LAPORAN

ARSITEKTUR DAN ORGANISASI KOMPUTER II

CACHE MEMORY (PRINSIP LOKALITAS, STRUKTUR ORGANISASI, DAN MANAJEMEN KONSISTENSI)

Oleh:

Muhammad Umar Mansyur (2020520018)

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNIK

UNIVERSITAS MADURA

2024

KATA PENGANTAR

Dalam era digital yang terus berkembang pesat, kecepatan dan efisiensi akses data telah menjadi faktor kritis dalam menentukan kinerja sistem komputasi. Di tengah tuntutan yang semakin tinggi untuk pemrosesan data yang cepat dan responsif, teknologi cache muncul sebagai solusi inovatif yang memainkan peran vital dalam arsitektur komputer modern. Cache, yang juga dikenal sebagai look-aside atau slave memory, berfungsi sebagai jembatan antara penyimpanan utama yang relatif lambat dan komponen sistem yang membutuhkan akses data dengan latensi rendah. Konsep ini didasarkan pada prinsip lokalitas referensi, yang menggambarkan kecenderungan aplikasi untuk mengakses sejumlah kecil data secara berulang dalam periode waktu tertentu.

Dalam laporan ini, penilis mencoba untuk menjelajahi dunia teknologi cache, mulai dari prinsip-prinsip dasarnya hingga implementasi kompleks yang menopang sistem komputasi modern. Hal ini mungkin akan bertentang dengan modul pembelajaran namun dalam tanda kutip penulisan laporan ini dimaksudkan bukan untuk memberikan kritik namun untuk menyempurnakan aspek pengetahuan dari sisi memeory cache. Mengingat pemahaman mendalam tentang teknologi cache tidak hanya penting bagi para insinyur perangkat keras dan pengembang perangkat lunak, tetapi juga bagi siapa pun yang tertarik pada peningkatan kinerja sistem komputasi terlebih kepada mahasiswa yang memiliki ide dan gagasan yang kritis.

Laporan ini bertujuan untuk memberikan pengalaman dan pemahaman yang mendalam mengenai teknologi memori cache dengan cara yang jelas dan terstruktur. Dengan memahami berbagai aspek dari teknologi cache, mulai dari prinsip dasar hingga aplikasi dan tantangan yang dihadapinya, diharapkan pembaca dapat mengaplikasikan pengetahuan ini untuk meningkatkan kinerja sistem komputasi dan mengatasi masalah yang terkait dengan efisiensi akses data.

Sumenep, 18 Juli 2021

Muhammad Umar Mansyur

DAFTAR ISI

DAFTAR GAMBAR

BAB I
PENDAHULUAN

1.1 Latar Belakang

Dalam dunia komputasi modern, kecepatan dan efisiensi akses data telah menjadi kebutuhan utama yang mendasari desain berbagai sistem perangkat keras dan perangkat lunak(Ramanujan et al., 2020). Salah satu inovasi utama dalam memenuhi kebutuhan ini adalah teknologi *cache*, yang dikenal dengan nama *look-aside* atau *slave memory*(Roy, 2022). *Cache* berfungsi sebagai perantara antara penyimpanan utama yang lebih lambat, seperti hard disk atau memori utama, dan prosesor atau aplikasi yang membutuhkan akses data cepat(Tran et al., 2022). Dengan menyimpan salinan data yang sering diakses, *cache* memungkinkan sistem untuk mengurangi waktu yang diperlukan untuk mendapatkan data tersebut dari sumber penyimpanan yang lebih lambat(Tran et al., 2022). Konsep ini berakar pada prinsip lokalitas referensi, yang menunjukkan bahwa aplikasi cenderung mengakses sejumlah kecil data dalam rentang waktu yang relatif singkat. Pemahaman mendalam tentang prinsip-prinsip ini sangat penting karena membantu merancang sistem yang dapat memanfaatkan cache secara efektif untuk meningkatkan kinerja secara keseluruhan(Roy, 2022).

Teknologi *cache* beroperasi berdasarkan beberapa prinsip kunci yang memungkinkan peningkatan kinerja sistem. Salah satu prinsip dasar adalah penggunaan struktur penyimpanan yang lebih cepat daripada penyimpanan utama. Misalnya, dalam komputer, *cache prosesor* biasanya dibangun dengan SRAM (*Static RAM*), yang memiliki kecepatan akses yang jauh lebih tinggi dibandingkan dengan DRAM (*Dynamic RAM*) yang digunakan sebagai memori utama(Wallach, 2022). Dengan menyimpan data yang sering diakses di *SRAM*, prosesor dapat mengakses informasi dengan *latensi* yang jauh lebih rendah, mempercepat proses komputasi dan eksekusi program. Selain itu, *cache* juga diterapkan dalam berbagai bentuk penyimpanan lain, seperti *disk cache* yang menggunakan memori cepat untuk menyimpan data yang sering diambil dari hard *disk*, atau *cache* jaringan yang menyimpan salinan data dari *server web* untuk mempercepat akses bagi pengguna

akhir. Ini menunjukkan betapa pentingnya *cache* dalam meningkatkan efisiensi sistem di berbagai level penyimpanan(Li et al., 2021).

Dari segi implementasi, teknologi *cache* dapat dibagi menjadi berbagai kategori berdasarkan komponen dan metode manajemennya. *Cache* perangkat keras, seperti *SRAM* dan *DRAM* yang digunakan untuk menyimpan data sementara, dirancang untuk memberikan akses data yang cepat dengan biaya yang lebih tinggi per bit dibandingkan dengan penyimpanan utama(Fang & Zhou, 2022). Di sisi lain, *cache* perangkat lunak, termasuk *cache file* sistem dan *cache web*, mengandalkan *algoritma* yang kompleks untuk menentukan data mana yang harus disimpan berdasarkan pola akses yang terjadi. *Cache* perangkat lunak sering kali menggunakan *heuristik* untuk mengelola data yang disimpan dan berusaha memaksimalkan manfaatnya berdasarkan analisis pola akses data dari aplikasi yang berjalan. Perbedaan ini menggarisbawahi kompleksitas desain dan implementasi *cache* yang harus dipertimbangkan dalam berbagai konteks sistem komputasi(Sorin et al., 2022).

Memahami prinsip-prinsip dan teknik pengelolaan cache sangat penting karena hal ini berdampak signifikan pada kinerja dan efisiensi sistem komputasi. *Heuristik* untuk manajemen konten, konsistensi data, dan organisasi cache memainkan peran penting dalam menentukan efektivitas *cache* dalam mempercepat akses data dan mengelola memori. Desain yang baik melibatkan pemilihan strategi yang tepat untuk organisasi *cache*, kebijakan pengambilan dan penghapusan data, serta mekanisme untuk menjaga konsistensi data antara cache dan penyimpanan utama. Dengan memahami berbagai aspek ini, pengembang dapat merancang sistem yang lebih cepat dan efisien, mampu mengatasi tantangan dalam akses data dan pengelolaan memori dengan lebih baik, serta memenuhi kebutuhan pengguna dan aplikasi yang semakin kompleks.

1.2 Rumusan masalah

Adapun rumusan masalah dari latar belakang di atas adalah sebagai berikut:

1. Bagaimana prinsip lokalitas referensi mempengaruhi performa sistem cache dalam akses data?

2. Apa jenis-jenis organisasi logis cache yang ada, dan bagaimana masing-masing mempengaruhi efektivitas pengelolaan cache dalam sistem komputer?
3. Bagaimana sistem cache mengelola konten dan konsistensi data untuk memastikan kinerja sistem yang optimal dalam berbagai aplikasi?

1.3 Tujuan Penulisan

Adapun tujuan penulisan dari rumusan masalah di atas adalah sebagai berikut:

1. Untuk mengetahui bagaimana prinsip lokalitas referensi mempengaruhi performa sistem cache dalam akses data.
2. Untuk mengetahui apa jenis-jenis organisasi logis cache yang ada, dan bagaimana masing-masing mempengaruhi efektivitas pengelolaan cache dalam sistem komputer.
3. Untuk mengetahui bagaimana sistem cache mengelola konten dan konsistensi data untuk memastikan kinerja sistem yang optimal dalam berbagai aplikasi.

BAB II
PEMBAHASAN

2.1 Pengaruh Prinsip Lokalitas Referensi terhadap Perfroma Sistem Cache

Prinsip lokalitas referensi memainkan peran krusial dalam menentukan efisiensi dan performa sistem cache. *Lokalitas* referensi merujuk pada pola akses memori yang berulang atau terlokalisasi dalam waktu dan ruang, yang dapat dimanfaatkan untuk meningkatkan kinerja sistem cache(Tan et al., 2020).

2.1.1 Temporal Locality

Temporal locality mengacu pada kecenderungan sebuah program untuk mengakses data yang sama berulang kali dalam waktu dekat. Fenomena ini sangat penting dalam desain cache karena memungkinkan cache untuk menyimpan data yang baru-baru ini diakses dan kemungkinan besar akan diakses kembali dalam waktu singkat. Dalam konteks cache, ini diterapkan melalui teknik seperti demand-fetch, di mana data yang diminta oleh program diambil dari memori utama dan disimpan di cache. Dengan menyimpan data ini, cache mengurangi waktu akses untuk permintaan data berikutnya, meningkatkan efisiensi sistem secara keseluruhan(Ivester & Sahasrabudhe, 2021; Tan et al., 2020).

Namun, efektivitas temporal locality bergantung pada ukuran cache. Jika cache terlalu kecil, data yang sering diakses dapat dengan cepat mengisi cache dan menyebabkan penggantian data yang kurang sering diakses. Hal ini dapat menyebabkan cache thrashing, di mana data yang sering diakses terus menerus diganti, mengurangi manfaat dari caching itu sendiri. Dengan demikian, ukuran cache yang memadai sangat penting untuk memaksimalkan manfaat temporal locality(Kazakov, 2023).

2.1.2 Spatial Locality

Spatial locality merujuk pada kecenderungan untuk mengakses data yang berdekatan dalam ruang memori. Pola ini sering terlihat dalam operasi array dan struktur data yang memproses elemen berdekatan secara berurutan. Dalam sistem

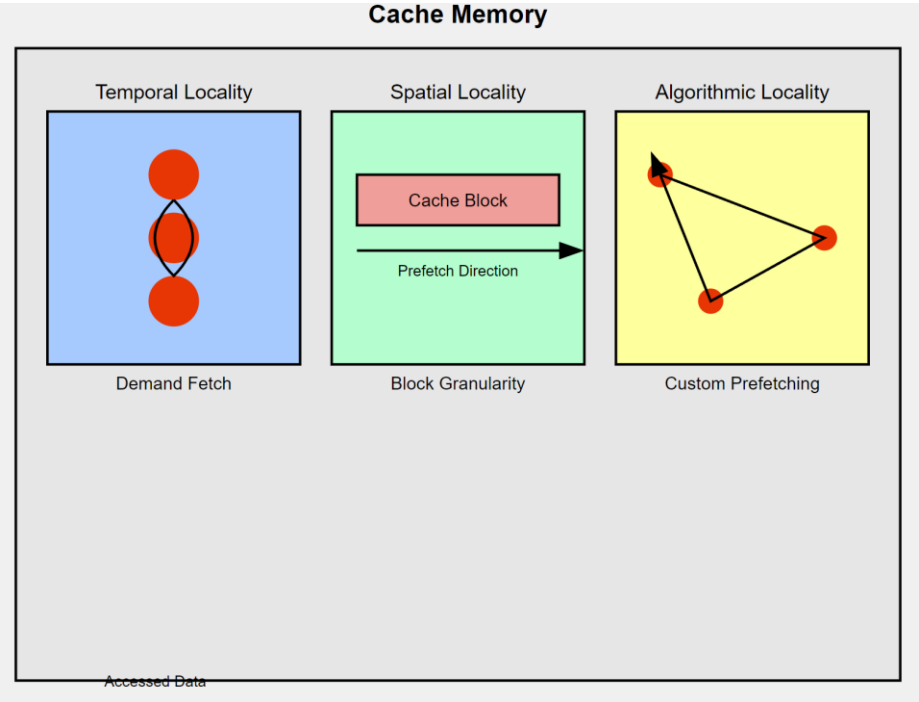
cache, spatial locality dieksploitasi melalui teknik seperti cache block granularity dan prefetching. Cache block granularity melibatkan pengambilan data dalam unit yang lebih besar daripada ukuran data yang diakses, sehingga ketika satu data item diambil, data di sekitarnya juga diambil dan disimpan di cache(Tan et al., 2020).

Prefetching, di sisi lain, adalah teknik aktif di mana sistem cache memprediksi data yang akan diakses berikutnya dan memuatnya ke dalam cache sebelum data tersebut benar-benar diminta. Prefetching dapat meningkatkan performa dengan meminimalkan waktu tunggu untuk data yang akan datang, tetapi keefektifannya tergantung pada akurasi algoritma prediksi. Jika program memiliki pola akses yang kompleks, teknik prefetching yang lebih canggih mungkin diperlukan untuk memanfaatkan spatial locality secara optimal(Kazakov, 2023).

2.1.3 Algorithmic Locality

Selain temporal dan spatial locality, algorithmic locality adalah fenomena baru yang terlihat pada algoritma yang mengakses data dalam pola yang sangat teratur tetapi tidak dapat dikategorikan secara klasik sebagai temporal atau spatial locality. Misalnya, dalam aplikasi grafis 3D atau simulasi sirkuit, data sering diakses dalam urutan tertentu, tetapi tidak secara bersamaan atau dalam jarak fisik yang dekat. Meskipun data tidak menunjukkan temporal atau spatial locality yang signifikan, pola akses yang teratur memungkinkan prefetching dan cache optimizations khusus yang didasarkan pada pemahaman mendalam tentang perilaku algoritma tersebut(Srivastava & Singh, 2022).

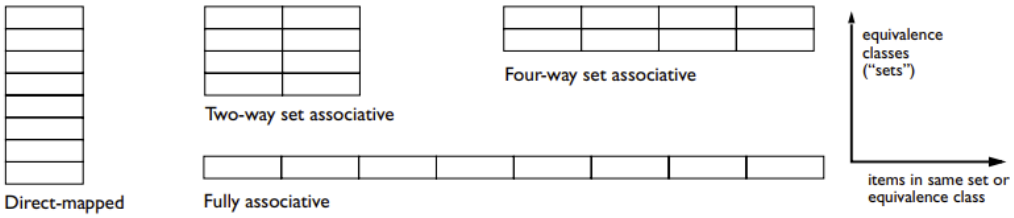
Secara keseluruhan, pemanfaatan prinsip lokalitas referensi baik temporal, spatial, maupun algorithmic dapat secara signifikan meningkatkan performa sistem cache dengan mengurangi waktu akses data dan meminimalkan latensi. Namun, tantangan terletak pada desain dan implementasi cache yang dapat menyesuaikan diri dengan pola akses data yang beragam dan sering berubah, sehingga memastikan penggunaan cache yang efisien dan efektif(Bhavikatti, 2023).



Gambar 2. 1 Ilustrasi Tipe Lokalitas dan Optimasi cache

Pada gambar 2.1 Temporal Locality ditunjukkan oleh lingkaran merah yang berulang kali diakses, dengan panah yang menggambarkan akses berulang ke data yang sama. Optimasi yang diterapkan untuk jenis lokalitas ini adalah demand fetch, di mana data yang sering diakses disimpan di cache untuk akses cepat di masa depan. Spatial Locality direpresentasikan oleh blok cache yang lebih besar, menunjukkan bahwa data yang berdekatan secara fisik cenderung diakses bersama. Panah pada bagian ini menunjukkan arah prefetch, mengantisipasi akses data berikutnya berdasarkan kedekatan spasial. Sementara itu, Algorithmic Locality digambarkan dengan pola akses yang teratur namun tersebar, menunjukkan data yang diakses dalam urutan yang dapat diprediksi tetapi tidak berdekatan secara fisik. Optimasi yang diterapkan untuk jenis lokalitas ini adalah custom prefetching, yang dirancang khusus berdasarkan pemahaman tentang pola akses algoritma. Ilustrasi ini mendemonstrasikan bagaimana sistem cache modern perlu mengakomodasi berbagai jenis lokalitas untuk mengoptimalkan kinerja, memungkinkan pengembangan strategi caching yang lebih efektif dan adaptif, sehingga meningkatkan efisiensi dan kinerja sistem secara keseluruhan(Paschos et al., 2020).

2.2 Jenis-Jenis Organisasi Logis Cache dan Dampaknya terhadap Pengelolaan Cache



Gambar 2. 2 Ilustrasi Jenis-Jenis Organisasi Logis Cache

Pada gambar 2.2 terdapat empat jenis organisasi cache yang berbeda, masing-masing menggunakan delapan blok cache. Organisasi pertama adalah direct mapped, di mana setiap blok berada dalam set tersendiri, ditunjukkan oleh delapan baris terpisah. Ini memberikan akses cepat namun terbatas dalam fleksibilitas penempatan data. Selanjutnya, fully associative ditampilkan sebagai satu baris panjang yang berisi semua delapan blok, menandakan bahwa setiap blok dapat ditempatkan di mana saja dalam cache, memberikan fleksibilitas maksimal namun dengan kompleksitas pencarian yang lebih tinggi. Dua jenis set-associative juga ditampilkan: two-way set associative membagi cache menjadi empat set dengan dua blok per set, sementara four-way set associative membaginya menjadi dua set dengan empat blok per set. Organisasi set-associative ini menawarkan keseimbangan antara fleksibilitas penempatan data dan efisiensi pencarian. Sumbu vertikal menunjukkan kelas ekuivalensi atau *set*, sedangkan sumbu horizontal menggambarkan item dalam set atau kelas ekuivalensi yang sama. Gambar ini secara efektif mendemonstrasikan spektrum organisasi cache, mulai dari yang paling terbatas namun sederhana (direct-mapped) hingga yang paling fleksibel namun kompleks (fully associative), dengan opsi set-associative sebagai kompromi di antara kedua ekstrem tersebut(Roy, 2022).

Secara umum Jenis Oraganisasi Logis cache dibedakan menjadi 3 yaitu sebagai berikut:

2.2.1 Direct Mapped (Pemetaan Langsung)

Cache direct mapped, atau pemetaan langsung, adalah salah satu metode organisasi cache yang paling sederhana dan efisien dalam hal implementasi perangkat

keras. Dalam cache direct mapped, setiap blok dalam cache hanya dapat menyimpan data dari satu blok memori tertentu. Ini dicapai dengan membagi alamat memori menjadi dua komponen utama: ID blok dan offset byte. ID blok menentukan lokasi cache yang tetap di mana data akan disimpan, sedangkan offset byte menunjukkan posisi byte spesifik dalam blok tersebut. Dengan cara ini, setiap blok memori dari backing store dipetakan secara unik ke satu blok cache, sehingga setiap kali data dari blok memori tersebut diakses, CPU tahu persis di mana mencarinya dalam cache.

Keuntungan utama dari metode direct mapped adalah kesederhanaan dan kecepatan aksesnya. Karena setiap blok memori hanya dapat dipetakan ke satu lokasi cache, waktu pencarian cache menjadi sangat cepat, memungkinkan akses data yang efisien. Hal ini juga membuat implementasi perangkat keras lebih sederhana dibandingkan dengan metode organisasi cache lainnya seperti fully associative atau set-associative. Namun, kesederhanaan ini datang dengan beberapa keterbatasan. Salah satu kelemahan utama dari cache direct-mapped adalah tingginya kemungkinan terjadinya konflik atau *collision*. Jika beberapa blok memori yang berbeda dipetakan ke lokasi cache yang sama dan sering diakses bergantian, data yang ada dalam cache akan sering digantikan (eviction). Hal ini dapat menyebabkan peningkatan *cache misses*, yang pada akhirnya mengurangi kinerja sistem secara keseluruhan.

Contoh praktis dari masalah ini adalah ketika program sering mengakses data yang terletak di alamat memori yang berulang, yang semuanya dipetakan ke blok cache yang sama. Dalam situasi ini, cache direct-mapped dapat mengalami performa yang lebih rendah dibandingkan dengan metode lain. Meski demikian, untuk banyak aplikasi di mana akses memori lebih terdistribusi secara merata, cache direct-mapped tetap menjadi pilihan yang efisien dan efektif. Oleh karena itu, pemilihan metode ini harus mempertimbangkan pola akses memori dari aplikasi yang akan dijalankan, serta kebutuhan kinerja dan efisiensi perangkat keras.

2.2.2 Fully Assosiative(Asosiatif Penuh)

Cache fully associative, atau asosiatif penuh, adalah salah satu metode organisasi cache yang menawarkan fleksibilitas tertinggi dalam hal penyimpanan data. Dalam cache fully associative, tidak ada pembatasan mengenai lokasi penyimpanan

data dari blok memori. Setiap blok memori dari backing store dapat disimpan di sembarang blok dalam cache. Dengan kata lain, seluruh cache terdiri dari satu set besar yang mencakup semua blok cache, sehingga data dapat disimpan di mana saja dalam cache tanpa memperhatikan ID blok tertentu(Gouk et al., 2022).

Keuntungan utama dari cache fully associative adalah kemampuan untuk mengurangi kemungkinan konflik atau *collision* secara signifikan. Karena data dapat ditempatkan di sembarang lokasi dalam cache, risiko bahwa dua atau lebih blok memori akan dipetakan ke blok cache yang sama berkurang drastis. Ini membuat cache fully associative sangat efektif dalam menangani situasi di mana akses memori tidak terdistribusi secara merata atau memiliki pola yang tidak terduga. Setiap kali CPU mengakses data, pencarian dilakukan di seluruh cache untuk menemukan apakah data tersebut sudah ada dalam cache atau tidak(Gouk et al., 2022).

Namun, fleksibilitas ini datang dengan biaya yang cukup tinggi dalam hal kompleksitas dan waktu akses. Karena pencarian dilakukan di seluruh blok cache, waktu yang dibutuhkan untuk menemukan data yang relevan lebih lama dibandingkan dengan cache direct-mapped atau set-associative. Hal ini memerlukan perangkat keras tambahan untuk mempercepat proses pencarian, seperti penggunaan komparator paralel atau algoritma pencarian yang efisien. Selain itu, implementasi perangkat keras cache fully associative menjadi lebih kompleks dan mahal karena harus mendukung kemampuan pencarian di seluruh blok cache secara simultan(Gouk et al., 2022).

Cache fully associative juga memiliki implikasi dalam hal manajemen memori dan efisiensi energi. Karena setiap blok cache harus diperiksa setiap kali ada akses memori, penggunaan daya bisa lebih tinggi, yang menjadi pertimbangan penting dalam desain sistem yang hemat energi. Meskipun demikian, untuk aplikasi di mana performa akses memori sangat kritis dan pola akses data sulit diprediksi, cache fully associative bisa menjadi pilihan yang optimal. Keputusan untuk menggunakan cache fully associative harus mempertimbangkan kebutuhan aplikasi spesifik, pola akses memori, dan batasan perangkat keras serta anggaran energi yang tersedia(Farshin et al., 2020).

2.2.3 Set Associative

Cache set associative adalah metode organisasi *cache* yang menggabungkan elemen dari direct mapped dan fully associative, menawarkan keseimbangan antara fleksibilitas dan efisiensi. Dalam organisasi ini, *cache* dibagi menjadi beberapa *set*, dan setiap *set* terdiri dari beberapa blok *cache*. Data dari blok memori dipetakan ke satu set tertentu berdasarkan bagian dari alamat memori yang dikenal sebagai *ID set*, namun data tersebut dapat disimpan di sembarang *blok* dalam *set* tersebut. Dengan demikian, *set associative cache* memungkinkan beberapa *blok* memori yang berbeda untuk berada dalam satu *set*, mengurangi kemungkinan konflik yang sering terjadi dalam *cache direct mapped*(Agarwal, 2021).

Keuntungan utama dari *set associative cache* adalah kemampuan untuk mengurangi konflik tanpa memerlukan kompleksitas penuh dari *fully associative cache*. Dengan memiliki lebih dari satu *blok* per *set*, data yang sering diakses bersama tetapi berada pada blok memori yang berbeda dapat disimpan dalam set yang sama, mengurangi frekuensi *cache misses*. Ini memberikan fleksibilitas yang lebih tinggi dibandingkan *direct mapped cache*, sementara tetap mempertahankan waktu akses yang lebih cepat daripada *fully associative cache* karena hanya perlu mencari dalam satu *set*, bukan seluruh *cache*(Leidel & Murphy, 2020).

Set associative cache dapat diimplementasikan dengan berbagai derajat asosiatif, yang dikenal sebagai *n-way set associative*. Misalnya, dalam *2-way set associative cache*, setiap siet berisi dua *blok cache*. Semakin tinggi derajat asosiatif, semakin besar fleksibilitasnya dalam menyimpan data, tetapi juga semakin kompleks implementasinya. Penggunaan *set associative cache* biasanya merupakan kompromi yang baik antara biaya, kinerja, dan kompleksitas. Dalam banyak sistem, tingkat asosiatif yang umum digunakan adalah *2-way* atau *4-way*, memberikan keseimbangan yang optimal antara kecepatan akses dan pengurangan konflik(Chong & Liu, 2023).

Namun, meskipun lebih fleksibel dan efisien, *set associative cache* tetap memiliki beberapa tantangan. Salah satunya adalah kebutuhan untuk menentukan blok mana dalam set yang akan digantikan ketika cache penuh. Berbagai algoritma penggantian, seperti *Least Recently Used (LRU)*, digunakan untuk mengelola ini, yang

menambah kompleksitas sistem. Selain itu, meskipun konflik berkurang, tetap ada kemungkinan terjadinya *conflict misses* jika beberapa blok memori sering dipetakan ke *set* yang sama(Chong & Liu, 2023).

Secara keseluruhan, *set associative cache* menawarkan kompromi yang menarik antara kecepatan akses dan kompleksitas implementasi. Ini menjadikannya pilihan yang populer dalam desain *cache modern*, memungkinkan pengelolaan data yang lebih efisien dan meningkatkan kinerja sistem secara keseluruhan. Pemilihan derajat asosiatif yang tepat sangat penting dan harus disesuaikan dengan kebutuhan spesifik aplikasi serta pola akses memori yang diharapkan(Chong & Liu, 2023).

2.3 Manajemen Konten dan Konsistensi Data dalam Sistem Cache

Manajemen konsistensi terdiri dari tiga tanggung jawab utama:

1. Menjaga cache tetap konsisten dengan dirinya sendiri
2. Menjaga cache tetap konsisten dengan backing store
3. Menjaga cache tetap konsisten dengan cache lain

Dalam banyak kasus, tujuan-tujuan ini dapat dicapai dengan cara yang sederhana. Namun, dalam banyak kasus, pemilihan organisasi logis dapat membuat semuanya menjadi sangat rumit(Purnal et al., 2021).

2.3.1 Menjaga Cache Tetap Konsisten dengan Dirinya Sendiri

Cache harus memastikan tidak ada dua salinan data yang sama. Cache dapat disimpan dilokasi yang berbeda asalkan memiliki nilaiyang sama. Karena biasanya tidak menjadi masalah jika cache diindeks dan ditandai menggunakan alamat fisik, yaitu alamat yang sesuai dengan memori utama (backing store). Namun, banyak sistem operasi memungkinkan dua alamat virtual yang tidak terkait untuk memetakan ke alamat fisik yang sama. Namun hal tersebut menciptakan masalah yang disebut *sinonim*. Sinonim terjadi ketika data yang sama dapat diakses melalui dua alamat virtual berbeda, dan jika cache diindeks menggunakan alamat virtual, kedua alamat tersebut bisa memetakan ke dua lokasi berbeda dalam cache. Ini bisa menyebabkan inkonsistensi ketika salah satu data diubah(Hyden et al., 2020).

Untuk mengatasi masalah ini, cache sebaiknya diindeks menggunakan alamat fisik yang sesuai dengan memori utama. Kemudian mengidentifikasi data secara unik, sehingga tidak ada masalah jika data yang sama diakses melalui alamat virtual berbeda. Selain itu, menggunakan tag yang sesuai dengan memori utama juga menyelesaikan masalah lain. Jika cache diatur secara set-asosiatif (di mana satu set dapat berisi beberapa blok), dua sinonim bisa memetakan ke set yang sama tetapi disimpan di blok berbeda, menciptakan dua salinan data berbeda dalam cache. Dengan menggunakan tag yang sesuai dengan memori utama dan mengindeks cache secara fisik, masalah ini dapat dihindari(Favor & Srinivasan, 2023).

2.3.2 Menjaga Cache Tetap Konsisten dengan Backing Store

Nilai yang disimpan di *backing store* (memori utama) harus selalu diperbarui mengikuti perubahan apa pun yang dilakukan pada versi yang disimpan di cache. Pembaruan ini dilakukan untuk memastikan bahwa permintaan baca ke versi di backing store, misalnya dari prosesor lain, mengembalikan nilai terbaru yang disimpan di cache. Dua mekanisme umum yang digunakan dalam cache untuk memenuhi tanggung jawab ini adalah kebijakan *write back* dan *write through*(Pulkit et al., 2023).

Pada kebijakan *write-through*, data di *backing store* diperbarui segera setelah data ditulis ke *cache*. Ini menawarkan keuntungan dengan memiliki jendela waktu yang sangat kecil di mana *backing store* memiliki salinan data yang usang (*stale*). Namun, kebijakan ini memiliki kerugian biaya yang tinggi karena setiap penulisan ke *cache* juga harus ditulis ke *backing store*, yang biasanya tidak dapat menyediakan *bandwidth* yang sama dengan *cache*(Wang et al., 2024).

Sebaliknya, kebijakan *write-back* memanfaatkan prinsip *locality*. Ketika sebuah blok ditulis, kemungkinan blok tersebut akan ditulis lagi dalam waktu dekat, sehingga tidak efisien untuk mengirimkan kedua nilai tersebut ke *backing store*. Cache *write-back* melacak blok-blok yang telah ditulis (*dirty*) dan hanya menulis isinya ke memori utama saat terjadi penggantian blok. Hal ini secara *signifikan* menghemat *bandwidth* ke *backing store*.

Untuk mengatasi kelemahan dari kebijakan *write-through*, sistem sering kali menyediakan memori cepat tambahan yang secara fisik adalah bagian dari *cache*, tetapi secara logis adalah bagian dari *backing store*. Pilihan populer untuk memori cepat ini adalah buffer penulisan *FIFO* (*first-in first-out*), buffer penulisan *koalesen*, atau *buffer* penulisan yang dapat dicari (*write cache*). Tujuan dari perangkat keras ini adalah untuk memisahkan sifat *bursty* dari operasi penulisan mikroprosesor dari *backing store* dan mengalirkan data yang ditulis ke *backing store* dengan kecepatan yang dapat ditangani(Sorin et al., 2022).

Kerugian utama dari *cache write-back* adalah interaksinya dengan prosesor lain dan proses lain. Dalam organisasi *multiprosesor*, jendela waktu yang panjang di mana *backing store* dapat memiliki nilai yang salah untuk data tertentu meningkatkan kemungkinan bahwa prosesor lain yang berbagi *backing store* akan membaca data yang salah tersebut. Selain itu, masalah sinonim dalam sistem *memori virtual* dapat memperumit situasi ini. Misalnya, ketika sebuah proses menulis ke ruang alamatnya, keluar, dan sebagian dari ruang datanya digunakan kembali oleh proses lain, tidak ada yang mencegah *cache write-back* yang diindeks secara virtual untuk menimpa ruang alamat proses baru ketika blok-blok *cache* tersebut akhirnya diganti. Bahkan jika sistem operasi membersihkan halaman data, data lama yang ditulis dari proses pertama masih dapat ditemukan di cache jika sistem operasi menggunakan alamat virtual yang berbeda atau pengenal ruang alamat yang berbeda dari proses asli(Sorin et al., 2022).

2.3.3 Menjaga Cache Tetap Konsisten dengan Cache Lain

Dalam sistem komputer modern, terutama yang menggunakan *multiprosesor*, menjaga konsistensi antara beberapa *cache* menjadi tantangan penting. Setiap *prosesor* mungkin memiliki *cache* sendiri, sehingga memastikan bahwa semua *cache* tetap sinkron sangat penting untuk menjaga integritas data. Terdapat beberapa tanggung jawab dan mekanisme yang perlu diimplementasikan untuk menjaga konsistensi ini. Konsistensi antar *cache* berarti setiap salinan data di berbagai *cache* harus selalu memiliki nilai yang sama. Jika satu cache memperbarui nilai suatu data, *cache* lain yang menyimpan salinan data tersebut juga harus diperbarui. Selain itu, *cache* harus tetap konsisten dengan memori utama atau *backing store*, yang berarti

bahwa setiap perubahan yang dibuat di salah satu *cache* harus tercermin di memori utama dan, pada gilirannya, di *cache* lain(Nagarajan et al., 2020).

Untuk menjaga konsistensi antara beberapa *cache*, digunakan beberapa protokol konsistensi *cache*, di antaranya adalah *protokol snooping* dan *protokol direktori*. Dalam *protokol snooping*, semua *cache* mengawasi *bus* memori untuk operasi baca/tulis yang relevan. Ketika satu cache memperbarui data, informasi ini disiarkan di *bus*, dan *cache* lain yang memiliki salinan data tersebut juga diperbarui atau ditandai sebagai tidak valid. Contoh *protokol snooping* yang umum adalah *MESI* (*Modified, Exclusive, Shared, Invalid*) dan *MOESI* (*Modified, Owner, Exclusive, Shared, Invalid*). *Protokol direktori* menggunakan *direktori* pusat yang menyimpan status setiap blok memori yang dibagi di antara berbagai *cache*. Ketika satu *cache* memperbarui data, *direktori* memastikan bahwa semua *cache* lain yang memiliki salinan data tersebut diberi tahu tentang perubahan tersebut. Protokol ini lebih cocok untuk sistem dengan jumlah *prosesor* yang besar, di mana snooping mungkin tidak efisien(Nagarajan et al., 2020).

Namun, menjaga konsistensi cache memiliki tantangan tersendiri, seperti kompleksitas implementasi, *latensi*, dan *overhead* komunikasi antar *cache*, serta masalah skala dalam sistem dengan banyak *prosesor*. Implementasi *protokol* konsistensi *cache* memerlukan tambahan perangkat keras dan logika untuk mengelola status *cache* dan komunikasi antar *cache*, yang menambah kompleksitas sistem. Mekanisme konsistensi cache dapat menambah latensi dan *overhead* komunikasi antar *cache*, yang bisa berdampak negatif pada kinerja sistem, terutama dalam lingkungan multiprosesor dengan beban kerja tinggi. Dalam sistem dengan banyak prosesor dan *cache*, menjaga konsistensi menjadi lebih sulit dan membutuhkan protokol yang lebih canggih dan *skalabel*(Nagarajan et al., 2020).

Secara keseluruhan, menjaga konsistensi antara beberapa *cache* adalah aspek kritis dalam desain sistem komputer modern, terutama dalam sistem *multiprosesor*. Dengan menggunakan protokol snooping dan direktori, sistem dapat memastikan bahwa semua *cache* tetap sinkron dan data yang diakses oleh prosesor

selalu mutakhir dan akurat. Implementasi yang efektif dari mekanisme ini adalah kunci untuk menjaga integritas data dan kinerja sistem yang optimal(Nagarajan et al., 2020).

BAB III
PENUTUP

3.1 Kesimpulan

Prinsip lokalitas referensi, yang meliputi *temporal locality*, *spatial locality*, dan *algorithmic locality*, berperan penting dalam meningkatkan performa sistem *cache*. *Temporal locality* mengacu pada kecenderungan sebuah program untuk mengakses data yang sama berulang kali dalam waktu dekat, memungkinkan *cache* untuk menyimpan data yang baru-baru ini diakses dan kemungkinan besar akan diakses kembali. Ini membantu mengurangi waktu akses data dan meningkatkan efisiensi sistem. *Spatial locality*, di sisi lain, melibatkan akses data yang berdekatan dalam ruang memori, seperti dalam *operasi array*. Dengan teknik seperti *cache block granularity* dan *prefetching*, *cache* dapat menyimpan data yang berdekatan untuk akses yang lebih cepat di masa depan. *Algorithmic locality* mengacu pada pola akses data yang teratur tetapi tidak terlokalisasi secara klasik, seperti dalam aplikasi *grafis 3D*. Dengan memahami dan mengoptimalkan pola ini, *cache* dapat lebih efektif mengelola data dan meningkatkan performa sistem secara keseluruhan.

Ada tiga jenis utama organisasi logis cache yaitu *direct mapped*, *fully associative*, dan *set associative*. *Direct mapped cache* memetakan setiap blok memori ke satu lokasi tertentu dalam *cache*, menawarkan kecepatan akses tinggi tetapi rentan terhadap konflik jika banyak *blok* memori dipetakan ke lokasi yang sama. *Fully associative cache* memungkinkan blok memori disimpan di sembarang lokasi dalam *cache*, mengurangi konflik tetapi menambah kompleksitas dan waktu akses karena pencarian dilakukan di seluruh *cache*. *Set associative* cache menggabungkan elemen dari kedua metode sebelumnya, membagi cache menjadi beberapa set dengan beberapa blok *per set*. Ini menawarkan keseimbangan antara kecepatan akses dan fleksibilitas penempatan data, mengurangi kemungkinan konflik dibandingkan *direct mapped cache* dan lebih efisien dibandingkan *fully associative cache*. *Set associative cache* juga memungkinkan penggunaan algoritma penggantian yang lebih canggih untuk mengelola data yang sering diakses.

Manajemen konten dan konsistensi data dalam sistem *cache* melibatkan menjaga konsistensi dengan dirinya sendiri, dengan *backing store*, dan dengan *cache* lain dalam sistem *multiprosesor*. Untuk menjaga konsistensi internal, cache diindeks menggunakan alamat fisik untuk menghindari masalah sinonim yang dapat menyebabkan inkonsistensi data. Konsistensi dengan backing store dicapai melalui kebijakan *write-through* atau *write-back*, yang masing-masing memiliki kelebihan dan kekurangan dalam hal kecepatan dan efisiensi *bandwidth*. Konsistensi antar *cache* dalam sistem multiprosesor dijaga melalui *protokol snooping* dan *direktori*, yang memastikan bahwa semua *cache* tetap sinkron dan data yang diakses selalu mutakhir. Tantangan utama dalam manajemen konsistensi adalah kompleksitas implementasi, *latensi*, dan *overhead* komunikasi, tetapi dengan protokol yang tepat, *sistem cache* dapat berfungsi secara optimal, menjaga integritas data dan meningkatkan kinerja sistem secara keseluruhan.

3.2 Saran

Untuk memaksimalkan performa sistem *cache,* penting untuk terus memperhatikan dan mengoptimalkan prinsip *lokalitas referensi*. Pengembang sistem harus memastikan bahwa aplikasi memanfaatkan *temporal locality* dengan efektif, misalnya dengan teknik *caching* yang menyimpan data yang baru diakses. *Spatial locality* dapat dioptimalkan melalui *desain algoritma* yang mengakses data berdekatan dalam memori secara berurutan. Selain itu, pengembangan algoritma dan perangkat lunak yang lebih *adaptif* terhadap *pattern access* data dapat mengoptimalkan penggunaan *algorithmic locality*, memungkinkan sistem untuk memprediksi dan memuat data secara lebih efisien.

Dalam memilih jenis organisasi *logis cache*, perlu dipertimbangkan pola *akses* memori dari aplikasi yang akan dijalankan serta kebutuhan kinerja dan efisiensi perangkat keras. *Direct mapped cache* bisa cocok untuk aplikasi dengan akses memori yang terdistribusi secara merata, tetapi *fully associative cache* atau *set associative cache* mungkin lebih sesuai untuk aplikasi dengan akses memori yang tidak terduga atau sering konflik. *Set associative cache* seringkali menawarkan kompromi terbaik antara fleksibilitas dan efisiensi. Selain itu, pemilihan algoritma penggantian yang

sesuai seperti *Least Recently Used* (*LRU*) dapat lebih lanjut meningkatkan efektivitas pengelolaan *cache*.

Untuk menjaga konsistensi dan kinerja sistem cache, perlu diterapkan protokol konsistensi *cache* yang efisien, terutama dalam lingkungan *multiprosesor*. Protokol *snooping* dan direktori harus dipilih dan diimplementasikan berdasarkan kebutuhan spesifik sistem dan jumlah *prosesor* yang terlibat. Kebijakan *write-through* dan *write-back* harus dipilih berdasarkan *trade-off* antara kecepatan akses dan bandwidth yang tersedia. Selain itu, pemantauan terus-menerus dan penyesuaian dinamis terhadap manajemen konsistensi dan kebijakan penulisan dapat membantu meminimalkan latensi dan overhead komunikasi, memastikan kinerja sistem yang optimal dalam berbagai aplikasi.

DAFTAR PUSTAKA

Agarwal, I. (2021). System, apparatus and method for processing remote direct memory access operations with a device-attached memory. *US Patent 11,036,650*. https://patents.google.com/patent/US11036650B2/en

Bhavikatti, N. (2023). On Algorithmic Cache Optimization. *ArXiv Preprint ArXiv:2311.07615*. https://arxiv.org/abs/2311.07615

Chong, B. I., & Liu, M. (2023). Cache memory and method of its manufacture. *US Patent App. 18/004,968*. https://patents.google.com/patent/US20230245691A1/en

Fang, L., & Zhou, Z. (2022). File pre-fetch scheduling for cache memory to reduce latency. *US Patent 11,366,757*. https://patents.google.com/patent/US11366757B2/en

Farshin, A., Roozbeh, A., Jr, G. Q. M., & ... (2020). Reexamining Direct Cache Access to Optimize \{I/O\} Intensive Applications for Multi-hundred-gigabit Networks. *2020 USENIX Annual …*. https://www.usenix.org/conference/atc20/presentation/farshin

Favor, J. G., & Srinivasan, S. (2023). Virtually-tagged data cache memory that uses translation context to make entries allocated during execution under one translation context inaccessible during …. *US Patent 11,625,479*. https://patents.google.com/patent/US11625479B2/en

Gouk, D., Lee, S., Kwon, M., & Jung, M. (2022). Direct access,\{High-Performance\} memory disaggregation with \{DirectCXL\}. *2022 USENIX Annual Technical …*. https://www.usenix.org/conference/atc22/presentation/gouk

Hyden, R. J., Spitalnik, J., & Sullivan, T. J. (2020). Mitigating costs of presentation and delivery of digital media assets based on geographically distributed cache memory contents. *US Patent App. 16/285,147*. https://patents.google.com/patent/US20200275139A1/en

Ivester, S., & Sahasrabudhe, K. (2021). Method and apparatus for performing atomic operations on local cache slots of a shared global memory. *US Patent 11,074,113*. https://patents.google.com/patent/US11074113B1/en

Kazakov, M. V. (2023). Processing device and method of sharing storage between cache memory, local data storage and register files. *US Patent App. 17/467,104*. https://patents.google.com/patent/US20230069890A1/en

Leidel, J., & Murphy, R. C. (2020). Stacked memory device system interconnect directory-based cache coherence methodology. *US Patent 10,838,865*. https://patents.google.com/patent/US10838865B2/en

Li, B., Wei, J., & Kim, N. S. (2021). Virtual-Cache: A cache-line borrowing technique for efficient GPU cache architectures. *Microprocessors and Microsystems*. https://www.sciencedirect.com/science/article/pii/S0141933121004622

Nagarajan, V., Sorin, D. J., Hill, M. D., & Wood, D. A. (2020). *A primer on memory consistency and cache coherence*. library.oapen.org. https://library.oapen.org/handle/20.500.12657/61248

Paschos, G., Iosifidis, G., & Caire, G. (2020). Cache optimization models and algorithms. *Foundations and Trends® in …*. https://www.nowpublishers.com/article/Details/CIT-104

Pulkit, A., Naval, S., & Laxmi, V. (2023). A Survey of Side-Channel Attacks in Context of Cache--Taxonomies, Analysis and Mitigation. *ArXiv Preprint ArXiv:2312.11094*. https://arxiv.org/abs/2312.11094

Purnal, A., Turan, F., & Verbauwhede, I. (2021). Prime+ Scope: Overcoming the observer effect for high-precision cache contention attacks. *Proceedings of the 2021 ACM …*. https://doi.org/10.1145/3460120.3484816

Ramanujan, R. K., Hinton, G. J., & Zimmerman, D. J. (2020). Dynamic partial power down of memory-side cache in a 2-level memory hierarchy. *US Patent 10,795,823*. https://patents.google.com/patent/US10795823B2/en

Roy, P. (2022). *Simncore: Multilevel cache memory design simulator for manycore system*. search.proquest.com. https://search.proquest.com/openview/9bf8a91bb100704ff3dbf50d4088bd2d/1?pq-origsite=gscholar%5C&cbl=18750%5C&diss=y

Sorin, D., Hill, M., & Wood, D. (2022). *A primer on memory consistency and cache coherence*. books.google.com. https://books.google.com/books?hl=en%5C&lr=%5C&id=84mbEAAAQBAJ%5C&oi=fnd%5C&pg=PP1%5C&dq=cache+memory%5C&ots=1_LemKf1g0%5C&sig=oDOvgspWPS6TrK7TBF2NRlaY0GA

Srivastava, S., & Singh, P. K. (2022). Proof of Optimality based on Greedy Algorithm for Offline Cache Replacement Algorithm. *International Journal of Next …*. https://search.ebscohost.com/login.aspx?direct=true%5C&profile=ehost%5C&scope=site%5C&authtype=crawler%5C&jrnl=09765034%5C&AN=160004946%5C&h=G%2BMzZLCeNfk4lcF2xX7nuBR4jVfmfDqj56Se9uxZGEQJMtlhRV0Oh7aAGCLc9YgHM02j5UatVitEsE5nG3HQiA%3D%3D%5C&crl=c

Tan, Q., Zeng, Z., Bu, K., & Ren, K. (2020). PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24086-paper.pdf

Tran, D. H., Chatzinotas, S., & ... (2022). Throughput maximization for backscatter- and cache-assisted wireless powered UAV technology. *IEEE Transactions on …*. https://ieeexplore.ieee.org/abstract/document/9723521/

Wallach, S. J. (2022). Cache systems and circuits for syncing caches or cache sets. *US Patent 11,360,777*. https://patents.google.com/patent/US11360777B2/en

Wang, Z., Jin, B., Yu, Z., & Zhang, M. (2024). Model Tells You Where to Merge: Adaptive KV Cache Merging for LLMs on Long-Context Tasks. *ArXiv Preprint ArXiv:2407.08454*. https://arxiv.org/abs/2407.08454