情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 1　13–24 (Mar. 2010)

Kemari：仮想マシン間の同期による耐故障クラスタリング

田村芳明[†1]　柳澤佳里[†1]
佐藤孝治[†1]　盛合敏[†1]

インターネット上で提供されるサービスの増加と高機能化に加え，PC サーバの小型化，高速化，低価格化により，企業では多数の PC サーバで構成された複雑なシステムのコスト削減とリソースの有効利用が求められている．この課題を解決するために，仮想マシンを利用して，1 つの物理マシン上に複数のサーバ機能を統合することが検討されている．しかし，ハードウェア障害発生時にサービスを継続するためには，特殊なハードウェア，アプリケーションや OS に依存しない，可用性の高い構成が必要である．本論文では，仮想マシン間の同期による耐故障クラスタリング技術，Kemari について述べる．Kemari は，アプリケーションや OS に依存しないで，障害発生時にサービスを継続することができる．Kemari を仮想マシンモニタである Xen に実装し，実験を行ったところ，運用系の電源断といった障害でも，アプリケーションや OS が待機系で透過的に継続できることを確認した．

Kemari: Virtual Machine Synchronization for Fault Tolerance

Yoshiaki Tamura,[†1] Yoshisato Yanagisawa,[†1]
Koji Sato[†1] and Satoshi Moriai[†1]

Internet services are growing in numbers and functionality. Commodity servers are well used for those services, and service providers face a problem to lower the cost of running numbers of servers. Consolidating multiple servers by using virtual machines is an effective approach to reduce unnecessary costs such as floor space, power, and management, which results in better resource utilization. On the other hand, consolidating multiple servers to a single server may increase the risk of single point of failure. Although high availability architectures should be considered for this, current solutions require specially designed hardware, or modifications to running software. In this paper we propose Kemari, a cluster system that synchronizes virtual machines for fault tolerance that does not require specific hardware or modifications to software. We present the design, implementation and evaluation of our system.

1. はじめに

近年，インターネットの普及により，様々なサービスがインターネット上で提供され，その数も増加している．また，インターネットサービスのインフラストラクチャに広く用いられている PC サーバは，技術の進歩とともにコモディティ化が進んでいる．PC サーバは手軽に導入できるため，企業で使われるサービスにも多く利用されている．しかし今日の企業では，多数の PC サーバで構成される複雑なシステムのコスト削減とリソースの有効利用が課題となっている．この課題を解決する方法として，仮想マシンを利用したサーバ統合が多くの企業で検討されている．サーバ機能は仮想マシンによって提供され，複数の仮想マシンが 1 つの物理マシン上で動作することになる．これにより，設置スペースの削減，消費電力の削減，管理対象となるハードウェアの削減やリソース利用率の向上が可能になると考えられている．

しかし，仮想マシンを使ったサーバ統合には問題もある．サーバを統合する前のシステムの場合，1 つのサーバにハードウェア障害が発生したとしても，他のサーバに影響はない．一方，サーバ統合環境では，統合先のサーバにハードウェア障害が発生した場合，故障したサーバで動作するすべての仮想マシンに影響が広がる．このためサーバ統合環境では，サーバ統合をする前よりも多くのサービスが停止するという問題があり，これを解決するためには高可用性が必要になる．

従来の高可用化技術としては，フォールトトレラントサーバ（FT サーバ），高可用クラスタ（HA クラスタ）のアクティブスタンバイ方式やレプリケーション方式があげられる．FT サーバではソフトウェアを変更する必要はないが，ハードウェアのコストが大きい．HA クラスタのアクティブスタンバイ方式では障害発生後の切替えに時間を要する．また，レプリケーション方式ではソフトウェアの変更を必要とするため，サーバ統合環境に適用すると，仮想マシン上のすべてのソフトウェアに変更を加えなければならない．

本論文では，安価な PC サーバを対象に，アプリケーションや OS に依存しないで，障害

†1 日本電信電話株式会社サイバースペース研究所
NTT Cyber Space Laboratories

発生時にサービスを継続する技術，Kemari について述べる．Kemari は，仮想マシンのイベントを契機に仮想マシン間の同期をとることで，アプリケーションや OS に依存せずに，サービスを継続することができる．

2. 既存の高可用化技術と問題点

2.1 FT サーバ

FT サーバでは，CPU，メモリ，ストレージ，ネットワーク，電源などの主要なハードウェアコンポーネントが冗長化されている．各コンポーネントは，運用系の状態が変化すると，待機系の状態を運用系と一致させる処理が行われる．ここで，入力によって，状態が変化することを状態遷移という．ここでは，状態を一致させる処理を同期と呼ぶ．FT サーバでは，運用系が 1 命令進むごとに，待機系に運用系と同じ状態遷移をさせることで同期を行っている．さらに，同期処理をハードウェアで実装することで，高速に同期を行うことができる．アプリケーションや OS が動作するのに必要なコンポーネントは，つねに同期されているため，運用系に障害が発生して停止したとしても，待機系が処理を引き続いて行うことができる．したがって，アプリケーションや OS が特別な処理を必要としないで，透過的にサービスを継続することができる．しかし，FT サーバは特殊なハードウェアで構成されるため，同程度の性能を持つ PC サーバと比較して 10 倍程度の導入コストがかかる．

2.2 HA クラスタ

HA クラスタのアクティブスタンバイ方式では，運用系と待機系のサーバを用意し，運用系と待機系でストレージを共有する．運用系に障害が発生して停止した場合は待機系に切り替える．これをフェイルオーバという．前にあげた FT サーバと異なり，汎用的なハードウェアを利用することができるため，安価に構築することができる．運用系は待機系を同期させるのに必要な，アプリケーションに依存した情報を共有ストレージに書き，待機系はフェイルオーバ時にこの情報を用いてリカバリを行う．このように，アクティブスタンバイ方式では，フェイルオーバ時にアプリケーションごとのリカバリ処理が必要になるため，アプリケーションや OS から見て，透過的に可用性を得ることができない．また，フェイルオーバには時間を要し，その間はサービスが提供できなくなるという問題がある．

HA クラスタのレプリケーション方式も，アクティブスタンバイと同様に，運用系と待機系を用意する．アクティブスタンバイ方式とは異なり，それぞれのサーバが個別にストレージを持つ．レプリケーションの具体的な方法として，運用系のアプリケーションに来たリクエストを待機系にも転送することがあげられる．運用系に障害が発生して停止した場合，待機系のアプリケーションの状態は運用系と同期されているため，運用系を切り離してサービスを継続することができる．しかし，クラスタリングするアプリケーションごとにレプリケーションの仕組みを追加しなければならないため，アプリケーションや OS から見て，透過的に可用性を得ることができない．

3. 仮想マシンによる高可用化とその課題

3.1 仮想マシン間の同期による耐故障クラスタリング

本論文では，既存の高可用化技術の問題を解決する，仮想マシン間の同期による耐故障クラスタリング技術（Kemari）について述べる（**図 1**）．Kemari では，運用系がどの時点で停止しても，待機系が同じ状態になるように同期を行うことで，アプリケーションや OS に依存せずに，障害発生時にサービスを継続することを可能にする．本技術は，企業内の情報共有サーバなど，コストと可用性の両立を求められるサービスを主な対象とする．

本技術を実現するためには以下の機能が必要となる．

(1)　仮想マシンの同期
(2)　ハードウェア障害の検知
(3)　障害発生後の切替え

このうち，(2) と (3) については，既存の HA クラスタで同様の機能が実現されているため，それらを Kemari に応用できる．したがって，本論文では (1) の仮想マシンの同期方式について述べる．

3.2 仮想マシン間の同期における課題

アプリケーションや OS から見て，透過的にサービスを継続するためには，運用系と待機系の仮想マシン状態が同じでなければならない．これを実現する仮想マシン間の同期方式はこれまでも研究されており，lock step 方式と checkpoint 方式の主に 2 つが有効とされて

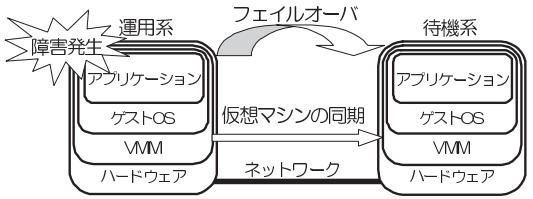

図 1　仮想マシン間の同期によるクラスタリング
Fig. 1　Clustering with VM synchronization.

いる．

lock step 方式は，運用系の仮想マシンに対する入力を，同期用ネットワークを通じて待機系の仮想マシンに対して与え，2 つの仮想マシンの状態遷移を同じに保ち続ける[4]．古くから研究されている技術で，製品化されているものもある[17),22)]．仮想マシンの同期に必要なデータ量が checkpoint 方式と比較して少ないとされ，効率的な仮想マシンの同期を実現している．一方，利用可能なプロセッサが限定され，同一プロセッサであってもプロダクトファミリによっては動作しないなど，実装コードのポータビリティが低い．

checkpoint 方式は，運用系の仮想マシンのイメージ（CPU，メモリ，ストレージなど）を待機系に送り，待機系の仮想マシンの状態を更新する．lock step 方式と比較して実装が容易であり，プロセッサの特定の機能に依存しないため，幅広い製品に実装することができる．一方，仮想マシンのイメージはデータ量が多いため，lock step 方式よりも 1 回の同期にかかるオーバヘッドが大きい．また，運用系の仮想マシンと待機系の仮想マシンをつねに同じ状態にしておくためには，定期的に同期を行うだけでは不十分である．特に，ストレージやネットワークを使用する場合，これらの状態と仮想マシンの状態が正しく遷移しないと，エラーやコネクションの切断が起きてしまう．このため checkpoint 方式では，待機系の仮想マシンの状態と，ストレージやネットワークなどの外部状態を一致させるために，運用系の仮想マシンからの出力を，待機系を更新するまでバッファする仕組みが必要となる[7),15)]．既存の checkpoint 方式の問題点は，lock step 方式と比べてオーバヘッドの大きい同期が定期的に行われてしまうことである．具体的には，CPU やメモリの負荷が中心のアプリケーションを実行している際に，lock step 方式であれば同期のオーバヘッドがほとんどないのに対して，既存の checkpoint 方式では Remus のように 25 ms という決まった間隔で同期を行うため，同期用ネットワークの通信量が定常的に高くなるとともに，一定の負荷がつねに生じるため，I/O 負荷がなかったとしても VM 上で動作するアプリケーションの性能を低下させてしまう．また，I/O をバッファリングする仕組みが必要になり，実装が複雑になる．

Kemari は，運用系と外部の間で発生するイベントに着目し，運用系の仮想マシンのイメージを，イベントドリブンで待機系に転送する．これは lock step 方式と checkpoint 方式の利点をあわせ持ち，低オーバヘッドかつ，幅広いハードウェアに実装可能な仮想マシンの同期を実現している．Kemari の新規性は，同期の頻度を最小限にし，定常的な同期のオーバヘッドが発生しない点，I/O をバッファリングする仕組みを必要としないため，実装がシンプルであるとともに，I/O が必要以上に遅延されない点である．

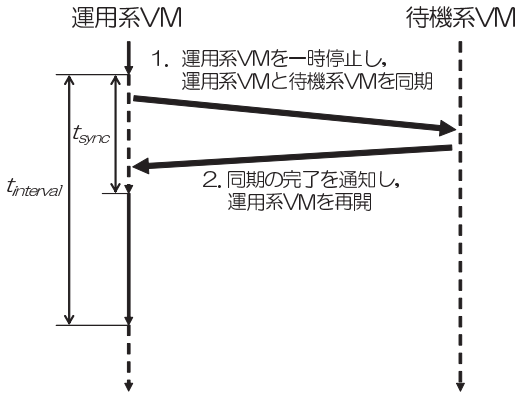

図 2　仮想マシン間の同期
Fig. 2　Procedure of VM synchronization.

4. 設　　計

図 2 に，仮想マシン間の同期の流れを示す．Kemari は，待機系の仮想マシンの状態を運用系の仮想マシンと同じにするために，まず運用系の動作を一時的に停止し，運用系と待機系の同期を開始する．続いて，待機系の状態が更新されたことを確認し，運用系を再開する．t_{sync} は同期にかかる時間，$t_{interval}$ は同期を行う間隔を表す．仮想マシン間の同期にかかるオーバヘッドは，$t_{interval}$ に対する t_{sync} の長さによって決まる．仮想マシン間の同期は，同期の準備，同期に必要なデータの転送，同期の完了待ちによって構成される．このうち，同期に必要なデータの転送が最も時間がかかると考えられる．同期に必要なデータの転送時間を短くするためには，転送するデータを小さくする必要がある．Kemari では，仮想マシンのイメージを対象に，前回の同期からの差分だけを転送している．

一方，同期の頻度を少なくし，間隔を長くすれば，同期のオーバヘッドを下げることができる．しかし，3.2 節で述べたように，待機系が障害発生時にサービスを継続するためには，待機系と外部状態の一貫性を保つ必要がある．したがって，同期のオーバヘッドを低くするためには，待機系と外部状態の一貫性を保ちつつ，同期の頻度を少なくしなければならない．

Kemari は，仮想マシンと外部の状態遷移を起こすイベントを，同期の契機として利用する．状態遷移には，決定的（deterministic）なものと非決定的（non-deterministic）なものがあり，仮想マシンと外部の間で発生するイベントは，非決定的な状態遷移を起こす可能

性がある．このため，イベントが伝播される前に，仮想マシンと外部の一貫性を保つように，運用系と待機系を同期する必要がある．一方，決定的な状態遷移は待機系で再現できるため，同期する必要はない．

本章では，仮想マシンに接続されるデバイスの種類とイベントの方向に着目し，Kemariが同期の契機とするイベントについて述べる．

4.1　同期の契機とするイベントの種類

仮想マシンに接続されるデバイスには，タイマ，ネットワーク，ストレージ，コンソールなどがある．仮想マシンとこれらのデバイスの間で発生するイベントが，同期の契機として必要かを考える．

まず，タイマのイベントについて説明する．仮想マシンがタイマからイベントを受け取ると，仮想マシンの状態は遷移する．しかし，実時間に依存したアプリケーションでなければ，受け取ったイベント以降の時刻に送られるイベントを用いて，同じ状態遷移を引き起こすことができる．Kemariは実時間に依存したアプリケーションを対象としないため，タイマのイベントでは仮想マシンを同期させない．

続いて，ネットワーク，ストレージ，コンソールのイベントについて説明する．仮想マシンがこれらのイベントを受け取ると，仮想マシンの状態は遷移する．反対に，仮想マシンがこれらのデバイスにイベントを送ると，デバイスの状態は遷移する．ここで起こる状態遷移は，やりとりされるイベントによって異なるため，ネットワーク，ストレージ，コンソールとのイベントでは，仮想マシンを同期させる必要がある．

4.2　同期の契機とするイベントの方向

4.1節で候補としたイベントのうち，ネットワークとストレージについて，仮想マシンから見た入力と出力のイベントに分け，各イベントを同期の契機とする必要性について述べる．なお，Kemariが想定しているサーバ環境ではコンソールのイベントは少ないため，イベントの方向に関する検討は行わない．

4.2.1　ストレージのイベント

図3に，Kemariで用いている同期方式と，運用系，待機系，ストレージの状態遷移を示す．運用系の仮想マシンからストレージに出力されるイベントは，ストレージに対する読み込み，または書き込みにより発生する．これらの操作により，ストレージはイベントの情報に基づいた状態遷移を行う．

運用系の仮想マシンからストレージへのイベントを契機に同期をしなかった場合，待機系はこのイベント以前の状態から再開することになる．しかし，待機系は運用系のストレージ

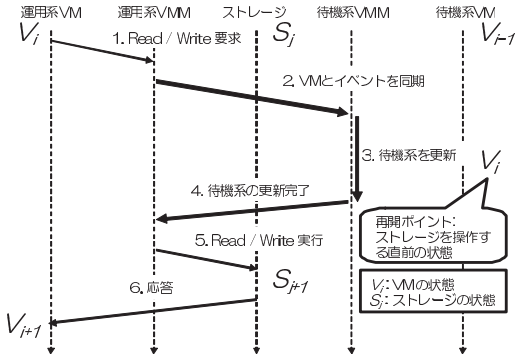

図3　ストレージへのイベントによる同期
Fig. 3　Synchronization on events to storage.

に対する操作を知らないため，待機系とストレージの関係は整合性のない状態になってしまう．このような状態では，待機系で動作するアプリケーションやOSが透過的に継続することができない．したがって，運用系の仮想マシンからストレージへのイベントを契機に，運用系と待機系の同期をとらなければならない．

一方，ストレージから仮想マシンへのイベントは，仮想マシンがストレージを操作した結果として発生する．前述した，運用系の仮想マシンからストレージへのイベントを契機に同期をさせた場合，待機系はストレージを操作する直前の状態から再開する．待機系は，運用系がストレージに行った操作を再び実行するため，運用系が受け取ったものと同じ応答をストレージから受けることができる．したがって，ストレージから仮想マシンへのイベントを契機に同期をさせる必要がない．

4.2.2　ネットワークのイベント

図4に，Kemariで用いている同期方式と，運用系，待機系，通信相手の状態遷移を示す．運用系の仮想マシンからネットワークへのイベントは，運用系の仮想マシンが通信相手に送信を行うときに発生する．このイベントにより，通信相手はイベントの情報に基づいた状態遷移を行う．

運用系の仮想マシンからネットワークへのイベントを契機に同期をさせなかった場合，待機系はこのイベント以前の状態から再開することになる．しかし，運用系がネットワークへイベントを送り，通信相手がそのデータを受信していた場合，待機系と通信相手の関係は整合性のない状態になってしまう．TCPを例にすると，運用系がACKを通信相手に送り，

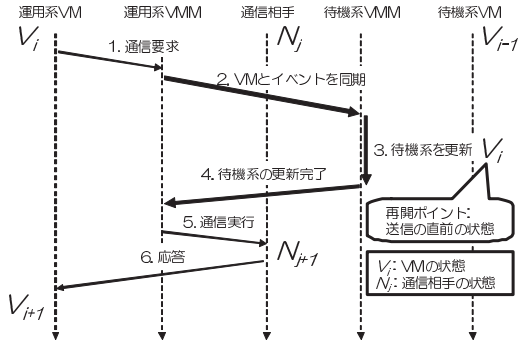

図 4　ネットワークへのイベントによる同期
Fig. 4　Synchronization on events to network.

通信相手がそれを受け取った状態である．この場合，通信相手は待機系よりも先の状態に遷移してしまっているため，待機系と通信相手のコネクションは回復できない．したがって，運用系の仮想マシンからネットワークへのイベントを契機に，運用系と待機系を同期させなければならない．

一方，運用系の仮想マシンがネットワークから受け取るイベントは，運用系の仮想マシンが受信を行うときに発生する．このイベントを契機に同期させなかった場合，運用系で受信したデータが待機系に存在しないことがある．これは，通信相手から見てパケットロスが起きている状態だが，TCP のように，信頼性を保証する通信プロトコルではプロトコルで対処し，UDP のように，信頼性を保証しないものではアプリケーションで対処することができるため，問題は生じないと考えられる．

なお，ネットワークから仮想マシンへのイベントのみを同期の契機とし，運用系と同じ状態遷移を待機系に行わせる手法も考えられるが，本論文では十分に検証していない．

5. 実　　装

Kemari は，オープンソースの仮想マシンモニタである Xen[2] をベースに実装されており，2009 年 7 月現在の最新版に加え，複数のバージョンに対応している．**図 5** に，実装したシステムの概観を示す．

Xen では，仮想マシンを実行する環境をドメインと呼ぶ．ドメインには，他のドメインを管理する権限を持ったドメイン 0 と，管理権限を持たないゲストがある．ドメイン 0 は

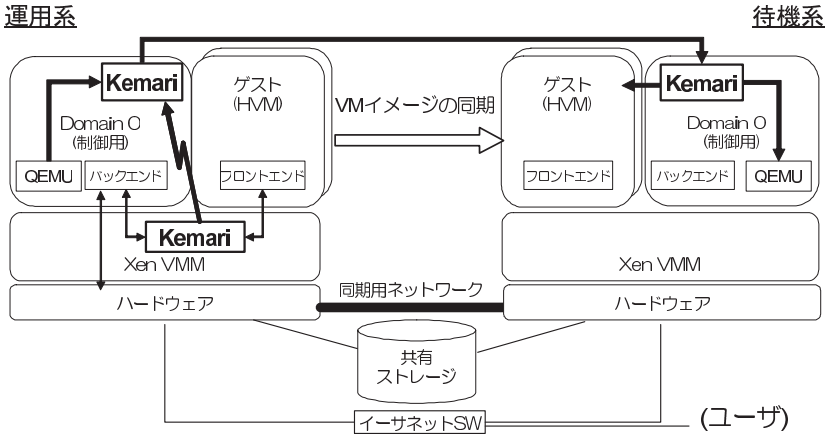

図 5　Xen に実装した Kemari の概観
Fig. 5　Architecture of Kemari implemented with Xen.

ゲストの操作に加え，I/O デバイスの仮想化も行う．Kemari が同期の対象とするのはゲストのみである．

Xen が提供するドメインで OS を実行するためには，Intel VT や AMD-V などのプロセッサの仮想化支援機構を利用し，OS をそのまま実行する方式と，OS の一部の命令をあらかじめ書き換えておく方式がある．前者は Full Virtualization，後者は Para Virtualization と呼ばれる，Full Virtualization で動作するドメインは HVM，Para Virtualization で動作するドメインは Domain U と呼ばれる．Kemari は現在 HVM を対象としている．

HVM で I/O を処理する方法には，Domain U でも使用される Para Virtulized Driver（PV Driver），QEMU[3] を拡張したエミュレーション，IOMMU を利用したパススルーがある．PV Driver は，ドメイン 0 が提供する仮想デバイスを扱う仕組みで，ゲストが持つフロントエンドと，ドメイン 0 が持つバックエンドで構成される．PV Driver は，I/O 性能がエミュレーションよりも高く，ライブマイグレーション[6] ができるなど，パススルーよりも柔軟性が高いという特徴を持ち，HVM を使用する際には広く利用されている．Kemari はこの PV Driver を前提としている．

5.1 構　　成

Kemari に必要なハードウェアは，2 つ以上の PC，ネットワーク，共有ストレージである．Kemari は HVM を対象としているため，PC には仮想化支援機構が必要になる．共有スト

レージは，保護対象となる仮想マシンの仮想ストレージを保存するもので，運用系と待機系からアクセスできなければならない．SAN で利用される，Fibre Channel（FC）や iSCSI のストレージに加え，ローカルディスクをミラーリングする DRBD [14] も利用できる．

Kemari は，運用系仮想マシンモニタのコンポーネント，運用系ユーザ空間のコンポーネント，待機系ユーザ空間のコンポーネントで構成される．なお，ユーザ空間はドメイン 0 のユーザ空間を指す．Xen の仮想マシンモニタはネットワークのドライバを持っておらず，運用系と待機系の通信にドメイン 0 を利用するため，仮想マシンモニタとユーザ空間に分けて実装した．保護する仮想マシンの送受信は，運用系ユーザ空間のコンポーネントと待機系ユーザ空間のコンポーネントが TCP で行う．Xen のライブマイグレーションも同様の理由で，ドメイン 0 のユーザ空間に実装されている．

運用系仮想マシンモニタのコンポーネントと運用系ユーザ空間のコンポーネントは共有のリングバッファを持つ．これは，データのやりとりを高速に行ったり，仮想マシンモニタとユーザ空間の切替えを少なくしたりするためである．

5.2　イベントチャネルのタップ

イベントチャネルのタップ機構は運用系仮想マシンモニタに実装されている．イベントチャネルとは，Xen で仮想化されたシステム内の通信を行う仕組みである．イベントチャネルを通じて，Xen とドメイン，または，ドメインどうしで通信することができる．イベントチャネルの両端にはポートがあり，イベントごとに送受信するポートが決まっている．仮想デバイスのフロントエンドドライバとバックエンドドライバ間の通信はイベントチャネルで実装されている．

イベントチャネルのタップ機構は Xen のイベントチャネルを拡張し，フロントエンドバックエンド間の通信のうち，あらかじめ指定されたイベントを捕捉する．捕捉するイベントは，ポート番号とイベントが流れる方向によって決まる．Kemari では，ユーザが Kemari を起動した際に，4 章で述べたイベントを捕捉し，仮想マシンの同期処理を開始するように設定している．なお，捕捉するイベントの設定および解除は，Kemari の動作中に行うこともできる．

5.3　仮想マシンの一時停止とページの抽出

指定されたイベントが捕捉されると，運用系仮想マシンモニタのコンポーネントは仮想マシンの同期処理を始める．4 章で述べたように，運用系と待機系の同期は，運用系の動作を停止したうえで行う．Xen には，ゲストを一時停止する方法として，suspend と pause の 2 つがある．

suspend はマイグレーションで用いられている仕組みで，ドメイン 0 の管理ツール，Xen，対象のゲストが協調し，ゲスト自身で転送できる状態にする．しかし，ゲストの状態遷移が発生するため，アプリケーションや OS から見た透過性が失われている．また，suspend は非同期に行われるため，イベントを捕捉した瞬間にゲストを停止させることができず，ゲストが停止するまでの間にアプリケーションや OS の状態が遷移する可能性がある．したがって，Kemari を実現する方法として，suspend は不十分である．

pause は，ゲストを Xen のスケジューリングから外すことにより，ゲストの動作を一時的に停止する仕組みである．Xen のスケジューラに対する操作だけで実現しているため，ゲストを瞬時に停止することができる．しかし，pause しただけでは，suspend のようにゲストを転送できる状態にはならない．Kemari は，shadow page table の特徴を利用することでこの問題を解決している．

shadow page table は，仮想化されたフレーム番号である pseudo-physical frame number（pfn）と物理的なフレーム番号である machine frame number（mfn）の変換を行う．shadow page table では，suspend のようにゲスト自身が転送できる状態にする必要がなく，pfn とそれに対応する mfn のデータを転送すればよい．HVM に加え，ドメイン U でも auto translated mode が指定された場合，ゲストは shadow page table を利用する．

運用系仮想マシンモニタのコンポーネントはゲストの一時停止後，転送対象となるページを抽出する．ページの抽出には，Xen のメモリ管理機構が持つ dirty bitmap を用いる．Kemari は高い頻度で転送を行うため，ゲストが書き込むページの数は限られ，局所性もある．そこで，dirty bitmap をビット単位ではなく，ワード長単位で調べ，ユーザ空間のコンポーネントと共有するリングバッファに書き込む．書き込みが完了すると，Xen とユーザ空間を結ぶイベントチャネルを通じて，運用系ユーザ空間のコンポーネントに通知する．

5.4　仮想マシンの継続的な差分転送

運用系ユーザ空間のコンポーネントは，Xen が管理するゲストのページと仮想 CPU のコンテキストに加え，QEMU が管理する仮想デバイスの情報と QEMU が書き込んだページを，待機系ユーザ空間のコンポーネントに継続的に転送する．QEMU は別プロセスで動作しているため，仮想デバイスの情報を保存するように，イベントチャネルで通知する．QEMU が書き込んだページも，仮想マシンモニタのコンポーネントと同様に，ワード長単位でフィルタする．

運用系ユーザ空間のコンポーネントは，初回の転送時にゲストのすべてのページを送る．その後は共有リングバッファからフィルタされた dirty bitmap を参照し，対象のページの

みを転送する．ページを転送するためには，Xen が参照を許可し，ページをユーザ空間にマップしなければならない．運用系ユーザ空間のコンポーネントは，初回の転送時にマップした情報を再利用することで，マップする回数を減らしている．

待機系ユーザ空間のコンポーネントは，運用系から送られるデータを一時バッファに読み込み，運用系に完了通知を送る．データの転送中に障害が検知された場合，一時バッファの内容は破棄され，ゲストの復元には，前の同期で送られた仮想マシンのイメージを使用する．

5.5　待機系バックエンドドライバの復元

待機系に送られたゲストのフロントエンドと待機系バックエンドは整合性がないため，仮想デバイスが使えない状態である．そこでゲストを再生する際，待機系バックエンドの設定を復元する．まず，フロントエンドとバックエンドで共有しているリングバッファのページをマップする．次に，BACK_RING_ATTACH マクロで共有リングの index を復元する．最後に共有リングの index の一部をバックエンドにコピーし，バックエンドの index を復元する．

6. 実　　験

Kemari を用いた際のオーバヘッドを測定するため，実験を行った．実験マシンの構成は 5 章で示したとおりであり，実験環境は**表 1** のとおりである．使用した仮想マシンモニタおよびホスト OS はそれぞれ Xen 3.4-testing および Xen 公式サイトで配布している Linux に Kemari 用の変更を加えたものである．実験では Kemari を用いない場合と Kemari を用いた場合の性能比較を行った．この際，Kemari に用いる同期用ネットワークに 1 GbE を用いた場合と，10 GbE を用いた場合を比較した．また，バックエンドに用いる共有ストレージとして，FC 接続されたストレージを用いた場合と DRBD によりサーバ間でローカルストレージのミラーリングを行ったものを用いた場合を比較した．なお，DRBD に利用するネットワークは Kemari の同期用ネットワークと共有している．実験では，まずシステムの全体的な性能を lmbench[18] を用いて測定した．それから Kemari が仮想マシンを同期する契機としているイベントであるネットワークとストレージの性能をそれぞれ netperf[11]，iozone[1] で測定した．

表 1　実験環境
Table 1　Experimentation environment.

サーバ	HP DL365G5
CPU	AMD Opteron 2356 (4core RVI 2.3 GHz) x 2
メモリ	PC2-5300 6G
NIC (GbE)	Broadcom NetXtreme II BCM5708
NIC (10 GbE)	Chelsio T310
FC アダプタ	QLogic QLE2460 4G FC
共有ストレージ	NetApp FAS 2020
ローカルストレージ	10 krpm 2.5 inch ホットプラグ SAS
仮想マシンモニタ	Xen 3.4-testing
ホスト OS	Debian Lenny (linux-2.6.18-xen.hg)
メモリ	1 GB
ゲスト OS	Debian Etch
メモリ	512 MB
仮想 CPU 数	1

6.1　基 本 性 能

Kemari を用いた際の基本性能を調べるために lmbench を用いて測定を行った．lmbench では，getpid の呼び出し時間を測定する `null calls`，int 型の足し算にかかる時間を測定する `int add`，fork システムコールにかかる時間を計測する `fork`，Hello World を表示する単純なプログラムの fork および exec にかかる時間を測定する `exec`，シェルから exec で用いた単純なプログラムを実行するのにかかる時間を測定する `sh`，ページフォルトによりファイルからページを読み込むのにかかる時間を測定する `page fault`，pipe によるコンテキストスイッチにかかる時間を測定する `context switch` を実行し，測定を行った．

表 2 は lmbench の結果である．この結果は 5 回測定を行った際の中央値を用いている．ここで，Base は FC 接続したホストにて Kemari を動かさず測定した値であり，このほかはすべて Kemari で同期を行っているホストにて測定した値である．Base 以外の行にある Kemari の後ろの括弧は同期用ネットワークの接続速度を示しており，それぞれ 10 GbE は 10 GbE での接続，1 GbE は 1 GbE での接続を示している．また，この後ろに続く FC，DRBD はそれぞれ共有ストレージとして FC を使う場合，DRBD によりミラーリングされたローカルディスクを使う場合を示している．

表 2 の結果では，null call や int add，page fault では Kemari の有無にかかわらずほぼ同等の性能を示したが，fork，exec，sh，context switch では Kemari で同期を行うことで 7%から 42.1%の性能劣化がおきた．前者のベンチマークと後者のベンチマークの違いはプロセス切替えを行うか否かである．Kemari では同期のために変更が行われたページをトラッキングしているが，後者のプログラムではプロセス切替えのために多くのページを変更するためそのオーバヘッドが発生したと考えられる．ここで，exec や shell など Kemari(1 GbE)+DRBD の方が Kemari(10 GbE)+DRBD や Kemari(1 GbE)+FC よりも

表 2　lmbench の測定結果（抜粋）
Table 2　Results of lmbench.

	null call [us]	int add [ns]	fork [us]	exec [us]	sh [us]	page fault [us]	context switch [us]
Base	0.24	0.44	114	349	1197	1.2845	8.2
Kemari(10 GbE)+FC	0.24	0.44	141	403	1280	1.2835	11.6
Kemari(10 GbE)+DRBD	0.24	0.44	161	415	1388	1.3145	11.6
Kemari(1 GbE)+FC	0.24	0.44	151	410	1335	1.3370	11.5
Kemari(1 GbE)+DRBD	0.24	0.44	162	413	1318	1.3239	11.6

性能が良い結果が出ている箇所があるが，これは実験中にデーモンなどが動いて外乱を起こした結果だと考えられる．実際，これらの結果の中央値以外の値を比較してみると，Base とこれらとの間には差があるものの，これらの値の間にはほとんど差がないことを確認している．

6.2　ネットワークの性能

Kemari によるオーバヘッドがネットワーク I/O に与える影響を調べるため，netperf でベンチマークを行った．ベンチマークでは 128 MB のデータをイーサネットスイッチでつながった外部のホストから TCP で送信するのに要した時間を測定し，そこからスループットを求めている．この際，実際にネットワークにつないで提供されるサーバのネットワークスペックを考慮し，100 Mbps で外部と接続を行って実験を行った．測定対象として 6.1 節と同様の環境を用い，それぞれ同じラベルをつけている．

実験の結果は**図 6** のとおりである．netperf の性能は同期用ネットワークの帯域が大きいほど Base に近づき，10 Gbps のネットワークを使った場合には Base の 90%程度の性能を達成している．1 Gbps のネットワークを使った場合でも Base の 66%程度の性能が達成された．netperf の結果は利用している共有ストレージの種類の影響を受けていないが，これは netperf がネットワークのスループットを測定するベンチマークであり，ディスク I/O をともなわないためであると考えられる．

netperf の性能は同期用ネットワークの帯域により変化するが，実際にどの程度のデータが流れているかを Kemari(10 GbE)+FC と Kemari(1 GbE)+FC の環境で調査した．調査方法は 1 秒ごとに ifconfig コマンドで NIC の統計情報をとる方法で行った．このとき，netperf の性能として 88 Mb/s 程度の性能を出している一方で，同期用ネットワークの最大通信量は 10 GbE で 1.3 Gb/s，1 GbE で 730 Mb/s であった．なお，10 GbE は実測値で 6.7 Gb/s，1 GbE は実測値で 940 Mb/s 程度出るが，GbE はこの実験を見る限りその性能を出しきれていない．Kemari は送信すべきデータをすべて送り終わった後，受信プログラムからの ACK を待つため，ネットワークのレイテンシがその性能に大きな影響を与える．実際，ping で 10 回の平均ラウンドトリップタイムを測定したところ 10 GbE では 0.090 ms に対し，1 GbE では 0.118 ms かかる．以上から，10 GbE を使った場合と 1 GbE を使った場合の netperf の結果の差は，ネットワークのレイテンシの差が原因であると考えられる．

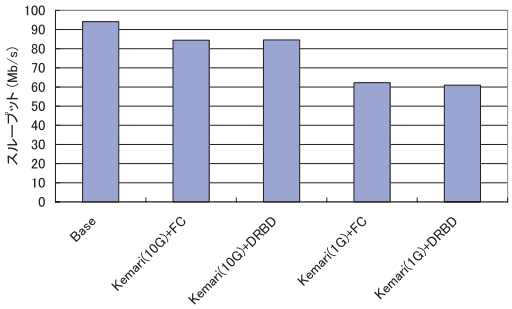

図 6　netperf の結果
Fig. 6　Results of netperf.

6.3　ファイル I/O の性能

次に，Kemari によるオーバヘッドがファイル I/O に与える影響を調べるため，iozone でベンチマークを行った．iozone が操作するファイルは PV ドライバで提供される仮想デバイス上に設け，ファイルシステムとして ext3 を用いている．また，測定のパラメータはファイルサイズを 128 MB，レコードサイズを 32 KB として測定した．iozone では O_SYNC オプションでファイルを開いて書き込みを行う同期書き込み，全データの書き込み後にのみ fflush および fsync システムコールによるフラッシュ行うバッファリング書き込みの 2 種類のベンチマークを 6.1 節と同様の測定対象に対して行い，それぞれ同じラベル付けを行った．

実験結果は**図 7**（同期書き込み）および**図 8**（バッファリング書き込み）のとおりである．

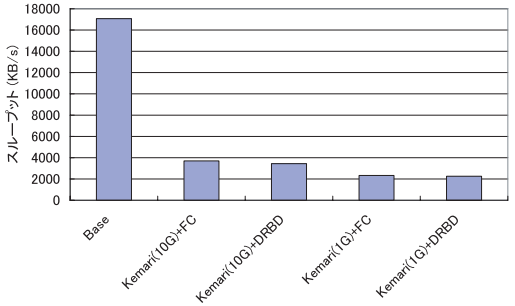

図 7　iozone（同期書き込み）の結果
Fig. 7　Results of iozone (synchronized write).

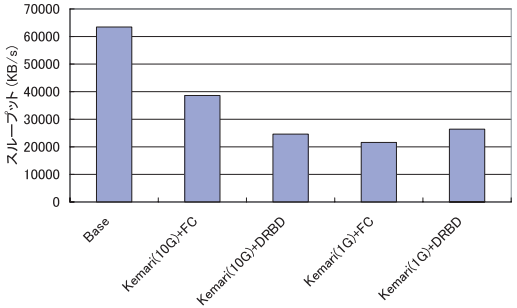

図 8　iozone（バッファリング書き込み）の結果
Fig. 8　Results of iozone (buffered write).

図 7 では Kemari を用いると著しく性能が劣化している．性能は同期用ネットワークに用いている線の帯域に依存し，10 Gbps で Base の 20%程度，1 Gbps で 13%程度の性能しか出ていない．これは Kemari がストレージへの出力を契機として同期をしているため，書き込みのつど仮想マシンの同期が起こり，I/O がその間停止するためであると考えられる．これに対し，ストレージへの書き込みを遅延させるバッファリング書き込み（図 8）では，これほどひどい性能劣化は起こっておらず，最悪の性能である Kemari(1G)+FC のときでも Base の 34%程度の性能が出ている．

同期書き込みの性能は同期用ネットワークの帯域により変化するが，実際にどの程度のデータが流れているかを 6.2 節同様，Kemari(10 GbE)+FC および Kemari(1 GbE)+FC の環境で調査した．その結果，iozone の性能として 4.0 MB/s および 2.4 MB/s 程度の性能を出す一方で，同期用ネットワークの最大通信量は 970 Mb/s，610 Mb/s であることを確認した．この結果はいずれもネットワーク帯域を限界まで使用していないことから，6.2 節同様，ネットワークのレイテンシがその性能劣化の一因であると考えられる．

6.4　checkpoint 方式との比較

Kemari は運用系の仮想マシンのイメージを，イベントドリブンで待機系に転送することで，低オーバヘッドかつ，幅広いハードウェアでの利用を実現している．特に，Kemari 方式の場合，CPU やメモリの負荷が中心のアプリケーションを実行している際のオーバヘッドが checkpoint 方式に比べて少なくなる．これを確認するため，6.3 節の Kemari(10G)+FC 構成を用い，iozone をディスクの同期を行わずに実行した際の性能，同期用ネットワークの通信量を測定した．なお，Kemari 方式ではイベントが発生しないとまったく同期を行わないため，オーバヘッドを適切に比較することができない．そこで，自発的なディスク書き込みを促すため，6.3 節のパラメータからファイルサイズを 512 MB に増やしている．実験では，Kemari そのもの（Kemari 方式）と Kemari を改造し，Remus にならってタイマで 25 ms ごとに割込みをかけて定期的に同期を行うようにしたもの（checkpoint 方式）とを比較した．

5 回測定した中央値を比較したところ，checkpoint 方式で 53 MB/s，Kemari 方式で 83 MB/s と Kemari 方式は checkpoint 方式の 1.6 倍程度の性能が出ている．なお，同期をしなかった場合の性能は 103 MB/s なので，10 GbE を使った Kemari 方式でも 20%程度の性能劣化がある．

実験中の同期用ネットワークの通信量の変化は**図 9** のとおりである．図では Kemari 方式で同期を行った場合の通信量（Kemari）と，checkpoint 方式で同期を行った場合の通信量（checkpoint）とを比較している．checkpoint 方式では定常的に 700 Mb/s 前後の通信量があるのに対し，Kemari 方式では定常時は通信がなく，突発的に 1.0 Gb/s 以上の通信量が発生している．これは，checkpoint 方式で 25 ms ごとに同期が行われるのに対し，Kemari 方式では OS によるダーティページのディスクへの書き戻しのタイミングでのみ同期が発生しているためと考えられる．また，図 9 の間に流れた総データ量は，Kemari 方式で 1.3 GB 程度，checkpoint 方式で 1.8 GB 程度と Kemari 方式の方が 30%程度少ない．このように，Kemari 方式ではイベントごとに同期させるため，同期用ネットワークの通信量が少なく，checkpoint 方式に比べて低いオーバヘッドを実現している．

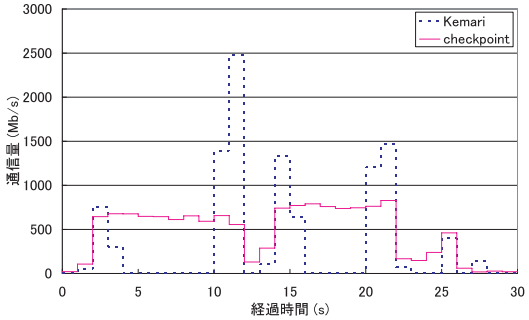

図 9　iozone（同期なし）時の同期用ネットワークの通信量
Fig. 9　Network traffics during iozone without sync.

6.5　継続性の確認

Kemari を用いることで障害発生時に処理が継続するか確認を行った．まず，Kemari で同期を行ったうえで，ベンチマークに用いた netperf，iozone を実行し，その間に iLO2（HP サーバ用リモート制御機構）を用いて電源のオフを行い擬似障害を発生させた．この際，待機系に運用系の故障を通知することでベンチマークが継続することを確認した．また，イーサネットスイッチでつながった外部のホストから OpenSSH で接続を行ったうえで xclock を動かし，上記の擬似障害発生，故障の通知により接続が切れることなく xclock の処理が継続することも確認した．OpenSSH には攻撃防止の目的で通信の整合性がとれない場合に接続を切る機能が搭載されているが，それに検出されることなく通信が継続できている．さらに，Windows XP SP2 を Kemari 上で動作させ，Windows Update 中に上記の擬似障害を発生，故障の通知を行い，問題なく処理が継続できることも確認している．現在の Kemari の実装は HVM を対象としているので，PV ドライバがあれば OS を問わず動作する．

なお，プロセッサなどの内部イベントを契機とした同期は行っていないため，そういったイベントに動作が依存するプログラムは保護対象とならない．具体的には，タイマや乱数生成器などに依存するプログラムは対象としていない．しかし，これらのプログラムは既存製品でも保護することができない．また，転送するページ数がプログラムによって異なるため，1 回の同期にかかる時間が予測できないが，これは Remus などのように定期的に同期を行ったとしても本質的には変わらない．したがって，同期の時間を保証する必要のあるプログラムについては，Kemari および Remus とも対応することができない．

7. 関 連 研 究

仮想マシンモニタを利用し，アプリケーションや OS の可用性を向上させる取り組みは活発に研究されている．Bressoud ら[4] は，仮想マシンの外部からのイベントや，仮想マシンで実行される命令のうち，演算結果が外部の状態に依存する命令を，運用系と待機系の仮想マシンモニタにシミュレートさせる lock step 方式を実装し，仮想マシン間の同期を行っている．しかし，仮想マシンモニタによるシミュレートの実装に PA-RISC[12] の recovery register を利用しているため，本研究が対象としている PC サーバには適用できない．lock step 方式を PC アーキテクチャに実装し，アプリケーションや OS から見て透過的に可用性を向上させる技術としては VMware FT[22] がある．また，対象とする OS が限られているが，同様の技術に everRun VM[17] がある．しかし，これらの製品の性能はまだ明らかになっていない．Stodden ら[20] も同様の試みを行ったが，実用には達していない．ExtraVirt[16] では，単一のシステムにある複数のプロセッサの同期をとることで，一方のプロセッサが故障したときに，他方のプロセッサが処理を引き継ぐという方式を提案しているが，プロセッサ以外の故障には対応できない．これらの lock step 方式に対し，Kemari は利用可能なプロセッサが多いこと，実装が簡易であること，さらに，仮想 SMP への対応が容易であることが優位性としてあげられる．一方，同期用ネットワークの通信量は lock step 方式の方が少ないと考えられる．

lock step 方式の技術を応用し，仮想マシンの動作を記録して，セキュリティやデバッグに役立てる研究も行われている．ReVirt[10] や VM-FIT[19] は仮想マシンモニタによるシミュレーションを PC アーキテクチャに実装し，セキュリティの研究に応用している．King らの研究[13] は，同様の仕組みをデバッグに役立てようという試みであり，Aftersight[5] はバグの自動検出も行う．しかし，これらの研究は，デバッグが複雑になる SMP 環境には対応していない．SMP-ReVirt[9] は，ReVirt[10] を SMP 環境に拡張した技術であり，実装には Xen を用いている．Kemari はこれらの研究とは目的が異なる．

Second Site[8] では，仮想マシンの checkpoint を継続的に外部ストレージに保存し，杉木ら[23] は，同様の技術を広域分散環境に適用している．しかし，運用系がどの時点で停止しても待機系が同じ状態になる同期方式ではないため，これらでは待機系が処理を透過的に引き継ぐことができない．Remus[7] や XSFT[15] では，仮想マシンからストレージやネットワークへの出力をバッファすることでこの問題を解決している．これらの checkpoint 方式に対し，Kemari は同期の頻度を最小限にするため，6.4 節で示したように，定常的な

同期のオーバヘッドが発生しない点で優位性がある．さらに，I/O をバッファリングする仕組みを必要としないため，実装がシンプルであるとともに，I/O が必要以上に遅延されないという点でも優れている．

Vizioncore vReplicator [21] や VMware HA [22] は，仮想マシンによる高可用性を実現する製品だが，同期の契機はポリシとしてユーザに委ねられており，アプリケーションや OS に依存した記述をする必要がある．

8. ま と め

本論文では，仮想マシン間の同期による耐故障クラスタリング技術，Kemari について述べた．Kemari は，イベントを契機に仮想マシンの同期を行うことで，アプリケーションや OS に依存しないで，障害発生時にサービスを継続することを可能した．また，実験により，Kemari によるオーバヘッドが最悪時に 90%，最良時に 10%程度であることを確認した．今後は，管理ツールや故障検知機構と組み合わせ，より実用的な検証を進めるとともに，性能について調査を深め，さらなる性能改善について検討を行う．Kemari は http://www.osrg.net/kemari から入手可能である．

参 考 文 献

1) IOzone. http://www.iozone.org/
2) 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, *19th ACM symposium on Operating systems principles*, pp.164–177 (2003).
3) Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *2005 USENIX Annual Technical Conference, FREENIX Track*, pp.41–46 (2005).
4) Bressoud, T.C. and Schneider, F.B.: Hypervisor-Based Fault Tolerance, *ACM Trans. Computer Systems*, Vol.4, No.1, pp.80–107 (1996).
5) Chow, J., Garfinkel, T. and Chen, P.: Decoupling Dynamic Program Analysis from Execution in Virtual Environments, *2008 USENIX Annual Technical Conference*, pp.1–14 (2008).
6) Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live Migration of Virtual Machines, *2nd USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, USA, pp.273–286 (2005).
7) Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. and Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication, *5th USENIX Symposium on Networked Systems Design and Implementation*, pp.161–174 (2008).
8) Cully, B. and Warfield, A.: SecondSite: disaster protection for the common server, *2nd Workshop on Hot Topics in System Dependability* (2006).
9) Dunlap, G., Lucchetti, D., Fetterman, M. and Chen, P.: Execution replay for multiprocessor virtual machines, *4th International Conference on Virtual Execution Environments*, pp.121–130 (2008).
10) Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A. and Chen, P.M.: ReVirt: enabling intrusion analysis through virtual-machine logging and replay, *5th symposium on Operating systems design and implementation*, Boston, MA, USA, pp.211–224 (2002).
11) Hewlett-Packard Company: Netperf. http://www.netperf.org/
12) Hewlett-Packard Company: PA-RISC architecture. http://www.hp.com/
13) King, S.T., Dunlap, G.W. and Chen, P.M.: Debugging operating systems with time-traveling virtual machines, *2005 USENIX Annual Technical Conference*, pp.1–15 (2005).
14) LINBIT: Distributed Replicated Block Device. http://www.drbd.org/
15) Lu, M. and cker Chiueh, T.: Fast Memory State Synchronization for Virtualization-based Fault Tolerance, *39th IEEE International Conference on Dependable Systems and Networks* (2009).
16) Lucchetti, D., Reinhardt, S.K. and Chen, P.M.: ExtraVirt: detecting and recovering from transient processor faults, *20th ACM symposium on Operating systems principles*, New York, NY, USA, pp.1–8 (2005).
17) Marathon Technologies, Corp.: everRun VM. http://www.marathontechnologies.com/
18) Mcvoy, L.W. and Staelin, C.: lmbench: Portable Tools for Performance Analysis, *1996 USENIX Annual Technical Conference*, pp.279–294 (1996).
19) Reiser, H.P. and Kapitza, R.: Hypervisor-Based Efficient Proactive Recovery, *26th IEEE Symposium on Reliable Distributed Systems*, pp.83–92 (2007).
20) Stodden, D., Eichner, H., Walter, M. and Trinitis, C.: Hardware Instruction Counting for Log-Based Rollback Recovery on x86-Family Processors, *3rd International Service Availability Symposium*, pp.106–119 (2006).
21) Vizioncore: vReplicator. http://www.vizioncore.com/
22) VMware, Inc.: VMware vSphere 4. http://www.vmware.com/
23) 杉木章義，大和崎啓，加藤和彦：広域分散環境のための仮想機械を利用したサービス協調複製基盤，情報処理学会論文誌：コンピューテングシステム，Vol.2, No.1, pp.1–11 (2009).

(平成 21 年 7 月 24 日受付)
(平成 21 年 11 月 14 日採録)

田村　芳明（正会員）

2003 年上智大学理工学部機械工学科卒業．2004 年米国 Northeastern University より M.S.（Computer Systems Engineering）．2005 年日本電信電話株式会社入社．以来，オペレーティングシステム，仮想化技術の研究に従事．現在，NTT サイバースペース研究所に所属．

柳澤　佳里（正会員）

2003 年東京工業大学理学部情報科学科卒業．2005 年同大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．2008 年同博士課程修了．同年日本電信電話株式会社入社．現在，NTT サイバースペース研究所に所属．博士（理学）．オペレーティングシステムの研究に従事．

佐藤　孝治（正会員）

1989 年慶應義塾大学理工学部数理科学科卒業．1991 年同大学大学院理工学研究科計算機科学専攻修士課程修了．同年日本電信電話株式会社入社．以来，分散システム，マルチメディアシステム，オペレーティングシステムの研究に従事．現在，NTT サイバースペース研究所に所属．日本ソフトウェア科学会会員．

盛合　　敏（正会員）

1983 年東北大学工学部電気工学科卒業．1988 年同大学大学院博士課程（情報工学専攻）修了．工学博士．同年日本電信電話株式会社入社．入社以来，プロトコル処理，インターネットシステム運用技術，分散 OS，リアルタイム OS（Real-Time Mach），セキュア OS の研究に従事．2001〜2004 年（株）ぷららネットワークス（現 NTT ぷらら）．2004 年より NTT サイバースペース研究所主幹研究員，高信頼 OS カーネル，仮想マシン，大規模分散システム，ユビキタスコンピューティング基盤の研究開発に従事．日本ソフトウェア科学会，電子情報通信学会，USENIX 各会員．The Linux Foundation Japan アドバイザリメンバ．