Vol.2019-OS-145 No.4
2019/2/28

ゲストOSのファイルキャッシュ識別による メモリ重複除外

久保田 曹嗣[1,a)]　鄭 俊俊[1]　芝 公仁[2]　瀧本 栄二[1]　齋藤 彰一[3]　毛利 公一[1]

概要：仮想化技術を用いたシステムを構築する際に，同一のゲスト OS やライブラリを利用する VM を複数実行することがある．このような環境では，メモリ内容の重複が発生し，この重複した部分をマージすることで，メモリの使用量を削減することができる．特に内容が重複することが多いメモリ領域は，ファイル内容がそのままマップされているファイルキャッシュである．そのため，メモリ重複除外の既存研究は，ファイルキャッシュを優先的にスキャンすることでマージ効率を向上させている．しかし，既存研究は，常に最新のファイルキャッシュのみをスキャンするため，マージ効率の向上に限界がある．以上の背景から，本論文では，ゲスト OS のファイルキャッシュとファイルパスを対応付けて管理し，その情報をもとにメモリ重複除外を行う手法を提案する．本手法により，ファイルキャッシュの中でも，同一内容のファイルがマップされている領域のみを優先的にスキャンすることができる．実装した提案手法を用いてページマージ数を計測する評価を行った結果，既存手法と比較してマージ効率を向上させることを確認した．

1. はじめに

仮想化技術を用いることで，1 つの物理マシン上で複数の仮想マシン（VM）を構築することができる．近年，この仮想化技術を用いたシステムやサービスが増加しており，それに比例して物理メモリの消費量が増大している．仮想化技術を用いたシステムとして，IaaS 型のクラウドサービスやマルウェア解析システムの cuckoo sandbox[1]，サイバー攻撃の演習システム [2] 等があり，これらは同一のゲスト OS やライブラリを利用する VM を複数実行する場合がある．このような環境では，図 1 のように，共有ライブラリの機械語命令列やプロセスが用いるメモリ内容，マップされたカーネルイメージの内容等が重複し，この重複した部分をマージすることでメモリの使用量を削減することができる．

Linux には，実際にメモリ重複除外を行う機能として Kernel Same page Merging(KSM)[3] が実装されている．KSM は，システムコールで指定されたメモリ領域をスキャンし，内容が一致するページをマージする機能を持つ．KSM のメモリスキャンは，スキャン対象のメモリ領域から 1 ページずつ取り出して，ページ内容を比較することで行う．スキャンするメモリ領域は，プロセスが発行する madvise システムコールで固定的に指定し，スキャンするページは，指定されたメモリ領域からアドレス順に取り出す．ただし，前述のように，KSM は，総当たりでメモリスキャンを行うため，スキャン時にマージ可能なページを発見できる確率（以下，ヒット率）が低く，マージ効率が悪いという課題がある．

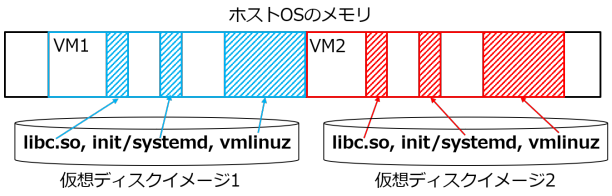

図 1　内容が重複するメモリ

KSM を改善した研究として，ファイルキャッシュに着目した XLH[4] がある．ファイルキャッシュは，I/O を高速化するため，ディスク上のファイルデータをメモリにキャッシュするものであり，ファイル内容がそのままメモリに格納されている．そのため，複数のゲスト OS が同一内容のファイルを用いる場合，各ゲスト OS が持つファイルキャッシュは内容が重複する可能性が高くなる．XLH は，KVM[5] が QEMU[6] を通じてディスク I/O をエミュレーションする際の I/O 先アドレスを用いて，ゲスト OS のファイルキャッシュを特定する．そして，特定したファ

[1] 立命館大学
Ritsumeikan University
[2] 龍谷大学
Ryukoku University
[3] 名古屋工業大学
Nagoya Institute of Technology
[a)] skubota@asl.cs.ritsumei.ac.jp

イルキャッシュから優先してスキャンすることで，ヒット率を向上させ，マージ効率を向上させる．しかし，XLH は，ファイルキャッシュかどうかを判断するのみであり，ファイルキャッシュが一致しているかに関わらず，常に最新のファイルキャッシュからスキャンページを取り出す．そのため，ヒット率の向上に限界があるという課題がある．

以上の背景から，本論文では，ゲスト OS のファイルキャッシュと仮想ディスクイメージ内のファイルパスを対応付けて管理し，その情報をもとにメモリ重複除外を行う手法を提案する．提案手法は，ゲスト OS の I/O に用いられるファイルのファイルパスをヒントとして，ゲスト OS のファイルキャッシュがどのファイルと対応付いているかを識別する．そして，識別した情報を用いて，同一内容のファイルがマップされているファイルキャッシュを優先的にメモリスキャンの対象とする．これにより，提案手法は，内容が一致するファイルキャッシュがマップされたページを優先的にスキャンすることができるため，既存手法よりヒット率向上させることができる．

提案手法を実現するために，ゲスト OS からホスト OS に対してファイルパスを送信する実装と，ホスト OS でファイルキャッシュを識別する処理を実装を行った．ゲスト OS からホスト OS へのヒント送信処理は，ゲスト OS から QEMU に対するヒント送信を virtio[7] を用いて実現し，QEMU からホスト OS へのヒント送信を新たなシステムコールを追加することで実現した．ホスト OS でファイルキャッシュを識別する処理は，ファイルパスが一致しているかを検索する file path tree を構築することで実現した．この実装を用いて，複数の VM 起動時のページマージ効率の計測を行った結果，既存手法と比較してマージ効率が向上することを確認した．

以下，本論文では 2 章で関連研究として KSM と XLH について述べ，3 章で提案手法の設計について述べ，4 章で提案手法の実装について述べる．その後，5 章でページマージ効率の評価について述べ，6 章で提案手法の課題と今後の展望について述べる．

2. 関連研究

2.1 Kernel Same page Merging

Kernel Same page Merging(KSM)[3] は，メモリをスキャンして同一内容のページをマージする Linux カーネルの機能であり，実体は ksmd という名前のカーネルスレッドとして動作する．

2.1.1 スキャン対象となるメモリ

KSM がスキャン対象とするメモリ領域を図 2 に示す．VM のメモリは，ヒープなどのスワップしなければ開放できないメモリ領域である Anonymous な領域にマッピングされる．KSM は，複数の VM 実行時を想定した機能であるため，Anonymous な領域のみをスキャン対象とする．ただし，全ての Anonymous なメモリ領域ではなく，madvise システムコールで指定された範囲のみが対象となる．

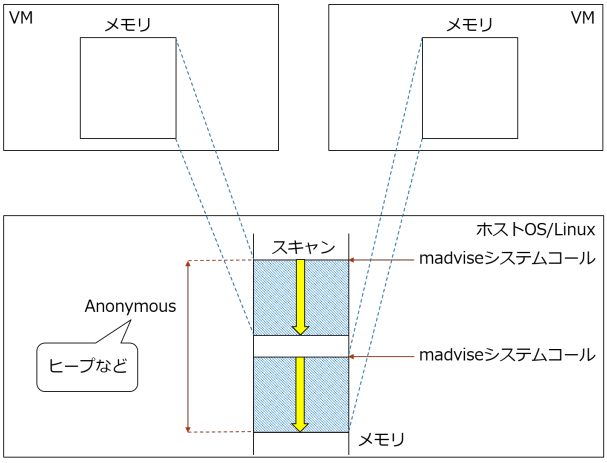

図 2　スキャン対象となるメモリ

2.1.2 メモリスキャンの方法

KSM は，一致する物理ページの探索に stable tree と unstable tree と呼ぶ 2 つの赤黒木 [8]（図 3）を用いる．赤黒木は，ノードに赤と黒の色を付けた二分探索木であり，KSM で用いる 2 つの赤黒木は，各ノードのインデックスに物理ページの内容を用いる．

stable tree

stable tree は，マージが完了した複数の仮想ページとそれらの実体となる 1 つの物理ページ (KSM ページ) を保持するための赤黒木である．マージした仮想ページは，Copy On Write(COW) により書込みを保護する．もし，マージした仮想ページに対して書込みが行われると，ページのアンマージを行う．KSM ページの管理する仮想ページが，アンマージにより 1 ページのみになった場合，該当の KSM ページを持つノードを stable tree から削除する．

unstable tree

unstable tree は，しばらくページの書換えが行われておらず，ヒットしやすいページを保持するための赤黒木である．unstable tree に挿入するページは，挿入時にページ内容のハッシュ値計算をすることで，ページ書換えの有無を検知する．

2.1.3 課題

KSM は，各プロセスが発行する madvise システムコールにより指定されたスキャン対象のメモリ範囲から，アドレス順にページを取り出してスキャンを行う．VM を 2 つ実行する場合，ヒットするのは 1 つ目の VM が指定したメモリ領域をスキャンし終わり，2 つ目の VM が指定したメモリ領域をスキャンし始めてからである．このように，単純にアドレス順にページを取り出し，総当たりでスキャンを行うため，マージ開始速度が遅くなる．また，ファイルキャッシュの破棄が起こるような環境では，マージ開始速

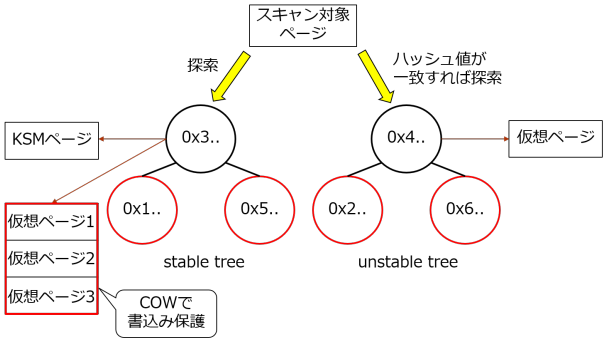

図 3　stable tree と unstable tree

度が遅くなることでファイルキャッシュの破棄によりマージ機会を失ってしまうこともある．以上で述べた理由から，KSM はヒット率が低くなるため，ページマージ効率が悪いという課題がある．

2.2　XLH

XLH[4] は，VM のファイルキャッシュは内容が重複しやすいというヒントを元に，重複除外の効率を向上させることを目的とした研究である．

2.2.1　ファイルキャッシュ

ファイルキャッシュは，ディスク上にあるファイルの内容を一時的にメモリ上に保存するキャッシュである．ファイルキャッシュは，ディスクの内容がそのままメモリにマップされているため，同一環境の VM 間で内容が一致しやすい．また，Linux は空きメモリを積極的にファイルキャッシュとして消費していくため，ファイルキャッシュはサイズが大きくなりやすい．以上の理由から，XLH は，ファイルキャッシュとして用いられているメモリ領域をスキャンの優先対象とする．

2.2.2　ヒントの生成と保存

XLH は，ゲスト OS のファイルキャッシュをヒントとして優先的にスキャンを行うため，ゲスト OS のファイルキャッシュを特定しなければならない．QEMU-KVM のディスク I/O（ファイル読込み時）とヒント生成を図 4 に示し，以下で説明する．

(1) VM が仮想ディスクに対して read 要求を発行する際，I/O 関係の命令がホストマシンの CPU によりトラップされることで VMExit が発生し，ホスト OS の KVM に処理が遷移する．

(2) KVM は，VMExit の要因を解釈し，QEMU に I/O 処理の依頼をする．

(3) QEMU は，ホスト OS に read システムコールを発行することで，依頼を受けた read 要求の I/O エミュレーションを行う．

(4) read システムコールを受けたホスト OS は，vfs_read 関数を呼び出し，ディスクに対して read 要求を発行する．read 要求のデータ転送先は VM のファイルキャッシュとなるため，XLH はこの関数内でデータ転送先のアドレスを用いてヒント生成を行う．

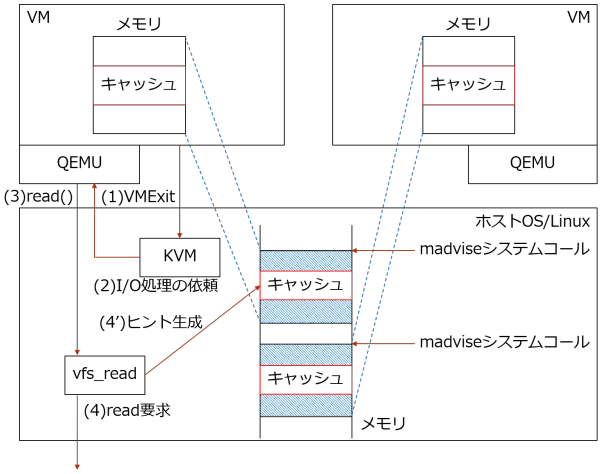

図 4　QEMU-KVM のディスク I/O と XLH のヒント生成

ファイルキャッシュの特定は，vfs_read 関数に渡された引数を参照することで，ヒント生成処理は，XLH 用に拡張を加えた madvise システムコール内の処理を呼び出すことで実現している．また，XLH は，ファイルの読み込み時だけでなく書き込み時にもヒント生成を行うため，vfs_write 関数でも同様の処理を行う．

生成したヒントは，ヒント保存用のスタックに保存し，ヒントの取り出し時は，常に最新のものから順に取得する．また，古いヒントを自動的に破棄するため，ヒント保存用スタックは循環スタックというものを用いる．循環スタックは，スタックの最大サイズまでデータがプッシュされると，スタックのトップを頂点から底に変更するものである．スタックの底には一番古いデータが格納されているため，古いヒントを破棄することができ，また，データ溢れも発生しない．

2.2.3　評価

XLH は，文献 [4] 内で KSM とページマージ効率の比較評価が行われている．評価内容は，複数の VM を立ち上げ，それぞれの VM でカーネルビルドを行っている状態でページマージ数を 10 分間計測している．実際に評価を行うと，マージの開始自体が KSM よりも早く，また，10 分後のマージ数も XLH の方が多いという結果が得られている．この結果から，XLH は有効であると結論付けている．

2.2.4　課題

XLH は，ファイルキャッシュかどうかの判断のみを行い，内容が一致しているかに関わらず，常に最新のヒントをスキャン対象ページとする．そのため，ヒット率の向上に限界がある．

3.　提案手法

3.1　概要

本論文では，ゲスト OS のファイルキャッシュが仮想ディ

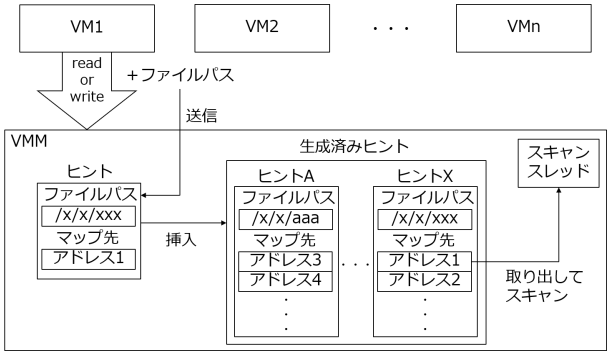

図 5　提案手法の全体構成

スクイメージ上のどのファイルパスと対応付いているかを識別し，その情報を元にメモリ重複除外を行う手法を提案する．提案手法では，VM が I/O に用いたファイルのファイルパスを VMM に送信する．VMM は，受信したファイルパスと I/O 先のアドレスを対応付けることで，ファイルキャッシュをファイル単位で識別し，同一内容のファイルキャッシュを優先的にスキャンする．これにより，スキャン時のヒット率を向上することができるため，マージ効率を向上させることができる．

提案手法の全体図を図 5 に示す．VM がディスク I/O を行う際，VMM に対して I/O 対象ファイルのファイルパスを送信する．ファイルパスを受け取った VMM は，ファイルパスとそのファイルが VM のファイルキャッシュとしてマップされるアドレスを対応付け，ヒントを生成する．生成したヒントは，既に生成済みのヒントから同一ファイルパスを持つヒントを探索し，見つけた場合にそのマップ先を追加する．もし同一ファイルパスを持つヒントが存在しなかった場合，新たなヒントとして保存する．重複除外を行うスキャンスレッド（KSM では ksmd）は，生成済みのヒントから一番新しいヒントを取り出し，メモリスキャンを行う．

3.2　設計

3.2.1　ヒント送信部分

提案手法は，ゲスト OS からホスト OS に対して，I/O に用いたファイルのファイルパスをヒントとして送信する．提案手法を QEMU-KVM 上で実現する場合の設計を図 8 に示し，以下で説明する．

(1) ゲスト OS 上で動作しているアプリケーションが read システムコールを発行すると，ゲスト OS に処理が遷移する．
(2) ゲスト OS は，ファイルの物理的な格納場所を特定し，ディスクに対する read 要求を発行するため，ブロックデバイス用のデバイスドライバを呼び出す．呼び出されたデバイスドライバは，ヒントを QEMU に送信する．
(3) デバイスドライバは，ヒント送信後に VMExit を発生させ，KVM に処理を遷移させる．
(4) KVM は，VMExit の要因を判断し，QEMU に I/O 処理の依頼をする．
(5) QEMU は，ホスト OS にシステムコールを発行することで，依頼を受けた read 要求の I/O エミュレーションとヒントの送信を行う．

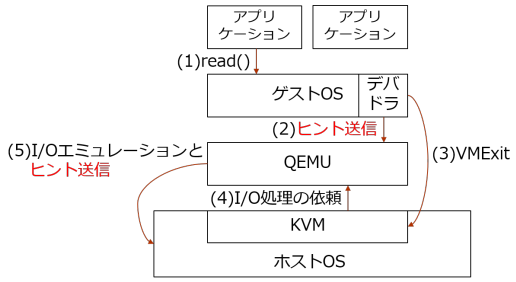

図 6　ヒント送信部分の設計

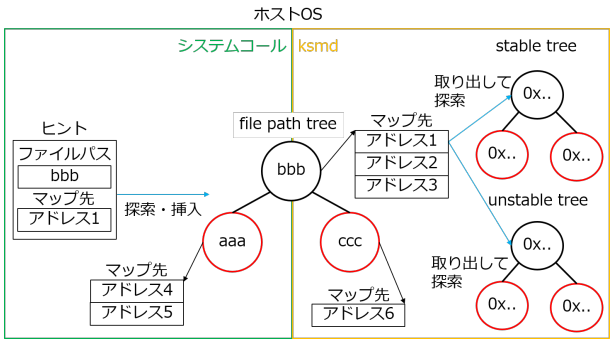

図 7　重複除外処理の設計

手順 (2) において，ファイルパスは，デバイスドライバを呼び出す段階ではディスク内のセクタ番号に置き換えられているため，取得することができない．そのため，デバイスドライバまでファイルパスを伝える必要がある．

3.2.2　重複除外処理

重複除外処理は，ホスト OS で I/O エミュレーション時のシステムコール内の部分と ksmd 内の部分で別々の処理を行う．重複除外処理の設計を図 7 に示す．システムコール内では，ヒントを生成し，生成したヒントと同一のファイルパスを持つヒントを探索する．ヒントの探索は，ハッシュ化したファイルパスを index とした赤黒木である file path tree を用いて行う．file path tree の各ノードは，それぞれのファイルが実際にマップされているファイルキャッシュのリストを保持している．

ksmd 内では，file path tree から最新のヒントを保持しているノードを参照し．そのノードが保持しているファイルパスのマップ先からアドレス順にスキャンページを取得する．スキャンページの取得以降は，XLH と同様の処理を行う．

XLH は，ヒントスタックでヒントの保存と取り出しを行っていた．提案手法は，file path tree にヒントを保存することでファイルキャッシュの識別を実現する．

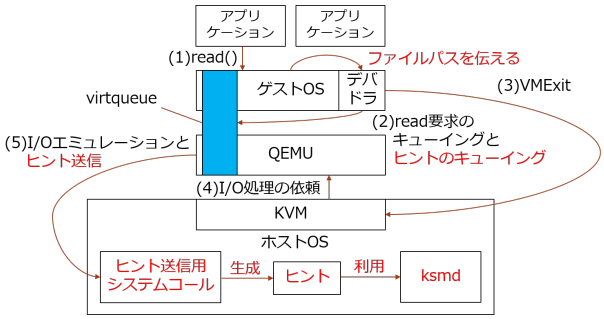

図 8　実装内容

4. 提案手法の実装

提案手法を実現するため，ゲスト OS，ホスト OS，QEMU を改変することで実装を行った．実装環境は，ゲスト OS，ホスト OS は Linux-4.4.0，QEMU はバージョン 2.5.0 である．本章では，提案手法の実装を，ゲスト OS から QEMU へのヒント送信，QEMU からホスト OS へのヒント送信，重複除外部分の 3 つに分けて述べる．

4.1 実装内容

行った実装を図 8 に示す．ヒント送信部分は，ゲスト OS から QEMU へのヒント送信に必要な，ゲスト OS からデバイスドライバへファイルパスを伝える処理と，virtio[7] 用のキューである virtqueue にヒントをキューイングする処理を実装した．また，QEMU からホスト OS へのヒント送信に必要な，ホスト OS にヒント送信用システムコールの追加と，追加したシステムコールを用いて QEMU からホスト OS にヒントを送信する処理を実装した．重複除外処理は，ヒント送信用システムコールでヒントを生成する処理と，ksmd で生成したヒントを用いて重複除外を行う処理を実装した．

4.2 ゲスト OS から QEMU へのヒント送信

4.2.1 デバイスドライバでファイルパスを取得する処理

提案手法は，ファイルパスをヒントとして用いる．しかし，デバイスドライバが呼び出される段階では，ファイルはディスク上のセクタ番号とサイズのみで表されるようになるため，ファイルパスを取得することができない．そのため，ファイル I/O を行うシステムコールを改変することで，デバイスドライバでのファイルパス取得処理を実装した．ファイルを読み込むシステムコールは，read システムコールを，ファイルに書き込むシステムコールは，write システムコールではなくファイルを同期する fsync システムコールを改変することで実装した．これは，write システムコールは，メモリ上のファイルキャッシュに書き込むため I/O エミュレーションが発生せず，また，QEMU からホスト OS へのヒント送信は I/O エミュレーション時に行うためである．実装は，デバイスドライバで参照できる構造体を拡張し，デバイスドライバ呼び出し前にその構造体にファイルパスを格納する処理を追加した．

提案手法は，ヒントとしてファイルパスを用いるが，ファイルパスはデータサイズが可変長であるため，ヒントとして扱いづらい．そのため，ゲスト OS でヒントをハッシュ化する処理を実装した．ハッシュ計算には，計算のコストが低く，また Linux カーネルに実装されており簡単に使用することができる xxhash[9] を用いた．提案手法の元としている Linux-4.4.0 ではまだ xxhash が実装されていなかったため，Linux-4.20 のソースコードから移植することで使用した．

4.2.2 virtio を用いたヒント送信処理

ゲスト OS から QEMU へのヒント送信は，オーバヘッドを最小限にするため，virtio を用いて実装を行う．virtio を用いた QEMU-KVM の I/O を，図 8 を用いて以下で説明する．

(1) ゲスト OS 上で動作しているアプリケーションが read システムコールを発行すると，ゲスト OS に処理が遷移する．
(2) ゲスト OS は，ファイルの物理的な格納場所を特定し，ディスクに対する read 要求を発行するため，ブロックデバイス用のデバイスドライバを呼び出す．呼び出されたデバイスドライバは，read 要求をゲスト OS と QEMU で共有しているキューである virtqueue にキューイングする．
(3) デバイスドライバは，キューに一定以上のデータが格納されていた場合，VMExit を発生させて KVM に処理を遷移させる．
(4) KVM は，VMExit の要因を判断し，QEMU に I/O 処理の依頼をする．
(5) QEMU は，virtqueue から I/O 要求を取り出し，ホスト OS にシステムコールを発行することで，取り出した I/O 要求のエミュレーションを行う．

従来の I/O は，ゲスト OS の処理で大量の VMExit が発生し，オーバヘッドが大きくなるという問題点があった．virtio を用いることで，virtqueue に I/O 要求をキューイングできるようになるため，複数の I/O 要求を VMExit なしで QEMU に伝えることができる．

ヒント送信の実装は，virtqueue に格納するデータを表す構造体を拡張し，ヒント送信用のメンバを追加した．また，ゲスト OS の virtio デバイスドライバを改変し，追加したメンバにヒントを格納する処理を追加した．

4.3 QEMU からホスト OS へのヒント送信

4.3.1 ヒント送信用のシステムコールの追加

QEMU は，ファイル I/O を行うシステムコールによりディスク I/O のエミュレーションを行う．QEMU からホスト OS へヒントとなるファイルパスを送信するため，図 8

に示すように，ホスト OS にヒント送信用のシステムコールを追加した．追加したシステムコールを以下に示す．

- ssize_t preadh(pread システムコールの引数, hint);
- ssize_t pwriteh(pwrite システムコールの引数, hint);
- ssize_t preadvh(preadv システムコールの引数, hint);
- ssize_t pwritevh(pwritev システムコールの引数, hint);

これらのシステムコールは，pread，pwrite，preadv，pwritev システムコールをヒント用に拡張したものである．基本的には元となったシステムコールと同様の処理を行い，引数で指定されたヒントを用いて提案手法用の処理を行う．

4.3.2 I/O エミュレーション時のヒント送信処理

QEMU の I/O エミュレーションは，virtqueue から I/O 要求を取得し，ゲスト物理アドレスのホスト仮想アドレスへの変換と I/O 対象の仮想ディスクイメージの特定を行い，それらのデータを引数として I/O 用のシステムコールを発行することで行われる．QEMU の I/O エミュレーションに用いられているシステムコールを前項で述べたシステムコールに置き換えることで，ヒント送信処理を実現した．

実際にヒント送信を行うためには，I/O エミュレーションを行う関数でヒントを参照できるようにしなければならない．そのため，I/O エミュレーションを行う関数で参照できる構造体を拡張し，ヒント格納用のメンバを追加した．これにより，I/O エミュレーションを行う関数でヒントが参照できるようになった．

4.4 重複除外部分

4.4.1 ヒント生成と file path tree の構築

提案手法は，4.3.1 項で説明したシステムコールにより，QEMU からヒントとしてファイルパスのハッシュ値を受け取る．提案手法の重複除外を実現するため，ホスト OS を改変し，これらシステムコール内にヒントの生成と file path tree を構築する処理を実装した．実装した処理を図 9 で示す．システムコールは，受け取った引数から提案手法で用いるヒントを生成する．生成したヒントは，ファイルパスのハッシュ値を index とした file path tree に挿入される．file path tree の各ノードは，ファイルキャッシュのアドレスを保存するヒントリストの先頭へのポインタを保持している，file path tree の探索時に，ファイルパスのハッシュ値が一致するノードがあれば該当ノードのヒントリストに生成したヒントを追加する，一致するノードがなければ file path tree に新たなノードを追加し，新たにヒントリストを構築する．

ヒントを取り出す速度よりもヒントを生成する速度のほうが速い場合，大量のヒントが file path tree の保存されてしまうことになる．XLH の場合は，ヒントを循環スタックに保存しているため，この問題を考慮する必要がなかった．しかし，提案手法は，ヒントを file path tree とヒントリストで保存するため，ヒントの最大数を制限する必要がある．ヒント最大数を制限するため，実装した処理を図 10 に示す．file path tree につなぐヒントリストとは別に，すべてのヒントを管理するためのヒント管理リストを作成する．そして，生成したヒントは，ヒントリストだけでなく，ヒント管理リストにも挿入する．ヒントの最大数を超えてヒント管理リストにデータを挿入する場合，ヒント管理リストに繋がれた一番古いヒントを破棄する．また，破棄するヒントと同じヒントを file path tree に繋がれたヒントリストからも削除する．これらの処理により，ヒントの最大数を制限し，且つ一番古いヒントを破棄する処理を実現することができる．

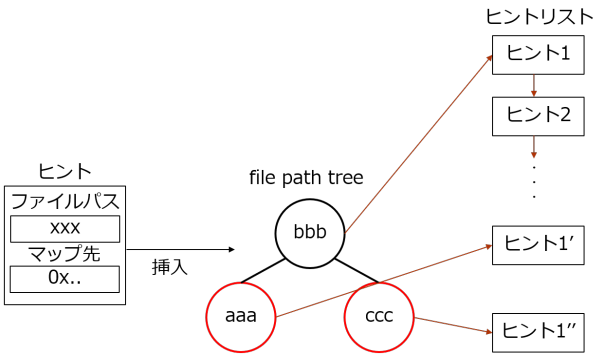

図 9　システムコールで行う処理

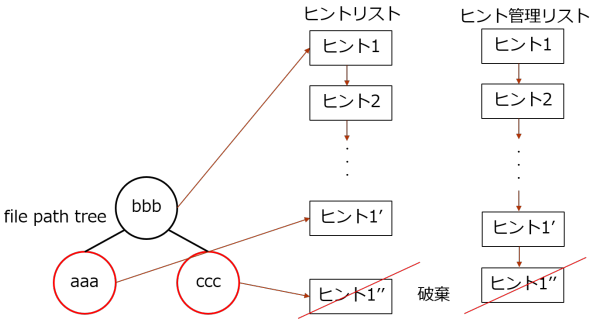

図 10　ヒントの最大数を制限する処理

4.4.2 スキャンページの取得

提案手法のスキャンを実現するため，ゲスト OS を改変し，ksmd で実行する処理に file path tree からスキャンページを取得する処理を実装した．スキャンページ取得時の処理を図 11 に示す．まず file path tree 内の直近で更新があったノードからヒントリストを参照する．そして，ヒントリストの先頭から最新のヒント（ヒント 1）を取り出し，ヒント 1 からアドレス順にスキャンページを取得する．ヒント 1 のすべてのアドレスを取得し終えるまで，常にヒント 1 からスキャンページを取得する．ヒント 1 のアドレスが終端に達した場合は，ヒント 1 をヒントリストとヒント管理リストから削除し，ヒントリストの先頭から次のヒント（ヒント 2）を取り出し，ヒント 2 からアドレス順にスキャンページを取得する．ヒントリスト内のヒント

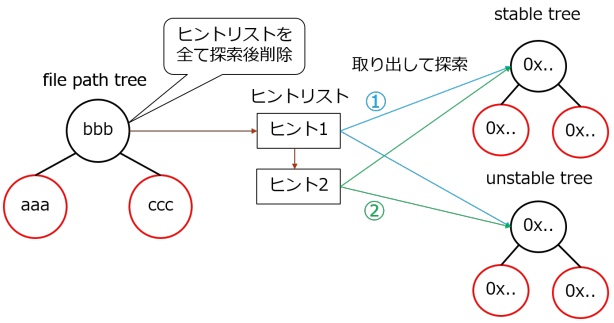

図 11　スキャンページの取得時の処理

を全て削除するまでは，この処理でスキャンページを取得する．ヒントリスト内のヒントが全て削除された場合は，file path tree から該当ヒントリストを保持しているノードを削除する．そして，file path tree から直近で更新があったノードから新たなヒントリストを参照する．

スキャンページ取得処理の後は，XLH と同様の処理を行う．また，KSM でのスキャン機能も残しているため，KSM との共存もできる．

5. 評価

実装した提案手法を用いて，ページマージ効率の評価を行った．本章では，評価に用いた環境と評価内容，評価結果について述べる．

5.1 評価環境

ホスト OS の環境を表 1 に示す．XLH は，Linux-3.4-rc3 を元にした実装が公開されている [10]．提案手法の実装は，Linux-4.4.0 を元にしているため，XLH を Linux-4.4.0 に移植した．そのため，ホスト OS は，XLH の評価では表記したバージョンの移植版のものを，提案手法の評価では表記したバージョンを元に改変したものを使用している．

ゲスト OS の環境を表 2 に示す．提案手法の評価では，QEMU とゲスト OS はそれぞれ表記したバージョンを元に改変したものを使用している．

5.2 評価内容

提案手法の有効性を確認するため，KSM，XLH，提案手法のそれぞれでページマージ効率の比較評価を行う．評価は，ホスト OS で 2 つの VM を立ち上げ，10 分間ページマージ数を計測することで行う．KSM のパラメータを操作し，100 ページスキャンすると 100ms のスリープをする設定にしているため，10 分間で 60 万ページのスキャンが行われる．また，ディスク I/O を大量に発生させるため，それぞれの VM ではカーネルビルドを行う．

ゲスト OS のファイルキャッシュ破棄の有無が提案手法に与える影響を評価するため，本評価は，ゲスト OS に 512MB もしくは 2GB のメモリを割り当てる 2 つのシナリオで行う．512MB のメモリ容量だとファイルキャッシュの破棄が発生し，2GB のメモリ容量だとファイルキャッシュの破棄が発生しない．

表 1　ホスト OS の環境

項目	内容
CPU	intel Core i7 6700(4 コア)
メモリ	16GB
OS(カーネル)	Ubuntu-16.04(Linux-4.4.0)

表 2　ゲスト OS の環境

項目	内容
QEMU	QEMU-2.5.0
CPU	仮想 CPU(1 コア)
メモリ	512MB or 2GB
OS(カーネル)	Ubuntu-16.04(Linux-4.4.0)

5.3 評価結果

5.3.1 2GB

評価結果を図 12 に示す．KSM は，スキャンメモリ範囲が大きいため，600 経過後もほとんどマージができていない．提案手法と XLH を比較すると，1.5〜2 倍のページをマージしており，600 秒で約 30 万ページをマージしている．10 分間のスキャンページ数が 60 万ページであるため，2 ページのスキャンで 1 ページをマージできていることになる．本評価では，VM を 2 つ立ち上げているため，ヒット率はほぼ限界まで向上している．この結果から，ファイルキャッシュを十分に確保できる環境において，提案手法はマージ効率を向上させることを確認した．

5.3.2 512MB

評価結果を図 13 に示す．提案手法と KSM を比較すると，600 秒後の最終結果で約 3 倍のページをマージできている．KSM は，約 250 秒経過後にページマージが開始するが，提案手法は同時点で既に約 7 万ページのマージが行われている．また，XLH と比較すると，約 180 秒までは 2 倍の効率でマージできている．600 秒後の最終結果でも，約 1.5 倍多くのページをマージできている．これらの結果から，提案手法は，キャッシュの破棄が起こるような環境でもページマージ効率を向上させることを確認した．

6. 課題と今後の展望

提案手法は，ホスト OS だけでなくゲスト OS と QEMU にも実装が必要となるため，導入コストが大きいという課題がある．QEMU の実装は，ホスト OS にシステムコールでヒントを伝える処理が必要となるため，削減することは難しい．ゲスト OS の実装は，ファイルパスをデバイスドライバまで伝える処理以外は全て virtio ドライバ内で完結している．そのため，virtio ドライバ内で取得できるデータをヒントとすることができれば，ドライバの改変のみで

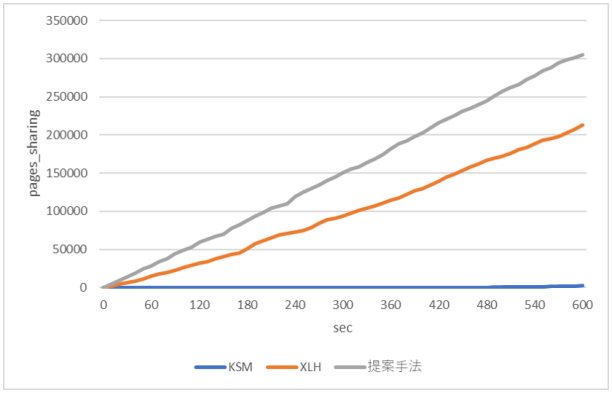

図 12 2GB の VM

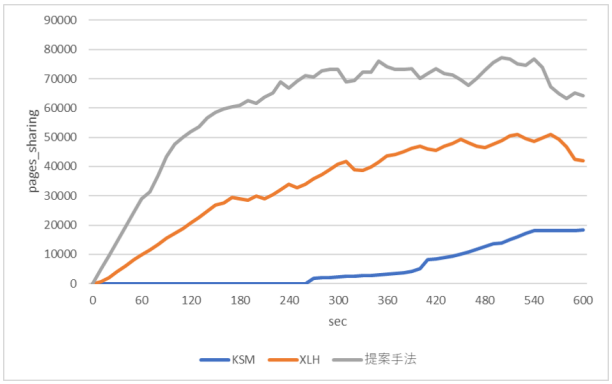

図 13 512MB の VM

提案手法を実現できるため，導入コストを小さくすることができる．

提案手法のヒントは，本論文ではファイルパスのハッシュ値を用いている．しかし，ファイルパスは動的なデータであるため，書き換えによりマージ効率が低下する可能性が考えられる．そのような場合は，/home 以下のファイルなど，内容が一致しない可能性が高いファイルをヒント生成の対象にしないことで，マージ効率の低下を防ぐような対処方法が有効であると考えている．また，環境に特化することで静的なデータをヒントとして用いるようなことも考えられる．例えば，仮想ディスクイメージを共有しているような環境を想定すると，同一ファイルの i ノード番号は各 VM で一致するため，i ノード番号がヒントの有力な候補となる．

提案手法の評価は，本論文では VM でカーネルビルドを行い，ページマージ数を計測している．これにより，カーネルビルドのような I/O が大量に行われる環境では，提案手法の有効性を確認できた．今後は，各 VM で Web サーバを立ち上げるような，実環境を模したシナリオで評価を行う必要がある．

7. おわりに

本論文では，ゲスト OS のファイルキャッシュをファイルパスと対応付けて識別することで，重複除外の効率を向上させる手法について述べた．ゲスト OS のファイルキャッシュはメモリ内容が重複する可能性が高いため，ファイルキャッシュを優先的にスキャンすることで KSM のマージ効率を向上させる既存研究が存在する．しかし，既存研究はスキャン対象ページの取り出し方に問題があった．そのため，ゲスト OS のファイルキャッシュを識別し，同一内容のファイルがマップされたファイルキャッシュを優先的にスキャン対象ページとする手法を提案した．

評価では，2 つの VM に対してそれぞれ 2GB もしくは 512MB のメモリを割り当て，それぞれでカーネルビルドを行い，ページマージ数を計測した．評価結果から，2GB を割り当てた場合は，60 万ページのスキャンで 30 万ページをマージできることを確認し，512MB を割り当てた場合は，既存研究より 1.5〜2 倍の効率でマージできることを確認した．そのため，ファイルキャッシュ破棄の有無に関係なく，提案手法は既存手法よりページマージ効率を向上させることを確認した．

参考文献

[1] Foundation, C.: Automated Malware Analysis - Cuckoo Sandbox, `https://cuckoosandbox.org/`.
[2] Pham, C., Tang, D., Chinen, K.-i. and Beuran, R.: CyRIS: A Cyber Range Instantiation System for Facilitating Security Training, *Proceedings of the Seventh Symposium on Information and Communication Technology*, SoICT '16, New York, NY, USA, ACM, pp. 251–258 (2016).
[3] ARCANGELI, A., E. I. and WRIGHT, C.: Increasing memory density by using KSM, *In Proceedings of the Linux Symposium (Montreal, Quebec, Canada)*, pp. 19–28 (2009).
[4] K. Miller, F. Franz, T. G. M. R. M. H.-l. and Bellosa, F.: XLH: more effective memory deduplication scanners through cross-layer hints, *Proceedings of the 2013 USENIX conference on Annual Technical Conference* (2013).
[5] KVM: Main Page — KVM,, `https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792` (2016).
[6] Bellard, F.: QEMU, a fast and portable dynamic translator, *Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 41–46 (2005).
[7] Russell, R.: virtio: towards a de-facto standard for virtual I/O devices, *ACM SIGOPS Operating Syst. Review (OSR*, p. 103 (2008).
[8] Hinze, R.: Constructing Red-Black Trees, Technical Report IAI-TR-99-6, Institut für Informatik III, Universität Bonn (1999).
[9] Collet, Y.: xxHash - Extremely fast non-cryptographic hash algorithm, `https://cyan4973.github.io/xxHash/`.
[10] Bellosa, F.: ITEC-OS Research - Memory Deduplication, `https://os.itec.kit.edu/2652.php`.