単一システムイメージを提供するための仮想マシンモニタ

金田憲二[†,††] 大山恵弘[†] 米澤明憲[†]

我々は，クラスタ上に共有メモリ型マルチプロセッサマシンを仮想的に構築する仮想マシンモニタを設計・実装した．この仮想マシンの機能によって，クラスタを非常に簡便に利用することが可能となる．例えば，共有メモリ型マルチプロセッサマシン用の並列アプリケーションを，無変更のままクラスタ上で実行することが可能になる．さらに，マルチプロセッサをサポートする OS （例えば Linux）を，少量の変更で仮想マシンにインストールすることができる．我々は，8 台の物理マシン上に仮想的に 8-way のマルチプロセッサマシンを構築した．そして，その仮想マシン上で互いに独立な粗粒度タスクを並列に実行し，その実行時間を測定した．この実験の結果は，我々のアプローチで現実的な性能を達成できることを示している．

A Virtual Machine Monitor for Providing a Single System Image

Kenji Kaneda,[†,††] Yoshihiro Oyama[†] and Akinori Yonezawa[†]

We have designed and implemented a virtual machine monitor that virtualizes a shared-memory multi-processor machine on a commodity cluster. This functionality greatly simplifies utilization of commodity clusters. For example, it enables parallel applications for shared-memory multi-processor systems to run on clusters without any change of the applications. Moreover, commodity operating systems that support multi-processors (e.g., Linux) can be installed in a virtual machine with a small amount of modification. We built a virtual 8-way multi-processor machine on eight physical machines. We ran parallel coarse-grain tasks on Linux installed in the virtual machine and measured the execution time. The experimental result demonstrates the feasibility of our approach.

1. はじめに

コモデティクラスタは，ネットワーク技術の進化や PC の価格低下に伴い，近年その重要度を増しつつある．例えば，これまで並列計算などに携わることのなかった一般の自然科学の研究者が，個人・ワークグループ用として 4 ～ 32 ノード程度のクラスタを所有する，ということも現実に可能となっている．

しかし，こうしたクラスタの利用に際しては，その利便性が低いという重大な問題が依然として存在する．例えば，クラスタの各ノードに Linux などの通常の OS をインストールしただけでは，複数のマシンに分散した CPU やディスクなどの資源を大域的に管理する機構が欠如している．そのため，(特に計算機科学を専門としない一般の人にとって）クラスタを効率的に利用するのは難しいものとなっている．

そこで，本研究の目標として，こうしたクラスタ上に単一システムイメージ (Single System Image: SSI) を構築し，クラスタの利便性を向上させることを目指す．SSI の実現方法としては様々な既存研究（例えば SCore[1)]）が存在するが，本研究では特に，仮想マシンモニタ (Virtual Machine Monitor: VMM) を利用するというアプローチをとる．VMM というのは，実マシンと同等の処理（例えば OS の実行）が可能な仮想マシンを，実機上に構築するミドルウェアシステムである．CPU・メモリ・I/O デバイス等のハードウェアをソフトウェアでエミュレーションすることで，これを実現している．有名な VMM としては，例えば VMware Workstation[2)] が挙げられる．

我々は，より具体的には，「ネットワークで結合された複数のマシン上に，共有メモリ型マルチプロセッサマシンを仮想的に構築する VMM」を設計・実装する．この VMM を，我々は *Virtual Multiprocessor* と呼ぶ．Virtual Multiprocessor は，例えば N 台のシングルプロセッサマシン上に，N プロセッサからなる仮想

† 東京大学 大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
†† 日本学術振興会
Japan Society for the Promotion of Science

マルチプロセッサマシンを構築することができる．クラスタを利用する際には，この仮想マシン上に Linux といったマルチプロセッサ対応の OS をインストールし，さらに，その OS 上で並列アプリケーションを記述・実行する．

SSI を提供するための既存研究と比較して，我々のアプローチは以下の 3 点で優れている．まず，1 つ目の利点として，共有メモリ型マルチプロセッサマシンを仮定して記述された様々な並列アプリケーション（科学技術計算から Web サーバまで）を，無変更のまま分散環境上で実行できることが挙げられる．特に，パラメータスイープ型の並列アプリケーションなどを記述する際に，MPI[3] などの分散プログラミング言語を使用する代わりに，利用者が慣れ親しんだ既存の言語やツール（例えば並列 `make` やシェルスクリプト）をそのまま使用することができる．

さらに 2 つ目の利点として，並列アプリケーションだけでなく，単に逐次プログラムを複数同時に走らせる場合にも，我々のアプローチが有用であることが挙げられる．マルチプロセッサ用の OS カーネルも，少量の変更を加えるだけで仮想マシン上で動作させることができる．そのため，分散環境を管理するのに，例えば Linux のプロセス空間やファイルシステムをそのまま用いることができる．仮想マシン上で走っている Linux から複数のプロセスを立ち上げると，それらのプロセスは，Linux のスケジューラによって，それぞれ異なる仮想プロセッサに対して割り当てられる．そして，最終的には，その仮想プロセッサが対応づけられた実プロセッサ上で実行されることになる．

最後に 3 つ目の利点として，VMM による資源のカプセル化 (resource encapsulation) を，安全性や信頼性の向上のために利用できることが挙げられる．例えば，クラスタをサーバーホスティングのために用いる際に，マシンを破壊する恐れのあるソフトウェアを安全に実行するためのサンドボックスとして，VMM を利用することができる[4),5)]．また，仮想マシンの実行状態のスナップショットを取得・復元できるようにして，OS に何か不具合が生じたらそれ以前の状態に復旧する，といったことも可能になる[2),6),7)]．

我々は，8 台の物理マシン上に仮想的に 8-way のマルチプロセッサマシンを構築した．その上でフィボナッチ数を計算するプロセスを 8 個並列に走らせたところ，仮想・物理シングルプロセッサマシン上で実行させた場合と比較して，最大約 6.6 倍の性能向上となった．この実験の結果から，VMM の頻繁な介入を必要としない（例えば，I/O デバイスに頻繁にアクセスしない）プログラムの場合，良い性能を達成できることが分かった．

本稿は以下のように構成される．2 節で本システムの概要について述べる．3 節でハードウェアの仮想化手法について説明する．4 節で共有メモリの一貫性制御について詳細に説明する．5 節で我々が行った予備実験について述べる．6 節で関連研究との比較を行い，最後に 7 節でまとめと今後の課題について述べる．

2. Virtual Multiprocessor の概要

本節では，まず，Virtual Multiprocessor が提供する仮想マシンの特徴について説明する．次に，仮想マシンと実マシンの資源の対応付けについて述べる．

2.1 構築される仮想マシンの特徴

Virtual Multiprocessor によって構築される仮想マシンの特徴として，まず，命令セットアーキテクチャ (Instruction Set Architecture: ISA) のレベルで仮想化を行うことが挙げられる☆．仮想マシン上では，その ISA を満たす OS を走らせることができる．今現在の実装では IA-32 アーキテクチャを対象としている．

仮想マシンの他の特徴として，準仮想化 (para-virtualization) であることが挙げられる．つまり，仮想マシンの ISA は，実ハードウェアの ISA とほぼ同一であるが，完全に同じではない．詳しくは後述するが，実マシンと仮想マシンの ISA の相違点は，コントロールレジスタや `EFLAGS` レジスタなどの特殊なハードウェアへのアクセスに関連したものである．よって，汎用レジスタのみアクセスする多くのユーザアプリケーションは，ソースコードの変更を必要としないで済む．特殊なハードウェアにアクセスする OS のカーネルやデバイスドライバは，ソースコード変更を必要とする．ただし，その変更も，独自のアセンブリ変換器を用いることで半自動的に行え，手動変更による手間は少ない[11)]．詳しくは 3.2 節で述べる．

以上のように仮想マシンを設計した理由は，主に，少ない実装コストで，十分な性能を持つ仮想マシンを実現するためである．ちなみに，今現在の実装では Linux をサポートしている．

2.2 仮想マシンと実マシンの資源の対応

Virtual Multiprocessor は，仮想マシンと実マシンの資源を以下のように対応づける（**図 1** 参照）．

プロセッサ 基本的には，仮想マシンのプロセッサと実マシンのプロセッサは 1 対 1 に対応する．例え

☆ User-mode Linux[8)]，Zap[9)]，SoftwarePot[10)] などのような，Application Binary Interface (ABI) のレベルでの仮想化ではない．

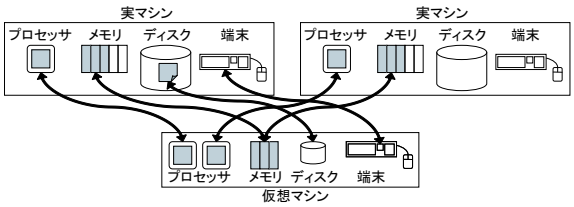

図 1 実マシンと仮想マシンの対応
Fig. 1 Mapping between physical machines and a virtual machine

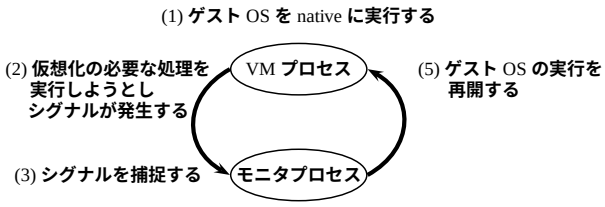

図 2 Virtual Multiprocessor の基本実行サイクル
Fig. 2 Basic execution cycle of Virtual Multiprocessor

ば，N 個のプロセッサからなる仮想マシンを構築するためには，複数の実マシンから総計 N 個のプロセッサを確保する必要がある．

メモリ 実マシンのメモリの一部を，仮想マシンのそれとして使用する．より具体的には，M MB の共有メモリを仮想化するためには，各仮想プロセッサごとに M MB のメモリを実マシンから確保する必要がある．

I/O デバイス どれか 1 つの実マシンにあるデバイスを，仮想マシンのそれとして使用する．例えばハードディスクの場合，実マシンのどれかに置かれたるファイルをディスクイメージとして利用する．シリアル端末の場合も同様に，どれかの実マシンの仮想端末を，仮想マシンのシリアル端末として利用する．

3. ハードウェアの仮想化

本節では，プロセッサ・メモリ・I/O デバイスといった各種ハードウェアの仮想化処理の実装について述べる．メモリの仮想化は，アドレス空間と一貫性制御のそれぞれの仮想化を必要とする．プロセッサおよびメモリのアドレス空間の仮想化手法は， LilyVM [11] とほぼ同様である．それ以外のメモリの一貫性制御と I/O デバイスの仮想化手法は，本システムに特有な実装となっている．

3.1 基本的な実装方針

Virtual Multiprocessor は，LilyVM[11] や FAUmachine[12] と同様に，実マシン上で走る OS の上に位置する．この方式は，VMM がハードウェア上に直に置かれる方式 [4),5),13)] と比較してオーバヘッドが大きい反面，少ない実装コストで IA-32 アーキテクチャの仮想化などを実現することができるといった利点をもつ[14]．

VMM は，大半の命令を実プロセッサ上で直に実行し，実ハードウェアの状態と干渉する命令（例えば，`cr3` などのシステムレジスタを読み書きする命令）のみ，ソフトウェアでエミュレーションする（このエミュレーションを必要とする命令を，センシティブ命令と呼ぶ）．以上の動作を実現するために，1 つの仮想プロセッサごとに，VM プロセスとモニタプロセスという 2 つのユーザプロセスを用意する．

VM プロセス 仮想マシンのプロセッサのうちどれか 1 つを担当し，ゲスト OS を native に実行する．VM プロセスが仮想化を必要とする処理を行おうとすると，シグナルが発生する．例えば，ユーザレベルでは実行することのできない特権命令を実行しようとすると，`SIGSEGV` シグナルが発生する．

モニタプロセス VM プロセスの実行を監視し，必要に応じてセンシティブ命令の実行の仮想化など，ハードウェアの仮想化処理を行う．例えば，VM プロセスがシグナルを発生させた場合，それを `ptrace` システムコールにより捕捉する．そして，捕捉したシグナルの種類と，その時点での VM プロセスのメモリやレジスタの状態を元に，必要とする仮想化処理を特定し，エミュレーションする．

この VM プロセスとモニタプロセスの実行サイクルを **図 2** に示す．ちなみに，ゲスト OS 上で新たにプロセスが生成された場合，そのプロセスは VM プロセスの内部で仮想的に実行される．新たにホスト OS 上でプロセスが生成されるわけではない．

3.2 プロセッサの仮想化

プロセッサの仮想化は，大きく分けて，センシティブ命令の仮想化と，割り込みや例外処理の仮想化からなる．まず，センシティブ命令の仮想化について述べる．センシティブ命令は，特権命令と非特権命令の 2 種類に分類される[15]．特権命令（例えば `lgdt` 命令）は，プロセッサの特権レベルが 0 より大きい場合，実行時に例外を発生させる．それに対して非特権命令（例えば `sgdt` 命令）は，低い特権レベルで実行しても例外は発生しない．

センシティブ命令の実行を捕捉する方法は，その命令が特権命令か非特権命令であるかによって異なる．

特権命令を捕捉するためには，実行時に発生した例外を捕捉すれば良い（VM プロセスはユーザプロセスであり，特権レベル 3 で動作していることに注意）．それに対して，非特権命令を捕捉するためには，より複雑な処理を必要とする．具体的には，ゲスト OS のカーネル・デバイスドライバを静的に書き換え，全てのセンシティブ命令の直前に不正な命令を挿入する．この不正な命令の挿入は，カーネルコンパイル時に，独自のアセンブリ変換器を用いて自動的に行われる．そして，モニタプロセスは，この不正な命令の実行によって発生した例外を捕捉することによって，非特権命令の実行直前に VM プロセスの実行を止める．

この静的なカーネルコードの変換は，小さな実装コストで非特権命令の仮想化が実現できるという利点を持つが，変換にソースコードを必要とするという欠点も持つ．例えば，バイナリしかない Linux のカーネルモジュールや，Windows などのソースコードの公開されていない OS は，現在の実装では扱うことができない．

次に，割り込みや例外処理の仮想化について述べる．割り込みや例外処理の仮想化は，基本的には，以下の手順で実装される．まず，モニタプロセスが割り込みや例外の発生を検知する．これは，`ptrace` システムコールによってシグナルやシステムコール呼び出しを捕捉したり，APIC のマップされたメモリ領域への書き込みを捕捉することによって実現する．次に，モニタプロセスは，その発生を検知した割り込み・例外を，適切な仮想プロセッサへと配送する．基本的には自分の担当する仮想プロセッサに対して割り込み・例外を配送するが，プロセッサ間割り込みが発生した場合は，TCP/IP で通信を行いながら指定された遠隔プロセッサに配送する．そして，割り込み・例外の配送された先の仮想プロセッサを担当するモニタプロセスは，割り込みディスクリプタテーブルなどを参照しながら，適切なハンドラへと VM プロセスの制御を移す．I/O デバイスの起こす外部割り込みについては，3.4 節で詳細に述べる．

3.3 共有メモリの仮想化

共有メモリの仮想化は大きく分けて，アドレス空間の仮想化と一貫性制御の実現の 2 つを必要とする．前述の通り，アドレス空間の仮想化に関しては，LilyVM とほぼ同様の手法である．一貫性制御に関しては，本システムに特有な処理である．

まず，アドレス空間の仮想化について簡単に説明する．アドレス空間の仮想化を実現するためには，セグメント機構とページング機構のそれぞれの仮想化を実現する必要がある．セグメント機構の仮想化は，Linux を動作させるのに必要な処理のみ実装されている．具体的には，セグメントレジスタへの読み書きの仮想化は実装されているが，アドレス変換は未実装で，各セグメントの先頭アドレスは常に 0 だと仮定している．

ページング機構の仮想化は以下のようにして実現している．まず，実マシン上に，仮想マシンのメモリイメージを保持するファイルを用意する．そして，仮想マシンのページディレクトリ・テーブルを参照することによって，VM プロセスの各ページをメモリイメージファイルへとマップする．より詳細には，VM プロセスがあるページにアクセスし `SIGSEGV` シグナルが発生した時点で，そのページを `mmap` システムコールによってイメージファイルへとマップする．また，ページディレクトリ・テーブルへの変更が有効になった時（例えば，`cr3` レジスタの値が更新された時）に，ページディレクトリ・テーブル中に既に存在しなくなったマッピングを `munmap` システムコールによって解放する．

また，仮想マシンがゲスト OS に提供する仮想アドレス空間の範囲を，`0x00000000` 以上 `0xb0000000` 未満に限定している．その理由は，`0xc0000000` 以上 `0xffffffff` 以下の領域はホスト OS のカーネルアドレス空間であり，`0xb0000000` 以上 `0xc0000000` 未満の領域はハードウェアの仮想化に必要な情報（例えばシステムレジスタの値）を格納するために使用されるからである．ゲスト OS のカーネルコードを静的に変更し，カーネルアドレス空間の先頭アドレスを `0xa0000000` にしている．

次に，一貫性制御の仮想化について説明する．既存のプログラムを変更することなく仮想マシン上で走らせることを可能にするために，基本的にはソフトウェア分散共有メモリシステムなどと同様の手法をとる．具体的には，VM プロセスのページの読み書き権限を `mprotect` システムコールによって適宜制限することによって，一貫性制御を実装する．例えば，あるページの情報が古くなった場合は，そのページへの読み書きを禁止する．すると，そのページに VM プロセスがアクセスした際に `SIGSEGV` シグナルが発生する．モニタプロセスはこのシグナルを捕捉すると，他のモニタプロセスと TCP/IP で通信をしながら，ページの最新の状態を取得する．そして，そのページへの読み書きを許可し，VM プロセスの実行を再開させる．この共有メモリの一貫性制御の詳細については 4 節で述べる．

以上の記述から分かるように，様々な理由によって

SIGSEGV シグナルが発生することに注意する必要がある．例えば，モニタプロセスは SIGSEGV シグナルを捕捉した際に，それがページフォルトの発生によるものなのか，それとも共有メモリの一貫性制御によるものなのかを判別する必要がある．具体的には，**図 3** に示されたフローチャートにしたがって，SIGSEGV シグナルの発生原因を特定し，それに応じた仮想化処理を行う．

3.4 I/O デバイスの仮想化

今現在の実装は，ハードディスクやシリアル端末などの I/O デバイスのエミュレーションをサポートしている．また，I/O デバイスへアクセスする手段としては，主に，プログラム I/O 方式（in/out 命令の発行），Direct Memory Access (DMA) 方式，メモリマップド I/O 方式が挙げられるが，現在の実装では，基本的には，プログラム I/O 方式と DMA 方式のみをサポートしている．

具体的には I/O デバイスのエミュレーションのために，全デバイスの状態をソフトウェア上で管理する I/O サーバが 1 つ用意されている．モニタプロセスは，必要に応じてこのサーバと通信を行う．例えば，ゲスト OS が in 命令によって p 番のポートから読み込みを行おうとする場合を考える．この場合，モニタプロセスは in 命令の実行を捕捉し，サーバに問い合わせをする．この問い合わせを受けたサーバは，p 番のポートへと対応付けれられたデバイスの動作をエミュレーションし，デバイスから読み出される値をモニタプロセスに返信する．

また，I/O サーバは各種デバイスの外部割込みを以下のようにエミュレーションする．まず，デバイスの状態を定期的に（少なくともタイマー割り込みの間隔ごとに）監視し，割込みを発生させる必要のあるデバイスが存在するか調べる．そして，もし割込みを発生させる必要がある場合は，基本的には，以下の手順で割り込みを仮想プロセッサに配送する．

(1) I/O サーバが，割り込み配送先の仮想プロセッサを決定する．現在の実装では，外部割り込みの配送先は基本的には固定されており，I/O サーバと同一の実マシン上にある仮想プロセッサに対して送られる☆．
(2) I/O サーバは，その仮想プロセッサを担当する VM プロセスに対して，適当なシグナル（例えば SIGUSR1）を送る．
(3) モニタプロセスが送信されたシグナルを捕捉する．
(4) モニタプロセスは，割り込みディスクリプタテーブルなどを参照しながら，適切なハンドラへと VM プロセスの制御を移す．

☆ 優先度の低いプロセッサに割り込みを配送する，I/O APIC による割り込みの動的な分配は，まだ実装されていない．

4. 共有メモリの一貫性制御

Virtual Multiprocessor は IA-32 アーキテクチャの仮想化を目指しており，共有メモリの一貫性制御も，その IA-32 のメモリモデルを満たしている必要がある．多くの既存のソフトウェア分散共有メモリシステムに関する研究は release consistency[16),17)] など他のメモリモデルを対象としていることから，この IA-32 のメモリモデル上での一貫性制御は我々のシステムに特有のものであるといえる．

本節では，まず，IA-32 のメモリモデルについて概説する．次に，我々が実装した simple なメモリ一貫性アルゴリズムについて述べる．

4.1 IA-32 のメモリモデルの概要

IA-32 のメモリモデルは，「あるプロセッサが行ったメモリアクセスが，自分および他のプロセッサにどういった順序で反映されるか」を定める，より具体的には，IA-32 のメモリモデルはプロセッサ順メモリモデル (Processor-ordered Memory Model) であり，以下の規則を満たす[18)]．

- あるプロセッサ x が行った読み書きについては，プログラムの文面通りの順で x に反映される．
- あるプロセッサ x が行った書き込みが x に反映される順序と，他のプロセッサに対して反映される順序は同一である．
- 異なるプロセッサが行った書き込みの間では，それらが反映される順序に制約はない．

また，IA-32 は，上記のメモリモデルを強めるための命令を提供している．例えば，mfence などの直列化命令 (serializing instruction) は，「直列化命令 i を実行した時点で，プログラムの文面上で i より前に位置する全メモリアクセス命令が，既に全プロセッサに反映済みである」ことを保障する．

4.2 Simple なアルゴリズム

我々は，simple な一貫性制御アルゴリズムを実装した．このアルゴリズムは，ページ単位でメモリの共有・非共有が管理される write invalidate プロトコル[19)] の一種である．同一ページに対して，読み込みは複数のプロセッサが同時に行うことができるが，書き込みは同時に 1 つのプロセッサしかできない．

図 4 と**図 5** は，上記のアルゴリズムのより詳細な記

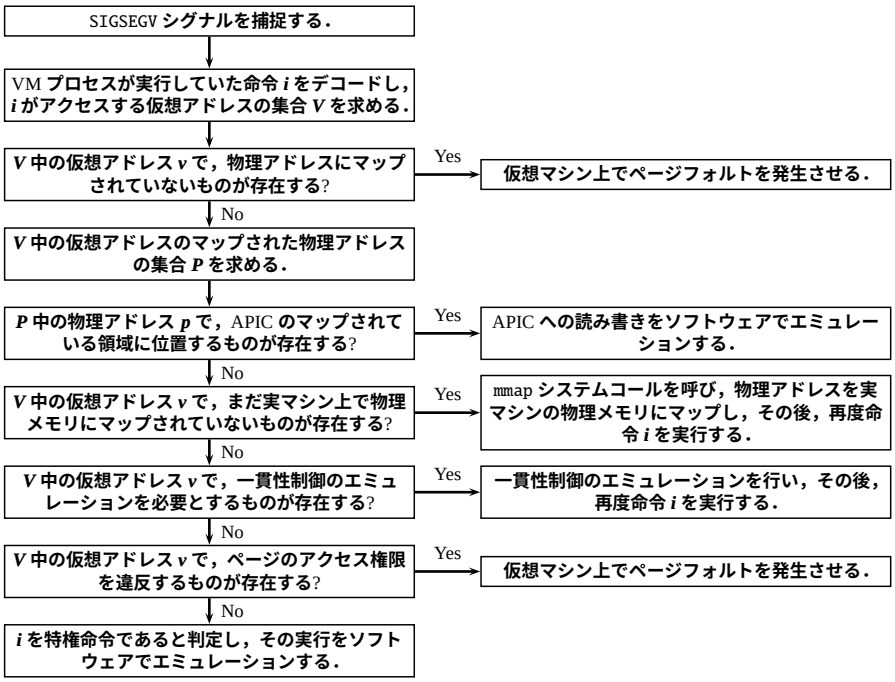

図 3 SIGSEGV シグナルの扱い
Fig. 3 Handling of SIGSEGV signals

$proc_i$	:	仮想プロセッサ i
$pages_i^n$	:	仮想プロセッサ i の n 番ページ
$p.state$	:	ページ p の状態 (`invalid`, `read_only`, or `read_write`)
$p.c$	:	ページ p の内容
$p.own$	:	ページ p のオーナー
$p.copyset$	:	ページ p の複製を持つプロセッサの集合
$p.busy$	:	ページ p を処理中かどうかを示すフラグ (`true` or `false`)

図 4 アルゴリズムの記述のための変数
Fig. 4 Variables for algorithm description

述である．図 4 には，アルゴリズム中に現れる変数が示されている．図 5 には，仮想プロセッサ i の動作が記述されている．各動作は $G \Longrightarrow A$ というシンタックスで記述され，G が満たされると A がアトミックに実行される．また，メッセージの送信先と送信元が同一プロセッサの場合，実際の実装ではメッセージは送信されず，ローカルに処理される．

5. 予 備 実 験

我々は，Virtual Multiprocessor のプロトタイプを実装し，予備実験を行った．このプロトタイプでは，最大 8 台の物理マシン上に 8-way の仮想マルチプロセッサマシンを構築し，その仮想マシン上で Linux を走らせることができる☆．また，我々は，仮想マシン上で `gcc` や `make` などの既存のプログラムが動作することも確認した．

予備実験として，システムの基本性能を評価するため，まず，共有メモリの一貫性制御以外の仮想化処理（例えば，センシティブ命令の実行や I/O デバイスへの読み書きのエミュレーション）にかかるオーバヘッドを測定するために，仮想シングルプロセッサマシン上でいくつかのベンチマークプログラムを実行した．そして次に，共有メモリの一貫性制御のオーバヘッドを測定するために，仮想マルチプロセッサマシン上で互いに独立なタスクを並列に実行した．

全ての実験において，Intel Xeon 2.4 GHz，2GB RAM，1 Gigabit Ethernet NIC からなるマシンを用いた．また，ホスト OS と ゲスト OS 共に Linux 2.4 を用いた．

5.1 仮想シングルプロセッサマシンの性能評価

共有メモリの一貫性制御以外の仮想化処理にかかるオーバヘッドを測定するために，物理および仮想シングルプロセッサマシン上で，いくつかの逐次プログラムを実行した．

表 1 は，実験に使用したベンチマークプログラムの

☆ 取得可能な BIOS の制限などのため，現在の我々の実装では 8 プロセッサまでしか扱えていない．

表 1 逐次ベンチマークプログラムの説明と，そのプログラムの物理・仮想シングルプロセッサマシン上での実行時間（単位: 秒）
Table 1 Description of Sequential benchmark programs and their execution time on a physical and a virtual single-processor machine (unit: seconds)

プログラム名	プログラムの説明	物理マシン上での実行時間 (P)	仮想マシン上での実行時間 (V)	オーバヘッド (V/P)
`fib`	フィボナッチ数を計算する	22.6	22.1	0.97
`getpid`	`getpid` を 100,000 回発行する	0.05	18.1	354
`ls`	数百のファイルの情報を表示する	0.03	6.64	255
`gcc`	C プログラムをコンパイルする	0.14	0.98	6.81

```
access nth page with a when violation(pages_i^n, a)
⟹ stop the execution of the VM process;
    send ⟨fetch, n, a, i⟩ to manager(n);

receive⟨fetch, n, a, s⟩ when pages_i^n.busy = false
⟹ let p be pages_i^n;
    p.busy := true;
    match a with
      read  ⇒ send ⟨inv, n, a, s, p.own⟩ to p.own;
      write ⇒ forall x ∈ S(p, s) do
                send ⟨inv, n, a, s, p.own⟩ to x;

receive ⟨inv, n, a, s, o⟩
⟹ let p be pages_i^n;
    match a with
      read  ⇒ p.state := read_only;
      write ⇒ p.state := invlid;
    if o = proc_i then send ⟨ack, n, a, p.c⟩ to proc_s;

receive ⟨ack, n, a, c⟩
⟹ let p be pages_i^n;
    p.c := c;
    match a with
      read  ⇒ p.state := read_only;
      write ⇒ p.state := read_write;
    send ⟨finish, n, a, i⟩ to manager(n);
    restart the execution of the VM process;

receive⟨finish, n, a, s⟩
⟹ let p be pages_i^n;
    match a with
      read  ⇒ p.copyset := p.copyset ∪ { proc_s };
      write ⇒ p.copyset := { proc_s };
               p.own := proc_s;
    p.busy := false;

ただし
manager(n) : n 番ページのマネージャー
             (例えば，プロセッサ数を N とするとき，
             manager(n) = proc_{n mod N})
violation(p, a) ≡ p.state = invalid ∨
                  (p.state = read_only ∧ a = write)
S(p, s) = {x ∈ p.copyset : x ≠ proc_s ∨ x = p.own}
```

図 5 Simple なメモリ一貫性アルゴリズム（仮想プロセッサ i における）
Fig. 5 Simple memory consistency algorithm (for virtual processor i)

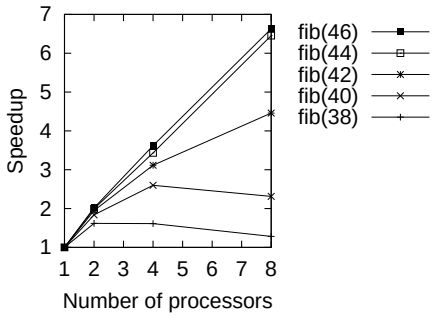

図 6 並列フィボナッチの性能向上
Fig. 6 Speedup of parallel fibonacci

説明と，物理・仮想シングルプロセッサマシン上におけるそのプログラムの実行時間を示している．この実験結果から分かるように，`getpid` や `gcc` の実行において，システムコール呼び出しや I/O デバイスの仮想化にかかるオーバヘッドが非常に大きいことがわかる．我々は，既存の VMM（例えば Xen[13] や CoVirt[20]）によって開発された最適化手法を適応することで，このオーバヘッドは削減可能であると考える．

5.2 仮想マルチプロセッサマシンの性能評価

1 ～ 8 台の物理マシン上にそれぞれ 1-way ～ 8-way の仮想マルチプロセッサマシンを構築し，その上でフィボナッチ数を計算するプロセスを 8 個並列に走らせた．このプログラムを実行した結果，**図 6** に示される性能向上を得た．`fib(n)` は，`n` 番目のフィボナッチ数を計算する際の性能向上を表している．この図から、実行されるタスクの粒度が大きければ台数効果がでていることが分かる．例えば，`fib(46)` を動作させたときは，8 プロセッサで約 6.6 倍の性能向上となった．

オーバヘッドの要因を調べるため，`fib(44)` を実行した際の実行時間の内訳を計測した．**表 2** はその結果を示しており，それぞれの値は，各プロセッサごとにかかった時間の平均値を示している．Total は，8 つのプロセス全てが実行を終えるまでにかかった全実行時間を表す．Native は，ゲスト OS が実機上で走っていた時間である．フィボナッチ数を計算するのにか

かった時間を主に表す．Mem は，モニタプロセスが行う仮想化処理のうち，共有メモリの一貫性制御の仮想化にかかった時間である．遠隔マシンとの通信にかかった時間を主に表す．Misc は，モニタプロセスが行う仮想化処理のうち，共有メモリの一貫性制御以外の処理（例えば，センシティブ命令の実行や I/O デバイスへの読み書きのエミュレーション）にかかった時間を示す．Idle は，実行可能なプロセスが存在せず仮想プロセッサが `hlt` 命令を実行していた時間を表す．表 2 から，プロセッサ数が増加するにつれて，共有メモリの一貫性制御の仮想化のオーバヘッドが増大していることが分かる．

そこで，共有メモリの一貫性制御のオーバヘッドについて，より詳しい分析を行った．具体的には，`fib(44)` 実行時における，ページのフェッチが発生した仮想アドレスの分布（**図 7** 参照）と，1 回のページのフェッチにかかる時間の分布（**図 8** 参照）とを測定した．図 7 から，プログラムの実行直後と終了直前に，ユーザ空間とカーネル空間の両方で，ページのフェッチが大量に発生していることが分かる（ゲスト OS のカーネル空間が `0xa0000000` から始まっていることに注意）．また，図 8 から，大半のページのフェッチの処理は数ミリ秒で完了するが，一部のフェッチは処理するのに数十ミリ秒必要としていることが分かる．以上のことから，false sharing などのために同一ページへのフェッチが集中した際に，オーバヘッドが大きくなっていることが考えられる．

6. 関 連 研 究

6.1 仮想マシンモニタ

vNUMA[21] は，我々の VMM と同様，分散環境上に共有メモリ型マルチプロセッサマシンを仮想的に構築するシステムである．この vNUMA と Virtual Multiprocessor の相違点として，まず，vNUMA が Itanium アーキテクチャを対象としているのに対して，

表 2 `fib(44)` の実行時間の内訳（単位: 秒）
Table 2 Breakdown of execution time of `fib(44)` (unit: seconds)

プロセッサ数	Total	Native	Mem	Misc	Idle
1	180.0	177.8	0.0	2.2	0.0
2	90.3	87.9	1.0	1.1	0.3
4	52.4	43.7	3.0	0.4	5.3
8	27.9	22.1	3.7	0.1	2.0

Total: 全実行時間
Native: ゲスト OS が native に実行されていた時間
Mem: 一貫性制御のエミュレーションにかかる時間
Misc: センシティブ命令等のエミュレーションにかかる時間
Idle: 仮想マシンが `hlt` 命令を実行していた時間

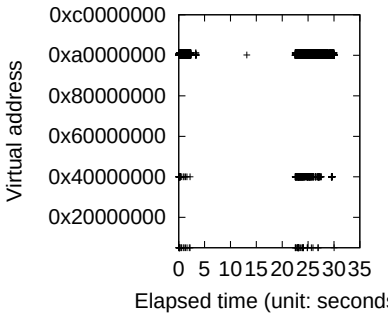

図 7 メモリ共有のためにモニタプロセスによってフェッチされた仮想アドレスの分布（`fib(44)` における）
Fig. 7 Distribution of virtual addresses fetched by the monitor processes for memory sharing (for `fib(44)`)

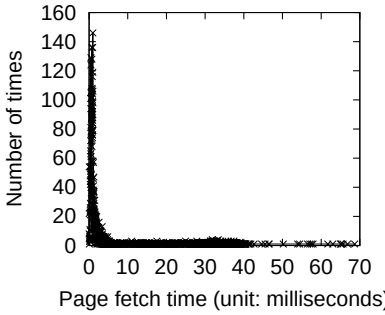

図 8 個々のページフェッチが完了するのにかかった時間の分布（`fib(44)` における）
Fig. 8 Distribution of times which individual page fetch requests took to complete (for `fib(44)`)

Virtual Multiprocessor は IA-32 アーキテクチャを対象としていることが挙げられる．

また，別の相違点として，共有メモリの一貫性制御アルゴリズムが挙げられる．vNUMA が元にしている Ivy[19] では，ページの invalidation が同期的に行われる（メッセージ送信後，アックを受信するまで待機する）のに対して，Virtual Multiprocessor では非同期に行われる．それによって，一回のページのフェッチにおけるメッセージの送信回数と，クリティカルパス長（ゲスト OS が実行を再開するまでに最低限送信の必要なメッセージ数）は，各システムで**表 3** のようになる．ただし，フェッチのリクエスト元のプロセッサを除く，ページの複製をもつプロセッサの数を c とする．この表から，read 時のメッセージ送信回数以外は，Virtual Multiprocessor と vNUMA は同性能，または，Virtual Multiprocessor の方が優れていることが分かる．ただし，定数倍の差であるので，実アプリケーションで有意な差が生じるかどうかを，今後実験を通して検証する．

Virtual Iron[22] も，分散環境上に共有メモリ型マ

ルチプロセッサマシンを仮想的に構築するシステムだが，詳細が未公開であるため十分な比較を行えていない（2005 年 10 月現在）．

6.2 クラスタ用ミドルウェアシステムと OS

SSI を実現するクラスタ用ミドルウェアシステムについて，数多くの研究がなされてきた．例として，SCore[1] や Condor[23] などが挙げられる．これらのミドルウェアシステムを用いると，遠隔マシンへの並列ジョブ投入などを行うことが可能になる．

しかし，これら既存のクラスタ用ミドルウェアシステムが提供する SSI は，Virtual Multiprocessor の提供する SSI と比較して，機能が制限されたものとなっている．例えば，共有メモリを仮定して記述された既存の並列プログラムをそのまま分散環境上で実行することはできない．また，CPU などの資源を大域的に管理する機構が存在するが，それぞれのシステムの持つ特有の操作・処理に習熟する必要がある．

また，Linux カーネルに一部変更を加え，分散環境上で動作させるという研究（例えば MOSIX[24] や Kerrighed[25]）も存在する．これらの研究で提案されたシステムには，カーネルの改変に多大な手間を必要とするという問題がある．

6.3 ソフトウェア分散共有メモリシステム

共有メモリを仮定して記述された既存の並列プログラムを分散環境上で実行可能にするソフトウェア分散共有メモリシステムの例として，Shasta[26] が挙げられる．Shasta は，コンパイラによって実行ファイルを変換し，ロード・ストアの直前にチェックコードを挿入することによって，共有メモリを実現する．

Shasta がユーザアプリケーションしか対象としていないのに対して，我々のシステムは，ユーザアプリケーションだけでなく OS カーネルも分散環境上で実行可能にする．また，対象としているメモリモデルも異なり，Shasta は MIPS アーキテクチャを対象としているが，Virtual Multiprocessor は IA-32 アーキテクチャを対象としている．

表 3 Ivy と Virtual Multiprocessor のメモリ一貫性アルゴリズムの比較
Table 3 Comparison of memory consistency algorithm between Ivy and Virtual Multiprocessor

	アクセス	マネージャ	メッセージ数	クリティカルパス長
Ivy	read	リクエスト元	2	2
		オーナ	2	2
		それ以外	3	3
	write	リクエスト元	$2+2c$	4
		オーナ	$2+2c$	4
		それ以外	$3+2c$	5
Virtual Multi-processor	read	リクエスト元	2	2
		オーナ	3	2
		それ以外	4	3
	write	リクエスト元	$2+c$	2
		オーナ	$3+c$	3
		それ以外	$4+c$	4

c: フェッチのリクエスト元のプロセッサを除く，ページの複製をもつプロセッサの数

7. おわりに

本稿では，ネットワークで結合された複数のマシン上に共有メモリ型マルチプロセッサマシンを仮想的に構築するシステム Virtual Multiprocessor について述べた．本システムによって，クラスタなどの分散環境を簡便かつ効率的に利用することが可能になる．

今後の課題としては，以下が挙げられる．

共有メモリの一貫性制御アルゴリズムの最適化 例えば，IA-32 のメモリモデルの持つ「同期命令を実行するまで，自プロセッサの行った書き込みが遠隔プロセッサに反映されるとは限らない」という性質を利用して，一つのページに対して複数のプロセッサが同時に書き込むことを可能にする．

仮想化処理のオーバヘッドの削減 既存の VMM（例えば Xen[13] や CoVirt[20]）によって開発された技術を我々の VMM に適応することで，仮想化処理にかかるオーバヘッドの削減を図る．より仮想化に適した設計のアーキテクチャ [27),28)] の利用も，オーバヘッドの削減手段として考えられる．

耐故障性 分散チェックポイント技術[29] や仮想マシンの複製技術[30] を利用し，物理マシンが故障した中でも VMM が動作し続けることを可能にする．

また，SPLASH-2 や Apache といった，より現実的なアプリケーションを仮想マシン上で走らせ性能評価を行う．よりプロセッサ数を増やした環境での実験も行う．本システムのプロトタイプは，`http://www.yl.is.s.u-tokyo.ac.jp/~kaneda/vmp` から取得できる．

謝辞 本研究の一部は，科学技術振興機構 戦略的創造事業の支援を受けた．

参考文献

1) SCore Cluster System Software: `http://www.pccluster.org/`.
2) VMware Inc.: `http://www.vmware.com/`.
3) The Message Passing Interface (MPI) standard: `http://www-unix.mcs.anl.gov/mpi/`.
4) Whitaker, A., Shaw, M. and Gribble, S. D.: Scale and Performance in the Denali Isolation Kernel, *In Proc. of OSDI*, pp. 195–209 (2002).

5) Waldspurger, C. A.: Memory Resource Management in VMware ESX Server, *In Proc. of OSDI*, pp. 181–194 (2002).
6) Whitaker, A., Cox, R. S. and Gribble, S. D.: Configuration Debugging as Search: Finding the Needle in the Haystack, *In Proc. of OSDI*, pp. 77–90 (2004).
7) Sato, O., Potter, R., Yamamoto, M. and Hagiya, M.: UML Scrapbook and Realization of Snapshot Programming Environment, *In Proc. of ISSS*, pp. 281–295 (2003).
8) User-mode Linux: `http://user-mode-linux.sourceforge.net/`.
9) Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments, *In Proc. of OSDI*, pp. 361–376 (2002).
10) Kato, K. and Oyama, Y.: SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation, *Software Security – Theories and Systems, volume 2609 of Lecture Notes in Computer Science*, pp. 112–132 (2003).
11) Eiraku, H. and Shinjo, Y.: Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions, *In Proc. of BSDCon*, pp. 91–102 (2003).
12) Hōxer, H.-J., Buchacker, K. and Sieh, V.: Implementing a User-Mode Linux with Minimal Changes from Original Kernel, *In Proc. of Linux-Kongress 2002*, pp. 72–82 (2002).
13) 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, *In Proc. of SOSP*, pp. 164–177 (2003).
14) Sugerman, J., Venkitachalam, G. and Lim, B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *In Proc. of the USENIX Annual Technical Conference*, pp. 1–14 (2001).
15) Robin, J. S. and Irvine, C. E.: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, *In Proc. of the USENIX Security Symposium*, pp. 129–144 (2000).
16) Keleher, P., Dwarkadas, S., Cox, A. and Zwaenepoel, W.: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *In Proc. of the USENIX Winter Technical Conference*, USENIX, pp. 115–131 (1994).
17) Bennett, J., Carter, J. and Zwaenepoel, W.: Munin: Distributed shared memory based on type-specific memory coherence, *In Proc. of PPoPP*, pp. 168–176 (1990).
18) Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide* (2003).
19) Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. on Computer Systems (TOCS)*, Vol. 7, No. 4, pp. 321–359 (1989).
20) King, S. T., Dunlap, G. W. and Chen, P. M.: Operating System Support for Virtual Machines, *In Proc. of the USENIX Annual Technical Conference*, pp. 71–84 (2003).
21) Chapman, M. and Heiser, G.: Implementing Transparent Shared Memory on Clusters Using Virtual Machines, *In Proc. of the USENIX Annual Technical Conference*, pp. 383–386 (2005).
22) Virtual Iron Software: `http://www.virtualiron.com/`.
23) Litzkow, M., Livny, M. and Mutka, M.: Condor - A Hunter of Idle Workstations, *In Proc. of ICDCS '88*, pp. 104–111 (1988).
24) Barak, A. and La'adan, O.: The MOSIX Multicomputer Operating System for High Performance Cluster Computing, *Journal of FGCS*, Vol. 13, No. 4-5, pp. 361–372 (1998).
25) Morin, C., Lottiaux, R., Vallée, G., Gallard, P., Utard, G., Badrinath, R. and Rilling, L.: Kerrighed: a Single System Image Cluster Operating System for High Performance Computing, *In Proc. of Euro-Par*, pp. 1291–1294 (2003).
26) Scales, D. J., Gharachorloo, K. and Thekkath, C. A.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *In Proc. of ASPLOS-VII*, ACM, pp. 174–184 (1996).
27) Intel Corporation: *Intel Virtualization Technology Specification for the IA-32 Intel Architecture* (2005).
28) Advanced Micro Devices: *AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual* (2005).
29) Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M. and Johnson, D. B.: A Survey of Rollback-recovery Protocols in Message-passing Systems, *ACM Computing Surveys (CSUR)*, Vol. 34, No. 3, pp. 375–408 (2002).
30) Bressoud, T. C. and Schneider, F. B.: Hypervisor-based fault tolerance, *ACM Trans. on Computer Systems (TOCS)*, Vol. 14, No. 1, pp. 80–107 (1996).