Vol.2014-OS-128 No.2
2014/3/6

仮想化環境におけるメモリ情報に着目したページシェアリング効率化手法

井上宗士[1,a)]　山田浩史[2,3]　河野健二[1,3]

概要：近年，仮想化環境における PaaS (Platform as a Service) が多く提供されている．各ユーザに仮想マシン (VM) 単位で開発・実行環境を提供しており，複数の VM を一台の物理マシンに集約することで稼働する物理マシン台数を減らすことができる．同じ OS・ミドルウェアが VM の台数分重複して稼働する分，冗長なメモリが存在する．ページシェアリングを用いて内容の同じページを共有することにより，冗長なメモリの除去が可能である．しかし，既存のページシェアリング手法では，共有可能なページの検出に時間がかかり，ワークロードの変化に追従できないという問題が存在する．本研究では，ゲスト OS から送るページの情報をヒントとすることにより，共有可能なページを既存手法と同等のオーバーヘッドで迅速に検出するページシェアリング手法を提案する．本手法の有用性を示すために，ベンチマーク実行中にページシェアリングを実行して共有できたページ数と CPU 使用率を計測したところ，提案手法は既存手法と同等のオーバーヘッドで最大 6 倍のページを共有することを確認した．

キーワード：仮想化，ページシェアリング，PaaS

1. はじめに

近年，仮想化技術は広く用いられるようになっており，その利用例のひとつとして，仮想化環境での Platform as a Service (PaaS) が多く提供されるようになってきている．仮想化とは，仮想マシンモニタ (VMM) 上に複数の仮想マシン (VM) を物理マシンに集約することで，稼働する物理マシンの台数を削減できる技術である．しかし，仮想化による PaaS では，開発・実行環境を各ユーザに VM 単位で提供する．このとき，同じ OS やミドルウェアが VM の台数分重複して稼働しているため，冗長なメモリが存在し，集約率増加のボトルネックとなっている．また，過去の調査により，集約率向上の主なボトルネックとなる物理資源はメモリとされており，一般的な実行環境における使用率は 40% 程度とされている [1].

仮想化環境における冗長なメモリを除去するための仕組みとして，ページシェアリング機構が VMM に導入されている．ページシェアリングでは，メモリスキャナを用いて VM のメモリを定期的に探索し，内容の同じメモリページを共有することで冗長なメモリを削減することができる．本機構を用いることにより，VM の集約率を向上することが可能となる．

既存のページシェアリングの手法では，共有可能なページの探索に要する時間が長くなってしまうという欠点が存在する．例として，VMware ESX Server のページシェアリングでは，メモリスキャナが全ページを探索する回数は最大で 10 分に 1 度となっている．このため，既存のページシェアリングではワークロードの変化に迅速に追従できず，頻繁に書き換えられるページを共有することができない．

Kernel-based Virtual Machine (KVM) [2] における Kernel Samepage Merging (KSM) [3] では，探索速度を手動で設定できるが，探索速度を上げると CPU 使用率が増加してしまうため，VM のパフォーマンスに影響が出てしまうという問題がある．Satori [4] のように，ディスク I/O を監視し，ページが書き変わった瞬間に共有可能かを調べる手法も存在するが，ブロック番号を利用していることから，各 VM が同一のディスクイメージ上で稼働している必要がある．そのため，本提案手法が想定している PaaS のような，複数のユーザに VM を提供する環境には適さない．

本研究は，各ゲスト OS が持つページの情報をヒントと

[1] 慶応義塾大学
Keio Uniersity
[2] 東京農工大学
Tokyo university of Agriculture and Technology
[3] JST CREST
[a)] shutoi@sslab.ics.keio.ac.jp

© 2014 Information Processing Society of Japan

して探索対象を限定することで，少ないオーバーヘッドでより多くのページを共有するページシェアリング手法を提案する．提案手法では，バッファキャッシュやテキスト領域に利用されるページが多く共有可能であることに着目し，これらのページを共有対象とする．特に，同じパス名のファイルの同一オフセット上に存在するページは内容が一致する可能性が高い．各ゲスト OS のバッファキャッシュのページが更新されるたびにページの情報を取得し，ホスト OS に送信する．ホスト OS はこれらの情報をもとに共有できる可能性の高いページを判断し，探索対象とする．

提案手法をオープンソースの仮想マシンモニタ KVM に実装する．ゲスト OS・ホスト OS の双方に改変を加える．ホスト OS 側では，KSM のメモリスキャナに改変を加える形で提案手法を実装する．

本手法の有用性を示すために，ベンチマーク稼働中にページシェアリングを実行し，共有したページ数とメモリスキャナの CPU 使用率を取得し，KSM と比較した．その結果，提案手法は KSM よりも少ないオーバーヘッドで，最大 6 倍以上のページを共有することができた．

本論文の構成を以下に示す．2 章ではページシェアリングの仕組みについて説明し．ページの用途毎の共有率について分析する．3 章では提案手法について説明する．4 章では提案手法の実装について説明する．5 章ではベンチマーク下における提案手法の有用性の調査とその結果について説明する．6 章でページシェアリングに対する関連研究を紹介する．7 章でまとめと今後の課題を述べる．

2. 背景

2.1 ページシェアリング

ページシェアリングを用いることで，内容の重複するページを除去することができ，より多くの VM を集約することができる．VMware ESX Server に導入されている Content-based page sharing [5] や KVM に導入されている KSM [3] では，VMM に実装されたメモリスキャナを用いてページシェアリングを行う．メモリスキャナは各 VM のメモリを探索し，内容の一致するページが存在するかを調べる．ページの内容のみに着目して共有可能かを決定することから，ページの用途や更新のタイミングに関わらず，共有可能なページを検出することが可能となる．

しかし，メモリスキャナを用いた既存のページシェアリングでは全ページを探索するため，共有可能なページを検出するまでに時間がかかってしまうという問題が存在する．実際に VMware ESX Server では 10 分に 1 度の探索が限界となっており，ワークロードの実行により書き変わるページを迅速に共有できない．そのため，同じ内容のページをすべて共有することができず，集約率が向上しない．Miller [6] が示すように，今回用いたカーネルビルドでは，共有可能なページの 80% は 5 分以内に更新されて

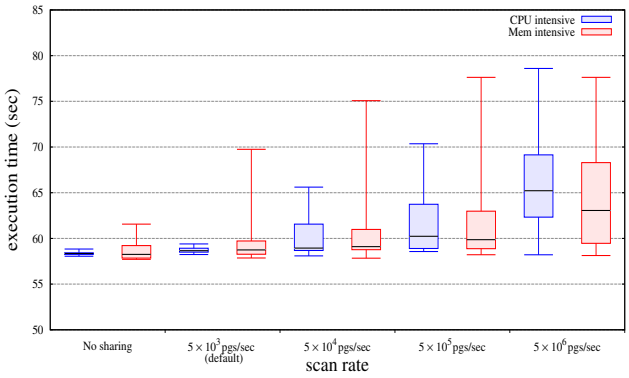


図 1 KSM の探索速度毎に応じた VM のパフォーマンスへの影響

いることから，ページを迅速に共有することは集約率の向上のために必要である．

KSM では，手動で探索速度を設定することが可能となっているが，探索速度を上げると VM のパフォーマンスに影響を与えてしまうことがわかっている．そのため，デフォルトで設定された速度よりも探索速度を高くすることは推奨されていない．なお，本研究における探索速度とは，単位時間あたりにメモリスキャナが探索するページの数を示す．KSM のメモリスキャナは初期状態では秒間 5000 ページ探索するように設定されている．VM のメモリを 1GiB と設定した場合，1 度の探索に 1 分程度の時間を要することになる．VM の台数や使用メモリ量の増加に応じて，探索時間は増加する．

KSM で探索速度を上げた場合に VM のパフォーマンスに与える影響を調査するための予備実験を行う．実験方法として，KSM の実行中に 4 台の同設定の VM 上で同じベンチマークを稼働させ，その実行時間を計測した．環境として，クアッドコアの 2.80GHz Intel Xeon X5560 CPU を一個搭載した物理マシン上で，各 VM に仮想 CPU を一つずつ割り当てている．また，各 VM のメモリとして 512 MiB 割り当てている．ベンチマークとしては，Wood らの実験で用いたもの [7] を利用したページに対する書き込みを一定回数繰り返すメモリインテンシブなマイクロベンチマーク，決められた計算を一定回数繰り返し続ける CPU インテンシブなマイクロベンチマークを作成した．

図 1 に示すように，KSM の探索速度を上げることにより，VM のパフォーマンスに影響を与えることが確認できる．左端のデータが KSM を実行しなかった場合の実行時間であり，右端が最も探索速度を上げた場合のデータを示す．グラフから，CPU・メモリインテンシブに関わらず最大で実行時間が 35% 長くなっていることがわかる．

Satori ではゲスト OS に対する I/O を監視し，ページが書き換えられると同時に共有を行うことでワークロードの変化に対する追従を可能としているが，こちらは PaaS 環境での仕様を想定されていない [4]．ブロック番号を用いてページを共有することから，各 VM が同じ仮想ディス

© 2014 Information Processing Society of Japan

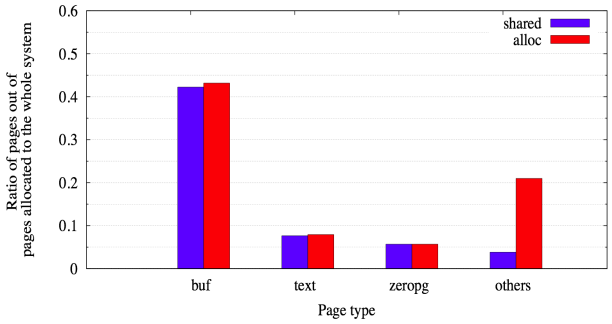


図 2 ブート直後の VM 2 台の間で共有可能なページの内訳

クイメージから起動している必要がある．そのため，ユーザが複数存在する環境での利用には適さない．

2.2 共有可能なページの種類

本研究では，テキスト領域やバッファキャッシュに利用されるページは多く共有可能であることから，これらのページのみを探索対象としている．Barker らの研究により，ワークロード実行中の共有ライブラリのページはページシェアリングにより共有されるページの 43% を占めることが報告されている [8]．また，Kloster らの研究では，ページキャッシュにおける共有可能なページは 64% から 94% 存在することが報告されている [9]．

また，ブート直後の VM 2 台の間で共有できるページの内訳を調べることで，バッファキャッシュ・テキスト領域・ゼロページに利用されるページの多くが共有可能であることを検証した．図 2 に調査の結果を示す．赤い棒グラフがその用途に割り当てられたページ数を示しており，青い棒グラフが共有したページ数を表している．バッファキャッシュ・テキスト領域・ゼロページは 95% 以上が共有可能であることがわかる．反して，その他のページ (ヒープ・スタック等) はおよそ 20% が共有可能であることがわかる．ブート直後の同設定の VM 2 台における内訳であることから，リアルワークロードが稼働している状況下ではこれ以上に共有できるページは減少すると考えられる．

.

3. 提案手法

3.1 概要

本研究では，ゲスト OS から与えられるヒント情報を活用することで探索対象を限定するページシェアリングの手法を提案する．既存手法の KSM では，すべてのページを探索するため，実行時のオーバーヘッドが大きくなる．提案手法を用いることで，探索するページ数を減らし，より少ないオーバーヘッドで多くのページを共有することが可能となる．

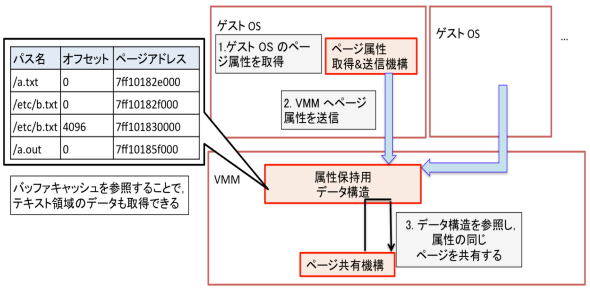


図 3 提案手法のアプローチ

3.2 アプローチ

PaaS 環境を想定をしていることから，同一のファイルやアプリケーションが多く存在すると考えられる．また，2.2 章の結果から，バッファキャッシュやテキスト領域，ゼロページが探索対象として妥当であることが検証されている．そのため，これらのページの情報をゲスト OS からハイパーバイザに送信する．また，同一パス名のファイルの同一オフセット上にあるデータは同じになる可能性が高いことから，これらの情報を用いる．本研究では，この情報をページ属性と名付ける．

提案手法を実現するために，VMM による仮想化環境を用いる．PaaS 環境を想定するため，ゲスト OS への改変が可能である．Intel VT による完全仮想化環境であるが，Intel VT の仮想化支援機能 [10] を用いることで VM exit によるページ属性の受け渡しが利用できる．

提案手法におけるページ属性送信の流れを図 3 に示す．ゲスト OS 側でページ属性を取得し，VMM へ送信する．VMM では受け取ったページ属性を保持しておき，ゲスト OS から新しい属性を渡されると同じ属性のページが存在するかを確認する．属性のマッチするページが存在すれば，そのページに関する情報を探索用のデータ構造に保持しておく．メモリスキャナはデータ構造を参照し，同じ属性のページが存在するページのみを探索する．

4. 実装

提案手法を，オープンソースの仮想マシンモニタ KVM に実装した．ホスト OS・ゲスト OS の双方に，Linux 3.0.4 を使用しており，またどちらにも改変を加える．ホスト OS に関しては KSM に用いるデータ構造やメモリスキャナへの改変も行う．更に，ページ属性と VM を対応付けるようにするため，qemu [11] に対しても修正を加えている．

実装イメージを図 4 に示す．パス名からファイルのバッファキャッシュにアクセスを行えるようにするため，ファイルオープン時に inode とパス名を紐付ける．バッファキャッシュ上のページが更新されるたびにバッファキャッシュから対応したパス名，オフセットをページ属性として取得する．ページ属性はハイパーバーザに VM exit による疑似的なハイパーコールでデータを受け渡し，データ構造

© 2014 Information Processing Society of Japan

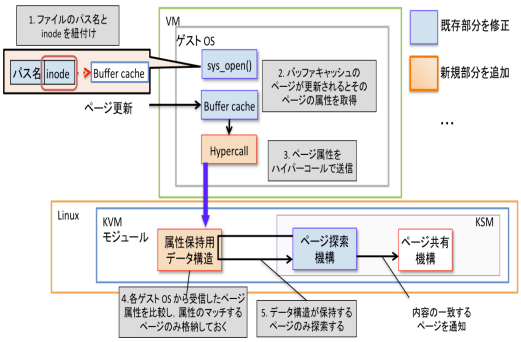


図 4 実装図

に属性を保持する．同じ属性の存在するページのみがメモリスキャナに受け渡され，探索される．共有可能なページが検出された場合，メモリスキャナは従来の KSM のページ共有機構にページを渡し，共有する．

4.1 ページヒントの保持

KSM のメモリスキャナは，探索実行時に属性とは別のページの情報を必要とする．本研究ではこの情報をページヒントと名付ける．属性のマッチするページが存在すると，それぞれのページのヒントが生成され，データ構造に保持される．提案手法では，メモリスキャナはデータ構造が保持しているページヒントから探索対象を決定する．これにより，ページが更新されるとすぐにそのページを探索する．

ページヒント保持用データ構造として，Miller らの XLH [6] にて導入されている bounded circular hint-stack を用いる．ヒントスタックでは，直近で生成された一定数のページヒントを保持する．彼らは，当初円形のキュー構造を用いてヒントを保持していたが，変化の早いワークロードではメモリスキャナがヒント生成の頻度に追いつくことができない．前述した通り，KSM のメモリスキャナはデフォルトで 5000 ページ / 秒の探索を行うよう設定されている．ワークロードによってはヒント生成の速度はメモリスキャナの探索速度よりも高くなるため，新しく更新されるページが共有できなくなってしまう可能性がある．

ヒントスタックを用いることで，最も新しく更新されたページを優先的に探索することができる．設定されたスタックサイズよりも多くのヒントが生成されている場合，古いヒントから順に除去される．これにより，変化の多いワークロードに追従してヒントをメモリスキャナに渡すことができる．

4.2 KSM のデータ構造への改変

KSM では，二種類の木構造を用いてページの探索を行っている．メモリスキャナは，探索対象のページについて，この二つの木に共有可能なページが存在するかを探索する．Stable tree には現時点で共有されているページ，unstable tree では共有されていないページを登録している．どちらの木も，ページの内容をキーとしている．

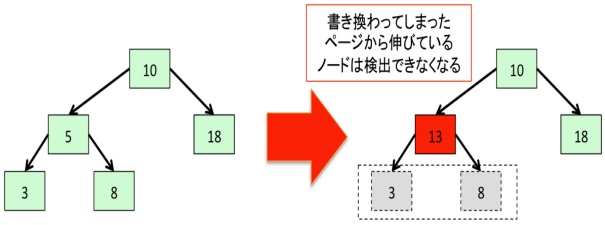


図 5 unstable tree の更新

探索対象のページについて，メモリスキャナはまず stable tree を探索する．共有可能なページが存在する場合は stable tree にそのページを追加する．Stable tree に共有可能なページが存在しない場合は，次にそのページが前回から更新されていないかを，チェックサムを用いて検証する．チェックサムは，ページの内容をハッシュ化したものを用いる．更新されていない場合のみ，unstable tree を探索し，共有可能なページを探索する．共有可能なページが存在すればそれらのページを unstable tree から削除し，stable tree に追加する．存在しない場合は，unstable tree に追加する．

Unstable tree では本提案手法で共有できるページ数を減少させてしまうという問題が存在する．Stable tree と違い，unstable tree では登録されているページの更新を監視しないため，各ノードのキーが書き変わってしまっても木構造の更新ができない．そのため，書き変わってしまったものより下のノードは図 5 のように探索が不可能になってしまい，結果的に共有できるページが減ってしまう，KSM では，全ページを探索後に unstable tree を削除，再構築しているが，提案手法では全てのページを探索する前に同じページを複数回探索する可能性が高く，より共有できるページが減ってしまう．また，Miller によると KSM においても unstable tree 内のノードは最大で 70% が探索不可能になるとされている．

提案手法では XLH と同じく，unstable tree を用いず，ハッシュテーブルにより未共有ページを管理する．これにより，ページの更新が他のノードの探索に影響しないため，共有できるページが減少することがなくなる．ハッシュ値には，KSM で利用していたチェックサムを用いる．

5. 実験

5.1 目的

提案手法が KSM よりも少ないオーバーヘッドで多くのページを共有できることを示すために，ベンチマークの稼働中にページシェアリングを実行し，ページ共有率と CPU 使用率を提案手法と KSM で比較した．ページ共有率は，システム全体のページ数に対して共有できたページ数の比率とする．CPU 使用率は，top コマンドを用いて取得す

© 2014 Information Processing Society of Japan

る．また，実験は三つの探索速度について行った．それぞれ，秒間 5000 ページ（デフォルト），秒間 1000 ページ，秒間 500 ページとなっている．これに対応し，KSM が全ページについて探索を 1 度行うまでに要する時間はそれぞれ 44 秒，220 秒，440 秒となる．

実験環境として，クアッドコアの 2.80GHz Intel Xeon X5560 CPU を一個搭載した物理マシン上に 2 台の VM を稼働させた．各 VM に仮想 CPU を一個，メモリを 512 MiB 割り当てている．ベンチマークとしてカーネルのビルドを実行する．初期状態として，各 VM がブート完了した状態で一旦ページシェアリングを行い，その時点で共有可能なページを全て共有しておく．

5.2 ページ共有率

実験で計測したページ共有率のデータを図 6 に示す．青線が KSM，赤線が提案手法の共有率を示す．縦軸が共有率，横軸が実行時間である．どちらのグラフについても，提案手法は既存手法よりも多くのページをほぼコンスタントに共有し続けており，デフォルト時で (図 6(a)) 最大 2 倍，秒間 1000 ページに設定した場合でも (図 6(b)) 最大 6 倍のページを共有している．

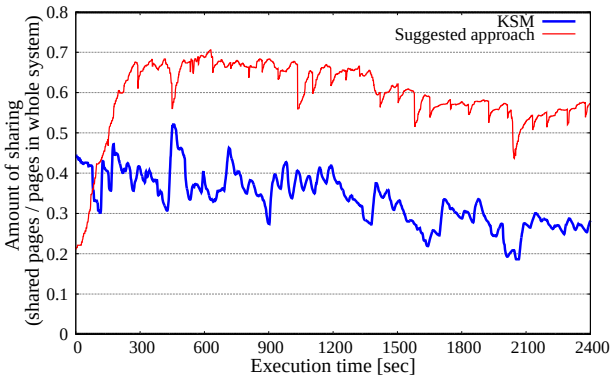
どちらのグラフについても，初期状態では KSM が提案手法よりも多くページを共有している．これは，提案手法がヒープやスタックなどを探索対象としていないためである．2.2 節にて説明したように，ブート直後の VM 二台間では，バッファキャッシュやテキスト領域ほどではないが，ヒープ領域やスタック領域も共有可能となる．KSM では全てのページを探索するため，実験開始時では提案手法が探索を行わないこれらのページを共有している．

また図 6(b) では，KSM がワークロードの変化に追従できていないことがわかる．初期状態から一度大きく低下した後，共有率がほとんど増加することがないことから，ベンチマーク実行中に更新されたページが共有できていないことがわかる．加えて，それ以降大きな減少もないということから，ここで共有され続けているページはベンチマークの実行によるものではなく，実験開始前に共有していたページと考えられる．図 6(c) にて，提案手法のページ共有率が大きく低下しているにも関わらず，KSM では変化がないことからこれらの共有率はベンチマークの実行によるものではないということがわかる．

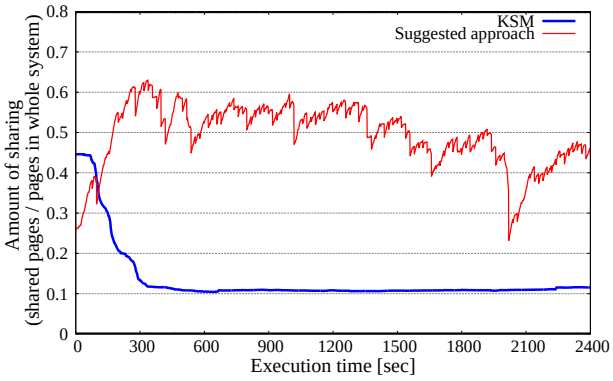
5.3 CPU 使用率

提案手法と KSM メモリスキャナの CPU 使用率の平均値を表 1 に示す．デフォルトでは，提案手法は KSM よりも CPU 使用率が低くなっているが，その他の設定では，KSM が提案手法よりも低い CPU 使用率を示している．

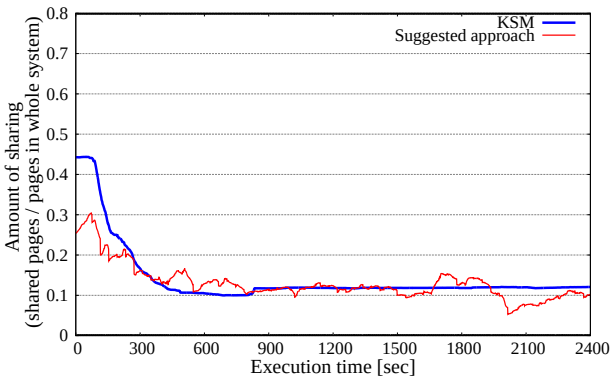
デフォルトにおいて，提案手法が KSM よりも少ないオーバーヘッドでページシェアリングを行うことができる



(a) 5000 ページ / 秒 (デフォルト)



(b) 1000 ページ / 秒



(c) 1000 ページ / 秒

図 6 探索時間毎のページ共有率

Approach	5000pages/sec (default)	1000pages/sec	500pages/sec
KSM	3.49%	0.65%	0.274%
提案手法	1.72%	1.15%	0.54%

表 1 CPU 使用率

理由は，KSM よりも探索を行う総回数が少ないためである．KSM が全てのページを探索するのに比べ，提案手法は更新があったページの中で同じページ属性が存在するもののみを探索する．提案手法では，ページヒントがスタックに格納されていない間は動作を停止している．そのため，現時点で共有可能なページが全て共有された場合は，新しくページヒントが生成されるまでオーバーヘッドが発生しなくなる．対して，KSM ではページの更新の有無に関わらず探索を続けるため，コンスタントにオーバーヘッ

© 2014 Information Processing Society of Japan

ドが発生している．

残り 2 つの設定について KSM が提案手法よりも CPU 使用率が低くなっている理由は，KSM が実験中に新しくページを共有していないためである．4.2 節にて説明したように，KSM では共有可能なページを探索する際にそのページが変化していないかをチェックサムによって確認し，変化している場合は unstable tree の探索を行わないようにしている．この探索速度では，KSM のメモリスキャナは unstable tree の探索を行わないため，結果的に探索回数が少なくなっている．そのため，書き換えられているページを優先的に探索する提案手法と比べて CPU 使用率が低くなっている．しかし，KSM はワークロードによって新しく書き換えられたページを共有できておらず，提案手法よりも低い共有率を示していることから，提案手法よりも不必要なオーバーヘッドが発生しやすいと言える．

探索速度を上げることで，提案手法は迅速にページを共有するため，結果的にメモリスキャナの CPU 使用率の平均値は大きく増加することがない．一時的に KSM よりも多くのページを探索することになるため，瞬間的な CPU 使用率は KSM より高くなるが，探索を行っている時間を短くするため，動作を停止できる時間が長くなる．KSM は提案手法のように瞬間的に高い CPU 使用率を示すことはないが，ワークロードの状況に関わらず探索を続けるため，オーバーヘッドが絶えず発生する．そのため，探索速度を上げると 2.1 節にて説明したように，VM のパフォーマンスに影響が出てしまう．

6. 関連研究

仮想化環境におけるページシェアリングの手法は数多く提案されている．しかし，これらの手法は PaaS の環境において，ワークロードの変化に迅速に追従し，VM の集約率を大きく向上することができない．提案手法は，ページの持つ情報をヒントとすることで少ないオーバーヘッドでページを迅速に共有し，より多くの VM の集約を可能にする．

メモリスキャナを用いて各 VM のメモリを探索し，ページの内容を比較して共有する content-based なページシェアリングを提案する研究は多くなされている．Waldspurger らは，VMware ESX における content-based page sharing を提案している [5]．各ページの内容のハッシュ値を保持しておき，ハッシュ値が一致するページ同士のみを比較するようにすることで，探索により発生するオーバーヘッドを抑えている．また，Difference Engine [1] では，通常のページシェアリングに加え，内容が完全一致しないページを共有する．ページ同士の一致する部分のみを共有し，異なる部分はパッチを作成することで補っている．また，使用頻度の低いページを検出し，圧縮することでメモリの使用率を減らしている [12]．Memory Buddies では，ページシェアリングにより最も多くページを共有できるようデータセンタ内で VM を再配置する [13]．メモリスキャナによりメモリを探索し，各ページのフィンガープリントを生成する．一致するフィンガープリントの数に応じて，VM の配置先を決定する．KSM では，ハッシュ値を利用したページの探索を行っているが，その利用法は VMware ESX とは異なる [3]．KSM では，頻繁に変化するページを探索しないようにするため，各ページ内容のハッシュ値を算出し，過去のハッシュ値と一致しない場合は探索しない．また，変化の早いワークロードに対応するために探索速度を手動で変更することが可能となっている．これらのページシェアリングの手法は，ページの用途やいつ更新されたかなどといった情報に依存せず，内容が一致しているかどうかのみを基準として共有するため，静的ワークロード下など，ページ内容が更新されにくい環境で多くのページを共有できる．しかし，全てのページを探索していることから，共有可能なページを検出するまでに時間がかかってしまい，共有前に書き換えられてしまう可能性が高い．KSM では探索速度を上げることが可能であるが，VM のパフォーマンスに影響を与えてしまうという問題が存在する．

本研究と同様にページの更新を監視し，更新されたページを優先的に共有するページシェアリングの手法も提案されている．Xen に導入された [14] Satori では，ゲスト OS に sharing-aware block device を追加し，I/O を監視することで共有可能なページを迅速に共有している [4]．しかし，ページの共有にブロック番号を利用していることから，共通する仮想ディスクイメージで稼働する VM を対象としている．したがって，提案手法が想定している PaaS のようにユーザが複数存在する環境での使用には適さない．KVM に導入された XLH では，VM の仮想ディスクイメージに対する read/write を監視している．このときに更新されたページのページヒントを生成し，KSM のメモリスキャナがこれを参照することで迅速なページ更新を可能にする．しかし，ゲスト OS に改変を加えていないことから，各ページの用途や対応するファイルのパス名と言った詳細な情報が取得できないため，提案手法と比較して原理的に共有できるページが少なくなると考えられる．また，更新されたページを全てヒントスタックに格納するため，頻繁にページが更新されるワークロードにおいては古いヒントがスタックから除去されてしまい，メモリスキャナが全てのヒントを参照できなくなる可能性が高い．

7. まとめ

本研究では，ページが持つ情報をヒントとすることで探索範囲を限定し，既存手法よりも小さいオーバーヘッドで多くのページを共有するページシェアリング手法を提案した．提案手法では，各ゲスト OS から更新されたページの情報をハイパーバイザに受け渡し，その情報をもとに探索

© 2014 Information Processing Society of Japan

対象を限定する．また，バッファキャッシュやテキスト領域に利用されるページが多く共有可能であることから，これらのページのみを共有するようにした．

KVM の，双方 linux 3.0.4 で動作するホスト OS，ゲスト OS に提案手法を実装した．ゲスト OS にバッファキャッシュの更新を監視してページ属性を生成，送信する機構を実装した．ホスト OS では KSM のメモリスキャナやデータ構造に改変を加えた．属性の一致するページのヒントをヒントスタックに保持し，メモリスキャナはページヒントを参照して探索を行う．

提案手法を用いることで，既存手法である KSM よりも少ないオーバーヘッドで最大 6 倍のページを共有することができた．提案手法は KSM よりも迅速に共有可能なページを検出するため，KSM では検出できないような頻繁に書き変わってしまうページを共有している．

提案手法が影響を受ける環境は大きく分けて 4 つに分けられる．パス名が同一であり共有可能であるファイルが存在する場合，パス名が異なり共有も不可能なファイルが存在する場合，パス名が同一であるが共有が不可能なファイルが存在する場合，パス名が異なるが共有が可能なファイルが存在する場合である．提案手法は最初の 2 パターンについては適切な対処が可能である．3 つ目に関しては探索が行われるが，共有はされない．最後のパターンについては，提案手法は対処できない．そのため，このようなファイルが多く存在する場合は提案手法は大きく効率が低下してしまう．今後は，これら 4 つのパターンを加味した実験を行い，既存手法との比較を行う必要がある．

また，本提案手法はファイル形式に依存した仕様などにも対処することが可能となる．例えば，microsoft office ドキュメントのように，全てのファイルの間でヘッダが共有可能である場合などである．パス名をページ属性としているため，これを参照して特殊なファイルを判別し，探索対象への追加や除去が可能である．加えて，KSM と切り替えて利用することで，提案手法では探索しないページを共有しておくことも可能である．

参考文献

[1] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M. and Vahdat, A.: Difference Engine: Harnessing Memory Redundancy in Virtual Machines, *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 309–322 (2008).

[2] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, *Proceedings of the Linux Symposium*, pp. 225 – 230 (2007).

[3] Arcangeli, A., Eidus, I. and Wright, C.: Increasing memory density by using KSM, *2009 Linux Symposium*, pp. 19–28 (2009).

[4] Milos, G., Murray, D. G., Hand, S. and Fetterman, M. A.: Satori: Enlightened page sharing, *Proceedings of 2009 USENIX Annual Technical Conference (ATC '09)*, pp. 1–14 (2009).

[5] Waldspurger., C. A.: Memory resource management in VMware ESX server, *Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI '02)*, pp. 181–194 (2002).

[6] Miller, K., Franz, F., Rittinghaus, M., Hillenbrand, M. and Bellosa, F.: XLH: More effective memory deduplication scanners through cross-layer hints, *Proceedings of 2013 USENIX Annual Technical Conference (ATC '13)*, pp. 279–290 (2013).

[7] Brown, A. D. and Mowry, T. C.: Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently, *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation (OSDI '00)*, pp. 31–44 (2000).

[8] Barker, S., Wood, T., Shenoy, P. and Sitaraman, R.: An Empirical Study of Memory Sharing in Virtual Machines, *Proceedings of 2012 USENIX Annual Technical Conference (ATC '12)*, pp. 273 – 284 (2012).

[9] Jacob Faber Kloster, Jesper Kristensen, A. M.: Determining the use of Interdomain Shareable Pages using Kernel Introspection , Technical report, Aalborg University (2007).

[10] 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, No. 3, pp. 167–178 (2006).

[11] Bellard, F.: Qemu, a fast and portable dynamic translator, *Proceedings of the USENIX Annual Technical Conference, ATEC '05*, pp. 41 – 46 (2005).

[12] Douglis, F.: The compression cache: Using on-line compression to extend physical memory, *Proceedings of the USENIX Winter Technical Conference*, pp. 519–529 (1993).

[13] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E. and Corner, M. D.: Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers, *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'09)*, pp. 31–40 (2009).

[14] Barham, P., Dragovic, B., Fraser, K., Hand, S. and Harris, T.: Xen and the art of virtualization, *Proceedings of the 19th ACM symposium on Operating systems principles(SOSP '03)*, pp. 164–177 (2003).

© 2014 Information Processing Society of Japan