دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

پروژه معماری کامپیوتر

عنوان:

پیاده‌سازی TAGE BRANCH PREDICTOR در شبیه‌ساز ChampSim

نگارش

عرفان تیموری ۴۰۲۱۰۵۸۱۳

پوریا رحمانی ۴۰۲۱۱۱۴۱۸

نیما قدیرنیا ۴۰۲۱۱۱۳۲۳

محمدرضا منعمیان ۴۰۲۱۰۶۶۰۴

تیر ماه ۱۴۰۴

فهرست مطالب

۱ مقدمه ۲

۲ پیاده‌سازی ۲

۲-۱ فایل tage-config.h ۳

۲-۲ فایل tage.h ۳

۲-۲-۱ hyper-parameter ها ۳

۲-۲-۲ struct های کمکی ۵

۲-۲-۳ سایر متغیرها و آرایه‌ها ۵

۲-۲-۴ constructor کلاس tage ۷

۲-۳ فایل tage.cc ۸

۲-۳-۱ تابع predict_branch ۸

۲-۳-۲ تابع last_branch_result ۹

۳ تست ۱۲

۳-۱ نحوه اجرای پیاده‌سازی در ChampSim ۱۲

۳-۲ ویژگی‌ها و دلایل انتخاب تریس‌ها ۱۳

۳-۳ پارامترهای قابل تنظیم در پیاده‌سازی ۱۶

۳-۴ روش بهینه‌سازی پارامترها و تست‌گیری ۱۷

۳-۴-۱ بهینه کردن هر پارامتر به صورت مجزا ۱۸

۳-۴-۲ به دست آوردن توزیع احتمالی برای هر پارمتر ۱۹

۳-۴-۳ تغییر همزمان پارمتر ها به صورت رندوم ۲۰

۳-۵ نتایج به‌دست‌آمده از تنظیمات بهینه برای TAGE ۲۰

۳-۵-۱ نتایج تغییر مجزای هر پارامتر ۲۰

۳-۵-۲ نتایج تغییرات همزمان پارامترها ۲۲

۳-۵-۳ تست با base های مختلف ۲۵
۳-۶ مقایسه عملکرد TAGE با سایر branch predictorها ۲۷

۱ مقدمه

پیش‌بینی شاخه یکی از اجزای حیاتی در طراحی پردازنده‌های مدرن محسوب می‌شود که نقش بسزایی در بهبود کارایی اجرای دستورالعمل‌ها و کاهش وقفه‌های ناشی از انشعاب دارد. در میان پیش‌بینی‌کننده‌های شاخه، الگوریتم TAGE (TAgged GEometric predictor) به دلیل دقت بالایی که دارد بسیار موثر واقع می‌شود.
در این پروژه ، یک نمونه ساده از این branch predictor پیاده‌سازی شد و دقت آن به‌ازای پارامترهای متفاوت مورد بررسی قرار گرفت تا پارامترهای مناسبی برای آن انتخاب شوند، بعلاوه عملکرد آن نیز در مقایسه با سایر predictor ها مورد سنجش قرار گرفت که توضیح جزئیات مراحل اجرای پروژه را در بخش‌های مختلف این گزارش نوشته‌ایم. فایل‌های مربوط به پیاده‌سازی و تست نیز به پیوست ارسال شده‌اند

۲ پیاده‌سازی

branch predictor ها در ChampSim ،در پوشه‌ای به نام branch هستند ، توی پوشه branch ، برای هر branch predictor یک پوشه به نام همان predictor وجود دارد که درون آن پوشه یک فایل header و یک کد cpp وجود دارد که در آن‌ها یک کلاس (یا struct) به نام آن branch predictor وجود دارد که از کلاس branch_predictor که در champsim به طور پیشفرض تعریف شده ارث‌بری می‌کنند.
بنابراین، ما هم درون پوشه‌ی branch ، یک پوشه به‌نام tage ساختیم و دو فایل tage.h و tage.cc را ایجاد کردیم(بعلاوه یک فایل tage_config.h که توضیح هر سه را در بخش پیاده‌سازی می‌بینید) و یک کلاس tage نیز در فایل header تعریف کردیم.
تمام predictor ها با صدا زدن دو تابع predict_branch که pc را می‌گیرد و یک boolean به عنوان پیش‌بینی predictor برمی‌گرداند و تابع last_branch_result که وظیفه update ساختارهای موجود در predictor را دارد کار می‌کنند. بنابراین ، وظیفه اصلی ما هم پیاده‌سازی همین دو تابع است و بقیه کارها در راستای آن انجام می‌شوند.

۲-۱ فایل tage-config.h

برای راحتی و سرعت script هایی که برای اجرای تست‌های مختلف با پارامترهای مختلف اجرا کردیم و با آن‌ها در بخش تست آشنا می‌شوید ، پارامترهایی که قرار بود مقدارشان را در تست‌ها تغییر دهیم دوباره در این فایل هم تعریف کردیم و در فایل tag.h که همه hyper-parameter ها در آن تعریف شده‌اند ، این دسته از hyper-parameter ها را با مقدار همین hyper-parameter ها در فایل tage_config.h مقداردهی کردیم.(برای این‌کار، در ابتدای tage.h ، config_tage.h include شده‌است تا بتوانیم از مقادیر موجود در آن برای پارامترهایی که در تست‌ها تغییر می‌کنند استفاده کنیم)

تعریف و کاربرد همه متغیرها را می‌توانید در بخش توضیح فایل tage.h ببینید و لیست همه hyper-parameter هایی که در تست‌ها آن‌ها را تغییر دادیم را می‌توانید در توضیحات بخش تست مشاهده کنید.

enum ای به نام TableSizePattern نیز در این فایل تعریف شده که مشخص می‌کند سایز هر جدول برحسب طول تاریخچه‌ای که آن جدول استفاده می‌کند از چه الگویی پیروی می‌کند.(با افزایش طول تاریخچه ، سایز جدول نیز ثابت بماند (CONSTANT) ، افزایش‌یابد(ASCENDING) ، کاهش‌یابد(DESCANDING) یا تا نیمه افزایش و سپس کاهش یابد (HILL-SHAPED))

۲-۲ فایل tage.h

۲-۲-۱ hyper-parameter ها

- NUMBER_OF_TABLES :
  تعداد جدول‌های tage (به جز جدول base_predictor) را مشخص می‌کند.
- LEAST_History_LENGTH و COMMON_RATIO :
  می‌دانیم در tage ، طول‌تاریخچه‌های جداول از یک دنباله‌ی هندسی پیروی می‌کنند.
  LEAST_HISTORY_LENGTH کمترین عضو این دنباله هندسی است و COM-MON_RATIO قدر نسبت این دنباله هندسی است.
- U_MAX و U_MIN :
  می‌دانیم ارزش پیش‌بینی‌ها در هر خانه از جداول tage با یک عدد دوبیتی موسوم به u سنجیده

می‌شود. از آنجا که داده‌ساختار دو بیتی در cpp به‌طور پیشفرض نداریم، برای راحتی کار ماکزیمم و مینیمم u را برابر ۳ و ۰ تعریف می‌کنیم تا در افزایش و کاهش u از مرزها عبور نکنیم.

- PRED_MAX و PRED_MIN :

به دلیل مشابه بالا، ماکزیمم و مینیمم برای عدد pred که نشان‌دهنده پیش‌بینی هر خانه از جداول tage است را به‌ترتیب برابر ۷ و ۰ تعریف می‌کنیم (چون pred یک عدد سه‌بیتی است)

- ALT_BETTER_THAN_PRED_MAX و ALT_BETTER_THAN_PRED_MIN :

alt_better_than_pred یک عدد چهار بیتی است که مشخص می‌کند چقدر پیش‌بینی ها با جدول alternative از پیش‌بینی‌ها با جدول provider بهتر بوده‌اند. اگر این عدد بیشتر یا مساوی ۸ بود و شرایط دیگری هم که در بخش tage.cc توضیح داده می‌شود برقرار بود، برای پیش‌بینی از جدول alternative استفاده می‌شود.

به‌دلیل مشابه بالا ، ماکزیمم و مینیمم آن را به‌ترتیب ۱۵ و ۰ در این دو متغیر ذخیره می‌کنیم

- TAG_SIZE_MAX و TAG_SIZE_MIN :

از این دو متغیر برای تنظیم سایز tag خانه‌های جداول استفاده کردیم. طبق پیشنهاد هوش مصنوعی ، جداول با طول history کمتر، از طول tag بیشتری استفاده می‌کنند تا احتمال collision در آنها کاهش یابد.

- MAX_TABLE_SIZE_LOG و MIN_TABLE_SIZE_LOG :

از این دو پارامتر هم برای تنظیم ابعاد جداول استفاده کردیم.(لگاریتم بیشترین و کمترین ابعاد جدول در مبنای دو در این دو متغیر ذخیره می‌شود)

- GHR_SIZE :

طول کل history ای که ذخیره می‌کنیم. این عدد برابر ۵۰۰۰ قرار داده شده‌است تا مطمئن باشیم از بیشترین طول تاریخچه‌ای که از آن استفاده می‌کنیم بیشتر است.

- RESET_RATE :

بیت‌های MSB و LSB عدد u همه‌ی خانه‌های جداول باید به‌صورت دوره‌ای reset شوند تا یک پیش‌بینی بازهم ارزش خود را ثابت کند! این دوره ، به پیشنهاد مقاله‌ی اصلی، برابر ۲۵۶۰۰۰ قرار داده‌شد.

- ACTIVE_PATTERN :

یک عضو از enum تعریف شده در فایل tage_config.h (به نام Table_Size_Pattern) است که الگوی تغییر سایز جداول برحسب طول تاریخچه استفاده در جدول را مشخص می‌کند

۲-۲-۲ struct های کمکی

- cell :

هر خانه از جداول tage ، یک instance از این struct است. این struct شامل یک عدد u برای ارزش پیش‌بینی ، یک عدد pred برای خود پیش‌بینی و یک عدد tag است. همچنین این struct یک default constructor دارد که عدد pred را برابر ۴ (taken ضعیف) و دو عدد دیگر را برابر صفر مقداردهی اولیه می‌کند.

- circular_fold : برای محاسبه index هایی که یک دستور با pc مشخص باید در جداول به آن‌ها مراجعه کند و همچنین tag هایی که باید برای بررسی tag_hit مورد استفاده قرار بگیرند، باید pc و طول تاریخچه مشخص را با هم hash کنیم.

برای این‌کار ، با توجه به تعداد بیتی از تاریخچه که هر جدول باید استفاده کند، بیت‌های آخر تاریخچه را درنظر می‌گیریم و با ترکیب بیت‌های آن ، آن را به یک عدد از طول index یا tag (که توسط table_sizes_log و tag_sizes تعیین می‌شوند) در می‌آوریم.

برای اینکه هربار از ابتدا این اعداد تبدیل‌شده را محاسبه نکنیم، این struct را برای ذخیره‌ی آن‌ها و update مرحله به مرحله آن‌ها می‌سازیم.

این struct سه عدد history_length ، target_size و folded_value دارد که به‌ترتیب طول تاریخچه‌ی مورد استفاده، طول مورد انتظار بعداز تبدیل و نتیجه تبدیل را در خود دارند.

۲-۲-۳ سایر متغیرها و آرایه‌ها

- alt_better_than_pred_count :

همان متغیر چهاربیتی است که مشخص می‌کند چقدر پیش‌بینی‌های جداول alternative از provider ها بهتر بوده‌است (توضیح آن در بخش hyper-parameter ها آمد) که با صفر مقداردهی اولیه شده‌است.

- reset_counter و is_MSB :

reset_counter دوره تناوبی که u ها reset می‌شوند را می‌شمارد و وقتی به ۲۵۶۰۰۰ رسید دوباره صفر می‌شوند. از آنجا که در هر دوره، به صورت یکی‌درمیان، باید MSB ها و LSB های u صفر شوند، عدد is_MSB مشخص می‌کند که نوبت reset شدن MSB اعداد u است یا نوبت LSB ها.

- داده‌های آخرین prediction :

از آنجا که طبق بررسی‌ها، هر update (call شدن تابع last_branc_result) دقیقا بعد از call شدن تابع predict_branch مربوطه اتفاق می‌افتد و از آنجا که به برخی‌داده‌ها که در روند انجام prediction بدست آوردیم، از جمله شماره جداول provider و ،alternative اندیس‌هایی از این جداول که برای انجام پیش‌بینی به آن‌ها رجوع کردیم ، پیش‌بینی‌هایی که هرکدام از provider و alternative دارند و پیش‌بینی نهایی خود نیاز داریم ، آن‌ها را به‌عنوان داده‌هایی در کلاس نگه می‌داریم و درون تابع predict_branch آن‌ها را update می‌کنیم.

- پیش‌بینی‌کننده پایه :

فایل‌های bimodal.h ، gshare.h، perceptron.h و hashed_perceptron.h در فایل include ، tage.h شده‌اند تا instance ای از یکی از این پیش‌بینی کننده‌ها به عنوان پیش‌بینی‌کننده پایه داشته باشیم.(constructor این پیش‌بینی کننده‌ها که در واقع همان constructor کلاس branch_predictor است در constructor کلاس tage صدا زده می‌شود تا in-stance ساخته شود.)

توجه کنید از آنجا که در champsim پیش‌بینی‌کننده‌هایی مثل bimodal خودشان یک جدول دارند، از جدول base_predictor ها استفاده نکردیم و از یک instance از base_predictor استفاده کردیم.

- allocation_rng : برای تولید عدد رندوم برای allocation خانه از یک جدول تصادفی وقتی allocation از چندخانه قابل انجام است استفاده می‌شود.

- آرایه‌ها برای size ها:

آرایه‌های history_lengths ، tag_sizes و table_sizes_log به‌ترتیب برای نگه‌داری طول‌تاریخچه، سایز تگ استفاده‌شده در خانه‌ها و لگاریتم مبنای دو ابعاد جداول استفاده می‌شوند.

- آرایه‌ها برای ذخیره fold ها :

آرایه‌های index_folds ، tag_folds1 و tag_folds2 ، آرایه‌هایی از circular_fold ها هستند و همانطور که در توضیح circular_fold اشاره‌شد، برای ذخیره اعداد folded استفاده‌شده

در محاسبه hash برای index و tag به‌کار می‌روند و هر بار در تابع last_branch_result طبق توضیحی که در بخش منطق محاسبه hash می‌دهیم بروزرسانی می‌شوند.

- آرایه‌ها برای ذخیره tag ها و index ها :

از آنجا که در طول یک روند prediction و update ، مقدار pc و همینطور بیت‌های -his tory ثابت می‌مانند، index هایی از جداول که باید به آن‌ها مراجعه کنیم و tag دستور کنونی برای آن جداول که hash ای از history و pc هستند، ثابت می‌مانند. بنابراین، به‌جای هربار محاسبه کردن آن‌ها، یکبار در predict_branch آن‌ها را محاسبه و در آرایه‌های tags و indexes ذخیره می‌کنیم.

- global_history :

یک bitset است که ۵۰۰۰ بیت انتهایی تاریخچه را نگه می‌دارد و در ابتدا همه بیت‌های آن با صفر مقداردهی اولیه شده‌اند.

- tage_tables :

vector ای از vector هایی از cell هاست که همه جداول tage را در خود دارد.

۴-۲-۲ constructor کلاس tage

این constructor ، constructor های predictor پایه و کلاس branch_predictor که از آن ارث‌بری می‌کند را صدا می‌زند و بعلاوه برای مقداردهی به آرایه‌های سایزها و پارامترهای -his tory_length و target_size برای circular_fold های توی index_folds و tag_folds1 و tag_folds2 استفاده می‌شود.

- برای مقداردهی به آرایه history_lengths ، اولین history_length را برابر LEAST_HISTORY_LENGTH قرار می‌دهیم و هر history_length را برابر گردشده‌ی حاصلضرب history_length قبلی در COMMON_RATIO قرار می‌دهیم تا -his tory_length استفاده‌شده در جداول، از دنباله‌ای هندسی پیروی کند.

- برای مقداردهی به آرایه tag_sizes ، از TAG_SIZE_MAX برای جدول اول (با کوتاهترین طول تاریخچه مورد استفاده) استفاده می‌کنیم و به‌طور خطی طول tag را تا TAG_SIZE_MIN

کاهش می‌دهیم تا جدول با بلندترین طول تاریخچه ، کوتاهترین طول tag مورد استفاده را داشته باشد.

- برای مقداردهی به آرایه table_sizes_log به ACTIVE_PATTERN توجه می‌کنیم ، اگر CONSTANT بود، از MIN_TABLE_SIZE_LOG برای همه جداول استفاده می‌کنیم، اگر ASCENDING بود ، اعضای این آرایه را از MIN_TABLE_SIZE_LOG تا MAX_TABLE_SIZE_LOG به‌طور خطی افزایش می‌دهیم تا جدولی که از بلندترین طوبل تاریخچه استفاده می کند، بیشترین ابعاد را هم داشته باشد، برعکس، اگر DESCANDING بود ، ابعاد را به‌طور خطی کاهش می‌دهیم و اگر HILL_SHAPED بود ، ابعاد را تا نیمه به‌طور خطی افزایش و سپس به‌طور خطی کاهش می‌دهیم.

- حال، با استفاده از resize ، اندازه‌های جداول را مطابق table_sizes تعیین می‌کنیم و سپس history_length های آرایه‌های مخصوص fold را با توجه به آرایه history_lengths ، target_size های index_folds را با توجه به table_sizes_log و target_size های tag_folds1 و tag_folds2 را با توجه به tag_sizes مقداردهی می‌کنیم:

```
1     for(size_t i = 0; i < NUMBER_OF_TABLES; i++) {
2      index_folds[i].history_length = tag_folds1[i].history_length =
           tag_folds2[i].history_length = history_lengths[i];
3      index_folds[i].target_size = table_sizes_log[i];
4      tag_folds1[i].target_size = tag_sizes[i];
5      tag_folds2[i].target_size = tag_sizes[i] - 1;
6     }
```

۲-۳ فایل tage.cc

۲-۳-۱ تابع predict_branch

هدف این تابع به وضوح predict کردن branch می‌باشد. برای این کار ابتدا با استفاده از pc برای همه جداول،index و tag مربوط به آن را به دست می‌آوریم. دقت کنید که محاسبه fold ها به صورت داینامیک صورت می‌گیرد که آن را در ادامه توضیح می‌دهیم و در این تابع تنها از مقدار آن‌ها استفاده می‌شود. در ادامه با شروع از جدول با history length طولانی‌تر یکی یکی جدول‌ها را بررسی می‌کنیم و در صورتی که tag محاسبه شده توسط ما با tag آن خانه index از آن جدول برابر باشد پیش‌بینی آن جدول را معتبر می‌نامیم. اولین جدولی که به این صورت پیدا کنیم را به عنوان

provider و دومین را به عنوان alternative در نظر می‌گیریم. در صورتی که دو جدول پیدا شود دیگر باقی جداول را بررسی نمی‌کنیم. حال برای پیش‌بینی این‌گونه عمل می‌کنیم:
در صورتی که هیچ جدولی، یعنی نه provider و نه ،alternative پیدا نشده بود از predictor base استفاده می‌کنیم. در صورتی که فقط provider داشتیم نیز به وضوح از آن استفاده می‌کنیم. اما در صورتی که هم provider داشتیم و هم alternative ، بر اساس شرایط تصمیم می‌گیریم که کدام را به عنوان پیش‌بینی در نظر بگیریم. در صورتی که ارزش پیش‌بینی provider برابر با صفر باشد و همچنین alt_better_than_pred_count نیز از ۸ بیشتر باشد و همچنین pred مربوط به پیش‌بینی provider برابر با ۳ یا ۴ باشد، یعنی ضعیف باشد، آنگاه از پیش‌بینی alternative استفاده می‌کنیم در غیر این صورت از provider استفاده خواهیم کرد. این شرایط گفته شده با نام is_newly_allocated در کد آمده است.
در نهایت با طی کردن این فرآیند پیش‌بینی branch به اتمام می‌رسد.

۲-۳-۲ تابع last_branch_result

ابتدا دقت کنید که در صورتی که provider نداشته باشیم از الگوریتم آپدیت base_predictor خواهیم کرد. حال اگر provider داشته باشیم آپدیت ما بدین‌گونه خواهد بود:

- آپدیت مقدار ارزش u

  این مقدار تنها در صورتی آپدیت می‌شود که provider و alternative هر دو موجود باشند و پیش‌بینی آن‌ها متفاوت باشد. در آن صورت با توجه به اینکه کدام پیش‌بینی درست بوده ارزش خانه provider تغییر می‌کند. ابته دقت کنید که این ارزش نمی‌تواند از بازه کمینه یا بیشینه آن خارج شود.

- آپدیت pred

  آپدیت این مقدار نیز واضح است، در صورتی که branch ما taken باشد مقدار pred یکی زیاد می‌شود و در صورتی که taken not باشد یکی کم خواهد شد. البته دقت کنید که از مرز کمینه و بیشینه تجاوز نخواهیم کرد.

- آپدیت شمارنده چهار بیتی مقایسه کننده عملکرد provider و alternative

  این شمارنده تنها در صورتی آپدیت می‌شود که alternative داشته باشیم و پیش‌بینی alter-native و provider متفاوت باشد، در واقع آپدیت این شمانده بسیار شبیه به آپدیت u است. در صورتی که alternative درست و provider غلط پیش‌بینی کند آنگاه شمارنده ما یکی

زیاد شده و در صورتی که provider درست و alternative غلط پیش‌بینی کند شمارنده ما یکی کم می‌شود. البته باز هم دقت کنید که از مرز کمینه و بیشینه تجاوز نخواهیم کرد.

- منطق و شیوه پیاده‌سازی allocation خانه‌های جدید

دقت کنید که allocation زمانی انجام می‌شود که پیش‌بینی ما غلط باشد. اگر پیش‌بینی ما غلط باشد، تلاش می‌کنیم که یک entry در جدول‌هایی که طول history length طولانی‌تری نسبت به provider ما دارند allocate کنیم. برای این کار از جدول با شماره $provider + ۱$ شروع کرده و به ترتیب جدول‌ها را بررسی می‌کنیم. برای هر جدول ارزش (یا همان u) خانه با index مربوط به آن جدول را بررسی می‌کنیم، اگر برابر با صفر بود آن را در لیستی قرار می‌دهیم. همچنین index مربوط به خانه‌ای از جدول که بررسی کردیم را نیز ذخیره می‌کنیم تا بعدا بدانیم چه entry ای باید تغییر کند. این کار را برای همه جداول انجام می‌دهیم.
حال اگر لیست ما خالی نبود، یکی از جداول را به صورت تصادفی انتخاب می‌کنیم، به طوری که احتمال انتخاب جدول با طول history length کمتر، بیشتر باشد. پس از اینکه یک جدول را انتخاب کرده، خانه‌ای که با index مربوط به آن نشان داده می‌شود را تغییر می‌دهیم. به این صورت که pred آن را برابر با ۴ قرار داده و tag آن را نیز برابر با مقدار محاسبه شده برای هر جدول (همان که به وسیله fold ها و hash کردن به دست می‌آمد) قرار می‌دهیم.
اگر لیست ما خالی بود یعنی ارزش همه خانه‌های موردنظر از جداول بررسی شده ناصفر بودند، در این حالت نمی‌توانیم allocate انجام دهیم اما برای جلگیری از تکرار بیش از حد این افاق ارزش یا همان u تمام این خانه‌های بررسی شده را یکی کاهش می‌دهیم تا به صفر نزدیک‌تر شوند.
بدین ترتیب فرآیند allocate کردن نیز به پایان می‌رسد.

- reset کردن u و آپدیت history global

برای reset کردن u از یک reset_counter استفاده می‌کنیم که هر بار که تابع را صدا میزنیم یکی زیاد می‌شود و در صورتی که به Reset_Rate برسد برابر با صفر شده و u همه خانه‌های همه جداول را تغییر می‌دهد. تغییر دادن آن نیز این‌گونه است که یک بار بیت کم‌ارزش u را برابر با صفر کرده و دفعه بعدی بیت پرارزش آن را و به همین شکل به طور یکی در میان ادامه می‌دهد. آپدیت شدن GHR نیز اینگونه است که بعد از صدا زدن تابع آپدیت نتیجه branch را به سمت راست global history که یک bitset است الصاق می‌کند. البته دقت کنید که چون طول bitset ثابت است پس بیت سمت چپ آن از دست می‌رود.

- منطق و شیوه پیاده‌سازی hash و محاسبه tag ها و index :

برای توضیح منطق hash ابتدا باید منطق fold کردن یک عدد از سایز history_length به سایز target_size را توضیح دهیم.
حاصل fold کردن عدد x که history_length بیتی است به target_size بیت ، عدد y خواهد بود که برای هر $0 \leqslant i \leqslant$ target_size - 1 ، بیت i ام آن ، برابر حاصل xor بیت‌هایی از x است که باقیمانده جایگاهشان به target_size برابر i است. مثلا اگر $x = \overline{a_4 a_3 a_2 a_1 a_0}$ و target_size برابر 3 باشد ، خواهیم داشت :

$$y = \overline{(a_2 \oplus a_5)\ (a_1 \oplus a_4)\ (a_0)}$$

برای این‌که هر بار ساختن y از روی x را از ابتدا انجام ندهیم که overhead زیادی داشته‌باشد، آن را به‌صورت dynamic ساخته و هربار update می‌کنیم.
به‌راحتی می‌توانید مشاهده‌کنید که منطق کد زیر، همواره با update شدن تاریخچه global ، همین مقدار folded را بدست می‌دهد.(کد زیر را برای tag_folds_1 و tag_folds_2 تکرار کرده‌ایم تا حتی از overhead صدا زدن تابعی برای این update هم جلوگیری کنیم!)

```
index_folds[i].folded_value = (index_folds[i].folded_value << 1) + global_history[0];
index_folds[i].folded_value ^= ((index_folds[i].folded_value & (1 <<
    index_folds[i].target_size)) >> index_folds[i].target_size);
index_folds[i].folded_value ^= (global_history[index_folds[i].history_length] <<
    (index_folds[i].history_length % index_folds[i].target_size));
index_folds[i].folded_value &= ((1 << index_folds[i].target_size) -1);
```

حالا برای محاسبه hash ها ، از xor مقادیر folded با pc و شیفت‌های آن استفاده می‌کنیم. بطوریکه :

```
for(size_t i = 0; i < NUMBER_OF_TABLES; i ++) {
  indexes[i] = static_cast<uint32_t>((pc ^ index_folds[i].folded_value) & ((1<<
      table_sizes_log[i]) -1 ));
  tags[i] = static_cast<uint32_t>(((pc >> 2)^ tag_folds1[i].folded_value ^
      (tag_folds2[i].folded_value << 1)) & ((1 << tag_sizes[i]) - 1));
}
```

۳ تست

۳-۱ نحوه اجرای پیاده‌سازی در ChampSim

برای اجرای فایل‌های پیاده‌سازی‌شده در ChampSim، ابتدا باید فایل champsim_config.json را باز کرده و مطابق تصویر زیر، مقدار branch predictor را برابر با tage قرار دهیم. همچنین، پارامترهای موردنظر باید در فایل tage_config.h تنظیم شوند.

شکل ۱: تنظیم مقدار branch predictor در فایل champsim_config.json

سپس، با رفتن به پوشه‌ای که فایل json در آن قرار دارد، می‌بایست دستورات زیر به‌ترتیب اجرا می‌شوند:

۱. ./config.sh champsim_config.json

۲. make

۳. bin/champsim –warmup-instructions NUM_OF_WARMUP_INST –simulation-instructions NUM_OF_SIMULATION_INST TRACE_NAME

دستورات اول و دوم باعث می‌شوند که پارامترهای موردنظر و پیش‌بینی‌کننده‌ی TAGE پیکربندی و آماده شوند. دستور سوم نیز اجرای شبیه‌سازی را روی یک trace مشخص و به اندازه‌ی مشخصی از دستورات آغاز می‌کند. مقداردهی به پارامترهای WARMUP_INST و SIMULATION_INST

به ترتیب تعیین می‌کند چه تعداد دستور برای فاز آماده‌سازی و چه تعداد برای فاز اصلی شبیه‌سازی استفاده شود.

با توجه به اینکه تعداد پارامترهای قابل تنظیم (و در نتیجه تعداد حالت‌های تست) بسیار زیاد است، اجرای دستی تمامی تست‌ها کاری بسیار زمان‌بر خواهد بود. به همین دلیل، یک اسکریپت پایتون به نام optimize_tage.py طراحی شده که فرآیند تست را به‌صورت خودکار انجام می‌دهد. نحوه عملکرد این اسکریپت به شرح زیر است:

۱. یک حلقه روی هر یک از پارامترها به‌صورت جداگانه اجرا شده و در هر مرحله تنها مقدار یکی از آن‌ها تغییر داده می‌شود. این تغییرات مستقیماً در فایل tage_config.h اعمال می‌شوند.

۲. سپس، سه دستور بالا (پیکربندی، ساخت، و اجرای شبیه‌سازی) به‌ترتیب اجرا می‌گردند.

۳. خروجی حاصل از هر تست در قالب سه فایل متنی ذخیره می‌شود:

- فایل output: شامل خروجی استاندارد دستور اجرا (شبیه‌سازی)
- فایل error: شامل پیام‌های خطا در صورت وقوع
- فایل params: شامل پارامترهایی که در آن اجرای خاص استفاده شده‌اند

با توجه به تعداد پارامترها و مقادیر ممکن برای هر یک، حدوداً ۹۰ تست مختلف اجرا شده و خروجی آن‌ها در پوشه‌های مشخص ذخیره می‌گردد.

توجه شود که در ابتدای اسکریپت، لازم است تریس ورودی مشخص شود. بنابراین برای گرفتن خروجی روی هر سه تریس موردنظر، اسکریپت باید سه بار اجرا شود تا نتایج به‌دست‌آمده برای تمامی تریس‌ها استخراج گردد.

۳-۲ ویژگی‌ها و دلایل انتخاب تریس‌ها

در این بخش، تریس‌های مورد استفاده در فرآیند تست معرفی می‌شوند. این تریس‌ها از لحاظ حجم، پیچیدگی رفتاری و تنوع الگوهای پرش متفاوت هستند تا بتوانند ارزیابی جامعی از عملکرد branch predictorها ارائه دهند. همچنین درصد نسبی تعداد دستورات branch در هر تریس با استفاده از فایل خروجی اجرای همان تریس محاسبه می‌شود تا نشان داده شود که این تریس‌ها تا چه میزان روی مکانیزم پیش‌بینی پرش فشار وارد می‌کنند.

برای محاسبه تعداد تخمینی دستورات branch از فرمول زیر استفاده می‌کنیم:

$$Branch\ Inst = \frac{MPKI \times Total\ Inst}{۱۰۰۰} \times \frac{۱۰۰}{Branch\ Prediction\ Accuracy} \quad (۱)$$

که در آن:

- MPKI: تعداد Misses Per Kilo Instructions
- Total Inst: تعداد کل دستورات اجراشده در تریس
- Branch Prediction Accuracy: درصد دقت پیش‌بینی پرش

در نهایت، درصد تقریبی دستورات branch در هر تریس از رابطه زیر به‌دست می‌آید:

$$Branch\ Percentage = \frac{Branch\ Inst}{Total\ Inst} \times ۱۰۰ \quad (۲)$$

- تریس perlbench_135B.trace: این تریس مربوط به برنامه perlbench با حجم ۱۳۵ میلیارد دستور است. این نسخه‌ی حجیم‌تر نسبت به سایر تریس‌های perlbench رفتاری پیچیده‌تر و متنوع‌تر در پرش‌ها دارد و برای سنجش پایداری پیش‌بینی‌کننده‌ها در بارهای طولانی مناسب است. در یکی از فایل‌های خروجی مربوط به این تریس خط زیر را مشاهده می‌کنیم:

CPU 0 Branch Prediction Accuracy: 94.85%

MPKI: 11.44

Average ROB Occupancy at Mispredict: 43.44

با فرض اجرای ۴۰ میلیون دستور (Total Instructions = 40M)، ابتدا تعداد پرش‌ها را به صورت زیر محاسبه می‌کنیم:

$$\text{Branch Inst} = \frac{۱۱٫۴۴ \times ۴ \times ۱۰^۷}{۱۰۰۰} \times \frac{۱۰۰}{۹۴٫۸۵} \approx ۴۵٫۷۶ \times ۱۰^۵ \times ۱٫۰۵۴۳ \approx ۴,۸۲۴,۴۷۷$$

و در نتیجه:

$$\text{Branch Percentage} = \frac{۴,۸۲۴,۴۷۷}{۴ \times ۱۰^۷} \times ۱۰۰ \approx ۱۲٫۰۶٪$$

- تریس xalancbmk_748B.trace: این تریس با ۷۴۸ میلیارد دستور از سنگین‌ترین تریس‌های مورد استفاده در این تست‌ها است. حجم بالای آن باعث می‌شود که ارزیابی دقت و کارایی branch predictorها در مقیاس وسیع‌تری انجام شود. همچنین، تنوع بالا در نوع داده‌ها و الگوریتم‌های مورد استفاده، این تریس را به گزینه‌ای مناسب برای بررسی رفتار پیش‌بینی‌کننده‌ها در سناریوهای متنوع تبدیل کرده است.

CPU 0 Branch Prediction Accuracy: 96.33%

MPKI: 10.24

Average ROB Occupancy at Mispredict: 37.31

$$\text{Branch Inst} = \frac{۱۰٫۲۴ \times ۴ \times ۱۰^۷}{۱۰۰۰} \times \frac{۱۰۰}{۹۶٫۳۳} \approx ۴۰٫۹۶ \times ۱۰^۵ \times ۱٫۰۳۸۱ \approx ۴,۲۵۲,۰۵۸$$

$$\text{Branch Percentage} \approx \frac{۴,۲۵۲,۰۵۸}{۴ \times ۱۰^۷} \times ۱۰۰ \approx ۱۰٫۶۳٪$$

- تریس perlbench_53B.trace: این نسخه سبک‌تر از برنامه perlbench با ۵۳ میلیارد دستور، برای مقایسه رفتار پیش‌بینی‌کننده‌ها در شرایط کم‌بارتر به کار گرفته شده است. این تریس امکان مشاهده تغییرات دقیق‌تر در خروجی‌ها و حساسیت به پارامترها را فراهم می‌کند.

CPU 0 Branch Prediction Accuracy: 95.7%

MPKI: 9.713

Average ROB Occupancy at Mispredict: 50.02

$$\text{Branch Inst} = \frac{۹٫۷۱۳ \times ۴ \times ۱۰^۷}{۱۰۰۰} \times \frac{۱۰۰}{۹۵٫۷} \approx ۳۸٫۸۵ \times ۱۰^۵ \times ۱٫۰۴۵ \approx ۴,۰۴۸,۱۷۰$$

$$\text{Branch Percentage} \approx \frac{۴,۰۴۸,۱۷۰}{۴ \times ۱۰^۷} \times ۱۰۰ \approx ۱۰٫۱۲٪$$

درصد دستورات branch در هر سه تریس درصد بالایی است. توجه کنید این درصدها حدودی هستند و لازم است در هربار تست به صورت جداگانه درصد حدودی دستورات branch محاسبه شود؛ با این حال این مقدار بین ۱۰ تا ۲۰ درصد متغیر است.

۳-۳ پارامترهای قابل تنظیم در پیاده‌سازی

برای به دست آوردن بهترین دقت برای TAGE پیاده سازی شده باید پارامتر هایی که در tage.h مقدار دهی شدند را تغییر دهیم و با تست گرفتن به بهترین شیوه مقدار دهی پارامتر ها برسیم. حال به توضیح هر کدام از پارامتر های قابل تغییر می‌پردازیم.

- تعداد جدول ها:

همانطور که در قسمت پیاده سازی مشاهده کردیم الگوریتم TAGE از چندین جدول برای پیش بینی استفاده میکند بطوریکه هر جدول با استفاده از یک طول تاریخچه معین به پیش بینی می‌پردازد. اینکه در الگوریتم در مجموع از چند جدول استفاده کنیم می‌تواند تاثیر گذار باشد .تعداد جدول زیاد ممکن است بار محاسباتی ما را زیاد کند و اگر جدول های ما از یک حد کمتر بود الگوریتم TAGE معنای خود را از دست می‌دهد. برای تست کردن ما مقدار های ۸ ، ۱۰ ، ۱۲ ، ۱۴ را انتخاب کردیم.

- طول تاریخچه هر جدول :

در قسمت پیاده سازی بیان شد که به جدول با توجه با یک طول مشخص از تاریخچه branch ها به پیش بینی می‌پردازد.طول تاریخچه مصرفی در هر یک از جدول ها به صورت هندسی افزایش می‌یباد. حال مقدار اولیه برای اولین جدول و همچنین قدر نسبت این دنباله ی هندسی می‌تواند پارامتر های تاثیر گذاری باشند. همچنین توجه داشته داریم که GHR (متغیری که کل تاریخچه ما را ذخیره می‌کند)باید چند بیتی باشد به این خاطر که این مقدار نباید از بیشترین طول تاریخچه مورد نیاز برای یک جدول کمتر باشد. برای تست کردن ما مقدار های ۴ ، ۵را برای مقدار اولیه یعنی least-history-length و مقدار های ۱٫۷ ,۱٫۶, ۱٫۵ را به عنوان قدرنسبت این دنباله هندسی یعنی common-ratioانتخاب کردیم.

- عمق هر جدول :

هر جدول شامل چند entry است که با توجه به هش پیاده سازی شده هر ip با توجه به GHR به یک خانه از جدول نسبت داده می‌شود. اینکه هر جدول چند entry داشته باشد نیز یکی از پارامتر هایی است که می‌تواند تاثیر گذار باشد. برای مقدار دهی عمق جدول ها همانطور که در پیاده سازی گفته شد یک enum ساختیم و انواع روش های عمق دهی به جداول یعنی هم صعودی هم نزولی و هم به صورت گاوسی(ابتدا صعودی و سپس نزولی)را پیاده کردیم. پارامتر هایی که می‌توان در این روش پیاده سازی تغییر داد max-table-size ، min-table-size ، active-pattern که نوع مقدار دهی عمق ها را مشخص می‌کند است. برای

تست مقادیر مختلف برای max-table-sie مقدار ۱۱ ، ۱۲ و min-table-size مقدار ۹، ۱۰ و برای نوع پیاده سازی روش های constant , ascending , descending , hill-shaped در نظر گرفتیم.

- تعداد بیت های tag برای هر جدول :

مقدار بیت هایی که برای tag برای هر جدول مصرف می‌کنیم نیز مهم است . به ویژه رابطه این مقدار به اندازه تاریخچه مصرفی هر جدول نیز ارتباط دارد . اینکه اگر جدول طول تاریخچه بیشتری داشته باشد بهتر است مقدار tag در آن بیشتر باشد یا کمتر . با کمی پرس و جو و کمک هوش مصنوعی دریافتیم بهتر است برای جدول هایی که طول تاریخچه بیشتری مصرف می‌کنند مقدار tag کمتر باشد . از این رو با استفاده از روشی که در قسمت پیاده سازی توضیح داده شد با استفاده از دو پارامتر بیشترین مقدار tag و کمترین مقدار آن این مقدار را برای هر جدول تعیین کردیم . بنابراین دو پارامتر مربوطه tag-size-max , tag-size-min است که برای اولی مقدار ۱۰ ، ۱۲ و برای دومی مقدار ۶ ، ۸ را در نظر گرفتیم که تست کنیم.

توجه داریم بازه اعدادی که برای هر پارامتر درنظر گرفتیم با پرس و جو از منابع مختلف هست. حال باید با انجام تست شیوه ای از مقداردهی پارامتر ها را بیابیم که بیشترین دقت را دریافت کنیم . در ادامه به شیوه به دست آوردن این پارمتر ها می‌پردازیم.

۳-۴ روش بهینه‌سازی پارامترها و تست‌گیری

توجه داریم که ممکن است تغییر یک پارامتر روی پارامتر دیگر تاثیر بگذارد برای همین اینکه در هر تست یک پارامتر را مقدارش را عوض کنیم و مشاهده کنیم روی یک تریس مشخص کدام حالت بهینه می‌شود قطعا ساده انگارانه است چرا که ممکن است تغییر همزمان چند پارامتر با همدیگر باعث بهبود دقت شود.
بنابراین برای حل این مشکل و پیدا کردن بهینه پارامتر ها ابتدا به روشی که در ادامه توضیح می‌دهیم برای هر کدام از پارامتر ها مقادیر مختلف در بازه ای که در تظر گرفتیم را تست کردیم . در ادامه بر اساس دقت به دست آمده از هر کدام از تست ها به هر کدام از مقادیر پارامتر ها احتمال دادیم به طرویکه آن مقداری از پارامتر مشخص که دقت بیشتری را به همراه آورده احتمال بیشتری داشته باشد. در ادامه با توجه به این احتمال ها پارامتر ها را به صورت رندوم مقدار دهی می‌کنیم و دوباره تست می‌کنیم . با این روش هم از اینکه همه حالت های ممکن برای مقدار دهی پارامتر ها را تست کنیم جلوگیری کردیم هم از اینکه فقط با تغییر یک پارامتر به صورت ساده انگارانه تست انجام دهیم.

(توجه داریم که زمان هر شبیه سازی حدود چند دقیقه است که البته با تغییر تعداد دستور ها و نوع تریس فرق می‌کند اما در نهایت هر شبیه سازی مقدار قابل توجهی زمان می‌برد و واضح است مدت زمان تست کردن کل حالت ها بسیار زیاد است)

۳-۴-۱ بهینه کردن هر پارامتر به صورت مجزا

در بخش ۳-۳ پارامتر های قابل تغییر را بیان کردیم. اگر بخواهیم در هر تست فقط یکی از پارامتر ها را تغییر دهیم باید به اندازه ۳۰ تست برای هر تریس انجام دهیم . همچنین ۳ تریس نیز برای تست روی برنامه های محک مختلف انتخاب کردیم پس در مجموع باید ۹۰ تست انجام ددهیم . قطعا انجام دستی و در ادامه تحلیل داده های به دست آمده از تست ها طاقت فرسا است . به همین دلیل برای انجام تست از چند اسکریپت مختلف که با استفاده از پایتون نوشته شده است کمک گرفتیم.
همچنین در قسمت های قبل نیز توضیح داده شد مقادیر قابل تغییر را در یک فایل هدر جدا به نام tage-config.h به صورت یک کلاس که شامل پارامتر های قابل تغییر است قرار دادیم و این فایل را در هدر اصلی include کردیم.
برای انجام این مرحله از اسکریپت optimize-tage استفاده می‌کنیم . در این اسکریپت فایل tage-config.h بر اساس پارامتر هایی که ما می‌خواهیم با استفاده از تابع write-tage-config مقدار دهی می‌شوند . در ادامه تابع run-command دستوراتی که برای ران کردن تست روی یک تریس است را با استفاده از کتابخانه os در پایتون اجرا می‌کند. حال با استفاده از این دو تابع اصلی run-test را پیاده می‌کنیم که لیستی از پارامتر ها گرفته ابتدا فایل هدر را سخته سپس دستورات مورد نظر را اجرا کرده و در نهایت نتایج حاصل از شبیه سازی و همچنین ارور ها و پارامتر های مورد نظر در این تست را در پوشه مربوط به آن تست در پوشه اصلی output ذخیره می‌کند.
حال کافی است با استفاده از حلقه پارامتر های گفته شده را مقدار دهی کنیم لیست params را بسازیم و به تابعrun-testورودی دهیم تا تست های مختلف انجام شود.
در ادامه با استفاده از یک اسکرپت دیگر به نام csv-provider پوشه هر کدام از تست ها را باز کرده روی فابل نتایج هر تست جست و جو انجام می‌دادیم تا به کلمه Branch Prediction Accuracy برسیم سپس مقدار جلوی آن را خوانده و در یک فایل csv به صورت جدول و با ذکر پارامتر های تست ذخیره می‌کنیم . در اصل این اسکریپت وظیفه جمع آوری و ذخیره سازی اطلاعات از تست ها را دارد.
در ادامه با فایل csv کلی را به ۴ فایل تبدیل کردیم به صورتی که هر فایل مختص به یکی از پارامتر های قابل تغییر ذکر شده است . حال در ادامه با یک اسکریپت دیگر هر کدام از جدول های csv را

به یک نمودار خطی تبدلی می‌کنیم تا تحلیل راحت تر شود. همه این کار ها برای این بود که برای هر تریس مشخص کنیم هر پارامتر با کدام یک از مقدار هایش به تنهایی باعث بهینه شدن دقت می‌شود. توجه داریم همه فایلهای اسکریپت گفته شده ضمیمه شده است .

۳-۴-۲ به دست آوردن توزیع احتمالی برای هر پارمتر

حال با استفاده از نتایج به دست آمده از تست کردن تکی پارامتر ها برای هر کدام از تریس ها با توجه به دقت به دست آمده از مقدار دهی هر پارامتر به مقدار آنها یک احتمال نسبت می‌دهیم. به این صورت که هر چقدر دقت به دست امده بیشتر باشد مقدار آن پارامتر احتمال بیشتری داشته باشد . برای این کار ابتدا دقت های به دست آمده را نرمالایز می‌کنیم (یعنی میانگین آنها را از هر کدام کم کرده سپس تقسیم بر انحراف معیار داده ها می‌کنیم)
سپس مقادیر به دست آمده را از تابع soft-maxعبور می‌دهیم یعنی e را به توان ضریبی از داده (مثلا0.5) می‌رسانیم و سپس با تقسیم بر مجموع همه اعداد به دست آمده مقادیر را صورت احتمال یعنی اعداد بین صفر و یکی که جمع همه آنها یک شود در می‌آوریم. این محاسبات را با استفاده از اکسل انجام دادیم. برای مثال در زیر نحوه به دست آوردن توزیع برای پارامتر number-of-table برای سه traceمختلف را مشاهده می‌کنیم.

	A	B	C	D	E	F	G	H	I	J	K
1	NUMBER_	Accuracy	Trace			AVG	STD	NORMAL	EXP	SUM	probability
2	8	96.19	748B		748B	96.0375	0.209821	0.726809	1.438217	4.34071	0.331332
3	10	96.18	748B		135B	94.1225	0.300818	0.679149	1.40435	4.354228	0.32353
4	12	96.04	748B		53B	94.955	0.37045	0.011915	1.005975	4.366163	0.231754
5	14	95.74	748B					-1.41787	0.492167		0.113384
6	8	94.39	135B					0.889241	1.559898		0.358249
7	10	94.31	135B					0.6233	1.365676		0.313644
8	12	94.07	135B					-0.17452	0.916437		0.210471
9	14	93.72	135B					-1.33802	0.512216		0.117637
10	8	95.34	53B					1.039276	1.681419		0.385102
11	10	95.13	53B					0.472398	1.266426		0.290055
12	12	94.87	53B					-0.22945	0.891611		0.204209
13	14	94.48	53B					-1.28222	0.526706		0.120634

شکل ۲: به دست آوردن توزیع برای پارامتر number-of-tableبرای traceهای مختلف

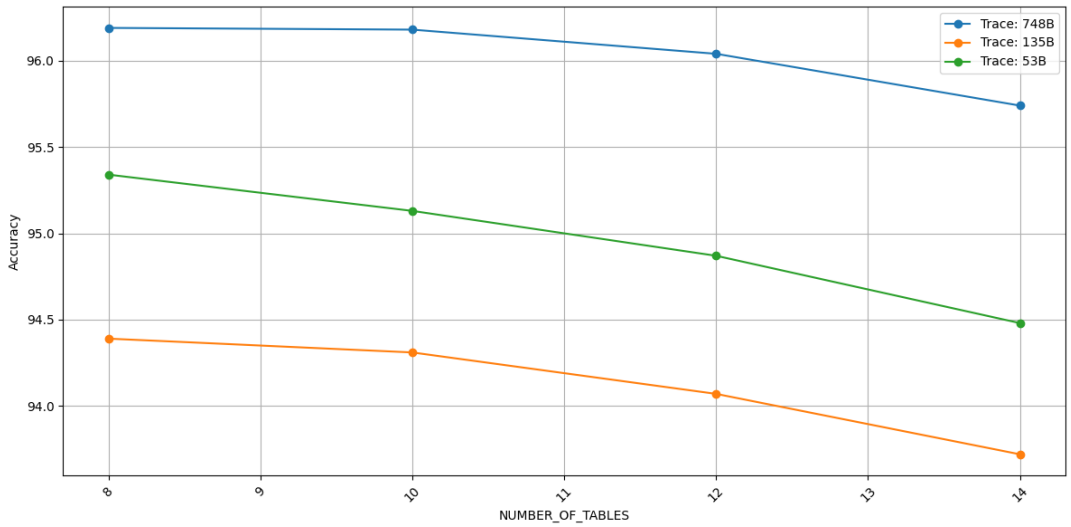
۳-۴-۳ تغییر همزمان پارمتر ها به صورت رندوم

حال که با استفاده از فرمول گفته شده برای هر کدام از پارامتر ها برای هر traceجداگانه توزیعی در آودریم که هرچقدر تستی که برای آن مقدار پارامتر انجام دادیم دقت بیشتری داشته باشد احتمال بیشتری به آن مقدار پارامتر داده شود.
حال با استفاده از یک اسکریپت به نام random-script مقادیر احتمالات را از اکسل های هر پارامتر استخراج کرده سپس با استفاده از آن هر پارامتر را به صورت رندوم انتخاب می‌کنیم . حال که پارامتر ها مشخص شد مانند قبل به تابع run-tst آن را وردی داده تا نتایج در پوشه های مربوطه به دست آید . در ادامه دوباره از اسکریپت csv-providerاستفاده کرده تا نتایج به دست آمده به صورت یک فایل csv به دست آید. در ادامه نتایج هر کدام از حالت های تست را مشاهده می‌کنیم.

۳-۵ نتایج به‌دست‌آمده از تنظیمات بهینه برای TAGE

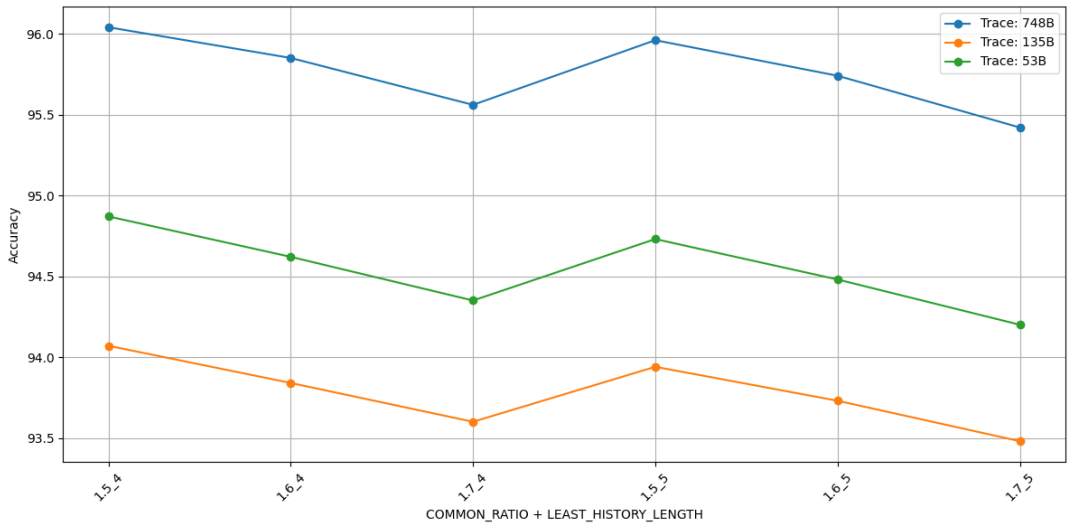
۳-۵-۱ نتایج تغییر مجزای هر پارامتر

با توجه به روشی که گفتیم در هر تست یک پارامتر را تغییر داده تا بهترین مقدار برای هر کدام و همچنین توزیع مربوط به آن پارامتر را به دست آوریم. نمودار نتایج مربوط به هر پارامتر به شرح زیر است



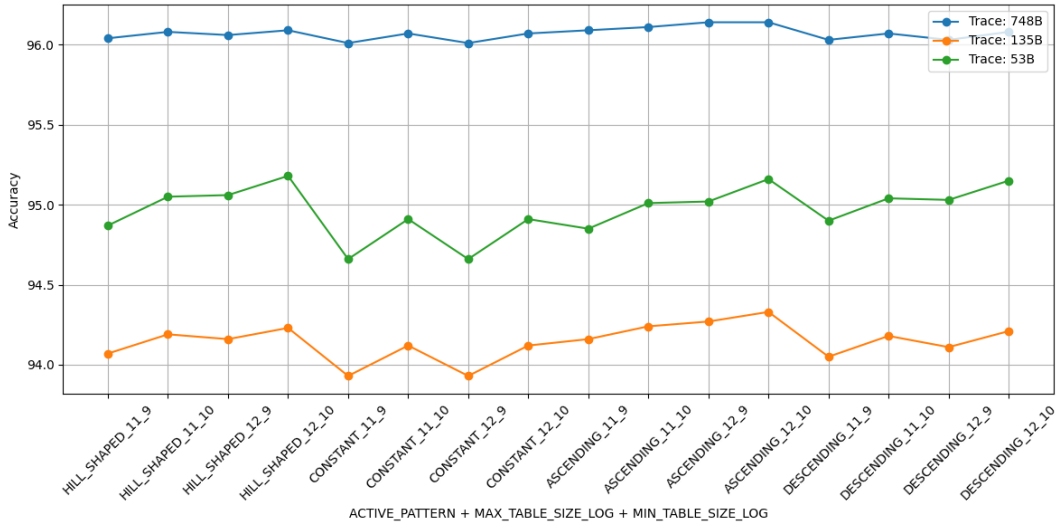
شکل ۳: نمودار به دست آمده از تغییر number-of-table

مشاهده می‌کنیم به صورت تکی برای این پارامتر یعنی number-of-table برای همه تریس ها مقدار ۸ باعث بهینه شدن دقت می‌شود.



شکل ۴: نمودار به دست آمده از تغییر پارمتر مربوط به طول تاریخچه جدول ها

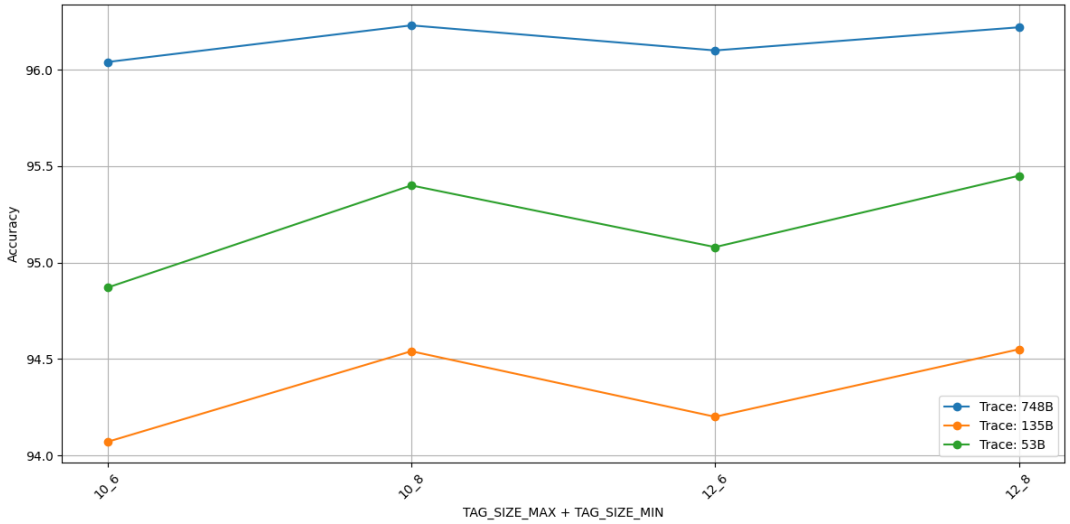
در این پارامتر نیز مشاهده می‌کنیم اگر بخواهیم به تنهایی در نظر بگریم بهتر است برای همه trace ها مقدارجدول اول را ۴ و قدر نسبت را ۱/۵ قرار دهیم.



شکل ۵: نمودار به دست آمده از تغییر پارامتر مربوط به عمق جداول

مشاهده می‌کنیم برای هر سه تریس تست شده اگر مقدار بیشینه جدول را ۱۲ و کمینه را ۱۰ قرار دهیم و همچنین در تریس های 135B , 748B نوع پیاده سازی را ASCENDING و در تریس 53B

HILL-SHAPED قرار دهیم به حالت بهینه تغییر تک hyper-parameterخواهیم رسید.



شکل ۶: نمودار به دست آمده از تغییر tag-size

مشاهده می‌کنیم برای این پارامتر نیز در هر سه trace اگر مقدار بیشینه را ۱۰ و مقدار کمینه را ۸ قرار دهیم قرار دهیم به حالت بهینه تغییر تک hyper-parameterخواهیم رسید.

حال که برای هر traceبرای هر پارامتر مقدار بهینه و توزیع آن را به دست آوردیم تست های رندوم را به روش گفته شده انجام می‌دهیم . همچنین یک تست نیز برای هر traceبه این صورت که هر پارامترش بهترین مقدار را که از تست های قبلی به دست آمده دارد اما این بار همزمان آن پارامتر ها قرار گرفتند انجام می‌دهیم .

۳-۵-۲ نتایج تغییرات همزمان پارامترها

- perlbench_135B.trace:

مشاهده می‌کنیم اگر هر پارامتر را بهترین مقدارش که به صورت تکی به دست آمده قرار دهیم در این traceبه دقت 95.69می‌رسیم در صورتی که به صورت رندوم به دقت 95.71نیز رسیدیم. به این صورت که باید پارامتر ها را به صورت زیر قرار دهیم :

TAG_SIZE_MIN	TAG_SIZE_MAX	NUMBER_OF_TABLES	MIN_TABLE_SIZE_LOG	MAX_TABLE_SIZE_LOG	LEAST_HISTORY_LENGTH	COMMON_RATIO	ACTIVE_PATTERN
۸	۱۲	۱۰	۱۰	۱۲	۴	۱۵	HILL_SHAPED

- perlbench_135B.trace:

	A	B	C	D	E	F	G	H	I	J	K	L
1	ACTIVE_PA	COMMON	LEAST_HIS	MAX_TABL	MIN_TABL	NUMBER_I	TAG_SIZE_	TAG_SIZE_	Accuracy	Test_ID		
2	DESCENDI	1.5	5	12	9	8	10	8	95.59	test_0		
3	HILL_SHAP	1.5	4	12	10	8	12	8	95.69	test_1		
4	CONSTAN	1.5	5	12	10	8	10	6	95.28	test_2		
5	ASCENDIN	1.5	4	12	10	10	12	8	95.68	test_3		
6	ASCENDIN	1.6	4	12	10	8	10	8	95.68	test_4		
7	HILL_SHAP	1.5	4	11	10	12	10	6	95.05	test_5		
8	DESCENDI	1.6	4	12	10	14	12	8	95.55	test_6		
9	DESCENDI	1.6	5	11	9	14	10	8	95.21	test_7		
10	DESCENDI	1.7	4	11	10	10	12	8	95.61	test_8		
11	ASCENDIN	1.6	4	11	9	8	12	6	95.32	test_9		
12	DESCENDI	1.5	4	11	10	10	12	8	95.59	test_10		
13	HILL_SHAP	1.5	4	11	9	10	12	8	95.49	test_11		
14	DESCENDI	1.6	4	12	9	10	10	8	95.59	test_12		
15	CONSTAN	1.7	5	12	9	14	12	8	94.97	test_13		
16	CONSTAN	1.5	4	12	10	12	12	6	95.09	test_14		
17	CONSTAN	1.5	4	12	9	8	10	8	95.3	test_15		
18	DESCENDI	1.5	5	11	10	12	10	8	95.51	test_16		
19	HILL_SHAP	1.5	4	12	9	10	12	8	95.61	test_17		
20	HILL_SHAP	1.6	4	12	10	12	10	8	95.58	test_18		
21	HILL_SHAP	1.5	5	12	10	10	10	8	95.65	test_19		
22	HILL_SHAP	1.6	4	12	10	8	10	8	95.7	test_20		
23	ASCENDIN	1.5	4	11	10	12	12	8	95.54	test_21		
24	DESCENDI	1.7	5	11	10	10	12	8	95.57	test_22		
25	DESCENDI	1.6	5	11	10	10	12	8	95.61	test_23		
26	ASCENDIN	1.7	5	11	10	14	10	8	95.01	test_24		
27	HILL_SHAP	1.6	5	12	9	10	10	8	95.52	test_25		
28	HILL_SHAP	1.5	4	12	10	10	12	8	95.71	test_26		
29	DESCENDI	1.5	4	12	9	8	12	6	95.44	test_27		
30	DESCENDI	1.5	4	12	9	10	12	8	95.6	test_28		
31	ASCENDIN	1.6	4	11	9	12	10	8	95.25	test_29		
32	HILL_SHAP	1.5	4	12	10	8	12	8	95.69	test_with_best_param		
33												

شکل ۷: نتایج به دست آمده با تست رندوم روی 153Btrace

	A	B	C	D	E	F	G	H	I	J	K
1	ACTIVE_PA	COMMON	LEAST_HIS	MAX_TABI	MIN_TABL	NUMBER_	TAG_SIZE_	TAG_SIZE_	Accuracy	Test_ID	
2	ASCENDIN	1.6	5	11	10	8	10	8	94.63	test_0	
3	HILL_SHAF	1.5	4	12	10	8	10	8	94.63	test_1	
4	DESCENDI	1.5	4	11	9	10	10	8	94.59	test_2	
5	DESCENDI	1.5	4	12	9	10	12	8	94.63	test_3	
6	CONSTANT	1.6	4	11	9	8	10	8	94.48	test_4	
7	DESCENDI	1.6	4	12	10	8	10	8	94.64	test_5	
8	HILL_SHAF	1.7	5	12	10	10	12	8	94.64	test_6	
9	HILL_SHAF	1.5	5	11	10	12	10	6	94.08	test_7	
10	DESCENDI	1.5	4	11	9	8	12	8	94.54	test_8	
11	HILL_SHAF	1.5	5	12	10	10	10	8	94.66	test_9	
12	ASCENDIN	1.7	5	11	9	14	10	8	94.12	test_10	
13	CONSTANT	1.6	4	12	9	8	10	6	94.29	test_11	
14	CONSTANT	1.5	4	11	10	10	12	8	94.61	test_12	
15	HILL_SHAF	1.6	4	12	10	12	12	8	94.63	test_13	
16	DESCENDI	1.6	4	11	9	12	12	6	94.07	test_14	
17	ASCENDIN	1.7	5	11	9	12	10	6	93.55	test_15	
18	CONSTANT	1.7	5	12	10	8	12	8	94.6	test_16	
19	HILL_SHAF	1.6	4	11	10	8	12	8	94.63	test_17	
20	HILL_SHAF	1.7	5	11	10	8	12	8	94.66	test_18	
21	ASCENDIN	1.6	5	12	10	10	12	6	94.46	test_19	
22	ASCENDIN	1.5	4	11	9	12	12	8	94.56	test_20	
23	HILL_SHAF	1.5	5	12	10	8	10	8	94.65	test_21	
24	CONSTANT	1.5	5	12	9	8	10	8	94.46	test_22	
25	HILL_SHAF	1.5	4	12	10	12	12	8	94.67	test_23	
26	ASCENDIN	1.5	4	12	10	8	10	8	94.64	test_24	
27	HILL_SHAF	1.5	4	11	9	12	12	8	94.55	test_25	
28	ASCENDIN	1.5	5	11	10	14	10	8	94.44	test_26	
29	ASCENDIN	1.5	5	12	9	10	10	8	94.62	test_27	
30	HILL_SHAF	1.6	4	11	10	12	10	6	93.98	test_28	
31	DESCENDI	1.7	4	12	10	8	12	6	94.52	test_29	
32	ASCENDIN	1.5	4	12	10	8	12	8	94.66	test_with_best_param	
33											

شکل ۸: نتایج به دست آمده با تست رندوم روی 135Btrace

مشاهده می‌کنیم اگر هر پارامتر را بهترین مقدارش که به صورت تکی به دست آمده قرار دهیم در این traceبه دقت 94.66می‌رسیم در صورتی که به صورت رندوم به دقت 94.67نیز رسیدیم. به این صورت که باید پارامتر ها را به صورت زیر قرار دهیم :

TAG_SIZE_MIN	TAG_SIZE_MAX	NUMBER_OF_TABLES	MIN_TABLE_SIZE_LOG	MAX_TABLE_SIZE_LOG	LEAST_HISTORY_LENGTH	COMMON_RATIO	ACTIVE_PATTERN
۸	۱۲	۱۲	۱۰	۱۲	۴	۱.۵	HILL_SHAPED

- xalancbmk_748B.trace:

	A	B	C	D	E	F	G	H	I	J
1	ACTIVE_PATTERN	COMMON_RATIO	LEAST_HISTORY_LENGTH	MAX_TABLE_SIZE_LOG	MIN_TABLE_SIZE_LOG	NUMBER_OF_TABLES	TAG_SIZE_MAX	TAG_SIZE_MIN	Accuracy	Test_ID
2	DESCENDING	1.6	4	11	10	10	12	6	96.17	test_0
3	CONSTANT	1.6	5	12	10	12	12	6	95.87	test_1
4	HILL_SHAPED	1.5	4	12	10	8	10	8	96.27	test_2
5	ASCENDING	1.5	5	11	9	12	12	8	96.22	test_3
6	ASCENDING	1.7	5	11	9	14	10	8	95.79	test_4
7	DESCENDING	1.6	5	12	9	12	12	8	96.16	test_5
8	ASCENDING	1.6	5	12	9	10	12	8	96.27	test_6
9	ASCENDING	1.5	4	12	10	8	12	8	96.28	test_7
10	ASCENDING	1.5	5	12	10	10	10	8	96.3	test_8
11	CONSTANT	1.6	5	12	9	8	10	8	96.25	test_9
12	CONSTANT	1.7	4	12	10	10	10	8	96.22	test_10
13	DESCENDING	1.6	4	11	10	12	12	8	96.2	test_11
14	CONSTANT	1.6	4	11	9	14	12	8	96.02	test_12
15	HILL_SHAPED	1.5	5	12	9	14	10	6	95.67	test_13
16	HILL_SHAPED	1.5	5	12	9	10	12	6	96.19	test_14
17	ASCENDING	1.5	4	12	10	8	10	6	96.26	test_15
18	DESCENDING	1.6	4	11	10	10	10	8	96.26	test_16
19	ASCENDING	1.5	4	12	10	8	12	6	96.26	test_17
20	HILL_SHAPED	1.7	4	12	9	10	12	6	96.07	test_18
21	HILL_SHAPED	1.6	4	12	10	14	12	6	95.45	test_19
22	DESCENDING	1.5	5	12	10	14	12	8	96.13	test_20
23	DESCENDING	1.6	5	11	10	12	12	6	95.88	test_21
24	HILL_SHAPED	1.5	4	12	10	12	12	6	96.13	test_22
25	ASCENDING	1.5	4	12	9	12	12	8	96.26	test_23
26	DESCENDING	1.5	4	11	9	10	12	6	96.19	test_24
27	ASCENDING	1.7	5	12	10	8	10	8	96.3	test_25
28	ASCENDING	1.6	4	12	9	8	12	6	96.24	test_26
29	ASCENDING	1.5	5	12	9	8	10	8	96.28	test_27
30	HILL_SHAPED	1.5	4	11	9	8	10	6	96.19	test_28
31	ASCENDING	1.7	5	12	9	14	10	6	94.89	test_29
32	ASCENDING	1.5	4	12	10	8	10	8	96.28	test with best for each param

شکل ۹: نتایج به دست آمده با تست رندوم روی 748Btrace

مشاهده می‌کنیم اگر هر پارامتر را بهترین مقدارش که به صورت تکی به دست آمده قرار دهیم در این traceبه دقت 96.28می‌رسیم در صورتی که به صورت رندوم به دقت 96.30نیز رسیدیم. به این صورت که باید پارامتر ها را به صورت زیر قرار دهیم :

TAG_SIZE_MIN	TAG_SIZE_MAX	NUMBER_OF_TABLES	MIN_TABLE_SIZE_LOG	MAX_TABLE_SIZE_LOG	LEAST_HISTORY_LENGTH	COMMON_RATIO	ACTIVE_PATTERN
۸	۱۰	۸	۱۰	۱۲	۵	۱.۷	ASCENDING

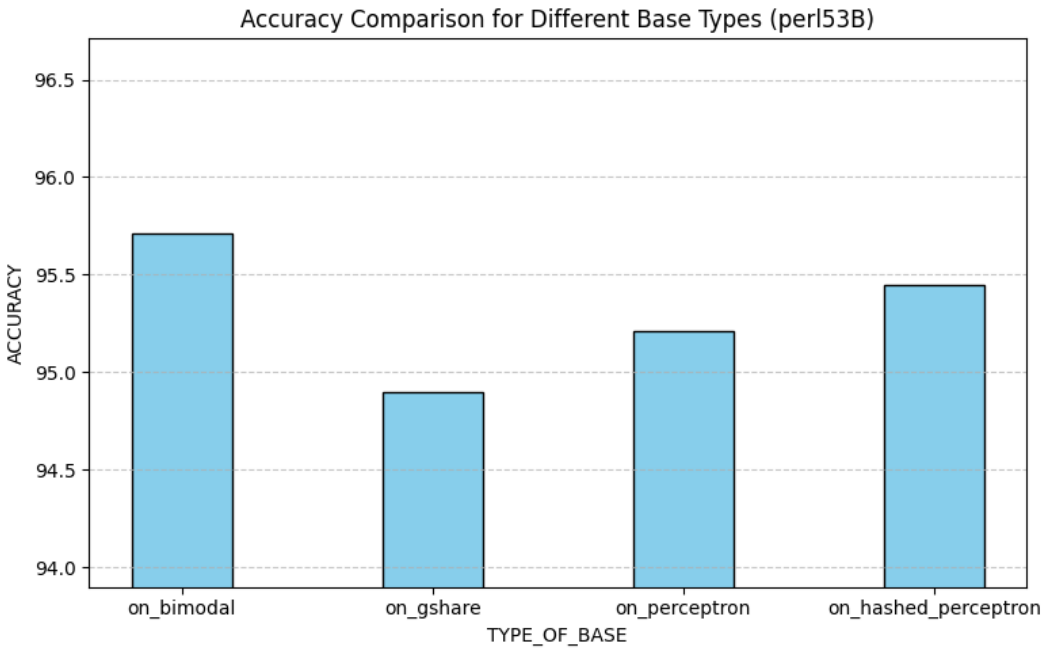
۳-۵-۳ تست با base های مختلف

همانطور که در قسمت پیاده سازی شرح داده شد اگر جدول مناسب در پیدا نشد الگوریتم TAGE از یک base-predictorاستفاده می‌کنیم. واضح است که نوع این base-predictorمهم است و خود می‌تواند یک hyper-parameterباشد .

از ما خواسته شده بود ابتدا با فرض اینکه این base-predictorما bimodalاست بقیه پارامتر ها را طوری مشخص کنیم که دقت بیشینه شود سپس با تغییر base-predictor و تست کردن با انواع مختلف آن اینکه کدام باعث بیشینه شدن دقت می‌شود را مشخص کنیم . ما نیز برای هر کدام از traceها با استفاده از بهترین پارامتر هایی که در قسمت قبل به دست آوردیم روی branch-

predictorهای مختلف تست می‌کنیم تا مقدار بهترین base را به دست آوریم.

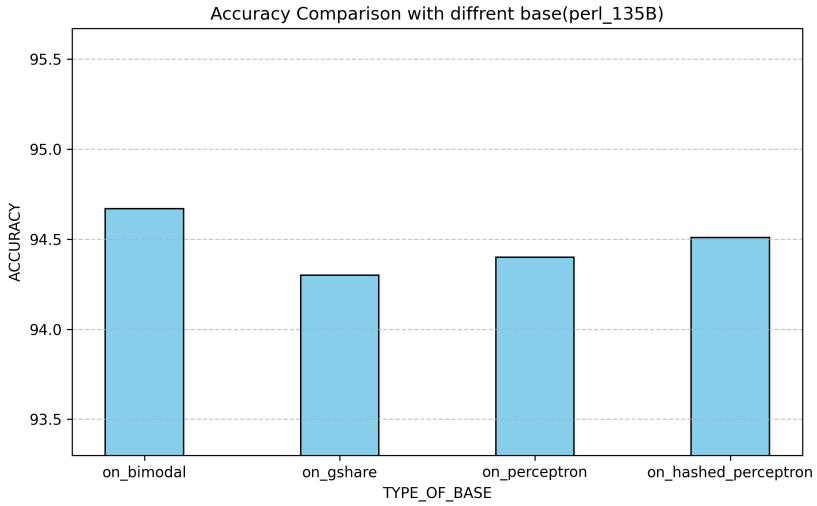
- perlbench_53B.trace:



شکل ۱۰: مقایسه عملکرد base-predictorهای مختلف روی 53B-trace

مشاهده می‌کنیم در این traceبهترین نتیجه با bimodalبه دست آمده است.

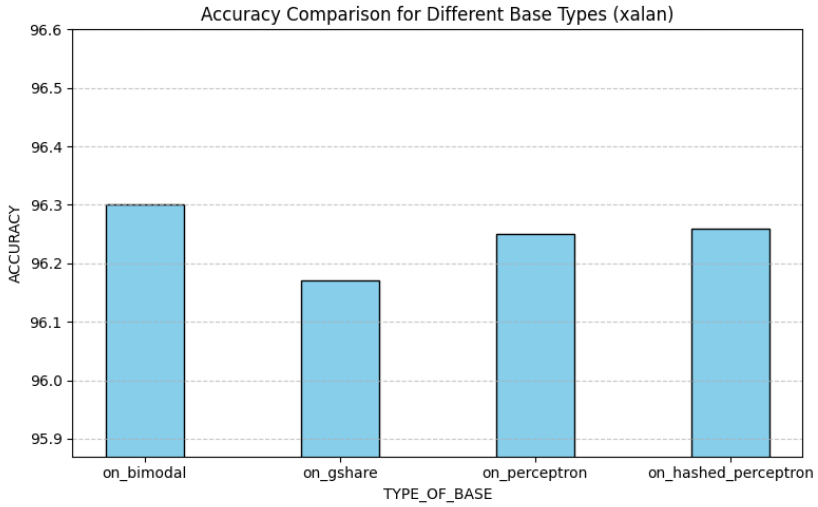
- perlbench_135B.trace:



شکل ۱۱: مقایسه عملکرد base-predictorهای مختلف روی 135B-trace

مشاهده می‌کنیم در این traceنیز بهترین نتیجه با bimodal : branch-pridictorبه دست می‌آید.

- xalancbmk_748B.trace



شکل ۱۲: مقایسه عملکرد base-predictorهای مختلف روی 758B-trace

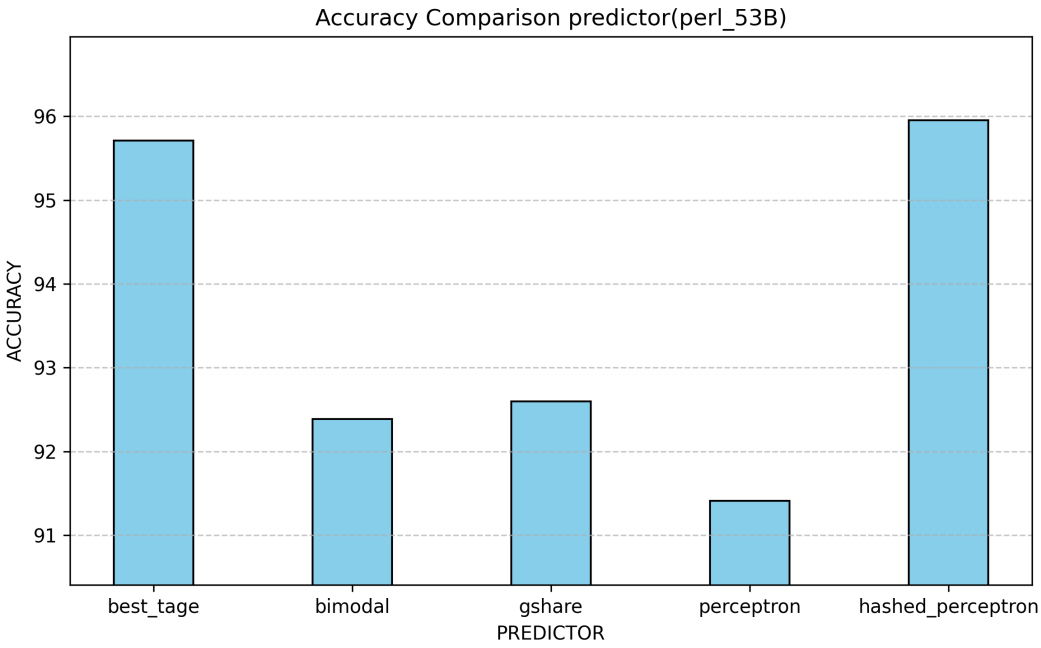
در این traceنیز اگر baseرا bimodalقرار دهیم به بهترین نتیجه خواهیم رسید.

مشاهده کردیم در چند بخش گذشته برای هر کدام از traceها با استفاده از روش گفته شده پارامتر هایی که باعث بهبود دقت می‌شود را پیدا کرده و همچنین اینکه base در پیاده سازی آن چه باشد نیز مشخص کردیم . در ادامه برای هر trace بهترین tage ای که به دست آوردیم را با branch-predictorهای دیگر مقایسه می‌کنیم.

۳-۶ مقایسه عملکرد TAGE با سایر branch predictorها

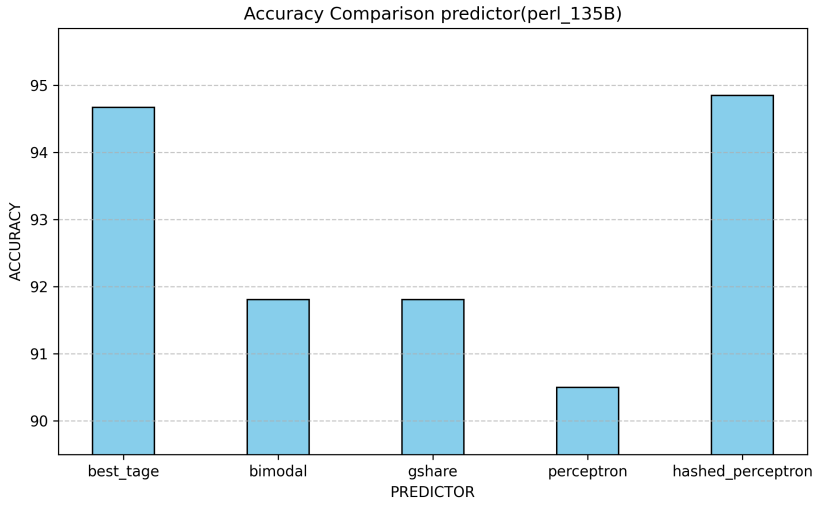
حال که برای هر کدام از trace ها بهترین پارمتر ها و همچنین بهترین baseها را به دست آوزدیم به مقایسه branch-predictorای که طراحی کردیم با سایر branch-predictorها می‌پردازیم . باز نیز برای هر traceجداگانه عمل می‌کنیم . مشاهده میکنیم برای هر سه traceای که شبیه سازی را روی آن ها تست کردیم tage ای که پیاده کردیم تنها به مقدار اندکی از hashed-perceptronدقت کمتری در پیش بنی داشت اما از سایر branch-predictorها دقیق تر بود.نتایج را به صورت نمودار ستونی در زیر مشاهده می‌کنیم :

- perlbench_53B.trace



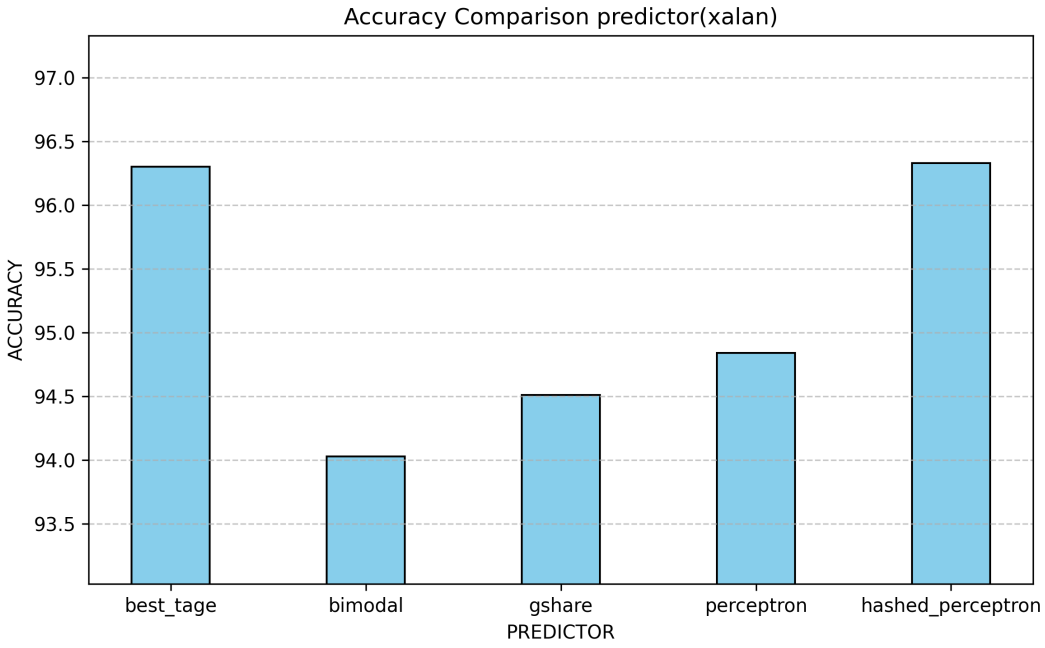
شکل ۱۳: مقایسه عملکرد branch-predictor های مختلف روی 53B-trace

- perlbench_135B.trace



شکل ۱۴: مقایسه عملکرد branch-predictorهای مختلف روی 135-trace

- xalancbmk_748B.trace



شکل ۱۵: مقایسه عملکرد branch-predictorهای مختلف روی 758B-trace