Senior Project - CPSC 490
2024 Spring Semester

Student: Evan Kirkiles
Advisor: Scott Petersen

Senior project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Optimizing Web Delivery: Applying CPU Prefetching Techniques to CDNs

Evan Kirkiles
`evan.kirkiles@yale.edu`
Scott Petersen
`scott.petersen@yale.edu`

Abstract

Predictive prefetching is a technique allowing a cache to improve cache hit ratios by speculatively preparing itself with data that will be requested from the cache ahead of time. This paper explores the application of predictive prefetching techniques, traditionally utilized in CPU caches, to enhance web caching strategies within Content Delivery Networks (CDNs). By drawing parallels between CPU and web caching strategies, the research introduces two prefetching implementations aimed at minimizing request latency and optimizing content delivery. The first method employs HTML parsing to prefetch potentially needed resources based on static analysis of links in web pages. The second leverages historical traffic data to predict and load future requests dynamically based on real user site traversals. Through simulated user testing using Cloudflare's local **`wrangler`** development environment, the efficacy of these methods is evaluated against a traditional caching strategy. Results indicate that predictive prefetching can significantly reduce user-perceived latency and increase cache hit ratios on unvisited pages, particularly in environments with fluctuating or sparse traffic that are prone to low cache hit ratios. The findings suggest that integrating predictive prefetching into CDNs could improve the efficiency of web resource delivery for smaller customers, albeit with considerations for increased complexity and potential origin server load.

Senior Project - CPSC 490
2024 Spring Semester

Student: Evan Kirkiles
Advisor: Scott Petersen

Table of Contents

1. Background 1
1.1. What Is a Cache? 1
Cache Policies 2
Cache Hierarchies 2
1.2. Caching in the CPU 3
Cache Hierarchy 5
Cache Coherence 6
Speculative Execution 6
1.3. Caching on the Web 8
Cache Hierarchy 9
Cache Coherence 10
Predictive Prefetching 11
2. Methods 11
2.1. Basic Cache 11
2.2. Prefetcher: HTML Parser 12
2.3. Prefetcher: Analytics Predictor 12
2.4. Testing & Metrics 13
3. Results 14
4. Discussion 15
4.1. Improvements 15
4.2. Conclusion 16
5. References 17
6. Tables & Figures 18

1. Background

"There are only two hard things in Computer Science: cache invalidation and naming things."

– Phil Karlton

In the digital age, the paradigm of caching—preparing high-latency data ahead of time, ensuring it can be retrieved instead locally at low latency—is a problem of paramount importance. In the CPU, where uncached memory reads cost precious clock cycles compared to reads from registers or caches, the caching solution results in a complex architecture of multi-level hardware caches and specialized modules for predicting memory accesses ahead of time—in some cases even involving hardware-based perceptrons for better branch prediction[1]. On the web, caching has led to a huge economy of Content Delivery Networks (CDNs), where files and cloud functions are distributed across the globe, as close as possible to the end-user, to prevent lengthy network calls back to origin servers. This is not to mention the several other levels of web caching found in browsers and on the origin servers themselves. In both cases, on CPU and on the Web, the end goal is the same: minimize the distance and time between the user and the data they require. This section aims to provide a basic technical background into the rationale behind caches and the problems that caching solves inside the CPU and globally on the Web.

1.1. What Is a Cache?

The word *cache* was first used in computing in 1967 in the IBM Systems Journal while describing the System/360 Model 85, which used a hardware buffer in front of memory dubbed the "High Speed Buffer".[2] IBM found this name too long, and thus the term cache was adopted to refer to a high-speed memory buffer, from its original roots in the French *cacher* for "to store" or "to hide".[3]

The idea behind the cache is similar to its etymology: it acts as a local store for the results of an expensive operation, be it a memory read or a function call. In its most basic version, it is simply a key/value lookup table. When a user executes a request for the expensive operation through the cache, the following steps occur:

1. A **`cache key`** is computed based on the user's request.
2. The **`cache key`** is used to perform a lookup in the cache storage.
   2a. If a **`cache entry`** matches, it is returned. This is a **`cache hit`**.

1 D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, Monterrey, Mexico, 2001, pp. 197-206, doi: 10.1109/HPCA.2001.903263.

2 .. G. C. Stierhoff and A. G. Davis, "A history of the IBM Systems Journal," in IEEE Annals of the History of Computing, vol. 20, no. 1, pp. 29-35, Jan.-March 1998, doi: 10.1109/85.646206.

3 .. Merriam-Webster.com Dictionary, s.v. "cache," accessed April 20, 2024, https://www.merriam-webster.com/dictionary/cache.

2b. If no entry matches the cache key, this is a **`cache miss`**.

In case of a cache miss, some cache implementations call the expensive operation themselves, handling their own population. Otherwise, it is up to the process requesting data from the cache to decide whether or not and how to populate the missing entry.

Caches are generally evaluated with respect to three metrics: **`cache hit ratio`** (the percentage of cache lookups which are hits), **`cache size`** (how much data the cache can hold), and the cost involved in a cache miss. A larger cache can lead to a higher cache hit ratio, but is more expensive to implement and often slower, especially if built in hardware. A smaller cache experiences lower cache hit ratios, but can often be co-located with the process requesting the data and have more expensive hardware, drastically reducing the lookup time. The tradeoff between cache size and cache hit ratio is navigated based on the cost of a cache miss. For example, if the environment means cache misses are quite expensive, you might take the tradeoff of the cost of a larger cache to minimize the cost of the cache misses. In such scenarios, a multi-level cache is often used as a compromise between speed, size, and cost, to be described in a following section.

» Cache Policies

Whereas the range of the cached function's outputs is often infinite, the storage capacity of the cache is definitively finite. For this reason, caches must have rules about which entries to keep and which entries to purge when the cache fills to its maximum size. This is known as a **`cache eviction policy`**. Basic policies include:

1. Least Recently Used (LRU)
2. Least Frequently Used (LFU)
3. First-In-First-Out (FIFO)
4. Random Replacement

When a cache needs to fill a new entry, these policies are executed across the cache entries, and a cache entry is removed to make space for the new one. In some cases, the removed entry (the "victim") is moved to another cache with slower speeds known as the "victim" cache.

Some caches also allow for entry expiration, in which cache entries "expire", or become "stale," when a certain condition is met (typically, a maximum age) based on some property of the cached value. Stale entries are prioritized for eviction, but are not removed immediately, only marked as "dirty". Caches supporting stale entries are often configured to return a stale entry if present for a given cache key, performing revalidation for that cache key in the background after the stale entry has been returned.

» Cache Hierarchies

As we will see in caching on both the CPU and the web, when the cached resource spans a huge range of potential cache entries, it is essential to use a cache hierarchy, or multiple levels of caches, in order to speed up cache performance. In a multi-level caching setup, caches located closer to the end destination of the data ("higher level caches") are smaller and faster to read from, often because of hardware implementation, cost tradeoffs that don't scale, or simply by being nearer to the processor. Caches further away ("lower level caches") are larger and act as a fallback for misses in the higher level caches.

The relationship between the data in a higher-level cache and its next-lowest-level cache is known as a **`cache inclusion policy`**, and can offer different guarantees and potential optimizations based on the approach taken. The three policies are:

1. An **`inclusive`** policy, in which all entries in a higher level cache are also present in the lower level cache.
2. An **`exclusive`** policy, in which all entries in a higher level cache must not be present in the lower level caches.
3. A **`non-inclusive non-exclusive (NINE)`** policy, in which the contents of the lower level cache are neither strictly inclusive nor exclusive of the higher level cache.[4]

The inclusive policy (1) provides lower miss latency and useful guarantees about the contents of each cache, particularly important in **`cache coherence`** of parallel systems which share the lower-level caches but have their own private highest-level caches. The exclusive policy (2) provides the greatest cache space, as the total cache size is the sum of all cache level sizes (no data duplication), but uses more bandwidth in maintaining the exclusivity guarantee as blocks evicted from a higher level cache must populate the next-lowest level cache in waterfall. The NINE policy (3), without any inclusive or exclusive guarantees, uses by far the least bandwidth in maintaining itself, at the expense of guarantees regarding when a cache lookup can be determined to be a miss.[5]

1.2. Caching in the CPU

At its essence, a CPU is an instruction fetch-and-execute system. Its task is to read instructions from memory, decode them, and then perform some action based on the opcode and operands of the instruction, most commonly reading or writing to the same memory device it fetched the instructions from (at least, in the **`Von Neumann architecture`**[6], where instructions and data can be accessed in the same way—see Figure 1). Billions of instructions are executed by a CPU pipeline each second, scheduled out-of-order and sometimes even executed speculatively before an instruction is guaranteed to need execution.

4 .. Solihin, Yan (2016). Fundamentals of Parallel Multicore Architecture. Chapman and Hall/CRC. pp. 146–150. ISBN 9781482211184.
5 .. Culler, David; Gupta, Anoop; Singh, Jaswinder Pal (1999). Parallel Computer Architecture: A Hardware/Software Approach. San Francisco: Morgan Kaufmann Publishers. pp. 369–372. ISBN 1558603433.
6 .. J. von Neumann, "First draft of a report on the EDVAC," in IEEE Annals of the History of Computing, vol. 15, no. 4, pp. 27-75, 1993, doi: 10.1109/85.238389.

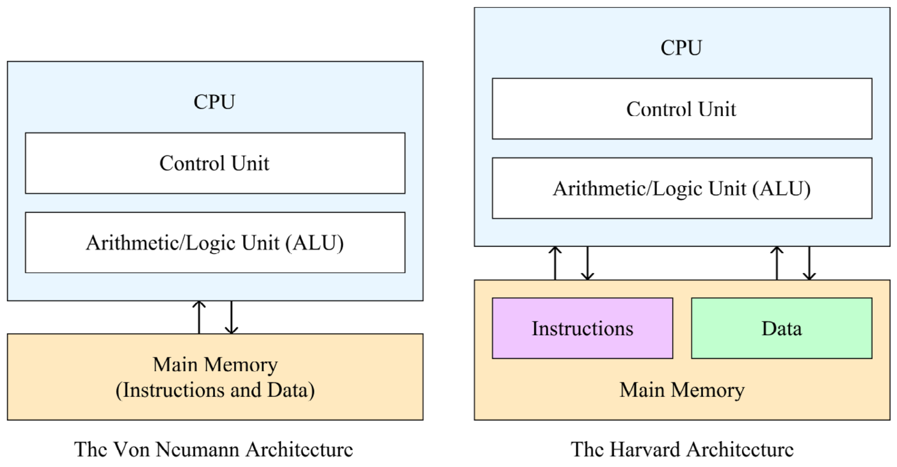


Figure 1: The Von Neumann Architecture (where instructions and data are accessed along the same patterns) vs. The Harvard Architecture (where instructions and data are accessed in different address spaces).

As both instructions and data are read from memory in a separate hardware device from the CPU, ensuring that the CPU can quickly read both instructions and data is a core problem in CPU design. Yet, while modern memory has seen exponential growth in terms of scale and information density, transfer rates from memory to CPU have not experienced the same level of improvement, meaning a CPU often has to sit idle while waiting for instructions or data to be funneled onto the chip from memory. This disparity between CPU speed and memory transfer speed has led to one of the largest problems in modern-day computing, known as the **`Von Neumann bottleneck`**, or, colloquially, the "**`memory wall.`**" While a faster CPU might be able to execute instructions at a quicker clock cyle, it will proportionally end up spending more clock cycles sitting idle waiting for memory reads—whose latency depends on other factors beyond just the CPU speed—as it has to fetch both instructions and data at this increased pace.[7] That is, merely increasing CPU clock speed is facing quickly diminishing returns in terms of processor performance.

To minimize this problem, almost every modern CPU takes on a split-cache architecture, or **`almost-von-Neumann architecture`**, with the end goal of minimizing the latency of memory operations and the blocking nature of instruction fetches vs. data fetches. This involves two components: first, and most importantly, the implementation of multi-level hardware caches to store frequently-accessed data closer to the CPU and avoid high latency memory reads, and secondly, the separation of instructions and data at the cache level (but not at the memory address space level) such that instruction and data fetches can occur simultaneously, at least from the cache. This approach preserves the logical benefits of the Von Neumann architecture in storing code and data in the same memory physical address space, while benefitting from the improved parallel instruction and data memory accesses from the Harvard architecture. The multi-level cache

7 .. Zou, X., Xu, S., Chen, X. et al. Breaking the von Neumann bottleneck: architecture-level processing-in-memory technology. Sci. China Inf. Sci. 64, 160404 (2021). https://doi.org/10.1007/s11432-020-3227-1

setup improves the speed of memory reads and writes across-the-board, at the expense of complexity in CPU architecture for data guarantees.

» Cache Hierarchy

CPU caches are most commonly organized into a three-level hierarchy of hardware caches: a Level 1 cache (**`L1`**), a Level 2 cache (**`L2`**), and a Last Level Cache (**`LLC`**), in increasing order of size and distance form the processor cores. Memory lookups cascade from L1 down, meaning that an uncached memory access has to miss in the three cache levels, L1, L2, and LLC, before actually being fetched from main memory. In chips with a wholly inclusive policy, any cache line found in the L1 cache must be present in the L2 cache, and likewise for the L2 cache being present in the LLC cache. However, chip manufacturers can take vastly different cache inclusion policies as they see fit—one approach taken in Intel processors, for example, is for a non-inclusive policy at the LLC with respect to the L2 cache, but inclusivity at the L2 cache with respect to the L1 caches. All of these caches are **`on-die`**—built on the CPU itself—with each core of a CPU corresponding to its own L1 caches and successive caches typically being shared across cores.

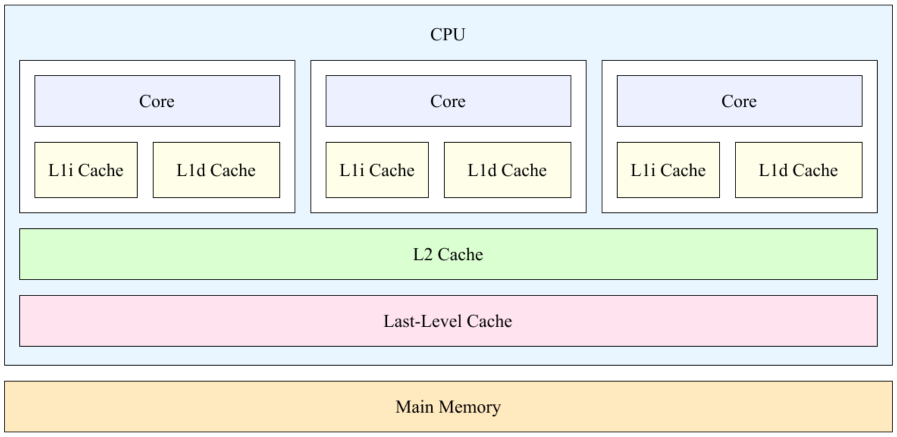


Figure 2: A simplified visualization of a three-level CPU cache hierarchy.

In the split-cache architecture shown in Figure 2, the L1 caches are composed of two hardware caches, the **`L1i`** and **`L1d`** caches—the former for instructions, the latter for data. By separating the instruction and data caches, memory operations for the different types of information can be dispatched in parallel, and, in the best case scenario (an L1 cache hit), resolve without blocking the other. This separation of instructions and data has been commonplace for decades, having first been introduced with the IBM 801 CPU in 1976[8], and is a limited version of the **`Harvard architecture`** in which instructions and data in a computer are stored in separate memory address spaces with separate access patterns entirely[9], enabling hardware supporting parallel reading and writing to each space at the same time.

8 .. Alan Jay Smith. 1982. Cache Memories. ACM Comput. Surv. 14, 3 (Sept. 1982), 473–530. https://doi.org/10.1145/356887.356892

9 .. "Harvard vs von Neumann Architectures." ARM Developer. Accessed April 20, 2024. https://developer.arm.com/documentation/ka002816/latest

» Cache Coherence

In maintaining data integrity across this multi-core, multi-level hierarchy of caches, the CPU must solve the problem of **`cache coherence`**—making sure that memory reads and writes that hit a cache from one core are propagated to the caches in the other cores, and ultimately to the lower-level caches and main memory. There are two conditions for cache coherence:

1. **`Write Propagation`**, i.e. that changes to the data in a cache must be propagated to other copies of that data in the peer caches.
2. **`Transaction Serialization`**, i.e. that reads and writes to a single memory location must be seen by all processors in the same order.[10]

Maintaining cache coherence most commonly involves either broadcasting memory transactions to all cores or performing all memory operations inside a shared memory space—either a directory or another shared cache. The fact that all transactions to shared caches take place on a bus—the hardware for transferring data—make a technique called **`bus snooping`** a way to maintain coherency, with each cache monitoring the bus and updating its internal state if it detects it has the same block as a transaction on the bus.[11]

This is not to mention the difficulty of cache coherence on Non-Uniform Memory Access (NUMA) machines[12]—where distributed nodes (cores) possess their own local memory on top of their local caches—but these are not the norm in consumer-grade computers.

» Speculative Execution

To fully capitalize on the resources of the CPU, execution of instructions is not done one-at-a-time, but rather broken down into stages of a pipeline operating on multiple instructions at once. This is known as **`superscalar`** execution, with these pipelines supporting execution of multiple instructions in "stages" known as superscalar pipelines. Each stage of a superscalar pipeline corresponds to one or more clock cycles, such that instructions march through the pipeline one clock cycle at a time. At any point in time, there may be as many instructions in the pipeline as there are stages and sub-steps of each stage. A common abstraction of pipeline stages includes the following 5 single-step stages:

1. Instruction Fetch (**`IF`**), where instructions are read from the cache.
2. Instruction Decode (**`ID`**), where opcodes and operands are parsed.

10 Culler, David; Gupta, Anoop; Singh, Jaswinder Pal (1999). Parallel Computer Architecture: A Hardware/Software Approach. San Francisco: Morgan Kaufmann Publishers. pp. 369–372. ISBN 1558603433.
11 Ibid., p. 279.
12 https://web.archive.org/web/20131228092942/http://www.cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf

3. Execute (**`EX`**), where (for example) the ALU performs arithmetic operations and multiplication operations may begin.
4. Memory Access (**`MEM`**), where memory operations start. This stage often has multiple steps, each one clock cycle, though not in this example.
5. Writeback (**`WB`**), where the results of an instruction are written to a buffer to be committed to memory (the cache, usually).

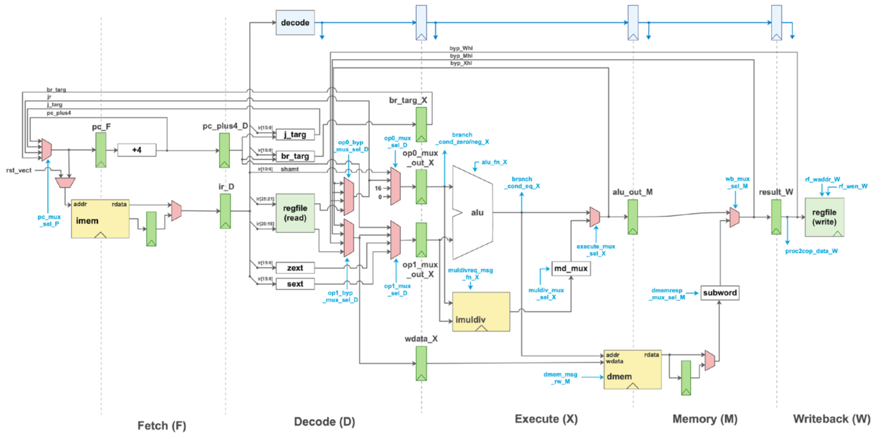


Figure 3: A rudimentary pipeline implementation for a 5-stage bypassing PARCv2 processor.

In the example pipeline in Figure 3, instructions technically take 5 clock cycles to complete, but a single instruction "finishes" every clock cycle, assuming no stalling of the pipeline. An integral issue arises in the fact that the condition of a conditional branch instruction (any **`if`** statement not compiled away, for example) is only evaluated in the Execute stage, three clock cycles in, meaning either the pipeline:

(a) inefficiently stalls in the Fetch stage for multiple clock cycles until the conditional branch is executed, the correct instructions are fetched from main memory, and the instruction cache is prepared, or

(b) the CPU predicts the outcome of the branch and begins executing instructions along its predicted path, rolling back the state of the pipeline if the prediction does not match the actual branch condition.

To remedy this problem, CPUs take option (b) in a complex process in what is called **`speculative execution`**.[13] Instructions are already being loaded pre-emptively in sequence into the instruction cache by a hardware device called the **`instruction prefetcher`**.[14] With speculative execution added to the mix, when the instruction prefetcher detects a conditional branch or jump in the instruction cache, it does not blindly fetch the following instructions sequentially from memory. Rather, it communicates with a hardware device called a **`branch predictor`** (at least, on Intel processors[15]) using an algorithm based on the current and historical state

13 The introduction of speculative execution to CPUs introduces a whole new category of side-channel attacks, most notably Spectre and Meltdown. See: https://spectreattack.com
14 Zhang, Zhiyuan, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. "{BunnyHop}: Exploiting the Instruction Prefetcher." In 32nd USENIX Security Symposium (USENIX Security 23), pp. 7321-7337. 2023.
15 *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation, February. 2022.

of the pipeline and the currently-cached instructions.[16] The output of the branch predictor is a best-guess as to where the instructions that will follow after the branch will come from. The instruction cache is then populated from that point. Once the conditional branch is evaluated, if the branch was predicted correctly, then:

- There has been a several-clock-cycle boost in efficiency, as no stalling was introduced in the pipeline while waiting for the branch condition to be evaluated.
- The instructions speculatively executing in the pipeline are valid.
- The instructions remaining in the instruction cache are all valid.

If the branch prediction was *wrong*:

- A rollback occurs of the speculatively-executing instructions.
- The instruction prefetcher must fetch the actual instructions it needs to execute and invalidate the speculatively cached instructions.
- Several other hardware devices must have their state reset back to their pre-speculation state.

Still, modern branch predictors are good enough at what they do that branch mispredictions are infrequent enough to offset their heavy cost and architectural complexity.

This concept of prefetching data for a cache is the focus of the implementation portion of this paper, where we shall apply this CPU concept to the globally distributed caches of the web. For now, however, with CPU caching covered, let us examine a much higher-level version of caching—that of web resources.

1.3. Caching on the Web

On the web, caching serves not to prevent against main-memory accesses, but rather to skip high latency network requests across servers for web resources, including but not limited to HTML, JavaScript, CSS, video, image, and document files. Caching here takes place conceptually in software, rather than hardware like the CPU, and is distributed globally across different servers in countless regions. Cache entries are keyed by a combination of URL, query parameters, cookies, and headers representing a user's request, and contain as value a single static file.

The essence of web caching exists in the Hypertext Transfer Protocol (HTTP), where a standard set of headers attached to messages from both client **`request`** and server **`response`** define instructions for how resources may be added to both shared caches and private caches. Of greatest importance of these headers is the **`Cache-Control`** header, containing a list of directives for managing cache behavior.[17] Cache implementations may vary, but this shared standard provides a common set of baseline rules

16 Mittal, Sparsh. "A survey of techniques for dynamic branch prediction." Concurrency and Computation: Practice and Experience 31, no. 1 (2019): e4666.
17 Mozilla Contributors. "Cache-Control". MDN Web Docs. Accessed April 20, 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

supported by all web caches. A table of the most common Response Cache-Control header directives can be found in Table 2.

When a resource is requested from a server, instead of going directly to the **`origin server`**, it is (assuming the hosting setup for a standard website) instead routed through a series of caching servers provided by a Content Delivery Network, or a CDN. If there is a cache hit at any of these intermediary nodes, the file is returned from the cache there instead of making its way back as a request to the origin server. This saves a huge amount of time in establishing TCP connections as the request travels throughout the web, as well as mitigate the need for origin servers to process huge numbers of requests. The Cache-Control header, specified by the origin server, is parsed by each of these caching layers, informing them about how long they may keep a file in their cache for.

» Cache Hierarchy

Like the CPU, caching on the Web takes place in a hierarchy of caches of increasing size. There is a rough correspondence in terms of cache levels here, as seen in Figure 4.

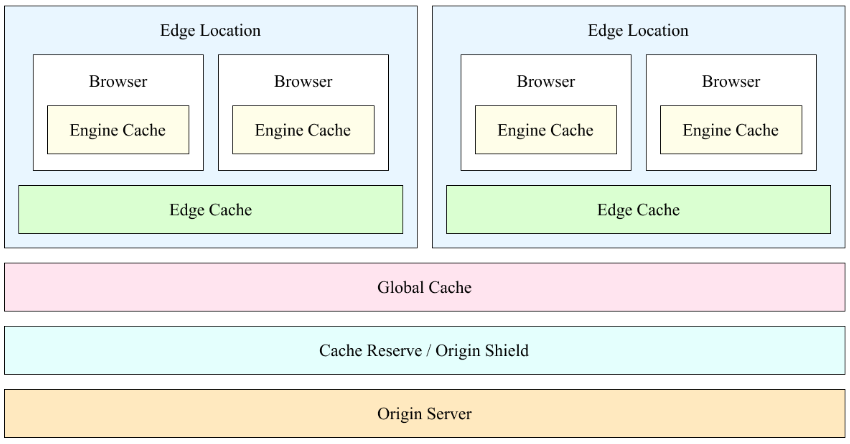


Figure 4: A simplified visualization of the caches involved in Web delivery.

Instead of processor cores with L1 instruction and data caches at the most local level, the "end nodes" are instead web browsers, with their own HTTP caches directly within their users' computers. Like L1 caches, these web engine caches are private to the computer and directly colocated with the place the data will be used.

Between the end-user web browser and the origin server, the caching layers are almost always provided by a specialized service known as a Content Delivery Network, or a CDN. These intermediary caches are not just bigger, they are also shared across different customers and applications as allocated by the CDN provider. Examples of CDN providers include Cloudflare, Akamai, Fastly, Bunny CDN, AWS CloudFront, and Google Cloud CDN, to name a few. Closest in locality to the end user are **`edge`**

`servers`, which have small capacity but are far more widespread.[18] By locating these specialized edge cache servers close to the end user, network speeds can be vastly reduced, as data has to travel a much shorter distance. However, the tradeoff is that edge servers often come with much less capacity and much less computing power, meaning they can't host entire servers or do much beyond simple calculations and serving static files.

Beyond the edge network is the **`global cache`**, CDN servers which are fewer in number but much larger in size and compute power. These are generally a handful per continent, in key locations with large populations. When there is a cache miss in the edge network, this is the next destination that is checked. A key insight for both the global cache and the edge cache is that they are heavily economical regarding which files they choose to cache. Even if a file is *able* to be cached for an extensive period of time, if it does not receive enough requests, it is likely that it will be automatically culled from the cache by the CDN provider before it fully expires and have to be revalidated, regardless of its maximum age (denoted by the **`s-maxage`** Cache-Control directive).

After the global cache is the **`origin shield`**, which can be considered a form of last level cache. The sole purpose of this cache is to catch any cache misses from the global cache, and is often a paid CDN add-on instead of a free offering like the other cache levels. The benefit of the origin shield is that resources found here are preserved indefinitely, safe from eviction until they reach their maximum age. In return, the host of the origin server is often charged for the storage costs of their files in the origin shield. The term "shield" is fitting, as this caching layer acts as a shield for requests to the origin server, making sure that only truly uncached requests hit the origin server, vastly reducing egress bandwidth and ultimately fees.

A major benefit of the CDN is the dedicated networking infrastructure that these platforms provide in distributing web resources globally. Beyond the simple servers they own, it is in the best interest of the CDN to also optimize routing, compression, security, and other network-related problems across their global network, which is the route that these companies generally take.

» Cache Coherence

The Cache-Control HTTP header goes a long way in ensuring data only persists in a cache as long as the origin server wants. However, sometimes the cache must be purged manually, for which CDNs typically offer some sort of cache invalidation mechanism. By submitting a cache key to such an API, the CDN will invalidate or delete the specified cache key across all of its points of presence—edge servers, global caches, and origin shield.

The software implementation, global distribution, and larger margin for performance issues of CDN caches means that satisfying cache coherence is a much different task than it is on the CPU—in general, when a cache entry is invalidated, it can broadcast its status across the CDN's

18 A common metric by which CDNs are judged is points of presence, or POPs, which refers to the locations of all CDN servers (including edge servers) distributed across the globe.

cache with software-based network requests, not hardware, meaning the problems solved by, for example, bus snooping in the CPU are much easier to handle.

» Predictive Prefetching

With a basic understanding of the role of CDNs and caching on the Web, how might we map concepts we understand and find in CPU caches across disciplines into the world of the Web? Such is the focus of the rest of this paper, focusing on the benefits of the prefetching mechanisms found in the CPU and determining if CDN caches might benefit from such a predictive prefetcher.

At its most basic level, prefetching in the CPU simply fetches and pre-populates hardware caches with data that might be needed—*before* it is requested. In a CDN, then, the obvious corollary is to pre-fetch web resources into the cache that might be needed—*before* they are requested.

For example, if a webpage contains an image and two internal links, it is highly likely that the successive resources that the user will request after requesting the initial webpage from the CDN are that image and one of those two internal links. If we can pre-populate the edge server's cache with those resources from a lower-level cache before the user fully downloads the page and actually requests the successive resources, we will effectively save the user the bandwidth of a network round trip from the edge server to the global cache—or even all the way back to the origin server—for each of those successive resources. See Figure 6 and Figure 7 for an example visualization of this procedure.

A review of the service offerings of the major CDN providers finds only a single CDN which offers such a cache-level prefetch capability, Akamai[19] (who has dubbed the concept "predictive prefetching").

2. Methods

Having established a technical foundation for how caches are implemented to maximize performance on both the CPU and the Web, this paper shall now outline methods for implementing a CDN layer augmented with the lookahead properties that CPU instruction prefetchers exhibit in predicting memory accesses for the CPU caches.

2.1. Basic Cache

The predictive prefetchers outlined in this paper are built in Cloudflare workers, enabling direct access to read and write from Cloudflare's edge cache through its Cache API.[20] Cloudflare workers are small JavaScript functions that run on edge servers, before requests hit the cache, in the

19 Akamai. "Predictive Prefetching." Akamai Tech Docs. Accessed April 20, 2024. https://techdocs.akamai.com/property-mgr/docs/predictive-prefetching
20 Cloudflare. "Workers Runtime APIs: Cache." Cloudflare Docs. Accessed April 20, 2024. https://developers.cloudflare.com/workers/runtime-apis/cache/

`workerd` execution environment (a JavaScript/WASM environment similar to Node.js, but lacking many of the built-in libraries Node.js offers like **`fs`**, **`crypto`**, **`path`**, etc.[21]). Code for a basic Worker that forwards requests to an origin server and caches the responses in Cloudflare's CDN can be implemented as in Figure 9 on page 21.

Importantly, Cloudflare makes adding items to their global cache network as simple as calling an asynchronous function in a worker. This `fetchAndCache` function forms the basis for the two predictive prefetching workers that follow. The basic cache implementation is used a baseline to compare against in the prefetching cache implementations.

2.2. Prefetcher: HTML Parser

The first predictive prefetching implementation takes a deterministic approach, using an HTML parser to find literal resource links within a returned HTML document. As an HTML response is streamed to the client through the caching worker in a Node.js stream, the function parses anchor tags and image srcs using Cloudflare's built in HTMLRewriter pattern[22]. If a link is relative (and thus is located behind our cache), the URL is prefetched into Cloudflare's global cache network. An example implementation can be found in Figure 8 on page 20.

A limitation of Cloudflare workers is that they have a maximum quota of 50 subrequests per invocation. This means that simply prefetching all the links and images in a page is not possible, and would already be hugely inefficient, as only one of those links will end up being taken and only some of those images will end up being shown. To protect against this, we limit the number of prefetchable resources in the parser. Currently, the prefetcher simply prefetches the first **`n < 6`** resources in the page, considering that a cache hit takes 1 subrequest and a cache miss takes 3 subrequests. This puts the prefetcher well under the subrequest limit, which can theoretically prefetch up to 15 resources (assuming a cache miss on all 15 of the prefetched resources, as well as a cache miss on the initially requested resource, making the number of subrequests 3 * (15 + 1) = 48).

2.3. Prefetcher: Analytics Predictor

The second prefetching implementation is traffic-based, similar to Akamai's predictive prefetching. Analytics data is captured in a simple SQLite database (bound to the Worker through Cloudflare's D1 service) and informed based on the `Referer` header of a request (if it is present) to build a graph of user traversals across the site. This analytics table is simple: a composite primary key made up of a **`src`** URL and a **`dest`** URL, and a positive integer **`freq`**uency count. Whenever a page is requested, we perform two simultaneous operations on our analytics database:

21 Varda, Kenton. "Introducing workerd: the Open Source Workers runtime." The Cloudflare Blog. Accessed April 20, 2024. https://blog.cloudflare.com/workerd-open-source-workers-runtime
22 Galloni, A. and Stepanyan, I. "A History of HTML Parsing at Cloudflare: Part 1." The Cloudflare Blog. Accessed April 20, 2024. https://blog.cloudflare.com/html-parsing-1

1. If the request contains a Referer HTTP header, we increment by 1 the frequency value in our database with composite key `src`=Referer, `dest`=Requested URL.
2. We query for the top `n` (by `freq`) `dest` URLs for the current `src`.

The results of the second query are then used as the list of URLs to prefetch into the cache, resolving after the requested page is sent back to the client. With this approach, cache prefetches are reliably driven by the resources actually requested by real users, not simply links found in HTML content. Additionally, sorting by navigation frequency imposes a natural heuristic for resources to prefetch, where the number of resources prefetched can be modified simply by increasing or decreasing this `n`. The implementation of this analytics-driven prefetcher can be found in Figure 10 on page 22.

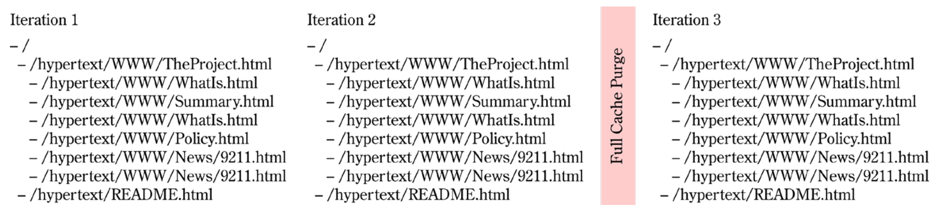
2.4. Testing & Metrics

Creating a representative, real-world cache testing environment for the predictive prefetchers is an immensely difficult task. It requires typical user traffic and cache behavior at the scale that CDNs operate on, this information being: (a) knowledge about when resources will be chosen for eviction from Cloudflare's CDN, which is impossible to know given the proprietary nature of Cloudflare's CDN, (b) a representative site with enough linked resources for prefetching to make sense, and (c) actual, notable user traffic.

Lacking the aforementioned components for real-world testing of predictive prefetching, testing was instead done locally using Cloudflare's `miniflare` development environment, which among its offerings provides a basic emulation of a cache layer, a SQLite database binding for analytics tracking, and network proxy which forwards local origin requests to a configured host. The host chosen to be cached in this paper is the first webpage ever created— http://info.cern.ch, picked largely for its simple HTML structure and high proportion of links. The testing suite traverses the site in the following order:

1. The landing page (/)
2. The first web page ever, by Tim Berners-Lee outlining the World Wide Web (/WWW/TheProject.html)
3. 5 of the sub-pages contained on that page, with one of the sub-pages being visited twice
4. And finally, back to the README file (/hypertext/README.html)

To see how different prefetching strategies affect the cache, this traversal is performed three times, and, after the first two traversals, the cache is fully purged (See Figure 5).



Iteration 1

```
– /
  – /hypertext/WWW/TheProject.html
    – /hypertext/WWW/WhatIs.html
    – /hypertext/WWW/Summary.html
    – /hypertext/WWW/WhatIs.html
    – /hypertext/WWW/Policy.html
    – /hypertext/WWW/News/9211.html
    – /hypertext/WWW/News/9211.html
  – /hypertext/README.html
```

Iteration 2

```
– /
  – /hypertext/WWW/TheProject.html
    – /hypertext/WWW/WhatIs.html
    – /hypertext/WWW/Summary.html
    – /hypertext/WWW/WhatIs.html
    – /hypertext/WWW/Policy.html
    – /hypertext/WWW/News/9211.html
    – /hypertext/WWW/News/9211.html
  – /hypertext/README.html
```

Iteration 3

```
– /
  – /hypertext/WWW/TheProject.html
    – /hypertext/WWW/WhatIs.html
    – /hypertext/WWW/Summary.html
    – /hypertext/WWW/WhatIs.html
    – /hypertext/WWW/Policy.html
    – /hypertext/WWW/News/9211.html
    – /hypertext/WWW/News/9211.html
  – /hypertext/README.html
```

Figure 5: The testing suite, consisting of three traversals across http://info.cern.ch.

The three summary statistics measured in testing are the **`perceived request latencies`**, the **`cache hit ratio`** across the test suite, and the **`number of subrequests`** in each worker (e.g., checking cache status for prefetch candidates and prefetches from origin), compared across all three cache implementations (basic, HTML parsing, and analytics-based). The goal is to analyze the benefits of a cache prefetcher for a CDN as opposed to the standard pull-style CDN cache, trading speculative backend egress fees for improved performance when a user navigates through multiple uncached pages.

3. Results

The full tabulated results of the testing suites with each cache prefetcher can be found in Table 3, Table 4, and Table 5 starting on page 23. The summary statistics of the tests are outlined below in Table 1.

Caching Mechanism	Average Request Latency	Cache Hit Ratio	Total Subrequests
Basic Cache	138.9ms	37.5%	**54**
HTML Parsing Prefetcher	**62.4ms**	**75%**	194
Analytics Prefetcher	91.1ms	62.5%	66

Table 1: Summary statistics across the experiments with different types of caching strategies. The traversal took place over 8 site pages and was repeated three times for a total of 24 navigations, with the last traversal having had the cache purged beforehand.

Both the HTML parsing prefetcher and analytic prefetcher easily surpass the cache hit ratio of the basic, unmodified caching Worker in the testing environment over the 24 requests in the testing suite.

As our testing suite begins with an empty cache, the cache hit ratios experienced are indicative of the strengths of prefetching pages when many of the pages in a site would result in cache misses. It should be noted that the testing environment simulates a tiny sliver of the number of operations and lifetime of objects in the cache as compared to a real-world cache. As the cache hit ratio of the Basic cache would increase as more resources are added and removed from the cache throughout its lifetime of the server, the difference in performance would drop off considerably, with standard cache hit ratios reaching 80-90% and a "good" cache hit ratio being >95%. For this reason, predictive prefetching works best as an early-stage cache supplement to boost initial cache hit ratios when a cache is unpopulated.

However, this comes at a cost in normal operation, when a cache has more pages. All requests now have the post-response overhead of several **`subrequest`**s to check and populate cache entries for linked pages. Particularly in the case of the HTML parsing prefetcher with its indiscriminate prefetching of links on a page, a huge number of subrequests are added to each request to check the cache status (and, subsequently, populate the cache on cache miss) of even the first 6 links in the page. As such, the far smaller overhead of using actual user traffic data to pick a set of best next destinations to prefetch, as in the analytics prefetcher, will likely prove far more cost effective in production.

The analytics prefetcher, too, has its drawbacks—pages that are more heavily navigated to are also more likely to be found in the CDN's cache, leading to minor duplication of logic. A good analytics-based prefetcher would also take into account how often the related pages are cached, and discount the priority of prefetching often-cached pages.

4. Discussion

Where is predictive prefetching in a CDN most useful? From its benefits for sites with low cache hit ratios, it appears to be for sites with low- to medium-size traffic, where CDNs will commonly evict the pages of the site from their caches due to lack of traffic. Predictive prefetching offers very little benefit compared to its cost for sites with a large amount of traffic, as pages for these sites are much more likely to be present in edge caches. Additionally, the prediction mechanism must necessarily run in front of the cache—on both cached and uncached resources (to not skip prefetching of resources linked by cached pages)—meaning every single request is costing that much more in edge compute time. This can blow up exponentially in size, particularly if using such a heavy-handed stroke as the HTML parser.

Additionally, CDN prefetching is only fitting where pages are static or, if generated at request time, generated with long time-to-lives (**`TTL`**s) and without needing user input on the current page, such that cache contents do not expect information in the request from a user during their navigation of the site. Fortunately, this is already an important consideration when using a cache itself, as it is unlikely for sites to cache user-based or time-sensitive resources at the CDN level, where cached resources are persisted and shared across users. In such situations, if prefetching is still desired, client-side prefetching (with the HTML `rel="prefetch"` attribute, or with JavaScript) is the better approach, as this form of prefetching can better opt into user credentials and time-based information.

4.1. Improvements

One remedy for requiring all requests to trigger the predictive prefetcher is to instead take an approach similar to the bus-snooping providing cache coherence in some CPU caches. Instead of putting the prefetcher directly inside the caching function itself, a prefetcher could instead watch server request logs and perform the pre-fetching with that stream as input, not as a direct addon to the request handlers itself.

Far more improvement can also be found in improving prefetch logic. The indiscriminate catch-all of the HTML parser is incredibly inefficient in terms of subrequests, and does not take into account which page will likely be taken out of a first page. This is the area the analytic prefetcher improves on, but additional weighting logic that de-prioritizies pages likely to be cached can make the analytic prefetcher a truly viable solution despite its cost for production websites.

Lastly, allowing for more information from the initial request to be incorporated into the subrequests hydrating the cache will be necessary if deploying a prefetching solution to any cache. The current implementation only caches by URL, whereas modern caches can incorporate all sorts of information, including cookies and headers. These must be forwarded through the subrequests for the prefetcher to cache the correct following resources.

4.2. Conclusion

In this paper, we have surveyed the process of caching on both the CPU and the Web. Using insights from prefetching tactics within the CPU, we proposed and analyzed two methods of prefetching web resources in a CDN, comparing the two methods to the performance of a basic cache. The results show that implementing prefetching at the CDN level can provide modest improvements in cache hit ratios for low- to medium-traffic sites, where cache hit ratios are already low, at the cost of higher egress from speculative cache fetching. Further steps involve improvements to the prefetching algorithms used in the CDN, as well as more rigorous real-world testing of the performance gains brought about by a prefetcher.

Many thanks to my advisor Scott Petersen and the CPSC 490 instructor Sohee Li for their instructoral support in this project. Prof. Abhishek Bhattarchajee's initial instruction in CPSC 420 was also essential in providing a conceptual foundation for CPU architecture and the hardware implementations of CPU caches. Lastly, I would like to thank Maria Wilson, Mia Toledo-Navarro, Collin Robinson, and Kennedy Anderson for their support in the process of writing this thesis.

5. References

Akamai. “Predictive Prefetching.” Akamai Tech Docs. Accessed April 20, 2024. https://techdocs.akamai.com/property-mgr/docs/predictive-prefetching

Alan Jay Smith. 1982. Cache Memories. ACM Comput. Surv. 14, 3 (Sept. 1982), 473–530. https://doi.org/10.1145/356887.356892

Cloudflare. “Workers Runtime APIs: Cache.” Cloudflare Docs. Accessed April 20, 2024. https://developers.cloudflare.com/workers/runtime-apis/cache/

Culler, David; Gupta, Anoop; Singh, Jaswinder Pal (1999). *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco: Morgan Kaufmann Publishers. pp. 279, 369–372. ISBN 1558603433.

D. A. Jimenez and C. Lin, “Dynamic branch prediction with perceptrons,” Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, Monterrey, Mexico, 2001, pp. 197-206, doi: 10.1109/HPCA.2001.903263.

G. C. Stierhoff and A. G. Davis, “A history of the IBM Systems Journal,” in IEEE Annals of the History of Computing, vol. 20, no. 1, pp. 29-35, Jan.-March 1998, doi: 10.1109/85.646206.

Galloni, A. and Stepanyan, I. “A History of HTML Parsing at Cloudflare: Part 1.” The Cloudflare Blog. Accessed April 20, 2024. https://blog.cloudflare.com/html-parsing-1

Intel 64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation, February. 2022.

Merriam-Webster.com Dictionary, s.v. “cache,” accessed April 20, 2024, https://www.merriam-webster.com/dictionary/cache.

Mittal, Sparsh. “A survey of techniques for dynamic branch prediction.” Concurrency and Computation: Practice and Experience 31, no. 1 (2019): e4666.

Mozilla Contributors. “Cache-Control”. MDN Web Docs. Accessed April 20, 2024. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

Solihin, Yan (2016). *Fundamentals of Parallel Multicore Architecture*. Chapman and Hall/CRC. pp. 146–150. ISBN 9781482211184.

Varda, Kenton. “Introducing workerd: the Open Source Workers runtime.” The Cloudflare Blog. Accessed April 20, 2024. https://blog.cloudflare.com/workerd-open-source-workers-runtime

Von Neumann, J. “First draft of a report on the EDVAC,” in IEEE Annals of the History of Computing, vol. 15, no. 4, pp. 27-75, 1993, doi: 10.1109/85.238389.

Zhang, Zhiyuan, Mingtian Tao, Sioli O’Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. “BunnyHop: Exploiting the Instruction Prefetcher.” In 32nd USENIX Security Symposium (USENIX Security 23), pp. 7321-7337. 2023.

Zou, X., Xu, S., Chen, X. et al. “Breaking the von Neumann bottleneck: architecture-level processing-in-memory technology.” Sci. China Inf. Sci. 64, 160404 (2021). https://doi.org/10.1007/s11432-020-3227-1

“Harvard vs von Neumann Architectures.” ARM Developer. Accessed April 20, 2024. https://developer.arm.com/documentation/ka002816/latest

6. Tables & Figures

Directive	Description
`max-age`	The `max-age=N` response directive indicates that the response remains fresh until N seconds after the response is generated.
`s-maxage`	The `s-maxage` response directive indicates how long the response remains fresh in a shared cache. The s-maxage directive is ignored by private caches, and overrides the value specified by the max-age directive or the Expires header for shared caches, if they are present.
`no-cache`	The `no-cache` response directive indicates that the response can be stored in caches, but the response must be validated with the origin server before each reuse, even when the cache is disconnected from the origin server.
`must-revali-date`	The `must-revalidate` response directive indicates that the response can be stored in caches and can be reused while fresh. If the response becomes stale, it must be validated with the origin server before reuse.
`no-store`	The `no-store` response directive indicates that any caches of any kind (private or shared) should not store this response.
`private`	The `private` response directive indicates that the response can be stored only in a private cache (e.g. local caches in browsers).
`public`	The `public` response directive indicates that the response can be stored in a shared cache. Responses for requests with Authorization header fields must not be stored in a shared cache; however, the public directive will cause such responses to be stored in a shared cache.
`stale-while-revalidate`	The `stale-while-revalidate` response directive indicates that the cache could reuse a stale response while it revalidates it to a cache.
`stale-if-error`	The `stale-if-error` response directive indicates that the cache can reuse a stale response when an upstream server generates an error, or when the error is generated locally. Here, an error is considered any response with a status code of 500, 502, 503, or 504.

Table 2: Common Response Cache-Control header field directives, courtesy of Mozilla Web Docs.
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control

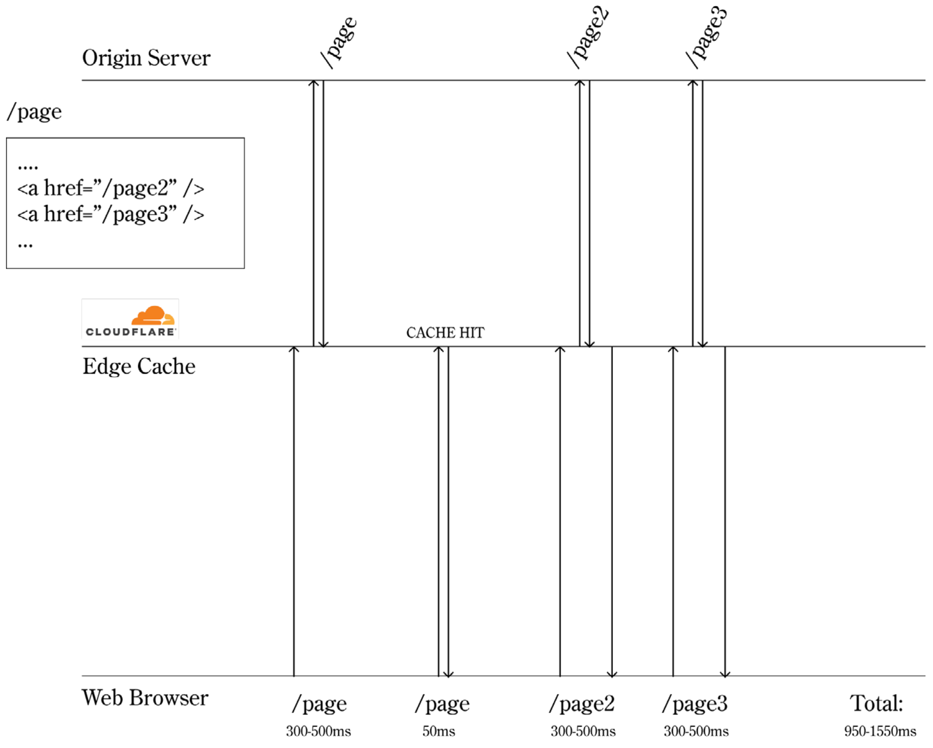


Figure 6: Fetching an example page "/page" without predictive prefetching, assuming no prior cached resources.

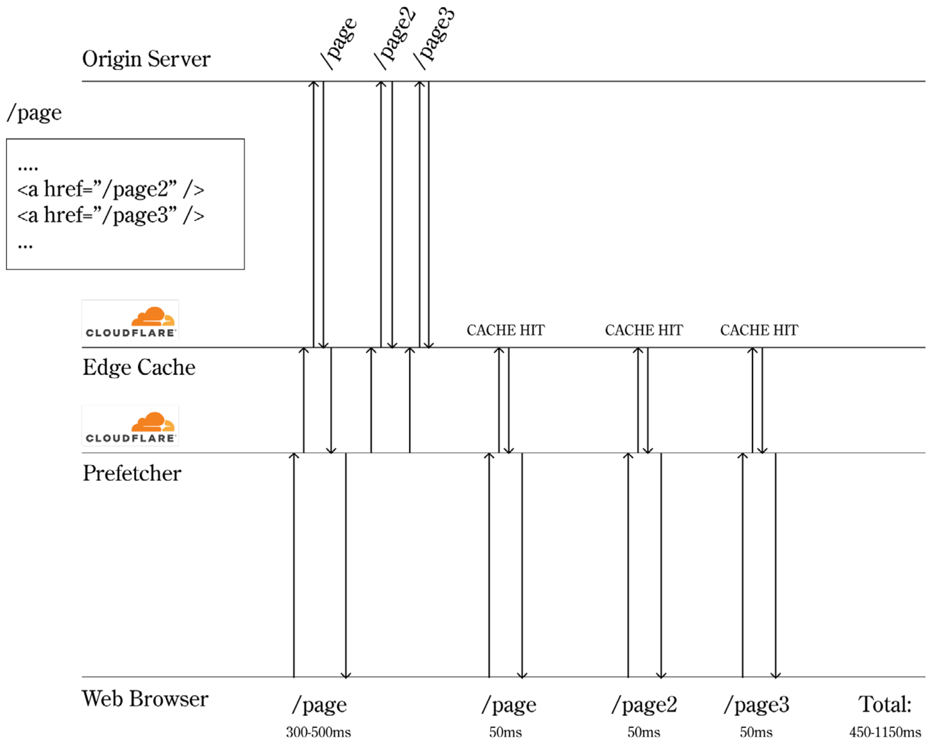
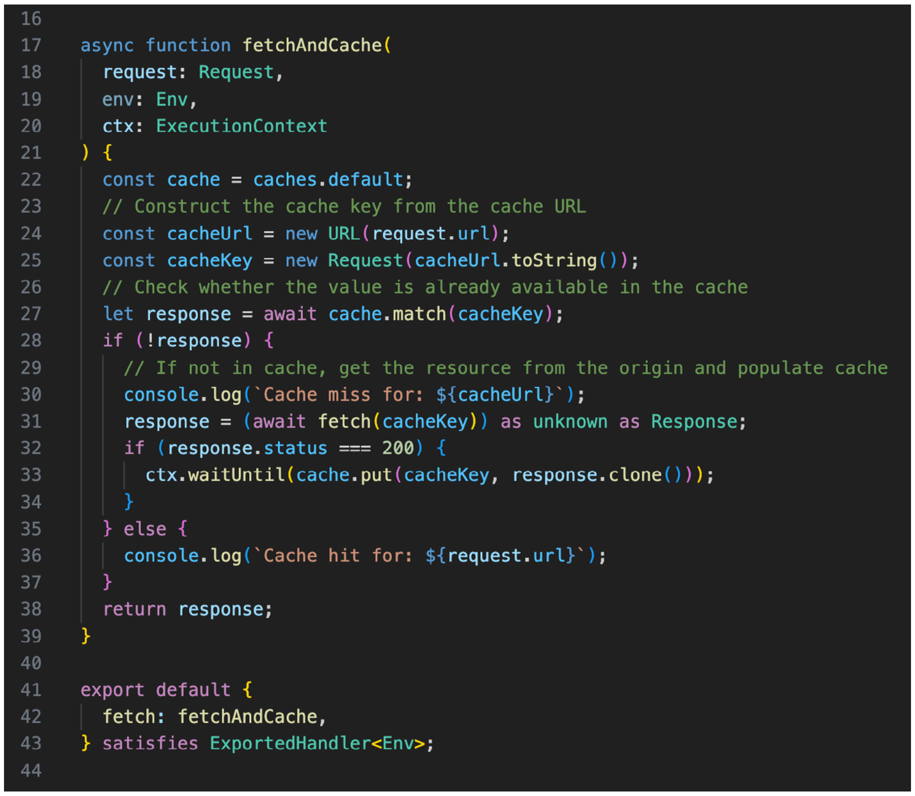
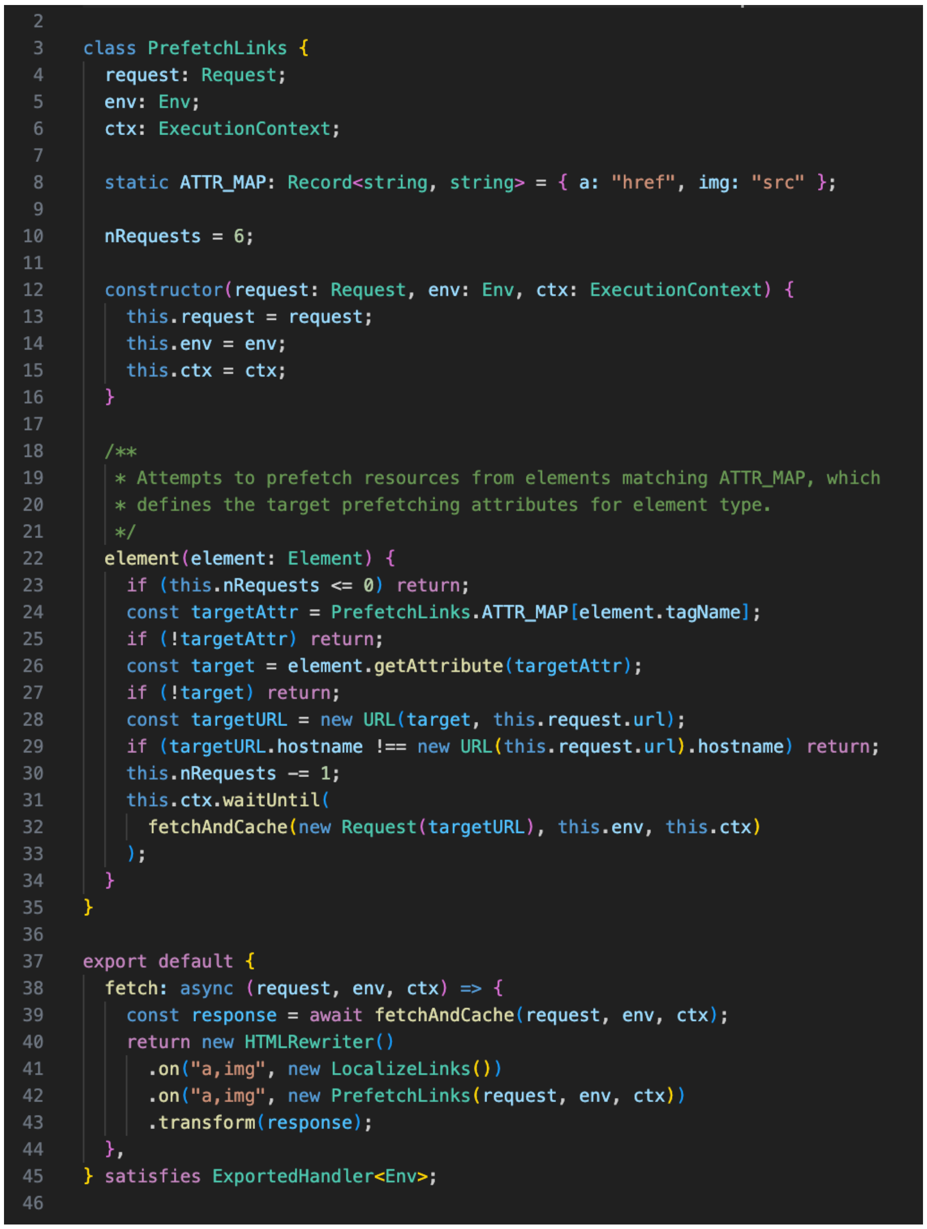


Figure 7: Fetching an example page "/page" with predictive prefetching, assuming no prior cached resources.



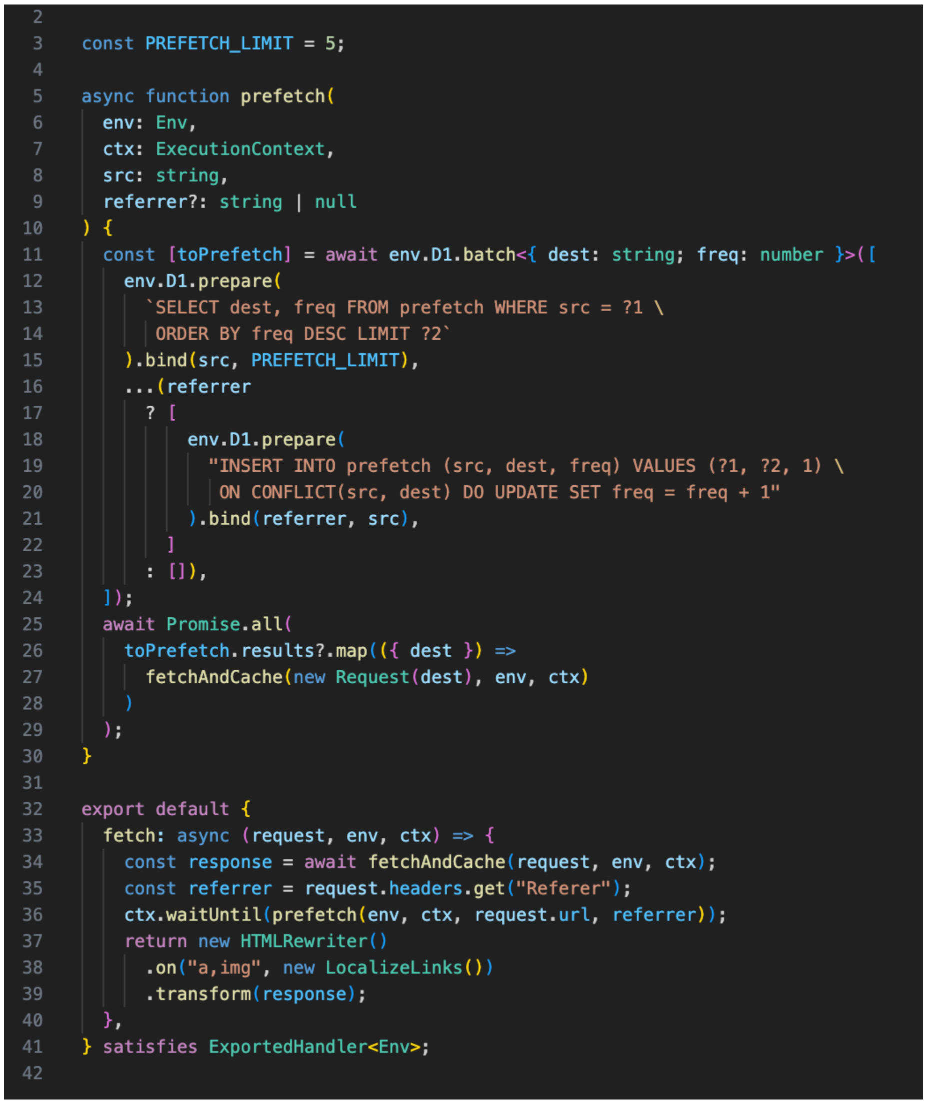
```typescript

async function fetchAndCache(
  request: Request,
  env: Env,
  ctx: ExecutionContext
) {
  const cache = caches.default;
  // Construct the cache key from the cache URL
  const cacheUrl = new URL(request.url);
  const cacheKey = new Request(cacheUrl.toString());
  // Check whether the value is already available in the cache
  let response = await cache.match(cacheKey);
  if (!response) {
    // If not in cache, get the resource from the origin and populate cache
    console.log(`Cache miss for: ${cacheUrl}`);
    response = (await fetch(cacheKey)) as unknown as Response;
    if (response.status === 200) {
      ctx.waitUntil(cache.put(cacheKey, response.clone()));
    }
  } else {
    console.log(`Cache hit for: ${request.url}`);
  }
  return response;
}

export default {
  fetch: fetchAndCache,
} satisfies ExportedHandler<Env>;

```

Figure 8: A basic implementation for a caching function in a Cloudflare worker. The fetchAndcache function serves as a shared foundation for the rest of the prefetchers.



```typescript

class PrefetchLinks {
  request: Request;
  env: Env;
  ctx: ExecutionContext;

  static ATTR_MAP: Record<string, string> = { a: "href", img: "src" };

  nRequests = 6;

  constructor(request: Request, env: Env, ctx: ExecutionContext) {
    this.request = request;
    this.env = env;
    this.ctx = ctx;
  }

  /**
   * Attempts to prefetch resources from elements matching ATTR_MAP, which
   * defines the target prefetching attributes for element type.
   */
  element(element: Element) {
    if (this.nRequests <= 0) return;
    const targetAttr = PrefetchLinks.ATTR_MAP[element.tagName];
    if (!targetAttr) return;
    const target = element.getAttribute(targetAttr);
    if (!target) return;
    const targetURL = new URL(target, this.request.url);
    if (targetURL.hostname !== new URL(this.request.url).hostname) return;
    this.nRequests -= 1;
    this.ctx.waitUntil(
      fetchAndCache(new Request(targetURL), this.env, this.ctx)
    );
  }
}

export default {
  fetch: async (request, env, ctx) => {
    const response = await fetchAndCache(request, env, ctx);
    return new HTMLRewriter()
      .on("a,img", new LocalizeLinks())
      .on("a,img", new PrefetchLinks(request, env, ctx))
      .transform(response);
  },
} satisfies ExportedHandler<Env>;

```

Figure 9: An HTML link-parsing prefetcher built in a CloudFlare worker.



```ts

const PREFETCH_LIMIT = 5;

async function prefetch(
  env: Env,
  ctx: ExecutionContext,
  src: string,
  referrer?: string | null
) {
  const [toPrefetch] = await env.D1.batch<{ dest: string; freq: number }>([
    env.D1.prepare(
      `SELECT dest, freq FROM prefetch WHERE src = ?1 \
      ORDER BY freq DESC LIMIT ?2`
    ).bind(src, PREFETCH_LIMIT),
    ...(referrer
      ? [
          env.D1.prepare(
            "INSERT INTO prefetch (src, dest, freq) VALUES (?1, ?2, 1) \
            ON CONFLICT(src, dest) DO UPDATE SET freq = freq + 1"
          ).bind(referrer, src),
        ]
      : []),
  ]);
  await Promise.all(
    toPrefetch.results?.map(({ dest }) =>
      fetchAndCache(new Request(dest), env, ctx)
    )
  );
}

export default {
  fetch: async (request, env, ctx) => {
    const response = await fetchAndCache(request, env, ctx);
    const referrer = request.headers.get("Referer");
    ctx.waitUntil(prefetch(env, ctx, request.url, referrer));
    return new HTMLRewriter()
      .on("a,img", new LocalizeLinks())
      .transform(response);
  },
} satisfies ExportedHandler<Env>;

```

Figure 10: An analytics-based prefetcher built in a CloudFlare worker.

t=	Referer	URL	Time (ms)	Cached?	Subreqs
1		/	265.9		3
2	/	/hypertext/WWW/TheProject.html	228.0		3
3	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	214.1		3
4	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	226.1		3
5	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	215.9		3
6	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	220.0		3
7	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	5.9	✓	1
8	/	/hypertext/README.html	207.5		3
9		/	7.3	✓	1
10	/	/hypertext/WWW/TheProject.html	4.4	✓	1
11	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	2.8	✓	1
12	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	1.7	✓	1
13	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	1.5	✓	1
14	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	1.8	✓	1
15	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	1.9	✓	1
16	/	/hypertext/README.html	2.2	✓	1
Full Cache Purge					
17		/	216.2		3
18	/	/hypertext/WWW/TheProject.html	213.4		3
19	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	209.7		3
20	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	230.8		3
21	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	212.8		3
22	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	210.6		3
23	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	211.4		3
24	/	/hypertext/README.html	223.7		3
			≈ 138.9	37.5%	54

Table 3: Performance of the basic caching layer. Pages must first be visited to be put into the cache, and the full cache purge forces all of the successive pages to result in cache misses felt by the user. This is the standard behavior of CDN caches. Note that a cache hit only requires a single subrequest (getting the item from the cache), whereas a cache miss requires three subrequests (the cache miss, fetching from origin, and cache population).

t=	Referer	URL	Time (ms)	Cached?	Subreqs
1		/	254.5		6
2	/	/hypertext/WWW/TheProject.html	7.3	✓	12
3	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	4.2	✓	11
4	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	9.9	✓	10
5	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	9.6	✓	6
6	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	230.0		15
7	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	5.1	✓	5
8	/	/hypertext/README.html	219.5		13
9		/	9.6	✓	2
10	/	/hypertext/WWW/TheProject.html	6.2	✓	6
11	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	6.0	✓	7
12	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	7.8	✓	4
13	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	8.1	✓	6
14	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	4.5	✓	5
15	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	8.0	✓	5
16	/	/hypertext/README.html	6.9	✓	5
Full Cache Purge					
17		/	236.8		6
18	/	/hypertext/WWW/TheProject.html	8.9	✓	12
19	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	5.2	✓	11
20	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	4.4	✓	10
21	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	3.6	✓	6
22	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	218.6		15
23	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	5.7	✓	5
24	/	/hypertext/README.html	217.6		13
			≈ 62.4	75%	196

Table 4: Performance of the HTML parsing-based prefetching caching layer, prefetching the first 6 links present in the page's HTML document.

t=	Referer	URL	Time (ms)	Cached?	Subreqs
1		/	250.4		3
2	/	/hypertext/WWW/TheProject.html	229.0		3
3	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	221.2		3
4	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	235.4		3
5	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	226.0		3
6	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	225.3		3
7	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	6.7	✓	1
8	/	/hypertext/README.html	225.5		3
9		/	6.5	✓	3
10	/	/hypertext/WWW/TheProject.html	10.1	✓	5
11	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	7.5	✓	1
12	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	4.0	✓	1
13	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	6.9	✓	1
14	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	3.5	✓	1
15	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	11.0	✓	1
16	/	/hypertext/README.html	5.3	✓	1
Full Cache Purge					
17		/	230.4		9
18	/	/hypertext/WWW/TheProject.html	5.4	✓	13
19	/hypertext/WWW/TheProject.html	/hypertext/WWW/WhatIs.html	3.6	✓	1
20	/hypertext/WWW/TheProject.html	/hypertext/WWW/Summary.html	7.4	✓	1
21	/hypertext/WWW/TheProject.html	/hypertext/WWW/Policy.html	231.1		3
22	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	6.7	✓	1
23	/hypertext/WWW/TheProject.html	/hypertext/WWW/News/9211.html	4.6	✓	1
24	/	/hypertext/README.html	4.0	✓	1
			≈ 91.1	62.5%	66

Table 5: Performance of the analytics-based prefetching caching layer, prefetching the 3 most popular outgoing links from a page. Traffic between internal pages of the site is tracked in a SQLite database.

FINAL DRAFT