Title

Mira: A Program-Behavior-Guided Far Memory System

Authors

Guo, Zhiyuan

He, Zijian

Zhang, Yiying

Publication Date

2023-10-23

DOI

10.1145/3600006.3613157

Mira: A Progam-Behavior-Guided Far Memory System

Zhiyuan Guo
z9guo@ucsd.edu
University of California San Diego
San Diego, California, USA

Zijian He
zih015@ucsd.edu
University of California San Diego
San Diego, California, USA

Yiying Zhang
yiying@ucsd.edu
University of California San Diego
San Diego, California, USA

Abstract

Far memory, where memory accesses are non-local, 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: transparently swap memory pages between local and far memory, and utilizing new programming models to explicitly move fine-grained data between local and far memory. The former requires no program changes but comes with performance penalty. The latter has potentially better performance but requires significant program changes.

We propose a new far-memory approach by automatically inferring program behavior and efficiently utilizing it to improve application performance. With this idea, we build *Mira*. Mira utilizes program analysis results, profiled execution information, and system environments together to guide code compilation and system configurations for far memory. Our evaluation shows that Mira outperforms prior swap-based and programming-model-based systems by up to 18 times.

ACM Reference Format:
Zhiyuan Guo, Zijian He, and Yiying Zhang. 2023. Mira: A Progam-Behavior-Guided Far Memory System. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3600006.3613157

1 Introduction

As memory becomes one of the most contended hardware resources in data centers and as more applications require huge memory to execute, a promising and popular approach is to allow applications to use memory beyond traditional main memory, such as unused memory on a remote server [25], disaggregated memory blades in a server or a rack [33, 46], and other forms of slower but cheaper memory [43, 57]. Some of these non-local memory has attached computation power (*e.g.*, an ARM processor) [31, 56]. In this paper, we call all of them *far memory*.

Because far memory is slower than local memory, existing systems have all utilized local memory as a cache for far memory, with two approaches. The first is to transparently swap memory pages between local and far memory [8, 9, 30, 58, 68]. These systems all suffer from the coarse granularity of a 4 KB page, which is often larger (2.3× to 31× [17]) than what is actually read/written by an application. Data amplification not only consumes extra network bandwidth but could also slow down overall application performance. The second far-memory approach is to use a new programming model or extend an existing one with new APIs for far-memory accesses [26, 31, 56, 65]. Through explicit and precise control of what to access in far memory, this approach reduces amplification but requires non-trivial application-programmer or library-writer effort.

These two approaches respectively perform optimizations dynamically by a run-time system and statically by programmers. The former is a completely transparent system-level approach that treats user programs as a black box, while the latter is a white-box approach that puts the responsibility of optimization on programmers. *Is it possible to overcome the drawbacks of these approaches, harness their benefits, and even surpass their best-case performance?*

Our answer lies in a *program-behavior-guided far-memory approach*, by exploring an unexplored layer in far-memory research: program-analysis tools and compilers. With compilers, we can automatically convert programs written for local memory to accessing far memory, optimize the transferred code for better performance, and do so without any programming burden. Program analysis can reveal information unknown to run-time systems or even programmers. For example, it can detect indirect memory accesses like `for (i=0; i< size; i++) B[A[i]]++;`. With this knowledge, a compiler can insert prefetching operations like `%1=(fetch A[i+distance])` and `fetch B[%1]` at `distance` elements ahead. In contrast, without program knowledge, a runtime-based far-memory system often exhibits amplification or prefetching of incorrect data based on history. While static program analysis and code optimization offer many benefits, a key limitation is their inability to incorporate run-time information, which may result in suboptimal decisions. As with prior solutions to static approaches' limitations [35, 51], we can leverage run-time profiling of applications and utilize profiling outcomes to steer program analysis and compilation for far-memory systems.

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613157

We leverage program analysis, compiling, and profiling together to automate and optimize far-memory accesses. These technologies have been extensively studied in a traditional server setting for optimizing the performance of CPU cache and main memory [19, 42, 48, 51]. However, cache for far memory is fundamentally different in one key aspect: *cache for far memory is DRAM-based and can be controlled by software.* This feature gives us a great opportunity to customize the cache for program behavior (which we acquire from program analysis and run-time profiling) and to generate and optimize code for far memory based on the customized cache configurations using a compiler. Together, they call for the *co-design of program analysis, compiler, profiling, and run-time cache systems for far memory.* This co-design opportunity also brings significant challenges: while traditional compiler optimizations target a fixed CPU cache architecture, we need to configure our cache based on our program analysis and profiling, and our compiler should generate and optimize code for far memory via this non-fixed cache.

To leverage opportunities and confront challenges, our core idea is to separate the local cache into spaces dedicated to and configured for different program behaviors. We observe that a program often exhibits several different memory access patterns with different objects or at different phases, and they benefit from different cache configurations. For example, sequential accesses fit a small directly mapped cache with a cache line size of multiple consecutive data elements, while accesses with good locality but large working sets fit a relatively large set-associative cache. With this observation, we propose to divide the local cache into different *cache sections*, each tailored to a distinct access pattern. Based on the analyzed and profiled behavior of one program scope for one object or multiple objects with the same behavior, we configure a cache section's size, cache structure (*e.g.*, set-/full- associative), cache line size, prefetching and eviction patterns, and communication method (*e.g.*, one-/two-sided RDMA). Our compiler then optimizes code in that scope to best fit the configuration. Section separation allows us to customize cache configurations for one access pattern at a time and to in turn optimize code for one cache configuration at a time. Additionally, we decompose a whole-program-whole-cache co-design problem into manageable per-access-pattern subproblems that we can more precisely solve.

With this core idea, we build *Mira*, a far-memory system that co-designs program analysis, compilation, a configurable cache layer, and run-time profiling. It follows an iterative approach shown in Figure 1. Initially, Mira profiles the application running on our generic swap layer to identify scopes for analysis. For these scopes and based on analysis and profiling results, Mira identifies objects to place in far memory, generates far-memory accessing code, and optimizes the code and cache configuration. Additionally, Mira identifies and compiles functions to offload to far memory with computation power, also based on program behavior. The next iteration uses the new configuration and code. If high overhead is detected, Mira performs another optimization iteration, until user-specified stopping criteria is met.

Apart from the co-design challenge with a configurable cache, Mira confronts two unique challenges in a far-memory environment: 1) inefficient implementation of far-memory pointers and their dereferences will largely hurt application performance; 2) larger program scopes and more objects need to be potentially analyzed, as far-memory accesses are slower and local cache is larger than CPU cache. For 1), we design a novel far-memory pointer dereferencing mechanism that is performance efficient and metadata-space efficient, by leveraging program behavior to turn as many dereferences into native memory loads as possible. For 2), we perform coarse-grained, cache-section-specific profiling to narrow down program scopes and objects to those with the highest potential gain from further optimization, and we analyze and optimize each of them while globally optimizing the partition of local cache space across them.

We implement Mira's static parts on top of MLIR [44], a Multi-Layer Intermediate Representation ecosystem that allows us to choose the proper abstraction levels to build our program analysis and compiler and to support a variety of front-end programs and back-end execution architectures. We build all run-time parts as user-level libraries. We evaluate Mira using micro-benchmarks and three real programs: MCF [10], DataFrame [34], and GPT-2 [50] inference [7]. We compare Mira with FastSwap [9], a kernel-level swap-based far-memory system, Leap [8], a run-time pre-fetching solution for swap-based far-memory system, and AIFM [56], a far-memory system with a new programming model. Our results show that Mira outperforms these prior swap-based and programming-model-based systems by up to 18 times.

Mira is available at https://github.com/WukLab/Mira.

2 Related Works

2.1 Existing Far-Memory Systems

Page-based far-memory swapping. A common way to build far-memory systems is via page-based memory swapping. InfiniSwap [30] is the first RDMA-based remote memory swap system. FastSwap [9] improves InfiniSwap's performance with better scheduling and polling mechanisms. Leap [8] prefetches memory pages to avoid remote-memory accesses in the critical path based on a process' majority access pattern. Canvas [68] and Hermit [54] are two recent works that improve Linux's swap system by enforcing better isolation mechanisms in a multi-application environment and by executing non-urgent but time-consuming tasks asynchronously. LegoOS [58] is a non-Linux based system that swaps 4 KB pages between a compute node's "extended cache" and disaggregated memory.

These swap-based systems all suffer from two common problems: 1) they are all 4 KB page based. Such coarse granularity could result in huge network bandwidth wastage and reduced application performance [17]; 2) they are all agnostic to program semantics. As we will show, program semantics are crucial in enabling a variety of optimizations.

3PO [15] is a recent system that uses an offline process to analyze memory accesses of *oblivious* applications, whose memory accesses are independent of program inputs. 3PO then uses the analysis results to perform prefetching. 3PO still performs prefetching in 4KB-page granularity. Moreover, it only works for completely oblivious applications.

Cache-line-based and other far-memory systems. New hardware like CXL [22] and research-based prototypes [18, 29] enable access to far memory in cache-line size and with much faster speed than today's network communication. Moreover, CXL allows CPU cache misses being directly served by a memory device connected to the CPU. Software systems on top of these hardware technologies can utilize the high speed and/or fine access granularity to improve far-memory performance [17, 46]. Unlike Mira, none of these existing software systems consider program semantics or configure local cache based on program behavior. Note that even though our implementation of Mira focuses on RDMA-based remote memory, our general designs apply to a broad definition of far memory, including CXL-based memory pools, local- or remote-node persistent memory, and slower storage layers, because Mira's optimizations can adapt to different far-memory accessing speeds and computation power.

New programming models. In addition to transparent approaches, another type of far-memory solution is introducing new far-memory-specific programming interfaces. FaRM [25, 26] and many other RDMA-based systems [31, 59, 65] use simplified or richer APIs for programmers to perform remote memory allocation, read, write, etc. AIFM [56] proposes a new programming model for far memory, including remotable pointers, dereferencing scope, eviction handler, etc. To avoid application programmers' burden, AIFM tries to confine far-memory-specific programming within libraries. A common limitation of these works is their burden on application or library developers, who can also make unoptimized decisions. Moreover, these works only optimize their added APIs or library calls and do not analyze other program behavior for further performance optimization opportunities. Finally, systems like AIFM incur high runtime overhead, as each far-memory pointer dereferencing requires the manipulation of fair amounts of metadata.

2.2 Non-Far-Memory Optimizations

Memory accesses in a traditional, non-far-memory environment have been highly optimized at various layers. However, as far as we know, there is no work that co-designs program analysis, compiler, and a configurable cache.

Compiler and system optimizations for CPU cache. A host of compiler-level and system-level solutions have been proposed to optimize applications' performance on CPU caches. They can be roughly categorized into three types. The first transfers programs and/or data layout to make memory accesses more cache friendly, *e.g.*, via data structure padding, peeling, field reordering, hot-cold code region separation, etc. [19, 42, 48]. The second allocates different CPU cache spaces to different parts of applications. For example, CPU cache coloring assigns different memory regions to different cache regions to avoid cache conflicts across memory regions [23, 71]. The third guides memory-access optimizations using run-time profiling results (*i.e.*, PGO) [51]. For example, APT-GET [35] improves prefetching of memory accesses to the CPU cache using profiling information collected from CPU counters.

These techniques cannot directly be used in a far-memory setting. Unlike Mira, they do not target a software configurable cache environment, do not co-design program analysis, compiler, and cache systems, do not work for far memory or perform any of our far-memory-oriented optimizations, and do not support function offloading.

Software-defined and configurable cache. There have been several works proposing configurable CPU cache architectures and software mechanisms to utilize such reconfigurable cache architectures [45, 64, 70]. For example, Jenga [64] proposes to assign different parts of CPU cache to different hierarchies (levels) based on measured cache miss curves for each application. Lee et al.[45] build a customized cache for streaming applications based on offline analysis of memory access traces. These solutions focus on the architecture and systems level, without the understanding or usage of program behavior and lack compiler optimization. Moreover, they all require non-traditional CPU cache hardware. Mira configures DRAM cache for far memory based on program behavior, with a run-time cache system, program-analysis, compiler, and profiling co-designed approach.

Another software-manageable cache hardware is scratchpad memory. Several works have focused on finding good ways to schedule what data to place in the space-limited scratchpad memory [36, 62, 66]. For example, Susu et al. [62] use static analysis and code transformation to perform space planning on scratchpad memory for an accelerator. Unlike Mira, these works do not configure scratchpad memory based on program behavior and instead seek good data placement and scheduling to fit the scratchpad.

Finally, TriCache [28] proposes to customize DRAM cache using a user-space block cache with a virtual memory interface to access fast storage devices. Unlike Mira, it does not utilize program behavior when configuring its block cache, and its usage scenario and cache designs are both different.

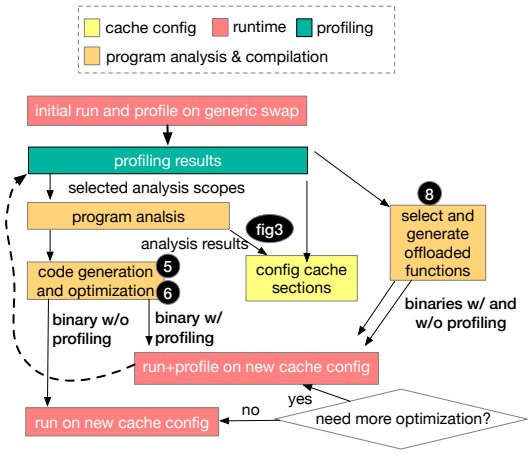

Figure 1. Mira Overall Flow.

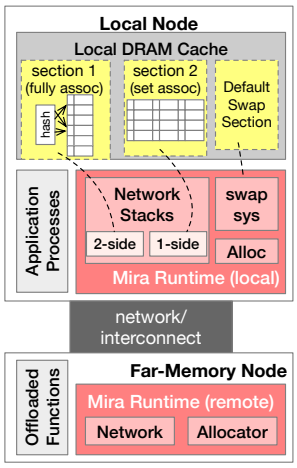

Figure 2. Mira Runtime.

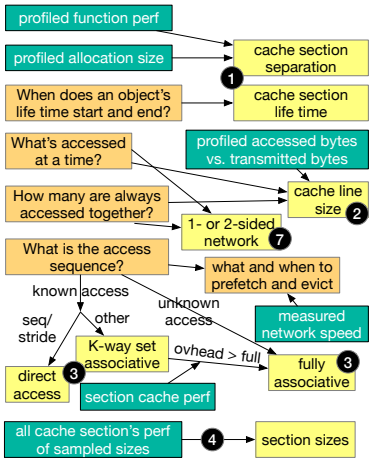

Figure 3. Deciding Cache Configuration.

3 Mira Overview

Mira consists of program analysis tools, a compiler, a run-time system for local nodes, a run-time system for far-memory nodes, and a profiling system. They work together to iteratively adapt system configurations and user programs for far-memory accesses, as shown in Figure 1. Figure 2 shows the run-time architecture of Mira. Mira takes an unmodified program as input and generates 1) a cache configuration based on the program's behavior and 2) a compiled code that runs on far memory via the Mira run-time system.

Overall flow. Initially, without run-time information or program analysis, Mira configures the local cache as a universal swap section and places all heap objects and static data in it (we never use far memory for stack or code, as they are small and frequently accessed). The initial execution works almost the same as traditional page swap-based systems, except for the profiling code our compiler inserts. At this and each of the later profiling runs, we collect per-function miss rate, miss latency, hit overhead (*i.e.*, the additional latency to access data in cache over a regular memory load), and function execution time. Additionally, we collect allocation sizes of all data objects.

We then decide how to split cache sections (initially, only the swap section) based on profiled per-function performance results and object sizes (§4.1). As each non-swap section needs program analysis and code generation/optimization, having many of them increases the static tools' complexity and is often unnecessary. Thus, we identify the functions that "suffer the most" from executing on the current cache configuration and compiled code, and we find larger objects in them to place in their own sections for further optimization.

Figure 3 illustrates the type of program analysis we perform and how we use the analysis results together with profiling results to determine various cache section configurations (details in §4.2). Overall, we use lifetime analysis to determine when to start and end a section, the amount of (batched) data accessed with profiled network performance to determine cache line size, and memory access sequences together with profiled cache section performance to determine cache structure. We determine the sizes of cache sections by globally optimizing the overall performance based on each section's profiled performance characteristics (§4.3).

For code ranges in each non-swap cache section, Mira compiles code to access the cache section or in the case of a cache miss, the far memory (§4.4). Mira converts memory operations like allocation, read, and write to *remotable* operations at the IR level, which then is lowered to either cache or network accesses. Afterward, we perform various code optimizations based on program analysis and profiling results, *e.g.*, prefetching data, batching far-memory accesses, flushing and marking data evictable, etc. (§4.5). We also generate code to access different network stacks of Mira based on program behavior (§4.7). Finally, we instrument the compiled code with coarse-grained profiling operations for the next round of profiling execution.

In addition to the above, we consider per-function computation load and network traffic to determine which functions to offload to far memory for optimal performance, and Mira generates binaries for them (§4.8).

Input adaptation. To adapt our compilation and cache configurations to inputs, we invoke profiling on sampled inputs. When the current compilation and cache configurations' performance degrades, we trigger a round of iterative code optimization in the background while the user invocation of a program keeps using the current compilation. Each iteration uses the previous iteration's profiling results to potentially set a new cache configuration and generates a new compilation. System administrators of Mira set an optimization target for each round (*e.g.*, at most 10 profiling-optimization iterations, or keep optimizing until no further gain is observed). The final compilation of a round is used for subsequent invocation of the program until another round of iterative optimization is needed. Each round of optimization

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

Figure 4. (Simplified) Code Example of Graph Traversal.

converges fast, usually in two to three iterations, and our profiling adds negligible performance overhead (§6). Our iterative approach reduces analysis and optimization scopes and complexity at each iteration while allowing for inaccuracy in one iteration to be fixed in the next one.

Overall, this sample-based input-adaptation approach has been taken by most prior profiling-guided-optimization (PGO) works [12, 35, 39–41, 51, 61, 63] and has been adopted in production [20, 51, 55]. As we (§6) and prior works show, this approach has only little mis-profiling overhead. This is because production workloads' inputs change slowly [20], and a fair amount of datacenter applications like machine learning benefit from the same sets of optimizations regardless of their inputs [15]. Additionally, as we will show in §4, many of Mira's designs are resistant to input changes.

Targeted applications. Mira optimizes code with memory access patterns that can be inferred from static analysis and dynamic profiling. Many datacenter applications fit this feature. Our evaluation results show the benefits of Mira for data analytics, machine learning, and graph processing applications (§6). Apart from our evaluated applications, Mira is potentially beneficial to other types of applications such as key-value stores and event-triggered applications. For applications or parts of an application that Mira does not optimize, we guarantee performance that is on par with existing swap-based far-memory systems.

Note that we assume each application to have its own cache space, far memory space, and cache runtime that are isolated from other applications. A datacenter/cloud manager can decide the amount of local/far memory space and CPU cores assigned to each tenant (application) and then run Mira in each tenant's container/VM.

4 Mira Design

This section presents Mira' design, including how we perform and utilize profiling, how we configure cache sections and their sizes, how we generate and optimize remote-access code, how we support multi-threading, different communication methods, and automated function offloading. We use a simple graph traversal program shown in Figure 4 as the rundown example of Mira's major designs. It traverses an edge array sequentially and updates the edge's source and destination nodes in a node array. Figure 5 shows the overall superior performance of this example when running on Mira as compared to FastSwap [9], Leap [8], and AIFM [56] for all local memory sizes. Figure 6 summarizes the effect of Mira techniques on this example. Here, and throughout the paper, we show relative performance that is normalized over native execution on full local memory (*i.e.*, no far memory).

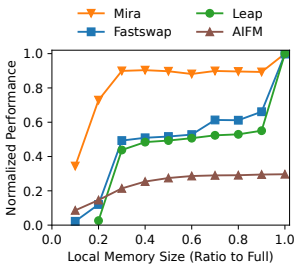

Figure 5. Edge Traverse Overall Performance.

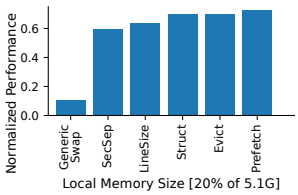

Figure 6. Performance Breakdown. *Orange lines show FastSwap performance.*

4.1 Profiling for Cache Configurations

As discussed in §3, to make program analysis more manageable, we leverage profiling results to pinpoint specific segments of a program that require analysis. Moreover, profiling results aid in identifying configurations that are challenging to determine through static analysis alone.

Profiling mechanism. Traditional profiling that happens at the run-time system can add fairly high-performance overhead that is not necessary for our profiling purpose. We instrument profiling code during compilation and only profile coarse-grained cache section performance at the function level or at allocation sites. Most of our profiling is related to a cache section's behavior (*e.g.*, miss rate, miss latency, hit overhead). These metrics are collected only when a non-native cache event happens, leaving native memory access intact and achieving lightweight profiling.

Determining cache sections and analysis scopes ❶. We leverage Mira's overall iterative flow to adaptively decide what data and code regions to place in a cache section, improving section selection with each iteration. After a profiling run, Mira collects the cache overhead and execution time of all functions. We compare the cache performance overhead across all functions and pick the highest 10% functions to analyze. Here and throughout the paper, we define cache performance overhead as the ratio of time spent in Mira runtime over the remaining program execution time, where the former includes handling cache hits (*e.g.*, cache lookup), misses (going across the network to fetch cache lines from far memory), and evictions. When selecting a function, we also implicitly select all its callee functions recursively for analysis. In the next iteration, if more optimization is needed, Mira uses new profiling results to pick the highest 20% functions to analyze, and so on (*i.e.*, 30%, 40% in the subsequent iterations until iteration stops).

After picking functions, we further nail down the analysis scope to large objects, as they need more space and will likely cause more cache misses. Similarly, we pick the largest 10% objects in the first iteration. If this function still needs to be analyzed in later iterations, we pick the largest 20% objects.

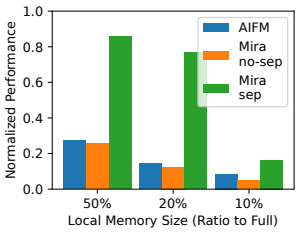

Figure 7. Effect of Cache Section Separation.

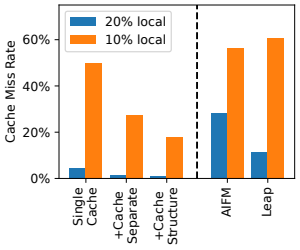

Figure 8. Miss Rate of Accessing Node Objects.

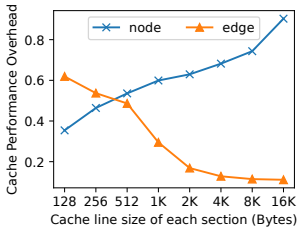

Figure 9. Effect of Different Line Size.

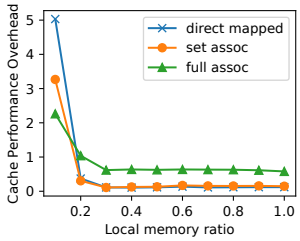

Figure 10. Effect of Cache Structures on Node Objects.

Users can set their own thresholds to replace these values we use for functions and objects. Even if we pick non-ideal objects and functions to optimize (*e.g.*, because of program input changes), our optimizations still improve application performance over generic swap-based systems.

After performing an analysis of the selected functions and objects and knowing their access patterns (§4.2), we group similar patterns into one section and leave different patterns in different sections. That means multiple objects can be in one section if their access patterns are similar, while one object can be in different sections at different times if its access pattern changes. Note that with the complexity and uncertainty of cache/code optimizations, separating a cache section may worsen its functions' performance. In this case, we roll back to the previous iteration's configuration.

Figure 7 shows the performance of Mira when not separating and separating cache sections with the graph traversal example. We also show AIFM [56]'s performance as a reference. Cache separation significantly improves Mira's performance. After initial iteration, Mira separates out two sections, one for the node array, and one for the edge array. To further understand where the benefit of cache separation comes from, we measure the miss rate of accessing the node array in Figure 8. In a joint cache, the sequentially accessed edge array could evict the randomly accessed node array and end up taking more space than what it needs (a few lines because of the sequentiality). After cache separation and assigning appropriate sizes (§4.3) to each section, the node array's miss rate drops by 44%-78%, while the edge array's miss rate stays the same. Cache separation also allows us to apply different cache structures to each section (more in §4.2), which further reduces the node array's miss rate.

4.2 Program Analysis for Cache Configurations

For the code regions selected using the profiling results (§4.1), we perform static program analysis to infer their access patterns, including their lifetime, access sequence, access granularity, access being read or write, and what data are often accessed together. We then use these analysis results together with profiling results to determine various cache section configurations to be used for the application's execution.

Determining cache line size ❷. A cache line in our system can contain one or multiple data items. We determine the cache line size of a section based on several factors. On the one hand, we want a cache line to be no larger than the data access granularity to avoid read/write amplification. On the other hand, if data items are accessed contiguously, we want to enlarge the cache line to cover as many of them as possible, as long as the line size is not bigger than what the network can transmit efficiently at a time. This is because accesses to each cache line need to go through a relatively costly pointer dereferencing process but accesses to an offset within a dereferenced cache line do not incur this overhead (§4.4). We take into consideration all these factors when setting cache line sizes.

Figure 9 shows the cache performance overhead (4.1) when using different cache line sizes for the node and the edge sections. For the node array, a smaller size is better, as it is accessed randomly. 128 bytes is the smallest size that can hold the accessed data unit. The edge array is accessed sequentially and thus benefits from larger line sizes. The cache overhead decreases dramatically when the line size is smaller than 2 KB because of our measured network characteristics.

Determining cache section structure ❸. Mira currently supports three cache section structures: directly mapped, set associative, and fully associative, following classical CPU cache architectures. Future works can add other structures. As with CPU cache, full associativity has the best utilization of cache space (*i.e.*, no conflict miss) but has a higher runtime overhead for cache lookup. This tradeoff shifts the other way with set associativity and then direct mapping.

To determine the structure of a cache section, we first analyze the access sequences of the program scope for a section to estimate the potential amount of conflicts that contend for a cache set or a direct location. If the access pattern is sequential or stride, then we use a directly mapped cache, as there will be no conflict. Otherwise, we analyze the locality set (*i.e.*, the entries of data that need to live in the local cache at the same time) and addresses in the locality set. If we cannot identify a locality set, we set the section to be fully associative. If we can find locality sets, we infer the potential amount of conflicts when using a K-way set-associative cache and set K accordingly.

Figure 10 shows the effect of using different cache structures on the node section. When local memory is large, full associativity has a constant overhead over set associativity and direct mapping. As local memory gets smaller, full

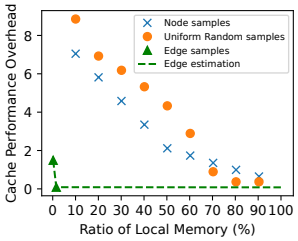

Figure 11. Cache Performance Overhead with Sampled Sizes

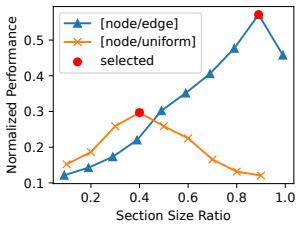

Figure 12. Section Size Selection.

```
1  @_redges, @_rnodes = remotable.alloc(..)
2
3  // parameter uses internal edge struct representation
4  remotable.func @trvs_graph_rmt(%arg0: !remotable<struct<edge>>){
5    scf.for %i = %0 to %num_edges { // scf is an MLIR dialect
6      // dereference remote pointer to local pointer
7      %1 = rmem.deref %arg0[%0]
8      %2 = rmem.deref %1->from
9      %3 = rmem.deref %1->to
10     func.call @update_node (%1, %2, %3)
11   }
12 }
```

Figure 13. Convert to Remotable for Graph Example.

associativity turns better than set associativity. Note that even though different cache structures for the node section have small differences, choosing the right cache structure for different sections has a larger impact on performance.

4.3 Determining Cache Section Size

As discovered by previous far-memory systems [9, 56, 58], the amount of local cache can largely impact far-memory system performance. Different from previous systems that only consider the effect of total cache size for an application's performance, we consider the effect of each cache section's size, as different objects and their access patterns can be affected differently by the amount of local cache. We use sampling and profiling to determine section sizes. ❹

We first sample a few sizes for each section. In each sampled run, we profile the cache performance overhead of the section. Sequential and strided cache sections only need a small size that can fit enough prefetched data to hide network delay. Beyond this size, the performance of these sections would stay the same. Thus, we only need to sample very few sizes to find a sequential/strided section's optimal size. For other cache sections, we sample a few section sizes as ratios of total local memory size (*e.g.*, 20%, 40%, 60%, 80%). After acquiring the relationship between section size and section performance for them and with our program analysis results of section lifetime, we construct an integer linear programming (ILP) problem with the target of minimizing the total cache overhead and the constraint that during any time, the total size of live sections should be no larger than the total application's local memory space. The solution to this ILP problem is the sizes we use for these sections.

Figure 11 shows different cache sections' performance overhead when sampling different section sizes. As the edge array is accessed sequentially, a small size can already achieve the same performance as the full size. The node array's accesses are indirect, and its section cache overhead is non-linear from our sampled results. To make the section size selection problem more interesting, we add a third array that is accessed uniformly randomly to be in another section, which also exhibits non-linear behavior with different section sizes. Figure 12 shows normalized application performance when partitioning the local memory differently across multiple sections and the partition ratios Mira's ILP solutions give (which are the optimal ratios). As expected, the optimal selection between node and edge arrays is to give most memory to the non-sequentially accessed node array. The ratio between the node array and the third array follows their sampled performance results.

4.4 Conversion to Remote Code

Converting to remote pointers and operations. Our compiler generates explicit remote operations for objects in non-swap cache sections; swap sections run the original code. As explained in §4.2 and discovered by previous API-based far-memory solutions [56], explicit remote operations can more precisely control far-memory accesses and thereby improve application performance. Specifically, Mira turns all pointers that point to selected objects (as in §4.1) in non-swap sections to remote pointers (defined in Mira's IR §5.1). It then turns allocation, load, and store operations of these remote pointers to their corresponding remote APIs (*e.g.*, remote load/store, see §5.1). Figure 13 shows a simplified code converted to remote operations for the graph-traversal example, using notations in §5.1.

Lowering remote operations ❺. When a remote pointer is dereferenced, resolving it could involve three steps: 1) looking up the pointer in the local cache; 2) if not found in the cache, fetching the data from far memory to the local cache; and 3) the actual data access. The third step is unavoidable. We perform prefetching to hide the overhead of far-memory accesses (step 2), to be discussed in §4.5.

We now describe how we optimize the first step of cache lookup. Normally, each cache lookup would require a set of instructions to locate whether or not and where the pointed-to data sits in the local cache. However, if we have already accessed a cache line and know that it is still in the local cache, we would know its local memory address. For future accesses of any data item in the same cache line, we can directly resolve the dereferencing by using the already obtained local address and an offset in the cache line. In these cases, the Mira compiler converts a remote pointer dereferencing to a native memory load instruction.

Note that the above optimization is only possible if the cache line is in the cache when the dereferencing happens. In a single-threaded program, Mira knows from program analysis whether or not there are any potential accesses to data that may fall into the same cache set (set-associative) or cache slot (direct mapped) before the dereferencing site. If

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

func.func @trvs_graph_opt(%arg0: !remotable<struct<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 corresponding phyiscal address (paddr) of cache line
    %wide_cache_line = rmem.paddr %SEdge, %arg0 + %i

    scf.for %j = %0 to %elements_per_line {
      // directly load element in (already resolved) cache line
      %1 = memref.load %wide_cache_line[%j]

      // use later element in the line to prefetch node elements
      %2 = memref.load %wide_cache_line[%j + %n_ahead_node]
      // 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 14. Mira Optimizations for Graph Example. *We show optimizatios of prefetching and eviction flush, not showing others for simplicity.*

no such "conflicting" accesses exist, we can safely know that the cache line will not be evicted and can perform the above optimization for that dereferencing site. When our analysis finds conflicting accesses or is unsure about the occurrence of conflict accesses, Mira can mark cache lines as "*dont-evict*" to indicate that evicting them would cause a huge overhead. Our runtime would choose to evict them the last. When our intended dereferencing sites all finish for a dont-evict cache line, we will remove the mark.

With these optimizations, we reduce not only the runtime overhead but also the metadata needed for far memory. Compared to Mira, AIFM's [56] library-based remote operation implementation has a much higher run-time overhead. AIFM needs to perform pointer dereferencing for each remote data item (*e.g.*, an element in a remote array), as AIFM does not perform program analysis and cannot apply native-instruction optimizations like ours. Moreover, AIFM maintains a significant amount of metadata for each remote pointer, *e.g.*, a "dereferencing scope" to manage pointer lifetime. It encounters high run-time overhead using and bookkeeping the metadata. Mira's analysis directly infers object lifetime and other information and uses them to compile code as native memory instructions if possible. Mira does not need any metadata for cache lines whose lifetime it can fully control. For example, in a loop whose accessed far-memory data can all be prefetched and do not have cache line access conflict, we do not need to maintain or access any metadata like cache line tags and pointers referencing to the line, all accesses are compiled as native memory instructions.

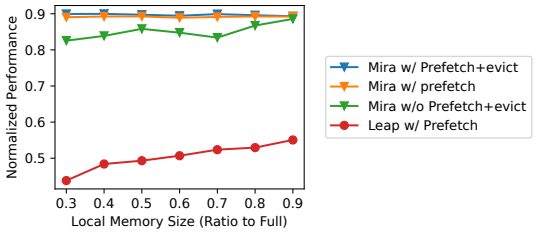

Figure 15. Effect of Prefetch and Eviction Hints.

4.5 Program Optimization

Apart from generating remote code, our compiler performs code optimizations in various ways as discussed below. ❻ These program-based optimizations provide benefits across different inputs without the need for recompilation.

Adaptive prefetching. Prefetching is a common technique used to reduce the overhead of far-memory data accesses. Previous systems [8, 16] use generic policies to determine what data to prefetch based on run-time access history. Instead of predicting future accesses based on run-time history, we use program analysis to determine what will be accessed in the future. For example, for a multi-level loop over a set of memory accesses, we prefetch them based on the loop pattern. Different from traditional CPU cache prefetching, we determine when to prefetch based on system environments (*e.g.*, measured network delay). Our compiler inserts prefetch operations at the program location that is estimated to be one network round trip earlier than actual access.

Eviction hints. From our program analysis (§4.2), in many cases, we can find the last access of a data element in a program scope (*e.g.*, a function). In these cases, our compiler inserts an asynchronous cache-line flushing operation after the last access and marks the line as *evictable*. When inserting a new cache line, we check which existing lines are marked evictable and evict those first. As the least useful lines are marked with our program-guided hints, Mira improves the local cache utilization. If there is no line marked as evictable, Mira uses a default LRU-like eviction policy.

Figure 15 shows the benefit of adding prefetching and eviction hints in Mira when running the graph-traversal example. Prefetching hides the latency for sequential edge accesses, and early eviction hides the write-back overhead behind the performance critical path. For this application, the former has a larger impact. We also evaluate Leap [8], which performs majority-history-based prefetching. Leap only aims at capturing global access patterns and cannot properly prefetch for an interleaved access pattern like this example. Moreover, Leap uses the default Linux global eviction policy, not getting any benefits from program hints.

Selective transmission. A major problem with swap-based systems is their coarse far-memory access granularity. New programming models like AIFM [56] allow programmers to define the exact data structures to move between local and far memory. However, programmers could make unoptimized decisions and fetch more data from far memory than needed.

For example, if a programmer defines a large data structure as a `remotable` object, AIFM fetches the entire data structure from far memory even when only a few fields are accessed.

To solve this problem and minimize traffic between local and far memory, our approach is to use program analysis to determine the parts in a data structure that are accessed in each program scope (*e.g.*, a function). We then generate code to only fetch or prefetch these parts.

Data access batching. For most networks and interconnects, one large communication event (*e.g.*, a message with multiple scatter-gathered data pieces) is more efficient than multiple smaller communication events. We seek program transformation opportunities leveraging this feature. If our program analysis identifies multiple addresses to be accessed at different locations, we batch them into a single network message by transforming the code. For example, when we identify two arrays to be accessed by two adjacent loops, we fuse the loops and batch access the two arrays.

Read/write optimization. In many cases, a read-only or write-only access pattern can be leveraged to achieve better performance. If a loop only contains read operations, we can safely discard the local cached objects after the loop. If it only contains writes that cover whole cache lines, we can avoid fetching the objects from far memory.

4.6 Multi-Threading Support

Multi-threaded programs have non-deterministic shared memory access behavior, bringing new challenges. Our solution for supporting multi-threading differs for programs that have no shared-memory writes and those that have them. For the former, *i.e.*, multi-threaded programs that are shared-nothing, read-only, or have unique ownership [13], we create separated cache sections for each thread. If multiple threads read the same data, each thread's cache section will have a copy of it. Thus, we could treat each cache section in isolation and apply all our optimizations discussed above.

We use shared cache sections for writable shared-memory multi-threading. We configure shared sections in a conservative way: full associative with cache line size being the largest access granularity among all accessing threads. We apply all optimizations presented in §4.5 except for eviction hints. Shared cache sections complicate the no-conflict analysis discussed in §4.4, as static analysis alone cannot determine whether a cache line could be evicted by another thread before it is accessed by the current thread. Instead, we mark a cache line as "dont-evict" from a thread's dereferencing time until the end of the line's lifetime in all threads. We perform lifetime analysis for "dont-evict" cache lines by keeping the reference count for each shared object and decreasing the count when all accesses from a thread finish.

Finally, traditional thread synchronization methods such as locks still work as is on Mira since we never make synchronization primitives remotable, and real data accesses only occur at local caches that are protected by traditional synchronization primitives.

4.7 Data Communication Methods

An important part of far-memory systems is the data communication between local and far-memory nodes, either over the network or over a local bus/interconnect. Many prior works have studied the benefits and use cases for one-sided communication where data is directly read/written from/to far memory vs. two-sided communication where data is sent as messages and far-memory nodes copy the messages to their final locations [65, 69]. These works manually design the communication methods for specific application domains.

We decide what communication method to use for each cache section based on its access pattern ❼. If our program analysis finds that a section's access pattern is reading/writing the entire data structure, then we use one-sided communication for this section to directly read/write the data structure with zero memory copy. If a section only accesses partial data structure (*e.g.*, one or two fields of it), then we use two-sided communication to only transfer the partial structure, avoiding read/write amplification. To achieve this, our compiler inserts code to prepare/process a message by copying from/to the partially accessed data fields.

4.8 Function Offloading

Certain types of far memory nodes have computation power that can execute application code [31, 56], allowing the offloaded computation to access data in far memory locally, reducing the network transfer overhead. To exploit this benefit, existing works require programmers to decide what computation to offload to far memory nodes and sometimes even rewrite offloaded computation. Mira automatically determines and offloads computation to far memory in the following program- and profiling-guided manner ❽.

To reduce the program-analysis complexity, we only consider program functions as the unit of offloading and functions that do not have shared writable data. Future work could include functions with shared writeable data with the support of new coherence hardware like CXL [22]. Among the candidate functions, we determine which ones to offload to far memory based on their amount of computation and required network communication. As far-memory nodes usually have less computation power (*e.g.*, with a low-power ARM processor), it is more beneficial to offload computation-light functions to far memory. Additionally, it reduces network communication to offload functions whose accessed data are already in far memory. Thus, we consider both factors when choosing functions to offload.

To implement function offloading, we insert code at the compute node to flush the local cache that contains data the function accesses before invoking the function. The compute node then calls the offloaded function with an RPC call and

sends the function inputs to far memory. After the far memory node finishes executing the offloaded function, it sends the return data to the local side.

Currently, Mira only supports offloading to CPU-based far-memory nodes. It could be extended to support other types of computing units by leveraging MLIR's capability of generating code for accelerators like GPU [38] and co-processors [1]. Similar to CPU-based nodes, offloading decisions for accelerators could be made based on computation needs and data-movement overhead.

5 Implementation

We implement Mira's program analysis and compiler on top of MLIR with 7.7K LOC in C++. We implement Mira's runtime libraries that run on the local node and far-memory node with 12.1K LOC in C++. This section discusses some of the implementation details.

Mira currently runs on one compute and one memory node. Supporting multiple memory nodes, or *memory pooling*, can be done via the integration of Mira and a distributed memory management layer such as the one used in LegoOS [58], where Mira decides what objects and functions to offload and the distributed memory manager decides which memory node to offload them to.

5.1 Far-Memory MLIR Abstractions

MLIR. MLIR (Multi-Level Intermediate Representation) [44] is a compiler ecosystem that allows multiple abstractions at different levels. Each abstraction is called a *dialect*. Currently, MLIR supports tens of dialects for common operations, such as memory accesses, control flow, arithmetic, machine learning, and LLVM [4]. We choose to build our compiler in the MLIR ecosystem because it supports multiple frontend languages and backend architectures. Moreover, it allows us to easily add various far-memory abstractions and code optimizations as dialects at different layers while reusing existing MLIR dialects and their optimizations. Note that Mira analyzes and optimizes all libraries whose source code is available (*e.g.*, C++ STL), in the same way as application programs. We run pre-compiled library calls on our generic swap cache.

We add two new MLIR dialects for far memory:

`remotable`. The `remotable` dialect defines a new abstraction for data objects in non-swap cache sections and for functions that can be offloaded. Lines 1 and 4 in Figure 13 shows the allocation of a `remotable` object and the definition of a `remotable` function.

`rmem`. The `rmem` dialect defines operations to access and manipulate `remotable` objects and functions, including two main types. The first is basic object accesses such as load and store, by extending traditional pointer operations and `memref` [5] operations in MLIR to work with `remotable` objects. For example, lines 7 and 8 in Figure 13 perform memory loading from `remotable` objects `%arg0[%0]` and `%1`. The second type is code optimizations such as prefetch. For example, lines 7 and 9 in Figure 14 perform an asynchronous `fetch` of an object to be accessed in a future loop iteration and blocking `wait` the data needed for the current iteration.

5.2 Static Analysis and Code Generation

We now discuss how we analyze programs and generate code with the `remotable` and `rmem` dialects. Our analysis is sound, as we trade completeness for correctness and fast analysis time. There could be rare cases where our analysis cannot infer (*i.e.*, "undecidable"), and we avoid their optimizations.

5.2.1 Implementing `remotable` and `rmem`. We now discuss how we implement `remotable` and `rmem` abstractions.

Converting to `remotable` and `rmem`. Mira identifies data objects to place in far memory based on analysis explained in §4.1 and turn them into `remotable` objects. If a field in a structure is identified, we turn the whole structure into `remotable`. Afterward, Mira finds all pointers pointing to `remotable` objects via forward dataflow analysis (lattice static-single-assignment, or SSA-based, analysis [2]) and type-based alias analysis [24]. These pointers all become `rmem` pointers, and we convert the original memory accesses to the corresponding `rmem` operations.

Afterward, we perform an SSA-based backward analysis to find all the functions where an `rmem` pointer is passed as a parameter. If a function only accesses `remotable` objects, stack variables, and heap variables allocated and released within the function scope, then we mark the function as `remotable`. Note that the same function may be called with a non-remote pointer (*i.e.*, pointing to a local object). In this case, we create another version of the function definition that is not `remotable`.

As the above backward and forward analysis involves the whole program, we avoid invoking them as much as possible by storing analysis results, including the relationships between functions and `remotable` objects and each function's references to `remotable` objects. Later compiler optimizations could reuse these results without going through the costly whole-program analysis again.

Implementing `remotable.alloc`. We use the combination of a local allocator and a remote allocator to implement the allocation of memory space on the far memory node. The remote allocator works like a low-level systems allocator (*e.g.*, `mmap` in Linux) and performs the actual memory allocation at far memory. The local allocator acquires allocated far-memory addresses from the remote allocator and buffers the addresses locally; so it works like an allocator in a language library (*e.g.*, `malloc` in clib). When a `remotable.alloc` is called, the local allocator first checks if there is a buffered memory address range that is no smaller than the allocated size. If so, it directly assigns one to the allocation site. Otherwise, it asks the remote allocator for more addresses. As the

allocated addresses are the virtual memory addresses at a far-memory node, our RDMA-based network stack can use them to perform one-sided accesses directly (§4.7).

Loading an `rmem` pointer from far memory. We now explain how Mira dereferences an `rmem` pointer. Initially, an `rmem` pointer has the value of an allocated far-memory address for a remotable memory space. When an `rmem.load` happens, Mira first checks if the data the `rmem` pointer points to has been fetched to the local cache already by searching for the far-memory address in the designated local cache section. If not, Mira fetches the data object from far memory and places it in the section. For the next step of this case or for the cache-hit case, we set the section ID and the offset of the object within the section as the value of the `rmem` pointer, with the former occupying the highest 16 bits and the latter occupying the lower 48 bits. Then to access the actual data, we map the section ID and offset to the virtual memory address of this cache line plus an offset within the line. This is the virtual memory address seen by the local node MMU, which performs the actual memory access. The Mira compiler generates corresponding code for all the above steps during compilation.

Pointers to both local and `remotable` objects. An `rmem` pointer could be set to point to a `remotable` object or a local object at runtime in different executions (*e.g.*, based on an `if` condition). A potential problem of such cases is that the `rmem` pointer will have a normal memory address when pointing to a local object but address constructed as section ID and offset when pointing to a `remotable` object. If we use the same process to dereference an `rmem` pointer by locating the cache section and offset, accesses to local objects would be wrong. To solve this problem, we use a simple method: reserving a dummy, non-existent cache section (section 0, as the highest 16 bits for normal addresses are 0) to represent all `rmem` pointers that point to local objects. When Mira finds a cache section ID zero during dereferencing, it treats it as a local object and maps it to the proper local address.

Generating offloaded function binaries. At compile time, Mira turns `rmem` accesses within the offloaded function into raw memory accesses and `rmem` pointers into raw pointers, as the function will run on the node that contains the `remotable` objects. The pointer addresses will be assigned by the remote allocator in the remote virtual address space. On the local node side, we implement the call of a `remotable` function as an RPC call. To ensure that a `remotable` function sees the up-to-date `remotable` objects during its execution, we flush all cached `remotable` objects that the `remotable` function accesses to far-memory before calling the function.

5.2.2 Behavior Analysis.

For the performance-critical sections we identified, Mira performs a detailed analysis of memory operations, concerning the range of addresses that will be accessed in each section. We use memory dependency analysis [49] together with scalar evolution [14] to reason about memory accesses and their patterns within a code block (access address sequence, granularity, read/write, possible batching). We further analyze memory accesses across code blocks and function boundaries. For example, if addresses touched within a basic block suggest certain locality, we can batch multiple `rmem` pointer dereferences within that block to reduce the runtime overhead. If this block happens to be the body of a loop, the address representation at the loop level will guide our prefetch optimization and reveal batching opportunities across iterations.

5.3 Cache Section Implementation

Fully-associative cache. We maintain remote-address-to-physical-address maps and a list of available free physical cache lines for fully associative caches. The former is used for cache lookup, while the latter is used for cache insertion. For our compiler-inserted prefetch and eviction hints, we implement the actual operations in our runtime. Additionally, we implement an approximation of LRU eviction using active and inactive lists for when an on-demand eviction is needed.

Swap-based cache section. Different from other sections that use compiler-generated code for cache accesses, the swap cache transparently executes the original code via our implemented user-space swap system (on top of Linux `userfaultfd` [11]). The line size in the swap cache is 4 KB, consistent with OS default page size. Mira manages a physical page pool in RDMA region for the swap section. Unlike other sections, the mapping between virtual addresses to physical pages in the swap section is dynamic. Mira sets up, tears down, or changes mappings when there are `userfaultfd` events, prefetching operations, or eviction hints. Mira evicts a page based on an approximate global LRU policy.

6 Evaluation

We evaluate Mira on a Cloudlab [27] cluster of eight c6220 servers, each equipped with two 8-core Intel Xeon E5-2560 CPUs (2.6 GHz), 64 GB RAM, and a 50 Gbps Mellanox FDR-CX3 NIC with 50 Gbps Infiniband network.

Applications. We select three applications to evaluate Mira: *DataFrame*, *MCF*, and *GPT-2 inference*, representing common code patterns (*e.g.*, data access pattern, threading model, etc.) and common datacenter application types (data analytic, ML inference, graph processing), being open sourced, and having fairly large memory consumption.

DataFrame [34] is a data analytic system written in 24.3K LOC C++. The Dataframe system provides a set of data analytic operations, such as filtering, grouping, etc., on a collection of named columns called a DataFrame. When operating on large data sets, DataFrame can be both compute and memory intensive, making it a good candidate for far memory.

GPT-2 [50, 60] is a transformer-based [67] large language machine-learning model with 100M to 1.5B parameters. We

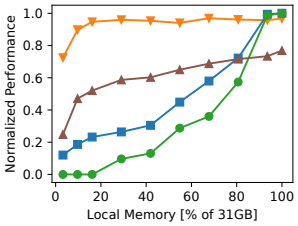

Figure 16. DataFrame Performance.

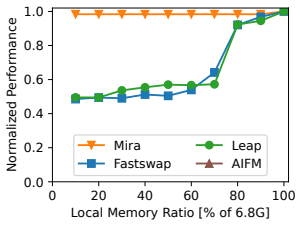

Figure 17. GPT-2 Performance.

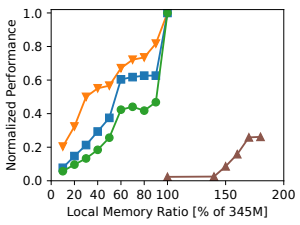

Figure 18. MCF Performance.

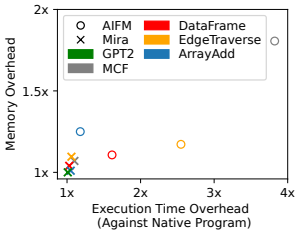

Figure 19. Runtime Overhead. *when running on full local memory.*

perform GPT-2 inference on ONNX [7], an open AI ecosystem that is compatible with MLIR [6]. 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 [3, 47, 52], as GPU is not always available (*e.g.*, in serverless computing services). A common technique used by transformer inference is to cache computed values called keys and values to avoid recomputation for better inference latency. Key-value caches consume device memory that can be several times bigger than the model itself [53]. Instead of manually figuring out what data to place in far memory, Mira automatically identifies key-value data for far memory.

MCF [10] is a benchmark from the SPEC 2006 benchmark suites [32]. MCF 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.

Systems in comparison. We compare Mira to three systems: AIFM [56], FastSwap [9], and Leap [8]. 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.

6.1 Overall Application Performance

DataFrame. Figure 16 shows the overall DataFrame performance of Mira, AIFM, FastSwap, and Leap as the local memory size increases. For Mira, we use a compilation that is "trained" using the 2014 year of the New York City taxi trip dataset [37] and tested on the dataset's 2015-2016 year data. Mira outperforms FastSwap and Leap because Mira separates and customizes cache sections (*e.g.*, precise prefetching for each section, proper cache line size, etc.). Without cache separation, FastSwap and Leap's swap-based global optimizations do not work well for each distinct program behavior. Leap performs worse than FastSwap even when it does majority-based prefetching, mainly because of FastSwap's more efficient data-path implementation in Linux. AIFM has significant runtime overhead in pointer dereferencing, as it needs to resolve every access of a remotable pointer. This overhead is also the reason why even at 100% local memory, AIFM is still a lot slower than other systems.

GPT-2 inference. Figure 17 shows the overall GPT-2 inference performance of Mira, FastSwap, and Leap. AIFM does not currently support any matrix or other machine-learning operations, thus we do not evaluate GPT-2 on AIFM. For Mira, we obtain GPT-2's compilation from a randomly generated input sequence batch and tested it on a few other sequence batches (*e.g.*, sentences from wiki pages). Mira's performance stays the same even when local memory size reduces to only 4.5% of full memory. DNN model inference like GPT-2 exhibits a layer-by-layer computation pattern, where data used in one layer (*e.g.*, that layer's weight matrix, the input matrix to this layer) is not needed anymore in the remaining layers. Our program and profiling analysis correctly capture this pattern and separate matrices in different layers into different cache sections, end their lifetime when their corresponding layers finish, perform batched access of data used in each layer, and generate precise prefetching and eviction hints. As a result, most remote access overhead can be hidden behind performance-critical paths, and even a small amount of local memory is enough to saturate computation throughput. In contrast, FastSwap and Leap both experience high performance downgrade as local memory size shrinks, because without the knowledge of program behavior, they are not able to fetch the precise set of data for computation and end up using most of the local memory to cache data that is not used soon.

MCF. Figure 18 shows the overall MCF performance. We ran MCF with the example graphs given by SPEC-2006 [32], one being 4x larger than the other. Here, we show the results for the smaller graph. Mira configures the default swap cache for the larger graph, achieving similar performance as FastSwap.

Being a graph-like application, MCF's memory accesses are highly dependent on pointer values and also on program control flows. Thus, it is the application that is the least friendly to program analysis tools among the three. Nevertheless, Mira is able to make appropriate cache configuration and prefetching/eviction decisions with our co-design approach. When local memory is larger than 70% of full memory, Mira uses the swap section for the main array (which is

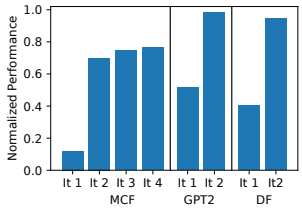

Figure 20. Iterative Optimization with Applications.

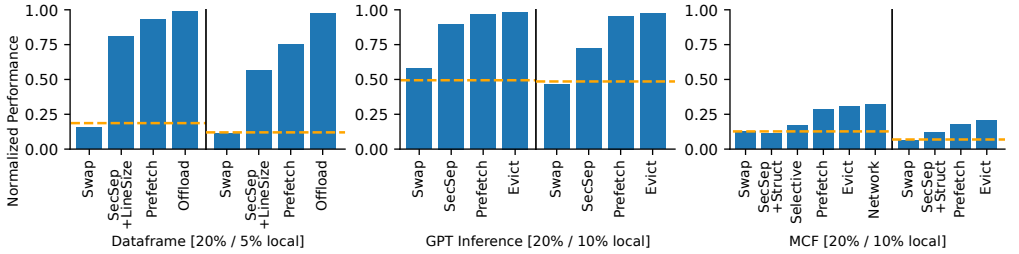

Figure 21. Mira Technique Breakdown. *One technique is added at a time. Orange lines show the performance of FastSwap.*

accessed indirectly by pointer values). When local memory is small, Mira discovers performance overhead at the swap-based cache sections from profiling and changes it to be a set-associative section. Mira prefetches data by following pointers, in a similar way as our rundown example.

In contrast, FastSwap and Leap use swap regardless of local memory size and do not prefetch data by following pointers due to their system-level-only approach. We use AIFM's array library to implement MCF's data structures allocated in continuous memory, as the current AIFM does not provide a library for linked pointers. Surprisingly, AIFM fails to execute when local memory is smaller than full size. Even at full memory, AIFM's performance is orders of magnitude worse than the other systems, and its performance only rises to 26% even when the local memory size is 80% *larger* than the full memory size. The reason behind this is that AIFM requires a significant amount of metadata for their remotable pointers, which reduces the local memory space usable by actual data. Mira's metadata is much smaller. Instead of maintaining various information like lifetime with each remote pointer, Mira directly uses such information during compilation. Moreover, Mira only maintains per-cache-line metadata and one cache line contains multiple elements. Besides metadata overhead, AIFM incurs costly pointer dereferencing for every element in an array, while Mira avoids most pointer dereferencing with the cache-line optimization in §4.4.

Runtime overhead. To understand the run-time behavior of Mira, we measure the run-time performance overhead and metadata overhead when running at full local memory with the three applications, the graph-traversal example, and a simple loop over an array for summing the array value. Other than GPT-2, we also show the run-time behavior of AIFM. Figure 19 shows the results compared to running original programs on regular Linux as the baseline. Even given full memory (*i.e.*, no need to use any far memory), AIFM has significant performance and metadata overhead.

Iterative Optimizations. Figure 20 shows the iterative process of Mira in three applications with 80% local memory. The initial iteration runs MCF on the generic swap cache. After the first iteration, Mira identifies two large objects, and only one of them causes high-performance overhead. Mira analyzes it and configures a set-associative section for the next iteration. In the next iteration, Mira detects read amplification and reduces the cache line size accordingly. MCF performance converges after iteration four. For GPT-2, from the first iteration, we identify 122 large objects to be placed in isolated sections. Most of them have highly predictable access patterns and non-overlap lifetime. Mira achieves optimal performance in two iterations. Similarly, Dataframe reaches the best configurations in the second iteration.

Analysis and compiling overhead. By leveraging profiling results, we are able to reduce the program analysis scope of MCF [10] from 1.8K lines of code to only three functions with 0.3K lines of code and that of ONNX GPT-2 inference from 1000+ allocation sites to 122. With the reduced scope, Mira's program analysis and compiler finish fast even for a large program (*e.g.*, 3.93 seconds for GPT2 with 36K LOC). Mira's runtime profiling adds 0.4% to 0.7% profiling performance overhead for the three applications we evaluate. Putting things into perspective, previous works [20, 40] incur 3.3% to 978% profiling overhead.

6.2 Performance Deep Dive

To understand where Mira's benefits come from, we evaluate the effect of different Mira techniques by adding one or two at a time, as in Figure 21. Different applications with different amounts of local memory benefit differently from these techniques, and we select several distinctive ones to present. The baseline Mira places entire heaps in the generic swap cache, *i.e.*, the initial run. Below, we explain the effect of each technique and their impact on the three applications.

Cache section separation. As shown in Figure 21, when adding cache section separation on top of the swap-cache baseline, all applications except for MCF have a huge performance improvement. This is both because we use separated sections with different configurations and because we end a section as soon as its lifetime in the program ends, leaving precious local memory space for other sections. The latter is especially useful for GPT inference, as Mira promptly and precisely release matrices used by one layer after that layer's computation finishes, which enables the freed space to be immediately used by the next layer. This type of memory management is a technique manually added in previous DNN systems [21], and Mira automatically generates the optimal memory usage based on program analysis and profiling. MCF

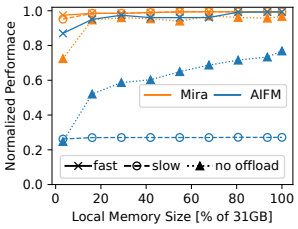

Figure 22. DataFrame with Function Offloading.

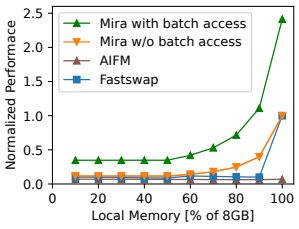

Figure 23. Access Batching with Dataframe.

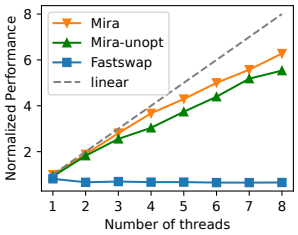

Figure 24. Read-only Shared Multi-threaded GPT-2.

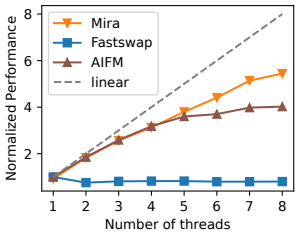

Figure 25. Writable Shared Multi-Threaded Dataframe.

does not benefit as much from cache separation, because a large portion of it has access patterns that fit swap the best.

Prefetching and eviction hints. With a program analysis approach, Mira can uncover more memory access information such as access sequences in a program, which we leverage to prefetch and evict data. As shown in Figure 21, adding prefetching and eviction hints on top of separated cache sections improves performance further. Eviction hints have smaller improvements compared to prefetching, as they have a longer-term, indirect impact on future accesses while prefetching has a direct impact on the subsequent accesses.

Function offloading. Mira identifies 7 functions and basic blocks to be offloaded to far memory for the DataFrame application. As shown in Figure 21, function offloading improves DataFrame's performance, and the effect is larger when the local memory size is low.

To further understand function offloading, we compare Mira with AIFM which also supports function offloading. Figure 22 shows their performance under different local memory. We evaluate our test bed where the far-memory node is a normal server (*fast* in the figure) and an emulated slow memory device (*e.g.*, a SmartNIC) where computation on the device runs five times slower than the normal server. Both Mira and AIFM's function offloading improves performance over their respective no-offloading setups, especially when the local memory size is small. Unlike AIFM, Mira's no-offloading setup already achieves near-optimal performance.

With a slow far-memory device, AIFM performs much worse than Mira. With AIFM, offloaded functions are selected manually and do not adjust to far-memory devices' capabilities. When a far-memory device is slow, offloaded functions' computation overhead can outweigh the benefit of offloading (*i.e.*, reducing data communication). Mira automatically adjusts what functions to offload based on profiling and system environments. With the slow device setup, Mira only selects two offloading targets, both functions are data-access heavy reduce operators.

Data access batching. Data access batching is an effective optimization technique when batching opportunities exist in a program. We report this effect in Figure 23 with a DataFrame job that performs three operators (avg, min, max) on the same vector, whose original code has three consecutive loops over the vector. Mira discovers the batching opportunity, fuses the loops together, and batch-fetches the vector. We compare Mira with and without batching and with AIFM and FastSwap. Batching largely and consistently improves Mira's performance with different local memory sizes. As a library-level approach, AIFM implements each operator in isolation. Without program knowledge, approaches like AIFM cannot do the type of data batching Mira does.

Multi-threading. We evaluate our support of read-only multi-threaded applications using GPT-2 inference. Figure 24 shows the relative performance when increasing the number of threads used for Mira and FastSwap. Mira scales much better than FastSwap with multi-threading. By separating cache sections to be private to each thread, Mira further improves performance over Mira-unopt. Moreover, FastSwap's limited scalability is related to its Linux-based swap system, which has various synchronization and locking bottlenecks [54].

To evaluate writable shared memory, we run the Dataframe "filter" operator which uses multiple threads to write to a shared result vector. As shown in Figure 25, Mira scales better than FastSwap and AIFM, as most of Mira's optimizations still apply to writable shared multi-threading.

7 Conclusion

We presented Mira, a far-memory platform that co-designed program analysis, compiler, run-time profiling, and runtime systems. By leveraging the unique opportunities of far-memory environments and by overcoming challenges, we show that Mira significantly outperforms existing far-memory systems.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd Andrew Quinn for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We are also thankful to Geoff Voelker, Tianyin Xu, and Linhai Song for their valuable feedback on our work.

This material is based upon work supported by the National Science Foundation under the following grant: NSF 2022675, and gifts from Google, Meta, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] 2020. *MLIR SPIR-V Dialect.* https://mlir.llvm.org/docs/Dialects/SPIR-V/

[2] 2021. *Writing DataFlow Analyses in MLIR.* https://mlir.llvm.org/docs/Tutorials/DataFlowAnalysis/

[3] 2023. *Llama.cpp 30B runs with only 6GB of RAM now.* https://news.ycombinator.com/item?id=35393284

[4] 2023. MLIR llvm Dialect. https://mlir.llvm.org/docs/Dialects/LLVM/.

[5] 2023. MLIR memref Dialect. https://mlir.llvm.org/docs/Dialects/MemRef/.

[6] 2023. ONNX-MLIR. https://github.com/onnx/onnx-mlir.

[7] 2023. Open Neural Network Exchange. https://onnx.ai/.

[8] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC '20)*. Virtual.

[9] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Heraklion, Greece.

[10] Andreas Löbel 2011. *Combinatorial optimization Single-depot vehicle scheduling.* https://www.spec.org/cpu2006/Docs/429.mcf.html

[11] Linux Kernel Archieves. 2018. *Userfaultfd — The Linux Kernel documentation.* https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html

[12] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Phoenix, Arizona.

[13] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System programming in rust: Beyond safety. In *Proceedings of the 16th workshop on hot topics in operating systems (HotOS '17)*. Whistler, Canada.

[14] Daniel Berlin, David Edelsohn, and Sebastian Pop. 2004. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit.*

[15] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2022. 3PO: Programmed Far-Memory Prefetching for Oblivious Applications. arXiv:2207.07688 [cs.OS]

[16] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA.*

[17] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. 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.

[18] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. 2019. Project PBerry: FPGA Acceleration for Remote Memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Bertinoro, Italy.

[19] Gautam Chakrabarti and Fred Chow PathScale. 2008. Structure Layout Optimizations in the Open 64 Compiler : Design , Implementation and Measurements.

[20] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '16)*. Edinburgh, UK.

[21] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG]

[22] CXL Consortium. 2019. . https://www.computeexpresslink.org/

[23] Chen Ding and Ken Kennedy. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel and Distrib. Comput.* 64, 1 (2004), 108–134.

[24] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-Based Alias Analysis. *SIGPLAN Not.* 33, 5 (May 1998), 106–117.

[25] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*. Seattle, WA.

[26] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Monterey, California.

[27] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (ATC '19)*. Renton, WA.

[28] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. 2022. 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.

[29] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA.

[30] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. Boston, MA.

[31] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. 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.

[32] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[33] Hewlett Packard. 2005. The Machine: A New Kind of Computer. http://www.hpl.hp.com/research/systems-research/themachine/.

[34] Hossein Moein 2023. *C++ DataFrame for statistical, Financial, and ML analysis.* Retrieved Sep, 2023 from https://github.com/hosseinmoein/DataFrame

[35] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Rennes, France.

[36] Mahmut Kandemir, Ismail Kadayif, and Ugur Sezer. 2001. Exploiting scratch-pad memory using presburger formulas. In *Proceedings of the 14th international symposium on Systems synthesis (ISSS 2001)*. Montrél, Canada.

[37] Kartik Kannapur 2017. *NYC Taxi Trips - Exploratory Data Analysis.* Retrieved Sep, 2023 from https://www.kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook

[38] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2022. MLIR-Based Code Generation for GPU Tensor Cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC 2022)*. Seoul, South Korea.

[39] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21).*

Virtual.

[40] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. Virtual.

[41] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*. Virtual.

[42] Thomas Kistler and Michael Franz. 2000. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 3 (2000), 490–505.

[43] Sohyang Ko, Seonsoo Jun, Yeonseung Ryu, Ohhoon Kwon, and Kern Koh. 2008. A New Linux Swap System for Flash Memory Storage Devices. In *2008 International Conference on Computational Sciences and Its Applications*.

[44] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*. Virtual.

[45] Junghee Lee, Chanik Park, and Soonhoi Ha. 2003. Memory access pattern analysis and stream cache design for multimedia applications. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference (ASP-DAC 2003)*. Kitakyushu, Japan.

[46] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. 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, Canada.

[47] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA.

[48] Pengcheng Li, Hao Luo, Chen Ding, Ziang Hu, and Handong Ye. 2014. Code layout optimization for defensiveness and politeness in shared cache. In *43rd International Conference on Parallel Processing (ICPP 2014)*. Minneapolis, MN.

[49] Diego Novillo. 2007. Memory SSA—A Unified Approach for Sparsely Representing Memory. In *Proc of the GCC Developers' Summit*.

[50] OpenAI. 2019. GPT-2: 1.5B release. https://openai.com/research/gpt-2-1-5b-release.

[51] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '19)*. Washington DC.

[52] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. *arXiv:1811.09886* (2018).

[53] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems (MLSys '23)*. Miami, FL.

[54] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. 2023. 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.

[55] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro* 30, 4 (2010), 65–79.

[56] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. Banff, Canada.

[57] Mohit Saxena and Michael M. Swift. 2010. FlashVM: Virtual Memory Management on Flash. In *2010 USENIX Annual Technical Conference (ATC '10)*. Boston, MA.

[58] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. 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.

[59] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SOCC '17)*. Santa Clara, CA.

[60] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, Miles McCain, Alex Newhouse, Jason Blazakis, Kris McGuffie, and Jasmine Wang. 2019. Release Strategies and the Social Impacts of Language Models.

[61] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: Profile-Guided Btb Replacement for Data Center Applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. New York, NY.

[62] Alexandru E Şuşu. 2020. A vector-length agnostic compiler for the connex-s accelerator with scratchpad memory. *ACM Transactions on Embedded Computing Systems (TECS)* 19, 6 (2020), 1–30.

[63] Akshitha Sriraman Joseph Devietti Gilles Pokam Heiner Litz Baris Kasikci Tanvir Ahmed Khan, Dexin Zhang. 2021. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA '21)*. Virtual.

[64] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Toronto, Canada.

[65] Shin-Yeh Tsai, Yizhou Shan, , and Yiying Zhang. 2020. 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.

[66] Sumesh Udayakumaran and Rajeev Barua. 2003. 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*. 276–286.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[68] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. 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.

[69] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. 2022. KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing. In *2022 USENIX Annual Technical Conference (ATC '22)*.

[70] Emmett Witchel and Krste Asanovic. 2001. The span cache: Software controlled tag checks and cache line size. In *Workshop on Complexity-Effective Design, 28th ISCA*.

[71] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards Practical Page Coloring-Based Multicore Cache Management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. Nuremberg, Germany.