GCRFP - *PAGE REPLACEMENT* UNTUK *SOLID STATE DRIVE* MENGGUNAKAN *GHOST-CACHE*

Wahyu Suadi[1)], Supeno Djanali[2)], Waskitho Wibisono[3)], Radityo Anggoro[4)], dan Ary Mazharuddin Shiddiqi[5)]
[1, 2, 3, 4, 5)] Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia 60111
e-mail: wsuadi@if.its.ac.id[1)], supeno@its.ac.id[2)], waswib@if.its.ac.id[3)], onggo@if.its.ac.id[4)], ary.shiddiqi@gmail.com[5)]

ABSTRAK

Solid State Drive (SSD) adalah satu alternatif yang penyimpanan data yang populer saat ini. Banyak digunakan sebagai media cache untuk mempercepat akses data ke hard disk (HDD). Paper ini mengusulkan satu teknik page replacement di SSD cache, menggunakan parameter frekuensi dan resensi bergantian secara adaptif untuk mengatasi perubahan pola akses sekaligus meminimalkan jumlah proses tulis ke SSD. Algoritma yang diusulkan mampu melakukan pemilihan teknik replacement yang sesuai dengan pola akses penguna sehingga didapatkan hit rate yang lebih baik. Algoritma yang diusulkan juga diintegrasikan dengan mekanisme ghost-cache sehingga didapatkan pengurangan jumlah proses penulisan ke SSD dengan signifikan. Ujicoba dilakukan dengan menggunakan dataset yang riiil untuk proses akses penulisan dan pembacaan dari kasus riil. Uji coba menunjukkan algoritma yang diusulkan mampu memberikan hasil yang baik dibandingkan algoritma sejenis yang lain.

***Kata kunci**: Block cache replacement, disk cache, solid state disk.*

GCRFP - PAGE REPLACEMENT FOR SOLID STATE DRIVE USING GHOST-CACHE

Wahyu Suadi[1)], Supeno Djanali[2)], Waskitho Wibisono[3)], Radityo Anggoro[4)], and Ary Mazharuddin Shiddiqi[5)]
[1, 2, 3, 4, 5)] Department of Informatics, Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia 60111
e-mail: wsuadi@if.its.ac.id[1)], supeno@its.ac.id[2)], waswib@if.its.ac.id[3)], onggo@if.its.ac.id[4)], ary.shiddiqi@gmail.com[5)]

ABSTRACT

State Drive (SSD) is an alternative to data storage that is popular today, widely used as a media cache to speed up data access to the hard disk (HDD). This paper proposes page replacement technique on SSD cache that used frequency and recency parameter, alternately. The algorithm is selected adaptively based on trace input. This method helps to overcome changes in access patterns while minimizing the number of write processes to SSD. The proposed algorithm can choose a replacement technique that suits the user access pattern so that it can bring a better hit rate. The proposed algorithm is also integrated with the ghost-cache mechanism so that the reduction in the number of writing processes to SSD is significant. The experiment runs using a real dataset, describing trace of data read, and data write taken from real usage. The trial shows that the proposed algorithm can give good results compared to other similar algorithms.

***Keywords**: Block cache replacement, disk cache, solid state disk.*

I. PENDAHULUAN

DALAM satu dekade ini media flash memory, khususnya *Solid State Disk* (SSD) menjadi satu teknologi yang merubah paradigma media penyimpanan massal. Media ini digunakan dari *notebook*, *desktop* hingga mesin kelas *server*. Media ini memiliki keunggulan dibandingkan media *hard disk*, seperti pemakaian daya yang kecil, tahan goncangan dan kemampuan baca-tulis yang lebih tinggi. Ini yang menjadikan SSD menjadi *cache* untuk media disk [1].

Teknologi *cache* bertujuan memasukkan data yang sering digunakan di memori yang lebih cepat, mengurangi akses di memori yang lebih lambat sehingga menghasilkan peningkatan kinerja sistem. Media SSD digunakan sebagai *cache* untuk media *disk*, artinya adalah sebagian data dari *disk* yang sering digunakan akan disimpan di SSD. Media SSD memiliki kecepatan 5 kali lebih cepat daripada *disk* [2], namun media SSD juga memiliki ciri khas.

SSD memiliki keterbatasan pada jumlah penulisan, dengan penulisan yang massif, media SSD akan mengalami penurunan umur pakai secara drastis. Algoritma *cache* untuk *disk* dan memori tidak bisa digunakan di SSD. Algoritma tersebut tidak memperhitungkan penulisan yang banyak akan menurunkan umur media. Algoritma LRU

 DOI: http://dx.doi.org/10.12962/j24068535.v18i2.a986

misalnya, hanya memperhitungkan *recency* akses, dengan cepat memindahkan data dari HDD ke SSD. Sebelum data tersebut bisa digunakan, sudah diganti dengan data yang lain. Menimbulkan jumlah penulisan yang tinggi. Algoritma LRU tidak sesuai untuk digunakan di SSD.

Penelitian pada algoritma *cache* menghasilkan banyak variasi algoritma. Untuk memaksimalkan pemakaian *cache*, algoritma berusaha untuk menyimpan blok yang populer. Secara umum, semua algoritma menggunakan dua asumsi empiris. Yang pertama adalah *temporal locality*, blok yang baru saja digunakan memiliki kemungkinan untuk digunakan lagi yang lebih tinggi daripada yang blok diakses sebelumnya. Yang lainnya adalah *skewed popularity* dari blok. Beberapa blok diakses lebih banyak daripada banyak blok yang lain. Dua algoritma yang mengusung konsep tersebut, LRU dan LFU. Algoritma LRU pertama kali dikembangkan untuk manajemen *cache* di memori. Tujuannya utamanya adalah untuk meningkatkan *hit ratio* dengan memanfaatkan *temporal locality*. Jarak waktu akses ke blok yang sama umumnya tidak besar. LRU diperbaiki lagi dengan algoritma seperti LRU-K [3], FBR [4], dan ARC [5].

Algoritme LRU dan LRU memiliki kelebihan masing-masing, namun pemakaiannya tergantungidari pola akses aplikasi. LRU unggul karena menyimpan data yang dianggap akan dipakai di masa mendatang sedang LFU menyimpan data yang sering digunakan. Pada pola akses yang terus berubah menyebabkan tidak ada satu algoritma yang selalu unggul, maksudnya pada satu waktu LRU akan unggul di waktu yang lain LFU akan lebih. Untuk mengatasi hal ini, dikembangkan algoritma CRFP [6].

Algoritma CRFP [6] menggunakan informasi *recency* dan *frequency* serta menggunakan algoritma LRU dan LFU bergantian secara adaptif. Pergantian dari satu algoritma ke yang lain berdasarkan pencatatan *hit* pada *ghost-cache*, antrian LRU yang hanya menyimpan informasi blok. *Ghost-cache* di CRFP menyimpan *victim cache*, yaitu blok yang terlempar dari *cache*. *Ghost-cache* pada CRFP digunakan untuk mendeteksi perubahan pola akses. Pada LARC [7] mekanisme *ghost-cache* digunakan sebagai filter terhadap data yang akan masuk ke SSD. Hanya blok yang telah diakses dua kali yang akan dimasukkan ke SSD.

Dalam paper ini kami mengusulkan algoritma *repl acement policy* baru untuk meningkatkan performa sistem berbasis SSD dan *disk*, yang kami sebut dengan *Ghost Cache Replace* (GCRFP). GCRFP memasukkan mekanisme *ghost-cache* pada CRFP. Sama seperti LARC, *ghost-cache* diletakkan di posisi depan dan berfungsi sebagai filter. Komponen ini dapat berfungsi dengan baik, karena distribusi pemakaian blok biasanya berpola *skew*, hanya ada sedikit blok yang diakses lebih dari sekali. Hanya sedikit blok yang berhak masuk kedalam SSD, sehingga jumlah penulisan SSD akan berkurang. *Ghost-cache* yang ada di CRFP berfungsi untuk mencatat perubahan pola akses. Jika *hit* di *ghost-cache* melewati *threshold* atau *threshold ratio* (*hit*/*miss*), maka pola akses berbasis *frequency* lebih dominan maka algoritma yang digunakan menjadi LFU. Berlaku sebaliknya, jika akses yang miss melewati *threshold* atau *threshold ratio* (*miss*/*hit*), maka algoritma diubah menjadi LRU. Simulasi menunjukkan bahwa algoritma ini mampu memberikan peningkatan performa yang signifikan.

II. PAGE REPLACEMENT POLICY DI SSD

Cache digunakan untuk menjembatani perbedaan waktu akses dari memori cepat dengan memori yang lebih lambat. *Cache* banyak digunakan untuk mengurangi perbedaan kecepatan antara register dan memori utama, antara memori utama dengan HDD. Di penelitian ini, *cache* digunakan untuk menjembatani perbedaan kecepatan HDD dan SSD dengan menyimpan data yang potensial di dalam SSD.

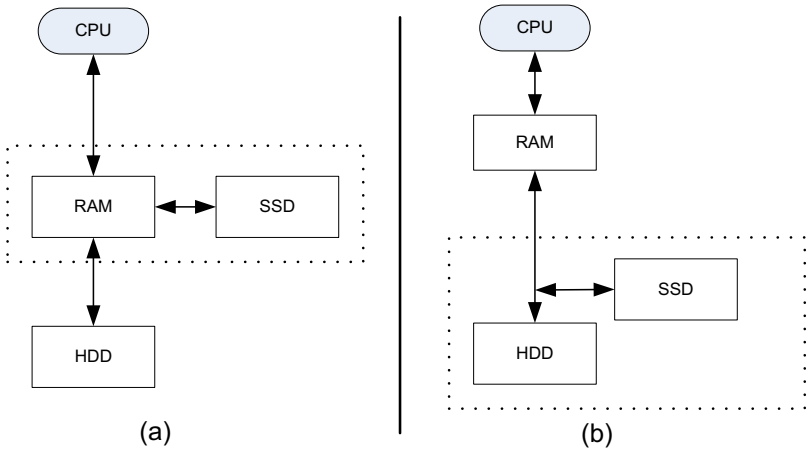
Dengan cache berukuran lebih kecil dari data utama, maka blok yang berpotensi akan digunakan kembali saja, yang disimpan dalam *cache*. Untuk memilih blok yang akan dikeluarkan dari dalam *cache* digunakan *Page Replacment Policy.*

A. Penggunaan SSD untuk cache

Ada dua macam model pemakaian SSD untuk *cache* [8] ditunjukkan pada Gambar 1. Pada model (a) SSD digunakan sebagai ekstensi dari sistem memori. Di model ini RAM dan SSD dijadikan satu *cache tier* terhadap HDD. Pada model (b) SSD digunakan sebagai *disk* ekstensi yang diakses menggunakan *interface* standar. Model ini transparan terhadap komponen yang lain. Model yang digunakan pada penelitian ini adalah model B, SSD bertugas sebagai *cache* terhadap HDD.

B. Algoritma Cache

Penelitian pada algoritma *cache* menghasilkan banyak variasi algoritma. Untuk memaksimalkan pemakaian cache, algoritma berusaha untuk menyimpan blok yang populer. Secara umum, semua algoritma menggunakan dua asumsi empiris. Yang pertama adalah *temporal locality*, blok yang baru saja digunakan memiliki kemungkinan untuk digunakan lagi yang lebih tinggi daripada yang blok diakses sebelumnya. Yang lain adalah *skewed popu-*



Gambar 1. Model pemakaian *cache* SSD dan HDD.

larity dari blok. Beberapa blok diakses lebih banyak daripada banyak blok yang lain. Dua algoritma yang mengusung konsep tersebut, LRU dan LFU. LRU banyak digunakan di sistem produksi, karena kesederhaan implementasi dan *overhead* yang rendah, O(1). LRU memiliki kelemahan pada akses yang memiliki *weak temporal locality*. Seperti mengakses data besar akan menggantikan isi *cache* dan mengisinya dengan blok yang hanya akan diakses satu kali. Ini lebih buruk pada sistem *multiuser*, satu *user* yang melakukan pembacaan data besar akan membuang blok yang lebih potensial dari user lain, membuat unjuk kerja sistem menurun. LRU tidak bisa mengakomodir pola akses seperti ini. Blok yang banyak diakses dapat digantikan oleh blok yang jarang digunakan. Beberapa algoritma diusulkan untuk memperbaiki kelemahan ini. Seperti MQ [9], 2Q [10], ARC [5], dan CRFP [6].

Algoritma ARC menggunakan dua buah LRU untuk mencatat *recency* dan *frequency* pada dua buah struktur LRU. LRU1 digunakan menyimpan informasi *recency* dan LRU2 digunakan untuk menyimpan blok yang diakses lebih dari satu kali. Total dari ukuran LRU1 dan LRU2 tetap, namun proporsi ukuran LRU1 dan LRU2 bisa adaptip sesuai dengan satu aturan.

Algoritma CRFP [6] menggunakan pendekatan yang berbeda dari ARC, algoritma ini menggunakan informasi *recency* dan *frequency* serta menggunakan algoritma LRU dan LFU bergantian secara adaptif. Pergantian dari satu algoritma ke yang lain berdasarkan pencatatan hit pada *ghost-cache*, antrian LRU yang hanya menyimpan informasi blok. *Ghost-cache* di CRFP menyimpan *victim cache*, yaitu blok yang terlempar dari *cache*. Jika blok di *ghost-cache* kembali masuk ke *cache*, maka nilai *frequency* akan dikembalikan dan ditambah 1. Jika *hit* di *ghost-cache* melewati *threshold* atau *threshold ratio* (*hit*/*miss*), maka pola akses berbasis *frequency* lebih dominan maka algoritma yang digunakan menjadi LFU. Berlaku sebaliknya, jika akses yang *miss* melewati *threshold* atau *threshold ratio* (*miss*/*hit*), maka algoritma diubah menjadi LRU.

Algoritma diusulkan di atas tidak ada yang memperhitungkan keterbatasan siklus tulis dari SSD. SSD memiliki siklus tulis yang terbatas, jika jumlah penulisan melebihi batas tertentu maka media akan rusak. Algoritma LRU, jika diterapkan pada SSD, akan menghasilkan *hit ratio* yang rendah dan menimbulkan banyak operasi tulis yang tidak perlu. Semakin banyaknya pemakaian SSD membuat pemakaian SSD sebagai cache ke HHD semakin tinggi.

Ada beberapa riset yang membahas tentang algoritma cache untuk SSD. Salah satunya adalah LARC. Algoritma *Lazy Adaptive Replacement Cache* (LARC) [7] mengindentifikasi blok yang jarang diakses dan membuatnya tidak masuk dalam *cache*. LARC menggunakan *ghost-cache*, diimplementasikan sebagai antrian LRU yang hanya menyimpan informasi blok, tidak menyimpan data blok sebenarnya. Blok yang diakses pertama kali, akan masuk ke *ghost-cache* sebagai blok yang potensial bisa masuk ke *cache*. Jika blok diakses kedua kali, maka blok termasuk dianggap populer dan akan dimasukkan kedalam SSD. *Cache* di SSD dikelola berdasarkan aturan LRU. Akses akan melihat apakah satu blok sudah ada di dalam *cache* SSD, jika ada maka data akan diambil dari SSD dan blok tersebut dipindahkan ke MRU (*most recently used*, blok yang baru saja diakses). Jika tidak ada di SSD, maka akan dicari di *ghost-cache*, jika ditemukan, maka data baru akan masuk ke bagian LRU (*least recenty used,* blok yang paling lama diakse) dari SSD. Jika blok tidak ada di SSD dan di *ghost-cache*, maka permintaan akan diambil dari HDD dan blok id akan disimpan di *ghost-cache*. Jika SSD dan *ghost-cache* penuh, maka blok LRU akan dibuang.

III. PAGE REPLACEMENT POLICY DENGAN GCRFP

Paper ini mengusulkan pendekatan baru untuk perbaikan mekanisme *page replacement* dalam lingkungan SSD dan *disk* yang kami beri nama denagn GCRFP. Algoritma ini menggunakan ide yang sama dengan CRFP, yaitu menggunakan informasi *recency* dan *frequency* serta menggunakan algoritma LRU dan LFU bergantian secara adaptif. GCRFP memasukkan mekanisme *ghost-cache* pada CRFP. Sama seperti LARC, *ghost-cache* diletakkan di posisi depan dan berfungsi sebagai filter. Komponen ini dapat berfungsi dengan baik, karena distribusi

pemakaian blok biasanya berpola *skew*, hanya ada sedikit blok yang diakses lebih dari sekali. Hanya sedikit blok yang berhak masuk kedalam SSD, sehingga jumlah penulisan SSD akan berkurang. *Ghost-cache* yang ada di CRFP berfungsi untuk mencatat perubahan pola akses. Jika *hit* di *ghost-cache* melewati *threshold* atau *threshold ratio* (*hit*/*miss*), maka pola akses berbasis *frequency* lebih dominan maka algoritma yang digunakan menjadi LFU. Berlaku sebaliknya, jika akses yang miss melewati *threshold* atau *threshold ratio* (*miss*/*hit*), maka algoritma diubah menjadi LRU.

Dalam pengembangan algoritma ini kami menggunakan data ujicoba yang untuk menguji performa dari GCRFP dibandingkan dengan algoritma yang lain. Untuk dapat melakuan ujicoba perlu dilaukan proses pengolahan *trace* dari data terlebih dahulu. Pengolahan trace kami jelaskan pada sub bagian berikut.

A. Pengolahan Trace

Trace yang digunakan berasal dari repository *Umass Trace Repository* [11] yang diambil dari *workload* sistem sesungguhnya. *Trace* ini banyak digunakan oleh penelitian sebelumnya. *Trace* tersebut mewakili dua pola *workload*, yaitu *websearch*, *trace* dari mesin pencarian web, *workload* dengan sedikit operasi tulis dan *financial*, *trace* dari mesin *database* transaksi, *workload* dengan banyak operasi tulis.

Mengingat setiap baris *trace* bisa mengandung lebih dari satu blok, maka *trace* harus diolah menjadi *trace* yang menunjuk ke satu blok. Hasilnya ditunjukkan di Tabel I.

Untuk mengetahui tentang bagaimana sebaran operasi baca unik dan operasi tulis unik terhadap blok unik, dapat dilihat di Tabel II. Blok unik dan operasi blok unik menjadi penting, karena informasi *cache* disimpan secara unik, tidak ada blok sama disimpan lebih dari satu kali. Di Tabel II bisa dilihat bahwa di *trace Financial*, operasi tulis bisa menyebar ke keseluruhan blok, sedangkan di *websearch* operasi tulis tampil dominan.

B. Desain LRU

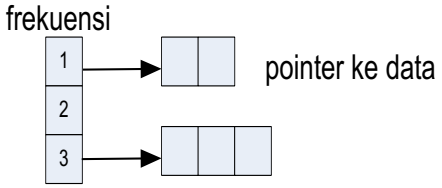
LRU disusun dengan dua struktur data, *red black tree* dan *linked list*. *Tree* digunakan sebagai informasi *key-value*, dengan *key* adalah lba dan nilainya adalah *pointer* ke informasi blok. Informasi yang perlu disimpan adalah waktu akses terakhir (diganti dengan nomor *trace*), nomor lba, operasi yang dikerjakan, dan pointer di elemen *list*. Untuk setiap akses maka elemen list yang ditunjuk akan dipindahkan ke posisi MRU (*most recently used)*. Jika struktur data sudah penuh, maka data yang ada di berada di posisi LRU (*least recently used)* akan dibuang dari struktur data.

C. Desain LFU

LFU disusun dengan dua struktur data, *red black tree* dan *array linked list*. *Tree* digunakan sebagai informasi *key-value*, dengan *key* adalah lba dan nilainya adalah pointer ke informasi blok. Informasi yang perlu disimpan adalah frekuensi, nomor lba, operasi yang dikerjakan dan pointer di elemen *list*. Gambar struktur data dapat dilihat di Gambar 2. Untuk setiap akses maka elemen *list* yang ditunjuk akan ditambah nilai frekuensinya, serta dipindahkan ke *list* di array frekuensi di atasnya, di posisi kiri. Jika struktur data sudah penuh, maka data yang ada di berada di *array* dengan frekuensi terkecil dan posisi kanan, akan dibuang dari struktur data.

D. Algoritma Combined the LRU and LFU Policies (CRFP)

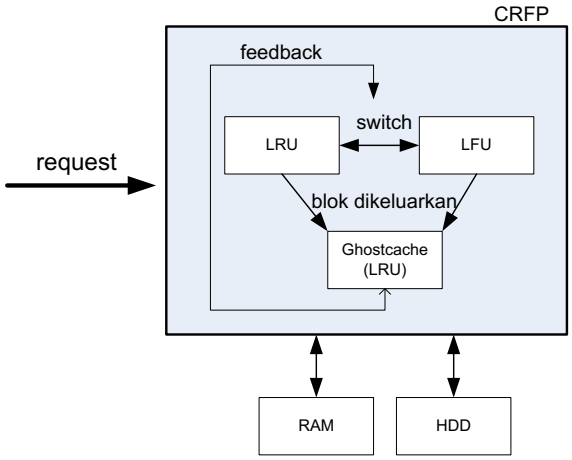
Setiap kali akses maka elemen list yang ditunjuk akan dipindahkan ke posisi MRU (*most recently used)*. Jika struktur data sudah penuh, maka data yang ada di berada di posisi LRU (*least recently used)* akan dibuang dari struktur data.



Gambar 2. Array frekuensi LFU, berisikan *list* ke *pointer* data.

TABEL I
SEBARAN OPERASI BACA DAN OPERASI TULIS.

Trace	***Read Request***	***Write Request***	**Total *Request***	**Rasio Baca**
Websearch	3.995.896	456	3.996.352	99.99%
Financial	13.882.741	3.810.800	17.693.541	78.46%



Gambar 3. Alur kerja CRFP.

TABEL II
SEBARAN DATA DAN OPERASI KE BLOK UNIK.

Trace	***Read Request* Unik**	***Write Request* Unik**	**Blok Unik**	**Rasio Baca**
Websearch	1.309.998	42	1.310.018	100.00%
Financial	831.654	755.866	909.660	52.39%

TABEL III
HIT RATE UNTUK *TRACE WEB*.

***Hit rate* relatif terhadap LARC**					
	100	**150**	**200**	**300**	**(x1000)**
GCRFP	1.12	1.13	1.13	1.12	
LARC	1.00	1.00	1.00	1.00	
GLRU	1.09	1.1	1.09	1.07	
LFU	1.16	1.21	1.23	1.25	
LRU	0.49	0.52	0.55	0.63	

Algoritma LRU dan LFU banyak digunakan dengan banyak varian algoritma *replacement policy* lain. Masing-masing memiliki kelemahan dan kelebihan masing-masing. LRU mampu menangkap aspek kekinian dari blok populer berdasarkan *temporal locality* dan LFU mampu menyimpan informasi blok mana yang paling popular. ARC [5, 3] adalah salah yang mengkombinasikan dari dari kedua algoritma tersebut. Masalah muncul jika ada perubahan pola akses dan satu dari algoritma diatas tidak bisa selalu memberikan hasil yang baik. CRFP berusaha menyelesaikan masalah di atas dengan memasukkan kedua algoritma secara bersamaan, bisa dilihat di Gambar 3.

Sedangkan untuk bisa menggunakan dua algoritma secara bersamaan diperlukan metoda *switching*. Algoritma di bawah ini menggambarkan algoritma yang digunakan. *H* adalah *hit ghost-cache out*. *O* adalah jika terjadi *miss*. *SWITCH_TIMES* adalah berapa kali jumlah *hit*/*miss*. *SWITCH_RATIO* adalah rasio antara *H*/*O* atau *O*/*H*.

E. Algoritma GCRFP

Algoritma *replacement policy* untuk SSD seperti LARC dan ARC memasukkan mekanisme *ghost-cache*. Di LARC digunakan untuk memfilter blok yang akan masuk ke SSD. Di ARC, digunakan untuk menampung blok-blok yang telah diakses lebih dari sekali untuk bisa masuk dalam LRU utama. Metode yang sama akan diterapkan juga di CRFP menjadi algoritma GCRP (*ghost-cache CRFP*). Cara kerja CRRP dapat dilihat pada algoritma di bawah ini.

IV. HASIL DAN PEMBAHASAN

Untuk menguji kinerja dari algoritma yang diusulkan, perlu diimplementasi simulasi dalam bahasa Go [12]. Menggunakan struktur data internal dan komponen *red-black-tree*, simulator disusun dalam sekitar 2600 baris kode. Dikompilasi dan dijalankan di mesin Linux dengan spesifikasi i5 7400 dengan memori 8GB. Sebelum bisa menjalankan simulator, perlu dilakukan preprocessing atas trace untuk mendapatkan akses blok individu, mengingat tiap trace memiliki ukuran blok yang berbeda serta ukuran akses yang berbeda pula. Algoritma yang diimplentasi adalah LARC, GLRU adalah algoritma LRU yang ditambahkan *ghost-cache* untuk memfilter blok yang dimasukkan, algoritma LRU standar sebagai pembanding dan algoritma LFU standar sebagai pembanding.

Percobaan pertama melakukan perhitungan *hit rate*, yaitu jumlah akses yang telah ada didalam cache (SSD),

semakin besar *hit rate* maka makin banyak akses yang diambil dari SSD maka makin baik dan makin cepat sistem berjalan. Sebaliknya makin rendah *hit rate*, maka makin banyak data yang diambial dari HDD dan sistem akan berjalan lebih lambat. Percobaan kedua melakukan perhitungan *write count*, yaitu jumlah penulisan yang terjadi di *cache* (SSD), semakin banyak *write count* maka jumlah penulisan ke *cache* (SSD) akan makin banyak pula. Ini tidak diinginkan, karena semakin banyak penulisan ke SSD akan mengurangi jumlah siklus pada SSD dan mempercepat kerusakan di media.

Untuk mempermudah membaca data dari tabel, kami menggunakan LARC sebagai *baseline*. LARC digunakan sebagai *baseline*, karena algoritma ini telah menggunakan *ghost-cache* untuk memfilter blok potensial. Apabila satu algoritma memiliki performa yang lebih baik, berarti algoritma *replacement policy* yang digunakan adalah lebih dari pada LRU yang diterapkan oleh LARC.

Tabel III menunjukkan *hit rate* dari masing-masing algoritma dengan menggunakan ukuran *cache* 100.000 hingga 300.000 untuk trace web. Trace ini dihasilkan pada aplikasi pencarian web, didominasi oleh operasi baca dengan operasi tulis yang sedikit. Untuk baseline digunakan data LARC. Tabel menunjukkan GCRFP memiliki kinerja lebih baik daripada LARC. LRU menunjukkan performa paling rendah, terutam pada cache ukuran 100.000 dan tetap buruk hingga ke ukuran 300.000. LFU memiliki *hit rate* terbaik, lebih baik daripada GCRP, baik dari ukuran *cache* kecil maupun besar.

Tabel IV menunjukkan *write count* dari masing-masing algoritma dengan menggunakan ukuran *cache* 100.000 hingga 300.000 untuk *trace web*. Tabel menunjukkan GCRP memiliki *write count* yang lebih rendah dari LFU (yang memiliki *hit rate* terbaik dari ujicoba sebelumnya). Performa terbaik dimiliki oleh GLRU, namun algoritma ini masih kalah dengan GCRFP pada *hit rate*.

Tabel V menunjukkan *hit rate* dari masing-masing algoritma dengan menggunakan ukuran *cache* 100.000 hingga 300.000 untuk *trace financial*. *Trace* ini dihasilkan pada aplikasi OLTP, didominasi oleh operasi baca dengan operasi banyak operasi tulis, lebih banyak daripada trace web. Untuk *baseline* digunakan data LARC. Tabel menunjukkan GCRFP memiliki kinerja lebih baik daripada LARC. LRU menunjukkan performa naik, terutama pada *cache* ukuran 100.000 dan tetap baik hingga ke ukuran 300.000. LFU memiliki *hit rate* terburuk, lebih buruk dibandingkan GCRP, baik dari ukuran *cache* kecil maupun besar.

Algoritma 1: CRFP dan *ghost-cache*.

```
reset  nilai H & O /* H=O=0 */
if request ada di LRUout then
   pindahkan ke LRU utama
   inc(H)
else
   inc(O)
}

if (SWITCH == 0) then /* LRU policy */
   if  H>SWITCH_TIMES and  (H / O) > SWITCH_RATIO  then
       reset  nilai H & O
       SWITCH = 1                  /* ubah ke LFU */
if (SWITCH == 1) then /* LFU policy */ {
    if  O>SWITCH_TIMES and   (O / H) > SWITCH_RATIO  then
       reset  nilai H & O
       SWITCH = 0                /* ubah ke LRU */
reset  nilai H & O /* H=O=0 */

if request ada di LRUout then
   pindahkan ke LRU utama
   inc(H)
else
   inc(O)
}

if (SWITCH == 0) then /* LRU policy */
   if  H>SWITCH_TIMES and  (H / O) > SWITCH_RATIO  then
       reset  nilai H & O
       SWITCH = 1                  /* ubah ke LFU */
if (SWITCH == 1) then /* LFU policy */ {
    if  O>SWITCH_TIMES and   (O / H) > SWITCH_RATIO  then
       reset  nilai H & O
       SWITCH = 0                /* ubah ke LRU */
```

TABEL IV
WRITE COUNT UNTUK *TRACE WEB*.

***Write count* relatif terhadap LARC**					
	100	150	200	300	(x1000)
GCRFP	1.11	1.11	1.14	1.20	
LARC	1.00	1.00	1.00	1.00	
GLRU	1.09	1.09	1.10	1.09	
LFU	50.34	35.69	28.08	18.62	
LRU	52.11	37.51	29.90	20.22	

TABEL V
HIT RATE UNTUK *TRACE FINANCIAL*.

***Hit rate* relatif terhadap LARC**					
	100	**150**	**200**	**300**	**(x1000)**
GCRFP	1.02	1.02	1.02	1.01	
LARC	1.00	1.00	1.00	1.00	
GLRU	1.01	1.01	1.01	1.01	
LFU	0.78	0.83	0.87	0.92	
LRU	1.10	1.09	1.09	1.08	

TABEL VI
WRITE COUNT UNTUK *TRACE FINANCIAL*.

***Write count* relatif terhadap LARC**					
	100	150	200	300	(x1000)
GCRFP	1.19	1.17	1.13	1.09	
LARC	1.00	1.00	1.00	1.00	
GLRU	1.09	1.10	1.09	1.08	
LFU	2.80	2.48	2.22	1.88	
LRU	2.12	1.89	1.72	1.52	

Tabel VI menunjukkan *write count* dari masing-masing algoritma dengan menggunakan ukuran *cache* 100.000 hingga 300.000 untuk *trace web*. Tabel menunjukkan GCRP memiliki *write count* yang lebih rendah dari LRU (yang memiliki *hit rate* terbaik dari ujicoba sebelumnya). Performa terbaik dimiliki oleh GLRU, namun algoritma ini masih kalah dengan GCRFP pada *hit rate*.

V. KESIMPULAN

Tabel I dan Tabel III menunjukkan bahwa perbedaan *trace* bisa membuat perbedaan kinerja LRU dan LFU. Penerapan LRU dan LRU secara adaptif di GCRFP memberikan hasil yang konsisten, walaupun tidak selalu memberikan hasil yang terbaik secara *hit rate*. Tabel II dan Tabel IV menunjukkan bahwa penambahan *ghost-cache* sebagai filter data yang tidak potensial untuk masuk ke SSD memberikan *write count* yang baik, walaupun tidak selalu memberikan hasil yang terbaik. Jika dikombinasikan hasil performa *hit rate* dan *write count*, GCRFP memberikan hasil yang baik dan konsisten, walaupun tidak selalu memberikan hasil yang terbaik.

DAFTAR PUSTAKA

[1] S. S. Rizvi dan T. Chung, "Flash SSD vs HDD: High Performance Oriented Modern Embedded and Multimedia Storage Systems," dalam *Proc. International Conference on Computer Engineering and Technology*, 2010.

[2] N. Fisher, Z. He, dan M. McCarthy, "A hybrid filesystem for hard disk drives in tandem," *Computing,* vol. 94, hal. 21-68, 2011.

[3] E. J. O'Neil, P. E. O'Neil, dan G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," dalam *Proc. ACM SIGMOD International Conference on Management of Data*, 1993.

[4] J. T. Robinson dan M. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," dalam *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990.

[5] N. Megiddo dan D. S. Modha, "ARC: a Self-tuning, Low Overhead Replacement Cache," dalam *Proc. USENIX Conference on File and Storage Technologies*, 2003.

[6] Z. Li, D. Liu, dan H. Bi, "CRFP: A Novel Adaptive Replacement Policy Combined the LRU and LFU," dalam *Proc. International Conference on Computer and Information Technology Workshops*, 2008.

[7] S. Huang, Q. Wei, J. Chen, C. Chen, dan D. Feng, "Improving flash-based disk cache with Lazy Adaptive Replacement," dalam *Proc. IEEE Symposium on Mass Storage Systems and Technologies*, 2013.

[8] G. Graefe, "The five-minute rule twenty years later, and how flash memory changes the rules," dalam *Proc. ACM International Workshop on Data Management on New Hardware*, 2007.

[9] Y. Zhou, J. Philbin, dan K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," dalam *Proc. USENIX Annual Technical Conference*, 2001.

[10] T. Johnson dan D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," dalam *Proc. International Conference on Very Large Data Bases*, 1994.

[11] Umass, "Umass Trace Repository," University of Massachusetts Amherst, [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage.

[12] Google, "Go Programming Language," [Online]. Tersedia: https://golang.org/. [Accessed January 2020].