Vol.2017-OS-140 No.14
2017/5/17

アプリケーションレベルのメモリ管理を考慮した HugePage 割り当て機構

清水 祐太郎[†1,a)]　山田 浩史[†1,b)]

概要：メモリ技術の発達により，メインメモリサイズの巨大化が進んでいる．こうした主記憶が大規模な環境では TLB ミスによるレイテンシが問題となる．TLB ミスを減らす手法の一つとして， HugePage の利用が挙げられる．HugePage を利用することによって，ページテーブルの 1 エントリあたりがカバーできるアドレスの範囲が拡大し， TLB のカバレッジも同様に増加する．これまでに HugePage を利用するための手法がいくつか提案されている．しかしながら，アプリケーション自身がメモリ管理を行う場合では，いずれの手法でも十分に効率的な割り当ては行えない．本研究では，アプリケーションレベルでのメモリ管理を考慮しながら，効率的に HugePage を割り当てるための手法を提案する．今回提案する手法によって，必要とする分だけ HugePage を利用することが可能になる．本研究では，提案手法を Linux kernel 4.7.10 と memcached 1.4.31 上に実装を行った．性能評価を行い既存手法と比較した結果，実メモリ使用量のデフォルトからの増加量を 93.6% 程度削減しながら，既存手法の約 99.4% のスループットを達成した．

1. はじめに

近年のメモリ技術の発達により，メインメモリサイズの巨大化が進んでいる．また，不揮発性メモリの発達により，メモリの集積度が飛躍的に向上した．それに伴い， x86 アーキテクチャがサポートしている 256TiB の仮想アドレス空間よりも大きいサイズのメモリを搭載可能なマシンが登場すると言われている [1].

こうした主記憶が大規模な環境では Translation Lookaside Buffer (TLB) ミスによるレイテンシが問題となる．プログラムがメモリにアクセスを行う際に，内部的にはページテーブルを辿って仮想アドレスから物理アドレスへ変換を行う必要がある．TLB は，メモリ管理ユニット内でその変換を高速に行えるように，変換の対応付けのキャッシュとしての役割を担っている．ハードウェア的に TLB の拡張は容易ではないため [1]， TLB サイズの成長はメインメモリのそれと比べると遥かに遅い．そのため，アプリケーションは大容量なメモリを利用できるにも関わらず， TLB ミスによって性能が劣化してしまう場合がある．実際に，現行のシステムにおいて 55% もの性能劣化を被るアプリケーションが存在する [2].

†1 現在，東京農工大学
Presently with Tokyo University of Agriculture and Technology
a) simiyu@asg.cs.tuat.ac.jp
b) hiroshiy@asg.cs.tuat.ac.jp

TLB ミスを減らす手法の一つとして， HugePage の利用が挙げられる．HugePage とは，通常のページよりも大きいサイズ単位でメモリを扱うことが可能となる，メモリ管理ユニットの機能である．たとえば，x86 や AMD64 では通常のページが 4KiB であるのに対し， HugePage を利用することで 2MiB，1GiB まで管理の単位サイズを大きくすることができる．HugePage を利用することによって，ページテーブルの 1 エントリあたりがカバーできるアドレスの範囲が拡大し， TLB のカバレッジも同様に増加する．

これまでに HugePage を利用するための手法がいくつか提案されている [3][4][5][6]．しかしながら，データベース管理システムや言語ランタイムなどアプリケーション自身がメモリ管理を行う場合では，いずれの手法でも十分に効率的な割り当ては行えない．HugePage を利用する既存の手法として，大きく分けて 2 通りの方法が存在する．一つ目はオペレーティングシステム (OS) が API を提供する方式である [3]．アプリケーションはこの API を介して OS に HugePage の利用をリクエストすることができる．この方式では既にマッピングしたページを HugePage に置き換えることは不可能であるため，予め確保しておく必要がある．そのため，アプリケーションが HugePage を確保したにも関わらず，アプリケーションの使用状況によってはそのページがあまり利用されないということが起こりうる．

二つ目は， OS がアプリケーション透過で通常のページを HugePage に置換する方式である [4][5][6]．この方式で

は，既存のアプリケーションにおいて，そのプログラム自体を一切変更することなく HugePage を利用することができる．しかし，アプリケーションが自身でメモリを管理する機構を備えている場合では，前者同様，効率的にこの方式を適用することは難しい．OS 側からすると，変換の対象とする領域がプロセスによって頻繁に使用されているのかどうかを判断することはできない．そのため，さほど使用されていない領域も含め，HugePage に置き換えるという非効率的な振る舞いが生じてしまう．このような挙動によって外部断片化の発生も懸念されるため，これを防ぐ手法 [7] の提案も行われている．また，HugePage 割り当ての公平性について考慮されていないため，先に起動したプロセスにばかり HugePage が割り当てられてしまい，各プロセス間で性能に差が出てしまうことが起こりうる [6].

本研究では，アプリケーションレベルでのメモリ管理を考慮しながら，効率的に HugePage を割り当てるための手法を提案する．本手法によって，アプリケーションは大きなコストをかけずに，必要とする分だけ HugePage を利用することが可能になる．提案手法では，OS カーネルがアプリケーションの各ページへのアクセス状況を監視し，頻繁に利用されているページをアプリケーションに通知する．通知を受けたアプリケーションは，通知されたページのアドレスを基に HugePage への置き換えを OS に依頼したり，メモリオブジェクトのコンパクションを行ったりすることが可能となる．参照される頻度の高いオブジェクトのみを HugePage 上に配置することで，不必要な HugePage の割り当てを極力避けつつ，かつアクセスオーバーヘッドの短縮を図る．

本論文の貢献は次のとおりである．

- 既存手法における無駄な HugePage の割り当てを避けるため，アクセスの集中しているページ群にのみ HugePage を割り当てる手法を提案した．
- ページへのアクセス数を記録し，HugePage へ変換のヒントをシグナルを用いてプロセスへ通知する機構を提案した．
- 提案手法を Linux kernel 4.7.10 と memcached 1.4.31 [8] 上に実装を行った．性能評価を行い既存手法と比較した結果，実メモリ使用量のデフォルトからの増加量を 93.6% 程度削減しながら，その約 99.4% のスループットを達成した．

本論文では，第 2 章で関連研究とその問題点について触れ，第 3 章でその解決策として本手法を提案する．第 4 章では提案手法を実現するためのシステムの設計，第 5 章では Linux カーネルおよびアプリケーションへの実装方法について述べる．第 6 章において提案手法の評価を行い，第 7 章で本研究のまとめと今後の課題を示す．

2. 関連研究

本章では，HugePage に関連した研究，および技術について述べる．現在，アプリケーションが HugePage を利用するには大別して 2 通りの方法が存在する．アプリケーションが能動的に HugePage を確保する方法と，透過的に OS カーネルが割り当てる方法である．ここではそれぞれの研究を紹介し，それらの問題点を挙げる．

2.1 能動的な HugePage の利用

アプリケーションが HugePage の利用を OS に対してリクエストする方式である．OS は HugePage を確保するための API を提供し，アプリケーションはこれを利用する．

2.1.1 hugetlbfs

hugetlbfs[3] は，現在 Linux において HugePage を利用するために提供されている仮想ファイルシステムである．アプリケーションは，mmap() システムコールの flags に MAP_HUGETLB を指定して呼ぶことで hugetlbfs を利用することができる．既存のアプリケーションにおいて hugetlbfs を利用したい場合には，libhugetlbfs を利用するなどして libc の mmap() をフックすることで適用が可能となる．

アプリケーションが hugetlbfs を利用する際には，いくつかの問題がある．まず，HugePage を確保できる数には限りがあるという点である．HugePage を利用するには連続した物理空きメモリが必要となるため，システムは予め一定量のメモリをプールする．割り当ては，このメモリプール内からしか行えない．

次に，通常のページと HugePage を動的に切り替えることができないという点が挙げられる．hugetlbfs は，マッピング時に HugePage としてメモリを確保する方式である．そのため，既にマッピングされている通常のページを HugePage に置き換えることはできない．

2.2 透過的な HugePage の利用

アプリケーションのメモリ空間内で，HugePage に置き換えが可能な連続領域が存在した場合，OS カーネルが自動的に HugePage に置き換えを行う方式である．前述の方式とは異なり，アプリケーションが HugePage の利用に関して意識をする必要はない．すなわち，既存のアプリケーションに対しても，何かしらの手を加えることなく HugePage の利用が可能である．

2.2.1 Transparent Huge Page (THP)

Linux のカーネルに実装されている，通常のサイズとして確保したページを HugePage に置き換えて提供する機構である．Transparent Huge Page[4] の動作は “always” と “madvise” の 2 通りに設定でき，それぞれ HugePage の割

り当て方が異なる．前者はアプリケーションの動作に関わらず，HugePage の割り当てができるページは可能な限り HugePage とする．実ページは，仮想ページに初めてアクセスした後に実際に割り当てが行われる．その際に呼ばれる Page fault handler 内で，対象の領域が 2MiB 以上でかつ anonymous であれば HugePage を割り当てる．

後者では，アプリケーションが HugePage の利用を OS に対してアドバイスする．アプリケーションは，HugePage の利用が妥当だとしたページを引数にとって，madvise システムコールを発行する．既存の通常ページを HugePage に置き換えるためのカーネルスレッドである khugepaged が，定期的にページ群を走査している．khugepaged は，アドバイスされたページ群を見つけると，新たに HugePage を確保してから内容をコピーし，以後はそのページを使用することになる．

しかしながら，THP の madvice ではあまり使用されていないページ群をまとめて HugePage に置き換えるという動作をしてしまうことがある．何故なら，OS 側からはそれらの領域がアクティブに使用されているのかどうかは判断できないからである．これらの手法では，実ページが割り当てられていないページ群は未使用と判断し，HugePage に置き換えることはしない．しかしながら，一度実ページが割り当てられれば，アンマッピングされるまでそのページを使用中であるとみなしてしまう．

2.2.2 Preferred superpage

Preferred superpage[5] では，アプリケーションの要求に対してシステムが適切な大きさの HugePage を割り当てることで，主記憶の断片化の抑制と最小限のディスクアクセスを図る．

HugePage を確保する際に無駄に大きな領域を割り当てると，未使用領域が点在してしまい断片化が進む．また，ページングの際に HugePage を丸ごと転送しなければならず，I/O コストも嵩む．そこで，システムが適切な大きさの HugePage を割り当てることを考える．HugePage を適切な大きさに保つためには，状況に応じて HugePage を併合して拡張したり，分割して縮小したりする必要がある．新たなページを確保した際に，HugePage が連続して存在している場合はそれらを一つにまとめる．それとは逆に，ページの解放や保護属性の変更など，ページ単位での制御が必要な場合は HugePage を小さく切り分け，それぞれ PageTable にマッピングし直すという処理を行う．

しかしながら，この方式でも THP と同様に，アプリケーションにおけるページの使用状況を考慮することはできない．そのため，HugePage の無駄な割り当てが生じてしまうことが考えられる．

2.2.3 Ingens

Ingens[6] では，既存の THP が有する問題点を洗い出し，それらを解決する手法を提案している．THP が持つ問題として，割り当て時間・内部断片化・割り当ての公平性・共有ページとの競合を挙げている．それぞれの問題を解決するための手法を提案し，それらを一つのシステムに統合している．

THP では，HugePage は一度確保されると，解放されるまでそのアドレスに確保され続ける．そのため，メモリ使用量が徐々に増加し，結果として利用できるメモリ領域が減ってしまう．これを避けるため，ページの利用状況に合わせて HugePage を割り当てたり分割したりする．カーネルスレッドはメモリ空間における HugePage に変換ができる枠内の使用状況を監視を行い，状況に応じて HugePage の割り当てや分割を行う．また，THP による動作は，割り当てができる箇所を見つけ次第割り当てを行う．すなわち，先に動作していたプロセスには多く割り当てられるが，あとから起動したプロセスにはほとんど割り当てられないなど，不均一な割り当てが行われる可能性がある．できる限り各プロセスに対して公平に HugePage を割り当てるために，Idleness penalty を用いる．各 HugePage に対してこれを適用し，Idle なページが少ないプロセスにより多くの HugePage の割り当てが行われるようにする．

Ingens ではページの利用状況は考慮されているが，局所的なアクセスの集中への対応は不十分である．この手法では各ページへのアクセス頻度を監視して，利用されているページの数が閾値を超えたら HugePage を割り当て，下回ったら分割という動作をする．しかし，アクセスが集中するページが HugePage へ変換ができるページ群のうちほんの一部だとすると，このページ群に HugePage が割り当てられることはない．たとえば，連続する 512 個の通常ページのうち，1 つのページに対してのみ集中的にアクセスがあるワークロードを考える．この場合，利用されているページ数は閾値を超えないため，このページ上にあるアイテムはいつまでたっても HugePage 上に配置されることはない．

3. 提案

本章では，アプリケーションと連動し，メモリの使用状況に合わせて HugePage を割り当てる手法を提案する．提案手法により，アクセスが集中するようなページ群に対して，必要とする分だけ HugePage を割り当てることが可能となる．それにより，スループットの改善とメモリ使用量の削減を図る．この提案は主に，ページに対するアクセス数のカウント，HugePage の割り当ての 2 つから構成される．

3.1 ページアクセスのカウントおよび通知

アクセスの集中するページを把握するため，各ページへのアクセス数をカウントする必要がある．アプリケーション自身で監視対象のページへの読み書きをすべて数えるこ

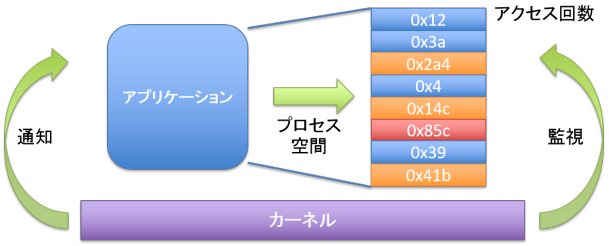

図 1 プロセス空間の監視と通知

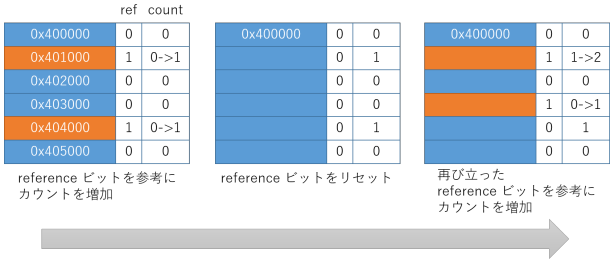

図 2 ページへのアクセスの監視の流れ

とは不可能ではない．しかしながら，これを行うのは非常にコストが大きく，アプリケーションのスループットの大幅な低下につながる．また，ソースコードが提供されていないような既存のアプリケーションに対しては，実現が極めて難しい．

そこで，本手法では OS カーネルによるプロセス空間の監視を行う．ハードウェアで提供されている機能を利用することで，カーネルは低コストでページに対してアクセスがあったか否かを判断することができる．カーネルは定期的にアクセスの有無の読み取りを行い，各ページの大まかなアクセス数を把握する．読み取りはアクセスが発生するたびに行われるわけではないため，正確なアクセス数のカウントは困難である．しかし，この手法を用いることで，プロセス空間全体を通したアクセスの偏りの傾向を把握することは可能である．

図 1 のようにプロセス空間の監視を行い，得られたアクセス数の情報を基に，カーネルはプロセスに対して HugePage の利用のヒントを与える．プロセスへの通知にはシグナルを採用することで，プロセスの通常の動作への影響を抑える．

3.2 HugePage の割り当て

通知を得たプロセスは，ページ群を HugePage へ置き換えることや，ページ内のオブジェクトのコンパクションなどを行うことができる．この HugePage に置き換えるかどうかなどの判断はアプリケーションが行う．カーネルは判断材料となる情報を提供するのみであり，判断自体には一切関わらない．なぜなら，実際にメモリをどのように利用しているかを把握しているのはアプリケーションのみであるからである．

アプリケーションは，通知を受けたページの数や，その分布の偏りを基に判断を行う．たとえば， 2MiB の連続した通常ページのうち，4 分の 1 以上にアクセスが集中すると，まとめて HugePage に置き換えると設定したとする．このとき，通知されたページ数がある閾値を超えた場合，その一帯のページ群を HugePage に変換するという依頼をアプリケーションが改めてカーネルに対して行う．アクセスが集中するページ数が，閾値を超えはしないがいくつか存在する，そのような場合では当該ページ上に存在していたメモリオブジェクトを別の HugePage 上にコンパクションを行う．

ただし，HugePage を割り当てることのできるページ数に限りがある．すなわち，先に動作していたプロセスにばかり HugePage が割り当てられると，後から動作を始めたプロセスにはそれほど割り当てが行えなくなる場合がある．そのような事態を避けるため，各プロセス間での HugePage の利用数を把握し，公平な割り当てを目指したい．

4. 設計

本章では 3.1 小節で述べたページアクセスのカウントおよび通知の設計について述べる．また，アプリケーションによる HugePage の利用についても触れる．

4.1 プロセス空間の監視

4.1.1 ページアクセスの追跡

各ページへのアクセスの監視は，ハードウェアで提供されている機能を利用して実現する．AMD64 アーキテクチャのページテーブルには，ページに対して読み込みや書き込みが行われるとセットされる reference ビットが存在する．このビットのセットは，ハードウェアで自動的に行われる．すなわち，カーネルは reference ビットを読み取ることで，プロセスが当該ページに対して読み書きを行ったかどうかを把握することができる．また，読み取った後にこのビットをクリアすることで，再びアクセスがあった時にビットが立つようになる．

ページアクセス数の記録は page 構造体に行う．page 構造体は全実ページにつき一つ存在しているため，ページのアクセス数を保持するには適当である．本研究では，page 構造体に新たな要素 acscount を追加し，この要素に対して記録を行う．図 2 に，reference ビットを読んで acscount を増加させる様子を示す．

アクセスカウントはカーネルスレッドによって行う．以下，本カーネルスレッドをアクセストラッカーと呼称する．このアクセストラッカーは，プロセスにマッピングされている全ページのページテーブルに対して読み取りを行う必要がある．vm_area_struct 構造体を走査することで，実際にマッピングされているページに対してのみページテーブルの取得を試みる．

アクセストラッカーは次のようにしてページへのアクセスを追跡，およびカウントを行う．全 vm_area_struct 構造体は task_struct 構造体， mm_struct 構造体を経由して単方向リストで得ることができる．監視対象とするプロセスの PID から task_struct 構造体を得た後，上記の通りページテーブルを得る．ページテーブルから reference ビットを読み取り，アクセスがあった場合は page 構造体の acscount を加算する．読み取った後は reference ビットをクリアすることで，次の読み取りのために準備を行う．監視対象とする全プロセスに対してひと通りのカウントを行った後，アクセストラッカーは一定時間休止する．

このアクセストラッカー自体は，監視対象のプロセスに対してコストを増加させる．この仕組みの実現は，先述の通りページテーブルの reference ビットを読み取り，クリアを行う．その際，キャッシュとの内容の一貫性を保持するために TLB の当該エントリがフラッシュされる．したがって，次回そのページにアクセスする際に TLB ミスが起き，全体としてスループットの低下が懸念される．そこで，アプリケーションは本手法を用いてある程度の HugePage への変換やコンパクションが済んだ後，自身を監視対象から外すことで性能の回復を図るようにする．

4.2 プロセスへの通知

アクセストラッカーがアクセスが集中するアクティブページを検知した際， HugePage を利用すべきページがあることをプロセスに通知するにはシグナルを用いる．シグナルを用いて情報をプロセスに渡す際，カーネルは siginfo_t 構造体にデータを格納してこれを受け渡す．この siginfo_t 構造体には，シグナルごとにいくつかのデータを格納できるように，多数の要素が存在している．しかしながら，この構造体を用いて通知を行うことのできる情報量には限りがある．

そこで本手法では，情報の取得には図 3 のようにアプリケーションが発行するシステムコールを組み合わせる．先述の通り，アクセストラッカーが提供するすべてのデータを siginfo_t 構造体でプロセスに渡すのは困難である．そのため，シグナルではアクセストラッカーが提供しようとしているアクティブページ数のみを通知する．シグナルを受け取ったプロセスは，システムコールを発行して当該ページのアドレスの取得を試みる．アプリケーションは通知されたページ数を基に相応のサイズのバッファを確保し，OS カーネルはそのバッファにページのアドレスを含む構造体を格納する．この仕組みにより，アプリケーションはアクティブページアドレスのリストを構造体の配列として取得し，扱うことができる．

4.3 HugePage の利用

アプリケーションが HugePage を利用するにあたり，

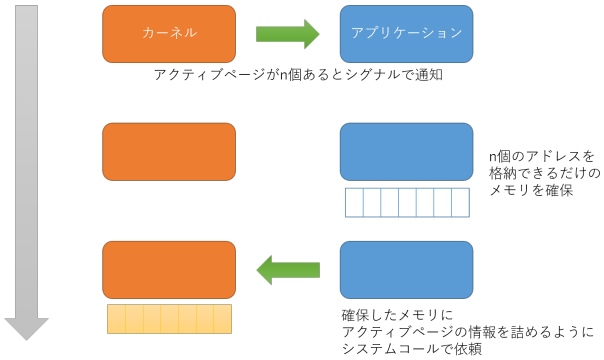

図 3 アクティブページ通知の流れ

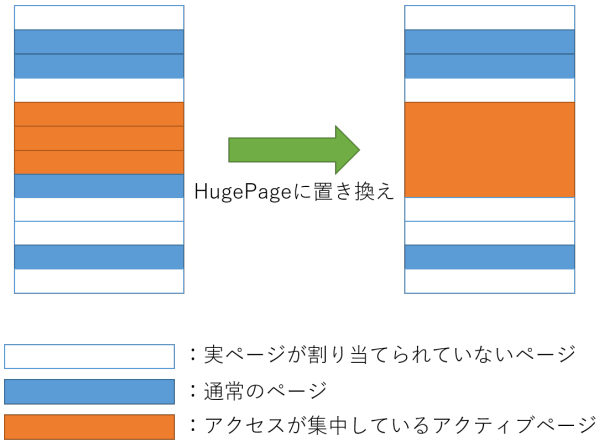

図 4 THP による HugePage への置き換え

本研究では新しい手法の提案は行わない．すなわち，既存の技術を組み合わせて利用することで実現する．まず，HugePage の利用を勧める通知を受けたプロセスが，図 4 に示すように一帯をまとめて HugePage に置き換えるという判断をした場合を考える．このとき，アプリケーションは THP を利用する．madvise() システムコールを発行し， OS カーネルに対して HugePage への置き換えを進言する．

次に，まとめて HugePage に置き換える事はしないが，一部のページ上のオブジェクトをコンパクションするという判断をした場合を考える．このときは，図 5 のように hugetlbfs を利用して別個に HugePage を用意し，その上にオブジェクトを移動させる．先ほどの THP の利用とは異なり，この場合はオブジェクトの配置されるアドレスが変化するため，対象アプリケーションのデータ構造を把握しておく必要がある．

5. 実装

本章では，提案機構の実装について述べる．図 6 に提案機構の概要を示す．対象とする OS カーネルは Linux kernel 4.7.10 ，アプリケーションは memcached 1.4.31 とする．OS カーネルには，ページアクセスの監視を行うカーネルスレッドを作成し，カウントの確認を行うための機能

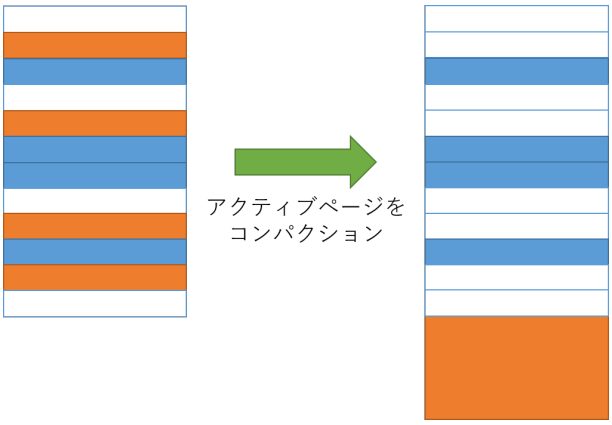

図 5 HugePage 上へのコンパクション

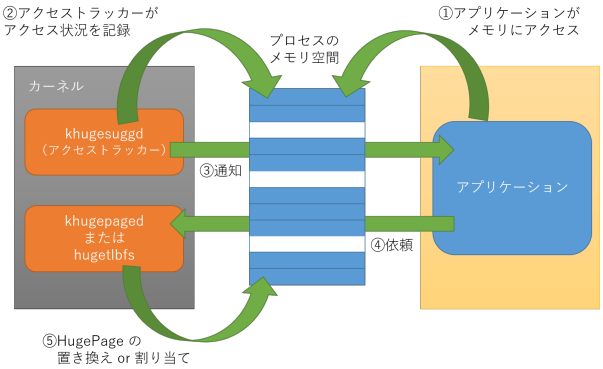

図 6 提案機構の概要

を procfs に追加した．アプリケーションには，通知内容に応じて HugePage に置き換える機能と， アイテムをコンパクションする機能を実装した．また，本提案手法をアプリケーションに対して容易に適用できるようにするため，ライブラリを作成し API を提供する．

5.1 Linux カーネルへの実装

5.1.1 ページアクセスの追跡

プロセスによる各ページへのアクセス追跡機能は，アクセスを検知して数え上げるカーネルスレッドと，それを確認するための機構から成り立つ．前者は 4.1.1 小節で示したアクセストラッカーによって実現する．後者は， procfs にカウント数を読み出す機能を追加して実現する．

5.1.1.1 アクセス数のカウント

アクセストラッカーと呼称するカーネルスレッドの動作について述べる．アクセストラッカーは図 7 に示す動作を繰り返す．4.3 節で述べたように，本手法では HugePage の置き換えに THP を使用している．そのため， THP の設定を madvise に設定しておくことが必要である．また，アクセストラッカーでは監視対象のプロセスを管理するために track_pages_tasks 構造体を扱う．この構造体には PID や通知を行う際の閾値，また通知を行うページ群がリストで繋がれている．

（1）THP の設定を確認し，“never” であれば “madvise” に変更して khugepaged を起動する
（2）track_pages_tasks 構造体のリストが空である場合は，追加されるまで動作を休止する
（3）track_pages_tasks 構造体のリストを追い，監視対象である全プロセスのページアクセスをカウントする
（4）一定時間動作を休止する

図 7 アクセストラッカー

```
00400000-00403000 r-xp 00000000 08:01 28588786
    00400000-0040800 : 0x00000121
    0040800-00402000 : 0x00000121
    00402000-00403000 : 0x0000002d

00602000-00603000 r--p 00002000 08:01 28588786
    00602000-00603000 : 0x00000004

00603000-00604000 rw-p 00003000 08:01 28588786
    00603000-00604000 : 0x00000025
```

図 8 page_acs 擬似ファイル

5.1.1.2 procfs を介したアクセス数の確認

アクセストラッカーによって加算された acscount の値を，シグナルによる通知を介さなくてもユーザによる確認を可能にする機構を用意している．この機能は， procfs にプロセスごとの擬似ファイル page_acs を追加することで提供される．procfs はプロセスに関わるカーネル情報にアクセスするために提供されている仮想ファイルシステムである．

擬似ファイル /proc/PID/page_acs を読むと，図 8 のような情報を得ることができる．ページの仮想アドレスと合わせて， acscount の値を表示している．

5.1.2 プロセスへの通知

プロセスに対して HugePage の利用を促すには，シグナルによる通知とシステムコールによる情報の取得を組み合わせて実現する．アクセストラッカーは，ページへのアクセス数が閾値を越したことを検出すると，シグナルを用いてプロセスにアクティブページの数の通知を行う．シグナルを受け取ったプロセスは，システムコールを発行してアクティブページのアドレスの取得を行う．

5.2 アプリケーションが利用する API とライブラリ

アプリケーションへ本提案手法を容易に適用できるようにするため，統一的に利用できる API を提供する．提案手法に関連する関数群を共有ライブラリにまとめ，アプリケーションはこのライブラリをロードすることで API を利用する．

このライブラリは，アプリケーションがカーネルと行うやりとりを肩代わりする役割を担っている．図 9 に本ライブラリのカーネル，アプリケーションとの関係，表 1 にライブラリが提供する API とその機能を示す．本節では，

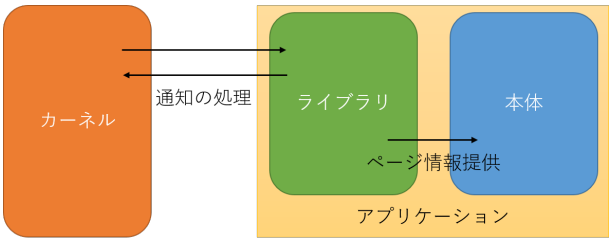

図 9 ライブラリの関係

表 1 API 一覧

API	機能
hps_enable_track	トラッキングを有効に設定
hps_disable_track	トラッキングを無効に設定
hps_set_functions	ユーザ定義の関数を設定
hps_wait_signal	シグナルが来るまで待機

API についてそれらの処理と実装について述べる．

5.2.1 通知シグナルへの対処

プロセスがシグナルを処理する方法は 2 通り存在する．1 つ目は，シグナルハンドラを設定しシグナル受信時に指定した関数を実行する方法である．2 つ目は，シグナルが配送されるまでプロセスを待機させ，受信後に以降の処理を継続させる方法である．

本ライブラリでは，その両者ともに対応するように実装を行う．シグナルハンドラを利用するのであれば，hps_enable_track() 関数で sigaction() 関数を呼び，ハンドラの設定を行う．シグナルを受信するまでプロセスを待機させるのであれば， 別スレッド内で hps_wait_signal() 関数を呼ぶ．この関数内では， sigtimedwait() 関数および sigwaitinfo() 関数でシグナルの待機を行う．しかしながら，この実装では処理を行っている最中に再びシグナルが飛んでくる状況も考えられる．その際にデフォルトの動作（無視）を行わないように，予め sigprocmask() 関数でマスクを行っておく．

ライブラリ内には sigpagesz_handler() 関数が存在しており，いずれの方法でもこの関数が呼ばれるようになっている．この関数がカーネルからアクティブページの情報取得の役割を担う．

5.2.2 アプリケーションと連携した HugePage の利用

カーネルから取得したアクティブページの情報を基に，HugePage への置き換えやオブジェクトのコンパクションを行う．まず，得られたアクティブページ群から，置き換えをするべきかコンパクションをするべきかの判断を行う．この判断は， 2MiB （最大で 512 ページ）ごとに行われる．デフォルトで判断を行うための閾値を定めているが，アプリケーションにその判断を仰ぐことも可能である．HugePage への置き換えを行うのであれば， madvise() を発行し， THP を利用する．コンパクションを行うのであれば，アプリケーション本体にその処理を行わせる．コンパクションをライブラリ内で行わない理由として，ライブラリからアプリケーションのデータ構造を把握することは不可能であることが挙げられる．図 9 のようにアプリケーション本体に情報を提供し，本体の中でコンパクションの処理をする．

6. 実験

6.1 実験環境

本提案手法の評価を，表 2 と表 3 に示すコンピュータで行う．いずれの環境でも OS は Ubuntu 16.04.1 LTS ，カーネルは提案手法を実装した Linux 4.7.10 である．

HP ProLiant DL980 G7 は NUMA マシン [9] であり，8 つのノードから構成される．各ノードに 10 個のコアと約 128GB のメモリが載っている．すなわち，合計で 80 コア，1TB のメモリを搭載している．Dell PowerEdge T80 II は 8 コア，16GB のメモリを搭載している．

6.2 節のアクセストラッカーのオーバーヘッド評価は実験環境 1 で実験を行う．6.3 節のマイクロベンチマークによる評価と 6.4 節の memcached による評価は実験環境 2 で実験を行う．

6.2 アクセストラッカーのコスト評価

アクセストラッカーが一つのプロセス空間全体を走査するのにかかる時間を計測する．本手法では，一つのプロセスにマッピングされている実ページすべてに対してページウォークを行う．したがって，割り当てられているページ数が多くなるほど，この処理にかかる時間は長くなると考えられる．本実験では，ページ数増加に伴う処理のコストの増加を確認する．

6.2.1 実験方法

一つのプロセス空間全体の走査をアクセストラッカーが行うのに要した CPU 時間を計測する．対象アプリケーションとして，巨大なサイズのメモリを確保するものを作成し，そこに提案手法を実装する．5 回ずつ計測を行い，その平均を算出する．

表 2 実験環境 1

機器名	HP ProLiant DL980 G7
CPU	Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz
キャッシュ	30720KB/コア
コア数	80
RAM	809909MB

表 3 実験環境 2

機器名	Dell PowerEdge T80 II
CPU	Intel(R) Xeon(R) CPU E31270 @ 3.40GHz
キャッシュ	8192KB/コア
コア数	8
RAM	15980MB

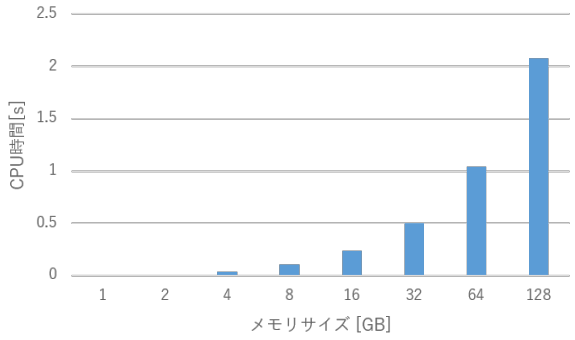

図 10 アクセストラッカーの処理時間

確保するメモリサイズを 1GiB から 128GiB に変化させ，それぞれの条件のもとで実験を行う．実験に使用する NUMA マシンにおいて，メモリの配置ポリシーは interleave とした．また，計測は全ページに対して実ページの割り当てが済んでいる状態で行う．すなわち，確保したメモリに対して 4KiB 単位で全ページに一度アクセスの後，トラッカーを有効にする．

6.2.2 実験結果

アクセストラッカーの処理時間を図 10 に示す．この結果から，プロセスが確保したメモリの合計サイズの増加に伴い，アクセストラッカーの処理時間も線形的に増加していくことが確認できる．プロセス空間に割り当てられたページの合計サイズが 1GB であれば 37ms，2GB であればその 2 倍弱の 72ms であった．128GB までサイズを拡大すると，処理にかかる時間は 2 秒を超えてしまう．

アクセストラッカーの動作自体がシステムに対して大きなコストになってしまう場合がある．アクセストラッカーが一つのプロセスのメモリ空間を走査している間は，そのCPU はコンテキストスイッチを起こさないようになっている．今回の実験に用いたサーバのようなコアが多数ある環境では，一つのコアを長時間占有しても他のプロセスに対してそれほど大きな影響は及ぼさない．しかしながら，コア数が少ない環境では残りのコアでしか他のプロセスが動作できなくなってしまい，システムの動作が遅くなってしまうと考えられる．より短時間で走査を終えられるようにアクセストラッカーを最適化することが今後の課題となる．

6.3 マイクロベンチマークによる評価

実装した HugePage 割り当て機構の評価を行うために，簡易的なベンチマークアプリケーションを実行する．提案手法と既存手法でスループットと実メモリ使用量を比較し，目的が達成できているか確認する．

6.3.1 実験方法

マイクロベンチマークプログラムで，メモリに対する読み書きのスループットを計測する．アクセス方法は，確保したメモリ空間の全範囲に対するシーケンシャルアクセス，全範囲に対するランダムアクセス，一部範囲に偏ったランダムアクセス（以下，スキューアクセス）の 3 つとする．一部範囲に偏ったシーケンシャルアクセスを行わない理由は，このアクセス方法は確保するメモリサイズを小さくした場合の全範囲シーケンシャルアクセスと同等であるからである．スキューアクセスは，全体の 3 分の 1 程度の範囲に限定してアクセスを行う．今回の実験では，確保するメモリサイズは 1GiB とする．

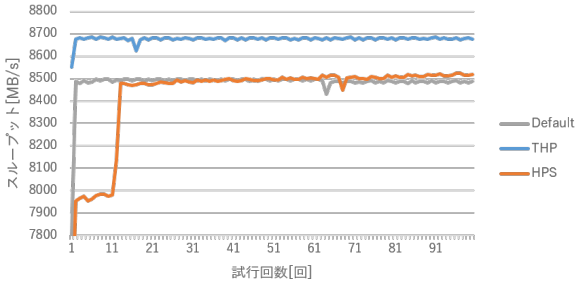

図 11 シーケンシャルアクセス時のスループット

スループットの計測は，次の通りに行う．読み書きの単位は 4KiB のページごととする．この 4KiB 単位のアクセスを指定された範囲に対して行い，これを 1 セットとする．16 セット読み書きを行い，それに要した時間を測る．1 セットの読み書きの中で，2 回以上同じページに対してアクセスすることはしない．アクセスした範囲のメモリサイズをかかった時間で割ることで，1 秒当たりの読み書き速度を算出する．この一連の動作を 80 回繰り返し，1 回毎のスループットの変化を記録する．

実メモリ使用量の計測は ps コマンドを使用する．5 秒ごとにベンチマークプログラムに割り当てられている実メモリ使用量の変化を監視する．

比較対象は HugePage を使用しないデフォルト状態 (Default)，THP を always に設定して常に HugePage を使う既存手法 (THP)，それと提案手法 (HPS) である．

6.3.2 実験結果

6.3.2.1 スループット

マイクロベンチマークを実行した際の，シーケンシャルアクセス時，ランダムアクセス時，スキューアクセス時のスループット結果をそれぞれ図 11，図 12，図 13 に示す．初回のアクセス時は実ページの割り当てが起きるため，いずれの条件でもスループットは低い．2 回目以降のアクセス時は初回時よりアクセス速度が上昇し，デフォルトでも既存手法である THP(always) でも安定したスループットを保っている．比較すると，常に HugePage を使用している既存手法のスループットが最も高く，デフォルトはそれより低いという結果になっている．

提案手法である HPS の結果を見ると，測定を始めてしばらくの間はスループットは非常に低い．しかしながら，

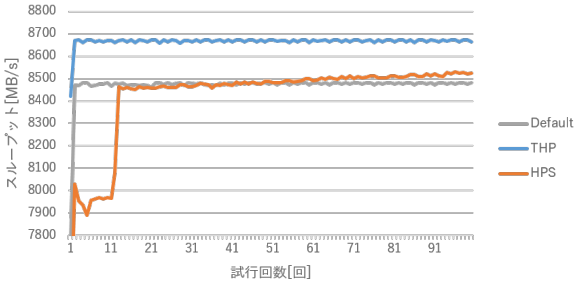

図 12 ランダムアクセス時のスループット

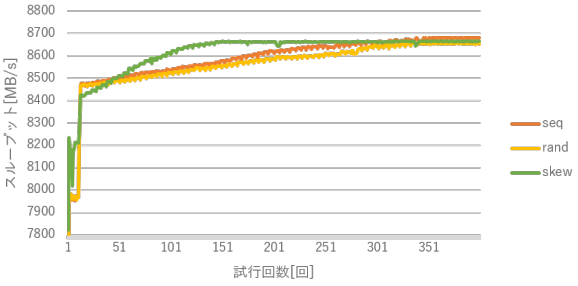

図 14 提案手法のスループット比較

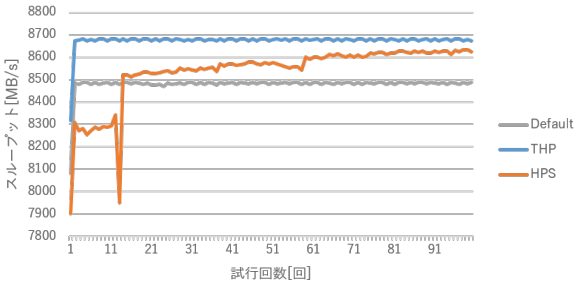

図 13 スキューアクセス時のスループット

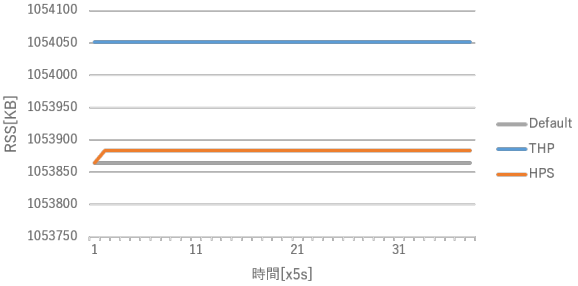

図 15 シーケンシャルアクセス時の実メモリ使用量

しばらくするとデフォルトと同程度まで回復し，その後スループットが徐々に上昇している．初期の低いスループットの原因として，アクセストラッカーが挙げられる．4.1.1 小節で述べたように，アクセストラッカーの動作には TLB エントリのフラッシュが伴う．そのため，結果としてアプリケーションでの TLB ミスが多発してしまう．アクセス速度がデフォルトと同程度まで回復したタイミングは，ある程度のコンパクションが済んだと判断してトラッカーを停止した時点と考えられる．

この後は徐々にスループットが上昇しているが，それを顕著に示しているのがスキューアクセスの結果である．スキューアクセスにおいて，提案手法を用いた場合ではデフォルトに対して 1.8% ほどスループットが改善している．しかしながら，シーケンシャルアクセスとランダムアクセスで提案手法を用いた場合では，最大でも 0.5% 程度の改善しかしていない．このよう結果になった原因として，HugePage への置き換えに利用した THP(madvise) の機構が考えられる．THP(madvise) のこの機能を実現しているカーネルスレッド khugepaged は，VM_HUGEPAGE フラグが立ったページを一括して HugePage に置き換えるわけではない．ある程度の時間をかけて順に置き換えが行われていく．したがって，参照範囲が狭いワークロードではそのアイテム全体が HugePage に載るのが他よりも早かったためであると考えられる．

6.3.2.2 スループット追加実験

先の小々節で示した結果から，提案手法を用いたいずれアクセス方法のスループットも，いまだ上昇中にあることがわかる．そこで，これ以降のスループットの変化を得るため，計測を 400 回行った．その結果を図 14 に示す．シーケンシャルアクセス，ランダムアクセス，スキューアクセスをそれぞれ seq，rand，skew とラベルをつけている．

先ほどの結果と同様，試行回数が 80 回程度の時点でスキューアクセスのスループットは既存手法と同程度まで達している．シーケンシャルアクセスとランダムアクセスは，だいたい 300 回を超えたあたりで同程度のスループットとなっている．

6.3.2.3 実メモリ使用量

マイクロベンチマークを実行した際の，シーケンシャルアクセス時，ランダムアクセス時，スキューアクセス時の実メモリ使用量をそれぞれ図 15，図 16，図 17 に示す．図 18 は図 17 と同じ結果のグラフであるが，デフォルトと提案手法のみの結果を示している．既存手法は常にデフォルトより多くのメモリを使用している．既存手法では，アプリケーションがメモリを使用しているかどうかに関わらず HugePage を割り当てるので，スキューアクセスでも他のアクセス方法と同等のメモリ量の割り当てが行われている．デフォルトでは，それとは対照的に実際に利用している分のメモリしか割り当てない．そのため，デフォルトのスキューアクセスでは他と比べて 40% 程度という低いメモリ使用量となっている．

既存手法，提案手法のそれぞれで，デフォルトと比べてどれほどメモリ使用量が増加したのか調べる．シーケンシャルアクセスとランダムアクセスでの増加量は，既存手法を用いた場合は約 0.017% ，提案手法では 0.002% である．スキューアクセスでは，既存手法を用いた場合は 147%，提案手法では 0.031% ほどのメモリ使用量が増加した．い

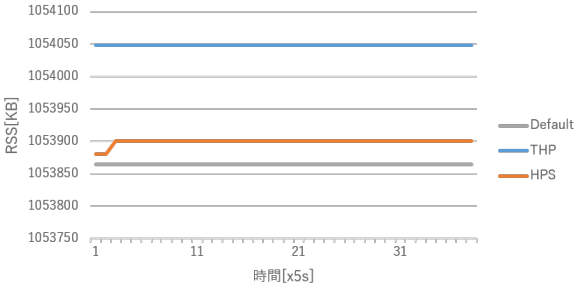

図 16 ランダムアクセス時の実メモリ使用量

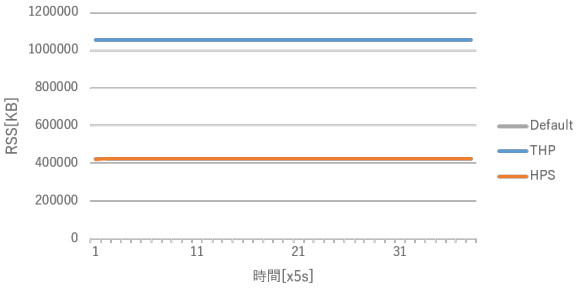

図 17 スキューアクセス時の実メモリ使用量 1

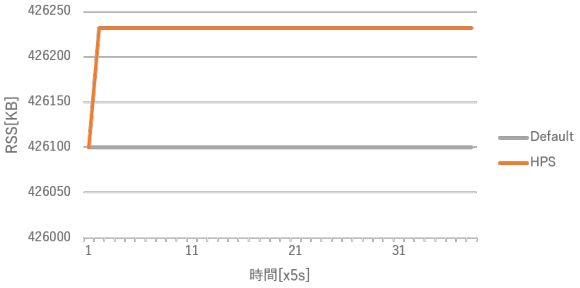

図 18 スキューアクセス時の実メモリ使用量 2

ずれの結果からも，提案手法は既存手法よりもメモリ使用量の増加を抑止できた．スキューアクセスの場合，それがより顕著に現れている．

6.4 memcached による評価

本手法を実装した memcached に対して，ベンチマーカを用いてスループットを測定する．ベンチマークプログラムには Yahoo! Cloud Serving Benchmark (YCSB) [10][11] を使用する．

6.4.1 実験方法

memcached から 1 秒当たりに取得できるアイテム数を計測する．YCSB では memcached に対してはじめにアイテムの設定を行い，その後ベンチマークの測定を行う．YCSB で実行するワークロードは表 4 に示すとおりである．ベンチマークプログラムを 10 回実行し，そのスループットの平均値を算出する．それと同時に，memcached の実メモリ使用量の変化の記録も行う．比較対象は 6.3 節同様に，デフォルト (Default)・既存手法 (THP)・提案手法 (HPS) の 3 つとする．

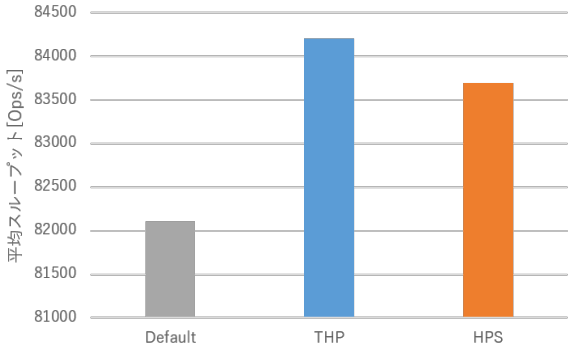

図 19 memcached のスループット

表 4 memcached にかけるワークロード

データサイズ	512 byte
設定するアイテム数	800000
取得するアイテム数	800000
アクセス種別割合	read:1
アクセス分布	zipfian（一部に偏る）

6.4.2 実験結果

提案手法を実装した memcached に対してワークロードをかけた際のスループットを図 19 に示す．それぞれの手法を用いて 10 回計測した際の平均値である．この実験においても既存手法が最も優れたスループットを示しており，6.3.2 小節でのマイクロベンチマークを用いた実験結果と傾向は一致する．既存手法のスループットがデフォルトに対して 2.55% 改善しているのに対し，提案手法を用いた場合では 1.94% 改善した．この実験において，提案手法は既存手法の 99.4% 程度のスループットを達成している．

また，同時に計測した実メモリ使用量を図 20 に示す．この結果には，memcached にワークロードをかけた直後からのメモリ使用量の変化が記録されている．既存手法を用いた場合では，デフォルトを基準として 96.2% 多いメモリをはじめから使用している．すなわち，デフォルトに対して 2 倍弱の実ページを使用しているということである．それに対して提案手法では，はじめはデフォルトとほとんど変わらないメモリ使用量を保っている．途中で HugePage への置き換え，ないしはコンパクションが行われたため増加しているが，その量は 6.17% 程度である．したがって，提案手法は既存手法に比べて，実メモリ使用量のデフォルトからの増加量を 93.6% 程度削減できた．

7. おわりに

本研究では，アプリケーションレベルでのメモリ管理を考慮しながら，効率的に HugePage を割り当てるための手法を提案した．OS カーネルによってプロセス空間を監視し，アクセスが集中しているページをアプリケーションに

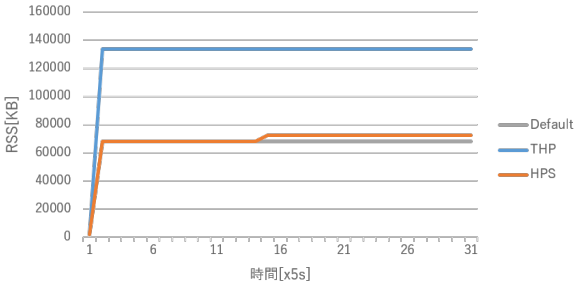

図 20 memcached の実メモリ使用量

通知することで，参照される頻度の高いオブジェクトのみを HugePage 上に配置することが可能となる．提案手法を memcached に実装して性能を測り，既存手法と比較したところ，実メモリ使用量のデフォルトからの増加量を 93.6% 程度削減できた．その上，既存手法の約 99.4% のスループットを達成した．

参考文献

[1] I. E. Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W. mei Hwu, T. Roscoe, and K. Schwan. SpaceJMP : Programming with Multiple Virtual Address Spaces. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 353–368, 2016.

[2] Collin McCurdy, Alan L. Cox, and Jeffrey Vetter. Investigating the TLB behavior of high-end scientific applications on commodity microprocessors. *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 95–104, 2008.

[3] Inc. the Linux Kernel Organization. Huge Pages.

[4] Inc. the Linux Kernel Organization. Transparent Hugepage Support.

[5] Alan Cox Juan Navarro, Sitaram Iyer, Peter Druschel. Practical, transparent operating system support for superpages. *Operating Systems Design and Implementiontion (OSDI)*, pp. 89–104, 2002.

[6] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 705–721, 2016.

[7] Mel Gorman and Patrick Healy. Supporting Superpage Allocation without Additional Hardware Support. *7th International Symposium on Memory Management*, pp. 41–50, 2008.

[8] Dormando. memcached - a distributed memory object caching system. `http://www.memcached.org/`.

[9] Jinsu Park, Myeonggyun Han, and Woongki Baek. Quantifying the performance impact of large pages on in-memory big-data workloads. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016*, pp. 209–218, 2016.

[10] brianfrankcooper. Yahoo! Cloud System Benchmark (YCSB). `https://github.com/brianfrankcooper/YCSB`.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, pp. 143–154, 2010.