دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

گزارش پروژه سیستم‌های عامل

نگارش

مهرشاد دهقانی

یوسف سدیدی

امیرعلی شیخی

کیهان مسعودی

استاد درس

جناب آقای دکتر اسدی

مسئول پروژه

جناب آقای مرادی

بهمن ۱۴۰۳

فهرست مطالب

فهرست تصاویر

فصل ۱

مقدمه

در دنیای مدرن پردازش و ذخیره‌سازی داده‌ها، کارایی سیستم‌های کامپیوتری نقش مهمی در بهینه‌سازی عملکرد دارد. یکی از چالش‌های اساسی در این حوزه، مدیریت حافظه نهان (Cache) به‌گونه‌ای است که دسترسی به داده‌های پرکاربرد سریع‌تر انجام شود. در این پژوهش، چهار الگوریتم مختلف برای سیاست‌های جایگزینی حافظه نهان شامل Oracle/Belady N-hit، ARC و LRU بررسی و پیاده‌سازی شده‌اند.

۱-۱ تعریف مسئله

با افزایش حجم داده‌ها و نیاز به پردازش سریع، مدیریت کارآمد حافظه نهان به یکی از مسائل کلیدی در سیستم‌های ذخیره‌سازی و پردازشی تبدیل شده است. هدف اصلی، انتخاب سیاستی بهینه برای جایگزینی داده‌ها در حافظه نهان است که بتواند نرخ برخورد (Hit Rate) را افزایش داده و زمان تأخیر دسترسی به داده‌ها را کاهش دهد.

۱-۲ اهمیت موضوع

انتخاب سیاست مناسب برای مدیریت حافظه نهان تأثیر مستقیمی بر عملکرد سیستم‌های کامپیوتری دارد. بهینه‌سازی این سیاست‌ها می‌تواند باعث کاهش بار پردازشی، بهبود سرعت پاسخگویی و افزایش بهره‌وری سیستم‌های ذخیره‌سازی شود. مقایسه و تحلیل این الگوریتم‌ها به درک بهتر نقاط

قوت و ضعف هر روش کمک می‌کند.

۱-۳ اهداف پژوهش

- بررسی و پیاده‌سازی الگوریتم‌های مختلف مدیریت حافظه نهان.
- تحلیل و مقایسه عملکرد الگوریتم‌های Oracle/Belady N-hit، ARC، و LRU.
- ارزیابی میزان کارایی هر الگوریتم بر اساس معیارهایی مانند نرخ برخورد و زمان پاسخگویی.
- رسم نمودار برای نشان دادن بهتر نتایج و مقایسه الگوریتم‌ها با هم

فصل ۲

الگوریتم‌ها

در اینجا به بررسی دقیق چهار الگوریتم ذکر شده می‌پردازیم و کدهای توسعه داده شده را توضیح می‌دهیم. همچنین تصاویر خروجی هر کدام از این الگوریتم‌ها روی چهار فایل متن‌باز Alibaba و تصاویر نمودارها را نشان می‌دهیم.
همه کدها به زبان C++ در پیوست این سند موجود است.

۲-۱ الگوریتم Oracle/Belady

۲-۱-۱ توضیح کارکرد الگوریتم

الگوریتم Oracle به این صورت عمل می‌کند که به هنگام وقوع cache miss درصورتی که حافظه نهان فضای خالی داشته باشد, بلاک مورد نظر از حافظه را در حافظه نهان قرار می‌دهد و در غیر این صورت, اگر دسترسی بعدی به این بلوک حافظه زودتر از دسترسی یکی از بلوک‌های موجود در حافظه نهان اتفاق بیوفتد, بلوکی از حافظه نهان که دسترسی بعدی به آن دیرتر از بقیه بلوک‌ها است, حذف می‌شود و بلوک دسترسی فعلی جایگزین آن می‌شود. به عبارت دیگر این الگوریتم بر اساس دسترسی‌های آینده تصمیم می‌گیرد که یک بلوک وارد حافظه نهان بشود یا خیر.

۲-۱-۲ نحوه پیاده‌سازی الگوریتم

در این نحوه پیاده‌سازی خودمان از الگوریتم را شرح می‌دهیم.

تابع process_future_occurrences

در این تابع که در تصویر زیر آمده است, vector مربوط به دسترسی‌ها به طور معکوس مورد بررسی قرار می‌گیرد و زمان دسترسی به هر آدرس درون vector مربوط به آن آدرس در یک hashmap به نام future_occurrences نگه داری می‌شود. بدین صورت در پایان اجرای این تابع برای مجموعه آدرس‌های ورودی, یک hashmap خواهیم داشت که در آن key , آدرس و value , یک آرایه از زمان‌های دسترسی به آن آدرس از حافظه است.

نکته ۱: در این جا منظور از زمان دسترسی به آدرس x آن است که در چندمین دسترسی به x می‌رسیم. نکته ۲: ترتیب زمان دسترسی‌ها در آرایه هر آدرس به صورت معکوس ذخیره می‌شود یعنی آخرین عضو آرایه اولین زمان دسترسی را نشان می‌دهد. نکته ۳: با دقت در کد متوجه می‌شوید که در هر آرایه در اولین خانه یک مقدار صفر ذخیره شده است. کاربرد این خانه مشخص کردن این است که آیا برای اولین بار آن آدرس miss خورده و یا خیر.

```cpp
// Function to preprocess future occurrences of pages
unordered_map<long long int, vector<int> > preprocess_future_occurrences(const vector<tuple<long long int, string> > &sequence, int piece_count, int piece_num) {
    unordered_map<long long int, vector<int> > future_occurrences;
    int size = sequence.size();
    int ceill = (size + piece_count - 1) / piece_count;
    int start = ceill * piece_num;
    int end = min(ceill * (piece_num + 1), size) - 1;

    for (int i = end; i >= start; --i) {
        long long int page = get<0>(sequence[i]);
        if (future_occurrences.find(page) == future_occurrences.end()) {
            future_occurrences[page].push_back(0);  // Dummy value for cold miss detection
        }
        future_occurrences[page].push_back(i);  // Store future occurrence index
    }
    return future_occurrences;
```

شکل ۲-۱: تابع process_future_occurrences

تابع optimal_cache_replacement_with_set

در تابع دوم از یک multiset برای حافظه نهان استفاده شده است. دلیل این موضوع کاهش زمان درج در حافظه نهان است. در این multiset اعضای cache بر حسب زمان دسترسی بعدی و به صورت نزولی نگه داری می‌شوند. در این تابع یک hashmap نیز داریم که به هر آدرس موجود در حافظه نهان پوینتر به محل آن بلوک در multiset را نسبت می‌دهد.

```
multiset<pair<int, long long int>, CompareNextUse> cache;  // Cache uses a custom comparator for decreasing order
unordered_map<long long int, multiset<pair<int, long long int> >::iterator> cache_map;  // Map storing iterators to set elements
```

شکل ۲-۲: دو داده ساختار جهت شبیه سازی حافظه نهان

بدین صورت در هر دسترسی بررسی می‌کنیم که آدرس مورد نظر در hashmap موجود است یا خیر و سپس در صورت وقوع miss و پر بودن cache, به وسیله if (it->first > next_use) بررسی می‌کنیم که اگر دسترسی بعدی به آدرس فعلی پیش از دسترسی بعدی به اولین آدرس cache باشد, این خانه را جایگزین اولین خانه موجود در حافظه نهان (یعنی خانه ای که دورترین دسترسی را دارد) می‌کنیم.

```
if (cache.size() == cache_size) {
    // Evict the page with the biggest "next use" (i.e., the one at the beginning)
    auto it = cache.begin();
    if (it->first > next_use) {
        cache_map.erase(it->second);  // Remove from cache_map
        cache.erase(it);  // Remove from cache set

        if (future_occurrences[page][0] == 0) {
            cold_misses++;  // Detect cold miss
            future_occurrences[page][0] = 1;
        }

        auto insert_it = cache.insert({next_use, page});  // Insert new page into cache
        cache_map[page] = insert_it;  // Store iterator in cache_map
    } else {
        out_misses++;  // Page can't be inserted because its next use is too far
    }
} else {
    // Cache is not full, insert page directly
    if (future_occurrences[page][0] == 0) {
        cold_misses++;  // Detect cold miss
        future_occurrences[page][0] = 1;
    }
    auto insert_it = cache.insert({next_use, page});  // Insert new page into cache
    cache_map[page] = insert_it;  // Store iterator in cache_map
}
```

شکل ۲-۳: کد بخش جایگذاری و eviction در حافظه نهان

تابع read_sequence

در تابع read_sequence آدرس و نوع دسترسی برای دسترسی‌هایی که در بازه زمان مشخص شده هستند, استخراج می‌شود. در واقع اولین time_stamp به عنوان first_time در نظر گرفته می‌شود و بعد از آن تنها مواردی در data قرار می‌گیرند که time_stamp برای آنها در بازه first_time+start و first_time+end باشد. خروجی این تابع یک vector از دوتایی‌های آدرس و نوع دسترسی است.

```cpp
// Function to read the offsets & request types from the CSV file
vector<tuple<long long int, string> > read_sequence(const string& filename, long long int start, long long int end) {
    long long int first_time;
    vector<tuple<long long int, string> > data;
    ifstream file(filename);
    if (!file) {
        cerr << "Error: Could not open the file." << endl;
        return data;
    }

    string line;
    int first_line = 1;
    while (getline(file, line)) {

        stringstream ss(line);
        string time_stamp, offset, temp, request_type;

        if (getline(ss, time_stamp, ',') &&
            getline(ss, temp, ',') &&
            getline(ss, offset, ',') &&
            getline(ss, temp, ',') &&
            getline(ss, request_type, ',')) {
            try {
                long long int time = stoll(time_stamp);
                if (first_line == 1)
                {
                    first_line = 0;
                    first_time = time;
                }

                if (time >= first_time + start && time <= first_time + end)
                    data.emplace_back(stoll(offset), request_type);
            } catch (const invalid_argument&) {
                cerr << "Warning: Non-integer value encountered and skipped: " << line << endl;
            } catch (const out_of_range&) {
                cerr << "Warning: Value out of range encountered and skipped: " << line << endl;
            }
        }
    }
    file.close();
    return data;
}
```

شکل ۲-۴: تابع read_sequence

نتیجه

با توجه به این موضوع که این الگوریتم زمان دسترسی به آدرس‌ها در آینده را مبنای کار خود قرار می‌دهد, واضح است که این الگوریتم به هیچ وجه قابل استفاده به عنوان policy برای یک cache واقعی نیست و تنها برای بررسی و مقایسه نتایج قابل استفاده است. نکته دیگر در مورد این الگوریتم این است که این الگوریتم با توجه به روش توضیح فوق, در واقع بهترین تصمیم را در مورد قرار گرفتن یا نگرفتن یک خانه در cache و همچنین محل قرارگیری آن می‌گیرد. با توجه به این موضوع

واضح است که عملکرد هر الگوریتم دیگری در شرایط مشابه از این الگوریتم پایین‌تر است.

۲-۲ الگوریتم N-hit

۲-۲-۱ توضیح کارکرد الگوریتم

الگوریتم N-hit دو پارامتر به نام‌های trigger_threshold و insertion_threshold دارد. این سیاست به این صورت کار می‌کند که تا وقتی که حافظه نهان به اندازه trigger_threshold پر نشده با هر بار miss داده را promote (وارد حافظه نهان) می‌کند. اما در این پروژه نیازی به پیاده‌سازی این مکانیزم نبوده‌است و الگوریتم از همان ابتدا به صورت عادی عمل می‌کند. سیاست ورود به حافظه نهان یا همان promote به این صورت که تعداد دسترسی‌ها به هر آدرس مجازی را می‌شمارد، با هر بار دسترسی چک می‌کند که آیا تعداد دسترسی‌ها به مقدار

برای سیاست خارج کردن از حافظه نهان یا همان eviction به این صورت است که اگر حافظه پر باشد آن داده‌ای که کمترین تعداد دسترسی را تا این لحظه دارد، از cache بیرون می‌رود. اگر چند داده تعداد دسترسی یکسان داشتند، آن داده که زودتر وارد شده را خارج می‌کنیم.

۲-۲-۲ نحوه پیاده‌سازی الگوریتم

در این نحوه پیاده‌سازی خودمان از الگوریتم را شرح می‌دهیم. برای پیاده‌سازی ابندا دو استراکت به نام‌های Request و CacheItem را به شکل زیر ساخته‌ایم.

تایپ Request فیلد زمان، آدرس منطقی و نوع Request دارد (خواندن یا نوشتن) دارد. تایپ CacheItem فیلد آدرسی منطقی، تعداد دسترسی و زمان ورود به حافظه نهان را دارد. همچنین یک تایع مقایسه برای این تایپ تعریف کردیم که مطابق سیاست خارج کردن از حافظه نهان است. در کلاس NhitCache متغیرهای زیر وجود دارند:

- cache size: اندازه کش بر حسب تعداد آدرس منطقی.
- insertion_threshold: در قسمت قبل بیان شد.

```cpp
struct Request {
    long long timestamp;
    std::string logical_address;
    std::string request_type;
};

struct CacheItem {
    std::string logical_address;
    int access_count;
    long long insertion_time;

    // Comparator for std::set
    bool operator<(const CacheItem& other) const {
        if (access_count == other.access_count) {
            return insertion_time < other.insertion_time; // FIFO if access counts are equal
        }
        return access_count < other.access_count; // Evict least accessed first
    }
};
```

شکل ۲-۵: استراکت Request و CacheItem

- start_time,end_time: بازه زمانی مورد بررسی.
- cache: هش مپ از آدرس‌های منطقی به cacheItem.
- access_counts: هش مپ از آدرس‌های منطقی به تعداد دفعات دسترسی.
- eviction_set: ست برای حذف کردن از حافظه نهان. به دلیل اینکه می‌شود در $O(logn)$ کوچکترین عنصر را طبق تابع مقایسه که پیش‌تر مطرح کردیم، می‌توانیم پیدا کنیم و همچنین یک عنصر دلخواه را در $O(logn)$ از آن حذف کنیم.

همچنین متغیرهای مربوط به معیارهای مقایسه نیز در این کلاس وجود دارند. سه تابع زیر در این کلاس پیاده‌سازی شده‌است:

- process_request: یک درخواست یعنی یک خط از فایل را بررسی می‌کند.
- evict: عنصر با کمترین تعداد دسترسی را حذف می‌کند.
- print_metrics: خروجی‌های مورد نظر را چاپ می‌کند.

در تابع request process منطق بیان شده در الگوریتم پیاده‌سازی شده‌است و همچنین برای پیدا کردن کوچکترین عنصر در set از تابع begin و برای حذف کردن عنصر از حافظه از تابع erase استفاده شده‌است. به دلیل استفاده از داده‌ساختار مناسب مانند set الگوریتم بهینه است.

۲-۳ الگوریتم ARC

الگوریتم ARC (Adaptive Replacement Cache) ترکیبی از دو سیاست حافظه نهان LRU و LFU است که به‌طور پویا بین داده‌های اخیراً استفاده‌شده و داده‌های پرتکرار تعادل برقرار می‌کند. پیاده‌سازی این الگوریتم شامل چهار لیست اصلی است:

- T1: لیستی از بلاک‌های اخیراً استفاده‌شده که هنوز پرتکرار نیستند.
- T2: لیستی از بلاک‌هایی که به‌دفعات استفاده شده‌اند و پرتکرار محسوب می‌شوند.
- B1: لیستی از بلاک‌هایی که قبلاً در T1 بودند اما به دلیل محدودیت فضا حذف شده‌اند.
- B2: لیستی از بلاک‌هایی که قبلاً در T2 بودند اما به دلیل محدودیت فضا حذف شده‌اند.

۲-۳-۱ نحوه پیاده‌سازی الگوریتم

پیاده‌سازی این الگوریتم در کلاس ARC_Cache شامل توابع زیر است:

- void access(int key, const std::string& request_type): این تابع برای پردازش درخواست‌های خواندن و نوشتن داده‌ها در کش استفاده می‌شود. ابتدا بررسی می‌کند که آیا داده در T1 یا T2 وجود دارد (در این صورت یک hit رخ می‌دهد) یا باید از حافظه اصلی دریافت شود (miss). همچنین در صورت نیاز داده را از B1 یا B2 بازیابی کرده و مقدار p را تنظیم می‌کند.
- void moveToT2(int key): اگر داده‌ای در لیست T1 باشد و مجدداً مورد استفاده قرار گیرد، به T2 منتقل می‌شود تا نشان دهد که این داده پرتکرار است.
- void replace(int key): عملیات جایگزینی را انجام می‌دهد. اگر کش پر باشد، یکی از داده‌های قدیمی از T1 یا T2 حذف شده و در B1 یا B2 ذخیره می‌شود. این تابع مشخص می‌کند که داده از کدام لیست حذف شود، که این انتخاب به مقدار پارامتر p بستگی دارد.
- void processTraceFile(const std::string &filename, int cache_size, long long start_time, long long end_time): این تابع فایل trace ورودی را پردازش کرده، داده‌های مورد نیاز را استخراج و برای هر درخواست تابع access را فراخوانی می‌کند. سپس آمار کلی عملکرد کش شامل تعداد درخواست‌ها، برخوردها و عدم برخوردها را نمایش می‌دهد.

۲-۳-۲ سیاست جایگزینی داده

وقتی حافظه نهان پر می‌شود، یکی از بلاک‌های قدیمی بر اساس مقدار پارامتر p از T1 یا T2 حذف شده و در لیست‌های B1 یا B2 نگهداری می‌شود. این فرایند باعث تنظیم پویا بین دو نوع دسترسی می‌شود:

۲-۴ الگوریتم LRU

۲-۴-۱ توضیح کارکرد الگوریتم

این برنامه یک شبیه سازی از الگوریتم جایگزینی افظه LRU(cache replacement) است.الگوریتمLRU یک استراتژی مدیریت حافظه است که در آن وقتی حافظه نهان (cache) پر می شود و نیاز به حذف یکی از عناصر دارد، داده‌ای حذف می‌شود که قدیمی‌ترین استفاده را داشته است زیرا احتمالا در آینده نزدیک مورد استفاده قرار خواهند گرفت.

۲-۴-۲ نحوه پیاده‌سازی الگوریتم

استراکت CacheStatistics: در این ساختار داده،آماری از عملکرد حافظه نهان شامل تعداد کل برخوردها و عدم برخوردها ،تعداد عملیات‌های خواندن و نوشتن و سایر فیلدهای آماری وجود دارد.

```
struct CacheStatistics {
    int total_hits = 0;
    int total_misses = 0;
    int cold_misses = 0;
    int total_reads = 0;
    int total_writes = 0;
    int total_read_hits = 0;
    int total_read_misses = 0;
    int total_write_hits = 0;
    int total_write_misses = 0;
};
```

شکل ۲-۶: استراکت CacheStatistics

تابع get_first_timestamp: از این تابع برای پیدا کردن اولین زمان درخواست از فایل CSV استفاده

می شود به این صورت که فایل را باز می‌کند و اولین سطر را می‌خواند و سپس مقدار اولین ستون را استخراج کرده و به عنوان مبدا زمانی استفاده می‌کند.

تابع lru_cache_simulation: در این تابع شبیه سازی الگوریتم LRU برای داده ها در بازه ای مشخص انجام می‌شود. در ایتدا از یک متغیر cache_map (unordered_map) برای نگه‌داری محل داده ها در لیست حافظه نهان و از یک متغیر(list) cache برای نگه‌داری ترتیب دسترسی به داده‌ها استفاده می‌کنیم. همچنین از یک seen_offest (unordered_set) برای بررسی این که آیا این آدرس برای اولین بار در cache قرار گرفته و coldMiss رخ داده است ،استفاده می‌شود. در این تابع فایل به صورت خط به خط خوانده شده و قسمت‌هایی که در بازه زمانی مشخص شده است را با دستورات لازم انتخاب کرده و در صورتی که در حافظه نهان موجود بود داده را به اول لیست متغیر cache می‌بریم و در غیر این صورت آخرین عضو cache را حذف کرده و آدرس جدید را در آن قرار می‌دهیم.

در تابع main نیز ورودی ها به ترتیب گرفته شده و بعد از یکسان سازی معیارهای اندازه گیری ،شبیه سازی cache را با این ورودی ها اجرا می کنیم.

۲-۵ الگوریتم امتیازی LARC

الگوریتم LARC (Learning Adaptive Replacement Cache) نسخه‌ای بهینه‌تر از ARC است که با استفاده از یادگیری الگوهای دسترسی، بهبودهایی در سیاست جایگزینی داده‌ها ارائه می‌دهد. تفاوت اصلی آن با ARC در این است که به جای استفاده صرف از دو لیست LRU و LFU، از یک روش یادگیری برای تنظیم میزان تخصیص حافظه به داده‌های اخیراً استفاده‌شده و داده‌های پرتکرار بهره می‌برد.

۲-۵-۱ نحوه پیاده‌سازی الگوریتم

پیاده‌سازی این الگوریتم شامل ساختارهایی مشابه با ARC است اما تفاوت‌هایی در مدیریت و تصمیم‌گیری‌ها دارد. کلاس LARC_Cache دارای توابع کلیدی زیر است:

- void access(int key, const std::string& request_type): این تابع بررسی می‌کند که آیا داده در T1 یا T2 وجود دارد. در صورت وجود، آن را به ابتدای لیست مربوطه منتقل کرده و

یک برخورد (hit) ثبت می‌کند. در غیر این صورت، مکانیزم جایگزینی را برای بارگذاری داده اجرا می‌کند.

- void lazyPromoteToT2(int key): اگر داده‌ای در T1 باشد و مجدداً به آن دسترسی پیدا شود، این تابع آن را به T2 منتقل می‌کند. این انتقال تحت تأثیر وزن یادگیری p انجام می‌شود تا کارایی حافظه نهان بهینه شود.

- void replace(int key): در هنگام نیاز به حذف یک بلاک، این تابع تصمیم می‌گیرد که آیا داده‌ای از T1 حذف شده و به B1 منتقل شود یا از T2 به B2. این تصمیم بر اساس مقدار p و نسبت اندازه‌های B1 و B2 گرفته می‌شود.

- void processTraceFile(const std::string &filename, int cache_size, long long start_time, long long end_time): این تابع فایل ردگیری را پردازش کرده و درخواست‌های خواندن و نوشتن را درون کش بررسی می‌کند. همچنین نرخ برخورد و نرخ عدم برخورد را محاسبه کرده و خروجی نهایی را نمایش می‌دهد.

فصل ۳

نتیجه‌گیری

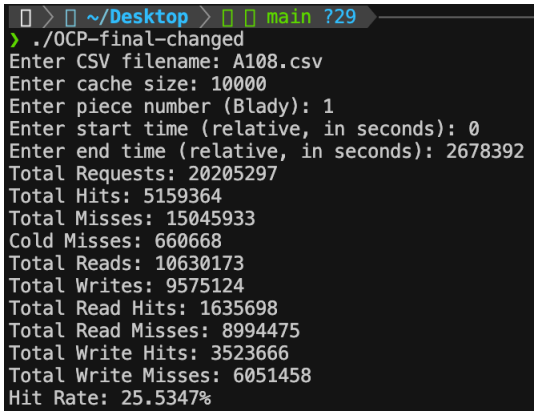
در ادامه تصاویر مربوط به خروجی الگوریتم‌های توضیح داده‌شده برای فایل‌های Alibaba آمده است.

۳-۱ خروجی Oracle/Belady

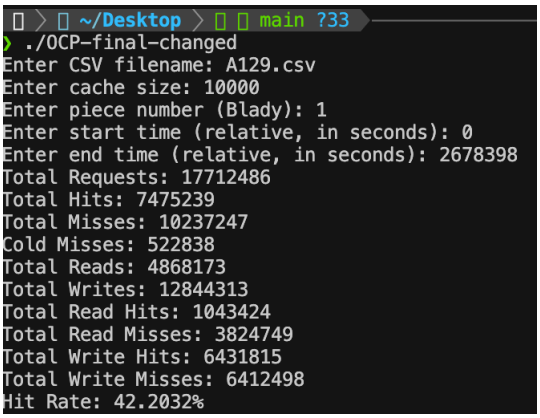
خروجی‌های خواسته شده برای فایل‌ها به شرح زیر است:

```
~/Desktop main ?28
❯ ./OCP-final-changed
Enter CSV filename: A42.csv
Enter cache size: 10000
Enter piece number (Blady): 1
Enter start time (relative, in seconds): 0
Enter end time (relative, in seconds): 2678385
Total Requests: 5087929
Total Hits: 1718526
Total Misses: 3369403
Cold Misses: 114526
Total Reads: 3098735
Total Writes: 1989194
Total Read Hits: 672056
Total Read Misses: 2426679
Total Write Hits: 1046470
Total Write Misses: 942724
Hit Rate: 33.7765%
```

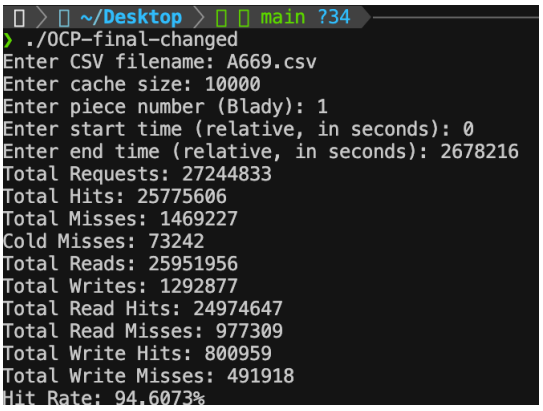
شکل ۳-۱: خروجی برنامه برای فایل A42



شکل ۳-۲: خروجی برنامه برای فایل A108



شکل ۳-۳: خروجی برنامه برای فایل A129



شکل ۳-۴: خروجی برنامه برای فایل A669

۳-۲ خروجی N-hit

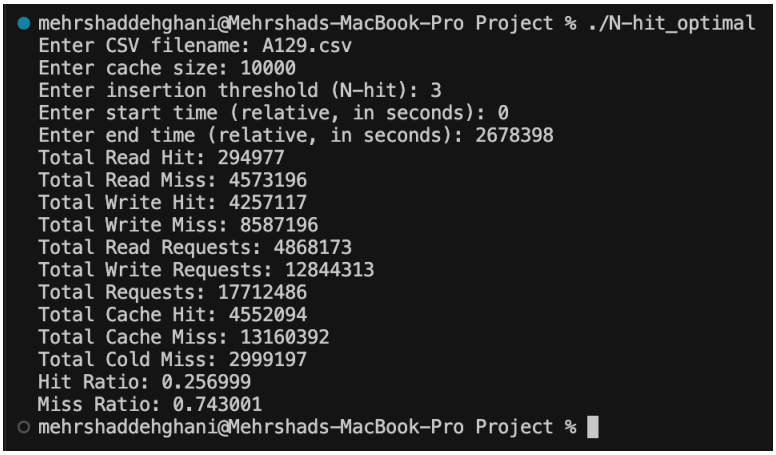
خروجی‌های خواسته شده برای فایل‌ها به شرح زیر است:

```
mehrshaddehghani@Mehrshads-MacBook-Pro Project % ./N-hit_optimal
Enter CSV filename: A42.csv
Enter cache size: 10000
Enter insertion threshold (N-hit): 3
Enter start time (relative, in seconds): 0
Enter end time (relative, in seconds): 2678385
Total Read Hit: 418021
Total Read Miss: 2680714
Total Write Hit: 890703
Total Write Miss: 1098491
Total Read Requests: 3098735
Total Write Requests: 1989194
Total Requests: 5087929
Total Cache Hit: 1308724
Total Cache Miss: 3779205
Total Cold Miss: 154348
Hit Ratio: 0.257221
Miss Ratio: 0.742779
mehrshaddehghani@Mehrshads-MacBook-Pro Project %
```

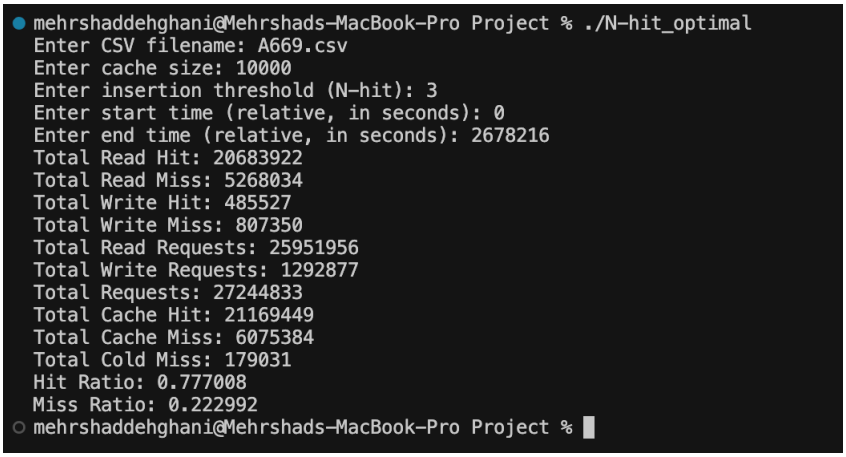
شکل ۳-۵: خروجی برنامه برای فایل A42

```
mehrshaddehghani@Mehrshads-MacBook-Pro Project % ./N-hit_optimal
Enter CSV filename: A108.csv
Enter cache size: 10000
Enter insertion threshold (N-hit): 3
Enter start time (relative, in seconds): 0
Enter end time (relative, in seconds): 2678392
Total Read Hit: 829604
Total Read Miss: 9800569
Total Write Hit: 1927889
Total Write Miss: 7647235
Total Read Requests: 10630173
Total Write Requests: 9575124
Total Requests: 20205297
Total Cache Hit: 2757493
Total Cache Miss: 17447804
Total Cold Miss: 1658530
Hit Ratio: 0.136474
Miss Ratio: 0.863526
mehrshaddehghani@Mehrshads-MacBook-Pro Project %
```

شکل ۳-۶: خروجی برنامه برای فایل A108



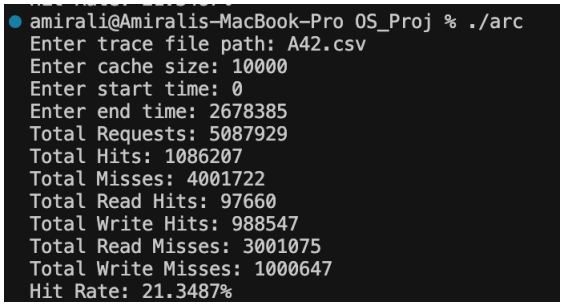
شکل ۳-۷: خروجی برنامه برای فایل A129



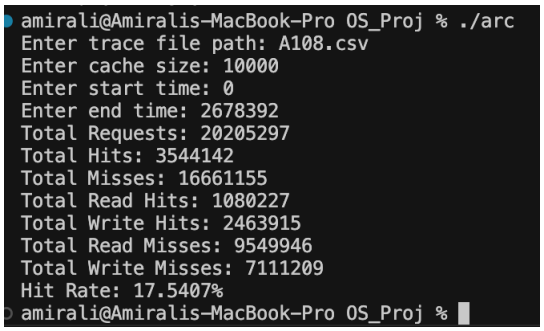
شکل ۳-۸: خروجی برنامه برای فایل A669

۳-۳ خروجی ARC

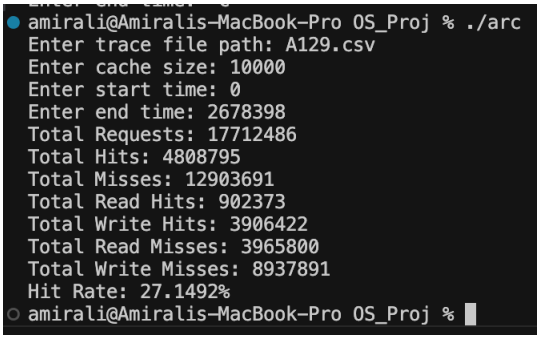
خروجی‌های خواسته شده برای فایل‌ها به شرح زیر است:



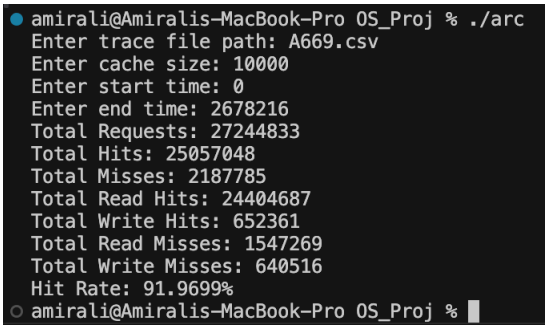
شکل ۳-۹: خروجی برنامه برای فایل A42



شکل ۳-۱۰: خروجی برنامه برای فایل A108



شکل ۳-۱۱: خروجی برنامه برای فایل A129



شکل ۳-۱۲: خروجی برنامه برای فایل A669

۳-۴ خروجی LARC

خروجی‌های خواسته شده برای فایل‌ها به شرح زیر است:

```
amirali@Amiralis-MacBook-Pro OS_Proj % ./larc
Enter trace file path: A42.csv
Enter cache size: 10000
Enter start time: 0
Enter end time: 2678385
Total Requests: 5087929
Total Hits: 1085554
Total Misses: 4002375
Total Read Hits: 97611
Total Write Hits: 987943
Total Read Misses: 3001124
Total Write Misses: 1001251
Hit Rate: 21.3359%
```

شکل ۳-۱۳: خروجی برنامه برای فایل A42

```
amirali@Amiralis-MacBook-Pro OS_Proj % ./larc
Enter trace file path: A108.csv
Enter cache size: 10000
Enter start time: 0
Enter end time: 2678392
Total Requests: 20205297
Total Hits: 3544595
Total Misses: 16660702
Total Read Hits: 1080362
Total Write Hits: 2464233
Total Read Misses: 9549811
Total Write Misses: 7110891
Hit Rate: 17.5429%
```

شکل ۳-۱۴: خروجی برنامه برای فایل A108

```
amirali@Amiralis-MacBook-Pro OS_Proj % ./larc
Enter trace file path: A129.csv
Enter cache size: 10000
Enter start time: 0
Enter end time: 2678397
Total Requests: 17712486
Total Hits: 4808479
Total Misses: 12904007
Total Read Hits: 902627
Total Write Hits: 3905852
Total Read Misses: 3965546
Total Write Misses: 8938461
Hit Rate: 27.1474%
```

شکل ۳-۱۵: خروجی برنامه برای فایل A129

```
amirali@Amiralis-MacBook-Pro OS_Proj % ./larc
Enter trace file path: A669.csv
Enter cache size: 10000
Enter start time: 0
Enter end time: 2678216
Total Requests: 27244833
Total Hits: 25054700
Total Misses: 2190133
Total Read Hits: 24407429
Total Write Hits: 647271
Total Read Misses: 1544527
Total Write Misses: 645606
Hit Rate: 91.9613%
```

شکل ۳-۱۶: خروجی برنامه برای فایل A669

۳-۵ خروجی LRU

خروجی‌های خواسته شده برای فایل‌ها به شرح زیر است:

```
D:\code\os_project\cmake-build-debug\os_project.exe
Enter CSV filename:A42.csv

Enter cache size:10000

Enter start time (relative, in seconds):0

Enter end time (relative, in seconds):2678385

Total Requests: 5087929
Total Hits: 943118
Total Misses: 4144811
Cold Misses: 154348
Total Reads: 3098735
Total Writes: 1989194
Total Read Hits: 1805
Total Read Misses: 3096930
Total Write Hits: 941313
Total Write Misses: 1047881
Hit Rate: 18.5364%
```

شکل ۳-۱۷: خروجی برنامه برای فایل A42

```
D:\code\os_project\cmake-build-debug\os_project.exe
Enter CSV filename:A108.csv

Enter cache size:10000

Enter start time (relative, in seconds):0

Enter end time (relative, in seconds):2678392

Total Requests: 20205297
Total Hits: 3468251
Total Misses: 16737046
Cold Misses: 1658530
Total Reads: 10630173
Total Writes: 9575124
Total Read Hits: 1039982
Total Read Misses: 9590191
Total Write Hits: 2428269
Total Write Misses: 7146855
Hit Rate: 17.1651%
```

شکل ۳-۱۸: خروجی برنامه برای فایل A108

```
D:\code\os_project\cmake-build-debug\os_project.exe
Enter CSV filename:A129.csv

Enter cache size:10000

Enter start time (relative, in seconds):0

Enter end time (relative, in seconds):2678398

Total Requests: 17712486
Total Hits: 4737397
Total Misses: 12975089
Cold Misses: 2999197
Total Reads: 4868173
Total Writes: 12844313
Total Read Hits: 884111
Total Read Misses: 3984062
Total Write Hits: 3853286
Total Write Misses: 8991027
Hit Rate: 26.7461%
```

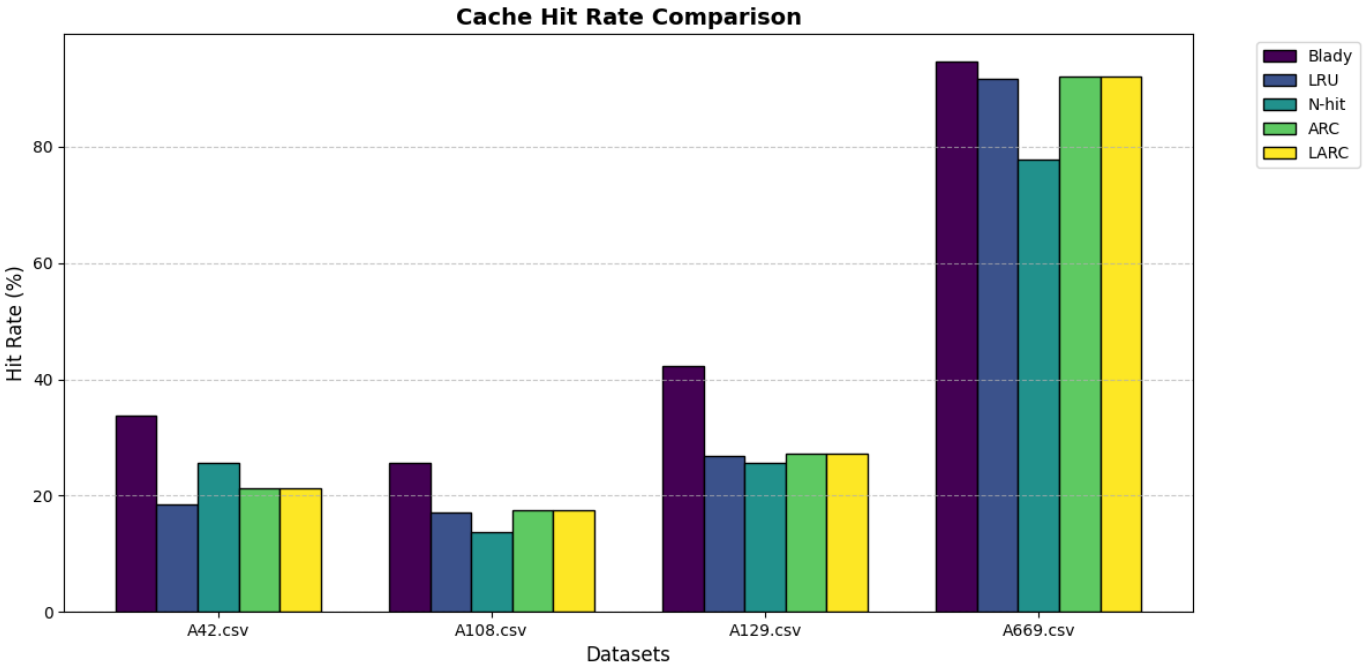
شکل ۳-۱۹: خروجی برنامه برای فایل A129

```
D:\code\os_project\cmake-build-debug\os_project.exe
Enter CSV filename:A669.csv

Enter cache size:10000

Enter start time (relative, in seconds):0

Enter end time (relative, in seconds):2678216

Total Requests: 27244833
Total Hits: 24986997
Total Misses: 2257836
Cold Misses: 179031
Total Reads: 25951956
Total Writes: 1292877
Total Read Hits: 24353813
Total Read Misses: 1598143
Total Write Hits: 633184
Total Write Misses: 659693
Hit Rate: 91.7128%
```

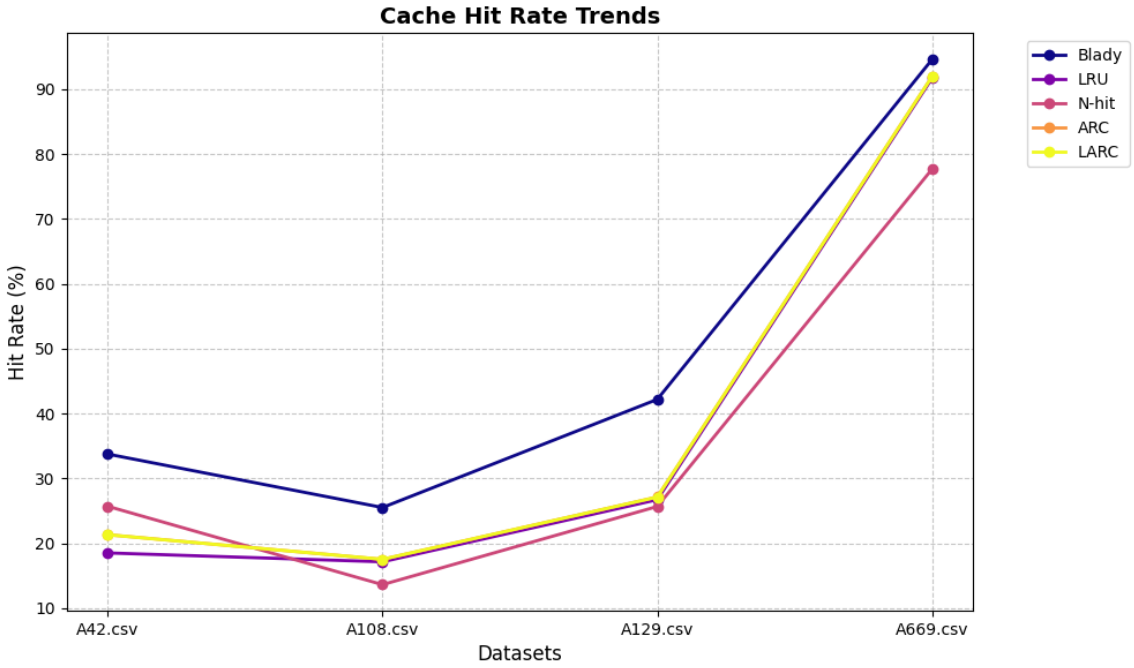
شکل ۳-۲۰: خروجی برنامه برای فایل A669

۳-۶ جمع بندی و مقایسه

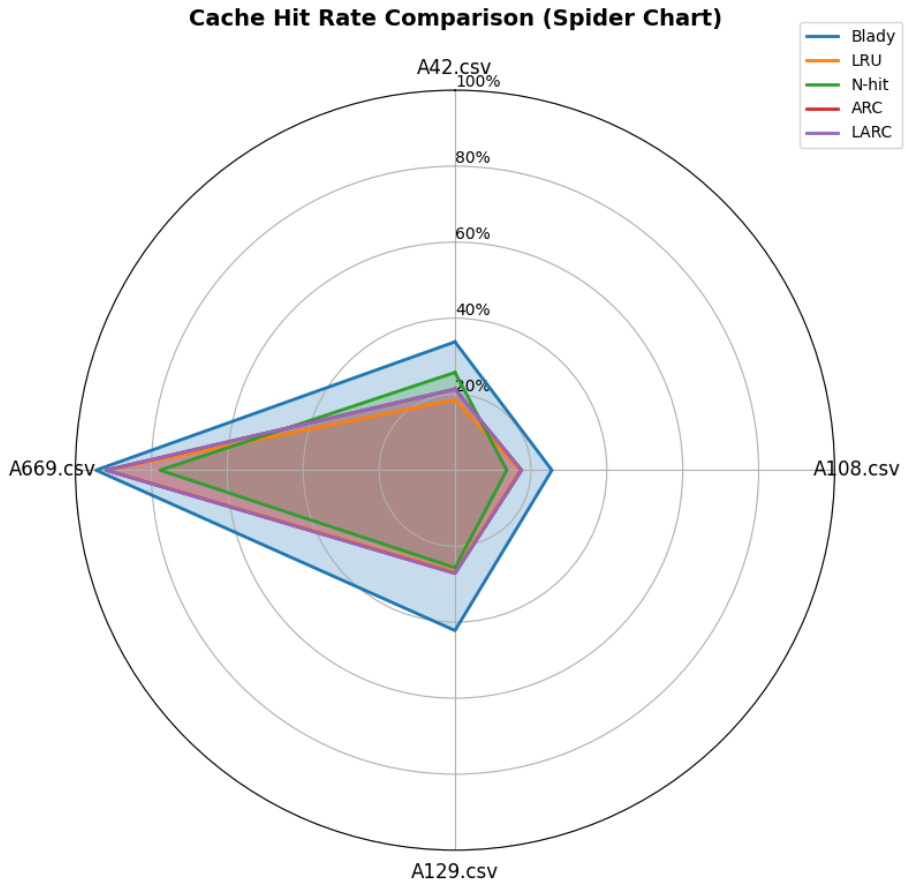
با بررسی خروجی‌ها و نرخ برخوردهای بدست آمده برای فایل‌ها و الگوریتم‌های مختلف به نمودارهای زیر می‌رسیم:



شکل ۳-۲۱: نمودار ستونی نرخ برخورد



شکل ۳-۲۲: نمودار خطی نرخ برخورد



شکل ۳-۲۳: نمودار عنکبوتی نرخ برخورد

۱. الگوریتم Blady در تمام موارد بهتر عمل می‌کند

دلیل: Blady یک الگوریتم پیشرفته یا تطبیقی است که استراتژی کش خود را به‌طور پویا بر اساس الگوهای دسترسی تنظیم می‌کند. این الگوریتم از تکنیک‌های پیش‌بینی برای شناسایی الگوها در داده‌ها استفاده کند، که به آن امکان می‌دهد ورودی‌های مفید بیشتری را در کش نگه دارد.

۲. الگوریتم LRU در مجموعه‌داده‌های کوچک ضعیف عمل می‌کند LRU در مجموعه‌داده‌های A42، A108 و A129 کمترین نرخ برخورد را دارد.

دلیل: LRU آیتم‌هایی که کمترین استفاده اخیر را داشته‌اند، از کش خارج می‌کند. این الگوریتم برای بارهای کاری با locality زمانی قوی خوب عمل می‌کند، اما اگر الگوهای دسترسی نامنظم باشند یا locality زمانی ضعیف باشد، عملکرد آن ضعیف خواهد بود. مجموعه‌داده‌های کوچک ممکن است الگوهای دسترسی تکراری کافی برای بهره‌برداری از LRU نداشته باشند.

۳. الگوریتم N-hit در بیشتر موارد ضعیف است N-hit در مجموعه‌داده‌های A108 و A129

کمترین نرخ برخورد را دارد و در A42 و A669 عملکرد متوسطی دارد.

دلیل: الگوریتم‌های N-hit معمولاً نیاز دارند که یک آیتم چندین بار مورد دسترسی قرار گیرد تا در کش نگه‌داری شود. اگر بار کاری الگوهای دسترسی تکراری نداشته باشد یا دارای درجه بالایی از تصادفی بودن باشد، این الگوریتم‌ها عملکرد ضعیفی خواهند داشت.

۴. الگوریتم‌های ARC و LARC عملکرد مشابهی دارند ARC و LARC در تمام مجموعه‌داده‌ها نرخ برخورد تقریباً یکسانی دارند.

دلیل: ARC و LARC الگوریتم‌های پیشرفته‌ای هستند که بین دو عامل "تازگی" و "تکرار دسترسی" تعادل برقرار می‌کنند. این الگوریتم‌ها به تغییرات در بار کاری تطبیق می‌یابند، که باعث می‌شود در مجموعه‌داده‌های مختلف عملکرد قوی‌تری داشته باشند. LARC ممکن است مکانیزم‌های یادگیری اضافی داشته باشد، اما در این مورد به نظر نمی‌رسد مزیت قابل‌توجهی نسبت به ARC ارائه دهد.

۵. همه الگوریتم‌ها در مجموعه‌داده A669 نرخ برخورد بالایی (بالای ۹۰٪) دارند.

دلیل: این مجموعه‌داده احتمالاً locality زمانی قوی یا الگوهای دسترسی تکراری دارد، که باعث می‌شود حتی الگوریتم‌های ساده‌تر مانند LRU نیز به‌راحتی بتوانند آیتم‌های مفید را در کش نگه‌دارند.