Cache-Control: A Survey in Fast Delivery Across CPU and Web

CPSC490: Senior Project
Student: Evan Kirkiles
Advisor: Scott Petersen

This file is a working draft for me to plot out my thoughts and initial writing. I will eventually put all of this into a nicely-formatted InDesign PDF when I wrap up the writeup. This will also include better data figures, and citations. Just be careful if attempting to read it as a text that makes sense outside of fragmented thoughts / outlines.

In terms of what's been done:
- Initial research into web caching / CPU caching
- Design wire framing for web tool
- Code for basic push-style cache built using CloudFlare R2 (this will serve as a rudimentary landscape for my experiment with branch prediction)

Introduction

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, the answer results in a complex architecture of multi-level caches and specialized hardware for predicting memory accesses ahead of time. On the web, caching has led to a huge economy of Content Delivery Networks (CDNs), where web resources—images, videos, CSS, HTML, and JavaScript, to name a few—are copied and hosted across the globe, as close as possible to the end user, to prevent lengthy network requests back to the origin server.

Done correctly, caching is one of the most effective ways of speeding up a slow process—even outside of file and data persistence, caching finds its way into most frameworks and libraries as a method of cutting back on unnecessary extra work. Techniques like dynamic programming capitalize off of cached function calls to eliminate deep, duplicate recursive calls.

In this project, we provide a survey of caching on both the web and the CPU, and examine areas in which cross-pollination can occur between the two—focusing mainly on the concept of branch prediction as it might apply to web resources.

Literature Review

CPU Cache

The modern, multi-core CPU is made up of countless moving parts. To name the most important…

<Brief overview of CPU architecture>

Underlying every functionality within the CPU are a huge system of caches—often three-layered, but sometimes more as in.

In terms of caching, the modern CPU is generally made up of three layers: the L1 instruction and data caches, the L2 caches which catch misses in either of the L1 caches, and an L3 cache shared across cores. The lower-numbered caches are physically closer to their respective cores, with cache coherency managed by a complex set of algorithms to ensure that data is up-to-date across caches across each core.

<Wrap up CPU architecture section>

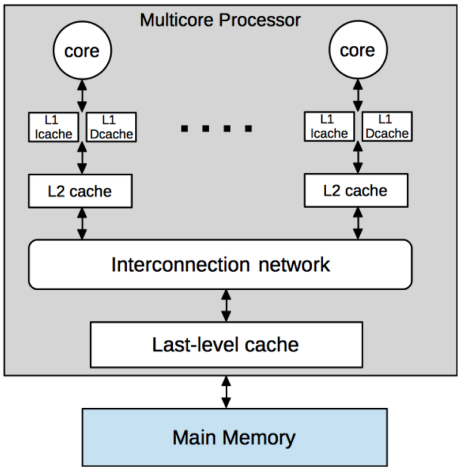


Figure 5: Block diagram of a multicore processor.

Bibliography Here:
https://arxiv.org/ftp/arxiv/papers/1801/1801.05215.pdf

Web Caching

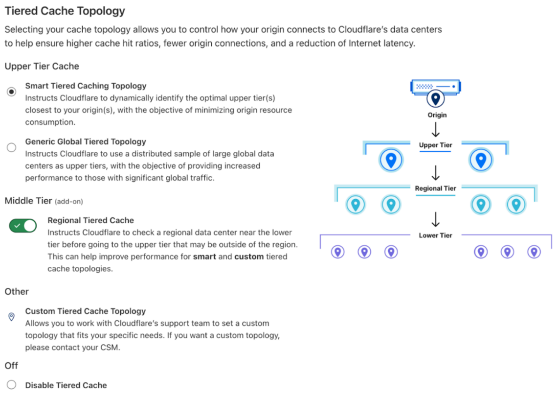
Caching on the web, while similarly built and implemented to the multi-level setup that CPUs implement, serves a slightly different motivation. Whereas the CPU cache aims to prevent calls to main-memory during execution, the CDN cache aims to prevent network requests to the origin server. The work, here, has hypothetically already been done by the origin server—it is egress fees, slow connections, and

---

The CDN

Caching for the web through the CDN takes place globally across a network of servers. As in the CPU, caches local to data are smaller and more numerous. In web-speak, this first tier is known as the “edge”—a network solely functioning to run small bits of code and deliver files as close to a user as possible.

In the CloudFlare example to the right, (their solution labelled “tiered caching”) the origin receives only a fraction of the requests which



users actually make to the domain it is hosted on. Rather, each cache level serves to capture most-recently used data, in decreasing levels of usage. Common cache eviction policies include LRU, MRU, and Least Frequently Used (LFU), though the actual algorithms used for eviction in CDN caches are generally not shared with the customer.

The Browser

There is another level of caching on the web which takes place locally, within users' browsers. Origin servers can provide rules for caching a specific file within HTTP headers like `Cache-Control` and `Expiration`, such that the browser knows to keep a file around and not send a new network request until that cached file has expired. An example of one such setting

This is done through the `Cache-Control` HTTP header.

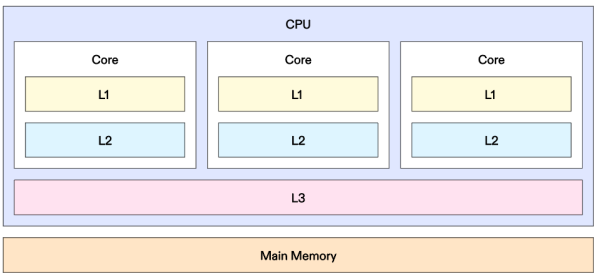
Interestingly, the API put forth for controlling browser caches through the `Cache-Control` header is often duplicated for CDNs—CDNs using the same format but often prepending the specific provider's identifier so that proxies know which cache policy applies to them. For example, CloudFlare primarily uses a `CloudFlare-Cache-Control` header from the origin server, but also supports `CDN-Cache-Control` and even `Cache-Control` in decreasing orders of precedence.

Existing Problems

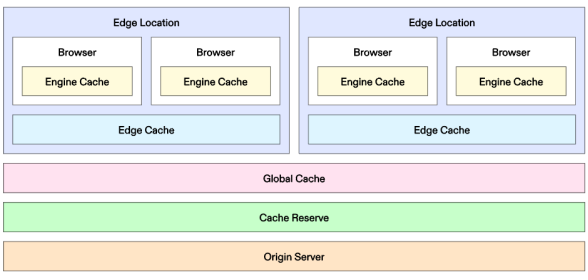
Cross-Pollination

From a birds-eye view, caches across the two domains are quite similar—particularly in the multi-core approach to caching. Each edge location can be thought of as a "core" serving all of its region-specific users

Example



Example



Area for Innovation

(The below needs citations and figures)

Being in such a critical location for computational performance, the CPU has reaped the benefits of intense scrutiny and thus intense innovation in the field of caching. There is arguably no place more highly optimized in terms of caches than within the CPU.

CPU Background

Knowing this, we can take insight from the world of CPU caching and apply it to web caching to begin trying to eke out as much performance as possible from our CDNs. This senior project covers the scope of a technique called “branch prediction.” Essentially, the problem boils down to prediction of resources required before they are accessed.

In the CPU, these are literal instructions: when the CPU loads a conditional branch or jump instruction into its instruction cache, it hypothetically must wait until that instruction is executed before it knows both what instructions will be required and what data will be requested from memory to be able to continue filling up caches. There is no way around this without knowing ahead-of-time the outcome of the branch condition—hence the creation of the “branch predictor” circuit, a piece of hardware which attempts to determine the direction a conditional branch or jump instruction will take before the condition is actually evaluated. When the

This explodes in complexity when considering multiple paths, as in switch statements and multiple if/else blocks.

The solution to this problem is rough around the edges, but works. When the CPU loads a branch instruction into memory, it runs the branch address and several other instructions related to the runtime state through the “branch predictor.” This branch predictor is tasked with determining what instructions to load *after* the branch—essentially, it must predict ahead-of-time the outcome of a conditional jump. If the correct branch is predicted, then

(Note that this is irrelevant for basic

CDN / Web Caching Applications

Taking inspiration from this approach to predicting data access patterns, there is benefit also in the sphere of web and networking. When a web page’s HTML is requested, we actually already know all of the potential next-destinations of the page. By streaming the HTML through a cloud function and parsing out anchor tags, image URLs, and other resources hosted by the origin server, we can “prime” our caches in a similar way to the CPU’s branch prediction: we can pre-fetch those resources from the origin server so that when the user ultimately requests those new assets—either through navigation in the website or by images simply being required for displaying the initial web page—our cache in their location will be pre-hot and no longer stale. Especially as cache size is generally not an issue in the web like it is for CPUs, we can ensure we have all of the resources a user might fetch from a page ready on-hand to serve from the cache, cutting out on a user-side cache miss in exchange for a slightly more complex cache implementation on the backend.

References (So Far)

https://en.wikipedia.org/wiki/Cache_prefetching
https://arxiv.org/pdf/2011.00477.pdf
https://web.dev/articles/predictive-prefetching
https://techdocs.akamai.com/property-mgr/docs/predictive-prefetching