arXiv:2505.21669v1 [cs.AR] 27 May 2025

NORTHWESTERN UNIVERSITY

Improved Prefetching Techniques for Linked Data Structures

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Computer Engineering

By

Nikola Vuk Maruszewski

EVANSTON, ILLINOIS

June 2025

© Copyright by Nikola Vuk Maruszewski 2025

All Rights Reserved

ABSTRACT

Improved Prefetching Techniques for Linked Data Structures

Nikola Vuk Maruszewski

With ever-increasing main memory stall times, we need novel techniques to reduce effective memory access latencies. Prefetching has been shown to be an effective solution, especially with contiguous data structures that follow the traditional principles of spatial and temporal locality. However, on linked data structures—made up of many nodes linked together with pointers—typical prefetchers struggle, failing to predict accesses as elements are arbitrarily scattered throughout memory and access patters are arbitrarily complex and hence difficult to predict. To remedy these issues, we introduce LINKEY, a novel prefetcher that utilizes hints from the programmer/compiler to cache layout information and accurately prefetch linked data structures. LINKEY obtains substantial performance improvements over a striding baseline. We achieve a geomean 13% reduction in miss rate with a maximum improvement of 58.8%, and a 65.4% geomean increase in accuracy, with many benchmarks improving from 0%. On benchmarks where LINKEY is applicable, we observe a geomean IPC improvement of 1.40%, up to 12.1%.

Acknowledgements

First, I would of course like to thank my advisor, Prof. Nikos Hardavellas. I have had the privilege to work with him for the past three years; his guidance and support has helped me become the researcher I am today. I'd also like to thank the other members of my thesis committee, Prof. Peter Dinda and Prof. Russ Joseph, for taking the time to review this dissertation and attend my defense.

I'd like to thank Karl Hallsby for his help in drafting and reviewing this thesis. Tommy McMichen's advice on data structures was also invaluable for designing benchmarks. Additionally, I'd like to thank Nick Wanninger and Steve Ewald for helping make the benchmarks more representative, and Atmn Patel for his advice on hardware resources. Finally, I must thank Joe Maruszewski for the information on octrees and their traversal patterns in Computational Fluid Dynamics programs.

I would also like to acknowledge my prior collaborators, who have helped me grow as a researcher and hone my skills. A big thank you to Prof. Kate Smith, Jessica Jeng, Michael Gavrincea, and Connor Selna.

Last but not least, I must thank my family for all they have done for me, and their continuing support throughout my academic journey. We are all products of our environment, and I thank my family for encouraging me to push myself and helping me become the person I am today.

List of abbreviations

AT: Address Table

BFQ: Backup Fetch Queue

BFS: Breadth-First Search

BST: Binary Search Tree

CAT: Child Association Table

CDP: Content Directed Prefetcher [**15**]

DFS: Depth-First Search

LDS: Linked Data Structure

Dedication

To my grandparents, Milan and Ljubica Egelja. Without your teachings and support, this thesis would have never been written.

Table of Contents

ABSTRACT 3

Acknowledgements 4

List of abbreviations 5

Dedication 6

Table of Contents 7

List of Tables 10

List of Figures 11

List of Algorithms 12

List of Listings 13

Chapter 1. Introduction 14

1.1. Contributions 17

1.2. Thesis Organization 17

Chapter 2. Background and Motivation 18

2.1. Linked Data Structures 18

2.2. Principle of Locality 20

2.3. Prefetcher Implementations . . . 21
2.3.1. Software Implementations . . . 22
2.3.2. Hardware Implementations . . . 23
2.3.3. Hybrid Implementations . . . 24
2.4. Memory System Trends . . . 25

Chapter 3. Effectively Prefetching Data Structures Linked with Pointers . . . 26
3.1. Principle of Operation . . . 27
3.2. Table-Based Fetching . . . 29
3.2.1. Table Search . . . 30
3.2.2. Table Building . . . 32
3.3. Backup Fetch Queue . . . 35
3.4. Issuing Requests . . . 35

Chapter 4. Experimental Evaluation . . . 39
4.1. Methodology . . . 39
4.1.1. Prefetcher Configuration . . . 40
4.2. Benchmarks . . . 41
4.2.1. Traversal Benchmarks . . . 43
4.2.2. Lookup Benchmarks . . . 44
4.3. Metrics . . . 47
4.4. Results . . . 48
4.4.1. Configuration Evaluation . . . 48
4.4.2. Performance Evaluation . . . 51

Chapter 5. Related Works 55

Chapter 6. Conclusion and Future work 61

6.1. Future Work 62

References 63

Appendix A. Other Contributions 71

A.1. Personal Contributions 72

Vita 73

List of Tables

3.1 Address Table entry 30

3.2 Child Association Table entry 30

4.1 Baseline system configuration parameters 40

4.2 Custom instructions added to interact with Linkey 41

4.3 Benchmarked configurations of Linkey 41

4.4 Benchmarks used to evaluate Linkey 42

List of Figures

2.1 Types of linked data structures 19

3.1 Populated AT and CAT example 31

3.2 Linkey fetch pipeline 36

4.1 Zipfian distribution on $[1, 100]$ 45

4.2 Normalized load misses with different Linkey configurations 49

4.3 Normalized IPC with different Linkey configurations 50

4.4 Normalized numbers of load misses 51

4.5 Prefetch accuracy results 52

4.6 Prefetch hit counts 52

4.7 Normalized IPC results 53

4.8 Normalized IPC results, with certain exclusions 54

List of Algorithms

3.1 AT Table Search 32

3.2 AT/CAT Table Building 34

3.3 Prefetch Request Issuing 38

List of Listings

2.1 Software prefetching example 22

4.1 Octree "W-cycle" traversal kernel 44

CHAPTER 1

Introduction

Effective main memory access times have become the main bottleneck in modern high-performance computing systems. While processor performance has followed Moore's law, doubling every 1.5 years, DRAM speeds have failed to keep up, with access latencies growing to at least **300 cycles** on modern processors [**44**]. As this processor-memory performance gap widens, new developments in processor technology will go unnoticed if memory stall times cannot be reduced. We must develop new techniques to hide this latency if we wish to maximize the performance of our systems.

Prefetching has been shown to be an effective solution for reducing the effective memory access latencies in modern high-performance processors [**54**]. By predicting the next memory access and ensuring the data is in the L1-D$ before the processor issues the corresponding memory operation, prefetchers can, in theory, completely hide the latencies of the L2$ and below. With contiguous data structures, such as arrays or hashtables, this technique has been incredibly effective, even with complex access patterns [**43**].

Unfortunately, modern prefetching techniques struggle when presented with linked data structures (LDSs), such as linked lists, trees, and graphs. As the nodes of these data structures are arbitrarily distributed throughout memory and traversed through pointer chasing, there is no inter-element spatial locality to take advantage of like in contiguous data structures. Additionally, as access patterns to these objects are often unpredictable

(e.g., searching for an arbitrary key in a binary search tree), prefetching techniques that take advantage of temporal locality such as correlation [**27**, **43**, **55**, **62**] often fail as well.

One might wonder, if these data structures are so detrimental to performance, are they really so commonly used to warrant special optimization? Programmers are taught to maximize spatial and temporal locality, if they wish to maximize performance, why would they use a data structure with poor locality? A map can be implemented as either a hashtable or a binary tree, it seems like the hashtable should be the clear choice for speed.

There are conditions where linked data structures are faster than their array-based counterparts, or even required to satisfy the requirements of a program. As they are constructed of nodes connected by pointers, it is rather trivial to make modifications to LDSs, irrespective of the location of the modified element. For example, an insertion into a linked list takes $\mathcal{O}(1)$ time, regardless of whether the location is the start, end, or middle, while a dynamic array only has $\mathcal{O}(1)$ insertion at the end. Linked data structures are also more memory efficient, as only the exact number of nodes needed to hold the data need be allocated, compared to array-based structures where an *exponentially growing* over-allocation is often required for good amortized time complexity. Some LDSs even provide guarantees not found elsewhere, such as binary search trees (BSTs) that offer both fast lookup and ordering.

It is no surprise then that linked data structures are incredibly common across many applications. The Linux kernel [**58**] makes heavy use of (intrusive) linked lists for tracking free objects, along with balanced BSTs for scheduling and I/O. Other types of trees are also used for scientific computing [**31**], database indexes [**14**], fast text search [**1**], and

bioinformatics [**40**]. Finally, sparse graphs, implemented using adjacency lists, are widely used for social networks analysis, web crawling, and biology [**5**].

Clearly, the importance and prevalence of these linked data structures justifies targeted optimization for their specific access patterns. Unfortunately, as stated previously, most prefetching techniques optimize for traditional locality, which often fails when presented with pointer-chasing applications. To be most effective, we must take advantage of *reference locality*—the idea that the set of objects referenced by a given object is typically unchanging [**4**].

Content Directed Prefetchers (CDPs), as introduced in [**15**], attempt to optimize for reference locality. Their distinguishing feature is the use of memory *responses*, in addition to request addresses, to search for possible "recursive" pointers (i.e., pointers to nodes of the same data structure) and issue prefetches. However, as many CDPs strive for a hardware-only implementation [**18**, **37**, **61**], they must speculate if a value is a pointer, leading to cache pollution and even security vulnerabilities [**12**, **60**]. Other implementations rely heavily on software/profiling [**6**, **61**, **68**, **69**] or require substantial hardware resources [**21**, **22**, **23**, **24**, **64**, **65**].

To remedy these issues and improve performance in pointer-chasing applications, we introduce a novel prefetching technique for linked data structures, which we have termed Linkey. We take a hybrid approach with hardware-software co-design to provide high-performance while eliminating speculation on whether a value is a pointer. Metadata specifying basic LDS node layout information is passed down to the hardware, alleviating the need to perform shape analysis in silicon. Linkey is also provided with at least one known memory location in the data structure (e.g., the root), from which it learns the

shape of the entire structure. With these simple bits of information, Linkey can continue on to effectively prefetch complex linked data structures.

1.1. Contributions

Our contributions are as follows:

(1) We introduce Linkey, a novel prefetching technique for linked data structures. With only a tiny amount of information passed down from the software, Linkey can effectively issue prefetch requests for pointer-chasing access patterns. Linkey is also very adaptable, supporting data structures with varying numbers of linking (i.e., recursive) pointers.

(2) We evaluate Linkey against traditional prefetcher technologies on a representative set of LDS benchmarks, showing substantial performance improvements over a striding baseline. We achieve a geomean 13% reduction in miss rate with a maximum improvement of 58.8%, and a 65.4% geomean increase in accuracy, with many benchmarks improving from 0%. On benchmarks where Linkey is applicable, we observe a geomean IPC improvement of 1.40%, up to 12.1%.

1.2. Thesis Organization

The remainder of this thesis is structured as follows: Chapter 2 gives background information on modern prefetching technologies and further motivates the design of Linkey. Chapter 3 delves into the design and implementation of Linkey. Next, Chapter 4 benchmarks Linkey against an array of other prefetching techniques. In Chapter 5, we discuss other related works for prefetching pointer-chasing access patterns. Finally, we conclude and discuss future work in Chapter 6.

CHAPTER 2

Background and Motivation

Linked data structures are a critical component of many algorithms and software programs. However, as they do not consistently follow the traditional principles of temporal and spatial locality, predicting accesses is a major challenge. Prefetchers have attempted to improve accuracy using software, hardware, and hybrid approaches with varying degrees of success and practicality. However, with the current trends in memory system design, even the best of these approaches will still struggle when presented with linked data structures.

2.1. Linked Data Structures

Linked data structures (LDSs) are one of the two main categories of data structures, the other being contiguous data structures. The main difference between them is how elements are found and traversed. A contiguous data structure is allocated as a single large block of memory, which enables fast random access times but means modifications can be slow if a re-allocation is required. Contiguous data structures are also often over-allocated to provide fast amortized time complexities, leading to memory overheads, as the size of the over-allocation must grow *exponentially* for the amortization to work.

On the other hand, LDSs are built from many small nodes **linked** together with pointers. This means each allocation happens individually, and nodes can be arbitrarily scattered throughout memory. This hurts random access times, as one may have to

traverse a long pointer chain to access a node. Modifications, on the other hand, are significantly simpler, as re-allocations and movements of large blocks of memory are not required—only a few pointers must be updated. Additionally, LDSs are space efficient—no over-allocations are *required* for their complexity guarantees, and any over-allocation for low-level performance optimizations (such as a node pool) only needs to grow *linearly*.

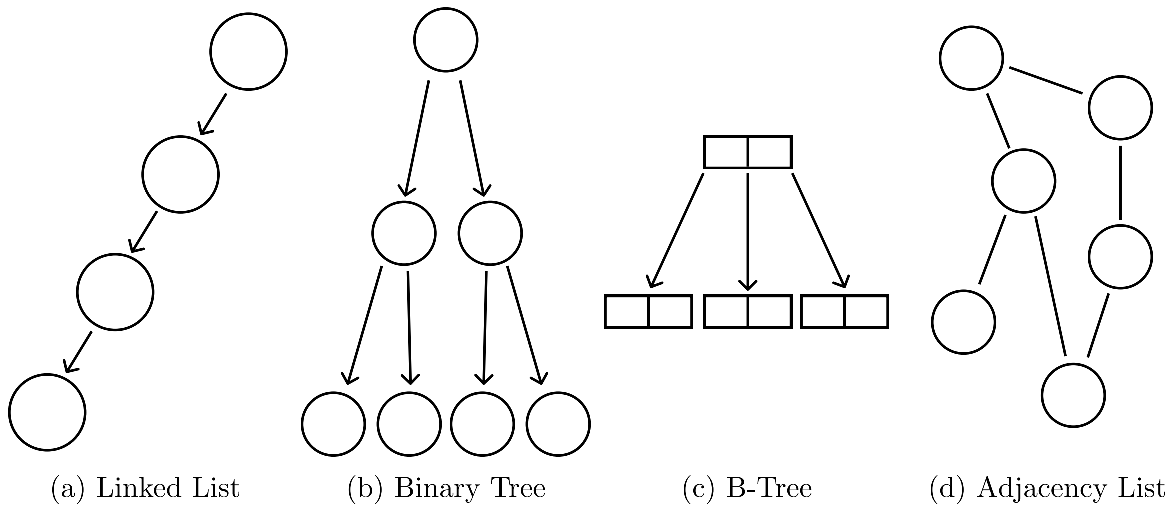

Figure 2.1. Common types of linked data structures. Circles and rectangles represent nodes, the arrows represent pointers (i.e., the links) between nodes in the linked data structure.

Due to these benefits, there are many applications that make use of linked data structures. Memory allocators in glibc and the Linux kernel utilize linked lists (shown in Figure 2.1a) to track free segments of memory. Self-balancing binary search trees (Figure 2.1b), such as red-black trees [**20**], are used in Linux's Completely Fair Scheduler for ordering processes, and in `epoll(7)` for tracking file descriptors [**58**]. Octrees (8-ary trees) have uses in 3D graphics [**42**] and scientific computing applications such as computational fluid dynamics [**31**] that partition 3D space. Additionally, tries (i.e., prefix

trees) [**17**] have been used for fast text search [**1**], web server routing [**45**], and bioinformatics [**40**]. In database systems, B-Trees (Figure 2.1c) and their variations have been heavily used, mainly for the implementation of table indexes [**14**]. Finally, sparse graphs are most often implemented using adjacency lists (Figure 2.1d), with applications in social networks analysis, web crawling, and biology [**5**].

In many of these applications, an LDS is used simply because there is no other alternative. The Linux kernel scheduler must have predictable runtime, thus a hashmap or array-based binary heap that may need to perform a lengthy resize would be unsuitable. Database indexes require a data structure that supports both point and range lookups, a feature unique to the B-Tree family; the use of two separate indexes for each feature would be untenable due to storage overheads. Most of all, the incredibly large sparse graphs used in social network analysis must be implemented using an adjacency list—any other implementation would consume orders of magnitude more memory. Clearly, LDSs are important enough to warrant targeted optimizations.

2.2. Principle of Locality

Code executed on modern CPUs tends to follow the principles of spatial and temporal locality: nearby objects are often accessed together, and frequently accessed objects will likely be accessed again in the future. Modern systems stacks optimize on this principle, and thus programmers optimize code by improving locality, creating a virtuous cycle. Prefetchers notably employ locality to perform shape analysis; they speculate on the layout of an object in memory using it and issue requests accordingly. This technique is often extremely effective, as contiguous data structures are very common (vectors,

hashmaps, etc.) and some pointer-based access patters are frequently repeated (e.g., virtual function calls).

Unfortunately, linked data structures break this paradigm. As each node is typically allocated individually, the user has no control over its placement in memory; this arbitrary distribution of nodes means any spatial locality-based prefetcher struggles when presented with an LDS, as the location of one node gives no indication of the location of others. While there is some temporal locality in access patterns, and certain nodes can be hotter than others depending on the type of traversal, it is still just an approximation. A prefetcher for linked data structures should instead consider **reference locality**: the principle that the set of nodes *referenced* by a given node tends not to change. This implies child pointers of a node are relatively constant, which also explains why temporal locality can sometimes serve as a proxy for reference locality when access patterns are crystalline (i.e., not noisy) and repetitive.

2.3. Prefetcher Implementations

Prefetchers appeared in research as early as 1982 [**54**], and have grown more and more complex since. Implementations have been done in either software or hardware, primarily focusing on the memory operation address stream. Only recently have new designs bridged the gap between software and hardware, or started using memory *responses* to inform future prefetches.

```
void vec_work(int arr*, size_t N) {
    for (int i = 0; i < N; ++i) {
        prefetch(arr[i + 1]);
        work(arr[i]);
    }
}

void ll_work(node_t* n) {
    for (node_t* cur = n; cur != NULL;) {
        prefetch(cur->next);
        work(cur->data);
        cur = cur->next;
    }
}
```

Listing 2.1. Software prefetching example for vectors and linked lists.

2.3.1. Software Implementations

A common method of implementing prefetching is by using a compiler to insert speculative load (i.e., prefetch) instructions into a program. When a processor hits these instructions, it will issue a load, but suppress any errors due to memory protection violations or invalid loads. The result of the load is also dropped, as the only purpose is to bring data into the cache. From a hardware perspective, this is trivial to implement—only a slightly modified load instruction is needed—and it allows prefetching schemes to be tailored to the program executed. An example of software prefetching for linked lists and vectors is shown in Listing 2.1.

Unfortunately, the separation of the compiler and runtime leads to issues with pure software implementations. As the compiler is not fully aware of microarchitecural details and system configurations, such as memory speeds, it must use an (inherently flawed) cost model to determine prefetch depth and scheduling. Additionally, when prefetching linked data structures, the compiler must either rely on alias/shape analysis or profiling

to decide what to prefetch [**4**, **61**, **69**]; the former is intractable as it maps to the halting problem [**48**], while to be accurate the latter requires coverage of all program inputs and/or extremely low overheads to run online, which is very difficult to accomplish in practice. Finally, the insertion of additional instructions into hot loops will raise code size, increasing L1-I$ pressure and hurting performance.

2.3.2. Hardware Implementations

Hardware has been the primary implementation of prefetching algorithms for many years. The simplest "Next-K" prefetchers bring in the next K lines after the current miss, while striding prefetchers fetch the n-th next cache line after the current one. Both strategies optimize for contiguous, sequential traversals such as vectors [**43**, **54**]. More advanced correlation-based prefetchers [**27**, **43**, **55**, **62**] observe repeated patterns of accesses at a constant offset (e.g., A, $A+8$, B, $B+8$, C, $C+8$); such designs have seen greater success with linked data structures and other pointer-chasing applications. Even more complex designs have turned to implementing entire neural networks in hardware, which are in theory adaptable across many applications [**47**].

Hardware prefetcher designs have been previously created with pointer-chasing access patterns in mind. Some techniques compute and store jump pointers during accesses to an LDS, allowing "long-distance" prefetches without traversing a pointer chain [**38**, **51**]. Other designs extract a *prefetch kernel* from the application and schedule it to run on a custom prefetch engine [**24**, **64**, **65**], a second SMT thread [**21**, **22**, **23**], or insert it directly into the application [**52**]. Techniques for improving the effective latency of indirect memory accesses use pointer caches for load address prediction [**13**] or track memory

access patterns to predict the indirection function and issue prefetches accordingly [**63**, **67**]. Finally, more advanced hardware-only designs attempt to infer an LDS's semantic structure (i.e., shape) from the memory request address stream [**37**].

2.3.2.1. Content Directed Prefetchers. This new class of prefetchers, introduced in [**15**], not only examines the address stream from the processor, but also the **responses** from the memory system. Cooskey et al. specifically attempt to find pointers in cache lines returned from memory and issue prefetches for those addresses, in the hopes that they are child pointers in an LDS. This technique was further enhanced in [**18**] to attempt to reduce prefetch misses. Apple also recently used this technique in their M-series of processors; unfortunately, it was shown in [**12**] and [**60**] that speculating if a value is a pointer opens up many security vulnerabilities. This is in addition to cache pollution caused by the inability to identify if a pointer is referencing a child node or simply points to other data that won't be needed in the traversal (such as parent nodes, metadata, etc.).

2.3.3. Hybrid Implementations

Recent prefetcher designs have attempted to cross the boundary between hardware and software for further optimization opportunities, combining high-level information easily determined by the compiler, such as object sizes and types, with the low-level runtime information readily available to the CPU, such as pointer values. Some of these techniques try to inform the hardware about every object pointer in the application [**6**] or build complex graphs to communicate program semantics to hardware [**68**]. Many rely on some sort of profiling that is then passed down to the hardware to help it issue prefetches [**61**, **69**], but as mentioned previously, representative profiling is extremely difficult to accomplish

in practice. A technique such as Compiler-Directed Content-Aware Prefetching (CDCAP, described in [**3**]) is likely most practical, as only minimal yet readily-available static information (LDS node layout) is passed down to hardware.

2.4. Memory System Trends

Recent developments in memory technology have seen R&D budgets utilized more and more towards improving bandwidth, rather than latency, in the hopes of mitigating the off-chip memory bandwidth wall [**49**]. Examples include signaling improvements such as three-level (-1, 0, +1) pulse-amplitude modulation (PAM-3) as used in GDDR7 to transfer 1.5 bits of data per cycle [**66**], double data rate interfaces that push data on both edges of a clock [**28**], and novel quad data rate systems that use two double-pumped clocks at a 90 degree phase offset to transfer **four** signals per cycle [**26**]. To increase bandwidth, some DRAM chips even utilize additional dimensions of signaling—HBM3 memory, for example, uses 3D-stacked memory dies connected with a silicon interposer to increase link density [**35**, **46**]. This increase in bandwidth reduces the cost of prefetching, as the likelihood of a prefetch causing congestion decreases, and with memory stall times ever-increasing, more aggressive data prefetchers may be the only solution for avoiding memory stalls.

CHAPTER 3

Effectively Prefetching Data Structures Linked with Pointers

We first begin by identifying the following requirements for any effective prefetcher of linked data structures:

(1) Get **ahead** of the application, ensuring prefetches are timely (i.e., issued early enough, but not too early [**37**]) so the data is on-chip (ideally in the L1-D$) when the application needs it. This is especially vital as memory stalls increase.

(2) **Stay** ahead as execution progresses, continuing to issue prefetches to travel down the LDS by utilizing child pointers found in memory responses.

(3) Be **accurate** with prefetch requests to avoid cache pollution.

Linkey uses three hardware structures to accomplish these goals. To *get ahead* of the application and ensure prefetches are timely, we add an **Address Table (AT)** and **Child Association Table (CAT)** that allows us to store the locations and associations between the most important nodes of the data structure. When accesses to roots are detected, these tables allow us to fetch the majority of likely subsequent accesses in parallel. Then, to *stay ahead* of the application, we add a **Backup Fetch Queue (BFQ)** to continue fetching after exhausting the data in the tables. We populate this queue using pointers found in memory responses and draw from it when additional memory bandwidth is available for prefetching, e.g., when addresses miss in the tables.

To ensure prefetches are accurate, LINKEY uses metadata provided by the software to only issue prefetches for useful linking pointers. This allows the user/compiler to control which linking pointers should be prefetched for a given traversal. While LINKEY cannot predict the next node to be accessed, it can accurately predict the *set* of nodes that will follow the current access, hence why it issues many prefetches at once. The other nodes in the prefetched set are also not useless—there is a (often strong) chance they will be needed soon. Our accuracy and hit count results in Chapter 4 reinforce this hypothesis.

A high-level overview of LINKEY's operation follows, along with further details on the organization and maintenance of the AT, CAT, and BFQ.

3.1. Principle of Operation

We chose a hybrid prefetcher design for LINKEY. Purely software prefetching approaches are intractable, as low-level information available to the runtime (e.g., pointer values) cannot be statically determined by the compiler. While hardware-only approaches are possible, as evidenced by Intel's CDP [**15**], the lack of high-level information leads to excessive prefetches and cache pollution. Thus, we choose a hybrid approach to utilize the strengths of both the software and the hardware. Specifically, LINKEY obtains node layout information from the user/compiler and melds it with pointer values available at runtime to effectively prefetch linked data structures.

A few basic assumptions are also made. We hypothesize that the majority of applications will follow these assumptions, and thus use them as optimization opportunities:

(1) Linked data structures have high reference locality, i.e., child pointers of nodes do not change often.

(2) The majority of traversals of a linked data structure start from a small set of nodes (the "roots"). This implies a skewed distribution of node "temperatures," with the roots and nodes near them very hot, while the rest are cold.

(3) A traversal starts when a root is accessed following an access to a different node.

(4) Traversals tend to access non-pointer fields of a node in the same order.

With these assumptions in mind, the LINKEY prefetcher is provided with a few small pieces of information: the size of a node in the LDS (*NodeSize*), the offsets of linking pointers to other nodes from the start of a node (*ChildOs*), and the addresses of the "root" nodes (stored in the AT and identified with table pointers in a *Roots* register). This information can be provided in many ways; possible implementations include custom instructions, MMIOs, I/O ports, and customized loads/stores. However, the implementation should ideally be serializing with respect to memory, so the prefetcher configuration finishes before memory requests are issued. The serialization can be implemented on the software side with a fence, and will not impact performance as the configuration must only be done once, before any hot loops.

Upon receiving a memory request from the core, LINKEY searches the AT for LDS nodes the request address may correspond to. If the address hits, LINKEY uses the AT and CAT to issue fetches for the children of the hit node. A check is then performed to see if a new traversal has started; this can also be explicitly indicated with a marker, but doing so is strictly optional. To take advantage of the increased bandwidth available in modern memory systems, as described in Section 2.4, we allow the prefetcher to output multiple requests per invocation. If the AT/CAT search does not allow the prefetcher to fill its request buffer, further addresses are drawn from the BFQ.

When responses arrive from the memory subsystem, the prefetcher is notified and allowed to read the data. If the cache block returned from memory corresponds to any of the addresses in the AT (determined using *NodeSize*), the values of all child pointers in the block (found using *ChildOs*) are extracted and inserted into the AT, then associated with the AT hit using the CAT. By using application-provided metadata, we ensure that only pointers known to be useful are added to the tables. Finally, we use a pseudo-LRU eviction system to allocate entries in both tables.

Memory responses are also used to build the BFQ, similar to Intel's CDP system [**15**]. When Linkey issues a prefetch, it adds a small amount of metadata to locate the node within the fetched cache block. When the response arrives, the Linkey prefetcher uses this metadata to push all child pointers found in the memory response into the BFQ, excluding those already present in the AT. Then, by pulling from the BFQ when memory bandwidth is available, the prefetcher "stays ahead" of the application once it exhausts the entries stored in the AT and CAT.

3.2. Table-Based Fetching

We use the AT to keep track of known nodes of the LDS. Each AT entry stores a virtual address corresponding to the start of an LDS node, in addition to a valid bit, two LRU bits (described in Section 3.2.2.1), and the indexes of any child entries in the CAT (each index with its own valid bit). The CAT associates entries of the AT in a parent/child relationship, with each CAT entry containing a parent and child index, in addition to a valid bit, two LRU bits, and a number specifying which child pointer the entry represents. As nodes of an LDS contain pointers, the nodes will be aligned to the

size of a pointer (8 bytes on modern systems), thus, we can elide the bottom 3 bits of the address field in the AT. The AT and CAT organization is further detailed in Table 3.1 and Table 3.2, and a populated example of the tables is shown in Figure 3.1.

Valid (1)	LRU (2)	Address	Child 0 Idx/Valid	...	Child N Idx/Valid

Table 3.1. Address Table entry. Bitwidths are shown in parentheses. Address bitwidth depends on machine configuration. Child index bitwidth depends on the number of CAT entries (i.e., $\log_2(|CAT|)$). N is equal to $|ChildOs| - 1$.

Valid (1)	LRU (2)	Parent Idx	Child Idx	Offset Idx

Table 3.2. Child Association Table entry. Bitwidths are shown in parentheses. Parent/child index bitwidth depends on the number of AT entries (i.e., $\log_2(|AT|)$). Offset index bitwidth depends on the number of child pointer offsets (i.e., $\log_2(|ChildOs|)$.

3.2.1. Table Search

When the Linkey prefetcher receives an address from the core, two searches happen in parallel: a root check to identify new traversals, and a full table scan to detect continuing traversals. For the first check, the roots (of which there are a small number) are searched using a base and bound check: $RootAddr_i \leq Addr < RootAddr_i + NodeSize$. If this check succeeds, we keep the index of the hit and compute the offset of the request address from the start of the root that hit (i.e., $Addr - RootAddr_i$). We assume this offset (termed *KeyO*) is the offset of the key used for the traversal (e.g., in a binary tree lookup; see the 4th assumption above) and update it when a new traversal begins. If the root check fails, Linkey uses the results of the second check, which is a content-addressable memory (CAM) lookup along the address field of the table, searching for $Addr - KeyO$. The

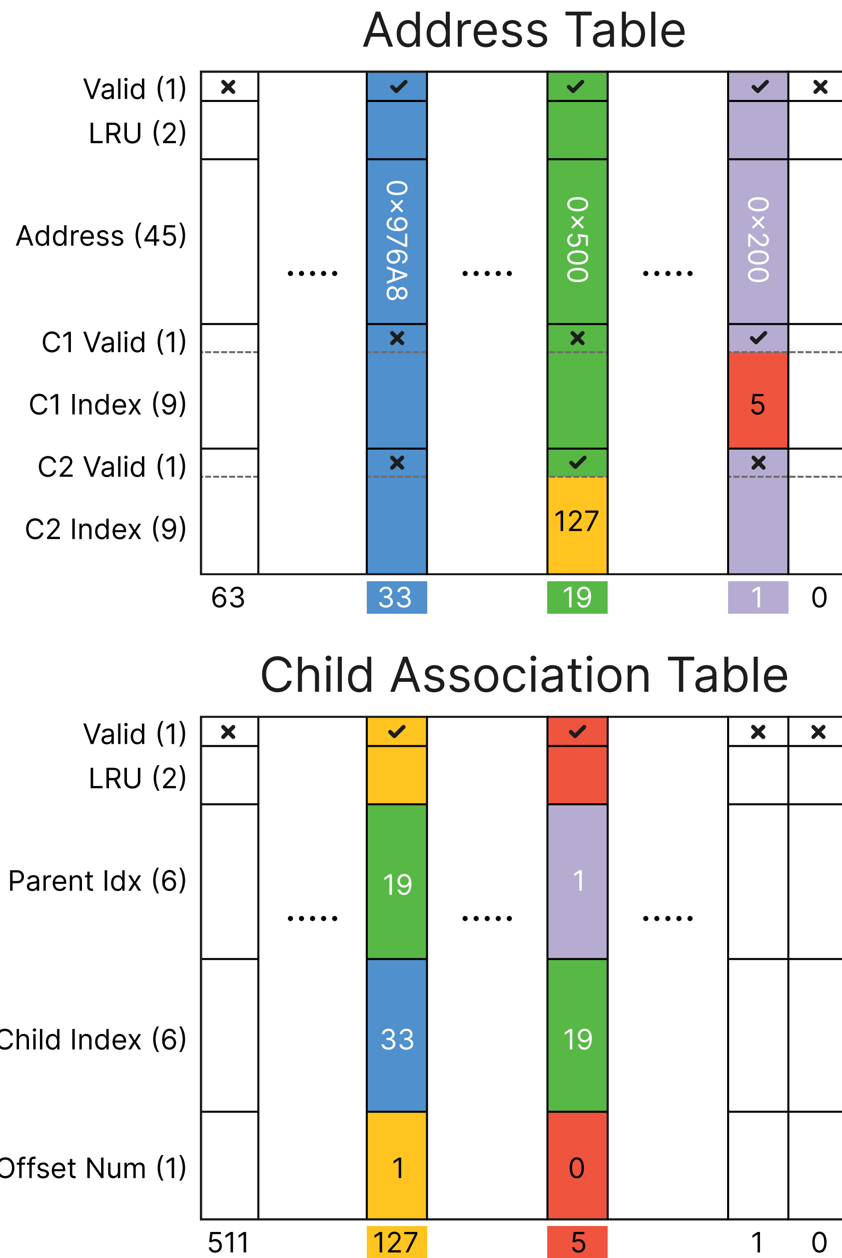

Figure 3.1. Populated example of the Address and Child Association Tables for a system with 48-bit virtual addresses. The AT holds 64 entries and supports up to two CAT pointers. The CAT holds 512 elements. Note that entries are displayed vertically. The numbers on the bottom of the table represent entry indexes. Colors are used to highlight specific entries.

use of *KeyO* allows us to avoid the base-and-bound check when searching non-root entries without losing performance. A pseudocode description of this process can be found in Algorithm 3.1.

Algorithm 3.1 AT Table Search

```
1: ▷ KeyO is the key offset from the start of the node. ◁
2: procedure SearchRoots(Addr, NodeSize, KeyO)
3:     for Root ∈ ValidRoots(AT) do
4:         RA ← Root.Address
5:         if RA ≤ Addr < RA + NodeSize then
6:             if a new traversal is starting then        ▷ See Section 3.1 for definition
7:                 Update KeyO to Addr − RA
8:             return Index(Root)
9:     return ∅
10:
11: procedure CAMTable(Addr, KeyO)
12:     for Entry ∈ ValidNonRoots(AT) do
13:         if Entry.Address = (Addr − KeyO) then
14:             return Index(Entry)
15:     return ∅
16:
17: procedure SearchAT(Addr, NodeSize, KeyO)        ▷ Entrypoint
18:     I ← SearchRoots(Addr, NodeSize, KeyO)
19:     if I = ∅ then
20:         I ← CAMTable(Addr, KeyO)
21:     return I
```

3.2.2. Table Building

Table building occurs upon receiving a memory response for any load, in parallel with the insertion into the cache, or after a store completes. This process runs independently of the table search, and thus can be completely asynchronous. First, a base and bound check is applied to the entire AT, to see which objects might lie in the relevant cache block. For every AT hit, the provided child pointer offsets are used to check if a child pointer may lie

in the cache block. If Linkey finds child pointers with non-`NULL` values within the cache block, it adds them to the AT (if not already present) and allocates new CAT entries to associate the parent AT entry with the child AT entries. If a CAT pointer was already present in the parent AT entry, we first invalidate its corresponding CAT entry. Finally, Linkey stores pointers to the newly created CAT entries in the parent AT entry. This process is detailed in Algorithm 3.2.

3.2.2.1. Evictions. We use a modified version of pseudo-LRU to allocate entries in the AT and CAT. Two bits are added to each entry: the conventional `UsedLRU` bit and a `JustBuilt` bit. When a table search hits in the AT, we set the `UsedLRU` bit on the AT entry that hit and any CAT entries that have the AT entry as a parent. The bit is unset when all entries have their `UsedLRU` bit set. On the other hand, we set the `JustBuilt` bit during table building, right as we add an entry to the table. Linkey unsets all `JustBuilt` bits when a new traversal begins (as described in Section 3.1). Having either LRU bit set prevents an entry from being evicted; we add the `JustBuilt` bit to ensure entries aren't evicted before they can be used. Furthermore, a root is **never** a valid eviction candidate, and neither is the *Parent* entry from the build algorithm (Algorithm 3.2, line 10). Note that this may mean sometimes there are no eviction candidates, in which case we skip insertion. Once an eviction candidate is chosen, Linkey invalidates it (as detailed in Section 3.2.2.2) and then overwrites it.

3.2.2.2. Invalidations. Invalidating a CAT entry is very simple: we mark the corresponding CAT pointer as invalid in the parent AT entry (using the parent index and offset number), and then unset the `Valid` bit of the CAT entry. To invalidate an AT

Algorithm 3.2 AT/CAT Table Building

```
1:  procedure SearchBaseBound(BlockAddr, NodeSize)
2:      R ← {}
3:      for Entry ∈ AT do
4:          SA ← CacheLine(Entry.Address)
5:          EA ← CacheLine(Entry.Address + NodeSize)
6:          if SA ≤ BlockAddr ≤ EA then
7:              Add Index(Entry) to R
8:      return R
9:
10: procedure BuildTableForEntry(Parent, BlockAddr, ChildOs)
11:     for o ∈ ChildOs do
12:         P ← Parent.Address + o
13:         if CacheLine(P) = BlockAddr then
14:             C ← *P                                              ▷ Data is in cache
15:             if Parent.Children[o].Valid then
16:                 ▷ See Section 3.2.2.2 for details on invalidations.  ◁
17:                 InvalidateCATEntry(Parent.Children[o].Index)
18:
19:             ▷ See Section 3.2.2.1 for details on evictions.  ◁
20:             if C ≠ NULL and AT and CAT have room after evictions then
21:                 Child ← AddOrFindATEntry(C)
22:                 CATIdx ← AddCATEntry(Parent, Child, o)
23:
24:                 Parent.Children[o].Valid ← true
25:                 Parent.Children[o].Index ← CATIdx
26:
27: procedure BuildTable(BlockAddr, ChildOs, NodeSize)                ▷ Entrypoint
28:     ▷ Called after a store or when a memory response is received.  ◁
29:     R ← SearchBaseBound(BlockAddr, NodeSize)
30:     for Entry ∈ R do
31:         BuildTableForEntry(Entry, BlockAddr, ChildOs)
```

entry, we invalidate all CAT entries that have that AT entry as either a parent or child (using the CAT invalidation process), then set the `Valid` bit of the AT entry to 0.

3.3. Backup Fetch Queue

The BFQ is also built asynchronously during the table building phase. When Linkey issues a prefetch, it attaches metadata specifying the offset of the start of the object from the beginning of the requested cache block. If this metadata is present when a block returns from memory, Linkey identifies child pointers with non-`NULL` values in the cache block using the metadata and *ChildOs*. The node specified by the metadata is also looked up in the AT; this search can be piggybacked off of the table building search in Section 3.2.2. If the node is not present in the AT, or is present and a child found in the cache block isn't present (which can be checked using the child indexes), Linkey adds that child pointer to the BFQ. This last check is to prevent the BFQ from filling with addresses that would already be fetched by the AT and CAT; we assume that a node isn't in the AT if the parent node from the memory response isn't, as the check would otherwise be too expensive.

The prefetch metadata could also be used to avoid the base-and-bound search described in Section 3.2.2 in the majority of cases, however, as the table building runs asynchronously the decision to do so is an implementation detail.

3.4. Issuing Requests

When we prefetch a node, we wish to ensure that the most important portions of the node are brought into the cache. This is extremely important as we allow nodes to span multiple cache blocks, and thus we wish to ensure that only the portions of the node needed for the current traversal are prefetched. To accomplish this goal, we issue prefetches for $\mathit{Node.Start} + \mathit{KeyO}$ and $\mathit{Node.Start} + o$, for all $o \in \mathit{ChildOs}$; we know these values will

are necessary for the traversal, given our assumptions and the metadata provided by the user. We deduplicate prefetches to ensure the same cache block isn't added to the request buffer twice, and we also prevent prefetches to the cache block specified by the core's demand request.

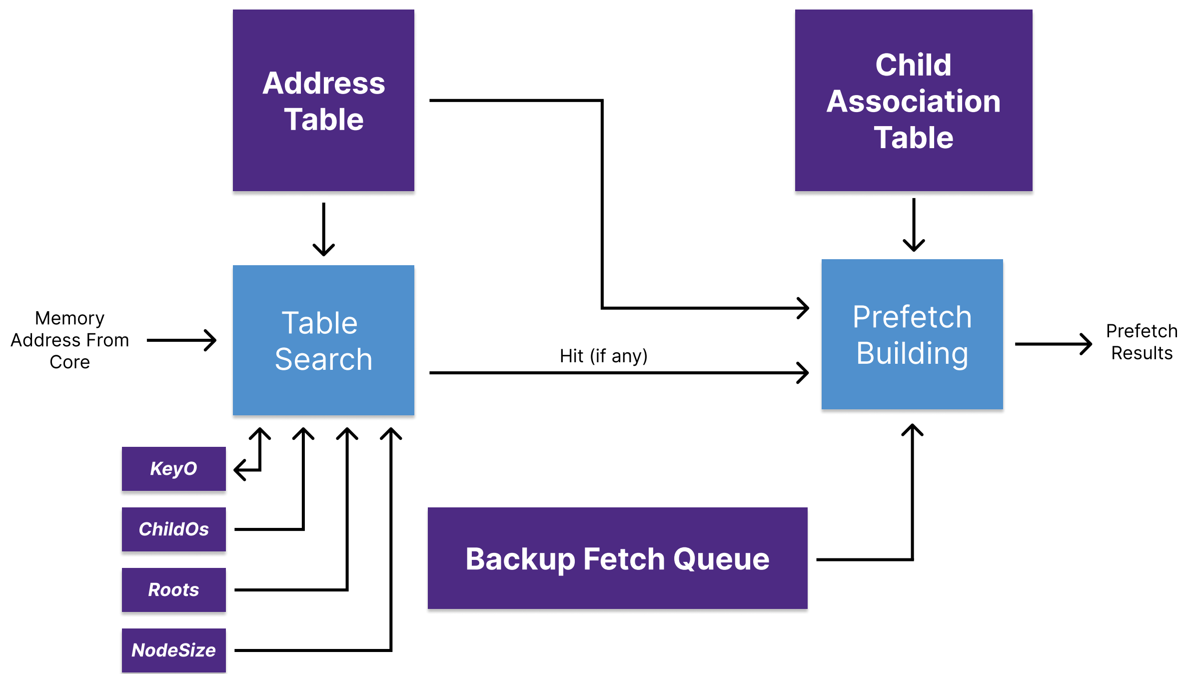

Figure 3.2. The LINKEY fetch pipeline. This sits on the critical path and runs in parallel to the core's L1-D$ access.

The full fetch pipeline is displayed in Figure 3.2. First, LINKEY performs the table search described in Section 3.2.1. If it finds an AT index matching the address, LINKEY adds the index to an internal queue. LINKEY then extracts the index from the same queue and issues prefetches for the specified AT entry as described above. The CAT pointers in the AT entry are also used to add the indexes of its child AT entries to the queue. This process continues iteratively, fetching the remainder of the current node and as many

child nodes as possible, within the size limit of the output buffer (which we capped at 8 entries). This process allows the prefetcher to "get ahead" of the application by fetching, in parallel, the cached shape of the most important nodes of the LDS.

If the output buffer is not full, LINKEY draws addresses from the BFQ. Prefetches are similarly issued to the $KeyO$ offset and each of the $ChildOs$. As the BFQ is filled using memory responses, employing the BFQ allows the prefetcher to "stay ahead" of the application, fetching nodes in the LDS that are not already present in the AT/CAT. This entire fetch process is shown in Algorithm 3.3.

Algorithm 3.3 Prefetch Request Issuing

```
1: ▷ CoreBlock is the core memory request's block address ◁
2: procedure IssueRequest(Addr, ObjectAddr, CoreBlock)
3:     CA ← CacheLine(Addr)
4:     if CA ≠ CoreBlock and CA has not yet been fetched then
5:         O ← ObjectAddr − CA
6:         Add (CA, O) to the request output buffer.
7:
8: procedure PrefetchObject(BaseAddr, ChildOs, KeyO, CoreBlock)
9:     IssueRequest(BaseAddr + KeyO, BaseAddr, CoreBlock)
10:    for o ∈ ChildOs do
11:        if request buffer is not full then
12:            IssueRequest(BaseAddr + o, BaseAddr, CoreBlock)
13:
14: procedure IssueTableFetches(EI, CoreBlock, ChildOs, KeyO)
15:    Initialize an empty queue Q
16:    Add EI to Q
17:
18:    while Q is not empty and request buffer is not full do
19:        Pop head of Q into I
20:        if I has not yet been seen then
21:            PrefetchObject(AT[I].Address, ChildOs, KeyO, CoreBlock)
22:
23:            for c ∈ AT[I].Children do
24:                if c.Valid then
25:                    CI ← CAT[c.Index].ChildIdx
26:                    Add CI to Q
27:
28: procedure HandleCoreReq(Addr, ChildOs, KeyO, NodeSize)        ▷ Entrypoint
29:    EI ← SearchAT(Addr, NodeSize, KeyO)
30:    CoreBlock ← CacheLine(Addr)
31:    if EI ≠ ∅ then
32:        IssueTableFetches(EI, CoreBlock, ChildOs, KeyO)
33:
34:    while request buffer is not full do
35:        Pop head of BFQ into A
36:        PrefetchObject(A, ChildOs, KeyO, CoreBlock)
```

CHAPTER 4

Experimental Evaluation

We evaluate LINKEY across a wide range of common linked data structures with representative access patterns. We use a modern system configuration as our baseline, and compare performance on key metrics. Finally, we discuss patterns and trends that emerge in the gathered data.

4.1. Methodology

We characterize the performance of LINKEY with the Sniper Multicore Simulator [**8**,**9**], although we only consider single-core performance to simulate the worst-case scenario for LDS applications (i.e., no interleaving of memory requests between threads). We modify the Sniper cache controller to notify the prefetcher when memory responses are returned. Prefetch requests are issued as non-exclusive reads; this is because coherency requests are significantly faster than DRAM as they stay on-chip. Finally, to model systems with higher memory bandwidth, as described in Section 2.4, we modify the cache controller to issue two prefetch requests at a time and set the prefetch output buffer is set to a size of eight elements.

We configure the simulator to be representative of a modern high-performance x86-64 system. We adapted simulator configuration parameters from [**11**]; the values are shown in Table 4.1. Virtual and physical address widths were taken from a modern processor

(AMD EPYC 7443P). We refer to the baseline L1-D$ striding prefetcher as `pre_simple` in our figures.

ISA	x86-64, 48-bit virtual and physical addresses
Instruction latency (int)	add(1c.), mul(4c.), div(12c.)
L1-I$	32 KiB, 8-way, 2-cycle latency
L1-D$	48 KiB, 12-way, 5-cycle latency, 64 MSHRs
L1-D$ Prefetcher	Striding, does not cross pages
L2$	1 MiB, 16-way, 16-cycle round trip
L3$	64 MiB, 16-way, 34-cycle round trip
Cache Replacement Policy	LRU for all
DRAM	160-cycle latency

Table 4.1. Baseline system configuration parameters. We refer to the baseline L1-D$ striding prefetcher as `pre_simple` in our figures.

4.1.1. Prefetcher Configuration

To provide basic LDS node layout information to LINKEY, as described in Section 3.1, we choose to use custom instructions inserted into the program. The added instructions are shown in Table 4.2. To enforce serialization, we insert a `mfence` into the code after these instructions. However, as the prefetcher configuration only runs once, these instructions are executed outside the hot loop (i.e., the region of interest) of the benchmarks. Additionally, simulator magic instructions are used to view memory response data, as Sniper does not keep track of memory responses; this is purely for simulation within Sniper and would not be needed in a real hardware implementation.

We limit the maximum node size (i.e., the distance from the start of the node to the end of the last child pointer) to 4 KiB, giving the *NodeSize*, *KeyO*, and *ChildOs* registers a width of 12 bits. We configure LINKEY to support up to eight child pointers, thus

Instruction	Description
`lds.reset`	Reset the LDS prefetcher and clear all tables.
`lds.set_root`	Inform the prefetcher of the address of the ith "root" node.
`lds.clear_roots`	Clear the prefetcher's list of "root" nodes.
`lds.add_offset`	Inform the prefetcher of a child pointer in the LDS node.
`lds.set_size`	Inform the prefetcher of the size of an LDS node (*NodeSize*).
`lds.new_traversal`	Inform the prefetcher a new traversal is about to begin, to reset *KeyO* (optional).

Table 4.2. Custom instructions added to interact with LINKEY.

ChildOs holds eight 12-bit entries. Finally, we allow up to four roots, meaning we must add four registers holding pointers into the AT with corresponding valid bits.

As shown in Table 4.1, we assume 48-bit virtual addresses. Thus, we use 45-bit addresses in the AT and BFQ—the lower three bits can be dropped due to structure alignment (see Section 3.2). The BFQ is limited to eight entries. For the AT and CAT, we test several sizes; the configurations are shown in Table 4.3.

Config Name	AT Entries	CAT Entries	HW Size (Bytes)
`pre_lds_a64_c256`	64	256	1599.5
`pre_lds_a256_c1024`	256	1024	7232.5
`pre_lds_a1024_c4096`	1024	4096	32 833.5

Table 4.3. Benchmarked configurations of LINKEY. The size values are totals, including the AT, CAT, BFQ, and all registers (*ChildOs*, *NodeSize*, *KeyO*, and *Roots*).

4.2. Benchmarks

We chose our benchmarks to represent a wide range of linked data structures with access patterns representative of a variety of applications. Broadly-speaking, benchmarks can be divided into two categories: *traversal* benchmarks that visit each node in an LDS

at least once, and *lookup* benchmarks that search for a certain value in an LDS with the minimum number of node accesses. Additionally, benchmarks can be split into static and dynamic categories, depending on if the LDS is modified during the traversal/search kernel. Descriptions and categorizations of the benchmarks can be found in Table 4.4.

Name	**Description**	**Type**	**Dynamic**
`ll`	Sum of a linked list.	Traversal	×
`ll_reverse`	Reversal and then sum of a linked list.	Traversal	✓
`dll`	Sum of a doubly linked list.	Traversal	×
`bintree_dfs`	DFS sum of a binary tree.	Traversal	×
`bintree_bfs`	BFS sum of a binary tree.	Traversal	×
`bintree_probe_uni`	Probe of a balanced binary search tree with uniformly-distributed random keys.	Lookup	×
`bintree_probe_zipf`	Probe of a balanced binary search tree with Zipfian-distributed random keys.	Lookup	×
`rbtree_uni`	Probe of a Red-Black Tree with uniformly-distributed random keys.	Lookup	✓
`rbtree_zipf`	Probe of a Red-Black Tree with Zipfian-distributed random keys.	Lookup	✓
`splay_uni`	Probe of a Splay Tree with uniformly-distributed random keys.	Lookup	✓
`splay_zipf`	Probe of a Splay Tree with Zipfian-distributed random keys.	Lookup	✓
`trie_uni`	Probe of a trie with uniformly-distributed random keys.	Lookup	×
`trie_zipf`	Probe of a trie with Zipfian-distributed random keys.	Lookup	×
`octree`	Traversal of an octree in a W-cycle.	Traversal	×
`graph_bfs`	BFS sum of a graph	Traversal	×

Table 4.4. Benchmarks used to evaluate LINKEY.

We run all benchmarks with several input sizes. The *small* size uses ≈1000 nodes, the *large* size uses ≈10 000 nodes, and the *huge* size uses ≈100 000 nodes. We use a random memory pool to allocate LDS nodes; this is to be representative of real-world applications, as at any point but a cold start, the application's node pool will be arbitrarily ordered.

The entire benchmark suite is written in C++ and compiled with GCC 7.3.1 at `-O3` with inlining disabled.

4.2.1. Traversal Benchmarks

Approximately half (7 out of 15) of the benchmarks ran are traversal-based benchmarks. These perform common traversals (BFS, DFS, etc.) on linked data structures. As these benchmarks visit each node only one or two times, benefits from LINKEY will likely be somewhat limited, as a central assumption of our design is that the distribution of node temperatures is skewed. However, as we are aware of the structure of the LDS, we do still expect to see some improvement.

4.2.1.1. Linked-List. These benchmarks (`ll`, `ll_reverse`, and `dll`) all traverse a type of linked list through its entirety, computing a sum along the way. The `ll_reverse` benchmark reverses the linked list between summations to evaluate effectiveness on changing data structures. The `dll` benchmark performs both a forwards and backwards traversal, evaluating effectiveness on data structures with multiple roots and pointer cycles.

4.2.1.2. Binary Tree. These two benchmarks (`bintree_dfs` and `bintree_bfs`) perform common traversals of binary tree data structures to compute a sum. The binary tree used consists of d full levels, meaning it is also perfectly balanced. The DFS benchmark is implemented recursively, while the BFS benchmark uses a `std::queue` from the C++ STL.

4.2.1.3. Octree. The `octree` benchmark performs a W-cycle traversal on an octree (8-ary tree). This type of traversal is commonly used in Computational Fluid Dynamics algorithms [**31**,**41**] to iteratively step a problem on a coarser/finer mesh, and an octree is

```
1 long sum_tree(node_t* tree) {
2     if (tree == NULL)
3         return 0;
4 
5     // Start with tree value
6     long sum = tree->value;
7 
8     // Sum children twice (W cycle)
9     for (int pass = 0; pass < 2; ++pass) {
10         for (int i = 0; i < 8; ++i)
11             sum += sum_tree(tree->children[i]);
12     }
13 
14     return sum;
15 }
```

Listing 4.1. Code to perform the "W-cycle" traversal on an octree. Note the double recursion pass.

specifically used to represent 3-D space. The traversal pattern looks like a recursive "W", hence the name. The benchmark kernel is shown in Listing 4.1.

4.2.1.4. Graph. The graph benchmark (`graph_bfs`) performs a BFS traversal on a connected, undirected graph of max degree five. Nodes have a random number of random children, but must have at least one. All non-`NULL` children are placed before `NULL` children. We give LINKEY the offsets of the five children of the node, in addition to the full node size and the start node of the traversal. This traversal is implemented using the C++ STL `queue` and `unordered_set`.

4.2.2. Lookup Benchmarks

This second category of benchmarks performs many lookups (probes) of a linked data structure. Under the current configuration, we carry out 1000 probes of an LDS of varying size. Keys are randomly generated using either a uniform distribution (`*_uni`, implementation from C++ STL) or a Zipfian distribution (`*_zipf`, implementation from [**19**, **36**])

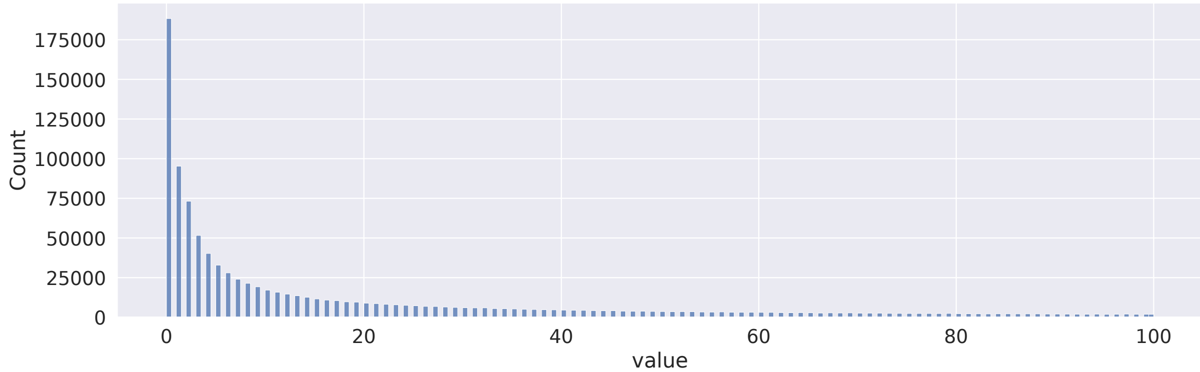

Figure 4.1. A histogram of a Zipfian distribution with parameter $\theta = 0.99$ on $[1, 100]$. The distribution was sampled 1 000 000 times.

with a fixed seed. Uniform distributions serve as a worst-case, while Zipfian distributions are more representative of real-world scenarios (a very small number of very hot keys). Our Zipfian distribution used a skew parameter $\theta = 0.99$ (taken from YCSB [**16**]); a plot of the distribution can be found in Figure 4.1.

For benchmarks with numerical keys, we generate the keys to range from 1 to $2^d - 1$, where d is the depth of the tree. For the dynamic lookup benchmarks, we set aside 5% of keys—as in YCSB [**16**]—to be randomly inserted during the kernel, rather than during the initial tree build, thus evaluating performance on insertions, changing trees, and missing keys. If the root of the data structure changes (as it might with balanced BSTs), we inform the prefetcher.

We expect to see better performance from these benchmarks compared to the traversal benchmarks, as some nodes will certainly be “hotter” than others, especially with Zipfian-distributed keys. However, we do also acknowledge the possibility of a performance hit on dynamic BST benchmarks (i.e., `rbtree_*` and `splay_*`), as the LDS can change on insertion and even lookup, hurting reference locality.

4.2.2.1. Binary Tree. These benchmarks (`bintree_probe_*`) use the same balanced binary search tree as described earlier in Section 4.2.1.2. This tree is static—only lookups are performed.

4.2.2.2. Red-Black Tree. The Red-Black Tree (RBTree) [**20**] is from of balanced BST. This type of tree is widely used in software development—not only is it used throughout Linux [**58**], it is the default implementation of `std::set` and `std::map` in the C++ STL [**33**, **34**]. This benchmark is dynamic—both insertions and lookups are performed. We adapted the implementation from [**56**].

4.2.2.3. Splay Tree. These benchmarks evaluate performance on Splay Trees [**53**], another form of balanced BST. Splay trees are very interesting because they attempt to optimize for temporal locality by moving recently-used keys (on both insertion **and** lookup) closer to the root. Overall lookup complexity still remains at $\mathcal{O}(\log n)$. This benchmark is dynamic—both insertions and lookups are performed. We adapted the implementation from [**56**].

4.2.2.4. Trie. A trie [**17**] is a data structure for efficiently storing strings and searching by prefixes. They have many uses, especially for fast text search [**1**] and web server routing [**45**]. The implementation was adapted from [**59**]—each node has 26 children (one for each letter of the alphabet), and a boolean marker to signify if the node represents the end of a word. All operations are done in lowercase. A random dictionary is used to populate the tree. As only a maximum of eight offsets can be added to the prefetcher, we chose the eight most common letters in the English language (e, t, a, o, i, n, s, and r). A trie might never change over the lifetime of an application (e.g., in a web server), thus, this benchmark is static—only lookups are performed.

4.3. Metrics

Many statistics are available to judge the efficacy of a hardware optimization. We selected the following four metrics, as they would be most impacted by a prefetcher, to evaluate LINKEY:

Miss Count: The direct goal of a prefetcher is to reduce the number of cache misses an application experiences. Thus, we look at miss counts to evaluate our performance. Theoretically, reducing the number of cache misses should lead to improved performance.

IPC: The goal of any hardware optimization is to improve application performance. IPC directly correlates with performance, and we hope to maximize it.

Prefetch Accuracy: Ideally, prefetchers should only prefetch blocks that will later be used by the application. We would like to minimize the number of unused prefetches, to minimize energy usage and contention on shared resources. However, as LINKEY is aggressive to account for high effective memory access times, this statistic will likely suffer.

Prefetch Hits: Similarly to prefetch accuracy, we would like to view our improvement over baseline on the number of hits to prefetched cache lines. This differs from prefetch accuracy, as it is not weighted by the total number of prefetch requests.

We present miss count and IPC statistics normalized to baseline. We divide the plots into columns by benchmark type (lookup vs. traversal). For an aggregate metric, we provide a geomean for both types of benchmark, along with an overall geomean.

4.4. Results

We first begin with an evaluation of different LINKEY configurations. Using the results, we choose the configuration that is most reasonable, given performance and hardware resource utilization. We then perform an in-depth evaluation of that configuration, across a wide range of metrics.

4.4.1. Configuration Evaluation

To choose the optimal configuration, we use the two metrics that most directly correlate with performance: miss count and IPC. The evaluation of load misses is shown in Figure 4.2. The 1.6 KiB version of LINKEY is shown in blue, the 7.2 KiB version in orange, the 32.8 KiB version in green, and the striding baseline in red. For readability, we divide the benchmarks by size into rows. Most benchmarks experience a decrease in load misses with the addition of LINKEY. At size *huge*, on the bottom, only `graph_bfs` and `bintree_bfs` experience increases in the number of load misses.

We show normalized IPC data in Figure 4.3. All configurations show improvements in IPC on some benchmarks, however, the results are not as consistent as miss count. With the 1.6 KiB configuration, geomean IPC even decreases when the benchmark size is *huge*. However, the 7.2 KiB and 32.8 KiB configurations do show improvement on most benchmarks at all sizes.

4.4.1.1. Discussion. The data does show that increased table size leads to improved performance. However, one must consider the scarcity of hardware resources. These graphs show considerable performance improvement with just a 256-entry AT table, and at a size of 7.2 KiB (about 10% of the L1 caches), the resource utilization is in-line with

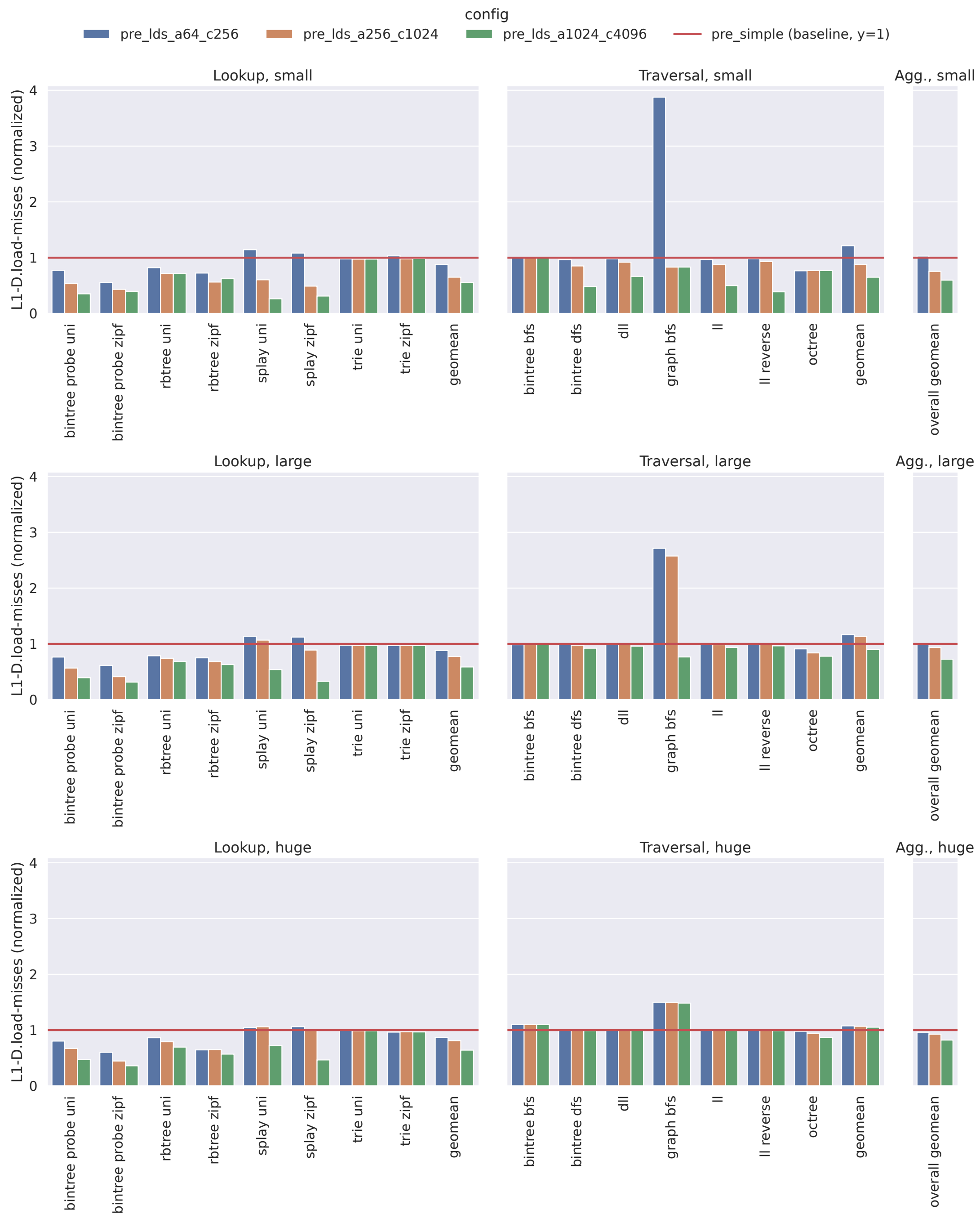

Figure 4.2. Normalized numbers of load misses with different prefetcher configurations. Rows represent benchmark sizes, colors represent configurations.

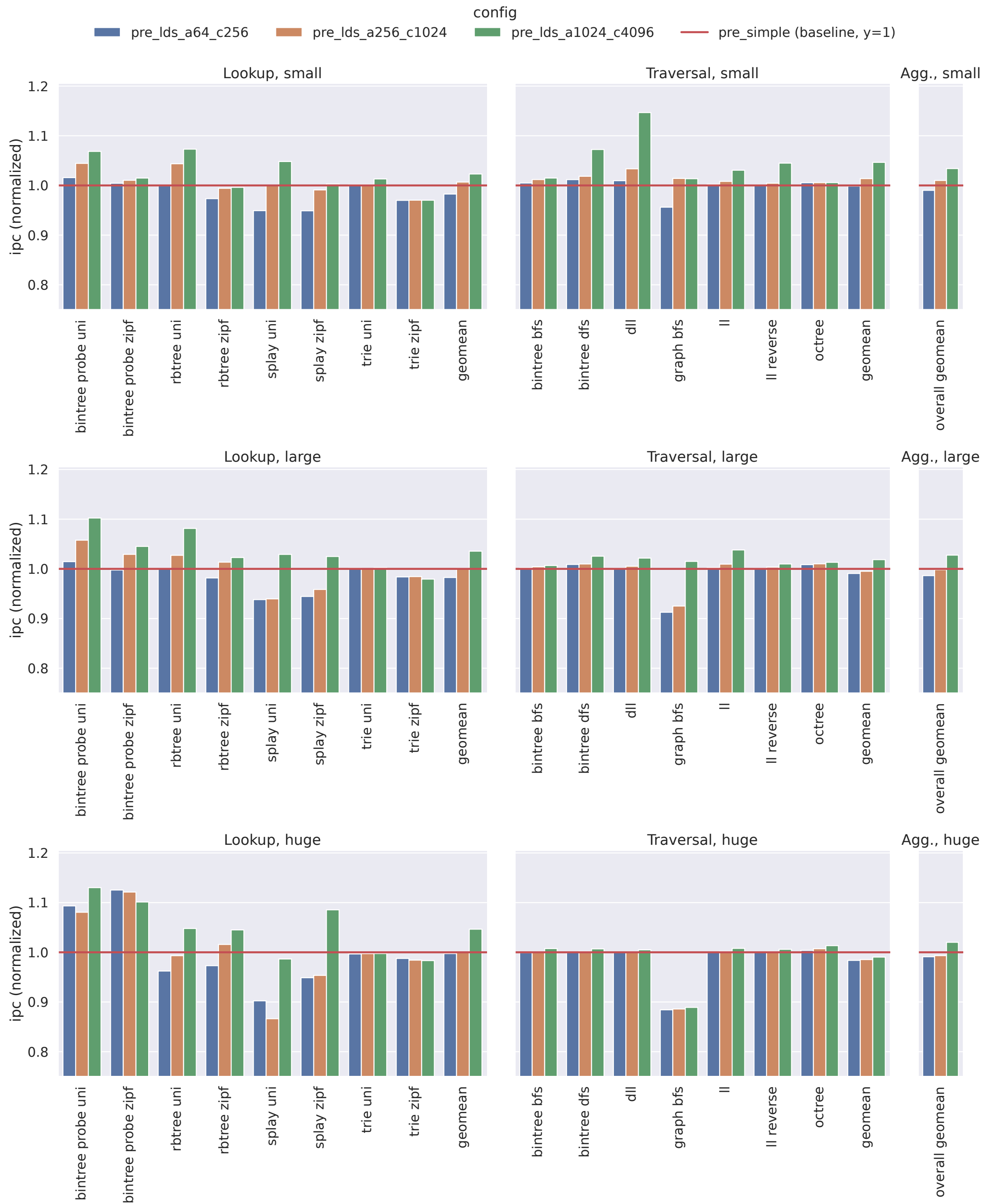

Figure 4.3. Normalized IPC with different prefetcher configurations. Rows represent benchmark sizes, colors represent configurations.

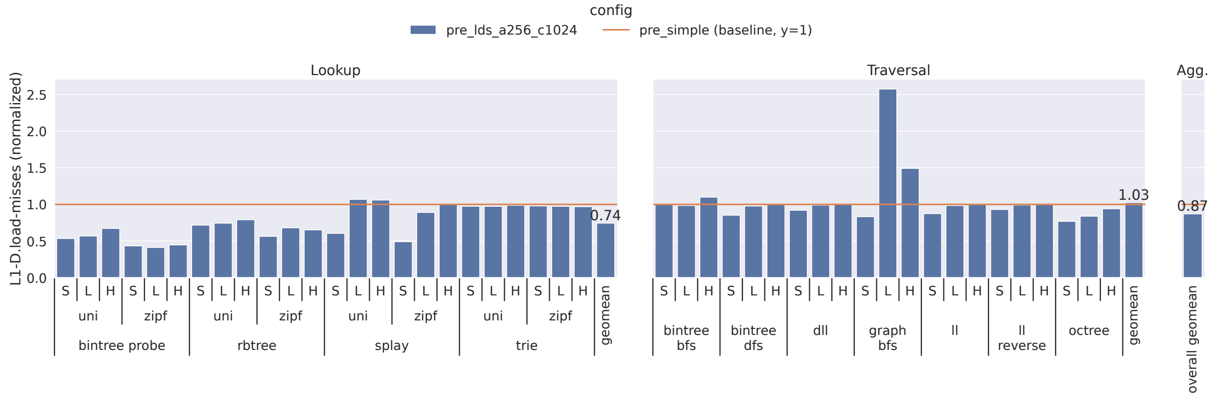

Figure 4.4. Normalized numbers of load misses when using the Linkey prefetcher.

other modern prefetchers [**47**]. Thus, for the remainder of the evaluation, we will consider only the 7.2 KiB Linkey configuration (`pre_lds_a256_c1024`).

4.4.2. Performance Evaluation

In Figure 4.4, we show the normalized load misses between Linkey and the baseline across our benchmark suite. While the Splay Tree and Graph BFS benchmarks see an increase in load misses, we observe a decease in almost all other cases, many times by over 2×. As test sizes increase, load misses tend to increase, although not substantially, demonstrating the scalability of the Linkey approach. Overall, the geomean sees a decrease of 13%.

We show prefetch accuracy results in Figure 4.5. Trends in miss rate line up with trends in accuracy, as expected—when accuracy is high, miss rate tends to be lower. Prefetch accuracy also decreases as sizes increase. However, in many cases the baseline had a prefetch accuracy of 0%—i.e., none of its requests hit at all, while almost all the Linkey prefetches hit in the same benchmarks. While the baseline did do better on some lookup benchmarks, it also issued very few prefetches, meaning the total number of hits

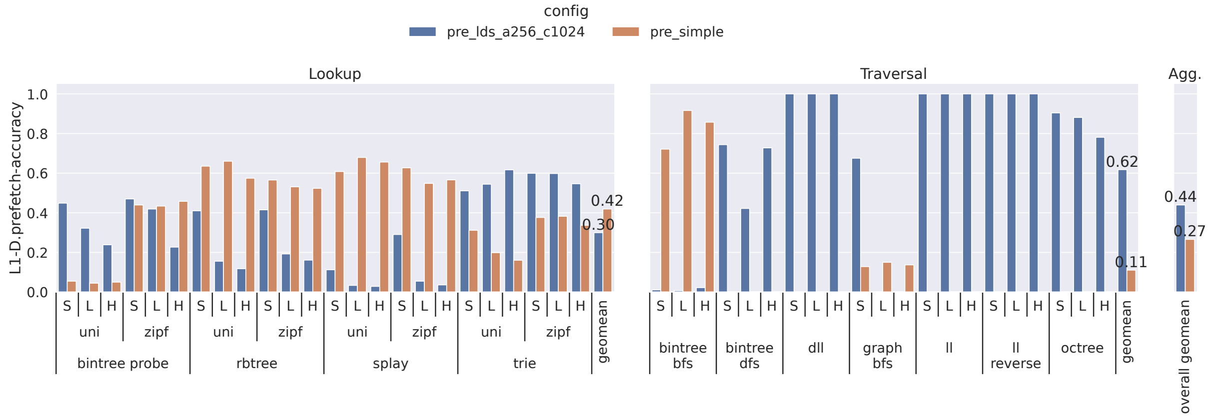

Figure 4.5. Prefetch accuracy of the Linkey prefetcher and the striding baseline.

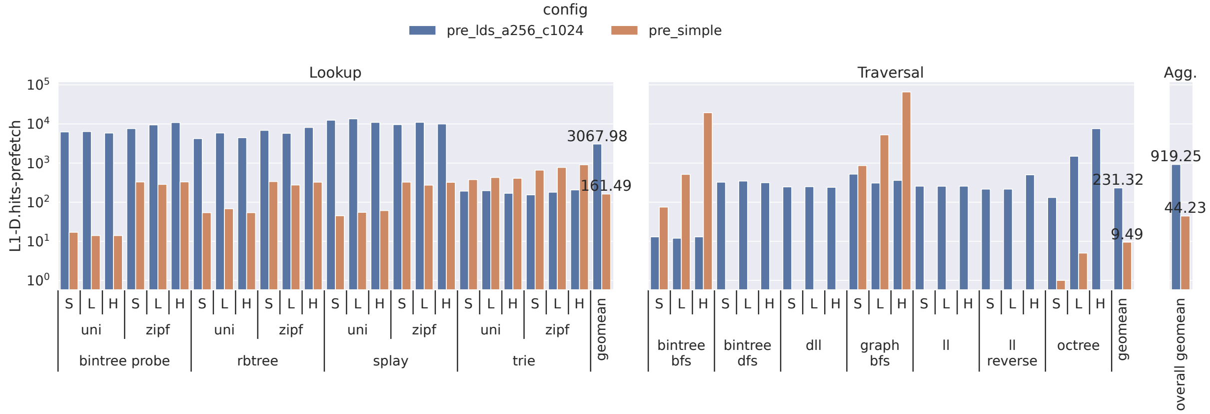

Figure 4.6. Prefetch hit counts of the Linkey prefetcher and striding baseline. Note the use of a log scale on the Y-axis.

was small. Overall, we observe a 65.4% improvement in mean prefetch accuracy, from 26.6% to 43.9%. This shows that Linkey can accurately identify and fetch important child pointers in an LDS.

We show prefetch hit counts in Figure 4.6. Note the use of a log scale on the Y-axis. In only two benchmarks (`bintree_bfs` and `graph_bfs`) do we observe significant decreases in the number of prefetch hits—overall numbers increase substantially, sometimes by

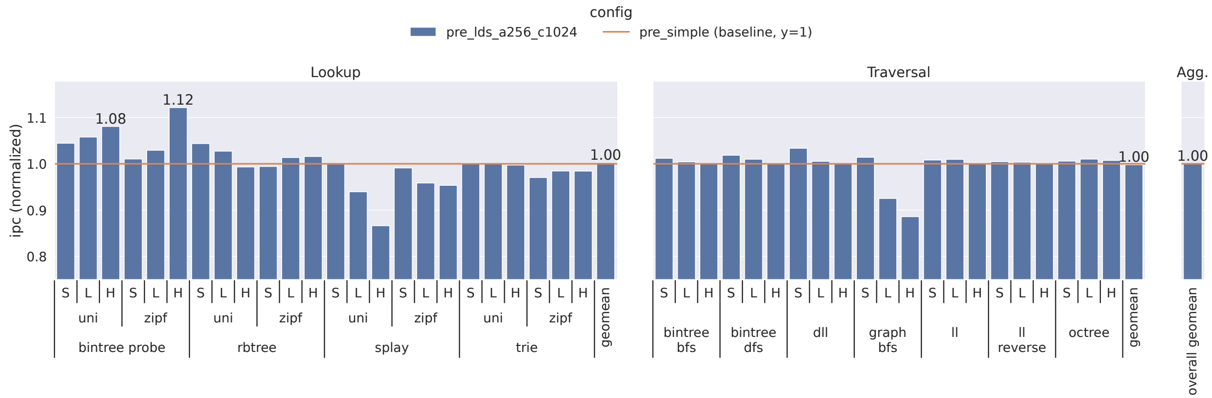

Figure 4.7. Normalized IPC when using the LINKEY prefetcher.

multiple orders of magnitude, especially on lookup benchmarks. The geomean increase in the number of hits was 19.8×; this proves our hypothesis that fetching several "levels" of an LDS at a time will improve cache performance.

In Figure 4.7, we show the IPC results. In cases with low accuracy and high miss counts, we do see some IPC decreases. However, the benchmarks with IPC improvements see substantial improvements—especially `bintree_probe_*` (with the highest improvement of 12.1%) and `rbtree_*`. Additionally, on benchmarks with benefits, IPC gains often improve as test size **increases**, further demonstrating the scalability of our approach. The overall geomean of normalized IPC increases by 0.05%.

4.4.2.1. Discussion. As expected, we see significant performance gains across many benchmarks, especially the lookup benchmarks. While there are IPC decreases on some benchmarks, almost all significant decreases are in the Graph BFS or Splay Tree benchmarks. The performance hit in the graph benchmark is from the nature of BFS—as we visit each node *exactly once*, our assumption that certain nodes are hot breaks down, and

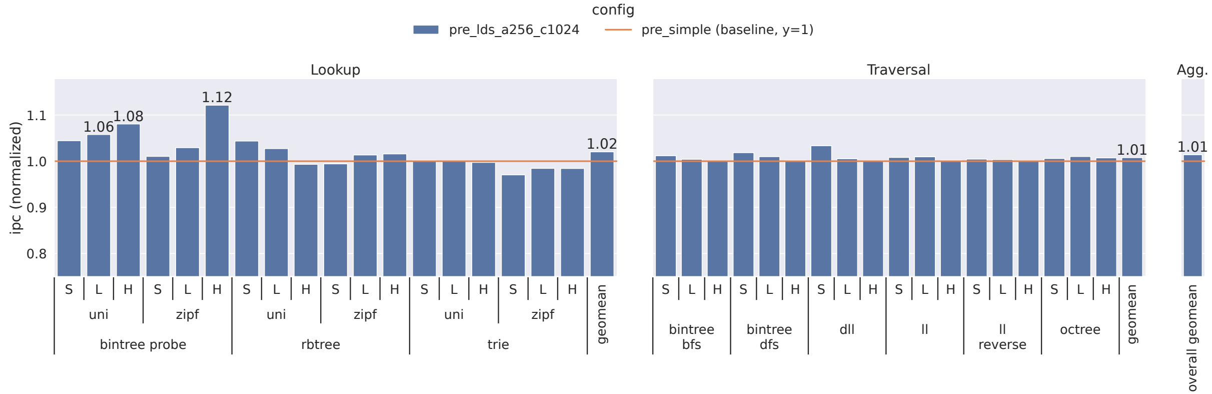

Figure 4.8. Normalized IPC when using the LINKEY prefetcher, with `graph_bfs` and Splay Tree benchmarks removed.

the cache ends up polluted with unused prefetches. For splay trees, the constant rearranging leads to *low reference locality*, which predictably hurts performance due to the need for continuous table invalidations. Our assumption that mutations are significantly rarer than lookups holds with dynamic Red-Black Trees, where we see substantial performance improvements (up to 4.3% on `rbtree_uni-small`). However, as our prefetcher is *configurable*, it can simply be disabled (i.e., just not configured) by the programmer on Splay Trees or graphs traversed in BFS order. IPC with those benchmarks removed is shown in Figure 4.8, where we observe a geomean performance increase of 1.40%, and up to 12.1% on lookup benchmarks.

CHAPTER 5

Related Works

With the long history of prefetching technologies, and the importance of linked data structures, there is a significant body of work on algorithms that benefit applications with pointer-chasing access patterns. Here, we focus on the work that best relates to our contributions.

Apple [**12**, **60**] and Intel [**15**] both developed CDPs that identify values in the cache likely to be pointers, with the help of the memory access stream from the core, and issue fetches accordingly. However, this approach has numerous security flaws, as the lack of pointer provenance means data values may be interpreted as pointers and thus fetched, opening a covert channel that attackers can use to read sensitive data through timing attacks. Additionally, these approaches will suffer from degraded performance as effective memory access latencies increase—every cache block must return from memory before it can be scanned for pointers and the next prefetch requests issued. Finally, even if all pointers in a cache block are valid, it is highly unlikely all will be accessed (e.g., cold data pointers), leading to memory system congestion and cache pollution. LINKEY avoids these performance issues with its knowledge of LDS node layout.

Efficient Content Directed Prefetching (ECDP) [**18**] builds on Content Directed Prefetching (CDP) as introduced in [**15**] with the goal of eliminating extraneous loads caused by fetching *every* pointer in a cache block. First, the application is profiled, with the goal of finding the offsets of useful loads from the location of the first load in a cache

block. When a block is accessed, ECDP searches for values that may be pointers, just like the original CDP, but then utilizes the profiling information to avoid performing certain loads that would not be useful. As a variation of CDP, this technique still falls victim to the issues of sequential loads as detailed above, in addition to the known issues with profiling (e.g., finding fully representative input data). However, it does attempt to solve the extraneous request problem similarly to Linkey—by passing information down from the compiler.

Al-Sukhni et al. [**3**] introduce Compiler-Directed Content Aware Prefetching (CDCAP), a technique that also evolves CDP [**15**]. Compiler-based profiling is used to statically identify offsets to prefetch, which are then provided to a hardware prefetch engine through special "Harbinger" instructions. These instructions are similar to the instructions Linkey uses to convey layout information to the hardware. However, CDCAP only supports a single "linking" (i.e., recursive) pointer, meaning performance will suffer on data structures with more than one child. Furthermore, the "Harbinger" instructions are inserted into hot loops, requiring compiler support to "time" them (using an inherently flawed cost model) and also increasing code size. Like most approaches, CDCAP must also wait for loads to return from memory before recursing—unlike Linkey, which uses its tables to get "ahead" of the application.

In [**61**], Wang et al. describe a prefetch engine that utilizes hints from a compiler analysis to inform issued requests. Notably, the analysis detects common cases of recursive pointers as used in linked-list traversals, which hardware then exploits to issue prefetches of the LDS up to six levels deep. However, layout information is not provided to the hardware prefetch engine—it instead speculates on values in a cache block being pointers,

falling victim to the same cache pollution flaws as other CDPs. Nodes are also assumed to be smaller than two cache blocks, which may not hold across real-world applications. The researchers also state the technique does not perform well on trees. Most importantly, prefetch requests for an LDS are issued sequentially, and thus performance degrades with increased effective memory access times.

Roth et al. describe a prefetching technique in [**50**] that utilizes dependency chains to determine the shape of a linked data structure node in hardware, and issue requests accordingly. While this approach requires no modification of the executable, it cannot learn the full structure of an object without observing accesses to all children. Additionally, the authors limit their prefetcher to only a single node ahead, meaning prefetches are only triggered when the core issues new loads, *and* blocks must be returned from the memory system before new prefetches can be issued—leading to significantly worse performance as effective memory access latencies increase.

Liu et al. [**37**] build on dependence based prefetching (DBP) as described in [**50**] to enable *timely* prefetches of children; this is needed as increased memory access times require prefetches to be issued earlier to avoid stalls. Similar to DBP, the instruction and memory access streams are used to build dependency relationships between instructions, however, the authors then classify the loads into direct data accesses, indirect data accesses, and recursive node accesses. This information is then used to issue prefetch loads when memory responses arrive at the core—effectively, this technique is similar to ECDP [**18**], but performs the shape analysis in hardware using methods inspired by DBP. As more information is extracted from the instruction stream, more aggressive prefetches can be issued, which helps to improve their "timeliness." However, as the technique attempts to guess

if values are pointers, it is vulnerable to cache pollution like other CDPs—even more so due to the increased aggressiveness of prefetching. Additionally, as the shape analysis is performed in hardware, the full structure of the node can only be learned if accesses to all children are observed. Finally, prefetches are still issued sequentially, meaning with ever-increasing effective memory access times, prefetch timeliness will still suffer. LINKEY similarly attempts to improve timeliness, but instead uses its AT/CAT tables to do so.

Jump pointers [**29**, **39**, **51**] are a prefetching technique to avoid "pointer chains" when prefetching LDSs. Extra pointer(s) are added to each node, referencing the LDS node(s) that will be accessed N timestamps later. Then, the prefetcher can issue requests during a traversal using these jump pointers. This works well if LDS accesses are correlated, however, if something like a tree is being searched based on arbitrary user input, this technique will fail. Additionally, the storage requirements for these pointers present significant overheads, especially on memory-constrained programs.

Correlation prefetchers [**27**, **55**, **62**] check for repeated *streams* of accesses, correlating access patterns with prefetch triggers. They utilize temporal locality to capture sequences of hot access patterns, regardless of complexity. This is very successful on linked data structures with a skewed distribution of node accesses, as the prefetcher can track the hottest streams and issue requests down to the last node. However, the effectiveness of these prefetchers drops as the skewed-ness of an LDS decreases, as only a fixed number of streams can be tracked at once. LINKEY takes the opposite approach: rather than issuing very precise fetches to a few nodes, it brings in many nodes that have a high likelihood of being useful in the near future. This allows greater adaptability across

access patterns, with the tradeoff that hotter nodes lose a bit of performance so the rest can gain substantially more.

Graph prefetching algorithms have been introduced in [**2**, **30**, **32**, **68**] and can significantly improve performance by heavily optimizing for the LDS representations commonly used by graphs. However, these techniques are all so heavily specialized towards a specific application and representation that it is unlikely they will be implemented in a practical system, or have code re-written to fit their programming paradigm.

Many techniques exist that extract a *prefetch kernel*—a small piece of code that mimics the access pattern of the original application—and either assign it to a helper thread running in SMT [**21**, **22**, **23**, **25**], a small processor in the memory hierarchy [**24**, **64**, **65**], or simply embed the kernel in the application code itself [**52**]. While these techniques can produce benefits, the cost of running an entire thread to perform prefetching or redesigning the memory hierarchy to include programmable prefetch engines is too high to be practical. Even just embedding the prefetching kernel in the application code will hurt performance due to code size increases.

Collins et al. [**13**] introduce a "pointer cache" that breaks serial dependence chains in code such as `p->next->data`. Effectively, it serves as a value predictor for the `p->next` load, allowing that load and the load of `data` to proceed in parallel. The algorithm for determining if a value is a pointer is taken from the CDP paper [**15**]. Linkey does not eliminate these dependency chains, but simply attempts to prefetch the pointer data for many nodes in parallel, preventing stalls from occurring.

Finally, while indirect memory prefetchers [**10**, **57**, **63**, **67**] and LDS prefetchers like Linkey have some overlap, the problem spaces are different. Indirect memory prefetchers

are typically more focused on array-of-pointers accesses, as would be found in virtual function calls or certain sparse matrix representations. Accesses to these structures do not involve pointer chasing, but instead a (possibly complex) indexing $A[f(B[i])]$. Indirect memory prefetchers attempt to predict the value of $f(B[i])$, a different problem than what LINKEY solves.

CHAPTER 6

Conclusion and Future work

Ever-increasing effective memory access times drive the need for novel memory subsystem optimization approaches. Prefetching, while an effective technique for contiguous data structures that follow traditional locality, has struggled when presented with linked data structures. Due to the prevalence and importance of these structures, we must develop new techniques to effectively issue prefetches, or else programs will suffer from ever-increasing memory stalls.

We introduce LINKEY, a novel prefetching technique for linked data structures. Using only basic node layout information, LINKEY can identify and cache layout information of linked data structures on the core. As memory requests arrive, LINKEY both effectively issues prefetches down the LDS and builds its internal representation of the data structure. Unlike other techniques, LINKEY does not depend on complex compiler analyses, profiling, or significant user instrumentation to accurately predict future accesses. By allowing the software to specify the link pointers to prefetch, LINKEY avoids the cache pollution caused by other CDPs. Furthermore, we designed LINKEY with modern hardware in mind, and used increased memory bandwidth available in modern systems to prefetch aggressively in the hopes of avoiding future miss penalties.

With these techniques, LINKEY manages to issue targeted prefetches to the most vital nodes of a linked data structure. As a result, we measure substantial performance improvements over a striding baseline. LINKEY achieves a geomean 13% reduction in miss

rate with a maximum improvement of 58.8%, and a 65.4% geomean increase in accuracy, with many benchmarks improving from 0%. On benchmarks where Linkey is applicable, we observe a geomean IPC improvement of 1.40%, up to 12.1%.

6.1. Future Work

The design and implementation of Linkey could be further extended. We could add more intrusive methods of tracking linked data structures to the hardware (such as pointer tagging to mark provenance), increasing accuracy while remaining fully transparent to the user. We could also perform further benchmarking using other real-world applications, such as the Olden benchmark suite [**7**]. Finally, we could add a separate prefetch buffer to reduce cache pollution, or configure prefetch requests to land in the much larger L2$—hopefully resolving the slowdowns experienced by the Splay Tree and Graph benchmarks.

References

[1] Aho, A. V., and Corasick, M. J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340.

[2] Ainsworth, S., and Jones, T. M. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the 2016 International Conference on Supercomputing* (New York, NY, USA, June 2016), ICS '16, Association for Computing Machinery, pp. 1–11.

[3] Al-Sukhni, H., Bratt, I., and Connors, D. Compiler-directed content-aware prefetching for dynamic data structures. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2003), pp. 91–100.

[4] Ayers, G., Litz, H., Kozyrakis, C., and Ranganathan, P. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne Switzerland, Mar. 2020), ACM, pp. 513–526.

[5] Besta, M., and Hoefler, T. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations, Apr. 2019.

[6] Burcea, I., Soares, L., and Moshovos, A. Pointy: A hybrid pointer prefetcher for managed runtime systems. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Sept. 2012), pp. 97–106.

[7] Carlisle, M. C. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, United States – New Jersey, 1996.

[8] Carlson, T. E., Heirman, W., and Eeckhout, L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle Washington, Nov. 2011), ACM, pp. 1–12.

[9] Carlson, T. E., Heirman, W., Eyerman, S., Hur, I., and Eeckhout, L. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization 11*, 3 (Oct. 2014), 1–25.

[10] Cavus, M., Sendag, R., and Yi, J. J. Informed Prefetching for Indirect Memory Accesses. *ACM Trans. Archit. Code Optim. 17*, 1 (Mar. 2020), 4:1–4:29.

[11] Cebrian, J. M., Jahre, M., and Ros, A. Temporarily Unauthorized Stores: Write First, Ask for Permission Later. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Nov. 2024), pp. 810–822.

[12] Chen, B., Wang, Y., Shome, P., Fletcher, C., Kohlbrenner, D., Paccagnella, R., and Genkin, D. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *33rd USENIX Security Symposium (USENIX Security 24)* (2024), pp. 1117–1134.

[13] Collins, J., Sair, S., Calder, B., and Tullsen, D. Pointer cache assisted prefetching. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* (Nov. 2002), pp. 62–73.

[14] Comer, D. Ubiquitous B-Tree. *ACM Computing Surveys 11*, 2 (June 1979), 121–137.

[15] Cooksey, R., Jourdan, S., and Grunwald, D. A stateless, content-directed data prefetching mechanism. *SIGOPS Oper. Syst. Rev. 36*, 5 (Oct. 2002), 279–290.

[16] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, June 2010), SoCC '10, Association for Computing Machinery, pp. 143–154.

[17] De La Briandais, R. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference* (New York, NY, USA, Mar. 1959), IRE-AIEE-ACM '59 (Western), Association for Computing Machinery, pp. 295–298.

[18] Ebrahimi, E., Mutlu, O., and Patt, Y. N. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture* (Feb. 2009), pp. 7–17.

[19] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. *SIGMOD Rec. 23*, 2 (May 1994), 243–252.

[20] Guibas, L. J., and Sedgewick, R. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (Sfcs 1978)* (Oct. 1978), pp. 8–21.

[21] Huang, Y., and Gu, Z. Performance Analysis of Prefetching Thread for Linked Data Structure in CMPs. In *2009 International Conference on Computational Intelligence and Software Engineering* (Dec. 2009), pp. 1–4.

[22] Huang, Y., Gu, Z.-M., Tang, J., Cai, M., Zhang, J., and Zheng, N. Estimating Effective Prefetch Distance in Threaded Prefetching for Linked Data Structures. *International Journal of Parallel Programming 40*, 5 (Oct. 2012), 465–487.

[23] Huang, Y., Tang, J., Gu, Z.-m., Cai, M., Zhang, J., and Zheng, N. The Performance Optimization of Threaded Prefetching for Linked Data Structures. *International Journal of Parallel Programming 40*, 2 (Apr. 2012), 141–163.

[24] Hughes, C. *Prefetching Linked Data Structures in Systems with Merged Dram-Logic.* Master of Science, University of Illinois at Urbana-Champaign, 2000.

[25] HuiDong, Z., Jing, C., Wei, D., and Yan, H. Performance evaluation of thread prefetching for recursive data structures. In *2011 IEEE 3rd International Conference on Communication Software and Networks* (May 2011), pp. 610–613.

[26] JEDEC. Graphics Double Data Rate (GDDR5X) SGRAM Standard, Mar. 2023.

[27] Joseph, D., and Grunwald, D. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (New York, NY, USA, May 1997), ISCA '97, Association for Computing Machinery, pp. 252–263.

[28] Kalter, H., Stapper, C., Barth, J., DiLorenzo, J., Drake, C., Fifield, J., Kelley, G., Lewis, S., Van Der Hoeven, W., and Yankosky, J. A 50-ns 16-Mb DRAM with a 10-ns data rate and on-chip ECC. *IEEE Journal of Solid-State Circuits 25*, 5 (Oct. 1990), 1118–1128.

[29] Karlsson, M., Dahlgren, F., and Stenstrom, P. A prefetching technique for irregular accesses to linked data structures. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)* (Jan. 2000), pp. 206–217.

[30] Kaushik, A. M., Pekhimenko, G., and Patel, H. Gretch: A Hardware Prefetcher for Graph Analytics. *ACM Trans. Archit. Code Optim. 18*, 2 (Feb. 2021), 18:1–18:25.

[31] Khokhlov, A. Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations. *Journal of Computational Physics 143*, 2 (July 1998), 519–543.

[32] Lakshminarayana, N. B., and Kim, H. Spare register aware prefetching for graph algorithms on GPUs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (Feb. 2014), pp. 614–625.

[33] Lattner, C. Llvm/llvm-project. LLVM, May 2025.

[34] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (Mar. 2004), pp. 75–86.

[35] Lee, J., Cho, K., Lee, C. K., Lee, Y., Park, J.-H., Oh, S.-H., Ju, Y., Jeong, C., Cho, H. S., Lee, J., Yun, T.-S., Cho, J. H., Oh, S., Moon, J., Park, Y.-J., Choi, H.-S., Kim, I.-K., Yang, S. M., Kim, S.-Y., Jang, J., Kim, J., Lee, S.-H., Jeon, Y., Park, J., Kim, T.-K., Ka, D., Oh, S., Kim, J., Jeon, J., Kim, S., Kim, K. T., Kim, T., Yang, H., Yang, D., Lee, M., Song, H., Jang, D., Shin, J., Kim, H., Baek, C., Jeong, H., Yoon, J., Lim, S.-K., Lee, K. Y., Koo, Y. J., Park, M.-J., Cho, J., and Kim, J. 13.4 A 48GB 16-High 1280GB/s HBM3E DRAM with All-Around Power TSV and a 6-Phase RDQS Scheme for TSV Area Optimization. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)* (Feb. 2024), vol. 67, pp. 238–240.

[36] Lersch, L. Llersch/cpp_random_distributions, Oct. 2024.

[37] Liu, G., Huang, Z., Peir, J.-k., and Shi, X. Semantics-Aware, Timely Prefetching of Linked Data Structure. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems* (Dec. 2010), pp. 213–220.

[38] Luk, C.-K., and Mowry, T. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers 48*, 2 (Feb. 1999), 134–141.

[39] Luk, C.-K., and Mowry, T. C. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, Sept. 1996), ASPLOS VII, Association for Computing Machinery, pp. 222–233.

[40] Martínez-Prieto, M. A., Brisaboa, N., Cánovas, R., Claude, F., and Navarro, G. Practical compressed string dictionaries. *Information Systems 56* (Mar. 2016), 73–108.

[41] Maruszewski, J. Octrees and Lattice Boltzmann, Apr. 2025.

[42] Meagher, D. Geometric modeling using octree encoding. *Computer Graphics and Image Processing 19*, 2 (June 1982), 129–147.

[43] Mittal, S. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Computing Surveys 49*, 2 (June 2017), 1–35.

[44] Mori, K., Kosugi, S., Yoshida, H., Shimada, H., and Ando, H. Localizing the Tag Comparisons in the Wakeup Logic to Reduce Energy Consumption of the Issue Queue. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Nov. 2024), pp. 493–506.

[45] Nouvertné, J., Schutt, P., Fincher, C., Unnikrishnan, V., Coffee, J., and Hirschfeld, N. Litestar, Apr. 2024.

[46] Park, M.-J., Cho, H. S., Yun, T.-S., Byeon, S., Koo, Y. J., Yoon, S., Lee, D. U., Choi, S., Park, J., Lee, J., Cho, K., Moon, J., Yoon, B.-K., Park, Y.-J., Oh, S.-m., Lee, C. K., Kim, T.-K., Lee, S.-H., Kim, H.-W., Ju, Y., Lim, S.-K., Baek, S. G., Lee, K. Y., Lee, S. H., We, W. S., Kim, S., Choi, Y., Lee, S.-H., Yang, S. M., Lee, G., Kim, I.-K., Jeon, Y., Park, J.-H., Yun, J. C., Park, C., Kim, S.-Y., Kim, S., Lee, D.-Y., Oh, S.-H., Hwang, T., Shin, J., Lee, Y., Kim, H., Lee, J., Hur, Y., Lee, S., Jang, J., Chun, J., and Cho, J. A 192-Gb 12-High 896-GB/s HBM3 DRAM with a TSV Auto-Calibration Scheme and Machine-Learning-Based Layout Optimization. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)* (Feb. 2022), vol. 65, pp. 444–446.

[47] Peled, L., Weiser, U., and Etsion, Y. A Neural Network Prefetcher for Arbitrary Memory Access Patterns. *ACM Trans. Archit. Code Optim. 16*, 4 (Oct. 2019), 37:1–37:27.

[48] Ramalingam, G. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5 (Sept. 1994), 1467–1471.

[49] Rogers, B. M., Krishna, A., Bell, G. B., Vu, K., Jiang, X., and Solihin, Y. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*

(New York, NY, USA, June 2009), ISCA '09, Association for Computing Machinery, pp. 371–382.

[50] Roth, A., Moshovos, A., and Sohi, G. S. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, Oct. 1998), ASPLOS VIII, Association for Computing Machinery, pp. 115–126.

[51] Roth, A., and Sohi, G. S. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (USA, May 1999), ISCA '99, IEEE Computer Society, pp. 111–121.

[52] Sankaranarayanan, K., Lin, C.-K., and Chinya, G. N. Helper Without Threads: Customized Prefetching for Delinquent Irregular Loads. *ArXiv* (Sept. 2020).

[53] Sleator, D. D., and Tarjan, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.

[54] Smith, A. J. Cache Memories. *ACM Comput. Surv. 14*, 3 (Sept. 1982), 473–530.

[55] Somogyi, S., Wenisch, T. F., Ailamaki, A., and Falsafi, B. Spatio-temporal memory streaming. *SIGARCH Comput. Archit. News 37*, 3 (June 2009), 69–80.

[56] Suchy, B., Campanoni, S., Hardavellas, N., and Dinda, P. CARAT: A case for virtual memory through compiler- and runtime-based address translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, June 2020), PLDI 2020, Association for Computing Machinery, pp. 329–345.

[57] Talati, N., May, K., Behroozi, A., Yang, Y., Kaszyk, K., Vasiladiotis, C., Verma, T., Li, L., Nguyen, B., Sun, J., Morton, J. M., Ahmadi, A., Austin, T., O'Boyle, M., Mahlke, S., Mudge, T., and Dreslinski, R. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Feb. 2021), pp. 654–667.

[58] Torvalds, L. Kernel/git/torvalds/linux.git - Linux kernel source tree, May 2025.

[59] Vedala, K., and Leal, D. TheAlgorithms/C. The Algorithms, May 2025.

[60] Vicarte, J. R. S., Flanders, M., Paccagnella, R., Garrett-Grossman, G., Morrison, A., Fletcher, C. W., and Kohlbrenner, D. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *2022 IEEE Symposium on Security and Privacy (SP)* (May 2022), pp. 1491–1505.

[61] Wang, Z., Burger, D., McKinley, K., Reinhardt, S., and Weems, C. Guided region prefetching: A cooperative hardware/software approach. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* (June 2003), pp. 388–398.

[62] Wenisch, T., Somogyi, S., Hardavellas, N., Kim, J., Ailamaki, A., and Falsafi, B. Temporal streaming of shared memory. In *32nd International Symposium on Computer Architecture (ISCA'05)* (June 2005), pp. 222–233.

[63] Xue, F., Han, C., Li, X., Wu, J., Zhang, T., Liu, T., Hao, Y., Du, Z., Guo, Q., and Zhang, F. Tyche: An Efficient and General Prefetcher for Indirect Memory Accesses. *ACM Trans. Archit. Code Optim. 21*, 2 (Mar. 2024), 30:1–30:26.

[64] Yang, C.-L., and Lebeck, A. A Programmable Memory Hierarchy for Prefetching Linked Data Structures. In *High Performance Computing* (Berlin, Heidelberg, 2002), H. P. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, Eds., Springer, pp. 160–174.

[65] Yang, C.-L., and Lebeck, A. R. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 14th International Conference on Supercomputing* (New York, NY, USA, May 2000), ICS '00, Association for Computing Machinery, pp. 176–186.

[66] Yang, J., Ko, H., Kim, K., Park, H., Park, J., Kang, J.-H., Cha, J., Kim, S., Kim, Y., Park, M., Lee, G., Lee, K., Lee, S., Jeon, G., Jeong, S., Joo, Y., Cha, J., Hwang, S., Kim, B., Byeon, S., Lee, S., Park, H., Cho, J., and Kim, J. 13.1 A 35.4Gb/s/pin 16Gb GDDR7 with a Low-Power Clocking Architecture and PAM3 IO Circuitry. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)* (San Francisco, CA, USA, Feb. 2024), IEEE, pp. 232–234.

[67] Yu, X., Hughes, C. J., Satish, N., and Devadas, S. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture* (New York, NY, USA, Dec. 2015), MICRO-48, Association for Computing Machinery, pp. 178–190.

[68] Zhang, X., Liu, C., Ni, J., Cheng, Y., Zhang, L., Li, H., and Li, X. PDG: A Prefetcher for Dynamic Graph Updating. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 43*, 4 (Apr. 2024), 1246–1259.

[69] Zhang, Y., Sobotka, N., Park, S., Jamilan, S., Khan, T. A., Kasikci, B., Pokam, G. A., Litz, H., and Devietti, J. RPG2: Robust Profile-Guided Runtime Prefetch Generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, Apr. 2024), vol. 2 of *ASPLOS '24*, Association for Computing Machinery, pp. 999–1013.

APPENDIX A

Other Contributions

This prefetching work was performed entirely during my final year at Northwestern University. However, prior to beginning this project, I was working on quantum computing research, specifically on compilation for modular quantum systems. While I carried out the bulk of that work over the prior two years, a substantial portion to wrap up the project and prepare for publication was done during my final year, concurrently with this thesis. The quantum work is currently under review at a top systems conference. The abstract to the paper (which can be found at arXiv:2501.08478) is quoted below.

> As quantum computing technology continues to mature, industry is adopting modular quantum architectures to keep quantum scaling on the projected path and meet performance targets. However, the complexity of chiplet-based quantum devices, coupled with their growing size, presents an imminent scalability challenge for quantum compilation. Contemporary compilation methods are not well-suited to chiplet architectures. In particular, existing qubit allocation methods are often unable to contend with inter-chiplet links, which don't necessary support a universal basis gate set. Furthermore, existing methods of logical-to-physical qubit placement, swap insertion (routing), unitary synthesis, and/or optimization, are typically not designed for qubit links of wildly

varying levels of duration or fidelity. In this work, we propose SEQC, a complete and parallelized compilation pipeline optimized for chiplet-based quantum computers, including several novel methods for qubit placement, qubit routing, and circuit optimization. SEQC achieves up to a 36% increase in circuit fidelity, accompanied by execution time improvements of up to 1.92×. Additionally, owing to its ability to parallelize compilation, SEQC achieves consistent solve time improvements of $2 - 4\times$ over a chiplet-aware Qiskit baseline.

A.1. Personal Contributions

I was the first person working on this quantum project, and the only one for the first year. My work involved formulating the problem as contained optimization; this led to an initial version of the compiler using MIP solvers. While we achieved good results, the MIP solver was impractically slow, and we pivoted to a greedy approach that later became SEQC. My work involved developing and testing initial mapping algorithms, including the Simulated Annealing technique that was utilized in the paper. I also contributed to results collection, analysis, and paper writing. These contributions led to my naming as a **co-first author** on the final paper.

Vita

Nikola (Nino) Maruszewski is an undergraduate student studying Computer Science at Northwestern University. He is also completing a combined MS in Computer Engineering, advised by Prof. Nikos Hardavellas. Nikola first started his research in quantum computing, where he investigated compilation techniques for modular quantum hardware architectures. However, Nikola now works mainly in the computer architecture space, with his current work in improving prefetching technologies for linked data structures. After his graduation from Northwestern in June 2025, Nikola will be attending Georgia Tech in the fall to pursue a PhD under the supervision of Prof. Josiah Hester.