Vol.2018-OS-142 No.7
2018/2/27

NUMA上のDockerコンテナスケジューリングにおけるメモリマイグレーションの改善

漆田 瑞樹[1,a)]　廣津 登志夫[1,b)]

概要：近年，多くの企業では保守性の観点からマイクロサービスを適用している．これにより分割された各サービスのリソースはDockerなどのコンテナ技術により管理されている．こういった環境ではパフォーマンス向上の観点からNUMAのサーバーが使われているため，その利点を活かすようなコンテナ配置が必要である．最適な配置は負荷に応じて変わるためマイグレーションを行うことが望ましい．しかし，現在cgroupに実装されているマイグレーション機能はCPUコアを1つ占有するため全体のパフォーマンス低下を招いてしまう．これを防ぐためにマイグレーションをスロットリングして実行したい．また，一般的にCPUとメモリはペアで切り替えるため，スロットリングをするとメモリアクセスレイテンシがローカルアクセスレベルに回復するまでに大きな遅延が発生してしまう．そこで本研究ではスロットリングをしつつメモリアクセスレイテンシの回復を可能な限り高速化する手法を提案する．

Improvement of Memory Migration in Docker container scheduling on NUMA systems

Mizuki Urushida[1,a)]　Toshio Hirotsu[1,b)]

1. 序論

近年，マイクロサービスを適用する企業が増えている．マイクロサービスでは単一の大きなサービスを機能ごとに分割し，モジュール化して運用する．これにより，各機能ごとに動作させることができるため保守性が向上する．その一方で各モジュールはプロセスとして独立しているため，特定のモジュールが一方的にCPUを使用するなどサーバーリソースの競合が生じることがある．そこで従来は仮想マシン(VM)を利用することによってセキュリティの強化とともに各モジュールのリソースを管理していた．しかしハードウェア資源の仮想化オーバーヘッドの大きさや，ブートストラップによる起動時間の遅さの面からVMではなくコンテナが利用されるようになった[1]．コンテナはLinuxのcgroup[2]とnamespace[3]を用いてホストOS上で隔離環境を実現しており，プロセスがホストOSの管理下となるためオーバーヘッドが限りなく小さい．中でも各種サービスをデプロイすることに特化したDocker[4]が広く使われている．

一方，大規模サービスを運用するような高性能サーバーでは，Non Uniform Memory Access(NUMA)と呼ばれるマルチCPU環境を使用することが多い．NUMAでは各CPUソケットにメモリを直結して構成されており，このCPUとメモリの組をNUMAノードと呼ぶ．この環境では負荷に応じて適したNUMAノードにコンテナをマイグレーションすることでより良いパフォーマンスが得られる．DockerにおけるNUMAノード間のマイグレーションは，各コンテナのリソースを管理しているcgroupの提供する機能により実現できるが，現状の実装ではこのマイグレーションが1つのCPUコアを占有して行われるため，その間はシステム上で稼働している他のサービスのパフォーマンスが低下してしまう．本研究ではそのマイグレーションに対して，一定量のページを移送してスリープするという

[1] 法政大学
Hosei Uniersity
[a)] m.urushida@dsl.k.hosei.ac.jp
[b)] hirosu@hosei.ac.jp

スロットリング手法を適用することでこの問題を解決する．また，このようなスロットリング手法のみではメモリアクセスレイテンシの回復が遅くなってしまうため，更にこのレイテンシを効率的に回復させるような手法を提案する．

2. 関連研究

NUMA は Symmetric Multiprocessing(SMP) システムの構成法の一つで，Intel のアーキテクチャではメモリを直接接続した CPU 同士が Intel QuickPath Interconnect(QPI) で接続されている．そのため，各 CPU が自身に接続されたメモリへアクセスすることでシステム全体のメモリバス幅を拡大できる．その一方で他のメモリへのアクセスレイテンシは大きくなってしまう．このレイテンシを本研究環境で実際に計測したものが**表 1** である．これを見ると，同一 NUMA ノードの CPU メモリ間の通信に比べて，異なる NUMA ノードの通信は約 1.5 倍遅くなっている．そのため，NUMA 上ではプロセスの使用するメモリと CPU は同一の NUMA ノードのものを使うことが望ましい．

表 1 ローカルアクセスとリモートアクセスのレイテンシ (ns)

		Memory	
NUMA Node		0	1
CPU	0	74.8	121.7
	1	119.2	72.7

この NUMA に対して，負荷に応じて VM の使用するメモリと CPU を最適に切り替えるスケジューリングの研究がかねてより行われている．関連研究 [5] ではメモリと I/O デバイスが異なる NUMA ノードに位置するときにも遅くなることから，I/O デバイスとの通信に重点を置いた VM のスケジューリングアルゴリズムを提案している．I/O デバイスとの通信はホスト OS 上の I/O スレッドによりハンドリングされ，その後に VM に届けられる．この手法ではその一連の通信ができるだけ QPI を通らないように，低負荷時には I/O デバイスが接続されている NUMA ノード上に VM と I/O スレッドを集約する．また，高負荷時には各 NUMA ノードに I/O スレッドを作成して VM を全体に分散させる．このような処置はコンテナに対しても有効である．そこで本研究ではスケジューリング時に発生するメモリマイグレーションの改善を行う．

3. Docker

Docker はコンテナを提供するミドルウェアである．コンテナでは内部のプロセスがホストのカーネル上で動作し，ホスト側からは自身のプロセスとして管理できるためオーバーヘッドが非常に小さい．また，Docker では単一のアプリケーションを単一のコンテナで動作させる設計をしているため軽量である．他に有名なコンテナミドルウェアとして LXD[6] が挙げられるが，こちらは init プロセスを起動して単一の仮想マシンとして動かすことが多い．そのため起動時のオーバーヘッドが小さい Docker がマイクロサービスによく使われている．

多くのコンテナ技術では物理リソース管理に cgroup という Linux の機能を用いている．これにより CPU，メモリ，ディスク I/O といった資源の利用について制御することが可能となる．cgroup では「cgroup」と呼ばれるプロセスグループ単位でリソース制御を行っており，ルートの「cgroup」から分岐するツリー状のヒエラルキーとなっている．そして子の「cgroup」は親の「cgroup」の制御情報を引き継ぐ．また，制御するリソースの種類はサブシステムと呼ばれる単位で分けられている．これらを組み合わせることで特定のプロセスに対するリソースを細かく制御することができる．

NUMA ノードの切り替えには cpuset サブシステムを利用する．このサブシステムを利用することで NUMA ノード間のコンテナマイグレーションを実現できる．これにより，そのときのリクエストに最適化されたキャッシュを持つコンテナなどを削除せずに別の NUMA ノードに移動することができる．cgroup はユーザー空間に展開された sysfs から操作でき，cpuset は`/sys/fs/cgroup/cpuset` から操作できる．Docker コンテナの cpuset 用 cgroup は `/sys/fs/cgroup/cpuset/docker/[container_id]` に生成される．**表 2** にその cpuset を操作する上で主要な設定ファイルを示す．

表 2 cpuset の設定

設定	概要
`cgroup.procs`	この cgroup に属すプロセス ID
`cpuset.cpus`	使用可能な CPU コア ID
`cpuset.mems`	使用可能なメモリノード ID
`cpuset.memory_migrate`	`cpuset.mems` 変更時のメモリマイグレーションの有無

マイグレーションの例を**図 1** に示す．この例では cgroupA と，その子である cgroupB，cgroupC がある．cgroupA に属すプロセスのメモリは制限されずに NUMA ノード 0 とノード 1 の両方を使用することができ，cgroupB は NUMA ノード 0 のみ，cgroupC は NUMA ノード 1 のみに使用制限がされている．CPU に関しては制限されておらず，加えてメモリの切替時には同時にマイグレーションを行うように設定されている．各プロセスの仮想メモリはページテーブルを介して物理メモリ上にマッピングされている．このとき，cgroupA のメモリノード ID を 1 にすると下記の流れでマイグレーションが行われる．

(1) A のメモリノード ID を 1 に設定する
(2) PID100 の NUMA ノード 1 へのマイグレーション
(3) B のメモリノード ID が 1 に書き換わる
(4) PID300, 400 の NUMA ノード 1 へのマイグレーション

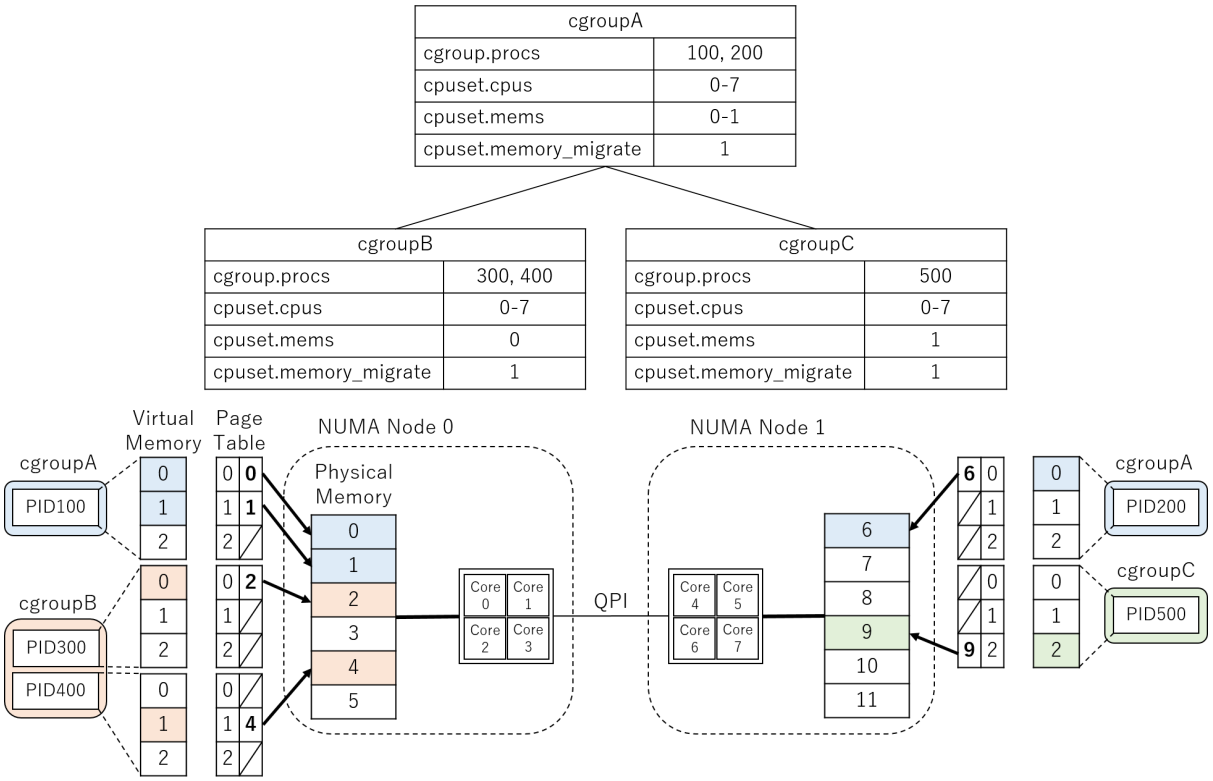


図 1 cpuset の cgroup ヒエラルキーの例

(5) C はすでに NUMA ノード 1 なので何も行われない

4. 既存手法

現在提供されている cgroup 上に実装されているマイグレーションでは，一度に全てのページを移送するため CPU コアの占有や QPI への負荷といった点で問題がある．ここではその既存のメモリマイグレーションの仕組みを説明しその問題点について述べる．

4.1 マイグレーションの流れ

メモリの切替と同時にマイグレーションを行う設定になっていると，メモリノード ID の変更に合わせて対象となるプロセスのマイグレーションタスクがワークキューに追加される．これらのタスクはワークキューに紐付いたカーネルスレッドによって 1 つずつ順に実行される．タスクの中の処理は大きく分けてページリスト生成フェーズと，対象ページを移送するマイグレーションフェーズの 2 段階で構成されている．ページリスト生成フェーズではプロセスのメモリ全体に対してページウォークを行い，宛先ノード以外に属するページをマイグレーション対象としてリストに加える．そして，マイグレーションフェーズでは与えられたページリストを全て宛先ノードにマイグレーションする．

ページウォークとはリニアな仮想アドレスを基にページテーブルを参照して，対応する物理アドレスを求めることを指す [7]．プロセスのメモリは複数の vm_area_struct 構造体 (以下 vma) と呼ばれるメモリリージョンから構成されており，**図 2** の例では vma1 の先頭にある仮想アドレスのページウォークを示している．Linux のページング環境はページテーブルエントリの容量を節約するために多段ページテーブルを採用している．多段ページテーブルでは，CR3 レジスタが指しているページグローバルディレクトリ (PGD) からページテーブルエントリ (PTE) までを，仮想アドレスのビット列を基に辿っていく．PTE はページフレーム番号 (PFN) を保持しており，これと仮想アドレス下位 12 ビットのページ内オフセットを合わせることで物理アドレスを求めることができる．

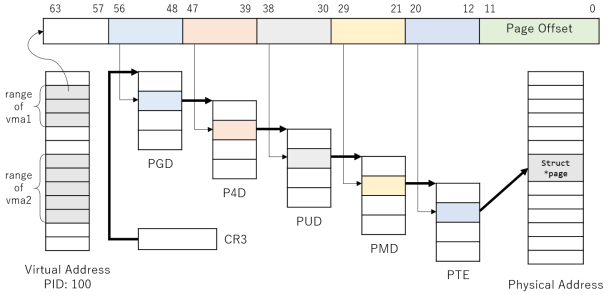


図 2 ページウォークの流れ

4.2 メモリマイグレーションの問題点

マイグレーションフェーズではページリスト内の全てのページを移送する．そのため，カーネルスレッドがこ

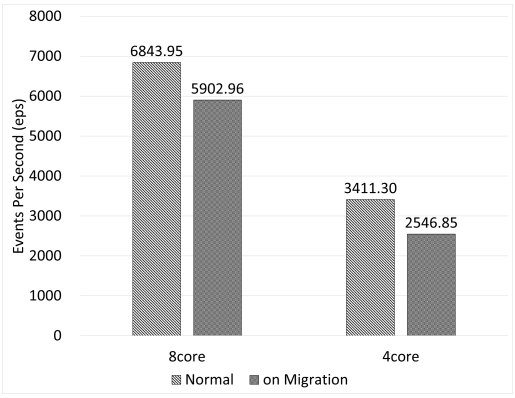


図 3 平常時とマイグレーション時の処理性能

の処理を終えるまで CPU コアを 1 つ占有してしまう．そこで，予備実験を行いマイグレーション処理が他のプロセスに与える影響を測定した．ここではベンチマークをとるタスクを 8 コア及び 4 コアを用いて実行させると同時に，8GB のプロセスのメモリマイグレーションを発生させた．実験環境には Supermicro X8DAH+-F(デュアルソケット)，Intel Xeon E5530(4cores)，メモリ 24GB 上で稼働する Linux 4.13.10 を用い，ベンチマークは sysbench 1.0.6[8] を使用した．sysbench で定められた一連の計算を 1 イベントと数えた際に，1 秒間に処理されたイベント数を平常時とマイグレーション時に分けて**図 3** に示す．これによると 8 コアの場合は，マイグレーション時に 1/8 の性能低下が見られ，4 コアの場合は 1/4 の性能低下が見られる．このようにコアを 1 つ占有するため，ホスト上の使用可能な CPU の数が少ないほど影響が大きくなってしまう．

次に，マイグレーション時に QPI の負荷によるネットワークへの影響について**図 4** の環境で測定した．この測定では iperf 3.1.7[9] を用い，パケット生成ノードに iperf クライアントを稼働させ，NUMA ノード上で iperf サーバーを次の (1)-(3) の条件で動作させた．

(1) node0: NUMA ノード 0 上で動作
(2) node1: NUMA ノード 1 上で動作
(3) node1(mig): NUMA ノード 1 上で動作 (バックグラウンドで無関係な他プロセスをマイグレーション)

1Gbps の UDP 通信を 10,000 秒間行ったときに 1 秒毎に 1 サンプル (約 82,000 パケット) としてパケットロス率を求めて，パケットロス率毎のヒストグラムにしたものを**図 5** に示す．**図 5a** がグラフの全体であり，**図 5b** が 0 から 500 サンプルの領域を拡大したものである．このグラフの node1 を見るとわかるように，QPI 負荷がなければ QPI を経由した通信にほとんどパケットロスは発生しない．しかし，QPI に負荷をかけると node1(mig) のように一気にパケットロスが生じる．本環境でのマイグレーション速度は最大で 1.8GB/s であり，QPI バンド幅は 9.8GB であるため，2 割程度の負荷でもパケットロスが発生することがわかる．

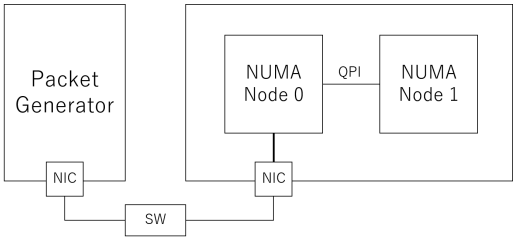
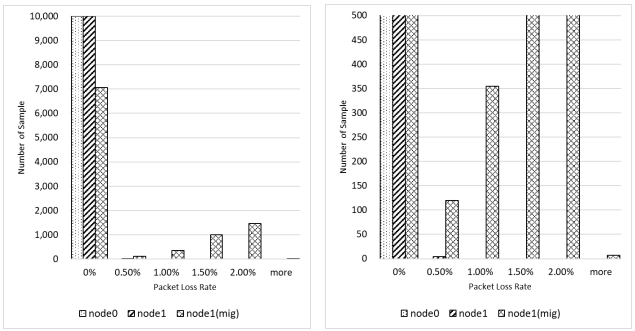


図 4 ネットワーク測定実験環境



(a) 全体　　(b) 0-500 サンプル領域の拡大図

図 5 マイグレーション時のパケットロス率

5. スロットリング手法の設計

既存のメモリマイグレーションのようにコアを占有しない手法を設計する．ここでは CPU 使用率を抑制するためのスロットリングとメモリアクセスレイテンシの回復を高速化する手法を提案する．

5.1 スロットリング

スロットリングによる CPU 使用率の抑制のためには，マイグレーションを処理するカーネルスレッドを定期的に休止させれば良い．ここではその制御のために 2 つのパラメータ n と t を定義する．n は一度にマイグレーションするページ数であり，t はマイグレーション後のスリープ時間 (ms) である．n ページマイグレーションして t スリープする処理の繰り返しにおいて，n を小さくすることによって一度のマイグレーションで使用する CPU 時間を減少させることができる．そしてマイグレーション処理のたびにスリープすることによって CPU 使用率を抑制する．

5.2 レイテンシ回復の高速化

メモリマイグレーションはコンテナの CPU を変更した際にメモリアクセスレイテンシを減らすために行う．これまでの実装では一気に全てのメモリを移送するため，移送後にはレイテンシは小さくなる．一方，提案手法ではスロットリングをするため，移送が完了するまでの間メモリアクセスレイテンシの大きな状態が発生することになる．この現象を避けるために，将来的にアクセスが発生する可能性が高いページを優先的にコピーする．将来アクセスさ

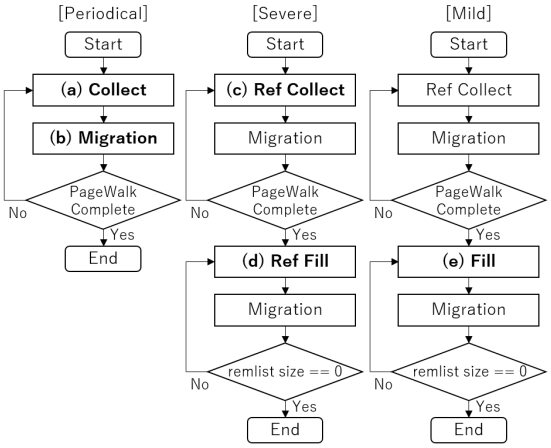


図 6 マイグレーションの処理フロー

れる可能性が高いページの判定には，最近アクセスされたページのフラグを利用する．これは参照の局所性より，最近アクセスされたページは近い将来アクセスされる可能性が高いという傾向を考慮したものである．このようなページを Referenced ページと呼ぶことにする．

5.3 スロットリング手法の提案

ここでは 3 つの手法，Periodical, Severe, Mild を提案する．まず，Periodical は 5.1 節で説明したスロットリングのみを行う．次に，Severe はスロットリングにより周期的に実行されるたびに，5.2 節で説明したレイテンシ回復の高速化のための Referenced チェックを行う．しかしこれはチェックに要するオーバーヘッドが大きいため，Mild ではそのチェックを初回のページウォーク時にのみ行うようにする．ここではページの回収先リストとして，移送優先度の高いページを格納する pagelist と，優先度の低いページを格納する remlist を定義している．これらの手法の詳細なマイグレーションフローについて**図 6** に示す．

Periodical 手法ではページウォーク時に (a) 全てのページを pagelist に回収する．そして，この pagelist サイズが n に達する度にページウォークを切り上げて (b) マイグレーションを行う．この後 pagelist の中身は空になる．マイグレーションにはスリープが組み込まれており，移送後には t ms スリープする．スリープ復帰後は切り上げたときの仮想アドレスからページウォークを再開する．これを繰り返し，ページウォークが終了した時点でマイグレーションが完了する．このようにページ量とスリープ時間を設定することでスロットリングができる．

Severe 手法ではページウォーク時に (c)Referenced ページを pagelist に，それ以外を remlist に回収する．そして，pagelist サイズが n に達する度にマイグレーションしてページウォークが完了するまで繰り返す．これによりレイテンシの回復が高速化される．次に，ページウォーク後にアクセスされたページを検出するために，(d)remlist を全探索して Referenced ページを見つけ次第 pagelist に移動する．この pagelist が n に達したらマイグレーションを行う．n に満たない場合は remlist の先頭から不足分を pagelist に移動する．これを繰り返し，remlist が空になった時点でマイグレーションが完了する．この手法には Referenced ページを抽出するためのオーバーヘッドが付随する．プロセスの総ページ数を N とすると，フラグチェック回数は初項 N，末項 0，公差 $-n$ の等差数列の和となるため，$N^2/2n + N/2$ となり，計算量は $O(N^2/n)$ となる．

Mild 手法ではそのオーバーヘッドを抑えるように設計している．ページウォークが完了するまでは Severe と同様の手順を行うが，(e) 残った remlist のチェックは行わずに n ページずつマイグレーションする．こうすることでチェックはページウォーク時にのみ行われて計算量は $O(N)$ となる．この方法ではページウォークが行われた後にアクセスされたページには対応できない．しかし，そもそもページウォークは n ページずつ切り上げて余分なチェックを行わないため，チェックの直前までアクセスされたページを追跡できる．そのため，最初だけ Referenced チェックを行うだけでも十分なレイテンシの回復が見込める．

6. 実装

5 章にて提案したスロットリング手法を Linux Kernel 4.13.10 上で実装した．

6.1 スロットリング有効化インターフェースの追加

本手法を実際に利用するために，現在の cgroup にマイグレーションタスクをスロットリングするフラグを管理するインターフェースを追加する．内部で管理しているフラグに `CS_LAZY_MIGRATE` を追加し，`cpuset.lazy_migrate` というインターフェースファイルに 1 という数値が書き込まれるとそのフラグを立てるように変更した．また，タスクのコア実装は cgroup ではなく，カーネルのメモリ管理の実装であるためその中にも `MPOL_MF_LAZY_MOVE_ALL` というフラグを追加した．これらのフラグによりタスクがスロットリングして実行される．

6.2 マイグレーションページの選定

Referenced ページは page 構造体の `PG_referenced` フラグと，PTE の `Accessed` フラグにより判定することができる．前者はプロセスがメモリにアクセスした際に OS がマークするフラグであり，後者は PTE に対応するページフレームへのアドレッシング時にページング回路 (ハードウェア) がマークするフラグである．この両者の内一方が立っているときに Referenced ページだと判断する．

pagelist は `list_head` という循環リストを用いて page 構造体を格納する．一方で remlist は `pte_node` という独自に定義した単方向連結リストで管理する．`pte_node` はペー

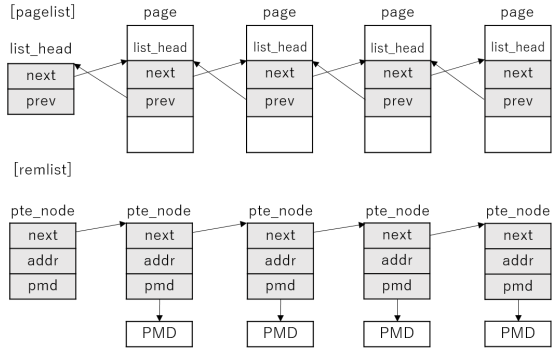


図 7 ページウォーク後のページリスト

ジが属す PMD エントリへのポインタ pmd と，仮想アドレス空間内でのページの先頭アドレス addr を保持している．pte_node を用いることで，addr から pmd 内のオフセットを計算して目的の PTE を求めることができる．page 構造体から PTE を引くのは逆マッピングという仕組みを利用して探索が可能だが，メモリディスクリプタの照合とページウォークが必要になるため pte_node を使用する方が高速である．また，PTE ではなく PMD で管理しているのは，Referenced チェック時に行う PTE ロックがし易いからである．PTE はロックの機構を持っておらず，ロックをする際にはその PTE が属すページテーブルの PFN をロックする必要がある．そのため，その PFN を保持している PMD で管理している．ページウォークが完了すると**図 7** のように，マイグレーション対象ページが pagelist と remlist に振り分けられる．

6.3 スロットリング機構

cpuset.mems が変更されるとその cgroup に属すプロセスのマイグレーションタスクが起動される．このとき 6.1 章で追加した cpuset.lazy_migrate が有効化されているとマイグレーションをスロットリングして実行する．ソースコード 1 にこのスロットリングの簡易的な実装を示す．

この関数 throttling_migration() は 5.3 で示したようにページウォーク開始からマイグレーション完了までの一連の処理を行う．引数としてマイグレーションを行う対象プロセスのメモリディスクリプタ (mm_struct) と，マイグレーション元 (src) と先 (dst) のビット列と，マイグレーションパラメータである移送ページ量 (limit) とスリープ時間 (sleep) を引数として受け取り，9 行目までに pagelist や remlist など各種変数の初期化処理を行う．

まず，11-12 行目について説明する．11 行目で評価される pw_contd という変数はページウォークの継続を表しており，true に初期化されている．そのため，まずページウォークを行う関数 queue_pages_range() が実行される．この関数では Referenced ページとそれ以外のページを pagelist と remlist に分けて回収する．Periodical の場合は全て pagelist に連結される．pagelist が limit に達してページウォークを切り上げた際は，start にその位置の仮想アドレスを記録することで次にそのアドレスから再開できるようにする．ページウォークが最後の仮想アドレス end に到達すると pw_contd に false が返される．

次に 18-22 行目について説明する．ここでは pagelist にページが入っていることを確認して migrate_pages() という関数を実行している．これは既存の関数であり，受け取った pagelist をマイグレーション先である dst に移送するものである．この後に，21 行目にて sleep で指定した時間分カーネルスレッドを停止させる．そしてスリープから復帰すると contd という真偽値をチェックして true なら 11 行目にループする．この変数はマイグレーションの継続を表しており，true に初期化されている．

ページウォークが終了すると pw_contd は false となり，ページウォークの代わりに 15 行目が実行される．move_pages() という関数は remlist から pagelist に limit 分移動する処理を行う．Severe では Referenced ページ優先で移動され，Mild では無差別に移動される．remlist が空になると contd に false が返される．以上の流れによりマイグレーションが完了する．

ソースコード 1 スロットリングを行うコード

```
1 void throttling_migration(struct mm_struct *mm,
2       int src, int dst, int limit, int sleep)
3   LIST_HEAD(pagelist);
4   LIST_HEAD(remlist);
5   bool pw_contd, contd;
6   unsigned long start = mm->mmap->vm_start;
7   unsigned long end = mm->task_size;
8
9   pw_contd = contd = true;
10  do {
11    if (pw_contd)
12      pw_contd = queue_pages_range(&start, end,
13        mm, src, &pagelist, &remlist, limit);
14    else
15      contd = move_pages(&pagelist,
16        &remlist, limit);
17
18    if (!list_empty(&pagelist))
19      migrate_pages(&pagelist, dst);
20
21    msleep_interruptible(sleep);
22  } while (contd);
23 }
```

7. 評価

提案した 3 つのスロットリング手法に対してオーバーヘッドとレイテンシ回復速度の 2 つの観点で評価実験を行った．また，QPI の負荷によるパケットロスといった従

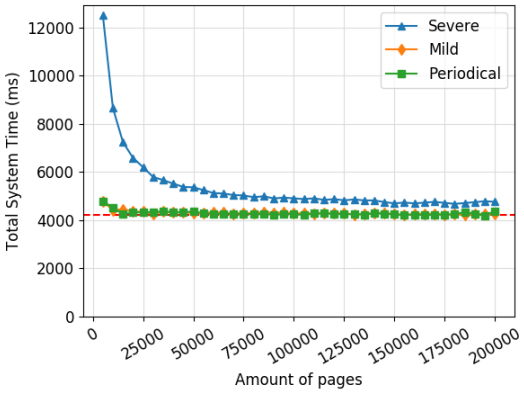


図 8 ページ量による CPU 時間

来のマイグレーションにおける問題点が提案手法により改善されたことを併せて示す．この章での実験環境は 4.2 章と同様のものである．

7.1 マイグレーションオーバーヘッド

まず，各手法を実現するためのオーバーヘッドを測定した．ここでは 8GB のプロセスをマイグレーションして，完了までに消費された CPU 時間を計測する．スロットリングパラメータのページ数 n を $5000 \leq n \leq 200000$ の範囲で行ったときの CPU 時間を**図 8** に示す．赤い点線は既存手法の CPU 時間 4200ms を表す．余計な処理がない既存手法の CPU 時間は最小になるため，この値との差が小さいほど良い．グラフを見ると，Severe はフラグチェック計算量が多いためオーバーヘッドが大きくなっており，Periodical や Mild は remlist のチェックがないためオーバーヘッドが小さい．Mild はページウォーク時にチェックを行うが 1 回のオーバーヘッドが小さいため差が現れなかった．

7.2 メモリアクセスレイテンシの回復速度

次に，メモリアクセスレイテンシの回復速度を測定した．この実験ではマイグレーションされているプロセスのレイテンシがローカルアクセスレベルに回復するまでの経過を見る．このプロセスはページに対してスパースにアクセスして単純計算を行うものであり，使用するメモリ量 8GB に対してその半分の 4GB にのみにアクセスする．1 回の計算にかかった時間をレイテンシと定義したときの結果を**図 9** に示す．赤い点線はリモート，青い点線はローカルの平均アクセスレイテンシを表す．提案した 3 手法はいずれも約 36 秒でマイグレーションが完了する．グラフを見ると Periodical のレイテンシがローカルレベルに回復するまでに約 36 秒かかるのに対して，Severe と Mild は Referenced ページを優先するため，その半分の約 18 秒でレイテンシが回復することがわかる．

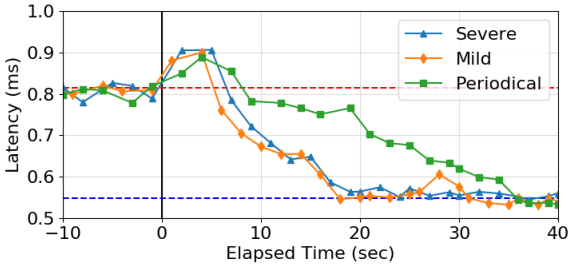
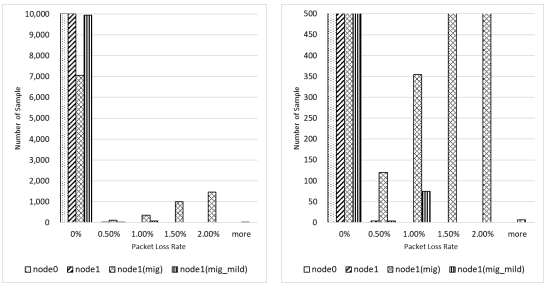


図 9 メモリアクセスレイテンシの回復速度



(a) 全体　　(b) 0-500 の区間

図 10 QPI 負荷を抑圧したときのパケットロス率

7.3 スロットリング効果

スロットリングにより QPI の負荷による影響を抑制できることを示す．ここでは 4.2 節と同様に図 4 の環境において，4.2 節 (1)-(3) に加えて次の (4) を提案手法として測定した．

(4) node1(mig_mild): NUMA ノード 1 上で動作 (バックグラウンドで無関係な他プロセスを Mild 手法でマイグレーション)

ここで 1 回にマイグレーションするページ数 n を 45,000(約 180MB)，スリープ時間 t を 1,000ms としている．このページ数は本環境のプロセッサが 1 秒間にマイグレーションできるページ数の 1 割であり，これにより QPI の負荷を従来の約 10 分の 1 程度に抑えるものである．結果を**図 10** に示す．これによると，従来のマイグレーション手法である node1(mig) では約 3,000 サンプルのロスがあるが，10 分の 1 にスロットリングした node1(mig_mild) ではロス率は約 80 サンプルと 2.5%程度に抑制された．スロットリングを行うと 10 倍の時間が掛かるが，ロス率が 2.5%に抑制されていることから，総合的に見てもパケットロス率が低下していることがわかる．

8. 考察

実験結果より，オーバーヘッドに関しては Mild と Periodical では既存手法とほとんど変わらないが，Severe ではページ数 n が小さくなるほどオーバーヘッドは大きくなった．Mild と Periodical のオーバーヘッドが変わらないことから，計算量 $O(N)$ のフラグチェックにはそれほど計算リソースが必要が無いことがわかった．その一方で計算

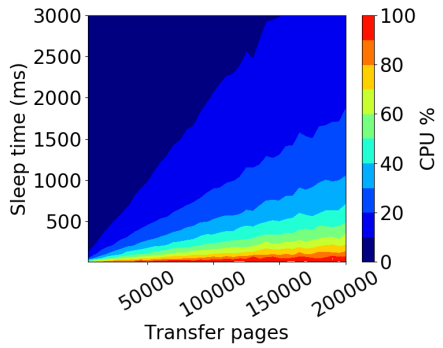


図 11 Mild 手法の CPU 使用率ヒートマップ

量 $O(N^2/n)$ の Severe のオーバーヘッドが大きくなってしまっていることから，計算量 $O(N)$ は小さいが無視できるほどのオーバーヘッドではないと考えられる．また，メモリアクセスレイテンシに関しては Periodical がマイグレーション完了までレイテンシが回復しないのに対し，Mild と Severe ではその半分の時間でレイテンシが回復した．これは Referenced ページを優先することにより，計算に使われているページが先に移送されたからである．これは実験で使用したプロセスが均等にスパースでアクセスしたため，このようにチェックするものとしないもので差が大きく現れたが，実際にはプロセスのアクセスする位置によって多少差が縮まると考えられる．

以上のオーバーヘッドとレイテンシ回復の両方の観点から，オーバーヘッドは同等であるのに対してレイテンシ回復は Mild が高速であることから Periodical に比べて Mild が良好な結果であると考えられる．Severe に関してはレイテンシ回復の高速化は実現できたが，オーバーヘッドが非常に大きいため今後改良の余地があると思われる．

このスロットリング手法を実際に適用する際には，スロットリングパラメータを適切に設定する必要がある．設定の指標は各パラメータごとの CPU 使用率を各自の環境で計測することにより得られる．例として一度に移送するページ数 n ($5000 \leq n \leq 200000$) とスリープ時間 t ($10 \leq t \leq 3000$)ms の組でマイグレーションを本環境で行った．このときの CPU 使用率をヒートマップとして **図 11** に示す．この指標によりマイグレーションタスクに対して許容する CPU 使用率を設定することが可能となる．

9. 結論

本研究では，コンテナのマイグレーションの際に発生する CPU 占有を抑制するために，メモリ移送をスロットリングする手法を提案した．ここでは 1 回の移送ページ数と移送間のスリープ時間を設定して単純にスロットリングするだけでなく，移送対象のページを選別することによりメモリアクセスレイテンシの回復を高速化する機構も実現した．この選別では最近アクセスしたページを先に移送する．実験結果から，提案手法により CPU コア占有を抑圧しつつレイテンシ回復の高速化が実現されることを示した．

謝辞 本研究は JSPS 科研費 JP15K00138 の助成を受けたものです．

参考文献

[1] Kang, H., Le, M. and Tao, S.: Container and microservice driven design for cloud infrastructure devops, *Cloud Engineering (IC2E), 2016 IEEE International Conference on*, IEEE, pp. 202–211 (2016).
[2] Menage, P.: CGROUPS, , available from ⟨https://www.kernel.org/doc/Documentation/cgroup-v1⟩ (accessed 2018-02-04).
[3] Biederman, E. W. and Networx, L.: Multiple instances of the global linux namespaces, *Proceedings of the Linux Symposium*, Vol. 1, Citeseer, pp. 101–112 (2006).
[4] Docker Inc.: Docker - Build, Ship, and Run Any App, Anywhere, , available from ⟨https://www.docker.com⟩ (accessed 2018-01-22).
[5] Banerjee, A., Mehta, R. and Shen, Z.: NUMA Aware I/O in Virtualized Systems, *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, IEEE, pp. 10–17 (2015).
[6] Canonical Ltd.: Linux Containers - LXD - Introduction, , available from ⟨https://linuxcontainers.org/lxd⟩ (accessed 2018-01-24).
[7] Gandhi, J., Basu, A., Hill, M. D. and Swift, M. M.: Efficient memory virtualization: Reducing dimensionality of nested page walks, *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, pp. 178–189 (2014).
[8] Kopytov, A.: akopytov/sysbench: Scriptable database and system performance benchmark, , available from ⟨https://github.com/akopytov/sysbench⟩ (accessed 2018-02-04).
[9] Jon Dugan, Seth Elliott, B. A. M. J. P. K. P.: iPerf - The ultimate speed test tool for TCP, UDP and SCTP, , available from ⟨https://iperf.fr/⟩ (accessed 2018-02-04).