Vol.2010-OS-115 No.20
2010/8/4

KVMを利用した耐故障クラスタリング技術の開発

大村 圭[†1] 田村芳明[†1] 吉川拓哉[†2]
フェルナンド バスケス[†2] 盛合敏[†1]

本論文では，KVM を利用した仮想マシン同期技術の設計と実装，および実験について述べる．我々は，先行研究において，仮想マシンの同期による耐故障クラスタリング技術 Kemari を提案し，仮想マシンモニタ Xen に実装を行った．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停止することなく継続することができる．しかし，Xen は Linux カーネルに統合されていないため，Linux の最新ドライバなどを利用することができないという問題がある．そこで，本研究では，Linux 標準の仮想マシンモニタである KVM に，Kemari を実装した．実験の結果，模擬的なハードウェア障害発生時も，アプリケーションや OS を停止せずに継続できることを確認した．

Fault Tolerant clustering using KVM

Kei Ohmura ,[†1] Yoshiaki Tamura ,[†1]
Takuya Yoshikawa ,[†2] Fernando Vazquez [†2]
and Satoshi Moriai[†1]

This paper describes the design and implementation of virtual machine synchronization using KVM, along with experimental results of the system. In the previous work, we proposed Kemari, a cluster system that synchronizes virtual machines for fault tolerance that does not require specific hardware or modifications to software. Although we implemented Kemari based on Xen, Xen itself does not provide device drivers, and Dom0 which is responsible hasn't been integrated to the mainline Linux kernel, meaning we cannot use the latest hardware. To overcome this shortage, we implemented a prototype based on KVM, a virtual machine monitor included in the mainline Linux kernel.

1. はじめに

様々なサービスが，インターネットを通して多くのユーザに提供されており，システム停止によりサービスの継続が不可能となった際に，影響を受けるユーザ数は莫大なものとなる．そのため，システム停止を避けるために冗長化などの対策がされている．システム停止の原因としてハードウェア故障や人為的なミス，ソフトウェア障害などが挙げられるが，近年，仮想化によるサーバ集約により，一つのハードウェア上に複数の仮想マシンを動作させることが一般的なものとなりつつあり，ハードウェア故障のサーバへの影響は増大している．そのため，ハードウェア故障に対応するための高可用化技術が，ますます重要なものとなっている．

従来の高可用技術として，フォールトトレラント (FT) サーバやハイアベイラビリティ (HA) クラスタが挙げられる．FT サーバは主要なハードウェアを全て二重化し完全に同期させることで，ハードウェア故障を隠蔽することができる．しかし，二重化のための高価な専用ハードウェアが必要であり，柔軟な構成が難しく，また，ローカル環境に限定されてしまい，ディザスタリカバリ対策に使うことはできない．一方，HA クラスタは，安価な IA サーバを複数用意するだけで利用可能であるが，アプリケーションに応じた構成を組む必要があり，集約された仮想マシン上の全てのアプリケーションに対応しなければならない．

我々は，この課題を解決するために，先行研究において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 を提案した[14]．Kemari は，プライマリ・セカンダリと 2 台のサーバを用意し，それぞれ仮想マシンを動作させ，プライマリで動作する仮想マシンの特定のイベントを契機に，セカンダリの仮想マシンと同期させることで，仮想マシンの一貫性を保つことができる．仮想マシンの実行状態をコピーするため，アプリケーションや OS に依存せずに同期が可能であり，また，障害発生時にセカンダリは，瞬時に仮想マシンを再開させ，停止直前の状態からサービスを提供することができる．

しかし，先行研究において利用した仮想マシンモニタ Xen[1] は，Linux カーネルに統合されていないため，Linux の最新ドライバなどを利用することができない．一方，Linux 標準の仮想マシンモニタである KVM[8] が，近年，注目を浴びている．KVM は 2007 年に

†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
†2 日本電信電話株式会社研究企画部門オープンソースソフトウェアセンタ
NTT Open Source Software Center

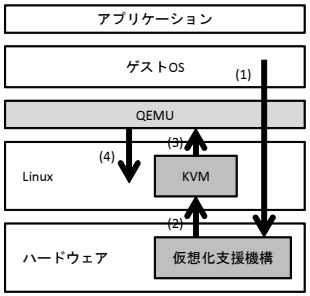

図 1 KVM の構成と，I/O 処理の流れ
Fig. 1 Architecture of KVM and I/O process.

Linux カーネルに統合された仮想マシンモニタであり，Linux カーネルが提供する機能が利用可能である．そのため，Linux に追加された新機能は，即座に利用が可能であり，仮想マシンモニタとして KVM を利用する利点は大きい．そこで，本研究では，KVM を利用した耐故障クラスタリング技術の設計と実装を行った．

本論文では，2 章で仮想マシン同期機構の設計と実装，3 章で実験結果および考察，4 章で関連研究に触れ，5 章でまとめと今後の課題について述べる．

2. KVM における仮想マシン同期機構の設計と実装

本研究では，Xen 版 Kemari と同様に，イベントドリブンでプライマリとセカンダリの仮想マシンを同期する方式を採用し，以下の設計方針において実装を行った．

- ユーザ空間に閉じた設計
- 既存のライブマイグレーション実装と親和性が高い設計

ユーザ空間に閉じることで，デバッグが容易，拡張性の高い設計となっている．また，既存のライブマイグレーションの拡張機能とすることで，ライブマイグレーションに機能追加などの変更があった場合に，容易に利用が可能な設計とした．

2.1 KVM とは

KVM (Kernel-based Virtual Machine) は，ホスト型仮想マシンモニタとして Linux に実装されており，ハードウェアによる仮想化支援機構や，PC エミュレータとして，QEMU[2] を KVM 用に変更して利用することで，完全仮想化を実現している．

KVM の構成と I/O 処理の流れを図 1 に示す．KVM は Linux カーネルモジュールとして実装されており，Linux カーネルは KVM モジュールをロードすることで，仮想マシンモニタになることができる．I/O エミュレーションには QEMU を利用しており，ゲスト OS は QEMU によりエミュレートされたデバイスを利用することができる．また，KVM のゲスト OS は，QEMU プロセスとして実行されるため，ホスト上の一プロセスとして扱うことができる．ゲスト OS が，ディスク I/O やネットワーク I/O を行おうとすると，(1) プロセッサの仮想化支援機構によりトラップされ，(2) 処理が KVM に移行する．(3) KVM は I/O アクセスの解析を行い，処理を QEMU へ依頼する．(4) QEMU は エミュレーションにより I/O 命令をシステムコールに変換し，ホストカーネルに実行を依頼し，実際のドライバによって処理が実行される．

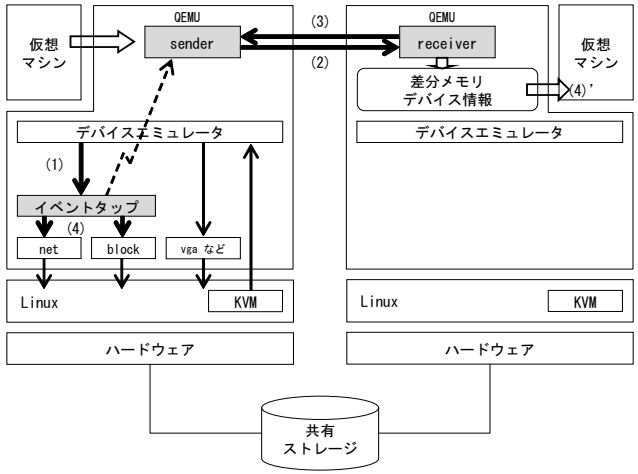

図 2 KVM に実装した Kemari の概観
Fig. 2 Architecture of Kemari implemented with KVM.

2.2 仮想マシン同期機構の概要

KVM に実装した，仮想マシン同期機構の概要を図 2 に示す．本システムの処理は，(1) 同期の契機となるイベントの捕捉，(2) 仮想マシンイメージの差分転送，(3) 受信完了の Ack，(4) イベントの発行，(4)' 受信したデータを仮想マシンに反映，という流れで進む．

Kemari は主に，同期の契機となるイベントを捕まえる (1) イベントタップ機能と，プライマリとセカンダリの仮想マシンの状態を同一に保つための，(2) 継続的な仮想マシンの同期機能からなる．イベントタップ機能は，QEMU による I/O エミュレーション時に，一

貫性を保つのに必要なイベントのみを捕まえるように実装している．同期機能は，ライブマイグレーションの拡張機能として実装してあり，メモリやデバイスの save，load といった機能は，基本的には既存のコードを利用している．また，継続的な同期や，高速化のための手法を施している．

2.3 QEMU による I/O エミュレーションのトラップ

イベントタップ機能は，QEMU により I/O エミュレーションが行われた際に，特定のイベントをトラップするように実装した．捕捉するイベントは仮想マシンと外部デバイスの状態遷移を，非決定的に起こすものであり，タイマ，ネットワーク，ディスク，コンソールなどが考えられる．Kemari は実時間に依存しないアプリケーションを対象とし，また，サーバ用途を想定しているため，タイマおよびコンソールのイベントに関しては，今回は，考慮していない．捕捉するイベントは，仮想マシンから，ディスクおよびネットワークへの出力のみとしている．

本実装では，図 2 で示しているように，ディスク，ネットワーク出力イベントのトラップを QEMU の net，block のレイヤで行っている．そのため，ゲスト OS が利用するドライバの種類 (e1000，virtio など) に関わらず，イベントの捕捉が可能であり，Xen 版 Kemari が PV ドライバ限定であったのとは異なり，ゲストがどのようなデバイスドライバを使用していたとしても対応することができる．そのため，ゲスト OS に特別なドライバをインストールさせることなく，同期が可能であるという利点がある．

Kemari の同期のタイミングは，I/O エミュレーションの途中であるため，プライマリで障害が発生した場合，セカンダリは I/O エミュレーションを完了させる必要がある．しかし，KVM は I/O エミュレーションの際に，仮想 CPU の RIP レジスタを更新してから，QEMU に依頼する実装となっており，セカンダリで仮想マシンを再開させる際に，RIP レジスタが進んでしまうという問題がある．この場合，QEMU は次の I/O のエミュレーションを行おうとするため，プライマリで，エミュレーションの途中であった I/O がリプレイされないという問題がある．そこで，I/O イベントをトラップする際に，書き込み命令を save し，セカンダリで load する関数を新たに追加した．これにより，セカンダリで仮想マシンを再開する際に，load したデータを基に，エミュレーションの途中であった I/O をリプレイさせることができ，正常にサービスを引き継ぐことを可能としている．

2.4 KVM のライブマイグレーションを利用した仮想マシンの同期

同期の契機となるイベントを捕捉した際に，仮想マシンイメージの差分をプライマリからセカンダリに転送する機構について述べる．

本機構は，KVM のライブマイグレーション機能により，プライマリとセカンダリの仮想マシン状態を合わせる初期同期と，前回同期時の差分データを転送し，常に仮想マシンの状態を合わせるようにライブマイグレーション機能を拡張した，反復同期からなる．

2.4.1 KVM におけるライブマイグレーション

ライブマイグレーションは，仮想マシンを停止する時間を最小限にして，ホスト間で仮想マシンを移動する手法である[4]．動作している仮想マシンのメモリは，常に変更されていくので，送信済みのメモリが変更されていないかを確認し，変更が生じた場合は，差分メモリを送る．メモリはページ単位で管理・転送され，変更が生じたページのことをダーティページと呼ぶ．KVM におけるライブマイグレーションでは，ダーティページと，CPU などのデバイス情報を全て転送し，転送が成功すると，移動先で仮想マシンを再開させる．ネットワークの接続などは保たれたままなので，ユーザにとっては透過的に仮想マシンの移動が可能である．以下に，ライブマイグレーション時の，移動元 (sender)，移動先 (receiver) のそれぞれの処理の流れを示す．

- sender
  - (1) receiver との接続の確立．まずは，全てのページをダーティページとし，ダーティログトラッキングを開始する．
  - (2) ダーティページが一定量以下になるまで，繰り返し転送を行う．また，一定時間内に転送できるデータ量は制限されており，制限量を越えると，仮想マシンに制御を移し，データの転送を一時停止する．
  - (3) ダーティページが一定量以下になると，仮想マシンを停止して，残りダーティページを全て転送．次に，CPU などを含むデバイス状態を，それぞれに対応する save 関数を呼び出して転送していく．
  - (4) 全データを転送後，ダーティログトラッキングの終了などの後処理を行う．
- receiver
  - (1) sender との接続の確立や，データを格納するバッファの準備などの前処理を行う．
  - (2) sender からデータを受信し，有限長のバッファに格納する．
  - (3) まず，バッファからヘッダを読み出し，ヘッダから，デバイスの種類 (メモリ，CPU など) を特定し，各デバイスに対応した load 関数を呼び出していく．
  - (4) 各 load 関数では，バッファからデータを順次，読み出していき，メモリやデバイスに，そのデータを反映していく．
  - (5) バッファの中身を全て読み終えて，かつ同期終了フラグの読み出しが完了してい

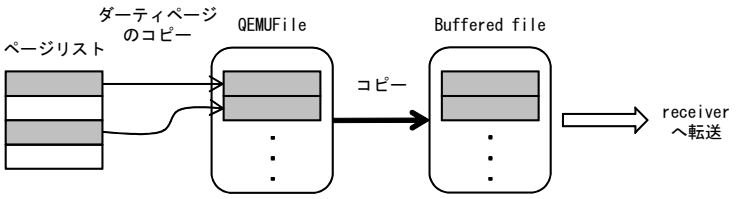

図 3 ダーティページの転送プロセス
Fig. 3 Transfer process of dirty pages

ない場合は，(2) に戻る．((3)，(4) のフェーズでバッファが空になると，その都度，データの受信を行う．)

以上の流れで，ライブマイグレーションは進んでいく．sender も receiver と同様に図 3 の，QEMUFile 構造体に有限長のバッファを保持しており，ダーティページなどの各データは，一旦 QEMUFile にコピーされ，一杯になると，receiver に転送し，クリアする．また，sender の (2) ダーティページを繰り返し転送するフェーズにおいて，制限量以上のデータを転送すると，処理途中のデータは，Buffered file というメモリ領域に一時的に格納され，仮想マシンに制御が移る．仮想マシンからライブマイグレーション機構への制御の移行は，タイマベースとなっており，100ms 経過すると移行する．制御が戻ると，Buffered file のデータはフラッシュされ，繰り返しダーティページを転送するフェーズを再開する．

2.4.2 継続的な仮想マシンの同期

本機能の実装の多くは，前述したライブマイグレーション機能を利用している．ライブマイグレーションと大きく異なる点は，sender に関しては，同期の契機となるイベント時にプライマリの仮想マシンを停止し，ダーティページとデバイス状態を一度に全て転送し，receiver からデータ受信完了の Ack を受け取った後に，仮想マシンを再開する点である．一方，receiver に関しては，全てのデータを受信し終わった後に，受信完了の Ack を sender に送信し，データの反映を行うという点である．このようにすることで，任意の時点での仮想マシン状態を確実に保存し，いつ障害が発生しても，セカンダリの仮想マシンを矛盾なく再開することができる．本機能を実現するために，行った実装について述べる．

Kemari における転送プロトコル ライブマイグレーションで扱う，QEMU における仮想マシンスナップショットの，フォーマットの概要を図 4 に示す．QEMU ヘッダは，QEMU のバージョンなどを示す．セクションタイプは，ライブマイグレーションのフェーズを表す．デバイスタイプは，後に続くデバイス情報が，CPU なのか，メモリなのかといった

図 4 仮想マシンスナップショットのファイルフォーマット
Fig. 4 File format of VM snapshot

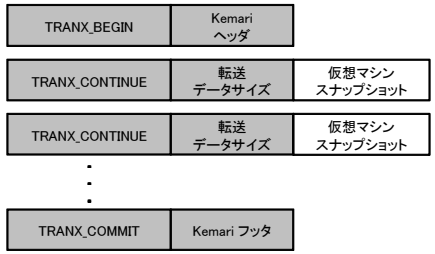

図 5 Kemari のデータフォーマット
Fig. 5 Data format of Kemari

ことを表しており，sender，receiver 共に，デバイス情報の save，load は，それぞれに応じた save，load 関数が存在する．receiver は，読み出したデバイスタイプに応じた，load 関数を呼び出し，データの反映処理を行う．Kemari の場合，データの受信・反映途中で，プライマリの仮想マシンが障害により停止してしまうと，プライマリとセカンダリの仮想マシンの一貫性が壊れてしまい，セカンダリで正常に仮想マシンを再開させることができない．これを避けるためには，receiver は十分に大きなバッファを用意して，全データを受信して，バッファにコピーした後に，データの反映を開始する必要がある．もし，全データを受信する前に，プライマリが故障した場合は，バッファのデータを破棄し，前回同期時の状態から仮想マシンを再開させることができる．receiver でのデータバッファリングを実現するために，receiver が，データを受信する際に，Kemari 用の関数でフックするようにした．Kemari 用関数では，全データを受信するまで，受信処理が続けられる．この際，受信完了を判断するために，データの終わりを示す QEMU_VM_EOF を読み込む必要があるが，load 関数は，受信と同時に，データの反映を行ってしまうため，利用することができない．そこで，sender が送信するデータに，図 5 に示すような，Kemari 用のヘッダなどを追加した．トランザクションタイプは，TRANX_BEGIN，TRANX_CONTINUE，TRANX_COMMIT の 3 つがあり，それぞれ，Kemari による同期の開始，CPU，メモリなどの転送，同期の終了を示す．特に，CONTINUE の場合は，これから，転送するデータのサイズと，図 4 の仮想マシンスナップショットが続く．これにより，receiver は，受信するデータサイズや同期の終了を判断できる．また，受信したデータの内，Kemari ヘッダなどは破棄し，仮想マシンスナップショットのみ，バッファにコピーすることで，ファイルフォーマットを維持している．そのため，全データを受信

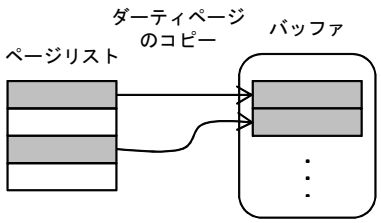

図 6 ダーティページをバッファにコピー
Fig. 6 Copy dirty pages to buffer.

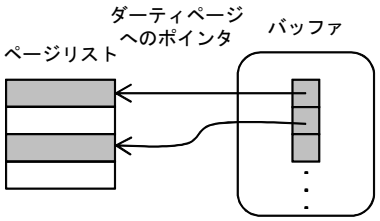

図 7 ダーティページへのポインタを参照
Fig. 7 Reference pointer to dirty pages.

後，バッファの中身を load 関数で読み出していくだけで，データの反映が可能である．

Buffered file のパススルー ライブマイグレーションでは図 3 に示すように，転送制限量を超えると，Buffered file にデータをコピーし，仮想マシンに制御を戻すことで，一定時間内の転送データ量を調整すると共に，転送によって仮想マシンが長時間ブロックされるのを回避している．しかし，Kemari では，ある同期のタイミングでの仮想マシンイメージをプライマリとセカンダリで共有することで，一貫性を保っているので，一回の同期で全データを転送し，その後，仮想マシンに制御を戻す必要がある．そこで，Kemari の場合は，QEMU の Buffered file をパススルーし，一回の同期で，確実に receiver にデータを転送するようにした．

writev() の利用 また，図 6 のように，ダーティページなどを一度，QEMUFile のバッファにコピーし，バッファが一杯になると，転送を行う．このバッファのサイズは，32KB である．例えば，ダーティページのサイズが 4KB/ページとして考えた際に，一回の write 命令で 8 ページしか送ることができない．そのため，write() システムコールを複数回呼ぶ必要があり，Kemari のように，一度に全てのデータを転送する方式にとっては，無駄なオーバヘッドを生じさせてしまう．そこで，図 7 に示すように，ダーティページを QEMUFile のバッファにコピーするのではなく，バッファには，各ダーティページのポインタを持たせ，writev() システムコールにより，転送する方式にした．これにより，無駄なコピーを減らすと同時に，システムコールを呼ぶ回数を減らし，効率の良い転送を実現している．

3. 実　　験

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．

表 1 実験環境
Table 1 Experimentation envirionment.

サーバ	HP DL360G6
CPU	Intel Xeon Quad Core 2.66GHz x 4
メモリ	PC2-5300 96G
NIC (10GbE)	Ghelsio 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	KVM (kernel 2.6.33)
QEMU	qemu.git
ホスト OS	Debian Lenny (kernel 2.6.33)
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	1G
ゲスト OS	Debian Etch
メモリ	512MB
仮想 CPU 数	1

また，KVM を利用しているため，PC には仮想化支援機構が必要になる．実験環境を表 1 に示す．

実験に使用した QEMU と Xen には，Kemari 用の変更が加えられている．Kemari による同期なしで測定した結果をそれぞれの仮想マシンモニタごとに，KVM-Base，Xen-Base として表し，Kemari による同期を行っている場合は，KVM-Kemari，Xen-Kemari として，実験を行った．実験結果の値は，5 回測定した際の中央値を用いている．

3.1 基 本 性 能

Kemari によるオーバーヘッドが，ゲスト OS の基本性能に与える影響について lmbench を用いて測定した．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測定する 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．結果を表 2 に示す．KVM 版，Xen 版共に，int add, page fault では，Kemari の有無に関わらずほぼ同等の性能を示したが，fork, exec，sh，context switch では，KVM 版は，3 %から 15 %程度，Xen 版は 8 %から 27 %程度の性能劣化が起きた．

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]
KVM-Base	0.37	0.17	75.2	257	855	0.75	2.42
KVM-Kemari	0.19	0.17	77.4	267	881	0.77	2.85
Xen-Base	0.32	0.17	59.4	189	681	0.56	1.23
Xen-Kemari	0.32	0.17	64.9	201	712	0.64	1.69

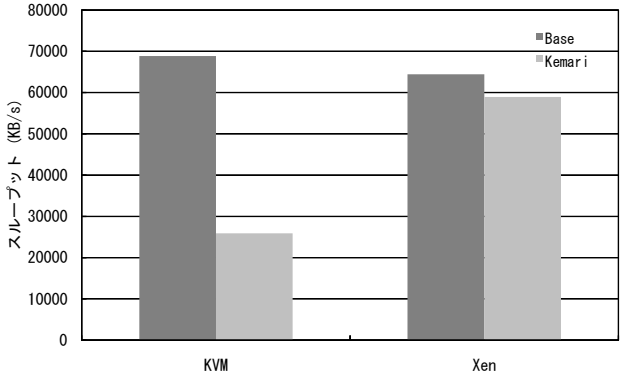

図 8 iozone (バッファリング書き込み) の結果
Fig. 8 Results of iozone (buffered write).

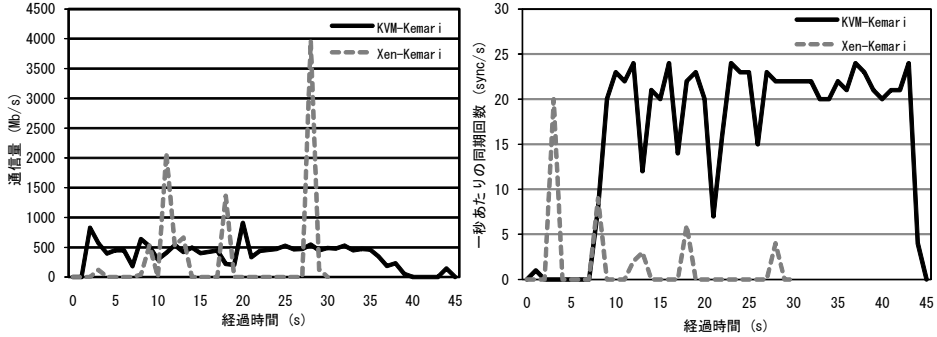

図 9 iozone (バッファリング書き込み) 実行時の同期用ネットワークの通信量
Fig. 9 Network traffics during iozone (buffered write).

図 10 iozone (バッファリング書き込み) 実行時の同期回数
Fig. 10 Number of synchronization during iozone (buffered write).

これは，プロセス切り替えのために，多くのページを変更するため，転送するダーティページが増加したためと考えられる．KVM 版の方が，性能劣化が小さいという結果が得られたが，KVM-Base の性能が，Xen-Kemari の性能より低く，元々の性能が異なるため，比較は難しい．また，null calls に関しては，測定値にばらつきがあり，KVM-Kemari は，2 倍近く性能が向上しており，Xen-Kemari に関しても，Base より良い性能を示すことがあった．これの原因は特定できていないが，Kemari のような処理をした際に，仮想マシンモニタのスケジューリングの優先度が上がった，などが考えられる．

3.2 ファイル I/O 性能

iozone を用いて実験を行い，ファイル I/O 性能を測定した．測定パラメタとして，ファイルサイズを 128MB，レコードサイズを 32KB としている．実験結果を図 8 に示す．KVM-Kemari は，Base に対して，38 %程度の性能しか出ていない．一方，Xen-Kemari は，91 %程度と高い性能を示している．これを解析するため，Kemari により発生する，同期用ネットワークの通信量と，1 秒あたりの同期回数を測定した (図 9，10)．KVM-Kemari が，頻繁に同期を行うことで，定期的に 500Mb/s 程度の通信量が発生しているのに対し，Xen-Kemari はほとんど同期を行わないため，一回の同期による通信量が大きく，場合によっては 4Gbps 程度の通信が発生している．総通信量は KVM-Kemari は 2.1GB 程度なのに対し，Xen-Kemari は 1.2GB 程度である．今回使用した，同期用ネットワークは 10Gbps と十分に帯域が大きいため，頻繁に同期を行うよりも，一回の同期で大きなデータを転送する方が効率が良いため，KVM-Kemari より Xen-Kemari の方が，良い性能を示したと考えられる．

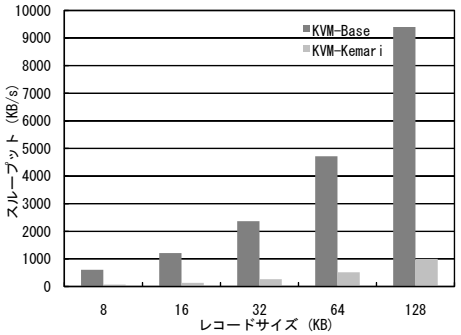

図 11 各レコードサイズでの iozone (同期書き込み) の結果
Fig. 11 Results of iozone (synchronized write) at each record size.

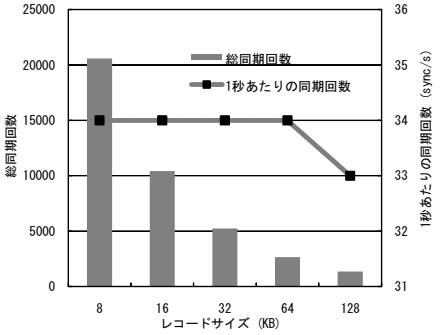

図 12 各レコードサイズでの総同期回数と 1 秒あたりの同期回数
Fig. 12 Number of synchronization at each record size.

KVM-Kemari の方が，同期が頻繁に発生してしまう原因として，イベントを捕捉するレイヤが異なることが原因と考えられる．Xen 版は，PV ドライバを繋ぐイベントチャネルでイベントを捕捉していたが，KVM 版は，より汎用性を高めるため，QEMU の net，block レイヤでイベントを捕捉している．おそらく，それぞれで I/O をためるキューの長さが異なり，Xen の PV ドライバがある程度キューにたまってから，I/O 処理を Dom0 に依頼しているのに対して，QEMU ではより小さい単位で I/O 命令が発行され，結果として net，block レイヤでのイベント捕捉回数が多くなったのではないかと考えられる．

3.3 レコードサイズによる性能および同期回数の変化

iozone での実験で，ファイルサイズを 128MB に固定し，レコードサイズを 8KB から 128KB に変化させて，同期書き込みをさせて，実験を行った．図 11 は各レコードサイズでのスループット，図 12 は各レコードサイズで，iozone を実行させてから，終わるまでに，発生した Kemari による総同期回数を示す．

図 11 から，Kemari の有無に関わらず，レコードサイズを大きくすることでスループットが向上していることがわかる．Kemari 有りの場合は，レコードサイズを大きくすることで，同期頻度が小さくなるため，同期オーバヘッドが小さくなり，より性能が向上すると考えられたが，Kemari 無しの場合と同程度の性能向上率となっている．同期回数の削減が，性能向上に大きく寄与しなかったことの原因として，同期回数を減らすことで，一回の同期における転送量が増加してしまい，結果的に，同期オーバヘッドは大きく変わらなかったためだと考えられる．次に，図 12 を見ると，レコードサイズを大きくすることにより，総同期回数が反比例して小さくなっている一方で，1 秒あたりの同期回数に大きな変化がないことがわかる．頻繁に同期を行うことで，同期処理がリソースを消費してしまっているため，スループットは低く抑えられてしまう．同期書き込みのようなワークロードで性能を向上する為には，単位時間あたりの同期回数を削減することが必要だと考えられる．

4. 関 連 研 究

仮想マシンモニタを利用して，サーバの可用性を高める研究は活発にされており，商用化されているものも複数存在する[11)10)]．仮想マシン同期手法については，主に，lock-step[3)] 方式と，checkpoint[5)] 方式が挙げられる．

lock-step 方式は，プライマリとセカンダリに同じ入力を与え，各仮想マシンで同様の処理を行わせることで，仮想マシンの状態を同一に保つことができる．各仮想マシンで計算を行わせるため，同期に必要なデータ量は小さくて済むが，マルチコアへの対応が難しく，利用可能なプロセッサが限定されてしまうという問題がある．また，lock-step 方式は，仮想マシンの実行履歴を記録し，再現が可能であることから，デバッグの効率化やセキュリティの向上といった観点からも研究がされており，Revirt[6)]，軽量仮想計算機モニタ[15)]，川崎ら[13)] の研究が挙げられる．これらの研究は Kemari とは目的・手法が異なる．

一方 checkpoint 方式は，継続的にプライマリの仮想マシンイメージ (CPU，メモリ，ストレージなど) をセカンダリに転送することで，仮想マシンの状態を同一に保っている．同期に必要なデータ量が比較的大きくなってしまうが，ライブマイグレーションの応用で実現可能であり，利用できるプロセッサが制約されるといったこともない．しかし，仮想マシンの外部デバイスと状態を合わせて，同期する必要があり，Kemari のようなイベントドリブンによる手法と，I/O バッファリングといった手法がある．イベントバッファリングでは，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し，同期のタイミングで出力を行わせることで，仮想マシンに対して，外部デバイスの状態遷移が先に進まないようにしている．しかし，バッファリングにより I/O が遅延されるという問題がある．Zhu[12)] らは，ダーティページの管理機構およびゲスト領域の map 処理を最適化することで，40 %から 120 %程度の性能改善を行っている．これらの手法は有効ではあるが，測定でネットワークおよびディスクのバッファリングは行っておらず，チェックポイント方式でボトルネックとなる，I/O 性能の低下については解決できていない．

また，ライブマイグレーションのオーバヘッドを削減する研究もいくつか行われている．CR/TR-motion[9] では，Revirt[6] の機構をライブマイグレーションに適用することで，継続的に転送されるデータ量を 90 %程度削減すると共に，ライブマイグレーション中のオーバヘッドも 8 %程度に抑えているが，仮想 SMP 環境における問題は解決されていない．Hirofuchi[7] らは，ポストコピー方式のライブマイグレーションを提案し，KVM に実装している．従来のライブマイグレーションでは同じアドレスのページを何回も送るのに対し，ポストコピー方式では移動する瞬間にアクセスしてるページを中心に送るため，ローカリティの高いアプリケーションでは，マイグレーションにかかる時間も短く，転送されるページ数も少ないという利点がある．しかし，Kemari の場合は，切替前に全てのページが転送されている必要があるため，提案されている手法を適用することはできない．

5. おわりに

本論文では，KVM を利用した仮想マシンの同期技術の設計と実装，性能測定を中心とした実験について述べた．本実装は，ユーザ空間に閉じた実装により，デバッグが容易であると共に，既存のライブマイグレーションのコードに大きな変更を加える必要がないため，シンプルで拡張性の高い設計となっている．また，イベントタップ機能を QEMU の net，block レイヤに実装することにより，PV ドライバを必要とする Xen 版とは異なり，e1000，virtio など，QEMU で利用可能な全てのデバイスのイベントを捕捉することができ，ゲスト OS に全く手を加えることなく，同期することを可能とした．実験では，lmbench により計測した Kemari による性能劣化は，Xen 版と同程度であった．一方，ファイル I/O は，Kemari 無しに対して，38 %程度の性能しか出ておらず，Xen 版と比較して大きく性能が落ちており，Xen 版が少ない頻度で同期しているのに対して，KVM 版はかなり高い頻度で同期しているのが明らかになった．これは，イベントを捕捉するレイヤの違いによると考えられ，適用可能なゲスト環境と性能がトレードオフであると予想される．今後は，汎用性を保ちつつ，ファイル I/O を中心とした性能の改善が必要である．また，KVM と Xen で，Base となる性能が大きく異なっていたため，これらに関しても，今後詳細な調査が必要である．

参考文献

1) 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).
2) Bellard, F.: QEMU, a fast and portable dynamic translator, *ATEC'05: Proceedings of the annual conference on USENIX Annual Technical Conference*, CA, USA, USENIX Association, pp.41–46 (2005).
3) Bressoud, T. C. and Schneider, F. B.: Hypervisor-Based Fault Tolerance, *ACM Trans. Computer Systems*, Vol.4, No.1, pp.80–107 (1996).
4) Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live Migration of Virtual Macihines, *2nd USENIX Symposium on NetworkedSystems Design and Implementation*, Boston, MA, USA, pp.273–286 (2005).
5) 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).
6) 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).
7) Hirofuchi, T., Nakada, H. and Sekiguchi, S.: Enabling Instantaneous Relocation of Virtual Machines with a Lightweight VMM Extension, *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp.73–83 (2010).
8) Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux virtual machine monitor, *OLS'07: The 2007 Ottawa Linux Symposium*, pp.225–230 (2007).
9) Liu, H., Jin, H., Liao, X., Hu, L. and Yu, C.: Live migration of virtual machine based on full system trace and replay, *18th International Symposium on High Performance Distributed Computing*, pp.101–110 (2009).
10) Marathon Technologies, Corp: everRun VM, http://www.marathontechnologies.com/.
11) VMware, Inc: VMware vSphere 4, http://www.vmware.com/.
12) Zhu, J., Dong, W., Jiang, Z., Shi, X., Xiao, Z. and Li, X.: Improving the Performance of Hypervisor-Based Fault Tolerance, *24th IEEE International Parallel and Distributed Processing Symposium*, pp.1–10 (2010).
13) 川崎　仁，追川修一：SMP を活用した Primary/Backup モデルによるリプレイ環境の構築，情報処理学会研究報告， Vol.2010-OS-113, No.12 (2010).
14) 田村芳明，柳澤佳里，佐藤孝治，盛合　敏：Kemari：仮想マシン間の同期による耐故障クラスタリング，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, pp. 13–24 (2010).
15) 竹内　理，坂村　健：軽量仮想計算機モニタを利用した OS デバッガのロギング & リプレイ機能の提案，情報処理学会論文誌， Vol.50, No.1, pp.394–408 (2009).