Mechanisms for Efficient Cache Access in Near-Cache Accelerators

Piratach Yoovidhya

CMU-CS-24-143

August 2024

Computer Science Department
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: Near-cache computing, Predictor, Data-centric, Cache-coherence

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 15%, and reduces additional energy consumption by 91% in other applications.

Acknowledgments

I would like to start off by thanking my advisor, Professor Nathan Beckmann, for his guidance and support throughout my research career. I am glad to have had the opportunity to work with him over the past few years as it has been an incredible learning experience for me every step of the way.

I am also grateful to the corgis, Baphy and Arya, both of whom play a crucial role in research meetings.

Thank you to Brian Schwedock who mentored me during my undergraduate years and was the person that got me started in my studies in computer architecture. Brian, along with Yatin Manerkar, Nicolai Oswald, and Jennifer Brana also collaborated with me on this research and I would like to thank them for all their help.

Jennifer Brana deserves an additional thank you for mentoring me throughout this project and bearing with me during my struggles (which, admittedly, happened all too often).

Finally, I want to thank Professor Phillip Gibbons for helping me as part of the committee and giving me valuable feedback to my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, data movement has emerged as the primary bottleneck of modern computer systems [10, 13, 14, 17]. Computation is much cheaper than data movement, and the gap has been widening due to the shift towards leaner and specialized core [7, 11, 13, 14, 19]. The fundamental issue with conventional computing systems is that data is only processed on cores, leading to excessive data movement through deep, multi-level cache hierarchies.

To address this trend, near-data processing designs have been proposed, adding accelerators near the main memory [4, 5, 12, 23]. This design aims to bring compute closer to data, potentially reducing the excessive data movement found in conventional hierarchies. Near-data processing works well when programs have little reuse, but in applications with significant locality, this design is often far less effective [2, 15, 28, 36]. Furthermore, the distance between the core and the accelerators is still far, making communication with the core extremely expensive.

Near-cache computing is a similar approach that offers a promising solution to the problems faced in near-data processing designs. In near-cache computing designs, the baseline multicore is augmented with a general-purpose engine (i.e., accelerator) on each tile. Near-cache engines benefit from the lower communication costs to the core, allowing for the core to offload tasks to the engines without significant overhead.

Challenges There are several challenges that arise with near-cache computing designs. For the engine to utilize the shared memory space effectively, coherence has to be maintained between the engine and the core. Naively extending baseline protocols can increase verification costs, network traffic, directory state, and can also lead to the engine polluting the core's caches [6, 25] Beyond that, existing protocols do not allow the engine to access both the L2 and the LLC efficiently, which is necessary for near-cache engines to be effective.

Kobold Kobold is a coherence protocol and implementation that addresses these problems [6]. A directory structure called the "Mis-direction Filter" is added to help the core maintain coherence with the engine without incurring significantly increasing verification costs, network traffics, and directory states. Kobold allows for the engine to efficiently access both levels of the cache hierarchy. Different applications prefer access to different levels of the cache hierarchy, and sending a request to the wrong level can increase the miss latency of the engine. On the

other hand, speculatively sending requests to both levels of the cache hierarchy in parallel can lead to higher energy consumption, and in some cases, can negatively affect the performance of the system.

Thesis Contributions We propose a Memory Access Predictor that predicts which level of the cache hierachy the requested data resides in. The predictor is used by the engine to make efficient cache accesses at the right level of the hierarchy. We present three implementations of the predictor, weighing the advantages and disadvantages of each design. Across a range of micro-benchmarks, we found that on average, our Memory Access Predictor is able to improve performance by 4.7% compared to the baseline approach, and saves 56.7% of the energy that would have been wasted on excessive speculative requests.

Outline The remainder of this thesis is organized as follows. Chapter 2 discusses relevant background information on data movement, cache coherence in near-cache computing designs, and Kobold. Chapter 3 describes three different Memory Access Predictor designs. Chapter 4 briefly summarizes the methodology used for the evaluation. Chapter 5 presents the results from our evaluation, showing how well the Memory Access Predictor performs compared to alternative approaches. Chapter 6 concludes, and discusses possible future work.

Chapter 2

Background and Motivation

To combat increasing data movement costs [10, 13, 14, 17], there have been many proposals for architectures that move compute closer to memory, rather than moving data to compute [8, 9, 18, 20, 30, 31]. In this chapter, we discuss one popular approach called "Near-Cache Computing". We will discuss some prior near-cache computing proposals, and how why data movement costs can be minimized through this design. We then introduce a near-cache computing system, Kobold, and outline some optimizations made to the system [6]. Finally, we motivate a mechanism built on top of Kobold that can 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 [1, 2, 19, 22, 25, 27, 32, 33, 35]. 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 [33]. Minnow augments a chip multiprocessor core with a programmable engine that supports worklist scheduling and performs worklist-directed prefetching [35]. täkō performs computations on data as it travels through the cache hierarchy, for example, decoding encrypted data [25].

A representative system, Livia, shows how these engines can be utilized to minimize data movement [19]. In a baseline system (Fig. 2.1) executing a tree lookup application, computations only execute on the core so data is always being moved into the core, often forcing it to traverse the full memory hierarchy. With Livia, engines are placed throughout the memory hierarchy, allowing tasks to be scheduled and executed at locations where the data naturally settles to, minimizing data movement. This can be seen in Fig. 2.2, where minimal data movement is needed to execute the same tree lookup application.

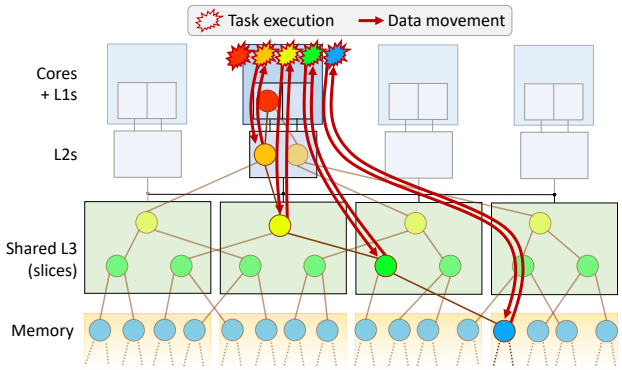


Figure 2.1: Baseline compute-centric systems end up moving data over long distances.

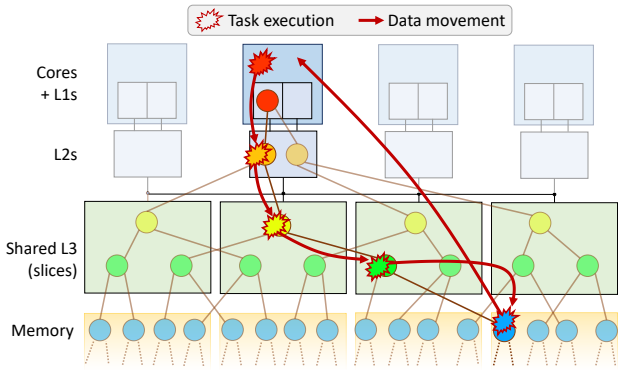


Figure 2.2: Livia executes tree lookup at locations that minimize data movement.

2.2 Coherence in Near-Cache Accelerators

For the engine to perform meaningful work, coherence has to be maintained between the core and the engine. Maintaining coherence between the core and the engine is a difficult problem; it is best to avoid much additional complexity to maintain coherence as it may negatively impact the performance of the system. With the right coherence protocol, the engine should be able to efficiently access both the L2 and the LLC without additional costs in network traffic, verfication, and maintaining directory states. Prior proposals for near-cache accelerators [19, 25] 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 a 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 [21]. A coherence request is handled by the directory, which determines what action to take based on the location and state of the block. Naively extending these directory-based protocols by treating the eL1D as a sharer under the LLC can be detrimental to the system's performance. The directory overhead is doubled, and any transfer between the core and the

engine requires data to be written back to the LLC, resulting in excessive communication with the LLC. This problem is further exacerbated when considering chip multi-processors, where the LLC banks are often far from the tile.

2.2.2 Kobold

To address the problems, the Kobold coherence protocol has been proposed [6]. Kobold is a hierarchical cache-coherence protocol that is able to improve scalability and limit coherence traffic by utilizing a directory-like structure called the "Mis-direction Filter" (MDF). Kobold augments the L2 cache with the MDF (Fig. 2.3), which is used to track the state of the eL1D, allowing for coherence requests to be handled without involving the LLC. This restricts the complexity of the engine within a tile by making the core's L2 and the eL1D appear like a single, unified cache to the LLC. The LLC is then able to use baseline protocols without any modifications, which keeps verification costs low.

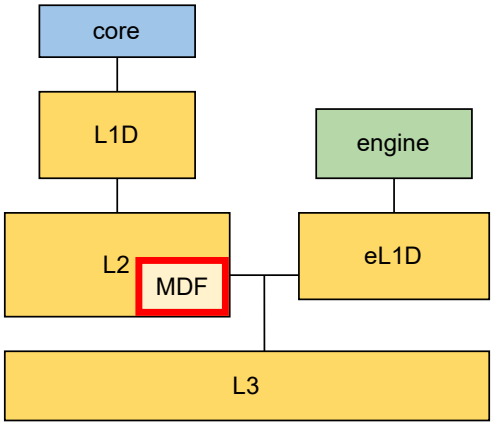


Figure 2.3: Cache hierarchy design with the mis-direction filter (MDF) augmenting the L2 cache.

When the L2 is inclusive of the eL1D, data is duplicated in both caches. It is possible for data brought into the L2 by the engine to remain in the L2 long after it has been evicted in the eL1D. As a result, engine is able to pollute the core's L2 with unnecessary data, worsening the baseline processor performance. This was shown in täkō [25], where a 4× slowdown was observed in certain benchmarks as a result of L2 pollution. Kobold makes the L2 non-inclusive of the eL1D, eliminating this problem entirely.

Despite these optimizations, this hierarchical design can still harm the overall performance for applications where the eL1D is better suited to be the child of the LLC; the current approach adds the L2 miss latency to the eL1D's miss path. In order to hide the L2 latency during an eL1D miss, the eL1D can speculatively forward a request to the LLC in parallel with the L2 request when a miss occurs. For correctness purposes, modifications were made to the protocol to handle additional scenarios that come with the optimization. Fig. 2.4 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 for correctness, 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 send a speculative request.

Many applications benefit from this optimization, and Fig. 2.5 illustrates 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 6% speedup over Kobold without the speculation optimization in place.

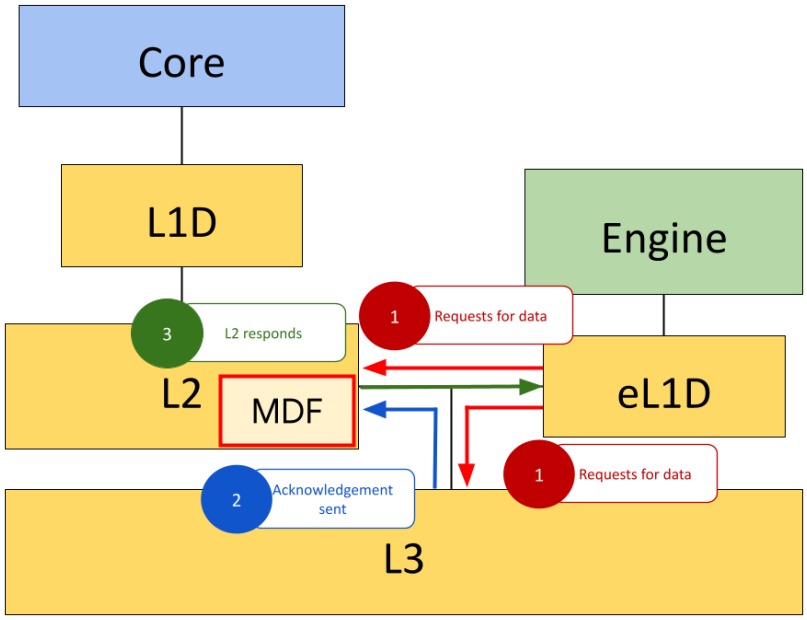


Figure 2.4: When the eL1D speculatively requests the LLC for data that is already in the L2, this request is ignored by the LLC. Instead, the L2 waits for an acknowledgment from the LLC before being able to respond to the eL1D with data.

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. For an application that mostly relies on accesses to the L2, these speculative requests to the LLC are simply wasted energy that could have easily been avoided. We can see this in Fig. 2.6a, 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. Furthermore, since this application mostly relies on accesses to the L2, the longer L2 hit path (due to having to wait for the LLC's acknowledgment) ends up leading to a 6% slowdown for the same application (Fig. 2.6b). For applications that rely on L2 accesses, never speculating is better than always speculating as it saves energy and results in higher performance.

The ideal scenario is for the eL1D to only send parallel requests when the requested data is in the LLC, and never send parallel requests otherwise. We can get close to this ideal scenario by using a *Memory Access Predictor* (MAP). We drew inspiration from various places [24, 26], which utilizes predictors for similar scenarios in their system. In Alloy Cache [24], the Memory

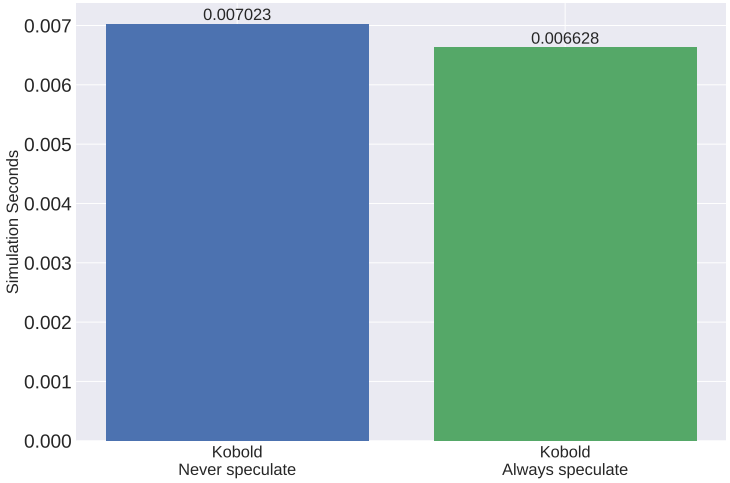
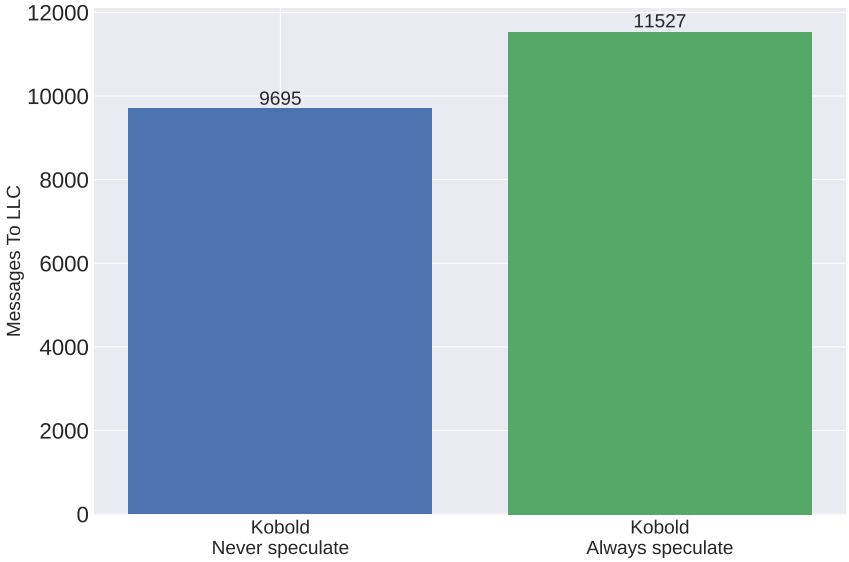
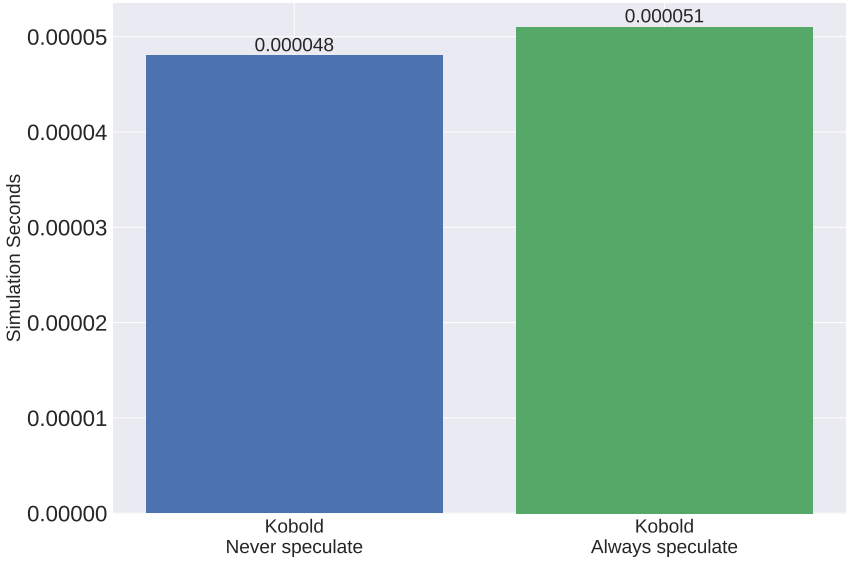


Figure 2.5: An application that mostly accesses data residing in the LLC sees a speedup from having speculative requests enabled.

Access Predictor is used to dynamically choose between accessing cache and memory serially, or accessing both in parallel. Similarly, in Kobold, the Memory Access Predictor will be used to dynamically choose between sending a request to the L2 and the L3 serially, or sending a request to both in parallel. If the predictor has a high enough accuracy, the engine will send requests to the correct cache levels, achieving good performance while keeping additional energy consumption low.



(a) A significant amount of messages is unnecessarily sent to the LLC with always speculating.



(b) The longer L2 hit path results in an overall slowdown for an application that mostly relies on accesses to the L2.

Figure 2.6: Speculation can adversely affect both the energy consumption and performance in an application that mostly accesses the L2.

Chapter 3

Design and Implementation

The Memory Access Predictor is key to enabling good performance without consuming much additional energy from excessive speculative requests. In this chapter, we discuss various designs for the Memory Access Predictor.

3.1 Memory Access Counter (MAP-C)

Our first design, inspired by Alloy Cache's predictor [24], 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. When the counter value is high, the

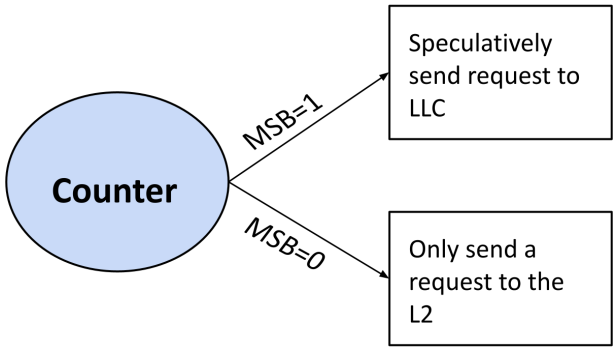


Figure 3.1: The Memory Access Counter predicts speculation when the MSB=1 and no speculation when the MSB=0.

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. Once the counter reaches the minimum or maximum value, the saturating nature of this counter ensures that the value will not change any further. This is by design to ensure that the counter remains responsive to changes in cache access patterns in the future.

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 maintaining a mapping of the cache line address to its respective saturating counter, instead of using a single counter for every cache access. 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 (shown in Fig. 3.2). 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. Currently, the hash function is a simple function that mods the cache line address by the size of the table. With more sophisticated hashing functions, collisions may not happen as frequently.

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 differentiate between hot and cold cache lines, incorrectly incrementing and decrementing for certain cache accesses. With this table approach, each cache line has its own counter and is less likely to experience the same problem.

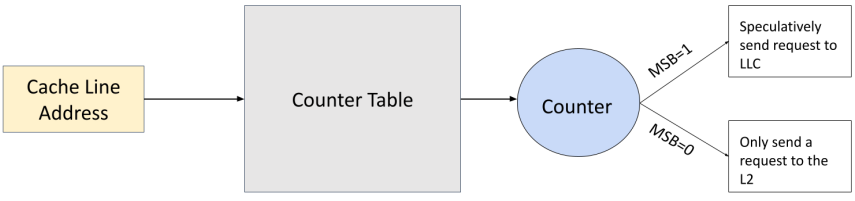


Figure 3.2: The Memory Access Counter Table hashes the cache line address to its respective counter, which is then used for predictions.

3.3 Memory Access Ratio (MAP-R)

Our final approach is inspired by DynAMO's predictor [26], and uses a ratio between the number of LLC accesses and the L2 accesses for predictions. 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. This is illustrated in Fig. 3.3. To avoid overflows, we shift the counter values to the right every certain number of predictions.

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. There are a few advantages with this over previous approaches. First, past memory accesses are used as part of the prediction scheme, unlike the Memory Access Counter, which ignores past memory accesses past a certain point. It still weighs recent memory accesses more, retaining one of the Memory Access Counter's key strengths. Second, utilizing a threshold ratio value for prediction is more descriptive than using the most-significant-bit of a 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. In practice, this predictor type may be difficult to implement as it relies heavily on division. It is also likely to be less responsive than the Memory Access Counter.

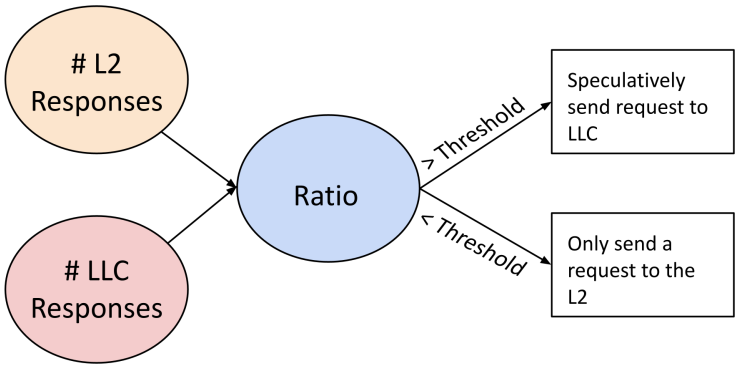


Figure 3.3: The Memory Access Ratio computes a ratio between the number of responses from the LLC and the number of responses from the L2. This is then used for prediction.

3.4 Road 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 [16, 24, 29, 34]. However, we found it

to be difficult to obtain the instruction address within the predictor, so we did not pursue this implementation further. We leave this as future work.

Chapter 4

Methodology

Cores	1 core, RISC-V ISA, 1 GHz, In-order CPU
Engines	1 engine, RISC-V ISA, 1 GHz, Modified In-order CPU
L1	16 KB, 8-way set-assoc, unified data and instruction caches, Tree-PLRU repl.
L2	128 KB, 8-way set-assoc, 4-cycle tag, 4-cycle data array, Tree-PLRU repl.
LLC	512 KB, 16-way set-assoc, 20-cycle tag, 20-cycle data array, inclusive, Tree-PLRU repl.

Table 4.1: System parameters in our experimental evaluation.

Simulation Framework We implemented our system in gem5 [3], an execution-driven, cycle-accurate simulator.

System Parameters Our system parameters are given in Table 4.1. We evaluated a system with one core and one engine, both of which utilizes a simple in-order CPU.

Micro-benchmarks We modified a number of micro-benchmarks used in evaluating Near-Stream Computing [33] to evaluate against our system.

Metrics Our evaluation focuses on the effectiveness of the three Memory Access Predictor designs. First, we show the overall performance results of Kobold with the Memory Access Predictor by comparing the simulation seconds, eL1D miss latency, and energy usage to baseline models without the predictor. Following that, we conduct a sensitivity study of each type of predictor under different parameters. We then compare each predictor's performance across different micro-benchmarks to identify the most consistently accurate predictor.

Chapter 5

Evaluation

In this chapter, we evaluate the Memory Access Predictor's performance against a variety of micro-benchmarks similar to the ones used to evaluate Near-Stream Computing [33]. In Sec. 5.1, we evaluate the overall performance benefits from the Memory Access Predictor. The performance of the predictor is compared against never speculating in Kobold, and always speculating in Kobold. The predictor design used for this section is the Memory Access Counter implementation with 3 bits. The reasoning behind using this predictor design is then shown in the sensitivity study in Sec. 5.2, where we compare different parameter values for each Memory Access Predictor design before comparing the overall performance of each design.

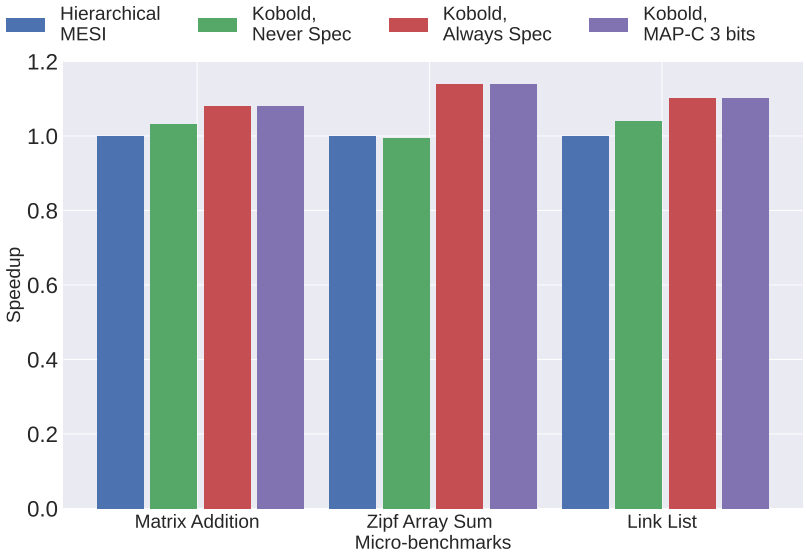
5.1 Memory Access Predictor Performance

This section aims to evaluate the Memory Access Predictor's ability to achieve similar speedups to always speculating for applications that rely on accesses to the LLC. For applications that rely on accesses to the L2, we also evaluate the Memory Access Predictor's ability to retain similar performance as never speculating. We also evaluate how well the Memory Access Predictor is able to save energy that would have been wasted in sending unnecessary speculative requests to the LLC when always speculating. Each of the micro-benchmarks are designed to rely on ether L2 accesses, LLC accesses, or an even split of both.

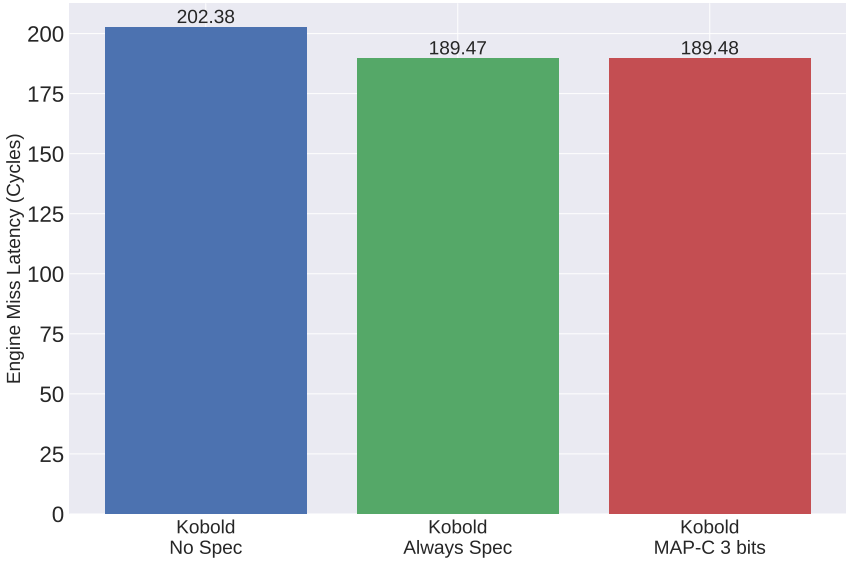
5.1.1 LLC Workloads

To understand how well the Memory Access Predictor is able to achieve the speedup gained from always speculating, we evaluate the predictor against workloads that rely on accesses to the LLC. The result from this is shown in Fig. 5.1a. The conclusion we can draw from this is that the Memory Access Predictor is able to consistently reach the same performance speedup as always speculating for simple micro-benchmarks that rely mostly on accesses to the LLC.

The speedup from always speculating and from the Memory Access Predictor is due to the lower eL1D miss latency compared to never speculating, as seen in Fig. 5.1b. This shows that the Memory Access Predictor is successful in enabling speculating for situations where only sending a request to the L2 would result in adding the L2 miss to the eL1D miss path.



(a) Performance with the predictor is similar to always speculating for applications that access the LLC.

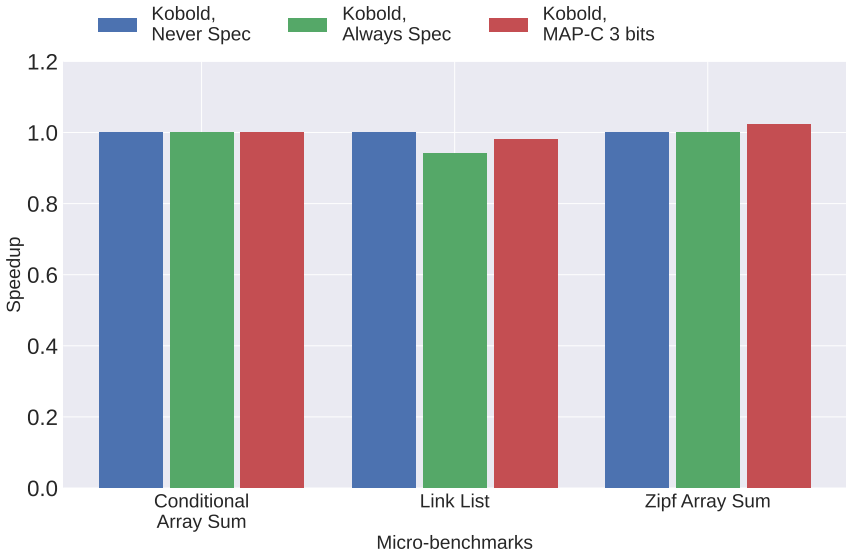


(b) MAP is able to successfully hide the L2 miss latency in a link list application with most of its data in the LLC.

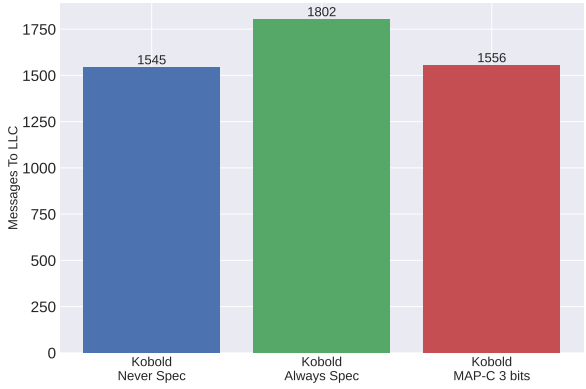
Figure 5.1: Evaluation of MAP against micro-benchmarks that rely on accesses to the LLC.

5.1.2 L2 Workloads

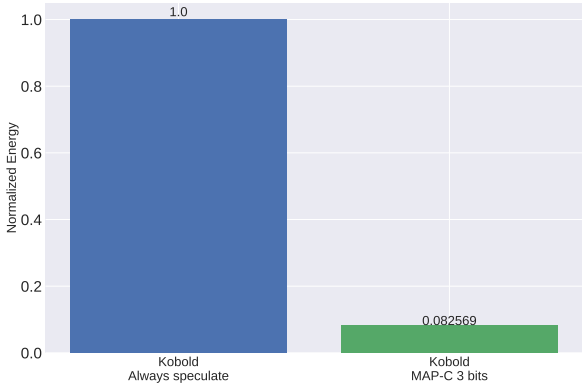
We now evaluate how well the Memory Access Predictor is able to retain similar performance to never speculating when running applications that rely mostly on accesses to the L2. Fig. 5.2a illustrates that the Memory Access Predictor retains similar performance to never speculating across the micro-benchmarks that mostly access the L2. For the link list application, the Memory Access Predictor performs 2% worse compared to never speculating, due to incorrectly sending



(a) MAP is able to retain similar performance to never speculating.



(b) MAP sends similar total number of messages to the LLC as never speculating.



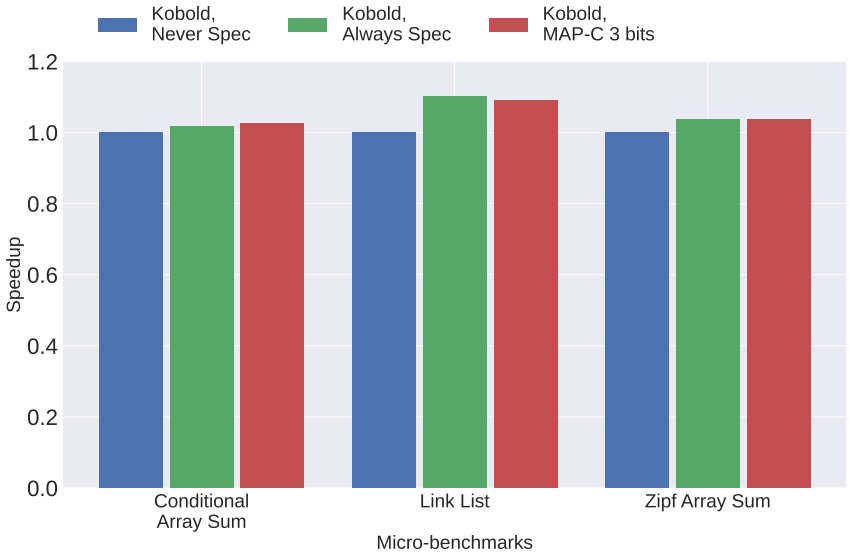
(c) MAP consumes significantly lower amount of energy compared to always speculating.

Figure 5.2: Evaluation of MAP against micro-benchmarks that rely on accesses to the L2.

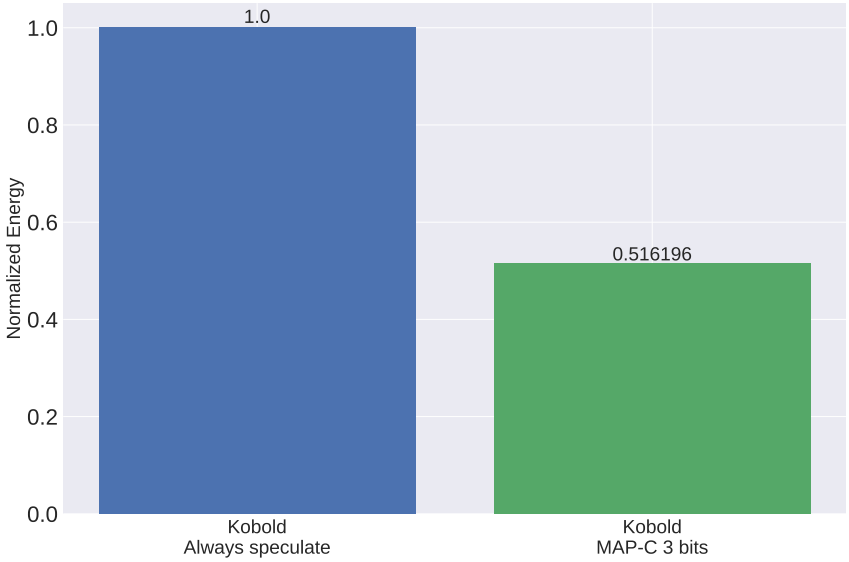
more speculative requests to the LLC than necessary. Fig. 5.2b shows the comparison of the number of messages sent to the LLC. The Memory Access Predictor is able to reduce the total number of messages sent to the LLC, reaching a similar total number as never speculating. This results in a significantly lower energy consumption compared to always speculating, as shown in Fig. 5.2c, where the Memory Access Predictor is able to save around 91% of the energy that would have been wasted to unnecessary speculative requests to the LLC.

5.1.3 Mixed Workloads

We have seen that across simple workloads that mostly access one level of the cache hierarchy, the Memory Access Predictor is able to achieve its goals well. Now we evaluate the Memory



(a) MAP improves overall performance compared to never speculating.



(b) MAP improves performance while saving nearly half of the energy wasted to unnecessary speculative requests to the LLC.

Figure 5.3: Evaluation of MAP against micro-benchmarks that rely on both accesses to the L2 and the LLC.

Access Predictor across a mixed workload that accesses both levels of the cache hierarchy.

Fig. 5.3a presents how well the Memory Access Predictor performs against micro-benchmarks that require access to both the L2 and the LLC. For the Conditional Array Sum application, the Memory Access Predictor is able to achieve an even better speedup than always speculating due to being able to hide the L2 miss latency when the requested data is in the LLC, and relying on the shorter L2 hit path when the data is in the L2. However, for the Link List application, the

Memory Access Predictor is not able to achieve the same speedup as always speculating, as the prediction accuracy is only around 80%, but it still performs better than never speculating.

Overall, the Memory Access Predictor consistently outperforms never speculating while retaining additional energy consumption low. Fig. 5.3b shows that in a representative application, the Memory Access Predictor saves around 50% of the energy that would have been spent on unnecessary speculative requests to the LLC, while being able to match the performance from always speculating. The average energy savings from speculation in the micro-benchmarks presented in Fig. 5.3a is around 56.7%.

5.2 Sensitivity Study on Parameters

In this section, we discuss how varying each of the predictor design's parameters can affect the overall prediction accuracy. We ran each predictor and the chosen parameter across different micro-benchmarks that rely on accesses to different levels of the cache hierarchy. The preferred cache level access is listed under the micro-benchmark name in parentheses.

5.2.1 Memory Access Counter

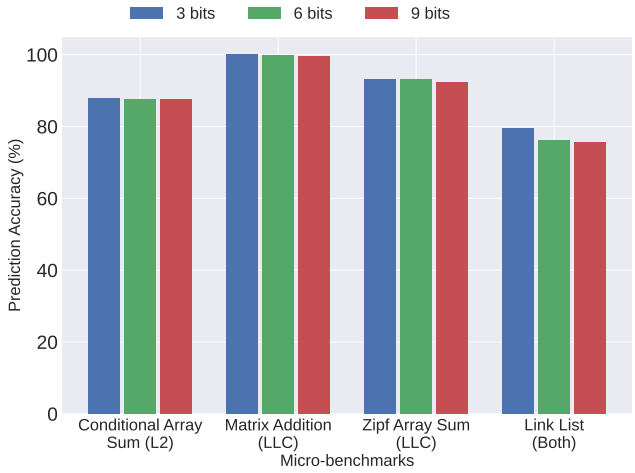


Figure 5.4: Micro benchmarks with results from running various sized counters with MAP-C.

For the Memory Access Counter design, we evaluated different counter sizes. The performance difference between each counter size did not vary much for each micro-benchmark, except for the Link List application, which roughly had half of its data in the L2, and the other half in the LLC. In this application, this predictor type did not perform as well as a result of using one saturating counter, so the predictor was not able to differentiate between different cache access patterns. The 3-bit counter consistently outperforms the other sizes, likely due to being small enough to still remain responsive to changes in access patterns, while still being big enough to not respond incorrectly to false patterns.

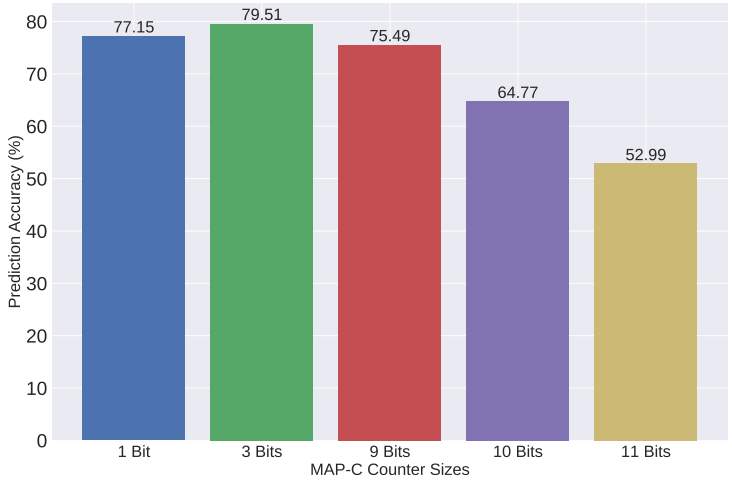


Figure 5.5: MAP-C's prediction accuracy falls off further if we decrease or increase the counter size more in the Link List benchmark.

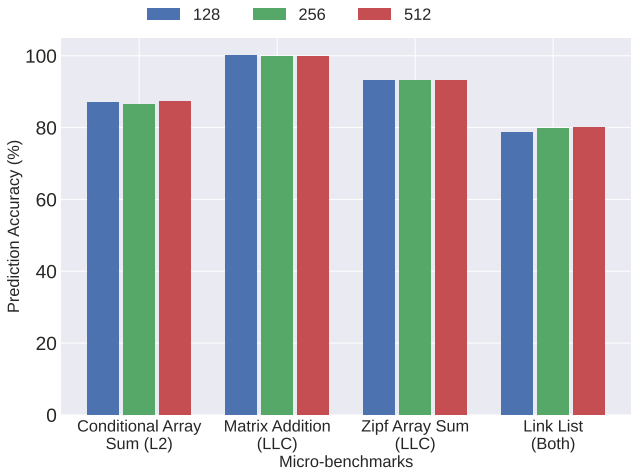


Figure 5.6: Micro benchmarks with results from running various sized tables with MAP-T.

Fig. 5.5 presents the prediction accuracies when we push the counter size more in both directions. The prediction accuracy has a steep drop off after 9 bits, and is slightly lower at 1 bit. This result highlights the short-lived and straightforward nature of our micro-benchmarks, where smaller counter sizes are preferred due to being more responsive.

5.2.2 Memory Access Counter Table

We evaluated different counter table sizes for the Memory Access Counter Table. The counter size used is 3-bits, which was chosen based on the best performing counter size from the previous experiment. Across all the micro-benchmarks, the table size that performs most consistently well is 512. Unlike the Memory Access Counter, the Memory Access Counter Table performs better on average for the Link List application, due to being able to differentiate between different cache

access patterns. Fig. 5.7 shows the effect of the higher average prediction accuracy. MAP-T is able to reduce the number of messages from the engine to the LLC by a small amount. The average speedup between the two designs are the same. The reason behind the unexpectedly small improvements in speedups and energy savings is likely due to the short-lived nature of the applications we performed our evaluation on. In longer-running benchmarks, we expect MAP-T's higher prediction accuracy to have a more significant improvement on the simulation seconds and energy consumption.

In Fig. 5.8, we can see that increasing the table size too much can cause the accuracy begins to drop significantly as a result of having too many separate counters to be able to track any meaningful pattern in cache accesses. Conversely, as we decrease the table size, the accuracy begins to drop slightly due to the table no longer being able to differentiate the two cache level access patterns well. Once the table size reaches 64, the prediction accuracy saturates at around the same accuracy as MAP-C with 3 bits.

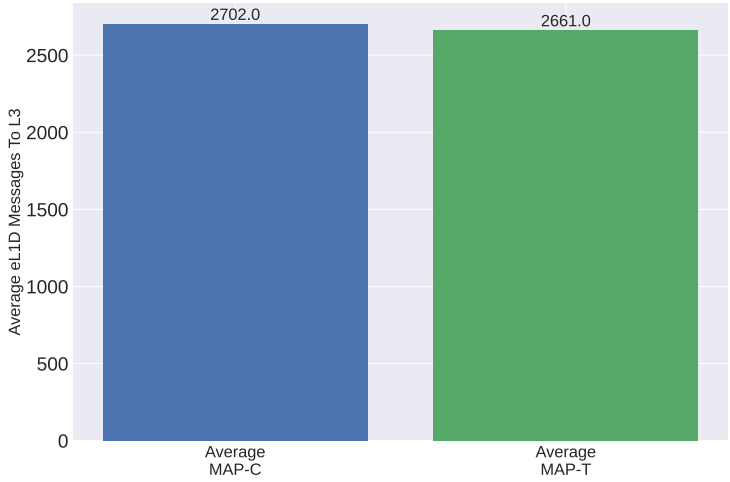


Figure 5.7: Average number of messages sent from the eL1D to the LLC across MAP-C counter sizes vs. MAP-T table sizes.

5.2.3 Memory Access Ratio

For the Memory Access Ratio design, we varied the threshold ratio value across the micro-benchmarks. A lower threshold ratio value means that the predictor requires a lower ratio of responses from the LLC to responses from the L2 for the predictor to predict speculation. Overall, the ratio value of 2.0 performed the best across the applications. Different ratio values do not change the accuracy much across our micro-benchmarks, and this predictor design seems to perform consistently worse compared to other predictor types. This is, again, highlights a limitation in our study—the Memory Access Ratio would likely perform better on longer-running benchmarks.

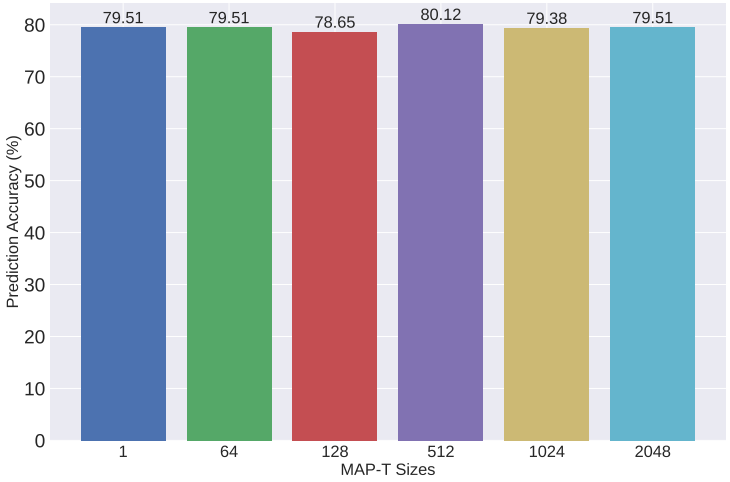


Figure 5.8: MAP-T's prediction accuracy falls off if we decrease or increase the table size more in the Link List benchmark.

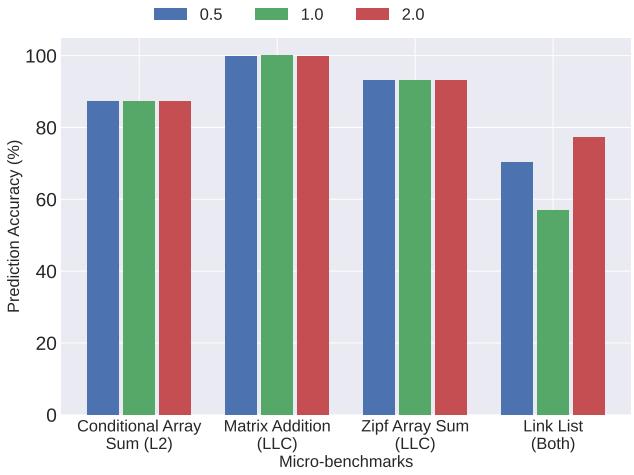


Figure 5.9: Micro benchmarks with results from running various threshold values.

5.3 Comparisons Across Predictor Types

Finally, we chose the best performing parameter value from each predictor type and compared each design's accuracy across our micro-benchmarks. The average prediction accuracy of each predictor design ends up being extremely close to one another, with MAP-C outperforming the other predictors at 90.1%, followed by MAP-T at 90.0%, and MAP-R at 89.9%. Evaluating against longer-running applications would likely yield a more varied average prediction accuracies across different predictors.

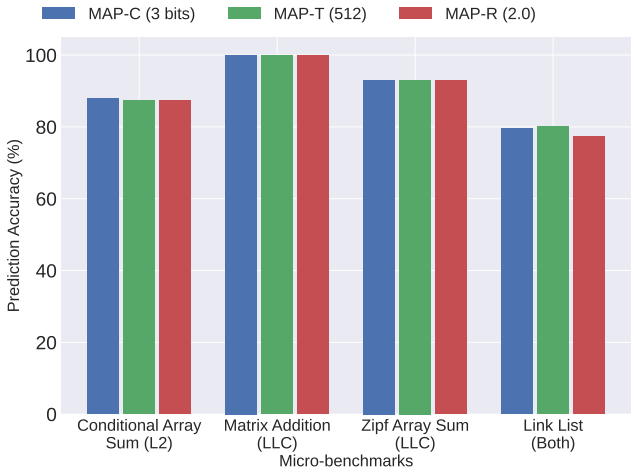


Figure 5.10: Micro benchmarks with results from running various predictor types

Chapter 6

Conclusion

In this thesis, we proposed a Memory Access Predictor that can be used by near-cache engines to access the cache level preferred by the running application. We evaluated the Memory Access Predictor across various micro-benchmarks, showing that the Memory Access Predictor is able to achieve high prediction accuracies. When these predictions are utilized by the engine, we show that we are able to achieve similar speedups as always sending speculative requests, while keeping the additional energy consumption from speculation low.

6.1 Future Work

The biggest limitation of our work is in its evaluation. To more accurately simulate the predictor under more realistic circumstances, we should model the CPU as an Out-of-Order core. The next step after that is to evaluate the predictor on a multi-core system. Finally, as mentioned earlier, many of the applications we evaluated the predictor on is too short to see much meaningful differences in prediction accuracy, speedups, and energy savings under different predictor configurations. Evaluating the predictor against long-running benchmarks with varied cache access patterns (e.g., with phase changes) would definitely be an important next step for this thesis.

Bibliography

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute caches. In *Proc. of the 23rd IEEE intl. symp. on High Performance Computer Architecture (Proc. HPCA-23)*, 2017. 2.1

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung 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. 1, 2.1

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL `https://doi.org/10.1145/2024716.2024718`. 4

[4] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H Loh, Don McCaule, Pat Morrow, Donald W Nelson, Daniel Pantuso, et al. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–479. IEEE Computer Society, 2006. 1

[5] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur 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. 1

[6] Jennifer Brana, Brian C. Schwedock, Yatin A. Manerkar, and Nathan Beckmann. Kobold: Simplified cache coherence for cache-attached accelerators. *IEEE Computer Architecture Letters*, 22(1):41–44, 2023. doi: 10.1109/LCA.2023.3269399. 1, 1, 2, 2.2.2

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2014. 1

[8] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, 2012. 2

[9] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013. 2

[10] William J. Dally. GPU Computing: To Exascale and Beyond. In *Supercomputing '10, Plenary Talk*, 2010. 1, 2

[11] Benoıt Dupont de Dinechin, Renaud Ayrignac, P-E Beaucamps, Patrice Couvert, Benoıt Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *Proc. of the High Performance Extreme Computing Conference*, 2013. 1

[12] Mingyu Gao, Grant Ayers, and Christos 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. 1

[13] John Hennessy and David 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. 1, 2

[14] Mark Horowitz. Computing's energy problem (and what we can do about it). In *ISSCC*, 2014. 1, 2

[15] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen 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. 1

[16] 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

[17] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495), 2020. 1, 2

[18] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013. 2

[19] 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. 1, 2.1, 2.2

[20] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. Snnap: Approximate computing on programmable socs via neural

acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015. 2

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

[22] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. Opportunistic computing in gpu architectures. In *Proc. of the 46th annual Intl. Symp. on Computer Architecture (Proc. ISCA-46)*, 2019. 2.1

[23] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, and Vijayalakshmi 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. 1

[24] 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

[25] 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. 1, 2.1, 2.2, 2.2.2

[26] 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, 3.3

[27] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *Proc. of the 50th annual IEEE/ACM intl. symp. on Microarchitecture (Proc. MICRO-50)*, 2017. 2.1

[28] Po-An Tsai, Changping Chen, and Daniel 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. 1

[29] 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

[30] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford 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*, ASPLOS XV, 2010. 2

[31] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven 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. 2

[32] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony 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. 2.1

[33] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. Near-stream computing: General and transparent near-cache acceleration. 2022. 2.1, 4, 5

[34] 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

[35] 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

[36] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proc. HPDC*, 2014. 1