UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Towards a Transparent and Efficient Far Memory System

Permalink
https://escholarship.org/uc/item/1qx9p6gx

Author
He, Zijian

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Towards a Transparent and Efficient Far Memory System

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science and Engineering

by

Zijian He

Committee in charge:

Professor Yiying Zhang, Chair
Professor Amy Ousterhout
Professor Jishen Zhao

2023

Copyright

Zijian He, 2023

All rights reserved.

The Thesis of Zijian He is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

TABLE OF CONTENTS

Thesis Approval Page iii

Table of Contents iv

List of Figures vi

Acknowledgements viii

Abstract of the Thesis ix

Introduction 1

Chapter 1 Background and Related Work 7
- 1.1 Far-Memory Systems 7
- 1.2 Optimizations for Memory Accesses 9
- 1.3 Multi-Level Intermediate Representation (MLIR) 11

Chapter 2 Mira Overview 13
- 2.1 Mira's Runtime System 13
- 2.2 Mira in Action 14

Chapter 3 Cache Section Configurations 17
- 3.1 Cache Sections 18
- 3.2 Cache Line Size 21
- 3.3 Cache Structure 23
- 3.4 Section Size 25

Chapter 4 Transparent Disaggregation 29
- 4.1 Convert to Far-Memory Accesses 29
- 4.2 Lowering Remote Operations 30

Chapter 5 Program Optimizations 34
- 5.1 Adaptive Prefetching 34
- 5.2 Eviction hints 37
- 5.3 Selective Transmission 38
- 5.4 Batching Requests 38
- 5.5 Data Communication Methods 39
- 5.6 Function Offloading 39

Chapter 6 Implementation 41
- 6.1 Far-Memory Abstraction in MLIR 41
- 6.2 Convert To `rmem` Dialect 43
- 6.3 Allocation with Far-Memory 43
- 6.4 More on Lowering Routines 44

6.5 Function Offloading 45
6.6 Cache Section Runtime 46
6.7 Multi-Threading Support 46

Chapter 7 Evaluation 48
7.1 Settings 48
7.2 End-to-End Performance 49
7.3 Runtime Overhead 52
7.4 Compilation Time 53
7.5 Performance Deep Dive 53

Chapter 8 Conclusion 57

Bibliography 59

LIST OF FIGURES

Figure 2.1. **Mira Overall Flow.** 14

Figure 3.1. **(Simplified) Code Example of Graph Traversal.** 17

Figure 3.2. **Overall performance (left) and breakdown (right) of the graph traversal program.** 18

Figure 3.3. **Effect of Cache Section Separation.** 20

Figure 3.4. **Sampled real network performance** 22

Figure 3.5. **Section-specific cache overhead (left) and illustration of data amplifications (right).** 23

Figure 3.6. **Illustration of cache structure effect** 24

Figure 3.7. **(Augmented) Code Example of Graph Traversal.** 26

Figure 3.8. **Illustration of the cache-capacity-performance relationship (left) and Mira's decisions on the section size (right) of the augmented graph traversal program.** 27

Figure 4.1. **Illustration of Mira's disaggregation flow.** 30

Figure 4.2. **Example of propagating remotability** 31

Figure 5.1. **Code example of accesses in nested loops.** *The given MLIR is simplified for readability.* 35

Figure 5.2. **Mira's analyze results for touched memory regions in each loop layer.** 36

Figure 5.3. **Effect of prefetch and eviction hint on the original graph traversing program.** 37

Figure 6.1. **Optimized graph traversal example. We only show prefetching and eviction flush in this case.** *The given MLIR is simplified for readability.* 42

Figure 6.2. **Example of lowering** `rmem.paddr` **for a specific cache structure.** 45

Figure 7.1. **DataFrame Performance.** *In this comparison, we do not consider function offloading.* . . 50

Figure 7.2. **GPT-2 Generation Performance.** 51

Figure 7.3. **MCF Performance.** 52

Figure 7.4. **Runtime Overhead Comparison. The memory is set to the required size of native program.** 53

Figure 7.5. **Mira Performance gain breakdown for all applications. Listed techniques are applied inclusively from left to right, with the generic swap-based section as the baseline.** . . 54

Figure 7.6. **Mira Iterative optimization of MCF and GPT2.** 56

ACKNOWLEDGEMENTS

I wish to express my gratitude to Professor Yiying Zhang, who served as the chair of my committee. I was lucky to find support in Yiying, it is her uncanny ability to contribute something useful to a discussion on almost any technical topic and her generous investment in improving the reasearch skills of her students that made this thesis possible.

I would also like to acknowledge Zhiyuan Guo, a diligent and insightful researcher at UCSD WukLab, whose dedication to efficiency and profound understaning of the matter sets a standard that I can only strive to attain. The work in this thesis would not have happened without him.

I am also grateful to Professor Jishen Zhao and Professor Amy Ousterhout for their time in serving on the committee and valuable feedbacks on my thesis.

This thesis, in full, is currently being prepared for submission for publication of the material, coauthors include Zhiyuan Guo and Yiying Zhang. The thesis author was one of the primary authors of this material.

ABSTRACT OF THE THESIS

Towards a Transparent and Efficient Far Memory System

by

Zijian He

Master of Science in Computer Science and Engineering

University of California San Diego, 2023

Professor Yiying Zhang, Chair

Memory-intensive applications suffer significant performance degradation when their working sets exceed available memory capacity, which can result in swapping with slow disks. Far memory, where memory accesses are directed to other connected nodes, has become more popular in recent years as a solution to expand memory size and avoid memory stranding.

Prior far memory systems have taken two approaches: 1) devising a swap system that uses far memory as a backup device and transparently exposes these regions to unmodified applications, and 2) introducing a new programming model/data structure that interfaces with far memory runtime. The former requires no program changes but comes with a

significant performance penalty, while the latter requires considerable developer efforts to adopt and tune new APIs despite potential performance gains.

Our key insight is that by capturing both statically known and dynamically monitored program behaviors, we can optimize both the program itself and the underlying runtime, resulting in a notable performance boost. Furthermore, we propose automating this process within the compiler to achieve a certain level of transparency. In this thesis, we introduce Mira, a far-memory system that transforms unmodified C/C++ programs to adopt remote memory accesses and optimizes them by tailoring runtime support to their specific behaviors. Our evaluation demonstrates that Mira significantly enhances workload performance, particularly in terms of execution time, surpassing previous swap-based systems and programming models by up to $18\times$.

Introduction

Today's data centers face significant constraints in terms of available memory (RAM) resources. The increasing popularity of memory-intensive workloads, such as in-memory key-value stores and machine learning applications, leads to a rapid growth in memory requirements. Typically, resources are allocated based on peak usage to avoid significant performance degradation. However, this approach often results in severe under-utilization and imbalanced provisioning across servers. According to Google's cluster trace, nearly 30% of server memory remains unused for minutes [1]. These observations highlight the need for addressing the memory bottleneck by leveraging memory beyond the physical boundary.

By integrating a second-tier memory into the existing memory hierarchy, the local DRAM can be effectively extended with reasonable overhead. This enables the same job to continue running even with lower memory capacity. In this thesis, our major focus is on sharing cold memory with other compute nodes connected through the network. The key idea is compatible with other implementations of far memory.

Existing approaches

Some existing systems utilize the swapping mechanism, which involves data exchange between secondary storage and RAM at a fixed-size unit (pages), to incorporate far memory as a slower-tier device [2, 3, 4, 5]. While this method can be easily integrated into the kernel and provides a high level of transparency, it introduces significant overhead. The overhead is primarily caused by pagefault handling and blocking during page arrival. In certain cases, a Linux kernel might be configured for busy waiting during swap events to avoid context switching and interrupt handling, resulting in wasted CPU cycles that could have been allocated for other tasks.

These page-based systems also introduce another issue, amplification, which occurs when physically fetched or written data exceeds the logically required size. It can be traced back to the inflexible and coarse granularity of swapping operations. For instance, even if only a small fraction of a page is modified, the entire page is marked as dirty, leading to substantial write amplification during the swap-out process. Similarly, a whole page is fetched despite the actual access range within that page, causing significant read amplification. Amplification issues are prevalent in real-world applications, with amplification ratios ranging from 2× to 31× for 4KB pages [6]. These amplifications not only decrease network utilization but also exacerbate performance degradation.

Another group of approaches proposes new programming models that directly embrace far-memory access through runtime APIs in applications [7, 8, 9]. This solution circumvents the need to handle page faults and provides more control over data transmission. However, rewriting existing applications to adopt these new APIs is often error-prone, time-consuming, and requires expert knowledge of the implications involved.

Our approach

This thesis aims to provide a solution that combines the advantages of both worlds: enhancing the efficiency of the far memory system while reducing the burden on developers when adopting the proposed system. Our answer lies in a program-behavior-guided far-memory system. Specifically, we leverage the knowledge of program behavior to guide the configuration of the runtime system and optimize far memory accesses jointly. To achieve this, we explore an uncultivated layer in far-memory research: the compiler. We utilize static analysis with runtime profiling to capture program behaviors. Additionally, our compiler transforms the input program to seamlessly embrace far memory without requiring explicit modifications from the programmers' side.

The introduction of program analysis techniques provides a means to unravel the inner workings of an application that was once regarded as a black box. This enables the identification and understanding of behaviors that are challenging to extract solely from traces. A prime example of such behavior is observed in the context of indirect memory access, represented by the expression `B[A[i]]`. Prefetching data accurately in these scenarios presents considerable difficulties due to the intertwined nature of touched addresses from `B` and `A`, as well as the strong data dependence exhibited by the loaded memory. These characteristics impede simple matching to well-defined patterns such as stride or sequential access [10]. However, through the use of static analysis techniques, the precise instructions responsible for computing the target address can be derived by traversing the intermediate representation (IR) graph. This capability enables the elimination of redundant memory faults resulting from misguided prefetches.

While many existing approaches rely on fixed runtime settings, we propose a novel approach that utilizes program behavior to guide system configurations. This decision is rooted in the observation that the efficiency of program execution can be greatly influenced by the underlying runtime settings, and mismatched configurations may even result in a significant performance degradation. For instance, employing a large management granularity can lead to

read/write amplification when the access pattern is predominantly random. On the other hand, adopting object-level granularity can introduce excessive space overhead for bookkeeping and suboptimal access latency when strong locality patterns are present. Based on these findings, we offer highly customizable parameters such as cache line size, cache architecture, and capacity for our runtime system that treats the local space as a cache buffer for far memory,

Despite the efficiency of new systems, users can not easily adopt them if they require hints or explicit use of proposed APIs. We find the compiler a promising layer to maintain the benefit of new abstractions while not requiring any additional efforts from the programmer. In detail, our compiler will perform program analysis, tune the runtime settings with analyzed results and profile information, transform the application, and optimize it for far-memory access. By automating this process, our system can achieve superior efficiency and transparency simultaneously.

Challenges

Using program analysis and dynamic profiling to optimize memory access is a well-established concept in conventional server settings, considering the CPU cache and main memory [11, 12, 13, 14]. However, achieving an optimal cache configuration that caters to the entire application is not always feasible. In many cases, a program exhibits multiple memory access patterns involving different objects or occurring at different phases, each of which can benefit from distinct cache settings.

Another challenge, which is commonly encountered in software-defined far memory systems, pertains to the efficiency of runtime implementations. For instance, the AIFM system [7] utilizes remoteable pointers to represent memory objects backed by far-memory. To access these objects, additional instructions such as validity checks, and swapping in/out need to be executed. While AIFM aims to offer fine-grained control over data transmission, the dereference overhead associated with each object introduces significant runtime overhead in terms of both

time and space. This issue becomes particularly pronounced when dealing with a large number of disaggregated memory objects that are small in size. In one of the benchmarks we tested, which involved a graph traversing algorithm, the memory overhead incurred from managing metadata for each pointer even exceeded the actual memory required by the program itself. Similarly, our runtime caches also incorporate additional instructions and metadata. To fully harness the benefits of our proposed system, it is crucial to minimize and marginalize this overhead, as otherwise, it can overshadow the advantages provided by the system.

To address the initial problem, we propose dividing the local cache into sections, each tailored to a specific behavior exhibited by the program. The parameters of each section can be individually customized to optimize them exclusively for the corresponding access pattern. For instance, a directly mapped cache with a large cache line size and moderate capacity can be a suitable match for sequential access patterns. On the other hand, random access patterns with poor locality may benefit from a large set-associative cache with a higher number of ways.

In a higher-level perspective, we transform the initially complex optimization challenge into a pattern-section matching problem, which is easier to comprehend and resolve. Specifically, we determine the capacity, line size, structure, prefetching/eviction policy, and RDMA operations (such as one-/two-sided communication) for each cache section based on the results of static analysis and dynamic profiling results. Furthermore, we can further optimize far-memory accesses that involve a dedicated cache section by leveraging the corresponding cache settings, enabling more efficient utilization of our runtime system.

For the second challenge, our efforts are mainly from two aspects. Firstly, at the code generation stage, we leverage access locality to convert as many far-memory-dereference to native main memory access as possible to avoid unnecessary API calls. Secondly, in terms of runtime implementation, we retain the concept of a cache line so that each metadata instance is associated with a group of objects.

Contributions

In this thesis,

1. We propose Mira, a transparent and efficient far-memory system that co-optimizes the program and the runtime system jointly through compiler techniques.

2. We implement an efficient and highly configurable cache system for the far-memory.

3. We implement the compiler part of Mira on top of Multi-Layer Intermediate Representation [15], MLIR, which allows us to introduce new abstractions easily.

4. We illustrate the effectiveness of detailed design using synthetic micro-benchmarks and evaluate the end-to-end performance on three real-world applications: MCF [16], DataFrame [17] and language model (GPT-2 [18]) inference. We compare Mira with systems from both worlds to demonstrate our superior performance: Fastswap [3], an efficient implementation of kernel-level swap system integrated with far-memory, Leap [19], another swap-based far-memory system equipped with a powerful prefetching mechanism and AIFM [7], a programming model that accesses far-memory under the hood.

Chapter 1

Background and Related Work

This chapter aims to provide background information on critical concepts and to review the relevant literature.

1.1 Far-Memory Systems

Far memory represents an emerging data center design paradigm that maximizes cluster resource utilization by leveraging the untapped memory resources of remote servers or memory blades. In a typical setup, an oversubscribed machine retrieves physically disaggregated memory through a low-latency network, utilizing technologies such as RDMA or even traditional TCP.

System-level far-memory solutions

Far-memory systems are commonly implemented using page-based swapping techniques. InfiniSwap [2] was the pioneering remote memory swap system that utilized RDMA, while FastSwap [3] further enhanced its performance with superior scheduling and polling mechanisms. Leap [19] leverages a process's predominant access pattern to prefetch memory pages, aiming to minimize remote-memory accesses in critical execution paths. More recent works, such as Canvas [5] and Hermit [20], have focused on improving Linux's swap system by enhancing isolation mechanisms in multi-application environments and asynchronously executing non-urgent but time-consuming tasks. Additionally, LegoOS [4] presents an alternative non-Linux system that facilitates swapping of 4 KB pages between a compute node's "extended cache" and

disaggregated memory.

These page-based systems commonly encounter two issues: firstly, they exhibit fixed and coarse swap granularity, typically based on 4 KB pages, resulting in considerable wastage of network bandwidth due to amplifications and a subsequent decline in application performance [21]. Secondly, these systems lack awareness of program semantics, which is essential for enabling a range of optimizations.

There are also attempts to address these barriers. Emerging hardware, such as CXL [22] and research prototypes [23, 24], facilitates cache-line-sized accesses to far memory at significantly faster speeds than current network communication. Additionally, CXL eliminates page fault overhead from the critical path by detecting far-memory access using CPU cache misses. Software-defined systems built upon this technique can benefit from this low latency and fine access granularity [21, 25]. However, none of these systems consider program semantics or configure local cache based on program behavior.

3PO [26] is a recent study that leverages the characteristic of *oblivious* applications, which exhibit access traces independent of the input, to proactively plan far-memory prefetching in a consistent execution environment. In contrast to 3PO, Mira utilizes program analysis and profiling techniques to capture program behavior without imposing strict requirements on application obliviousness. Moreover, we adopt a co-design approach for the runtime system based on the observed program behavior.

New programming models

This type of far-memory system introduces new libraries that facilitate access to remote memory under the hood. Many RDMA-based systems [9, 27, 28, 29] expose RDMA-like APIs to enable low-level operations such as direct memory read/write at remote servers, while others implement interfaces resembling data structures to align with legacy applications [7, 8, 30]. These systems offer promising performance by granting greater control over far-memory accesses, but at the same time, they place an additional burden on programmers. Manually porting existing

applications presents two challenges:

1. Remotealizing even a single object can be intrusive, requiring modifications to access interfaces wherever this remoteability propagates. In our initial attempt to disaggregate the MCF workload [16] using our runtime supports, we observed that more than 90% of functions needed modification when placing only one data structure in far-memory. This renders the rewriting process error-prone and time-consuming.

2. Although library developers can strive for optimal optimization of each API, additional effort is required to understand the implications of their interaction with the original program behavior in order to utilize these libraries effectively.

There is also a recent work that aims to leverage the efficiency of user-space block cache: Tricache [31]. It achieves user transparency by instrumenting address translation at compile time, prior to the actual load/store operations. However, similar to other software-defined caches, Tricache still encounters efficiency concerns due to the overhead associated with executing library codes for each memory dereference. While Tricache relies on a two-level cache structure resembling a hardware TLB to optimize runtime calls, Mira takes a step further by incorporating program analysis to capture access locality and eliminate unnecessary instructions. Additionally, Mira possesses the capability to configure the cache structure based on the program's behavior, a feature not explored in Tricache.

1.2 Optimizations for Memory Accesses

While memory accesses in a traditional, non-far-memory environment have been extensively optimized across various layers, to the best of our knowledge, no research has focused on co-designing program analysis, compilers, and a configurable cache.

CPU cache optimizations

Numerous compiler-level and system-level approaches have been suggested to enhance application performance on CPU caches. These solutions can be broadly categorized into three categories. The first category involves modifying programs and/or data layouts to create more cache-friendly memory access. Techniques such as data structure padding, peeling, field rearrangement, and separation of hot and cold code regions are employed for this purpose [11, 12, 13]. The second approach assigns distinct CPU cache spaces to different parts of applications. An example of this technique is CPU cache coloring, which assigns different memory regions to different regions or levels of the cache to mitigate cache conflicts [32, 33, 34]. The third type utilizes run-time profiling results, such as profile-guided optimization (PGO), to guide memory-access optimizations. APT-GET [35] is an example of a system that combines compiler-based prefetching with dynamically profiled execution times to enhance prefetching timeliness.

In contrast to these approaches, Mira takes a departure by introducing a novel co-design that integrates configurable caches, program analysis, and the compiler to optimize memory accesses in a far-memory environment. This unique combination enables Mira to effectively address the challenges posed by existing far-memory systems and achieve optimized memory performance.

Configurable caches

There has been some work toward configurable CPU caches and utilization of software mechanisms to configure such architectures [36, 37, 38]. For instance, Jenga [37] reconfigure the cache hierarchy, both in the number of levels and size of each level, according to the cache access latency that can be calculated from the hardware miss counter. Lee et al. [38] proposed a customized cache structure for streaming applications specifically by analyzing the memory access traces offline. These solutions concentrate on the architecture and systems level and do not consider compiler optimizations. Moreover, they all require special hardware to realize the configurable cache, which may not be feasible in current data centers. Mira, on the other hand,

whose local cache is fully implemented at software and is jointly optimized with the program itself through program analysis and profiling results.

Scratchpad memory is another type of hardware-manageable cache that can be controlled by software. Several studies have concentrated on developing effective strategies for determining which data to store in the space-limited scratchpad memory [39, 40, 41]. For example, Susu et al. [41] utilizes static analysis and code transformation to perform space planning on scratchpad memory for an accelerator. However, these works do not configure scratchpad memory hierarchy based on program behavior but merely seeking for good data placement and scheduling to fit the scratchpad.

1.3 Multi-Level Intermediate Representation (MLIR)

MLIR [15] is a compiler framework that enables multiple abstractions at various levels. These abstractions are referred to as *dialects*. Presently, MLIR supports numerous dialects for common operations, encompassing areas such as machine learning, LLVM, memory, control flow, arithmetic, etc. We have opted to develop our compiler within the MLIR framework since it facilitates the use of multiple frontend languages and backend architectures. Additionally, it provides us with the flexibility to include various far-memory abstractions and code optimizations as dialects at various layers while reusing the existing MLIR dialects and their optimizations.

The code snippets given in this thesis are all in the form of MLIR with certain simplifications. Apart from some self-explanatory operations such as for loops or memory loads, we explain other instructions/types that also occur in this thesis below.

geteleptr

It is utilized to retrieve the address of a subelement within an aggregate data structure. Its sole purpose is to perform address calculation and does not involve any memory access.

affine.for

This control flow is nothing more than a normal for loop with some additional constraints. It often appears in the machine-learning community.

memref type

It represents a reference to a memory region, providing similar functionality to a buffer pointer but with implications to the underlying data layout. `memref<1024x1024xi32>` stands for a group of 32-bit integers arranged in the format of a row-major 2-D matrix.

vec.load

This operation literally reads a vector from the memory. The resulting value will be used in other operations that support vectorization.

affine.load/store

These operations are essentially normal memory read/write with specific constraints on its offset indices and the target that is being accessed.

Chapter 2

Mira Overview

Mira consists of several components, including a compiler responsible for analysis and transformations, a runtime system designed for both local-compute and far-memory nodes, and a profiling system. The overall flow of Mira is illustrated in Figure 2.1, which follows an iterative approach to adapt system configurations and user programs for far-memory accesses. The subsequent part of this section will provide an introduction to our runtime system, followed by a detailed walkthrough of the execution-compilation process.

2.1 Mira's Runtime System

Our runtime treats the local DRAM as the cache buffer for the far-memory node. Instead of utilizing a single cache for the entire program, we divide the local memory into independent sections that can be configured individually. These sections can be fine-tuned using parameters such as cache line size, capacity, and cache architecture, which are determined by our compiler based on the observed program behaviors.

The runtime system provides a set of APIs for accessing far-memory. These APIs facilitate the translation of remote addresses to native addresses that can be used by the local memory management unit (MMU). Additionally, it offers other primitive operations, including explicit prefetching and evictions, which the compiler can leverage for better performance.

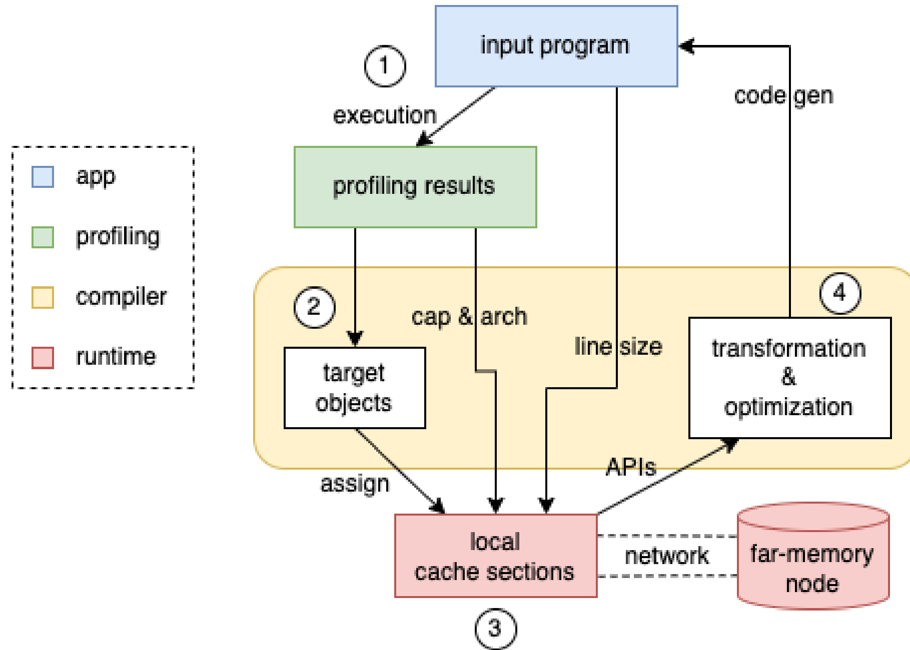


Figure 2.1. **Mira Overall Flow.**

2.2 Mira in Action

For the initial run, Mira employs a generic swap cache, similar to traditional page swap-based systems, for all heap objects. This initial execution serves as a starting point and allows for the profiling of necessary metrics for future optimizations. The information collected during each profiled run remains consistent and includes details such as memory allocation sizes, function execution time, and cache overhead within each function.

Based on the collected performance of individual functions and the sizes of objects, informed decisions are made regarding the division of cache sections. It is important to acknowledge that the complexity of program analysis and code generation/optimization increases with each additional section, which may not always be necessary. Hence, our focus lies in identifying the functions that are most affected by the current cache configuration and compiled code. We then isolate larger objects within these functions and allocate dedicated sections for them. These selected objects subsequently undergo our analysis and compilation process.

For the identified cache sections, we proceed to configure their parameters based on the observed program behaviors. To determine the cache line size, we analyze the object size and

examine the symbolic representation of a set of load/store addresses within the filtered function. This allows us to estimate the access granularity and strike a balance between amplification and data movement efficiency. The analyzed access sequence also aids in determining the appropriate cache structure, as we can minimize cache conflicts by adjusting the number of ways accordingly. However, it is important to avoid using an overly complex cache structure that may introduce noticeable per-access overhead. Deciding the optimal section architecture statically can be challenging. Therefore, we profile the cache overhead and make adjustments in subsequent execution-optimization trials. Additionally, by considering the profiled performance characteristics of each cache section, we can determine their respective sizes by interpolating the overall performance with a given local memory constraint.

For accesses to objects that are backed by far-memory, Mira applies code transformations to enable interfacing with local cache sections or even remote memory nodes directly. Our compiler converts ordinary operations like allocation, read and write, etc. to *remoteable* operations in the IR, which we refer to as *disaggregating* the program at the compile time. These remoteable operations remain at a high level during the analysis and optimizations and will be lowered to cache/rdma instructions eventually.

Apart from the conversion of memory accesses, we also offload functions as the far-memory node can be equipped with certain computation capabilities. However, the benefit of co-locating the function and data might be overshadowed by the longer execution time if the remote computation is slower. Therefore, our policy also takes the current cluster settings into consideration, and we assume an external monitor will keep our system informed about any discernible changes that might require re-compilation. Currently, our compiler is able to cache different function signatures that correspond to offloaded or native versions, and only change the call site to adapt to new decisions quickly. We envision this can be extended to just-in-time (JIT) compilers, enabling dynamic function offloading at runtime.

During the lowering pass, the inclusion of profiling code instrumentation can be initiated when necessary. This need may arise due to iterative optimizations on similar inputs, changes

in system environments, or adaptation to new inputs that result in different program behaviors. During normal runs, the binary is executed without the profiling instructions. Users have the flexibility to modify the default triggering conditions for a profiling and optimization trial using various methods, such as specifying a threshold for recompile iterations/performance gains or leveraging frameworks like AutoFDO [42] and Ding [43] to identify a "new" input.

Chapter 3

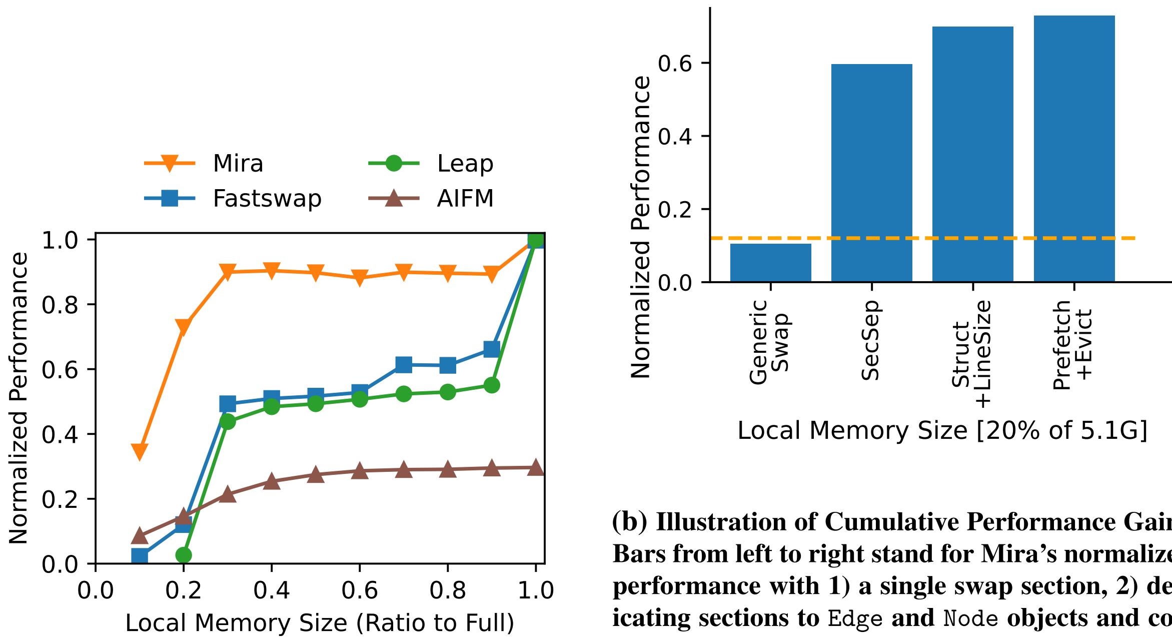
Cache Section Configurations

This chapter provides a detailed account of the process involved in identifying what to put into a cache section and determining various cache parameters, including section capacity, cache line size, and cache structure. To illustrate the key designs of Mira, we employ a simple graph traversal program depicted in Figure 3.1 as an example application. The program consists of two arrays: one for *Edges* and another for *Nodes*. It sequentially follows the edges and modifies the source and destination nodes of each traversed edge.

Figure 3.2a demonstrates the superior overall performance of Mira compared to other systems across various local memory sizes. Furthermore, Figure 3.2b provides a breakdown of performance gains, highlighting the effectiveness of several design components implemented in Mira. Throughout the paper, we present relative performance metrics, which are normalized against the execution of the native binary. A higher value indicates better performance (a value of 1.0 signifies comparable performance to the execution of the unmodified program with sufficient memory).

```
1 edges, nodes = malloc()
2 void traverse_graph(struct edge *edges) {
3   for (int i = 0; i < num_edges; i++)
4     update_node(edges[i], edges[i].from, edges[i].to);
5     // edges[i].from and edges[i].to point to nodes
6 }
```

Figure 3.1. **(Simplified) Code Example of Graph Traversal.**



(a) Graph Traverse Overall Performance.

(b) Illustration of Cumulative Performance Gains. Bars from left to right stand for Mira's normalized performance with 1) a single swap section, 2) dedicating sections to `Edge` and `Node` objects and configuring the section capacity, 3) line size and section structure tuning 4) far-memory access optimizations. *Orange lines show FastSwap performance.*

Figure 3.2. Overall performance (left) and breakdown (right) of the graph traversal program.

3.1 Cache Sections

By partitioning the local memory into sections, we have the flexibility to configure cache parameters based on specific program behaviors. However, it is essential to exercise caution when creating new sections, as this can introduce additional overhead in terms of program analysis and optimizations, and may result in suboptimal utilization of the local memory. Given these considerations, Mira adaptively determine which data and code regions to be paired with a cache section by focusing on large objects in functions that ”suffer” most from accessing the far-memory in each execution-optimization trial.

To measure this ”sufferance”, we have designed an algorithm with a dual purpose: 1) to identify the appropriate scope where significant improvement potential exists and the optimization context is sufficient, and 2) to account for control flow, which is crucial for capturing the program's runtime behavior. With these design principles, we estimate the suffering of a

function f w.r.t the current runtime setting S using the following formula:

$$\texttt{Suffering}_{f,S} = \frac{\texttt{Overhead}_{f,S}}{\texttt{Execution}_f}$$

$$\texttt{Overhead}_{f,S} = \sum_{c \in S} \texttt{Access_Lat}_{f,c} + \sum_{f' \in \lambda(f)} \texttt{Overhead}_{f',S}$$

where $\lambda(f)$ denotes callees within f, $\texttt{Access_Lat}_{f,c}$ refers to time spent within cache c in function f, and $\texttt{Execution}_f$ is the execution time of function f.

As the equation suggests, the cache overhead of f will further be recursively aggregated to the parent functions of f when analyzing its callers, and we count each function only once in case there exists mutual recursion (e.g., two functions calling each other). The recursive process prevents Mira from focusing solely on last-level functions, as optimizations such as prefetch (Section 5.1) and API call elimination (Section 4.2) can be better performed with a border analysis scope. It also embeds the execution graph efficiently without monitoring the call stack at runtime, leading to a more accurate description of the program's dynamic behavior. The overhead is also weighted against the function execution time to moderate interference from noises (i.e., code that executes efficiently).

We rank functions based on this index and by default select the top 10% of functions for analysis. These functions tend to have more potential for optimizing cache configurations and far-memory code. Since the overhead of callees is attributed to callers, we also include all functions called within the selected functions in the analysis scope. Once the functions are selected, we further narrow down the analysis to focus on large objects that are more likely to result in far-memory access and require significant space. By default, we choose the largest 10% of objects accessed within the selected functions. Users have the flexibility to customize these two thresholds to strike a balance between analysis overhead and convergence rate. In our empirical evaluations, we have found that the default settings perform well for real-world applications tested.

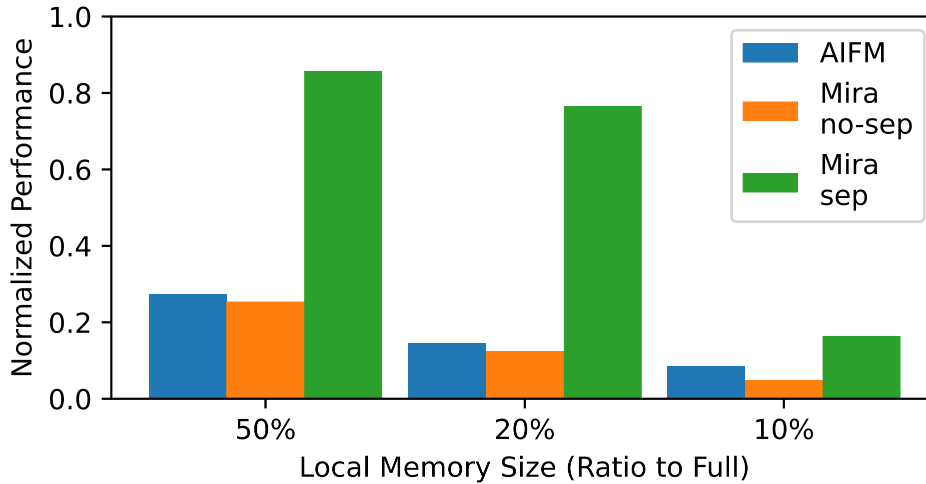


Figure 3.3. **Effect of Cache Section Separation.**

After analyzing the selected functions and objects and determining the suitable cache line size/architecture for their access patterns (as described in Section 3.2), we group similar patterns into one section while leaving the remaining objects in their own sections. This approach allows multiple objects to share a section if their access patterns are similar. Conversely, a single object may be allocated to different sections at different times if its access pattern changes.

It is important to note that there may still be discrepancies between our analysis/estimations and the actual runtime executions. As a result, allocating a separate cache section for an object may actually degrade the performance of its associated functions. For instance, if the original section contains two objects with certain discrepancies in their access patterns but only a small overlap in their lifetimes, separating the section for each object may lead to decreased available space for both objects, overshadowing the potential benefit of tuned parameters. However, identifying such information at compile time is difficult. Currently, if we detect any performance degradation through profiling, we revert to the configuration used in the previous iteration. In the worst case, a generic swap section is used for all target objects, as this is the default setting for the initial run, which aligns with other swap-based far-memory implementations.

In Figure 3.2b, we can observe that separating the cache sections (with capacity configuration as explained in Section 3.4) can achieve a 5.7× and 4.9× performance boost compared with

using our generic swap and with Fastswap. Figure 3.3 shows this effect under different memory pressures and includes AIFM's performance as a reference. Following the initial iteration, Mira divides the data into two distinct sections: one for the node array and another for the edge array. This segregation is based on the analysis result that the edge array is accessed sequentially, while the node array is not. The isolation prevents the interference of two access patterns and grants more local space to the memory-sensitive one (node section), resulting in a significant reduction in cache misses for the node array. However, at 10% capacity, node objects are experiencing severe conflict miss regardless of a dedicated cache section, thus the overall performance is not promising as shown in Figure 3.3.

3.2 Cache Line Size

Each of our runtime caches operates with a fixed size, known as the cache line size, which is inspired by CPU caches. When determining the cache line size, we adhere to two principles:

1. We aim to ensure that the size of a cache line does not exceed the data access granularity. This principle helps us mitigate read/write amplifications that can occur when the cache line size is too large.

2. It is beneficial to increase the cache line size when data items are frequently accessed contiguously, as long as the line size remains within the maximum transmission capacity of the network. This principle allows us to leverage network latency characteristics and exploit runtime access locality.

By adhering to these principles, we strike a balance between minimizing amplifications and taking advantage of access patterns and network characteristics to optimize the performance of our runtime caches.

We begin by estimating the optimal point at which data can be efficiently transmitted in blocks over the network. To achieve this, we profile the latency of one-sided writes against

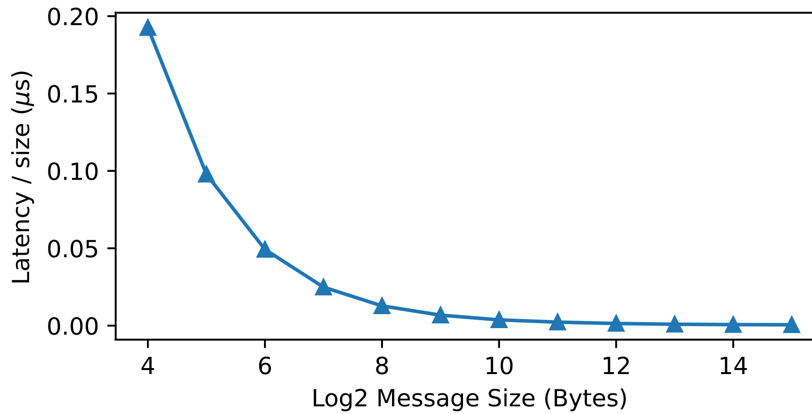
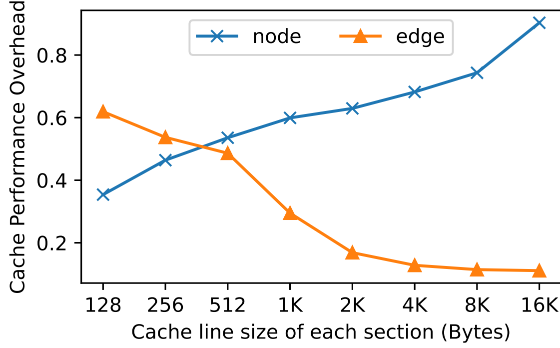


Figure 3.4. **Sampled real network performance**

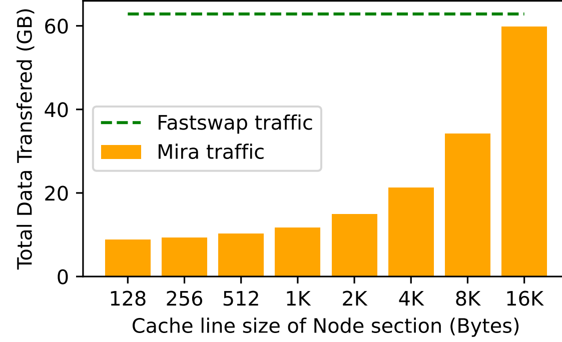
different message sizes. We identify the ”balanced” point as the pivot where the latency stabilizes. Beyond this point, further increasing the line size does not improve network efficiency. Using a profiled network latency curve, as depicted in Figure 3.4, which corresponds to the same environment in which we run the graph traversing benchmark, our analysis determines that a line size of 4KB satisfies the requirements for efficient data transmission.

Then for each access, we extract the instruction sequence for calculating the remote address and examine the access offset. Accesses to the same object will be averaged within a block (such as a function body or a for loop) and severe as the representative stride for that block when analyzing other regions. In the code example shown in 3.1, the average gap between accesses to edge nodes is 1 inside the loop block whereas the offset for nodes is not available. Since there is only one loop block in the function, the same results will be used when analyzing callers of *traverse_graph*. If the stride is a constant known at compile time and is smaller than 4KB, the compiler will configure the corresponding cache line size to the balanced point since 1) a larger line size can benefit access locality, and 2) data movement is relatively efficient at this point despite potential amplification. If the gap exceeds the balanced point, the compiler will treat it as random access and enforces a more conservative strategy since further increasing the line size will imply more cache conflicts.

We also consider all other cases where the offset is not statically known as random access and balance between the fine-grained control granularity and the space overhead for



(a) Effect of cache line size. The y-axis represents the cache access latency as described before, involving the execution of cache code and waiting for data arrival (lower the better).



(b) Monitoring of total data transmitted with different cache line sizes for *Node* objects.

Figure 3.5. **Section-specific cache overhead (left) and illustration of data amplifications (right).**

metadata management. The compiler will choose the smaller one between the size of access granularity (i.e., statically known structure size) and the lower bound of the line size, which can be customized by setting the space overhead threshold.

Figure 3.5 illustrates the cache overhead associated with utilizing various cache line sizes for the node and edge sections. For the node array, which is accessed randomly and requires a minimum size of 128 bytes to accommodate the data unit, adopting a smaller line size helps reduce conflicts and amplifications (as shown in Figure 3.5b). It is worth noting that Fastswap transfers 3$\times$ more data compared to Mira with the same 4K size setup, which can be attributed to the reduced miss rate achieved through section separation. On the other hand, the edge array is accessed sequentially. Using a larger line size allows the compiler to amortize the cost of far-memory dereferencing by offsetting subsequent accesses within the same line, which is already at the local side. However, this effect diminishes when the line size becomes relatively large.

3.3 Cache Structure

Mira supports three cache section structures: directly mapped, set associative, and fully associative, which are analogous to traditional CPU cache architectures. Similar to CPU caches,

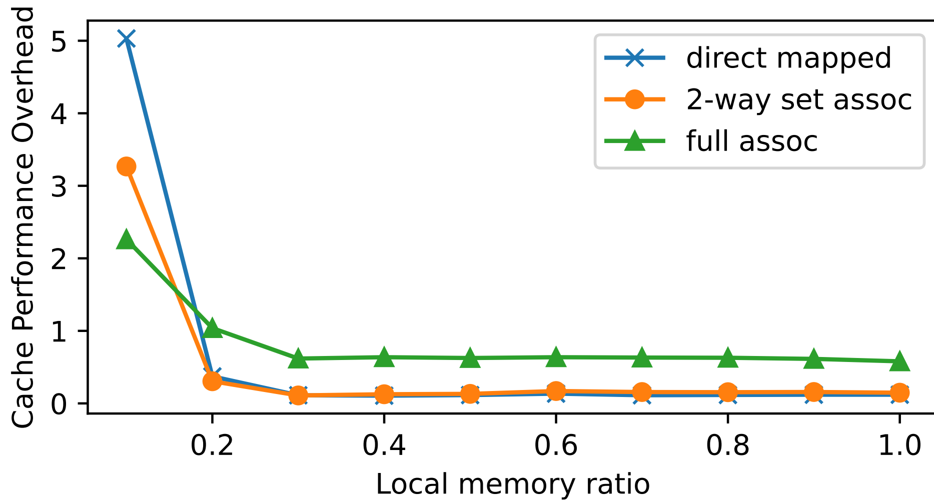


Figure 3.6. **Illustration of cache structure effect**

fully associative cache sections provide the most efficient utilization of cache space with no conflict misses. However, they come with a higher runtime overhead for cache lookup. This tradeoff shifts as we move towards set associativity and direct-mapped cache section structures. To determine the appropriate cache section structure, we analyze the program's access sequences within a code block and extrapolate the potential level of conflict that may occur. This analysis helps us make an informed decision about which cache structure is most suitable for the given program's behavior.

In detail, we estimate the number of data items K that are better to be cached within the function and choose the structure with the least complexity that can fulfill this requirement. If this information cannot be determined statically, we configure the cache section to be fully associative by default. In other cases, we select a direct-mapped cache if $K = 1$, a set-associative cache with K ways if $K \leq 16$, or a fully associative cache otherwise. This ensures that the accessed data items can be stored without evicting each other. In the example provided, we choose a direct-mapped cache for `Edge` to take advantage of the low runtime latency of this architecture without deteriorating the miss rate. For `Node`, we select a 2-way set-associative cache to accommodate two data items simultaneously.

There are other factors that can add complexity to the cache line configuration process.

One such factor is the consideration of prefetch optimizations. We perform cache line configuration after prefetch optimizations to take into account the requirements of the prefetch pipeline. For example, if Mira decides to prefetch node objects a few steps before the loop iteration that actually needs it, we increase the number of ways accordingly to ensure the prefetched items remain valid between the prefetch site and the access site. Another factor to consider is the difference in access speed between cache structures of varying complexity. Sometimes, this difference outweighs the impact of reduced miss rates. The effect of distinct cache structures on the `Node` section can be observed in Figure 3.6. With relatively sufficient local memory, full associativity always leads to suboptimal performance due to its high overhead. However, as the amount of local memory decreases, full associativity becomes a more favorable option. However, it can be challenging to anticipate this effect at compile time, we rely on profiling results to adjust the cache structure. For sections that adopt fully-associative structures, we gradually decrease the associativity until the overall far-memory performance (including miss latency and hit overhead) starts to increase. By default, a 2-way set-associative cache uses a direct-mapped cache as its next candidate, while a fully associative cache uses a 16-way set-associative cache as its next option.

3.4 Section Size

Previous far-memory systems [3, 4, 7] have found that the performance of a far-memory system can be significantly impacted by the size of the local cache. However, in contrast to these earlier systems that focused solely on the total cache size and its effect on application performance, we take a more fine-grained approach by considering the impact of each cache section's size. This is important because different objects and their access patterns may be affected differently by the amount of local cache.

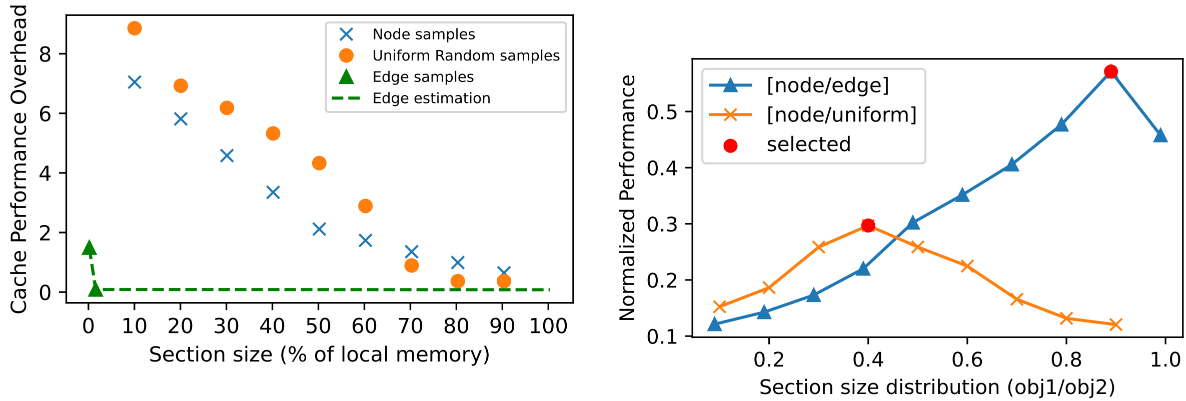
However, it is difficult to determine the exact relationship between cache size and performance, we employ sampling and profiling techniques to determine the optimal section

```
1  edges, nodes, R = malloc()
2  void traverse_graph(struct edge *edges) {
3    for (int i = 0; i < num_edges; i++)
4      update_node(edges[i], edges[i].from, edges[i].to);
5      // edges[i].from and edges[i].to point to nodes
6  }
7
8  void third_access(struct third *R) {
9    for (int i = 0; i < num_edges; i++) {
10     foo(R[rand1(i)], nodes[rand2(i)]);
11   }
12 }
13
14 traverse_graph(edges)
15 third_access(R)
```

Figure 3.7. **(Augmented) Code Example of Graph Traversal.**

capacity. Initially, we sample several different sizes for each cache section, and during each sampled run, we profile the cache overhead, which includes the total miss latency and hit overhead. For sections that correspond to sequential accesses, only a small amount of memory is required to accommodate enough preloaded data and hide the network latency. In general, the cache overhead decreases linearly as we increase the memory size to extend the prefetch pipeline (i.e., local slots for requested data). The overhead stops decreasing when the capacity is large enough to fulfill the required prefetch distance. Based on this knowledge, we only perform coarse-grained sampling for this group of sections and infer the optimal size.

For other sections, we collect denser sampling points for each cache (e.g., from 10% to 90% of the total required memory for the section). With the profiling results and the estimated cache lifetime obtained from program analysis, we formulate the configuration problem as an integer linear programming (ILP) problem. The objective of the ILP is to minimize the total cache overhead (weighted by the function execution time), while ensuring that the aggregated section sizes do not exceed the local memory limit at any time. To better demonstrate this effect with different access patterns, we augment the original graph traversal program with another array *R* that is accessed in a random manner, as shown in Figure 3.7.



(a) Sampled cache overhead with different section sizes. The performance of section for `Node` in function `traverse_graph` is omitted for clarity.

(b) Section size selection. The y-axis shows function-specific normalized performance.

Figure 3.8. **Illustration of the cache-capacity-performance relationship (left) and Mira's decisions on the section size (right) of the augmented graph traversal program.**

The impact of section capacity on the cache performance is depicted in Figure 3.8a. As the plot suggests, an extremely small section size will suffice to achieve the optimal performance for sequential accesses (edge nodes). On the other hand, for the `Node` and `R` arrays, the relationship between section capacity and cache overhead is obscure: `Node` overhead drops rapidly from 30% to 50% and `R` overhead decreases significantly from 50% to 70%. These different capacity-performance relationships necessitate different configurations in response to varying levels of local memory pressure.

Figure 3.8b shows the normalized performance of two functions respectively if assigning different section sizes when the local memory ratio is 20%. Since there is no overlap between the lifetime of `Node` and `R`, Mira can partition these two sets of caches, i.e., node/edge and node/uniform separately, and exploit full local memory in each case. Mira's solution generates the optimal memory distribution solution for both functions. As anticipated, the optimal approach is to allocate the majority of the memory to the `Node` array, which is accessed randomly. The optimal ratio between the `Node` section and the `R` section also aligns with their relative performance obtained through sampling.

In Figure 3.8b, the normalized performance of two functions is shown when different section sizes are assigned, assuming a local memory ratio of 20%. Since the lifetimes of node

array and R array do not overlap, Mira can partition the caches separately for each function using the whole local memory. As the plot shows, Mira's solution generates the optimal memory distribution for both functions. As expected, the majority of the memory is allocated to node objects in the first function. The optimal ratio between the Node section and the R section aligns with their relative performance obtained through sampling, demonstrating the effectiveness of Mira's approach in identifying promising cache configurations.

Chapter 4

Transparent Disaggregation

In this chapter, we introduce the integration of our runtime cache sections through a compile-time transformation process known as ”disaggregation” of user programs. The overall process consists of a `conversion` and a `lowering` pass. In Figure 4.1, we provide an example that illustrates how the Mira compiler converts conventional operations involving selected objects into remote operations and subsequently lowers these remote operations to low-level dialects. In the example, the transformed operations, after being converted to far-memory accesses, are represented by high-level descriptive semantics (`rmem.load`). This abstraction level simplifies the analysis and optimization process, as the resulting interface closely resembles the original one. From this point, we can progressively lower these operations to conventional dialects (`normal.load` for memory read) or to more primitive remote operations that describe low-level functionalities (`rmem.deref` for resolving remote address translation).

4.1 Convert to Far-Memory Accesses

While accesses to data managed by our generic swap section do not require any modification, we use the compiler to instrument additional operations for loading from other cache sections. Given the filtered set of large objects (as described in Section 3.1), Mira converts their allocation sites to far-memory allocations, whose resulting pointers are of type `remote memref` (defined in our MLIR dialect named `rmem`). This remoteability shall be propagated

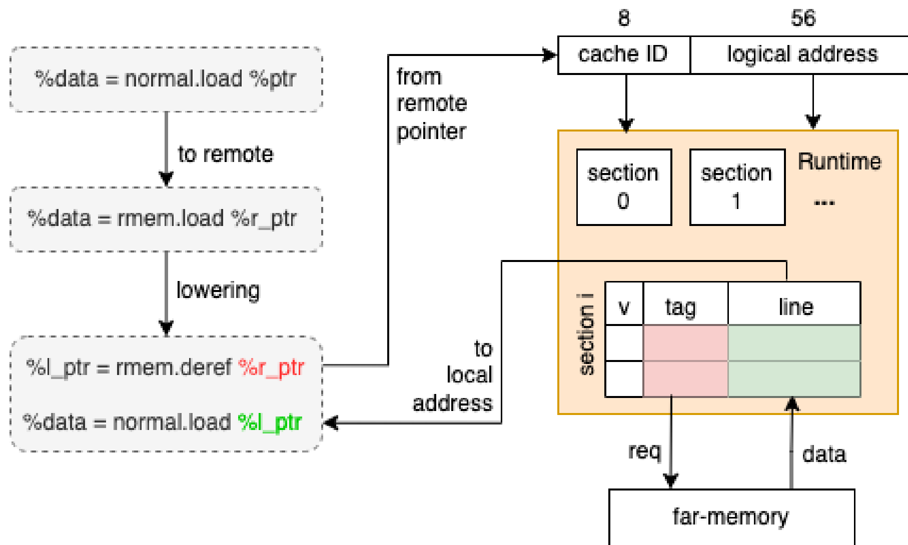


Figure 4.1. **Illustration of Mira's disaggregation flow.**

through def-use chains. For example, operations that only manipulate the address will pass on the remoteability to its result if a remote memref is its operand. On the other hand, operations that load the data will deprive this property, meaning the resulting data is available in the local memory. Operations with no result will be the end of the current propagation chain. Mira will transform operations that are involved with remote memref types into instructions defined in `rmem`, which we refer to as remote operations. Figure 4.2 shows an example of this process, in which line 12 is obtaining the address of the target element by inter(first index) and intra(second index) offsetting the source structure, whereas line 13 performs remote memory read, extracting one layer of remoteability leaving inner types intact.

4.2 Lowering Remote Operations

The conversion result only produces IR that contains high-level descriptive semantics, our compiler will eventually lower these operations to more basic instructions that actually perform far-memory accesses, either by interacting with our runtime system or sending RDMA requests directly. In this section, we describe how Mira lowers a remote pointer dereference and the optimization it performs.

Initially, the value of a rmem pointer represents its far-memory address in the remote

```
1  // Original IR (simplified)
2  %dat: struct.A {payload: int, next: ptr<A>} = malloc()
3
4  %2nd_next = normal.geteleptr %dat[1, 1] -> ptr<ptr<A>>
5  %target = normal.load %2nd_next -> ptr<A>
6  %paddr = normal.geteleptr %target[0, 0] -> ptr<int>
7  %payload = normal.load %paddr -> int
8
9  // After conversion and propagation (simplified)
10 %r_dat: struct.rA {int, rmref<rA>} = remotable.alloc()
11
12 %2nd_next = rmem.geteleptr %r_dat[1, 1] -> rmref<rmref<rA>>
13 %target = rmem.load %2nd_next -> rmref<rA>
14 %paddr = rmem.geteleptr %target[0, 0] -> rmref<int>
15 %payload = rmem.load %paddr -> int
```

Figure 4.2. **Example of propagating remotability**

memory space. For each `rmem.load/store`, Mira instruments API call to obtain the local address that corresponds to the given remote pointer. The translation routine involves checking the presence of data by mapping the far-memory address to the block index inside the corresponding section and examining its tag and valid bits. The first 8-bit of a remote address indicates its current section id and the runtime will use the corresponding translation function to perform the mapping. If it is not cached, Mira retrieves the data from far memory and places it in the designated slot. With the presence of objects in the cache, the runtime will return the virtual address that is meaningful to the local node MMU for actual accesses.

The naive approach is to ask the compiler to inject this process before each far-memory access and obtain the real local address first, followed by conventional memory load or store. However, this approach would introduce significant overhead to far-memory access as the expensive lookup-and-access step would need to be performed each time. To address this issue, we propose a compiler approach that leverages static analysis of access locality. The idea is that if we can determine the presence of a cache line in the local section based on previous requests, any subsequent accesses to the same cache line can be performed directly by offsetting the already resolved address. This allows us to completely avoid the overhead or reduce it to a

single address calculation instruction. This technique can be applied in various scenarios, such as a loop that sequentially accesses a large array or accesses multiple fields of a remote structure.

Note that the proposed optimization requires the rigid assumption on the presence of the target cache line in case there are multiple preceding far-memory dereferences happening in the same section and are not likely on the same line. Since our cache configuration satisfies most of the cases (as described in Section 3.3), we can apply the optimization extensively. Moreover, in cases when Mira sees excessive requests to the same cache section that might result in unsafe evictions, the compiler can use the ”lock” mechanism exposed by the runtime to mark a cache line as `unevictable` until the lifetime of an optimized dereferenced address ends.

Also note that the proposed optimization relies on the assumption that the target cache line is present, especially when there are multiple preceding far-memory dereferences occurring in the same section and not likely on the same line. In cases where Mira detects excessive requests to the same cache section that could potentially lead to unsafe evictions, the compiler can utilize the ”lock” mechanism provided by the runtime to designate a cache line as ”unevictable” until the dereferenced address's lifetime ends. Thanks to our co-design approach that configures the runtime setting according to program behaviors, we can still apply the elimination of redundant dereference extensively.

The efficiency issue is common in other works that introduce new user-space runtimes. For instance, AIFM [7] requires pointer dereferencing for each remote data item, even when those items are programmed to be accessed together, as each element is managed by the runtime in isolation, resulting in the need for individual dereference operations. In contrast, Mira attempts to determine whether two objects reside on the same cache line. If they do, resolving a remote address can eliminate the need for the entire dereference process for subsequent accesses. Additionally, AIFM relies on user-specified lifetime protection to ensure the validity of already dereferenced pointers. This approach introduces overhead for entering and leaving these scopes and places an additional burden on programmers. On the other hand, Mira designs the runtime structure based on estimated conflicts and deduces the reusability of cache lines at compile time.

This approach adds no overhead to ensure pointer lifetime while providing transparency to the programmer.

Chapter 5

Program Optimizations

Apart from transparently adopting far-memory in a user application, our compiler performs code optimization for far-memory accesses as discussed below.

5.1 Adaptive Prefetching

Prefetching is a commonly used technique to mitigate data movement delays. Previous systems [19, 44] employ heuristics based on access history to determine which data to prefetch. In contrast, we take a different approach by leveraging the compiler to identify data that is likely to be accessed in the near future. The compiler inserts prefetching operations at a program location estimated to be one network round trip earlier than the actual access, effectively avoiding blocking.

When dealing with memory operations within nested loops, an additional step is required to determine the level at which prefetching should be performed. This decision depends on the loop's body interval and the data being accessed, which can vary across different levels of the loop. In this section, we will explain the analysis process involved to estimate the accessed memory range, using the example depicted in Figure 5.1. The code example performs matrix multiplication on `%0` and `%1`, and stores the result in `%2`. The algorithm has been pre-optimized with loop tiling. The `memref` type is similar to a pointer, but it also provides information about the memory layout of the underlying resource.

```
1  func.func private @foo(f32,
2    vector<8xf32>,
3    memref<1xvector<8xf32>>)
4
5  %0 = rmem.alloc_memref: rmref<memref<256x1x1024xf32>>
6  %1 = rmem.alloc_memref: rmref<memref<1024x50264xf32>>
7  %2 = rmem.alloc_memref: rmref<memref<256x1x50264xf32>>
8
9  loop1: affine.for %arg0 = 0 to 256 {
10   loop2: affine.for %arg1 = 0 to 50264 step 8 {
11     loop3: affine.for %arg2 = 0 to 1024 step 8 {
12       %alloca = normal.alloca_memref() {alignment=16} :
13         memref<1xvector<8xf32>>
14       %4 = rmem.vec.load %2[%arg0, 0, %arg1]
15       affine.store %4 -> %alloca[0]
16       loop4: affine.for %arg3 = 0 to 8 {
17         %6 = arith.addi %arg2, %arg3 : index
18         %7 = rmem.affine.load %0[%arg0, 0, %6]
19         %9 = rmem.vec.load %1[%6, %arg1]
20         call @foo(%7, %9, %alloca)
21       }
22       %5 = affine.load %alloca[0]
23       rmem.vec.store %5 -> %2[%arg0, 0, %arg1]
24     }
25   }
26 }
```

Figure 5.1. Code example of accesses in nested loops. *The given MLIR is simplified for readability.*

For each remote access, we generate a symbolic representation of the involved memory region at the innermost loop that encloses it. This representation consists of three components: a remote object as the base address, an expression to calculate the offset given all induction variables of enclosing loops, and the number of objects being accessed by this operation. We use the access at line 18 in Figure 5.1 as an example. The starting address, %0, is the object allocated at line 5. Its memory layout indicates a three-dimension array, and sequentially indexing through each of its dimensions results in strided accesses with step sizes $\{1024, 1024, 1\}$ respectively. Thus, loading an element at `%0[%arg0,0,%6]` corresponds to the formula below:

$$mem_{L18} = \{\%0, \%arg0 * (1024) + (\%arg2 + \%arg3), 1\} \tag{5.1}$$

where $\%arg0, \%arg2, \%arg3$ stands for induction variables of loop1, loop3 and loop4 respectively.

Then, for each layer of the loop, we transform the above representation (Equation (5.1)) by peeling out induction variables belonging to loops nested within the current level one at a time. This process is straightforward: we iterate through the range of an induction variable, substitute it into the formula, and calculate the new offset symbolically. If the gaps between any pair of new offsets are smaller or equal to the access size, we merge them to form a contiguous accessed region. Using the same example, the representation for the memory region touched by the operation at line 18 with respect to loop3 will exclude $\%arg3$ and aggregate the resulting 8 small pieces into an integral region, leading to the final representation that matches line 11 in Figure 5.2.

```
1  access_mem = {
2  // range = { base address , offset expr , access size }
3  loop1 = ...
4  loop2 = ...
5  loop3 = [
6    {%0, <(%arg0,$arg1,%arg2) -> (%arg0 * 1024 + %arg2)>, 8},
7    {%1, <(%arg0,$arg1,%arg2) -> (%arg2 * 50264 + $arg1)>, 8},
8    ...
9    {%1, <(%arg0,$arg1,%arg2)
10     -> ((%arg2+7) * 50264 + %arg1)>, 8},
11   {%2, <(%arg0,$arg1,%arg2) -> (%arg0 * 50264 + $arg1)>, 8}
12 ]
13 loop4 = [
14   {%0, <(%arg0,$arg1,%arg2,%arg3)
15     -> (%arg0 * 1024 + (%arg2+%arg3))>, 1},
16   {%1, <(%arg0,$arg1,%arg2,%arg3)
17     -> ((%arg2+%arg3) * 50264 + %arg1)>, 8}
18 ]
19 }
```

Figure 5.2. **Mira's analyze results for touched memory regions in each loop layer.**

After we obtain the estimated memory access at each loop layer, we proceed to decide the appropriate location for prefetch. The principle that guides this decision is two-fold: 1) the total access of an object at a specific layer should not exceed the network's capacity, and 2) loops

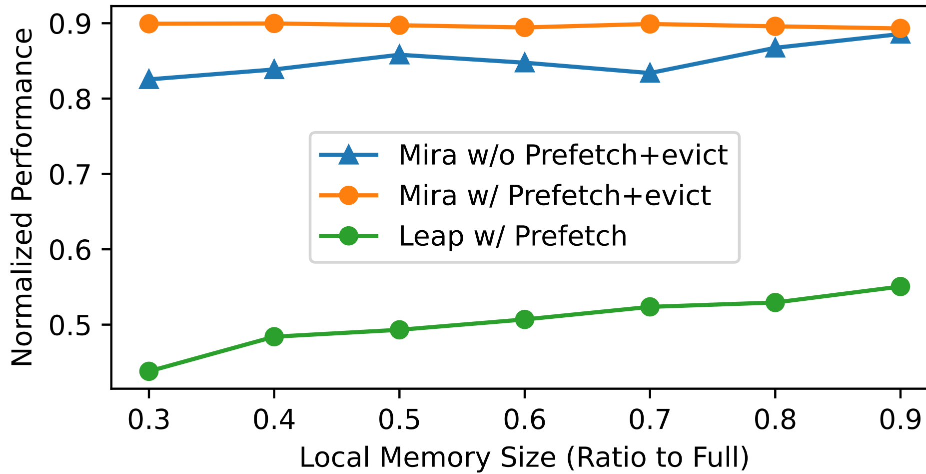


Figure 5.3. **Effect of prefetch and eviction hint on the original graph traversing program.**

whose induction variables are insignificant to the offset expression should be ignored. The first principle can facilitate an efficient prefetch pipeline. The second principle is to avoid redundant prefetch operations. For access operations whose memory ranges cannot be determined at compile-time, we select the immediately enclosing loop as the default prefetch site.

5.2 Eviction hints

With static analysis, Mira can determine the last access to a far-memory object within a code block in some cases. Leveraging this information, when all references to a cache line have ended their lifetime, the Mira compiler will insert an asynchronous flushing operation and mark the line as evictable. Future insertions can first check which existing lines are marked evictable and benefit from proactive movement beforehand. Without this optimization, all eviction events are blocking and delay the critical path since they are passively triggered by new accesses.

Figure 5.3 demonstrates the benefit of adding prefetching and eviction hints in Mira when running the graph-traversal example (Figure 3.1). In addition, we assess the performance on Leap [19], which implements prefetching based on majority history. However, Leap is designed to capture global access patterns and is not effective in prefetching for interleaved access patterns like the one demonstrated in this example, where each loop step involves both sequential and

indirect accesses. Furthermore, Leap employs the standard Linux global eviction policy and does not take advantage of any program hints.

5.3 Selective Transmission

A major drawback of swap-based systems is their fixed and coarse far-memory access granularity. While recent programming models like AIFM [7] allow programmers to specify precise data structures for movement between local and remote memory, there is a risk of suboptimal decisions leading to unnecessary data fetching. For example, designating a large data structure as a remoteable object may result in fetching the entire structure from remote memory even if only a few fields are accessed.

To address this issue, we propose a compiler-based solution. Our approach involves utilizing program analysis to identify the specific portions of a data structure accessed within each program scope, such as a function. Subsequently, we register handlers at the remote side and generate code that fetches or prefetches only these identified portions, avoiding the unnecessary transfer of unused data.

5.4 Batching Requests

For the majority of networks and interconnects, a single large communication event, such as a message containing multiple scatter-gathered data pieces, is more efficient than multiple smaller communication requests. Therefore, we aim to leverage this characteristic by altering program behaviors. In particular, if our analysis identifies multiple data items that are accessed at different locations, we modify the original code so that their network messages can be bound together. For instance, if we discover two arrays that are sequentially accessed by two adjacent loops without data dependency, we merge the loops and batch access the two arrays. We also apply conventional loop fusion beforehand to disclose more batching opportunities.

5.5 Data Communication Methods

Communication between the local node and the far-memory node, either over a network or a local bus/interconnect, is a crucial aspect of far-memory systems. Several previous studies have examined the advantages and potential applications of one-sided communication, where data is directly read/written from/to far memory, versus two-sided communication, where data is sent as messages and the far-memory node copies the messages to their final destinations [9, 45]. These works manually design the communication methods for a specific application domain.

Our selection of the appropriate communication method for each cache section is based on its access pattern. When our program analysis reveals that a section's access pattern involves reading/writing the entire data structure, we employ one-sided communication to fetch/write the entire structure in a single operation. On the other hand, if a section only accesses partial data structure, such as one or two of its fields, we opt for two-sided communication to transfer only the required partial structure. This approach helps avoid read/write amplification. To achieve this, we insert code at the compile time to either prepare a message by copying the partially accessed data structure or ask the remote memory node to assemble a reply with chosen fields.

5.6 Function Offloading

Some far memory nodes possess computational capabilities that can execute application code [7, 28], enabling offloaded computation to access data in far memory locally without network transfers. To take advantage of this feature, previous studies require programmers to determine which computation should be offloaded to far memory nodes, and in some cases, even rewrite the offloaded computation. On the contrary, Mira automatically and transparently determine and offloads computation to far memory with the following policy.

To reduce the program-analysis complexity, we only consider program functions as the unit of offloading. We first identify functions that do not have any shared writable data with other (local) functions as offloading candidates. Shared writes are hard to support in

today's far-memory environment that does not provide coherence between local and far-memory nodes. In the future, if hardware-provided coherence between compute and memory nodes, such as CXL [22], becomes available, we may consider functions with shared writes as potential candidates for offloading.

From the set of candidate functions, we decide which ones to offload to far memory based on the amount of computation to be offloaded and the level of network communication required. Since far-memory nodes typically have lower computation power (such as a low-power ARM processor), we avoid offloading computation-intensive functions to far memory. Meanwhile, we aim to minimize network communication, and if all the data accessed by a function is already in remote memory, it is advantageous to place the function in far memory as only the transfer of function parameters and results is required.

Once the offloading targets are finalized, we insert code at the local side to flush relevant local caches, ensuring that the remote node has up-to-date data. We also register the function handler on the remote side, where memory operations are converted back to native ones. The Mira compiler then packages the function inputs, instruments RPC calls to the offloaded function and blocks for the result.

Chapter 6

Implementation

We implement Mira runtime with 12.1K lines of code in C++ for both the local node and far-memory node. Additionally, we implement Mira compiler on top of Multi-Level Intermediate Representation (MLIR) with 7.7K lines of code in C++. In the following sections, we will delve into some of the representative parts.

6.1 Far-Memory Abstraction in MLIR

We add two new MLIR dialects for far memory:

1. `remoteable`. This dialect introduces a new abstraction for data objects that belong to cache sections and for functions that can be offloaded to far memory. Figure 6.1 shows the allocation of a remote object and the definition of a remoteable function at line 6 and line 8&9.

2. `rmem`. This dialect defines the interaction with remoteable objects and functions. The operations we implemented have two major objectives: 1) to support conventional pointer/memref instructions when targeting remoteable types, and 2) to coordinate with our runtime APIs and maintain metadata for optimizations.

Figure 6.1 is a comprehensive code example that covers most representative remote operations for reference.

```
%SEdge = rmem.cache_section
  {#type = "direct", #line = 2M, ...}
%SNode = rmem.cache_section
  {#type = "full", #line = 128B, ...}

@_redges , @_rnodes = remotable.alloc (..)

func.func @trvs_graph_opt(
  %arg0: !rmem.rmref<struct<_r_edge>>){
  scf.for %i <- %0 to %num_edges step %elements_per_line {
    // prefetch %n_ahead elements ahead from far memory
    rmem.fetch %SEdge, %arg0 + %i + %n_ahead
    // wait for current requested data (at %i) to be in cache
    rmem.wait %SEdge, %arg0 + %i
    // get phyiscal address of line starting address
    %line = rmem.paddr %SEdge, %arg0 + %i

    scf.for %j = %0 to %elements_per_line {
      // directly load element with resolved cache line
      %addr1 = normal.geteleptr %line[%j]
      %1 = normal.load %addr1

      // also prefetch node elements
      %addr2 = normal.geteleptr %line[%j + %n_node_ahead]
      %2 = normal.load %addr2
      // node elements may be in cache already, fetch if not
      rmem.fetch_if_not_in_cache %SNode, %2 -> from
      rmem.fetch_if_not_in_cache %SNode, %2 -> to

      // wait for node elements to be in cache and access
      rmem.wait %SNode, %1 -> from
      %3 = rmem.paddr %SNode, %1 -> from
      rmem.wait %SNode, %1 -> to
      %4 = rmem.paddr %SNode, %1 -> to
      func.call @update_node (%1, %3, %4)
    }
    // flush used %i element for eviciton hint
    rmem.flush %SEdge, %i
  }
}
```

Figure 6.1. Optimized graph traversal example. We only show prefetching and eviction flush in this case.
The given MLIR is simplified for readability.

6.2 Convert To `rmem` Dialect

Once Mira identifies a data object to be remoteable (Section 3.1), we trace its allocation site and perform forward dataflow analysis (Section 4.1) to convert involved pointers/memrefs and operations to remote counterparts. The remoteability within types is also propagated recursively. Afterward, we perform a backward analysis to find all the functions where a `rmem` pointer is passed as a parameter. Note that the same function may be called with a native local pointer, we then create another version of the function definition with a different signature.

Since both the backward and forward analyses require analyzing the entire program, we endeavor to minimize their usage by caching the analysis results. These results encompass each function's references to remoteable objects. By doing so, later compiler optimizations can make use of these results without the need to repeat the expensive whole-program analysis.

6.3 Allocation with Far-Memory

We have implemented remote memory allocation, which involves using a local allocator and a remote allocator jointly, to request memory from remote address space. The remote allocator operates similarly to a low-level system allocator, such as the mmap function in Linux, and handles the actual memory allocation in far memory. The local allocator, on the other hand, receives the allocated far-memory addresses from the remote allocator and buffers them, similar to user-space allocator libraries like malloc in clib.

When an allocation is requested, the local allocator first checks if there is an available memory region within the buffered far-memory addresses. If there is, it returns that region to the allocation site. If not, it requests additional addresses from the remote allocator. Since the allocated addresses correspond to virtual memory addresses at a far-memory node, our RDMA-based network stack can utilize them for one-sided accesses.

6.4 More on Lowering Routines

All high-level descriptive remote operations we inserted need to be converted to basic dialects, which contain primitive functionalities such as function calls or arithmetic operations. MLIR provides convenient translations from thesis dialects to LLVM-IR, from which we can produce the final binary. Lowering for some operations will call runtime APIs while others will be converted to instructions directly for performance.

- `rmem.cache_section` create cache metadata as a global data structure. Most values configured for the cache, e.g. line size and section size, are compile-time constant values and will be optimized out.
- `remoteable.alloc` will call the generated `@remote_alloc` function directly. The overall allocation process is described in Section 6.3.
- `rmem.deref` serves as an interface for translating remote addresses into local ones, as described in Section 4.2. We transform it to other remote operations, a process known as partial lowering. Specifically, we replace it with `rmem.fetch_if_not_in_cache` followed by a `rmem.wait` and `rmem.paddr`. However, we do not expect any of deref operations to be a direct conversion target in the final lowering pass, as we would transform all of them into the above three instructions and reschedule the fetch instruction in the earlier prefetch optimization pass.
- `rmem.fetch_if_not_in_cache` and `rmem.fetch` will call generated function `@cache_request_cond` and `@cache_request_ncond` respectively. Both lowering results will map a remote address to a local slot within the section to receive the incoming data, with the former one checking the tag bit before making RDMA requests.
- `rmem.paddr` will be lowered to instructions directly to obtain the local address of desired remote address. Instructions in Figure 6.2 assume a direct mapped cache with a line size

of 4096 bytes.

```
1  // local memory starting address
2  llvm.mlir.global external @_rbuf : !llvm.ptr<i8>
3
4  // global section metadata
5  llvm.mlir.global external @s0 {
6    #local_base = 0,
7    #remote_base = 0,
8    #linesize = 4096,
9    #section_id = 0,
10   #num_blocks = 16
11 }
12
13 // %7 = rmem.paddr @s0, %raddr
14 %c12_i64 = arith.constant 12 : i64
15 %c4095_i64 = arith.constant 4095 : i64
16 %c15_i64 = arith.constant 15 : i64
17 // get line id
18 %0 = arith.shrsi %raddr, %c12_i64 : i64
19 %1 = arith.andi %0, %c15_i64 : i64
20 %2 = llvm.mlir.addressof @_rbuf : !llvm.ptr<ptr<i8>>
21 %3 = llvm.load %2 : !llvm.ptr<ptr<i8>>
22 // offset to line
23 %4 = arith.shli %1, %c12_i64 : i64
24 // offset within line
25 %5 = arith.addi %4, %raddr : i64
26 %6 = arith.andi %5, %c4095_i64 : i64
27 %7 = llvm.getelementptr %3[%6] -> !llvm.ptr<i8>
```

Figure 6.2. **Example of lowering `rmem.paddr` for a specific cache structure.**

6.5 Function Offloading

Mira generates one object file for each remoteable function and links them with the remote server's runtime. Mira reverts `rmem` operations in the function back to normal memory accesses and remote pointers into normal pointers, as the function will run on the node that contains the remoteable objects locally. On the local side, we implement the invocation of an offloaded function as an RPC call. To ensure that the function can see the up-to-date remoteable objects during its execution, we flush all sections accessed by this function to far memory.

6.6 Cache Section Runtime

Fully-associative cache.

We maintain a remote address to physical address map for full associative caches and also a list of idle physical cache lines. On top of physical space management, we implemented an approximation of LRU eviction using active and inactive lists [46]. Our runtime could take our compiler-inserted code to perform prefetching and cache line flushing for eviction hints.

Swap-based cache section.

Different from other sections that directly generate program statements for cache accesses, the swap cache transparently executes the original code via a system-level run-time swap system. Line size in the swap cache to be 4KB, align with OS page size. We build our user-space swapping system on top of Linux `userfaultfd` [47].

6.7 Multi-Threading Support

Analyzing, generating code for, and optimizing multi-threaded programs is challenging because it is difficult to infer the order in which threads access shared data. To prevent potential race conditions in such programs, we lock a cache line, increasing the reference counter and rendering it ”unevictable” if it is dereferenced by any thread until the end of its lifetime. This ensures that no conflicting accesses from other threads can take over that slot. It is worth noting that traditional thread synchronization mechanisms still function as expected with Mira since we will not make synchronization primitives remoteable and keep real data accesses only occurring at the local side so that they are protected by traditional synchronization mechanisms.

When multiple threads share a section, certain cache configurations, and code optimizations may become impractical. For instance, because we cannot determine the locality set across multiple threads statically, we are no longer able to determine whether to use a directly mapped or set-associative structure by estimating the potential level of conflict. By default, we use a full-associative structure for all shared sections. Nevertheless, we can still optimize performance

by utilizing each thread's access pattern, such as by employing techniques like prefetching.

In order to circumvent the numerous overheads associated with multiple threads sharing a cache section, we undertake data ownership analysis. By determining that an object is exclusively owned by one thread or is read-only shared by multiple threads within the lifetime of a cache section, we can isolate or duplicate the section respectively for each thread to create a lock-free environment.

Chapter 7

Evaluation

We evaluated Mira on a Cloudlab cluster of 8 C6220 servers, each equipped with two 8-core Intel Xeon E5-2560 CPUs (2.80 GHz), 64GB RAM, and a 50 Gbps Mellanox FDR-CX3 NIC with 50G Infiniband network.

7.1 Settings

Applications

Three industrial-quality applications are selected to assess the scalability and the effectiveness of proposed techniques, including *DataFrame*, *MCF*, and *GPT-2 inference*. DataFrame [17] is a data analytics framework written in 24.3K LOC C++. The Dataframe system provides a set of data analytic operations, such as filtering, grouping, etc., on a data structure called DataFrame, a collection of named columns. When working with large data sets, DataFrame can be both compute and memory intensive, making it a good fit for far-memory environments. We run New York City taxi trip analysis workload [48] on top of this framework.

GPT-2 [18, 49] is a transformer-based [50] large language machine-learning model with 100M $\sim$ 1.5B parameters. We perform GPT-2 inference on ONNX [51], an open AI ecosystem that is compatible with MLIR [52]. The MLIR representation of GPT-2 inference on ONNX has more than 36K lines of code. We run this inference on sequences of 256-token length with a batch size of 64 in a CPU-based far memory environment. Both industry and academia have adopted the use of CPU to perform large machine learning model inference [53, 54, 55], as GPU

is not always available (e.g., in serverless computing services). A common technique used by these inference tasks is to use more memory (e.g., to cache certain computed values) for better inference latency [56], making it a good fit for far memory.

MCF [16] is a benchmark from the SPEC 2006 benchmark suites [57]. It is derived from a program used for single-depot vehicle scheduling in public transportation and performs graph-based computation. It is written in C and contains 1.8K LOC. Even though MCF is a smaller application than DataFrame and GPT-2 inference, it is representative of graph-processing applications that are common in data centers and can benefit from far memory.

Baselines

We compare Mira to three systems: AIFM [7], Fastswap [3], and Leap [19]. AIFM is a far-memory system that introduces a new programming model. We use AIFM's DataFrame implementation for DataFrame and its array library for MCF. Fastswap is a Linux-based optimized swap system for far memory. Leap is a Linux-based swap system that performs majority-based prefetching.

7.2 End-to-End Performance

DataFrame

Figure 7.1 provides an overview of the DataFrame performance of Mira, AIFM, Fastswap, and Leap at different local memory sizes. Mira exhibits superior performance compared to Fastswap and Leap due to its capacity to segregate and customize cache sections, such as implementing precise prefetching for each section, adopting appropriate cache line sizes, and so forth. Without cache segregation, Fastswap and Leap's swap-based global optimizations do not function effectively for each distinct program behavior. Although Leap performs majority-based prefetching, its performance is inferior to that of Fastswap, mainly owing to Fastswap's more efficient data-path implementation in Linux. AIFM experiences a significant runtime overhead in pointer dereferencing since it must resolve every access of a remoteable pointer. This overhead

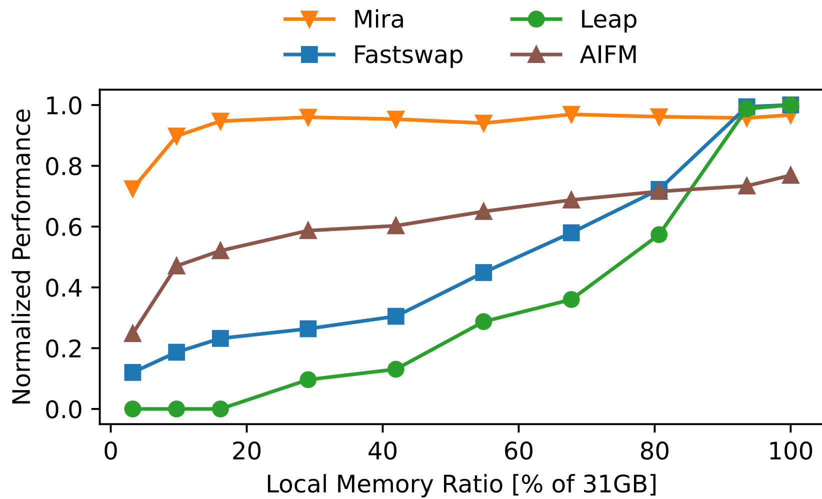


Figure 7.1. DataFrame Performance. *In this comparison, we do not consider function offloading.*

is why, even at 100% local memory, AIFM is significantly slower than other systems. On the contrary, Mira converts remote accesses in the same cache line to native memory instructions, thus amortizing the overhead of the first dereferences in each line.

GPT-2 Inference

Figure 7.2 depicts the overall GPT-2 inference performance of Mira, Fastswap, and Leap. AIFM is not evaluated for this application since it presently does not support any matrix structures or machine-learning operations. By leveraging domain knowledge and investing user effort into crafting these interfaces, we anticipate that AIFM will demonstrate performance on par with the native execution. Mira's performance remains steady even when the local memory size reduces to a mere 4.5% of the full memory. DNN model inference, such as GPT-2, has a layer-by-layer computation pattern, where the data used in one layer (such as weight matrix or input to the layer) is not required for the subsequent layers. Our program and profiling analyses accurately capture this pattern by reusing section space for matrices in distinct layers, ending their lifetime when their corresponding layers finish, performing batched access of data used in each layer, and generating precise prefetching and eviction hints. Consequently, the majority of remote access overhead can be concealed behind performance-critical paths, and even a small quantity of local memory is adequate for saturating computation throughput. In contrast,

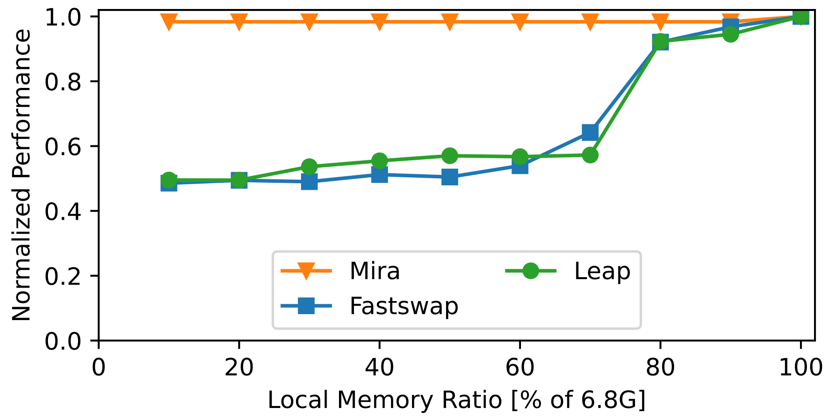


Figure 7.2. GPT-2 Generation Performance.

Fastswap and Leap encounter a significant decrease in performance as the local memory size decreases because, without knowledge of the program's behavior, they are unable to retrieve the specific data set required for computation. As a result, they end up utilizing considerable local memory to cache data that is no longer needed or required in the distant future.

MCF

Figure 7.3 displays the overall MCF performance. Since MCF is a graph-like application, its memory accesses rely heavily on pointer values as well as program control flows. Consequently, it is the least accommodating application for program analysis tools among the three. Nonetheless, Mira can make appropriate cache configuration and give software prefetching/eviction hints through our co-design approach. Mira utilizes a generic swap section for the primary object (whose access pattern is largely pointer indirection) when the local memory exceeds 70% of the required size. When the local memory is limited, Mira identifies performance overheads in the swap-based cache sections through profiling and optimizes them to use a set associative cache. Mira prefetches data accurately by analyzing the address expression, similar to our graph example.

In contrast, Fastswap and Leap are swap-based regardless of local memory size and program behavior. As a result, Mira outperforms them when the local space is relatively low. Since MCF adopts a space-efficient representation of the static graph, i.e., storing edges

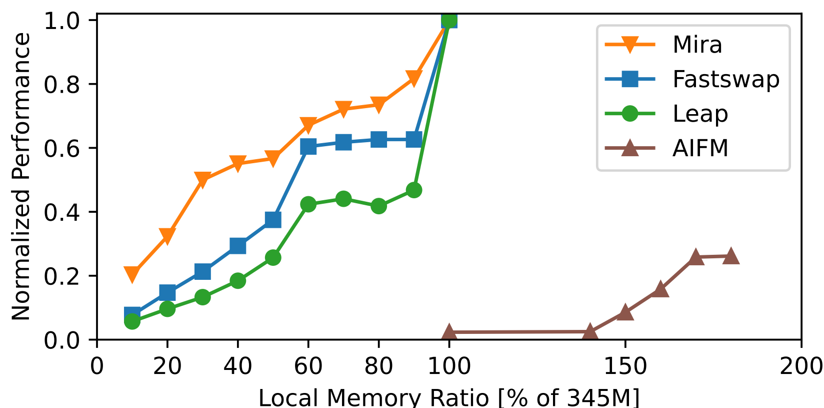


Figure 7.3. **MCF Performance.**

continuously in a single array, we employ AIFM's array library to implement a similar layout. However, we fail to execute the program even when the local memory is slightly lower than what is required. When given sufficient memory, AIFM's performance is significantly worse than the other systems, and its performance only improves to 26% when the size exceeds the full memory size by 80%. The reason for this is that AIFM requires a significant amount of metadata for their remoteable pointers, which decreases the amount of local memory available for actual data. Since the access pattern in this graph workload is largely random, the impact of space overhead is most pronounced among the three applications. Mira, on the other hand, has considerably smaller space overhead. Rather than storing various information such as lifetime with each remote pointer, Mira employs such information during compilation. Moreover, we attach the metadata to each cache line that can contain multiple objects. Besides space overhead, AIFM incurs costly pointer dereferencing for every element in an array, while Mira avoids this with the dereference optimization introduced in Section 4.2.

7.3 Runtime Overhead

To illustrate the efficiency of Mira, we assess the performance overhead and metadata overhead at full local memory while running the following applications: MCF, DataFrame, GPT-2, the graph-traversal example, and a trivial loop that sums over a big array. We also collect the number for AIFM except for GPT-2 inference. Figure 7.4 shows the overhead compared to

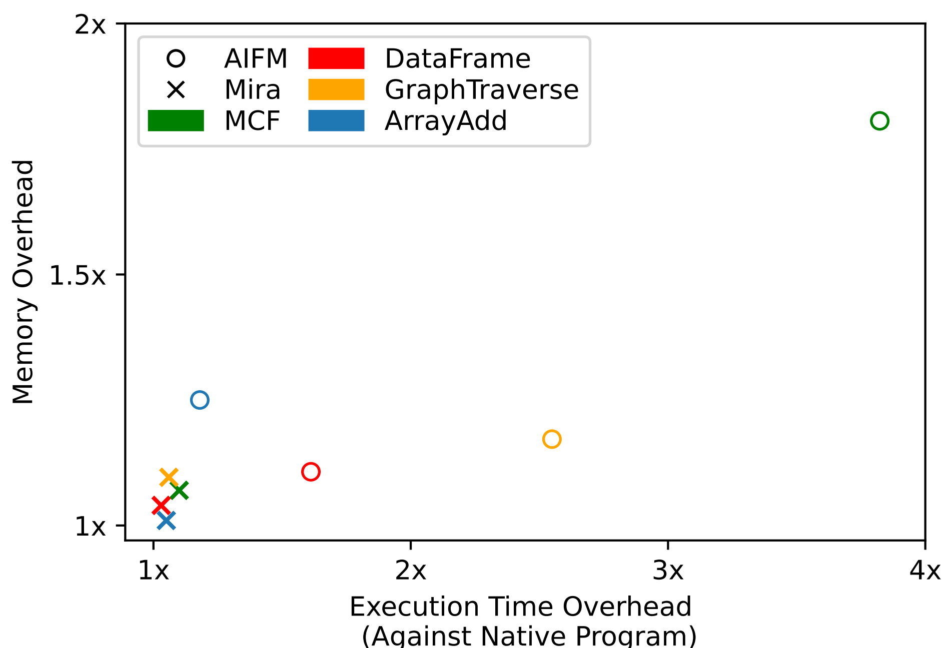


Figure 7.4. **Runtime Overhead Comparison. The memory is set to the required size of native program.**

executing the unchanged program as the baseline. For all workloads, Mira exhibits near-native performance and minimum space overhead.

7.4 Compilation Time

Using the insights gained from profiling, we are able to narrow down the program analysis required for MCF from 1.8K lines of code to just three functions with 300 lines of code in total. Similarly, for ONNX GPT-2 inference, we have reduced the number of allocation sites from 1000+ to 122. This reduction in scope has enabled Mira program analysis and compilation to run much faster even for large programs, such as GPT2 with 36K LOC, which can now be processed in just 3.93 seconds.

7.5 Performance Deep Dive

In order to identify the factors contributing to Mira's performance gains, we conducted an evaluation in which we add techniques incrementally, as depicted in Figure 7.5. The benefits of these techniques varied depending on the application and the amount of local memory

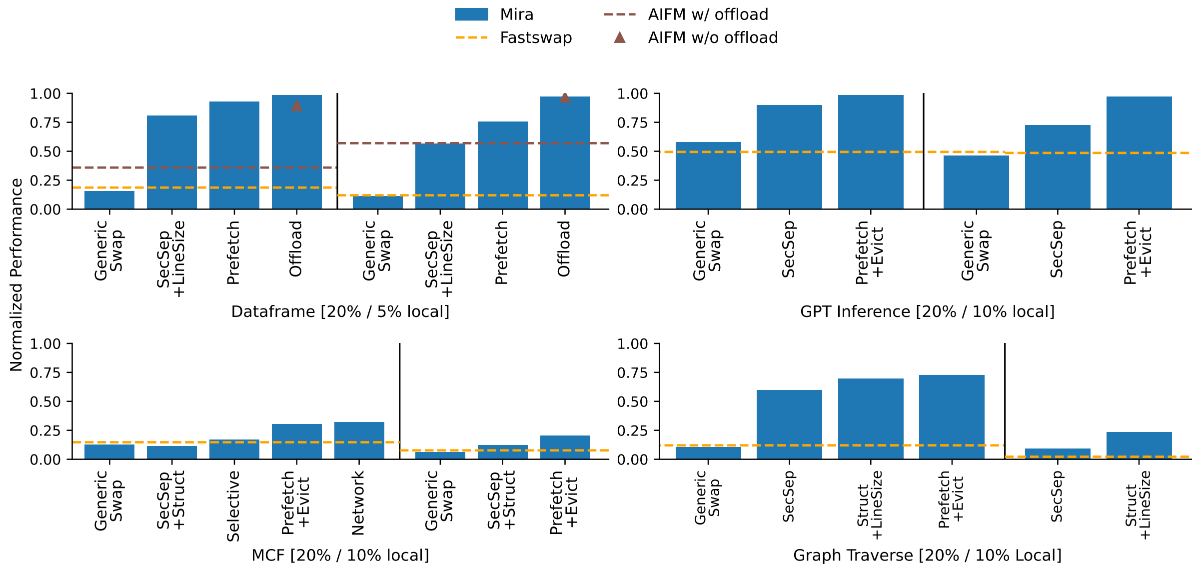


Figure 7.5. Mira Performance gain breakdown for all applications. Listed techniques are applied inclusively from left to right, with the generic swap-based section as the baseline.

available, and we have chosen several representative applications to illustrate the impact of these techniques. We only include AIFM in the Dataframe workload comparison since the reference implementation is provided in [7]. The same terminologies in Figure 3.2b are used in this plot. Below, we will explain the effect of each technique and its impact on the three applications.

Cache Section Separation

Applying cache separation results in a significant performance boost for all applications, except MCF, compared to using a generic swap-based section. This improvement is partly due to the customized configurations for each cache, which better cater to different program behaviors. Additionally, we leverage static lifetime analysis to optimize local memory utilization, instead of relying solely on runtime heuristics like replacement policies. Mira's ability to promptly release matrices used by one layer after computation finishes is particularly advantageous for machine-learning inference. This feature allows free spaces to be immediately reused by other layers, increasing the likelihood of satisfying a layer's working set. Previous DNN systems propose a similar memory management policy, but it requires manual adoption [58], while Mira can automatically generate the optimal scheme based on program analysis and profiling.

However, cache separation brings limited benefits for MCF, as its memory accesses are random and require a large space to accommodate the working set. Even a small reduction in available memory significantly degrades performance. Allocating a separate section for each behavior can prevent interference with others but also introduces more conflicts within each section. Currently, our policy does not always achieve a perfect trade-off between these two factors. For example, with a 20% local memory ratio, the performance of MCF slightly deteriorates, while a 10% ratio yields noticeable improvement. A future direction would be to employ performance counters, such as memory reuse distance and miss rate, to infer the effect of this separation and optimize our decision-making

Prefetching and Eviction Hints

Mira employs program analysis to uncover memory access information, including precise access sequences and the control flow graph. This information is then utilized to implement techniques such as accurate prefetching and proactive data eviction, further optimizing the memory usage of the system. As shown in Figure 7.5, while tuning cache parameters only generates marginal improvement, our software prefetching and eviction hints still help to improve MCF's performance by 48.1% and 67.9% when the memory ratio is at 0.1 and 0.2 respectively. The reason MCF benefits significantly from this optimization can be attributed to its fine-grained and pointer-chasing memory accesses. These types of accesses are difficult to optimize for history-based runtime systems, making software techniques more appealing. On the other hand, prefetching and eviction have a smaller impact on cache sections with large line sizes (such as those with multiple consecutive loads like GPT-2 inference), as the latency of blocking on a cache miss can be amortized by considerable accesses to the fetched line.

Function OffLoading

We only apply this technique for the DataFrame application as a proof of concept. As shown in Figure 7.5, the effect is more eminent when the local memory is low.

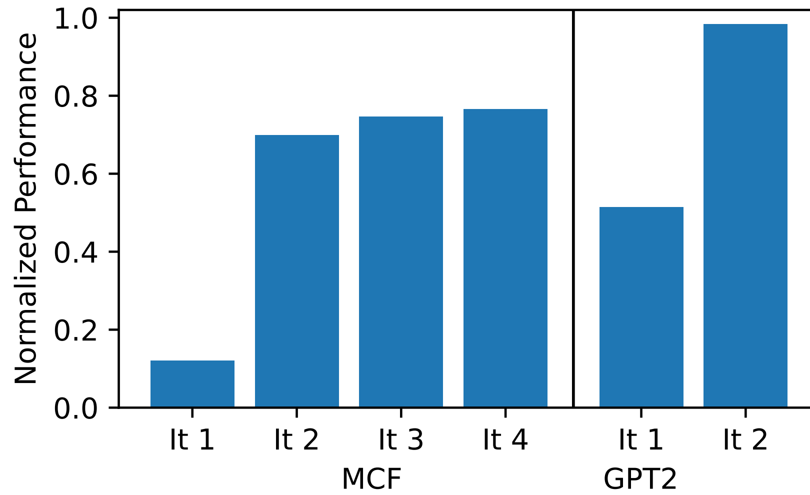


Figure 7.6. **Mira Iterative optimization of MCF and GPT2.**

Optimizations Convergence

The iterative profile-and-optimize process employed by Mira, as demonstrated in Figure 7.6, is evaluated using MCF and GPT-2 inference. In the initial iteration, MCF uses the generic swap cache globally. After the first execution, Mira identifies two large objects, one of which causes significant performance overhead. Mira analyzes this object and selects a set-associative section for the next iteration. After the next run, Mira detects read amplification and accordingly reduces the cache line size. MCF performance converged after 4 iterations.

In the case of GPT-2 inference, we identifies 122 large objects that could be placed in isolated sections right from the first iteration. The access patterns for these objects are highly predictable, and their lifetime can be clearly separated and is oblivious to the input. As a result, Mira achieves optimal performance within just two iterations.

Chapter 8

Conclusion

In this thesis, we have delved into the utilization of program behavior to design and optimize far-memory systems. As a result, we introduced Mira, a comprehensive framework that integrates program analysis, compiler optimizations, profiling systems, and runtime support. Through the combined power of static and dynamic program analysis techniques, we co-optimize the far-memory runtime with the program itself, enabling user applications to seamlessly harness the benefits of far-memory while minimizing the associated deployment burden and achieving superior performance.

This thesis, in full, is currently being prepared for submission for publication of the material, coauthors include Zhiyuan Guo and Yiying Zhang. The thesis author was one of the primary authors of this material.

Bibliography

[1] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[2] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. Shin, "Efficient Memory Disaggregation with Infiniswap," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, (Boston, MA), April 2017.

[3] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, (New York, NY), April 2020.

[4] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, (Carlsbad, CA), October 2018.

[5] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu, "Canvas: Isolated and Adaptive Swapping for Multi-Applications on Remote Memory," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), Apr. 2023.

[6] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, (New York, NY, USA), p. 79–92, Association for Computing Machinery, 2021.

[7] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-Performance, Application-Integrated Far Memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, (Banff, Canada), November 2020.

[8] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No Compromises: Distributed Transactions with Consistency, Availability, and Performance," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, (Monterey, California), 2015.

[9] S.-Y. Tsai, Y. Shan, , and Y. Zhang, “Disaggregating Persistent Memory and Controlling Them from Remote: An Exploration of Passive Disaggregated Key-Value Stores,” in *Proceedings of the 2020 USENIX Annual Technical Conference (ATC ’20)*, (Boston, MA, USA), July 2020.

[10] S. Ainsworth and T. M. Jones, “Software prefetching for indirect memory accesses: A microarchitectural perspective,” *ACM Trans. Comput. Syst.*, vol. 36, jun 2019.

[11] G. Chakrabarti and F. C. PathScale, “Structure layout optimizations in the open 64 compiler : Design , implementation and measurements,” 2008.

[12] T. Kistler and M. Franz, “Automated data-member layout of heap objects to improve memory-hierarchy performance,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 3, pp. 490–505, 2000.

[13] P. Li, H. Luo, C. Ding, Z. Hu, and H. Ye, “Code layout optimization for defensiveness and politeness in shared cache,” in *2014 43rd International Conference on Parallel Processing*, pp. 151–161, IEEE, 2014.

[14] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, “Bolt: a practical binary optimizer for data centers and beyond,” in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, IEEE, 2019.

[15] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, “MLIR: Scaling compiler infrastructure for domain specific computation,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’21)*, 2021.

[16] A. Löbel, “Combinatorial optimization Single-depot vehicle scheduling.” https://www.spec.org/cpu2006/Docs/429.mcf.html.

[17] “C++ DataFrame for statistical, Financial, and ML analysis..” https://github.com/hosseinmoein/DataFrame.

[18] “GPT-2: 1.5B release.” https://openai.com/research/gpt-2-1-5b-release.

[19] H. Al Maruf and M. Chowdhury, “Effectively Prefetching Remote Memory with Leap,” in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC ’20)*, 2020.

[20] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu, “Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, (Boston, MA), Apr. 2023.

[21] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, “Rethinking Software Runtimes for Disaggregated Memory,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*, (Virtual, USA), 2021.

[22] CXL Consortium. https://www.computeexpresslink.org/.

[23] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam, "Project PBerry: FPGA Acceleration for Remote Memory," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*, (Bertinoro, Italy), 2019.

[24] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct Access, High-Performance Memory Disaggregation with DirectCXL," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, (Carlsbad, CA), July 2022.

[25] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*, (Vancouver, BC Canada), March 2023.

[26] C. Branner-Augmon, N. Galstyan, S. Kumar, E. Amaro, A. Ousterhout, A. Panda, S. Ratnasamy, and S. Shenker, "3PO: Programmed Far-Memory Prefetching for Oblivious Applications," 2022.

[27] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast Remote Memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, (Seattle, WA), April 2014.

[28] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A Hardware-Software Co-Designed Disaggregated Memory System," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, (Lausanne, Switzerland), Mar. 2022.

[29] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proceedings of the 8th Annual Symposium on Cloud Computing (SOCC '17)*, (Santa Clara, CA, USA), September 2017.

[30] K. Wang, G. Xu, Z. Su, and Y. D. Liu, "GraphQ: Graph query processing with abstraction refinement – programmable and budget-aware analytical queries over very large graphs on a single PC," in *USENIX Annual Technical Conference (USENIX)*, pp. 387–401, 2015.

[31] G. Feng, H. Cao, X. Zhu, B. Yu, Y. Wang, Z. Ma, S. Chen, and W. Chen, "TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, (Carlsbad, CA), July 2022.

[32] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, (Nuremberg, Germany), 2009.

[33] C. Ding and K. Kennedy, “Improving effective bandwidth through compiler enhancement of global cache reuse,” *Journal of Parallel and Distributed Computing*, vol. 64, no. 1, pp. 108–134, 2004.

[34] X. Su, X. Liao, H. Jiang, C. Yang, and J. Xue, “Scp: Shared cache partitioning for high-performance gemm,” *ACM Trans. Archit. Code Optim.*, vol. 15, oct 2018.

[35] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, “Apt-get: Profile-guided timely software prefetching,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 747–764, 2022.

[36] E. Witchel and K. Asanovic, “The span cache: Software controlled tag checks and cache line size,” in *Workshop on Complexity-Effective Design, 28th ISCA*, 2001.

[37] P.-A. Tsai, N. Beckmann, and D. Sanchez, “Jenga: Software-defined cache hierarchies,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 652–665, 2017.

[38] J. Lee, C. Park, and S. Ha, “Memory access pattern analysis and stream cache design for multimedia applications,” in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pp. 22–27, 2003.

[39] M. Kandemir, I. Kadayif, and U. Sezer, “Exploiting scratch-pad memory using presburger formulas,” in *Proceedings of the 14th international symposium on Systems synthesis*, pp. 7–12, 2001.

[40] S. Udayakumaran and R. Barua, “Compiler-decided dynamic memory allocation for scratch-pad based embedded systems,” in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 276–286, 2003.

[41] A. E. Şuşu, “A vector-length agnostic compiler for the connex-s accelerator with scratchpad memory,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 6, pp. 1–30, 2020.

[42] D. Chen, T. Moseley, and D. X. Li, “AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications,” in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016.

[43] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O’Reilly, and S. Amarasinghe, “Autotuning Algorithmic Choice for Input Sensitivity,” *SIGPLAN Not.*, vol. 50, p. 379–390, jun 2015.

[44] P. Braun and H. Litz, “Understanding memory access patterns for prefetching,” in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, 2019.

[45] X. Wei, F. Lu, R. Chen, and H. Chen, "{KRCORE}: A microsecond-scale {RDMA} control plane for elastic computing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 121–136, 2022.

[46] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, (San Francisco, CA, USA), p. 439–450, Morgan Kaufmann Publishers Inc., 1994.

[47] L. K. Archieves, "Userfaultfd — The Linux Kernel documentation." https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.

[48] "NYC Taxi Trips - Exploratory Data Analysis." https://www.kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook.

[49] I. Solaiman, M. Brundage, J. Clark, A. Askell, A. Herbert-Voss, J. Wu, A. Radford, G. Krueger, J. W. Kim, S. Kreps, M. McCain, A. Newhouse, J. Blazakis, K. McGuffie, and J. Wang, "Release strategies and the social impacts of language models," 2019.

[50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[51] "Open Neural Network Exchange." https://onnx.ai/.

[52] "ONNX-MLIR." https://github.com/onnx/onnx-mlir.

[53] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, (Carlsbad, CA), July 2022.

[54] "Llama.cpp 30B runs with only 6GB of RAM now." https://news.ycombinator.com/item?id=35393284.

[55] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, J. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy, "Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications," 2018.

[56] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," 2022.

[57] "Standard Performance Evaluation Corporation 2006." https://www.spec.org/cpu2006/.

[58] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016.