PENJADWALAN *GARBAGE COLLECTION* PADA RAID SSD UNTUK MEMINIMASI *COLLISION*

Laporan Tugas Akhir

Disusun sebagai syarat kelulusan tingkat sarjana

Oleh

Fadhil Imam Kurnia

NIM : 13515146

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO & INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

MEI 2019

PENJADWALAN *GARBAGE COLLECTION* PADA RAID SSD

UNTUK MEMINIMASI *COLLISION*

Laporan Tugas Akhir

Oleh

Fadhil Imam Kurnia

NIM : 13515146

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Draf Seminar Tugas Akhir II

di Bandung, pada tanggal 23 Mei 2019

Pembimbing I,	Pembimbing II,
A. Imam Kistijantoro ST, M.Sc, Ph.D.	Riza Satria Perdana ST, MT.
NIP. 19730809 200604 1 001	NIP. 19700609 199512 1 002

LEMBAR PERNYATAAN

Dengan ini saya menyatakan bahwa:

1. Pengerjaan dan penulisan Laporan Tugas Akhir ini dilakukan tanpa menggunakan bantuan yang tidak dibenarkan.
2. Segala bentuk kutipan dan acuan terhadap tulisan orang lain yang digunakan di dalam penyusunan laporan tugas akhir ini telah dituliskan dengan baik dan benar.
3. Laporan Tugas Akhir ini belum pernah diajukan pada program pendidikan di perguruan tinggi mana pun.

Jika terbukti melanggar hal-hal di atas, saya bersedia dikenakan sanksi sesuai dengan Peraturan Akademik dan Kemahasiswaan Institut Teknologi Bandung bagian Penegakan Norma Akademik dan Kemahasiswaan khususnya Pasal 2.1 dan Pasal 2.2.

Bandung, 23 Mei 2019

Fadhil Imam Kurnia

NIM. 13515146

ABSTRAK

PENJADWALAN *GARBAGE COLLECTION* PADA RAID SSD UNTUK MEMINIMASI *COLLISION*

Oleh

Fadhil Imam Kurnia

NIM : 13515146

Media penyimpanan *Solid State Drive* (SSD) sudah umum digunakan oleh masyarakat. Pada tahun 2021 diprediksi jumlah pengiriman SSD akan menyalip jumlah pengiriman *Hard Disk Drive* (HDD). Dalam operasinya, SSD perlu menjalankan proses GC untuk menghapus *invalid data* yang sudah tidak terpakai. Namun proses GC tersebut menimbulkan kenaikan latensi karena *request* I/O harus menunggu GC selesai sebelum *request* tersebut dapat dilayani. Dampak dari GC menjadi lebih buruk pada RAID SSD karena tidak hanya satu SSD yang terlibat di dalamnya. Dampak dari GC dapat dieliminasi dengan melakukan rekonstruksi data menggunakan paritas yang sudah umum digunakan pada RAID. Namun rekonstruksi hanya dapat dilakukan ketika dalam satu waktu hanya ada satu GC saja yang berlangsung, atau tidak ada *collision* antar GC dalam RAID. Dalam penelitian ini diusulkan strategi penjadwalan GCSync, GCSync+, dan GCLock yang dapat meminimasi *collision* tersebut pada RAID SSD. GCSync merupakan strategi penjadwalan yang menggunakan *time window*, untuk proses GC, yang berbeda-beda pada setiap SSD, sedangkan GCSync+ merupakan pengembangan dari GCSync dengan menambahkan *buffer time* antara dua *time window* untuk menghilangkan semua *collision*, sedangkan GCLock merupakan strategi penjadwalan dengan *lock* pada *controller* RAID. Pengujian yang dilakukan dengan modifikasi simulator SSDSim menunjukan hasil yang cukup baik. GCSync mampu mengurangi *GC collision* hingga 94,27% dan hanya menimbulkan kenaikan latensi sebesar 1,20 – 1,54 kali lipat. Bahkan pada salah satu *workload* yang digunakan untuk pengujian, GCSync mampu menurunkan latensi hingga 1,37 kali lipat. GCSync+ dan GCLock mampu menghilangkan semua *collision* pada semua pengujian, pengaruh strategi tersebut pada kenaikan latensi hanya sebesar 1,23 – 2,13 kali lipat.

Kata kunci: SSD, *garbage collection*, penjadwalan, RAID.

KATA PENGANTAR

Puji syukur kepada Tuhan Yang Maha Kuasa, karena atas rahmat dan karunia-Nya tugas akhir penulis yang berjudul "Penjadwalan *Garbage Collection* pada RAID SSD untuk Meminimasi *Collision*" dapat diselesaikan dengan baik. Tugas akhir ini disusun sebagai salah satu syarat untuk menyelesaikan program sarjana di Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung.

Penyusunan tugas akhir ini tidak lepas dari bimbingan, dukungan, doa, dan bantuan banyak pihak. Oleh karena itu penulis mengucapkan terima kasih kepada:

1. Bapak A. Imam Kistijantoro ST, M.Sc, Ph.D dan Bapak Riza Satria Perdana ST, MT selaku pembimbing yang telah membagikan ilmu dan pengalamannya untuk mendukung pengerjaan tugas akhir ini.
2. Prof. Haryadi S. Gunawi Ph.D dan Huaicheng Li, serta rekan-rekan di Laboratorium Garuda Ilmu Komputer (GIK Lab) yang telah mengenalkan penulis dalam dunia penelitian tentang media penyimpanan (*storage technology*) dan sistem terdistribusi.
3. Bapak Yudistira Dwi Wardhana Asnar ST, Ph.D selaku dosen wali yang mengarahkan penulis selama penulis menuntut ilmu di Program Studi Teknik Informatika.
4. Seluruh staf pengajar Program Studi Teknik Informatika yang telah memberikan banyak ilmu, baik keilmuan Informatika maupun non keilmuan, selama masa perkuliahan.
5. Bapak Suranto, Bapak Edi, Ibu Puji, Ibu Dedeh dan segenap staf serta karyawan Program Studi Teknik Informatika yang membantu banyak keperluan administrasi penulis.
6. Kedua orang tua penulis yang selalu mendoakan dan mendukung dari jauh selama proses studi penulis di Bandung.

7. Jauhar Arifin, Muhammad Hilmi Asyrofi, Martin Lutta Putra, dan segenap teman-teman seangkatan lainnya yang banyak memberikan bantuan dalam proses penyusunan tugas akhir ini, maupun selama proses perkuliahan.
8. Teman-teman Remaja Masjid Baitussalam dan Remaja Masjid Al-Haq yang memberikan keceriaan dan dukungan selama penulis menjalani studi di Bandung.
9. Fajar Fadhil Khoir, Abdurohman, dan Brian Ibnu Syam, selaku teman dan pengelola kafe Sans.Co yang menyediakan tempat serta fasilitas untuk pengerjaan tugas akhir ini.
10. Serta seluruh pihak yang telah membantu dan memberikan dukungan kepada penulis yang tidak dapat disebutkan satu persatu.

Penulis menyadari bahwa penyusunan laporan tugas akhir ini belum sempurna karena keterbatasan waktu, kemampuan, pengalaman, dan pengetahuan yang dimiliki oleh penulis. Oleh karena itu, penulis menerima kritik dan saran untuk perbaikan tugas akhir ini. Semoga tugas akhir ini dapat memberikan manfaat dan sumbangan dalam keilmuan informatika.

Bandung, Mei 2019

Penulis

DAFTAR ISI

ABSTRAK .... **iv**

KATA PENGANTAR .... **v**

DAFTAR ISI .... **vii**

DAFTAR LAMPIRAN .... **x**

DAFTAR GAMBAR .... **xi**

DAFTAR TABEL .... **xiii**

BAB I PENDAHULUAN **1**

I.1 Latar Belakang 1

I.2 Rumusan Masalah 3

I.3 Tujuan 3

I.4 Batasan Masalah 3

I.5 Metodologi 4

I.6 Sistematika Pembahasan 5

BAB II STUDI LITERATUR **7**

II.1 *Solid-State Drive* (SSD) 7

II.1.1 Arsitektur Internal 8

II.1.2 Operasi Penulisan dan Pengubahan Data 11

II.2 *Garbage Collection* (GC) 13

II.3 *Redundant Array of Independent Disk* (RAID) 14

II.4 Penelitian Terkait 15

II.4.1 Harmonia: Penjadwalan GC pada RAID SSD 15

II.4.2 Tiny-Tail Flash: Eliminasi Efek GC pada SSD 16

II.5 *Garbage Collection Collision* (GCC) pada RAID SSD 18

II.6 SSDSim: Simulator SSD Berbasis *Tracefile* 19

II.7 Simulator dan Emulator SSD Lainnya 21

BAB III ANALISIS DAN DESAIN RANCANGAN SOLUSI 23

III.1 Analisis Pentingnya Minimasi GC *Collision* 23

III.2 Analisis Dampak *GC Collision* terhadap *Response Time* 24

III.3 Penjadwalan GC pada RAID SSD 26

III.4 Desain Strategi Minimasi *Collision* 27

III.4.1 GCSync 27

III.4.2 GCSync+ 28

III.4.3 GCLock 29

BAB IV IMPLEMENTASI DAN PENGUJIAN 30

IV.1 Implementasi dengan SSDSim 30

IV.1.1 Menghasilkan Data Log GC 31

IV.1.2 Menandai I/O yang Terhalang Proses GC 33

IV.2 Implementasi Strategi Minimasi Collision 35

IV.2.1 Implementasi *Controller* RAID 35

IV.2.2 Implementasi GCSync 39

IV.2.3 Implementasi GCSync+ 40

IV.2.4 Implementasi GCLock 40

IV.3 Pembuatan Kakas untuk Pengujian 41

IV.3.1 Grafik Proses GC Selama Simulasi 41

IV.3.2 Grafik *GC Collison* 42

IV.3.3 Grafik CDF Latensi Proses Pembacaan Data 43

IV.4 Deskripsi *Workload* untuk Pengujian 44
IV.4.1 TPCC untuk Pengujian Basisdata 45
IV.4.2 *Developer Tools Release Server* (DTRS) 45
IV.4.3 MSN *Storage Metadata and File Server* (MSNFS) 46
IV.5 Hasil Pengujian Minimasi *Collision* 47
IV.5.1 Pengukuran Sebelum Implementasi 47
IV.5.2 Hasil Pengujian GCSync 48
IV.5.3 Hasil Pengujian GCSync+ 52
IV.5.4 Hasil Pengujian GCLock 56
IV.6 Evaluasi Hasil Pengujian GCSync, GCSync+, dan GCLock 59
BAB V KESIMPULAN DAN SARAN 61
V.1 Kesimpulan 61
V.2 Saran 62
DAFTAR REFERENSI 63

DAFTAR LAMPIRAN

Lampiran A. Grafik *GC Collision* dari Pengujian 65

Lampiran B. Grafik CDF Latensi *Read Request* 66

Lampiran C. Grafik Proses GC pada RAID SSD 67

DAFTAR GAMBAR

Gambar I.1 Variabilitas latensi pada SSD karena proses GC (Yan, 2017)............ 2

Gambar II.1. Tren pengiriman SSD dan HDD (Sumber: Statista.com)................. 8

Gambar II.2. Diagram blok arsitektur dalam SSD (Eshghi, 2018:20)................... 9

Gambar II.3. *Blok* dalam *flash chip* (Swanson, 2011) 9

Gambar II.4. Rantai *flash cells, flash chip, block*, dan *page* (Swanson, 2011)..... 10

Gambar II.5. Manajemen LBA dan PBA pada SSD (Eshghi, 2018) 11

Gambar II.6. Tabel konversi LBA ke alamat fisik dalam SSD (Swanson, 2011). 12

Gambar II.7. Mekanisme *Garbage Collection* (GC) (Micheloni, 2018) 13

Gambar II.8 Ilustrasi GC pada RAID SSD. (a) kondisi normal, dan (b) kondisi GC dengan Harmonia (Kim, 2011) 16

Gambar II.9 Eliminasi pengaruh GC dengan memanfaatkan paritas (Yan, 2017) 17

Gambar II.10 Ilustrasi *collision* pada RAID SSD 18

Gambar II.11 Deviasi rata-rata antara SSDSim dan purwarupa SSD (Hu, 2011) 19

Gambar II.12 *Cumulative Distribution Function* (CDF) dari *response time* SSDSim dan purwarupa SSD dengan berbagai *workload* (Hu, 2011)................................ 20

Gambar II.13 Contoh *tracefile* yang digunakan SSDSim 21

Gambar II.14. Cosmos OpenSSD, salah satu *hardware-based emulator* (Sumber: www.openssd.io)... 22

Gambar III.1 Potongan simulasi RAID5 dengan *workload* TPCC yang menunjukan proses GC di setiap *disk* 23

Gambar III.2 Contoh grafik CDF untuk pengujian.. 25

Gambar III.3 Contoh grafik GC *collision* untuk pengujian 26

Gambar III.4 Ilustrasi *time window* pada GCSync... 28

Gambar IV.1 Skema penggunaan SSDSim untuk simulasi SSD 30

Gambar IV.2 Contoh *log* latensi I/O yang dihasilkan pada akhir simulasi........... 31

Gambar IV.3 *Struct* gc_operation untuk menampung informasi sebuah proses GC dalam SSDSim 32

Gambar IV.4 Contoh *log* proses GC yang dihasilkan setelah simulasi 33

Gambar IV.5 Contoh *log request* I/O yang disimulasikan dengan *flag* meet_gc yang menandakan apakah *request* tersebut terhalang oleh GC atau tidak..................... 34

Gambar IV.6 *Struct* raid_info untuk memodelkan *controller* RAID dalam SSDSim 35

Gambar IV.7 Skema simulasi RAID dengan SSDSim 36

Gambar IV.8 Diagram alir proses kerja *controller* dalam SSDSim...................... 37

Gambar IV.9 *Struct* raid_sub_request untuk memodelkan *request* di SSD.......... 38

Gambar IV.10 Modifikasi struct ssd_info untuk GCSync 39

Gambar IV.11 Struct gclock_raid_info untuk memodelkan *lock* pada *controller* 40

Gambar IV.12 Contoh grafik GC untuk memvisualisasikan GC pada RAID 42

Gambar IV.13 Contoh grafik GC *Collision* untuk menunjukan collision pada RAID 43

Gambar IV.14 Contoh Grafik CDF latensi *request* I/O 44

Gambar IV.15 Grafik GC *Collision* sebelum implementasi 47

Gambar IV.16 Proses GC dalam simulasi sebelum integrasi 48

Gambar IV.17 Grafik CDF pada RAID sebelum implementasi 48

Gambar IV.18 Ilusttrasi terjadinya 2GC pada GCSync 51

DAFTAR TABEL

Tabel IV.1 Karekteristik *workload* yang digunakan untuk pengujian 45

Tabel IV.2 Parameter penting yang digunakan dalam simulasi............................ 46

Tabel IV.3 Proses GC sebelum dan sesudah implementasi GCSync.................... 49

Tabel IV.4 Jumlah GC *collision* sebelum dan setelah implementasi GCSync 50

Tabel IV.5 Grafik CDF latensi sebelum dan sesudah implementasi GCSync 52

Tabel IV.6 Proses GC sebelum dan sesudah implementasi GCSync+ 53

Tabel IV.7 Jumlah GC *collision* sebelum dan setelah implementasi GCSync+ ... 54

Tabel IV.8 Grafik CDF latensi sebelum dan sesudah implementasi GCSync+.... 55

Tabel IV.9 Proses GC sebelum dan sesudah implementasi GCLock 56

Tabel IV.10 Jumlah GC *collision* sebelum dan setelah implementasi GCLock ... 57

Tabel IV.11 Grafik CDF latensi sebelum dan sesudah implementasi GCLock.... 58

BAB I
PENDAHULUAN

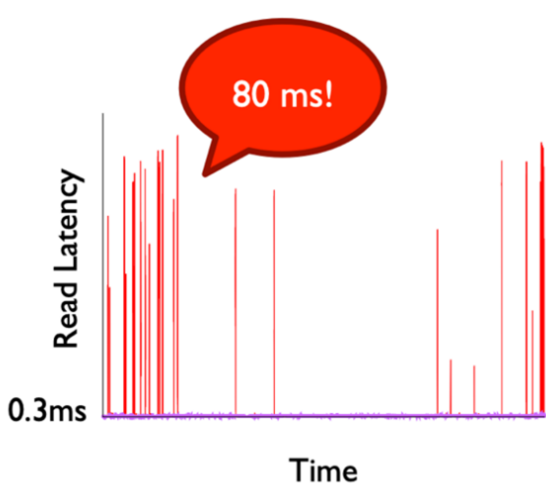
I.1 Latar Belakang

Teknologi penyimpanan *Solid State Drive* (SSD) sudah umum digunakan oleh masyarakat karena kecepatan pengaksesan data yang dimilikinya. Saat ini saja, kecepatan dari SSD yang menggunakan *interface* NVMe dapat mencapai 3GB/s, sedangkan kecepatan SSD dengan interface SATA mencapai 278MB/s, sekitar 350 kali lipat kecepatan *Hard Disk Drive* (HDD) (Xu, 2015). Data dari Statista menunjukan tren penurunan pengiriman HDD, dan tren kenaikan pengiriman SSD, bahkan pada tahun 2021 diprediksikan bahwa jumlah pengiriman SSD akan melebihi jumlah pengiriman HDD. Walaupun saat ini harga SSD masih tergolong mahal, namun dengan adanya produksi masal harga penyimpanan per *byte* menggunakan SSD sudah semakin turun. Bukan tidak mungkin kedepannya harga SSD dapat menyerupai harga HDD.

Walaupun SSD memiliki kecepatan yang relatif tinggi atau latensi yang rendah, mekanisme *Garbage Collection* (GC) yang ada dalam SSD dapat mengakibatkan proses I/O menjadi lebih lambat daripada biasanya. Proses *read* atau *write* pada SSD harus menunggu proses GC selesai terlebih dahulu, sebelum proses I/O tersebut dapat dilanjutkan. Hal tersebut dapat menjadi lebih parah jika SSD digunakan untuk RAID. Umumnya proses I/O pada RAID melibatkan lebih dari satu SSD, sehingga latensi dari proses I/O sangat dipengaruhi oleh SSD yang paling lambat. Jika salah satu SSD yang terlibat dalam proses I/O harus melakukan GC maka proses I/O harus menunggu SSD tersebut selesai melakukan GC, walaupun SSD lainnya yang terlibat dalam proses I/O tersebut tidak sedang melakukan GC. Oleh karena itu, dengan adanya mekanisme GC dalam SSD, latensi yang biasanya rendah pada SSD, terkadang dapat meningkat drastis, bahkan hingga 100 kali lipat (Dean, 2013). Jika biasanya proses pembacaan pada SSD hanya memerlukan waktu kurang dari 1 ms, terkadang proses pembacaan dapat mencapai 100 ms. Oleh karena itu proses GC menjadi salah satu faktor penyumbang besarnya variabilitas latensi pada RAID SSD. Seperti ditunjukan pada Gambar I.1, proses *read* yang biasanya

hanya memerlukan 0.3 ms, dapat menjadi 80 ms ketika proses GC berjalan (Yan, 2017).

Adanya variabilitas latensi pada RAID SSD dapat memberikan dampak pada layanan yang berjalan di atasnya. Penelitian sebelumnya (Card, 1991) menunjukan bahwa layanan yang memberikan tanggapan (*response*) dengan cepat, dibawah 100 ms, lebih memberikan pengalaman yang natural kepada penggunanya dibandingkan dengan layanan yang lebih lama memberikan tanggapan. Selain itu, perkembangan layanan *online* juga semakin meningkatkan kebutuhan terhadap layanan yang mampu beroperasi dengan cepat, walaupun harus melakukan pemrosesan dengan data yang terdistribusi. Contohnya seperti mesin pencari yang dapat memberikan hasil pencarian kurang dari 100 ms setiap kali satu huruf diketikkan.



Gambar I.1 Variabilitas latensi pada SSD karena proses GC (Yan, 2017)

Sebelumnya sudah ada peneliti yang berusaha mengeliminasi efek dari proses GC dalam SSD dengan memanfaatkan redundansi (Skourtis, 2013) atau paritas (Yan, 2017). Mekanime eliminasi GC pada SSD dengan paritas dapat dilakukan secara optimal jika dalam satu waktu hanya ada satu SSD saja yang melakukan proses GC, karena pada saat itu data yang belum dapat dibaca karena ada proses GC dapat

direkonstruksi menggunakan paritas dan data yang ada pada SSD lainnya. Dengan kata lain, kita harus meminimasi terjadinya *Garbage Collection Collision* (GCC) atau *collision* pada RAID SSD untuk dapat mengeliminasi efek dari GC dan membuat latensi dari SSD selalu rendah. Oleh karena itu, pada tugas akhir ini dilakukan penelitian untuk meminimasi *collision* pada RAID SSD dengan menerapkan suatu algoritma penjadwalan GC.

I.2 Rumusan Masalah

Berdasarkan latar belakang yang dijelaskan sebelumnya, rumusan masalah yang akan diselesaikan dalam tugas akhir ini adalah sebagai berikut:

1. Bagaimana strategi minimasi *Garbage Collection Collision* (GCC) atau *collision* pada RAID SSD?
2. Bagaimana cara implementasi penjadwalan GC pada RAID SSD untuk meminimasi *collision*?

I.3 Tujuan

Tujuan akhir dari penelitian ini adalah sebagai berikut:

1. Mendefinisikan strategi yang dapat digunakan untuk dapat meminimasi *collision* pada RAID SSD sehingga proses rekonstruksi dapat dilakukan secara optimal ketika salah satu SSD sedang menjalankan proses GC.
2. Mengimplementasikan strategi minimasi *collision* yang sudah didefinisikan, serta mengukur efektivitas strategi tersebut.

I.4 Batasan Masalah

Terdapat beberapa batasan lingkup pengerjaan penelitian dalam tugas akhir ini. Batasan-batasan tersebut adalah.

1. Implementasi minimasi *collision* hanya dilakukan pada simulator SSD yang sudah divalidasi sebelumnya, atau sudah dipastikan dapat mendekati SSD sesungguhnya.

2. Jenis RAID yang akan digunakan untuk pengujian adalah RAID 5 karena pada jenis tersebut terdapat paritas yang dapat digunakan untuk rekonstuksi data.

I.5 Metodologi

Untuk menyelesaikan penelitian dalam tugas akhir ini, metodologi pengerjaan yang dilakukan adalah sebagai berikut.

1. Eksplorasi Simulator SSD

   Karena penelitian yang dilakukan menggunakan simulator SSD, maka perlu dilakukan eksplorasi terhadap simulator yang akan digunakan. Perlu dipelajari bagaimana simulator memroses setiap permintaan I/O dan melakukan mekanisme GC. Selain itu, juga perlu diamati data statistik apa saja yang dihasilkan dari simulasi yang dilakukan.

2. Perancangan dan Persiapan

   Setelah memahami simulator yang akan digunakan, kemudian perlu dilakukan perancangan penelitian dengan menentukan statistik apa saja yang akan digunakan untuk melihat efek dari solusi yang diberikan. Selain itu dimungkinkan juga dilakukan modifikasi terhadap simulator agar dapat menyimulasikan RAID SSD dengan berbagai macam *workload* simulasi. Pada tahap ini juga dilakukan perancangan mekanisme yang akan digunakan untuk meminimasi *collision* pada RAID SSD.

3. Implementasi

   Tahap implementasi merupakan tahap inti dari penelitian yang dilakukan. Pada tahap ini dilakukan implementasi mekanisme minimasi GCC yang sudah dirancang pada tahap sebelumnya.

4. Pengujian

   Setelah implementasi selesai dilakukan, selanjutnya akan dilakukan pengujian dengan membandingkan statistik hasil simulasi sebelum dilakukan implementasi dan sesudah dilakukan implementasi. Jika hasil

yang dicapai belum terlalu memuaskan, tahap perancangan, implementasi, dan pengujian dapat dilakukan beberapa kali untuk mendapatkan hasil yang dirasa cukup baik.

5. Penarikan Kesimpulan

   Setelah melakukan pengujian dan didapatkan hasil yang dirasa cukup dalam rentang waktu yang sudah disesuaikan, tahap terakhir dari penelitian ini adalah penarikan kesimpulan. Kesimpulan akan dibuat secara ringkas dan jelas sehingga mudah dipahami oleh pembaca.

I.6 Sistematika Pembahasan

Laporan Tugas Akhir ini terdiri dari lima bab yang berisi penjelasan penelitian yang dilakukan, diawali dengan pendahuluan serta diakhiri dengan kesimpulan dan saran. Pada Bab 1 dijelaskan pendahuluan tentang penelitian yang dilakukan dalam tugas akhir ini. Terlebih dahulu akan dijelaskan latar belakang permasalahan, kemudian dilanjutkan dengan rumusan masalah yang ingin dijawab. Setelah itu juga dipaparkan tujuan, batasan masalah, serta metodologi yang digunakan dalam penelitian.

Bab 2 berisi ringkasan studi literatur yang berisi beberapa dasar teori terkait dengan penelitian yang dilakukan. Pada bab tersebut dijelaskan sekilas mengenai *Solid-State Drive* (SSD), proses *Garbage Collection* (GC) dalam SSD, *Redundant Array of Independent Disk* (RAID), dan *Garbage Collection Collision* (GCC) atau *collision* dalam RAID SSD. Selain itu juga dipaparkan penelitian sebelumnya yang terkait dengan penelitian ini.

Selanjutnya pada Bab 3 dijelaskan lebih detail tentang *GC collision* dalam RAID SSD. Setelah itu terdapat penjelasan tentang strategi yang digunakan untuk meminimasi *GC collision*. Secara garis besar terdapat dua strategi minimasi *collision* yang dijelaskan dalam Bab 3.

Pada Bab 4 dijelaskan simulasi yang dilakukan untuk memvalidasi strategi yang sudah dijelaskan sebelumnya. Penggunaan simulator dan modifikasi simulator juga

dijelaskan dalam bab ini. Pengukuran efektifitas implementasi strategi minimasi *collision* dilakukan dengan menggunakan beberapa skenario *workload*.

Terakhir, pada Bab 5 dipaparkan kesimpulan dan saran dari penelitian yang dilakukan. Kesimpulan disampaikan dalam bentuk poin-poin penting. Saran-saran yang disampaikan diharapkan dapat menjadi bahan untuk penelitiaan selanjutnya terkait dengan penjadwalan GC pada RAID SSD.

BAB II
STUDI LITERATUR

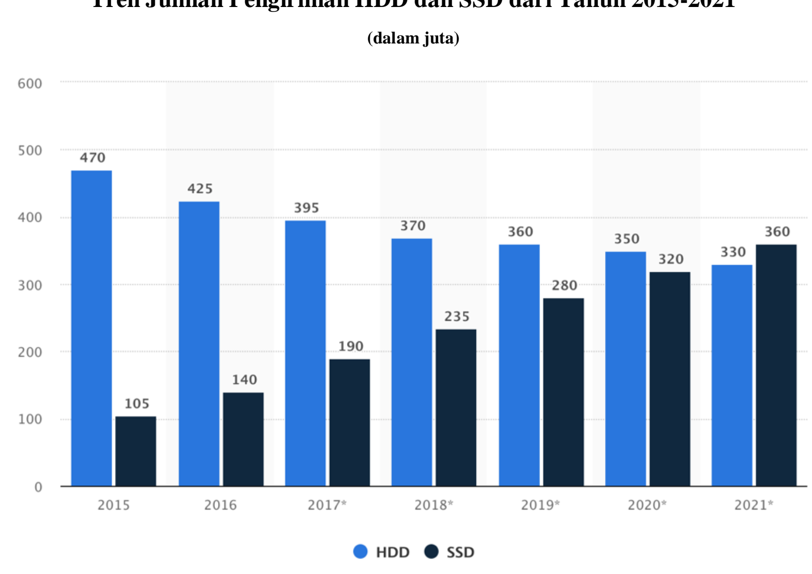
Untuk menunjang penelitian yang dilakukan dalam Tugas Akhir ini, pada bab ini dijelaskan beberapa konsep yang berkaitan dengan SSD dan *Garbage Collection* (GC) pada SSD. Konsep dasar mengenai SSD dan komponen di dalamnya dijelaskan pada subbab II.1, kemudian dilanjutkan dengan pembahasan proses GC dalam SSD pada subbab II.2. Subbab selanjutnya menjelaskan mengenai *Garbage Collection Collision* (GCC) yang dapat terjadi pada RAID SSD. Terdapat juga subbab yang menjelaskan mengenai penelitian yang sudah dilakukan sebelumnya terkait dengan GC pada SSD.

II.1 *Solid-State Drive* (SSD)

Seiring dengan berjalannya waktu, teknologi penyimpanan data yang bersifat *non-volatile* semakin berkembang. *Solid-State Drive* (SSD) merupakan teknologi yang relatif baru untuk penyimpanan data. Sesuai namanya, SSD tidak memerlukan komponen yang bergerak, berbeda dengan *Hard Disk* (HDD) yang memerlukan putaran piringan magnetik. Karena tidak memerlukan komponen yang bergerak, dibandingkan dengan HDD, SSD dapat memberikan waktu akses data yang jauh lebih singkat, penggunaan energi yang lebih sedikit, dan ketahanan terhadap guncangan dan getaran. SSD juga memiliki densitas penyimpanan yang lebih padat (Kim, 2011). Hal tersebut membuat penguatan lantai rak-rak pada ruangan *server* dapat dikurangi, dan biaya pengiriman media penyimpanan juga menjadi lebih murah. Adanya proses produksi masal, inovasi penyusunan cip, dan efisiensi dalam produksi, juga membuat harga per *gigabyte* (GB) penyimpanan pada SSD semakin murah.

Karena beberapa keunggulannya tersebut, SSD mulai banyak digunakan untuk *website server*, komputasi awan, *data mining*, serta penggunaan pribadi. Berdasarkan data dari Statista yang dipublikasikan pada Mei 2017, jumlah pengiriman HDD terus menurun, sebaliknya jumlah pengiriman SSD terus

meningkat, diprediksikan jumlah pengiriman SSD akan melampaui HDD pada tahun 2021.



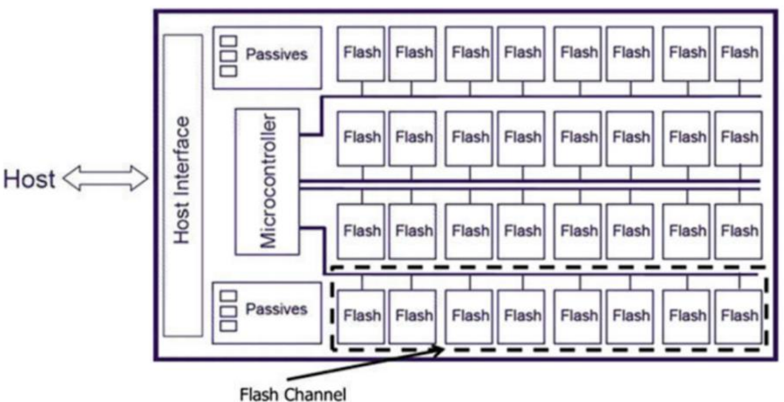
Gambar II.1. Tren pengiriman SSD dan HDD (Sumber: Statista.com)

Arsitektur dalam SSD sangat berbeda dengan arsitektur HDD yang menggunakan piringan magnetik untuk menyimpan biner 1 dan 0. Dalam SSD penyimpanan data dilakukan dengan *flash cell* dalam *flash chip memory* / *flash chip*. Penjelasan lebih detail mengenai arsitektur SSD dapat dilihat pada subbab II.1.1, kemudian penjelasan mengenai proses penulisan dan pengubahan data dalam SSD dapat dilihat pada subbab II.1.2.

II.1.1 Arsitektur Internal

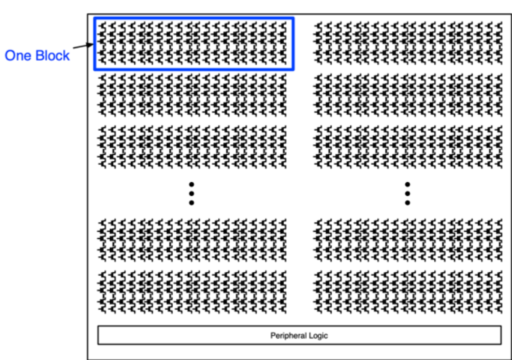
Untuk mengakomodir penyimpanan data dalam jumlah besar, sebuah SSD dapat memiliki puluhan hingga ratusan *flash chip* didalamnya. Sebelumnya, *flash chip* sudah banyak digunakan sebagai media penyimpanan di PDA, *flash disk*, telepon genggam, dan perangkat lainnya (Eshghi, 2018). Namun *flash chip* yang digunakan dalam SSD memiliki kualitas yang lebih baik, seperti umur yang lebih panjang, densitas yang lebih besar, dan akses data yang lebih cepat. Saat ini ada beragam

jenis *flash chip* yang sudah dikembangkan, diantaranya adalah *NAND*, *NOR*, dan *AND flash*. Mayoritas SSD yang ada saat ini menggunakan jenis *NAND flash* karena ukurannya yang lebih kecil dan proses penulisan maupun penghapusan data yang lebih cepat dibandingkan jenis lainnya. Untuk lebih jelasnya, posisi *flash chip* dalam SSD dapat dilihat di diagram blok SSD pada Gambar II.2.

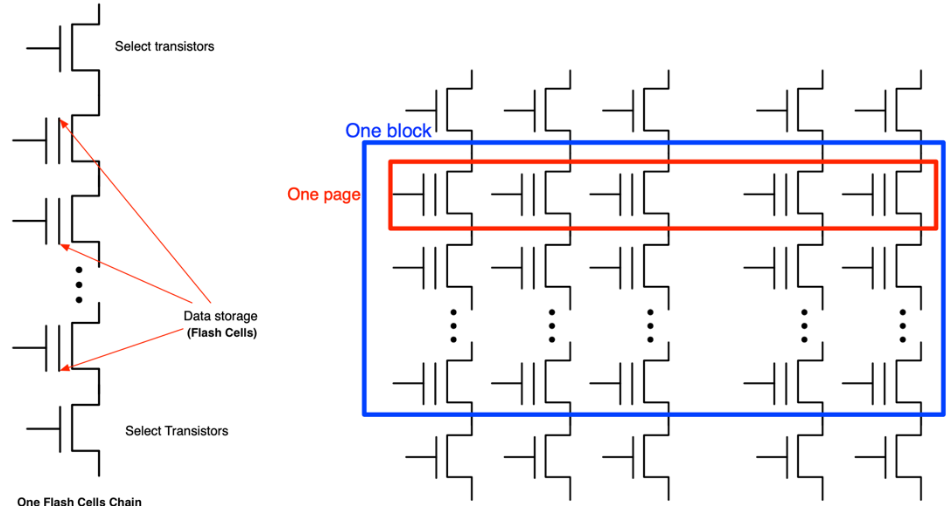


Gambar II.2. Diagram blok arsitektur dalam SSD (Eshghi, 2018:20)

Pada setiap *flash chip* terdapat banyak *flash cell* yang dikelompokkan menjadi *block* dan *page*. Sebuah *flash chip* memiliki sejumlah *block*, dan untuk setiap *block* terdapat sejumlah *page*. Gambaran dalam *flash chip* dapat dilihat pada Gambar II.3 dan Gambar II.4. Data yang disimpan dalam sebuah *page* biasanya berukuran 256 bytes, 512 bytes, hingga 1024 bytes.



Gambar II.3. *Blok* dalam *flash chip* (Swanson, 2011)



Gambar II.4. Rantai *flash cells, flash chip, block*, dan *page* (Swanson, 2011)

Terdapat beberapa metode yang dapat digunakan untuk menyimpan data biner dalam *flash cell.* Metode *Single-Level Cell* (SLC) menyimpan satu bit (1 dan 0) pada setiap *cell*, sedangkan *Multiple-Level Cell* (MLC) menyimpan dua bit untuk setiap *cell* (00, 01, 10, 11), terdapat juga metode *Triple-Level Cell* (TLC) dan *Quad-Level Cell* (QLC). Semakin banyak bit yang disimpan dalam sebuah *Cell* maka akan semakin sulit bagi SSD untuk membedakan representasi satu nilai dengan nilai lainnya.

Selain komponen utama *flash chip*, komponen lainnya yang ada dalam SSD adalah *host interface*. Komponen tersebut menghubungkan SSD dengan komputer (*host)* menggunakan protokol standar industri yang sudah disepakati, seperti SATA, PCIe, SAS, NVMe, dan lain sebagainya. Komponen ini menangani permasalahan kompatibilitas *logic* dan elektrik antara SSD dan komputer.

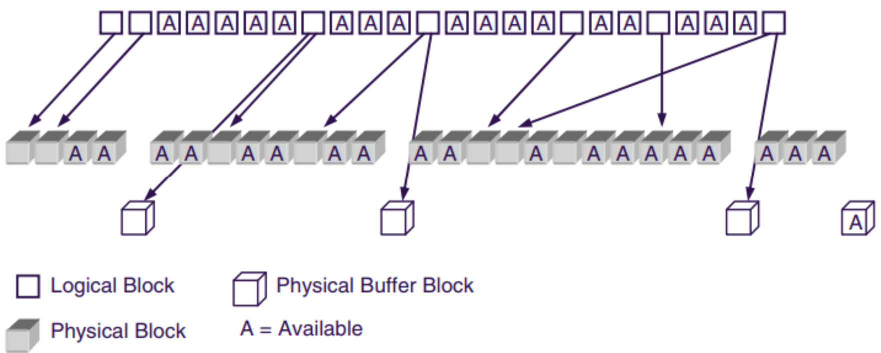
Komponen *microcontroller* merupakan salah satu komponen penting dalam SSD. Adanya *microcontroller* tersebut membuat SSD dapat berkomunikasi menggunakan protokol yang biasanya digunakan pada HDD. Hal tersebut membuat banyak sistem operasi dapat mengakomodir HDD dan SSD. Di dalam *microcontroller* juga terdapat program yang berfungsi sebagai *Flash Translation*

Layer (FTL), *Garbage Collection Management*, dan *Bad Block Management* (Micheloni, 2017). FTL berfungsi untuk menerjemahkan *Logical Block Address* (LBA) yang digunakan oleh komputer menjadi lokasi memori sebenarnya dalam fisik SSD (*Physical Block Address* atau PBA).

Komponen lainnya yang mungkin ada dalam SSD adalah sensor suhu yang digunakan untuk managemen energi, serta *quartz* untuk *clocking* yang lebih baik. SSD juga memiliki *fast DDR memory* yang digunakan untuk *data caching*. Proses *caching* dapat digunakan saat ada permintaan penulisan data, memori DDR dapat digunakan untuk menampung data sebelum data tersebut dituliskan ke *flash chip*, hal tersebut membuat pengubahan data menjadi lebih cepat serta dapat mengurangi penggunaan *flash chip* yang dapat mengurangi umurnya.

II.1.2 Operasi Penulisan dan Pengubahan Data

Dalam arsitektur SSD, terdapat komponen *Flash File System* (FFS) yang membuat SSD dapat diakses seperti media penyimpanan magnetik yang kontigu (Eshghi, 2018). Komputer menganggap penulisan dan pembacaaan data dilakukan secara langsung pada *block* yang ditujunya, padahal SSD bisa saja menempatkan LBA yang bersebelahan pada bagian memori (PBA) yang berbeda. Manajemen konversi alamat tersebut dapat dilihat pada Gambar II.5. Hal tersebut dilakukan karena keterbatasan pada teknologi *flash cell* yang tidak dapat melakukan proses penulisan ulang (*overwrite*) secara langsung.



Gambar II.5. Manajemen LBA dan PBA pada SSD (Eshghi, 2018)

LBA	Physical Page Address	
0	Block 5	Page 7
2k	Block 27	Page 0
4k	Block 10	Page 2

Gambar II.6. Tabel konversi LBA ke alamat fisik dalam SSD (Swanson, 2011)

Jika ada permintaan pembacaan data (*read*) pada suatu *block address* tertentu, SSD akan mencari alamat sebenarnya pada tabel translasi, kemudian melakukan proses pembacaan data pada alamat yang sebenarnya tersebut. Hal tersebut dilakukan oleh FTL yang sudah diprogram dalam *microcontroller*. Tabel yang digunakan untuk konversi juga disimpan pada *flash chip*, sama seperti data lainnya, namun lokasinya sudah ditentukan sebelumnya. Jika ada permintaan penulisan data (*write*), proses pencarian alamat sebenenarnya oleh FTL juga akan dilakukan.

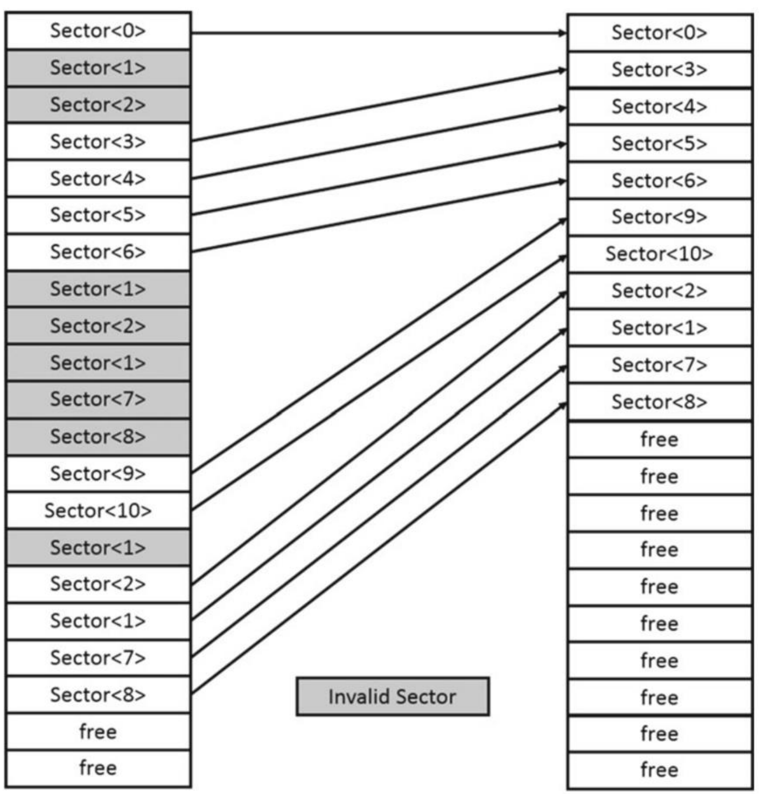
Berbeda dengan *sector* pada HDD yang dapat ditulis ulang secara langsung, operasi *write* pada SSD hanya dapat dilakukan pada *page* yang masih kosong. Jika komputer melakukan *overwrite* pada suatu LBA yang sama, maka SSD akan menempatkan data baru di *page* yang berbeda dari *page* yang digunakan untuk menyimpan data sebelumnya, kemudian *page* yang sebelumnya akan ditandai sebagai *invalid page*. Sebuah *invalid page* baru dapat digunakan untuk menulis data baru setelah dilakukan proses penghapusan (*delete*). Proses *write* juga harus memperhatikan jumlah *write* yang sudah dilakukan pada sebuah *page*, hal ini karena *flash chip* memiliki batas maksimum jumlah *write* yang dapat dilakukan sebelum *flash* tersebut menjadi kurang efisien untuk digunakan *(bad block*). Hal ini juga membuat SSD sebisa mungkin membuat penulisan data seimbang di setiap *flash chip*, sehingga sisa umur dari semua *flash* dapat relatif sama, mekanisme tersebut biasa dikenal dengan sebutan *wear leveling*.

Jika proses *read* dan *write* dapat dilakukan pada tingkat *page*, proses *delete* hanya bisa dilakukan pada tingkat *block*. Ini berarti dalam sekali penghapusan, sejumlah

page akan secara sekaligus dihapus datanya. Mekanisme penghapusan ini biasa dikenal dengan istilah *garbage collection* (GC) yang akan diterangkan pada subbab II.2.

II.2 *Garbage Collection* (GC)

Mekanisme *wear leveling* sangat tergantung dengan ketersediaan *block* kosong sebagai tempat penulisan data yang baru. Seiring dengan penggunaan SSD, jumlah *free page* akan semakin berkurang, dan jumlah *block* yang *sparse* juga akan bertambah. Pada batas tertentu proses *garbage collection* (GC) akan diaktifkan untuk menambah jumlah *free page*.



Gambar II.7. Mekanisme *Garbage Collection* (GC) (Micheloni, 2018)

Proses GC dilakukan dengan cara menghapus *block* yang semua *page* didalamnya memuat *invalid data*. Jika dalam sebuah *block* masih terdapat *page* yang berisi *valid data*, maka terlebih dahulu data tersebut dipindahkan ke *page* pada *block* lainnya,

proses ini dapat dilihat pada Gambar II.7. Setelah pemindahan tersebut, proses penghapusan dapat dilakukan pada tingkat *block*.

Walaupun SSD dapat melakukan pengaksesan data dengan cepat, terkadang proses GC yang berjalan dalam SSD dapat membuat proses pembacaan data menjadi 100 kali lebih lambat dari biasanya (Dean, 2013). Hal tersebut dapat terjadi karena saat proses GC berlangsung, SSD tidak bisa melayani permintaan pembacaan data (*read*). Sebuah proses *read* yang menarget sebuah *flash chip* harus menunggu proses GC selesai dilakukan pada *chip* tersebut sebelum proses *read* dapat dilakukan. Beberapa SSD bahkan melakukan penghalangan (*blocking*) di tingkat *channel,* ini berarti proses I/O (*read* atau *write*) tidak dapat dilakukan pada semua *flash* yang ada pada satu *channel*, selama salah satu *flash* dalam *channel* tersebut sedang menjalankan proses GC. Inilah yang membuat variansi latensi dari proses *read* pada SSD menjadi besar.

II.3 *Redundant Array of Independent Disk* (RAID)

Konsep tentang *Redundant Array of Independent Disk* atau *Redundant Array of Inexpensive Disk* (RAID) diperkenalkan pada sekitar tahun 1980an. Teknologi RAID ini memungkinkan untuk dilakukannya penyusunan banyak *disk* menjadi satu sistem, sehingga dapat diperoleh media penyimpanan dengan kapasitas yang lebih besar, dan kinerja yang lebih baik. Kinerja yang lebih baik didapatkan karena proses I/O dapat dilakukan secara paralel pada setiap *disk* yang berbeda (Perumal, 2004).

Salah satu implementasi RAID yang popular adalah RAID 5. Pada RAID 5 digunakan paritas untuk membenarkan data jika terjadi *error*. Penggunaan paritas dapat membantu dalam rekonstruksi data yang hilang, selain itu ruang penyimpanan tambahan yang dibutuhkan untuk menambahkan paritas juga tidak terlalu besar, dibandingkan dengan melakukan duplikasi data seutuhnya.

RAID 5 yang menggunakan banyak HDD sudah banyak diimplementasikan, saat ini orang-orang mulai mencoba menggunakan SSD untuk menyusun RAID. Dengan menyusun beberapa SSD, diharapkan performa yang didapatkan akan

menjadi jauh lebih baik. Namun sebenarnya kecepatan pengaksesan data pada RAID ditentukan oleh *disk* yang paling lambat diantara semua *disk* yang menyusun RAID tersebut. Permasalahan tersebut lebih parah terjadi pada RAID yang menggunakan SSD (RAID SSD), karena terdapat mekanisme GC didalam SSD. Jika ada salah satu SSD yang sedang menjalankan proses GC, maka suatu proses dalam RAID tersebut harus menunggu GC tersebut selesai, walaupun proses I/O pada *disk* yang lainnya sudah selesai dilakukan. Jika pada sebuah SSD variansi latensinya sudah besar, maka pada RAID SSD variansi dari latensi tersebut dapat menjadi lebih besar lagi.

Beberapa penelitian sebelumnya mencoba memanfaatkan paritas yang ada pada RAID untuk meminimalisir permasalahan tersebut (Yan, 2017). Jika sebuah *disk* dalam RAID SSD sedang menjalankan proses GC, maka *controller* dari RAID dapat merekonstuksi data pada SSD tersebut, terlebih lagi proses rekonstruksi dengan paritas dapat dilakukan lebih cepat dibandingkan harus menunggu proses GC selesai. Solusi ini dapat membuat variansi pada RAID SSD dapat lebih ditekan.

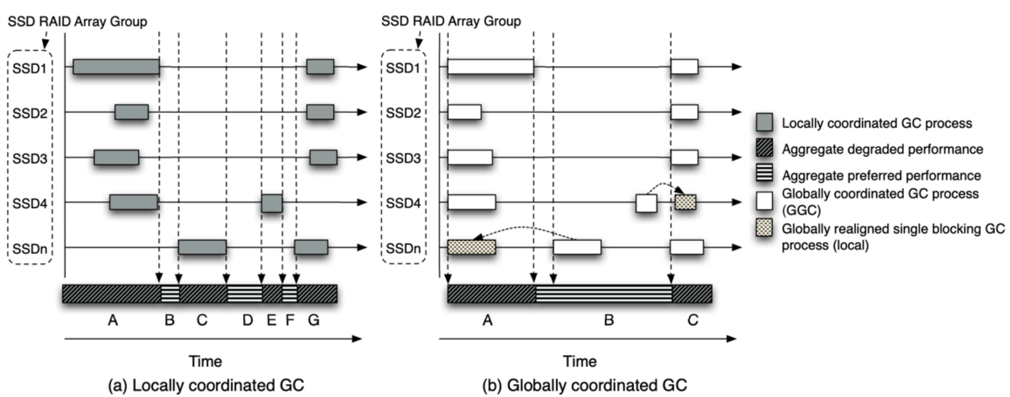
II.4 Penelitian Terkait

II.4.1 Harmonia: Penjadwalan GC pada RAID SSD

Beberapa penelitian sebelumnya sudah pernah dilakukan untuk meminimalisir efek dari GC pada RAID SSD. Salah satu penelitian tersebut adalah Harmonia yang diusulkan oleh Kim. Harmonia merupakan metode penjadwalan yang diusulkan untuk mengoordinir GC yang berjalan dalam RAID SSD, mekanisme tersebut disebut dengan istilah *Globally Coordinated Garbage Collector* (GGC).

Pada penelitian tersebut, GGC dilakukan dengan mengupayakan supaya GC pada setiap *disk* dalam RAID berjalan pada waktu yang bersamaan. c*ontroller* pada RAID dapat memerintahkan setiap *disk* yang terhubung dengannya untuk melakukan GC secara serempak, hal tersebut dilakukan ketika salah satu *disk* sudah mencapai *threshold* dan harus melakukan GC. Ilustrasi penjadwalan GC pada Harmonia dapat dilihat pada Gambar II.8. Implementasi Harmonia dilakukan pada simulator DiskSim yang sudah dimodifikasi. Terdapat dua komponen untuk

menjalankan Harmonia, yaitu O-RAID dan O-SSD. O-RAID adalah RAID *controller* yang sudah dioptimasi sehingga dapat mendukung penjadwalan GC, sedangkan O-SSD adalah SSD yang dapat beroperasi optimal dalam konfigurasi RAID.



Gambar II.8 Ilustrasi GC pada RAID SSD. (a) kondisi normal, dan (b) kondisi GC dengan Harmonia (Kim, 2011)

Ketika GC berjalan serempak pada setiap *disk*, latensi dari proses I/O akan menjadi tinggi, namun hal tersebut hanya berlangsung sementara. Selebihnya RAID dapat memberikan kinerja yang lebih baik, yaitu dengan latensi yang rendah. Penerapan metode tersebut mampu mengurangi rata-rata latensi hingga 69% (Kim, 2011).

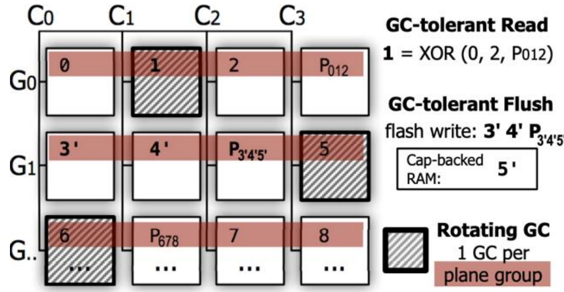
II.4.2 Tiny-Tail Flash: Eliminasi Efek GC pada SSD

Penelitian Tiny-Tail Flash atau ttFlash berusaha mengiliminasi efek dari GC dengan cara rekonstruksi menggunakan paritas (Yan, 2017). Paritas yang biasanya berfungsi untuk merekonstruksi data jika terjadi kesalahan, juga dimanfaatkan jika salah satu *plane* tidak dapat diakses karena ada proses GC yang sedang berlangsung pada *plane* tersebut. Metode tersebut dapat dilakukan jika paling banyak hanya ada satu proses GC yang sedang berlangsung pada suatu *plane group*, penjadwalan GC seperti itu disebut dengan istilah *Rotating Garbage Collection* (RGC).

Ilustrasi dari metode ini dapat dilihat pada Gambar II.9, pembacaan data pada *plane* 0, 1, dan 2 secara sekaligus terhalang karena pada *plane* 1 proses GC sedang berlangsung. Mekanisme ttFlash memungkinkan data pada plane 1 direkonstruksi dengan menggunakan paritas, yaitu menggunakan operasi

$$\texttt{XOR}(0,\ 2,\ P_{012})$$

dengan P_{012} adalah paritas dari gabungan *plane* 0, 1, dan 2. Adanya rekonstruksi tersebut membuat proses pembacaan tidak harus menunggu proses GC selesai dilakukan. Jika pada SSD biasa, proses GC dapat menyebabkan perlambatan hingga 5 - 138 kali, mekanisme eliminasi GC ini hanya menyebabkan perlambatan 1,0 - 2,6 kali saja.

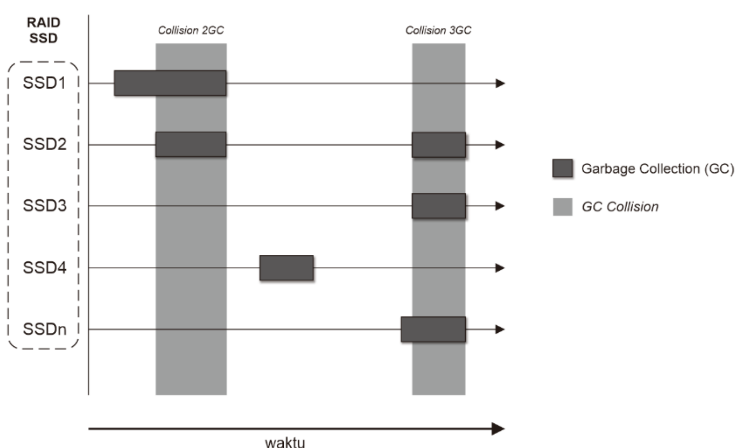


Gambar II.9 Eliminasi pengaruh GC dengan memanfaatkan paritas (Yan, 2017)

Penelitian yang dilakukan oleh Yan dapat juga diterapkan dalam RAID SSD, terutama RAID 5 yang juga menggunakan paritas didalamnya. Namun mekanisme pada SSD tersebut sangat bergantung pada RGC. Secara sederhana, RGC pada RAID SSD dapat dilakukan jika dalam suatu RAID sebisa mungkin hanya ada satu SSD saja yang sedang menjalankan proses GC. Terlebih lagi proses rekonstruksi data dapat dilakukan relatif lebih cepat jika dibandingkan dengan harus menunggu proses GC selesai.

II.5 *Garbage Collection Collision* (GCC) pada RAID SSD

Garbage Collection Collision (GCC) atau *collision* pada RAID SSD terjadi ketika ada lebih dari satu *disk* yang menjalankan operasi GC dalam waktu yang bersamaan. Hal tersebut dapat membuat kinerja dari RAID SSD menurun, karena *controller* RAID harus menunggu lebih dari satu proses GC untuk dapat melanjutkan proses I/O. Ilustrasi *collision* pada RAID SSD dapat dilihat pada Gambar II.8, pada bagian kiri terdapat *collision* karena 2 GC terjadi secara bersamaan, dan pada bagian kanan gambar tersebut terdapat *collision* karena 3 GC terjadi pada waktu yang beririsan. *Collision* tersebut menjadi kendala ketika *controller* RAID ingin melakukan rekonstruksi dengan paritas, seperti dalam konfigurasi ttFlash. Ini karena beberapa algoritma rekonstruksi menggunakan paritas maksimal hanya mengizinkan satu *disk* saja yang *offline* atau mati.

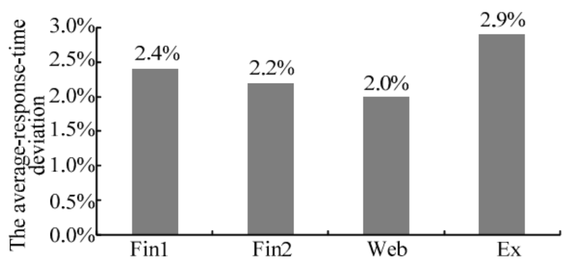


Gambar II.10 Ilustrasi *collision* pada RAID SSD

Misalkan pada mode RAID 5, dengan 4 *disk* berisi data dan 1 *disk* berisi paritas, maka rekonstruksi hanya dapat dilakukan jika maksimal hanya satu disk saja yang sedang mengalami gangguan. Kasus ideal untuk meminimalisir *collision* pada konfigurasi tersebut terjadi ketika dalam satu waktu hanya ada maksimal satu proses GC saja yang berjalan.

II.6 SSDSim: Simulator SSD Berbasis *Tracefile*

Kebanyakan penelitian tentang SSD dan RAID SSD dilakukan dengan menggunakan simulator atau emulator. Hal tersebut dilakukan karena susahnya melakukan implementasi rancangan desain baru SSD pada perangkat keras secara langsung. Oleh karena itu banyak simulator atau emulator yang dikembangkan untuk keperluan penelitian SSD. Salah satu simulator yang cukup populer dan sudah teruji adalah SSDSim yang dikembangkan oleh Yang Hu menggunakan bahasa pemrograman C.

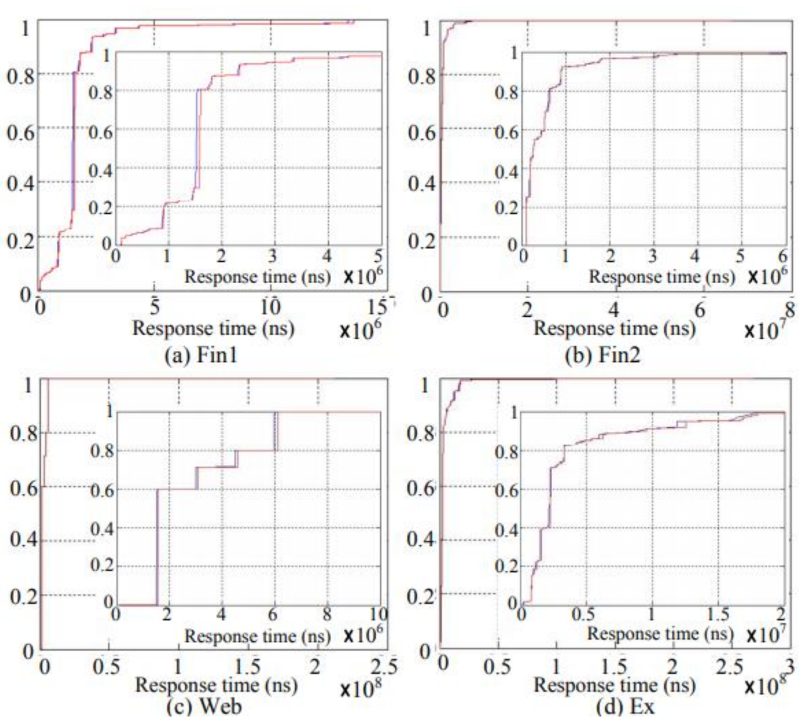


Gambar II.11 Deviasi rata-rata antara SSDSim dan purwarupa SSD (Hu, 2011)

SSDSim telah divalidasi dengan menggunakan purwarupa SSD sungguhan berbentuk fisik. Hasil evaluasi SSDSim pada Gambar II.11 menunjukan deviasi yang kecil, antara 2%-2,9%, menggunakan berbagai *workload*. Hal tersebut menunjukan hasil simulasi dengan SSDSim mendekati SSD sungguhan. Hasil evaluasi SSDSim juga dapat dilihat dengan grafik CDF dari *response time* pada Gambar II.12, belokan pada grafik CDF dapat dilihat lebih jelas pada subgrafik yang ada. Garis merah adalah CDF *response time* dari purwarupa SSD, sedangkan garis biru adalah CDF *response time* dari SSDSim. Gambar II.12 tersebut menunjukan kedua garis merah dan biru hampir beririsan sepenuhnya, hal itu menunjukan tingkat akurasi SSDSim yang tinggi dalam menyimulasikan *response time* SSD.

Karena akurasinya yang tinggi dan implementasinya yang sederhana, SSDSim telah banyak digunakan pada beberapa penelitian terkait SSD. Beberapa diantaranya

adalah ttFlash (Yan, 2017), PA-SSD (Zhang, 2018), dan masih banyak lagi. Oleh karena itu dalam penelitian ini, SSDSim yang telah dimodifikasi juga digunakan untuk keperluan pengukuran dan pengujian. SSDSim merupakan simulator SSD berbasis *tracefile*, hal tersebut berarti SSDSim memerlukan masukan berupa *tracefile* yang berisi urutan proses I/O yang diterima oleh sebuah SSD. Contoh *file* masukan yang diterima oleh SSDSim dapat dilihat pada Gambar II.13. Setiap baris dalam tracefile menunjukan I/O *request* yang dikirimkan ke SSD. Pada setiap baris terdapat empat bilangan, bilangan pertama adalah waktu datangnya *request* tersebut dalam satuan *nanosecond* (ns), bilangan kedua adalah nomor *id disk* tujuan *request* tersebut, bilangan ketiga adalah alamat memori atau *Logical Sector Number* (LSN) sebagai target dari *request* tersebut, lalu bilangan keempat adalah ukuran *request* tersebut dalam satuan *sector* (512 bytes), dan bilangan terakhir adalah jenis *request* tersebut, 1 menandakan *write request*, sedangkan 0 menandakan *read request*.



Gambar II.12 *Cumulative Distribution Function* (CDF) dari *response time* SSDSim dan purwarupa SSD dengan berbagai *workload* (Hu, 2011)

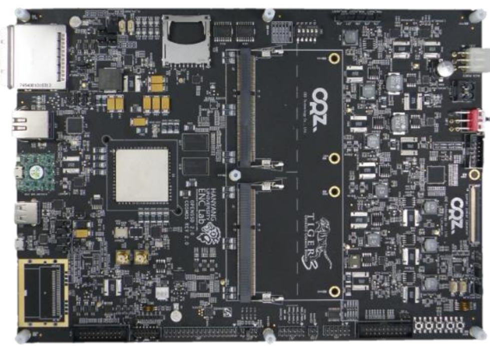
```
0 0 32020016 16 0
505 0 16814064 16 0
624 0 42387552 16 0
1106 0 25872944 16 0
1276 0 121286448 16 0
2015 0 72382240 16 1
2892 0 51178080 16 0
3237 0 65747328 16 1
3801 0 122653824 32 0
3941 0 39945520 16 1
6063 0 27051872 16 0
7875 0 40989360 32 0
....
```

Gambar II.13 Contoh *tracefile* yang digunakan SSDSim

II.7 Simulator dan Emulator SSD Lainnya

Untuk membantu perancangan desain arsitektur maupun algoritma dalam SSD, serta pengukuran efektivitas dari usulan baru tersebut, maka diperlukan kakas (*tool*) khusus yang dapat memodelkan SSD. Sebelumnya para peneliti sudah mengajukan beberapa kakas yang dapat digunakan, salah satunya adalah SSDSim yang telah dijelaskan pada subbab II.6. Secara umum kita dapat membagi kakas tersebut menjadi dua kelompok yaitu simulator berbasis perangkat lunak (*software-based simulator*) dan emulator berbasis perangkat keras (*hardware-based emulator*) (Yoo, 2013).

Hardware-based emulator adalah kakas yang paling akurat untuk mengamati proses dalam SSD secara aktual (*real-time*). Namun karena bagian internal dari emulator tersebut tidak dapat diubah-ubah dengan mudah, maka emulator tersebut tidak dapat digunakan untuk mengamati SSD dengan desain internal yang berbeda-beda. Kita tidak dapat mengamati SSD dengan jumlah *channel* yang beragam, jumlah *bus clock* yang beragam. Kita juga tidak dapat mengatur latensi proses I/O, jumlah *page* per *block*, ukuran *page*, dan lain sebagainya. Sehingga eksperimen yang dapat dilakukan dengan *hardware-based emulator* sangat terbatas. Salah satu kakas yang masuk kategori ini adalah OpenSSD (Lee, 2011), bentuknya dapat dilihat pada Gambar II.14.



Gambar II.14. Cosmos OpenSSD, salah satu *hardware-based emulator* (Sumber: www.openssd.io)

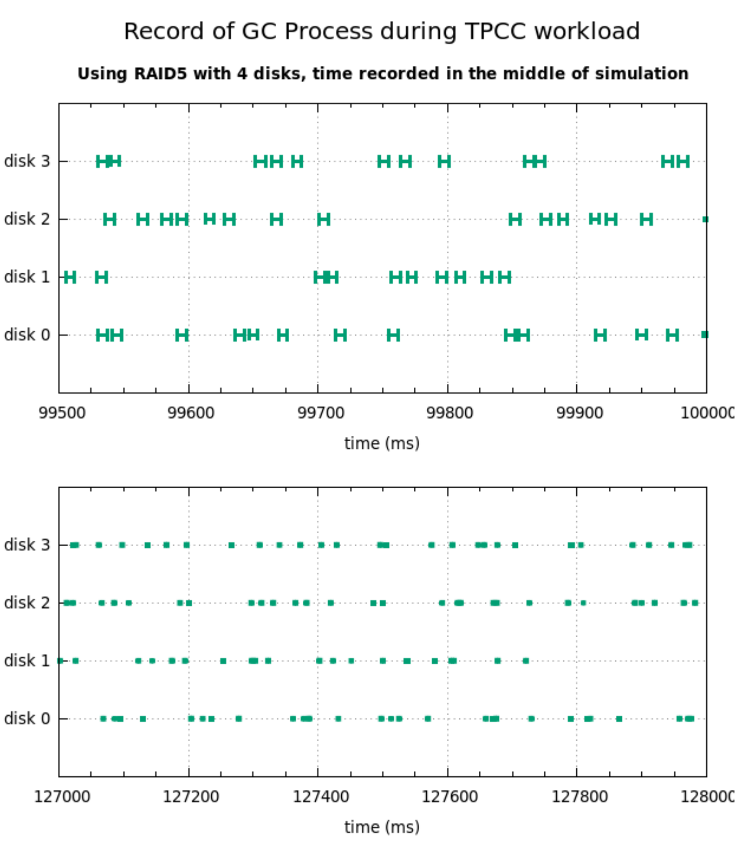
Sesuai dengan namanya, *software-based simulator* tidak menggunakan perangkat keras untuk memodelkan SSD. Simulator berbasis perangkat lunak ini dapat dibagi lagi menjadi dua jenis, yaitu *trace-driven simulator* yang berjalan secara *off-line* dan *virtual devices based simulator* yang berjalan secara *on-line* dengan terhubung pada *host* sebagai *software based block devices*. *Trace-driven simulator* dapat memodelkan komponen internal dari SSD secara detail, dan sering digunakan untuk berbagai eksperimen terkait desain SSD. Selain itu *trace-driven simulator* juga lebih sederhana untuk dikonfigurasi karena berbentuk program yang biasanya sudah menggunakan paradigma *object-oriented programming* (OOP). Untuk menjalankan simulasi pada *trace-driven simulator* dibutuhkan sebuah *tracefile* yang berisi urutan proses I/O yang diterima oleh SSD. Beberapa contoh dari simulator tipe ini adalah SSDSim (Hu, 2011) dan DiskSim (Agrawal, 2008).

Sedangkan *virtual device based simulator* merupakan ekstensi pada program virtualisasi yang sudah ada, biasanya simulator ini terhubung ke *host* sebagai *software based block device* atau ekstensi *filesystem*. Karena menggunakan program virtualisasi, kita dapat mengamati proses dalam SSD secara *on-line*, seperti menjalankan sistem operasi dan menjalankan berbagai program di atasnya. Beberapa contoh simulator berbasis *virtual device* ini adalah VSSIM (Yoo, 2013) dan FEMU (Li, 2018). Namun saat ini pengembangan *virtual device based simulator* masih belum banyak. Oleh karena itu, beberapa penelitian sebelumnya biasanya dilakukan dengan menggunakan *trace-driven simulator*.

BAB III
ANALISIS DAN DESAIN RANCANGAN SOLUSI

III.1 Analisis Pentingnya Minimasi GC *Collision*

Seperti yang dijelaskan pada subbab II.4.2 mengenai ttFlash, dampak dari GC pada SSD sebenarnya dapat dieliminasi dengan cara rekonstruksi menggunakan paritas. Sebelumnya penggunaan paritas sudah lebih umum dijumpai pada RAID, terutama RAID 5. Penggunaan paritas dalam RAID tersebut ditujukan untuk menghindari kesalahan penyimpanan data. Namun paritas tersebut juga dapat digunakan untuk melakukan rekonstruksi data ketika proses GC sedang berlangsung, sama halnya seperti yang dilakukan pada ttFlash.



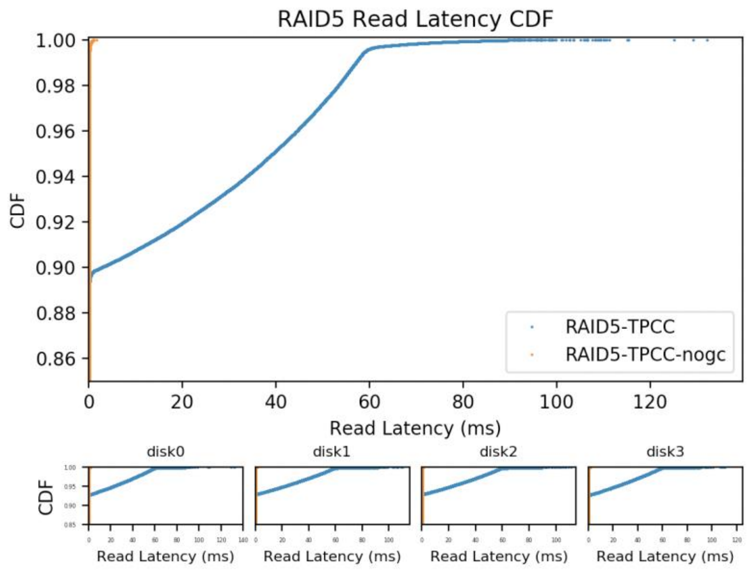
Gambar III.1 Potongan simulasi RAID5 dengan *workload* TPCC yang menunjukan proses GC di setiap *disk*

Namun metode rekonstruksi tersebut dapat dilakukan hanya ketika dalam satu waktu hanya ada satu proses GC yang sedang berlangsung. Dengan kata lain, metode rekonstruksi tersebut dapat dilakukan jika *GC collision* dapat dikurangi. Oleh karena itu diperlukan sebuah strategi penjadwalan GC pada RAID SSD untuk meminimasi *collisison*. Simulasi RAID 5 yang dilakukan dengan SSDSim secara sekilas menunjukan adanya *GC collision* pada semua *workload* yang digunakan. Hal tersebut dapat dilihat pada grafik berbentuk Gantt Chart pada Gambar III.1, garis hijau merepresentasikan proses GC yang sedang berjalan, absis pada grafik tersebut adalah waktu berjalannya simulasi.

Sebelumnya penelitian terkait penjadwalan GC pada RAID SSD sudah pernah dilakukan oleh Kim (2011) dengan mekanisme Harmonia. Penelitian Harmonia tersebut dilakukan sebelum dikenalnya metode rekonstruksi untuk meminimasi dampak dari GC. Alih-alih meminimasi *GC collision*, mekanisme Harmonia justru memaksimalkan *GC collision* dengan membuat proses GC pada setiap disk sebisa mungkin dilakukan serentak atau bersamaan. Hingga sekarang, masih belum ada yang mempublikasikan penelitian mengenai implementasi penjadwalan GC pada RAID SSD untuk meminimasi *collision*. Oleh karena itu, jika efek dari GC pada RAID SSD ingin dieliminasi dengan rekonstruksi data, maka penjadwalan GC untuk meminimasi *collision* harus dilakukan terlebih dahulu.

III.2 Analisis Dampak *GC Collision* terhadap *Response Time*

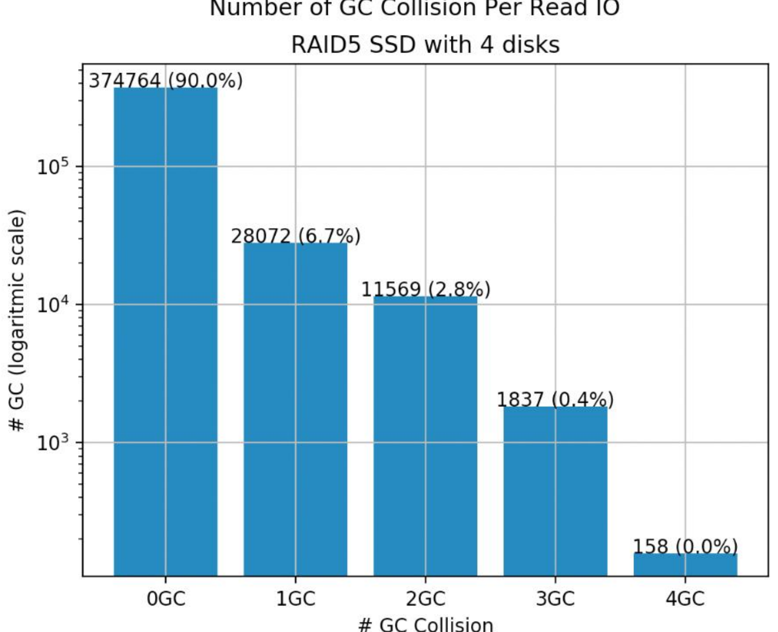
Untuk menunjukan pengaruh GC dan *GC Collision* pada RAID SSD terhadap *response time*, perlu dilakukan pengukuran yang mendetail. Modifikasi pada SSDim dilakukan untuk mengukur dampak dari GC dan *GC collision* terhadap *response time* pada RAID SSD. Penjelasan mengenai modifikasi SSDSim yang dilakukan dapat dilihat pada bab 4. Grafik *Cumulative Distribution Function* (CDF) dari *response time* digunakan untuk menunjukan persebaran *response time* secara keseluruhan dari awal simulasi hingga akhir.



Gambar III.2 Contoh grafik CDF untuk pengujian

Pengukuran CDF dari simulasi *workload* TPCC pada RAID 5 dapat dilihat pada Gambar III.2 dengan garis berwarna biru, sedangkan garis jingga menunjukkan latensi jika GC dihilangkan dalam simulasi. Terlihat bahwa grafik CDF berbelok pada sekitar persentil 90, hal tersebut menunjukkan sekitar 10% dari *read request* memiliki latensi yang lebih tinggi dari biasanya. GC dan *GC collision* diduga menjadi penyebab dari membesarnya latensi tersebut.

Dugaan tersebut dikonfirmasi dengan pengukuran jumlah *read request* yang terhalang oleh GC, hasil pengukuran tersebut dapat dilihat pada Gambar III.3. Sumbu x pada Gambar III.3 menunjukkan jumlah *read request* yang terhalang oleh n proses GC, 1GC menunjukkan jumlah *read request* yang terhalang oleh sebuah proses GC, sedangkan 2GC menunjukkan jumlah *read request* yang terhalang oleh dua proses GC dari *disk* yang berbeda, begitu seterusnya. Dari simulasi tersebut terlihat ada 10% *read request* yang terhalang oleh GC, hal tersebut sejalan dengan berbeloknya grafik CDF pada persentil 90.



Gambar III.3 Contoh grafik GC *collision* untuk pengujian

Metode ttFlash yang melakukan rekonstruksi untuk mengeliminasi dampak dari GC berhasil mengurangi pertambahan latensi karena proses GC. Hal tersebut dilakukan dengan meminimasi *GC collision* pada SSD. Oleh karena itu jika efek dari GC dan *GC collision* ingin dieliminasi pada RAID SSD maka penting untuk meminimasi *collision.*

III.3 Penjadwalan GC pada RAID SSD

Dalam sebuah konfigurasi RAID, sebuah SSD hanya mengetahui informasi lokal pada SSD itu sendiri. SSD tersebut tidak mengetahui informasi tentang SSD lainnya yang ada pada konfigurasi RAID yang sama. Oleh karena itu peran *controller* pada RAID menjadi sangat penting untuk mengoordinir semua SSD yang terhubung padanya. Sehingga SSD dan *controller* perlu saling bertukar informasi untuk melakukan penjadwalan GC.

Controller RAID yang menerapkan penjadwalan GC perlu diprogram khusus agar pernjadwalan tersebut dapat berjalan dengan baik, *controller* tersebut harus dapat bertukar informasi dengan SSD. Begitu pula dengan SSD yang ingin menerapkan

algoritma penjadwalan GC, SSD tersebut harus dapat berkomunikasi dengan *controller* RAID dan melaksanakan GC saat diizinkan oleh *controller*. Koordinasi antara SSD dan *controller* RAID dapat dilakukan sekali saja saat instalasi atau secara terus menerus selama RAID tersebut masih digunakan.

III.4 Desain Strategi Minimasi *Collision*

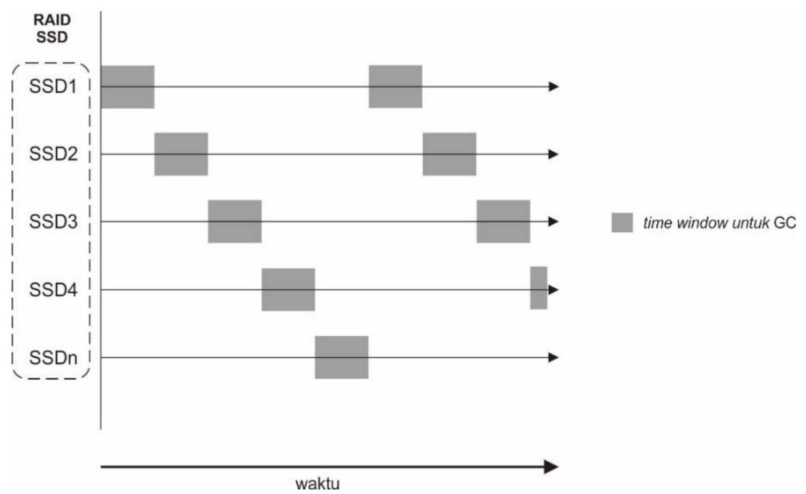
Dalam penelitian ini diusulkan tiga strategi yang dapat digunakan untuk meminimasi *GC collision* pada RAID SSD. Ketiga strategi tersebut berusaha menjadwalkan GC sehingga *collision* dapat dikurangi atau bahkan dihilangkan sama sekali. Ketiga strategi tersebut adalah GCSync, GCSync+, dan GCLock, berikut adalah penjelasan ketiga strategi tersebut.

III.4.1 GCSync

Salah satu strategi yang dapat dilakukan adalah dengan hanya mengizinkan SSD melakukan GC pada *time window* tertentu, metode ini selanjutnya akan disebut dengan nama GCSync. Setiap SSD memiliki *time window* yang berbeda-beda tergantung dengan nomor *id* dari SSD tersebut, ini untuk memastikan tidak ada GC yang dapat terjadi saat di SSD lain proses GC sedang berlangsung. Misalkan pada RAID dengan jumlah SSD sebanyak n dan durasi *time window* sebesar t_w, maka untuk SSD dengan nomor *id* d_i, *time window* untuk SSD tersebut adalah:

$$[t_w(ni+d_i)]ms \text{ s/d } [t_w(ni+d_i+1)]ms \text{ , } i \in Z$$

Misalkan pada konfigurasi RAID dengan 4 jumlah SSD, t_w = 100ms, dan id dari SSD yang terhubung dengan RAID *controller* adalah 0, 1, 2, dan 3. Maka SSD dengan id 2 hanya dapat menjalankan GC pada 200–300ms, 600-700ms, dst, sedangkan SSD dengan id 3 dapat menjalankan GC ada 300-400ms, 700–900ms, dst. Ilustrasi metode GCSync pada RAID SSD dapat dilihat pada Gambar III.4.



Gambar III.4 Ilustrasi *time window* pada GCSync

III.4.2 GCSync+

Kelemahan dari strategi GCSync adalah masih dimungkinkan terjadinya *GC collision* pada SSD yang berbeda. Strategi GCSync+ merupakan pengembangan dari strategi GCSync biasa. Pada GCSync biasa, *collision* masih dimungkinkan terjadi jika ada GC yang dimulai pada akhir *time window* suatu SSD, dan GC tersebut belum selesai walaupun *time window* dari SSD tersebut sudah selesai. Kemudian pada SSD lainnya, *time window* untuk proses GC sudah diaktifkan, hal tersebut dapat mengakibatkan dua proses GC berjalan bersamaan pada SSD yang berbeda.

Strategi GCSync+ berusaha menghilangkan GC *collision* tersebut dengan menambahkan *buffer time* di antara dua *time window*. Hal ini memberikan kesempatan proses GC berjalan tanpa dibarengi oleh proses GC pada SSD lainnya, karena SSD lainnya tersebut harus menunggu selama *buffer time* sebelum mengaktifkan *time window*. Oleh karena itu, jika konfigurasi RAID dengan n buah SSD, d_i sebagai nomor *id* dari setiap *disk*, durasi *time window* sebesar t_w dan durasi *buffer time* sebesar t_b, maka *time window* pada setiap SSD menjadi:

$$[(t_w+t_b)(ni+d_i)]ms \text{ s/d } [(t_w+t_b)(ni+d_i+1)-t_b]ms \text{ , } i \in Z$$

III.4.3 GCLock

Strategi lainnya yang dapat digunakan adalah dengan menggunakan *lock* pada *controller* RAID, metode ini selanjutnya akan disebut dengan nama GCLock. Setiap SSD yang akan melakukan proses GC perlu mengecek apakah *lock* tersedia di *controller*, jika *lock* tersebut tersedia dan dapat digunakan maka proses GC dapat dijalankan oleh SSD tersebut. Lalu setelah SSD tersebut selesai melakukan proses GC maka *lock* tersebut perlu dikembalikan ke *controller* RAID sehingga dapat digunakan oleh SSD lainnya.

Jika dibandingkan dengan GCSync, metode ini lebih kompleks karena diperlukan komunikasi antara SSD dan controller setiap proses GC akan dilaksanakan. Hal tersebut dapat membuat proses GC menjadi lebih lama. Pada GCSync dan GCSync+, koordinasi antara *controller* dan SSD hanya dilakukan saat inisiasi saja, namun pada strategi GCLock ini komunikasi antara SSD dengan *controller* perlu dilakukan setiap saat.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

IV.1 Implementasi dengan SSDSim

Setelah melakukan eksplorasi awal terhadap beberapa simulator SSD, dalam tugas akhir ini akan digunakan SSDSim untuk menyimulasikan pemrosesan I/O dan GC dalam SSD. SSDSim merupakan simulator SSD berbasis *tracefile*, hal tersebut berarti SSDSim memerlukan masukan berupa *tracefile* yang berisi urutan proses I/O yang diterima oleh sebuah SSD.

Gambar IV.1 Skema penggunaan SSDSim untuk simulasi SSD

SSDSim dipilih karena cara penggunaannya yang sederhana dibandingkan jenis simulator lainnya, selain itu SSDSim juga dapat dimodifikasi untuk menyimulasikan mekanisme tertentu dalam SSD. Pembuat SSDSim, Yang Hu, sudah memastikan bahwa simulatornya dapat berjalan mendekati SSD pada dunia nyata. Beberapa penelitian yang dilakukan sebelumnya juga menggunakan SSDSim untuk pengujiannya. SSDSim diimplementasikan dengan bahasa C dan dapat menghasilkan statistik terkait dengan latensi I/O. Skema penggunaan SSDSim dapat dilihat pada Gambar IV.1, sedangkan contoh statistik yang mencatat latensi tiap proses I/O dapat dilihat pada Gambar IV.2. Setiap baris pada Gambar IV2 adalah catatan untuk setiap *request* yang diproses selama simulasi.

```
     300000   87880544     16  1          300000              0     142575
  441000000   39949408     16  1       441000000              0     142575
 1111300000   54489088     16  1      1111300000              0     142575
 1625300000   72261008     16  1      1625300000              0     142575
 3657100000   70004480     16  1      3657100000              0     142575
 6372200000   55184816     32  1      6372200000              0     142575
 6373022575   55184816     32  0      6373022575     6373023575       1000
 6407800000   90290208     32  1      6407800000              0     142575
 6408622575   90290208     32  0      6408622575     6408623575       1000
 6417700000  102397728     32  1      6417700000              0     142575
 6418522575  102397728     32  0      6418522575     6418523575       1000
 7892600000   57201648     16  1      7892600000              0     142575
 8203000000   56142288     16  1      8203000000              0     142575
 8554600000   53224288     16  1      8554600000              0     142575
 9323500000   58212192     16  1      9323500000              0     142575
 9324322575   58212192     16  0      9324322575     9324323575       1000
 9407000000   34897632     16  1      9407000000              0     142575
 9644100000   54829776     16  1      9644100000              0     142575
10667500000   44825168     16  1     10667500000              0     142575
10845300000   92258384     16  1     10845300000              0     142575
10941800000   68706112     16  1     10941800000              0     142575
11492600000   66971392     16  1     11492600000              0     142575
12678000000   53224336     16  1     12678000000              0     142575
13084400000   54489104     16  1     13084400000              0     142575
....
```

Gambar IV.2 Contoh *log* latensi I/O yang dihasilkan pada akhir simulasi

Karena penelitian pada tugas akhir ini terkait dengan GC dan RAID SSD maka ada beberapa modifikasi yang perlu ditambahkan pada SSDSim. Modifikasi tersebut adalah dengan membuat SSDSim menghasilkan data terkait GC, menandai I/O yang terhalang oleh proses GC, implementasi *controller* RAID pada SSDSim, dan yang terakhir adalah implementasi metode minimasi *GC collision*.

IV.1.1 Menghasilkan Data Log GC

Sebelumnya SSDSim hanya mengahasilkan data jumlah *request*, latensi per *request*, dan data lainnya terkait dengan *request* I/O. Walaupun SSDSim sudah menyimulasikan proses GC pada SSD, namun data dari proses GC tersebut belum diberikan pada akhir simulasi. Oleh karena itu pertama-tama perlu dilakukan modifikasi pada SSDSim untuk menghasilkan data terkait dengan GC. Dalam SSDSim, informasi terkait sebuah proses GC disimpan dalam *struct* `gc_operation`, strukturnya dapat dilihat pada Gambar IV3.

```c
struct gc_operation {
  unsigned int chip;
  unsigned int die;
  unsigned int plane;
  unsigned int block;
  unsigned int page;
  unsigned int state;
  unsigned int priority;
  struct gc_operation * next_node;

  int64_t x_start_time;
  int64_t x_end_time;
  double x_free_percentage;
  unsigned int x_moved_pages;
};
```

Gambar IV.3 *Struct* gc_operation untuk menampung informasi sebuah proses GC dalam SSDSim

Atribut `chip`, `die`, `plane`, `block`, dan `page` digunakan untuk mengacu lokasi terjadinya proses GC. SSDSim melakukan proses GC secara *channel-blocking*, hal tersebut berarti ketika GC dilakukan pada sebuah *channel* maka *request* I/O dengan alamat pada *channel* tersebut harus menunggu proses GC selesai terlebih dahulu, namun proses I/O pada *channel* lain masih dapat dilakukan. SSDSim menyimpan semua proses GC dalam struktur data *linked-list*, atribut `next_node` digunakan untuk mengacu ke proses GC selanjutnya pada *linked-list* dalam SSDSim.

Untuk keperluan pencatatan waktu berjalnnya proses GC selama simulasi, maka ditambahkannlah atribut `x_start_time`, dan `x_end_time`, prefix `x` menunjukan atribut tersebut adalah atribut tambahan untuk pengerjaan penelitian ini. Sesuai dengan *timestamp* lainnya pada SSDSim, waktu mulai dan selesainya proses GC dicatat dalam satuan *nanosecond* (ns). Selain itu juga terdapat atribut `x_free_percentage` untuk mencatat seberapa banyak free page yang tersedia saat proses GC tersebut diinisialisasi, sedangkan atribut `x_moved_pages` digunakan untuk mencatat berapa banyak *page* yang dipindahkan sebelum sebuah *block* dihapus dalam suatu proses GC.

```
6      5      0      0      23.68      59      3067321875      3128985725      61663850
7      5      0      0      23.68      59      3067321875      3128985725      61663850
4      0      0      0      23.54      58      3070269500      3130888200      60618700
5      0      0      0      23.54      58      3070269500      3130888200      60618700
6      4      0      0      23.57      59      3130401175      3192065025      61663850
7      4      0      0      23.57      59      3130401175      3192065025      61663850
2      0      0      0      23.55      59      3072346575      3134010425      61663850
3      0      0      0      23.55      59      3072346575      3134010425      61663850
4      7      0      0      23.66      59      3132990775      3194654625      61663850
5      7      0      0      23.66      59      3132990775      3194654625      61663850
0      7      0      0      23.65      59      3073702975      3135366825      61663850
1      7      0      0      23.65      59      3073702975      3135366825      61663850
0      0      0      0      23.55      59      3136269400      3197933250      61663850
1      0      0      0      23.55      59      3136269400      3197933250      61663850
6      0      0      0      23.57      59      3194207600      3255871450      61663850
7      0      0      0      23.57      59      3194207600      3255871450      61663850
....
```

Gambar IV.4 Contoh *log* proses GC yang dihasilkan setelah simulasi

Setiap kali proses GC selesai dilakukan dan *struct* `gc_operation` akan dihapus, sebelumnya data terkait dengan proses GC tersebut dituliskan dalam *file* eksternal dengan nama `gc.dat`. Contoh hasil pencatatan GC ini dapat dilihat pada Gambar IV.4. Data *log* terkait GC ini dapat digunakan untuk analisis lebih lanjut dan divisualisasikan menggunakan grafik tertentu. Setiap baris pada *log* merupakan sebuah proses GC yang terjadi dalam simulasi. Data tiap kolom berturut turut adalah `chip`, `die`, `plane`, `block`, `channel`, `x_free_percentage`, `x_moved_pages`, `x_start_time`, `x_end_time`, dan durasi proses GC tersebut.

IV.1.2 Menandai I/O yang Terhalang Proses GC

Sebelumnya SSDSim hanya menghasilkan data waktu mulai, waktu selesai, dan latensi untuk setiap I/O, baik proses *read* ataupun *write*. Karena penelitian yang dilakukan terkait dengan pengukuran GC *collision*, pengukuran hasil akhir akan lebih mudah dilakukan jika I/O yang terhalang oleh proses GC dapat ditandai. Oleh karena itu perlu dilakukan modifikasi terhadap SSDSim dengan menambahkan *flag* pada object I/O request. *Flag* yang aktif menandakan bahwa sebuah I/O terhalang oleh GC saat I/O tersebut berusaha mengakses data yang ditujunya. Sehingga untuk

setiap I/O yang terhalang oleh proses GC, kita dapat menandainya pada log yang dihasilkan.

Pada SSD sebuah *request* I/O akan dipecah menjadi beberapa *sub-request* yang lebih kecil untuk membaca atau menulis pada lokasi yang berbeda-beda. Hal tersebut juga disimulasikan dalam SSDSim, khususnya pada prosedur `ditribute()` dan `no_buffer_distribute()`. Jika salah satu *sub-request* dari sebuah *request* terhalang oleh proses GC, maka *request* tersebut akan langsung ditandai. Semua *request* yang disimulasikan pada SSDSim dicatat dan disimpan pada akhir simulasi dalam file `io.dat`, contoh *file log* yang dhasilkan dapat dilihat pada Gambar IV.5, formatnya masih sama dengan *file log* pada Gambar IV.2, hanya saja ada tambahan dua kolom terakhir yaitu *flag* `meet_gc` dan `gc_remaining_time`. Atribut *flag* `meet_gc` bernilai 1 jika *request* tersebut terhalang oleh GC, sedangkan 0 sebaliknya. Lalu atribut `gc_remaining_time` menunjukan waktu yang harus ditunggu oleh *request* I/O tersebut sebelum proses GC selesai dan *request* tersebut dapat dieksekusi.

```
7141335400000   57979568    32  1   7141335400000          0    142575  0         0
7141392600000   57014064    32  1   7141392600000          0      1000  1  47575425
7141437900000   54307456    16  1   7141437900000          0      1000  1   2275425
7141381100000   56896032    32  1   7141381100000          0  59280575  1  59075425
7141449300000   53080096    32  1   7141449300000          0    142575  0         0
7141471900000   56975920    32  1   7141471900000          0    142575  0         0
7141517100000  102408208    16  1   7141517100000          0    142575  0         0
7141528400000   54756992    16  1   7141528400000          0    142575  0         0
7141551400000   53255600    32  1   7141551400000          0    142575  0         0
7141587900000   57164576    32  1   7141587900000          0    142575  0         0
7141599200000   56605344    32  1   7141599200000          0    142575  0         0
7141655400000   54835120    32  1   7141655400000          0      1000  1  44032325
7141666600000   54770800    16  1   7141666600000          0      1000  1  32832325
7141644300000   54833488    16  1   7141644300000          0  55274900  1  55132325
7141678100000  101110192    32  1   7141678100000          0  23474900  1  21332325
7141734500000   58394496    16  1   7141734500000          0    142575  0         0
7141745600000   55419904    16  1   7141745600000          0    205150  0         0
7141780200000  102122336    16  1   7141780200000          0    142575  0         0
7141803000000   58348992    16  1   7141803000000          0    142575  0         0

........
```

Gambar IV.5 Contoh *log request* I/O yang disimulasikan dengan *flag* meet_gc yang menandakan apakah *request* tersebut terhalang oleh GC atau tidak

IV.2 Implementasi Strategi Minimasi Collision

Simulator yang digunakan dalam penelitian ini, SSDSim, hanya menyimulasikan sebuah SSD saja. Padahal objek dalam penelitian ini adalah RAID SSD yang melibatkan lebih dari satu SSD. Oleh karena itu terlebih dahulu perlu dilakukan implementasi *controller* RAID yang mengintegrasikan beberapa SSD. Setelah itu strategi GCSync, Gcsync+, dan GCLock dapat diimplementasikan pada *controller* RAID tersebut. Berikut adalah penjelasan kedua tahap tersebut.

IV.2.1 Implementasi *Controller* RAID

Controller RAID bertugas untuk menerima *request* dan menyebarkannya pada SSD yang terhubung pada *controller* tersebut. Implementasi *controller* RAID ini dapat berbeda-beda tergantung pada jenis RAID yang digunakan, dalam penelitian ini *controller* yang diimplementasikan adalah *controller* RAID 5. Implementasi *controller* RAID diimplementasikan dalam SSDSim dalam file `raid.h` dan `raid.c`, diperlukan sekitar 1000 lebih baris kode untuk membuat c*ontroller* tersebut.

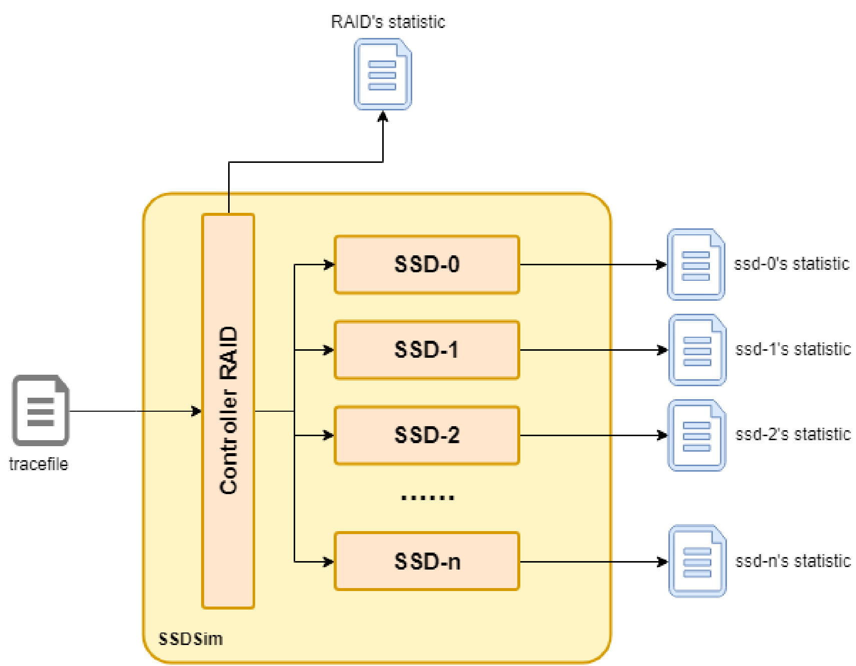
```
struct raid_info {
    unsigned int raid_type;
    unsigned int num_disk;
    unsigned int block_size;
    unsigned int stripe_size;
    unsigned int stripe_size_block;
    unsigned int strip_size_block;

    char tracefilename[80];
    char logfilename[80];
    FILE * tracefile;
    FILE * logfile;

    int64_t current_time;
    unsigned int max_lsn;
    struct raid_request *request_queue;
    struct raid_request *request_tail;
    unsigned int request_queue_length;
    struct ssd_info **connected_ssd;
};
```

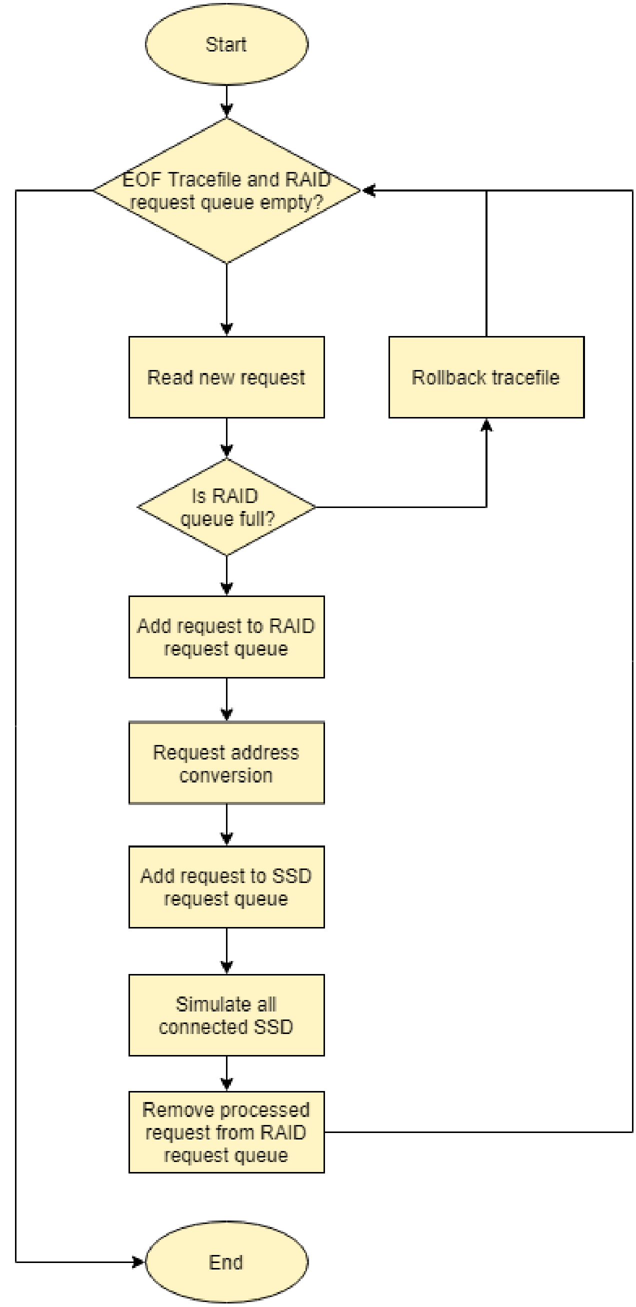
Gambar IV.6 *Struct* raid_info untuk memodelkan *controller* RAID dalam SSDSim

Objek *controller* dimodelkan dengan *struct* `raid_info`, strukturnya dapat dilihat pada Gambar IV.6. Beberapa atribut dalam *struct* tersebut diantaranya adalah `raid_type` yang menandakan jenis *controller* RAID dan `num_disk` yang menunjukan jumlah SSD yang terhubung *controller* tersebut. SSD dimodelkan dengan *struct* `ssd_info`, dan semua SSD yang terhubung dengan *controller* dapat diakses melalui atribut `connected_ssd`.



Gambar IV.7 Skema simulasi RAID dengan SSDSim

Penambahan *controller* dalam SSDSim memungkinkan dilakukannya simulasi RAID sesuai dengan Gambar IV.7. Simulasi tetap memerlukan sebuah *tracefile*, kemudian *controller* akan memecah setiap *request* yang ada dalam *tracefile* menjadi beberapa *request* ke SSD yang terhubung dengan *controller* tersebut. Dalam penelitian ini, *controller* yang diimplementasikan adalah untuk RAID 5 sehingga terdapat proses penghitungan paritas di dalamnya.



Gambar IV.8 Diagram alir proses kerja *controller* dalam SSDSim

Sebuah *read request* akan dipecah ke beberapa read request untuk SSD yang berbeda-beda. Sedangkan sebuah *write request* tidak dapat langsung dipecah ke beberapa SSD, tetapi perlu ada proses penghitungan didalamnya. Sebuah *write request* pada RAID 5 akan diubah menjadi *read request* untuk membaca data lama

dan paritas lama, dan *write request* untuk menulis data baru dan paritas baru. *Write request* yang menulis paritas baru harus menunggu *read request* sebelumnya selesai terlebih dahulu. Hal tersebut karena penghitungan paritas memerlukan data lama dan paritas lama.

Saat simulasi dijalankan, *controller* akan membaca *request* dari *tracefile* baris per baris. Sebuah *request* yang telah dibaca kemudian dipecah menjadi beberapa *request* ke beberapa SSD yang berbeda. *Request* dari RAID ke SSD ditampung dalam *struct* `raid_sub_request` seperti pada Gambar IV.9. Saat pemecahan *request* tersebut diperlukan konversi *Logical Sector Address* (LSA) dari tingkat RAID ke tingkat SSD. Proses kerja *controller* ini dapat dilihat lebih detail melalui diagram alir yang terdapat pada Gambar IV.8.

```
struct raid_sub_request {
    unsigned int disk_id;
    unsigned int stripe_id;
    unsigned int strip_id;
    unsigned int strip_offset;

    int64_t begin_time;
    int64_t complete_time;
    unsigned int current_state;

    unsigned int lsn;
    unsigned int size;
    unsigned int operation;

    struct raid_sub_request *next_node;
};
```

Gambar IV.9 *Struct* raid_sub_request untuk memodelkan *request* di SSD

Setiap *request* yang berasal dari *controller* dan ada dalam *queue* di SSD memiliki beberapa *state*, diantaranya adalah `R_SR_WAIT_PARITY`, `R_SR_PENDING`, `R_SR_PROCESS`, dan `R_SR_COMPLETE`. Sebuah *request* yang sedang diproses dalam SSD memiliki *state* `R_SR_PROCESS`, sedangkan *request* yang sudah selesai diproses dalam SSD memiliki *state* `R_SR_COMPLETE`. Jika semua *request* di SSD

yang berasal dari satu *request* di RAID sudah selesai diproses semuanya, maka *request* tersebut akan dihapus dari *queue* yang ada di *controller*. Penghapusan *request* tersebut dilakukan agar *controller* dapat memroses *request* lainnya yang datang dari *tracefile*.

IV.2.2 Implementasi GCSync

Objek SSD pada SSDSim dimodelkan dengan `struct ssd_info` yang ada pada file `initialize.h`. Implementasi GCSync dilakukan dengan menambahkan atribut `gc_time_window` pada `struct ssd_info`. Atribut tersebut berisi durasi *time window* yang dimiliki oleh suatu SSD, proses GC pada SSD tersebut hanya dapat dilakukan saat *time window* yang bersesuaian dengan *id* SSD tersebut. Misalkan sebuah konfigurasi RAID dengan 4 SSD dan mengaktifkan mode GCSync memiliki durasi *time window* selama 100 ms, maka SSD dengan id 0 hanya dapat melakukan GC pada waktu 0-100 ms, 400-500 ms, dan seterusnya. Penghitungan time window tersebut dijelaskan lebih detail pada subbab III.4.1.

```
struct ssd_info{

    ...

    int is_gcsync;
    int ndisk;
    int diskid;
    int64_t gc_time_window;

    ...

};
```

Gambar IV.10 Modifikasi struct ssd_info untuk GCSync

Atribut lainnya yang ditambahkan dalam `struct ssd_info` adalah `is_gcsync` yang bernilai `TRUE` jika mode GCSync diaktifkan. Selain itu juga terdapat atribut `ndisk` untuk menyimpan jumlah SSD yang terhubung dengan *controller* RAID. Terdapat juga atribut `diskid` untuk menyimpan *id* SSD tersebut, penomoran *id*

tersebut dimulai dari 0 hingga n-1, dengan n adalah jumlah SSD yang terhubung pada *controller* RAID.

Semua atribut yang terkait dengan GCSync tersebut diinisialisasi saat awal simulasi dimulai, atau saat awal SSD diinisialisasi. Oleh karena itu tidak diperlukan lagi komunikasi antara SSD dengan *controller* setelah inisialisasi tersebut dilakukan. Hal tersebut dapat mengurangi kompleksitas implementasi. Hasil pengujian lebih lanjut mengenai GCSync dapat dilihat pada Subbab IV.5.2.

IV.2.3 Implementasi GCSync+

Implementasi GCSync+ dilakukan dengan melakukan sedikit modifikasi terhadap strategi GCSync biasa. Terdapat tambahan konstanta `GC_BUFFER_TIME` yang merepresentasikan waktu tunggu antar dua *time window* pada SSD yang bebeda. Dengan adanya tambahan konstanta tersebut, sebelum SSD memulai proses GC perlu dilakukan pengecekan apakah SSD tersebut sudah memasuki *time window* untuk GC atau belum. Sebenarnya GCSync adalah bentuk khusus dari GCSync+ dengan *buffer time* sama dengan nol.

IV.2.4 Implementasi GCLock

Mode GCLock diimplementasikan dengan menambahkan model *lock* pada *controller* RAID. *Lock* pada *controller* dimodelkan dengan `struct gclock_raid_info` seperti yang dapat dilihat pada Gambar IV.11. Selanjutnya atribut *lock* tersebut ditambahkan pada *controller* dalam `struct raid_info`.

```
struct gclock_raid_info {
    int64_t begin_time;
    int64_t end_time;

    int is_available;
    int holder_id;
};
```

Gambar IV.11 Struct gclock_raid_info untuk memodelkan *lock* pada *controller*

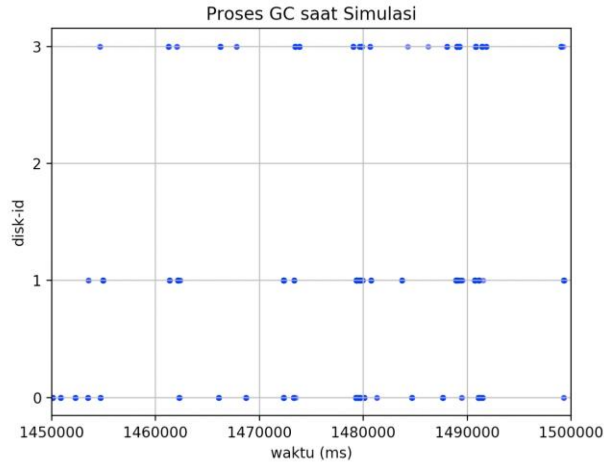
Atribut `is_available` bernilai `TRUE` jika tidak ada SSD yang sedang memegang *lock* tersebut, sebaliknya `FALSE` menandakan *lock* tersebut dapat digunakan. Jika atribut `is_available` bernilai `FALSE`, atribut `holder_id` akan berisi id SSD yang sedang menggunakan *lock* tersebut. Sedangkan atribut `begin_time` berisi waktu saat *lock* tersebut digunakan oleh SSD dengan id tertentu, atribut `end_time` berisi waktu perkiraan *lock* tersebut dikembalikan ke *controller*. Atribut `begin_time` dan `end_time` dapat digunakan untuk pengemabangan strategi penjadwalan lainnya yang belum diuji dalam penelitian ini.

IV.3 Pembuatan Kakas untuk Pengujian

Untuk mengukur efektifitas dari metode minimasi yang diimplementasikan, diperlukan tambahan kakas untuk memroses *file log* yang dihasilkan dari simulasi RAID dengan SSDSim. Kakas yang digunakan dalam penelitian ini dibuat dengan bahasa Python 3, dengan tambahan kakas Matplotlib untuk menghasilkan grafik. Adanya kakas yang menghasilkan grafik ini membuat pembandingan antara sebelum dan sesudah implementasi menjadi lebih mudah. Berikut adalah beberapa kakas yang dibuat untuk pengujian.

IV.3.1 Grafik Proses GC Selama Simulasi

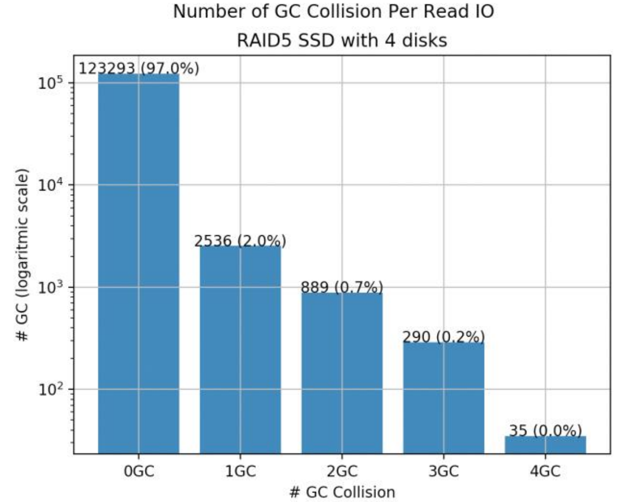
Kakas pertama digunakan untuk menghasilkan grafik proses GC pada *disk* yang berbeda selama simulasi dilakukan. Bentuk dari grafik ini mirip seperti Gantt Chart, namun proses GC ditandai dengan sebuah titik saja. Titik tersebut merupakan titik tengah antara waktu mulai dan waktu selesainya proses GC. Grafik ini hanya menampilkan penggalan waktu pada tengah-tengah simulasi, saat banyak proses GC yang terjadi, sehingga tidak semua proses GC dari awal hingga akhir simulasi dapat ditampilkan. Sumbu y pada grafik tersebut menunjukan nomor *id* dari *disk* yang ada dalam simulasi. Contoh grafik yang dihasilkan dari kakas ini dapat dilihat pada Gambar IV.12.



Gambar IV.12 Contoh grafik GC untuk memvisualisasikan GC pada RAID

IV.3.2 Grafik *GC Collison*

Jika kakas pertama hanya berfungsi untuk memvisualisasikan proses GC saja, kakas kedua ini berfungsi untuk mengukur *collision* secara kuantitatif. Kakas ini menghasilkan grafik yang menunjukkan jumlah GC pada *disk* yang berbeda yang menghalangi proses pembacaan data pada RAID SSD. Sumbu x pada grafik menunjukkan jumlah GC yang menghalangi proses pembacaan data. Sebagai contoh, 2GC menunjukkan jumlah *read request* yang terhalang oleh 2 proses GC sekaligus, sedangkan 3GC menunjukkan jumlah *read request* yang terhalang oleh 3 proses GC sekaligus pada *disk* yang berbeda, begitu juga dengan 4GC, dan seterusnya. Contoh grafik *GC Coliision* ini dapat dilihat pada Gambar IV.13.

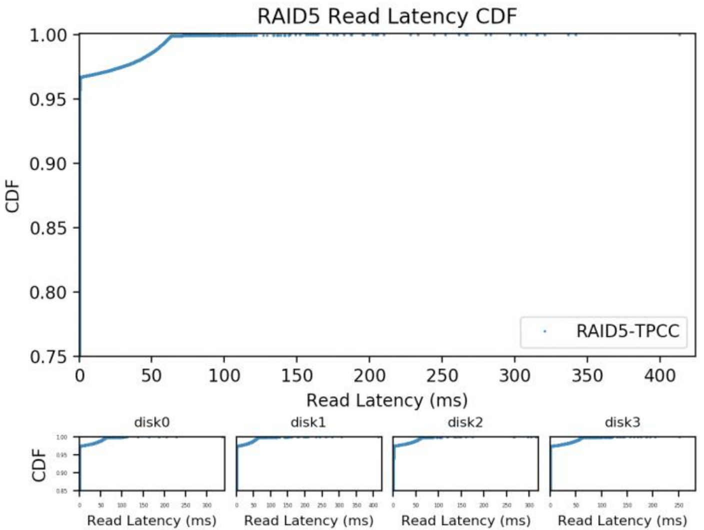


Gambar IV.13 Contoh grafik GC *Collision* untuk menunjukan collision pada RAID

Read request yang terhalang hanya oleh satu proses GC (1GC), dapat diselematkan dengan adanya rekonstruksi menggunakan paritas. Sedangkan 2GC, 3GC, dst tidak memungkinkan untuk dilakukan rekonstruksi dengan paritas. Oleh karena itu semakin sedikit jumlah dari 2GC, 3GC, dst maka minimasi *collision* semakin baik.

IV.3.3 Grafik CDF Latensi Proses Pembacaan Data

Selain memvisualisasikan dan mengukur *collision* yang terjadi pada RAID SSD, diperlukan juga sebuah kakas untuk menghasilkan grafik *Cumulative Distribution Function* (CDF) dari latensi *read request*. Grafik ini menggambarkan variabilitas latensi yang ada selama proses simulasi. Biasanya kurva akan berbelok pada persentil yang mirip dengan persentasi proses GC. Selain menghasilkan grafik, kakas ini juga mengukur rata-rata latensi proses I/O. Hal tersebut digunakan untuk mengukur bagaimana dampak dari implementasi strategi minimasi *collision* terhadap latensi dari RAID SSD. Contoh dari grafik ini dapat dilihat pada Gambar IV.14.



Gambar IV.14 Contoh Grafik CDF latensi *request* I/O

Pada grafik CDF yang dihasilkan oleh kakas ini, beberapa sub-grafik yang ada pada bagian bawah grafik adalah CDF latensi dari masing-masing disk yang ada pada konfigurasi RAID. Sedangkan grafik utamanya menunjukan CDF latensi dilihat dari *controller* RAID. Belokan pada kurva biru menandakan variabilitas yang ekstrem, hal tersebut dapat disebabkan oleh proses GC yang menghalangi *request* I/O.

IV.4 Deskripsi *Workload* untuk Pengujian

Untuk menguji strategi minimasi *collision* yang sudah diimplementasikan, pada penelitian ini digunakan tiga buah *workload* yang sudah umum digunakan dalam penelitian terkait dengan topik media penyimpanan. Ketiga *workload* tersebut berbentuk *block-level tracefile* yang diperoleh dari Microsoft Windows Server. Karakteristik ketiga workload tersebut sudah pernah dideskripsikan oleh Kavalenekar (2008) dalam makalahnya. Secara singkat karakteristik dari ketiga workload yang digunakan untuk evaluasi dapat dilihat pada Tabel IV.1.

Tabel IV.1 Karekteristik *workload* yang digunakan untuk pengujian

Workload	**Durasi (jam)**	**IO Rate**	**Avg. Req size Read/Write (KB)**	**Read (%)**	**Int. Arrv-Time (ms)**
TPCC	0:50		8,57/ 7,99	36,16	7,20
DTRS	8:20		53,38 / 55,57	10,96	236,14
MSNFS	8:20		9,02 / 15,75	11,61	55,66

IV.4.1 TPCC untuk Pengujian Basisdata

TPC-C merupakan pengujian *Online Transaction Processing* (OLTP) yang dikeluarkan oleh The Transaction Processing Performance Council (TPC) pada tahun 1992. *Workload* yang sudah umum digunakan untuk pengujian sistem basisdata ini merupakan bentuk yang lebih kompleks dibandingkan dengan pengujian OLTP sebelumnya, yaitu TPC-A. Selain digunakan untuk pengujian pada sistem basisdata, *workload* ini juga umum digunakan untuk pengujian pada penalitian terkait dengan media penyimpanan. TPC-C terdiri dari campuran lima transaksi konkuren dengan jenis yang berbeda-beda. Transaksi tersebut berusaha mengakses basisdata yang terdiri dari sembilan tabel dengan ukuran yang beragam.

IV.4.2 *Developer Tools Release Server* (DTRS)

DTRS merupakan *tracefile* dari sebuah *file server* yang menyimpan hasil kompilasi dari aplikasi Microsoft Visual Studio. Sebelumnya aplikasi Miscrosoft Visual Studio tersebut dikompilasi di *server* yang berbeda untuk kemudian diduplikasi ke DTRS. Terdapat sekitar 3000 pengguna yang mengakses DTRS. Memori yang

digunakan dalam DTRS adalah sebesar 2 GB, dan media penyimpanannya menggunakan *disk* virtual sebesar 40 GB yang dikonfigurasi dalam RAID 10.

IV.4.3 MSN *Storage Metadata and File Server* (MSNFS)

MSNFS menyimpan informasi *metadata* yang mengaitkan antara pengguna dengan *file* yang tersimpan dalam Back-End file Server (BEFS). Server tersebut digunakan dalam platform Live yang dimiliki oleh Microsoft. *Tracefile* MSNFS tersebut merekam *request* I/O selama sekitar 8,5 jam.

Tabel IV.2 Parameter penting yang digunakan dalam simulasi

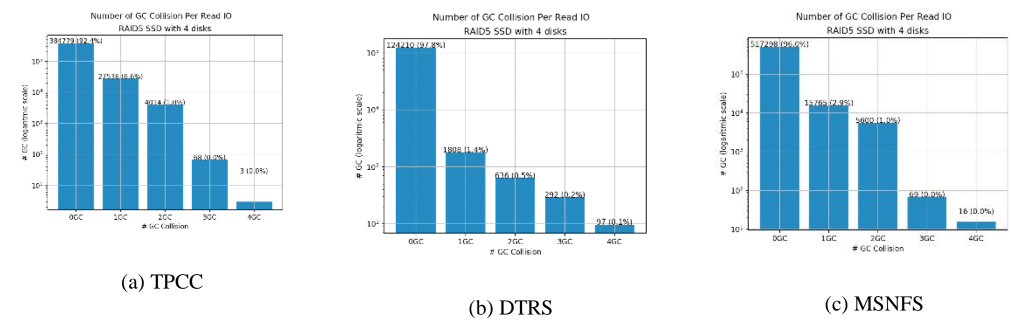
Parameter	Nilai
Page read to register	40000 ns
Page write from register	800000 ns
Block erase	2 ms
Aged ratio	0,76
GC threshold	0,30
RAID queue depth	20
SSD queue depth	8
SSD capacity	275 GB
#Disk	4
#Channel/Disk	8
#Chip/Channel	64
#Die/Chip	1
#Plane/Die	1
#Block/Plane	4096
#Page/Block	256
Page size	4096 bytes

IV.5 Hasil Pengujian Minimasi *Collision*

Untuk menjalankan simulasi pada SSDSim, terdapat file konfigurasi yang menyatakan spesifikasi dari SSD yang ingin disimulasikan. Beberapa parameter penting dalam file konfigurasi yang digunakan untuk menjalankan pengujian ini dapat dilihat pada Tabel IV.2. Karena pengujian dilakukan dengan menyimulasikan RAID SSD, maka semua SSD yang disimulasikan memiliki konfigurasi yang sama.

IV.5.1 Pengukuran Sebelum Implementasi

Sebelum implementasi dilakukan, sebelumnya perlu dilakukan pengukuran untuk nantinya dibandingkan dengan hasil setelah implementasi strategi minimasi *colliision*. Pengukuran ini dilakukan dengan tiga *workload* yang sudah dijelaskan sebelumnya. Visualisasi penggalan proses GC saat simulasi dapat dilihat pada Gambar IV.16, sedangkan nilai dari *GC collision* dapat dilihat pada Gambar IV.15. Pengukuran latensi sebelum implementasi menunjukan rata-rata latensi *read request* adalah berturut-turut 1,20 ms, 0,50 ms, dan 3,05 ms untuk *workload* TPCC, DTRS, dan MSNFS.



(a) TPCC

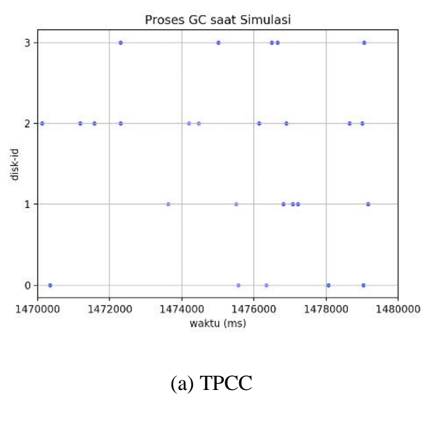
(b) DTRS

(c) MSNFS

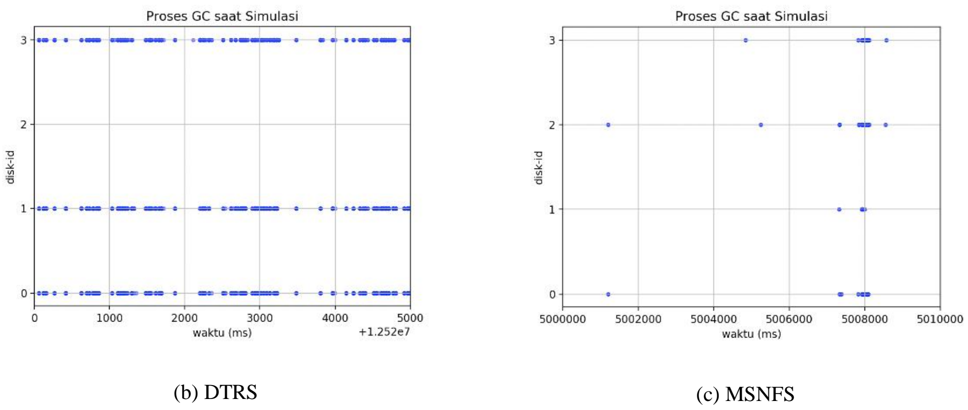
Gambar IV.15 Grafik GC *Collision* sebelum implementasi

Dari pengukuran awal ini didapatkan bahwa persentase *collision* yang melibatkan lebih dari dua GC adalah berturut-turut 0,98% 0,81% dan 1,06% untuk *workload* TPCC, DTRS dan MSNFS. Persentase jumlah GC dan GC *collision* tersebut sejalan

dengan persentil berbeloknya kurva pada grafik CDF yang dapat dilihat pada Gambar IV.17.

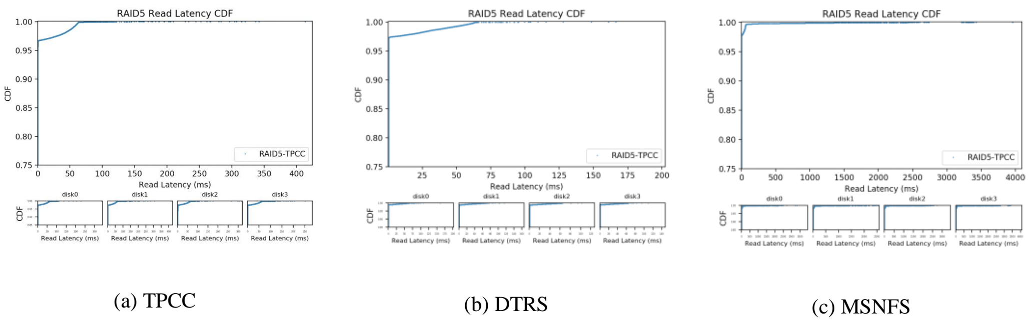


(a) TPCC



(b) DTRS (c) MSNFS

Gambar IV.16 Proses GC dalam simulasi sebelum integrasi



(a) TPCC (b) DTRS (c) MSNFS

Gambar IV.17 Grafik CDF pada RAID sebelum implementasi

IV.5.2 Hasil Pengujian GCSync

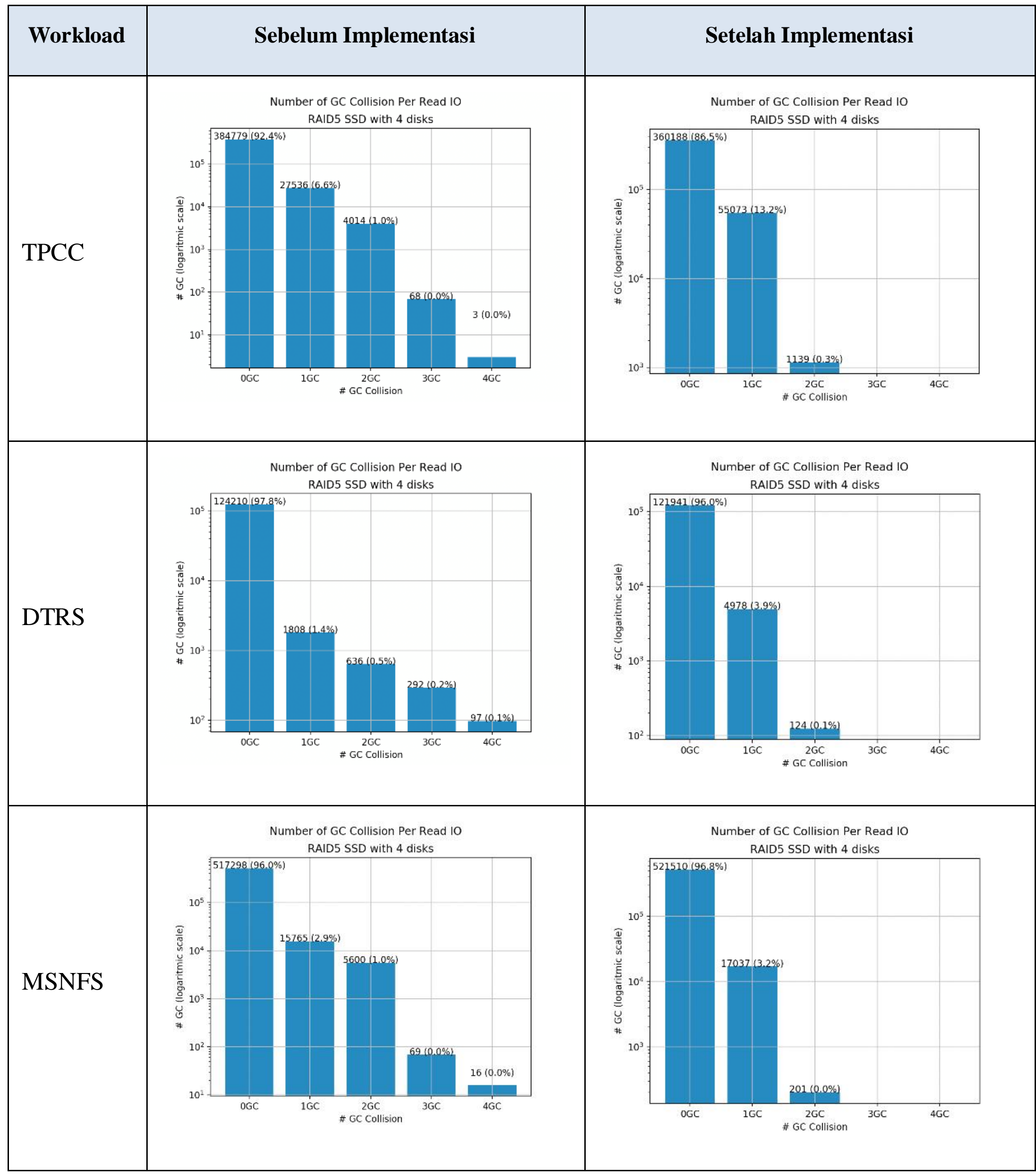
Metode GCSync berhasil diimplementasikan dalam SSDSim seperti yang sudah dijelaskan pada subbab IV.2.2. Secara visual perbandingan proses GC sebelum dan

sesudah implementasi dapat dilihat pada grafik GC yang ada pada Tabel IV.3. Pengujian ini dilakukan dengan mengatur *time window* pada setiap *disk* berdurasi selama 62.8ms. Pemilihan durasi tersebut mempertimbangkan durasi yang diperlukan untuk sebuah proses GC sesuai dengan *threshold* yang digunakan.

Tabel IV.3 Proses GC sebelum dan sesudah implementasi GCSync

Workload	Sebelum Implementasi	Setelah Implementasi
TPCC		
DTRS		
MSNFS		

Tabel IV.4 Jumlah GC *collision* sebelum dan setelah implementasi GCSync

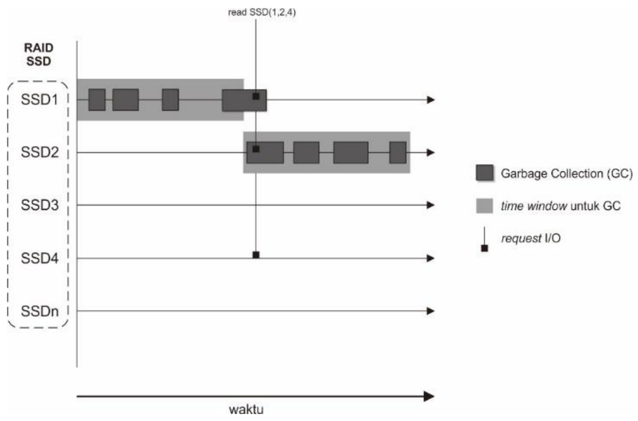


Workload	Sebelum Implementasi	Setelah Implementasi
TPCC		
DTRS		
MSNFS		

Implementasi GCSync mampu meminimasi *collision* dengan menurunkan nilai 2GC, dan menghilangkan 3GC serta 4GC. Berturut-turut pada *workload* TPCC, DTRS, dan MSNFS terjadi penurunan *collision* sebesar 65,72% 89,79% dan 94,27%,. Minimasi yang paling maksimum terjadi pada *workload* MSNFS.

Masih adanya 2GC pada setiap *workload* diakibatkan oleh GC pada salah satu *disk* yang terjadi pada akhir *time window* dan GC pada *disk* lainnya yang terjadi pada

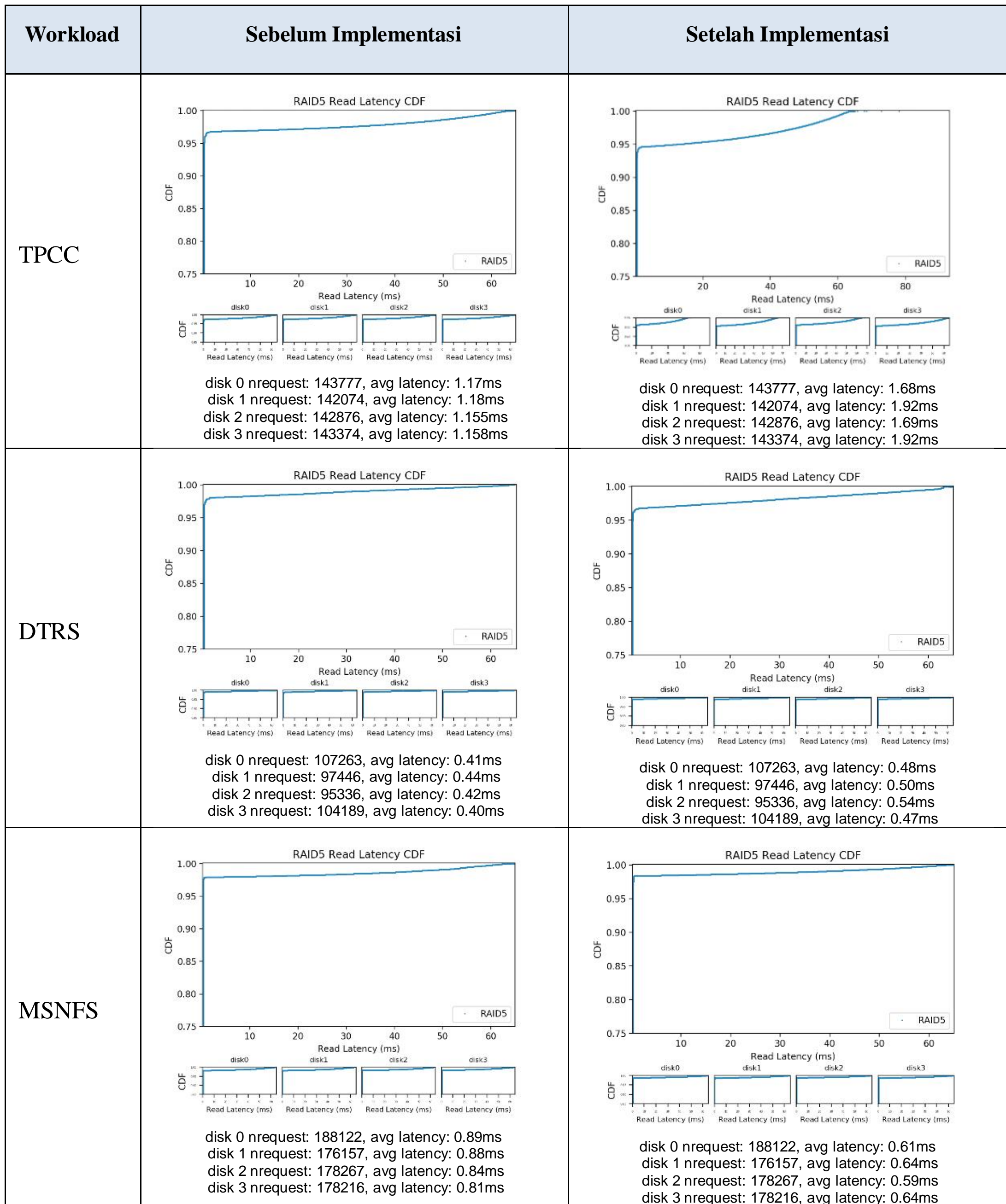
awal *time window disk* tersebut. *Read request* yang datang pada waktu tersebut, terpaksa masih harus menunggu 2 proses GC selesai terlebih dahulu, sebelum *request* tersebut dapat dieksekusi. Ilustrasi dari kejadian ini dapat dilihat pada Gambar IV.18, pada akhir *time window* pada SSD1 proses GC masih terjadi. Kemudian pada saat itu datang ke SSD1, SSD2, dan SSD4, maka request tersebut harus menunggu GC pada SSD1 dan SSD2, sehingga *collision* masih terjadi. Salah satu metode yang dapat digunakan untuk benar-benar menghilangkan 2GC tersebut adalah dengan menambahkan *time buffer* diantara dua *time window,* hal tersebut dilakukan pada strategi GCSync+.



Gambar IV.18 Ilusttrasi terjadinya 2GC pada GCSync

Walaupun GCSync mampu meminimasi *collision*, metode tersebut juga mengakibatkan kenaikan latensi pada RAID. Kenaikan latensi tersebut dapat dilihat pada Tabel IV.5. Pada simulasi *workload* TPCC dan DTRS berturut turut terjadi kenaikan latensi sebesar 1,54 dan 1,20 kali lipat. Sedangkan pada simulasi dengan workload MSNFS justru terjadi penurunan latensi sebesar 1,37 kali lipat.

Tabel IV.5 Grafik CDF latensi sebelum dan sesudah implementasi GCSync



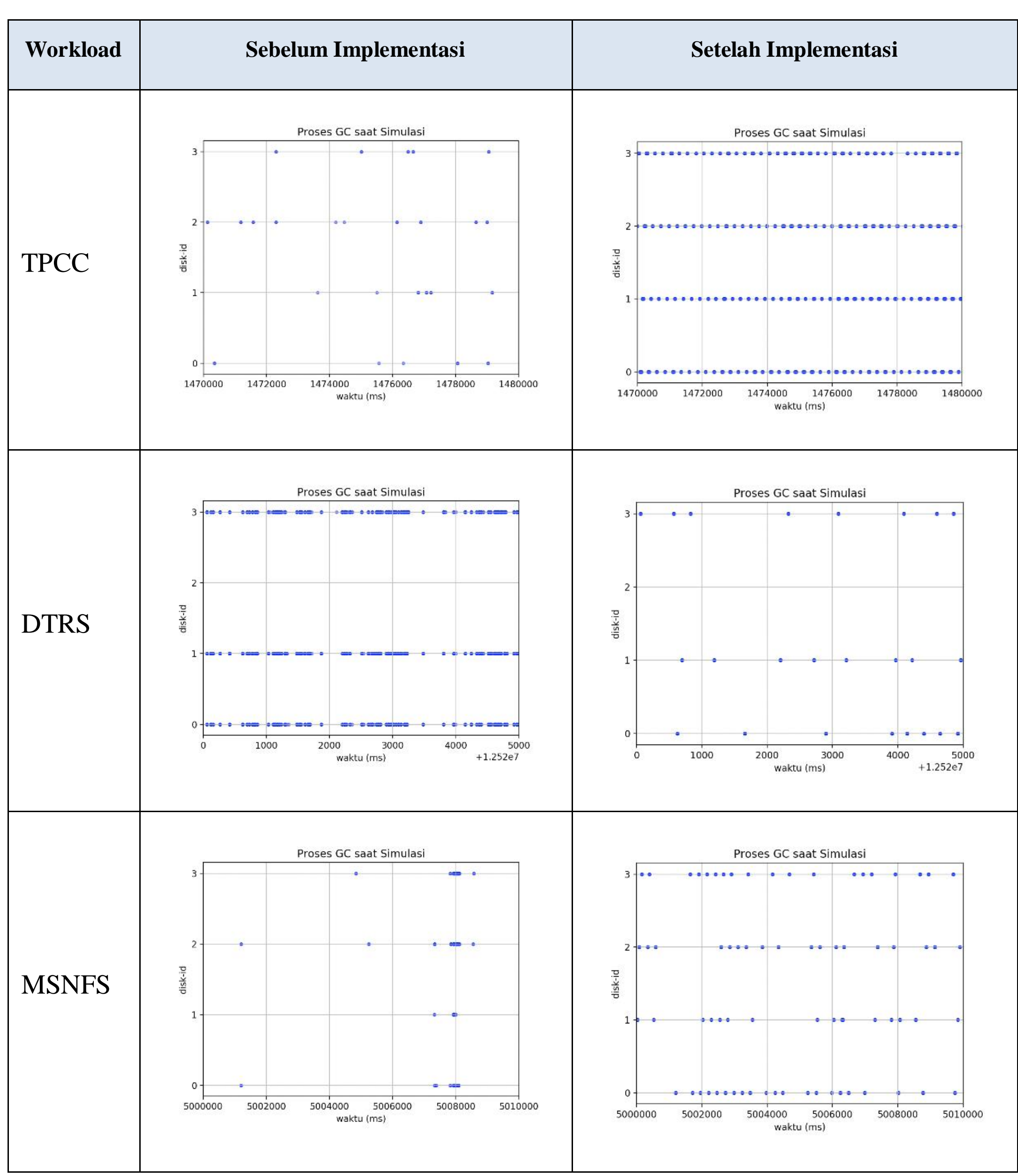
Workload	Sebelum Implementasi	Setelah Implementasi
TPCC	RAID5 Read Latency CDF disk 0 nrequest: 143777, avg latency: 1.17ms disk 1 nrequest: 142074, avg latency: 1.18ms disk 2 nrequest: 142876, avg latency: 1.155ms disk 3 nrequest: 143374, avg latency: 1.158ms	RAID5 Read Latency CDF disk 0 nrequest: 143777, avg latency: 1.68ms disk 1 nrequest: 142074, avg latency: 1.92ms disk 2 nrequest: 142876, avg latency: 1.69ms disk 3 nrequest: 143374, avg latency: 1.92ms
DTRS	RAID5 Read Latency CDF disk 0 nrequest: 107263, avg latency: 0.41ms disk 1 nrequest: 97446, avg latency: 0.44ms disk 2 nrequest: 95336, avg latency: 0.42ms disk 3 nrequest: 104189, avg latency: 0.40ms	RAID5 Read Latency CDF disk 0 nrequest: 107263, avg latency: 0.48ms disk 1 nrequest: 97446, avg latency: 0.50ms disk 2 nrequest: 95336, avg latency: 0.54ms disk 3 nrequest: 104189, avg latency: 0.47ms
MSNFS	RAID5 Read Latency CDF disk 0 nrequest: 188122, avg latency: 0.89ms disk 1 nrequest: 176157, avg latency: 0.88ms disk 2 nrequest: 178267, avg latency: 0.84ms disk 3 nrequest: 178216, avg latency: 0.81ms	RAID5 Read Latency CDF disk 0 nrequest: 188122, avg latency: 0.61ms disk 1 nrequest: 176157, avg latency: 0.64ms disk 2 nrequest: 178267, avg latency: 0.59ms disk 3 nrequest: 178216, avg latency: 0.64ms

IV.5.3 Hasil Pengujian GCSync+

Strategi GCSync+ berusaha menghilangkan *collision* yang masih terjadi pada strategi GCSync biasa, hal tersebut dilakukan dengan menambahkan *buffer time* antara dua *time window*. Strategi ini telah berhasil diimplementasikan dalam

SSDSim sesuai dengan penjelasan pada subbab IV.2.3. Pengujian GCSync+ dilakukan menggunakan *time window* sebesar 62,8ms dan *buffer time* juga sebesar 62,8ms. Pemilihan durasi *buffer time* dilakukan dengan mempertimbangkan durasi maksimum proses GC. Pemilihan *buffer time* yang terlalu lama dapat membuat SSD menunggu terlalu lama untuk menjalankan proses GC. Perubahan proses GC sebelum dan sesudah implementasi GCSync+ dapat dilihat pada Tabel IV.6.

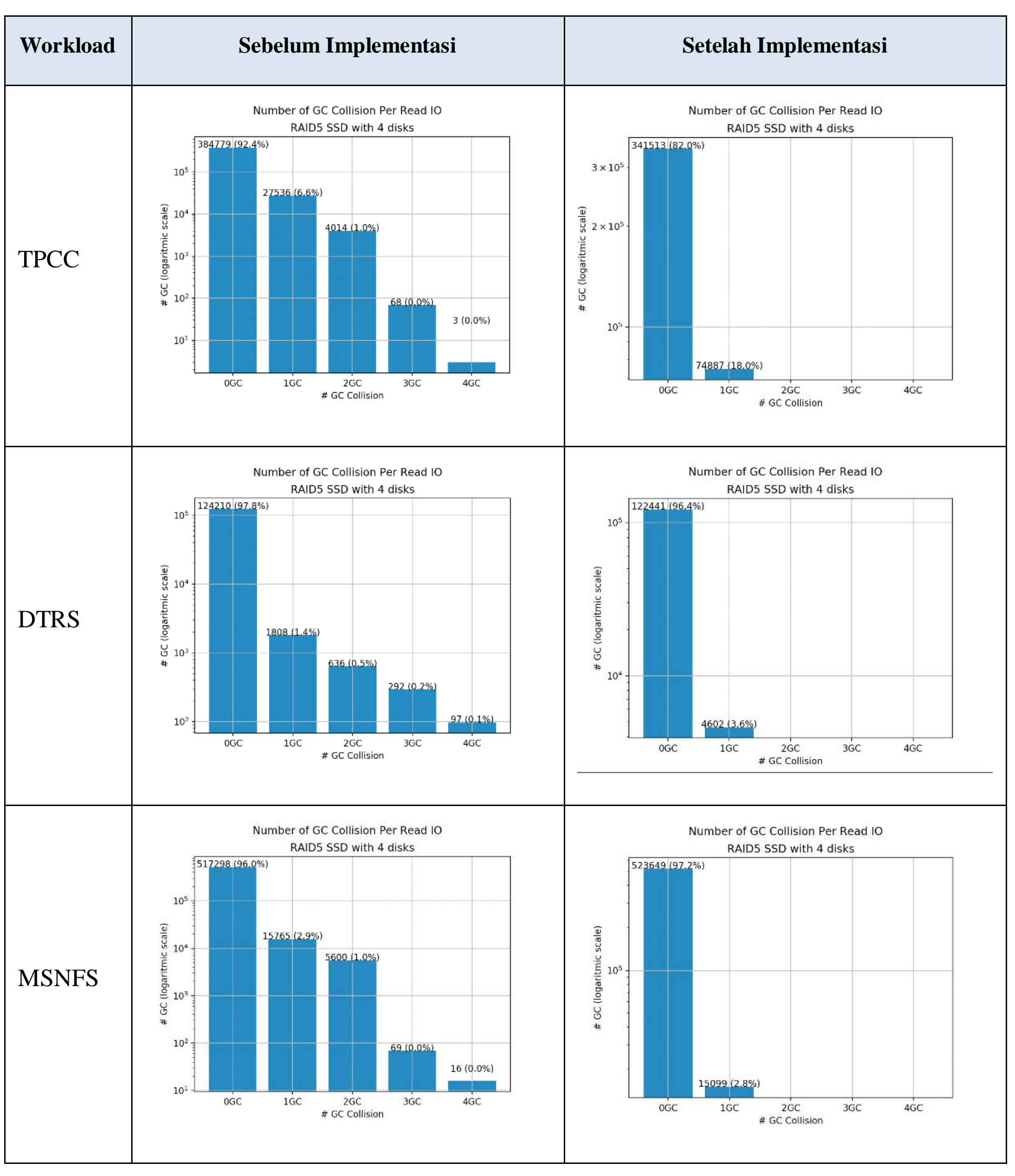
Tabel IV.6 Proses GC sebelum dan sesudah implementasi GCSync+



Workload	Sebelum Implementasi	Setelah Implementasi
TPCC		
DTRS		
MSNFS		

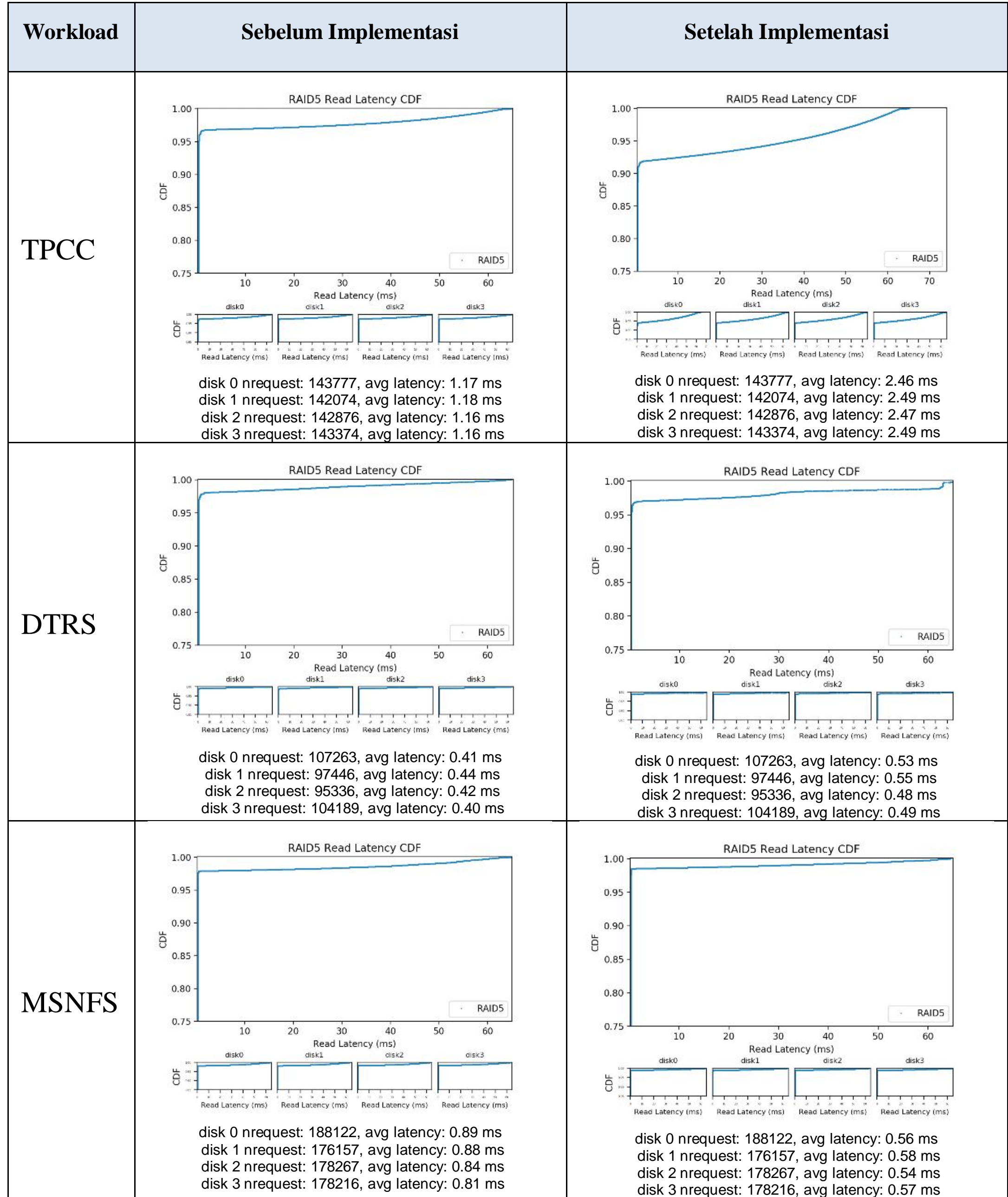
Pengujian strategi GCSync+ menunjukan efektifitas strategi tersebut untuk meminimasi *GC collision* pada RAID SSD, bahkan semua collision antara dua proses GC atau lebih dapat dihilangkan. Penggunaan *time window* dan *buffer time* yang tidak saling tumpang tindih antar SSD membuat tidak adanya *GC collision* yang terjadi. Jumlah *collision* sebelum dan sesudah implementasi GCSync+ dapat dilihat pada grafik *GC collision* pada Tabel IV.7.

Tabel IV.7 Jumlah GC *collision* sebelum dan setelah implementasi GCSync+



Workload	Sebelum Implementasi	Setelah Implementasi
TPCC		
DTRS		
MSNFS		

Tabel IV.8 Grafik CDF latensi sebelum dan sesudah implementasi GCSync+



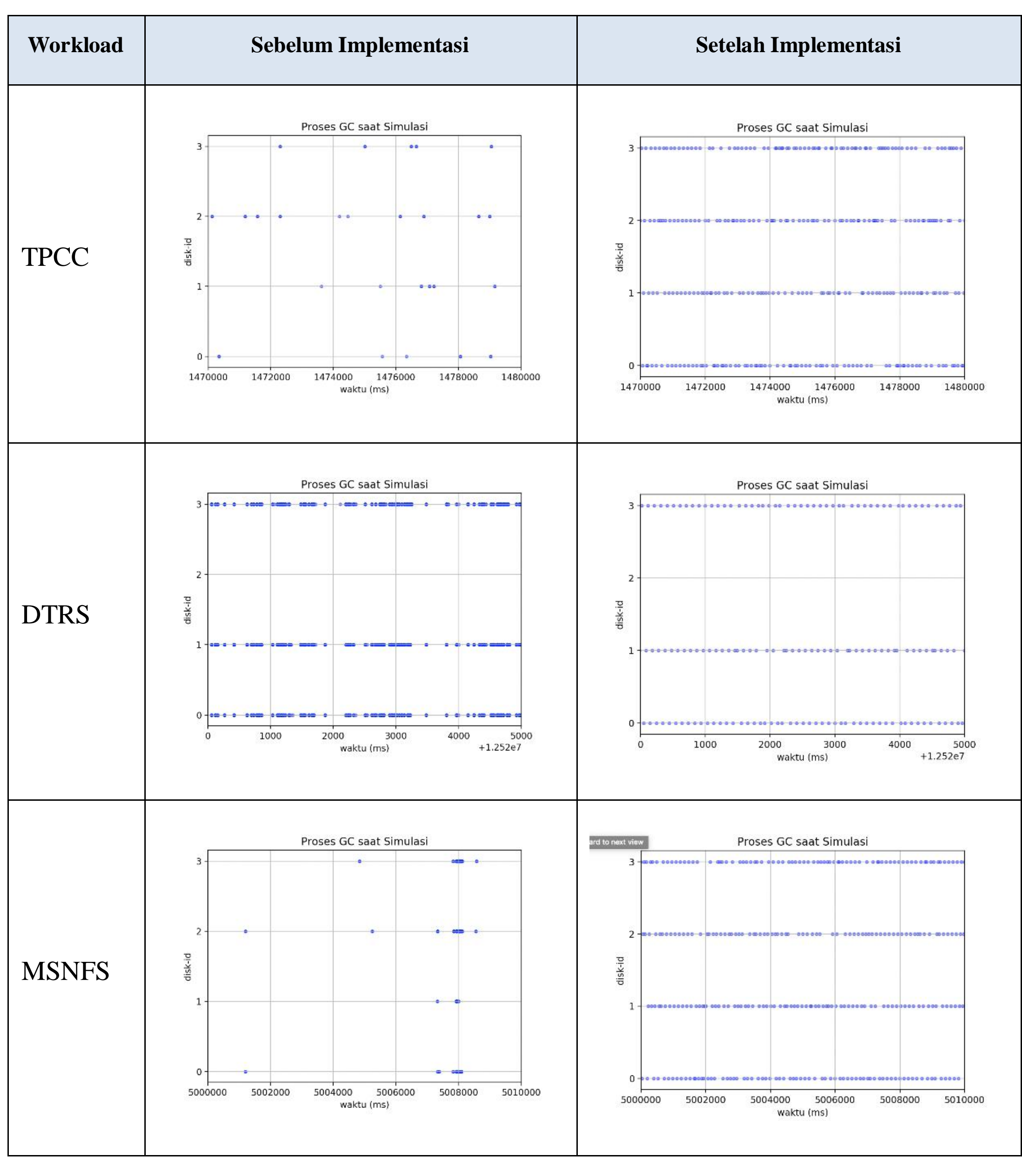
Workload	Sebelum Implementasi	Setelah Implementasi
TPCC	disk 0 nrequest: 143777, avg latency: 1.17 ms disk 1 nrequest: 142074, avg latency: 1.18 ms disk 2 nrequest: 142876, avg latency: 1.16 ms disk 3 nrequest: 143374, avg latency: 1.16 ms	disk 0 nrequest: 143777, avg latency: 2.46 ms disk 1 nrequest: 142074, avg latency: 2.49 ms disk 2 nrequest: 142876, avg latency: 2.47 ms disk 3 nrequest: 143374, avg latency: 2.49 ms
DTRS	disk 0 nrequest: 107263, avg latency: 0.41 ms disk 1 nrequest: 97446, avg latency: 0.44 ms disk 2 nrequest: 95336, avg latency: 0.42 ms disk 3 nrequest: 104189, avg latency: 0.40 ms	disk 0 nrequest: 107263, avg latency: 0.53 ms disk 1 nrequest: 97446, avg latency: 0.55 ms disk 2 nrequest: 95336, avg latency: 0.48 ms disk 3 nrequest: 104189, avg latency: 0.49 ms
MSNFS	disk 0 nrequest: 188122, avg latency: 0.89 ms disk 1 nrequest: 176157, avg latency: 0.88 ms disk 2 nrequest: 178267, avg latency: 0.84 ms disk 3 nrequest: 178216, avg latency: 0.81 ms	disk 0 nrequest: 188122, avg latency: 0.56 ms disk 1 nrequest: 176157, avg latency: 0.58 ms disk 2 nrequest: 178267, avg latency: 0.54 ms disk 3 nrequest: 178216, avg latency: 0.57 ms

Walaupun strategi GCSync+ dapat menghilangkan *GC collision*, pengaktifan strategi tersebut mengakibatkan perubahan latensi pada proses pembacaan data. Secara berturut-turut pada workload TPCC dan DTRS terjadi kenaikan latensi sebesar 2,13 dan 1,23 kali lipat. Sedangkan pengujian pada workload MSNFS justru

memperlihatkan penurunan latensi sebesar 1,53 kali lipat. Perubahan latensi tersebut dapat dilihat lebih detail pada grafik CDF dalam Tabel IV.8.

IV.5.4 Hasil Pengujian GCLock

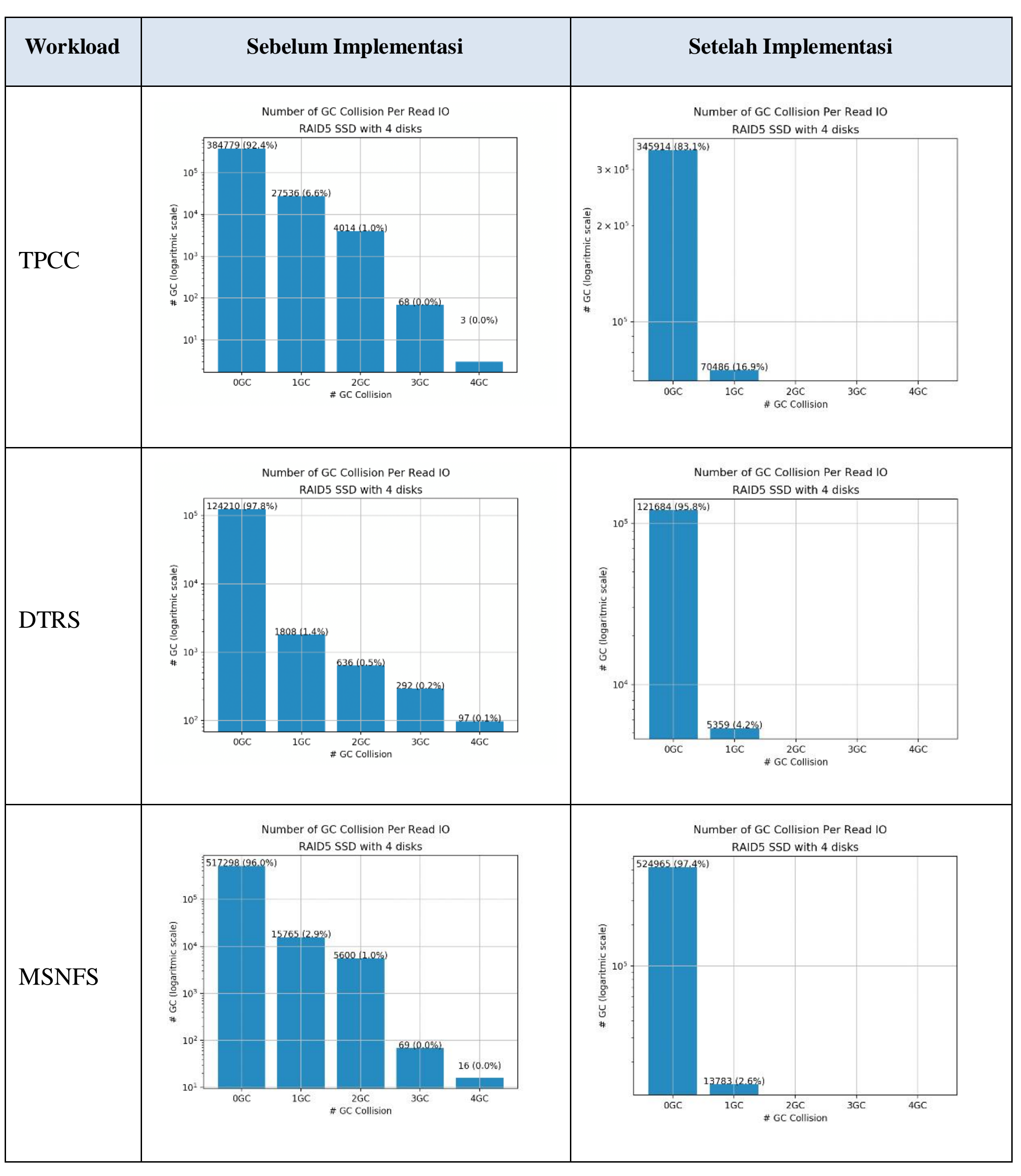
Tabel IV.9 Proses GC sebelum dan sesudah implementasi GCLock



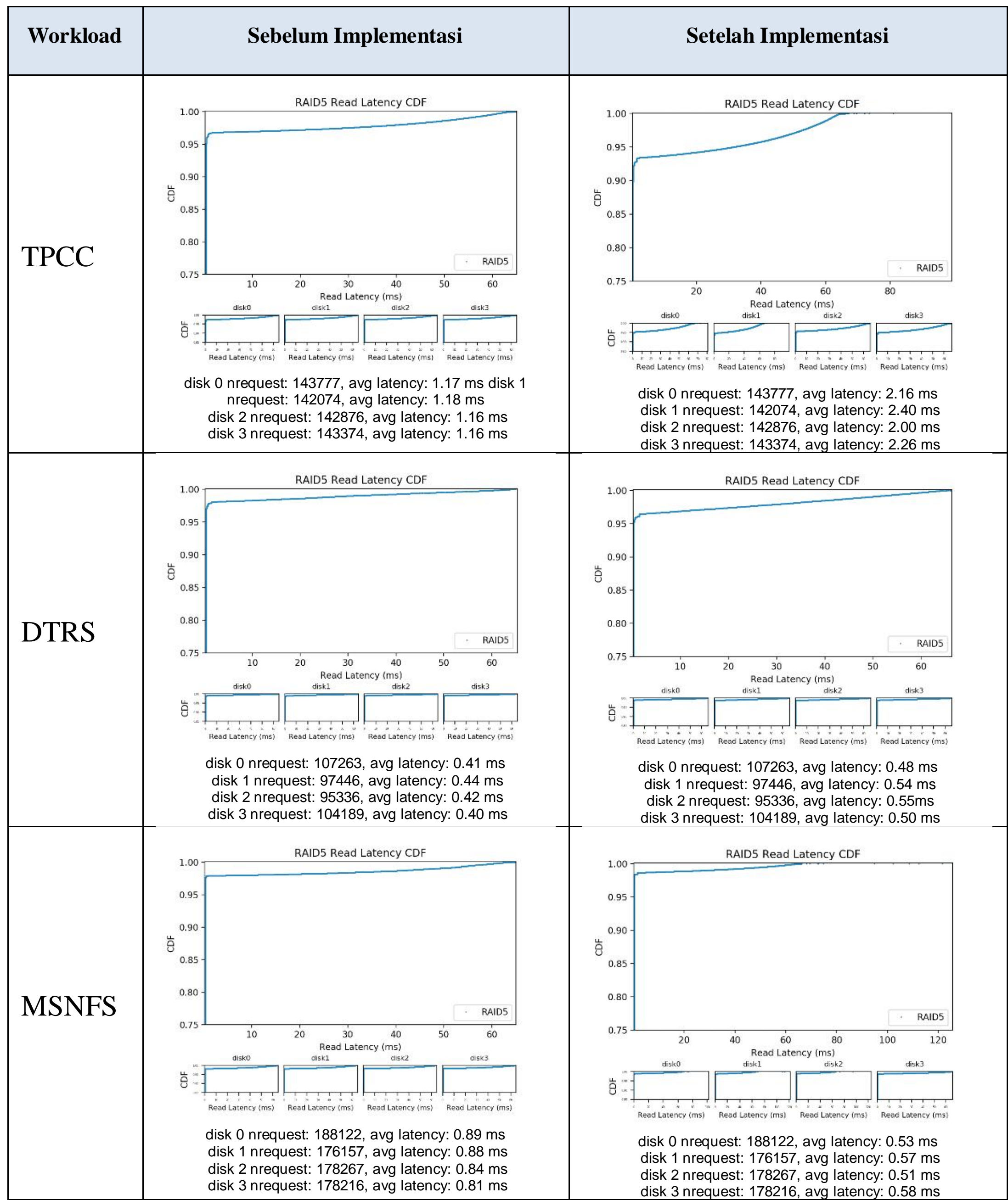
Workload	Sebelum Implementasi	Setelah Implementasi
TPCC		
DTRS		
MSNFS		

Metode GCLock mengharuskan SSD untuk mendapatkan *lock* dari *controller* sebelum SSD tersebut dapat melakukan proses GC. Pada penelitian ini, metode tersebut sudah berhasil diimplementasikan dalam SSDSim, mekanismenya seperti yang dijelaskan pada subbab IV.2.4. Jika dilihat secara sekilas melalui grafik GC pada Tabel IV.9, *collision* sudah dapat dikurangi dengan GCLock.

Tabel IV.10 Jumlah GC *collision* sebelum dan setelah implementasi GCLock



Tabel IV.11 Grafik CDF latensi sebelum dan sesudah implementasi GCLock



Workload	Sebelum Implementasi	Setelah Implementasi
TPCC	RAID5 Read Latency CDF disk 0 nrequest: 143777, avg latency: 1.17 ms disk 1 nrequest: 142074, avg latency: 1.18 ms disk 2 nrequest: 142876, avg latency: 1.16 ms disk 3 nrequest: 143374, avg latency: 1.16 ms	RAID5 Read Latency CDF disk 0 nrequest: 143777, avg latency: 2.16 ms disk 1 nrequest: 142074, avg latency: 2.40 ms disk 2 nrequest: 142876, avg latency: 2.00 ms disk 3 nrequest: 143374, avg latency: 2.26 ms
DTRS	RAID5 Read Latency CDF disk 0 nrequest: 107263, avg latency: 0.41 ms disk 1 nrequest: 97446, avg latency: 0.44 ms disk 2 nrequest: 95336, avg latency: 0.42 ms disk 3 nrequest: 104189, avg latency: 0.40 ms	RAID5 Read Latency CDF disk 0 nrequest: 107263, avg latency: 0.48 ms disk 1 nrequest: 97446, avg latency: 0.54 ms disk 2 nrequest: 95336, avg latency: 0.55ms disk 3 nrequest: 104189, avg latency: 0.50 ms
MSNFS	RAID5 Read Latency CDF disk 0 nrequest: 188122, avg latency: 0.89 ms disk 1 nrequest: 176157, avg latency: 0.88 ms disk 2 nrequest: 178267, avg latency: 0.84 ms disk 3 nrequest: 178216, avg latency: 0.81 ms	RAID5 Read Latency CDF disk 0 nrequest: 188122, avg latency: 0.53 ms disk 1 nrequest: 176157, avg latency: 0.57 ms disk 2 nrequest: 178267, avg latency: 0.51 ms disk 3 nrequest: 178216, avg latency: 0.58 ms

Grafik GC *collision* menunjukan pengurangan *collision* pada simulasi RAID yang dijalankan. Jika diukur secara kuantitatif, ternyata *collision* pada RAID SSD dapat dihilangkan dengan metode GCLock ini. Hilangnya *collision* tersebut dapat dilihat

lebih detail pada grafik GC *collision* pada Tabel IV.10. Semua *workload* yang digunakan untuk simulasi, baik TPCC, DTRS, maupun MSNFS menunjukan hilangnya 2GC, 3GC, dan 4GC. Hal tersebut menunjukan bahwa metode GCLock mampu mengeliminasi *collision* pada RAID SSD.

Seperti strategi penjadwalan GC sebelumnya, strategi GCLock juga mengakibatkan perubahan latensi saat pembacaam data. Pada simulasi *workload* TPCC dan DTRS berturut-turut terjadi kenaikan latensi sebesar 1,89 dan 1,25 kali lipat. Sedangkan pada simulasi workload MSNFS justru terjadi penurunan latensi sebesar 1,56 kali lipat. Hasil pengukuran latensi sebelum dan sesudah implementasi strategi GCLock dapat dilihat pada Tabel IV.11.

IV.6 Evaluasi Hasil Pengujian GCSync, GCSync+, dan GCLock

Hasil pengujian GCSync menunjukan adanya penurunan *GC collision* sebesar 65,72 – 95,27% dan pengaruh terhadap kenaikan latensi paling tinggi hanya sebesar 1,54 kali lipat. Bahkan pengujian dengan *workload* MSNFS menunjukan penurunan latensi sebesar 1,37 kali lipat. Oleh karena itu strategi GCSync dianggap cukup berhasil dalam meminimasi *collision* pada RAID SSD. Selajutnya untuk mengurangi *collision* yang masih tersisa dapat ditambahkan *buffer time* antar *time window* di SSD yang berbeda, hasil dari penambahan *buffer time* ditunjukan pada hasil pengujian GCSync+.

Pengujian GCSync+ menunjukan bahwa strategi tersebut dapat menghilangkan *GC collision* pada RAID SSD. Implementasi strategi GCSync+ juga mengakibatkan perubahan latensi pada proses pembacaan data. Hasil pengujian dengan berbagai *workload* menunjukan kenaikan latensi paling tinggi adalah sebesar 2,13 kali lipat pada *workload* TPCC. Pengukuran latensi dengan MSNFS justru menunjukan penurunan latensi sebesar 1,53 kali lipat.

Sedangkan hasil pengujian pada GCLock menunjukan bahwa strategi tersebut mampu menghilangkan semua *collision* pada RAID SSD. Hal tersebut karena strategi tersebut menggunakan *lock* yang memastikan dalam satu waktu hanya ada

satu SSD yang memegang *lock* tersebut. Walaupun *collision* dapat dihilangkan, metode GCLock juga mengakibatkan kenaikan latensi hingga 1,89 kali lipat.

Secara umum pengujian strategi GCSync, GCSync+, dan GCLock pada *workload* TPCC dan DTRS mengakibatkan kenaikan latensi yang tidak terlalu besar. Sedangkan pada pengujian dengan *workload* MSNFS, semua strategi penjadwalan GC mengakibatkan penurunan latensi antara 1,37 – 1,56 kali lipat. Adanya kenaikan dan penurunan latensi tersebut sangat berkaitan dengan jumlah *read request* yang tidak bertemu dengan proses GC sama sekali, atau jumlah 0GC jika pada grafik *GC collision*.

BAB V
KESIMPULAN DAN SARAN

V.1 Kesimpulan

1. Dalam penelitian ini telah berhasil dilakukan implementasi strategi minimasi GC *collision* pada RAID SSD yang dinamakan GCSync dan GCLock.
    a. GCSync menggunakan *time window* untuk menjadwalkan GC pada SSD yang terhubung dengan *controller* RAID. SSD hanya dapat menjalankan GC pada *time window* tertentu saja, selain itu *controller* juga memastikan tidak ada *time window* yang saling tumpang tindih.
    b. GCSync+ menggunakan *time window* untuk menjadwalkan GC, sama seperti GCSync. Selain itu juga ditambahkan *buffer time* antara dua time window sehingga GC *collision* dapat benar-benar dihilangkan.
    c. GCLock menggunakan *lock* yang ada pada controller RAID untuk menjadwalkan GC. SSD yang akan melakukan GC harus berusaha mendapatkan *lock* dari *controller* sebelum menjalankan proses GC. *Controller* memastikan dalam satu waktu hanya ada satu SSD saja yang memegang *lock* tersebut.
2. Strategi GCSync berhasil mengurangi *collision* sebesar 60-71% dan hanya memberikan kenaikan latensi antara 0,7-1,5 kali lipat.
3. Strategi GCSync+ berhasil menghilangkan *collision* dan memberikan kenaikan latensi antara 0,7-2,1 kali lipat.
4. Strategi GCLock juga berhasil menghilangkan seluruh *collision* pada RAID SSD, dan dari pengujian yang dilakukan, strategi ini memberikan kenaikan latensi sebesar 0,6-1,9 kali lipat.
5. Strategi penjadwalan GCSync, GCSync+ dan GCLock cukup efektif untuk meminimasi GC *collision* pada RAID SSD, bahkan strategi GCSync+ dan GCLock dapat menghilangkan seluruh GC *collision*.

V.2 Saran

1. Penelitian ini hanya berfokus pada pengurangan hingga penghilangan GC *collision* pada RAID SSD, sebaiknya perlu dilakukan penelitian lebih lanjut untuk mengukur dampak strategi yang diimplementasikan terhadap berbagai macam *tracefile* yang lebih beragam.
2. Untuk menghindari efek dari operasi dalam *operating system* (OS), penelitian ini dilakukan menggunakan simulator berbasis *tracefile*. Sebaiknya dilakukan penelitian lanjutan yang mengimplementasikan GCSync, GCSync+, dan GCLock pada simulator berbasis *virtual machine* untuk melihat pengaruh proses OS terhadap penjadwalan GC tersebut.
3. Penelitian lanjutan yang dapat dilakukan adalah dengan membuat metode rekonstruksi data pada RAID SSD, ketika proses *read* terhalang oleh proses GC. Proses rekonstruksi dapat dilakukan dengan maksimal ketika *GC collision* sudah diminimumkan dengan strategi GCSync, GCSync+, dan GCLock.

DAFTAR REFERENSI

Aggrawal, Nitin dkk. (2008). *Design Tradeoffs for SSD Performace*. In Proceedings of the USENIX Annual Technical Confrence (ATC), Boston, 2008.

Card, S.K., Robertson, G.G., dan Mackinlay, J.D. (1991). *The information visualizer: An information workspace*. Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (New Orleans, Apr. 28– May 2). ACM Press, New York, 1991, 181–188.)

Dean, Jeffrey dan Barroso, Luiz Andre. (2013). The Tail at Scale. Communications of the ACM, vol 56, pp. 74-80.

Eshghi K., Micheloni R. (2018). SSD Architecture and PCI Express Interface. Dalam: Micheloni R., Marelli A., Eshghi K. (eds) Inside Solid State Drives (SSDs). Springer Series in Advanced Microelectronics, vol 37. Springer, Singapore.

Hu, Yang dkk. (2011). *Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity*. Proceedings of the 25th International Confrence on Supercomputing (ICS '11), Munich.

Kavalanekar, S., Worthington, B., Qi Zhang, & Sharda, V. (2008). Characterization of storage workload traces from production Windows Servers. 2008 IEEE International Symposium on Workload Characterization. doi:10.1109/iiswc.2008.4636097.

Kim, Youngjae dkk. (2011). *Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-state Drives*. 7th IEEE Symposium on Massive Storage Systems and Technologies and Co-located Events (MSST 2011), Denver.

Lee, Sang-Won dan Kim, Jin-Soo. (2011). *Understanding SSDs with the OpenSSD Platform*. Seoul: Sungkyunkwan University.

Li, Huaicheng dkk. (2018). *The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Simulator*. Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18), Oakland.

Martindale, Jon. (2018). New laptops may see more storage as SSD prices expected to fall through 2019. https://www.digitaltrends.com/computing/ssd-prices-fall-2019/. Diakses pada 25 Oktober 2018.

Miceloni, Roni. (2017). *Solid-State-Drives (SSD) Modeling: Simulation Tools & Strategies.* Singapore: Springer.

Perumal, Sameshan dan Kritzinger, Pieter. (2004). *A Tutorial on RAID Storage Systems.* Cape Town: University of Cape Town.

Skourtis, Dimitris dkk. (2013). *High Performance & Low Latency in Solid-State Drives Through Redundancy*. Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13), San Jose.

Statista. (2017). Shipment of Hard and Solid-State Disk (HDD/SSD) Drives Worldwide from 2015 to 2021. https://www.statista.com/statistics/285474/hdds-and-ssds-in-pcs-global-shipments-2012-2017/. Diakses pada 30 September 2018.

Swanson, Steven. (2011). Flash Memory Overview. San Diego: University of California San Diego.

The OpenSSD Project. (2011). The Open SSD Project. http://www.openssd-project.org/wiki/The_OpenSSD_Project. Diakses pada 21 Desember 2018.

TPC. (1992). TPC-C: an On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/default.asp. Diakses pada 27 April 2019.

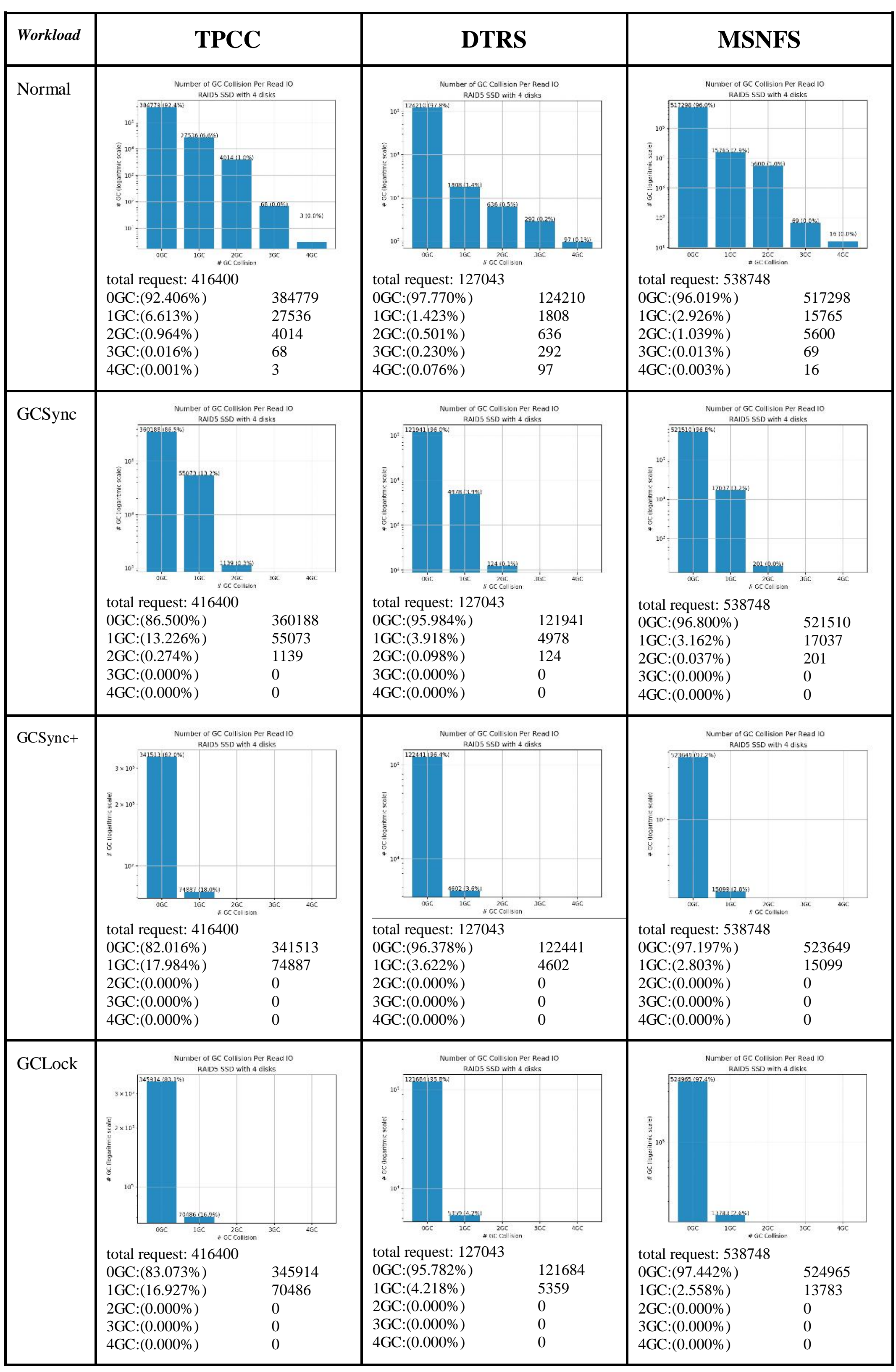
Yan, Shiqin dkk. (2017). *Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs*. Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17), Santa Clara.

Xu, Qiumin dkk. (2015). *Performance Analysis of NVMe SSDs and their Implication on Real World Databases*. Proceedings of the 8th ACM Internaional Systems and Storage Confrence (Systor '15). Haifa.

Yoo, Jinsoo dkk. (2013). *VSSIM: Virtual machine-based SSD simulator*. Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST' 13), Long Beach.

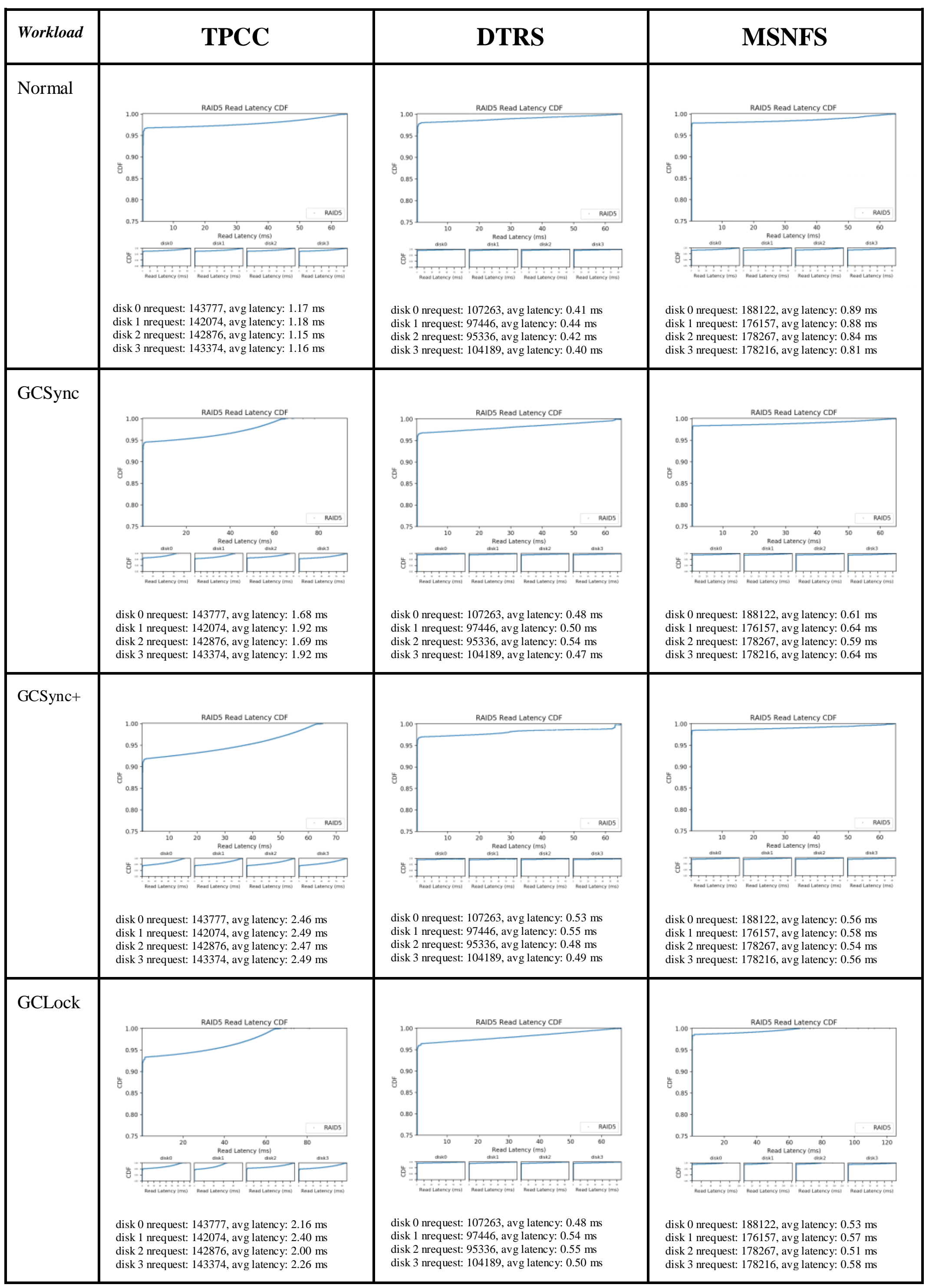
Zhang, Wenhui dkk. (2018). *PA-SSD: A Page-Type Aware TLC SSD for Improved Write/Read Performance and Storage Efficiency*. Proceedings of the 2018 International Conference on Supercomputing (ICS '18). ACM, New York, NY, USA, 22-32.

Lampiran A. Grafik *GC Collision* dari Pengujian



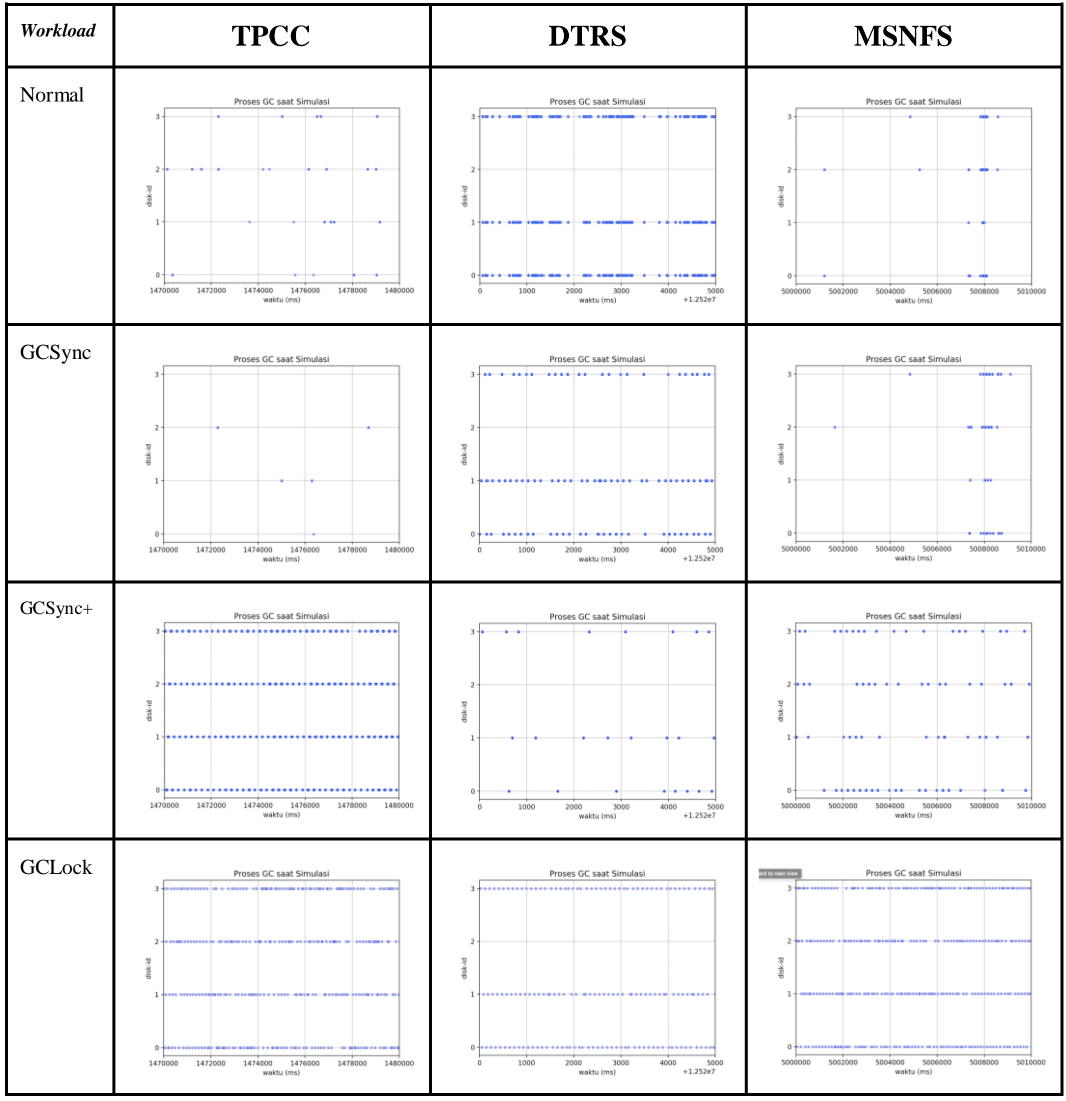
Workload	TPCC	DTRS	MSNFS
Normal	total request: 416400 0GC:(92.406%) 384779 1GC:(6.613%) 27536 2GC:(0.964%) 4014 3GC:(0.016%) 68 4GC:(0.001%) 3	total request: 127043 0GC:(97.770%) 124210 1GC:(1.423%) 1808 2GC:(0.501%) 636 3GC:(0.230%) 292 4GC:(0.076%) 97	total request: 538748 0GC:(96.019%) 517298 1GC:(2.926%) 15765 2GC:(1.039%) 5600 3GC:(0.013%) 69 4GC:(0.003%) 16
GCSync	total request: 416400 0GC:(86.500%) 360188 1GC:(13.226%) 55073 2GC:(0.274%) 1139 3GC:(0.000%) 0 4GC:(0.000%) 0	total request: 127043 0GC:(95.984%) 121941 1GC:(3.918%) 4978 2GC:(0.098%) 124 3GC:(0.000%) 0 4GC:(0.000%) 0	total request: 538748 0GC:(96.800%) 521510 1GC:(3.162%) 17037 2GC:(0.037%) 201 3GC:(0.000%) 0 4GC:(0.000%) 0
GCSync+	total request: 416400 0GC:(82.016%) 341513 1GC:(17.984%) 74887 2GC:(0.000%) 0 3GC:(0.000%) 0 4GC:(0.000%) 0	total request: 127043 0GC:(96.378%) 122441 1GC:(3.622%) 4602 2GC:(0.000%) 0 3GC:(0.000%) 0 4GC:(0.000%) 0	total request: 538748 0GC:(97.197%) 523649 1GC:(2.803%) 15099 2GC:(0.000%) 0 3GC:(0.000%) 0 4GC:(0.000%) 0
GCLock	total request: 416400 0GC:(83.073%) 345914 1GC:(16.927%) 70486 2GC:(0.000%) 0 3GC:(0.000%) 0 4GC:(0.000%) 0	total request: 127043 0GC:(95.782%) 121684 1GC:(4.218%) 5359 2GC:(0.000%) 0 3GC:(0.000%) 0 4GC:(0.000%) 0	total request: 538748 0GC:(97.442%) 524965 1GC:(2.558%) 13783 2GC:(0.000%) 0 3GC:(0.000%) 0 4GC:(0.000%) 0

Lampiran B. Grafik CDF Latensi *Read Request*



Workload	**TPCC**	**DTRS**	**MSNFS**
Normal	disk 0 nrequest: 143777, avg latency: 1.17 ms disk 1 nrequest: 142074, avg latency: 1.18 ms disk 2 nrequest: 142876, avg latency: 1.15 ms disk 3 nrequest: 143374, avg latency: 1.16 ms	disk 0 nrequest: 107263, avg latency: 0.41 ms disk 1 nrequest: 97446, avg latency: 0.44 ms disk 2 nrequest: 95336, avg latency: 0.42 ms disk 3 nrequest: 104189, avg latency: 0.40 ms	disk 0 nrequest: 188122, avg latency: 0.89 ms disk 1 nrequest: 176157, avg latency: 0.88 ms disk 2 nrequest: 178267, avg latency: 0.84 ms disk 3 nrequest: 178216, avg latency: 0.81 ms
GCSync	disk 0 nrequest: 143777, avg latency: 1.68 ms disk 1 nrequest: 142074, avg latency: 1.92 ms disk 2 nrequest: 142876, avg latency: 1.69 ms disk 3 nrequest: 143374, avg latency: 1.92 ms	disk 0 nrequest: 107263, avg latency: 0.48 ms disk 1 nrequest: 97446, avg latency: 0.50 ms disk 2 nrequest: 95336, avg latency: 0.54 ms disk 3 nrequest: 104189, avg latency: 0.47 ms	disk 0 nrequest: 188122, avg latency: 0.61 ms disk 1 nrequest: 176157, avg latency: 0.64 ms disk 2 nrequest: 178267, avg latency: 0.59 ms disk 3 nrequest: 178216, avg latency: 0.64 ms
GCSync+	disk 0 nrequest: 143777, avg latency: 2.46 ms disk 1 nrequest: 142074, avg latency: 2.49 ms disk 2 nrequest: 142876, avg latency: 2.47 ms disk 3 nrequest: 143374, avg latency: 2.49 ms	disk 0 nrequest: 107263, avg latency: 0.53 ms disk 1 nrequest: 97446, avg latency: 0.55 ms disk 2 nrequest: 95336, avg latency: 0.48 ms disk 3 nrequest: 104189, avg latency: 0.49 ms	disk 0 nrequest: 188122, avg latency: 0.56 ms disk 1 nrequest: 176157, avg latency: 0.58 ms disk 2 nrequest: 178267, avg latency: 0.54 ms disk 3 nrequest: 178216, avg latency: 0.56 ms
GCLock	disk 0 nrequest: 143777, avg latency: 2.16 ms disk 1 nrequest: 142074, avg latency: 2.40 ms disk 2 nrequest: 142876, avg latency: 2.00 ms disk 3 nrequest: 143374, avg latency: 2.26 ms	disk 0 nrequest: 107263, avg latency: 0.48 ms disk 1 nrequest: 97446, avg latency: 0.54 ms disk 2 nrequest: 95336, avg latency: 0.55 ms disk 3 nrequest: 104189, avg latency: 0.50 ms	disk 0 nrequest: 188122, avg latency: 0.53 ms disk 1 nrequest: 176157, avg latency: 0.57 ms disk 2 nrequest: 178267, avg latency: 0.51 ms disk 3 nrequest: 178216, avg latency: 0.58 ms

Lampiran C. Grafik Proses GC pada RAID SSD



Workload	TPCC	DTRS	MSNFS
Normal			
GCSync			
GCSync+			
GCLock			