情報処理学会論文誌　コンピューティングシステム　Vol.7　No.2　11–24 (May 2014)

プロセスの耐障害性向上のための多重OSの開発と評価

吉田 健二[1,a)]　齋藤 彰一[1]　毛利 公一[2,b)]　松尾 啓志[1]

受付日 2013年10月17日, 採録日 2014年1月30日

概要：オペレーティングシステム（OS）には高い耐障害性が求められる．しかし，耐障害性を向上させる既存手法は，専用ハードウェアや大きな実行時オーバヘッドが随伴するという問題がある．我々はOSを計算機上に複数動作させてアクティブ/バックアップ構成を組み，プロセスとファイルキャッシュを保護する耐障害性向上手法を提案する．本提案手法では，保護するデータは障害発生後に取得することで事前の実行状態保存による実行時オーバヘッドをゼロに抑えることができる．また，提案手法を実現するために必要となるリソースはCPU 1コアと少量のメモリ領域のみである．本提案手法を実装した結果，リカバリの時間は最短で0.4秒，2 GB程度のデータの復元が必要となった場合でも10秒程度であることを確認した．また，テキストエディタ，NFSサーバ，データベースサーバ，HTTPサーバで障害を発生させた場合の停止時間は最長1.5秒であった．

キーワード：リカバリ，オペレーティングシステム，プロセスマイグレーション，ファイルキャッシュマイグレーション

Development and Evaluation of Multiple Operating System for Process Reliability

Kenji Yoshida[1,a)]　Shoichi Saito[1]　Koichi Mouri[2,b)]　Hiroshi Matsuo[1]

Received: October 17, 2013, Accepted: January 30, 2014

Abstract: Operating Systems (OS) require high reliability. However, existing methods of fault tolerance have problems that they have a large run-time overhead or a dedicated hardware. We propose a method to recover from OS failures that preserves processes and file caches. This method runs two OSes and configures them as an active-backup structure in one computer. By obtaining data after a crash, the active OS does not have run-time overhead to backup a process execution status and file cache data. In addition, the resources consumed to build the active-backup structure are only one CPU core and a small amount of memory. In the implementation, we confirmed that recovery time when using the proposed method is about 0.4 seconds at a minimum and up to about 10 seconds even if 2 GB memory is restored. The downtime was up to about 1.5 seconds when the active OS of the proposed system crashed while running a text editor, an NFS server, a database server, and an HTTP server.

Keywords: recovery, operating system, process migration, file cache migration

1. はじめに

OSが安定して動作することは非常に重要である．なぜなら，OSは計算機上の処理の正常な実行に責任を持つからである．しかし，OSには必ずバグが存在し，機能追加や修正を行うたびにバグは増加する [1], [2]．バグの要因をすべて消し去ることは不可能であることから，障害発生を前提とした耐障害性を持ったOSが求められる．

障害が発生しOSが停止した場合，一般的には計算機のリブートによってリカバリが行われる．しかし，リブートは長時間計算機が使用不可能になることで計算機の可用性が低下するという問題点を持つ．さらに，リブートは動作中のプロセスや未保存のファイルキャッシュ，ネットワー

[1] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi 466–8555, Japan
[2] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[a)] orthros@mail.ssn.nitech.ac.jp
[b)] mouri@cs.ritsumei.ac.jp

クコネクションといった計算機の実行状態を破壊する．特に，リブートによってネットワークコネクションが切断された場合，リカバリは通信相手との協調動作を必要とする．このため，リブートによって発生する損害は接続中であった他の計算機へと及び，そのリカバリ時間は長くなる．我々は，計算機のリカバリ時間を削減し，実行状態を保護することによって計算機の耐障害性を高める手法を提案する．

耐障害性を高める既存の方式として，High-Availability Cluster（HA Cluster）が存在する [3]．しかし，HA Cluster は計算機を構成するためのコストが高いという欠点がある．さらに，つねにノード間のデータ同期を行うための実行時オーバヘッドが大きい．また，計算機の実行状態を保護する既存手法としてチェックポイント/リスタート（C/R）が存在する [4], [5]．この手法はチェックポインティングの実行時オーバヘッドが大きいという問題がある．このオーバヘッドを減少させるためにはチェックポインティングの実行間隔を広げればよいが，その際にはリスタートによる実行状態の巻き戻しが大きくなるという問題がある．

以上から，我々は耐障害性 OS に求められる要件を以下のように設定した．

- 最小限の計算機構成
- 最小限の性能低下
- 高速なリカバリ
- プロセスとファイルキャッシュの保護

最小限の計算機構成とは，HA Cluster 等と比較して複数の計算機および特殊なハードウェアを用意する必要がないという要件である．この要件を実現した耐障害性向上手法は導入コストが下がり，より広範囲の実システムへと適用可能である．たとえば，汎用計算機 1 台のみで動作しているサーバへの適用が追加投資なしで可能となる．次に，最小限の性能低下とは，導入コストを抑えたうえで実行時のオーバヘッドを最小限とする要件である．そして，高速なリカバリおよびプロセスとファイルキャッシュの保護は，質の向上を目指す耐障害性の性質を示す要件である．

設定した要件の実現を目指す既存手法に，障害発生時のメモリイメージを保存して活用する手法 [6] と 1 台の計算機上で OS を複数動作させる手法 [7] がある．Otherworld [6] では障害発生時に Warm-boot を用いて新たな OS を起動させる．Warm-boot を用いて起動した OS は，障害が発生した OS のメモリイメージを破壊することなしにメモリ上に残す．これによって障害が発生した OS のメモリイメージを走査することでプロセスの実行状態を復元する．しかし，Warm-boot は OS の起動処理をともなうため通常高速ではない．また，Otherworld はファイルキャッシュの破損を考慮しないため，プロセスが扱うファイルはリカバリ後に信頼性が失われる．さらに，Otherworld はネットワークコネクションを保護しないため，リカバリ時に再接続処理が必要となる．次に，Shimos2 [7] は Software Logical Partitioning（Software LPAR）[8] を採用することで OS を CPU のコアごとに動作させて冗長性を確保する．さらに，C/R を併用することによってプロセスの実行状態保護が可能である．しかし，C/R の使用はその実行時オーバヘッドによる性能低下を引き起こすという問題がある．また，C/R は必ずしも最新の実行状態を保護することができないという問題もある．

我々は，既存手法の利点と欠点を検討した結果，それぞれが補い合えることに着目し，新たな耐障害性向上手法 Orthros（ORganized Transmigratory High-Reliability OS）を提案する．Otherworld の問題である低速なリカバリは，Shimos2 と同様の Software LPAR を用いた複数 OS 同時実行によって解決する．Software LPAR は仮想化機構を経由せずに実計算機上で直接命令を実行できるため仮想化オーバヘッドが存在しない．さらに，HA Cluster のように冗長なハードウェア構成をとる必要がない．一方，Shimos2 の C/R のオーバヘッドは，Otherworld と同様に停止した OS のメモリイメージから必要なプロセスの実行状態を取得することで解決する．以上により，提案手法は目標要件である「最小限の計算機構成」と「最小限の性能低下」を達成する．つまり提案手法は，Software LPAR を用いた複数 OS 構成によるウォームスタンバイと，メモリイメージ走査による C/R 不要のプロセスの実行状態取得を用いたフェイルオーバ手法である．

提案手法は，プロセスおよびファイルキャッシュ保護と高速なフェイルオーバを，ハードウェア要求と大きな性能低下なしに実現可能である．また，Orthros が設定した 4 つの要件のそれぞれに特化した手法には性能で劣るが，高い水準で同時に実現する初めての手法である．具体的には，障害発生時の停止時間は最短で約 0.4 秒であり，要するリソースは CPU の 1 コアと 512 MB のメモリの専有のみである．さらに，要求ハードウェアはマルチコアの CPU と OS ごとのディスクのみである．以上より，本稿による貢献をまとめると以下の各手法の確立となる．

- カーネルの異常停止からの高速なリカバリ手法
- リカバリのための実行時負荷の軽減手法
- 異常停止した OS のイメージ走査によるプロセスマイグレーション手法

本稿の構成は次のとおりである．まず 2 章では提案手法の概要と利点について述べる．3 章でシステムの設計と実装を説明する．そして 4 章で評価を行い，5 章で実際のアプリケーションへの適用を行う．6 章では関連研究について述べ，比較を行う．最後に，7 章でまとめと今後の課題について述べる．

2. Orthros

本章では，Orthros の目的と概要について述べる．OS に障害が発生した際の計算機の停止時間は，管理者が障害に

気付くまでの時間と計算機をリブートしてプロセスを新たに起動するまでの時間の合計である．この停止時間が長いほど，システム管理者とシステム利用者が被る損害は大きくなる．また，OS を新たに起動することで，プロセスが保存していない一時データが損失するという問題も生じる．Orthros はこれら 2 つの問題を解決する手法である．

2.1　対象とする障害

Orthros が対象とする障害は，バグによる OS の停止であり，ユーザ空間で発生する障害に対してプロセスの保護は行わない．OS を停止させるバグには，スケジューラおよび排他制御のデッドロックや Null Pointer Dereference 等様々なものがある．これらは特にデバイスドライバに多く分布する [9]．これらのバグによる障害からプロセスおよびファイルキャッシュを保護することで，システムの可用性を向上させる．

Orthros は，OS のすべてのバグに対してプロセスを保護できるわけではない．Orthros はメモリ上に残されたデータを用いてプロセスとファイルキャッシュを保護するため，これらのカーネルデータを直接取り扱う処理に障害が発生しデータが破壊された場合には保護することができない．しかし，これら以外の処理で発生した障害からは保護が可能である．なぜなら，カーネル内のデータを処理する関数は各所で変数の値をチェックすることで誤った値の伝播を防いでいるからである [10]．このため，プロセスやファイルキャッシュに関係のない処理でこれらのデータが破壊されることは少ない．なお，発生した障害が上記のどちらの場合に属するかを Orthros 自身が判定することは今後の課題である．

また，Orthros はハードウェアの障害にも対処することができない．しかし，ディスクの故障に対しては RAID 構成等の別の対処方法が存在し，停電に対しては無停電電源装置が存在する．これらの対策を併用することで，ハードウェアの障害に対する耐障害性を向上させることは可能である．

2.2　構成

システム構成を**図 1** に示す．高速なリカバリを行うために，Orthros は 2 つの OS を用意してそれらをアクティブ/バックアップ構成で運用する．ActiveOS は主だった仕事のすべてを処理する OS であり，BackupOS は ActiveOS に障害が発生したときに使用される予備の OS である．計算機構成を最小限で済ませるために，この 2 つの OS は 1 台の計算機上で同時動作する．この際，BackupOS は通常時は何も仕事をせず，割り当てた計算機のリソースが無駄になる．したがって，通常実行時には ActiveOS にほぼすべてのリソースを割り当て，BackupOS には動作するために必要最低限の CPU とメモリのみを割り当てる．

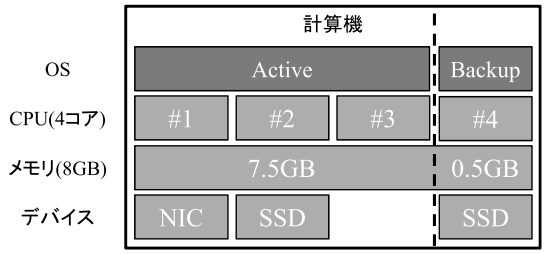

図 1　提案手法によるシステム分割例
Fig. 1　Example of proposed system.

BackupOS を動作させる際の性能低下を最小にするために，OS の同時実行には LPAR を利用する．LPAR は Xen [11] や KVM [12] のような仮想化システムと異なり仮想化オーバヘッドがほとんど存在しない．BackupOS 導入による ActiveOS のリソース減少は CPU の 1 コアと少量のメモリであるが，消費するリソースは計算機のメニーコア化と使用可能メモリの増大によって問題にならないと考える．また，LPAR をソフトウェア実装することで，BackupOS 導入のためのハードウェア要件はマルチコアプロセッサのみとなる．これにより，提案手法の導入に要するハードウェアの入手は容易であり，導入コストは低い．

2.3　動作概要

管理者が障害に気付くまでの時間を削減するために，BackupOS は ActiveOS に障害が発生していないかをつねに監視する．そして，障害を検知した場合には自動的にフェイルオーバを開始し，システム管理者によって指定された ActiveOS のプロセスおよびファイルキャッシュを保護する．プロセスの保護はプロセスマイグレーションで，ファイルキャッシュの保護はファイルキャッシュマイグレーションによって実現する．

プロセスマイグレーションを行うために C/R を用いると，通常実行時のオーバヘッドが大きい．このオーバヘッドを発生させないために，Orthros は C/R を用いずに，ActiveOS のメモリをマイグレーションしイメージを読み取ることで実行状態を取得する．そして同じ実行状態を持つプロセスを BackupOS 上に生成することでプロセスマイグレーションを実現する．ファイルキャッシュの保護も同様に，ActiveOS のメモリをマイグレーションして読み取ることで行う．ファイルキャッシュを保護することで，ファイルのデータに動作が依存するプロセスについても正常な動作を継続することが可能となる．さらに，ファイルキャッシュと同様に ActiveOS が用いていた NIC およびネットワーク設定も BackupOS 上にマイグレーションする．そして同時にプロセスが保持するソケットをマイグレーションすることで，Orthros はネットワークを隔てて通信している相手に対して透過的なリカバリを可能にする．

3. 設計と実装

本章では，まずフェイルオーバ処理のために各 OS が行う処理の概要を述べる．次に，ActiveOS が行う事前処理と，BackupOS が行う ActiveOS の監視処理，ファイルキャッシュマイグレーションとプロセスマイグレーションの実装について詳しく述べる．実装は Linux（version 2.6.38, processor type x86_64）に行った．

3.1 処理概要

ActiveOS はフェイルオーバ処理のために，以下の事項を準備として行う．

- BackupOS が用いるハードウェアの予約と BackupOS の起動
- 保護するプロセスを専用の方法で起動
- フェイルオーバに必要なデータを BackupOS に通知

また，ActiveOS と BackupOS は協調して，ActiveOS が正常な動作にあるかをつねに確認する．これによって，BackupOS は ActiveOS に異常が発生した場合に即座にフェイルオーバを開始できる．

フェイルオーバ処理の流れを**図 2** に示す．BackupOS は異常を検知すると，ActiveOS に Non-Maskable Interrupt（NMI）を発行しレジスタの保存を促す．次に，メモリ領域を BackupOS が読み取れるように拡張し，さらに SSD デバイスをマイグレーションして BackupOS から利用可能としてファイルシステムをマウントする．その後，ファイルキャッシュマイグレーションを行う．また，ネットワーク関係では，NIC デバイスをマイグレーションして利用可能にし，各種ネットワークの設定を ActiveOS と同様になるように行う．最後に，プロセスマイグレーションを実施する．以降の節では，ActiveOS の 3 種の準備処理と，BackupOS が行うファイルキャッシュマイグレーションとプロセスマイグレーションについて述べる．また，プロセスマイグレーションによるプロセスへのエラー伝播とそのエラーハンドリング方法について述べる．

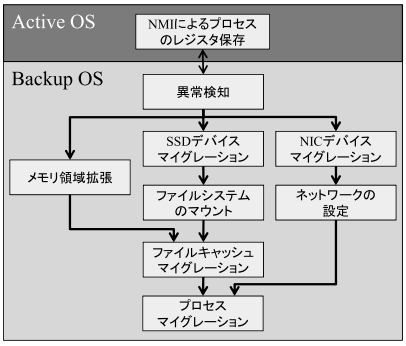

図 2　フェイルオーバ処理の流れ
Fig. 2　Failover flowchart.

3.2 ActiveOS の準備処理

ActiveOS は障害時にフェイルオーバを行えるようにするために，3 種の準備処理を行う．これらの動作は 1 度限り行われるため，実行時オーバヘッドにはならない．

3.2.1 OS の起動と保護

Orthros の Software LPAR の実装には，SHIMOS [8] および Mint [13] を参考にした．OS の起動は，まず最初に ActiveOS を起動し，その後 ActiveOS が BackupOS を起動することにより行う．この起動処理中のハードウェア初期化処理を改変することにより，CPU コアとメモリおよびデバイスの排他的な利用と OS 間の保護を実現する．一方の OS は起動時に他方の OS が用いる CPU コアを使用しないように起動オプションで指定する．この際，割込みが入る CPU コアを一意に識別するために，両 OS による論理 APICID の CPU コアへの割当てが重複しないように変更する．さらに，IO-APIC に関しても，他方の OS が書き込んだ割込み設定を上書きしないようにする．また，ActiveOS は BackupOS の物理メモリ領域を OS 起動時のみしかアクセスせず，起動後は読み出し専用として設定する．BackupOS は，起動オプションに memmap オプションを用いることで ActiveOS の物理メモリ領域を不使用のメモリとして設定し，BackupOS の通常動作では利用不可能にする．ディスクや NIC 等のデバイスは，他方の OS が使用するものに対して初期化処理を行わないことで制御対象から除外する．以上により，ActiveOS と BackupOS 間の排他的なデバイス利用を実現し，一方の障害の他方への波及を防止している．

しかし，不使用にされた ActiveOS の物理メモリ領域とデバイスは，フェイルオーバ時に BackupOS で利用可能にする必要がある．具体的には，ファイルキャッシュとプロセスをマイグレーションする際に，BackupOS は ActiveOS の物理メモリ領域を読み出せなければならない．また，SSD と NIC は，ActiveOS と同等のプロセスとファイルキャッシュ利用環境を整えるためにデバイスマイグレーションを行わなければならない．そのため，BackupOS が ActiveOS のハードウェアを使用しない設定は，後から適切な処理により変更できなければならない．具体的には，BackupOS が起動時に用いる memmap オプションを用いて指定した物理メモリ領域は，BackupOS に使用しないメモリとして認識されるが，ページテーブルの操作により BackupOS から読み出せるように変更できる．そしてファイルキャッシュとプロセスのマイグレーション時に，このページテーブル操作処理を行い，ActiveOS の物理メモリ領域を読み出せるようにする．次に，BackupOS は ActiveOS の起動時にデバイスの初期化処理を行わないことでデバイスの排他的利用を実現するが，同時に初期化処理を後から実行できるように初期化パラメータを保存する．これにより BackupOS は，デバイスマイグレーション時に，保存され

た初期化パラメータを用いて初期化処理を行うことでデバイスを利用可能となる．なお，マイグレーション処理のためのページテーブルの操作とデバイス初期化には複雑な手順が必要であり，OS の異常により偶発的に発生する可能性はきわめて低い．したがって，これらマイグレーションのための各処理が OS 間保護を劣化させることはない．

3.2.2　保護プロセス群の起動と指定

保護プロセス群は，pid-namespace を作成してその内部で起動する．pid-namespace の利用により，フェイルオーバ後の BackupOS 内でのプロセス ID の重複を防止できる．また，保護すべきプロセス群を ActiveOS 上で一意に指定する手段としても，pid-namespace 機能を利用する．これにより，システム管理者は pid-namespace を用いて保護するプロセス群を ActiveOS に指定できる．また，BackupOS も，指定された pid-namespace に属するプロセスが保護対象であることを容易に識別できる．

3.2.3　フェイルオーバに必要なデータの送信

システム管理者は，保護すべき pid-namespace のトップとなる init プロセスとファイルキャッシュを保護したいパーティションを，新規に作成したシステムコールにより ActiveOS に通知する．これにより，ActiveOS は BackupOS へ**表 1** で示されるデータを引き渡す．これらのデータは ActiveOS と BackupOS の共有メモリを介することで引き渡される．そして BackupOS は，これらのデータが壊れていないことを前提としてフェイルオーバ時にデータを読み出す．なお，この共有メモリはデータの書き込み中のみ ActiveOS から書き込み可能とし，その後は読み出し専用の領域として ActiveOS の異常動作から保護する．

3.3　死活監視

BackupOS は起動後に Inter-Processor Interrupt（IPI）による ActiveOS の死活監視を行う．使用する IPI は 2 種類あり，ActiveOS に障害が発生したことを伝える異常通知と正常に動作していることを伝える生存通知である．ActiveOS が自らの障害を検知しカーネルパニックが発生した場合，`panic()` 関数の中で BackupOS に対して異常通知を送信する．しかし，ActiveOS のコアがデッドロック等で停止した場合は `panic()` 関数が呼ばれないため異常通知を送ることができない．このため，ActiveOS は各コアから定期的に BackupOS に対して生存通知を表す IPI を送信する．BackupOS は一定時間生存通知を受信しない場合に ActiveOS が異常停止したと認識する．この生存通知の間隔は短いほど異常検知が素早く行われるが，割込みの増加は好ましくない．現在の設定では 0.1 秒ごとに送信しており，この間隔での割込みオーバヘッドは十分小さいと考える．以上の死活監視によって障害が検出された場合に，BackupOS はフェイルオーバを開始する．

表 1　BackupOS へ渡されるデータ
Table 1　Data passed to BackupOS.

アドレス	目的
保護するプロセスの `task_struct`	プロセスを構成するほとんどの情報
ランキュー	カレントプロセスの識別
CPU コア固有領域	割込みスタックにあるレジスタ情報
保護するパーティションの `super_block`	dirty な inode 番号およびそのページ

3.4　ファイルキャッシュマイグレーション

障害発生時まで ActiveOS が使用していたファイルシステムを BackupOS が使用できるようにするために，BackupOS はまず ActiveOS が使用していたディスクの初期化を再開してデバイスマイグレーションを行い制御を得る．そしてファイルシステムを ActiveOS 上で使用されていたときと同じパスでマウントする．以上の処理により，これらの処理後にマイグレーションされるプロセスは同じパスで同じ内容のファイルを扱うことが可能になる．

ファイルキャッシュマイグレーションは，ActiveOS のメモリ上の dirty なファイルキャッシュを検索し，BackupOS 上にコピーすることで行う．dirty なファイルキャッシュページはディスクの各ファイルシステムに対応してメモリ上に存在する `super_block` 構造体からたどることができる．そしてすべての dirty ページの内容をコピーして，BackupOS 上の `inode` 構造体に関連付けることで保護が可能である．ファイルキャッシュマイグレーションは ext3 ファイルシステムを対象に実装した．

なお，現在は保護すべき dirty ページの量が BackupOS のメモリサイズより大きい場合にはメモリ不足となりマイグレーションが不可能である．この場合に対応するための追加実装として，メモリ不足が生じた場合にはマイグレーションを中断し，コピーしたページを即座にディスクに書き込むことでメモリを解放する実装を行う予定である．

3.5　プロセスマイグレーション

プロセスマイグレーションは，ActiveOS で事前に指定された pid-namespace に含まれるプロセス群を対象とし，BackupOS が ActiveOS のメモリイメージからプロセス管理構造体を読み出した後に BackupOS 内に再構成することで行う．その実装は `fork` システムコールのコードを基盤として，カレントプロセスから実行状態のコピーを行う部分を，ActiveOS 上の停止したプロセスから実行状態のコピーを行うように改変することで実現した．

プロセスマイグレーションにおいて，プロセスのカーネル空間での実行状態を引き継ぐと同時に障害も引き継ぐ可能性が高い．Orthros では OS に発生した障害からのリカバリを目標とするため，障害の原因となるカーネル空間の実行状態は引き継がず，システムコールの実行状態は破棄

し失敗したものとしてプロセスに返す．また，プロセスマイグレーションでは，無関係なプロセスへのリンクやスケジュール優先度等の実行状態はコピーを行わない．なぜなら，これらの実行状態はプロセスの周辺環境に依存しリカバリの前後で意味が変動するためである．また，Orthros はデバイスドライバが保持するデータは保護しないため，プロセスが使用するグラフィックカード等の情報は初期化される．これらの情報に対応することは今後の課題である．

本節では，プロセスの実行状態の保護と通信状態のマイグレーションについて述べる．そして次節でシステムコール実行中のプロセスに対するマイグレーション時のエラーハンドリングについて述べる．

3.5.1　プロセスの実行状態保護

Linux カーネルにおける複雑なプロセス構造のすべてを OS を越えて取得して，そのまま使用することは整合性の観点から難しい．そこで，Orthros では単一のプロセスとして動作するために最低限保護すべき状態を，プロセスのレジスタの値とメモリ内容と開いているファイルの管理状態とし，これらのみを保護する．

まず，プロセスが使用していたレジスタの値の取得方法について述べる．障害発生時に実行中でなかったプロセスについてはメモリ上に値が保存されているため，そこから値を得ることができる．しかし，プロセスが実行中であった場合は CPU 内にしか値の情報が存在せず，メモリから読み取ることができない．そのため，障害発生時に ActiveOS の各コアに対して NMI を発生させることで強制的に割込みを発生させ，メモリ上にレジスタの値を保存させることでレジスタの値を得る．

次に，メモリ内容の保護は，仮想メモリの管理構造とそれをたどり得られる dirty ページの内容のみがコピーの対象となる．dirty でないページはフェイルオーバを行った後でもデマンドページングによって自動的に再生成されるため，コピーを行わないことでプロセスマイグレーションの時間を短縮する．プロセスの持つ dirty ページのコピーについても，3.4 節で述べたファイルキャッシュの場合と同様に BackupOS のメモリサイズに依存する問題がある．この問題に対応するため，メモリ不足時にはプロセスマイグレーションを一時中断しコピーしたページをディスクに書き出す実装を行う予定である．

最後に，ファイル管理状態の保護について述べる．ファイルの内容はファイルキャッシュマイグレーションによって保護されているため，プロセスが持つファイルディスクリプタの状態を BackupOS 上に復元するだけでよい．ファイルディスクリプタのコピーを行う場合に，`file` 構造体以下のデバイスドライバに直結するデータ構造が障害の原因となっている可能性があるため，そのままコピーすることは困難である．したがって，`file` 構造体をコピーするのではなく，同じファイルを開いて新たな `file` 構造体を作成し，その後にフラグやシーク位置等の必要な情報のみをコピーする．

3.5.2　プロセス間の通信状態保護

Linux においてプロセス間通信にはパイプとシグナルと共有メモリと UNIX ドメインソケットの 4 種類が存在する．このうち，Orthros が現在保護可能なものはパイプと古典的な共有メモリである．他については将来的に実装を行う予定である．なお，5 章で述べるアプリケーションが未対応の方法を使う場合には，保護可能な機能で代用するようにビルド時のコンフィグレーションを改変している．

まずパイプは，マイグレーションするために BackupOS が新たなパイプ構造体を作成し，ActiveOS 上のプロセスが保持していたパイプ構造体からバッファの内容および管理情報をコピーする．次に，古典的な共有メモリは，通常のファイルを各プロセスのメモリにマップする方式で実現されている．したがって，ファイルキャッシュマイグレーションで保護済みのファイルに対してマップ処理をすることで状態を保護可能できる．

3.5.3　TCP/IP の通信状態保護

ネットワークを隔てた通信相手に透過的なリカバリを実現するためには，通信の不具合を通信プロトコルの各層で自動修正可能な範囲に抑えればよい．ネットワーク層以下の不具合をなくすために，BackupOS は ActiveOS が使用した NIC をデバイスマイグレーションして制御を取得し，IP およびルーティング等に同一の設定を適用することで ActiveOS と同じネットワーク環境を得る．次に，セッション層以上の制御情報はプロセスが独自で管理する部分であるため，プロセスの実行状態を保護することで，セッション層以上の保護が可能である．最後に，トランスポート層の TCP/UDP での不具合をなくすために，プロセスの `task_struct` に関連付けられたソケットの情報をコピーする．TCP 通信ではネットワーク層についての情報とウィンドウ制御情報とパケットキューとソケットオプションによって生成されるソケットの状態について整合性を保つ必要があり，Orthros はそれらをコピーする．TCP プロトコルは再送制御を持ち，UDP プロトコルは信頼性の確保が必要ないため，これによりそれまで通信していた相手との通信継続が可能となる．TCP ソケットのマイグレーションは Linux kernel 3.5 の TCP repair mode のコードを参考にした．

Orthros は上記の機能を実装し，これにより 5 章で紹介するアプリケーションは通信を継続することができた．ただし，現在の実装は TCP/UDP の状態をマイグレーションできるが，OS 間の時間のずれを完全になくすことはできないため，TCP Timestamp やタイムアウト等の時間に関する内容について完全にマイグレーションすることは困難である．たとえば，パケットに添付される `jiffies` は単調増加することのみ想定され，`jiffies` の逆戻りが生じ

たパケットは通常は破棄されるため，BackupOS は自身の jiffies が ActiveOS の jiffies を追い越すまでの間通信を再開できない．そのため，BackupOS の jiffies は事前に ActiveOS のものよりも遅らせておくことでこのような状態を発生させないようにする．

3.6　エラーハンドリング

3.5 節で述べたように，障害発生時にプロセスがシステムコールを実行していた場合，Orthros はシステムコールを中断してプロセスに失敗したと認識させる．しかし，システムコールが失敗した場合のエラーハンドリングを実装していないプログラムは異常動作が発生する．また，エラーハンドリングを実装しているプログラムでも，システムコールの再実行を行わずに停止する場合には，プロセスマイグレーション後の処理継続が実現しない．

このように，Orthros 上で動作するユーザプログラムには，フェイルオーバ時のためのエラーハンドリングの追加実装が必要となる場合がある．この問題を解決するために，Orthros はプロセスに対してシステムコールの失敗を隠蔽するエラーハンドリングライブラリを提供する．

3.6.1　エラーハンドリングライブラリの機能

アプリケーションプログラマはエラーハンドリングライブラリをロードすることにより，プロセスマイグレーションが中断させたシステムコールを自動的に再実行するようにプログラムの動作を変更できる．また，アプリケーションプログラマがこの対応を行っていない場合にも，システム管理者がプロセス起動時に LD_PRELOAD を用いてライブラリをロードすることでも同様の対応が可能である．

しかし，システムコールの再実行は異常な結果をもたらす場合がある．たとえば，ファイルのシークを行うシステムコールの途中で障害が発生し再実行した場合には，余分にシークされた状態を発生させる可能性がある．このようなシステムコールの二重実行による影響を排除するためには，システムコールごとに実行前の値を保存する等，再実行の判断を確実に行う方法を検討する必要があり今後の課題である．

また，エラーハンドリングライブラリはプロセスマイグレーション後に任意の処理を行うフック関数を実行する機能をアプリケーションプログラマに提供する．フック関数はプロセスマイグレーション後に特殊な処理を実行したい場合に，プログラマが作成する関数である．フック関数を用いることで，たとえば Otherworld [6] の Crash Procedure と同様に，保持している重要なデータを保存して終了する処理や，ログを出力する処理等を独自に記述可能である．また，フック関数は，中断されたシステムコールの種類と引数を受け取り，当該システムコール再実行の可否の判断を行うことが可能である．

3.6.2　ライブラリのロードとフック関数登録

本ライブラリの利用には，障害発生前のプロセスの起動時にロードすることが必要である．本ライブラリは，ロード時にエラーハンドリング関数のアドレスを OS に通知する．エラーハンドリング関数は，プロセスマイグレーション後にフック関数の呼び出しおよびシステムコール再実行を行う関数である．BackupOS への関数アドレスの通知方法は，専用のシステムコールを作成することで実装した．

また，本ライブラリは前項で述べたプログラマが独自に作成したフック関数のアドレスを専用システムコールを通して OS に通知する機能を持つ．プロセスはライブラリをロード後にプログラムの任意の場所でフック関数登録用システムコールを実行し，フック関数のアドレスを OS に通知することができる．

3.6.3　プロセスマイグレーション時のエラーハンドリング

BackupOS は，エラーハンドリング関数の登録の有無によってライブラリのロードの有無を確認し，エラーハンドリング関数を実行するか否かの判断を下す．実行する場合には，プログラムカウンタとスタックの改変によりエラーハンドリング関数への遷移と同関数の実行に必要な情報の受け渡しをする．具体的には，エラーハンドリング関数のアドレスをプロセスのプログラムカウンタにセットする．さらに，同関数の実行に必要な情報として，元のプログラムカウンタの値とフック関数のアドレスとシステムコール中か否かを示すフラグがあり，これらはスタックに積むことで受け渡す．

プロセスマイグレーション後のプロセスは，BackupOS によるプログラムカウンタの改変により最初にエラーハンドリング関数を実行する．エラーハンドリング関数は，まず現在のレジスタの値をスタックに退避させ，さらにそのレジスタの値からシステムコールの種類と引数を認識する．同時に，スタックから，BackupOS が積んだフック関数のアドレスとシステムコール中であるか否かの情報を得る．その後，フック関数のアドレスが設定されていればフック関数を呼び出し，その結果としてシステムコールを再実行すべきか否かを受け取る．システムコール中でありかつフック関数によって再実行が拒否されなかった場合には，システムコールを再実行し返り値を得る．次に，スタックをエラーハンドリング関数実行前の状態に復元し，退避させておいたレジスタの値を復帰させる．ただし，システムコールを再実行した場合にはシステムコールの返り値が入るべきレジスタに再実行の結果をセットする．最後にリターン命令を実行することにより，システムコール再実行の結果をともない元のコードからの実行を継続する．

4.　評価

本章ではフェイルオーバに必要な時間およびシステム導入による性能低下の測定を行う．評価に使用する計算機の

構成は図 1 と同様であり，CPU には Core i5 760（2.80 GHz）を用いた．BackupOS への CPU，メモリ，SSD の割当ては動作に必要な最低限の容量である．特に，メモリ割当ては評価時にメモリ不足に陥らない量で 2 のベキ乗となる最小量の 512 MB を用いた．プロセスおよびファイルキャッシュの dirty メモリ量がこの割当てを超える場合の処理の実装は今後の課題である．なお，現在の OS 同時実行の実装には，BackupOS の動作に専用のディスクを 1 台必要とする．これは BackupOS の動作にファイルシステムを必要とするためであり，将来的に Ramdisk の技術を用いて解決する予定である．

4.1　リカバリ時間の測定

評価環境でのフェイルオーバ処理の概略を**図 3** に示す．フェイルオーバ処理による停止時間は，(A) 必須停止時間，(B) データ依存停止時間と (C) ネットワーク依存停止時間の 3 種類からなる．時間測定は図 3 に示す 3 種類の処理ごとに行う．なお，各測定結果は 3 回の試行による平均で求めている．また，フェイルオーバの開始は `panic()` 関数を呼び出すシステムコールを作成し，保護対象のプロセスとは別のユーザプロセスから人為的に行った．

4.1.1　必須停止時間に関する処理

必須停止時間は計算機構成のみで処理量が一定に定まる処理であり，図 3 において A で示される．測定した平均必須停止時間の内訳を**図 4** に示す．左のグラフは必須停止時間の全体を示すものであり，右のグラフは見やすさのために全体から SSD デバイスマイグレーションの時間を除いたものである．時間は合計で 415 msec であり，最も時間を要する処理は SSD のデバイスマイグレーション処理であった．この SSD にかかる時間をさらに調査したところ，udev によるデバイスファイルの作成が 9 割を占めた．

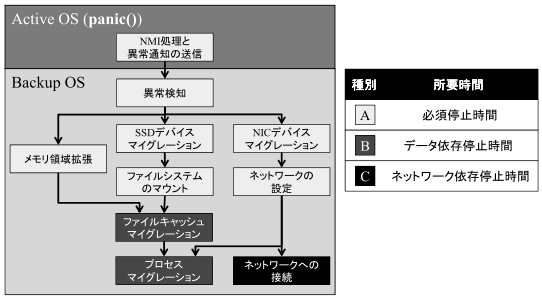

図 3　フェイルオーバ処理にかかる時間の分類
Fig. 3　Classification of failover time.

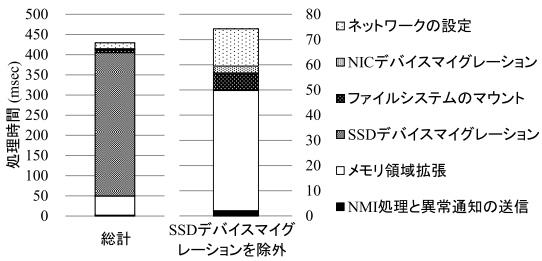

図 4　必須停止時間の内訳
Fig. 4　Breakout of fundamental downtime.

4.1.2　データ依存停止時間に関する処理

図 3 において B で示される処理は，ユーザの操作によって生じる保護データの特性に応じて時間が変動すると考えられるため，この処理時間をデータ依存停止時間と呼び個別に測定を行う．本項では，メモリコピー量とオープンしたファイル数とマイグレーションするスレッド数の 3 要素を変動させ処理時間を測定する．ファイルキャッシュマイグレーションの時間測定では，ファイルを生成後，ディスクに書き出される前に `panic()` を発生させた．また，各プロセスマイグレーションの時間測定では，プロセスが各リソース保持後にアイドル状態となっている間に `panic()` を発生させた．なお，測定対象以外の要素については処理時間が最小となるように調整した．

まず，メモリコピー量によるフェイルオーバ所要時間への影響を調査する．測定対象は，dirty なページ量を変化させた際のファイルキャッシュマイグレーション所要時間とプロセスマイグレーション所要時間である．測定結果を**図 5** に示す．この結果から，これら 2 つの要素のサイズとデータ依存停止時間は比例関係であることが分かる．1 MB あたりの処理時間はファイルキャッシュが平均 0.60 msec，プロセスが平均 0.34 msec 程度である．これによって，GB オーダの dirty ページが保持されない限りは大きな影響はないといえる．ファイルキャッシュの方が処理時間が長い理由は，ファイルキャッシュの管理構造がプロセスのメモリ管理構造よりも複雑であり，走査するオーバヘッドが大きいためである．

次に，オープンしたファイル数とスレッド数によるプロセスマイグレーション所要時間をそれぞれ**図 6** と**図 7** に示す．これらの結果から，データ依存停止時間はオープンしたファイル数とスレッド数についても比例関係にあることが分かる．1 ファイルあたり平均 4.7 μs，1 スレッドあたり平均 12 μs の処理時間が必要であるが，プロセスが

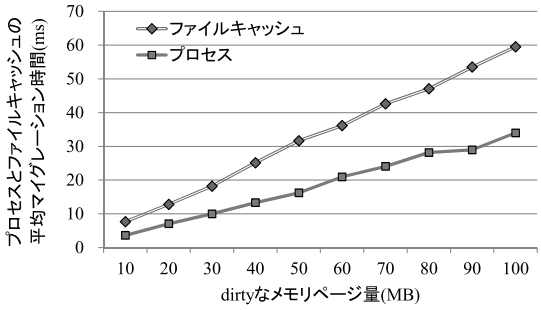

図 5　dirty なメモリ量によるプロセスおよびファイルキャッシュマイグレーション時間の推移
Fig. 5　Time required to migrate a process/filecaches from the amount of dirty memory.

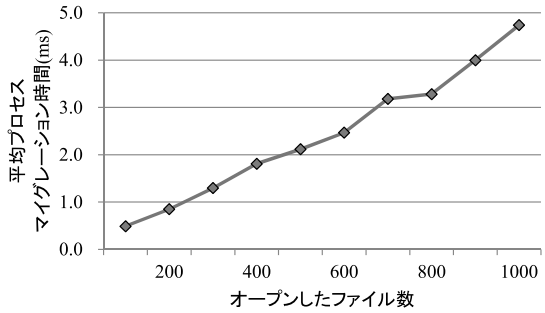

図 6　ファイル数によるプロセスマイグレーション時間の推移
Fig. 6　Time required to migrate a process from the number of files.

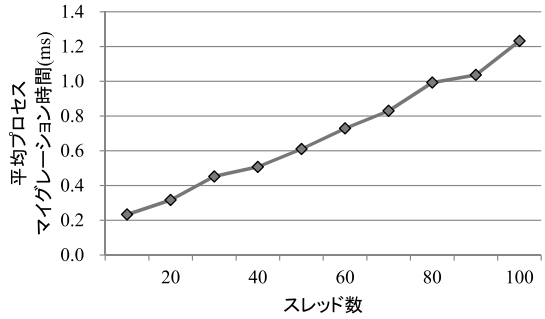

図 7　スレッド数によるプロセスマイグレーション時間の推移
Fig. 7　Time required to migrate a process from the number of threads.

1,000 ファイルと 100 スレッドを保持していた場合でも約 6.0 msec 程度でマイグレーションが終了し，400 msec 以上要する必須停止時間と比較して十分に短いため，マイグレーションの全時間に大きな変化はない．

以上から，データ依存停止時間について注意すべき時間は GB オーダでのメモリコピーが生じた場合のみであることが分かる．コピーが必要なメモリの量は最大でも物理メモリサイズ以下であるが，最近の物理メモリサイズの増加を考えれば無視できない時間である．しかし，それだけのメモリコピーを必要とする巨大なシステムのリカバリ時間という観点では十分に高速であると考える．

4.1.3　ネットワーク依存停止時間に関わる処理

図 3 における C の処理は，デバイスマイグレーション後の NIC に対するネットワーク設定を終えてから，実際にネットワーク上の他ノードと通信可能になるまでの待機処理である．この時間をネットワーク依存停止時間と定義する．このネットワーク依存停止時間は計算機が置かれるネットワークの構成によって大きく変わる可能性がある．また，実際に通信が行われていなければ隠蔽される時間であるため，他の時間とは別の扱いにして測定を行う必要がある．

今回の評価では，NIC に対する設定が終了してから同一ネットワーク上の計算機への icmp echo パケットを 0.1 秒間隔で送信し，それが成功するまでの時間を測定した．なお，通信速度と通信モードの設定にオートネゴシエーションを使用している場合 4 秒程度の時間を必要とするため，

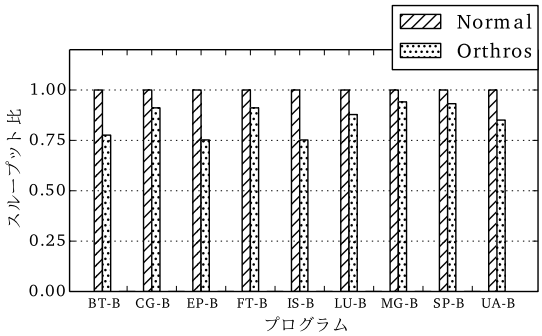

図 8　NAS Parallel Benchmark のスコア
Fig. 8　Score of NAS Parallel Benchmark.

Orthros では手動での設定を行う．結果，上記の環境では 1,007 msec 程度で通信が回復した．この時間の詳細な内訳は現在調査中であるが，アウトオブバンド管理や arp 等ではないかと予想される．今後，これらの情報のマイグレーション方法を検討する．

4.1.4　総計リカバリ時間

4.1.1 から 4.1.3 項までの結果より，ネットワークを使用しない小さなプロセスを保護した場合の最短停止時間は約 0.4 秒であることが分かる．そして，プロセスがリカバリ直後にネットワークを使用する場合の最短停止時間は約 1.4 秒である．また，故障の検知が異常通知ではなく，生存通知が停滞したことによるフェイルオーバであった場合には，生存通知による検出のための時間としてさらに 1 秒が必要である．さらに，dirty ページ量が多くなると，物理メモリサイズを上限として 1 GB あたり 0.3 秒から 0.6 秒の遅延が生じる．

4.2　実行時性能低下幅の測定

評価環境に Orthros を導入した際の最も大きな性能低下要因は CPU のコア数減少である．図 1 の構成の場合は，コア数の減少が 1 コアであることから 4 コアの CPU の場合には最大で 25%程度の計算リソースが失われると考えられる．そこで，並列処理マイクロベンチマークを用いて性能低下の割合を測定する．NAS Parallel Benchmark の BT，CG，EP，FT，IS，LU，MG，SP，UA の B クラスを用いて 10 回測定した平均の評価結果を**図 8** に示す．図 8 は，Orthros（Orthros 導入時のスコア）を Normal（非導入時のスコア）で正規化したグラフである．

Orthros 導入時の性能はいずれも非導入時に比べて 75%以上であり，コア数の減少分を超える性能低下は発生しないことが分かる．また，必ずしも 25%の性能が低下するわけではなく，アプリケーションによっては，性能が数%低下するだけの場合もある．

5.　アプリケーションへの適用

本章では提案システムを対話型アプリケーション，NFS

サーバ，データベースサーバと HTTP サーバに適用し，その応用可能性について議論する．各アプリケーションに対して適用による利点について検証し，実際に停止時間の測定を行う．停止時間の測定には，4 章と同じ測定環境およびフェイルオーバ処理を用い，3 回試行した平均の時間を用いた．なお，いずれのアプリケーションも障害発生は `panic()` を呼び出すことにより行った．また，計算機外部との通信が必要な場合には 1 Gbps ネットワークスイッチを介して接続した．

5.1　対話型アプリケーション

対話型アプリケーションとして代表的なアプリケーションはテキストエディタである．たとえば nano [14] はユーザが保存操作をしない限り編集途中の内容をディスクに保存せず，一時データとしてメモリ上に保持する．nano はオートセーブ機能を備えていないため，大量の実行状態を損失した場合の復旧には長時間を要する．Orthros を nano のテキスト編集に適用することで，ユーザは OS の不意の障害から編集中のデータを保護することが可能である．

Orthros 上で nano を用いて 4 MB のテキストを編集中に障害が発生した場合の停止時間を測定した．nano の実行時には，エラーハンドリングライブラリをロードすることでシステムコールの失敗による終了処理を行わないようにした．また，`panic()` の実行箇所は，(1) nano がファイルを開くときの `read` システムコールによりページキャッシュからのデータコピーを行うカーネル処理中，(2) `nano` がファイルを保存するときの `write` システムコールによりページキャッシュへのデータコピーを行うカーネル処理中，(3) 標準入力からの入力待ち中に別プロセスから `panic()` を呼び出す場合の 3 通りである．ユーザが BackupOS 上で編集が継続可能になったのは 0.46 秒後から 0.48 秒後で平均 0.47 秒後であり，不整合なくディスクの読み出しと書き込みを完了した．また，編集途中の内容はすべてメモリ上で保護され保存することができた．

5.2　NFS サーバ

NFS は，ネットワークの切断については大きな被害を受けることはない．これは，ファイル操作中に NFS サーバが異常停止した場合，NFS クライアントは成功するまで操作要求を繰り返し，通信が復旧すると自動的に操作が再開されるためである．しかし，ファイルへの書き込み中に OS に障害が発生してファイルキャッシュが破壊された場合には，操作要求は通信復旧後も正しく処理されないことがある．このように，NFS サーバはファイルキャッシュの損失に対して脆弱であるといえる．また，NFS サーバに障害が発生すると，クライアント上の NFS に依存するシステムはリブートが完了するまで長時間停止することも問題である．この問題は一般的には NFS サーバの冗長化により解決可能であるが，トレードオフとして管理の複雑化および機器導入コストの増大をもたらす．

Orthros を NFS サーバへ適用することで，ファイルの破壊とクライアントシステムの長時間停止という 2 つの問題を解決することができる．ファイルキャッシュを保護し，さらに ActiveOS と同じネットワーク環境を得ることによって，BackupOS はファイルの破壊を防ぎ NFS クライアントとの透過的な通信が可能である．なお，NFS サーバはステートレスなアプリケーションであるため，プロセスマイグレーションを行う必要はなく，BackupOS で NFS サーバを起動していれば透過的に要求を処理することができる．

Orthros を NFS サーバに対して適用し，NFS クライアントが NFS サーバと通信不可能になってから通信が再開されるまでの時間を測定した．この際，NFS サーバおよびクライアントのプロセス自体に変更は加えていない．`panic()` の実行箇所は，(1) nfsd がクライアントの要求を受けてファイルを読み出す際のページキャッシュからのデータコピーを行うカーネル処理中，(2) 同ファイルを書き込む際のページキャッシュへのデータコピーを行うカーネル処理中の 2 通りである．なお，TCP では操作要求の再送間隔が長くリカバリに時間を要するため，通信プロトコルに UDP を設定しタイムアウト時間を 0.1 秒に設定した．測定した停止時間は 1.34 秒から 1.37 秒で平均 1.35 秒となり，4.1.4 項の最短停止時間に近い値となった．

5.3　データベースサーバ

データベースサーバが動作する OS に障害が発生した場合，サーバとクライアント間のセッションは断絶し，実行中のトランザクション処理はロールバックされる．したがって，データベースクライアントはデータベースサーバがリカバリするまで待機したうえで再接続し，さらに実行していたトランザクション処理を最初からやり直さなければならない．この問題によってデータベースシステムの可用性は低下する．しかし，Orthros を適用することで解決することが可能である．また，Orthros をデータベースサーバに適用した場合のもう 1 つの利点として，高速化が可能になることがあげられる．Orthros によってメモリ上に存在する実行状態の信頼性が向上するため，たとえば sync の削減やインメモリデータベース使用が容易になると考える．

Orthros を MySQL（InnoDB）のサーバに適用し，障害発生時の停止時間を測定した．この際，プログラムのビルド時のコンフィグレーションスクリプトを改変することで UNIX ドメインソケットを使用しないようにした．また，システムコール失敗時のプロセス終了を防ぐためにエラーハンドリングライブラリをロードした．障害は，8 秒程度かかるクエリの実行開始から 4 秒程経過した時点で外部プロセスから `panic()` を呼び出し発生させた．この瞬間，スレッドの 1 つは CPU コアを使用してユーザ空間でクエ

リを実行中であり，他スレッドおよびプロセスはアイドル状態で待機中であった．クエリ実行時間を，障害を発生させた場合と同条件で発生させなかった場合で計測して比較した結果，停止時間は 0.54 秒から 0.55 秒で平均 0.54 秒となった．この際，ネットワーク依存停止時間はプロセスマイグレーションを行ってからクエリの処理が完了するまでの時間で隠蔽されている．そのため，クエリの実行時間が短くなるにつれて最大約 1 秒の遅延が表面化する．

5.4 HTTP サーバ

HTTP はステートレスなプロトコルであるため，セッション情報を保持しなければ OS に障害が発生しても大きな情報損失は発生しない．しかしリカバリにかかる時間はシステムの可用性を低下させる．また，SSL 通信によるログイン処理や連続するフォーム処理等，HTTP サーバがクライアントとの通信においてセッション情報を保持する場合，セッション情報保持の信頼性が問題となる．これらのセッション情報が損失した場合に，ユーザはセッション確立処理を再度行う必要がある．また，最悪の場合ではセッション確立から障害発生までの間に蓄えられた処理をすべてやり直す必要があり，大きなユーザビリティ低下となる．

サーバ側でのセッション情報の保存場所はメモリ上とディスク上が存在し，ディスク上ではデータベースを用いる場合と用いない場合がある．メモリは高速性に優れるが障害発生時の信頼性に劣り，ディスクは信頼性に優れるが高速性を犠牲にする．Orthros はメモリ保存時の信頼性を向上させることが可能である．また，ディスク保存時にもデータベースを使用しない場合はファイルキャッシュレベルでの信頼性を向上させることができる．さらに，データベースを使用する場合には前節で述べた場合と同様の効果がある．

さらに，非常に長い送受信処理の途中でサーバが異常停止した場合には通信の再開処理かやり直し処理が必要であり，ユーザビリティが低下する．たとえば，HTTP ファイルサーバやプログレッシブダウンロードによる動画配信は長時間通信を継続する可能性が高く，通信処理途中での異常停止はユーザにとって大きなユーザビリティ低下となる．Orthros はファイル送受信中の状態も保護可能であるため，このような用途のサーバの信頼性を向上させることができる．

Orthros を Apache2 に適用し，障害発生時の停止時間を測定した．この際，プログラムビルド時のコンフィグレーションファイルを変更することで，共有メモリの方式に通常ファイルのマップを使用するようにした．また，システムコールの失敗によるプロセスの終了を防ぐためにエラーハンドリングライブラリをロードした．障害は HTTP クライアントからのリクエストで html ファイルを読み出すときの `read` システムコール，およびアクセスログを書き込むときの `write` システムコールによって発生するページキャッシュとのデータコピーを行うカーネル処理中に `panic()` を呼び出し発生させた．HTTP リクエストを 0.01 秒間隔で送り続けて応答のない時間を測定した結果，停止時間は 1.47 秒から 1.48 秒で平均 1.48 秒であった．これは，4.1.4 項で述べた最短停止時間に近い値である．

6. 関連研究

本章では，OS に発生する障害に対して耐障害性を高めるための関連研究について述べる．

6.1 データ保護

Otherworld [6] は，OS をマイクロリブートすることによってプロセスの実行状態を保護する手法である．OS に障害が発生した際に，計算機を停止せずに新たな OS を Warm-boot することで故障した OS のメモリイメージを残存させる．新しい OS はそのメモリイメージを参照することによって，プロセスに関するデータを読み出して復元することで実行を再開する．これによって OS の障害からプロセスおよびプロセスが持つデータを保護することができる．しかし，Otherworld はファイルキャッシュとネットワーク通信状態の保護を行わないため，それらに依存するプロセスはリカバリ後に正しい動作を行うことができない．Orthros はプロセス保護時にそれらの実行状態を同時に保護するため，プロセスの正しい動作を保証するものである．また，Orthros は OS の初期化時間を必要としないため Otherworld よりも高速にリカバリを行うことができる．さらに，リカバリ処理は Warm-boot した新しい OS ではなく同時実行される独立した OS が行うことで，障害の発生した OS が Warm-boot 処理を行う場合よりも高い信頼性を得る．

Chen らの手法は障害発生時のファイルキャッシュ書き出しを高い信頼性で保証する [15]．そのために，OS のクラッシュ時に必ず `sync` が実行されるようにクラッシュハンドラを設定し，その書き出し処理は OS のデータ構造に極力依存しないように実現している．Orthros は障害が発生していない独立した OS が障害発生確認とファイルキャッシュ保護を行うため，Chen らの手法よりも高いファイルの信頼性を保証する．

実行状態保護という点では Le らの手法も提案手法に類似している [16]．VMM に障害が発生しリブートする際，障害が発生した VMM のメモリイメージを取得して利用することで高速なリカバリおよび保存オーバヘッドのない実行状態保護を実現している．しかしこの手法はあくまで VM の状態保護であり，VM 内の OS の障害には対応していない．

6.2 リカバリの改良

Shimos2 [7] は OS の冗長化による高速なリカバリと C/R

によるプロセス保護を行う手法である．Shimos2 では Software LPAR を用いてコアごとに OS が動作するため 1 つの OS に障害が発生しても他の OS が動作を継続でき，高速なフェイルオーバが可能になる．しかし，プロセスの実行状態保護に C/R を用いており実行時オーバヘッドが大きく，50 msec ごとのチェックポインティングで約 45%，1 秒ごとでも約 10%の性能低下をもたらすことが報告されている．また，最後のチェックポイントから障害までの最新の実行状態は保護不可能である．Orthros では Shimos2 と同じく OS の同時実行によるフェイルオーバを行うが，実行状態保護を C/R ではなく障害が発生した OS のメモリを読み取ることによって実現する．これによって実行時オーバヘッドを発生させずにプロセスとファイルキャッシュの最新の状態を保護可能である．

同様に複数の OS を用いる研究として，仮想環境を用いた OS のセルフヒーリングシステムが存在する [17]．この手法では冗長構成をとるためではなく，主となる OS に発生した不整合を修正するために 2 つ目の OS が存在する．しかし，これによりリブートのタイミングを多少ずらすことは可能であるが，最終的に必要なリブートは長時間を要する．

単一の OS で高速なリカバリを提供する研究としては Baker らの手法があげられる [18]．この手法は再生成に時間を要する実行状態をバックアップ/リストアするフレームワークを提供し，OS およびアプリケーションの高速なリカバリを実現している．この手法では実行時オーバヘッドはほぼ存在しないが，リカバリ時間は依然として長く数十秒程度であるとの報告がなされている．また，この手法が扱える実行状態はシリアライズ可能なデータのみである．

OS に障害が発生した際のリブートを高速化する手法として，Yamakita らの手法は C/R を用いて起動途中の VM のスナップショットを段階的に保存する [19]．そして障害が発生した際に最適なスナップショットから再開することによって，リブート処理の一部をスキップし高速化することが可能である．この手法は高速だが，仮想環境を利用するという制限が生じ，またプロセスの最新の実行状態を保護することができない．

6.3 障害の局所化

OS の耐障害性向上に対する別のアプローチとして，障害を局所化することが考えられる．たとえば，いくつかの手法 [20], [21] は OS の構成形式をマイクロカーネルにする．デバイスドライバをユーザ空間で実行することで OS 本体を保護し，デバイスドライバのマイクロリブートが可能になる．また，カーネル空間内のロックアップを検知して実行途中の処理を破棄する手法 [22] や，OS 内の処理要求のやりとりを限定し障害の波及を抑えるアプローチ [23] が存在する．また，Ishikawa らはドライバが self-healing を行うためのフレームワークを提供している [24]．

これらの手法は，事前にデバイスドライバ等の障害発生箇所の限定をしてから対策をするものである．範囲を限ったリカバリは高速であり，障害発生箇所以外の状態を保護可能である．我々の手法も最も大きな対象はデバイスドライバより発生する障害であるが，障害の発生箇所の厳密な特定を必要とせず，より一般的な対策となる．

7. まとめ

本稿では複数の OS を同時実行し，フェイルオーバを行うシステムとして Orthros を提案した．Orthros は OS に障害が発生した際に，プロセスとファイルキャッシュを保護し高速なリカバリを行う．本稿では各種アプリケーションに Orthros を適用した際の利点と，リカバリが数秒で終了することを示した．これは，計算機のリブートや OS の Warm-boot よりも高速な時間である．

Orthros は Software LPAR を最小限の計算機構成と最小限の実行時オーバヘッドを実現するために使用した．また，実行時オーバヘッドをともなわないプロセスおよびファイルキャッシュの保護のために障害が発生した OS のメモリを読み取った．これによって，Orthros の実現に必要なハードウェアはマルチコアプロセッサとディスクのみである．また，性能低下要因については CPU 1 コアと 512 MB のメモリ専有と，生存通知を送信するためのシンプルな割込み処理のみである．

今後の予定は，フェイルオーバ後の障害発生への対応である．現在はフェイルオーバ後に BackupOS に障害が発生した場合にリブートが必要になる．この問題に対処するため，フェイルオーバ後に新たな BackupOS を起動可能にするシステムを作成する予定である．

参考文献

[1] Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D.: An empirical study of operating systems errors, *Proc. 18th ACM Symposium on Operating Systems Principles*, *SOSP '01*, pp.73–88 (2001).

[2] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in linux: Ten years later, *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS '11*, pp.305–318 (2011).

[3] OpenClovis: SAFplus, available from ⟨http://help.openclovis.com/index.php⟩.

[4] Hargrove, P.H. and Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters, *Journal of Physics: Conference Series*, Vol.46, No.1, pp.494–499 (2006).

[5] Liao, J. and Ishikawa, Y.: A New Concurrent Checkpoint Mechanism for Real-Time and Interactive Processes, *Proc. 2010 IEEE 34th Annual Computer Software and Applications Conference*, *COMPSAC '10*, pp.47–52 (2010).

[6] Depoutovitch, A. and Stumm, M.: Otherworld: Giv-

ing applications a chance to survive OS kernel crashes, *Proc. 5th European Conference on Computer Systems*, *EuroSys '10*, pp.181–194 (2010).
[7] Liao, J., Shimosawa, T. and Ishikawa, Y.: Configurable Reliability in Multicore Operating Systems, *Proc. 2011 14th IEEE International Conference on Computational Science and Engineering*, *CSE '11*, pp.256–262 (2011).
[8] Shimosawa, T., Matsuba, H. and Ishikawa, Y.: Logical Partitioning without Architectural Supports, *Proc. 2008 32nd Annual IEEE International Computer Software and Applications Conference*, *COMPSAC '08*, pp.355–364 (2008).
[9] Ahmed, M.F. and Gokhale, S.S.: Linux bugs: Life cycle, resolution and architectural analysis, *Inf. Softw. Technol.*, Vol.51, No.11, pp.1618–1627 (2009).
[10] Yoshimura, T., Yamada, H. and Kono, K.: Is Linux kernel oops useful or not?, *Proc. 8th USENIX Conference on Hot Topics in System Dependability*, *HotDep '12* (2012).
[11] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles*, *SOSP '03*, pp.164–177 (2003).
[12] Kivity, A.: kvm: The Linux virtual machine monitor, *The 2007 Ottawa Linux Symposium*, *OLS '07*, pp.225–230 (2007).
[13] Nomura, Y., Senzaki, R., Nakahara, D., Ushio, H., Kataoka, T. and Taniguchi, H.: Mint: Booting Multiple Linux Kernels on a Multicore Processor, *Proc. 2011 International Conference on Broadband and Wireless Computing, Communication and Applications*, *BWCCA '11*, pp.555–560 (2011).
[14] GNU: nano, available from ⟨http://www.nano-editor.org/⟩.
[15] Chen, P.M., Ng, W.T., Chandra, S., Aycock, C., Rajamani, G. and Lowell, D.: The Rio file cache: Surviving operating system crashes, *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS '96*, pp.74–83 (1996).
[16] Le, M. and Tamir, Y.: ReHype: Enabling VM survival across hypervisor failures, *Proc. 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *VEE '11*, pp.63–74 (2011).
[17] Katori, T., Sun, L., Nilsson, D.K. and Nakajima, T.: Building a self-healing embedded system in a multi-OS environment, *Proc. 2009 ACM Symposium on Applied Computing*, *SAC '09*, pp.293–298 (2009).
[18] Baker, M. and Sullivan, M.: The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment, *Proc. USENIX Summer Conference*, *USENIX '92*, pp.31–43 (1992).
[19] Yamakita, K., Yamada, H. and Kono, K.: Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery, *Proc. 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, *DSN '11*, pp.169–180 (2011).
[20] Boyd-Wickizer, S. and Zeldovich, N.: Tolerating malicious device drivers in Linux, *Proc. USENIX Annual Technical Conference*, *USENIX ATC '10*, pp.117–130 (2010).
[21] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A.: Microreboot – A technique for cheap recovery, *Proc. 6th Conference on Symposium on Opearting Systems Design & Implementation*, *OSDI '04*, Vol.6, pp.31–44 (2004).
[22] David, F.M., Carlyle, J.C. and Campbell, R.H.: Exploring recovery from operating system lockups, *Proc. USENIX Annual Technical Conference*, *USENIX ATC '07*, pp.351–356 (2007).
[23] Lenharth, A., Adve, V.S. and King, S.T.: Recovery domains: An organizing principle for recoverable operating systems, *SIGPLAN Not.*, Vol.44, No.3, pp.49–60 (2009).
[24] Ishikawa, H., Courbot, A. and Nakajima, T.: A Framework for Self-Healing Device Drivers, *Proc. 2008 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, *SASO '08*, pp.277–286 (2008).

吉田 健二　（学生会員）

2012 年名古屋工業大学工学部情報工学科卒業．同年同大学大学院工学研究科情報工学専攻博士前期課程入学，現在に至る．

齋藤 彰一　（正会員）

1993 年立命館大学理工学部情報工学科卒業．1995 年同大学大学院博士前期課程修了．1998 年同大学院博士後期課程単位習得中退．同年和歌山大学システム工学部情報通信システム学科助手．2003 年同講師，2005 年同助教授．2006 年名古屋工業大学大学院助教授，2007 年同准教授，現在に至る．オペレーティングシステム，インターネット，セキュリティ等の研究に従事．博士（工学）．ACM，IEEE-CS 各会員．

毛利 公一　（正会員）

1972 年生．1994 年立命館大学理工学部情報工学科卒業，1996 年同大学大学院理工学研究科修士課程情報システム学専攻修了，1999 年同研究科博士課程後期課程総合理工学専攻修了．同年東京農工大学工学部情報コミュニケーション工学科助手，2002 年立命館大学理工学部情報学科講師，2004 年同大学情報理工学部情報システム学科講師，2008 年同准教授となり，現在に至る．博士（工学）．オペレーティングシステム，仮想化技術，コンピュータセキュリティ等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE-CS，USENIX 各会員．

松尾 啓志　（正会員）

1983 年名古屋工業大学工学部情報工学科卒業．1985 年同大学大学院修士課程修了．1989 年同大学院博士課程修了．同年名古屋工業大学電気情報工学科助手．講師，助教授を経て，2003 年同大学大学院教授．2006 年同大学情報基盤センターセンター長（併任），現在に至る．分散システム，分散協調処理に関する研究に従事．工学博士．電子情報処理学会，人工知能学会，IEEE 各会員．