社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－OS－109 (4)
2008／8／7

マルチコアSHにおける複数カーネル実行機構の設計と実装

下沢 拓[†1] 藤田 肇[†2] 石川 裕[†1,†2]

SMPマシン上において複数のOSを動作させる機構SHIMOSのSHにおける実装を示す．これは，既存の仮想マシンを用いる手法と異なり，資源を分割占有させることにより，オーバヘッドなくOSを実行することができる．本論文では，この機構の設計及びSHアーキテクチャ上での実装を述べ，SHIMOSの移植可能性を示し，特権命令の実行時間やI/O処理と計算処理の同時実行などのベンチマークによってその評価を示す．

Design and Implementation of a Multiple Kernel Execution Mechanism in Multicore SH Processors

TAKU SHIMOSAWA,[†1] HAJIME FUJITA[†2] and YUTAKA ISHIKAWA[†1,†2]

This paper shows an implementation on the SH architecture of the SHIMOS mechanism, which we have proposed for running multiple operating systems on an SMP machine. This mechanism enables executing operating systems without any overhead, by partitioning and dedicating resources for the operating systems. In this paper, we describe the design and the implementation on SH of the mechanism to present the portability of SHIMOS, and evaluate it by running several benchmarks to measure the time for privileged instructions and simultaneous execution of I/O processing and calculation.

1. まえがき

近年，マルチコアプロセッサの普及は著しく，高性能計算から組み込み用途まで幅広く使われ始めている．計算資源の並列性が高まっているが，その一方で個々のプログラムが高い並列性を有しているとは言い難く，複数の異なるプログラムを同時に実行することにより有効的に活用できる場面も多い．

その一つの形として，複数のOSを一つのマシン上で動作させることは，複数のアプリケーションが異なるOSを要求する場面で有効である．また，一般的なOSと実時間OSといった，異種のOSを同時に実行することでそのシステム全体の機能を増やすこともできる．

複数OSの単一マシンでの並列実行を実現する機構として，論理分割(LPAR)がある．これはIBM/System 370[1]で導入された機能であり，ハードウェアによりメモリなどの資源を分割してOSを複数起動させるものである．

OSの複数実行方法として，最もよく利用されるのが仮想マシン[2]である．仮想マシンは，LPARのようにハードウェア機構を必要とせず，ソフトウェアのみで処理を行うことができる．ただし，そのために特権命令の実行やデバイスの仮想化にオーバーヘッドが存在する．

我々は，SHIMOS (Single Hardware with Independent Multiple Operating Systems) と呼ばれる機構を提案し，x86アーキテクチャ上での実装を行い評価を示した[3]．SHIMOSは，CPUやメモリといった資源を分割し，同時に動作する複数のカーネルにそれぞれ専有させる機構である．この資源の専有により，SHIMOSでは，各CPUは特権命令を直接実行でき，デバイス操作に対してもオーバーヘッドなく処理することができる．SHIMOSによる分割の概要を**図1**に示す．

本論文では，SHIMOSのx86アーキテクチャでの実装を元に，SHアーキテクチャ上でのLinuxカーネルへの実装を示し，Apache Benchmark等による評価を行い，その分割の効果を示す．

本論文の構成は以下のとおりである．第2節において，仮想マシンの実装とその問題点について述べる．第3節では，実現への課題を挙げ，第4節において，それに対するx86及びSHアーキテクチャでの実装の対処についてそれぞれ述べる．第5節において，本手法の実験評価とその考察を行い，第6節で関連研究を挙げ，第7節で結論を述べる．

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 東京大学情報基盤センター
Information Technology Center, The University of Tokyo

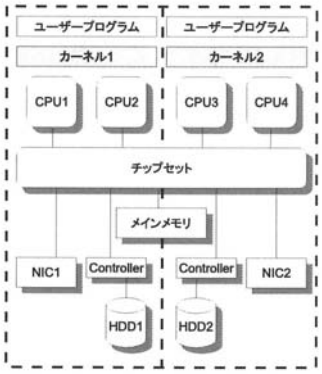

図 1　SHIMOS による資源分割の概要

2. 背　　景

2.1 仮想マシン

Popek[4] が提唱した仮想化可能性は，古典的な "trap and emulate" 方式による可能性を定義するものである．x86 アーキテクチャにおいては，一部の特権命令 (popf 命令など) は，ユーザーモードで実行されても trap することができないため，この方法による仮想化はできない．そのため，仮想マシンを実現するために，いくつかの方法が用いられている．

一つは，VMWare[5] などの仮想マシンで使われている動的コード変換である．ゲスト OS のコードのうち，特権命令やページテーブルや I/O 操作のコードを動的に変換し，VMM(仮想マシンモニタ) がその処理を代わりに行うものである．このため，この変換にコストが生じる．

もう一つは，Denali[6]，Xen[7] などが用いる準仮想化 (para-virtualization) である．VMM として Hypervisor と呼ばれるマイクロカーネルを置く．ゲスト OS には以下のような変更が加えられる．低い特権レベルでもカーネルが動作するようにし，実際にページテーブルや I/O 処理を行う場合には Hypervisor が提供するインタフェースを用いる．

そのほかに，ハードウェアにより "trap and emulate" 方式を可能にする仮想化支援機構が，最近の CPU[8),9)] に導入されている．これを用いることにより，古典的な方式による仮想マシンの実現が可能になる．

いずれの方法も，特権命令や I/O 処理にオーバーヘッドが存在する．SHIMOS では仮想化を用いず，直接カーネルを実行する．

2.2 x86 SMP ハードウェアにおける処理

本節では，x86 アーキテクチャでの SMP 環境におけるブートプロセスなど，SHIMOS の実装に関係する CPU アーキテクチャについて記述する．

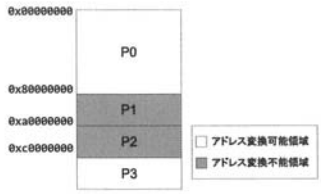

図 2　SH アーキテクチャの仮想メモリマップ

2.2.1 割り込み処理

x86 SMP ハードウェアにおいては，割り込みは二つの種類の APIC (Advanced Programmable Interrupt Controller) によって制御される．一つは，Local APIC と呼ばれるもので，もう一つは IOAPIC と呼ばれるものである．Local APIC は各 CPU コアごとに存在し，割り込みバスによって自分宛ての割り込みが伝達された場合に，CPU の INT 線を上げる．また，プロセッサ間割り込みを行う場合には，割り込みバスを通じて他の Local APIC にそれを伝達する．IOAPIC は外部割り込みと割り込みバスの間で，外部割り込みを I/O Redirection Table に従って，指定された LAPIC に対して割り込みを伝達する．このうち，logical destination mode を用いることで，一つ以上の CPU に割り込みを伝達することが可能である．

2.2.2 ブートプロセス

SMP 環境では，まず電源が入るとハードウェアによって一つの CPU コアが BIOS 起動のために選ばれ，実行される．この CPU は BSP(Bootstrap Processor) と呼ばれる．BSP はその後，OS のブートを開始し，この中で AP(Application Processor) と呼ばれるその他の CPU を起動する．AP のブートは，特殊な IPI を用いて AP の INIT シグナルを発生させ，リセットすることによって行われる．AP は特殊なリセットベクタの値にしたがって，その番地から実行を開始するが，リアルモードで実行を開始するため，AP のプロテクトモードへの移行も含めた初期化コードは，1MB 以下の下位メモリに置かなければならないという制限がある．

2.3 マルチコア SH における処理

本節では，SH アーキテクチャでの実装を行った RP1 プロセッサ[10] の，SHIMOS の実装上重要なアーキテクチャの特徴について述べる．

2.3.1 メ　モ　リ

29 ビットの物理アドレスモードでは，SH アーキテクチャでの仮想メモリマップは，**図 2** に示すように，P0 から P3 と呼ばれる 4 つの領域に分かれている[11]．

このうち，図で灰色で描かれた P1，P2 領域は MMU が有効な状態でも，それぞれ物理アドレスの 0 番地から線形に固定的にマップされる領域である．

CPU の非特権モード下ではこれらのうち P0 領域のみアクセスが有効であるので，SH アーキテクチャで

の Linux は 0x80000000 未満のアドレスをユーザー領域アドレスとし，それ以上をカーネルが使用するアドレスとしている．

2.3.2 他コアの起動処理

x86 アーキテクチャと同様に，システムの起動時には一つのコアのみが起動し，そのコアが他のコアの特定のレジスタに書き込むことで，他のコアの起動を行うことができ，またその番地も指定できる．

Linux においては，`arch/sh/head.S` の非 BSP 用初期化コードから実行を開始するように設定される．

3. 設　　計

本節では，SHIMOS の設計とその実装上の課題を明らかにする．

3.1 基本設計

SHIMOS では，CPU，メモリや，ハードディスクドライブ，ネットワークカードなどの周辺機器は分割され，各カーネルに割り当てられる．

この分割，割り当て処理をソフトウェアで行うために，動作させるカーネルに変更を加えたものになる．認識する CPU やハードウェアを制限する変更を加えたカーネルを動作させることにより，分割と専有が実現できる．

また，分割したカーネル間通信及びネットワークデバイスの共有を実現するために，共有メモリとプロセッサ間割り込みを利用した仮想ネットワークデバイスを作成する．

3.2 カーネルローダーモジュール

第 2.2.2 節で述べたようなアーキテクチャ上の制約及び，初期化の処理を逐次化するために，各カーネルの起動は順次行われる．すなわち，一つのカーネルが完全に起動した後に，そのカーネルからその他のカーネルの起動が行われる．この起動は，専用のカーネルモジュール（カーネルローダーモジュール）を用いることによって行われる．

カーネルローダーモジュールは，一般のブートローダが行うのと同様に，メモリ上に新しいカーネルを配置し，新しい CPU の動作を開始させ，その CPU は新しいカーネルを実行する．

3.3 課　　題

前々節及び前節で述べた設計によって，SHIMOS を Linux カーネルに変更を加えることで実装する上で，解決しなければならない課題をここで挙げる．

3.3.1 CPU

標準の Linux カーネルは，マシン上に存在する全ての CPU を利用しようとする．`maxcpus` コマンドラインオプションによって，CPU の数を制限することはできるが，CPU の認識順に使用していくため，そのままでは，2 番目以降のカーネルは最初のカーネルに属する CPU を利用しようとしてしまう．

3.3.2 メ モ リ

メモリも同様に，標準の Linux カーネルでは，存在する領域全体を利用しようとする．`mem` コマンドラインオプションにより，上限を設けることは可能であるが，低位のメモリ使用を制限することはできない．また，アーキテクチャによっては CPU の起動時などに特殊なメモリ領域を必要とするものが存在する．

その他の問題として，SHIMOS では，各カーネルは異なるメモリ領域を使い，従って仮想メモリマッピングもカーネル間で独立したものである．その一方で，カーネル間通信を実現するには，データ授受のためのメモリを共有する必要がある．

3.3.3 デ バ イ ス

Linux において，デバイスの選択を明示的に行うオプションは用意されていない．すなわち，カーネルは利用できる全てのデバイスを使用する．これは，カーネルに割り当てられていないデバイスも使用してしまう可能性がある．

3.3.4 外部割り込み

デバイスは各カーネル間で分割されるが，割り込みの一部と，割り込みを制御するコントローラをカーネル間で共有する必要がある．

これは，x86 においては IO APIC(Advanced Programmable Interrupt Controller) と呼ばれるもので，SH においては INTC(Interrupt Controller) と呼ばれるものに相当する．

これらは，通常の Linux ではカーネルが完全に初期化し，全ての割り込みを，利用するものについてはそのカーネルに属する CPU に対して発生するようにし，利用しないものについてはマスクを行ってしまう．

3.3.5 プロセッサ間割り込み

TLB のクリアなど，SMP においては，プロセッサ間割り込み (IPI) が重要な役割を持っている．

しかし，カーネル内で使われる IPI は，そのカーネル内で完結する必要があり，別のカーネルに対して割り込みが行われてはならない．

3.3.6 ブート処理

2 番目以降のカーネルは，前述したカーネルローダーモジュールによって起動されるが，ブート処理はアーキテクチャ依存の処理が多く，特殊な処理を要求されることがある．

また，正常に既にカーネルが起動している状態から，新たなカーネルの起動処理を行わせる必要があり，影響を与えずにブートさせる必要がある．

4. 実　　装

我々は，SHIMOS を x86 での Linux 2.6.24，SH で Linux 2.6.16 ベースの RP1 向け Linux に対して実装を行った．本節では，この実装について，それぞれのアーキテクチャごとに述べる．

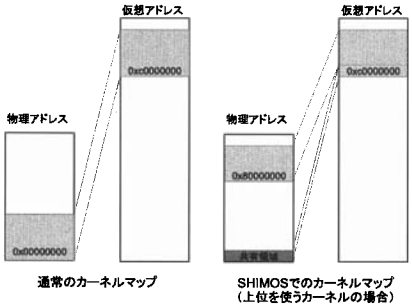

図 3　通常と SHIMOS でのカーネルのメモリマップ

4.1 x86 における実装

4.1.1 CPU

Linux の二番目以降の CPU の初期化処理を変更し，そのカーネルをブートした CPU の Local APIC ID より大きい CPU のみを初期化するように変更した．このため，カーネルローダーモジュールは，ロードするカーネルが使用する CPU のうち，もっとも Local APIC ID が若い CPU を起動 CPU とする．

4.1.2 メ モ リ

各カーネルのメモリマップは，**図 3** に示したように，各カーネル間で共通に使う低位の領域と独立に使う領域の二つの部分からなるように変更した．

低位領域には，二つの用途がある．一つは，第 2.2.2 節で述べた CPU の初期化時及びレガシーデバイスのアクセスに使うためであり，もう一つは，カーネル間での通信を実現するためのものである．

独立に使う領域では，オリジナルの Linux カーネルでは，使用するメモリの最低位アドレスを 0 と仮定しているのを変更する必要があった．

これらを実現するために，以下の変更を行うことで，メモリマップの変更を行った．

- *bootmem allocator* や *zone allocator* といったアロケータの初期化のコードの変更
- 仮想アドレスと物理アドレスの変換を行う `__va`, `__pa` マクロの変更
- 物理ページオフセットを表す `PAGE_PFN_OFFSET` の変更
- アドレス変換テーブルの初期化処理

4.1.3 デ バ イ ス

デバイスのうち，シリアルポート・キーボードといったレガシーなデバイス以外の割り当ては，各カーネルで使うドライバ（カーネルモジュール）の有無によって容易に制御できる．また，同じドライバを使うデバイスを複数持っている場合のために，PCI デバイスに関しては，PCI ID（バス番号，デバイス番号）によって認識するかどうかを分けるコードを追加した．

レガシーなデバイスについては，デバイスドライバのコードを変更し，例えばシリアルポートであれば一部のポートのみ認識するようにした．

4.1.4 外部割り込み

共有しなければならない割り込みは，第一に，タイマー割り込みといったレガシーな割り込みである．第二には，PCI 割り込みである．これは，PCI デバイスが PCI の割り込み線を共有している場合は，デバイスを分割していても割り込みを共有しなければならない．

これらの外部割り込みを制御する IO APIC の割り込み伝達方法の一つとして "fixed" と呼ばれるモードがあり，これを利用することにより，複数の CPU に対して同時に一つの割り込み要求を伝達することができる．

その際に用いられる論理 APIC ID は，元の Linux のコードでは `cpumask` と同一であるので，これを全カーネルの CPU で重ならないようにオフセットを設けた．加えて，割り込みのマスク処理は，他のカーネルが割り込みを有効にしている可能性があるので，その時の IO APIC の設定を読み，完全にマスクするか，割り込みの伝達先からそのカーネルに属する CPU を除くのみかを判断するように変更した．affinity の設定処理部分も同様に変更した．

他方，MSI(Message Signaled Interrupt) を利用している PCI デバイスの場合は，デバイスごとに割り込み先及びベクタを設定できるため，デバイス分割のみによって割り込みの課題は解決できる．

4.1.5 プロセッサ間割り込み

通常の Linux では，IPI の宛先の指定方法として short hand モードと呼ばれるモードを利用する．これは，宛先を "全ての CPU", "自分以外の全ての CPU", "自分" の 3 つから指定するものである．

ここでは，直接論理 APIC ID で宛先を指定することによって，他のカーネルの CPU に対して割り込みが発生しないようにした．

4.1.6 ブート処理

カーネルローダーモジュールは，第 4.1.2 節で示した低位領域に，リアルモードからプロテクトモードへの移行処理コードを含む部分をロードし，カーネルの展開コード及び圧縮カーネルを，起動されるカーネルの独立領域にロードする．

このとき，起動されるカーネルの独立領域は，元のカーネルのメモリマップ外であるので `ioremap` を使って一時的に元のカーネルのメモリ空間にマップを行った上でロードを行う．

4.1.7 仮想ネットワーク

仮想ネットワーク機器間での通信は，`alloc_skb` 関数を置き換え，共有メモリ領域から `sk_buff` を確保することにより，データ領域のアドレスをそのまま宛先のカーネルに渡すことで行われる．受け取ったカーネルは，データ領域を共有メモリの解放関数に与えることにより解放を行う．この方法により，カーネル間においてメモリコピーが発生しない通信を実現すること

ができる．

送信，受信の通知は IPI を用い，相手のカーネルの一つのコアに対して割り込みをかけることにより実現する．

4.2 SH における実装

本節では，SH RP1 用の Linux(2.6.16 ベース) に対しての SHIMOS の実装について述べる．

4.2.1 CPU

CPU コアの分割は，カーネルが使用できる CPU をその ID のリストで指定するコマンドラインオプションを用いることで，`maxcpus` だけでは実現できない，認識される CPU の制限を行うことができる．

4.2.2 メモリ

SH アーキテクチャでは，メモリマップの一部が固定されており，特にカーネルが利用する領域は全ての物理メモリの固定マップとなっている．このため，図 3 で示したようなメモリマップは行えず，各カーネルは，動作・利用する仮想アドレスを違えることによってメモリの分割を実現する．

他方，アーキテクチャ上，特定の物理メモリ領域がカーネル動作に必須になることはないので，物理メモリの開始アドレスとサイズを指定する `MEMORY_START` や `MEMORY_SIZE` の値を変更することにより，メモリの分割を実現できる．

なお，共有領域は最下位にある必要がなく，全物理メモリ領域は明示的なメモリマップなしに，各カーネルからアクセスすることができるので，任意の場所に配置することができる．

4.2.3 デバイス

デバイスを統一的に扱うインタフェースは存在しないため，各デバイスの初期化処理に対し，カーネルごとにそれを行うかどうかの設定変更を行うことで，デバイス分割を実現する．

また，例えばシリアルインタフェースなどはコア数のデバイスが存在するので，コマンドラインオプションなどからそのデバイスをそれぞれ分割するか，初期化処理を変更して認識するデバイスを制限することにより実現できる．

4.2.4 外部割り込み

SH においては，割り込みを共有する必要のあることは少ない．たとえばタイマーも複数存在し，各コアのローカルタイマーとして異なるタイマーを用い，異なる IRQ となる．

このため，SHIMOS の SH における実装については，INTC の初期化処理のうち，不必要な割り込みの無効化処理を取り除くことで，割り込みの割り振りを行うことができる．

4.2.5 プロセッサ間割り込み

プロセッサ間割り込みは，各コアに対応するレジスタに書きこむことで発生させることができる．

x86 での short hand モードに対応するモードはなく，対象となるコアの物理 ID に対応するレジスタへの書き込みを行う．また，この際，Linux 内の論理的なプロセッサ番号 (`smp_processor_id`) から物理コア番号への変換テーブルにより変換を行うので，一部のコアを使っている場合でも正常にプロセッサ間割り込みを処理することができる．

表 1 評価に用いたマシンの仕様

CPU	Intel Xeon 5130 (dual-core, 2.0GHz) x 2 (x86)
	Renesas RP1 600MHz (SH-4A 4 コア CPU) (SH)
メモリ	DDR2 667MHz FB-DIMM 1024 MB x 2 (x86)
	DDR2 200MHz SDRAM 64 MB x 2 (SH)
HDD	SATA 250GB x 3, PATA 120GB (x86)
	PATA 120GB (SH)
OS	Linux 2.6.24 (x86 版)
	Linux 2.6.24.3 (x86 Xen Dom0 用 Ubuntu 版を使用)
	Linux 2.6.16 ベース (SH)
NIC	Broadcom (`tg3`, オンボード), Intel (`e1000`, PCI-X) (x86)
	SMSC (100Mbps Ethernet) (SH)

4.2.6 ブート処理

アーキテクチャ上起動できるアドレスに制限がないため，カーネル展開コード等を含めて起動されるカーネルの独立領域に配置する．`ioremap` 等を行う必要がなく，配置したカーネルコードからブートさせることで別パーティションの起動が実現できる．

5. 評価

5.1 実行環境

この節では，SHIMOS で同じ二つの OS を実行した場合の評価を示す．

x86 アーキテクチャにおいては，SHIMOS と Xen, VMWare の二種類の仮想マシンを用いて，二つの OS を実行させ，その上で各種ベンチマークプログラムを動かし，その結果を比較した．

比較に使用したマシンの仕様は，**表 1** に示す通りである．使用した仮想マシンのバージョンは，Xen 3.2.0 と VMWare ESX Server 3.5 Update 1 である．

x86 での評価では，各 OS に，CPU は 2 コアずつ，メモリは 896MB を与え，それぞれ一台ずつ SATA と PATA の HDD を認識させた．ただし，VMWare はカーネルの制約により両方 SATA の別の HDD とし，直接 HDD を扱うことはできなかったので仮想ディスクイメージを別の HDD 上に生成した．また，NIC は，SHIMOS ではそれぞれ別の NIC を認識させ，各仮想マシンでは別の NIC にブリッジするデバイスを認識させた．

SH での評価では，各 OS に，CPU を 2 コアずつ，メモリを 64MB ずつ与え，片方には NIC を認識させ，もう片方には HDD を認識させ，それぞれ NFS と HDD 上の ext3 パーティションを OS のルートファイルシステムとした．

```
for(i=0;i<100000;i++){
  pid = fork();
  if(pid == 0){
    return 0;
  }
  waitpid(pid,NULL,0);
}
```

図 4　forkwait の要点

表 2　x86 上での forkwait 実行結果
(Native・単独実行を 100 としたときの速度．高いほど高性能)

	単独実行	並列実行
Native	100.0	79.9
SHIMOS	99.7	99.1
Xen	27.5	27.4
VMWare	20.7	20.1

5.2　マイクロベンチマーク

前節で述べた各アーキテクチャの評価環境上で，いくつかのマイクロベンチマークを動かし，SHIMOS の特徴である少ないオーバーヘッドを確かめた．

まず，`forkwait` ベンチマークを各環境上で動かした．このベンチマークは，**図 4** で示すようなプログラムで，プロセス生成・切替のオーバーヘッドを計測するものである．これをネイティブ，SHIMOS 及び各 VM 上で動作させた．この結果を**表 2** に示す．

以下の結果において，「単独実行」とは 1 台の計算機上で 1 つの OS を動作させて，ベンチマークを 1 つ動かした場合の性能である．「並列実行」は，1 台の計算機上で 2 分割し，それぞれの分割で OS を実行し，ベンチマークを 1 つずつ同時に実行した場合の性能である．ただし，ネイティブでの並列実行の評価は，分割せず計算機全体で 1 つの OS を走らせ，2 つのベンチマークプログラムを実行したものである．

この結果により，特権命令の実行に関して VM では 4 倍弱から 5 倍程度の時間がかかっていることがわかる．他方，SHIMOS では，CPU が直接実行を行うため，通常のネイティブのカーネル実行とほぼ同様の速度が得られている．同時実行では，ネイティブでは 20% 遅くなるが，SHIMOS 及び各仮想マシンではほぼ変わらないことがわかる．

次に，SH での単独実行及び並列実行結果を**表 3** に示す．この結果から，SHIMOS では特権命令に関わる速度がネイティブの場合と同様であることがわかる．また，同時に実行した場合の結果では，ネイティブではほぼ 2 倍の時間がかかるのに対し，単独で実行した場合より SHIMOS では 3% 程度しか遅くならず，二つのカーネルが十分分離されていることがわかる．

5.3　コンパイルベンチマーク

次に，実際のアプリケーションに近いベンチマークとして，それぞれの実行環境上で，Linux 2.6.24 のカーネルのコンパイルを行い，その時間を測定した．

表 3　SH 上での forkwait 実行結果
(Native・単独実行を 100 としたときの速度．高いほど高性能)

	単独実行	並列実行
Native	100.0	50.4
SHIMOS	99.6	93.1

表 4　x86 上での Linux Compile 実行結果
(Native・単独実行を 100 としたときの速度．高いほど高性能)

	単独実行	並列実行
Native	100.0	99.5
SHIMOS	99.8	99.7
Xen	71.7	69.5
VMWare	84.8	81.8

表 5　SH 上での Linux Compile 実行結果
(それぞれ Native 単独実行を 100 としたときの速度)

	単独実行 (NFS)	並列実行 (NFS)
Native	100.0	88.2
SHIMOS	99.8	93.0
	単独実行 (HDD)	並列実行 (HDD)
Native	100.0	96.0
SHIMOS	99.4	103.5

コンパイルするカーネルは `kernel.org` の配布しているものをそのまま用い，x86 に関してはデフォルトの設定 (`defconfig`) で，SH については最小の設定 (`allnoconfig`) でコンパイルを行った．なお，コンパイル時の並列数は x86 では 4，SH では 2 である．

x86 上で，このベンチマークを片方のパーティションのみで行った場合，及び，同時に両方のパーティションで行った場合の結果を**表 4** に示す．

この結果から，直接ネイティブで実行した場合に比べ，仮想マシンではおよそ 15.2% から 38.3% 遅くなっていることがわかる．他方，SHIMOS ではほとんど直接実行した場合と変わらないことが確認できる．並列に実行した場合もほぼ同様の結果となっている．なお，forkwait の結果と異なって，VMWare のほうが Xen よりコンパイル時間は短いが，これは，VMWare では仮想ディスクでなく直接 HDD の仮想マシンでのマウントが行えなかったためで，ディスク環境が厳密には異なるためである．

また，SH 上で同様に，ベンチマークを変更を加えない Linux と，SHIMOS 上で実行した場合の結果を**表 5** に示す．なお，第 5.1 節で述べたように，各 OS のファイルシステムは異なるので，NFS と HDD の上での結果をそれぞれ別に示す．

単独実行での結果からネイティブでの実行とほぼ変わらない速度で実行できていることがわかる．また，同時実行の結果では，ネイティブの 1 カーネルでの実行よりも 5.4% から 7.8% 改善している．

5.4　分割効果のベンチマーク

最後に，SH 上で SHIMOS による分割の効果を確かめるために，Apache と Dhrystone (Unix Bench[12] に含まれるもの) を同時に動かすベンチマークを行った．

表 6 Apache/Dhrystone ベンチマークの実行結果
(それぞれ Native・単独実行を 100 としたときの速度)

	単独実行 (Apache)	並列実行 (Apache)
Native	100.0	99.2
SHIMOS	99.4	99.4
	単独実行 (Dhrystone)	並列実行 (Dhrystone)
Native	100.0	97.8
SHIMOS	100.0	100.0

二つの OS が同時に動いている状態で，片方の OS 上では Apache を起動し，もう一方では Dhrystone を 2 プロセス同時に起動させた．このとき，Dhrystone を繰り返し実行する間に，1Gbps Ethernet で接続されている Core Duo 2.16GHz (メモリ 2GB) のマシンから，Apache Bench[13] によって評価マシンのメモリ上の 16MB の静的ファイルに対して，同時接続数 4 でリクエストを発行し，このときの Apache Bench の記録及び Dhrystone の実行速度を測定した．それぞれのベンチマーク結果を，単独で実行した場合の性能とともに，**表 6** に示す．

単独実行時では，SHIMOS では Apache Bench の結果は若干悪いが，同時実行した場合に，ネイティブの場合では Dhrystone の値が 2.2% ほど低下するのに対し，SHIMOS による分割をした各 OS では両ベンチマークともに単独実行と比べて変化が極めて小さい．これは，IO 処理と計算を SHIMOS による分割で別カーネル上で動かしたことで，相互の作用が減った効果と考えられる．

6. 関連研究

OpenVZ[14], Linux VServer[15] は，OS レベルで資源を分割するものである．このため，OS の特権命令に対するオーバーヘッドは存在しないので，高速に動作させることができる．しかし，カーネルは一台のマシンで一つであり，異なる OS を複数動作させることはできない．

TwinOS[16] は，一つのマシン上で複数の Linux を起動する方式である．これは CPU を時分割し，それ以外の資源は分割するもので方法であるが，単一プロセッサを対象としており，OS の切替について時分割によるコストが存在する．またそのため，実時間性を保障することも簡単ではない．

OS Switching[17] は，マシンの電源管理機能であるサスペンド・レジューム機能を利用し，複数の OS をネイティブスピードで動作させるものである．一つの OS がサスペンドした後，切替コードが動作し，別の OS が再開できるようにハードウェアの状態を変更する．この方法では，各 OS は，全てのデバイスを使うことができるが，同時に複数の OS を動かすことはできない．

マルチコア・マルチプロセッサマシンを OS で有効に利用する方法の一つとして，AsyMOS[18], Piglet[19] がある．これは，マシン上の CPU のうち 1 つでデバイスの処理のみを行う軽量カーネルを実行し，残りの CPU で Linux を動かすものである．デバイスへのアクセスは，軽量カーネルを介して行われる．またデバイスの状態取得は，割り込みでなく軽量カーネルからのポーリングによるため，全体として I/O 処理の改善が行われるものである．

組み込み用途の仮想マシンとして，L4VM[20] がある．これは，マイクロカーネル L4 上に QEMU を用いて，複数の OS を起動するだけでなく他アーキテクチャの OS を修正なしに起動することができる．しかし，これは他アーキテクチャの資産を利用することを目的としており，コード変換には大きなオーバーヘッドが存在するので実行速度の面では劣る．

7. 終わりに

本論文では，複数カーネルの並列実行機構である SHIMOS の x86 に加えて，SH アーキテクチャ上での実装を示し，複数のアーキテクチャに適用可能であることを確認した．

また，SH アーキテクチャでの実装においても同様にオーバーヘッドが少なく，直接実行した場合とほぼ同じ結果が得られることをベンチマークで示し，さらに，Linux コンパイルを同時に実行した場合，5.4% から 7.8% 改善すること，重い IO 処理と計算処理をカーネルを分割して実行させる場合で 1 カーネル内で実行時よりも 2.2% 改善することを示した．

謝辞 本研究の一部は，科学技術振興機構 戦略的創造研究推進事業 (CREST) (領域名：実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム) 技術課題：「高信頼組込みシングルシステムイメージ OS」による．

本論文におけるマルチコアチップ及び実行環境は，NEDO リアルタイム情報家電用マルチコアプロジェクトにて早稲田大学（笠原・木村研究室），(株) ルネサステクノロジ，(株) 日立製作所により開発されたものを利用している．

本論文における VMWare ESX Server 上でのベンチマーク評価は，VMWare 社の Free Academic License によって，性能評価が許可された製品を用いて行われた．

参考文献

1) Borden, T.L., Henessy, J.P. and W.Rymarczyk, J.: Multiple operating systems on one processor complex, *IBM Systems Journal*, Vol.28 No.1, pp. 104–123 (1979).
2) Meyer, R. A. and Seawright, L. H.: A virtual machine time-sharing system, *IBM Systems Journal*, Vol.9 No.3, pp.199–218 (1970).

3) Shimosawa, T., Matsuba, H. and Ishikawa, Y.: Logical Partition without Architectural Supports, *IEEE International Computer Software and Applications Conference* (2008).
4) Popek, G. J. and Goldberg, R. P.: Formal Requirements for Virtualizable Third Generation Architectures, *Communications of the ACM*, Vol.17 No.7, pp.412–421 (1974).
5) Adams, K. and Agesen, O.: A Comparison of Software and Hardware Techniques for x86 Virtualization, *ACM SIGOPS Operating Systems Review*, Vol.40 Issue 1, pp.95–99 (2006).
6) Whitaker, A., Shaw, M. and Gribble, S.D.: Scale and performance in the Denali isolation kernel, *ACM SIGOPS Operating Systems Review*, Vol.36 Issue SI, pp.195–209 (2002).
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, *Proceedings of the ACM Symposium on Operating Systems Principles*, pp.164–177 (2003).
8) Neiger, G., Santoni, A., Leung, F., Rodgers, D. and Uhlig, R.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization, *Intel Technology Journal*, Vol.10 Issue 3, pp.167–177 (2006).
9) Advanced Micro Devices: *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual* (2005).
10) Yoshida, Y., Kamei, T., Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., Odaka, T., Takada, K., Kimura, K. and Kasahara, H.: A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption, *2007 IEEE International Solid-State Circuits Conference(ISSCC2007)*, pp. 100–590 (2007).
11) ルネサステクノロジ：SH-4A ソフトウェアマニュアル (2004).
12) Niemi, D. C.: Unixbench 4.1.0, `http://www.tux.org/pub/tux/benchmarks/System/unixbench/`.
13) The Apache Software Foundation: ab - Apache HTTP server benchmarking tool, `http://httpd.apache.org/docs/2.2/programs/ab.html`.
14) OpenVZ Development Team: OpenVZ, `http://wiki.openvz.org/`.
15) Linux-VServer project Team: Linux VServer, `http://linux-vserver.org/`.
16) 田渕正樹，伊藤健一，乃村能成，谷口秀夫：二つの Linux を並存走行させる機能の設計と評価，電子情報通信学会論文誌， Vol.J88-D1 No.2, pp. 251–262 (2005).
17) Sun, J., Zhou, D. and Longerbeam, S.: Supporting Multiple OSes with OS Switching, *USENIX Annual Technical Conference*, pp.357–362 (2007).
18) Muir, S. and Smith, J.: AsyMOS-an asymmetric multiprocessor operating system, *Proceedings of Open Architectures and Network Programming*, pp.25–34 (1998).
19) Muir, S. and Smith, J.: Functional divisions in the Piglet multiprocessor operating system, *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pp.255–260 (1998).
20) Kinebuchi, Y., Koshimae, H., Oikawa, S. and Nakajima, T.: Virtualization Techniques for Embedded Systems, *The 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006)* (2006).