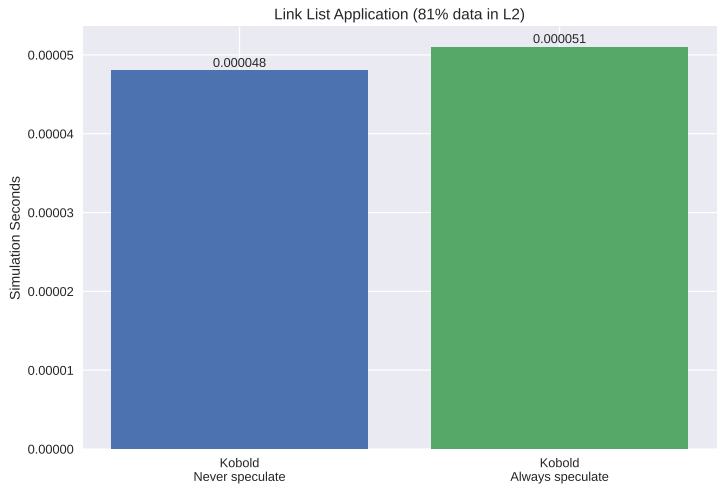

Figure 2.7: In a linked list traversal application where most of the data is in the L2, speculation actually harms the performance.

Chapter 3

Design and Implementation

The Memory Access Predictor is key to enabling good performance while retaining low additional energy consumption. In this chapter, we discuss various designs for the Memory Access Predictor, delving into the reasoning behind each implementation, and situations where one may excel over the others.

3.1 Memory Access Counter (MAP-C)

Our first design, inspired by Alloy Cache's predictor [4], utilizes a single saturating counter in making its prediction. The counter is updated everytime a response is received from either the L2 or the LLC, informing the predictor where the requested data came from. When the response is from the L2, the counter is decremented, and when the response is from the LLC, the counter is incremented. To make a prediction, the predictor uses the most-significant-bit (MSB) of the counter as shown in Fig. 3.1. If the MSB=1, the predictor assumes that the data is in the LLC, telling the eL1D to send a speculative request to the LLC. Otherwise, the predictor assumes the data is in the L2, and the eL1D will not send a speculative request.

This design relies on the descriptiveness of the counter. When the counter value is low, this informs the predictor that the majority of the recent responses have been from the L2, and that it is likely for future accesses to be for data in the L2 as well. Conversely, when the counter value is high, the predictor will know that the recent responses are from the LLC, so it predicts that future accesses will also be to the LLC. In our implementation, the saturating counter starts at zero, meaning that our approach is more on the conservative side— only predict speculation when the predictor has seen enough evidence in order to do so.

With the right counter size, we believe that this counter should be responsive enough to changes in memory access patterns. One downside to this approach is that it is a saturating counter, which means that the counter value is only representative of recent memory accesses. This can be adjusted to a certain extent by using a larger-sized saturating counter, but that could affect the responsiveness of the predictor to changes in memory access patterns. There is a fine line to walk between responsiveness and descriptiveness, which will be discussed in greather depth in chapter 5.

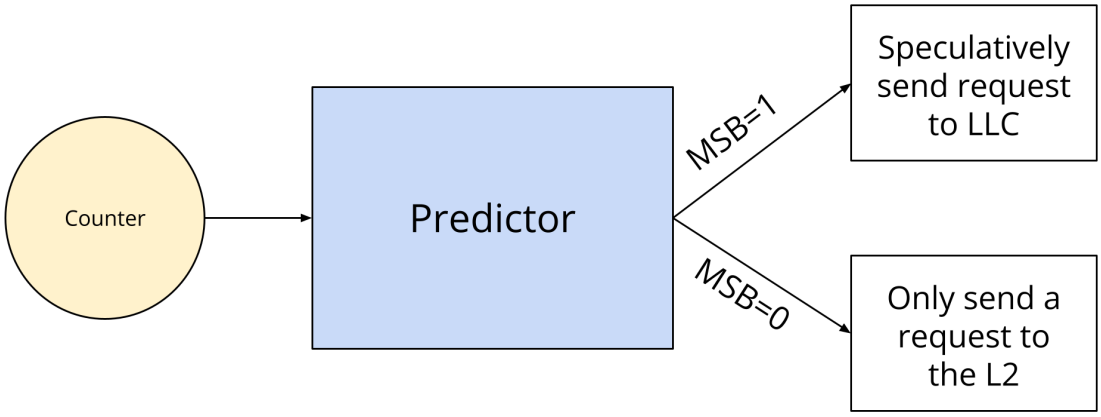

Figure 3.1: The Memory Access Counter predicts speculation when the MSB=1 and no speculation when the MSB=0.

3.2 Memory Access Counter Table (MAP-T)

Our second approach to the predictor is to maintain a table that maps cache line addresses to a counter. The design for the Memory Access Counter Table is almost identical to the Memory Access Counter. The big difference in this design is instead of using one saturating counter, we now maintain one for every entry within table. When there is a response, the predictor first checks the cache line address associated to the response. The cache line address is then used to hash into the table in order to retrieve the corresponding memory access counter. Much like before, if the response is from the L2, the counter is decremented, and if the response is from the LLC, the counter is incremented. When making predictions, the cache line address is hashed and used to index into the table to retrieve the right counter to use for predictions. The prediction mechanism then exactly follows the Memory Access Counter's mechanism as described in the previous section, using the MSB to determine whether or not to send a speculative request to the LLC.

This design aims to solve the issue of only having one counter for the entire workload in the first approach. In a long-running application, the previous approach would not be able to distinguish between hot and cold cache lines, incorrectly incrementing and decrementing for certain memory accesses. With this table approach, each cache line has its own counter and would not run into the same problem. However, it is entirely possible for hash collisions to happen either due to the hashing function used or due to the size of the table. We discuss these parameters further in chapter 5, and how adjusting them can affect the prediction accuracy.

3.3 Memory Access Ratio (MAP-R)

Our final approach to the predictor uses a ratio between the number of LLC accesses and the L2 accesses. The predictor keeps track of the number of responses from the LLC and the number of responses from the L2. When making a prediction, we compute the ratio of these two values, and compare it to a threshold value. If the ratio is greater than the threshold value, then the predictor

tells the eL1D to speculatively send a request to both the LLC and the L2. Otherwise, the predictor tells the eL1D to not speculate, and only send a request to the L2. To avoid overflows, we shift the counter values to the right every certain number of predictions (or cycles!).

This approach effectively computes an Exponentially Weighted Moving Average (EWMA) that informs the predictor how likely it is for the requested data to be in the LLC, based on past memory accesses. It addresses a few of the weaknesses the Memory Access Counter design had. Firstly, it is able to remember past memory accesses unlike the Memory Access Counter, which utilizes a saturating counter. It also weighs recent memory accesses more, retaining one of the Memory Access Counter's key strengths. Second, this approach utilizes a ratio being above a chosen threshold value before speculating. These values are more descriptive than using a single counter. A higher threshold value means that the predictor will operate on a more conservative side, only telling the eL1D to send a speculative request to the LLC when a high enough ratio of LLC responses to L2 responses have been reached. This means that the predictor can opt for a more performance-oriented prediction policy or a more energy-conservative prediction policy based on the threshold value alone. In our design, when the predictor has no information as it is starting out, we opted for a more conservative approach, predicting no speculative requests. In practice, this predictor type may be difficult to implement as it heavily utilizes division, which would not be supported by the predictor.

3.4 Roads Not Taken

We intended to implement another version of MAP-T that instead maintains a mapping of the instruction address to a counter instead of the cache line address. This may have been a good idea, as it is known that there is a strong correlation between cache access information and the instruction address that triggers the cache access [1, 4, 7, 9]. However, we found it to be difficult to obtain the instruction address within the predictor, so we did not pursue this implementation further.

Chapter 4

Methodology

Chapter 5

Evaluation

Chapter 6

Conclusion

6.1 Future Work

Bibliography

[1] Samira Khan, Daniel Jiménez, Doug Burger, and Babak Falsafi. Using dead blocks as a virtual victim cache. pages 489–500, 11 2010. doi: 10.1145/1854273.1854333. 3.4

[2] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan 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. 2.2

[3] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David Allen Wood. *A Primer on memory consistency and cache coherence*. 2020. 2.2.1

[4] Moinuddin K Qureshi and Gabriel H Loh. Fundamental Latency Trade-offs in Architecting DRAM Caches. In *Proc. of the 45th annual IEEE/ACM intl. symp. on Microarchitecture*, 2012. 2.3, 3.1, 3.4

[5] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan 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. 2.1, 2.2

[6] Víctor Soria-Pardos, Adrià Armejach, Tiago Mück, Dario Suárez-Gracia, José Joao, Alejandro Rico, and Miquel Moretó. Dynamo: Improving parallelism through dynamic placement of atomic memory operations. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, 2023. 2.3

[7] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, 1995. 3.4

[8] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. Near-stream computing: General and transparent near-cache acceleration. 2022. 2.1

[9] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proc. of the 44th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, 2011. 3.4

[10] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (Proc. ASPLOS-XXIII)*, 2018. 2.1