Limoncello: Prefetchers for Scale

Akanksha Jain*
avjain@google.com
Google, USA

Hannah Lin*
hylin@google.com
Google, University of Washington,†
USA

Carlos Villavieja
villavieja@google.com
Google, USA

Baris Kasikci
kasikci@google.com
Google, University of Washington,†
USA

Chris Kennelly
ckennelly@google.com
Google, USA

Milad Hashemi
miladh@google.com
Google, USA

Parthasarathy Ranganathan
partha.ranganathan@google.com
Google, USA

Abstract

This paper presents Limoncello, a novel software system that dynamically configures data prefetchers for high-utilization systems. We demonstrate that in resource-constrained environments, such as large data centers, traditional methods of hardware prefetching can increase memory latency and decrease available memory bandwidth. To address this issue, Limoncello disables hardware prefetchers when memory bandwidth utilization is high, and it leverages targeted software prefetching to reduce cache misses when hardware prefetchers are disabled. Limoncello is software-centric and does not require any modifications to hardware. Our evaluation of the deployment on Google's fleet reveals that Limoncello unlocks significant performance gains for high-utilization systems: It improves application throughput by 10%, due to a 15% reduction in memory latency, while maintaining minimal change in cache miss rate for targeted library functions.

ACM Reference Format:
Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. 2024. Limoncello: Prefetchers for Scale. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3620666.3651373

*Akanksha Jain and Hannah Lin contributed equally to this work.

†This work was done entirely at Google.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651373

1 Introduction

In today's data-driven world, hyperscale fleets operate under increasing pressure to maximize efficiency and resource utilization. Memory stalls account for nearly 40% of cycles in large data centers and are a major performance bottleneck [1, 2]. Hardware prefetchers [3] can mitigate memory stalls by retrieving program data from main memory ahead of time, but the effectiveness of hardware prefetchers is highly dependent on workload characteristics. In fact, it is common for CPU designers to design and tune hardware prefetchers using benchmarking suites, such as SPEC [4–8].

This paper is motivated by the observation that this approach of designing hardware prefetchers does not work well for large data centers and hurts performance at scale. We study the impact of disabling hardware prefetchers across tens of thousands of machines in Google's data center fleet, and we find that while disabling hardware prefetchers increases cache miss rates by 20%, it also reduces memory latency—the time spent waiting for a last level cache miss to return—by 15%. The increase in cache miss rates is not surprising, but the memory latency penalty due to hardware prefetching is a tremendous opportunity cost, especially considering that improvements to DRAM latency have essentially stalled [9].

Figure 1 sheds more insight on the impact of hardware prefetchers on memory latency. As Figure 1 shows, a state-of-the-art server CPU observes a 2× increase in load-to-use latency (y-axis) as memory bandwidth utilization increases (x-axis). Hardware prefetchers further increase load-to-use latency at high memory bandwidth utilization (blue line in Figure 1). This happens due to longer queuing delays in the memory system and inefficiencies introduced by inaccurate prefetches. In the low memory bandwidth utilization regime (left side of the graph in Figure 1), where hardware prefetchers do not impact memory latency, it makes sense to optimize for cache miss rates. But in the high memory bandwidth utilization regime (right side of Figure 1), where large fleets

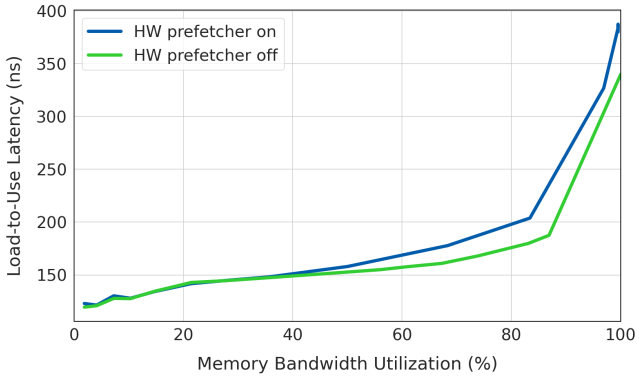


Figure 1. Average load-to-use latency per DRAM request reduces by 15% when hardware prefetchers are disabled. Data is gathered using the Intel MLC tool [10].

typically operate during peak traffic, we need new prefetching solutions that provide a better tradeoff between cache miss rates and memory latency.

Our key insight is that vertical integration between hardware and software can provide a more favorable performance tradeoff and is a better prefetching solution at scale. In particular, we make the observation that in bandwidth-constrained regimes, inserting software prefetch instructions directly in application code is much more judicious and targeted compared to hardware prefetching. We also note that data center operators have high visibility across the stack—they can analyze both the microarchitectural characteristics of hardware and the applications running on that hardware—and thus, they have the unique ability to dynamically modulate between hardware and software prefetching based on runtime fleetwide telemetry.

More concretely, we present Limoncello, a software system that achieves this vertical integration without any hardware changes. At low memory bandwidth utilization, Limoncello relies on hardware prefetchers to minimize cache miss rates. When memory bandwidth exceeds a certain threshold, Limoncello disables hardware prefetchers to optimize for memory latency. Finally, Limoncello uses fleetwide profiling to guide software prefetch insertions into select library functions and maintain a low cache miss rate even at high memory bandwidth utilization.

One unique aspect of our solution is our methodology for identifying software prefetching targets. Traditional profile-guided methods to insert software prefetches [11–13] do not work at scale because they rely heavily on the availability of representative benchmarks and inputs for generating useful performance profiles. This is challenging in a large fleet with huge diversity in both workloads and inputs. Our methodology instead uses large-scale hardware ablation studies, where we turn off (and thus "ablate") hardware prefetchers on hundreds of thousands of machines in the Google fleet, and we employ fleetwide profiling tools [1, 2] to characterize the impact of disabling hardware prefetchers. In particular, we disable hardware prefetchers on our experimental machines and monitor the impact of disabling hardware prefetchers on the cache miss rates of different functions. Interestingly, we find that data center tax operations, such as copying, compression, and hashing, suffer the most when hardware prefetchers are disabled and are the most profitable targets for software prefetching. While these functions are named after data centers, they are widely used as building blocks in many distributed applications. We discuss the results of this characterization in Section 4.1.

Limoncello can be easily deployed at scale because it does not require new hardware support. We designed, implemented, deployed, and evaluated Limoncello over a period of several months in Google's data center fleet running tens of thousands of diverse workloads and with machines that each run several hundred services. From our evaluation in the Google fleet, we find that Limoncello improves memory latency at peak utilization by 15% and application throughput by 10%. More broadly, this paper highlights that traditional prefetching (both software and hardware) face significant challenges at scale, and it points to several new research directions in an otherwise mature research area.

In summary, this paper makes the following contributions:

- We demonstrate that when memory bandwidth is scarce, hardware prefetchers can hurt system performance by increasing memory bandwidth usage and memory latency.
- We show that **fleetwide profiling** along with a carefully-designed **hardware ablation** study can accurately surface opportunities for precise and fine-grained software prefetching. We find that data center tax functions are highly prefetch-friendly and make good targets for software prefetching.
- We show that hardware-software collaboration can provide a better prefetching solution than either hardware prefetching or software prefetching alone.
- We design, implement, deploy, and evaluate **Limoncello** in Google's fleet and show that it improves application throughput by **10%**.

The rest of this paper is organized as follows: We motivate Limoncello by analyzing recent trends related to memory bandwidth and hardware prefetching (Section 2). We then describe Limoncello's hardware component, which we call Hard Limoncello in Section 3, and then Limoncello's software component, which we call Soft Limoncello in Section 4. We present Limoncello's evaluation in Section 6, discuss insights and research directions in Section 8, and then conclude in Section 9.

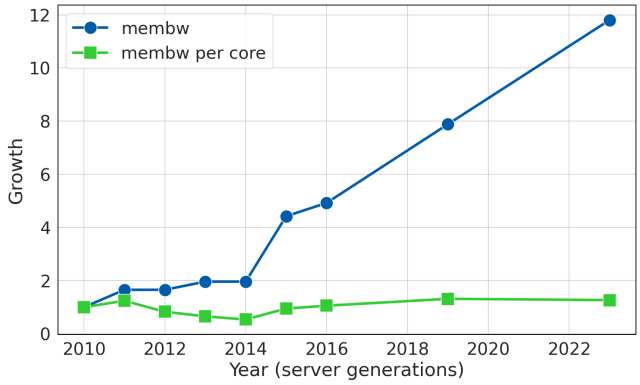


Figure 2. Memory bandwidth per core has plateaued over several generations of server CPUs.

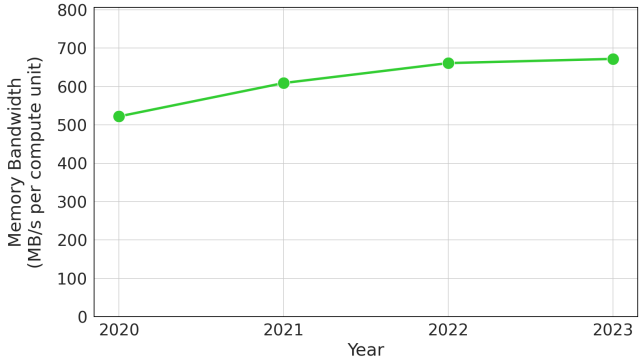


Figure 3. Average memory bandwidth usage of fleet workloads has increased over the last 4 years. Each point in the graph shows memory bandwidth usage per compute unit averaged across all workloads in the fleet.

2 Motivation

We now motivate Limoncello by discussing hardware trends with respect to memory bandwidth and data prefetching.

2.1 Hardware Trends

Figure 2 shows that memory bandwidth per core has stagnated over the last decade. Although the number of cores on servers has continued to increase, available memory bandwidth has been fundamentally limited by the number of pins [9]. DDR5 is likely to offer a one-time bump in memory bandwidth, but this trend is not expected to continue due to physical device limitations [9]. High-bandwidth memories (HBMs) [14] offer an attractive alternative, but their price point limits their widespread deployment in server CPUs.

At the same time, memory bandwidth usage of workloads has increased at a steady pace, as workloads have become more data-intensive. Figure 3 shows the average memory bandwidth usage of all workloads in the Google fleet for the last four years normalized by the amount of compute they use. The compute unit is defined as an abstraction of physical CPU cores such that it represents the same amount of computational power on any platform [15]. We see that since 2020 the average memory bandwidth usage has increased by nearly 1.4×, with an average 10% year-on-year increase.

Impact of memory bandwidth on server efficiency. To understand the impact of memory bandwidth on server efficiency and utilization, we study two state-of-the-art platforms in the Google fleet. The achievable memory bandwidth per core for these platforms is roughly 3 GB/s per core. Figure 4 shows that for these platforms, server CPU utilization—defined as the proportion of the total available processor cycles that are spent doing useful work—is limited by memory bandwidth. In particular, the figure shows the average memory bandwidth consumption (y-axis) for the servers in the fleet bucketed by their CPU utilization (x-axis). For optimal server utilization, the target CPU utilization for

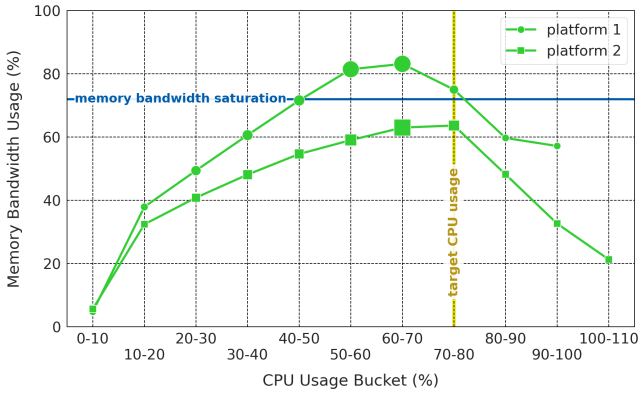


Figure 4. Memory bandwidth can saturate with just 50% CPU utilization in a bandwidth-bound platform. Sizes of the markers are in proportion to the number of platform servers in the CPU usage bucket.

the Google fleet is 70-80%. However, memory bandwidth gets saturated at 40-60% CPU utilization. When a server starts reaching memory bandwidth saturation, the cluster scheduler avoids scheduling workloads on the machine to prevent workloads from encountering performance cliffs due to memory bandwidth contention. As a result, many servers are unable to reach the target 70% CPU utilization, and CPU resources on servers remain idle due to lack of sufficient memory bandwidth.

Impact of hardware prefetchers on memory bandwidth. Hardware prefetchers contribute significantly to the memory bandwidth usage in the fleet. Table 1 shows that across the entire fleet, disabling hardware prefetchers reduces average memory bandwidth by 11.2%–15.7%, and it has a significant impact on the peak and P99 tail memory bandwidth consumption as well.

Memory Bandwidth Reduction	Platform 1	Platform 2
Average	15.7%	11.2%
P99	10.4%	2.8%
Peak	5.6%	5.5%

Table 1. Disabling hardware prefetchers reduces both average and tail memory bandwidth.

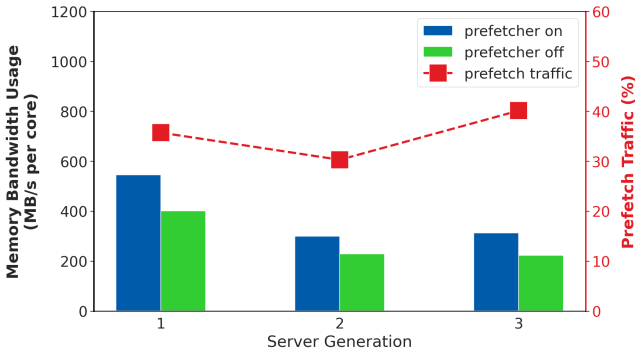


Figure 5. Memory bandwidth usage on SPEC with and without hardware prefetching over 3 generations of a server platform. We see a 30-40% increase in memory bandwidth usage when hardware prefetching is enabled.

To better understand why hardware prefetchers increase memory bandwidth usage, we profile the memory bandwidth usage of SPEC workloads over three generations of a leading server platform in the Google fleet. Figure 5 shows that the memory bandwidth usage of running SPEC increases by 30-40% when hardware prefetchers are turned on. Furthermore, the memory bandwidth overhead of prefetchers in the latest generation has increased to 40% compared to the 30% overhead in the previous generation.

One factor contributing to this growth is aggressive hardware prefetching. The trend of increasing hardware prefetch traffic reflects that hardware vendors optimize for prefetch coverage to obtain high cache hit rates under conditions that are not memory-bandwidth constrained (the left half of the latency curve in Figure 1). Consequently, prefetching algorithms have evolved to increase coverage at the expense of memory traffic. Data centers, on the other hand, operate under a different regime (the right half of the latency curve in Figure 1). Prefetchers, especially aggressive ones, exacerbate the pressure on the memory controller. Furthermore, prefetch requests from one workload can interfere with the demand requests from other co-located workloads, adding even more pressure on the memory system.

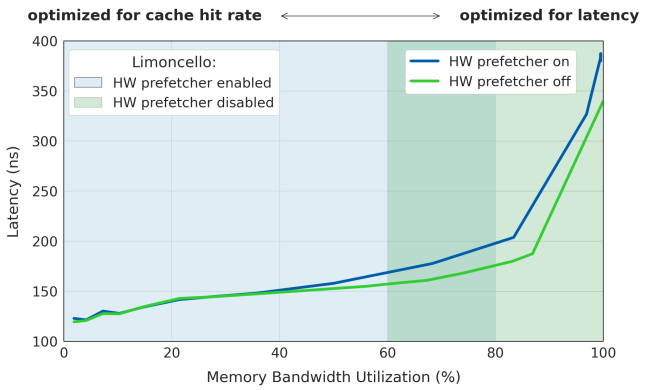


Figure 6. Limoncello disables hardware prefetchers when memory bandwidth utilization is high to optimize for memory latency.

3 Hard Limoncello

Limoncello disables hardware prefetchers for a machine socket when the machine load is high. In particular, when the memory bandwidth utilization on the socket exceeds a pre-defined upper threshold, we disable hardware prefetchers, and when the memory bandwidth utilization falls below a pre-defined lower threshold, we re-enable hardware prefetchers. As show in Figure 6, this optimizes cache hit rate when memory bandwidth utilization is low, and memory latency when it is high. The thresholds for disabling and enabling hardware prefetchers were determined through fleetwide experimentation and analysis of last-level cache (LLC) miss latency curves. Note that when hardware prefetchers are off, not only is memory latency reduced, but the amount of memory bandwidth that the socket supports before becoming exponential is also higher.

Telemetry. Our solution is based on socket-level memory bandwidth telemetry that is collected every 1 second. We use the perf tool to profile memory bandwidth levels on every socket every 1s and redirect the result to our software controller.

Design. Figure 7 shows a time series of memory bandwidth measurements taken every minute from a representative machine in the data center. As demonstrated in the figure, memory bandwidth can be a volatile metric that is difficult to predict. Reacting to short bursts in memory bandwidth can be counterproductive as we would constantly be toggling prefetchers on and off, which may lead to unstable performance and poor prefetching behavior.

To allow for smoother transitions, we introduce hysteresis in our software controller in two ways: First, we define separate upper and lower thresholds. The upper threshold is used to determine whether prefetchers should be disabled, and the lower threshold is used to determine whether prefetchers should be enabled again. Second, we define a time duration

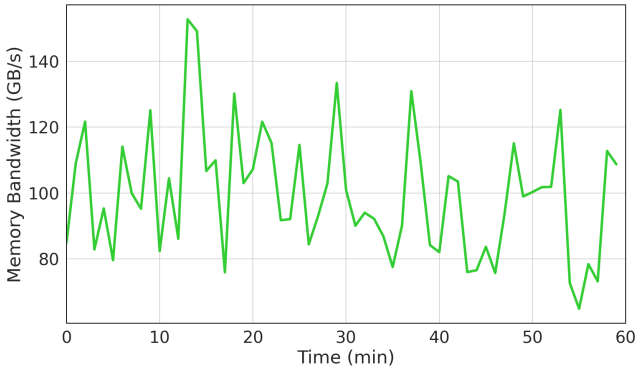


Figure 7. Memory bandwidth variability in a machine.

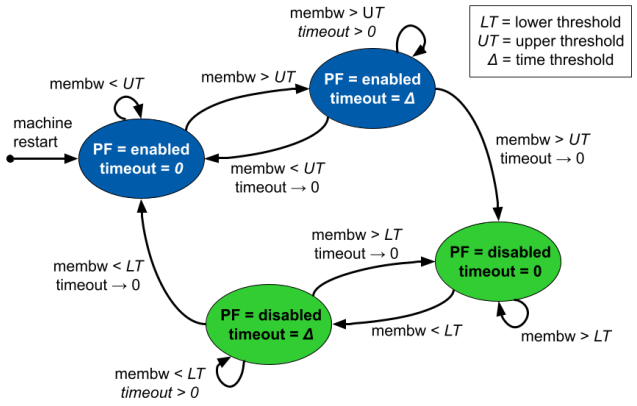


Figure 8. State diagram of Hard Limoncello controller for modulating hardware prefetchers. Green (lighter) indicates hardware prefetchers are off, blue (darker) that they are on.

during which memory bandwidth must stay above the upper threshold or below the lower threshold before the controller will change the state of the prefetchers. The overall flow is encapsulated in the state machine in Figure 8.

As a concrete example, consider a socket with the memory bandwidth utilization profile shown in Figure 9. Consider the upper threshold to be 80% memory bandwidth utilization and lower threshold to be 60% memory bandwidth utilization. As the memory bandwidth crosses the upper threshold for a sustained duration of time at t=0, we disable prefetchers (denoted in red). Note that prefetchers are not turned back on although memory bandwidth falls below the upper threshold at t=7.5 because it does not fall below the lower threshold. However, at time t=10, memory bandwidth falls below the lower threshold for a sustained period of time, so we enable the prefetchers. Before t=20, the prefetchers are left on even though memory bandwidth exceeds the lower threshold because it does not exceed the upper threshold.

Actuating Prefetcher Controls. The controller in Limoncello enables and disables hardware prefetchers by writing to

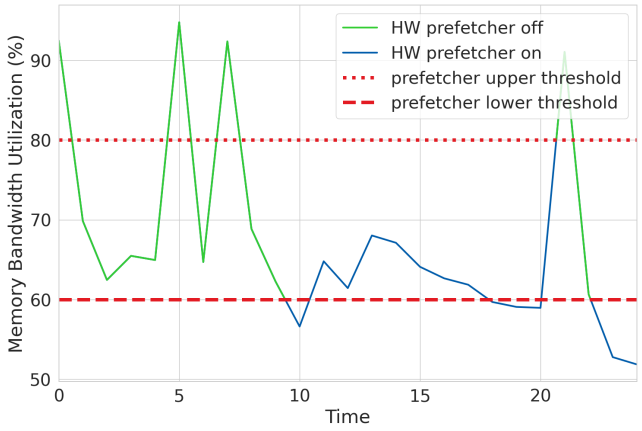


Figure 9. Hardware prefetcher state over time. The green lines in the figure show when hardware prefetchers are disabled. When machine load exceeds an upper threshold, hardware prefetchers are disabled and remain disabled until machine load returns below a lower threshold.

the model-specific registers (MSRs) for prefetchers. The register addresses and values vary for different vendors/platforms. For a given platform, we disable all prefetchers in the platform.

Thresholds. To identify the upper and lower thresholds for Hard Limoncello, we run a hardware ablation study on tens of thousands of machines in the data center. In particular, we create two sets of 10k experimental and 10k control machines; the experimental group runs Hard Limoncello, where the control group has all hardware prefetchers enabled and no software prefetchers added. The machines are chosen such that they lie across several geographical zones and a diverse set of workloads to ensure representative serving traffic. The experiment is run continuously over two weeks, so as to capture variations during the day as well as variations across days of the weeks.

We define the upper and lower thresholds as a percentage of the platform's memory bandwidth saturation threshold. Memory bandwidth saturation is defined as follows: For each machine socket, we define a memory bandwidth saturation threshold as the maximum memory bandwidth capacity for the machine and that we establish during a machine qualification process. Running at higher bandwidth than this threshold increases memory latency significantly.

As shown in Figure 10, we examined various lower and upper memory bandwidth thresholds for Hard Limoncello chosen with memory bandwidth saturation thresholds in mind by analyzing application performance trends.

4 Soft Limoncello

When hardware prefetchers are disabled with Hard Limoncello, some functions are expected to increase in cache misses.

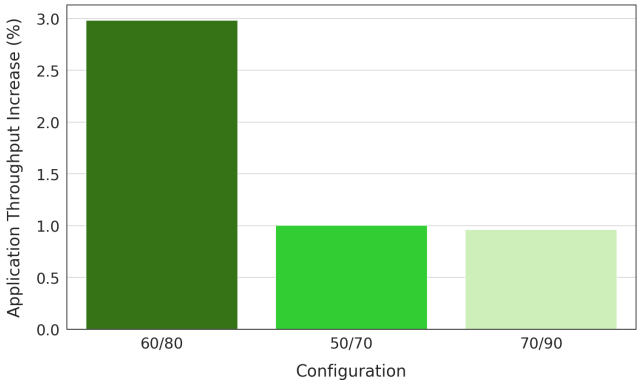


Figure 10. Application throughput based on different Hard Limoncello configurations. X/Y indicates the lower (X%) and upper (Y%) memory bandwidth thresholds in the configuration. Thresholds are expressed as a percentage of memory bandwidth saturation.

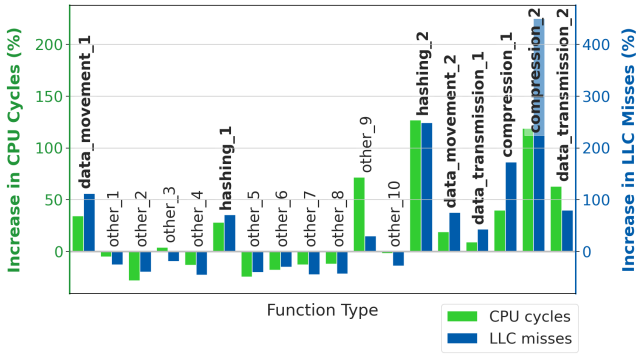


Figure 11. Change in CPU cycles (%) from Hard Limoncello. Functions that regress in performance when hardware prefetchers are disabled show an increase in CPU cycles (green) and an increase in LLC MPKI (blue). Data center tax functions in particular show performance regressions from disabling hardware prefetchers while other functions tend to improve in performance and decrease in cycles and MPKI.

Soft Limoncello addresses this by using careful insertion of software prefetches. Inserting software prefetches everywhere is not feasible due to its lack of scalability for the large range of applications running in a large data center fleet. We instead target specific software prefetch-friendly functions and we describe our process for identifying software prefetching targets and inserting the prefetches below.

4.1 Software Prefetching Targets

To determine software prefetching targets for Limoncello, we analyze function performance using the ablation study described in Section 3. During the experiment, we leverage a global profiling tool that continuously captures the program counter, CPU usage, and LLC misses of all applications in the fleet. The profiler samples a limited number of random machines at any given time and it is activated only for small time intervals to reduce profiling overhead. Although the profiler only runs for brief periods, the fleet is large enough such that aggregated samples can effectively capture the impact of code changes over a long period of time.

Profiling the machines in the experiment group when hardware prefetchers are disabled provides us with performance data at a function granularity in the absence of hardware prefetchers. Simultaneously profiling machines in the control group provides us with data in the presence of hardware prefetchers. This helps us generate a detailed snapshot of code sections that experience an increase in CPU usage and LLC misses from disabling hardware prefetchers. By leveraging this profiling data, we identify where we should introduce software prefetchers.

The results are mixed: For a few of our workloads, we see significant performance improvements. For example, disabling hardware prefetchers results in a >10% QPS gain in a memory-bound search application, a >30% improvement of QPS in an ML model server, and >1% throughput increase in a database server. However, disabling hardware prefetchers also hurt performance for other workloads and produces an average 5% performance drop in our fleet.

We analyze these regressions more closely by examining our profiling data at a function level and looking at LLC misses per kilo instructions (MPKI). Figure 11 shows the impact of disabling hardware prefetchers on several hot functions in the data center. Profiling data reveals that when hardware prefetchers are disabled, many functions regress in performance (increase in CPU cycles) due to a significant increase in LLC MPKI. We consider these functions to be prefetch-friendly. Not surprisingly, several other functions also gain performance (decrease in CPU cycles) due to lower memory latency and less cache pollution. We consider these functions to be prefetch-unfriendly: for these functions, the hardware prefetcher may have difficulty accurately predicting memory access patterns, leading to untimely or incorrect prefetches and wasted memory bandwidth.

Our function-level profiling data indicates that many of the prefetch-friendly functions are data center tax functions, common leaf functions that can be attributed to data center overhead. These operations account for 30-40% of the cycles in large data centers [1, 2]. From the profiling data, we identify the following functions as hardware-prefetch friendly and worthwhile targets for software prefetching:

1. Compression Many compression and decompression libraries are optimized for speed and are extremely memory latency sensitive. Prefetching can help hide this memory latency. When operations in these libraries are done over blocks, they access memory in contiguous patterns that are prefetch-friendly.

Figure 12. Aggregated change in CPU cycles (%) from Hard Limoncello. Data center tax functions (green) increased in CPU cycles under Hard Limoncello. In contrast, overall non-data center tax functions (blue) decreased in cycles.

2. Data transmission Serializing and deserializing data for RPCs are actions that often involve either copying from or writing to addresses in a predictable manner. In certain settings, the data may also be stored contiguously in memory. This can lead to memory access patterns that are both easily detectable by hardware prefetchers and possible to mimic with software prefetchers.

3. Hashing Like compression, hashing commonly involves data processing over block sizes. Hashing algorithms manipulate data in predefined sequences. This leads to predictable memory-access patterns that a hardware prefetcher can effectively prefetch.

4. Data movement Functions performing data movement like `memcpy` and `memmove` involve actions over contiguous sections of memory. These memory access patterns are easily detected by a hardware prefetcher, especially when the data movement is done over a large stream.

Figure 12 shows the overall changes in cycles comparing compression, data transmission, hashing, and data movement with other functions in the fleet. While each of the data center tax categories highlighted show significant increases in cycles with hardware prefetchers disabled, other functions in the fleet together show a performance improvement. Of course, some non-tax functions also regress with hardware prefetchers disabled, but many of these functions are not hot enough to warrant standalone optimizations.

Data center tax functions are particularly amenable to software prefetching for several reasons. First, they are well-contained in library functions and account for a disproportionately large number of fleet cycles. Second, their function definitions provide direct knowledge of what will be accessed and how much data will be accessed. In particular, each function performs computations over a stream of sequential data and reads data from a source, writes data to a destination, or both. For such functions, hardware prefetchers require a warm-up period to learn the access patterns and often lack knowledge about when to stop prefetching, which is problematic when the streams are short. With software prefetching, we can be much more precise as we know the exact addresses we want to prefetch, and we also know how much data should be prefetched.

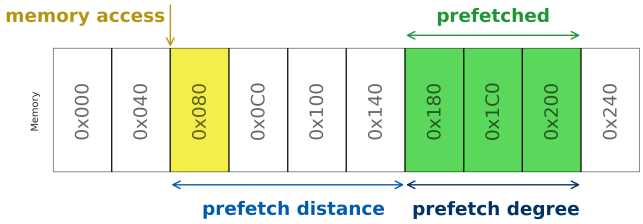


Figure 13. Software prefetch instructions act on addresses at a distance and can prefetch different degrees of data.

4.2 Software Prefetcher Design Space

Many design parameters need to be carefully considered when inserting software prefetches in a piece of code. In Soft Limoncello, we focus on three key design parameters:

- **Prefetch address**: Determining the exact location to software prefetch is an important design decision. If a prefetch is inserted for data that is not actually used, it can lead to wasted prefetches. Similarly, not all loads provide enough context for effective prefetching and adding in software prefetches for loads arbitrarily can lead to untimely prefetching.
- **Prefetch distance**: Prefetch distance refers to how far in advance data is fetched. Inserting the prefetch too close to the actual load will result in an untimely prefetch, whereas inserting the prefetch too far could lead to a stale prefetch. For example, in Figure 13, address 0x180 is prefetched when address 0x800 is loaded, resulting in a prefetch distance of 4 cache lines. A short prefetch distance has a higher chance of being accurate but will be less timely, whereas a long prefetch distance is more likely to be timely but has a higher chance of being inaccurate.
- Finally, the third consideration is **prefetch degree**, which refers to how much data is prefetched at once. A small degree results in less prefetch traffic but could lead to lower prefetch benefit, whereas a large degree could lead to greater inaccuracy.

Since small, short data access spread over a region do not benefit as much from hardware prefetching and are difficult to software prefetch, we target larger streams of data access which we identify by observing trends in the size of function call arguments. Using a fleetwide profiling tool, we gather the call argument sizes for certain functions as well as cache misses and CPU cycles at an instruction level. For functions with multiple potential prefetch sites, we look at the instruction-level metrics to identify prefetch addresses that

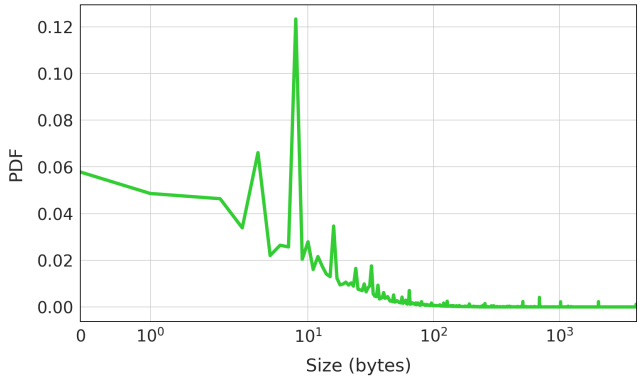


Figure 14. `memcpy` argument size distribution. The chart shows the probability density function (PDF) of the number of times each copy size appears in the profiling data. Most copy sizes are small.

have the highest potential for improved performance. When prefetch-friendly addresses are function call arguments, we examine function call argument distributions in the profiling data to help determine prefetch distances and degrees.

Since the effectiveness of software prefetching is tied to application memory access patterns, we carefully choose a series of microbenchmarks and load tests for testing software prefetches for Limoncello. Some of these benchmarks reflect the memory access patterns of our fleet, like Fleetbench [16], while others exercise a range of memory bandwidth usage levels to ensure testing under load and capturing both conditions where hardware prefetchers are enabled and where they are disabled.

We first use these benchmarks to sweep a chosen set of prefetching addresses, distances, and degrees. Then we select the best performing parameters for load testing to determine performance improvements. If either microbenchmarks or load tests fail to return positive performance improvements, we choose a different set of prefetching addresses, degrees, or distances for testing. By iterating on software prefetch strategies and testing, we are able to identify software prefetches that allowed us to remove application regressions initially seen in the ablation experiment of Limoncello. (See Section 6.)

Because we target data center tax functions with well-understood memory access patterns, we expect our inserted software prefetches to remain effective even as our data center fleet evolves. Regardless, our system continually profiles workloads and monitors CPU cycles and cache hit rate, and our profiling and experimental harness for fleetwide performance studies can be used to make adjustments to software prefetches if necessary.

4.3 An Example for Clarity: `memcpy`

First, we note that `memcpy` was one of the data center tax functions identified by fleetwide profiling during hardware ablation as a prefetch-friendly function (Section 4.1). This is not surprising, as `memcpy` copies data from a source to a destination, iterating over a contiguous piece of memory. `memcpy` is also very suitable for software prefetching since we can identify the locations that will be accessed by `memcpy` directly in software. This means that we can perform software prefetching without over-prefetching and we only need to determine proper prefetch degrees and distances to perform timely prefetches.

From our profiling data, we also note that data movement functions are on average done on short streams. Figure 14 shows that most copy calls are short, but there is a long tail of large copies. Our profiling data shows that workloads experiencing a performance regression from hardware prefetchers disabled have on average 26% larger `memcpy` sizes than non-regressing workloads. Since hardware prefetchers require a warmup period, it is likely easier for the hardware prefetcher to recognize and issue timely prefetches for larger copies.

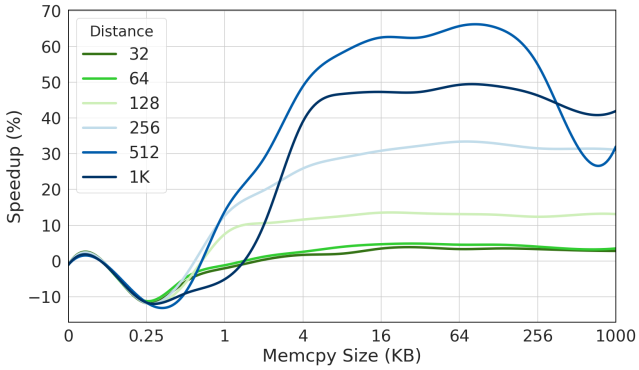
Following this data, we chose to focus on the larger `memcpy` sizes for inserting software prefetchers. As Figure 15 shows, the optimal prefetching distance and degree can vary based on call size. Conditioning software prefetching on larger call sizes for `memcpy` allowed us to ensure prefetches are timely enough. We used a few microbenchmarks that exercise a variety of different `memcpy` size distributions, such as the LLVM libc microbenchmark suite [17], and we searched for the optimal prefetching degree and distance by sweeping a range of degrees and distances, fixing the prefetch degree while varying the prefetch distance (Figure 15a) and fixing the prefetch distance while varying the prefetch degree (Figure 15b). After running the microbenchmarks, we verified results against multiple loadtests.

5 Methodology

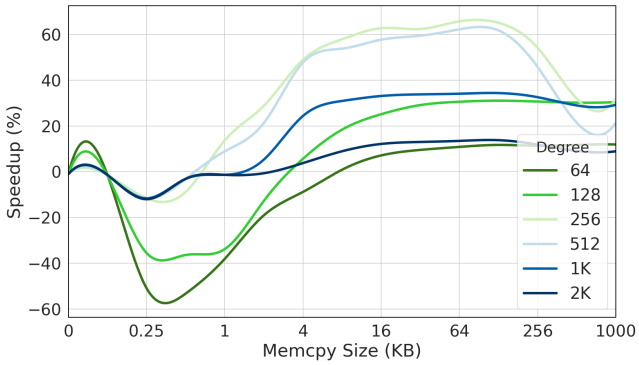
We deployed Limoncello in Google's production fleet. Each machine in the fleet runs hundreds of services that are scheduled using a centralized scheduler, and Limoncello was evaluated on production traffic. **Platform 1** and **Platform 2** are two different generations of large x86 out-of-order multicores.

Hard Limoncello. Using the hardware ablation study results described in Section 3, we set upper and lower memory bandwidth thresholds for Hard Limoncello as 80% and 60% of the platform's memory bandwidth saturation threshold. We modulated our hardware prefetchers using a controller as described in Section 3.

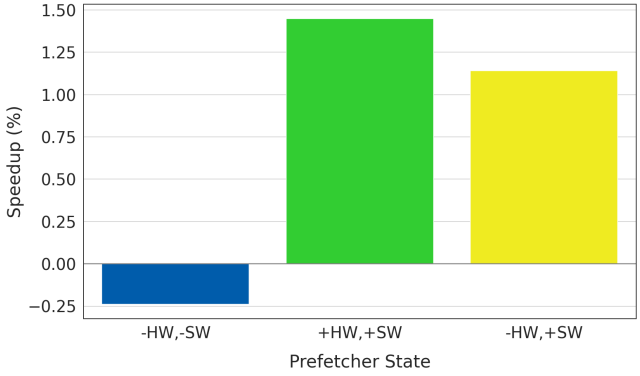
Soft Limoncello. We used profiling data from the ablation study to identify software prefetching targets and inserted



(a) Microbenchmark exercising different prefetch distances. Prefetching is fixed at a degree of 256 bytes.



(b) Microbenchmark exercising different prefetch degrees. Prefetching is fixed at a distance of 512 bytes.



(c) LLVM-libc microbenchmarks. +HW/SW and -HW/SW indicates benchmarks run with and without prefetchers, respectively. Speedup is measured relative to +HW,-SW

Figure 15. Soft Limoncello `memcpy` microbenchmarking.

software prefetchers as described in Section 4. To map performance metrics to code structures, we leveraged the global profiling tool to capture the program counter, CPU usage, IPC, LLC misses of all applications in the fleet.

Metrics. We report both machine level and workload-level metrics. At the machine level, we report memory bandwidth, DRAM latency reduction, and CPU utilization. CPU utilization is a good proxy for server utilization as a higher utilization indicates more useful work being done. At the workload level, we report application throughput for all workloads running in the fleet. Application throughput refers to the number of requests served by each service per unit of time. We report application throughput in different CPU utilization bands for completeness.

6 Evaluation

Due to the size of the fleet, we rollout Limoncello to the entire fleet over a period of a few weeks. Figures 16, 17, and 18 provide a comparison of average fleetwide performance metrics before the rollout, when Limoncello had not been applied, and after the rollout, when both Hard and Soft Limoncello were in full effect.

Figure 16 shows application throughput over three different CPU utilization bands (60%, 70%, 80%). We see that Limoncello significantly improves performance at high utilization—the desired operating point. In particular, at peak utilization (70% and 80% utilization), Limoncello improves throughput by 10%. At moderate utilization (line with 60% usage), Limoncello does not degrade performance due to its dynamic modulation of hardware prefetching

To understand Limoncello's performance improvements at high utilization, realize that when utilization is high, application performance is bottlenecked by memory bandwidth. Applying Limoncello reduces the memory bandwidth and lowers the latency to memory as shown in Figure 17. In particular, we saw a 13% reduction in median L3 latency and a 10% reduction in P99 L3 latency.

Figure 18 shows the impact in socket memory bandwidth at different percentiles. We observe a clear 15% reduction in memory bandwidth, with the number of saturated sockets falling by nearly 8%.

In Section 2.1, Figure 4 showed how the data center hits a CPU utilization ceiling when sockets are saturated. This starts occurring at the 40% CPU utilization band. After deploying Limoncello, CPU utilization increases as shown in Figure 19 for the platforms evaluated. Memory bandwidth usage increases at a slower rate and does not hit saturation until the 70-80% CPU utilization band. This allows us to increase machine utilization while still delivering good application performance.

Through Soft Limoncello, workload regressions seen during ablation experiments were effectively removed. Unlike hardware prefetchers, software prefetchers do not need a warm-up period. Software also offers greater foresight into expected data patterns. As a result, software prefetching can enable much more precise prefetching than hardware prefetchers and make more judicious use of memory bandwidth. These characteristics allowed our targeted insertions of software prefetchers to maintain a low MPKI in data center tax functions when hardware prefetchers were modulated.

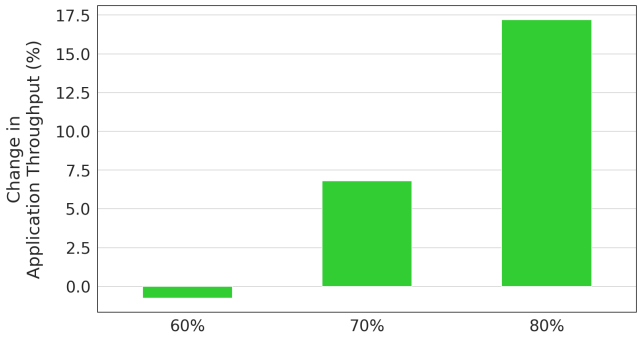


Figure 16. Limoncello application throughput gain. Application throughput increases by 6-13% increase, dependent on the CPU utilization level of the machines.

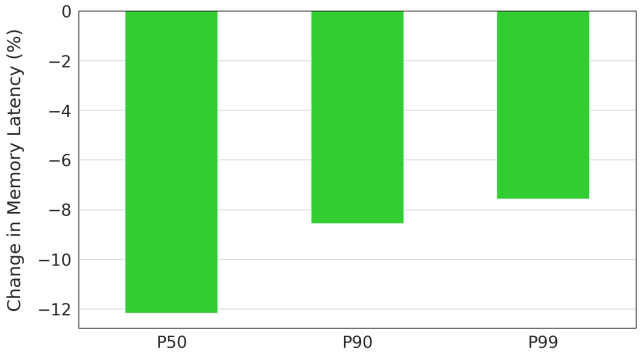


Figure 17. Limoncello memory latency reduction. Memory latency reduces by 13% in the median and 10% in the P99.

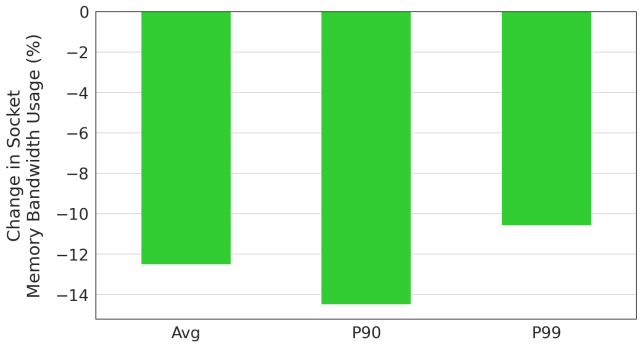


Figure 18. Limoncello socket bandwidth usage reduction. Average socket bandwidth reduces by 15%.

Figure 20 shows a breakdown of the improvement in CPU cycles for the four type of data center tax functions we targeted. The small increase in CPU cycles for the functions under Hard Limoncello reflects how useful hardware prefetchers are for these functions. However, the Soft Limoncello bars indicate that software prefetchers are extremely beneficial for the functions when operating in low memory bandwidth

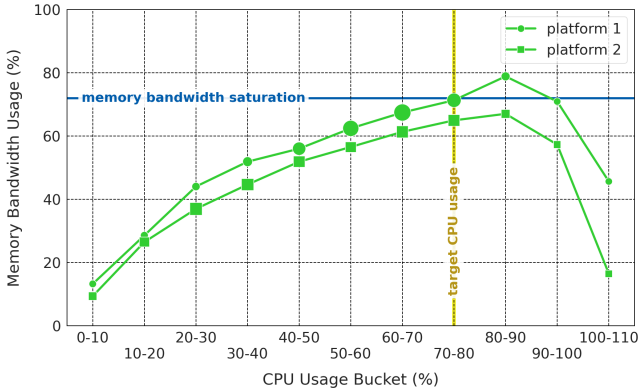


Figure 19. Increase in CPU utilization due to Limoncello. Before Limoncello, memory bandwidth saturation was reached in the 40-50% CPU utilization band (Figure 4), but with Limoncello memory bandwidth saturation is not met until the 70-80% band. Sizes of the markers are in proportion to the fraction of platform servers in the CPU usage bucket.

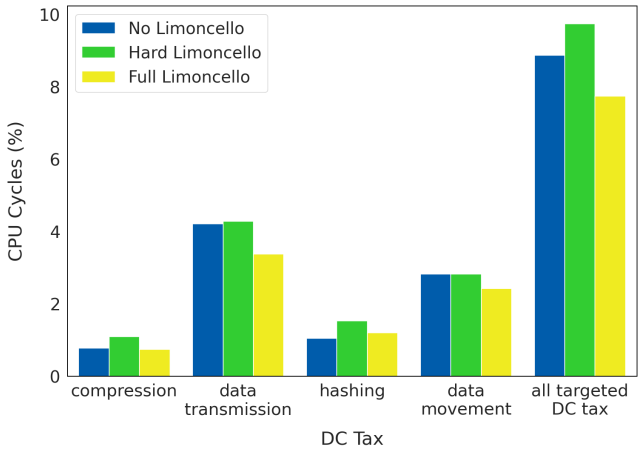


Figure 20. Software prefetcher impact in Limoncello. The y-axis shows the portion of fleetwide cycles spent in the respective function categories. The center bar (green) shows Hard Limoncello deployed without any software prefetchers. Adding software prefetchers into Limoncello lowered CPU cycles spent in targeted functions by 2% (yellow).

systems. If we look at an aggregated view of all targeted data center tax functions in Figure 20 we observe that the trend remains with an increase in cycles due to Hard Limoncello and a significant reduction when combined with Soft Limoncello.

Together, Hard and Soft Limoncello led to a higher capacity in the fleet. Our results demonstrate that much of the original data being prefetched by the hardware prefetchers harmed more than it benefited our large data center workloads. While the hardware prefetchers were designed to decrease memory latency, the retrieval of data excessively

congested memory traffic and lowered overall system performance. Turning the hardware prefetchers off under high memory bandwidth saturation reduced congestion, and inserting software prefetchers maintained lower MPKI for prefetch-friendly functions. By combining hardware and software techniques, Limoncello improved the performance of our data center fleet.

7 Related Work

Prefetching is a well-studied research area rich in literature. Since this paper is about hardware-software collaboration for more improving prefetch efficiency, we break down prior work into three broad categories: (1) hardware throttling, (2) software prefetching, and (3) hardware-software co-design. We note that most prior work involving hardware-software co-design requires changes to the hardware or the ISA. Limoncello avoids this entirely by taking a software-based approach to modulating hardware.

7.1 Hardware Prefetcher Throttling

Most hardware prefetch throttling mechanisms rely on bandwidth contention and prefetch accuracy for modulating hardware prefetcher aggressiveness [18–20]. These solutions tend to be reactive and coarse-grained, as they enable or disable prefetchers for all code irrespective of their relative importance. As a result, they are not effective for workloads where prefetch-friendly and prefetch-unfriendly code is interleaved at a fine granularity. In contrast, Limoncello leverages software prefetching to prefetch for a few code segments that are critical for overall performance, enabling proactive and finer-grained prefetch control.

More recent work in the hardware literature has considered finer-grained solutions [6, 7]. For example, PPF uses a perceptron filter and a range of program features to predict when to prefetch and when not to. CLIP predicts the criticality of load addresses and throttles hardware prefetchers for non-critical loads. Both solutions aim to eliminate inaccurate prefetches, but they are reliant on the accuracy of the underlying hardware prefetcher to decide whether to prefetch the targeted code segment or not. If the hardware had moderate accuracy for certain code segments, data for those code sites would not be prefetched, leading to a loss in coverage. Limoncello differs from these prior works as it leverages software prefetching, which is inherently more accurate, as software has knowledge about how much data is going to be accessed. As a result, Limoncello is able to achieve a superior coverage and accuracy tradeoff and is not limited by the underlying hardware prefetching algorithm's effectiveness. Moreover, Limoncello is a completely software-centric approach which does not require invasive hardware changes.

7.2 Software Prefetching

Software prefetching [21] has commonly been approached through the compiler. Static code analysis performed by compilers is used to identify simple memory access patterns and find targets for software prefetching [22–25]. A key challenge with software prefetching is to identify when and where to insert a software prefetch instruction. Compiler-based methods have been effective in identifying prefetch locations and timing for simple memory access patterns [26–29]. Further work has also been done to examine ways to prefetch more complex data structures, such as linked data structures [30–33]. However, static analysis based prefetch methods are inherently limited due to lack of dynamic execution information.

Profiling-based methods instead use runtime information to both identify prefetching candidates and improve prefetch timeliness [11–13]. Limoncello builds on this line of work but instead of using benchmarks to generate profiles, it uses fleetwide profiling tools [1, 2] alongside hardware ablation studies to determine prefetch locations and strides. Moreover, unlike prior work, Limoncello combines software prefetching with hardware prefetching such that hardware prefetchers are enabled in regimes they can provide the most benefit in, and software prefetches can provide targeted coverage in regimes where the hardware prefetcher is expensive.

7.3 Hardware-Software Codesign

Most prior work in designing cross-stack solutions for prefetching involve building programmable prefetching engines for specific data structures and require invasive hardware, compiler and/or ISA changes. For example, Ainsworth et al. design a programmable prefetch engine for graph applications in particular [34, 35], and Lebeck et al. design prefetchers for linked data structures [22, 36]. Limoncello offers a simpler solution and can be more readily implemented and deployed because it requires no hardware, ISA or compiler changes. Furthermore, Limoncello is more general as it is not tied to any particular data structure.

Another line of work looks at software-directed hardware prefetching. Most of these solutions require ISA support and compiler modifications. The work most closely related to our work is by Wang et al., and it involves using compiler hints to modulate the aggressiveness of a hardware prefetcher [37]. Other work leverages software to provide information about strides and addresses to prefetch [38, 39]. Finally, Zhang and Torrellas use the compiler to mark blocks in memory as belonging to contiguous spatially local regions or containing indirection pointers [40].

To the best of our knowledge, no current work effectively combines hardware and software prefetchers without requiring hardware modifications. Limoncello uniquely pairs existing hardware and software interfaces together, dynamically disabling hardware prefetchers and inserting software

prefetchers to outperform hardware and software prefetchers alone.

8 Discussion

Reflecting on our results, this work offers several insights and suggests interesting avenues for future research.

8.1 Improving Hardware Prefetching

While hardware prefetchers have become increasingly sophisticated, they face unique challenges in the data center environment. Recent literature in hardware prefetching has embraced aggressive and complex prefetching designs, with designers often prioritizing miss coverage over memory traffic. This design philosophy is ill-suited for data center workloads, where prefetchers have to compete with other useful work for memory bandwidth. For example, available memory bandwidth could be used to prefetch for an existing workload, or it could be used to run another thread or application that does useful work.

This suggests two avenues for improving hardware prefetchers. First, designs that make accuracy a first-class concern would be more efficient and well-suited for data center environments. Second, designs that are more resilient to frequent context switching and inter-core communication are also likely to work better in the fleet.

8.2 Improving Software Prefetching

Software prefetching is a task that must be done with extreme care. If prefetches are not inserted well, they will waste instructions and memory bandwidth and hurt application performance. Inserting prefetches for Soft Limoncello required many days of extensive testing and careful tuning.

The toil of inserting software prefetches is largely due to two factors: (1) lack of visibility into application memory access patterns, and (2) lack of production-representative benchmarks for testing prefetches. Better visibility into memory layouts and memory access patterns can help with removing some of the guesswork in software prefetching.

Testing the effectiveness of software prefetching is also a difficult task. Some libraries today include microbenchmarks for assessing performance. However, few of these microbenchmarks reflect deployments in a data center. Research into developing microbenchmarks that can mimic behavior at scale would allow for ease of development in inserting software prefetches.

8.3 Better Hardware-Software Interfaces

Both hardware and software prefetching have unique strengths: Software prefetching allows developers to carefully choose what to prefetch, whereas hardware prefetchers are able to issue prefetch requests much more quickly and timely. Unfortunately, existing ISA interfaces do not allow for any communication between hardware and software prefetching; prefetching must be done in either hardware or software alone. Research into better hardware-software interfaces that allow for ease of collaboration between the two will undoubtedly lead to much more powerful and efficient prefetching systems.

9 Conclusion

In this paper, we have shown that at scale, memory bandwidth is a scarce resource, and as a result, hardware prefetching can result in performance degradation at high utilization. With Limoncello, we demonstrated that collaboration between hardware and software prefetching can provide a prefetching solution with much better tradeoffs than hardware or software prefetching alone. We deployed Limoncello in Google's datacenter fleet and demonstrated that it can improve application throughput by 10% for production workloads. More broadly, this paper highlighted that hardware-software co-design is a promising avenue for improving the performance of datacenter systems, where machine utilization is high and where workloads are very diverse.

10 Acknowledgements

Limoncello builds on the engineering work of many teams at Google, and we would like to especially thank Eric Zhang, Ashish Naik, Gaurang Upasani, Bhuvan Sajja, Ilya Tokar, Shiyu Hu, Srividhya Balaji, and Xiangling Kong for their contributions. We would also like to thank Urs Hölzle, Tipp Moseley, Rama Govindaraju, our anonymous reviewers, and our shepherd for their valuable feedback.

References

[1] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015.

[2] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–750, 2020.

[3] Babak Falsafi and Thomas F Wenisch. Data prefetching. In *A Primer on Hardware Prefetching*, pages 15–37. Springer, 2014.

[4] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480. IEEE, 2016.

[5] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[6] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Perceptron-based prefetch filtering. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2019.

[7] Biswabandan Panda. Clip: Load criticality based data prefetching for bandwidth-constrained many-core systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*,

MICRO '23, page 714–727, New York, NY, USA, 2023. Association for Computing Machinery.

[8] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 996–1008, 2019.

[9] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, pages 171–243. Springer, 2022.

[10] Thomas Willhalm Sri Sakthivelu Sharanyan Srikanthan Vish Viswanathan, Karthik Kumar. Intel® memory latency checker v3.11. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2021.

[11] Jiwei Lu, H. Chen, Rao Fu, Wei-Chung Hsu, B. Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 180–190, 2003.

[12] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. Profile-guided post-link stride prefetching. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, page 167–178, New York, NY, USA, 2002. Association for Computing Machinery.

[13] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 747–764, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017.

[15] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *EuroSys'20*, Heraklion, Crete, 2020.

[16] Google. Fleetbench. https://github.com/google/fleetbench, 2023.

[17] LLVM-libc. Libc mem* benchmarks. https://github.com/llvm/llvm-project/tree/main/libc/benchmarks, 2023.

[18] Biswabandan Panda. Spac: A synergistic prefetcher aggressiveness controller for multi-core systems. *IEEE Transactions on Computers*, 65(12):3740–3753, 2016.

[19] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, page 63–74, USA, 2007. IEEE Computer Society.

[20] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 316–326, 2009.

[21] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 40–52, New York, NY, USA, 1991. Association for Computing Machinery.

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

[23] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen-mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In Yashwant K. Malaiya, editor, *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 24, Albuquerque, New Mexico, USA, November 18-20, 1991*, pages 69–73. ACM/IEEE, 1991.

[24] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, page 128–142, New York, NY, USA, 1990. Association for Computing Machinery.

[25] Seung Woo Son, Mahmut Kandemir, Mustafa Karakoy, and Dhruva Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. *SIGPLAN Not.*, 44(4):209–218, feb 2009.

[26] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[27] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. A case for resource efficient prefetching in multicores. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, BRACIS '14, page 101–110, USA, 2014. IEEE Computer Society.

[28] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Not.*, 27(9):62–73, sep 1992.

[29] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 210–221, 2002.

[30] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, page 62–73, Washington, DC, USA, 2002. IEEE Computer Society Press.

[31] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 222–233, New York, NY, USA, 1996. Association for Computing Machinery.

[32] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 115–126, New York, NY, USA, 1998. Association for Computing Machinery.

[33] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, page 111–121, USA, 1999. IEEE Computer Society.

[34] Sam Ainsworth and Timothy M. Jones. An event-triggered programmable prefetcher for irregular workloads. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 578–592, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–386, 2019.

[37] Zhenlin Wang, D. Burger, K.S. McKinley, S.K. Reinhardt, and C.C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 388–398, 2003.

[38] Tatsushi Inagaki, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming*

Language Design and Implementation, PLDI '03, page 269–277, New York, NY, USA, 2003. Association for Computing Machinery.

[39] Ibrahim Hur and Calvin Lin. Feedback mechanisms for improving probabilistic memory prefetching. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 443–454, 2009.

[40] Zheng Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 188–199, 1995.