

PENGEMBANGAN KEANDALAN SISTEM ARINC 653

DENGAN MENGGUNAKAN

FAULT-TOLERANT PARTITION SCHEDULING

Laporan Tugas Akhir

Disusun sebagai syarat kelulusan tingkat sarjana

Oleh

AUFAR GILBRAN

NIM: 13513015



**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG**

Agustus 2017

PENGEMBANGAN KEANDALAN SISTEM ARINC 653

DENGAN MENGGUNAKAN

FAULT-TOLERANT PARTITION SCHEDULING

Laporan Tugas Akhir

Oleh

AUFAR GILBRAN

NIM: 13513015

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan untuk diajukan sebagai Laporan Tugas Akhir

di Bandung, 22 Agustus 2017

Pembimbing,

Achmad Imam Kistijantoro, ST., M.Sc., Ph.D.

NIP. 197308092006041001

LEMBAR PERNYATAAN

Dengan ini saya menyatakan bahwa:

1. Pengerajan 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, 22 Agustus 2017

Aufar Gilbran

NIM 13513015

ABSTRAK

PENGEMBANGAN KEANDALAN SISTEM ARINC 653 DENGAN MENGGUNAKAN *FAULT-TOLERANT PARTITION SCHEDULING*

Oleh

AUFAR GILBRAN

NIM: 13513015

Standar ARINC 653 memberikan spesifikasi pada komponen-komponen milik sistem operasi guna menyediakan isolasi antara partisi pada sistem tersebut. Hal tersebut mengakibatkan kegagalan pada sebuah partisi tidak berakibat apapun pada partisi lain. Meskipun demikian, kegagalan pada partisi tersebut tetap terjadi. ARINC 653 memberikan spesifikasi komponen *health monitor* untuk mendeteksi dan menangani kegagalan yang terjadi pada sistem. Namun, beberapa kegagalan dapat bersifat persisten dan tidak dapat ditangani dengan metode penanganan yang dispesifikasikan oleh ARINC 653. Kegagalan pada partisi dapat mengakibatkan layanan yang diberikan oleh partisi tersebut tidak bekerja sebagaimana seharusnya sehingga mengakibatkan kegagalan sistem secara keseluruhan.

Pada Tugas Akhir ini, sebuah *scheduler* yang memanfaatkan skema *primary-backup* untuk meningkatkan keandalan sistem berbasis ARINC 653 telah dikembangkan. *Scheduler* dibangun dengan memodifikasi kode *scheduler* ARLX, yaitu prototipe sistem berbasis ARINC 653 yang dibangun di atas Xen hypervisor. Modifikasi dilakukan agar *scheduler* dapat melakukan penjadwalan partisi *backup* apabila partisi *primary* mengalami kegagalan.

Scheduler yang telah dikembangkan diuji untuk melihat apakah terjadi peningkatan keandalan serta mengukur *latency* maksimal pada sistem ketika menggunakan *scheduler* tersebut. Hasil pengujian menunjukkan peningkatan keandalan sistem secara signifikan, namun *latency* maksimal dari sebuah layanan masih tergolong terlalu tinggi sehingga *scheduler* tersebut belum layak digunakan pada lingkungan produksi.

Kata kunci: *hierarchical scheduling*, sistem *real-time*, *scheduling*, keandalan, *fault-tolerance*

KATA PENGANTAR

Puji dan syukur penulis ucapkan kepada Allah SWT atas berkat rahmat dan karunia-Nya selama ini sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul ”Pengembangan Keandalan Sistem ARINC 653 dengan Menggunakan *Fault-Tolerant Partition Scheduling*” dengan baik. Pada kesempatan ini, penulis juga ingin mengucapkan terima kasih yang sebesar-besarnya kepada:

1. Bapak Achmad Imam Kistijantoro, selaku dosen pembimbing Tugas Akhir, yang telah memberikan banyak masukan, saran, serta mengarahkan penulis untuk menyelesaikan permasalahan-permasalahan yang dihadapi selama pengerjaan Tugas Akhir.
2. Bapak Afwarman Manaf dan Ibu Mastura Diana Marieska selaku dosen penguji, yang telah memberikan kritik dan saran yang berharga sehingga hasil dari Tugas Akhir ini menjadi lebih baik.
3. Orang tua penulis, yang selalu memberikan motivasi untuk menyelesaikan Tugas Akhir kepada penulis.
4. Pipin Kurniawati, yang telah membantu penulis dalam menyusun isi dari laporan Tugas Akhir.
5. Ahmad Zaky, yang telah menjadi teman diskusi penulis dalam implementasi program selama pengerjaan Tugas Akhir.
6. Muhamad Visat Sutarno, yang telah membantu penulis menyelesaikan administrasi tugas akhir ketika penulis sakit.
7. Semua pihak yang telah membantu penulis dalam menyelesaikan tugas akhir ini baik secara langsung maupun tidak langsung.

Semoga Tugas Akhir ini dapat memberikan manfaat bagi semua pihak yang membutuhkannya.

Bandung, Agustus 2017

Penulis

DAFTAR ISI

ABSTRAK	iv
KATA PENGANTAR.....	v
DAFTAR ISI	vi
DAFTAR GAMBAR.....	ix
DAFTAR KODE PROGRAM.....	xii
BAB I. PENDAHULUAN	1
I.1 Latar Belakang	1
I.2 Rumusan Masalah.....	3
I.3 Tujuan	4
I.4 Batasan Masalah.....	5
I.5 Metodologi	5
I.6 Sistematika Pembahasan.....	6
BAB II. STUDI LITERATUR.....	7
II.1 ARINC 653.....	7
II.1.1 Sistem <i>Real-Time</i>	8
II.1.2 <i>Real-Time Scheduling</i>	10
II.1.3 Arsitektur Sistem.....	12
II.1.4 Fungsionalitas Sistem	13
II.1.4.1 Manajemen Partisi	13
II.1.4.2 Manajemen Waktu	14
II.1.5 <i>Health Monitoring</i>	15
II.2 <i>Fault-Tolerant Partition Scheduling</i>	17
II.3 Virtualisasi.....	18
II.4 <i>Hypervisor</i>	19
II.5 Xen	20
II.6 ARLX	22

BAB III. ANALISIS PERMASALAHAN DAN RANCANGAN SOLUSI	25
III.1 Analisis Permasalahan.....	25
III.2 Rancangan Solusi	26
III.2.1 Implementasi <i>Primary-Backup Partition Scheduling</i> pada ARLX.....	27
III.2.2 Daftar Jadwal.....	27
III.2.3 Informasi Partisi.....	28
III.2.4 Algoritma <i>Scheduling</i>	29
III.2.5 Mekanisme Penanganan Kegagalan.....	32
III.3 Rancangan Pengujian	33
III.3.1 Pengujian Keandalan	34
III.3.2 Pengujian <i>Latency</i> Layanan	36
BAB IV. PENGEMBANGAN DAN PENGUJIAN.....	38
IV.1 Pengembangan <i>Primary-Backup Partition Scheduling</i>	38
IV.1.1 Modul <i>scheduler</i>	39
IV.1.1.1 Pendefinisian <i>scheduler</i>	40
IV.1.2 Daftar Jadwal.....	41
IV.1.2.1 Informasi Partisi	43
IV.1.2.2 Fungsi <i>Scheduling</i>	44
IV.1.3 Modul <i>domain hypercall</i>	45
IV.1.4 Modul libxc	46
IV.1.4.1 Fungsi Pengiriman Informasi Partisi	47
IV.1.4.2 Fungsi Permintaan Informasi Partisi	48
IV.1.5 Modul libxl.....	48
IV.2 Pengujian.....	49
IV.2.1 Pengujian Keandalan	50
IV.2.2 Pengujian <i>Latency</i>	65
IV.2.3 Pembahasan Hasil Pengujian	72
BAB V. Kesimpulan dan Saran	79
V.1 Kesimpulan.....	79
V.2 Saran.....	80

DAFTAR PUSTAKA	82
-----------------------------	-----------

DAFTAR GAMBAR

Gambar II.1. Ilustrasi perbedaan permasalahan <i>scheduling</i> pada aplikasi <i>real-time</i> dan <i>non-real-time</i>	11
Gambar II.2. Contoh arsitektur <i>module</i>	12
Gambar II.3. Contoh <i>partition scheduling</i> pada sistem ARINC 653	15
Gambar II.4. Perbedaan penggunaan <i>ring native</i> dan <i>paravirtualized</i>	21
Gambar II.5. Jalur sebuah paket jaringan yang dikirim dari domU	22
Gambar II.6. <i>Hypervisor</i> ARLX (VanderLeest, 2010).....	23
Gambar II.7. <i>Schedule</i> partisi pada ARLX (VanderLeest, 2010)	23
Gambar III.1. Daftar jadwal layanan.....	28
Gambar III.2. Alur informasi sistem	29
Gambar III.3. Diagram <i>Flowchart</i> Algoritma <i>Scheduling</i>	30
Gambar III.4. Contoh <i>primary-backup scheduling</i>	32
Gambar III.5. Arsitektur sistem pengujian keandalan layanan	35
Gambar III.6. Komunikasi pada sistem pengujian keandalan layanan.....	36
Gambar IV.1. Hasil pengujian keandalan sistem menggunakan skenario 1	53
Gambar IV.2. Hasil pengujian keandalan sistem menggunakan skenario 2	54
Gambar IV.3. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 1)..	54
Gambar IV.4. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 2)..	55
Gambar IV.5. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 3)..	56
Gambar IV.6. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 4)..	56
Gambar IV.7. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 5)..	57

Gambar IV.8. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 6) ..	57
Gambar IV.9. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 7) ..	58
Gambar IV.10. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 8)	58
Gambar IV.11. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 9)	59
Gambar IV.12. Hasil pengujian keandalan sistem menggunakan skenario 4	60
Gambar IV.13. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 1)	60
Gambar IV.14. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 2)	61
Gambar IV.15. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 3)	62
Gambar IV.16. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 4)	63
Gambar IV.17. Hasil pengujian keandalan sistem menggunakan skenario 6 (layanan 1)	63
Gambar IV.18. Hasil pengujian keandalan sistem menggunakan skenario 6 (layanan 2)	64
Gambar IV.19. Hasil pengujian keandalan sistem menggunakan skenario 6 (layanan 3)	65
Gambar IV.20. Hasil pengujian <i>latency</i> sistem menggunakan skenario 1	67
Gambar IV.21. Hasil pengujian <i>latency</i> sistem menggunakan skenario 2 (layanan 1)	67
Gambar IV.22. Hasil pengujian <i>latency</i> sistem menggunakan skenario 2 (layanan 2)	68
Gambar IV.23. Hasil pengujian <i>latency</i> sistem menggunakan skenario 2 (layanan 3)	68
Gambar IV.24. Hasil pengujian <i>latency</i> sistem menggunakan skenario 2 (layanan 4)	69
Gambar IV.25. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 1)	70
Gambar IV.26. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 2)	70
Gambar IV.27. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 3)	71
Gambar IV.28. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 4)	71
Gambar IV.29. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 5)	72
Gambar IV.30. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 6)	73
Gambar IV.31. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 7)	73

Gambar IV.32. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 8)	74
Gambar IV.33. Hasil pengujian <i>latency</i> sistem menggunakan skenario 3 (layanan 9)	74
Gambar IV.34. Hasil pengujian <i>time drift</i> pada sistem dengan satu buah partisi yang harus dijadwalkan.....	75
Gambar IV.35. Hasil pengujian <i>time drift</i> pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 1).....	76
Gambar IV.36. Hasil pengujian <i>time drift</i> pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 2).....	76
Gambar IV.37. Hasil pengujian <i>time drift</i> pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 3).....	76
Gambar IV.38. Hasil pengujian <i>time drift</i> pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 4).....	77

DAFTAR KODE PROGRAM

Kode IV.1. Deklarasi fungsi <i>scheduling</i> pada ARLX	40
Kode IV.2. <i>Interface</i> dari fungsi <i>scheduler</i>	41
Kode IV.3. Struktur untuk menyimpan daftar jadwal	42
Kode IV.4. Struktur data untuk pemanggilan <i>global hypercall</i>	42
Kode IV.5. Struktur untuk menyimpan informasi partisi	43
Kode IV.6. Fungsi inisialisasi partisi.....	43
Kode IV.7. Fungsi penghapusan partisi	44
Kode IV.8. Implementasi algoritma <i>scheduling</i>	45
Kode IV.9. Data struktur untuk komunikasi yang dapat dimengerti <i>scheduler</i> dan <i>user space</i>	45
Kode IV.10. Data struktur untuk pemindahan data pada saat melakukan <i>domain hyper-call</i>	46
Kode IV.11. Fungsi yang digunakan <i>user space</i> mengirim informasi kepada <i>scheduler</i>	47
Kode IV.12. Fungsi yang digunakan <i>user space</i> meminta informasi partisi dari <i>scheduler</i>	48
Kode IV.13. xl berkomunikasi dengan menggunakan fungsi milik xc.....	49

BAB I

PENDAHULUAN

I.1 Latar Belakang

Sistem penerbangan atau avionik adalah sistem yang terdiri dari perangkat elektronik yang digunakan pada pesawat terbang. Avionik berfungsi untuk memberikan fasilitas untuk mengoperasikan pesawat terbang. Sistem yang termasuk dalam avionik adalah sistem komunikasi, navigasi, *dashboard* untuk manajemen sistem, dan sistem-sistem yang dipasang pada pesawat terbang untuk melakukan fungsionalitas tersendiri.

Avionik merupakan komponen penting dalam sebuah pesawat terbang dan umumnya merupakan *safety-critical system*, yaitu sistem yang apabila mengalami kegagalan dapat mengakibatkan kematian, kerugian besar, atau kerusakan lingkungan. Sistem operasi yang digunakan oleh avionik pada sebuah pesawat umumnya merupakan sistem *real-time*, yaitu sistem yang kebenaran komputasinya tidak hanya ditentukan oleh kebenaran hasil komputasi secara logika, tetapi juga waktu pada saat hasil komputasi dapat digunakan (K. G. Shin dan Ramanathan, 1994, p. 6). Dengan demikian, manajemen waktu komputasi adalah aspek yang sangat mendasar pada sistem *real-time*. Perangkat yang digunakan untuk avionik memiliki beberapa standar yang wajib dipenuhi. Standar tersebut dibuat oleh Aeronautical Radio, Incorporated (ARINC) dalam sebuah seri standar ARINC 600. Penelitian ini akan mendalami lebih lanjut mengenai sistem operasi *real-time* pada avionik yang dispesifikasi-an pada penerbangan bernama ARINC 653.

Standar ARINC 653 menspesifikasi-an partisi ruang dan waktu pada sistem operasi *real-time* agar avionik yang menggunakan sistem operasi tersebut *safety-critical*. Setiap proses aplikasi disebut sebagai partisi. Selain spesifikasi partisi, ARINC 653 juga mendefinisikan API untuk memudahkan manajemen partisi. Sistem operasi yang memenuhi standar ARINC 653 berfungsi untuk memanajemeni beberapa partisi beserta komunikasi antar partisi. Manajemen yang dilakukan meliputi *resources* komputasi umum seperti CPU, *timing*, memori, dan *devices* serta *resources* untuk komunikasi. Penelitian ini akan mendalami lebih lanjut

mengenai manajemen CPU, khususnya proses *partition scheduling*, yang dispesifikasi oleh standar ARINC 653.

Partisi pada standar ARINC 653 dapat membuat eksekusi dan pembaharuan perangkat lunak pada sistem tersebut lebih tepercaya dan fleksibel (Jin, 2013). Partisi dapat diganti atau ditambahkan selama partisi tersebut telah mendapatkan sertifikasi. Sebagai contoh, aplikasi navigasi yang dijalankan pada suatu partisi tertentu pada avionik dapat diganti dengan versi terbaru yang menggunakan algoritma navigasi yang lebih baik tanpa mengganggu partisi lain. Namun, standar ARINC 653 tidak menghasilkan sistem yang andal. Akan terdapat kemungkinan terjadinya *fault* pada perangkat lunak selama sistem beroperasi, meskipun keamanan dan kebenaran perangkat lunak tersebut sudah terverifikasi.

Permasalahan pada partisi yang mengalami kegagalan adalah layanan yang disediakan oleh partisi tersebut akan berhenti bekerja sebagaimana seharusnya. Pada avionik, hal ini dapat berakibat terjadinya kesalahan kalkulasi, sehingga meningkatkan ancaman keselamatan bagi manusia maupun lingkungan yang terlibat. Standar ARINC 653 memberikan spesifikasi komponen untuk melakukan *health monitoring*. Dengan adanya proses *health monitoring*, sistem dapat mendeteksi kegagalan yang terjadi dan memberikan respons yang sesuai berdasarkan jenis kegagalan yang terjadi. Namun, dalam kasus tertentu, keadaan partisi tidak dapat dikembalikan seperti semula karena kegagalan yang terjadi bersifat tetap. Oleh karena itu, perlu ada mekanisme untuk menggantikan layanan pada partisi yang mengalami kegagalan.

Untuk meningkatkan keandalan sistem dalam sistem *real-time*, salah satu cara yang dapat digunakan adalah dengan menggunakan *fault-tolerant scheduling* (Liestman dan Campbell, 1986) (Han dkk. 2003) (I. Shin dan Lee, 2008). Sebuah studi menunjukkan bahwa peningkatan keandalan *hierarchical scheduler* seperti *scheduler* ARINC 653 dapat dilakukan dengan menggunakan skema *primary-backup* (Hyun dan Kim, 2012). Untuk menghindari permasalahan yang akan muncul akibat partisi yang gagal, *scheduler* dapat menjalankan partisi pengganti yang memiliki layanan yang bersesuaian dengan partisi yang mengalami kegagalan. Partisi pengganti ini disebut sebagai partisi *backup*. Layanan yang diberikan partisi *backup* akan identik dengan layanan yang diberikan oleh partisi *primary*. Apabila partisi *primary* mengalami kegagalan ketika menyediakan sebuah layanan, partisi *backup*

akan melindungi sistem dari kegagalan dengan mengambil alih peran partisi *primary* sebagai penyedia layanan. Hal tersebut mengakibatkan layanan yang disediakan oleh partisi yang mengalami *fault* akan tetap tersedia dan sistem akan tetap berjalan sebagaimana seharusnya, sehingga sistem dapat dikatakan menjadi lebih andal.

Meski demikian, peningkatan tersebut hanya terlihat secara teoretis dan belum ada studi mengenai penggunaan *primary-backup scheduling* secara spesifik pada sistem ARINC 653. Selain itu, tidak ada implementasi sistem ARINC 653 menggunakan *primary-backup scheduling* yang tersedia secara gratis sehingga peningkatan keandalan sistem dengan memanfaatkan *primary-backup scheduling* sulit untuk dikuantifikasi. Untuk dapat melihat perbandingan antara *primary-backup partition scheduling* dengan *scheduling* sistem ARINC 653, perlu dilakukan pengujian keandalan sistem pada sistem yang menggunakan *primary-backup partition scheduling* dan sistem yang menggunakan *scheduling* seperti pada spesifikasi ARINC 653.

Arsitektur komputer yang dispesifikasikan pada standar ARINC 653 dapat divirtualisasikan dengan menggunakan *hypervisor*. *Hypervisor* adalah sebuah komponen yang membuat dan menjalankan *virtual machine*. Setiap *virtual machine* akan memiliki ruang memori dan waktu tersendiri sehingga dapat digunakan untuk mengimplementasikan sebuah partisi. Salah satu solusi sistem ARINC 653 yang menggunakan *hypervisor* adalah Xen ARINC 653. AR-LX adalah sebuah prototipe sistem ARINC 653 yang dibangun di atas Xen, yaitu *hypervisor open-source* dengan lisensi GPL sehingga ARLX juga *open-source*. Solusi *open-source* akan mempermudah pengembang aplikasi dan peneliti untuk melakukan pengembangan dan percobaan pada lingkungan ARINC 653 sebelum membeli solusi berbayar. Xen ARINC 653 merupakan solusi yang sangat tepat untuk melakukan studi penggunaan *fault-tolerant partition scheduling* dan perbedaan antara sistem yang menggunakan *fault-tolerant partition scheduling* dengan sistem yang menggunakan *partition scheduling* pada spesifikasi ARINC 653.

I.2 Rumusan Masalah

Permasalahan yang terjadi berdasarkan uraian latar belakang adalah standar ARINC 653 tidak membuat sistem menjadi andal. Padahal, fungsi dari standar ARINC 653 adalah seba-

gai panduan pengembangan sistem operasi *real-time* pada *safety-critical system*. Masalah tersebut dapat diselesaikan dengan menggunakan *fault-tolerant partition scheduling*. Meski *fault-tolerant partition scheduling* membuat sistem menjadi andal secara teoretis, perlu adanya studi lebih lanjut apakah metode *fault-tolerant partition scheduling* dapat digunakan pada sistem ARINC 653. Selain itu, perlu adanya perbandingan antara *scheduling* normal dengan *fault-tolerant partition scheduling* pada ARINC 653 untuk melihat kemungkinan peningkatan atau penurunan efisiensi sistem.

Berikut adalah pertanyaan permasalahan yang akan dijawab dan diselesaikan pada tugas akhir ini.

1. Bagaimana cara mengimplementasikan *fault-tolerant partition scheduling* pada sistem ARINC 653 sehingga keandalan sistem meningkat?
2. Bagaimana cara melakukan pengujian keandalan sistem ARINC 653 setelah menggunakan *fault-tolerant partition scheduling*?
3. Apakah terjadi peningkatan keandalan sistem akibat penggunaan *fault-tolerant partition scheduling* pada sistem yang memenuhi standar ARINC 653?
4. Apakah terdapat peningkatan atau penurunan kinerja akibat peningkatan keandalan menggunakan *fault-tolerant partition scheduling* pada sistem yang memenuhi standar ARINC 653?

I.3 Tujuan

Berdasarkan uraian pada rumusan masalah, tujuan yang ingin dicapai adalah meningkatkan keandalan sistem ARINC 653 tanpa mengurangi efisiensi sistem secara signifikan. Hal ini dapat dicapai dengan melakukan hal-hal berikut.

1. Mengimplementasikan *fault-tolerant partition scheduling* pada sistem ARINC 653.
2. Melakukan pengujian keandalan sistem.
3. Melakukan perbandingan keandalan sistem pada saat sebelum dan sesudah implementasi *fault-tolerant partition scheduling*.

I.4 Batasan Masalah

Berdasarkan rumusan masalah yang ada, terdapat beberapa asumsi terkait dengan sistem avionik yang akan dikembangkan yaitu.

1. Implementasi hanya akan dilakukan pada sistem dengan arsitektur x86-64.
2. Implementasi hanya akan diuji coba dengan menggunakan sistem operasi Linux untuk setiap partisinya. Hasil pada sistem operasi lain mungkin berbeda.

I.5 Metodologi

Metodologi yang akan digunakan dalam penelitian tugas akhir adalah sebagai berikut.

1. Studi Literatur

Pengerjaan tugas akhir diawali dengan mempelajari referensi berupa jurnal ilmiah dan dokumen resmi terkait dengan spesifikasi ARINC 653, arsitektur Xen, dan cara kerja ARLX. Pembelajaran mengenai spesifikasi ARINC 653 dilakukan untuk dapat mengetahui kriteria penilaian yang akan digunakan sebagai patokan dalam pembuatan kerangka pengujian. Pembelajaran arsitektur Xen dilakukan untuk mengetahui cara memodifikasi ARLX terkait dengan cara melakukan modifikasi dan pengujian pada prototipe tersebut akan dilakukan.

2. Analisis

Pada tahap ini dilakukan analisis permasalahan yang berkaitan dengan topik tugas akhir ini. Analisis permasalahan akan membahas spesifikasi ARINC 653 dan arsitektur Xen untuk kemudian dijadikan sebagai solusi terhadap permasalahan.

3. Perancangan Solusi dan Pengujian

Pada tahap ini dilakukan perancangan solusi dan pengujian yang akan dilakukan yang dapat menyelesaikan masalah-masalah yang telah dijelaskan pada Subbab I.4.

4. Pengembangan dan Pengujian

Pada tahap ini dilakukan pembangunan sistem pengujian serta pengembangan modul pengujian pada ARLX.

5. Kesimpulan dan Saran

Pada tahap ini akan dilakukan analisis hasil pengembangan dan pengujian serta efek-

nya terhadap sistem. Hasil analisis akan digunakan dalam penarikan kesimpulan serta pembuatan saran untuk penelitian selanjutnya.

I.6 Sistematika Pembahasan

Struktur laporan tugas akhir ini adalah sebagai berikut.

1. Bab I Pendahuluan: Bab ini berisi latar belakang, masalah yang dibahas, tujuan, batasan masalah, metodologi, dan sistematika pembahasan tugas akhir.
2. Bab II Studi Literatur: Bab ini berisi dasar teori yang digunakan pada tugas akhir.
3. Bab III Analisis Masalah dan Perancangan Solusi: Bab ini berisi analisis terhadap permasalahan yang telah dipaparkan pada Pendahuluan serta perancangan solusi yang dapat menyelesaikan permasalahan tersebut.
4. Bab IV Pengembangan dan Pengujian: Bab ini menjelaskan pembangunan solusi pada ARLX secara rinci.
5. Bab V Kesimpulan dan Saran: Bab ini berisi kesimpulan dari hasil pengujian dan saran untuk penelitian selanjutnya.

BAB II

STUDI LITERATUR

II.1 ARINC 653

Standar ARINC 653 menspesifikasikan partisi ruang dan waktu pada sistem operasi *real-time* avionik. Avionik adalah sebuah perangkat elektronik yang berfungsi untuk memberikan fasilitas untuk mengoperasikan pesawat terbang. Avionik umumnya merupakan sistem *safety-critical*, yang berarti kegagalan pada sistem dapat berakibat terjadinya hal-hal berikut.

1. Kematian atau cedera serius pada manusia.
2. Hilangnya atau rusaknya perlengkapan atau properti.
3. Kerusakan lingkungan.

Pada avionik, *Integrated Modular Avionics* (IMA) merupakan sebuah konsep yang bertujuan memperkecil penggunaan daya dan meringankan perangkat (Garside dan Pighetti, 2009, p. 2.A.2-1). Dalam arsitektur IMA, fungsionalitas avionik yang diharapkan menggunakan *resources* yang sama. Arsitektur IMA melakukan optimasi dengan cara memberikan *resource* seminimum mungkin pada masing-masing fungsionalitas avionik. Arsitektur ini memungkinkan eksekusi beberapa aplikasi avionik pada sebuah komputer yang sama. Hal tersebut dapat dicapai apabila sistem melakukan mekanisme partisi aplikasi avionik untuk mengisolasi eksekusi antar aplikasi, sehingga menjamin aplikasi dari satu partisi tidak dapat mengubah aplikasi, *devices*, aktuator, maupun data pada partisi lain (Rushby, 2000, pp. 11-12). Pada IMA, keseluruhan sistem disebut sebagai *integrated module* yang terdiri dari *core module* dan aplikasi. Sebuah *core module* terdiri atas *core software* dan *core hardware* yang menyediakan fungsionalitas *core module* dari sisi perangkat lunak dan perangkat keras. *Core module* berfungsi untuk menjadi tempat eksekusi aplikasi dan menyediakan layanan untuk melakukan partisi. Pada buku ini, istilah *integrated module* dan *core software* dapat dipertukarkan dengan *module* dan sistem operasi.

Standar ARINC 653 bertujuan untuk mengurangi biaya pengembangan, ukuran sistem, dan biaya sertifikasi dengan cara menggabungkan beberapa aplikasi pada satu komputer namun

tetap menjaga aplikasi-aplikasi tersebut saling terisolasi (Airlines Electronic Engineering Committee, 2012, pp. 3-30). Untuk mencapai tujuan tersebut, ARINC 653 mendefinisikan *interface* yang disebut sebagai *Application Executive* (APEX) untuk manajemen partisi, proses dan *timing*, komunikasi antar proses/partisi, dan penanganan *error* pada arsitektur IMA. Pada ARINC 653, satu unit partisi aplikasi disebut sebagai partisi.

Setiap partisi pada dasarnya sama seperti sistem dengan satu aplikasi karena partisi memiliki *device*, aktuator, konteks, dan data tersendiri. Hal tersebut mengakibatkan sistem memiliki partisi ruang dan waktu, sehingga masing-masing partisi tidak akan mempengaruhi partisi lain pada sistem. Proses dalam sebuah partisi diperbolehkan untuk melakukan *multitasking*.

Perangkat lunak pada *platform* ARINC 653 meliputi.

1. Partisi aplikasi avionik yang dispesifikasikan secara langsung oleh ARINC 653. Partisi aplikasi adalah target utama dari partisi ruang dan waktu dan dibatasi hanya menggunakan *system-call* yang telah disediakan oleh ARINC 653.
2. Sebuah *kernel* sistem operasi yang menyediakan *interface* dan perilaku yang telah dispesifikasikan oleh standar ARINC 653. *Kernel* harus menyediakan lingkungan eksekusi standar untuk menjalankan aplikasi.
3. Partisi sistem untuk melakukan fungsi di luar lingkup APEX. Fungsi yang dilakukan partisi sistem meliputi manajemen komunikasi antar *device* perangkat lunak maupun manajemen *fault*.
4. Fungsi spesifik sistem seperti *device driver*, *debug*, dan *test*.

II.1.1 Sistem *Real-Time*

Sistem *real-time* adalah sistem yang kebenaran perhitungannya tidak hanya ditentukan oleh kebenaran hasil perhitungan secara logika, tetapi juga waktu pada saat hasil perhitungan dapat digunakan (K. G. Shin dan Ramanathan, 1994, pp. 6-7). Sistem *real-time* digunakan untuk mendukung aplikasi yang membutuhkan perhitungan *real-time*. Aplikasi yang membutuhkan perhitungan *real-time* umumnya terdiri atas beberapa *task* yang saling terkait. Sebuah *task* dapat berupa periodik dan aperiodik. *Task* yang bersifat periodik harus dikerjakan dalam interval reguler dan memiliki *deadline* yang berarti *task* tersebut harus sudah selesai pada waktu yang telah ditentukan. *Task* yang bersifat aperiodik tidak memiliki

interval reguler, namun tetap memiliki *deadline* yang harus dipenuhi.

Pada umumnya, *task* periodik merupakan *time-critical task* yang harus dijamin dapat diselesaikan sesuai dengan *deadline*-nya dan kegagalan dalam menyelesaikan *task* tersebut dapat mengakibatkan kegagalan sistem secara keseluruhan. Sebagai contoh, pada aplikasi kendali pesawat, sebuah *task* periodik mungkin melakukan pengaturan kekuatan mesin dengan cara menghitung jumlah bahan bakar yang harus diberikan dan memberikannya sejumlah hasil perhitungan pada mesin. Kegagalan sistem operasi menyelesaikan *task* tersebut dalam *deadline* yang telah ditentukan dapat mengakibatkan jatuhnya pesawat dan hilangnya nyawa manusia. Maka, sistem operasi *real-time* pada aplikasi kendali pesawat harus dapat memberikan jaminan bahwa *task* periodik aplikasi tersebut dapat diselesaikan sesuai dengan *deadline*-nya.

Tidak semua *task* harus dikerjakan dalam interval reguler. Beberapa *task* dikerjakan apabila suatu kejadian terjadi. *Task* demikian disebut dengan *task* aperiodik. Sebagai contoh, pintu pesawat dibuka dengan menekan tombol untuk membuka pintu pesawat. *Task* tersebut tidak memiliki *deadline* penyelesaian. Namun, *task* aperiodik juga dapat memiliki *deadline* seperti *task* periodik. Maka, *task* aperiodik demikian harus diselesaikan sebelum *deadline*-nya.

Berdasarkan penjelasan di atas, *deadline* dari sebuah sistem *real-time* dapat diklasifikasikan menjadi *hard*, *firm*, dan *soft*. Sebuah *deadline* dikatakan *hard* apabila konsekuensi kegagalan penyelesaiannya fatal. Sebuah *deadline* dikatakan *firm* dan *soft* apabila konsekuensi kegagalan penyelesaiannya tidak fatal, namun akan menurunkan kualitas layanan yang ditawarkan sistem. Perbedaan antara *soft* dan *firm* adalah apabila hasil komputasi masih dapat digunakan setelah *deadline*, maka *deadline* tersebut merupakan *soft deadline*. Sebaliknya, apabila hasil komputasi tidak dapat digunakan setelah *deadline*, maka *deadline* tersebut merupakan *firm deadline*. Pada umumnya, *task* periodik memiliki *hard deadline*, sedangkan kebanyakan jenis *task* aperiodik yang memiliki *deadline* termasuk dalam kategori *firm deadline*, seperti transaksi pada sistem basis data (Haritsa dkk. 1992, pp. 203-241).

Penentuan *deadline* suatu aplikasi datang dari aplikasi itu sendiri. Sebagai contoh, pada sistem pengaturan kekuatan mesin, kekuatan mesin harus dapat diregulasi setiap 50 ms. *De-*

adline tersebut menjatuhkan *deadline* pada *subtask* pengaturan kekuatan mesin seperti perhitungan jumlah bahan bakar yang dibutuhkan dan pemberian bahan bakar. Sebagai contoh, perhitungan jumlah bahan bakar harus diselesaikan dalam waktu 5 ms dan pemberian bahan bakar harus diselesaikan dalam waktu 20 ms. *Deadline* tersebut akan menjatuhkan *deadline* pada *subtask* masing-masing, dan seterusnya.

Dari gambaran di atas, oleh karena pengaturan kekuatan mesin harus dapat diselesaikan tanpa terkecuali, terlihat bahwa sebuah sistem *real-time* harus dapat diprediksi. Sebuah sistem *real-time* dikatakan dapat diprediksi apabila pada saat desain dapat ditunjukkan bahwa seluruh *deadline task* aplikasi dapat dipenuhi selama beberapa asumsi mengenai sistem tersebut terpenuhi. Sebagai contoh, apabila asumsi mengenai sistem adalah *disk fault* hanya terjadi maksimal sebanyak f selama t detik, maka sistem tersebut dapat ditunjukkan pada saat desain bahwa seluruh *deadline task* aplikasi dapat dipenuhi selama *disk fault* yang terjadi tidak lebih dari f selama t detik. Asumsi tersebut mengakibatkan penentuan sebuah sistem dapat diprediksi ditentukan oleh aplikasi yang dijalankan pada sistem tersebut.

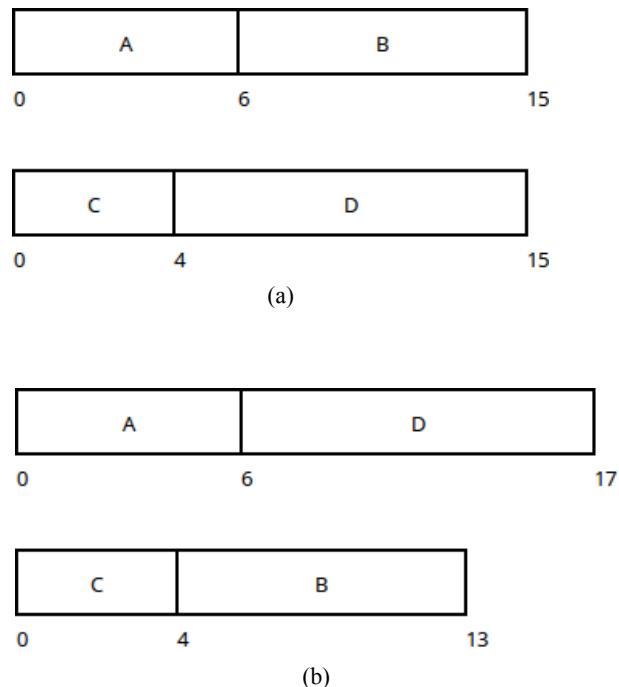
Pada sistem operasi Linux, kapabilitas *real-time* dapat dicapai dengan menggunakan *patch PREEMPT_RT* (The Linux Foundation, 2017). *Patch* tersebut akan meminimalisasi bagian dari *kernel* yang tidak *preemptible* serta menambah kebijakan mengenai prioritas pada masing-masing *process* sehingga *process* dengan prioritas yang lebih tinggi dapat mengambil alih giliran eksesksi dari *process* dengan prioritas yang lebih rendah.

II.1.2 Real-Time Scheduling

Diberikan beberapa *real-time task* dan *resources* yang terdapat pada sistem, *real-time scheduling* adalah proses penentuan kapan dan dimana *task* tersebut akan dikerjakan untuk menjamin seluruh *task* memenuhi *deadline* (K. G. Shin dan Ramanathan, 1994, pp. 8-9). *Scheduler* adalah sebuah proses yang melakukan *scheduling* pada *real-time task* yang diberikan. Dalam buku ini, istilah *real-time scheduling* dan *real-time scheduler* secara berurutan dapat dipertukarkan dengan *scheduling* dan *scheduler*. Sistem operasi akan secara periodik memberikan sejumlah *real-time task* yang tersimpan pada memori pada *scheduler*. Pada aplikasi *non-real-time*, tujuan utama *scheduling* adalah untuk meminimumkan total waktu yang dibutuhkan untuk menyelesaikan semua *task*. Sementara itu, pada aplikasi *real-time*, tujuan

utama *scheduling* adalah untuk memenuhi *deadline task* aplikasi.

Sebagai contoh, Gambar II.1(a) dan (b) memperlihatkan perbedaan antara *scheduling* pada sistem *real-time* dengan *scheduling* pada sistem *non-real-time*. *Task A* membutuhkan waktu $6 \mu\text{s}$, *task B* membutuhkan waktu $9 \mu\text{s}$, *task C* membutuhkan waktu $4 \mu\text{s}$, dan *task D* membutuhkan waktu $11 \mu\text{s}$. Seluruh *task* tersebut akan dijalankan pada sistem dengan dua buah CPU. Apabila sistem merupakan sistem *non-real-time*, maka *scheduling* yang terbaik dapat dicapai dengan pemilihan urutan proses seperti Gambar II.1(a). *Scheduling* seperti pada Gambar II.1(b) juga dapat digunakan, namun tidak optimal. Namun, apabila sistem merupakan sistem *real-time*, dan *task B* dan *D* mempunyai *deadline* $14 \mu\text{s}$ dan $17 \mu\text{s}$, maka satu-satunya *scheduling* yang memenuhi adalah seperti pada Gambar II.1(b).



Gambar II.1. Ilustrasi perbedaan permasalahan *scheduling* pada aplikasi *real-time* dan *non-real-time*. (a) *Scheduling* aplikasi *non-real-time* (b) *Scheduling* aplikasi *real-time*

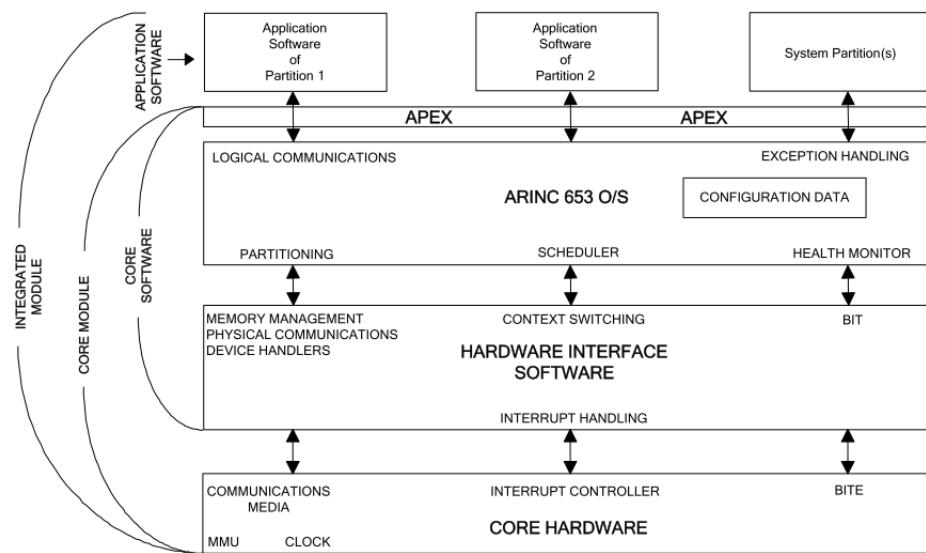
Algoritma untuk *scheduling* dapat diklasifikasikan dalam banyak dimensi. Beberapa algoritma *scheduling* dibuat untuk *task* periodik sedangkan algoritma lain dibuat untuk *task* aperiodik. Demikian juga beberapa algoritma dibuat untuk *preemptible task* sedangkan algoritma lain dibuat untuk *nonpreemptible task*. Algoritma *scheduling* dipengaruhi oleh faktor-faktor

seperti tingkat kekritisan, keterkaitan, *resource*, dan *deadline* aplikasi.

Algoritma *scheduling* juga berbeda tergantung komputer tempat sistem akan dijalankan. Beberapa faktor komputer yang memengaruhi algoritma *scheduling* adalah arsitektur *processor* (*uniprocessor* atau *multiprocessor*), metode komunikasi *shared memory* atau *message-passing*, dan jaringan.

II.1.3 Arsitektur Sistem

Standar ARINC 653 menspesifikasikan arsitektur sistem menggunakan arsitektur IMA. Setiap *core module* dapat memiliki satu atau lebih *processor*. Arsitektur dari *core module* memengaruhi implementasi sistem operasi, tetapi tidak memengaruhi *interface APEX* yang digunakan oleh *aplikasi* pada masing-masing partisi. Aplikasi harus mudah dipindahkan antar *core module* dan antar *processor* dari sebuah *core module* tanpa perlu adanya modifikasi *interface* dengan sistem operasi. Gambar II.2 menunjukkan contoh arsitektur sistem ARINC 653 serta komunikasi antar komponen pada sistem.



Gambar II.2. Contoh arsitektur *module* (Airlines Electronic Engineering Committee, 2012)

II.1.4 Fungsionalitas Sistem

Pada tingkat *core module*, sistem operasi memanajemeni partisi dan komunikasi antar partisi. Pada tingkat partisi, sistem operasi memanajemeni proses aplikasi dalam sebuah partisi dan komunikasi antar proses dalam partisi tersebut. Sistem operasi menyediakan fasilitas pada kedua level tersebut, dengan beberapa perbedaan pada fungsi dan konten tergantung pada lingkup operasinya. Oleh karena itu, definisi dari fasilitas yang diberikan sistem operasi membedakan pada tingkat apa fasilitas tersebut beroperasi.

Pada setiap saat, sebuah *module* berada dalam keadaan inisialisasi, operasional, atau diam. Pada saat dinyalakan, *module* memulai eksekusinya dalam keadaan inisialisasi. Setelah inisialisasi berhasil, *module* masuk ke dalam keadaan operasional. Selama *module* dalam keadaan operasional, sistem operasi memanajemeni partisi, proses aplikasi, dan komunikasi. *Module* akan terus berada pada keadaan operasional sampai dimatikan atau fungsi *health monitoring* pada *module* memerintahkan *module* untuk masuk ke dalam keadaan inisialisasi kembali atau keadaan diam.

Berikut adalah fungsionalitas sistem yang dispesifikasikan oleh standar ARINC 653.

1. Manajemen Partisi.
2. Manajemen Proses.
3. Manajemen Waktu.
4. Manajemen Memori.
5. Komunikasi Antar Partisi.
6. Komunikasi Dalam Partisi.

Penelitian ini hanya akan membahas manajemen partisi dan manajemen waktu yang dispesifikasikan oleh ARINC 653.

II.1.4.1 Manajemen Partisi

Inti dari standar ARINC 653 adalah konsep partisi, yaitu aplikasi yang beroperasi dalam sebuah *module* dipartisi terhadap ruang (partisi memori) dan waktu. Sebuah partisi dapat dikatakan sebagai satu unit aplikasi yang didesain untuk memenuhi batasan-batasan partisi.

Sistem operasi pada dasarnya sudah memiliki cara untuk melakukan partisi *resource* yang dimilikinya. Sistem terpartisi memungkinkan partisi dengan tingkat kekritisan yang berbeda berjalan pada satu *module* yang sama tanpa memengaruhi partisi lain secara *ruang* dan *waktu*. Bagian dari sistem operasi yang bekerja pada tingkat *core module* bertanggung jawab untuk melakukan partisi dan memanajemen partisi dalam *module* tersebut.

Partisi ditentukan urutannya dengan melakukan *scheduling* secara melingkar dan tetap. Untuk dapat melakukan *scheduling* secara melingkar, sistem operasi menyimpan *major time frame* untuk durasi tertentu yang kemudian akan diulang secara terus menerus selama *module* berjalan. Partisi akan terdaftar pada *major time frame* sebagai *partition window* yang memiliki nilai *offset* sesuai dengan urutan yang telah ditentukan. Partisi yang akan dijadwalkan terlebih dahulu akan mendaftarkan *partition window* dengan nilai *offset* yang lebih kecil. Urutan yang didapatkan oleh suatu partisi ditentukan oleh pengaturan urutan yang dibuat pada saat integrasi. Metode ini menjamin bahwa *scheduling* yang dilakukan akan bersifat deterministik sehingga dapat diprediksi.

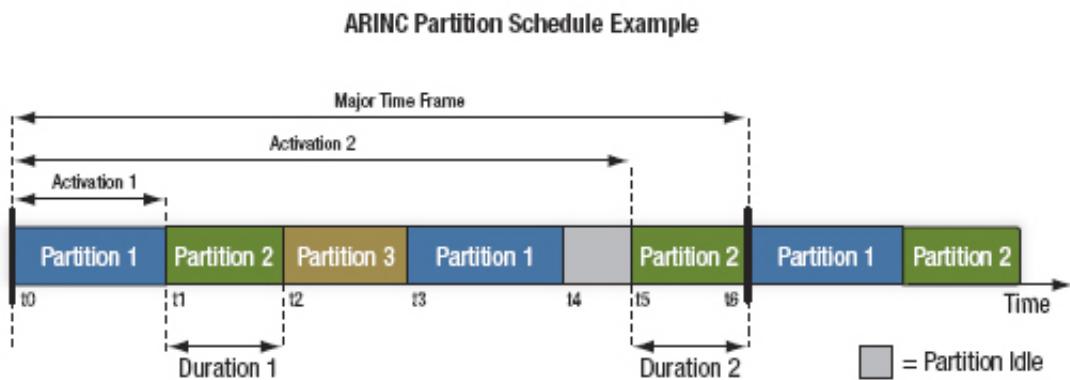
Sebuah *module* dapat memiliki beberapa partisi yang memiliki periode penggerjaan yang berbeda-beda. *Major time frame* didefinisikan sebagai kelipatan dari kelipatan persekutuan terkecil dari seluruh periode penggerjaan partisi pada *module* tersebut. Setiap *major time frame* mempunyai partisi yang sama. Periode penggerjaan masing-masing partisi harus terpenuhi dengan mengatur frekuensi dan ukuran *partition window* dalam *major time frame*.

II.1.4.2 Manajemen Waktu

Manajemen waktu adalah proses penting yang menjadi karakteristik sebuah sistem operasi yang digunakan untuk sistem *real-time*. Waktu harus unik dan tidak bergantung pada eksekusi partisi apa pun dalam *module* pada sistem ARINC 653. Seluruh nilai waktu yang digunakan harus terkait dengan waktu unik tersebut. Sistem operasi menyediakan *time-slicing* untuk *scheduling*, *deadline*, dan *periodicity* dari sebuah partisi. *Scheduler* yang dispesifikasi oleh ARINC 653 merupakan *hierarchical scheduler*, yaitu *scheduler* yang bekerja pada beberapa tingkat. *Scheduling* pada sistem ARINC 653 bekerja pada dua tingkat, yaitu pada tingkat partisi dan tingkat aplikasi. Pada tingkat partisi, *scheduler* bekerja untuk menjadwalkan partisi-partisi pada sistem tersebut. Proses ini disebut sebagai *partition scheduling*.

Pada tingkat aplikasi, *scheduler* bekerja untuk menjadwalkan aplikasi pada masing-masing partisi. Proses ini disebut sebagai *process scheduling*. Penelitian ini akan berfokus pada *partition scheduling*

Ilustrasi *partition scheduling* pada sistem ARINC 653 dapat dilihat pada Gambar II.3.



Gambar II.3. Contoh *partition scheduling* pada sistem ARINC 653 (Kinnan, 2009)

Sebuah kuantum waktu terasosiasi dengan setiap partisi (Kinnan, 2009). Kuantum waktu merepresentasikan waktu maksimal partisi tersebut dapat berjalan. Pada setiap kuantum waktu, partisi akan berjalan untuk menyediakan fungsi yang diharapkan. *Deadline* merupakan batas waktu maksimum ketika hasil perhitungan partisi tersebut dapat digunakan. Batasan tersebut merupakan syarat yang menentukan apakah partisi tersebut berhasil dikerjakan atau tidak. Selama partisi tersebut selesai dikerjakan tanpa melebihi *deadline*, maka *deadline* partisi tersebut dikatakan terpenuhi. Jika tidak, maka *deadline* partisi tersebut tidak terpenuhi dan sistem akan menghasilkan *error*. *Error* yang diakibat oleh *deadline* yang tidak terpenuhi menjadi acuan bahwa sistem tidak berhasil menjamin 100% *real-time*.

II.1.5 *Health Monitoring*

Standar ARINC 653 menspesifikasikan komponen untuk melakukan *health monitoring*, yakni proses untuk mendeteksi kegagalan yang terjadi pada sistem dan melakukan *recovery action* untuk mengatasi kegagalan tersebut. Setelah melakukan pendekteksian dan penanganan kegagalan, komponen tersebut harus dapat melaporkan kegagalan yang terjadi pada sistem.

Health monitoring pada sistem ARINC 653 bekerja pada tiga tingkatan kegagalan yang dapat terjadi pada sistem.

1. Kegagalan pada tingkat proses

Kegagalan pada level ini berakibat pada satu atau lebih proses pada sebuah partisi. Contoh kegagalan pada tingkat proses seperti: kegagalan aplikasi, IRQ pada O/S yang ilegal, dan kegagalan pada proses eksekusi (*stack overflow, page fault, dll*).

2. Kegagalan pada tingkat partisi

Kegagalan pada level ini berakibat pada satu buah partisi. Contoh kegagalan pada tingkat partisi seperti: kegagalan inisialisasi partisi, kegagalan yang terjadi akibat manajemen proses, dan kegagalan yang terjadi karena gagal menangani kegagalan aplikasi.

3. Kegagalan pada tingkat modul

Kegagalan pada level ini berakibat pada seluruh partisi pada sebuah modul. Contoh kegagalan pada tingkat modul seperti: kegagalan pada saat melakukan pergantian partisi dan kegagalan pada saat mengeksekusi fungsi milik sistem.

Respons yang diberikan oleh sistem pada saat kegagalan terdeteksi pada sistem ARINC 653 bergantung pada implementasi sistem. Namun, standar ARINC 653 memberikan cara pendekripsi kegagalan serta mekanisme untuk dapat memberikan respons.

1. Kegagalan pada tingkat proses

Kegagalan pada level ini dideteksi menggunakan proses dengan prioritas tertinggi. Proses berfungsi hanya untuk menangani kegagalan pada aplikasi-aplikasi yang terdapat pada sebuah partisi.

2. Kegagalan pada tingkat partisi dan tingkat modul

Kegagalan pada level ini dideteksi dengan melakukan *error handling* pada sistem operasi dari sistem ARINC 653.

Meski respons yang diberikan bergantung pada jenis kegagalan yang terjadi, standar ARINC 653 memberikan spesifikasi umum respons.

1. Kegagalan pada tingkat proses

Respons yang dapat dilakukan pada tingkat proses adalah:

- (a) abaikan sambil mencatat kegagalan yang terjadi;
 - (b) abaikan sampai n kali sebelum melakukan *recovery action* lain;
 - (c) mengulang proses yang mengalami kegagalan;
 - (d) mengulang partisi yang memiliki proses tersebut;
 - (e) mematikan partisi yang memiliki proses tersebut.
2. Kegagalan pada tingkat partisi
- Respons yang dapat dilakukan pada tingkat proses adalah:
- (a) abaikan;
 - (b) mematikan partisi yang mengalami kegagalan;
 - (c) mengulang partisi yang mengalami kegagalan.
3. Kegagalan pada tingkat modul
- Respons yang dapat dilakukan pada tingkat proses adalah:
- (a) abaikan;
 - (b) mematikan modul yang mengalami kegagalan;
 - (c) mengulang kembali modul yang mengalami kegagalan.

II.2 *Fault-Tolerant Partition Scheduling*

Studi untuk membuat sistem *real-time* menjadi *fault-tolerant* telah banyak dilakukan (Liestman dan Campbell, 1986) (Bertossi dkk. 2006). Liestman dan Campbell (1986) menunjukkan penambahan *deadline* pada setiap *task* akan menghasilkan sistem yang *fault-tolerant*. Meski demikian, penambahan *deadline* hanya menjamin waktu *response* sistem terhadap permintaan aplikasi dan hanya berfungsi sebagai indikator pada saat melakukan verifikasi sistem. *Fault* yang dialami aplikasi tidak hanya berupa waktu *response* saja, sehingga metode tersebut hanya membuat sistem menjadi *fault-tolerant* secara parsial. Untuk dapat mengatasi *fault* yang tidak terduga pada saat verifikasi maupun *fault* pada proses verifikasi itu sendiri, salah satu solusi yang diusulkan adalah dengan menggunakan skema *primary-backup*.

Sebuah studi menunjukkan bahwa peningkatan keandalan *hierarchical scheduler* seperti *scheduler* ARINC 653 dapat dilakukan dengan menggunakan skema *primary-backup* (Hyun dan Kim, 2012). *Primary-backup partition scheduling* merupakan proses *scheduling* yang

terinspirasi oleh metode replikasi data. Pendekatan paling sederhana untuk dapat melakukan *primary-backup partition scheduling* adalah dengan memberikan salinan untuk setiap *task*, dan memberikan salinan tersebut pada *processor* yang berbeda dengan *task* aslinya. Namun, pendekatan tersebut tidak efisien karena memerlukan banyak *processor* tambahan (Oh dan Son, 1994). Bertossi dkk. (2006) menunjukkan cara mengimplementasikan *primary-backup partition scheduling* yang lebih efisien pada sistem *real-time* yang menggunakan algoritma *scheduling Rate- Monotonic-First-Fit* (RMFF) dengan memberikan *task* salinan pada *scheduler* jika dan hanya jika *task* utama mengalami kegagalan. Jin (2013) berhasil membuktikan secara matematis bahwa penggunaan *primary-backup partition scheduling* dapat dilakukan pada sistem *uniprocessor* yang terpartisi dengan melakukan perluasan model *resource*.

II.3 Virtualisasi

Dalam ranah komputasi, virtualisasi adalah tindakan untuk membuat versi virtual dari suatu hal. Virtualisasi pertama kali digunakan sekitar tahun 1960 sebagai metode untuk membagi *resources* sebuah komputer secara logika kepada beberapa aplikasi yang akan menggunakan kannya. Sejak saat itu, arti dari istilah virtualisasi mengalami perluasan makna.

Virtualisasi dapat dilakukan pada berbagai macam hal terkait dengan komputer. Sebagai contoh, sistem operasi modern sudah menggunakan konsep virtualisasi sederhana. Sebuah sistem operasi akan menangani pembagian *resources* untuk diberikan pada aplikasi-aplikasi yang berjalan pada saat yang bersamaan. Dalam kasus ini, sistem operasi memberikan ilusi kepada masing-masing proses bahwa proses tersebut memiliki seluruh *resources* yang terdapat pada komputer tersebut. Namun, yang terjadi adalah sistem operasi membagi *resources* komputer dengan metode sedemikian rupa sehingga pemakaian *resources* tetap konsisten dengan ilusi yang diberikan oleh sistem operasi kepada masing-masing proses.

Sebagai contoh, virtualisasi CPU dilakukan dengan melakukan *preemption* pada proses yang sedang menggunakan CPU sehingga memberikan kesempatan bagi proses lain untuk menggunakan CPU. Sistem operasi melakukan melakukn *context switch* pada proses yang terkena *preemption* sehingga pada saat proses tersebut mendapatkan CPU pada giliran selanjutnya, proses tersebut dapat melanjutkan apa yang dikerjakan sebelumnya. Sementara itu,

virtualisasi memori dilakukan dengan melakukan pemetaan alamat memori virtual dengan kapasitas 2^{32} (2^{64} pada sistem 64-bit) ke alamat memori fisik dengan kapasitas sesuai dengan perangkat keras pada komputer tersebut. Kapasitas yang dapat digunakan oleh sebuah proses pada alamat memori fisik serta cara pemetaannya dilakukan oleh perangkat *memory management unit* pada komputer. *Devices* yang dimiliki oleh sebuah komputer juga divirtualisasikan oleh sistem operasi. Contoh *device* yang divirtualisasikan oleh sistem operasi adalah *storage*.

Salah satu kegunaan metode virtualisasi pada ranah komputasi adalah untuk membuat *virtual machine*. Sebuah *virtual machine* adalah representasi mesin secara logika yang memiliki *resources* layaknya mesin biasa. Umumnya, *virtual machine* digunakan untuk memberikan ilusi sebuah komputer sehingga dari sudut pandang perangkat lunak yang berjalan di atasnya, perangkat lunak tersebut seakan berjalan satu buah komputer fisik tanpa ada pembagian *resources*. Tentunya hal tersebut hanya berlaku untuk sistem operasi atau perangkat lunak yang berjalan langsung di atas perangkat keras. Sebuah komputer dapat direpresentasikan menjadi beberapa *virtual machine* sehingga memungkinkan beberapa sistem operasi beroperasi secara bersamaan pada komputer tersebut.

II.4 *Hypervisor*

Virtualisasi dilakukan menggunakan perangkat lunak virtualisasi yang disebut sebagai *hypervisor*. *Hypervisor* adalah sebuah perangkat lunak yang bertugas untuk membuat dan menjalankan beberapa *virtual machine*. *Hypervisor* berfungsi untuk memanajemeni *resources* fisik yang tersedia untuk digunakan oleh *virtual machine* yang berjalan di bawah *hypervisor* tersebut. Dengan demikian, *hypervisor* memiliki peran sama seperti sistem operasi, hanya saja ketimbang memanajemeni *resources* fisik untuk digunakan oleh proses perangkat lunak, *hypervisor* memanajemeni *resources* untuk sebuah proses yang juga mengatur *resources* yang diberikan pada proses tersebut untuk proses perangkat lunak yang berjalan di bawahnya. Istilah *hypervisor* didapatkan karena *hypervisor* membawahkan *supervisor*, pada kasus ini *supervisor*-nya merupakan sistem operasi.

Terdapat dua tipe *hypervisor*, *hypervisor type-1* dan *hypervisor type-2*. Pada *hypervisor type-1*, *hypervisor* berjalan langsung di atas perangkat keras. Karena *hypervisor type-1* ter-

hubung langsung dengan perangkat keras, maka aplikasi yang berjalan di atas *hypervisor type-1* hanya memerlukan satu lapisan tambahan sebelum dapat dijalankan oleh perangkat keras. Berbeda dengan *hypervisor type-1*, *hypervisor type-2* berjalan di atas sistem operasi yang sudah berjalan. *Hypervisor* berjalan sebagai aplikasi biasa pada sistem operasi, dengan demikian terdapat dua lapisan tambahan sebelum kode aplikasi dapat dijalankan oleh perangkat keras.

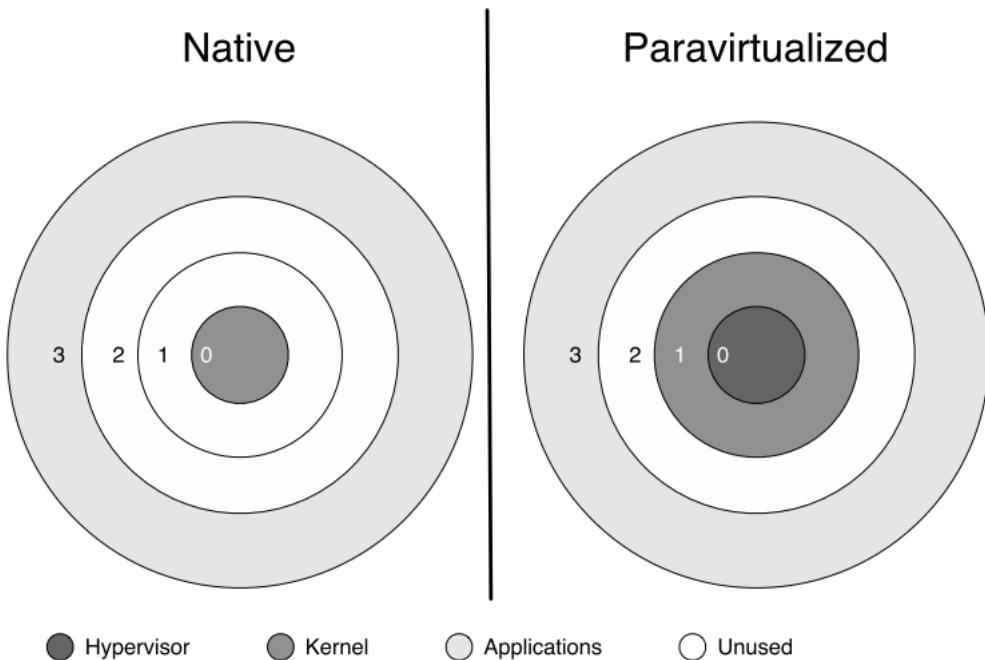
II.5 Xen

Sampai saat ini, sudah terdapat banyak solusi virtualisasi yang tersedia. Karena fokus dari penelitian ini adalah pada bidang avionika, maka virtualisasi yang digunakan harus memiliki performa yang tinggi. Dengan demikian, solusi virtualisasi akan dipilih apabila menggunakan *hypervisor type-1*. Terdapat beberapa solusi virtualisasi dengan *hypervisor type-1* seperti VMWare ESXi, Microsoft Hyper-V, Oracle VM Server dan Xen. Namun, solusi virtualisasi dengan *hypervisor type-1* yang gratis dan *open source* hanya Xen.

Kelebihan Xen sebagai *hypervisor open-source* memudahkan pengembang dalam menuliskan komponen baru yang dibutuhkan. Xen juga memberikan *interface* yang relatif mudah untuk melakukan pengembangan komponen baru. Fokus utama pada buku ini adalah pembuatan *scheduler* baru dan Xen telah menyediakan *interface* untuk membuat *scheduler* baru dengan mudah. Berdasarkan kelebihan-kelebihan tersebut, maka penelitian akan menggunakan Xen sebagai solusi virtualisasi.

Xen merupakan sebuah perangkat lunak untuk melakukan virtualisasi yang bekerja di antara sistem operasi dan perangkat keras. Terdapat tiga komponen utama pada sistem yang menggunakan Xen, yaitu *hypervisor*, *kernel*, dan aplikasi yang bekerja pada *user space*.

Pada Xen, *kernel* sebuah sistem operasi tidak lagi beroperasi pada *ring 0*. *Ring 0* digunakan oleh *hypervisor* yang akan mengatur *resources* yang akan diberikan pada *virtual machine* di bawahnya. *Kernel* sistem operasi dijalankan bergantung pada arsitektur *processor*-nya. Sebagai contoh, pada sistem IA32, *kernel* sistem operasi berjalan pada *ring 1* seperti pada Gambar II.4. Hal tersebut mengakibatkan *kernel* yang berjalan di atas Xen tetap dapat mengakses memori yang teralokasi untuk aplikasi yang berjalan di atas *kernel* tersebut dan *kernel* tetap tidak dapat mengakses maupun diakses oleh aplikasi yang terdapat pada *kernel*

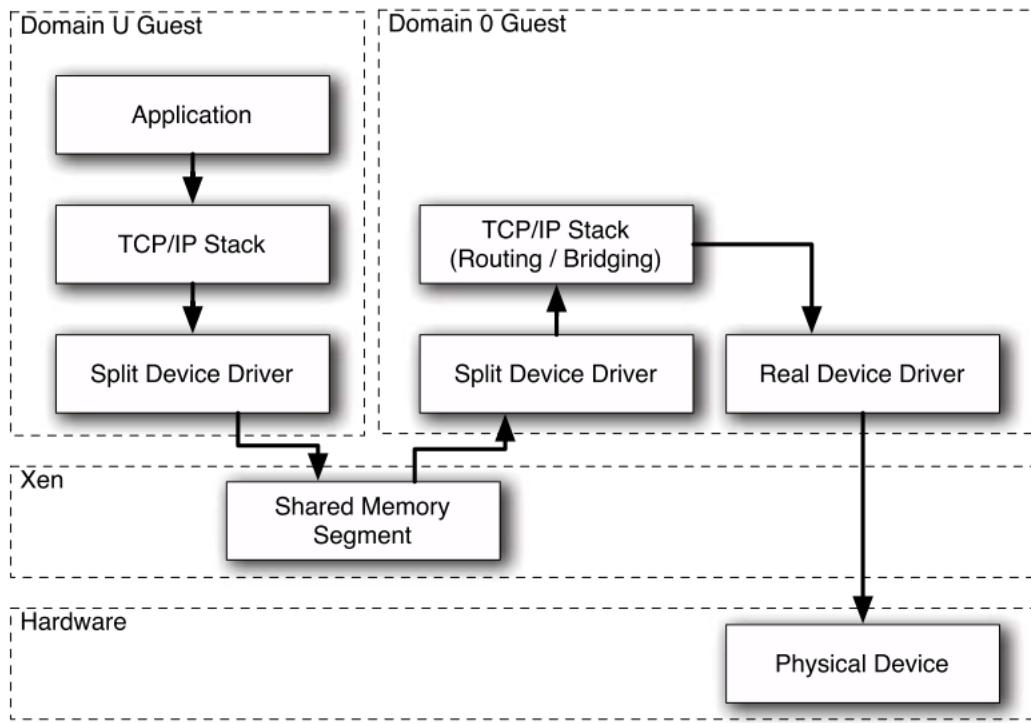


Gambar II.4. Perbedaan penggunaan *ring native* (kiri) dan *paravirtualized* (kanan) (Chisnall, 2014)

lain. *Hypervisor*, yang berjalan pada *ring 0*, terproteksi dari *kernel* yang berjalan pada *ring 1* dan aplikasi yang berjalan pada *ring 3*. Oleh karena *kernel* berjalan pada *ring 1*, *kernel* tidak dapat menjalankan *privileged instructions* seperti biasanya. *Hypervisor* pada Xen menyediakan *hypercall* sebagai pengganti *privileged instructions*. *Hypercall* digunakan oleh *hypervisor* untuk menjembatani permintaan *kernel* pada perangkat lunak. Hal ini dilakukan oleh *hypervisor* dengan cara menangkap *trap* yang dilakukan oleh *kernel*, kemudian meng eksekusi *trap* tersebut melalui *ring 0*. Dengan demikian, *kernel* yang seharusnya berjalan pada *ring 0* tidak perlu mengubah kode programnya dan tetap dapat melakukan *privileged instructions* sebagaimana bila *kernel* berjalan di atas *ring 0*.

Fungsi dari sebuah *hypervisor* adalah untuk menjalankan *virtual machine*. Pada Xen, *hypervisor* menjalankan *virtual machine* yang disebut sebagai *domain*. Terdapat dua jenis *domain* pada Xen, yaitu dom0 dan domU. Dom0 merupakan *domain* yang dapat menggunakan *privileged instructions* untuk memanajemen *devices* yang dijalankan oleh Xen pada saat sistem pertama kali beroperasi. DomU merupakan *domain* yang tidak dapat menggunakan *privileged instructions*. Kewajiban dom0 dalam menangani *devices* adalah dom0 harus dapat menyediakan *devices* untuk semua domU. Oleh karena perangkat keras tidak dapat diakses oleh beberapa sistem operasi sekaligus, maka akses *devices* oleh domU harus mela-

lui dom0. Gambar II.5 mengilustrasikan bagaimana paket jaringan yang dikirim oleh domU melalui dom0. Perbedaan utama pada pengiriman paket jaringan adalah pada *layer TCP/IP*, *layer* tersebut tidak meneruskan langsung ke *network device* tetapi menulis paket pada *shared memory*. Kemudian dom0 akan membaca paket yang terdapat pada *shared memory* untuk dilanjutkan ke *network device* yang sebenarnya. Penggunaan *shared memory* seperti ilustrasi tersebut banyak digunakan pada berbagai komponen yang membentuk Xen.



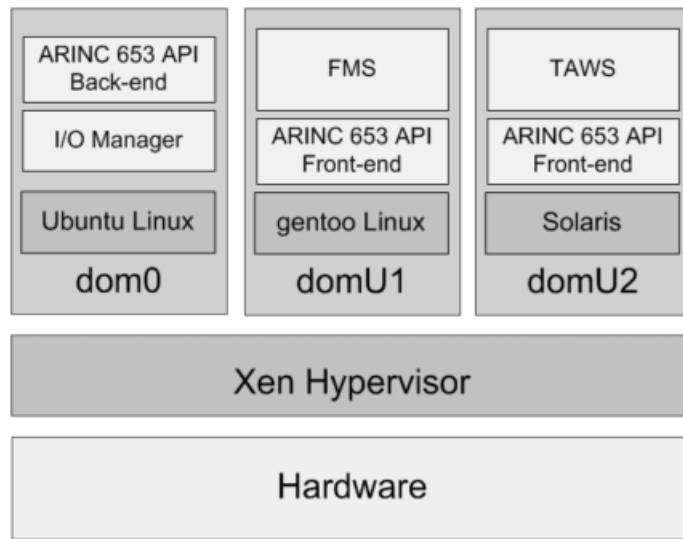
Gambar II.5. Jalur sebuah paket jaringan yang dikirim dari domU (Chisnall, 2014)

II.6 ARLX

ARLX merupakan prototipe sistem ARINC 653 yang dibangun di atas Xen oleh DornerWorks (VanderLeest, 2010). Prototipe dibangun dengan membuat *scheduler* seperti pada spesifikasi standar ARINC 653 pada Xen. Dalam melakukan partisi memori, prototipe tersebut memanfaatkan mekanisme partisi memori menggunakan *memory management unit* yang sudah merupakan mekanisme bawaan pada Xen.

DornerWorks membuka kode sumber *scheduler* ARINC 653 tersebut dan kode sumber ter-

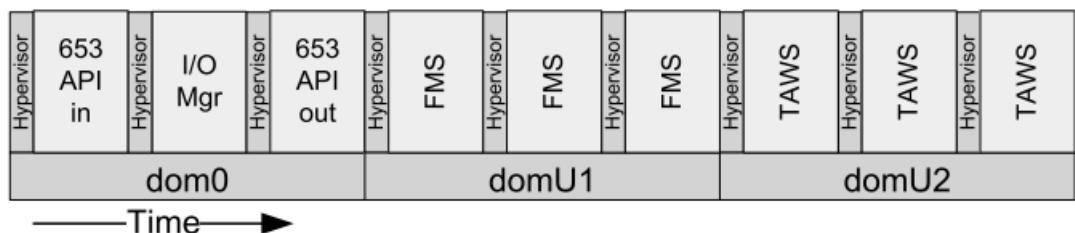
sebut telah digabungkan pada *source tree* milik Xen. Dengan demikian, *scheduler* ARINC 653 yang dikembangkan pada ARLX juga dapat digunakan oleh Xen tanpa memerlukan lapisan eksekusi tambahan. Untuk selanjutnya, ARLX akan merujuk pada Xen yang menggunakan *scheduler* tersebut.



1

Gambar II.6. *Hypervisor* ARLX (VanderLeest, 2010)

Setiap partisi akan direpresentasikan dengan menggunakan *domain* pada Xen. Dom0 pada Xen akan digunakan sebagai partisi tempat APEX dan *device driver* berada, sedangkan partisi untuk aplikasi avionik akan direpresentasikan menggunakan domU seperti pada Gambar II.6.



Gambar II.7. *Schedule* partisi pada ARLX (VanderLeest, 2010)

Partition scheduling dilakukan seperti yang telah dispesifikasikan pada standar ARINC 653. Layanan sebuah partisi akan didaftarkan dengan menggunakan *hypercall* seperti pada Xen. Partisi akan dipilih sesuai dengan urutan pada daftar jadwal yang diberikan pada saat melakukan *hypercall*. Ilustrasi penjadwalan partisi yang dilakukan oleh *partition scheduler* milik ARLX dapat dilihat pada Gambar II.7. Pada ilustrasi tersebut, ARLX akan menyediakan dua buah layanan masing-masing pada sebuah partisi tersendiri. Pada partisi pertama (domU1), ARLX akan menyediakan layanan bernama FMS. Pada partisi kedua (domU2), ARLX akan menyediakan layanan bernama TAWS.

Secara umum, algoritma tersebut sama seperti algoritma *scheduling* yang telah dipaparkan pada Subbab II.1.4.2. Pada saat *major time frame* dimulai, dom0 akan berjalan dan menjalankan fungsi-fungsi yang seharusnya dilakukan oleh dom0, seperti manajemen I/O, menyediakan API ARINC 653 untuk partisi-partisi pada sistem, dan lain-lain. Setelah kuantum waktu yang dimiliki oleh dom0 habis, domU1 akan berjalan dan menyediakan layanan FMS. Setelah kuantum waktu yang dimiliki oleh domU1 habis, domU2 akan berjalan dan menyediakan layanan TAWS. Hal ini akan dilakukan secara terus menerus sampai sistem dimatikan. Slot *hypervisor* pada selang *major time frame* digunakan oleh *hypervisor* untuk menentukan partisi mana yang akan mendapatkan waktu CPU.

Spesifikasi ARINC 653 berlaku apabila sistem operasi yang digunakan merupakan sistem operasi *real-time*. Pada dasarnya, Xen tidak dibangun sebagai *hypervisor* dengan kapabilitas *real-time*. Terdapat sebuah proyek yang ingin membangun kapabilitas *real-time* pada Xen. Namun, proyek tersebut tidak mendukung *scheduler* ARINC 653 yang terdapat pada ARLX. Dengan demikian, penelitian ini akan tetap menggunakan *scheduler* ARINC 653 pada ARLX meskipun Xen tidak memiliki kapabilitas *real-time*.

BAB III

ANALISIS PERMASALAHAN DAN RANCANGAN SOLUSI

III.1 Analisis Permasalahan

Standar ARINC 653 berfungsi sebagai panduan pengembangan sistem operasi *real-time* pada *safety-critical system*. Banyak faktor yang menentukan apakah sebuah sistem dapat dikategorikan sebagai *safety-critical system*, salah satunya adalah *fault-tolerancy*. Pada *safety-critical system*, *fault-tolerant* adalah properti yang sangat dicari. Sistem yang *fault-tolerant* akan memberikan jaminan bahwa sistem akan terus bekerja sebagaimana seharusnya meskipun terjadi kegagalan selama sistem beroperasi.

Fokus dari standar ARINC 653 adalah menjamin sistem operasi melakukan partisi lingkungan eksekusi agar aplikasi yang berjalan pada sebuah lingkungan eksekusi tidak mengganggu aplikasi pada lingkungan eksekusi lainnya. Meski standar tersebut menjamin kegagalan yang terjadi pada sebuah aplikasi tidak akan mengganggu aplikasi lain yang berada pada lingkungan eksekusi berbeda (*fault containment*), kegagalan yang terjadi pada sebuah aplikasi tetap akan terjadi. Standar ARINC 653 mendefinisikan mekanisme untuk mendeteksi dan memberikan tanggapan apabila terjadi kegagalan. Namun, standar tersebut tidak mendefinisikan tanggapan apa yang harus dilakukan untuk menjamin aplikasi yang ditangani tidak akan mengalami kegagalan yang sama dalam waktu dekat. Meskipun aplikasi avionik pada umumnya akan melalui proses verifikasi untuk menjamin tidak akan terjadi kegagalan, proses verifikasi tersebut mungkin sangat lama dan/atau tidak berhasil melakukan pengujian pada kasus khusus yang mengakibatkan kegagalan. Karena itu, mekanisme penanganan kegagalan tidak dapat diabaikan begitu saja.

Seperti yang telah dipaparkan pada Subbab II.1.5, sistem ARINC 653 mengisolasi kegagalan yang terjadi pada sebuah partisi sehingga kegagalan tersebut tidak akan memberikan efek apapun pada partisi lain. Kemudian, partisi yang mengalami kegagalan akan terdeteksi oleh *health monitor* pada sistem. *Health monitor* akan melakukan *recovery action* sesuai dengan jenis kegagalan yang terjadi. Meski demikian, *recovery action* yang dispesifikasikan akan

mengakibatkan partisi tidak berjalan untuk rentang waktu tertentu dan tidak dapat mengatasi permasalahan apabila penyebab kegagalan persisten pada partisi tersebut sehingga partisi tetap mengalami kegagalan.

Permasalahan yang terjadi apabila terdapat partisi yang tidak berjalan atau mengalami kegagalan adalah layanan yang disediakan akan berhenti bekerja dan/atau memberikan hasil perhitungan yang salah. Dalam avionika, hal ini dapat mengakibatkan kesalahan pada sistem kendali sehingga meningkatkan kemungkinan terjadinya bahaya pada manusia dan lingkungan.

III.2 Rancangan Solusi

Solusi dari permasalahan yang dipaparkan pada Subbab III.1 adalah dengan membuat sistem menjadi lebih andal. Untuk membuat sistem ARINC 653 menjadi lebih andal, salah satu solusi yang telah dipelajari sebelumnya adalah dengan menggunakan *primary-backup scheduling* pada *hierarchical scheduler* (Liestman dan Campbell, 1986) (Bertossi dkk. 2006). Penggunaan skema *primary-backup* dapat menjamin apabila partisi mengalami kegagalan, maka akan terdapat partisi *backup* yang dapat menyediakan layanan tersebut. Dengan demikian, layanan tidak akan mengalami *fault* dalam waktu panjang. Namun, meski sudah banyak studi mengenai *primary-backup scheduling* pada *hierarchical scheduler*, belum ada studi penggunaan *primary-backup scheduling* spesifik pada sistem ARINC 653 dan belum ada implementasi sistem ARINC 653 dengan menggunakan *primary-backup scheduling* yang tersedia secara gratis.

Primary-backup scheduling hanya akan diterapkan pada *partition scheduler*. Hal ini dikarenakan *process scheduler* bergantung pada sistem yang digunakan oleh partisi. Agar solusi dapat bekerja untuk kasus umum, solusi yang ditawarkan harus merupakan solusi yang tidak bergantung pada *process scheduler*, dengan asumsi mekanisme *process scheduler* yang digunakan tetap sesuai spesifikasi pada standar ARINC 653.

Dengan menggunakan *primary-backup*, layanan yang diberikan oleh partisi yang mengalami kegagalan akan tetap dapat berjalan dengan memberikan kuantum waktu CPU pada partisi *backup*. Skema umum *primary-backup* akan membuat *backup* tersebut identik dengan partisi *primary*. Meski skema tersebut dapat membuat sistem terlihat tidak pernah

mengalami kegagalan sama sekali, partisi yang identik berarti kegagalan yang terjadi pada partisi *primary* cepat atau lambat akan terjadi pada partisi *backup*.

Agar sistem yang menggunakan *primary-backup partition scheduling* dapat berjalan secara terus menerus, partisi *backup* tidak harus merupakan partisi yang identik dengan partisi *primary*. Partisi *backup* cukup memberikan layanan yang identik dengan layanan yang diberikan oleh partisi *primary*. Layanan tersebut dapat berupa aplikasi dengan versi yang sudah terbukti lebih stabil, meski tidak seoptimal layanan dengan versi baru. Solusi ini mencakup kemungkinan yang lebih umum ketimbang partisi *backup* yang identik.

Implementasi dan pengujian *primary-backup partition scheduling* akan dilakukan pada ARLX, yaitu prototipe sistem yang bersesuaian dengan spesifikasi ARINC 653 pada Xen. Hal ini dikarenakan prototipe tersebut tersedia gratis dan *open-source*, sehingga penelitian maupun pengembangan terkait dengan sistem ARINC 653 tidak memerlukan biaya banyak dan mudah untuk mengembangkan komponen baru.

III.2.1 Implementasi *Primary-Backup Partition Scheduling* pada ARLX

Pada ARLX, pembuatan sebuah partisi membutuhkan waktu yang cukup lama. Agar partisi *backup* dapat langsung menggantikan partisi *primary* ketika mengalami mengalami kegagalan, maka partisi *backup* sudah harus dibuat pada tahap inisialisasi atau sistem.

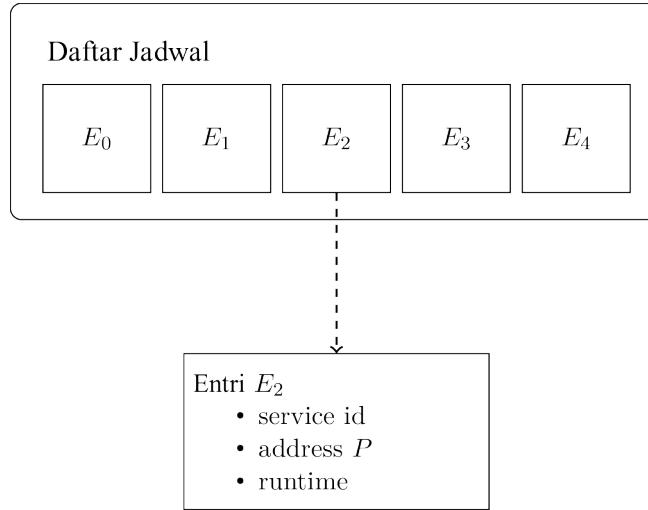
Scheduler membutuhkan daftar jadwal layanan-layanan yang akan disediakan pada setiap *major time frame*. *Scheduler* akan mendapatkan daftar jadwal layanan melalui *hypercall*. Kemudian, *scheduler* akan memilih partisi yang akan diberi kuantum waktu CPU. Partisi hanya akan diberikan kuantum waktu CPU jika partisi tersebut tidak mengalami kegagalan.

Selain itu, *scheduler* juga perlu mengetahui kapan sebuah partisi *primary* dikategorikan mengalami kegagalan. Oleh karena itu, perlu ada mekanisme untuk mendeteksi *fault*, dan memberikan informasi tersebut kepada *scheduler*.

III.2.2 Daftar Jadwal

Partition scheduler akan bekerja pada sebuah daftar jadwal yang terdapat pada konteks global milik *scheduler*. Dengan demikian, informasi tersebut dapat diakses kapan pun apabila

dibutuhkan oleh *scheduler*. Daftar jadwal dapat diberikan oleh *user space* melalui *hypercall* yang disediakan. Setiap perubahan yang terjadi pada *daftar jadwal* akan mengakibatkan seluruh proses penjadwalan yang sedang dilakukan oleh *scheduler* diulang kembali. Dengan demikian, daftar jadwal sebaiknya hanya dilakukan pada saat pengaturan awal dan tidak dilakukan pada saat sistem sedang bekerja pada lingkungan produksi.



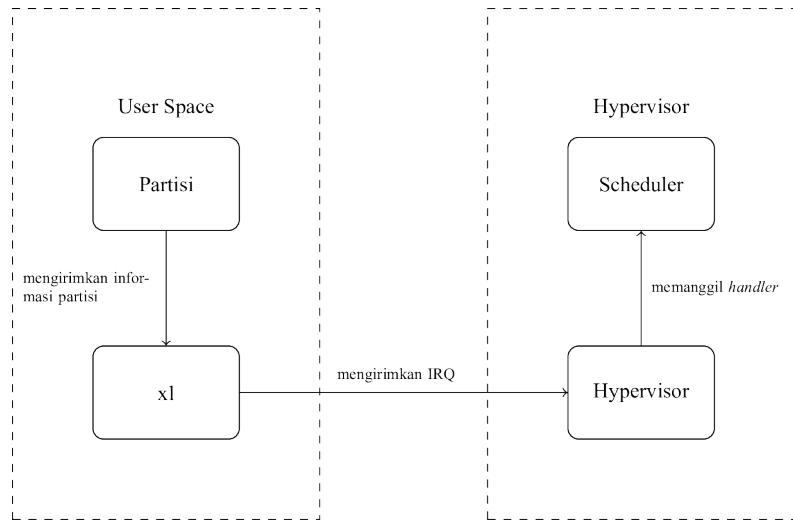
Gambar III.1. Daftar jadwal layanan

Daftar jadwal berupa sebuah struktur data *array*. Masing-masing elemen pada *array* tersebut disebut sebagai sebuah entri. Setiap entri akan memiliki informasi berupa batasan *real-time* layanan tersebut beserta penunjuk yang mengarah pada daftar partisi-partisi yang akan menyediakan layanan tersebut apabila diberikan kuantum waktu CPU. Dengan demikian, pada saat *scheduler* sedang memproses sebuah layanan, *scheduler* dapat memilih partisi yang akan dijalankan dengan cepat. Setiap entri hanya akan memiliki sebuah daftar partisi yang dapat menyediakan layanan, sehingga entri dapat dikatakan menunjuk pada daftar penyedia layanan. Ilustrasi daftar jadwal dapat dilihat pada Gambar III.1.

III.2.3 Informasi Partisi

Agar skema *primary-backup* dapat dilakukan, *scheduler* harus mengetahui beberapa informasi terkait keadaan partisi agar dapat menentukan partisi mana yang harus mendapatkan

kuantum waktu CPU. *Scheduler* harus mengetahui apakah partisi tersebut mengalami kegagalan atau tidak. Apabila partisi sedang mengalami kegagalan, maka partisi tidak boleh mendapatkan jatah kuantum waktu CPU. Selain itu, *scheduler* juga harus mengetahui layanan apa saja yang sudah dikerjakan sebelumnya pada *major time frame* saat ini. Dengan demikian, partisi yang memiliki layanan yang tidak bersifat *idempotent* tidak akan mendapatkan kuantum waktu CPU lebih dari sekali.

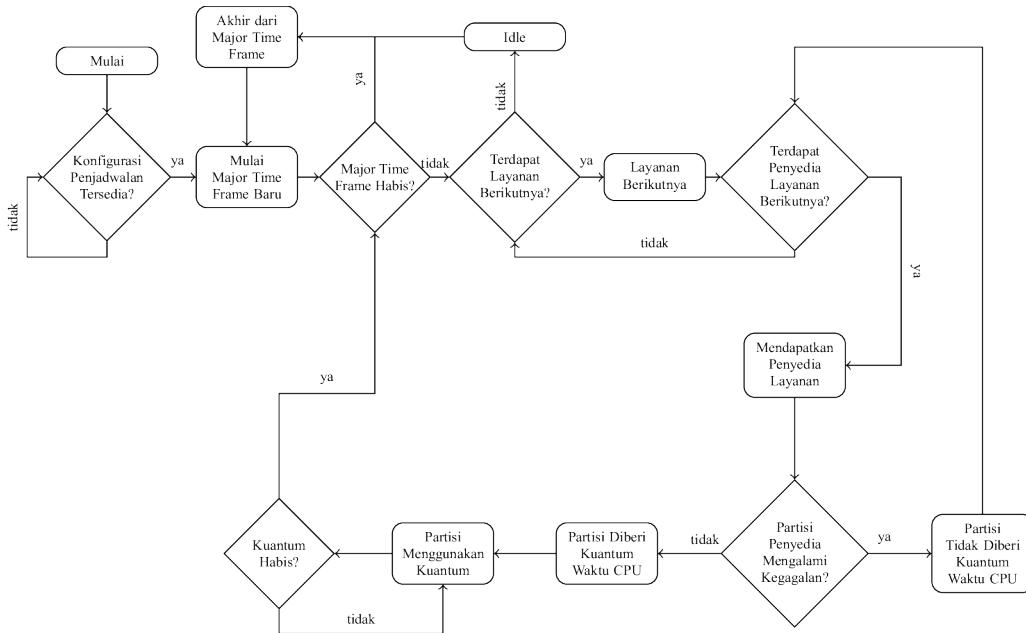


Gambar III.2. Alur informasi sistem

Informasi mengenai keadaan partisi dapat disimpan dengan menggunakan *boolean*. Informasi mengenai identifikasi layanan yang terdapat pada partisi dapat disimpulkan berdasarkan daftar jadwal yang didapat melalui *hypercall* seperti yang telah dipaparkan pada Subbab III.2.2. Apabila sebuah partisi P berada pada rentang waktu milik layanan A , maka partisi P diasumsikan menyediakan layanan A . Alur pengiriman informasi dapat dilihat pada Gambar III.2.

III.2.4 Algoritma *Scheduling*

Algoritma *scheduling* pada sistem *real-time* harus deterministik agar administrator sistem dapat mengatur konfigurasi proses-proses *real-time* sehingga memenuhi batasan-batasan masing-masing proses. Agar algoritma *scheduling* tetap deterministik, algoritma tersebut



Gambar III.3. Diagram Flowchart Algoritma *Scheduling*

harus tidak bergantung pada sumber eksternal. Dengan demikian, algoritma *primary-backup scheduling* yang dibuat harus secara penuh bergantung kepada informasi masing-masing partisi. *Scheduler* akan memberikan kuantum waktu CPU kepada partisi yang tidak sedang mengalami kegagalan dan layanan tersebut belum pernah disediakan. Sebuah layanan dikatakan belum pernah disediakan jika dan hanya jika dan hanya jika tidak ada partisi dengan layanan yang sama dengan layanan partisi tersebut yang sudah diberikan kuantum waktu CPU pada suatu *major time frame*.

Algoritma yang akan digunakan untuk melakukan *scheduling* digambarkan seperti pada Gambar III.3. Scheduler akan menunggu daftar layanan yang akan disediakan pada sebuah *major time frame*. Apabila *scheduler* telah mendapatkan daftar layanan tersebut, *scheduler* akan membuat *major time frame* baru. Setelah membuat *major time frame* baru, *scheduler* akan memilih layanan yang akan disediakan.

Pada saat partisi akan menyediakan sebuah layanan S , maka *scheduler* akan mendapatkan entri E_S dari daftar jadwal. Kemudian *scheduler* akan memilih salah satu partisi yang terdapat pada daftar penyedia layanan P yang ditunjuk oleh entri E_S . Setiap partisi memiliki informasi partisi seperti yang telah dijelaskan pada Subbab III.2.3. Dengan demikian

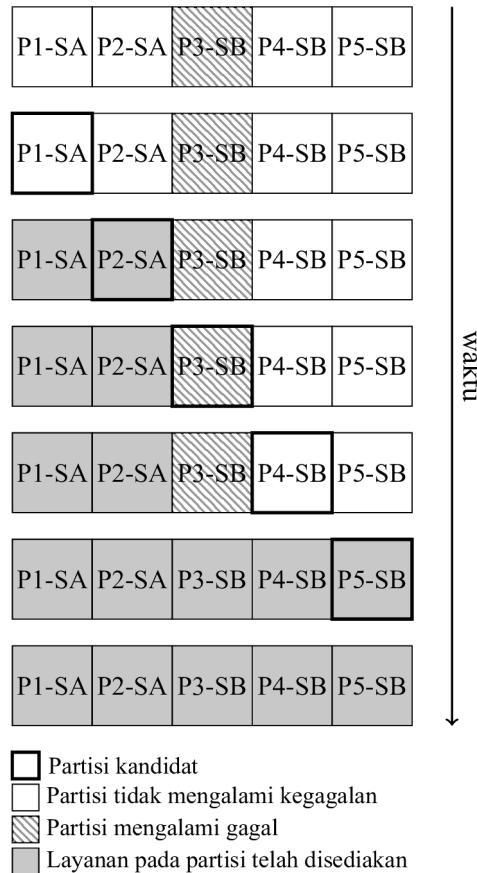
scheduler dapat menentukan apakah partisi P_i siap untuk diberi kuantum waktu CPU atau tidak.

Jika partisi P_1 tidak siap untuk diberi kuantum waktu CPU, maka *scheduler* akan mengecek partisi P_2 , jika partisi P_2 tidak siap untuk diberi kuantum waktu CPU, *scheduler* akan mengcek partisi P_3 , dan seterusnya. Apabila tidak terdapat partisi pada P yang tidak mengalami kegagalan, maka *scheduler* akan menunggu sampai waktu pada entri E_S habis dan akan mengambil entri berikutnya. Apabila terdapat partisi pada daftar penyedia layanan P yang tidak sedang mengalami kegagalan, maka partisi tersebut akan diberi kuantum waktu CPU. Pada saat waktu yang diberikan sudah habis, maka *scheduler* akan mengambil entri berikutnya.

Scheduler akan menghabiskan entri layanan dan menunggu sampai periode *major frame* tersebut habis. Kemudian, *scheduler* akan membuat *major time frame* baru dan mengulangi proses yang sama. Algoritma ini akan dilakukan secara terus menerus selama sistem berjalan.

Illustrasi bagaimana algoritma *primary-backup scheduling* pada *scheduler* ARINC 653 bekerja dapat dilihat pada Gambar III.4. Pada ilustrasi, partisi diberi label dengan format $PX - SY$, yang berarti partisi X akan menyediakan layanan Y jika diberikan kuantum waktu CPU oleh *scheduler*. Pada skenario tersebut, *scheduler* akan bekerja sebagai berikut.

1. Pada partisi $P1$, partisi tidak sedang mengalami kegagalan dan layanan A belum pernah disediakan, sehingga partisi diberikan waktu CPU oleh *scheduler*.
2. Pada partisi $P2$, partisi tidak diberikan kuantum waktu CPU karena pada saat partisi $P1$ sudah menghabiskan kuantumnya, layanan A sudah pernah disediakan.
3. Pada partisi $P3$, layanan pada partisi tersebut belum pernah disediakan, namun partisi sedang mengalami kegagalan sehingga partisi tidak diberikan waktu CPU.
4. Partisi $P4$ tidak sedang mengalami kegagalan dan karena partisi $P3$ tidak diberikan kuantum waktu CPU, layanan B belum pernah disediakan. Karena semua kondisi memenuhi, maka partisi $P4$ diberikan kuantum waktu CPU oleh *scheduler*.
5. Partisi $P5$ tidak diberikan kuantum waktu CPU karena layanan B sudah pernah disediakan.



Gambar III.4. Contoh *primary-backup scheduling*

Pada akhir dari *major time frame*, algoritma ini menjamin setiap layanan pernah disediakan apabila terdapat setidaknya satu partisi yang memiliki layanan tersebut yang tidak sedang mengalami kegagalan. Algoritma ini akan berulang secara terus menerus tiap akhir *major time frame*.

III.2.5 Mekanisme Penanganan Kegagalan

Penanganan kegagalan dengan menggunakan *primary-backup scheduler* dilakukan dengan mengubah nilai keadaan sebuah partisi dengan menggunakan *hypercall*. *Hypercall* tersebut tersedia pada dom0. *Hypercall* tersebut tidak dapat diakses oleh partisi yang terdapat pada sistem. Partisi pada sistem dapat mengatasi permasalahan tersebut dengan menggunakan API yang tersedia pada dom0. Namun, pada penelitian ini, API tersebut tidak akan diimplementasikan dan perubahan nilai keadaan akan dilakukan secara manual.

Sama seperti pada spesifikasi ARINC 653, metode pendekripsi kegagalan dapat dilakukan

seperti yang telah dipaparkan pada Subbab II.1.5. Namun, pada saat menggunakan *primary-backup scheduler*, *recovery action* yang dilakukan oleh sistem operasi adalah mengubah nilai keadaan sebuah partisi.

III.3 Rancangan Pengujian

Untuk menjamin *scheduler* yang dibahas pada Subbab III.2 dapat bekerja sebagaimana seharusnya, perlu dilakukan pengujian terkait fungsionalitas *scheduler*. Berikut adalah fungsionalitas sistem yang akan diukur pada saat menggunakan *primary-backup partition scheduler*.

1. Keandalan sistem
2. *Latency* layanan pada sistem

Pengukuran kedua fungsionalitas tersebut dilakukan guna mencapai tujuan yang telah dipaparkan pada Subbab I.3. Pengukuran keandalan sistem diperlukan untuk mengkuantifikasi peningkatan keandalan yang diakibatkan oleh penggunaan *primary-backup partition scheduler*. Pengukuran *latency* layanan pada sistem diperlukan untuk melihat efek penggunaan *primary-backup partition scheduler* pada kinerja sistem ARINC 653. Kedua jenis pengujian tersebut akan mengumpulkan data dengan melakukan *timing*, yaitu pengukuran waktu yang terlewat dari satu titik tertentu pada masa lampau. Data tersebut kemudian akan dianalisa untuk melihat perilaku dan kinerja dari sistem. Berdasarkan data tersebut, kinerja sistem dapat dianggap baik apabila sistem dapat dikategorikan sebagai sistem *real-time* seperti yang telah dipaparkan pada Subbab II.1.1.

Perangkat lunak akan dibangun dengan menggunakan bahasa pemrograman C dan berjalan pada *user space*. Dengan demikian, hasil pengukuran yang didapat memuat *delay* yang diakibatkan oleh pemindahan data melalui *system call* ketika akan meminta nilai waktu yang dicatat oleh sistem operasi. Namun, hal tersebut diperlukan karena beberapa alasan yang akan dipaparkan pada Subbab III.3.1 dan Subbab III.3.2.

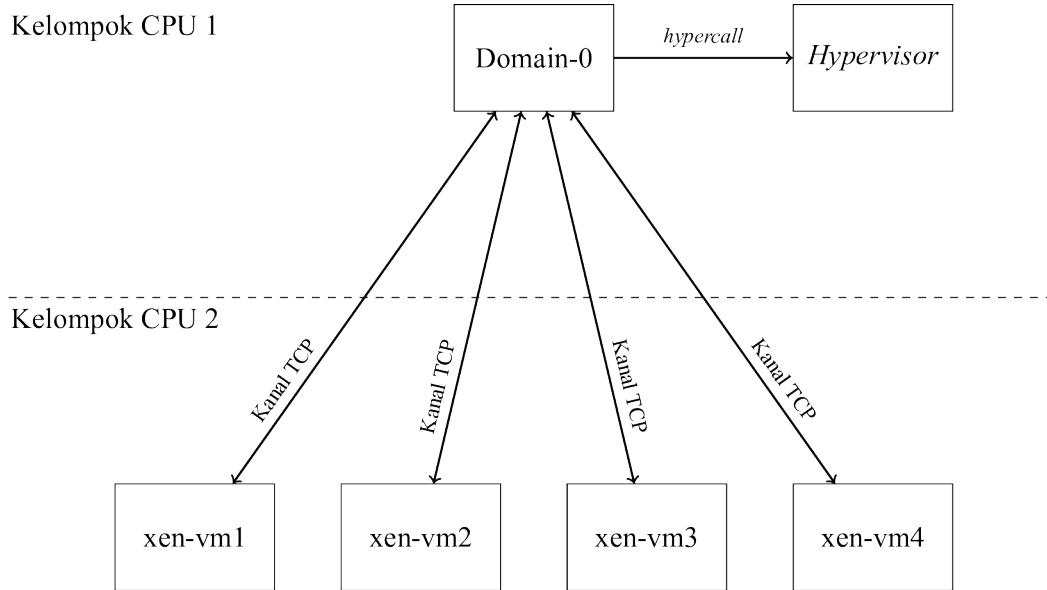
III.3.1 Pengujian Keandalan

Untuk mengukur peningkatan keandalan sistem akibat penggunaan *primary-backup partition scheduler*, sistem akan menjalani pengujian keandalan pada saat menggunakan *scheduler* tersebut. Tujuan utama dari pengujian keandalan adalah untuk mengukur *mean time to failure* sistem setelah menggunakan *scheduler* tersebut. Untuk mempermudah pengukuran, asumsikan bahwa sistem akan selalu mengalami kegagalan apabila terdapat sebuah layanan pada sistem tersebut yang mengalami kegagalan dan layanan-layanan merupakan komponen paling tidak andal pada sistem. Dengan demikian, *mean time to failure* sistem sama dengan minimum dari *mean time to failure* seluruh layanan yang terdapat pada sistem tersebut. Keandalan sistem dikatakan meningkat saat menggunakan *primary-backup scheduler* apabila *mean time to failure* sistem tersebut menjadi lebih lama. Selain itu, hasil pengujian dapat digunakan untuk menentukan apakah algoritma bekerja seperti yang seharusnya.

Pengujian keandalan dilakukan dengan melakukan pengecekan apakah setiap partisi dapat mengirimkan *heartbeat* dengan baik. Sebuah partisi dikatakan dapat mengirim *heartbeat* apabila partisi mengirimkan *heartbeat* jika dan hanya jika partisi tersebut memenuhi kondisi untuk mendapatkan kuantum waktu CPU. Cara termudah untuk melakukan pengujian tersebut adalah dengan menggunakan kerangka uji dengan arsitektur *server-client* seperti pada Gambar III.5.

Setiap partisi berlaku sebagai *client* pada arsitektur tersebut. *Server* dapat berupa komputer apa pun yang terhubung dengan seluruh partisi menggunakan kanal TCP. TCP dipilih karena kerangka uji didesain agar dapat digunakan antar komputer sehingga memerlukan komunikasi antara komputer melalui jaringan. Selain itu, protokol komunikasi dipilih menggunakan TCP karena protokol TCP menjamin paket yang dikirim akan sampai pada tujuan. Untuk mempermudah pengujian, mesin yang digunakan untuk *server* adalah dom0 pada sistem. Pada awalnya, dom0 akan menggunakan *hypercall* untuk mengubah penjadwalan partisi-partisi pada sistem sesuai dengan skenario yang ditentukan. Kemudian, setiap partisi mengirimkan *heartbeat* dengan selang waktu tertentu kepada *server*. Kemudian, *server* akan mencatat setiap layanan sudah pernah disediakan pada tiap akhir *major time frame*. Pada saat pengujian tersebut dijalankan, dom0 akan memanggil *hypervisor* secara acak untuk mengubah keadaan salah satu partisi. Ilustrasi langkah-langkah pengujian dapat dilihat

pada Gambar III.6.

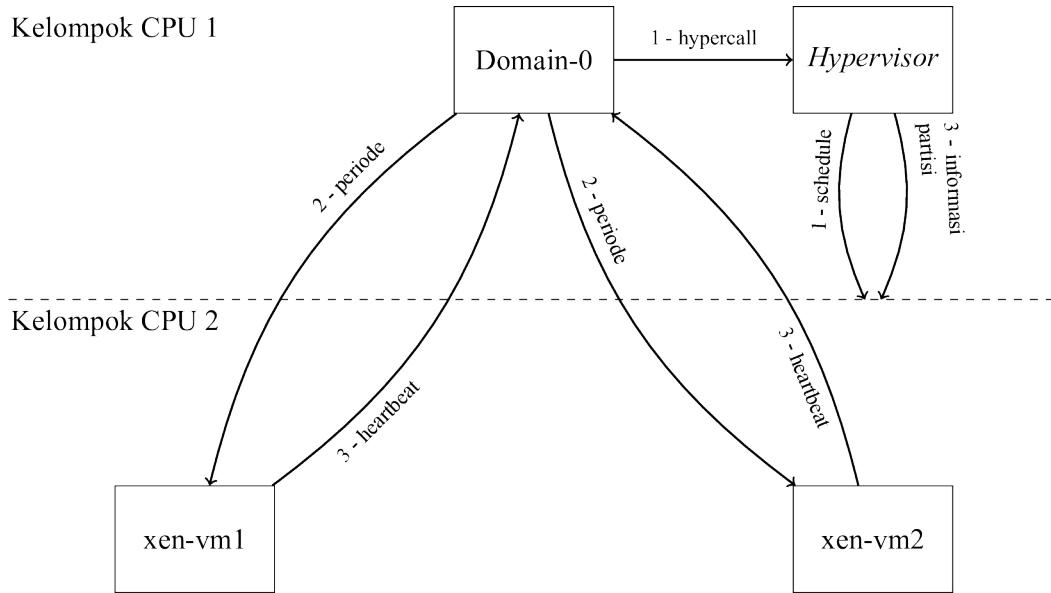


Gambar III.5. Arsitektur sistem pengujian keandalan layanan

Agar pengujian keandalan tersebut dapat bekerja sebagaimana seharusnya, maka batasan-batasan berikut harus terpenuhi pada saat melakukan pengujian.

1. Selang waktu harus dipilih sedemikian sehingga setiap partisi akan memberikan layanan tersebut apabila mendapatkan kuantum waktu CPU.
2. *Server* dan *client* tidak boleh menggunakan CPU yang sama. Hal ini menjamin agar *client* tidak mengganggu waktu eksekusi *server* sehingga *heartbeat* dari *client* dapat langsung diproses oleh *server*.
3. Waktu yang digunakan untuk melakukan pengukuran pengujian adalah waktu pada *server*. Karena waktu yang dipakai antara *server* dan *client* mungkin berbeda, perlu adanya satu sumber kebenaran waktu pada saat pengujian untuk menghasilkan pengukuran yang dapat terkuantifikasi.

Perlu diperhatikan bahwa penggunaan kanal TCP akan menambah *delay* pada waktu hasil pengukuran. Ditambah dengan *delay* yang diakibatkan oleh *system call*, maka hasil yang didapat memiliki resolusi waktu yang rendah. *Delay* tersebut mungkin dapat diperkecil



Gambar III.6. Komunikasi pada sistem pengujian keandalan layanan sesuai dengan urutannya

apabila komunikasi dilakukan menggunakan UDP. Namun, UDP tidak menjamin paket akan sampai pada tujuan, sehingga UDP bukanlah metode komunikasi yang andal. Meski demikian, tujuan utama pengujian keandalan dilakukan adalah untuk melihat keandalan sistem dan bukan untuk melihat kinerja sistem. Apabila terdapat paket yang hilang, maka hasil perhitungan keandalan sistem akan berbeda secara signifikan. Dengan demikian, kanal TCP tetap dipilih sebagai metode komunikasi antara *client* dan *server* pada pengujian keandalan.

Pengujian keandalan juga dapat digunakan untuk menghitung *latency* dari masing-masing layanan. Namun, seperti yang telah dijelaskan sebelumnya, akan terdapat *delay* pada waktu hasil pengukuran dikarenakan I/O. Meski demikian, *latency* yang didapat akan tetap bermafaat untuk melihat kinerja sistem apabila layanan yang berjalan pada partisi akan menggunakan I/O (*I/O-bound*).

III.3.2 Pengujian *Latency* Layanan

Pengujian *latency* layanan dilakukan dengan melakukan *periodic task* pada partisi-partisi yang berada pada sistem. Selisih waktu antara waktu yang seharusnya dengan waktu ketika *task* tersebut dapat diselesaikan merupakan *latency* yang akan diukur pada saat pengujian. Pengujian akan dilakukan menggunakan POSIX *timer*, yaitu *system interface* yang digu-

nakan untuk mendapatkan notifikasi pada waktu yang akan datang dan juga secara berkala (Institute of Electrical and Electronics Engineers, 2016). Pada saat *task* tersebut mendapatkan waktu CPU, *task* tersebut akan langsung mencatat waktu pada saat itu. Pada saat program pengujian selesai dijalankan, maka program tersebut akan menuliskan catatannya kedalam sebuah *file*.

Pengujian akan dilakukan pada *user space* dikarenakan aplikasi yang akan dibangun pada sistem ini akan berupa aplikasi yang berjalan pada *user space* juga. Dengan demikian, hasil pengujian diharapkan dapat memberikan gambaran perilaku sistem serta jaminan yang akan diberikan oleh sistem apabila pengembang mengembangkan aplikasi pada sistem tersebut.

Karena pengujian *latency* tidak membutuhkan komunikasi dengan partisi lain, maka *timing* dapat dilakukan tanpa membutuhkan I/O. Dengan demikian, hasil pengujian *latency* merepresentasikan kemampuan maksimal dari sistem. Selain itu, hasil pengujian dapat dijadikan referensi pada saat membuat desain layanan yang kinerjanya berdasarkan kecepatan CPU (*CPU-bound*).

BAB IV

PENGEMBANGAN DAN PENGUJIAN

IV.1 Pengembangan *Primary-Backup Partition Scheduling*

Pengembangan dilakukan dengan memodifikasi kode *scheduler* ARINC 653 *source-tree* pada ARLX. ARLX telah menggabungkan sumber kode *scheduler* miliknya dengan *source-tree* milik Xen. Versi Xen yang digunakan untuk implementasi adalah versi *4.8-stable*. Masing-masing partisi akan menggunakan Linux 4.4.75 dengan *patch* PREEMPT_RT versi 4.4.75-rt88. Hal ini diperlukan untuk menjamin *latency* seminimal mungkin pada partisi.

Modifikasi kode dilakukan pada modul-modul berikut.

1. Modul libxl

Modul ini digunakan untuk membuat *tools* yang dapat digunakan pada *user space domain* dom0 untuk melakukan pengaturan *hypervisor*.

2. Modul libxc

Modul ini berisi fungsi-fungsi untuk menangani permintaan *hypercall* dari *domain*.

3. Modul *domain hypercall*

Modul ini berisi definisi *interface* yang digunakan untuk mengatur struktur data yang dikirimkan pada saat mengirim data dari *domain* kepada *hypervisor*.

4. Modul *scheduler*

Modul ini berisi fungsi-fungsi yang digunakan untuk mengisi definisi *interface scheduler* pada Xen. Fungsi-fungsi tersebut meliputi algoritma *scheduler* dan penanganan *hypercall* yang ditujukan kepada *scheduler*.

Agar algoritma *primary-backup partition scheduling* yang dijelaskan pada Subbab III.2 dapat berjalan dengan seharusnya, perlu ada mekanisme penanganan data yang dibutuhkan pada masing-masing domain. Hal tersebut dapat dicapai dengan mendefinisikan struktur data yang dapat dimengerti oleh *hypervisor* dan *user space* untuk berkomunikasi melalui *hypercall*, membuat struktur data untuk menggunakan *hypercall*, membuat mekanisme pe-

ngiriman data melalui *hypercall*, membuat struktur data untuk menyimpan informasi *domain* pada *scheduler*, serta mengubah data yang didapat oleh *hypervisor* setelah menangani *hypercall* sesuai dengan struktur data *domain* pada *scheduler*.

Implementasi mekanisme pengiriman data dari *user space* kepada *hypervisor* akan dipaparkan pada Subbab IV.1.5. Mekanisme *hypercall* yang diimplementasikan akan dipaparkan pada Subbab IV.1.4. Pendefinisian struktur data yang digunakan pada saat melakukan *hypercall* akan dipaparkan pada Subbab IV.1.3. Mekanisme konversi data yang didapat oleh *hypervisor* setelah menangani *hypercall* serta definisi struktur data masing-masing *domain* pada *scheduler* akan dijelaskan pada Subbab IV.1.1.

Penjelasan akan dilakukan secara *bottom-up* dimulai dari definisi *interface/struktur data*, kemudian fungsi dari *interface/struktur data* yang sudah didefinisikan. Masing-masing mekanisme akan dijelaskan apabila seluruh *interface/struktur data* yang digunakan pada mekanisme tersebut telah dipaparkan. Secara garis besar, penjelasan akan dilakukan dimulai dari bagian *low-level* dan *high-level* pada *hypervisor*, kemudian memasuki bagian *low-level* dan *high-level* pada *user space*. Urutan penjelasan dilakukan sedemikian dengan harapan akan mempermudah pembaca dalam mengerti masing-masing langkah yang akan dipaparkan pada setiap mekanisme.

IV.1.1 Modul *scheduler*

Subbab ini akan membahas implementasi dari *primary-backup partition scheduling* pada modul *scheduler*. Implementasi *scheduler* pada ARLX meliputi tiga buah bagian.

1. Pendefinisian *scheduler*

Pada ARLX, algoritma *scheduling* yang akan digunakan *hypervisor* untuk menjadwalkan partisi dapat dipilih pada saat *boot*. Hal ini mempermudah dalam melakukan uji coba layanan dengan *scheduler* tertentu guna menentukan algoritma *scheduling* yang akan digunakan.

2. Daftar Jadwal

Scheduler ARINC 653 memerlukan daftar partisi yang akan dijadwalkan pada rentang waktu tertentu. Selain itu, *scheduler* ARINC 653 juga memerlukan periode sebuah penjadwalan. Daftar partisi yang akan dijadwalkan dapat diberikan melalui *hypercall*.

Daftar jadwal perlu dimodifikasi untuk memodelkan daftar jadwal seperti yang telah dipaparkan pada Subbab III.2.2.

3. Informasi partisi

Scheduler ARINC 653 yang terdapat pada ARLX tidak memerlukan informasi mengenai partisi. Pengguna hanya perlu memberikan daftar partisi beserta batasan *real-time* yang harus dipenuhi. Daftar tersebut diberikan melalui *global hypercall*, yaitu *hypercall* yang tidak dalam konteks partisi tertentu. *Primary-backup partition scheduling* memerlukan informasi mengenai partisi tertentu seperti yang telah dipaparkan pada Subbab III.2.3. Dengan demikian, *scheduler* memerlukan implementasi *hypercall* spesifik domain agar *user space* dapat mengatur informasi partisi pada *scheduler*. Untuk selanjutnya, *hypercall* akan mengacu pada *hypercall* spesifik *domain* karena *global hypercall* tidak relevan dengan tugas akhir ini.

4. Fungsi *scheduling*

Algoritma *scheduling* pada ARLX diimplementasikan dengan mendefinisikan fungsi yang memiliki deklarasi seperti pada Kode IV.1. Fungsi tersebut kemudian akan dipanggil apabila *hypervisor* menerima *timer interrupt* atau partisi yang sedang mendapatkan waktu CPU telah menghabiskan quantum waktunya.

```
1 struct task_slice (*do_schedule) (const struct scheduler *, s_time_t,  
2                                bool_t tasklet_work_scheduled);
```

Kode IV.1. Deklarasi fungsi *scheduling* pada ARLX

IV.1.1.1 Pendefinisan *scheduler*

Pada Xen, pendefinisan *scheduler* dilakukan dengan menggunakan *interface* yang telah disediakan (Chisnall, 2014, p. 218). Kode IV.2 menampilkan *interface* tersebut. *Interface* tersebut kemudian didefinisikan sebagai struktur pada saat *runtime* dengan masing-masing *function pointer* menunjuk pada sebuah fungsi.

Setiap fungsi yang ditunjuk oleh *function pointer* pada struktur tersebut akan berlaku sebagai

```

1 struct scheduler {
2     char *name ;
3     char *opt_name ;
4     unsigned int sched_id ;
5
6     void      (*free_domdata) (const struct scheduler *, void *);
7     void *    (*alloc_domdata) (const struct scheduler *, struct domain *);
8     int       (*init_domain) (const struct scheduler *, struct domain *);
9     void      (*destroy_domain) (const struct scheduler *, struct domain *);
10
11    struct task_slice (*do_schedule) (const struct scheduler *, s_time_t,
12
13    int       (*adjust)      (const struct scheduler *, struct domain *,
14    int       (*adjust_global) (const struct scheduler *,
15 );

```

Kode IV.2. *Interface* dari fungsi *scheduler*

handler yang akan dipanggil oleh *hypervisor* apabila terjadi *event* yang bersesuaian. Asosiasi antara *handler* dan *event* telah ditentukan sebelumnya. Setiap *handler* pada struktur tersebut kemudian diisi dengan alamat dari fungsi yang nantinya akan menangani permintaan pada *scheduler*. Setiap *scheduler* diharuskan mengisi *handler* *do_schedule()*. Apabila *scheduler* tidak memerlukan sebuah *handler*, maka *handler* dapat diisi dengan nilai NULL.

Pada saat *boot*, konfigurasi *scheduler* yang diinginkan dapat dipilih dengan menambahkan argumen ”*sched={scheduler}*”. *Hypervisor* kemudian akan mencari *scheduler* yang didefinisikan dengan nilai *opt_name* sama dengan ”*scheduler*”.

Untuk mengimplementasikan *scheduler*, *function pointer* *do_schedule()* harus diisi dengan alamat dari fungsi yang akan melakukan pemilihan partisi. Pada implementasi *primary-backup partition scheduler*, *function pointer* tersebut akan diisi dengan alamat dari fungsi *a653sched_do_schedule()* yang akan dipaparkan pada Subbab IV.1.2.2. Agar *hypervisor* dapat mengenali *scheduler* yang diimplementasikan, *opt_name* akan diberi nilai ”*arinc653pb*”. Dengan demikian, *primary-backup partition scheduler* dapat digunakan dengan cara memberikan parameter ”*sched=arinc653pb*” pada *hypervisor* ketika *boot*.

IV.1.2 Daftar Jadwal

Pada implementasi *primary-backup partition scheduler*, daftar jadwal akan merepresentasikan layanan yang harus dijadwalkan pada rentang waktu tertentu. Masing-masing layanan akan memiliki daftar partisi yang dapat menyediakan layanan tersebut dan memiliki durasi

layanan tersebut akan disediakan. Pemodelan daftar jadwal yang baru dapat dilihat pada Kode IV.3

```
1 typedef struct sched_providers_s
2 {
3     xen_domain_handle_t dom_handle;
4     int vcpu_id;
5     struct vcpu * vc;
6 } sched_providers_t;
7
8 typedef struct sched_entry_s
9 {
10    int service_id;
11    s_time_t runtime;
12    unsigned int num_providers;
13    sched_providers_t providers[ARINC653_MAX_DOMAINS_PER_SERVICE];
14 } sched_entry_t;
```

Kode IV.3. Struktur untuk menyimpan daftar jadwal

Pada saat sistem baru dimulai, maka sistem akan menggunakan jadwal tiruan yang akan menjalankan dom0 secara terus menerus. Daftar jadwal dapat diberikan dengan menggunakan *global hypercall*. Untuk mendukung perubahan jadwal, *global hypercall* akan diperbarui agar dapat menerima pemodelan data baru. Modifikasi *global hypercall* dapat dilihat pada Kode IV.4.

```
1 struct xen_sysctl_arinc653_schedule {
2     uint64_aligned_t major_frame;
3     uint8_t num_sched_entries;
4     struct {
5         int service_id;
6         uint64_aligned_t runtime;
7         uint8_t num_providers;
8         struct {
9             xen_domain_handle_t dom_handle;
10            unsigned int vcpu_id;
11            } service_providers[ARINC653_MAX_DOMAINS_PER_SERVICE];
12        } sched_entries[ARINC653_MAX_SERVICES_PER_SCHEDULE];
13    };
```

Kode IV.4. Struktur data untuk pemanggilan *global hypercall*

Setelah melakukan *hypercall*, *scheduler* akan mengganti daftar jadwal yang sebelumnya dengan daftar jadwal yang baru.

IV.1.2.1 Informasi Partisi

```
1 typedef struct a653sched_domain_s {
2     bool healthy;
3 } a653sched_domain_t;
```

Kode IV.5. Struktur untuk menyimpan informasi partisi

Pada implementasi *primary-backup partition scheduler*, informasi partisi akan disimpan pada struktur data seperti yang ditampilkan oleh Kode IV.5. *Scheduler* dapat mengetahui keadaan partisi apabila partisi tersebut memberikan informasi tentang dirinya melalui *hypercall* yang akan didefinisikan pada Subbab IV.1.3.

```
1 static int
2 a653sched_init_domain(const struct scheduler *ops,
3                      struct domain *dom)
4 {
5     a653sched_domain_t *sdom;
6
7     sdom = xzalloc(a653sched_domain_t);
8     if ( sdom == NULL )
9         return -ENOMEM;
10
11    sdom->healthy = true;
12
13    dom->sched_priv = sdom;
14
15    return 0;
16 }
```

Kode IV.6. Fungsi inisialisasi partisi

Pada saat pembuatan partisi, *scheduler* akan memanggil fungsi yang ditunjuk oleh *function pointer* *init_domain* pada struktur yang telah dijelaskan pada Subbab IV.1.1.1. Tujuan fungsi tersebut adalah untuk mengalokasikan memori serta menginisialisasi informasi partisi pada saat dibuat. Fungsi tersebut diimplementasikan seperti pada Kode IV.6. Setelah dibuat, informasi partisi akan memiliki nilai awal yang menandakan bahwa partisi tersebut sehat.

Pada saat penghapusan partisi, *scheduler* akan memanggil fungsi yang ditunjuk oleh *function pointer* *destroy_domain* pada struktur yang telah dijelaskan pada Subbab IV.1.1.1.

```

1 static void
2 a653sched_destroy_domain(const struct scheduler *ops, struct domain *dom)
3 {
4     dom->sched_priv = NULL;
5     xfree(dom->sched_priv);
6 }
7 }
```

Kode IV.7. Fungsi penghapusan partisi

Tujuan fungsi tersebut adalah untuk dealokasi memori yang digunakan oleh informasi partisi yang tersimpan pada *scheduler*.

Implementasi fungsi penghapusan informasi partisi dapat dilihat pada Kode IV.7. Dealokasi memori akan membuat pengaksesan memori tersebut mengakibatkan kegagalan pada *hypervisor* sehingga *hypervisor* akan berhenti bekerja. Karena itu, memori tempat penyimpanan struktur informasi partisi milik partisi yang dihapus tidak boleh diakses lagi. Fungsi tersebut hanya akan dipanggil apabila terdapat *interrupt* yang meminta agar *domain* tersebut dihapus. Maka, penggunaan informasi partisi sebaiknya dimulai dengan permintaan *lock* dan mematikan *interrupt*, kemudian pada saat ingin melakukan pembacaan maupun penulisan pada informasi, *scheduler* perlu melakukan pengecekan nilai *pointer* untuk memastikan informasi partisi masih valid, dan diakhiri dengan pelepasan *lock* dan menyalakan *interrupt*.

IV.1.2.2 Fungsi *Scheduling*

Algoritma *scheduling* yang akan digunakan pada *primary-backup partition scheduler* memiliki banyak kesamaan dengan algoritma *scheduling* yang digunakan oleh *scheduler* ARINC 653 pada ARLX. Dengan demikian, implementasi algoritma *scheduling* akan menggunakan fungsi yang sudah ada pada *scheduler* ARINC 653 yang kemudian dimodifikasi sehingga bekerja sebagaimana telah dipaparkan pada Subbab III.2.4. Modifikasi yang dilakukan dapat dilihat pada Kode IV.8.

Karena fungsi *scheduling* akan dipanggil setiap kali sebuah partisi menghabiskan kuantum waktu atau *hypervisor* mendapatkan *timer interrupt*, keadaan algoritma pada fungsi *scheduling* harus bersifat tetap pada setiap pemanggilan. Fungsi *scheduling* ARINC 653 pada ARLX memiliki kompleksitas waktu $O(1)$ untuk setiap pemanggilan. Untuk menjaga efisi-

```

8     ...
9     static sched_entry_t * entry;
10
11    if ( sched_priv->num_schedule_entries < 1 )
12        sched_priv->next_major_frame = now + DEFAULT_TIMESLICE;
13    else if ( now >= sched_priv->next_major_frame )
14    {
15        sched_index = 0;
16        entry = sched_priv->schedule;
17
18        sched_priv->next_major_frame = now + sched_priv->major_frame;
19        next_switch_time = now + entry->runtime;
20        current_provider = providers_candidate(entry);
21    }
22    else
23    {
24        while ( (now >= next_switch_time)
25               && (sched_index < sched_priv->num_schedule_entries) )
26        {
27            /* time to switch to the next domain in this major frame */
28            sched_index++;
29            entry = sched_priv->schedule + sched_index;
30            next_switch_time += entry->runtime;
31        }
32
33        /* Choose the most appropriate provider for current entry */
34        current_provider = providers_candidate(entry);
35    }
36
37    if ( current_provider != NULL )
38        new_task = current_provider->vc;
39    ...

```

Kode IV.8. Implementasi algoritma *scheduling*

```

1 typedef struct xen_domctl_sched_arinc653 {
2     uint8_t healthy;
3 } xen_domctl_sched_arinc653_t;

```

Kode IV.9. Data struktur untuk komunikasi yang dapat dimengerti
scheduler dan *user space*

ensi kinerja sistem secara keseluruhan, algoritma *scheduling* pada *primary-backup partition scheduler* diimplementasikan sedemikian sehingga memiliki kompleksitas waktu yang sama dengan algoritma *scheduling* pada *scheduler* ARINC 653.

IV.1.3 Modul *domain hypercall*

Subbab ini akan membahas implementasi dan modifikasi pada modul *domain hypercall* agar dapat mengimplementasikan *hypercall* sebagai metode komunikasi dan pemindahan data antara *user space* dengan *hypervisor* terkait informasi partisi seperti yang sudah dibahas pada Subbab III.2.3.

```

1 struct xen_domctl_scheduler_op {
2     uint32_t sched_id;
3     uint32_t cmd;
4
5     union {
6         xen_domctl_sched_credit_t credit;
7         xen_domctl_sched_credit2_t credit2;
8         xen_domctl_sched_rtds_t rtlds;
9         xen_domctl_sched_arinc653_t arinc653;
10        struct {
11            XEN_GUEST_HANDLE_64(xen_domctl_schedparam_vcpu_t) vcpus;
12            uint32_t nr_vcpus;
13            uint32_t padding;
14        } v;
15    } u;
16 };

```

Kode IV.10. Data struktur untuk pemindahan data pada saat melakukan *domain hypercall*

Implementasi *hypercall* dilakukan dengan mendeklarasikan struktur seperti pada Kode IV.9.

Deklarasi struktur tersebut bertujuan agar terdapat *interface* struktur data yang dapat dimengerti oleh *scheduler* dan *user space* sehingga keduanya dapat berkomunikasi. Struktur tersebut kemudian menjadi salah satu *union member* pada struktur generik yang digunakan oleh *Xen* untuk pemindahan data. Struktur tersebut dapat dilihat pada Kode IV.10. Metode transfer informasi partisi menggunakan struktur tersebut akan dipaparkan pada Subbab IV.1.4.

IV.1.4 Modul libxc

Subbab ini akan membahas implementasi dan modifikasi pada modul libxc, yaitu modul yang menangani komunikasi antara *hypervisor* dengan *user space* pada *low-level*. Fungsi-fungsi pada libxc akan dikompilasi menjadi sebuah *library* milik sistem sehingga seluruh *tools* pada *user space* dapat menggunakan fungsi-fungsi yang terdapat pada libxc. Modifikasi pada modul libxc diperlukan agar *user space* mengetahui cara untuk berkomunikasi dengan *scheduler* yang akan dibuat.

Agar *primary-backup partition scheduling* dapat bekerja, *libxc* harus dapat berkomunikasi dua arah dengan *hypervisor*. Komunikasi dilakukan dengan mengirimkan struktur Kode IV.10 melalui *hypercall*. Member *sched_id* pada struktur tersebut diberi nilai yang menandakan bahwa struktur yang dikirimkan melalui *hypercall* tersebut ditujukan untuk komponen *scheduler*. Kemudian, member *cmd* akan diberi nilai yang menandakan apakah *user space* akan mengirimkan informasi partisi kepada *scheduler* atau meminta informasi partisi

dari *scheduler*.

Implementasi fungsi untuk mengirimkan informasi partisi kepada *scheduler* oleh *user space* dapat dilihat pada Kode IV.11 sedangkan fungsi untuk meminta informasi dari *scheduler* oleh *user space* dapat dilihat pada Kode IV.12.

IV.1.4.1 Fungsi Pengiriman Informasi Partisi

```
1 int
2 xc_sched_arinc653_domain_set(
3     xc_interface *xch,
4     uint32_t domid,
5     struct xen_domctl_sched_arinc653 *sdom)
6 {
7     int rc;
8     DECLARE_DOMCTL;
9
10    domctl.cmd = XEN_DOMCTL_scheduler_op;
11    domctl.domain = (domid_t) domid;
12    domctl.u.scheduler_op.sched_id = XEN_SCHEDULER_ARINC653;
13    domctl.u.scheduler_op.cmd = XEN_DOMCTL_SCHEDOP_putinfo;
14    domctl.u.scheduler_op.u.arinc653 = *sdom;
15
16    rc = do_domctl(xch, &domctl);
17
18    return rc;
19 }
```

Kode IV.11. Fungsi yang digunakan *user space* mengirim informasi kepada *scheduler*

Fungsi untuk mengirimkan informasi partisi kepada *scheduler* membutuhkan identifikasi partisi dan struktur yang akan dikirim melalui *hypercall*. Identifikasi partisi harus berupa nilai yang valid (terdapat partisi yang memiliki identifikasi dengan nilai tersebut) agar pemanggilan *hypercall* berhasil. Struktur yang dikirim adalah struktur yang dapat dimengerti oleh *scheduler* seperti yang telah ditampilkan pada Kode IV.10.

Fungsi pada Kode IV.11 diimplementasikan dengan meminta alokasi memori pada *hypervisor* yang berkorespondensi dengan memori pada *user space*. Permintaan tersebut dilakukan dengan menggunakan *macro* DECLARE_DOMCTL. Dengan menggunakan hasil dari alokasi memori, *hypervisor* dapat menyalin isi dari memori pada *user space* menjadi isi dari memori korespondensinya pada *hypervisor*.

IV.1.4.2 Fungsi Permintaan Informasi Partisi

```
1 int
2 xc_sched_arinc653_domain_get(
3     xc_interface *xch,
4     uint32_t domid,
5     struct xen_domctl_sched_arinc653 *sdom)
6 {
7     int rc;
8     DECLARE_DOMCTL;
9
10    domctl.cmd = XEN_DOMCTL_scheduler_op;
11    domctl.domain = (domid_t) domid;
12    domctl.u.scheduler_op.sched_id = XEN_SCHEDULER_ARINC653;
13    domctl.u.scheduler_op.cmd = XEN_DOMCTL_SCHEDOP_getinfo;
14
15    rc = do_domctl(xch, &domctl);
16
17    if ( rc == 0 )
18        *sdom = domctl.u.scheduler_op.u.arinc653;
19
20    return rc;
21 }
```

Kode IV.12. Fungsi yang digunakan *user space* meminta informasi partisi dari *scheduler*

Fungsi untuk meminta informasi partisi kepada *scheduler* membutuhkan identifikasi partisi dan memori yang akan digunakan untuk menyimpan informasi partisi yang didapat melalui *hypercall*. Identifikasi partisi harus berupa nilai yang valid (terdapat partisi yang memiliki identifikasi dengan nilai tersebut) agar pemanggilan *hypercall* berhasil. Ukuran dari memori yang digunakan untuk melakukan *hypercall* harus sama dengan ukuran struktur pada Kode IV.10.

Fungsi tersebut diimplementasikan seperti pada Kode IV.12. Secara umum implementasi fungsi tersebut sama seperti implementasi fungsi permintaan informasi partisi seperti pada Subbab IV.1.4.2. Perbedaannya hanya pada nilai yang digunakan oleh *hypervisor* untuk menentukan arah komunikasi *hypercall*.

IV.1.5 Modul libxl

Subbab ini akan membahas implementasi dan modifikasi pada modul libxl, yaitu modul yang menangani konfigurasi *domain* pada *high-level*. Libxl digunakan untuk membangun kertas xl yang digunakan oleh administrator sistem untuk mengubah konfigurasi terkait de-

ngan *hypervisor* melalui *user space*. Fungsionalitas yang ditawarkan libxl pada umumnya menggunakan fungsi-fungsi yang ditawarkan oleh libxc. Dengan demikian, administrator dapat dengan mudah melakukan *hypercall* tanpa perlu membuat program lain.

```

1 static int sched_arinc653_domain_get(libxl__gc *gc, uint32_t domid,
2                                     libxl_domain_sched_params *scinfo)
3 {
4     struct xen_domctl_sched_arinc653 sdom;
5     int rc;
6
7     rc = xc_sched_arinc653_domain_get(CTX->xch, domid, &sdom);
8
9     libxl_domain_sched_params_init(scinfo);
10    scinfo->sched = LIBXL_SCHEDULER_ARINC653;
11    scinfo->parent = sdom.parent;
12    scinfo->healthy = sdom.healthy;
13
14    return rc;
15 }
16
17 static int sched_arinc653_domain_set(libxl__gc *gc, uint32_t domid,
18                                     const libxl_domain_sched_params *scinfo)
19 {
20     struct xen_domctl_sched_arinc653 sdom;
21     int rc;
22
23     rc = xc_sched_arinc653_domain_get(CTX->xch, domid, &sdom);
24     if (rc != 0) {
25         LOGE(ERROR, "getting domain sched arinc653");
26         return ERROR_FAIL;
27     }
28
29     if (scinfo->parent != LIBXL_DOMAIN_SCHED_PARAM_PARENT_DEFAULT)
30         sdom.parent = scinfo->parent;
31     sdom.healthy = scinfo->healthy;
32     rc = xc_sched_arinc653_domain_set(CTX->xch, domid, &sdom);
33
34     return rc;
35 }
```

Kode IV.13. xl berkomunikasi dengan menggunakan fungsi milik xc

Modifikasi yang dilakukan pada libxl adalah menambahkan opsi pada xl sehingga administrator sistem dapat mengubah informasi mengenai partisi dan memanggil fungsi yang digunakan untuk berkomunikasi dengan *hypervisor* melalui *user space*. libxl akan memanggil kode libxc seperti pada Kode IV.13. Kode tersebut akan memanggil fungsi-fungsi yang telah dibahas sebelumnya pada Subbab IV.1.4.

IV.2 Pengujian

Pengujian dilakukan dengan menggunakan perangkat lunak yang melakukan pengukuran keandalan serta *timing* seperti yang telah dipaparkan pada Subbab III.3. *Platform* yang

digunakan untuk melakukan pengujian sama dengan *platform* yang telah dipaparkan Subbab IV.1. Pengujian dilakukan pada Laptop Lenovo® G40 Series. Laptop tersebut menggunakan Intel® Core™ i5.

Hasil pengujian akan ditampilkan dengan menggunakan dua buah *scatter plot*. *Scatter plot* pertama akan menampilkan hubungan antara waktu sekarang dengan *delta*, yaitu selisih waktu antara dua buah *timing* yang berurutan. *Scatter plot* kedua akan menampilkan hubungan antara waktu sekarang dengan *galat*, yaitu selisih waktu antara *delta* dengan nilai ekspektasi. Secara formal, perhitungan *delta* dapat dilihat pada Persamaan IV.1 dan perhitungan *galat* dapat dilihat pada Persamaan IV.2.

$$\textit{delta} = \textit{now} - \textit{last} \quad (\text{IV.1})$$

Pada perhitungan nilai *delta*, nilai *now* adalah waktu pada saat hasil *timing* sekarang didapatkan dan nilai *last* adalah waktu pada saat hasil *timing* sebelumnya didapatkan. Nilai *delta* berguna untuk melihat selisih antara dua buah *timing* serta melihat arah dari perbedaan tersebut.

$$\textit{galat} = \textit{delta} - \textit{expected} \quad (\text{IV.2})$$

Pada perhitungan nilai *galat*, nilai *delta* didapat melalui Persamaan IV.1 dan nilai *expected* adalah nilai ekspektasi *timing*. Nilai ekspektasi yang digunakan dalam perhitungan *galat* adalah periode dari *timing*. Nilai *galat* berguna untuk melihat seberapa jauh penyimpangan antara dua buah *timing* yang didapat.

IV.2.1 Pengujian Keandalan

Pengujian keandalan akan dilakukan dengan menggunakan skenario-skenario berikut.

1. Skenario 1

Pada skenario ini, daftar jadwal berisi 1 layanan dan memiliki 1 partisi penyedia. Masing-masing akan berjalan selama 20 ms dengan *major time frame* keseluruhan selama 20 ms.

Hasil pengujian dengan menggunakan skenario ini akan digunakan sebagai referensi untuk *latency* maksimal terkecil yang mungkin dialami oleh sebuah layanan pada pengujian ini. Perlu diperhatikan karena pengujian akan menggunakan protokol TCP seperti yang telah dipaparkan pada Subbab III.3, maka *latency* maksimal terkecil yang mungkin dialami oleh sebuah layanan seharusnya lebih kecil. Namun, apabila layanan memang membutuhkan I/O, maka hasil pengujian dengan menggunakan skenario ini juga dapat dijadikan referensi pada saat pembuatan desain layanan.

2. Skenario 2

Pada skenario ini, daftar jadwal berisi 1 layanan dan memiliki 2 partisi penyedia. Masing-masing akan berjalan selama 20 ms dengan *major time frame* keseluruhan selama 20 ms.

Hasil pengujian dengan menggunakan skenario akan digunakan untuk melihat kinerja *scheduler* dalam menerapkan skema *primary-backup*. Skenario akan membuat partisi *primary* gagal dalam rentang waktu tertentu. Data yang terkumpul pada saat pergantian partisi penyedia layanan dari partisi *primary* menjadi partisi *backup* akan menunjukkan waktu pergantian serta *latency* layanan tersebut apabila terjadi pergantian.

3. Skenario 3

Pada skenario ini, daftar jadwal berisi 9 layanan dan memiliki 1 partisi penyedia. Masing-masing akan berjalan selama 20 ms dengan *major time frame* keseluruhan selama 180 ms.

Skenario ini dilakukan untuk melihat kinerja sistem apabila terdapat banyak partisi tanpa pergantian. Sama seperti skenario 1, skenario ini hanya digunakan sebagai referensi untuk *latency* maksimal terkecil yang mungkin dialami oleh sebuah layanan. Hasil pengujian dengan menggunakan skenario ini akan memperlihatkan hasil yang berbeda dari hasil pengujian apabila terdapat perbedaan kinerja sistem pada saat terdapat banyak layanan.

4. Skenario 4

Pada skenario ini, daftar jadwal berisi 2 layanan yang sama dan memiliki 1 partisi penyedia. Karena layanan tersebut merupakan layanan yang sama, maka partisi penyedia layanan 1 dan layanan 2 merupakan partisi yang sama. Partisi akan berjalan selama 20 ms pada masing-masing layanan. *Major time frame* keseluruhan pengujian

adalah 40 ms.

Skenario ini dilakukan untuk mengukur waktu tambahan pada *latency* yang diakibatkan oleh pergantian partisi. Apabila terdapat perbedaan antara hasil pengujian pada saat menggunakan skenario ini dengan hasil pengujian pada skenario dengan lebih dari satu partisi, maka dapat disimpulkan bahwa pergantian partisi pada Xen akan menambah *latency*. Meski secara sekilas skenario ini digunakan hanya untuk mengukur *latency*, namun hasil pengujian dengan menggunakan skenario ini akan dibandingkan dengan skenario pengujian keandalan lain. Dengan demikian, skenario ini dimasukkan sebagai pengujian keandalan.

5. Skenario 5

Pada skenario ini, daftar jadwal berisi 4 layanan dan masing-masing memiliki 2 partisi penyedia. Masing-masing akan berjalan selama 10 ms dengan *major time frame* keseluruhan selama 50 ms (terdapat 10 ms partisi *idle*).

Hasil skenario ini digunakan untuk melihat kinerja *scheduler* dalam menerapkan skema *primary-backup* pada saat sistem memiliki banyak layanan. Partisi *primary* pada masing-masing layanan akan digagalkan secara berkala. Pemrosesan data akan sama dengan skenario 2 hanya saja pada beberapa layanan yang berjalan pada saat yang bersamaan.

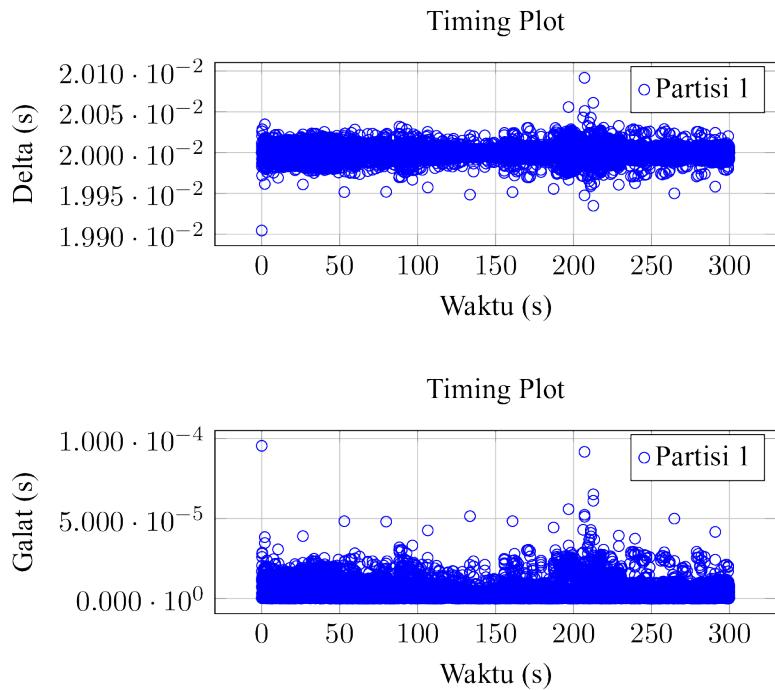
6. Skenario 6

Pada skenario ini, daftar jadwal berisi 3 layanan. Layanan 1 memiliki 2 partisi penyedia dan layanan akan berjalan selama 30 ms. Layanan 2 memiliki 3 partisi penyedia dan layanan akan berjalan selama 20 ms. Layanan 3 memiliki 1 buah partisi penyedia dan layanan akan berjalan selama 10 ms. *Major time frame* keseluruhan pengujian adalah 60 ms.

Skenario ini dilakukan untuk melihat kinerja sistem pada kasus umum. Selain itu, skenario juga akan menguji *scheduler* apabila durasi setiap layanan berbeda-beda.

Hasil pengujian keandalan sistem berdasarkan skenario-skenario pengujian akan dipaparkan. Pengujian diambil dengan menjalankan program pengujian selama 2 jam. Namun, hasil pengujian hanya akan ditampilkan sampai terlihat sebuah kecenderungan pada data.

1. Skenario 1

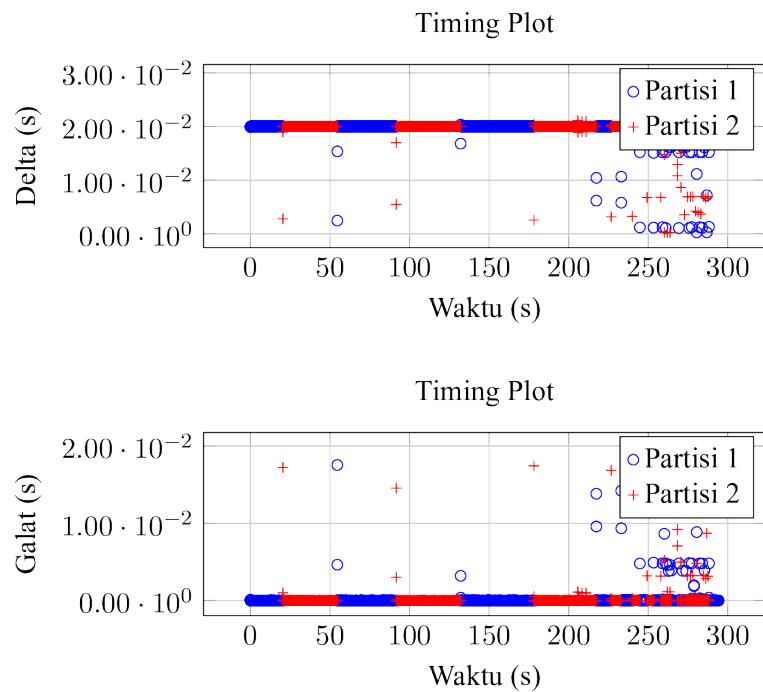


Gambar IV.1. Hasil pengujian keandalan sistem menggunakan skenario 1

Hasil pengujian keandalan dengan menggunakan skenario 1 dapat dilihat pada Gambar IV.1. Pengujian dengan menggunakan skenario 1 menghasilkan *timing* yang sangat presisi, dengan galat *timing* tertinggi dari ekspektasi periode *timing* yang seharusnya hanya sekitar $100 \mu\text{s}$. Pada pengujian ini, perilaku algoritma *primary-backup partition scheduling* tidak dapat terlihat. Namun, pengujian ini dapat digunakan sebagai referensi *timing* yang seharusnya didapatkan pada pengujian-pengujian berikutnya.

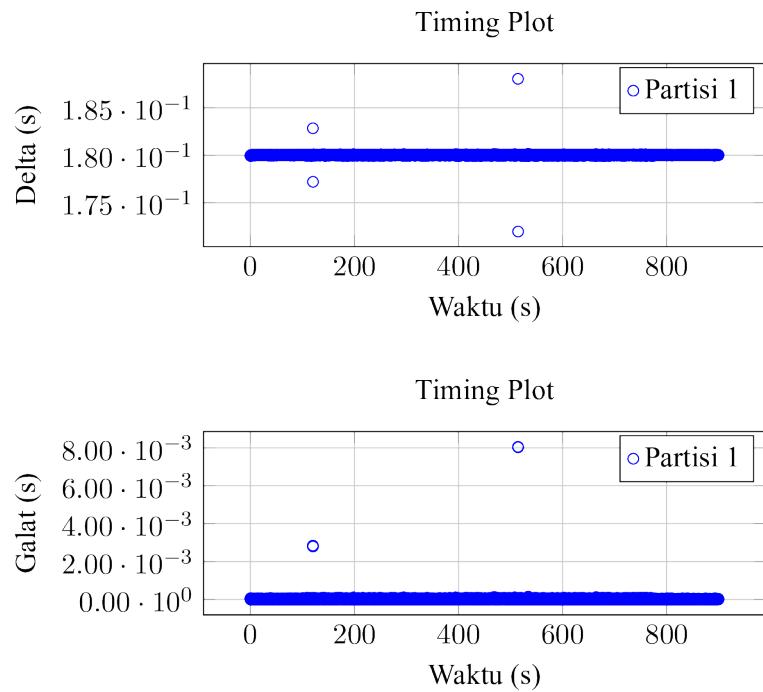
2. Skenario 2

Hasil pengujian keandalan dengan menggunakan skenario 2 dapat dilihat pada Gambar IV.2. Pada pengujian dengan menggunakan skenario 2, dapat terlihat bahwa layanan tersedia selama berjalananya program pengujian. Ketika partisi 1 mengalami kegagalan, partisi 2 mengambil alih peran partisi 1 dalam menyediakan layanan tersebut. Namun, semakin lama program pengujian berjalan, maka semakin banyak *timing* yang tidak sesuai dengan ekspektasi. *Timing* tersebut didapat lebih cepat dari yang seharusnya. Selain itu, dapat dilihat bahwa pada saat pergantian partisi, *timing* layanan menjadi semakin cepat.



Gambar IV.2. Hasil pengujian keandalan sistem menggunakan skenario 2

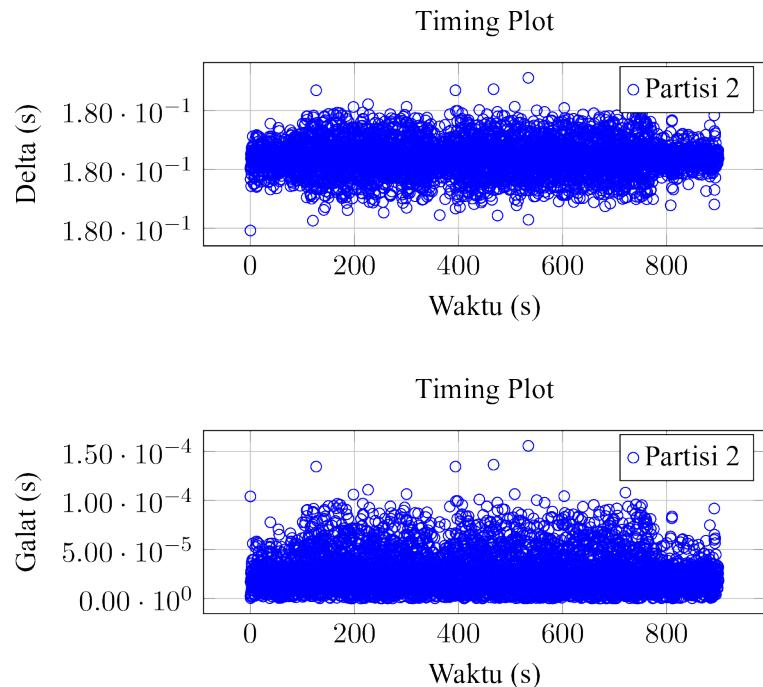
3. Skenario 3



Gambar IV.3. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 1)

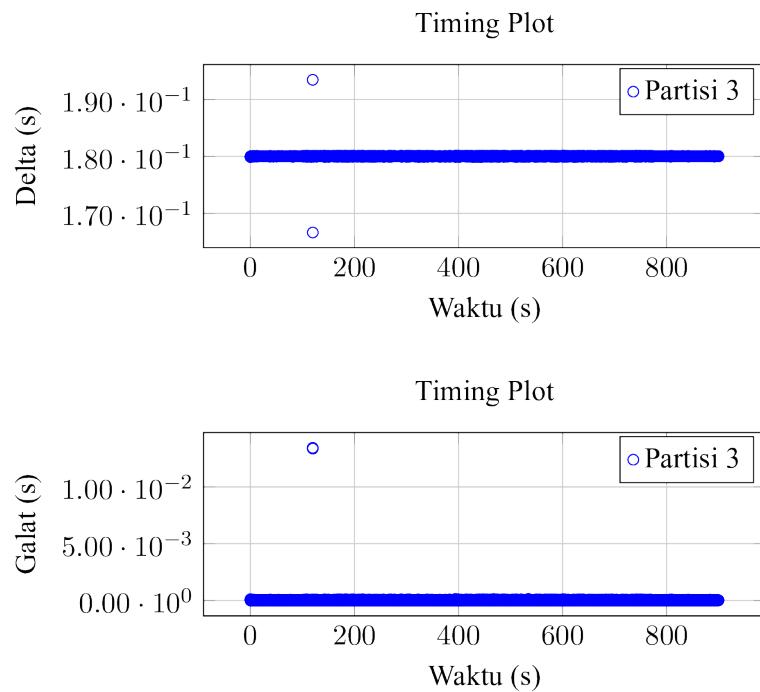
Hasil pengujian keandalan dengan menggunakan skenario 3 dapat dilihat pada Gambar IV.3, Gambar IV.4, Gambar IV.5, Gambar IV.6, Gambar IV.7, Gambar IV.8, Gambar IV.9, Gambar IV.10, dan Gambar IV.11. Hasil pengujian pada masing-masing layanan adalah sebagai berikut.

- (a) Pada layanan 1, partisi berjalan dengan cukup stabil. Galat antara maksimal pengukuran *timing* dengan *timing* ekspektasi cukup tinggi yaitu $800 \mu\text{s}$.

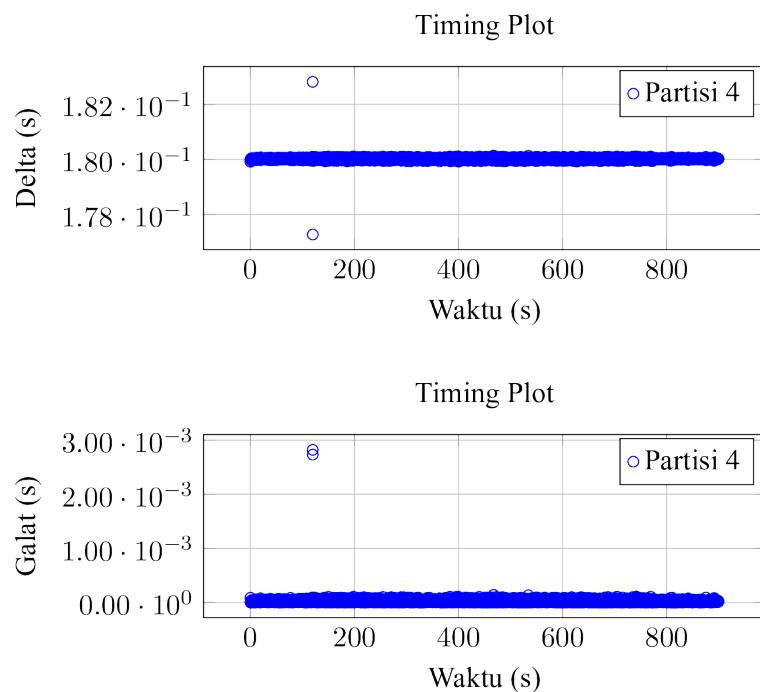


Gambar IV.4. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 2)

- (b) Pada layanan 2, partisi berjalan dengan sangat stabil. Galat antara maksimal pengukuran *timing* dengan *timing* ekspektasi sangat rendah yaitu $15 \mu\text{s}$.
- (c) Pada layanan 3, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 150 ms.
- (d) Pada layanan 4, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 3 ms.
- (e) Pada layanan 5, partisi berjalan dengan sangat stabil. Galat antara maksimal



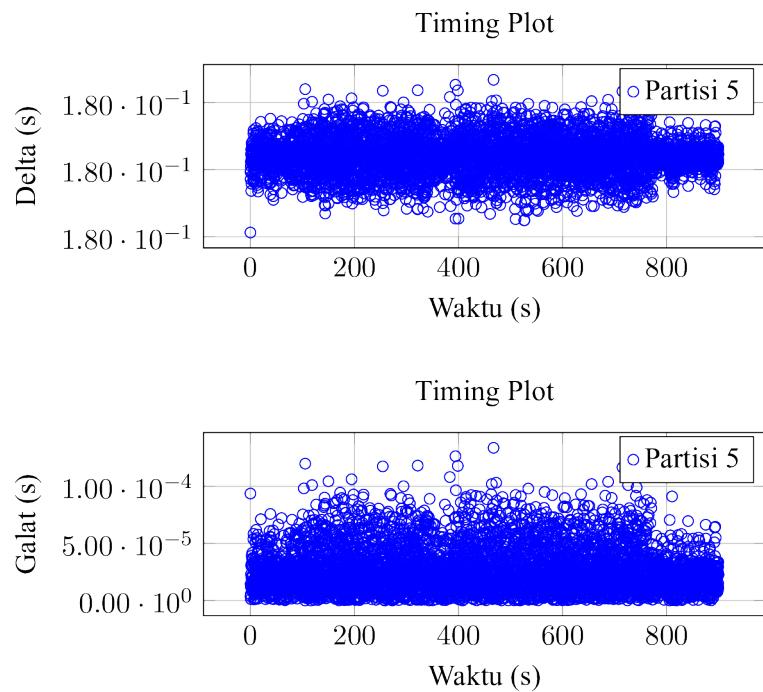
Gambar IV.5. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 3)



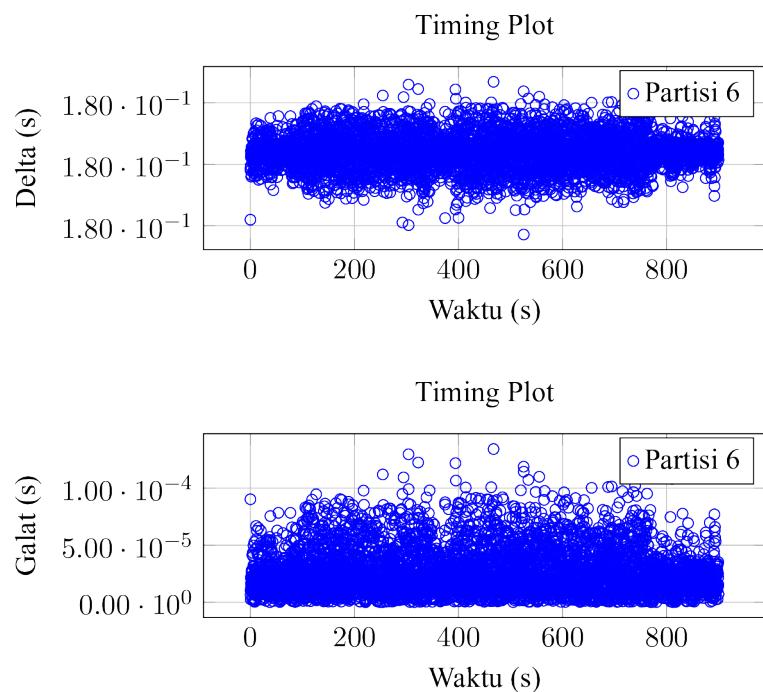
Gambar IV.6. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 4)

pengukuran *timing* dengan *timing* ekspektasi sangat rendah yaitu sekitar $13 \mu\text{s}$.

- (f) Pada layanan 6, partisi berjalan dengan sangat stabil. Galat antara maksimal



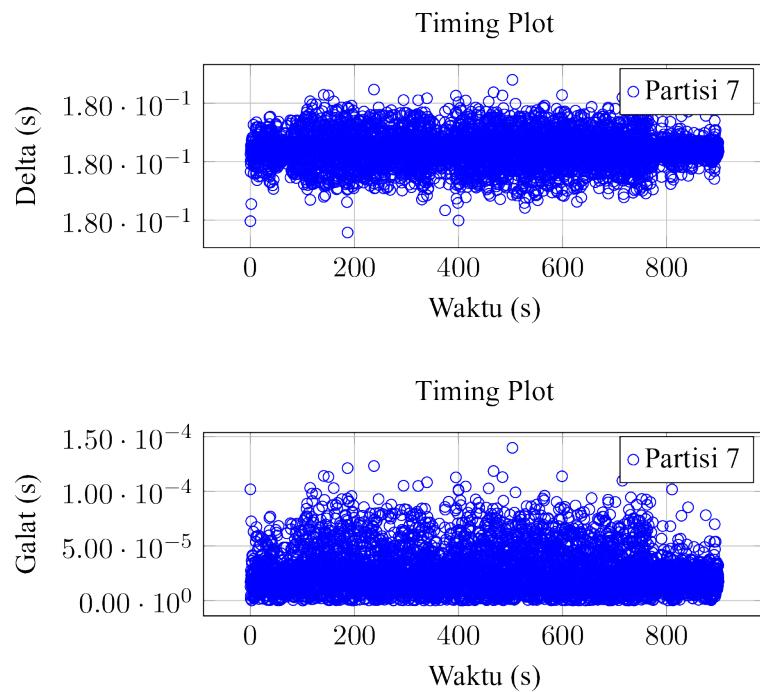
Gambar IV.7. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 5)



Gambar IV.8. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 6)

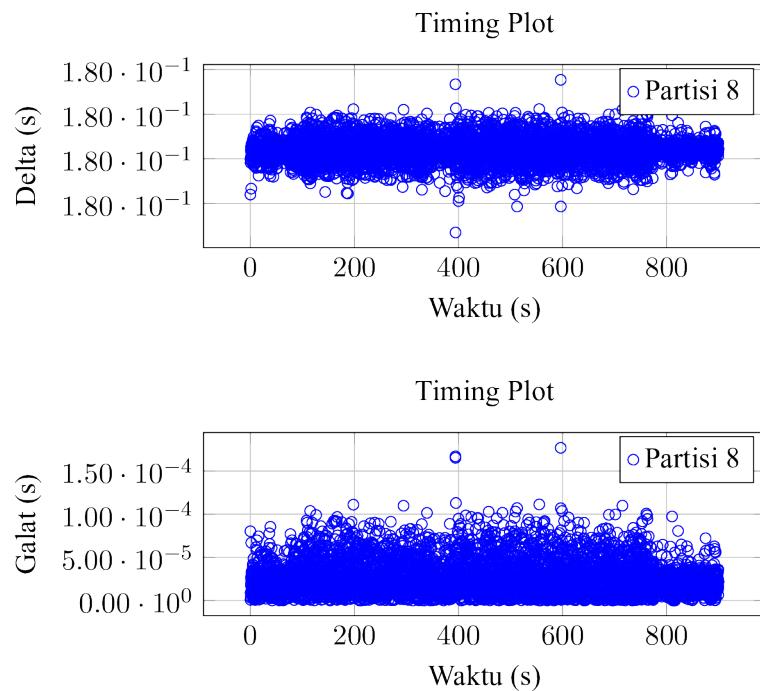
pengukuran *timing* dengan *timing* ekspektasi sangat rendah yaitu $13 \mu\text{s}$.

- (g) Pada layanan 7, partisi berjalan dengan sangat stabil. Galat antara maksimal



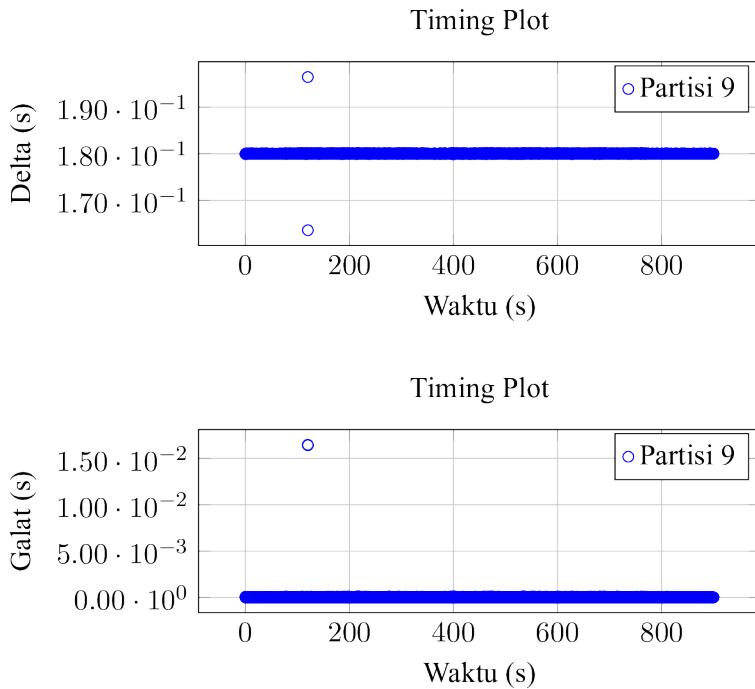
Gambar IV.9. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 7)

pengukuran *timing* dengan *timing* ekspektasi sangat rendah yaitu $15 \mu\text{s}$.



Gambar IV.10. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 8)

- (h) Pada layanan 8, partisi berjalan dengan sangat stabil. Galat antara maksimal pengukuran *timing* dengan *timing* ekspektasi sangat rendah yaitu $15 \mu\text{s}$.



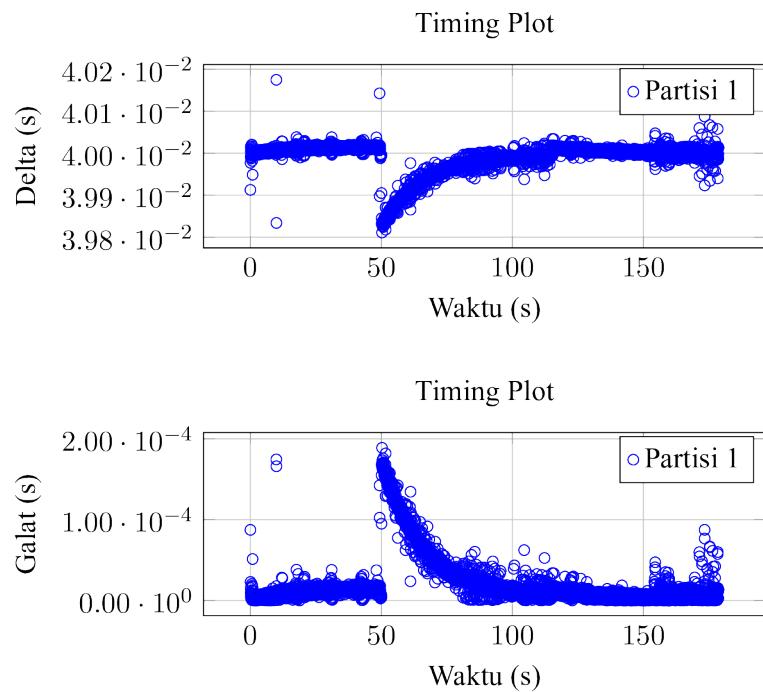
Gambar IV.11. Hasil pengujian keandalan sistem menggunakan skenario 3 (layanan 9)

- (i) Pada layanan 9, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 15 ms .

Secara umum, partisi berjalan dengan sangat stabil. Namun, pada sistem *real-time*, hasil pengukuran terpenting adalah *latency* maksimal sebuah *task* selesai dari jadwal *task* tersebut yang seharusnya. Maka, dapat dikatakan bahwa hasil pengujian dengan menggunakan skenario 3 sangatlah buruk.

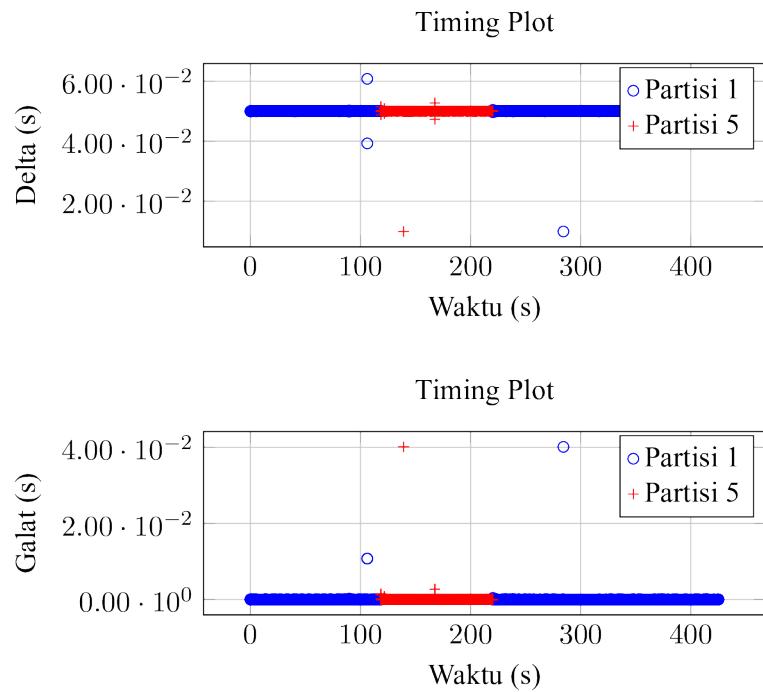
4. Skenario 4

Hasil pengujian keandalan pada skenario 4 dapat dilihat pada Gambar IV.12. Hasil pengujian dengan menggunakan skenario 4 menunjukkan bahwa apabila tidak terdapat *partition switching*, maka *timing* yang didapat cukup presisi. Namun, pada beberapa waktu tertentu tetap terdapat galat yang cukup besar pada perbedaan *timing* yang didapat dengan *timing* ekspektasi.



Gambar IV.12. Hasil pengujian keandalan sistem menggunakan skenario 4

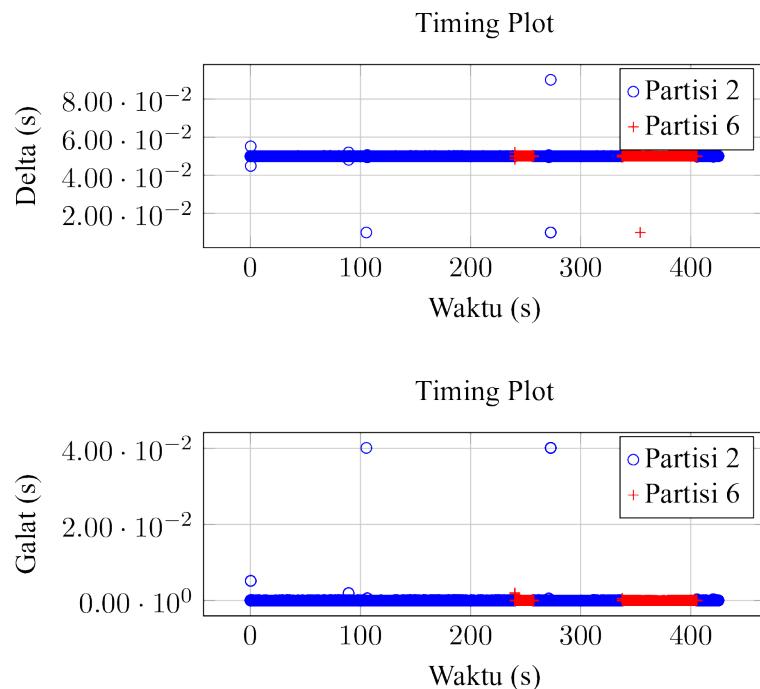
5. Skenario 5



Gambar IV.13. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 1)

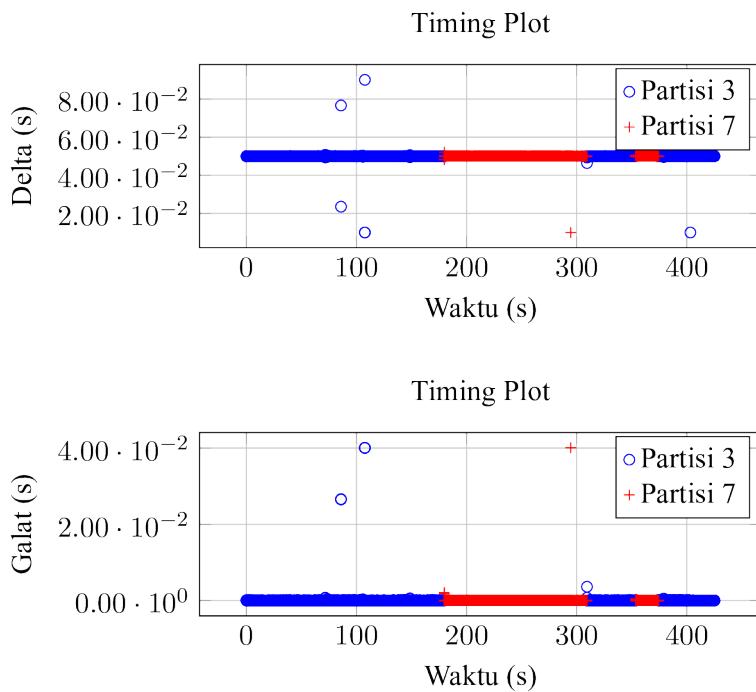
Hasil pengujian keandalan pada skenario 5 dapat dilihat pada Gambar IV.13. Gambar IV.14. Gambar IV.15. dan Gambar IV.16. Hasil pengujian untuk masing-masing layanan adalah sebagai berikut.

- (a) Pada layanan 1, partisi berjalan dengan stabil. Dari hasil pengujian, terlihat bahwa partisi 5 dapat mengambil alih peran partisi 1 dalam menyediakan layanan 1 ketika partisi 1 mengalami kegagalan. Dengan demikian, layanan 1 tersedia selama rentang waktu pengujian. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 40 ms.



Gambar IV.14. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 2)

- (b) Pada layanan 2, partisi berjalan dengan stabil. Sama seperti pada layanan 1, terlihat bahwa partisi 6 dapat mengambil alih peran partisi 2 dalam menyediakan layanan 2 ketika partisi 2 mengalami kegagalan. Dengan demikian, layanan 2 tersedia selama rentang waktu pengujian. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 40 ms.
- (c) Pada layanan 3, partisi berjalan dengan stabil. Sama seperti pada layanan-layanan



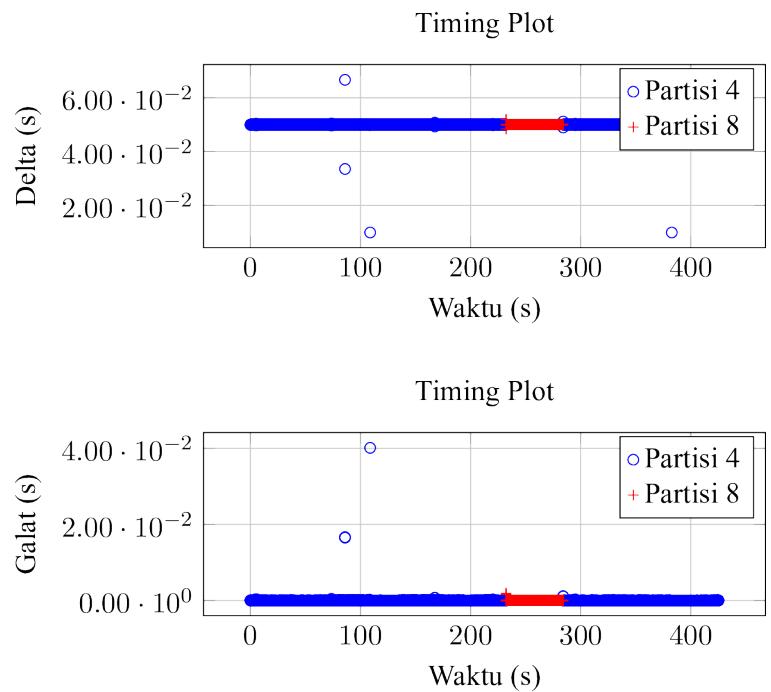
Gambar IV.15. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 3)

sebelumnya, terlihat bahwa partisi 7 dapat mengambil alih peran partisi 3 dalam menyediakan layanan 3 ketika partisi 3 mengalami kegagalan. Dengan demikian, layanan 3 tersedia selama rentang waktu pengujian. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 40 ms. Selain itu, terlihat beberapa titik dengan galat yang cukup tinggi berada pada waktu sekitar pergantian partisi *primary* dengan partisi *backup*.

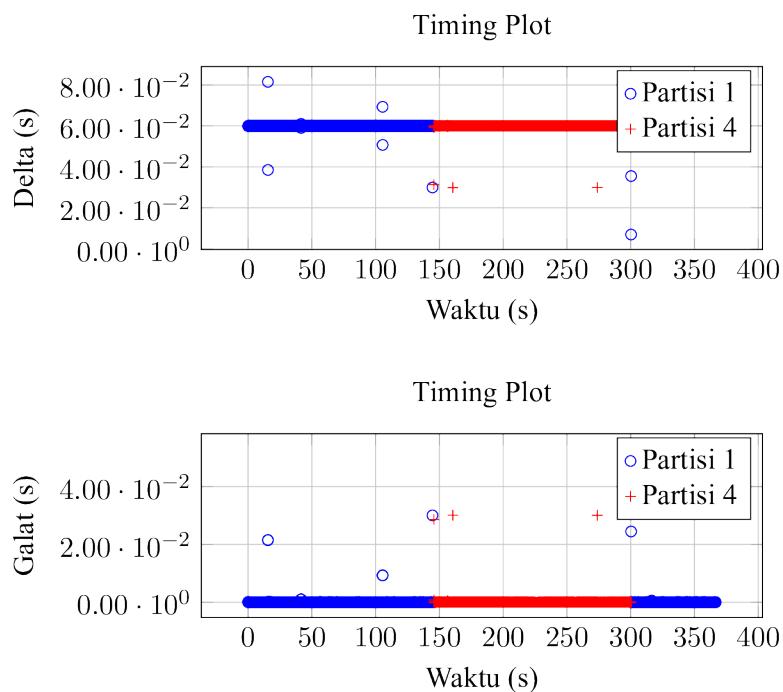
- (d) Pada layanan 4, partisi berjalan dengan stabil. Sama seperti pada layanan-layanan sebelumnya, terlihat bahwa partisi 8 dapat mengambil alih peran partisi 4 dalam menyediakan layanan 4 ketika partisi 4 mengalami kegagalan. Dengan demikian, layanan 4 tersedia selama rentang waktu pengujian. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 40 ms.

Sama seperti pada skenario 3, secara umum partisi berjalan dengan sangat stabil. Namun, pada pengujian ini terlihat bahwa banyak titik tidak stabil berada pada sekitar waktu pergantian antara partisi *primary* dengan partisi *backup*.

6. Skenario 6



Gambar IV.16. Hasil pengujian keandalan sistem menggunakan skenario 5 (layanan 4)

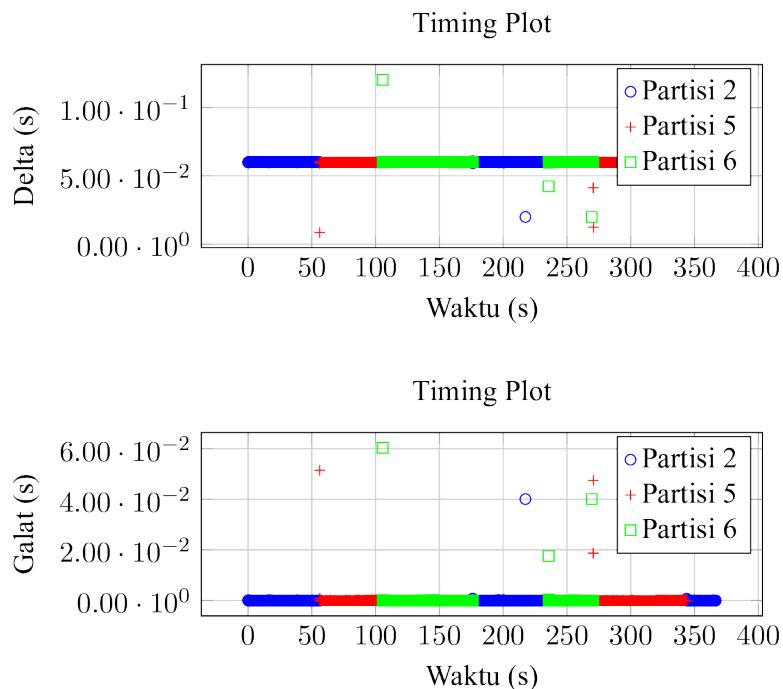


Gambar IV.17. Hasil pengujian keandalan sistem menggunakan skenario 6 (layanan 1)

Hasil pengujian keandalan pada skenario 6 dapat dilihat pada Gambar IV.17, Gambar IV.18, Gambar IV.19. Hasil pengujian pada masing-masing layanan adalah seba-

gai berikut.

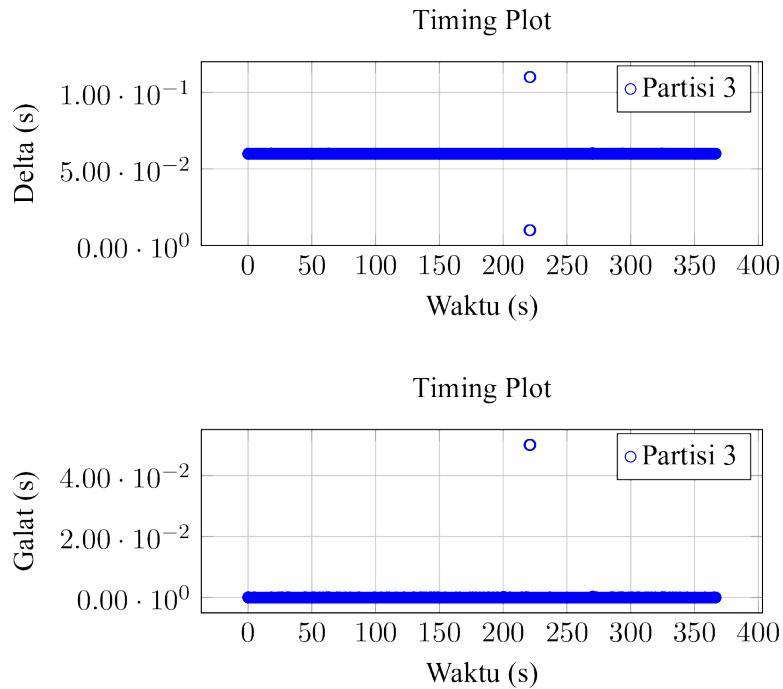
- (a) Pada layanan 1, partisi berjalan dengan stabil. Dari hasil pengujian, dapat terlihat bahwa partisi 4 dapat menggantikan peran partisi 1 ketika mengalami partisi 1 sedang mengalami kegagalan. Dengan demikian, layanan dapat tersedia selama rentang waktu pengujian. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu hampir mencapai 60 ms. Selain itu, terlihat beberapa titik dengan galat yang cukup tinggi berada pada waktu sekitar pergantian partisi *primary* dengan partisi *backup*.



Gambar IV.18. Hasil pengujian keandalan sistem menggunakan skenario 6 (layanan 2)

- (b) Pada layanan 2, partisi berjalan dengan stabil. Dari hasil pengujian, terlihat bahwa partisi 5 dapat menggantikan peran partisi 2 ketika terjadi kegagalan pada partisi 2. Dan apabila partisi 5 juga mengalami kegagalan, partisi 6 dapat menggantikan peran partisi 5. Ketika partisi 2 kembali dari kegagalan, maka partisi 6 tidak lagi berjalan dan digantikan oleh partisi 2. Dengan demikian, layanan 2 selalu tersedia selama pengujian. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu sekitar 60 ms. Selain itu,

terlihat beberapa titik dengan galat yang cukup tinggi berada pada waktu sekitar pergantian partisi *primary* dengan partisi *backup*.



Gambar IV.19. Hasil pengujian keandalan sistem menggunakan skenario 6 (layanan 3)

- (c) Pada layanan 3, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu sekitar 10 ms.

IV.2.2 Pengujian *Latency*

Pada pengujian *latency*, *backup* untuk masing-masing layanan tidak diperlukan karena diasumsikan partisi tidak akan mengalami kegagalan. Dengan demikian, hasil yang ditampilkan hanyalah partisi yang merupakan partisi *primary*.

Pengujian *latency* akan dilakukan dengan menggunakan skenario-skenario berikut.

1. Skenario 1

Pada skenario ini, daftar jadwal berisi 1 layanan. Layanan akan berjalan selama 20 ms dengan *major time frame* keseluruhan selama 20 ms.

Skenario ini akan menguji *latency* layanan apabila sistem hanya memiliki satu buah

layanan. Hasil pengujian akan menjadi referensi *latency* terbaik yang dapat dicapai oleh sebuah layanan pada sistem.

2. Skenario 2

Pada skenario ini, daftar jadwal berisi 4 layanan. Masing-masing layanan akan berjalan selama 10 ms dengan *major time frame* keseluruhan selama 50 ms.

Skenario ini akan menguji *latency* layanan apabila sistem memiliki banyak layanan, namun masih terdapat banyak *resource* yang tersisa. Hasil pengujian akan dianalisa bersamaan dengan skenario 1 dan skenario 3 untuk melihat apakah terdapat kecenderungan tertentu pada kinerja sistem terhadap jumlah layanan pada sistem.

3. Skenario 3

Pada skenario ini, daftar jadwal berisi 9 layanan. Masing-masing akan berjalan selama 20 ms dengan *major time frame* keseluruhan selama 180 ms.

Skenario ini akan menguji *latency* layanan apabila sistem memiliki banyak layanan. Hasil pengujian akan menunjukkan *latency* pada sistem dengan banyak layanan.

Hasil pengujian *latency* sistem berdasarkan skenario-skenario pengujian akan dipaparkan. Pengujian diambil dengan menjalankan program pengujian selama 2 jam. Namun, hasil pengujian hanya akan ditampilkan sampai terlihat sebuah kecenderungan pada data.

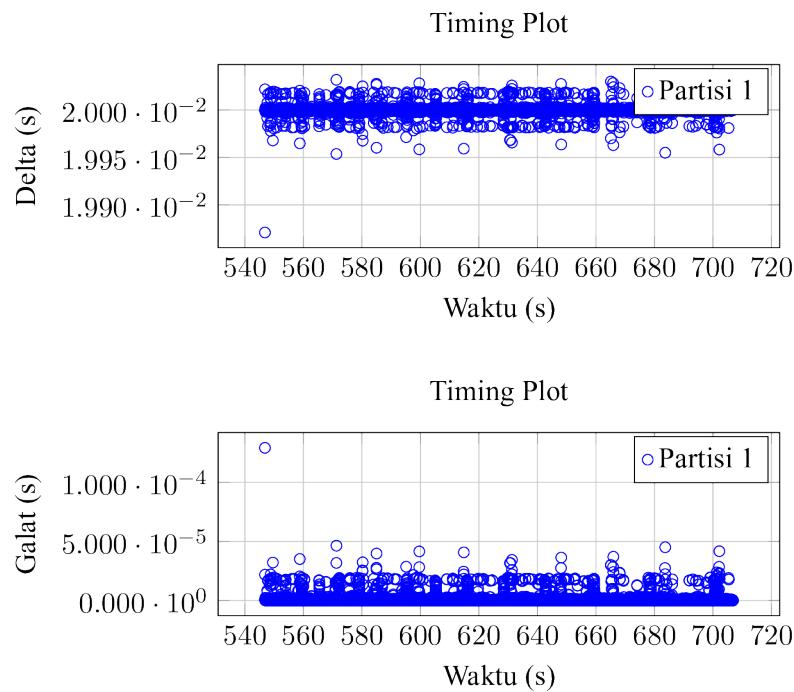
1. Skenario 1

Hasil pengujian *latency* dengan menggunakan skenario 1 dapat dilihat pada Gambar IV.20. Pengujian dengan menggunakan skenario 1 menghasilkan *timing* yang sangat presisi, dengan galat *timing* tertinggi dari ekspektasi periode *timing* yang seharusnya hanya sekitar 100 μ s. Pada pengujian ini, perilaku algoritma *primary-backup partition scheduling* tidak dapat terlihat. Namun, pengujian ini dapat digunakan sebagai referensi *timing* yang seharusnya didapatkan pada pengujian-pengujian berikutnya.

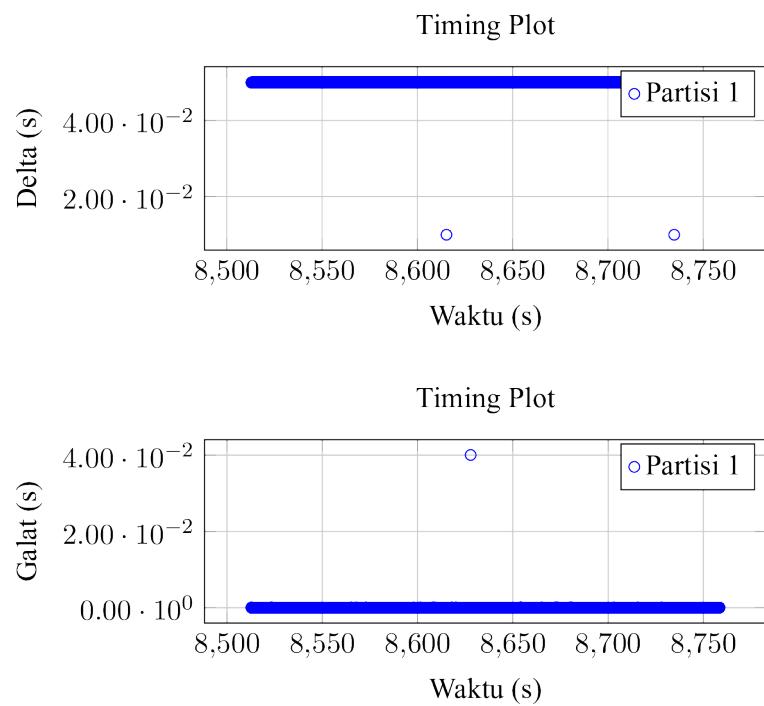
2. Skenario 2

Hasil pengujian *latency* pada skenario 2 dapat dilihat pada Gambar IV.21, Gambar IV.22, Gambar IV.23, dan Gambar IV.24.

- (a) Pada layanan 1, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu sekitar 40 ms.

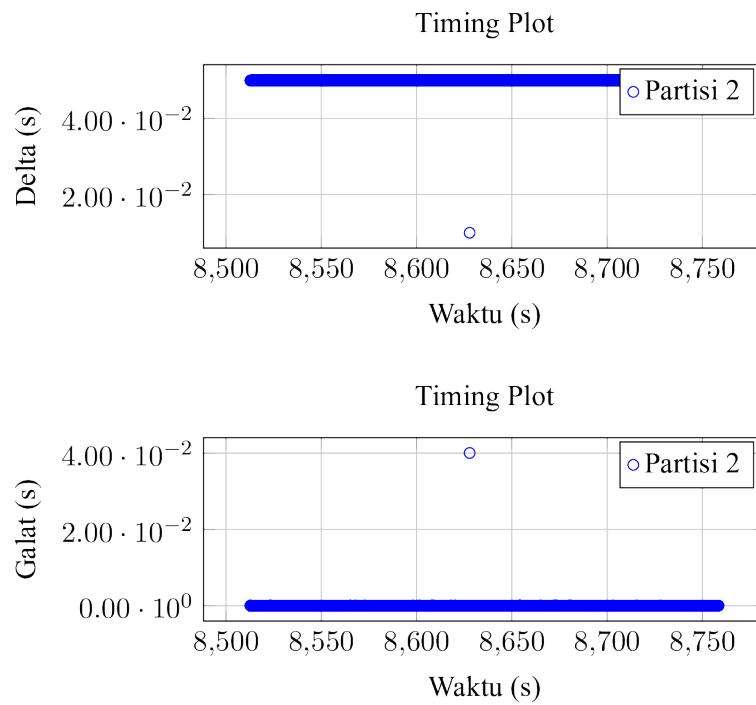


Gambar IV.20. Hasil pengujian *latency* sistem menggunakan skenario 1

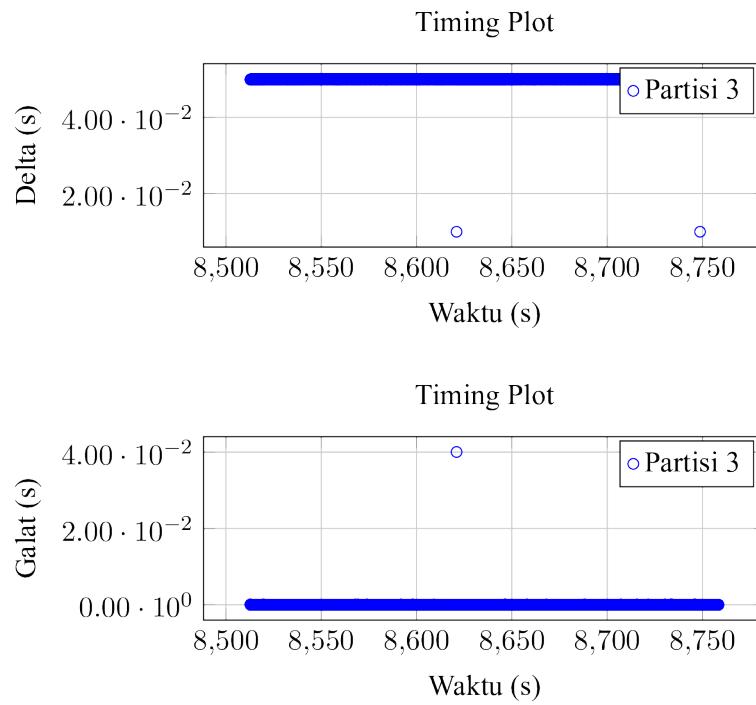


Gambar IV.21. Hasil pengujian *latency* sistem menggunakan skenario 2 (layanan 1)

- (b) Pada layanan 2, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu sekitar 40 ms.

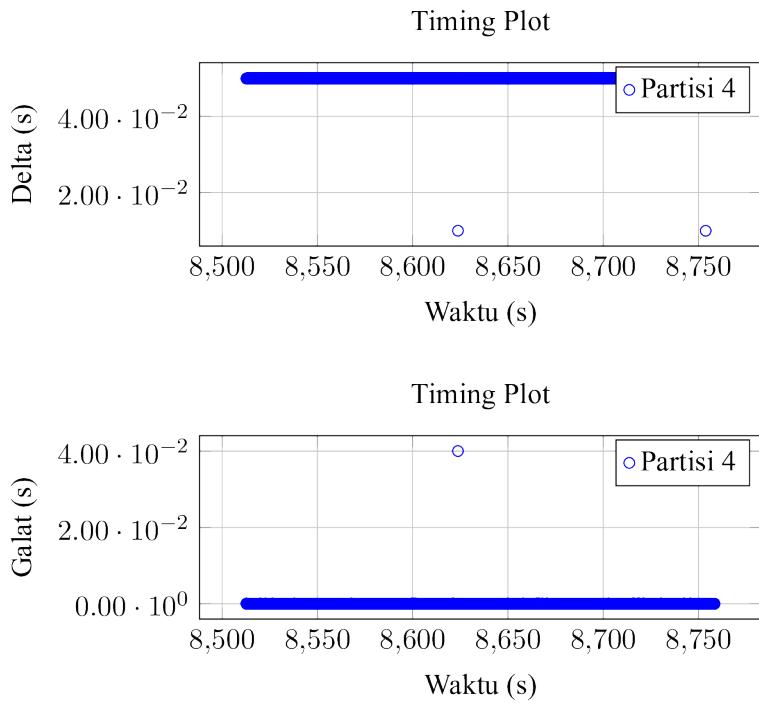


Gambar IV.22. Hasil pengujian *latency* sistem menggunakan skenario 2 (layanan 2)



Gambar IV.23. Hasil pengujian *latency* sistem menggunakan skenario 2 (layanan 3)

- (c) Pada layanan 3, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu sekitar 40 ms.



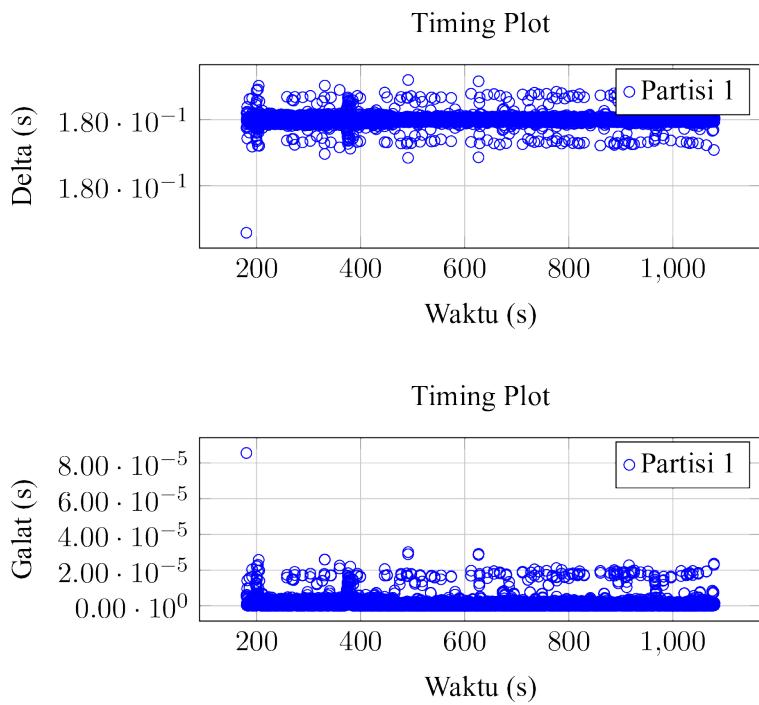
Gambar IV.24. Hasil pengujian *latency* sistem menggunakan skenario 2 (layanan 4)

(d) Pada layanan 4, partisi berjalan dengan stabil. Namun, galat maksimal antara pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu sekitar 40 ms. Hasil pengujian *latency* dengan menggunakan skenario 2 memperlihatkan bahwa *timing* dengan galat tinggi diakibatkan karena *timing* berjalan lebih cepat dari yang seharusnya.

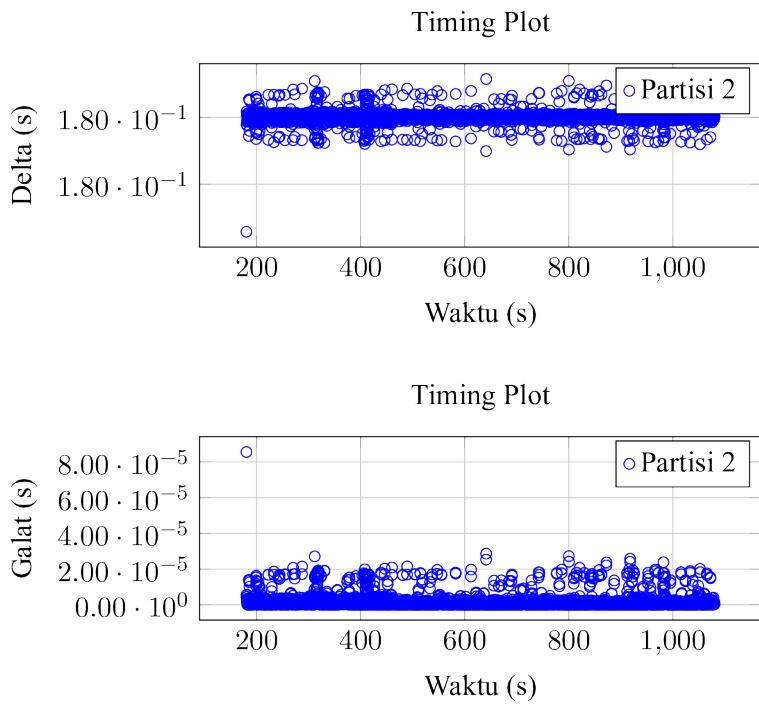
3. Skenario 3

Hasil pengujian *latency* pada skenario 3 dapat dilihat pada Gambar IV.25, Gambar IV.26, Gambar IV.27, Gambar IV.28, Gambar IV.29, Gambar IV.30, Gambar IV.31, Gambar IV.32, dan Gambar IV.33. Hasil untuk masing-masing layanan adalah sebagai berikut.

- (a) Pada layanan 1, partisi berjalan dengan sangat stabil. Namun, galat antara maksimal pengukuran *timing* dengan *timing* ekspektasi cukup tinggi yaitu sekitar 80 μ s.
- (b) Pada layanan 2, partisi berjalan dengan sangat stabil. Namun, galat antara maksimal pengukuran *timing* dengan *timing* ekspektasi cukup tinggi yaitu sekitar 80 μ s.

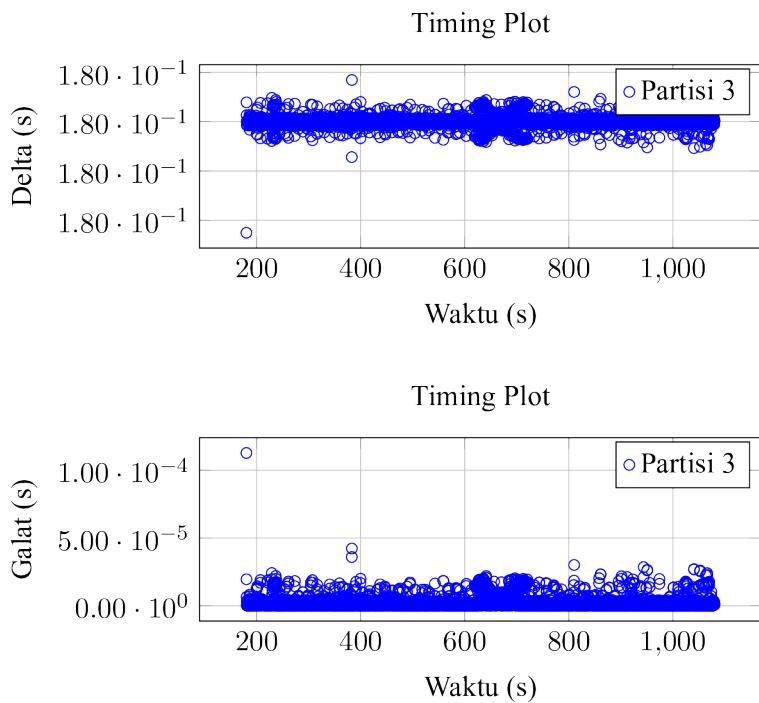


Gambar IV.25. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 1)

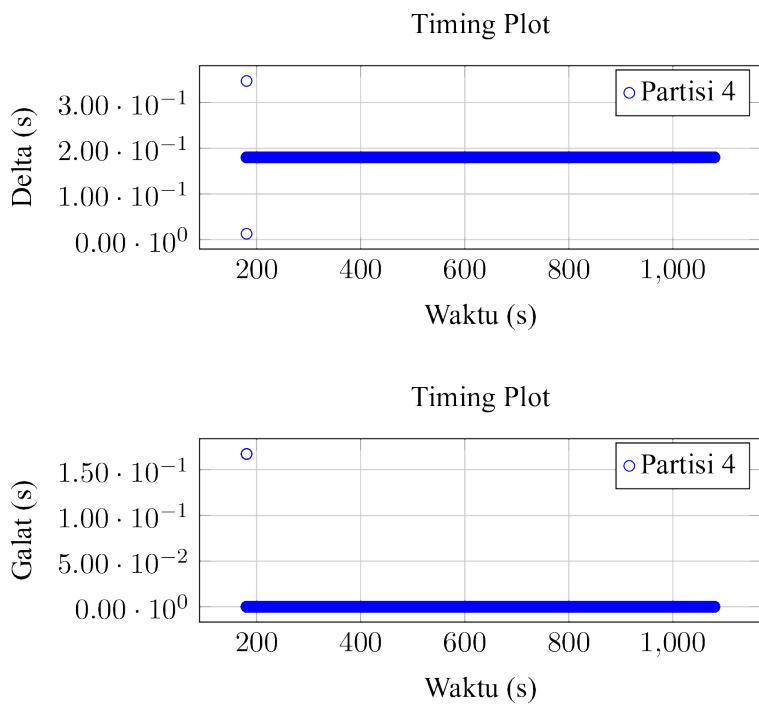


Gambar IV.26. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 2)

- (c) Pada layanan 3, partisi berjalan dengan cukup stabil. Namun, galat maksimal pengukuran *timing* dengan *timing* ekspektasi cukup tinggi yaitu sekitar 100 μ s.

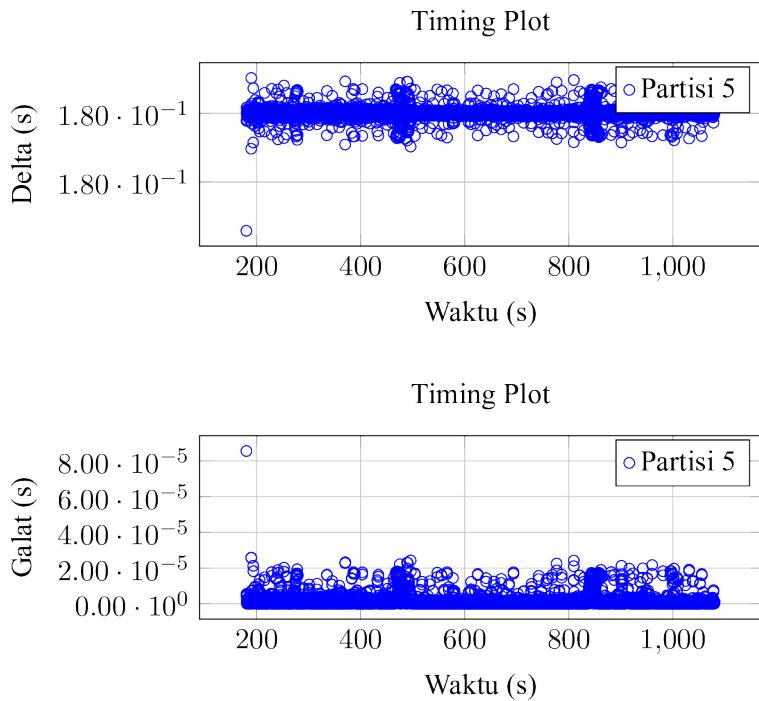


Gambar IV.27. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 3)



Gambar IV.28. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 4)

- (d) Pada layanan 4, partisi berjalan dengan stabil. Namun, galat maksimal pengukuran *timing* dengan *timing* ekspektasi sangat tinggi yaitu sekitar 160 ms.

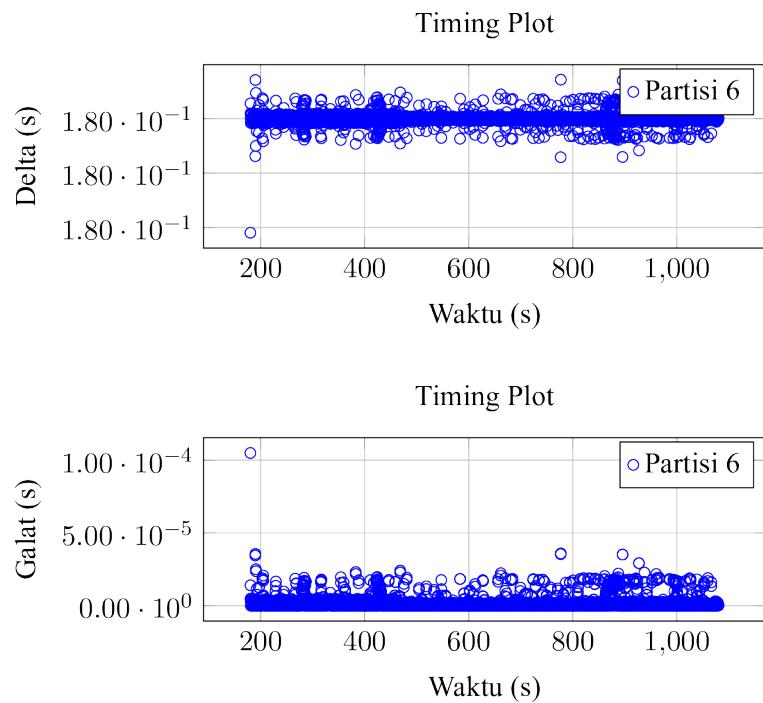


Gambar IV.29. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 5)

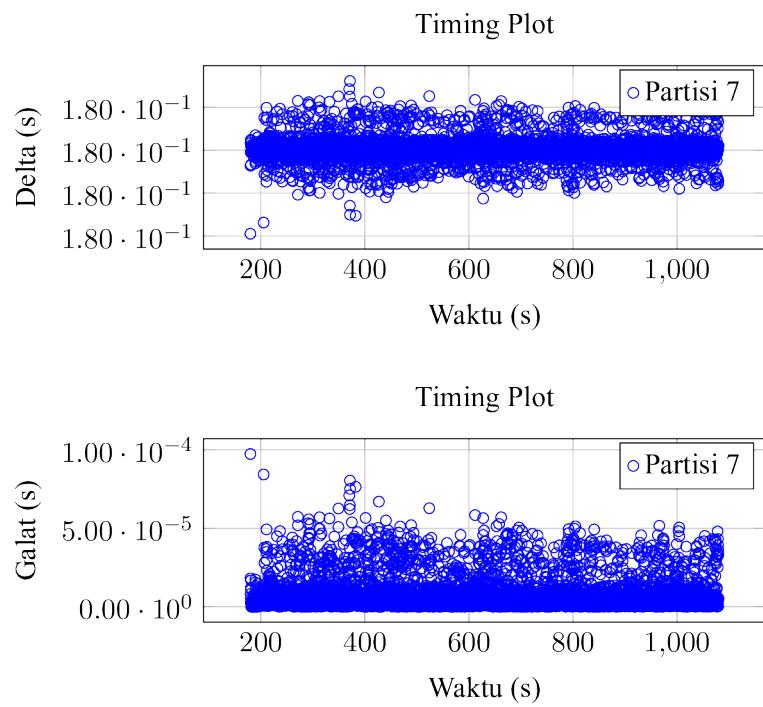
- (e) Pada layanan 5, partisi berjalan dengan sangat stabil. Namun, galat maksimal pengukuran *timing* dengan *timing* ekspektasi cukup tinggi yaitu sekitar 80 μ s.
- (f) Pada layanan 6, partisi berjalan dengan sangat stabil. Namun, galat maksimal pengukuran *timing* dengan *timing* ekspektasi cukup tinggi yaitu sekitar 100 μ s.
- (g) Pada layanan 7, galat maksimal pengukuran *timing* dengan *timing* ekspektasi tidak terlalu tinggi yaitu sekitar 100 μ s. Meski demikian, *timing* yang didapat dari layanan tersebut tidak stabil.
- (h) Pada layanan 8, partisi berjalan dengan sangat stabil. Namun, galat maksimal pengukuran *timing* dengan *timing* ekspektasi cukup tinggi yaitu sekitar 80 μ s.
- (i) Pada layanan 9, galat maksimal pengukuran *timing* dengan *timing* ekspektasi tidak terlalu tinggi yaitu sekitar 100 μ s. Meski demikian, *timing* yang didapat dari layanan tersebut tidak stabil.

IV.2.3 Pembahasan Hasil Pengujian

Primary-backup partition scheduler yang dihasilkan dapat menjadwalkan partisi pengganti apabila partisi *primary* mengalami kegagalan. *Scheduler* juga dapat menjadwalkan partisi

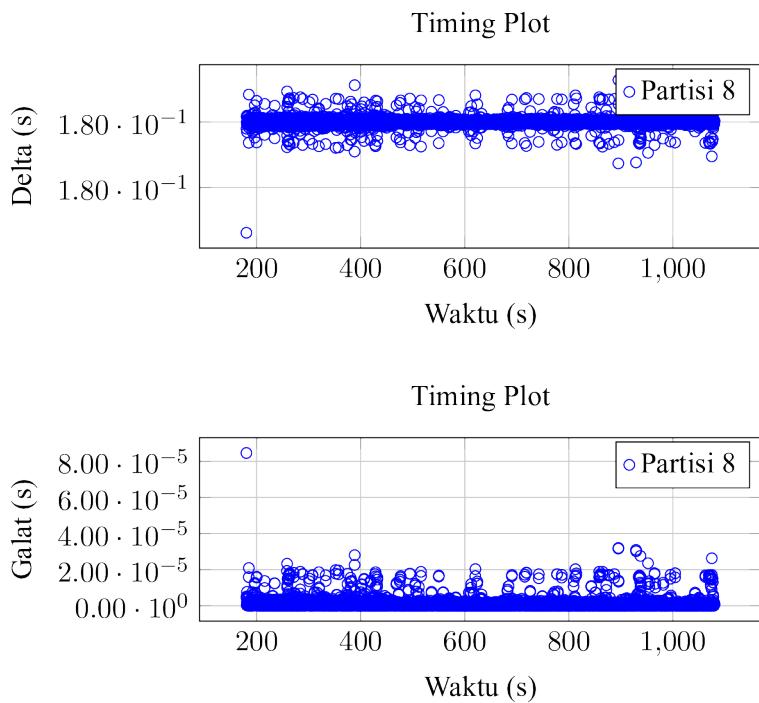


Gambar IV.30. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 6)

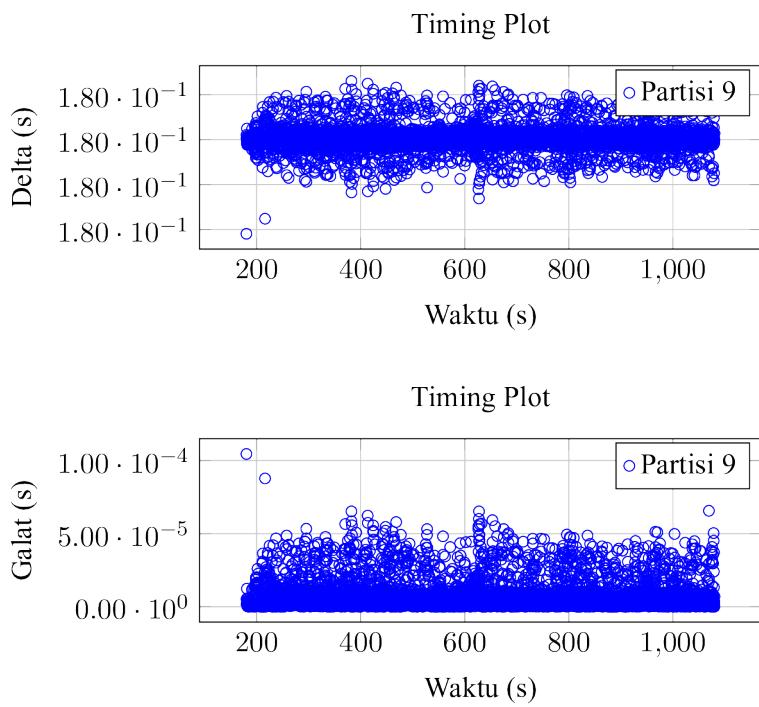


Gambar IV.31. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 7)

pengganti apabila partisi *backup* juga mengalami kegagalan, seperti yang dapat dilihat pada Gambar IV.18.



Gambar IV.32. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 8)



Gambar IV.33. Hasil pengujian *latency* sistem menggunakan skenario 3 (layanan 9)

Primary-backup partition scheduler telah menyelesaikan permasalahan keandalan sistem. Namun, berdasarkan hasil pengujian *latency*, dapat disimpulkan bahwa sistem hasil imple-

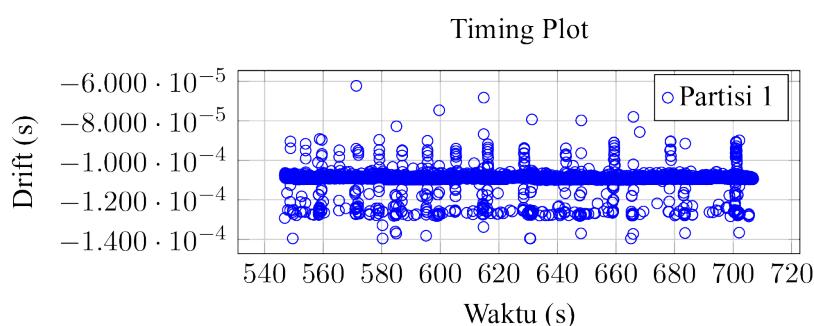
mentasi bukan merupakan sistem *real-time* dikarenakan *latency* maksimal yang sangat tinggi (*latency* dari *timing* hampir mencapai 100%) atau *timing* didapat jauh lebih cepat dari waktu ekspektasi *timing* tersebut didapatkan.

Untuk mencari tahu penyebab *timing* yang didapat pada waktu yang tidak seharusnya, akan dilakukan pengujian *time drift* untuk mengecek apakah terdapat *time drift* ketika *task* periodik berjalan. Pengukuran *time drift* dapat dilakukan dengan menggunakan rumus pada Persamaan IV.3.

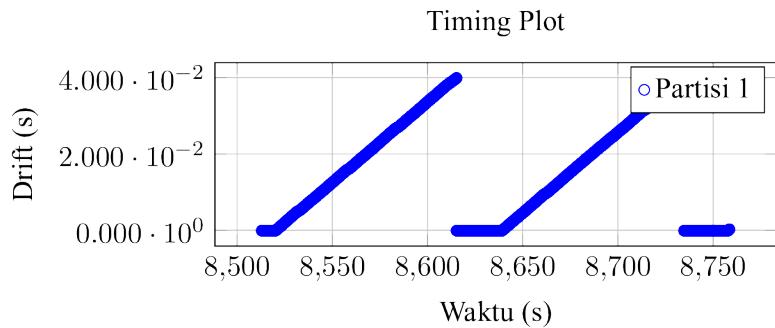
$$drift = now - (start + samples * expected) \quad (\text{IV.3})$$

Pada perhitungan *drift*, nilai *now* adalah waktu pada saat *timing* sekarang didapatkan, nilai *start* adalah waktu pada saat pengujian dimulai, nilai *samples* adalah jumlah sampel yang sudah terkumpul, dan nilai *expected* adalah nilai ekspektasi. Nilai ekspektasi yang digunakan dalam perhitungan *drift* adalah periode dari *timing*. Nilai *drift* akan menunjukkan seberapa jauh penyimpangan waktu *timing* dengan waktu *timing* yang seharusnya.

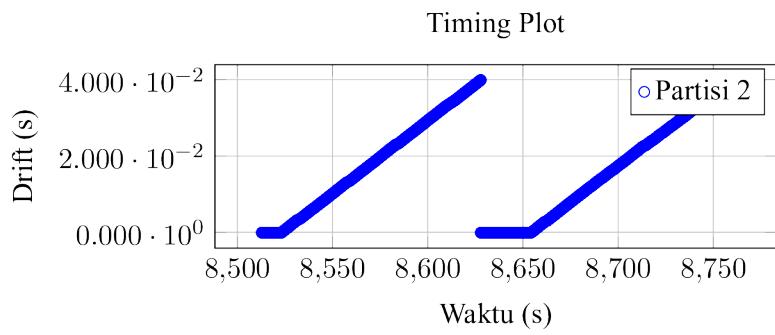
Secara intuitif, terlihat bahwa kesalahan *timing* terjadi apabila terjadi pergantian partisi, termasuk pergantian partisi pada saat pergantian layanan. Oleh karena itu, pengujian *time drift* akan dilakukan dengan melakukan pengujian pada sistem ketika memiliki satu buah partisi yang akan dijadwalkan dan sistem ketika memiliki lebih dari satu buah partisi yang akan dijadwalkan. Hasil pengujian *time drift* pada sistem ketika memiliki satu buah partisi yang akan dijadwalkan dapat dilihat pada Gambar IV.34, sedangkan hasil pengujian *time drift* pada sistem ketika memiliki lebih dari satu buah partisi yang akan dijadwalkan dapat dilihat pada Gambar IV.35, Gambar IV.36, Gambar IV.37, dan Gambar IV.38.



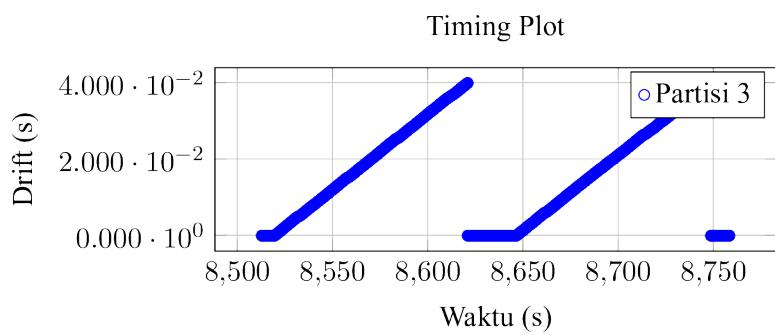
Gambar IV.34. Hasil pengujian *time drift* pada sistem dengan satu buah partisi yang harus dijadwalkan



Gambar IV.35. Hasil pengujian *time drift* pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 1)

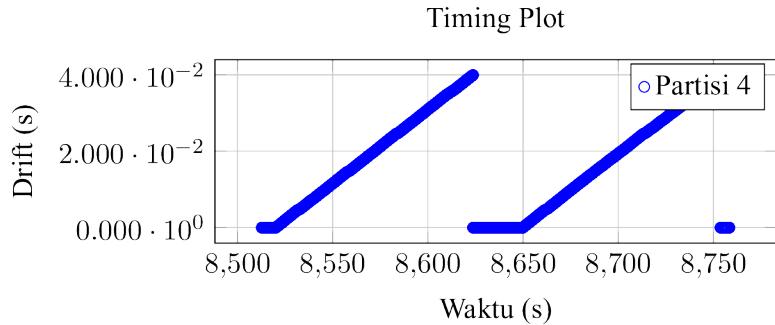


Gambar IV.36. Hasil pengujian *time drift* pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 2)



Gambar IV.37. Hasil pengujian *time drift* pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 3)

Hasil pengujian tersebut menunjukkan bahwa apabila terdapat dua atau lebih partisi yang berbeda, maka akan terdapat *time drift* pada waktu *task* periodik akan dijalankan. Selain



Gambar IV.38. Hasil pengujian *time drift* pada sistem dengan lebih dari satu buah partisi yang harus dijadwalkan (layanan 4)

itu, hasil pengujian tersebut juga menunjukkan bahwa terdapat titik puncak maksimum untuk nilai *time drift* sebelum akhirnya *time drift* kembali menjadi 0. Proses kembalinya nilai *time drift* menjadi 0 hanya dapat terjadi apabila sampel yang didapat sebelum nilai *time drift* kembali menjadi 0 didapat dengan sangat cepat, sehingga jumlah sampel bertambah tanpa terdapat perubahan yang signifikan pada nilai waktu sekarang. Dengan demikian, dapat disimpulkan bahwa *timing* yang didapat terlalu cepat pada pengujian keandalan dan pengujian *latency* diakibatkan karena adanya *time drift* pada *periodic task*. Selain itu, *timing* dengan *latency* terjadi dikarenakan *time drift* tersebut, yang mengakibatkan *task* periodik tidak mendapatkan kesempatan jalan dikarenakan waktu pada saat *task* tersebut mendapatkan giliran di luar dari rentang waktu layanan tersebut.

Timekeeping pada Xen merupakan permasalahan laten pada Xen (Adamczyk dan Chydzinski, 2015) (Broomhead dkk. 2010). Hasil pengujian *time drift* menunjukkan bahwa kesalahan mulai terjadi apabila Xen perlu melakukan *partition switching*. Dalam kasus ini, terdapat beberapa kemungkinan permasalahan yang terjadi. Berikut adalah daftar kemungkinan penyebab terjadinya *time drift* pada saat pengujian.

1. Perbedaan mekanisme *time keeping* pada Linux dan Xen.

Linux memiliki metode yang berbeda dengan Xen untuk melakukan *time keeping*. Mekanisme tersebut ditunjang oleh Xen dengan menggunakan *virtual timer*. Meski demikian, sumber waktu yang digunakan oleh Xen dan Linux merupakan sumber yang berbeda. Perbedaan metode kalibrasi waktu antara Xen dengan Linux dapat mengakibatkan perbedaan waktu sehingga menghasilkan *time drift*. Dengan demikian, pada saat partisi dengan sistem operasi Linux tidak mendapatkan waktu CPU,

kalibrasi waktu CPU yang dilakukan oleh Linux tidak sesuai dengan yang seharusnya.

2. Kinerja *partition switching* pada Xen

Pada saat *partition switching*, Xen akan menyimpan keadaan sebuah partisi sebelum melakukan pergantian. Pada saat Xen akan kembali menjalankan partisi tersebut, Xen perlu mengembalikan keadaan partisi seperti sebelumnya. Hal ini akan mengakibatkan tambahan waktu kerja sehingga pada saat konteks kembali kepada aplikasi, aplikasi sudah melewatkannya ketika *task* periodik seharusnya berjalan.

3. Desain yang tidak sesuai

Xen dan Linux tidak dibangun untuk menunjang aplikasi *real-time*. Dengan demikian, mekanisme internal Xen dan Linux dibangun untuk mendukung aplikasi secara umum. Aplikasi secara umum memilih *throughput* yang tinggi ketimbang *latency* yang rendah. Dengan demikian, mungkin terdapat ketidaksesuaian antara kebutuhan *real-time* aplikasi dengan kapabilitas sistem.

Sayangnya, mekanisme internal Xen dan Linux bukan merupakan lingkup penelitian Tugas Akhir ini, sehingga hasil pengujian tidak dapat digunakan untuk menyimpulkan penyebab permasalahan tersebut.

Untuk mengatasi permasalahan tersebut, pembuat aplikasi *real-time* dapat mendefinisikan keadaan sistem secara global yang akan diacu oleh setiap aplikasi. Setiap aplikasi akan menjalankan fungsionalitas tersebut apabila sudah saatnya fungsionalitas tersebut dijalankan. Periode dari aplikasi secara langsung akan dicapai apabila daftar jadwal yang diberikan pada *scheduler* sesuai dengan batasan waktu dan periode yang seharusnya.

BAB V

Kesimpulan dan Saran

V.1 Kesimpulan

Pada penelitian ini, telah berhasil dikembangkan sebuah *scheduler* yang menggunakan skema *primary-backup* guna meningkatkan keandalan sistem berbasis ARINC 653. Oleh karena *scheduler* menggunakan skema *primary-backup* dan hanya bekerja untuk melakuk-an penjadwalan partisi, maka *scheduler* tersebut disebut sebagai *primary-backup partition scheduler*. *Scheduler* bekerja dengan cara memberikan waktu CPU pada partisi apabila par-tisi tersebut dapat menyediakan layanan yang seharusnya berjalan pada waktu tersebut dan tidak sedang mengalami kegagalan.

Untuk dapat memberitahukan keadaan partisi melalui *user space*, *hypercall* untuk mengu-bah informasi mengenai keadaan partisi juga sudah diimplementasikan. Dengan demikian, kegagalan pada tingkat aplikasi juga dapat diberitahukan kepada *scheduler* untuk kemudi-an ditindaklanjuti. *Hypercall* dapat dilakukan melalui aplikasi dengan menggunakan *libxc* maupun secara manual oleh administrator sistem dengan menggunakan *libxl*.

Primary-backup partition scheduler dapat menangani kegagalan dengan sangat baik. Hasil pengujian keandalan menunjukkan bahwa terdapat 100% *uptime* pada setiap layanan. Hasil pengujian mungkin menaksir nilai *uptime* terlalu tinggi dikarenakan tidak pernah terjadinya kegagalan seluruh partisi penyedia setiap layanan. Selain itu, peningkatan keandalan hanya berlaku untuk kasus dimana *recovery action* yang dispesifikasi oleh ARINC 653 tidak dapat menangani kegagalan yang terjadi. Meski demikian, *primary-backup partition scheduler* tetap meningkatkan keandalan sistem pada kasus tersebut secara signifikan.

Meski hasil pengujian menunjukkan bahwa *primary-backup partition scheduler* mening-katkan keandalan sistem, terlihat bahwa terdapat permasalahan pada hasil *timing* yang dida-pat pada saat pengujian. Kesalahan *timing* sangat signifikan karena pada suatu saat sebuah layanan tidak memberikan pengukuran *timing* pada satu siklus *major time frame*. Hal ini me-nunjukkan bahwa fungsionalitas aplikasi dapat terlewatkhan apabila fungsionalitas tersebut

dilakukan pada selang periode tertentu. Untuk mengatasi permasalahan tersebut, pembuat aplikasi *real-time* dapat mendefinisikan keadaan sistem yang dapat diakses secara global untuk memberitahukan kapan sebuah fungsionalitas harus dijalankan ketimbang menggunakan sistem *timing*.

V.2 Saran

Hasil implementasi *primary-backup partition scheduler* sudah cukup baik. Hal ini diindikasikan oleh tercapainya peningkatan keandalan sistem ketika menggunakan *primary-backup partition scheduler*. Namun, sebelum *scheduler* tersebut dapat digunakan pada lingkungan produksi, terdapat beberapa permasalahan yang harus diselesaikan. Berikut adalah daftar permasalahan yang harus ditangani.

1. *Time drift* pada sistem

Seperti yang sudah dibahas pada Subbab IV.2.3, terdapat permasalahan pengukuran waktu pada sistem. Perlu ada studi lebih lanjut mengenai penyebab perilaku tersebut.

2. Kerangka uji yang lebih kuat

Pengujian yang dilakukan tidak mengikuti sebuah standar tertentu. Untuk menguji *scheduler*, metode yang akan menghasilkan hasil yang lebih menggambarkan kejadian nyata adalah dengan menggunakan pemodelan dan simulasi. Selain itu, penentuan saat kegagalan seharusnya terjadi pada saat pengujian sebaiknya disesuaikan dengan *mean-time-failure* dari layanan. Penggunaan metode tersebut juga dapat menunjukkan nilai optimal untuk beberapa parameter penjadwalan yang sangat penting seperti jumlah partisi yang sebaiknya disediakan untuk sebuah layanan dan waktu CPU yang harus diberikan oleh *hypervisor* untuk sebuah layanan.

Selain itu, perlu adanya pengembangan beberapa fitur sehingga sistem menjadi layak pakai. Berikut adalah daftar fitur yang dapat ditambahkan pada sistem.

1. Minimalisasi penggunaan memori

Memiliki partisi *backup* memerlukan memori tambahan linear terhadap rata-rata *redundancy* yang dibutuhkan agar sistem tetap berjalan sebagaimana seharusnya. Pada umumnya, *mean time to failure* sebuah aplikasi akan jauh lebih lambat ketimbang waktu yang dibutuhkan untuk membuat partisi baru. Dengan demikian, kita dapat

membuat partisi *backup* pada saat dibutuhkan saja. Hal ini mengakibatkan *trade-off* antara penggunaan memori dan performa sistem secara keseluruhan.

2. Pendeksiian partisi gagal otomatis

Untuk memberitahukan pada *scheduler* bahwa sebuah partisi sedang mengalami kegagalan atau tidak sudah dapat dilakukan dengan menggunakan *hypercall*. Namun, hal ini tidak dapat dilakukan apabila aplikasi yang bertugas untuk mendeksi kegagalan pada partisi tersebut juga mengalami kegagalan. Sistem harus dapat mendeksi kasus tersebut secara otomatis sehingga *scheduler* dapat memilih partisi yang tepat untuk menyediakan sebuah layanan.

3. Dukungan untuk *multicore*

Salah satu permasalahan yang mungkin terjadi adalah kegagalan *core* dari sebuah CPU. Dengan memiliki beberapa *core*, maka sistem dapat tetap berjalan meski terdapat kegagalan *core*.

DAFTAR PUSTAKA

- Adamczyk, Blazej dan Andrzej Chydzinski (2015). "Achieving high resolution timer events in virtualized environment." Dalam: *PLoS ONE* 10.7, hal. 1–25. ISSN: 19326203. DOI: 10.1371/journal.pone.0130887.
- Airlines Electronic Engineering Committee (2012). "653P1-3 Avionics Application Software Standard Interface Part 1 - Required Services." Dalam: *Arinc 653*, hal. 85.
- Bertossi, Alan A. dkk. (2006). "Scheduling hard-real-time tasks with backup phasing delay." Dalam: *Proceedings - IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT*, hal. 107–116. ISBN: 0769526977. DOI: 10.1109/DS-RT.2006.33.
- Broomhead, Timothy dkk. (2010). "Virtualize everything but time." Dalam: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, hal. 1–6. URL: http://www.usenix.org/events/osdi/tech/full%7B%5C_%7Dpapers/Broomhead.pdf.
- Chisnall, David (2014). *The Definitive Guide to the Xen Hypervisor*. Vol. 25. 9, hal. 307. ISBN: 9780874216561. DOI: 10.1007/s13398-014-0173-7.2. URL: <http://arxiv.org/abs/1011.1669%20http://dx.doi.org/10.1088/1751-8113/44/8/085201%20http://stacks.iop.org/1751-8121/44/i=8/a=085201?key=crossref.abc74c979a75846b3de48a5587bf708f%20http://www.ncbi.nlm.nih.gov/pubmed/15003161%20http://www.ncbi.nlm.nih.gov/pubmed/150>.
- Garside, Richard dan F. Joe Pighetti (2009). "Integrating modular avionics: A new role emerges." Dalam: *IEEE Aerospace and Electronic Systems Magazine* 24.3, hal. 31–34. ISSN: 08858985. DOI: 10.1109/MAES.2009.4811086.
- Han, Ching Chih dkk. (2003). "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults." Dalam: *IEEE Transactions on Computers* 52.3, hal. 362–372. ISSN: 00189340. DOI: 10.1109/TC.2003.1183950.
- Haritsa, Jayant R. dkk. (1992). "Data access scheduling in firm real-time database systems." Dalam: *Real-Time Systems* 4.3, hal. 203–241. ISSN: 09226443. DOI: 10.1007/BF00365312.

- Hyun, Jongsoo dan Kyong Hoon Kim (2012). “Fault-tolerant scheduling in hierarchical real-time scheduling framework.” Dalam: *Proceedings - 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2012 - 2nd Workshop on Cyber-Physical Systems, Networks, and Applications, CPSNA* i, hal. 431–436. DOI: 10.1109/RTCSA.2012.45.
- Institute of Electrical and Electronics Engineers (2016). *IEEE Std 1003.1™-2008 and The Open Group Technical Standard Base Specifications, Issue 7*. URL: <http://pubs.opengroup.org/onlinepubs/9699919799/>.
- Jin, Hyun-Wook (2013). “Fault-tolerant hierarchical real-time scheduling with backup partitions on single processor.” Dalam: *ACM SIGBED Review* 10.4, hal. 25–28. ISSN: 15513688. DOI: 10.1145/2583687.2583693. URL: <http://dl.acm.org/citation.cfm?doid=2583687.2583693>.
- Kinnan, Larry M. (2009). *Linux, ARINC 653 and OpenGL Team Up for Avionics Prototyping*. URL: http://archive.cotsjournalonline.com/articles/print_article/100736.
- Liestman, Arthur L. dan Roy H. Campbell (1986). “A Fault-Tolerant Scheduling Problem.” Dalam: *IEEE Transactions on Software Engineering* SE-12.11, hal. 1089–1095. ISSN: 00985589. DOI: 10.1109/TSE.1986.6312999.
- Oh, Yingfeng dan Sang H Son (1994). “Enhancing Fault-Tolerance in Rate-Monotonic Scheduling.” Dalam: 329, hal. 315–329.
- Rushby, John (2000). “Partitioning in avionics architectures: requirements, mechanisms and assurance.” Dalam: *Work March*, hal. 67. DOI: DOT/FAA/AR-99/58.
- Shin, Insik dan Insup Lee (2008). “Compositional real-time scheduling framework with periodic model.” Dalam: *ACM Transactions on Embedded Computing Systems* 7.3, hal. 1–39. ISSN: 15399087. DOI: 10.1145/1347375.1347383. URL: <http://portal.acm.org/citation.cfm?doid=1347375.1347383>.
- Shin, Kang G. dan Parameswaran Ramanathan (1994). “Real-Time Computing: A New Discipline of Computer Science and Engineering.” Dalam: *Proceedings of the IEEE* 82.1, hal. 6–24. ISSN: 15582256. DOI: 10.1109/5.259423.
- The Linux Foundation (2017). *The Real Time Linux collaborative project*. URL: <https://wiki.linuxfoundation.org/realtime/documentation>.

VanderLeest, Steven H. (2010). "ARINC 653 hypervisor." Dalam: *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*. ISBN: 9781424466160. DOI: 10.1109/DASC.2010.5655298.