دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر
گزارش پروژه معماری کامپیوتر

عنوان:

پروژه ششم:پیاده‌سازی سیاست‌های جایگزینی در حافظه نهان با استفاده از ChampSim

Project Six: Implementation of Cache Replacement Policies in the ChampSim Simulator

نگارش

محمدمهدی عابدینی ،کسری منتظری هدشی ، محمدپارسا جعفرنژادی ، سیدمحمدرضا جوادی

استاد درس

دکتر اسدی

مسئولان پروژه

آقایان صداقت‌گو و غفوری

تیر ۱۴۰۴

فهرست مطالب

۱ مقدمه

با افزایش سرعت پردازنده‌ها و محدودیت‌های سرعت حافظه‌های اصلی، فرآیند عملکردی بین این دو به چالشی بزرگ در طراحی سیستم‌های کامپیوتری تبدیل شده است. حافظه‌های نهان به عنوان لایه‌ای میان، در کاهش تأخیر دسترسی به حافظه‌ی اصلی و بهبود عملکرد سیستم نقش کلیدی ایفا می‌کنند. سیاست‌های جایگزین حافظه‌ی نهان تصمیم می‌گیرند که کدام بلوک داده حذف و جایگزین شود. انتخاب بهینه این الگوریتم‌ها تاثیر زیادی بر کارایی سیستم دارد. در این پروژه، هدف پیاده‌سازی و مقایسه چند سیاست جایگزین حافظه‌ی نهان است تا عملکرد آن‌ها در سناریوهای مختلف تحلیل شده و به درک بهتری از مدیریت حافظه‌ی نهان در سیستم‌های کامپیوتری برسیم. ابزار اصلی شبیه‌سازی ما برای بررسی سیاست‌های مختلف، چَمپسیم (ChampSim) است که می‌تواند سناریوهای مختلف حافظه نهان و سیاست‌های جایگزین را به صورت دقیق و مفصل شبیه‌سازی کند. با مقایسه نتایج و تحلیل و تفسیر داده ها ما می توانیم سیاست هارا ارزیابی کرده و مفاهیم تئوری درس را در عمل پیاده سازی کنیم و نتایج را به چشم ببینیم.

۱-۱ چشم‌انداز گزارش

در این گزارش، قدم به قدم مراحل انجام پروژه را بررسی می‌کنیم. مراحل پروژه عبارتند از:

۱. لینک‌های مرتبط با هر سیاست را مطالعه می‌کنیم تا با نحوه کارکرد و پیاده‌سازی آن‌ها آشنا شویم.

۲. درک خود از نحوه عملکرد این سیاست‌ها را در گزارش آورده و توضیح می‌دهیم.

۳. این سیاست‌ها را در شبیه‌ساز ChampSim پیاده‌سازی می کنیم.

۴. شبیه‌ساز را با استفاده از این سیاست‌ها و سیاست‌های Random ،LRU و SRRIP که توسط خود شبیه‌ساز پیاده‌سازی شده است، پیکربندی کرده و برنامه‌های مختلف را بر روی آن اجرا می‌کنیم.

۵. عملکرد این سیاست‌ها را با پارامترهای مختلف که توسط شبیه‌ساز ارائه می‌شود، مقایسه و تحلیل می‌کنیم.

پس نتیجتا در این گزارش ما ابتدا به توضیح سیاست های جایگزینی می پردازیم و بعد از آن نحوه

پیاده سازی را گزارش می دهیم و نهایتا با شبیه سازی سیاست های مختلف به نتیجه گیری و تحلیل عملکرد هریک می پردازیم

۱-۲ اهداف

در این پروژه ما اهداف زیر را دنبال می کنیم.

- شناخت و آشنایی با شبیه‌ساز ChampSim و نحوه کار با آن.
- آشنایی با سیاست‌های مختلف جایگزین در حافظه نهان.
- مقایسه و تحلیل عملکرد سیاست‌های مختلف حافظه نهان بر عملکرد کل سیستم.

۲ سیاست‌های جایگزینی

در این بخش نتایج مطالعات خود از لینک های داده شده را شرح می دهیم. به طور خلاصه، در این بخش مطابق گام دوم پروژه انواع سیاست هارا نام می بریم و آن هارا شرح می دهیم. قبل از آن که وارد هرکدام به طور خاص شویم، ابتدا بیایید به طور کلی به شرح مسئله به پردازیم. مهم است که بدانیم اصلا این سیاست ها چه معنی دارند و به طور کلی در پاسخ به چه مشکلی این راه حل ها ارائه شدند.

۲-۱ شرح مسئله

فرکانس کاری CPU بسیار بالاست. در بسیاری از پیاده‌سازی‌ها، مانند پیاده‌سازی چند چرخه‌ای در مرحله حافظه فرصت محدودی برای کار با حافظه وجود دارد و انتظار می‌رود در یک چرخه عملیات انجام شود. با این وجود به دلیل فرکانس کاری بسیار پایین در حافظه‌ها این عمل غیرممکن است. حتی اگر از تکنولوژی‌های جدید برای حافظه استفاده کنیم تا سرعت آن به پردازنده برسد هزینه به شکل سرسام آوری بالا می رود و طراحی را ناکارآمد می کند. اینجا ایده ایجاد واسطه شکل می گیرد. حافظه نهان وظیفه دارد تا با ایجاد واسطه بین پردازنده و حافظه مشکل را رفع کند.
حافظه نهان خود اندازه محدودی دارد، پس برای قرار گیری بلوک های حافظه اصلی ما با محدودیت روبرو هستم. اینجا چندین سوال مطرح می شود:
۱.بلوک در کدام مجموعه قرار گیرد؟
۲.در صورتی که مجموعه متناظر پر بود با کدام بلوک در مجموعه جایگزین شود؟
....۳

ما در این پروژه به سوال دوم می پردازیم. هدف این است که با پیدا کردن سیاست مناسب برای موقعیت‌های مختلف بتوانیم نرخ برخورد را حداکثر کنیم. با وجود ارائه الگوریتم های مختلف باید توجه داشته باشید که در لایه‌های نزدیک حافظه فرصت محدودی داریم و نباید زمان زیادی صرف خود الگوریتم شود یا هزینه‌های سخت‌افزاری زیادی برایمان بتراشد. با بهینه کردن تمامی این عوامل موثر ما به جستجوی سیاست مطلوب می پردازیم. در ادامه چند سیاست که متداولند یا در این پروژه پیاده می شوند می پردازیم.

۲-۲ سیاست‌های جایگزینی

در این بخش به سیاست ها و به مثال هایی از آن ها می پردازیم. هدف این است که درک و توصیف خود از این سیاست ها را ارائه دهیم.
روش پیاده‌سازی به طور کلی استفاده از لیست یا داده ساختاری برای آنها است. درهر بار مراجعه در سیاست‌های مختلف براساس اولویت در داده ساختار درج یا حذف انجام می‌شوند و پیچیدگی های مختلف بر اساس نوع داده ساختار و الگوریتم جستجو و درج در آن است. علاوه بر این در حافظه نهان به دلیل محدودیت‌های سخت‌افزاری پیچیدگی فضا نیز بسیار مهم است.

۲-۲-۱ LRU

سیاست LRU بلوکی را که بیش‌ترین زمان از آخرین استفاده از آن گذشته است، حذف می‌کند.

این سیاست از Temporal locality بهره می‌برد، به طوری که فرض می‌کند بلوک‌هایی که اخیراً از آن‌ها استفاده شده با احتمال خوبی دوباره نیز استفاده خواهند شد و آن‌هایی که زمان زیادی از استفاده از آن‌ها گذشته است، احتمالا دیگر در آینده نزدیک مورد استفاده قرار نخواهند گرفت. این سیاست در مواقعی که دسترسی به یک خانه به طور مداوم و در یک مقطع انجام می‌شود ایده‌آل است.

۲-۲-۲ MRU

سیاست MRU بلوکی را که اخیراً استفاده شده است، برای جایگزینی انتخاب می‌کند، برعکس سیاست LRU.

این نوع سیاست از ساختار های ساده کد بهره می برد. مثلا اگر مراجعه به خانه های حافظه در حلقه باشد متغیری که اخیرا استفاده شده یحتمل در تا اجرای بعدی حلقه استفاده نمی‌شود و می توان آن را کنار گذاشت. طبیعتا این نوع دقیقا خلاف LRU عمل می‌کند و اگر ساختار کد به شکل دسترسی

مداوم به یک خانه باشد نرخ برخورد ما پایین می آید. پس این ساختار به شدت به نوع دسترسی وابسته است.

۳-۲-۲ Random

به طور تصادفی یک بلوک را انتخاب کن و جایگزین کن.

در دنیای ماشین لرنینگ می گویند اگر بین چند فرض، بایاسی نداشته باشیم انتخاب رندوم بهترین امید ریاضی بازدهی را دارد. در حافظه نهان نیز اگر فرض خاصی نکنیم انتخاب رندوم کارآمد است. مهم ترین دلیل کارآمدی این سیاست این است که پیاده‌سازی آنی بسیار آسان و کم هزینه است چه از نظر سخت افزاری و چه پیچیدگی زمانی.

۴-۲-۲ SRRIP

بلوک‌هایی که کمترین مقدار ریفرنسیال را دارند جایگزین کن.

در تلاش برای قرار دادن پارامتری برای مشخص شدن ارزش بلوک، روش LRU می گوید تازگی براساس آخرین استفاده و LFU براساس تعداد استفاده. در این روش با معرفی پارامتر دیگری که ارزش گذاری و آپدیت آن متفاوت از دو الگوریتم گفته شده است ما معیار جدیدی برای جایگزینی انتخاب می کنیم. به این معیار مقدار رفرنسیال گوییم. الگوریتم به این شکل است: مقادیر رفرنسیال کران بالا و پایین دارند مثلا کران بالا مانند ۳. ابتدا مقدار آغازین برای همه خانه ها یکسان است ولی فول نیست مثلا ۲.
حال دوحالت داریم: در هنگام دسترسی: ناگهان مقدار ران کامل می کنیم. در گذر زمان و دسترسی نداشتن: آن را به مقدار پیش بینی شده ای تغییر می دهیم.
این نوع نگاه باعث ترکیبی از تازگی و تکرار می شود
تصویر زیریک نوع پیاده سازی خاص در یک مقاله را هم تحت دایاگرام و هم تحت شبه کد نشان

می دهد

```
// Candidates and their RRPVs are indexed by w in [0, W - 1]

if access is a hit on way w:
      RRPVs[w] = 0

else: // access is a miss
      while (maximum value of RRPVs != 2^M - 1): // 3 if M == 2
            Increment all RRPVs by 1
      // Now there must be at least an RRPV with value 2^M - 1
      Select w as the minimum index s.t. RRPVs[w] == 2^M - 1
      // w-th candidate is selected as the victim
      Replace the old data of candidate w with the new data
      RRPVs[w] = 2^M - 2 // 2 if M == 2
```

The following state-transition diagram illustrates how the RRPVs are managed for 2-bit SRRIP, i.e., when using M=2-bit RRPVs, a common implementation choice.

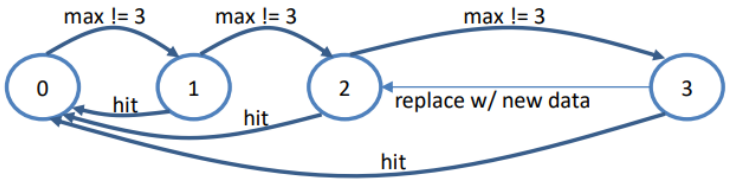


Figure 1. State-transition diagram of an RRPV in 2-bit SRRIP.

LFU ۵-۲-۲

در این سیاست، بلوک‌هایی که کم‌ترین تعداد دسترسی را داشته‌اند، برای جایگزینی انتخاب می‌شوند.

در مبحث Locality معتقدیم گذشته دسترسی‌ها بهترین پیشبینی برای آینده آنها است. در این روش با بهره مندی از این اصل ما به نوعی تاریخچه دسترسی به بلوک را نگهداری می‌کتیم و بر اساس تعداد دسترسی‌ها خانه‌ای که کمترین مراجعه را داشته جایگزین می‌کنیم. در صورت تساوی از LRU استفاده می کنیم یعنی خانه ای که زودتر وارد شده در صورت برابر بودن تعداد مراجعه اول بیرون می رود. این سیاست در الگوهایی که دسترسی به حافظه منظم است و تعداد دسترسی‌ها ناگهانی نیست می تواند مفید باشد اما یک عیب بزرگ دارد، اگر توجه کنید بلوک‌هایی که در ابتدا قرار می گیرند و به کرّات استفاده می‌شوند تا زمان طولانی در حافظه خواهند ماند و بلوک‌های جدیدتر که تازه قرار گرفته‌اند به سرعت خارج می‌شوند. این موضوع به خصوص اگر دسترسی‌ها در یک زمان و پشت سر هم باشد و دیگر به آن خانه رجوع نشود بازدهی بسیار پایینی دارد زیرا جلوی خانه‌هایی که احتمالا الان اسفتاده خواهند شد را می‌گیرد و خانه‌هایی که دیگر نیازی به آن‌ها نبود به خاطر نیاز

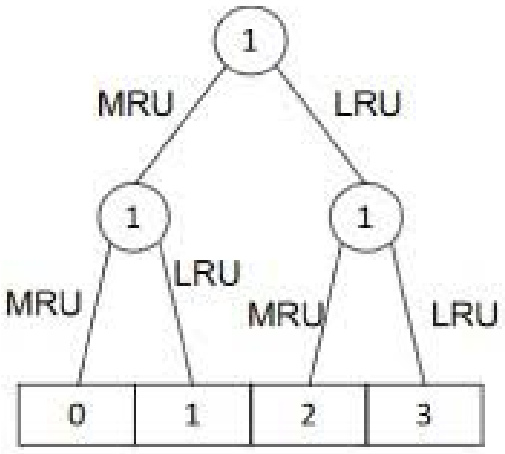
در گذشته جایشان را اشغال می کنند. به نوعی در این سیاست گذشته بسیار مهم است. یک روش پیاده‌سازی می تواند استفاده از شمارنده‌ها و آپدیت کردن هر کدام در هربار استفاده از بلوک باشد.

Tree-PLRU ۶-۲-۲

این سیاست از ساختار درختی برای پیگیری وضعیت بلوک‌ها و تصمیم‌گیری در مورد بلوک جایگزین بهره می‌برد و نوعی LRU است

اولین و بدیهی ترین راهی که ممکن است به ذهن هرکسی برسی LRU است. در ادامه ارائه راه حل ها نیز این روش توسعه داده شده یا اشکال متفاوتی برای آن آمده. یک مشکل در روش اصلی این است که تعداد بیت های زیادی صرف نگهداری سن می شود و هزینه زمانی و سخت افزاری زیادی برایمان دارد. یکی از این ایده ها برای بهبود هزینه، استفاده از داده ساختار های دیگر است. ساختار درختی اجازه می دهد تا هم از تعداد بیت های کمتری استفاده کنیم و هم هزینه جستجوی پایین تری داشته باشیم

یک مثال از نحوه پیاده سازی برای این مورد میزنیم، فرض کنید حافظه نهان ما ۴ سویه است. اگر از روش اصلی استفاده کنیم برای دانستن ترتیب آن ها در لیست برای هر بلوک ۲ بیت نیاز داریم، که برای هر مجموعه می شود ۸ بیت. اما حالا بیایید بجای آن از روش درختی استفاده کنیم. درخت برای هر مجموعه مشابه زیر می شود:



در اینجا ۰ به معنی این است که زیر شاخه سمت چپ از همه زودتر استفاده شده یا LRU است و ۱ به معنی اینکه زیرشاخه راستیLRU است. به این ترتیب باlogn بیت می توانیم LRU را مشخص کنیم. هنگام جستجو نیز کافی است در این درخت جستجو کرده و از هر مسیر که رفتیم بیت ها را تغییر دهیم. همان طور که می بینید بجای ۸ بیت ۳ بیت استفاده کردیم و هزینه سرشکن تغییر و جستجو و ... هم بسیار کمتر از حالت اصلی است.
در نهایت مانند سیاست اصلی بلوکی که کمترین استفاده را اخیرا داشته پس از جستجو جایگزین می شود

ARC ۷-۲-۲

سیاست LRU تمرکز بر تازگی دارد و سیاست LFU تمرکز بر تکرار. هردو مزایای خود را دارند. سیاست ARC با ترکیب این دو و مدیریت هوشمند حافظه نهان برای هربلوک سیاست بهینه را انتخاب می کند

در این سیاست ما ابتدا کش را به دو قسمت تقسیم می کنیم. بخش اول بخشی است که تمرکز بر تازگی دارد و سیاست LRU در آن اجرا می شود. بخش دوم برای اجرای سیاست تکرر است و سیاست LFU اجرا می شود. اندازه این بخش ها ثابت نخواهد بود ولی در آغاز برایشان مرزی تعیین می کنیم که بعدا طی عملیات ها جابجا می شود.

- T۱ (recently used)
  - آیتم‌هایی که فقط یک‌بار اخیراً استفاده شده‌اند.
- T۲ (frequently used)
  - آیتم‌هایی که بیش از یک‌بار استفاده شده‌اند.
- B۱ (ghost list for T۱)
  - آیتم‌هایی که از T۱ حذف شده‌اند، اما هنوز در حافظه کش نیستند؛ فقط نام آن‌ها نگه داشته می‌شود (نه داده‌شان).
- B۲ (ghost list for T۲)

– مشابه B۱، اما برای آیتم‌های حذف‌شده از T۲.

T۱ و B۱ با همدیگر بخش اول، و T۲ و B۲ بخش دوم را می‌سازند. همچنین در الگوریتم ARC یک پارامتر تطبیقی به نام p دارد که نسبت بین T۱ و T۲را تعیین می‌کند.

اگر آیتم‌های B۱ زیاد hit بخورند (یعنی آیتم‌هایی که فقط یک‌بار استفاده شده‌اند دوباره درخواست شوند)، مقدار p را افزایش می‌دهد

با اینکار فضای بیشتری به T۱ اختصاص میابد که باعث میشود توجه به recency افزایش یابد.

به طرز مشابه با hit شدن ایتم های B۲ ، مقدار p کاهش میبابد تا اندازه ماکسیمم سایز T۲ زیاد شود و باعث افزایش توجه به Frequency خواهد شد بدین ترتیب به صورت کش را به‌صورت پویا بین دو لیست اصلی تقسیم می‌کند

LARC ۸-۲-۲

نسخه تغییریافته ARC است. تفاوت این است که به محض اتفاق افتادن Cache miss حافظه نهان را به‌روزرسانی نمی کند.

در تصویر زیر از مقاله اصلی کلیت پیادهسازی سیاست مشهود است این الگوریتم در حقیقت مشابه ARC دارای دو صف اصلی است که هر دو مطابق FIFO عمل می‌کنند اما یکی از آنها تنها دسترسی‌های ۲ یا بیشتر را نگه می‌دارد. تفاوت این سیاست این است که صف اول (معادل T۱) خارج از حافظه نهان فیزیکی نگهداری می‌شود، بدین معنی که این صف در ساختار سیاست جایگزینی وجود دارد و در آن اشاره‌گرهایی به بلوک‌های حافظه (یعنی بخشی از آدرس فیزیکی) نگه‌داری می‌شود، اما این بلوک‌ها هنوز درون خود حافظه نهان بارگیری نشده‌اند. در حقیقت این سیاست تنها بلوک‌هایی که بیش از یک بار استفاده بشوند را وارد حافظه نهان می‌کند و در غیر این صورت آنها را خارج از آن نگه‌می‌دارد.
نام صف حافظه فیزیکی Q و نام صف مجازی Q_r است.حداکثر طول صف مجازی را C_r می‌نامیم که پارامتر مهمی است، چرا که اگر زیاد باشد آنگاه بعد مدتی تعداد زیادی از بلوک‌ها در این صف خواهند بود که باعث می‌شود سیاست مانند LRU عمل کند. از طرفی اگر اندازه آن را کم بگیریم

بلوک‌هایی که کمی قبل‌تر آمده‌اند شانس ورودی به حافظه نهان را ندارند. اما مقدار بهینه با توجه به برنامه‌های مختلف متفاوت است چرا که در هر بخش از هر برنامه فواصل بین استفاده دوباره از یک بلوک متفاوت است. این فاصله Reuse Distance نام دارد و برنامه باید مقدار C_r را با توجه به این تنظیم کند.

برای تنظیم پویای این مقدار از روابطی که در تصویر نوشته شده استفاده می‌شود که به طور کلی بدین‌ترتیب عمل می‌کند:

- اگر بلوک فعلی درون Q یافت شد، اندازه C_r را کاهش بده. هر چه مقدار قبلی آن بیشتر بود آن را بیشتر کاهش بده.
- در غیر این صورت اندازه C_r را افزایش بده، هر چه مقدار قبلی آن کمتر بود آن را بیشتر افزایش بده.

Algorithm 1: Routine to access block B

Initilize: $C_r = 0.1 * C$
if B is in Q **then**
  move B to the MRU end of Q
  $C_r = \max(0.1 * C, C_r - \frac{C}{C - C_r})$
  redirect the request to the corresponding SSD block
  return
end if
$C_r = \min(0.9 * C, C_r + \frac{C}{C_r})$
if B is in Q_r **then**
  remove B from Q_r
  insert B to the MRU end of Q
  if $|Q| > C$ **then**
    remove the LRU block D in Q
    allocate the SSD block of D to B
  else
    allocate a free SSD block to B
  end if
else
  insert B to the MRU end of Q_r
  if $|Q_r| > C_r$ **then**
    remove the LRU block of Q_r
  end if
end if

۳ پیاده‌سازی

در این مرحله ما باید براساس مستندات چمپسیم و سیاست های پیاده سازی شده آماده، سیاست های جدید را پیاده می کردیم. پس این بخش دو زیربخش دارد:

۱. مطالعه مستندات و بررسی پیاده سازی های ازپیش آماده

۲. پیاده سازی سیاست های جدید

۳-۱ مطالعه مستندات و آشنایی با پیاده‌سازی سیاست‌ها

در این مرحله با مطالعه مستندات ، راهنمای پیاده سازی ، سایت مفید برای مقالات و مستندات ، راهنمای خود ریپازیتوری و تمامی مقالات اشاره شده در صورت پروژه با نحوه پیاده سازی سیاست ها در این شبیه ساز آشنا شدیم
ابتدا با بررسی LRU Random، و SRRIP با نحوه پیاده سازی آشنا شدیم. قبل از آن اشاره کنیم کارکرد تک تک توابع را در داکیومنت های چمپسیم می توانید مشاهده کنید.

۳-۱-۱ Random

الگوریتم این سیاست در بخش پیشین توضیح داده شده است در اینجا به بررسی نحوه پیاده سازی می پردازیم.
هر سیاست برای پیاده سازی نیاز به یک فایل هدر برای پروتوتایپ توابع نیاز دارد و یک فایل cc که توابع را پیاده سازی کند.

```cpp
#include "random.h"

random::random(CACHE* cache) : random(cache, cache->NUM_WAY) {}

random::random(CACHE* cache, long ways) : replacement(cache), dist(0, ways - 1) {}

long random::find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const CACHE::BLOCK* current_set, uint64_t ip, uint64_t full_addr,
                         access_type type)
{
  return dist(rng);
}
```

فایل هدر

```h
#ifndef REPLACEMENT_RANDOM_H
#define REPLACEMENT_RANDOM_H

#include <random>

#include "cache.h"
#include "modules.h"

struct random : public champsim::modules::replacement {
  std::mt19937_64 rng{};
  std::uniform_int_distribution<long> dist;

  explicit random(CACHE* cache);
  random(CACHE* cache, long ways);

  // void initialize_replacement();
  long find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const CACHE::BLOCK* current_set, uint64_t ip, uint64_t full_addr, access_type type);
  // void update_replacement_state(uint32_t triggering_cpu, long set, long way, uint64_t full_addr, uint64_t ip, uint64_t victim_addr, access_type type, uint8_t
  // hit);
  //  void replacement_final_stats()
};

#endif
```

فایل پیاده سازی

همانطور که می بینید رندوم یکی را برمی گرداند.

LRU ۲-۱-۳

الگوریتم این سیاست در بخش پیشین توضیح داده شده است.

```h
#ifndef REPLACEMENT_LRU_H
#define REPLACEMENT_LRU_H

#include <vector>

#include "cache.h"
#include "modules.h"

class lru : public champsim::modules::replacement
{
  long NUM_WAY;
  std::vector<uint64_t> last_used_cycles;
  uint64_t cycle = 0;

public:
  explicit lru(CACHE* cache);
  lru(CACHE* cache, long sets, long ways);

  // void initialize_replacement();
  long find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                   champsim::address full_addr, access_type type);
  void replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                              access_type type);
  void update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                access_type type, uint8_t hit);
  // void replacement_final_stats()
};

#endif
```

فایل هدر

```cpp
#include "lru.h"

#include <algorithm>
#include <cassert>

lru::lru(CACHE* cache) : lru(cache, cache->NUM_SET, cache->NUM_WAY) {}

lru::lru(CACHE* cache, long sets, long ways) : replacement(cache), NUM_WAY(ways), last_used_cycles(static_cast<std::size_t>(sets * ways), 0) {}

long lru::find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                      champsim::address full_addr, access_type type)
{
  auto begin = std::next(std::begin(last_used_cycles), set * NUM_WAY);
  auto end = std::next(begin, NUM_WAY);

  // Find the way whose last use cycle is most distant
  auto victim = std::min_element(begin, end);
  assert(begin <= victim);
  assert(victim < end);
  return std::distance(begin, victim);
}

void lru::replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                 access_type type)
{
  // Mark the way as being used on the current cycle
  last_used_cycles.at((std::size_t)(set * NUM_WAY + way)) = cycle++;
}

void lru::update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip,
                                   champsim::address victim_addr, access_type type, uint8_t hit)
{
  // Mark the way as being used on the current cycle
  if (hit && access_type{type} != access_type::WRITE) // Skip this for writeback hits
    last_used_cycles.at((std::size_t)(set * NUM_WAY + way)) = cycle++;
}
```

فایل پیاده سازی

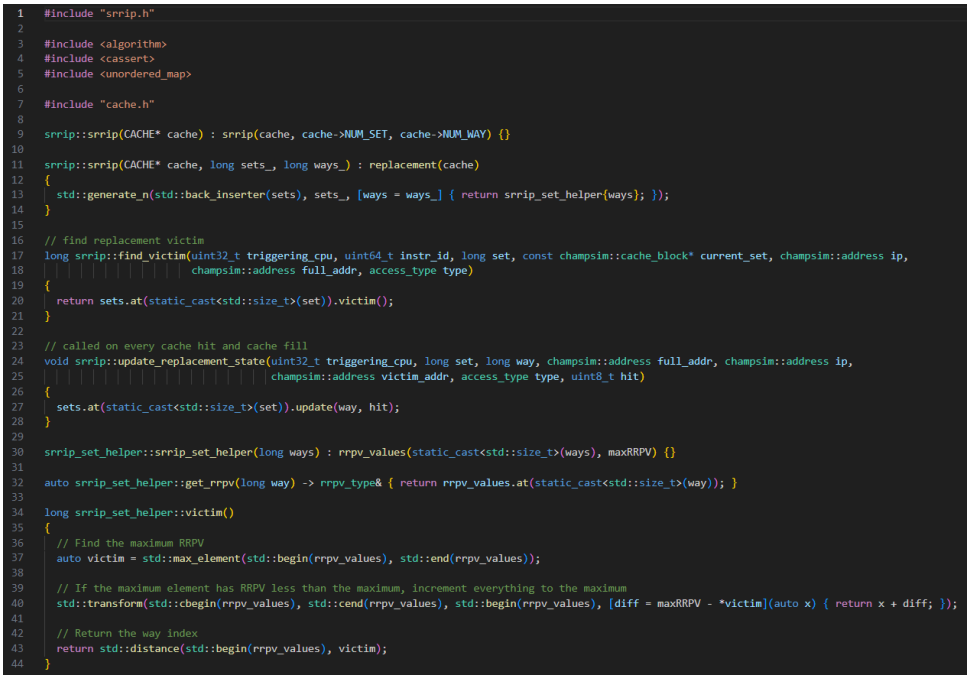
می بینید که در این پیاده سازی آرایه ای نگهداری می کند که هرچه مقدار آن کمتر باشد یعنی بیشترین مدت از طول عمر آن می گذرد. هرگاه نوبت جایگزینی رسد از اعضای آن مینیمم گرفته و آن را جایگزین می کند. توجه کنید اینجا تابع آپدیت شدن وضعیت نیز پیاده شده که اگر برخورد هم داشت باید سن آن یکی افزوده شود.

SRRIP ۳-۱-۳

الگوریتم این سیاست در بخش پیشین توضیح داده شده است.

```h
#ifndef REPLACEMENT_SRRIP_H
#define REPLACEMENT_SRRIP_H

#include <cstdint>
#include <vector>

#include "cache.h"
#include "modules.h"

struct srrip_set_helper {
  using rrpv_type = int;
  static constexpr rrpv_type maxRRPV = 3;

  std::vector<rrpv_type> rrpv_values;
  rrpv_type& get_rrpv(long way);

  explicit srrip_set_helper(long ways);

  long victim();
  void update(long way, bool hit);
};

struct srrip : public champsim::modules::replacement {

  std::vector<srrip_set_helper> sets;

  explicit srrip(CACHE* cache);
  srrip(CACHE* cache, long sets_, long ways_);

  // void initialize_replacement() {}
  long find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                   champsim::address full_addr, access_type type);
  void update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                access_type type, uint8_t hit);

  // use this function to print out your own stats at the end of simulation
  // void replacement_final_stats() {}
};

#endif
```

فایل هدر



```cpp
#include "srrip.h"

#include <algorithm>
#include <cassert>
#include <unordered_map>

#include "cache.h"

srrip::srrip(CACHE* cache) : srrip(cache, cache->NUM_SET, cache->NUM_WAY) {}

srrip::srrip(CACHE* cache, long sets_, long ways_) : replacement(cache)
{
  std::generate_n(std::back_inserter(sets), sets_, [ways = ways_] { return srrip_set_helper{ways}; });
}

// find replacement victim
long srrip::find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                        champsim::address full_addr, access_type type)
{
  return sets.at(static_cast<std::size_t>(set)).victim();
}

// called on every cache hit and cache fill
void srrip::update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip,
                                     champsim::address victim_addr, access_type type, uint8_t hit)
{
  sets.at(static_cast<std::size_t>(set)).update(way, hit);
}

srrip_set_helper::srrip_set_helper(long ways) : rrpv_values(static_cast<std::size_t>(ways), maxRRPV) {}

auto srrip_set_helper::get_rrpv(long way) -> rrpv_type& { return rrpv_values.at(static_cast<std::size_t>(way)); }

long srrip_set_helper::victim()
{
  // Find the maximum RRPV
  auto victim = std::max_element(std::begin(rrpv_values), std::end(rrpv_values));

  // If the maximum element has RRPV less than the maximum, increment everything to the maximum
  std::transform(std::cbegin(rrpv_values), std::cend(rrpv_values), std::begin(rrpv_values), [diff = maxRRPV - *victim](auto x) { return x + diff; });

  // Return the way index
  return std::distance(std::begin(rrpv_values), victim);
}
```

فایل پیاده سازی

در این پیاده سازی به دلیل پیچیدگی داده ساختار ها و متغیر ها و ... بیشتری استفاده شده و پیاده سازی الگوریتم با استفاده از فرایند های کمکی انجام گرفته تا آپدیت شدن ها به طور خودکار در هر برخورد یا خطارفت انجام شود.

۳-۲ پیاده‌سازی سیاست‌های جدید

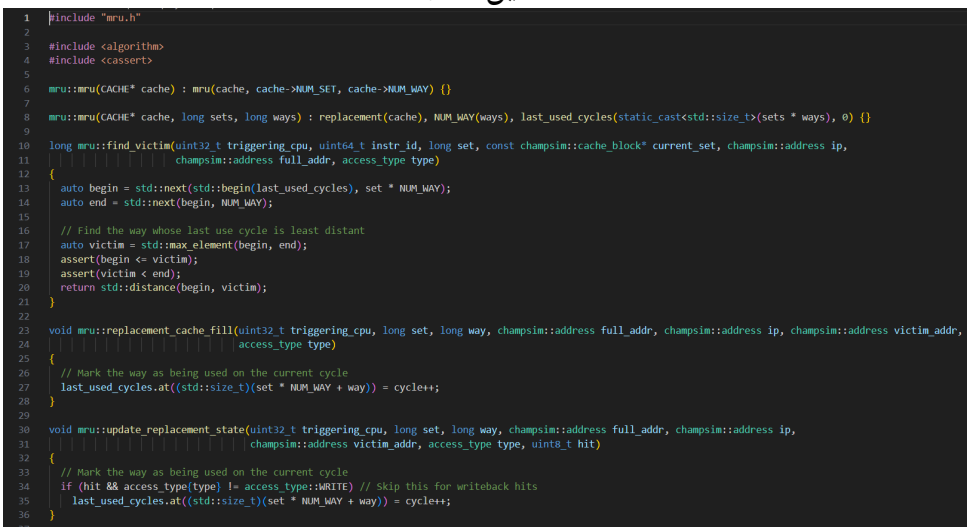
حال وقت پیاده سازی سیاست های جدید است.

۳-۲-۱ MRU

پیاده سازی این سیاست آسان بود زیرا همان LRU را با ماکس گرفتن به MRU می توان تبدیل کرد. پس با تغییر نام و استفاده از تابع ماکسیمم به جای مینمم این سیاست را پیاده کردیم.

```h
#ifndef REPLACEMENT_MRU_H
#define REPLACEMENT_MRU_H

#include <vector>

#include "cache.h"
#include "modules.h"

class mru : public champsim::modules::replacement
{
  long NUM_WAY;
  std::vector<uint64_t> last_used_cycles;
  uint64_t cycle = 0;

public:
  explicit mru(CACHE* cache);
  mru(CACHE* cache, long sets, long ways);

  // void initialize_replacement();
  long find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                   champsim::address full_addr, access_type type);
  void replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                              access_type type);
  void update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                access_type type, uint8_t hit);
  // void replacement_final_stats()
};

#endif
```

فایل هدر



```cpp
#include "mru.h"

#include <algorithm>
#include <cassert>

mru::mru(CACHE* cache) : mru(cache, cache->NUM_SET, cache->NUM_WAY) {}

mru::mru(CACHE* cache, long sets, long ways) : replacement(cache), NUM_WAY(ways), last_used_cycles(static_cast<std::size_t>(sets * ways), 0) {}

long mru::find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                      champsim::address full_addr, access_type type)
{
  auto begin = std::next(std::begin(last_used_cycles), set * NUM_WAY);
  auto end = std::next(begin, NUM_WAY);

  // Find the way whose last use cycle is least distant
  auto victim = std::max_element(begin, end);
  assert(begin <= victim);
  assert(victim < end);
  return std::distance(begin, victim);
}

void mru::replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                 access_type type)
{
  // Mark the way as being used on the current cycle
  last_used_cycles.at((std::size_t)(set * NUM_WAY + way)) = cycle++;
}

void mru::update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip,
                                   champsim::address victim_addr, access_type type, uint8_t hit)
{
  // Mark the way as being used on the current cycle
  if (hit && access_type{type} != access_type::WRITE) // Skip this for writeback hits
    last_used_cycles.at((std::size_t)(set * NUM_WAY + way)) = cycle++;
}
```

فایل پیاده سازی

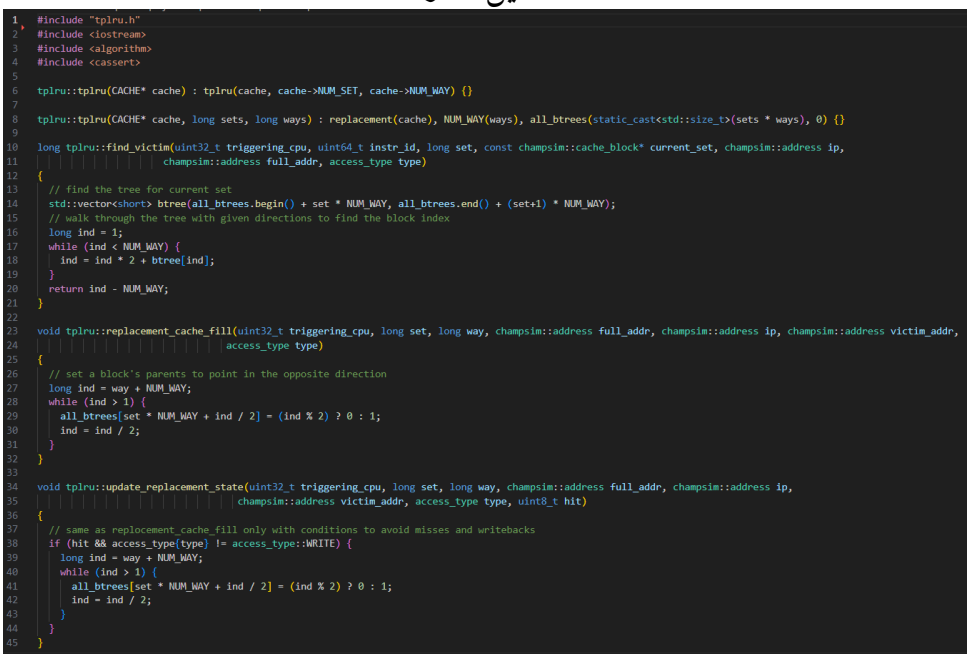
Tree-PLRU ۲-۲-۳

همان طور که در الگوریتم این یک نسخه تغییریافته برای LRU است. در فایل هدر علاوه بر توابع، یک وکتور برای ذخیره سازی همه درخت ها وجود دارد. در فایل پیاده سازی می بینید که در ابتدا بر اساس ویژگی های حافظه درخت هارا می سازیم.

در واقع، این رویکرد تلاش می‌کند تا علاوه بر بهره‌برداری از منطق LRU، از ساختارهای درختی برای افزایش کارایی، کاهش زمان جستجو، و بهبود تصمیم‌گیری در فرآیند جایگزینی و مدیریت حافظه بهره‌مند شود. نتیجه نهایی، الگوریتمی است که نه تنها به صورت خطی بلکه به کمک ساختارهای درخت، عملیات را انجام می‌دهد و پاسخ‌های بهینه‌تری در مواجهه با حالت‌های مختلف حافظه ارائه می‌دهد. منطق کلی با LRU یکی است تفاوت در نوع داده ساختار است که اینجا بجای مینمم گرفتن از یک لیست یا افزودن مقدار در لیست، به جستجوی درختی یا درج در درخت می پردازیم.

```h
#ifndef REPLACEMENT_TPLRU_H
#define REPLACEMENT_TPLRU_H

#include <vector>

#include "cache.h"
#include "modules.h"

class tplru : public champsim::modules::replacement
{
  long NUM_WAY;
  std::vector<short> all_btrees;
  /* above is the vector that contains all binary tree
  sequentially, tree with index n can be found at index n*NUM_WAY */

public:
  explicit tplru(CACHE* cache);
  tplru(CACHE* cache, long sets, long ways);

  // void initialize_replacement();
  long find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                   champsim::address full_addr, access_type type);
  void replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                              access_type type);
  void update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                access_type type, uint8_t hit);
  // void replacement_final_stats()
};

#endif
```

فایل هدر



```cpp
#include "tplru.h"
#include <iostream>
#include <algorithm>
#include <cassert>

tplru::tplru(CACHE* cache) : tplru(cache, cache->NUM_SET, cache->NUM_WAY) {}

tplru::tplru(CACHE* cache, long sets, long ways) : replacement(cache), NUM_WAY(ways), all_btrees(static_cast<std::size_t>(sets * ways), 0) {}

long tplru::find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                        champsim::address full_addr, access_type type)
{
  // find the tree for current set
  std::vector<short> btree(all_btrees.begin() + set * NUM_WAY, all_btrees.end() + (set+1) * NUM_WAY);
  // walk through the tree with given directions to find the block index
  long ind = 1;
  while (ind < NUM_WAY) {
    ind = ind * 2 + btree[ind];
  }
  return ind - NUM_WAY;
}

void tplru::replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                   access_type type)
{
  // set a block's parents to point in the opposite direction
  long ind = way + NUM_WAY;
  while (ind > 1) {
    all_btrees[set * NUM_WAY + ind / 2] = (ind % 2) ? 0 : 1;
    ind = ind / 2;
  }
}

void tplru::update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip,
                                     champsim::address victim_addr, access_type type, uint8_t hit)
{
  // same as replocement_cache_fill only with conditions to avoid misses and writebacks
  if (hit && access_type{type} != access_type::WRITE) {
    long ind = way + NUM_WAY;
    while (ind > 1) {
      all_btrees[set * NUM_WAY + ind / 2] = (ind % 2) ? 0 : 1;
      ind = ind / 2;
    }
  }
}
```

فایل پیاده سازی

LFU ۳-۲-۳

این را هم مشابه MRU به سادگی از تغییر در LRU بدست می آوریم. در LRU ما هنگام قرار دادن در کش به آن یک امتیاز اضافه می کردیم اما اینجا امتیاز آن باید از صفر شروع شود و بعد از این درهربار استفاده یکی افزوده شود. اینجا امتیاز ها نمایانگر تکرر استفاده هستند نه ترتیب آخرین استفاده.

```h
#ifndef REPLACEMENT_LFU_H
#define REPLACEMENT_LFU_H

#include <vector>

#include "cache.h"
#include "modules.h"

class lfu : public champsim::modules::replacement
{
  long NUM_WAY;
  std::vector<uint64_t> last_used_cycles;
  uint64_t cycle = 0;

public:
  explicit lfu(CACHE* cache);
  lfu(CACHE* cache, long sets, long ways);

  // void initialize_replacement();
  long find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                   champsim::address full_addr, access_type type);
  void replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                              access_type type);
  void update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                access_type type, uint8_t hit);
  // void replacement_final_stats()
};

#endif
```

فایل هدر

```cpp
#include "lfu.h"

#include <algorithm>
#include <cassert>

lfu::lfu(CACHE* cache) : lfu(cache, cache->NUM_SET, cache->NUM_WAY) {}

lfu::lfu(CACHE* cache, long sets, long ways) : replacement(cache), NUM_WAY(ways), last_used_cycles(static_cast<std::size_t>(sets * ways), 0) {}

long lfu::find_victim(uint32_t triggering_cpu, uint64_t instr_id, long set, const champsim::cache_block* current_set, champsim::address ip,
                      champsim::address full_addr, access_type type)
{
  auto begin = std::next(std::begin(last_used_cycles), set * NUM_WAY);
  auto end = std::next(begin, NUM_WAY);

  // Find the way whose last use cycle is most distant
  auto victim = std::min_element(begin, end);
  assert(begin <= victim);
  assert(victim < end);
  return std::distance(begin, victim);
}

void lfu::replacement_cache_fill(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip, champsim::address victim_addr,
                                 access_type type)
{
  // Mark the way as being used on the current cycle
  last_used_cycles.at((std::size_t)(set * NUM_WAY + way)) = 0;
}

void lfu::update_replacement_state(uint32_t triggering_cpu, long set, long way, champsim::address full_addr, champsim::address ip,
                                   champsim::address victim_addr, access_type type, uint8_t hit)
{
  // Mark the way as being used on the current cycle
  if (hit && access_type{type} != access_type::WRITE) // Skip this for writeback hits
    last_used_cycles.at((std::size_t)(set * NUM_WAY + way))++;
}
```

فایل پیاده سازی

۳-۲-۴ ARC

سیاست جایگزینی ARC با توجه به مقاله اصلی این سیاست

Algorithm Cache Replacement Adaptive an with LRU Outperforming

نوشته شده است. در ادامه PseudoCode مربوطه آورده شده است:

ARC(c) INITIALIZE $T_1 = B_1 = T_2 = B_2 = 0, p = 0$. x - requested page.

Case I. $x \in T_1 \cup T_2$ (a hit in ARC(c) and DBL($2c$)): Move x to the top of T_2.

Case II. $x \in B_1$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \min\{c, p + \max\{|B_2|/|B_1|, 1\}\}$. REPLACE($p$). Move x to the top of T_2 and place it in the cache.

Case III. $x \in B_2$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \max\{0, p - \max\{|B_1|/|B_2|, 1\}\}$. REPLACE($p$). Move x to the top of T_2 and place it in the cache.

Case IV. $x \in L_1 \cup L_2$ (a miss in DBL($2c$) and ARC(c)):

case (i) $|L_1| = c$:
If $|T_1| < c$ then delete the LRU page of B_1 and REPLACE(p).
else delete LRU page of T_1 and remove it from the cache.

case (ii) $|L_1| < c$ and $|L_1| + |L_2| \geq c$:
if $|L_1| + |L_2| = 2c$ then delete the LRU page of B_2.
REPLACE(p).

Put x at the top of T_1 and place it in the cache.

Subroutine REPLACE(p)

if $(|T_1| \geq 1)$ and $((x \in B_2$ and $|T_1| = p)$ or $(|T_1| > p))$ then move the LRU page of T_1 to the top of B_1 and remove it from the cache.

else move the LRU page in T_2 to the top of B_2 and remove it from the cache.

این سیاست جایگزینی از دو لیست اصلی T_1 و T_2 و دو لیست "روح" B_1 و B_2 برای پیاده سازی الگوریتم خود استفاده می‌کند. اجتماع دو لیست اول حافظه نهان ما را تشکیل می‌دهند و مقادیر درایه های حافظه نهان را ضبط می‌کنند. دو لیست دوم نیز متناظر در ادامه حافظه اصلی می‌آیند و نقش آن ها تنها نگهداری metadata درایه های حافظه است (یعنی تنها آدرس و برچسب یک درایه).

اگر قرار دهیم:

$$C = W \times S$$

where C is the cache size, W is the number of ways, and S is the number of sets.

آنگاه می‌توان قرارگیری این چهار لیست را بدین گونه تصور کرد که T_1 و T_2 در کنار همدیگر C ای به اندازه c تشکیل می‌دهند، و ابتدای B_1 به انتهای T_1 و ابتدای B_2 به انتهای T_2 متصل است.

وظیفه T_1 نگه داشتن درایه هایی است که اخیرا یک بار استفاده شده اند. T_2 درایه هایی را نگه می‌دارد که اخیرا بیش از یک بار نگهداری شده اند. B_1 و B_2 هم به ترتیب برچسبی از درایه های خارج شده از T_1 و T_2 نگه می‌دارند. همچنین پارامتر p در این سیاست، هدف ما از نسبتی از C

که باید به T_1 مختص یابد را نگهداری می‌کند. در طول زمان و با پیشروی stream این نسبت بروز رسانی می‌شود تا جای بیشتری به لیست مهم تر اختصاص دهد.

در ادامه منطقی که با درخواست پردازنده از درایه x طی می‌شود را آورده ایم:

Algorithm 1 REPLACE(p)

if $|T_1| \geqslant 1$ **and** $((x \in B_2$ **and** $|T_1| = p)$ **or** $|T_1| > p)$

Move the LRU page of T_1 to the top of B_1

Remove it from the cache

else

Move the LRU page of T_2 to the top of B_2

Remove it from the cache

تابع بالا، منطق تخلیه کردن یک درایه در سیاست ARC است. هنگام پر بودن حافظه نهان اگر لیست T_1 نسبتی بیش از میزان دلخواه ما (p) از حافظه نهان را اشغال کرده است، پایین ترین عضو آن را حذف کرده و وارد لیست روح B_1 می‌کنیم. در غیر این صورت T_2 بیش از میزان مطلوب حافظه نهان را اشغال کرده است و باید پایین ترین درایه آن را حذف کرده و در لیست روح B_2 قرار دهیم.

حال منطق اصلی سیاست:

۱. حالت نخست: x در T_1 یا T_2 قرار دارد

این به معنی یک hit است. درایه x باید از هر کجایی در این دو لیست که قرار دارد خارج شود و به ابتدای لیست T_2 برود، زیرا hit به این معنی است که این درایه اخیرا بیش از یک بار استفاده شده است.

۲. حالت دوم: x در B_1 قرار دارد این به معنی miss است. درایه x در حافظه نهان نیست، اما اخیرا در آن حضور داشته است. در این صورت باید نسبت مطلوب T_1 افزایش یابد - یعنی افزایش مقدار p. سپس تابع $REPLACE$ صدا زده می‌شود و درایه x به آغاز لیست T_2 می‌رود

۳. حالت سوم: x در B_2 قرار دارد این به معنی miss است. به طور مشابه درایه x در حافظه نهان نیست، اما اخیرا در آن حضور داشته است. در این صورت باید ینسبت مطلوب T_2 افزایش یابد - یعنی کاهش مقدار p. سپس تابع $REPLACE$ صدا زده می‌شود و درایه x به آغاز لیست T_2 می‌رود

۴. حالت چهارم: x در هیچ یک از لیست ها قرار ندارد این به معنی miss است اما این بار درایه x

اخیرا استفاده نشده است. به کمک تابع $REPLACE$ یک مکان در حافظه نهان خالی می‌شود و درایه x به دلیل این که تنها یک بار اخیرا استفاده شده است در اول لیست T_1 قرار می‌گیرد

بدین گونه سیاست ARC به شکل انعطاف پذیر ترکیبی از Used Recently Least و Least Used Frequently را به ما ارائه می‌دهد که با پیشروی برنامه قدرت هر یک از این سیاست ها به کمک پارامتر p در حافظه نهان بیشتر یا کمتر می‌شود.

LARC ۵-۲-۳

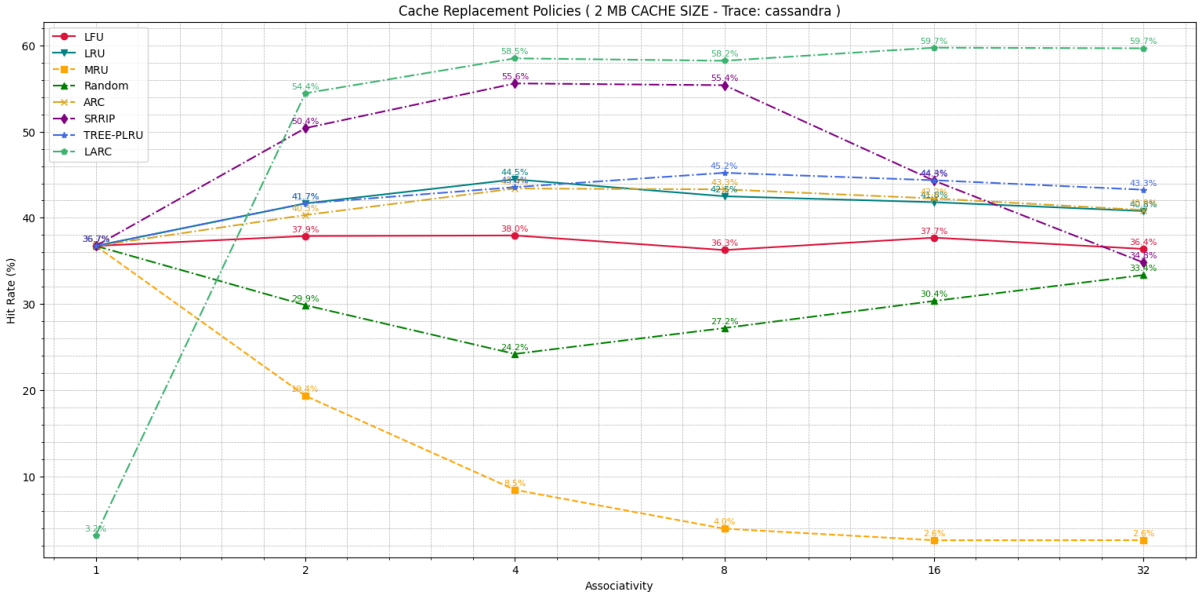
برای این سیاست سعی کردیم مطابق شبه‌کد درون مقاله با داده‌ساختارهای صف‌گونه الگوریتم را پیاده‌سازی کنیم.
برای ذخیره داده‌های Q و Q_r برای تمام مجموعه‌ها، یک متغیرها را از جنس Vector تعریف می‌کنیم که خود حاوی چندین Vector است که هر کدام متناظر این صف‌ها برای یک مجموعه خاص هستند. از آنجایی که اعضای Q درون حافظه نهان وجود دارند، برای ذخیره‌سازی آن‌ها کافی است که اندیس آن‌ها درون مجموعه (یعنی مقدار (way را ذخیره کنیم. هر زمان که بلوک جدیدی وارد شد یا Hit اتفاق افتاد آن را به سر این صف می‌بریم. همچنین مقدار اولیه این لیست‌ها به دلیل پیاده‌سازی مستقیم‌تر حاوی تمامی اندیس‌ها می‌باشد.
برای Q_r نیز به شکل مشابه یک متغیر در کلاس تعریف می‌کنیم، با این فرق که اعضای آن باید نشان‌دهنده خود بلوک‌های حافظه باشند، برای این کار از داده‌ساختار `champsim::address` استفاده می‌کنیم. این نوع داده حاوی آدرس فیزیکی کامل است که به دلیل این که در حافظه‌نهان سطح آخر (LLC) هستیم بیت‌های آفست این آدرس‌ها صفر خواهند بود که یعنی هر آدرس نشان‌دهنده یک بلوک در حافظه اصلی خواهد بود.
حال کافی است شروط الگوریتم را پیاده کنیم. شرط Hit در تابع `update_replacement_state` پیاده‌سازی شده است و Miss معادل تابع `find_victim` است. برای این که یک بلوک را اصلا وارد حافظه نهان نکنیم کافی است تعداد Way را به عنوان اندیس قربانی در این تابع خروجی بدهیم. همچنین نکته‌ای در اینجا وجود دارد آن هم این است که ممکن است نیاز به Write داشته باشیم اما این سیاست اجازه ورود به حافظه نهان را ندهد، در این صورت با خطا مواجه می‌شویم چرا که ساختار حافظه شبیه‌ساز ChampSim از نوع Write back است و ممکن نیست بدون ورود به حافظه نهان در بلوکی بنویسیم. برای حل این مشکل شرطی اضافه کردیم که در صورت Write، همیشه انتقال به حافظه نهان اتفاق بیفتد.

۴ شبیه‌سازی و پارامترها

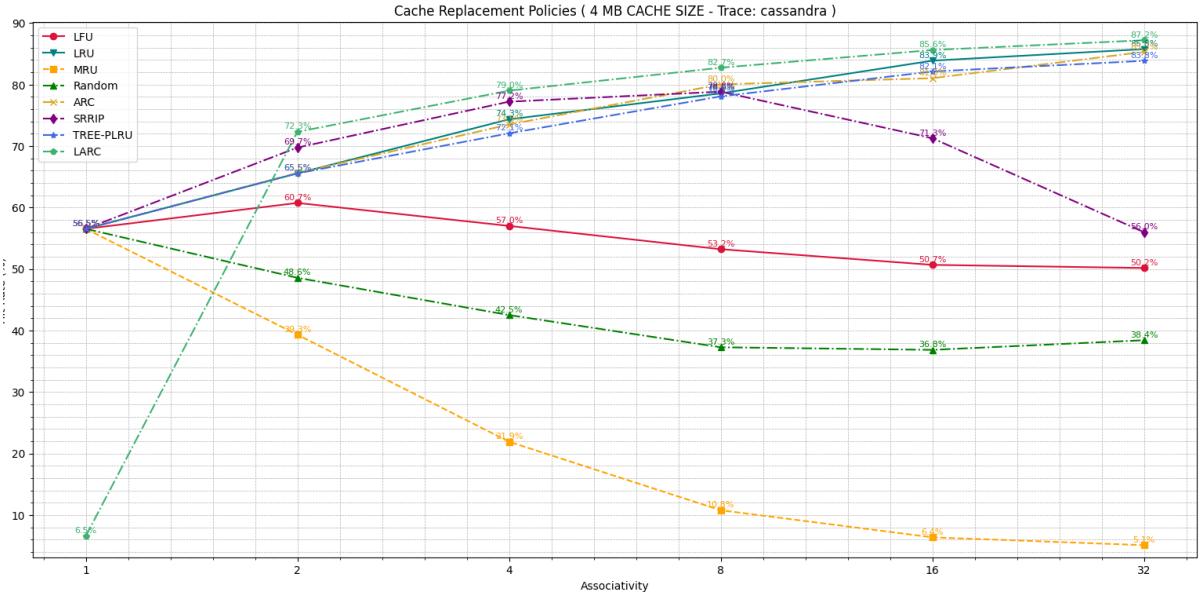
برای شبیه‌سازی و کسب داده موارد زیر در نظر گرفته شده است:

- Warmup Instructions: که تعداد دستورات گرم کردن است، در همه شبیه‌سازی‌های برابر $800,000$ گرفته شده است.
- Simulation Instructions: که تعداد دستورات اصلی است، در تمام شبیه‌سازی‌های برابر $2,400,000$ گرفته شده است.
- Associativity: که نشان‌دهنده تعداد بلوک در هر مجموعه است، برای بررسی عملکرد سیاست‌ها، این پارامتر را با مقادیر $1, 2, 4, 8, 16, 32$ مقداردهی کردیم تا نمودار نرخ برخورد را بر حسب این مقادیر رسم کنیم.
- Block Size: اندازه بلوک‌های حافظه نهان را در تمامی شبیه‌سازی‌ها برابر 64 بایت گرفتیم.
- Cache size(Set count): اندازه کل حافظه نهان لایه آخر معادل ضرب سایز بلوک در تعداد مجموعه و تعداد بلوک هر مجموعه است، شبیه‌سازی‌ها را در دو سری 2MB و 4MB برای اندازه حافظه اجرا کردیم که با تغییر تعداد مجموعه‌ها انجام شد. (در هر سری ضرب Associativity در تعداد مجموعه باید ثابت بماند)
- Trace: عبارت است از مجموعه دستوراتی که برای شبیه‌سازی استفاده می‌شوند. ما در این پروژه از دو مجموعه Cassandra و Streaming برای مقایسه مجموعه‌های مختلف استفاده شده است.

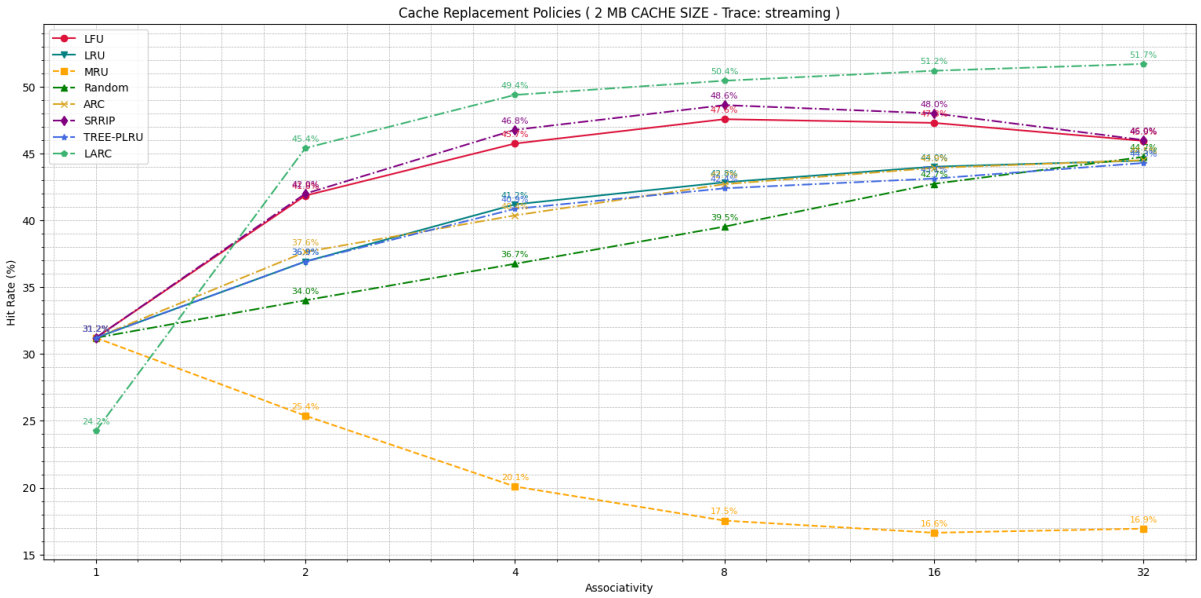
۵ تحلیل نتایج



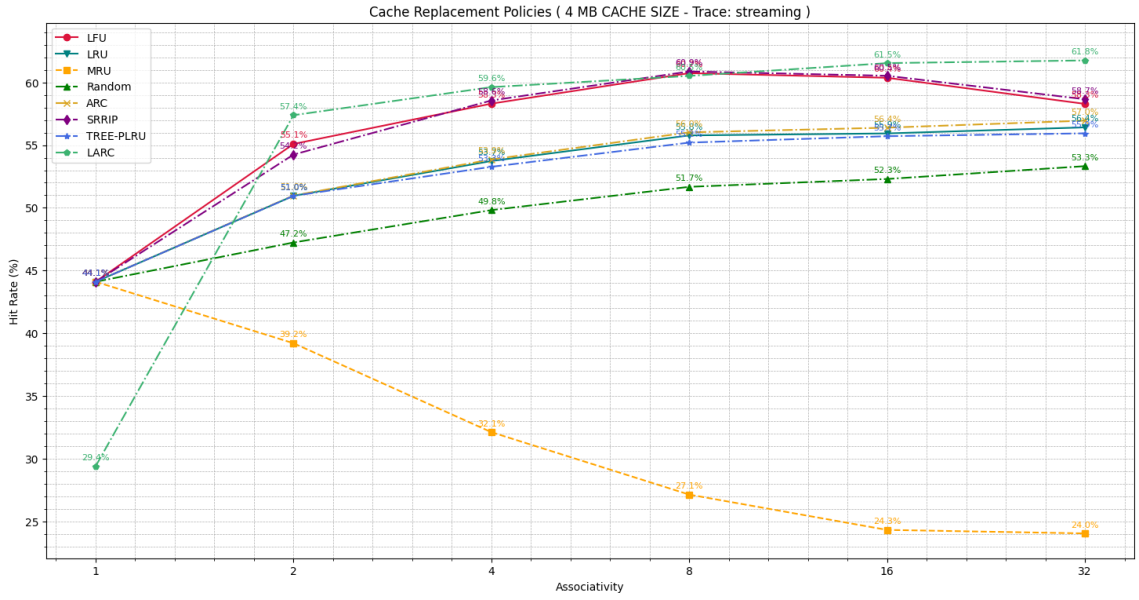
شکل ۱: 2MB-cache size:cassandra



شکل ۲: 4MB-cache size:cassandra



شکل ۳: 2MB-cache size:streaming



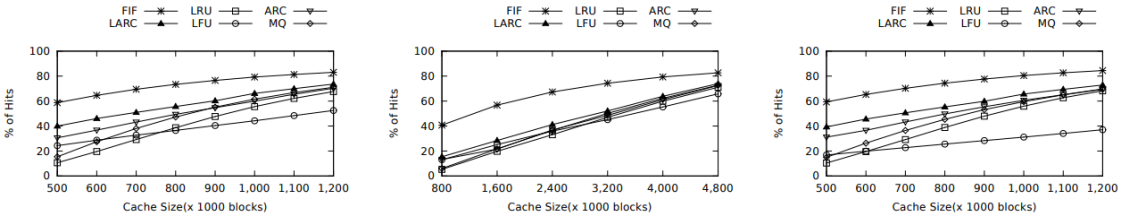
شکل ۴: 4MB-cache size:streaming

ابتدا رابطه نرخ برخورد هرمنحنی با تعداد راه را جداگانه بررسی می کنیم و بعد به رابطه آن با بقیه می پردازیم. در نظر داشته باشید تحلیل دقیق ریاضی تحدب و تفعر و اثبات روابط از حوصله این مقاله خارج است و تاکید ما در این مقاله بر الگوریتم، پیاده سازی و شبیه سازی سیاست ها و ارزیابی آن ها براساس نرخ برخورد است. بنابراین در تحلیل نتایج نیز ما به دنبال قسمت ریاضیاتی اثبات مکشوفات خود نیستیم و صرفا به پیدا کردن روابط و دلیل شهودی آن ها بر اساس آموخته های خود می پردازیم.

توجه کنید way در ۱ چندان معنایی در جایگزینی ندارد برای همین عموما آن را نادیده می گیریم.

همچنین توجه کنید ما اختلافات نرخ برخورد کم و قابل چشم پوشی را تحت "هم قدرت" یا "هم رده" تحلیل می کنیم، این لغات معمولا وقتی به کار می روند که رتبه بندی ها دائما در تست های مختلف عوض می شود ولی اختلاف همانطور که گفته شده کم است.
ابتدا با MRU شروع می کنیم. این منحنی در تمامی حالات محدب بود و کمترین مقدار را دارد. این نشان می دهد این الگوریتم هرچند در تئوری در برخی حالات که درباره کد اجرایی اطلاعاتی داریم بهترین بازدهی را دارد اما در حالت کلی شکست خورده است.
سیاست Random بستگی به نوع دستورات می تواند محدب یا مقعر باشد پس نتیجه گیری خاصی نمی توان کرد، اما با این وجود رتبه دوم از آخر را دارد و همچنان از MRU بالاتر است. این نشان می دهد اگر هیچ فرضی راجع به دستورات نداشته باشیم همجنان از فرض بیشترین استفاده بازدهی بهتری دارد. همجنین در اکثر مواقع به بهترین نرخ برخورد بین همه سیاست ها از بدترین حالت نزدیک تر است. این مورد درکنار سادگی پیاده سازی و هزینه سخت افزاری می تواند گزینه مطلوبی برای طراحی ها و معماری های خاصی باشد.
سیاست بعدی که بررسی می کنیم SRRIP است. این سیاست همواره مقعر است و در نسخه ۸ مسیره بیشینه می شود. به طرز جالبی با LFU همگرا می شود، اما ممکن است تقعرشان فرق کند که این نشان می دهد اگر تعداد مسیر ها کمی زیاد شود بازدهی این دو تقاوت کمی دارد. این سیاست تا قبل از way برابر ۸ رقیب جدی LARC است و از بقیه بالاتر است اما بعد از آن بازدهی چندانی ندارد و عملا در تعداد مسیر های محدودتر این روش برای تعیین امتیاز بهتر است. این نشان می دهد این نسخه که بر زمان استفاده تاکید دارد توانسته در مسیر های محدود از روش سنتی یعنی آخرین استفاده پیشین بگیرد. این توسعه مدیون مدیریت سن اعضا در طی زمان است.
حال که صحبت از LFU شد باید بگوییم هرچند همواره اکسترمم گلوبال دارد اما لزوما محدب یا مقعر نیست و این به نوع دستورات بستگی دارد. به دلیل تقاوت های در دو تریس مورد استفاده می توانیم بفهمیم می تواند از LRU بهتر باشد اما این به شدت به نوع استفاده بستگی دارد.
LRU به طور کلی صعودی است و هم قدرت .ARC نسخه اصلاحی این الگوریتم یعنی Tree-PLRU تقریبا هم قدرت است و به نظر می رسد صرفا برتری در هزینه دارد. اما به این حقیقت توجه کنید که به طرز جالبی در way برابر ۲ دقیقا همان نرخ برخورد را دارد که این بخاطر این است که وقتی ۲ مسیر داریم درخت ما با یک بیت نشان داده می شود، و فلش جهت درخت همواره به سمت MRU می رود و نتیجتا همواره نتیجه یکسانی دارند در این حالت.
تمامی الگوریتم های پیشین معمولا از منطق مشابهی پیروی می کردند منتها یا داده ساختار متفاوتی استفاده می کردند یا نوع نگرششان به داده ساختار یکسان متفاوت بود. الان می خواهیم به الگوریتم ARC و نسخه اصلاحی آن یعنی LARC را بررسی کنیم. این الگوریتم ها تفاوت این الگوریتم

های پیشین را درک می کنند و تصمیم دارند با ایجاد یک فرایند پویا، در هر حالت بهینه ترین الگوریتم را انتخاب کنند و در صورت اشتباه خودشان را اصلاح کنند.
ARC معمولا روند صعودی را دارد که یعنی با افزایش تعداد مسیر بازدهی اش بهتر می شود اما به طور کلی با LRU و Tree-PLRU هم قدرت است، به طور که اکثر مواقع بالاتر است. برعکس LRU که نسبتا در یک ردیف قدرت با نسخه اصلاحی خود داشت، نسخه اصلاحی ARC به طرز چشمگیری از همه سیاست ها من جمله خود ARC بالاتر است و تابع صعودی دارد. این مورد را می توان در این تفاوت جستجو کرد که عمده تفاوت LRU و TPLRU در داده ساختار مورد استفاده است اما در ARC و LARC تفاوت مهم تری داریم. این تفاوت در مسئله مواجه با عدم برخورد است و نشان می دهد به طرز جالبی با هرنوع پارامتر گذاری در LARC همواره نتایج از همه سیاست ها بهتر است. این نشان می دهد که ایده ترکیب دو سیاست LRU و LFU پتانسیل زیادی برای بهبود دارد. نتایجی که ما به دست آوردیم با دانش تئوری ما و مقاله های مرتب مانند LARC و ARC همخوانی دارد، و دانش ما در عمل تایید شد. در تصویر زیر می توانید نتایج خود مقاله LARC را مشاهده کنید. همانطور که مشخص است نتایج ما نیز این بر نتایج این مقالات صحه می گذارد. طبیعتا در مقادیر تفاوت داریم به دلیل تریس های متفاوت اما در همگی LARC برتری خود را ثابت کرده.



(a) *websearch* (b) *ads* (c) *ws_con*

Fig. 4. Hit rate of different algorithms under read-dominate traces

حال با نتیجه گیری نهایی این قسمت را پایان می دهیم. شایان ذکر است که در این تحلیل و نتیجه گیری way برابر یک مورد توجه دقیقی قرار نگرفته، اما خوب است اشاره کنیم همواره بین همه نرخ یکسان است و دلیل یکسان نبودنLARC تفاوت در سیاست نوشتن است.

سیاست MRU بدترین سیاست در حالت کلی است. Random می تواند در شرایطی که بحث هزینه و سرعت مطرح است بهترین باشد. سیاست های LRU و Tree-PLRU تفاوت کمتری نسبت به ARC و LARC دارند. LARC بهترین سیاست در همه شرایط است. باقی سیایت ها همگرا می شوند، و معمولا هم رده اند و در تریس های مختلف ممکن است عملکرد درخشان تری داشته باشند.

۶ پیچیدگی‌های پیاده‌سازی

بهترین طراحی، ساده‌ترین راه‌حلی است که کار می‌کند.

— آلبرت اینشتین

در تمام این مقاله، هدف ما بررسی سیاست های مختلف و ارزیابی آن ها بر اساس نرخ برخورد بوده است. در این بخش می خواهیم مختصر به هزینه های پیاده سازی بپردازیم.
از آنجا این موضوع هدف این مقاله نیست ما وارد جزئیات دقیق و ریاضیاتی آن ها نمی شویم بلکه به طور نسبی هزینه هارا بررسی می کنیم.
هزینه رندوم از همگی سیاست ها کمتر است. این به خاطر سادگی طراحی سخت افزاری است که عدد تصادفی تولید کند. پیچیدگی سخت افزارMRU و LRU از یک مرتبه است و به لحاظ هزینه معمولا یک لیست برای نگهداری ترتیب نیاز داریم. مثلا برای نگهداری ترتیب در یک مجموعه n بیتی می توانیم $logn$ بیت برای ترتیب هر بلوک اختصاص دهیم که در می شود $nlogn$ اما در سیاست Tree-PLRU به دلیل استفاده از ساختار درختی هزینه نگهداری ما کاهش پیدا می کند.
SRRIP نیز معمولا با استفاده از شمارنده ها پیاده سازی می شود و از نظر اردر فضا در اردر LRU می شود(جدای هزینه خود الگوریتم در این درس نمی گنجد، اما به طور کلی باید از LRU بیشتر باشد). هزینه LFU نیز بستکی به طراحی دارد اما به طور کلی باز در اردر LRU است، این را در پیاده سازی ما هم می توانید ببینید که ساختار مشابهی دارند.
ARC و LARC به طور کلی هزینه پیاده سازی بیشتری دارند این موضوع در فایل های ما نیز مشهودند، خصوصا که نیاز به ۴ صف دارند و باید اندیس آن هایی که حذف شده اند را هم به نوعی نگهدارند. پیچیدگی پیاده سازی این دو از همه بیشتر است و به سخت افزار بیشتری نیز نیاز دارد. به دلیل همین پیچیدگی می بایست به صورت نرم افزاری پیاده شوند و معمولا برای هارد دیسک مورد استفاده قرار می گیرند.

References

[1] Wikipedia contributors, "Adaptive replacement cache — wikipedia, the free encyclopedia," 2024. Accessed: 2025-07-19.

[2] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, USENIX Association, 2003.

[3] S. Jiang and X. Zhang, "Making lru friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance," *IEEE Transactions on Computers*, vol.62, no.12, pp.2344–2357, 2013.

[4] M. Hashemi, O. Mutlu, and B. Falsafi, "Accelerating dependent cache misses with an enhanced memory controller," *IEEE Micro*, vol.36, no.4, pp.60–71, 2016.