仮想 NUMA マシンの性能及び弾力性の向上

味曽野 雅史[1,a) 林 遼[1] 品川 高廣[1]

概要：NUMA マシンの普及に伴い，仮想化環境でも NUMA マシンを利用することが一般的になってきている．NUMA マシン上で仮想化をおこなう一つの方法が，ゲストにホストの NUMA 構成を一部または完全に再現する仮想 NUMA (vNUMA; Virtual NUMA) である．このときゲスト上では実マシンの NUMA 構成を活用したスケジューリングやメモリ利用が可能となる．vNUMA で VM を作成する場合，ホストの NUMA トポロジを完全にゲスト上で再現した場合（Full vNUMA と呼ぶ）が最も性能が良くなると考えられるが，Full vNUMA に関して詳細に性能を調査した研究は少ない．また，Full vNUMA 環境で他の VM を作成した場合は必然的にオーバコミットが発生するため，vNUMA の優位性が低くなるという課題がある．
本論文ではまず Linux の QEMU/KVM を利用して Full vNUMA 環境を作成し，その性能評価をおこなった．この結果．我々は現在の Linux には，vNUMA 環境において，1) idle 時の halt 状態時の不適切な PV 機能の処理による vCPU の利用率の低下，2) idle 時の過渡なプロセスマイグレーションによる性能低下の問題が生じることを発見した．この問題の解決策として，PV 機能の修正及び idle 時のマイグレーションを抑制する方法を提案する．
また，Full vNUMA 構成で複数 VM を作成した場合のオーバコミットを避けるための方法として，VM 間で協調的かつ排他的にリソースを共有する方法を提案する．これにより Full vNUMA の利点を生かしつつ，状況に応じて柔軟に複数 VM によりリソースを活用することができるようになる．

1. はじめに

今日クラウドコンピューティングは社会の様々なシステムの実現に不可欠な要素となっている．従来クラウド環境は主に商用のアプリケーションやウェブサービスの運用に用いられてきたが，近年は機械学習やハイパフォーマンスコンピューティングなど，高性能のマシンを必要とするワークロードにも利用されるようになっている [21, 40]．クラウドで提供されている高性能な仮想マシンの例として，代表的なクラウドプロバイダである AWS EC2 [1] では 96vCPU (Virtual CPU)，192GiB メモリ及び 5 つの NVMe SSD を搭載したインスタンスを貸し出している．

一方で，多量のリソースを割り当てた仮想マシンが，同量のリソースを持つベアメタルマシンと同等の性能を発揮するかというと，多くの場合はそうはならない．実際に，多くの研究が仮想化環境はベアメタル環境と比較して性能が出ない，あるいはスケールしないといった問題を報告している [22, 33, 34, 47, 48]．これらの仮想化環境での性能低下の要因は主に二つに分けられる．一つは，仮想化によりホストのハードウェア構成が隠蔽されることに伴うハードウェアリソースの非効率的な利用である．そしてもう一つは他の仮想マシンとの競合である．

前者の問題について説明をおこなう．現在の高性能なマシンでは一般に，NUMA (Non Uniform Memory Access) と呼ばれるアーキテクチャ構成が用いられている．前述の高性能な仮想マシンインスタンスでもホストマシンには NUMA マシンが採用されている．NUMA マシンでは CPU とメモリはノードという組に分けられ，あるノード上の CPU からのメモリアクセス速度は，そのノード上のメモリと他のノードのメモリで異なるという性質がある．このようなメモリ構成は I/O 競合を減らしメモリ量のスケールを可能にする．

NUMA マシンの性能を最大限に引き出すためには，OS は NUMA 構成を意識してスレッドのスケジューリングやデータ配置などをおこなう必要がある．仮想化した際にゲストにこの NUMA 情報が伝わらない場合，ゲストは NUMA 構成を意識した動作ができなくなるため，結果としてアプリケーションの性能低下に繋がってしまう．

ゲストにホストの NUMA 情報を伝えるための方法として，ゲストにホストの NUMA 構成の一部または全体を再現

1 東京大学
The University of Tokyo
a) misono@os.ecc.u-tokyo.ac.jp

ⓒ 2021 Information Processing Society of Japan

する仮想 NUMA (vNUMA; Virtual NUMA) がある．本論文では特にホストの NUMA 構成全体を再現する vNUMA を Full vNUMA と呼ぶ．vNUMA ではゲストに NUMA を構成した上で，ゲスト vCPU とホストの pCPU (Physical CPU) 及びメモリを NUMA 構成に合わせてピンニングすることで，ゲストが NUMA 情報を正しく活用できるようにする．

NUMA 上の仮想化環境でアプリケーションの性能を最大限に引き出すには，vNUMA を利用することに加え，VM 間のリソース競合を避けるために特定の vNUMA の仮想マシンに割り当てたリソースを他 VM に割り当てないことが望ましい．しかしながら，このような構成は以下 2 つの問題がある．1 つは，リソース使用率低下の問題である．一般にクラウドで利用されるマシンのハードウェア利用率は多くの場合低いことが知られており [49]，前述のように排他的にリソースを利用する構成はこの問題をより顕著にさせる可能性が高い．そしてもう一つは，VM に割り当てた CPU やメモリのリソースの変化が難しいという問題である．この理由の一つに多くの OS が NUMA 構成の動的変化を想定していないことが挙げられる．リソース量を動的に変化させる手法としては hotplug/unplug があるが，hotplug/unplug は遅かったり，メモリに関しては粒度がスロット単位で大きいといった課題がある．

本論文はこれらの問題を踏まえ，NUMA マシン上で性能を最大限引き出しつつリソース使用効率を向上させるための仮想化手法の実現を目標とする．本研究ではまず，オーバコミットをしない Full vNUMA の仮想マシンが最も性能面で優れた VM であることを確認するために，Linux の QEMU/KVM を利用して複数の仮想マシンを作成し，それらの性能評価をおこなった．ここで我々は予想に反し，Full vNUMA 上で一部のワークロードがベアメタルと比べ顕著な性能低下を引き起こすことを確認した．この問題を分析した結果，現在の Linux には，vNUMA 環境において，1) idle 時の halt 状態時の不適切な PV 機能の処理による vCPU の利用率の低下，2) idle 時の過渡なプロセスマイグレーションによる性能低下 という問題があることが判明した．本論文ではこれらの問題を修正するためのスケジューリングの改善案を提示する．スケジューラの改善により，オーバコミットをしない Full vNUMA は他 VM と比べ最も性能が良くなった．

次に，vNUMA 環境でのリソース利用率を高めるために，排他的リソース共有という手法を提案する．排他的リソース共有では，各 VM は Full vNUMA 構成として作成されるが，実際に利用するリソースは他 VM と協調し，被らない範囲で利用する．このような構成にすることで，オーバコミットされた Full vNUMA 構成でありながら各 VM は実質的に排他的な vNUMA 構成と同等となり，性能低下を最小限に留めることが可能となる．さらに，各 VM が利用できるリソース量を変化させることで，柔軟なリソース変更が可能となる．排他的リソース共有を利用することにより，Full vNUMA の利点を生かしつつ，状況に応じて柔軟に複数 VM によりリソースを活用することができるようになる．

表 1　NUMA ノード間距離の例 (Intel Xeon Platinum 8160)

	ノード 0	ノード 1
ノード 0	10	21
ノード 1	21	10

本論文の貢献は以下の通りである．

- Linux の QEMU/KVM を利用して Full vNUMA 構成の仮想マシンの性能を詳細に評価し，現在の Linux スケジューラに存在する性能低下の問題を明らかにした上で，その改善案を提示した
- Full vNUMA 仮想マシンの弾力性を向上させる手法を Linux の QEMU/KVM に対して提案した

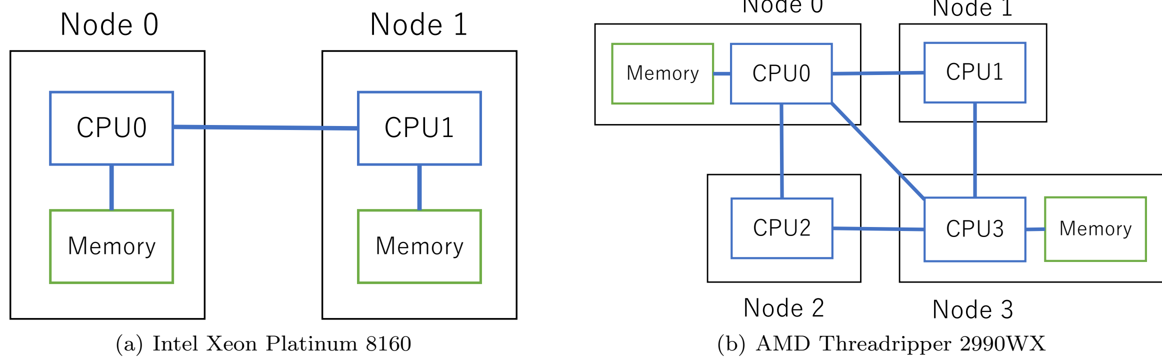
本論文の以降の構成は以下の通りである．まず 2 章で本研究の背景となる NUMA，vNUMA 及び二重スケジューリング問題の基礎知識を説明する．3 章で本研究の目的を述べ，4 章では Linux の QEMU/KVM を用いた Full vNUMA の性能評価をおこない，一部ワークロードで引き起こされる性能低下の改善案を提示する．5 章では vNUMA 環境の弾力性を向上させるための排他的リソース共有について提案する．6 章で本論文に関する議論をおこない，7 章で関連研究についてまとめる．最後に，8 章を本論文の終わりとする．

2. 背景

2.1 NUMA

NUMA アーキテクチャはメモリデザイン手法の一種であり，CPU とメモリは複数のノードという単位に分けられる．UMA (Uniform Memory Access) アーキテクチャとは異なり，NUMA アーキテクチャでは CPU からメモリまでの距離（アクセス速度）が双方のノード位置に応じて異なるという特性がある．ある CPU から見て同一ノードに存在するメモリのことをローカルなメモリ，それ以外のメモリをリモートなメモリといい，それぞれに対するアクセスをローカルアクセス及びリモートアクセスという．ノードによる分割は I/O によるメモリアクセス競合の減少に繋がり，結果としてメモリをスケールさせることが可能となる．今日 NUMA アーキテクチャは CPU とメモリを多く搭載するマシンに一般的である．アプリケーションの性能を引き出すためには，そのアプリケーションをスケジューリングする CPU とそれが利用するデータを配置するメモリは，NUMA 的に同じか近いノードのものを選択することが重要である．

図 1 に NUMA アーキテクチャの例を示す．CPU やメ



(a) Intel Xeon Platinum 8160　(b) AMD Threadripper 2990WX

図 1　NUMA アーキテクチャの例

表 2　NUMA ノード間距離の例 (AMD Threadripper 2990WX)

	ノード 0	ノード 1	ノード 2	ノード 3
ノード 0	10	16	16	16
ノード 1	16	10	16	16
ノード 2	16	16	10	16
ノード 3	16	16	16	10

モリを含む四角で囲まれた領域が 1 つの NUMA ノードを構成する．また 表 1及び 表 2 は 図 1の NUMA アーキテクチャの NUMA ノード間距離を示している．NUMA ノード間距離は，その値が小さいほどそのノード間で CPU からメモリへのアクセスが速いことを意味している．これらの情報は numactl コマンド[*1] [12] を用いて取得したものである．Intel Xeon Platinum 8160 は 2 ノード，AMD Threadripper 2990WX は 4 ノード構成である．なお後者にはメモリが存在しないノードが 2 つ存在している．

2.2　Virtual NUMA (vNUMA)

NUMA の構成情報は ACPI の Static Resource Affinity Table (SRAT) の中に含まれている．仮想マシン起動時にこの SRAT を適切に作成することで仮想マシン上に NUMA を構成することができる．ゲストに NUMA を構成した仮想マシンを Virtual NUMA (vNUMA) という．vNUMA は Hyper-V や QEMU/KVM, Xen, VMWare などの主要なハイパーバイザで利用可能な機能である．

vNUMA を利用する場合，通常はホストの NUMA 構成に沿ってゲスト上に NUMA を再現する．すなわち，ゲスト上に再現した NUMA とホストの NUMA が対応するように vCPU のピンニングやメモリ割り当てをおこなう．明示的な場合を除き，本論文では vNUMA はゲストとホストの NUMA 構成の対応が取れているものを指す．また本論文では特に，ホストの NUMA 構成全体をゲストに再現する vNUMA を Full vNUMA と呼ぶ．

図 2に vNUMA の例を示す．図 2(a)では，2 つの vNUMA インスタンスが存在し，それぞれがホストの資源を均等に分け合っている．vCPU から pCPU へと伸びる直線はその vCPU が pCPU に対してピンニングされていることを，メモリを示す四角の色は対応するホストのメモリを割り当てていることを示している．なお図では CPU は vCPU，pCPU ともにソケット単位で示しており，実際にはこの中に複数のコア及びハードウェアスレッドが存在しうる．図 2(b)は Full vNUMA の例であり，1 つの VM の中でホストの NUMA 構成全体が再現されている．

vNUMA によりゲストは物理的な NUMA の構成を生かしたスケジューリングやメモリ管理ができるようになる．これはゲストに NUMA 構成を隠蔽してホスト側で vCPU スケジューリングやメモリ配置を工夫する手法 [16,29,43,44,50] よりもセマンティックギャップ [23] が小さくなるため，アプリケーション性能的には望ましい [20,22,31,34,45]．4 章では Full vNUMA の詳細な性能評価結果を示す．

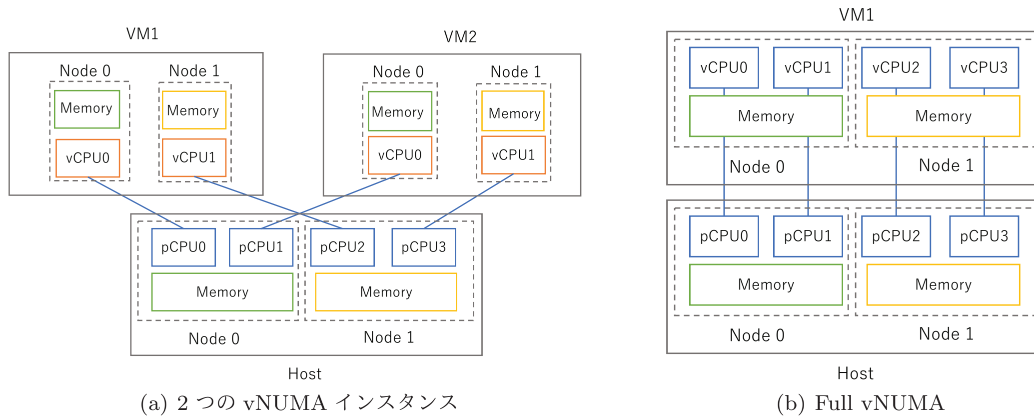
2.3　二重スケジューリング問題

仮想化環境ではゲストによるスケジューリングと，ホストによる vCPU のスケジューリング，二つのスケジューリングが存在する．この二重のスケジューリングは様々な性能低下問題を引き起こすことが知られており，それらを総称して二重スケジューリング問題と呼ぶ．

代表的な二重スケジューリング問題の一つに，Lock Holder Preemption (LHP) 問題がある [26,48]．これはスピンロックを保持している vCPU をプリエンプトするときに発生する．このときそのスピンロックを取得したい他の vCPU は，プリエンプトされた vCPU がリスケジュールされロックを開放するまで待つ必要があり，ロック待ちの vCPU を先にスケジュールすることは CPU 時間の無駄に繋がる．この問題は排他制御が多い並列プログラムを実行したときに特に大きな問題となる．

この他に，順序つきロックで優先度が一番高い状態で待機している vCPU をプリエンプトすることにより発生する Lock Waiter Preemption (LWP) 問題 [48] や，仮想化環境での IPI が遅いことに起因するブロック開放待ち vCPU の wakeup 時に発生する Blocked-Waiter Wakeup (BWW) 問

[*1] `numactl --hardware`



(a) 2 つの vNUMA インスタンス　(b) Full vNUMA

図 2　vNUMA の例— Best viewed in color

題 [25]，割り込みや RCU コンテキストの vCPU をプリエンプションすることで起こる問題 [33, 41] などがある．また vCPU がクリティカルセクション内でプリエンプトされなくても，仮想マシンによる I/O 完了時に vCPU がスケジューリングされていない場合はそれによる I/O の遅延が発生する [51]．

二重スケジューリングの対処に関する研究は多数存在 [25, 26, 33, 41, 48, 51] するが，本質的には仮想マシンに専用の pCPU を割り当て，オーバコミットを避けることで回避可能である．しかしながら，一般に VM の利用率は低い [49] ことが知られており，オーバコミットを避けた場合リソースの有効使用率が低下するという問題がある．この問題の対処方法として，[38, 47] では仮想マシンに割り当てる vCPU 数（=pCPU 数）を動的に変更することで，ある VM が利用していない CPU リソースを他の VM に一時的に貸し出す手法を提案している．なお，いずれの研究もメモリの貸し出しや NUMA の考慮は研究の対象外である．

3. 本研究の目的

クラウド利用のユースケースが広がるにつれ，高性能なクラウドインスタンスの需要が高まっている．NUMA マシン上に構成した仮想マシンで最も良い性能を得る方法は，Full vNUMA を利用し，さらに二重スケジューリング問題を避けるためにオーバコミットを避けることだと考えられるが，これでは 1VM がマシンを占有することになり，その VM が一時的に使用していないリソースを有効活用することができない．

ハイパーバイザ側で Full vNUMA マシンのリソースの一部を一時的に取得する手段があれば，クラウドベンダはそれを利用して，VM の一時的に余ったリソースを他 VM に貸し出し [24, 47] たり，そのリソースを利用したインスタンスを作成して貸し出し [49] たりすることができる．また，プライベートクラウドであれば利用者は既存の VM 環境を壊すことなくそのリソースを利用してもう一つ VM を作成し，他の作業をおこなうこともできる．

本研究は，NUMA マシン上に構成した仮想マシンで性能を最大限引き出しつつ，柔軟に複数 VM によりマシンのリソースを活用することができる弾力性の高い機構の実現を目的とする．このために我々はまず Full vNUMA 仮想マシンの詳細な性能評価をおこない，既存の Full vNUMA の問題点を洗い出し，その修正をおこなう．その上で，vNUMA の一部のリソースを他の用途に利用できるようにするための手法を提案する．

4. 仮想 NUMA マシンの性能向上

本章では QEMU/KVM を用いて Full vNUMA の仮想マシンの性能評価をおこなう．その結果，我々は予想に反して Full vNUMA では一部のケースで性能が極端に性能低下することを観測した．調査の結果，それらの問題はホスト側の Linux のスケジューラの動作に起因しており，スケジューラの修正により性能を改善できることを発見した．本章ではまず性能評価結果について述べた後，性能低下原因についての要因について記述し，さらにそれを修正する方法を示す．

4.1 実験環境

表 3に今回の実験に使用した実験環境を示す．評価マシンは 図 1(a)に示した 2 ノードの NUMA 構成を持ち，1 ノードは 64GB のメモリを持つ．また 1CPU は 24 コアを有し，ハイパースレッディングは無効化した．加えて C State や Turbo Boost などの CPU 周波数を動的に変化させる機能も無効化した．ゲスト OS 及びホスト OS にはともに Linux 5.9 を，ハイパーバイザには QEMU/KVM を利用した．vNUMA 作成には libvirt [10] を用いた．

表 4に実験に利用した仮想マシンの名称と設定を示す．これは次の実験結果で示す図での判例に対応するものである．仮想マシンはいずれも 48vCPU で，メモリはホストの各ノードに 4GB ずつ残し残り全てを VM に割り当てた．

ⓒ 2021 Information Processing Society of Japan

表 3 評価環境

名称	使用物
マザーボード	SuperMicro X11 DPH-T
CPU	Intel Xeon Platinum 8160 x 2
メモリ	SK Hynix DDR4 ECC 32GB x 4
OS (ホスト)	Linux 5.9
OS (ゲスト)	Linux 5.9
ハイパーバイザ	KVM/QEMU (QEMU 5.0)

表 4 実験に利用した VM の設定

名称	説明
baremetal	実マシン上で実行（仮想化なし）
numa	Full vNUMA
numa_sockets	Full vNUMA だがコアを作成しない（本文参照）
pin	UMA，CPU のピンニングだけ実施
uma	UMA
uma_numad	UMA かつホストで numad [16] を動作

また，各 VM では QEMU/KVM でデフォルトの準仮想化機能[*2]を有効にした．

VM の numa_sockets に関して補足する．この VM は，Full vNUMA と同様に NUMA を再現するものではあるものの，CPU の設定がホストのものとは異なり，1CPU 1 コアとして作成される．従って，ゲスト上では 1 ノードに 24CPU があるようにみえる．QEMU で VM を作成する場合，vCPU の設定で（ソケット数，コア数，スレッド数）を指定することができるが，これらを何も設定しない場合ソケット数 = vCPU 数となる．numa_sockets はこのデフォルトの挙動を再現したものになる．numa はホストと同じ 2 ソケット 24 コア構成の VM である．

4.2 実験内容

今回の実験では，OpenMP 版の NAS Paralell Benchmarks (NPB) ベンチマーク [11] を利用した．NPB には十数個の並列計算プログラムが含まれている．今回は NPB の中でディスク I/O の多い “dc” 及び，実行が 1 秒以下で終了する “is” を除いた残り全てのプログラムを実行した．NPB のデータサイズは C を利用し，OpenMP には GNU libomp を利用した．

OpenMP では `OMP_WAIT_POLICY` 環境変数 [13] により，スレッドがロック解放を待つ際にどれだけユーザ空間で待つか（スピンループを実施するか）を指定することができる．`OMP_WAIT_POLICY` の有効な値は “active” あるいは “passive” であり，その挙動は実装依存である．今回実験で利用した GNU libomp では `GOMP_SPINCOUNT` 環境変数 [7] でスピンループする回数を直接設定できるが，もしこの環境変数が未設定の場合，スピンループ回数は `OMP_WAIT_POLICY` が “passive” のとき 0，未設定のとき 300,000 (300k)，“active” のとき 30,000,000,000 (30b (billion)) である．スピンループ回数は大きければ大きいほどその間にロックが解放される確率が高くなり，このスピンループ中にロックが取得できれば最速でのロック取得になるため一般にソフトウェアの性能は向上するが，CPU 利用の観点から見るとスピンした分が無駄がとなり使用電力量も増加する．どの値が適切かはワークロード及びその時の目的に依るため，今回の実験では明示的に `GOMP_SPINCOUNT` を 0, 300k, 30b の 3 つに設定してその三種類それぞれで NPB ベンチマークを実行した．なお，スピンループ回数が上限に達した場合は futex(2) システムコール [6] により当該スレッドはロックが利用可能になるまでスリープする．

4.3 実験結果

図 3に実験結果を示す．ベンチマークは `GOMP_SPINCOUNT` の値に応じてそれぞれ 5 回実行した．グラフの各点はその 5 回の実行結果を示している．グラフ縦軸は実行時間（秒）であり，この値は小さい方が良い．また表 5，表 6，表 7に対応した実験のベアメタルに対する比率を示す．この値が 1 に近ければ近いほどベアメタルと同等の性能であることを意味する．実験結果から，以下のことが示される．

(1) `GOMP_SPINCOUNT` が 30b のとき，VM numa が VM uma と比べ平均で 10%程度，VM numa_sockets よりも 4%性能が良い
(2) `GOMP_SPINCOUNT` が 0 または 300k のとき，VM numa_sockets が VM uma と比べ平均で 5 から 10% 程度性能が良い
(3) `GOMP_SPINCOUNT` が 0 または 300k のとき，VM numa は VM numa_sockets や VM uma と比べて平均で 80% 程度性能悪い

(1) 及び (2) に関してはゲストに NUMA を再現した vNUMA の方が性能がベアメタルに近くということを意味しており，これは想定通りである．しかしながら，(3) に関しては NUMA 構成をよりホストマシンに近づけたもの (VM numa) の方がそうでないもの (VM numa_sockets) よりも性能が極端に悪いという結果となった．次節でこの要因について分析する．

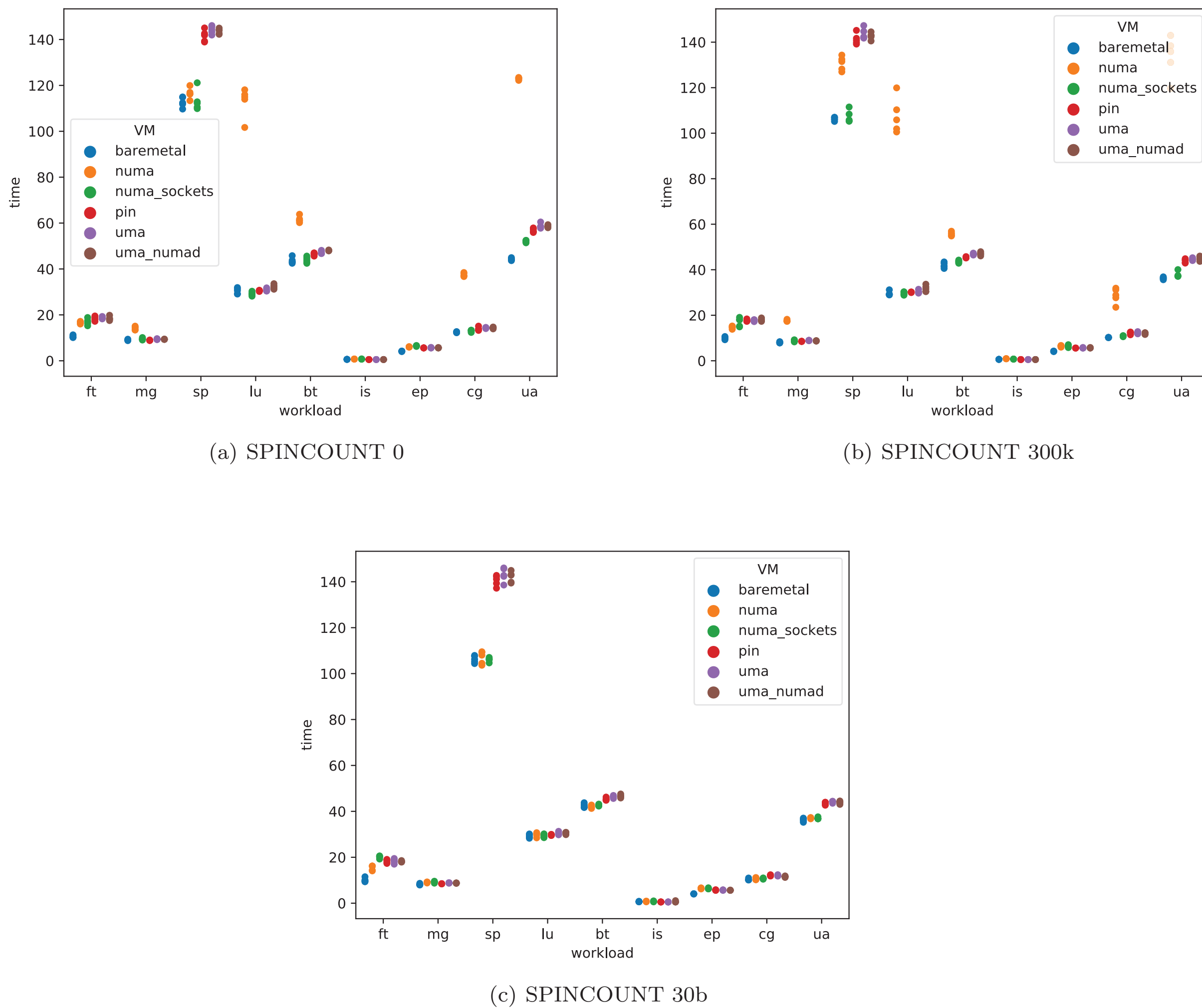
4.4 性能低下の主要因

今回の実験では `GOMP_SPINCOUNT` の値が小さいほど，VM numa の性能低下が観測された．`GOMP_SPINCOUNT` の値が小さいとはすなわち，その分ロック待ちのスレッドがロックをスピンループの間に取得できず，その結果スリープする確率が高いということである．調査の結果，我々はこの性能低下要因を引き起こす二つの Linux スケジューラの問題を発見した．

QEMU/KVM では vCPU の実体は一つの QEMU のスレッドである．そして，各 vCPU は Linux のスケジュー

[*2] https://github.com/qemu/qemu/blob/stable-5.0/target/i386/cpu.c#L4076-L4088 で設定されている

ⓒ 2021 Information Processing Society of Japan



(a) SPINCOUNT 0

(b) SPINCOUNT 300k

(c) SPINCOUNT 30b

図 3 NPB 実験結果 — Best viewed in color

表 5 ベアメタルに対する実験の平均値の相対的な比率 (SPINCOUNT 0)

	ft	mg	sp	lu	bt	ep	cg	ua	幾何平均
numa	1.47	1.63	1.06	3.93	1.42	1.45	3.02	2.77	1.90
numa_sockets	1.64	1.07	1.00	0.97	0.99	1.50	1.01	1.18	1.15
pin	1.67	1.01	1.27	1.00	1.04	1.34	1.15	1.28	1.20
uma	1.72	1.05	1.30	1.04	1.08	1.35	1.15	1.31	1.23
uma_numad	1.71	1.06	1.29	1.07	1.08	1.37	1.17	1.31	1.24

ラ（CFS; Complete Fair Scheduler）によってスケジュールされる．CFS の中で今回重要となるのは，CFS が保持するスケジューラドメイン [14] である．スケジューラドメインは階層的な構造で，一つの階層（ドメイン）には複数の CPU グループが存在する．このドメイン構成はアーキテクチャに依存するが，今回実験で利用した NUMA マシンの場合，ハードウェアスレッド，コア，ソケットでドメイン階層が構成される．CFS は定期的に各 CPU のランキュー*3の偏りを調整（ロードバランス）するが，このときの調整はスケジューラドメインごとにおこなわれる．

*3 実行可能でスケジューリングを待っているスレッドが格納されるキュー

今回 VM numa と VM numa_sockets で差が生じた原因は，前者は 24 コアからなるコアによるスケジューラドメインが構成されたのに対し，後者は 1 ソケット 1 コア構成なのでコアによるスケジューラドメインが構成されない点にある．以下で二つの性能低下要因について具体的に説明する．

4.4.1 PV steal time 機能に起因する性能低下

この問題の具体的な発生条件は以下の通りである．

- KVM の PV SPINLOCK (`KVM_FEATURE_PV_UNHALT`) が有効
- KVM の steal time (`KVM_FEATURE_STEAL_TIME`) が有効

表 6 ベアメタルに対する実験の平均値の相対的な比率 (SPINCOUNT 300k)

	ft	mg	sp	lu	bt	ep	cg	ua	幾何平均
numa	1.50	2.18	1.26	3.53	1.33	1.55	2.64	3.61	2.03
numa_sockets	1.77	1.06	1.02	1.03	1.04	1.54	1.08	1.00	1.17
pin	1.84	1.03	1.33	1.05	1.09	1.38	1.24	1.18	1.25
uma	1.83	1.08	1.39	1.07	1.13	1.40	1.17	1.23	1.27
uma_numad	1.84	1.08	1.39	1.07	1.13	1.38	1.18	1.18	1.26

表 7 ベアメタルに対する実験の平均値の相対的な比率 (SPINCOUNT 30b)

	ft	mg	sp	lu	bt	ep	cg	ua	幾何平均
numa	1.27	1.09	1.00	0.99	1.02	1.57	1.05	1.02	1.11
numa_sockets	1.64	1.09	1.02	1.01	0.99	1.60	1.06	1.02	1.15
pin	1.61	1.04	1.31	1.02	1.05	1.40	1.19	1.20	1.22
uma	1.69	1.08	1.35	1.04	1.09	1.41	1.19	1.21	1.24
uma_numad	1.63	1.11	1.36	1.03	1.09	1.43	1.19	1.19	1.24

- VM の中の CPU がマルチコア

KVM の steal time とは，vCPU が実行されるまでに pPCU を待った時間（ホストが他の処理をおこなっていた時間）をゲストに対して通知する機能である．この機能を利用するためには，ゲストは CPU ごとに用意された専用の MSR にゲストの空いているメモリのアドレスを格納する．KVM は非同期にそのメモリに対して情報を書き込み，ゲストはそのメモリの値を参照することで steal time を確認する．

この KVM steal time 機能で書き込まれる情報には，KVM PV SPINLOCK が有効の場合，steal time の値の他に，その vCPU がプリエンプトしているかという情報が含まれる[*4]．ゲスト側の CFS では，この情報をもとに，コアの CPU ドメインではホスト側で vCPU がプリエンプトされている場合，その vCPU に対してロードバランスをおこなわないようになっている．プリエンプトされた vCPU をロードバランス対象から外す理由は，その vCPU のランキューにプロセスを積んでも結局はホスト側でスケジュールされない限りは実行されないためである．

しかしながら，今回の実験で `GOMP_SPINCOUNT` が小さい時は，スリープによるロック待ちになり，vCPU は HLT VMEXIT を引き起こす．ホスト側では他のスレッドがスケジュールされるが，この時も vCPU にはプリエンプトされたというフラグがつけられる．この vCPU は他 vCPU からの Rescheduling IPI あるいは外部割り込みを待つことになる．ゲスト上ではその vCPU はプリエンプトされたというフラグにより，コアによるスケジューラドメインのロードバランスの対象から外れる．結果として，ホスト側では他に何も動かすものがない状態になっても，ゲストは vCPU がプリエンプトされているものだと勘違いし，その vCPU は全く活用されず，結果大幅な性能低下に繋がる．

この問題の理想的な解決方法は，HLT VMEXIT してプリエンプトのフラグが付けられた後，その vCPU スレッドよりも優先度の高いスレッドがあればそのスレッドを実行後に，そうでなければ直後に，プリエンプトされたというフラグをオフにすることである．そうすることで HLT VMEXIT した vCPU に対してゲストが必要に応じてロードバランスすることが可能となる．今回は簡易的に HLT で VMEXIT したときは vCPU をプリエンプトされているとマークしないことで修正した．

4.4.2 短時間に sleep/wake を繰り返すスレッドによる性能低下

表 5で CPU ピンニングだけした VM pin や numa_sockets が，lu のベンチマークで仮想化のオーバヘッドがあるにもかかわらず，ベアメタルと同じかそれ以上の性能を示しているのはここで説明する問題が理由である．

この問題は以下の手順で発生する．

(1) ロック待ちでスレッドがスリープ
(2) アイドル時ロードバランス[*5]により，別の NUMA ドメインの vCPU から実行可能スレッドがマイグレーションされる
(3) その vCPU 上で最初のスレッドがすぐ wake することで vCPU のランキューが詰まる

すなわち，性能向上を目的としておこなったマイグレーションが実際には余計なマイグレーションになってしまうことが要因である．この問題は仮想化関係なく，CFS の NUMA ドメインのロードバランスに関する問題であり，[36] で指摘されている “Overloaded wake on bug” と実質的に同じ現象である．対策としては以下の 2 つが考えられる．

- sleep/wake が短時間に繰り返されている時，アイドル時マイグレーションを抑制する
- タスクを wake up するとき，NUMA ドメインに関係なくアイドルな CPU があればそこにマイグレーショ

[*4] これはもともと Lock Holder Preemption 検出のために導入されたものである．steal time とは関係の無い情報だが，ゲストに非同期に情報を伝達する手段として steal time があったので，その機構を活用したものだと推測される

[*5] CPU がアイドル時になったときに発生するロードバランス

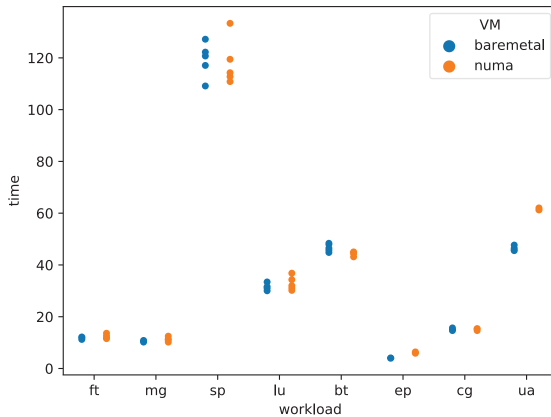


図 4 スケジューラ修正後の実験結果 (SPINCOUNT 0) — Best viewed in color

ンする [*6]

後者は [36] で提案されていた方法であるが，カーネルメンテナの同意を得られず問題は今も修正されていない．前者はこの問題となるそもそもの原因となるアイドル時マイグレーションを抑制するため，結果的に後者よりも無駄な処理が減ると考えられるが，今回は実装が容易な後者の手法を用いてスケジューラを修正した．

4.5 改善方法及び再評価結果

図 4に，今回発見した 2 点の結果を修正した後の実験結果 (SPINCOUNT 0) を載せる．図より．修正により VM numa の性能低下が改善され，ベアメタルと同等の性能が得られていることが確認できる．また，SPINCOUNT 30k でも同様の実験結果を得ている．一方で，今回の修正を施してもなおベンチマーク "ua" に関しては性能低下が見られる．この件に関しては現在調査中である．

4.6 まとめ

Linux の CFS にはスケジューラドメインごとの処理が多く含まれており，単なる CPU 数だけでなく，各 CPU のコア数やスレッド数が CFS に影響することとなる．vNUMA を作成する際は，本来であれば厳密に CPU 構成を再現した方がそれらの CFS の恩恵を得られるはずであるが，今回の実験では Linux に現存する NUMA に関する 2 つスケジューリングの問題（1 つは既知）が明らかとなり，NUMA 上でのスケジューリングアルゴリズムの設計の難しさが露呈する結果となった．

[36] では当該研究で発見した 4 つのスケジューラのバグ全てが，他部分のスケジューリング処理の最適化によるものだと述べている．今回新たに発見した steal time に関する問題も，もとはホスト側のスケジューリングの状態の一部をゲストに伝えることでゲストの性能向上を図るために導入されたものである．しかしながら，準仮想化機能に対する考慮不足が，特定条件での性能低下に繋がっている．QEMU/KVM 上で Linux をゲストで動かす場合，ホストでもゲストでも CFS が利用されるが，ゲストの CFS は準仮想化による最適化によりホストと異なる動作をすることがある．スケジューラ設計の際にはそのことを考慮することも大事である．

*6 CFS は NUMA ドメイン間を跨いだ wake 時マイグレーションをしない

5. 仮想 NUMA マシンの弾力性向上

前章ではスケジューラの問題を修正すれば, Full vNUMA のマシンがベアメタル性能に最も近くなることを示した．本章では Full vNUMA の仮想マシンに対して，他の仮想マシンと排他的にリソースを共有することにより，二重スケジューリングの問題を避けつつ仮想マシンの弾力性を向上させる手法を提案する．この手法を利用することで，vNUMA の利点を生かしたまま，柔軟に複数 VM によりマシンのリソースを活用することができるようになる．

5.1 提案手法

我々は仮想 NUMA マシンの弾力性を向上させるため，排他的リソース共有という手法を提案する．図 5に提案手法の概要を示す．提案手法では，どの仮想マシンも（図の場合は 2 つ）Full vNUMA として作成する．しかし，各仮想マシンが実際に利用するリソースは，他の仮想マシンと干渉しないもののみとする．図の場合，仮想マシン上の灰色で示された部分が，実際にはその仮想マシンは利用しない部分である．

この方法の要点は，どの仮想マシンにも起動時に ACPI を通じて全体の NUMA 構成が再現されることにある．これにより，その NUMA 構成の範囲でのリソース変更が容易になる．加えて，他の仮想マシンと排他的にリソースを利用をすることで，オーバコミットを避け，二重スケジューリングの問題を回避することができる．また hotplug/unplug を利用せずに実質的に利用できるリソース量だけをすることで，高速かつ既存のアプリケーションへの影響を最小限にしたリソース変更が可能となる．なお，実行する仮想マシンは最初から全て起動されている必要はない．必要に応じて起動または終了が可能である．

本研究ではリソース量の配分の管理は管理者がおこなうものとする．リソース割り当て量の変更フローは以下のようになる．

(1) リソースを受け渡す側の仮想マシンに対して，受け渡す分のリソースを利用不可にする
(2) リソースを受け取る側の仮想マシンに対して，受け渡された分のリソースだけ利用可能にする

この手法は，一時的余剰リソースの他 VM の貸し出し [24,47] や，それを利用したスポットインスタンスの作成 [49] に応

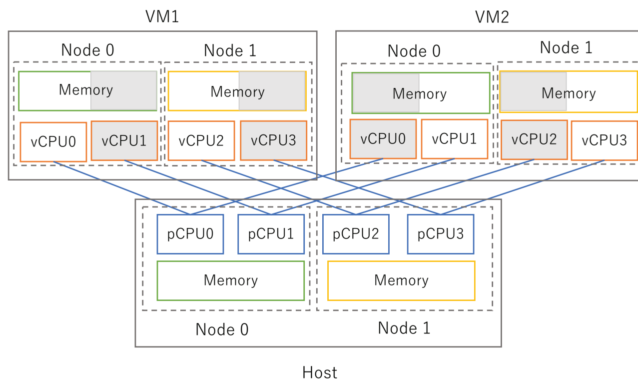


図 5　提案手法の概要 — Best viewed in color

用できる．

本手法実現の課題となるのは排他的リソース共有の実現方法である．以下にゲストと協調できる場合（プライベートクラウドなど）及びゲストと協調しない場合（パブリッククラウドなど）それぞれのケースで，QEMU/KVM 及びゲストとして Linux を対象とした場合の排他的リソース共有の実現方法を述べる．

5.2　ゲスト協調型

ゲストとホストが協調できる場合，オーバヘッドを最低限に抑えるためにゲストの方で利用できるリソースを制限する．Linux には各プロセスが利用できるリソースを制限するためのフレームワークとして cgroup [4] が存在する．cgroup ではプロセスごとに利用できる CPU やメモリ量などを制限することができる．この cgroup を利用して全プロセスに対して利用可能な CPU を指定することで，その仮想マシンは特定の CPU しか実質的に利用しなくなる．このことを応用し，ハイパーバイザからのリクエストに応じて全プロセスに対して cgroup を設定するエージェントプロセスを動作させておき，それを利用することで仮想マシンが利用できる実質的な CPU を制限する．

なお，Linux にはマイグレーション不可能なカーネルスレッド（per-CPU kthread）が複数存在するが，それらはその CPU にユーザスレッドが存在しない場合，何も仕事がなくなり動作しなくなる [24] ため，それらをマイグレーションさせる必要はない．もし vCPU 上で何もスケジューリングするものがなくなれば，その vCPU は自動的にアイドル状態となり HLT VMEXIT が発生する．Tickless Kernel の機能が有効になっていれば，その vCPU は他の CPU からの IPI (Rescheduling IPI 含む）あるいは他の外的割り込みによってのみ起動する．RCU の grace period は HLT 状態の CPU は考慮する必要がないので，RCU による IPI は発生しない．結果的に HLT した vCPU に対してハイパーバイザ側で割り込みを抑制すれば大部分で vCPU の停止状態は保たれる．

メモリに関しては，cgroup でもメモリ量の制限をかけることができるが，それはユーザプロセスに対してのみである．そこで，メモリのリソース制限に関しては virtio-mem [30] を利用する．virtio-mem はメモリデバイスに関する準仮想デバイスである．ゲストからは通常のメモリとして扱うことができるが，実際に利用可能な値をバルーニングのように変更することができる．そして既存のバルーニングドライバ（virtio-balloon）とは異なり NUMA 単位でのリソース制限にも対応している．virtio-mem は既に Linux のメインラインに導入されており，カーネルの変更なしに利用することができる．

5.3　ゲスト非協調型

ゲストとホストが協調できない場合は，ホストの方からリソース制限をかける．このリソース制限にも cgroup を利用することができる．

CPU の制限に関しては，ゲストで使用不可にしたい vCPU に対応するスレッドに関して cgroup によりスケジューラの割り当て CPU 時間（`cpu.cfs_quota.us`）を最小にする．これは当該スレッドを完全に停止するものではないが，その vCPU に関して最小限のスケジュールしかされなくなる．またメモリに関しても cgroup を利用し，仮想マシンのプロセス全体が利用できるメモリをノード毎に制限する．

この手法はもし仮にゲスト協調型であっても協調しない仮想マシンがあった場合に，強制的に制限をかけるためにも利用できる．

5.4　まとめ

本章では仮想 NUMA マシンの性能を維持しつつ弾力性を高めるための手法として排他的リソース共有を提案した．提案手法を利用することで，元の Full vNUMA 仮想マシンの利点を極力維持したまま，その VM の一部のリソースを利用して別の VM を実行することが可能となる．今後は提案手法について，リソースマイグレーション時間や排他的共有によるオーバヘッドなどを詳細に測定する予定である．

6.　議論

6.1　リソース割り当ての自動化

本研究では排他的共有において割り当てるリソースは管理者が設定するという形を取った．よりリソース効率性を高めるための手法として，リソースの使用率に応じて自動的に各 VM に割り当てる量を変化する方法がある．リソースの自動割り当てに関しては近年いくつかの研究が存在する [24, 47]．これらは本研究と直行する研究であり，これらの研究を応用することで全体のリソース使用効率をより高めることができると考えられる．

6.2 アプリケーションによる動的リソース変化の対応

動的に変化する CPU やメモリ量を効率的に利用するためには，アプリケーション自身がマシン上のリソース量変化を検知し，それらに応じて使用するリソースを増大あるいは減少させることが望ましい．しかしながら，アプリケーションごとにそのような機能を実装することは労力を要することであり，また既存のアプリケーションの書き換えも現実的ではない．従って，ライブラリあるいはランタイムレベルでのサポートが望まれる．

一つのアプローチとして，スレッドを管理して利用するランタイム（go [15] や Arachne [42] など）に対して，利用可能な CPU に応じたスレッド数の変化のサポートを追加することが考えられる．また VirtFlex [52] は，OpenMP ライブラリレベルで NUMA 構成の動的変化の対応をおこなっている．このような研究は我々が本論文で提案した手法を実際に利用する際に応用できると考えられる．

6.3 マイグレーションサポート

vNUMA の課題として，通常ゲストは NUMA 構成の動的変化に対応していないため，異なる NUMA アーキテクチャを持つマシンに vNUMA のインスタンスをライブマイグレーションした場合，ゲストとホストの NUMA 構成にずれが生じ vNUMA 本来の利点が生かせなくなるという問題が挙げられる．本論文ではこの問題は研究の対象外であるが，このような問題に対応するための手法の一つとして，XPV [20] ではゲスト OS に NUMA 構成の動的変化に対応する機構の追加を提案している．

IBM のプライベートクラウド利用に関する論文 [19] によれば，78%の VM はマイグレーションをおこなわない．また大多数のマイグレーションで，ホストのリソース量は大幅には変化しないとも述べている．これはクラウド環境では一括で同一のハードウェアを多量に購入してそれを利用することが多いためだと考えられる．従って，実際的には VM マイグレーションにより NUMA 構成のずれが生じる確率は低く，問題になることは少ない可能性がある．

6.4 他ハイパーバイザや OS への応用

本研究では Linux の QEMU/KVM を用いて実装や性能評価をおこなった．本論文で報告した性能低下要因は Linux のスケジューラの実装に起因するものであるが，他のハイパーバイザを実装する際にも注意すべき点である．また，排他的リソース共有に関しても他ハイパーバイザや OS の多くは Linux の cgroup 相当の機能 [5,8] を有している場合が多く，十分実装可能であると考えている．

6.5 I/O デバイスの対応

NUMA マシンでは，PCIe デバイスもノードごとに分かれている．すなわち，あるノードに接続した PCIe デバイスの DMA 速度は，ローカルなメモリに対しての方がリモートメモリに対するものよりも高速となる．IOctpus [46] では，DMA 先として常にローカルなメモリを選択することでデバイスの I/O 性能を高めている．vNUMA かつ PCIe デバイスをパススルー接続すれば，仮想マシン上でも [46] のような手法を利用できる．一方で，vNUMA 間での I/O デバイスのマイグレーション対応に関しては今後の課題の一つである．

7. 関連研究

7.1 仮想化環境における NUMA の活用

NUMA マシン上で仮想マシンを作成する場合に一番率直な方法は，ゲストには NUMA を作成しない方法である．この場合，NUMA 構成を生かすにはホスト側で vCPU のスケジューリングやゲストのメモリ配置を工夫する必要がある．Linux には定期的にパフォーマンスカウンタの値に応じてアプリケーションスレッドのマイグレーションあるいはそのアプリケーションが利用するデータのマイグレーションをおこなう Automatic Numa Balancing [29] という機能が存在する．また同様の機能を提供するデーモンとして numad [16] がある．これらの機能は仮想化用途に特化したものではなく，一般アプリケーションに向けたものであり，QEMU/KVM に対しても利用可能である．

この他，ハイパーバイザレベルで NUMA を考慮し VM の性能を向上させる研究には [43, 44, 50] がある．[43] は VM に割り当てたメモリが，バルーニングなどのメモリ操作後も変わらないようにする（リモートアクセスが増えないようにする）メモリ管理システムを提案した．[44,50] はパフォーマンスカウンタを用いて取得した値を考慮して，ローカルアクセスが多くなるような vCPU スケジューラを考案した．

このような方法はゲスト透過に利用でるという点でメリットがあるが，性能的な観点からは本論文でも示したように vNUMA よりも一般に劣ることになる [20,22,31,34,45]．

7.2 排他的な CPU 割り当てによる二重スケジューリング問題の回避

vCPU-BAL [47] は，利用可能な pCPU に応じて動的に割り当てる vCPU 数を変化させるという，vCPU ballooning を提案した．pCPU を一つの vCPU に排他的に割り当てることにより，ハイパーバイザによる vCPU のスケジューリングを除去することができ，結果的に二重スケジューリング問題を回避することができる．そして，ゲストの CPU 使用率に応じて動的に VM の vCPU 数を変化させることで，ある VM が利用していない pCPU を他の VM が利用することができるようになる．vCPU-BAL は Linux の hotplug/unplug の機能を用いて QEMU/KVM に対して実装された [38]．

仮想化環境であっても，Linux の CPU hotplug/unplug は数十ミリ秒程度かかる [24]．これは通常のスケジューリングのタイムスライス（1ms から 4ms 程度) や，軽量 VM の起動時間（数ミリ秒） [37] よりも遅い時間である．また CPU hotplug/unplug をおこなう際にはスピンロックを用いて全体の同期を取る処理が複数回必要である．従って，CPU hotplug/unplug は余った CPU 時間を瞬時的に他の VM が利用するのには低速であり，また他に動作しているアプリケーションの性能にも影響を与えうることになる．

vScale [24] は，vCPU ballooning を高速化するために軽量な vCPU 停止機構を提案した．vScale では Linux のカーネルスレッドとユーザスレッドについて深く調査し，マイグレーション可能なスレッドを他 CPU にマイグレーションし，その CPU に対する割り込みを抑制することで，CPU hotunplug せずとも CPU を数ミリ秒以内に実質的なオフライン状態にできることを示した．vScale ではこの手法を用いて VM が利用していない CPU の瞬時的な貸与を実現した．

我々が今回提案した手法は，動的かつ排他的に vCPU を割り当てるという点で vCPU ballooning と同じである．一方で我々の手法は vCPU を疑似的に停止するため，hotplug/unplug を利用する vCPU-BAL よりも高速に CPU の譲渡が完了する．加えて Linux に既に存在する機能を利用しており，vScale のような追加の変更は必要ない．また vCPU-BAL，vScale ともに NUMA については特に考慮されていない．我々の手法ではメモリに関しても動的な割り当てが可能である．

jailhouse [17] や ACRN [35] のような論理分割方式を採用したハイパーバイザも，仮想マシンに対して排他的に pCPU の割り当てをおこなう．しかし，そうしたハイパーバイザの割り当て設定は静的なものであり，動的な再構成やオーバコミットは通常サポートされていない．

7.3 ベアメタルクラウド

近年，複数のクラウドベンダが一つの物理マシン全体を貸し出す，いわゆるベアメタルクラウドのサービスを開始している [3,9,18]．ベアメタルクラウドを利用することで，仮想化のオーバヘッドを完全に除いた性能を得ることができる．ホストが NUMA マシンならば NUMA をそのまま扱うことが可能であり，当然二重スケジューリングの問題も存在しない．一方で，ベアメタルクラウドは通常のクラウドよりも高価であり，また仮想化が提供する弾力性や安全性が損なわれることになる．例えば，ベアメタルインスタンスのライブマイグレーションはハードウェアサポートが不十分であるために簡単ではない [27,32]．また，複数の研究がベアメタルクラウドのセキュリティ上の問題を指摘している [28,39]．あるクラウドベンダでは，セキュリティを担保するためにベアメタルクラウドに専用のハードウェアを利用している [2]．

我々の提案手法は専用のハードウェアを利用せずに，ベアメタルクラウドに近い性能を出しつつ，仮想化による保護と弾力性を提供するものである．また，我々の提案手法をベアメタルインスタンスに適用することも可能である．

8. 結論

NUMA マシンの利用も，高性能なクラウドの需要も今後の社会のシステムの発展とともにより一層増えてくるものと思われる．その際に重要になるのが，仮想 NUMA マシンの有効な活用方法である．本論文ではまず仮想 NUMA マシンの性能評価を実施し，Linux に存在するスケジューラの問題点を明らかにし，その修正案を提示した．そして，仮想 NUMA マシンで最大限の性能を引き出しつつ弾力性を確保する方法についての提案をおこなった．提案手法により，仮想 NUMA の利点を生かしつつ，状況に応じて柔軟に複数 VM で NUMA リソースを活用することが可能となる．今後はより大規模なベンチマークを利用して提案手法の評価を進める予定である．

参考文献

[1] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/ (Visted on 2021-05-06).
[2] AWS Nitro System. https://aws.amazon.com/ec2/nitro/ (Visted on 2020-7-11).
[3] Bare Metal Solution — Google Cloud. https://cloud.google.com/bare-metal (Visted on 2021-05-06).
[4] Cgroups - The Linux Kernel documentaion. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (Visted on 2021-05-06).
[5] CPUSET(2) FreeBSD System Calls Manual. https://www.freebsd.org/cgi/man.cgi?query=cpuset&sektion=2 (Visted on 2021-05-06).
[6] futex(2) — Linux manual page. https://man7.org/linux/man-pages/man2/futex.2.html (Visted on 2021-05-06).
[7] GOMP_SPINCOUNT (GNU libgomp). https://gcc.gnu.org/onlinedocs/libgomp/GOMP_005fSPINCOUNT.html (Visted on 2021-05-06).
[8] How to launch a process with CPU affinity set - Debugging Internet. https://blogs.msdn.microsoft.com/santhoshonline/2011/11/24/how-to-launch-a-process-with-cpu-affinity-set/ (Visted on 2021-05-06).
[9] IBM Cloud Bare Metal Servers. https://www.ibm.com/cloud/bare-metal-servers (Visted on 2021-05-06).
[10] libvirt: The virtualization API. https://libvirt.org/ (Visted on 2021-05-06).
[11] NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html (Visted on 2021-05-06).
[12] numactl/numactl: NUMA support for Linux. https://github.com/numactl/numactl (Visted on 2021-05-06).
[13] OMP_WAIT_POLICY. https://www.openmp.org/spec-html/5.1/openmpse64.html (Visted on 2021-05-06).
[14] Scheduler Domains - The Linux Kernel documentaion. https://www.kernel.org/doc/Documentation/

scheduler/sched-domains.txt (Visted on 2021-05-06).
[15] The Go Programming Language. https://golang.org/ (Visted on 2021-05-06).
[16] Overview - numad - Pagure.io, 2021. https://pagure.io/numad (Visted on 2021-05-06).
[17] siemens/jailhouse: Linux-based partitioning hypervisor, 2021. https://github.com/siemens/jailhouse (Visted on 2021-05-06).
[18] J. Barr. Amazon EC2 Bare Metal Instances with Direct Access to Hardware, 2017. https://aws.amazon.com/blogs/aws/new-amazon-ec2-bare-metal-instances-with-direct-access-to-hardware/ (Visted on 2021-05-06).
[19] R. Birke, et al. Virtualization in the Private Cloud: State of the Practice. *IEEE Transactions on Network and Service Management*, Vol. 13, No. 3, pp. 608–621, 2016.
[20] B. Bui, et al. When eXtended Para - Virtualization (XPV) Meets NUMA. In *EuroSys'19*. ACM, 2019.
[21] R. Buyya, et al. A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade. *ACM Computing Surveys*, Vol. 51, No. 5, pp. 105:1–105:38, 2018.
[22] N. Chakthranont, et al. Exploring the Performance Impact of Virtualization on an HPC Cloud. In *CloudCom'14*. IEEE, 2014.
[23] P. M. Chen, et al. When Virtual is Better Than Real. In *HotOS'01*. IEEE, 2001.
[24] L. Cheng, et al. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Eurosys'16*. ACM, 2016.
[25] X. Ding, et al. Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications. In *ATC'14*. USENIX, 2014.
[26] T. Friebel, et al. How to Deal with Lock Holder Preemption. In *Xen Summit North America*, 2008.
[27] T. Fukai, et al. OS-Independent Live Migration Scheme for Bare-Metal Clouds. In *UCC'15*. IEEE, 2015.
[28] T. Fukai, et al. BMCArmor: A Hardware Protection Scheme for Bare-Metal Clouds. In *CloudCom'17*, 2017.
[29] M. Gorman. The CPU Scheduler in on for Automatic NUMA Balancing, 2012. https://lwn.net/Articles/523065/ (Visted on 2021-05-06).
[30] D. Hildenbrand, et al. virtio-mem: Paravirtualized Memory Hot(un)plug. In *VEE'21*. ACM, 2021.
[31] K. Z. Ibrahim, et al. Characterizing the Performance of Parallel Applications on Multi-socket Virtual Machines. In *SoCC'11*. IEEE, 2011.
[32] J. Im, et al. On-Demand Virtualization for Live Migration in Bare Metal Cloud. In *SoCC'17*. ACM, 2017.
[33] S. Kashyap, et al. Scaling Guest OS Critical Sections with eCS. In *ATC'18*. USENIX, 2018.
[34] A. Kudryavtsev, et al. Modern HPC Cluster Virtualization Using KVM and Palacios. In *HiPC'12*. IEEE, 2012.
[35] Hao L., et al. ACRN: A Big Little Hypervisor for IoT Development. In *VEE'19*. ACM, 2019.
[36] J. Lozi, et al. The Linux Scheduler: a Decade of Wasted Cores. In *EuroSys'16*. ACM, 2016.
[37] F. Manco, et al. My VM is Lighter (and Safer) Than Your Container. In *SOSP'17*. ACM, 2017.
[38] T. Miao, et al. FlexCore: Dynamic virtual machine scheduling using VCPU ballooning. *Tsinghua Science and Technology*, Vol. 20, No. 1, pp. 7–16, 2015.
[39] A. Mosayyebzadeh, et al. Supporting Security Sensitive Tenants in a Bare-Metal Cloud. In *ATC'19*. USENIX, 2019.
[40] M. A. S. Netto, et al. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *ACM Computing Surveys*, Vol. 51, No. 1, pp. 8:1–8:29, 2018.
[41] A. Prasad, et al. The RCU-Reader Preemption Problem in VMs. In *ATC'17*. USENIX, 2017.
[42] H. Qin, et al. Arachne: Core-Aware Thread Management. In *OSDI'18*. USENIX, 2018.
[43] D. S. Rao, et al. vNUMA-mgr: Managing VM Memory on NUMA Platforms. In *HiPC'10*, pp. 1–10. IEEE, 2010.
[44] J. Rao, et al. Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In *HPCA'13*. IEEE, 2013.
[45] S. A. R. Shah, et al. Improve Performance and Throughput of VMs for Scientific Workloads in a Cloud Environment. In *PlatCon'16*. IEEE, 2016.
[46] I. Smolyar, et al. IOctopus: Outsmarting Nonuniform DMA. In *ASPLOS'20*. ACM, 2020.
[47] X. Song, et al. Schedule Processes, Not VCPUs. In *APSys'13*. ACM, 2013.
[48] V. Uhlig, et al. Towards Scalable Multiprocessor Virtual Machines. In *VM'04*. USENIX, 2004.
[49] Y. Wang, et al. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *EuroSys'21*, New York, NY, USA, 2021. ACM.
[50] S. Wu, et al. vProbe: Scheduling Virtual Machines on NUMA Systems. In *Cluster'16*. IEEE, 2016.
[51] C. Xu, et al. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In *ATC'13*. USENIX, 2013.
[52] R. Zhang, et al. Virtflex: Automatic Adaptation to NUMA Topology Change for OpenMP Applications. In *IWOMP'20*. Springer, 2020.

ⓒ 2021 Information Processing Society of Japan