ゲストOSにおけるNUMA対応スケジューリング

林　遼[1,a)]　味曽野 雅史[1]　品川　高廣[1]

概要：近年，クラウドコンピューティングのように仮想マシンを活用するシステムにおいても，Non-Uniform Memory Access（NUMA）アーキテクチャのマシンが利用されるようになってきた．しかし，従来の仮想マシンでは，仮想マシン間の資源配分を柔軟におこなうために，ホストの物理マシンの NUMA 構成が隠蔽されていたり，部分的にしか公開されていなかったりするため，ゲスト OS が NUMA 構成を考慮した最適な資源スケジューリングをおこなうことが難しかった．本研究では，ゲスト OS において NUMA 構成を考慮した資源スケジューリングを可能にしつつ，仮想マシン間の柔軟な資源配分も可能にする手法を提案する．提案手法では，ホストの物理マシンの NUMA 構成全体を全ての仮想マシン上に完全に再現することにより，ゲスト OS による NUMA 構成の正確な認識を可能にする．一方，仮想マシンモニタとゲスト OS の間で協調することにより，仮想マシン間での動的な資源受け渡しを可能にする．提案手法を Linux の cgroup を用いた CPU 動的受け渡し機構と，NUMA ノードを指摘できるように拡張したバルーンドライバによるメモリ動的受け渡し機構により実装した．NAS Parallel Benchmarks を用いた評価実験をおこなった結果，提案手法によって実行時間が最大 55%削減されることを確認した．

1. はじめに

Non-Uniform Memory Access（NUMA）アーキテクチャは，クラウドコンピューティングのためのデータセンターにおいても広く利用されるようになってきた [14,17]．NUMA アーキテクチャでは，CPU とメモリが同一のノード（ローカルノード）にある場合はメモリアクセスが高速であるが，異なるノード（リモートノード）にある場合は低速であるという非対称構造になっており，メモリアクセスの局所性を利用して頻繁に参照するデータをローカルノードに置くことで，システム全体としてメモリアクセスの高速化を実現している．従って，メモリアクセスを頻繁におこなうアプリケーションを効率よく実行するためには，OS が NUMA ノードの構成を考慮してプロセスの資源スケジューリングをおこなう必要がある．例えば Linux では，OS の NUMA 支援機構 [1,2] がプロセスのメモリアクセス状況を監視して，CPU やメモリのリモートノードからローカルノードへのマイグレーションなどをおこなっている．

一方，クラウドコンピューティングにおいては仮想化環境を用いることが多くあるが，従来の仮想化環境では仮想マシン上で動作するゲスト OS が NUMA 構成を十分に活用した資源スケジューリングをおこなうことが難しい．これは，従来の仮想化環境では，仮想マシンに割り当てる CPU やメモリなどの資源を動的に変更して仮想マシン間の資源配分を柔軟におこなえるようにするために，通常はホストの物理マシンとは独立した仮想化されたハードウェア環境が提供されており，ホストの物理マシンの NUMA 構成がゲスト OS からは隠蔽されているためである．

仮想化環境においても NUMA 構成を活用する既存手法としては，主に 1) 仮想マシンの起動時にホストの物理マシンの NUMA 構成を部分的に再現する **vNUMA 方式**と，2) 仮想マシンモニタが仮想マシンの稼働状況を監視して NUMA 構成を考慮した資源スケジューリングをおこなう **Blackbox 方式**の 2 種類に分類される [10]．vNUMA 方式では，ゲスト OS が NUMA 構成に基づいた効率のよい資源スケジューリングをおこなうことが可能であるが，現在の OS は NUMA 構成が実行時に動的に変化することを想定していないため，仮想マシンの NUMA 構成は起動時に静的に決定する必要があり，仮想マシン間の資源配分を実行時に柔軟におこなうことが難しい．一方 Blackbox 方式では，仮想マシンモニタが仮想マシン間の資源配分を自由におこなえる一方，仮想マシンモニタからは仮想マシン上のゲスト OS のプロセスの稼働状況を正確に把握することが難しく，プロセスの統計情報を活用した最適な資源スケジューリングをおこなうことが難しい．

本研究では，ゲスト OS が NUMA 構成を考慮した最適な資源スケジューリングをおこなえるようにするために，まず物理マシンの NUMA 構成全体を全ての仮想マシン上

[1] 東京大学
University of Tokyo
a) ryo_hayashi@ipc.i.u-tokyo.ac.jp

に正確に再現する．これにより，ゲスト OS が物理的な NUMA 構成を正確に認識してプロセス単位での資源スケジューリングをおこなえるようにする．一方，CPU やメモリのオーバーコミット状態を解消して各仮想マシンに動的かつ柔軟に資源配分をおこなえるようにするために，仮想マシンモニタがゲスト OS と連携することによって，ゲスト OS が認識している CPU やメモリの一部の利用を制限することにより，ゲスト OS が全ての CPU やメモリを認識した状態でありつつも，実質的には仮想マシン間での動的な資源受け渡しがおこなえるようにする．

提案手法の実装には，ゲスト OS として Linux，仮想マシンモニタとして QEMU/KVM を用いた．CPU の動的受け渡しには Linux の cgroup 機構を利用し，仮想マシンモニタと連携するゲスト OS 内のエージェントでゲスト OS の全てのプロセスが所属するプロセスグループを作成し，このグループに対して CPU のアフィニティを設定することで利用可能な CPU を制限した．一方，メモリの動的受け渡しには，既存のバルーンドライバを拡張して NUMA ノードを指定したバルーニングをおこなえるようにし，仮想マシンモニタが指定した NUMA ノードのメモリを縮小したり拡張したり出来るようにした．

性能評価として，Intel Xeon X5690 と AMD Threadripper 2990WX を NUMA 構成に設定した 2 台のマシンを使用した．実験の結果，仮想マシン間で CPU とメモリの受け渡しが想定通り動作することを確認した．また，NAS Parallel Benchmarks による MPI 性能のベンチマーク実験をおこなった結果，AMD マシン上ではゲスト OS 上でのベンチマークの実行時間が Blackbox 方式と比べて最大 55%，vNUMA 方式と比べても最大 34%削減されることを確認した．

2. 関連研究

仮想化環境における NUMA アーキテクチャの活用に関する従来の研究は，大きく分けて vNUMA 方式と Blackbox 方式の 2 種類に分類することができる [10]．

2.1 vNUMA 方式

vNUMA 方式は，Advanced Configuration and Power Interface （ACPI）などを用いて仮想マシンを仮想的に NUMA 構成であるように見せる手法である．vNUMA 方式を提供している仮想マシンモニタとしては，Hyper-V [4]，Xen [5]，VMware [6]，Oracle 仮想マシン Server [3] 等がある．この手法では，ゲスト OS が NUMA を意識したスケジューリングをおこなえるが，仮想マシンを起動後に NUMA 構成を変更することが難しい．

vNUMA-mgr [14] では，ホストの物理マシンの NUMA ノードのサブセットを仮想マシンに再現し，ゲスト OS による NUMA を考慮したスケジューリングを可能にしている．また，NUMA ノードに基づくバルーニングにより，仮想マシン間のメモリ分配を可能にしている．しかし，vNUMA-mgr では仮想マシンの NUMA 構成は依然として起動時に指定する必要があり、起動後に NUMA 構成を変更することは難しい．

XPV [10] は，準仮想化のインターフェイスを拡張して，仮想マシンの NUMA 構成の変化をゲスト OS やソフトウェア実行時ライブラリ（SRL）に通知することで，動的な NUMA 構成の変更に追随出来るようにしている．これにより，ゲスト OS のカーネルやメモリアロケータ等の SRL が，変更後の NUMA 構成を意識したスケジューリングおよびメモリ割り当てをおこなえるようにし，メモリアクセス中心のアプリケーションの実行時間短縮を実現している．しかし，XPV ではゲスト OS のカーネルや SRL が専用の拡張インターフェイスに対応している必要がある．

2.2 Blackbox 方式

Blackbox 方式は，仮想マシンを NUMA 構成にはせず，仮想マシンモニタやホスト OS が NUMA 構成を意識した資源スケジューリングをおこなう手法である．Blackbox 方式を提供している仮想マシンモニタとしては，KVM [1] や Xen [7] が挙げられる．この手法では，仮想マシンに割り当てる資源が一つの NUMA ノードで足りる場合には，単一ノードのみを仮想マシンに割り当てることでリモートノードへのアクセスを防ぐことができる．しかし，複数の NUMA ノードにまたがる資源が必要な場合，ゲスト OS が NUMA 構成の情報を認識できないため，効果的な資源スケジューリングが難しくなる．

vProbe [17] では，仮想マシンモニタが一定の周期ごとに各 vCPU のメモリアクセス量をパフォーマンスカウンタで取得し，ローカルノードへのアクセスが多くなるように vCPU を適切な NUMA ノードに割り当てる．また，Rao [15]，Han [12]，Liu [13] らも，CPU の統計情報を用いることで，仮想マシンモニタがゲスト OS からは透過的に NUMA ノードを考慮した資源スケジュールを実現する手法を提案している．しかし，Blackbox 方式は vNUMA 方式に比較して資源スケジューリング効率が低下することが指摘されている [10]．

3. 提案手法の設計と実装

3.1 概要

提案手法では，ゲスト OS による効率的なスケジューリングを実現するために，まず仮想マシン上で物理マシンと全く同じ NUMA 構成を再現する．NUMA ノード数，CPU 数，メモリ容量，各ノード内の CPU 数およびメモリ容量を再現することで，物理マシンと等価な NUMA 構成の物理マシンを作成し，ゲスト OS がプロセスの資源スケジューリングを最適化出来るようにする．

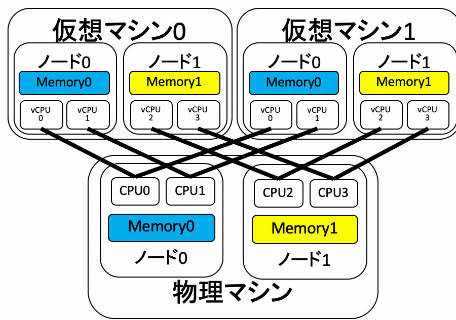

図 1　NUMA 再現図．黒線がピンニング，メモリの色がメモリバインドの対応関係を表す．

NUMA の再現には，仮想マシンモニタの仮想マシン作成オプションを用いた．本手法では，libvirt を用いて QEMU の仮想マシン作成オプションを指定した．さらに，ピンニングを用いて任意の vCPU のスケジューリング対象となる pCPU を一対一対応させた．また，メモリバインドを用いて仮想マシン上の各 NUMA ノードのメモリを対応する物理マシンの NUMA ノードのメモリから割り当てるよう設定した．これらにより，仮想マシンに割り当てられるリソース（CPU，メモリ）を固定化し，起動後のトポロジーの変化を防ぎつつ，物理マシンと等価な NUMA トポロジーの再現を実現した．

しかし，この手法では物理マシン上に複数の仮想マシンが動作している場合，物理リソースに対応する仮想リソースが仮想マシンの台数分だけ生じる．このリソースオーバーコミットにより，同一の物理リソースの仮想マシン間での奪い合いが生じた場合，性能劣化を招きうる．この問題を解決するために，上記の手法に加えて，仮想マシン間負荷分散機構の導入を提案する．

3.2　仮想マシン間負荷分散機構

仮想マシン間負荷分散機構では，各仮想マシンに物理リソースを分配して，物理リソースに対応する仮想リソースが高々一個である状態を保ち，リソースオーバーコミットを防ぐ．さらに，各仮想マシンの負荷の変化に応じて動的に仮想マシン間でリソースを受け渡し合う機構を動作させることで，起動後の負荷変動への対応も行う．

負荷分散機構は，ホスト OS 上で動作するシステムデーモン balanced と，各仮想マシンにあらかじめ用意されたリソース受け渡し機構が協調して動作する．balanced は各仮想マシンのリソース利用率を監視し，ある仮想マシンにおいてリソース利用率が閾値を超えていることを検知した場合，引き渡す側の仮想マシンと受け取る側の仮想マシン上のリソース受け渡し機構を動作させて，当該リソースの受け渡しを行う．

CPU の受け渡し機構は cgroup を用いて実装し，メモリの受け渡し機構は，NUMA ノードを指定したバルーニングが可能なように拡張したバルーンドライバおよび，同じく拡張した QEMU を用いて実装した．仮想マシン間負荷分散機構の全体像を図 2 に示す．各要素の詳細を次節以降にて述べる．

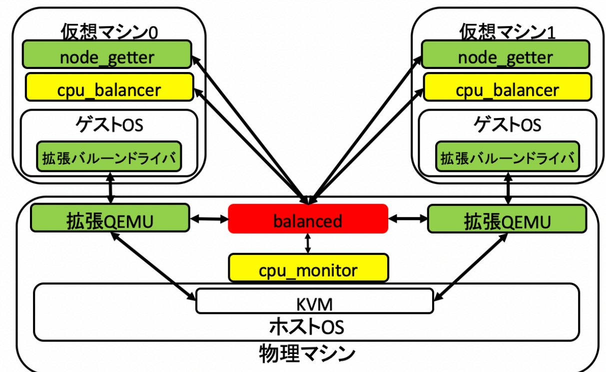

図 2　仮想マシン間負荷分散機構全体図．図中の黄色が CPU，黄緑色がメモリ，赤色が両方の受け渡しに用いる要素を表す．

3.3　cgroup を用いた CPU 受け渡し機構

cgroup は Linux で提供されている機構で，プロセスグループを作成し，このグループに対して一括でリソース利用を制限・隔離する機能を提供する．CPU 受け渡し機構は，cgroup を用いて仮想マシン上の全プロセスを一つのプロセスグループに登録し，このプロセスグループの CPU アフィニティを設定する，シェルスクリプト “cpu_balancer” により実装した．

CPU 受け渡しは，balanced と cpu_balancer および cpu_monitor（後述）を用いて以下の手順で行う．

(1) （balanced の初期化操作 1）各仮想マシン上で cpu_monitor を起動し，全プロセスを一つのプロセスグループに登録

(2) （balanced の初期化操作 2）物理マシン上の CPU 数を仮想マシン数で割った数の CPU が各仮想マシンで利用可能になり，かつ CPU アフィニティが仮想マシン間で被らないように，各仮想マシンのアフィニティを設定

(3) ホスト OS 上の balanced が，libvirt の API を通じて各仮想マシンの CPU 利用率を取得するよう実装した C プログラム cpu_monitor を用いて，CPU 負荷状況をモニタ

(4) ある仮想マシンで CPU 利用率が閾値を超えた場合，負荷が閾値を超えていない仮想マシンに対して，balanced が cpu_balancer を起動して CPU を要求

(5) 要求を受けた仮想マシンは，渡す CPU を外した CPU アフィニティを再設定することで，この CPU を仮想マシン上の任意のプロセスのスケジューリング対象から外す

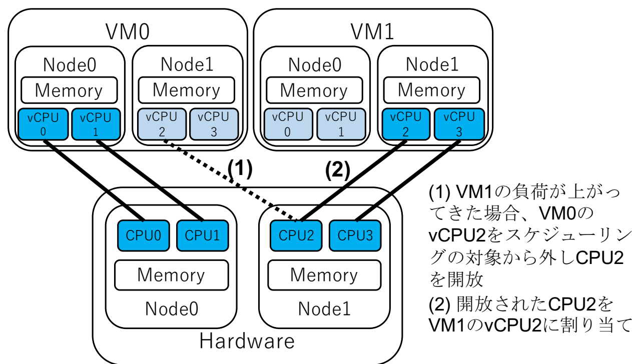

図 3　CPU 受け渡し図.

(6) 渡す側の仮想マシン上で cpu_balancer の実行が終わると，balanced が受け取る側の仮想マシン上で cpu_balancer を起動して CPU を引き渡す
(7) 受け取る側の仮想マシンが，受け取る CPU を含めた CPU アフィニティを再設定することで，この CPU を仮想マシン上の任意のプロセスのスケジューリング対象に含める

また，図 3 に概要図を示す．

3.4　拡張バルーンドライバを用いたメモリ受け渡し機構

バルーンドライバは，仮想マシン上のメモリ容量を動的に拡大・縮小することのできる特殊なデバイスドライバである，KVM の virtio balloon ドライバや，VMware の vmmemctl [8] など様々な実装が存在する．本実装では仮想マシンモニタとして QEMU/KVM を用いたため，virtio balloon を用いた．

virtio balloon は，仮想マシン上のメモリ容量を減らす場合，ゲスト OS 上のページを alloc_pages() 関数で確保し，確保したページを madvise システムコールの MADV_DONTNEED アドバイスによって，しばらく利用することがないと宣言することで，ホスト OS および他の仮想マシンで利用可能にする．この操作を inflate と呼ぶ．

メモリ容量を増やす場合には，inflate 操作した際に確保したページに対し，madvise システムコールの MADV_WILLNEED アドバイスによって，近いうちに利用すると宣言することで，このページをゲスト OS 上で再度利用可能にする．この操作を deflate と呼ぶ．

この機構を利用して，引き渡す仮想マシン上でメモリ容量を減らし，受け取る仮想マシン上でメモリ容量を増やすことで，CPU 受け渡しと同様に仮想マシン間のメモリ受け渡しを行う．CPU 受け渡しとの相違点として，balanced によるメモリ利用率のモニターには，libvirt のコマンドライン API である virsh の，

```
# virsh dommemstat <domain> | grep unused
```

コマンドを用いて，各仮想マシンのメモリ空き容量（/proc/meminfo の MemFree に相当する値）を取得した．ある仮想マシン上でこの値が閾値を下回った場合に，仮想マシン間メモリ受け渡しを実行する．

ただし，現行の virtio balloon では NUMA ノードを指定したメモリ容量の変更を行うことができず，NUMA を意識したメモリの受け渡しを行うことができない．そのため，本稿では virtio balloon を拡張し，NUMA ノードを指定したバルーニングを実現した．具体的には，以下の変更を行った．

- バルーンドライバがゲスト OS 上でページを確保する alloc_pages() 関数を，alloc_page_node() 関数に差し替え
- balloon_dev_info（バルーンデバイス情報記述子）および，QEMU とメモリ情報をやりとりをする virtio_balloon_config 構造体を，配列に拡張
- balloon_dev_info および virtio_balloon_config に関連する処理を，配列を処理できるよう変更

実装には Linux カーネルのバージョン 5.3.8 を用いた．

また，virtio balloon に対してバルーンコマンドを発行する際には，QEMU がコマンド受付インタフェースとなっている，そのため，QEMU も同様に，ノードを指定したコマンドの受付と，変更したコマンドの virtio balloon への転送ができるよう拡張した．具体的には以下の実装を行なった．

- ノードを指定することのできる，新規 QMP（QEMU Monitor Protocol）および HMP（Human Monitor Interface）コマンドの追加
- virtio balloon ドライバとメモリ情報のやりとりをする virtio_balloon_config 構造体の配列への拡張
- virtio_balloon_config に関連する処理を，配列を処理できるよう変更

実装にはバージョン 4.1.91 の QEMU を用いた．

これらの機構と，node_getter（後述）を用いて，以下の手順で仮想マシン間メモリ受け渡しを行う．

(1) balanced が各仮想マシン上のメモリ容量が等しくなるようバルーンコマンドを発行してバルーニング
(2) ある仮想マシンにおいてメモリ空き容量が閾値を下回った場合，balanced が当該仮想マシン上の CPU 利用率が最も高いノードを返す node_getter を動作させ，受け渡すメモリのノードを決定
(3) 引き渡す仮想マシンに対してノードを指定したバルーンコマンドを発行して，一定のメモリ量を減少させる
(4) 受け取る仮想マシンに対してノードを指定したバルーンコマンドを発行して，一定のメモリ量を増加させる

4.　評価

4.1　評価方法

提案手法を動作させた仮想マシン上でのベンチマーク実行速度の評価とその内部状態の計測をおこなった．ベンチ

ⓒ 2020 Information Processing Society of Japan

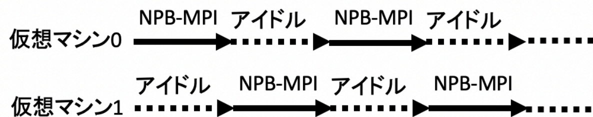

図 4　実験概要図

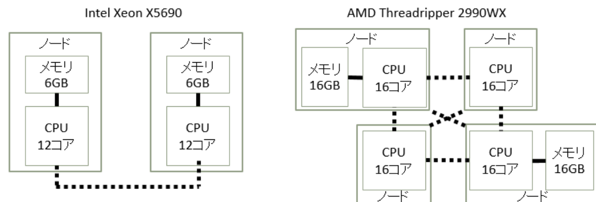

図 5　Intel Xeon, AMD Threadripper のトポロジー図

マーク実行速度に関しては，vNUMA 方式および Blackbox 方式との比較評価を行う．評価にあたり，図 4 に示すように仮想マシンを二つ用意し，一方がアイドル状態を保つ間もう一方がベンチマーク実行，を交互に 5 回ずつ繰り返し，計 10 回の平均実行時間を計測した．この実験を，

CPU 負荷分散セット　i)NUMA 再現なし（Blackbox 方式）
ii)NUMA 再現あり（vNUMA 方式）
iii)NUMA 再現と CPU 負荷分散あり（提案手法）

メモリ負荷分散セット　i)NUMA 再現なし（Blackbox 方式）
ii)NUMA 再現あり（vNUMA 方式）
iii)NUMA 再現とメモリ負荷分散あり（提案手法）

の二つの実験セットに対して行った．ただし，Blackbox 方式

また，Intel Xeon X5690（以下 Intel マシン）と AMD Threadripper 2990WX（以下 AMD マシン）の二つのマシンを用意し，それぞれのマシン上で上記の実験を行うことで，NUMA トポロジーの異なるマシンに対して同様に本手法が有効であるかを検証した．それぞれのトポロジーは図 5 に示す．

OS はホスト・ゲストともに Linux，仮想マシンモニタとして QEMU/KVM を用いた．ベンチマークには並列計算ベンチマークである NAS Parallel Benchmarks [9] の MPI 版（NPB-MPI）を用いた．

本手法では，複数の仮想マシンが存在し，それぞれの仮想マシン上では Web サーバのような計算負荷変動の激しいアプリケーションではなく，科学技術計算のように計算負荷が高い段階 （計算状態）と低い段階（アイドル状態）がはっきり別れているようなアプリケーションが動いている状況を想定する．また，仮想マシンの計算負荷が高い場合に，別の物理マシンへの仮想マシン全体のマイグレートはしない環境を仮定する．

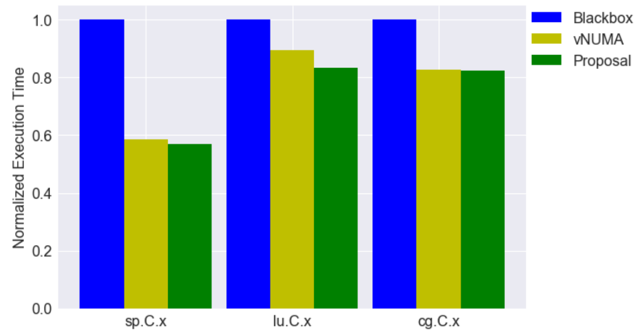

図 6　Intel マシンにおける CPU 負荷分散実験結果

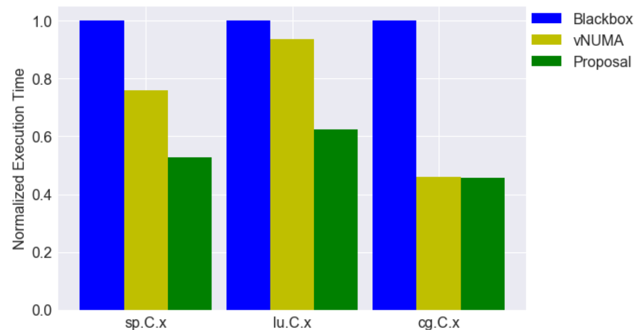

図 7　AMD マシンにおける CPU 負荷分散実験結果

4.2　CPU 負荷分散

CPU 負荷分散セットでは，NPB-MPI が提供する sp.C.x, lu.C.x, cg.C.x の三つのアプリケーションを実行した．Intel マシン，AMD マシン上でのそれぞれの結果を図 6，図 7 に示す．図より，Intel マシン上では Blackbox 方式に対して最大 43%，vNUMA 方式に対して最大 6.8%，AMD マシン上では Blackbox 方式に対して最大 55%，vNUMA 方式に対して最大 34%の実行時間の削減という結果を得た．AMD マシンに比べて Intel マシン上で vNUMA 方式と比較した性能向上が限定的であるのは，NPB-MPI の設定上指定できる並列度が平方数に限られており，Intel マシン上では仮想マシンあたりのコア数が 12 に対して並列度が 16，AMD マシン上では仮想マシンあたりのコア数 16 に対して並列度が 25 であるため，AMD マシンの方が CPU 負荷分散の効果を受けやすいことに起因すると思われる．

また，CPU 負荷実験中の各仮想マシンの CPU 数と CPU 負荷（利用率）の時間変化の様子を図 8 に示す．仮想マシンの CPU 負荷（図の実線）が増減するのに追従して，CPU 数（図の破線）が変化していることがわかる．

4.3　メモリ負荷分散

メモリ負荷分散セットは NPB-MPI が提供するアプリケーション sp.C.x を実行した．Intel マシン，AMD マシン上でのそれぞれの結果を図 9 に示す．図より，Intel マシ

ⓒ 2020 Information Processing Society of Japan

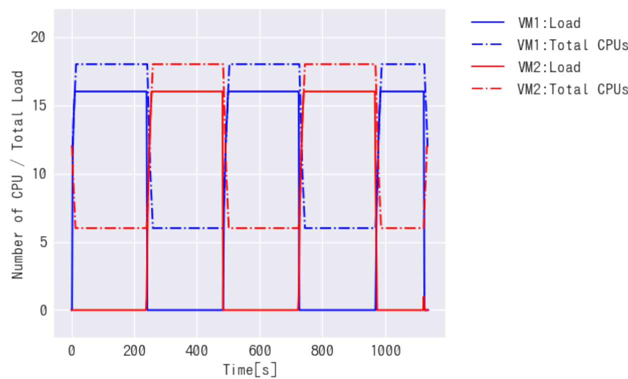

図 8　CPU 負荷分散有効時の CPU 数と CPU 利用率の時間変化グラフ

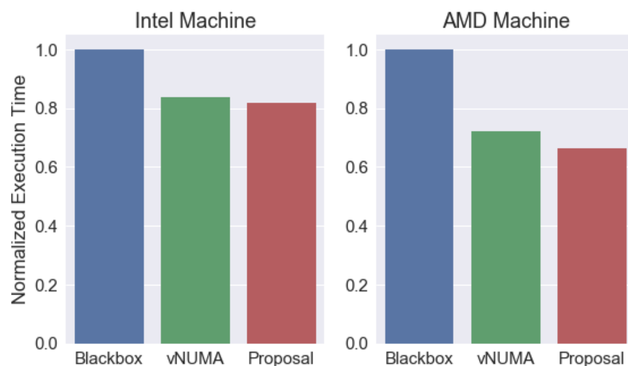

図 9　メモリ負荷分散実験結果

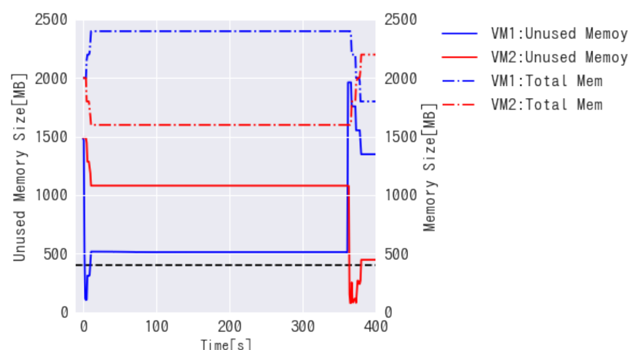

図 10　メモリ負荷分散有効時のメモリ容量とメモリ空き容量の時間変化グラフ

ン上では Blackbox 方式に対して 18%，vNUMA 方式に対して 2.2%，AMD マシン上では Blackbox 方式に対して最大 33%，vNUMA 方式に対して最大 7.8%の実行時間の削減という結果を得た．

また，メモリ負荷実験中の各仮想マシンのメモリ容量とメモリ負荷（空き容量）の時間変化の様子を図 10 に示す．仮想マシンのメモリ空き容量（図の実線）が 400MB に設定された閾値（図の黒点線）を下回ると，メモリ容量（図の破線）が増加していることがわかる．

4.4　負荷分散実行速度評価

本稿の負荷分散機構が実用上有用な速度で動作することを評価するために，CPU 受け渡し，メモリ受け渡しそれぞれの速度を計測した．

まず CPU 受け渡し速度は，渡す側の仮想マシンが CPU アフィニティの再設定の実行を開始してから，受け取る側の仮想マシンの CPU アフィニティの再設定が終了するまでの時間を測定した．

その結果は Intel マシン上では 1.378 秒，AMD マシン上では 1.272 秒で，仮想マシン上でのアフィニティの設定のみの時間は Intel マシンの仮想マシン上では 0.33 秒，AMD マシンの仮想マシン上では 0.27 秒であった．この速度は，高速な vCPU 受け渡しを実現している先行研究 [11] [16] が数十ミリ秒単位での受け渡しを取り扱っていることに比較すると劣後するが，本研究では負荷の変動が緩やかである環境を想定しており，評価実験においても支障が無いことが示されている．

また，メモリ受け渡し速度に関しては，バルーンコマンドを発行してからバルーニングが完了するまでの実行時間を，元のバルーンドライバの実行時間と比較した．400MB を減らした時（inflate），増やした時（deflate）の 2 種類について，Intel マシンと AMD マシンのそれぞれで計測した．その結果は表 1 に示す．

表 1　バルーニング時間計測結果

マシン	ドライバ	inflate[s]	deflate[s]
Intel	バルーンドライバ	0.404	0.367
	拡張バルーンドライバ	0.955	0.837
AMD	バルーンドライバ	0.198	0.168
	拡張バルーンドライバ	0.596	0.557

拡張版バルーンドライバでは，元のバルーンドライバより処理時間が伸びている．これは，ノード数の分だけドライバが処理する情報量およびホスト OS とゲスト OS 間でやりとりする情報量が増えることに起因すると考えられる．

4.5　内部状態評価

本稿ではベンチマーク実行時の仮想マシンの内部状態を評価するために，ベンチマーク実行時の LLC キャッシュミス数とページフォルト数を計測して，Blackbox 方式，提案手法の NUMA 再現のみの場合を比較した．計測には perf を用いて，ホスト OS 上でベンチマーク実行直前に仮想マシンに対応する QEMU プロセスを指定して計測を開始し，ベンチマークの実行完了後すぐに計測を終了した．

図 11 に Intel マシン上の計測結果，図 12 に AMD マシン上の計測結果を示す．それぞれ提案手法によって L3 キャッシュミス数およびページフォルトが減っていることがわかる．この結果より，NUMA 再現を行うことによってキャッシュ効率が向上したこと，そして Blackbox 方式では Automatic NUMA Balancing が NUMA アクセスの統計情報を取るためのページフォルトが発生してしまう一方で，提案手法ではこの機構を必要としないためにページ

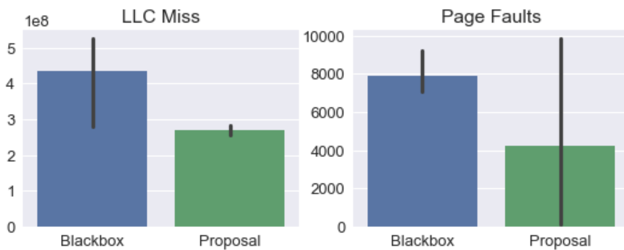

図 11 Intel Xeon における内部状態計測結果

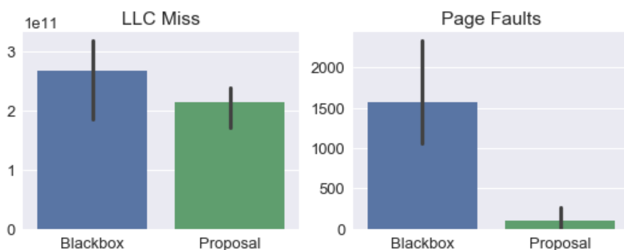

図 12 AMD Threadripper における内部状態計測結果

フォルト数が少ないことがわかる．

5. おわりに

本稿では，近年クラウドでも利用が広がる NUMA アーキテクチャが，仮想化環境では効率的に利用することが難しいという問題に対し，仮想マシン上で NUMA を正確に再現してゲスト OS による効率的なスケジューリングを実現しつつ，仮想マシンモニタとゲスト OS 間で協調して仮想マシン間での資源再配分を行うことで，動的な負荷分散にも対応可能な手法を提案した．提案手法を cgroup および NUMA ノード指定可能なように拡張したバルーンドライバを用いて実装し，評価実験によって，提案手法上でのベンチマークの実行速度が既存手法と比較して最大 55%削減されること，また負荷変化が緩やかな場合に対応可能な速度で動くこと，及び NUMA 構成の再現によりリソース利用効率が向上することを確認した．

参考文献

[1] AutoNUMA: the other approach to NUMA scheduling [LWN.net]. https://lwn.net/Articles/488709/. (Accessed on 01/12/2020).
[2] numad(8) - Linux man page. https://linux.die.net/man/8/numad. (Accessed on 01/12/2020).
[3] Optimizing Oracle VM Server for x86 Performance. https://www.oracle.com/technetwork/server-storage/vm/ovm-performance-2995164.pdf. (Accessed on 01/20/2020).
[4] VMM 2016 コンピューティング ファブリックで仮想マシンの設定を構成する — Microsoft Docs. https://docs.microsoft.com/ja-jp/system-center/vmm/vm-settings?view=sc-vmm-2019. (Accessed on 01/20/2020).
[5] vNUMA in Xen. https://events.static.linuxfound.org/sites/events/files/slides/vNUMA%20in%20Xen_XenDev.pdf. (Accessed on 01/20/2020).
[6] vSphere のリソース管理 - VMware vSphere 6.5. https://docs.vmware.com/jp/VMware-vSphere/6.5/vsphere-esxi-vcenter-server-651-resource-management-guide.pdf. (Accessed on 01/20/2020).
[7] Xen on NUMA Machines - Xen. https://wiki.xen.org/wiki/Xen_on_NUMA_Machines. (Accessed on 02/03/2020).
[8] メモリ バルーン ドライバ. https://docs.vmware.com/jp/VMware-vSphere/6.5/com.vmware.vsphere.resmgmt.doc/GUID-5B45CEFA-6CC6-49F4-A3C7-776AAA22C2A2.html. (Accessed on 01/17/2020).
[9] David H Bailey. NAS parallel benchmarks. *Encyclopedia of Parallel Computing*, pp. 1254–1259, 2011.
[10] Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. When eXtended Para-Virtualization (XPV) Meets NUMA. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pp. 1–15. ACM, 2019.
[11] Luwei Cheng, Jia Rao, and Francis Lau. vScale: automatic and efficient processor scaling for SMP virtual machines. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pp. 1–14. ACM, 2016.
[12] Jaeung Han, Jeongseob Ahn, Changdae Kim, Youngjin Kwon, Young-ri Choi, and Jaehyuk Huh. The effect of multi-core on HPC applications in virtualized systems. In *Proceedings of European Conference on Parallel Processing (EuroPar)*, pp. 615–623. Springer, 2010.
[13] Ming Liu and Tao Li. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In *Proceedings of 41st International Symposium on Computer Architecture (ISCA)*, pp. 325–336. IEEE, 2014.
[14] Dulloor Subramanya Rao and Karsten Schwan. vNUMA-mgr: Managing VM memory on NUMA platforms. In *Proceedings of International Conference on High Performance Computing (HiPC)*, pp. 1–10. IEEE, 2010.
[15] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. Optimizing virtual machine scheduling in NUMA multicore systems. In *Proceedings of 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 306–317. IEEE, 2013.
[16] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule Processes, not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, pp. 1–7. ACM, 2013.
[17] Song Wu, Huahua Sun, Like Zhou, Qingtian Gan, and Hai Jin. vProbe: Scheduling virtual machines on numa systems. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 70–79. IEEE, 2016.

ⓒ 2020 Information Processing Society of Japan