Vol.2010-OS-113 No.12
2010/1/28

SMP を活用した Primary/Backup モデルによるリプレイ環境の構築

川 﨑 仁[†1] 追 川 修 一[†1]

オペレーティングシステム（OS）の開発時やシステムの運用時にカーネルパニックによる障害が発生することがある．こうした場合，一般的に障害時点から遡ってデバッグを行うことになるが，これは過去の状態を推定することであり容易ではない．そこで本研究では，SMP 環境上にロギング＆リプレイを適用した手法を提案する．本手法では，SMP 環境上でロギング用とリプレイ用の 2 つの OS を起動し，リプレイ用 OS の実行を遅らせることで常時過去の状態を保持する．障害時にはリプレイ用 OS を利用して実行を再現することで，デバッグの効率化が可能であり，特に再現性の低いタイミングに依存するバグのデバッグが可能になる．ロギング＆リプレイ機構の設計と実装を行い，安定動作には至っていないがゲスト OS が起動することを確認した．

Kernel Debugging System using SMP environment by Primary/Backup Model

Jin Kawasaki[†1] and Shuichi Oikawa[†1]

When an operating system (OS) fails to operate a computer system, an OS kernel panic occurs. Then, we need to debug the kernel program going back from the failing condition. It is, however, difficult to debug the kernel in that way because we must assume the past condition of the OS. In this paper, we propose a method of debugging using the logging & replay functionality by utilizing an SMP environment. On this method, we boot up an OS on the logging side that records logs during the OS execution, and run an OS on the replay side that replays the OS execution by using the logs. The replay side always has a past state of the logging side by maintaining a time difference between the both sides. Our approach can provide an efficient debug method. Especially, it makes it much easier to debug timing bugs that are poorly reproducible. We designed and implemented the logging & replay functionality, and performed some experiments while a guest OS is booting up.

1. はじめに

オペレーティングシステム（OS）の開発時やシステムの運用時にカーネルパニックによる障害が発生することがある．そのような場合，パニックの原因を特定し修正するためにデバッグを行うことになるが，これは過去の状態を推定することであり容易ではない．なぜなら，一般的に障害時点から遡ってデバッグを行うことになるが，利用できる情報がクラッシュダンプデータなどパニック後のものに限られるためである．特に，再現性の低いバグに起因する場合やダンプデータが破壊されている場合にはさらに困難なものとなり，デバッグ作業は経験や知識を積み上げた場合においても依然難しい作業となっている．また，近年のカーネルの多機能化や複雑化は，バグ混入の確率を高めると共にデバッグをさらに困難にさせる方向にも影響を与えている．加えて，高機能なデバイスが様々登場しており，それらのデバイスドライバ開発においても同様に，カーネルデバッグが困難なものとなっている．このように，カーネルデバッグの効率化の必要性はますます高まっている．

そこで本研究では，仮想マシンモニタを応用し，カーネルパニック後の状態だけではなくカーネルパニック前の状態も提供することで，効率的なデバッグ環境を実現する手法を提案する．既存の研究においても同様の手法が提案されているが，本研究では新たに現在一般的になっているマルチプロセッサの SMP 環境を活用し，Primary/Backup モデル[1)] により，2 つの OS を時間差を設けて実行する方法を提案する．2 つの OS をそれぞれ Primary と Backup に割り当て，それらを時間差を設けて実行することで，Primary のカーネルパニック時に Backup 側ではカーネルパニックが起きる前の状況を作り出すことができる．そして，この状況から，デバッグを進めていくこととなる．

実際に本研究で核となるロギング＆リプレイ機能を実装し，2 つのゲスト OS を同一状態を保ったまま起動できることを確認した．現在のところ安定動作には至っていないが，実験としてロギング＆リプレイを利用したゲスト OS 起動時のログの分析と性能測定を行った．結果，ゲスト OS 起動時に関しては，ロギング＆リプレイを使用しない場合と比較して，性能低下が小さいことが分かった．

本稿においては，初めに提供するシステム全体の概要について述べる．次に，システムの中心となる仮想マシンモニタと，仮想マシンモニタが提供するロギング＆リプレイに関して

†1 筑波大学システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, Tsukuba University

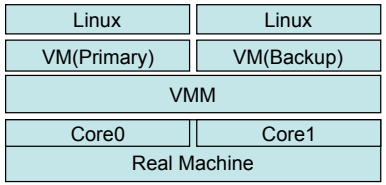

図 1 システムの概要図
Fig. 1 Figure of system

設計と実装を示す．その後，ゲスト OS 起動時のログの分析と性能測定を行う．そして，関連研究を示し，最後にまとめを述べる．

2. システム概要

本研究では，単一計算機の SMP 環境上で Primary と Backup の 2 つのゲスト OS を実行する．そして，Primary/Backup 間でロギング&リプレイを実現することにより，Backup 側で Primary の過去の状態を作り出す．デバッグ時にはその過去の状態を利用する．本研究で提案するシステムの概要図を**図 1** に示す．

2.1 機　　能

本システムが実現する機能は次の通りである．まず，Primary でゲスト OS を起動し，Backup は Primary よりも遅れて実行を開始する．このとき，Primary と Backup は完全に同一の挙動を示すようにする．その結果，Primary のカーネルが不具合を起こし停止した時点において，Backup 側は不具合が発生する前の状態（つまり，停止した時点の Primary にとって，過去にあたる状態）となっている．デバッグでは，この時点の Backup 側の OS を利用し，そこから命令をステップ実行により進めていくことでカーネルパニックが発生するまで処理を継続していく．また，このリプレイは何度でも繰り返して実行することが可能である．このようにして，カーネルパニックが発生する前の情報を利用した効率的なカーネルデバッグの方法を提供することができる．

2.2 要　　求

前節の機能を実現するためには，Primary と Backup 間で仮想マシンが同一の状態に保たれなければならない．もし同一の状態でなければ，命令実行に対する結果が異なることになり，正確なデバッグが不可能なためである．2 つの仮想マシンを同一の状態に保つためには以下の要件を満たす必要があることが，既存研究から明らかになっている[1)–3)]．

- イベントを同一のタイミングで処理し，同一の結果を得ること
- 命令列を同一の順序で処理し，同一の結果を得ること

イベントのうち非同期的なイベントについては，それが発生したタイミングと内容を Primary から Backup へと通知し，Backup 側で非同期的なイベントを同一のタイミングと同一の内容で OS に通知することで対応可能である．一方，命令列については，命令列だけを考えれば決定的な処理であり，一度実行を開始すると同一の順序で実行される．しかし，前述の非同期的なイベントが命令列の実行中に発生するために，環境によって命令実行順が変わってしまう可能性がある．この問題も，前述の非同期的なイベントに対する処理で解決可能である．また，同一の結果という点に関しては，デバイスの値の取得命令など実行結果が外部要因に影響を受ける命令では，実行環境で結果が変わる可能性があるため，命令の実行結果を Primary から Backup へと通知し，Backup 側がその命令の実行時に同一の結果を得られるように対応することが求められる．

2.3 実 現 方 法

本研究では，以上の点を踏まえて，仮想マシンモニタを利用することでこれを実現する．2 つの仮想マシンをそれぞれ SMP のコアの 1 つに割り当て，それぞれの仮想マシンを Primary と Backup とに対応させることで，SMP を活用した Primary/Backup モデルを構築する．Primary では OS の実行を通常に行ってログの取得と保存（ロギング）を行い，Backup ではログの読み出しと OS の再現実行（リプレイ）を行う．ここで述べたログとは，OS の動作を再現するために必要な情報のことであり，本稿では実行履歴と呼ぶ．実行履歴の詳細については後述する．また，対象とするアーキテクチャは一般的に広く利用されている IA-32 アーキテクチャとし，対象とする OS は Linux を想定する．

また，仮想マシン間で非同期的なイベントと命令に関する情報を転送することが必要となるため，本研究では Primary と Backup 間を共有メモリを用いて結合し，情報を Primary から Backup へと通知する方法を採用する．既存の手法においては，別マシン上で OS 状態を再現するためにネットワークを利用して転送を行う方法や，同一計算機上であっても同時に OS 状態を再現するのではなく後に実行履歴を元に再現を行う方法がとられてきた．ネットワークを利用する手法では，転送速度やエラー処理を検討し対応することが必要となるが，提案手法では SMP を活用し共有メモリを利用するため転送速度は高速でありエラー処理の必要も無い．また，全ての実行履歴を保存する必要もなく，実行履歴のサイズを削減できる．

以下の節では，まず本研究で提案するシステムの基礎となる仮想マシンモニタについて述

べ，続いて Primary から Backup へのイベントの通知を実現する仕組みについて述べる．

3. システムの設計と実装

本研究の目的を達成するために，ハードウェア仮想化支援機構を用いた仮想マシンモニタを開発している．また，この仮想マシンモニタに Primary と Backup 間の通信機能と，ロギング＆リプレイ機構を実装している．これらの点について本節で詳細を述べる．

3.1 仮想マシンモニタ：Sesta

本研究では，仮想マシンモニタ Sesta[4] をベースとして，SMP を利用して Primary/Backup モデルを実現した仮想マシンモニタを開発する．Sesta は，筆者の所属する研究室においてスクラッチから実装が行われた仮想マシンモニタである．また，Sesta は単一の仮想 CPU をゲスト OS に対して提供する仮想マシンモニタであり，単一の仮想マシンを提供する．そして，OS のマイグレーションを目的として開発されており，ホスト OS を必要とせず直接実機上で動作する Type1 型の仮想マシンモニタである．また，CPU，メモリ，一部のデバイスドライバの仮想化を実現しており，仮想化の実現のためにハードウェア仮想化支援機構 Intel VT-x[5),6)] を利用している．

Intel VT-x を利用することで，特権命令実行時のゲスト OS から仮想マシンモニタへの遷移やイベント発生時のハンドリングを比較的容易に実装することが可能である．これは，特権命令の実行や例外の発生により，自動的にゲスト環境から仮想マシンモニタの環境へと遷移し，仮想マシンモニタ側で処理を行うことができるためである．なお，Intel VT-x の MMU の仮想化に対するハードウェアによる支援はまだ一部のプロセッサに限られているため，特権命令と例外による仮想マシンモニタへの遷移と仮想メモリを利用した仕組みによりメモリの仮想化を実現し，ゲスト OS と仮想マシンモニタのアドレス空間の切り分けを実現している．

仮想メモリ機構では，CPU にアドレス変換表であるページテーブルを登録し，メモリアクセスの際にそのページテーブルをたどることで物理アドレスを求める．さらに，Intel VT-x では，仮想マシンモニタが利用するページテーブルとゲスト OS が使用するページテーブル（シャドウページテーブル）の 2 つを準備し，仮想マシンモニタとゲスト OS の環境を遷移する際にそれらを切り替えることが可能である．これらの仕組みを用いて，ゲスト OS と仮想マシンモニタ間のアドレス空間の切り分けを次のように実現している．

まず，ゲスト OS がページテーブルを変更する命令を実行すると，Intel VT-x の機能により仮想マシンモニタに遷移が起きる．仮想マシンモニタ側では，ゲスト OS が CPU に登録しようとしたページテーブルを走査し，シャドウページテーブルに設定する．そして，ゲスト OS に遷移する際に，このシャドウページテーブルを登録することで，仮想マシンモニタによるゲスト OS のメモリアクセスを管理を実現している．Sesta ではゲスト OS の物理アドレス空間を定められたサイズに制限しており，その物理メモリ空間を仮想メモリ空間と 1 対 1 で対応づけている．また，仮想マシンモニタをゲスト OS の物理アドレス空間より上位の領域に配置することで，仮想マシンモニタとゲスト OS の切り分けを実現している．

加えて，Sesta は，デバイスドライバについても仮想化を実現しており，一部のデバイスについてはデバイス状態を含め仮想化している．しかし，その他のデバイスについては単に I/O 命令を直接デバイスに通知するようになっている．

3.2 Sesta の拡張

本研究のシステムでは，2 つのプロセッサで個別に OS を実行しなければならないが，ベースとする Sesta はマルチプロセッサをサポートしていないため機能の追加が必要である．また，2 つの OS を同一の物理メモリ空間上で実行するために，アドレス空間の仮想化もそれらに対応させなければならない．これら 2 点とデバイスドライバに関連する拡張について，以下の節で順に述べる．

3.2.1 複数コアの利用

IA-32 アーキテクチャでは，初めに立ち上がるプロセッサを BSP と呼び，それ以外のプロセッサを AP と呼んでいる．BSP は電源投入時に自動的に立ち上がるが，AP は自動的には立ち上がらず，BSP から IPI（interprocessor interrupts）を送ることで立ち上げる必要がある．

また，レガシーな IA-32 アーキテクチャでは，割り込みコントローラとして i8259A が利用されてきた．アーキテクチャのマルチプロセッサ化により，割り込みコントローラもマルチプロセッサに対応した Advanced Programmable Interrupt Controller（APIC）が用いられている．APIC には，外部からのイベントを初めに受け取る I/O APIC と，各プロセッサごとにイベントを受け取る Local APIC とがある．イベント伝達の流れとしては，外部からのイベントはまず I/O APIC に伝わり，事前に設定しておいた情報に従い，各プロセッサの Local APIC に伝わる．そして，Local APIC が各プロセッサにイベントを通知する．なお，電源投入時には，互換性維持のためにレガシーな i8259A が用いられるため，マルチプロセッサ環境に対応するには APIC を利用するための準備を行った後に，APIC を有効にしなければならない．

Sesta では，各仮想マシンで提供する仮想 CPU は 1 つであり，本研究の拡張においても

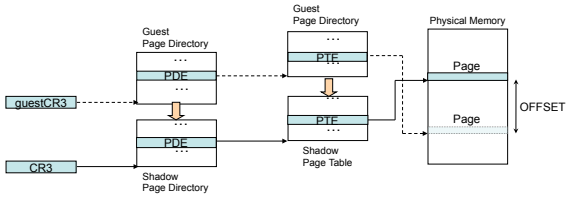

図 2 Backup におけるメモリ管理の概要図
Fig. 2 Figure of managing memory on Backup

この点は変更しない．それで，実機においては割り込みコントローラとして APIC を利用するが，ゲスト OS に提供する仮想マシンでは仮想 i8259A を提供することとした．

また，Backup では Primary の実行結果を元に動作を再現するため，外部イベントを直接受け取り処理する必要が無い．そこで，I/O APIC からは Primary の Local APIC のみに外部イベントを通知する．Primary から Backup への外部イベントの情報転送については，後述する．

3.2.2 メモリ管理

本研究で提案するシステムでは，同一の物理メモリ上にゲスト OS のための物理アドレス空間を複数持つため，別のゲスト OS の物理アドレス空間を書き換えてしまわないように，それらを保護しなければならない．加えて，Primary の動作を再現するために，Backup 用の物理アドレス空間上でゲスト OS が動作したとしても，Primary の物理アドレス空間上で動作しているものと変わらない透過的なアクセスを実現しなければならない．これらの課題を，前述のシャドウページングの仕組みを応用することで実現する．

まず，物理アドレス空間上の下位オフセット分を Primary に，続くオフセット分を Backup に割り当てる．したがって，Primary の物理アドレスと対応する Backup の物理アドレスは，Primary のアドレスにオフセットを加えたものとなる．それで，Backup 側においては，シャドウページングの設定の際に，ゲスト OS により登録されているアドレスにオフセットを加えたものを変換後の物理アドレスとして登録する．また，Backup 側でゲスト OS の登録しようとするページテーブルを走査し辿っていく際にも，オフセットを加えたアドレスを用いて辿っていく．このようにすることで，Backup 側のゲスト OS は自身が Primary と異なる物理アドレス空間で動作していることを認識すること無く動作する．Backup におけるメモリ管理の概要図を**図 2** に示す．

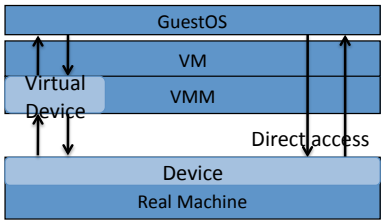

図 3 デバイスドライバの処理
Fig. 3 Processing of device drivers

3.2.3 デバイスドライバ

本研究においては，Backup で Primary のゲスト OS の動作を再現するために，Primary でのゲスト OS のデバイスアクセスとその結果を Backup に転送し再現する必要がある．再現する際に，Backup のゲスト OS がデバイスドライバにアクセスしようとする場合には，ゲスト OS から仮想マシンモニタに遷移し，実デバイスにアクセスした結果ではなく Primary から転送されたデバイスアクセスの結果をゲスト OS に通知することになる（**図 3**）．これを実現するためには，デバイスの仮想化が必要であり，本研究では Sesta の仮想デバイスの仕組みを活用することにする．従って，本システムで利用できるデバイスは，Sesta が仮想デバイスを提供しているデバイスとなる．Primary から Backup へのデバイスアクセス結果の情報転送については，後述する．

耐故障性を念頭に置いて Primary/Backup モデルを実現する場合には，Primary と Backup のそれぞれが実デバイスにアクセスする必要ある．本研究で提案するシステムにおいて Primary と Backup の OS は，前述の通り仮想デバイスにはアクセスするが実デバイスにはアクセスする必要が無い．

3.2.4 プロセッサ間通信

ここまで述べてきたように，Primary と Backup の OS を同一の状態を保って実行するには，Primary から Backup への情報転送が必要となる．そこで，Sesta を拡張しプロセッサ間通信を実現する．プロセッサ間通信の実現には，SMP の特徴を活かし共有メモリを用いることにする．共有メモリの詳細に関しては，3.3.3 で述べる．

3.3 ロギング＆リプレイ

本節では，Primary の動作を再現するために Backup 側に転送される実行履歴と，実行履歴の取得，転送，再現方法について述べる．これらの設計に関しては，既存研究で明らか

になっている知見を参考にしている[1)–3)].

3.3.1 実行履歴の種類

Primary での動作を Backup で再現するためには，Primary と Backup 間で同一の状態を保つことが求められる．そのためには，前述の通り 2 つの仮想マシンにおいて割り込みなどの非同期的なイベントを同一のタイミングで処理し同一の結果を得ること，また命令列を同一順序で処理し同一の結果を得ることが必要である．そして，これらを実現するために，Primary から Backup に対して必要な実行履歴を送る必要がある．具体的には，

- 非同期的なイベントの種類と起きたタイミング，その内容
- 結果が外部要因に影響を受ける命令と，その結果

を実行履歴として転送することが求められる．

本稿では，プロトタイプの開発を行うため最小限の構成として，タイマおよびシリアルの 2 つを実行履歴取得の対象である外部要因として設計を行う．このうちシリアルは，ゲスト OS の操作とその実行結果を表示するために，コンソールの入出力として利用する．

通常のシステムにおいては，最小限の構成としては上記に加えてディスクデバイスが存在する．本研究においては，initramfs を利用することでディスクデバイスを利用することなくゲスト OS を実行している．initramfs は，Linux カーネルの起動時に利用される小さなルートファイルシステムのことであり，起動時に物理メモリ空間上に展開される．本来はその後の起動シーケンスにおいて，ディスクデバイスのファイルシステムがマウントされることになるが，initramfs をそのまま利用している．ゲスト OS に対して十分大きな物理メモリ空間を割り当てることで，実験環境として実用上の問題は無い．

実行履歴として，本研究では以下の情報のうちイベントごとに必要なものを実行履歴として保存する．

- イベントの種類
  Backup で発生したイベントの種類を識別するために使用する．具体的には，割り込みなのか命令なのか，割り込みであればどの種類の割り込みなのか，命令であればどの種類の命令なのか，といった点を識別する値を保存する．
- イベントの発生した命令アドレス
  Backup に対して，Primary と同一の命令アドレスでイベントの通知や結果の適用を行うために必要である．
- 命令の実行結果
  非決定的な命令を実行した際に得られた結果を保存し，Backup で同一の命令が実行された時にこちらの値を返すために利用する．
- ECX
  ECX をカウンタとして利用する命令に対して，Primary と同一のタイミングで Backup に対してイベントを通知するために必要である．
- 分岐回数
  分岐命令を経由すると同一アドレスを複数回通過することになり，命令アドレスだけでは同一のタイミングを特定することができない．そこで，分岐命令の回数を保存しておき，分岐命令の回数と命令アドレスにより同一のタイミングを特定する．

3.3.2 実行履歴の取得

Primary 側では，非同期的なイベントが発生した際に，それを契機としてゲスト OS から仮想マシンモニタに実行が遷移する．そこで，仮想マシンモニタが実行履歴を作成し，共有メモリにそれを格納する．Backup 側の仮想マシンモニタでは，実行履歴を共有メモリから読み出し，ゲスト OS に適切に通知する．このとき，Primary と Backup とでイベント処理後の仮想マシン状態を同一のものにするため，イベントは Primary で起きたのと同一のタイミングで同一の内容を通知する．これを実現する方法としては，イベントを決められたタイミングでまとめて適用する方法[1)] や分岐命令数をカウントすることによりタイミングを求める方法が知られている[2),3)].

本研究においては，後者の手法を採用し，IA-32 アーキテクチャの Performance Monitoring Counter（PMC）を利用する．PMC は様々なハードウェアイベントを計測するために利用できるカウンタ群であり，複数のハードウェアイベントを同時に計測することができる．IA-32 では分岐命令数を直接カウントする機能は提供されていないが，同等の機能として retired-branch をカウントする機能が提供されている．これは，分岐命令や割り込み，例外の様に，命令ポインタを更新するような処理をカウントする機能である．また，投機的に実行されたが後にキャンセルされた命令はカウントアップの対象に入らない．この機能により，分岐回数をカウントすることで，イベント発生のタイミングを確定させる．実行履歴の取得では，結果が外部要因に影響を受ける命令についても同様にイベントとして扱う．本手法における実行の流れは以下に示すようになる．

(1) PMC を設定し，分岐回数のカウントを開始する
(2) イベント発生時に分岐回数を含む実行履歴を取得し，共有メモリに格納する
(3) 1 に戻って処理を継続する

Backup 側でのイベントの再現については，3.3.5 で取り上げる．

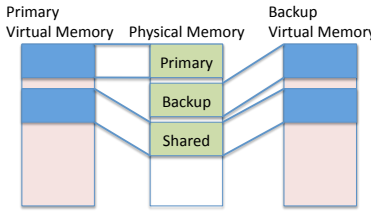

図 4 共有メモリを利用した転送
Fig. 4 Transfer using shared memory

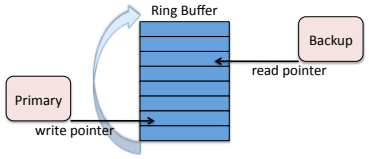

図 5 リングバッファの概要図
Fig. 5 Figure of the ring buffer

3.3.3 実行履歴の転送

Primary 側で実行履歴を取得し，それを Backup 側へと転送するために，前述の通り共有メモリを利用する．そのため，物理メモリ上に Primary と Backup の両方からアクセスできる同一の物理アドレス空間を確保する．これにより，SMP である特徴を活かし，同一マシン上のメモリを共有した高速な情報転送が可能となる．共有メモリ領域を確保した場合のメモリアーキテクチャを**図 4** に示す．

3.3.4 リングバッファ

実行履歴の転送は共有メモリを通して行うが，共有メモリへアクセスする仕組みとして，リングバッファを実装する（**図 5**）．リングバッファとは，バッファ領域をリング状に見立て，切れ目の無いアクセスを可能にした仕組みである．通信の分野で広く用いられており，仮想マシンモニタ Xen においては仮想マシン間の通信に採用されている[7]．

リングバッファの処理を高速化するために，個々のエントリを固定長として定義する．したがって，実行履歴もその種類とは無関係に固定長で保存することとし，空の領域は 0 でパディングする．今回，1 つのエントリは 32 byte に設定して実装を行った．

また，Backup 側での実行の最大遅延時間はリングバッファ全体のサイズに影響を受けるため，遅延時間を長くするためにはリングバッファ全体のサイズを大きくすることになる．一方，Primary と Backup は時間差はあるものの同時にゲスト OS を実行するため，Primary 側において実行を開始してからカーネルパニックが起きるまでの時間がリングバッファ全体のサイズに影響することは無い．

3.3.5 実行履歴の再現

Backup では，共有メモリから実行履歴を読み出すと，それをゲスト OS に通知することにより仮想マシン状態を Primary のものと同一の状態に保つ．まず，非同期的なイベントについては次のように処理する．

(1) 実行履歴を共有メモリから取得する
(2) 実行履歴の命令アドレスを元に，その命令アドレスまで実行する
(3) 分岐回数と ECX を比較する
  (a) 一致すれば次に進む
  (b) 一致しなければ (2) に戻り繰り返す
(4) ゲスト OS にイベントを通知する
(5) 1 に戻って処理を継続する

また，非決定的な命令については，該当命令を実行時に実際に命令を実行して結果を得るのではなく，実行履歴から得た結果をゲスト OS に返すことで再現する．

表 1 実験環境
Table 1 Experimental environment

項目名	内容
実機	Dell Precision 490
CPU	Xeon 5130 2.00GHz
Memory	1GByte
入出力	シリアル× 2，ディスプレイ
ゲスト OS	Linux 2.6.23

4. 実験と考察

現在，ゲスト OS をロギング＆リプレイを利用して起動できている．安定動作にまでは至っていないため，ゲスト OS の起動を開始してから起動が完了する前までのロギング＆リプレイの負荷と実行履歴の量を測定した．測定は，Linux カーネルが命令アドレスの 0x100000 以降に展開された状態から開始し，nash が sh を起動する直前での間で行った．実験環境を**図 1** に示す．

4.1 ロギング＆リプレイの負荷

ロギング＆リプレイが Linux カーネルの起動処理に与える負荷を測定するために，経過時間を比較した．時間は，クロックサイクル数をカウントする Time Stamp Counter（TSC）の値を元に算出した．TSC はロギング＆リプレイの対象とはしていない．結果を**表 2** と**図 6** に示す．この結果では，Primary の実行履歴の取得と保存によるオーバヘッドは数ミリ秒と小さく，システムに与える影響は小さいと考えられる．Backup の実行履歴の読み込みと再現によるオーバヘッドは数十ミリ秒とやや大きいが，2 %以内に収まっている．

表 2 ゲスト OS の起動時間
Table 2 Results of time

	起動時間 [s]
ロギング&リプレイなしの Primary	2.284
ロギング&リプレイありの Primary	2.286
ロギング&リプレイありの Backup	2.319

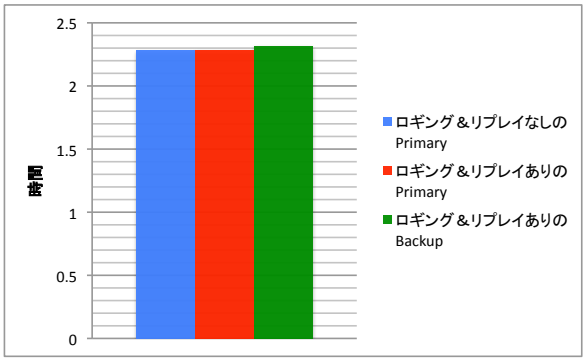

図 6 ゲスト OS の起動時間
Fig. 6 Results of time

4.2 実行履歴の総量と内訳

Primary が取得した実行履歴の総量と内訳について確認した（**表 3**，**表 4**）．値は複数回繰り返した結果の平均値である．タイマ割り込みの数が 207.5 となっているのは，そのためである．起動時間が約 2.3 s で，総量は約 81 KiB となっている．イベントの種類に関わらず 1 つの実行履歴を 32 byte の固定長で保存しているため未使用の領域もあり，削減することは可能である．また，起動中もシリアルを利用しているにも関わらず内訳を見るとシリアルの割り込みが非常に少ない結果となっているが，これはシリアルの初期化処理が起動シーケンスの終わり近くで行われているためだと考えられる．

起動中と同程度の量のイベントが起動完了後も発生するとすれば，Backup 側を 1 分遅れて実行するためには 2 MiB 以上の領域が必要となる．今後，実際に起動完了後のログの量を計測していく予定である．

表 3 実行履歴の総量
Table 3 Total of logs

総量 [KiB]
81.08

表 4 実行履歴の内訳
Table 4 Details of logs

	個数	割合 [%]
タイマ割込み	207.5	7.998
シリアル割込み	2	0.077
in 命令	2385	91.93

5. 関 連 研 究

本研究と同様のロギング&リプレイを実現した研究として，セキュリティの観点から行われた ReVirt[2]，デバッグの効率化の観点から行われた軽量仮想計算機モニタ[3] や King ら[8] の研究がある．本研究は，ロギング&リプレイの実現方法としてこれらの手法を参考にしているが，実現方法として SMP 環境を活用し，ゲスト OS の実行と再現を同時に行う．また，軽量仮想計算機モニタは，対象とするゲスト OS を独自 OS としており，アドレス変換機構は利用できないが，我々の研究ではアドレス変換機構を利用している Linux をゲスト OS として実行させている．

Bresoud ら[1] は，仮想マシンモニタにより Primary/Backup モデルに基づいたゲスト OS のレプリケーションを実現し，耐故障性が得られることが明らかにした．この研究では，命令列を epoch と呼ばれる間隔に区切り，epoch の終了時点で非同期的なイベントを適用する手法が示されている．本研究においても Primary/Backup モデルを採用したが，命令列を区切ること無く Primary でイベントや命令が発生したタイミングでイベントを適用する．

また，耐故障性の向上を目的として，ゲスト OS のスナップショットを転送することでゲスト OS を同一状態に保つ研究も行われている．Remus[9] ではチェックポイントを用いて実現されており，Kemari[10] では同期すべきイベントに着目して実現されている．本研究はこれらの研究とは目的が異なる．

6. お わ り に

本稿では，カーネルデバッグの支援方法として，ロギング&リプレイを SMP を活用し

Vol.2010-OS-113 No.12
2010/1/28

て実現する手法を提案した．これは，SMP の各プロセッサを Primary/Backup モデルの Primary と Backup に割り当てる方法であり，仮想マシンモニタにより実現する．Backup を Primary に対して時間差を設けて実行することで，Primary の過去の状態を Backup に作り出すことができる．実際に本システムの実装を進めており，機能の一部であるロギング＆リプレイを現在実装中である．現状で動作する範囲内で性能測定を行ったところ，ゲスト OS の起動時に関しては性能低下が小さいことを確認した．今後さらに研究を進め，ロギング＆リプレイ機能の実装を完了させ，本機能を利用した効率的なカーネルデバッグ環境の提供を目指す．

参 考 文 献

1) Bressoud, T. and Schneider, F.: Hypervisor-based fault tolerance, *ACM Transactions on Computer Systems (TOCS)*, Vol.14, No.1, pp.80–107 (1996).
2) Dunlap, G., King, S., S., C., Basrai, M. and Chen, P.: ReVirt: Enabling intrusion analysis through virtual-machine logging and replay, *ACM SIGOPS Operating Systems Review*, Vol.36, pp.211–224 (2002).
3) 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).
4) 青柳信吾：組込みシステムのための VMM による OS マイグレーションの実現，修士論文，筑波大学システム情報工学研究科コンピュータサイエンス専攻 (2009).
5) Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual*.
6) Neiger, G., Santoni, A., Leung, F., Rodgers, D. and Uhlig, R.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization, Technical report, Intel Corporation (2006).
7) 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, *ACM SIGOPS Operating Systems Review*, Vol.37, No.5, pp.164–177 (2003).
8) King, S., Dunlap, G. and P.M., C.: Debugging operating systems with time-traveling virtual machines, *In the Proceedings of the 2005 USENIX Annual Technical Conference* (2005).
9) Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. and Warfield, A.: Remus: High availability via asynchronous virtual machine replication, *In the Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008).
10) 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，コンピュータシステム・シンポジウム論文集， Vol.2009, No.13, pp.11–20 (2009).