輪講資料 2009 年 11 月 27 日

クラウドコンピューティングに対するスレッドマイグレーション技術の適用
Application of Thread Migration Techniques to Cloud Computing

情報理工学系研究科 電子情報学専攻 近山・田浦研究室 修士 1 年 48096419 原健太朗

Abstract

With the increase of parallel and distributed applications which require many computational resources, the importance of Cloud Computing is rising, in which a user can use only as many computational resources as needed when needed in a pay-as-you-go system. Although Cloud Computing services have been featured in many ways, they must at least meet the following two common requirements; firstly they must support flexible scale-up/scale-down in response to dynamic load fluctuation; secondly they must schedule shared computational resources between multiple users (according to their policies). In this paper, I analyze two representative Cloud Computing services, Amazon EC2 and Google App Engine, focusing on how each service achieves these two requirements. On the basis of these analyses I propose a thread migration-based model as an intermediate approach between Amazon EC2 and Google App Engine. Moreover, as elemental techniques for achieving the thread migration-based model efficiently, I survey techniques for kernel thread migration and fast memory migration.

1 序論

1.1 背景

近年，SNS やオンラインゲームなどの Web アプリケーションや，遺伝子解析や地震シミュレーションなどの高性能数値計算アプリケーションなどを始めとして，多数の計算資源を要求する並列分散アプリケーションが増加している．従来，企業や大学などの組織がこのようなアプリケーションを実行しようと思えば，その組織が自前でデータセンタを構築する必要があった．すなわち，従来のコンピューティング形態では，各組織が，サーバやネットワーク機器などのインフラストラクチャや OS などのプラットフォーム，そしてその上で動作する各種ソフトウェアなどを備えたデータセンタを構築し，それらを日常的に管理する必要があった．

しかし，このようなデータセンタ “所有型” のコンピューティング形態にはいくつかの欠点がある．第一の欠点として，データセンタの管理コストが大きい．各組織の目的は，あくまでもアプリケーションを実行することであるため，複雑で専門的な管理技術を要するデータセンタの管理は回避したい作業である．第二の欠点として，データセンタを構築する段階では適切なサーバ台数を見積もることが難しい．当然，データセンタを構築する上では設置するサーバ台数を決定しなければならないが，過小に見積もれば高負荷に耐えられないし，過大に見積もれば投資が無駄になってしまう．また，負荷を監視しつつサーバ台数を増強するとしても，サーバの発注・設置・ソフトウェア設定など多くの作業が必要になるため，負荷が高くなったからと言ってすぐに増強できるものではない．第三の欠点として，固定のサーバ台数で運用されるデータセンタでは動的な負荷変動に効率的に対応できない．データセンタのサーバ利用率は，平均で 5%〜20% であり，ピーク時にはその 2 倍〜10 倍の負荷が加わると言われている [15]．つまり，固定のサーバ台数で運用されるデータセンタでは，普段は処理能力が余剰になっている一方で高負荷時には処理能力が不足する事態が起きやすく，動的な負荷変動を効率的に吸収することができない．

1.2 クラウドコンピューティングとは

以上のような現状を背景として，近年注目を浴びてきているコンピューティング形態がクラウドコンピューティング [15, 14, 8] である．クラウドコンピューティングでは，クラウドプロバイダと呼ばれる組織が大規模なデータセンタを構築し，インフラストラクチャ，プラットフォーム，ソフトウェアなどを整備して，それらをサービスとして利用者に提供する．そして利用者は，それらのサービスを必要なときに必要な量だけ利用することができ，実際に利用した量だけ課金される．つまり，従来のコンピューティング形態が “所有型” なのに対して，クラウドコンピューティングは “利用型” のコンピューティング形態である．

クラウドコンピューティングが提供するサービスは，IaaS (Inflastructure as a Service)，PaaS (Platform as a Service)，SaaS（Software as a Service）の 3 つに大きく分類できる．IaaS は，仮想サーバやストレージなどの計算機資源を提供するサービスであり，後述する Amazon EC2 や Amazon S3[1] などが代表例である．PaaS は，利用者が作成したアプリケーションの実行環境を提供するサービスであり，後述する Google App Engine[2]，Windows Azure[5]，Salesforce.com[4] などが代表例である．PaaS では，各サービスごとに実行可能なアプリケーション領域がある程度限定されている．SaaS は，クラウド環境上で実現されるソフトウェアサービスであり，利用者が

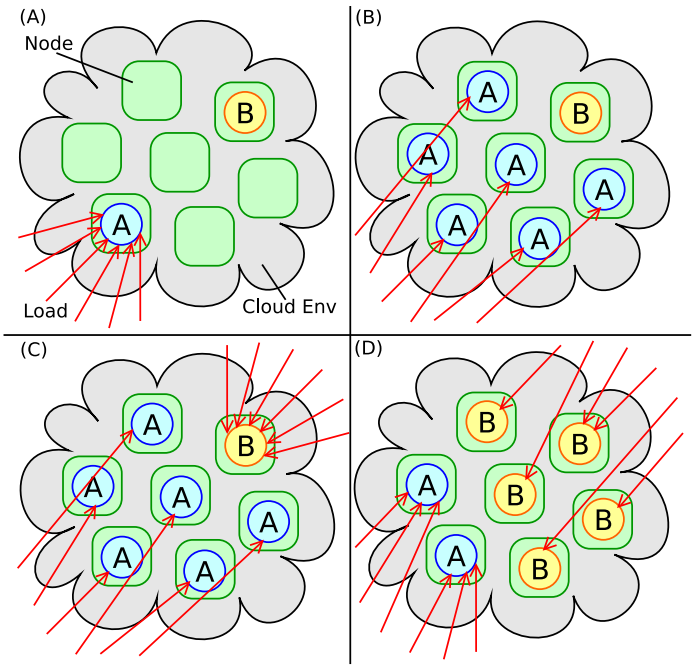

Fig.1 An example of computational scale-up/scale-down in Cloud Computing.

ブラウザ経由で利用する Web サービスが中心である．たとえば，Google Docs[3] や Salesforce.com の CRM などが代表例である．これら “利用型” のクラウドコンピューティングサービスでは，利用者はサービスが内部的にどう実現されているかを意識する必要がないため，煩雑なサーバ管理技術などは不要である．また，従量制課金モデルの中で必要なときに必要な量だけ利用できるため，動的な負荷変動に効率的に対応できる．

具体例を示す．あるクラウド環境の中に利用者 A と利用者 B がいるとし，今利用者 A の負荷が増大したとする（Fig.1(A)）．すると，利用者 A のアプリケーションの計算規模が拡張し負荷が分散される（Fig.1(B)）．やがて，利用者 B の負荷が利用者 A の負荷より増大したとすると（Fig.1(C)），今度は利用者 A のアプリケーションが縮小する代わりに利用者 B のアプリケーションが拡張して負荷分散が図られる（Fig.1(D)），というような変化がたとえば起きる．当然，どういう条件が成立したときにどの利用者の計算規模を拡張/縮小させるかは各サービスに依存するが，いずれにせよ，資源を利用者間で適宜スケジューリングしつつ，負荷に応じて計算規模を拡張/縮小することで負荷分散を図るというのがクラウドコンピューティングにおける基本的な仕組みである．ここで重要なのは，多数の利用者で多数の資源を共有することにより各利用者の負荷変動が上手に吸収されるという点である．たとえば，月曜日に負荷が高い利用者と火曜日に負荷が高い利用者と水曜日に負荷が高い利用者と木曜日に負荷が高い利用者を 1 個のクラウド環境に詰め込むことにより，各利用者が自前でデータセンタを所有する場合と比較して，資源の利用率を高く維持しつつ，全体として少ない資源数で負荷変動を吸収できる．要するに，10 人で 10 台のサーバを持つより，10000 人で 10000 台のサーバを持つ方が良いというのが，クラウドコンピューティングの基本的な考え方である [18]．

クラウドコンピューティングという単語は依然バズワードであると言われ，学術的に捉えるか商用的に捉えるかでもその定義が変わり，事実，多様な実現形態が存在する [15, 14]．しかし，以上の観察に基づくと，最低限の共通の要請として，

- 負荷の増減に応じて柔軟に計算規模を拡張/縮小できること
- （何らかのポリシーに基づいて）共有資源を利用者間でスケジューリングできること

の 2 点は，クラウドコンピューティングサービスを実現する上で必須と言える．そこで本稿では，この 2 つの要請をどう実現するかに着眼しつつ，既存のクラウドコンピューティングサービスを分析すると同時に，新たにスレッドマイグレーション型モデルを提案し，それに適用できる要素技術について検討していく．

1.3 本稿の構成

第 2 節では，上記の 2 つの要請に着眼しつつ，Amazon EC2 および Google App Engine について分析し，その分析に基づいてスレッドマイグレーション型モデルによるクラウドコンピューティングサービスを提案する．第 3 節では，スレッドマイグレーション型モデルにとって必要なカーネルスレッドマイグレーションに関する技術を紹介し，第 4 節では，高速なメモリマイグレーションの技術を紹介する．

2 クラウドコンピューティングサービス

2.1 動機付け

本節では，Amazon EC2，Google App Engine，スレッドマイグレーション型モデルの各クラウドコンピューティングサービスに関して，

- 何を単位として計算規模の拡張/縮小を実現しているか
- 利用者間で資源をどのようにスケジューリングしているか

に着眼してそれぞれの特徴を分析する．後者に関しては，Fig.1(C) に示す状況が発生した場合，つまり，すでに利用者 A が多くの資源を利用している状況で利用者 B への負荷が増大した場合に，利用者 B のアプリケーションをどう拡張すれば良いかというシナリオを具体例にして考える．

2.2 Amazon EC2

Amazon EC2 では VM（Virtual Machine）が計算規模の拡張/縮小の単位であり，利用者は，必要なときに VM を起動/停止することで計算規模の拡張/縮小を実現する．Amazon EC2 の利点は，VM という汎用的な計算機資源が提供されるため，利用者にとって自由度が大きく，実行可能なアプリケーション領域が広いという点である．特に，多くの高性能数値計算アプリケーションのような，長時間を要するアプリケーションも実行できる．一方で，第一の欠点として，VM は存在しているだけで無視できない量の資源を消費するため，課金が CPU の使用時間などではなく VM の起動時間に基づいて行われる．

よって，たとえば Web アプリケーションの場合，クライアントからのリクエストが存在しない限り CPU は消費しないが，リクエストを待ち受けるために VM を起動しているだけで課金されてしまう．第二の欠点として，VM の起動/停止には数分を要するため，負荷変動に対して高速に対応しづらい．これも，VM の資源消費量が大きいことに起因している．

次に，Amazon EC2 が Fig.1(C) に示す状況にどう対処しているかを考える．Amazon EC2 では，各利用者が起動できる VM 数がデフォルトでは 20 個に制限されており，それ以上の VM を使用するには詳細な申請を提出しなければならないというルールがある．つまり，Amazon EC2 では，一時的に急激な高負荷状態に陥ったとしても，それに対応して VM 数を急増させることができない．このように Amazon EC2 は，負荷変動に対する VM 数の急激な変化を抑制するためのルールを敷くことで，クラウド環境上に起動される VM 数の変化を緩やかなものにし，利用者全体の VM 数を常に大まかに把握できるようにしている．そして，利用者全体の VM 数が把握できれば，できるだけ資源を枯渇させないような利用者への VM 割り当てを行えるため，そもそも Fig.1(C) に示すような，利用者 B のアプリケーションを拡張しようにも拡張できない事態には陥りにくい．要するに，Amazon EC2 では，負荷変動に対する高速な適応性を犠牲にして，できる限りクラウド環境内の資源数を枯渇させないような資源のスケジューリングを実現している．

2.3 Google App Engine

Google App Engine は，利用者が Java もしくは Python で記述した Web アプリケーションを，Google の効率的なインフラストラクチャ上で実行させるためのプラットフォームを提供する．Google App Engine における計算規模の拡張/縮小の単位は，Web アプリケーションに対するクライアントからのリクエストであり，リクエストの増減に応じて，Web アプリケーションが実行される資源数が自動的に拡張/縮小され，利用者が何の意識を払わずとも負荷分散が図られる．2009 年 11 月時点における無料コースでは，1 分間に最大 7400 個ものリクエストが処理可能とされている．よって，第一の利点として，負荷変動に対して非常に高速にしかも自動的に適応できる点が挙げられる．第二の利点として，実際に使用した CPU 時間やネットワークバンド幅に基づいた “真の” 課金が行われる．すなわち，VM を起動しておくだけで課金される Amazon EC2 とは異なり，Google App Engine では実際にリクエストが届いて資源が利用されない限り課金されない．これは，Google App Engine における計算資源の拡張/縮小の単位が，存在するだけで資源を消費する VM ではなく，存在しなければ資源を消費しないリクエストであるという点に起因している．

次に，Google App Engine が Fig.1(C) に示す状況にどう対処しているかを考える．Google App Engine では，各リクエストは 30 秒以内に処理されなければならず，30 秒以上かかるリクエストは強制終了させられるというルールがある．このルールにより，Fig.1(C) の状況が起きたとしても，高々 30 秒待てば利用者 A のアプリケーションは消滅するため，利用者 B のアプリケーションを割り当てることが可能になる．要するに，Google App Engine は，各リクエストの処理時間を制限することで短時間単位の資源のスケジューリングを実現している．したがって，欠点として，Google App Engine では短時間のアプリケーションしか実行できない．たとえば，ソーティングや連立方程式ソルバなどの長時間を要する高性能数値計算アプリケーションを 30 秒区切りでプログラムすることは困難であり，Google App Engine 上で実行させることは難しい．また，効率的な計算規模の拡張/縮小を自動的に実現するために，プラットフォーム的にも典型的な Web アプリケーションに特化した作りになっている．

Tab.1 Characteristics comparison between three Cloud Computing Services （サービス名に関しては，EC2=Amazon EC2，スレッド型=スレッドマイグレーション型モデル，GAE=Google App Engine）.

サービス名	EC2	スレッド型	GAE
拡張/縮小の単位	VM	スレッド	リクエスト
資源消費量	大	中間	小
課金の粒度	粗粒度	中間	細粒度
負荷変動への適応性	低速	中間	高速
実行可能なアプリ領域	広い	中間	狭い
長時間アプリの実行	可能	可能	不可能

2.4 スレッドマイグレーション型モデル

以上で述べた Amazon EC2 および Google App Engine の特徴を Tab.1 にまとめる．Tab.1 より，Amazon EC2 と Google App Engine は，これらの特徴に関して対照的な性格を持つことが読み取れる．そこで，両者の中間的なアプローチ，つまり両者の利点を混合させるアプローチとして，私はスレッドマイグレーション型モデルを提案する．

スレッドマイグレーション型モデルでは，計算規模の拡張/縮小の単位としてスレッドを用いる．スレッドマイグレーション型モデルの第一の利点は，スレッドは VM よりも資源消費量が少ないため，Google App Engine のように実際の CPU 使用時間などに基づく細粒度な課金が可能な点である．また第二の利点として，資源消費量の少ないスレッドの起動/停止は VM の起動/停止より軽量なため，VM よりも負荷変動に対して高速に適応できる．さらに第三の利点として，スレッドマイグレーション型モデルでは，Google App Engine のように実行時間に制限を設けず，高性能数値計算アプリケーションのような長時間を要するアプリケーションも実行可能である．

問題は，このような利点を両立させつついかにして Fig.1(C) に示す状況に対処するかであるが，スレッドマイグレーション型モデルでは，実行中のスレッドをあるノードから別のノード

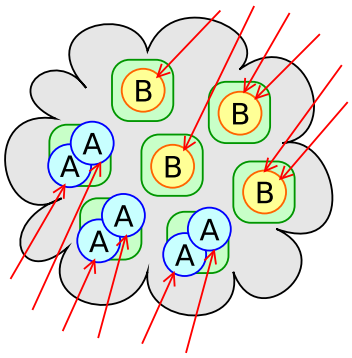

Fig.2　Resource scheduling in a thread migration-based model.

に移動させることで資源のスケジューリングを行う．たとえば，Fig.1(C) の場合，利用者 A の実行中のスレッドの何個かを別のノードに移動させ，空いたノードに利用者 B のアプリケーションを拡張することで資源のスケジューリングを実現する（Fig.2）．したがって，欠点として，1 個のノード上で多数のアプリケーションが実行される可能性があるため，実行中のアプリケーションの品質を保証できない．各アプリケーションの品質はクラウド環境全体の負荷状況によって決定される．すなわち，クラウド環境全体の負荷が低ければスレッドは多数のノードに効率的に分散されて実行されるが，クラウド環境全体の負荷が高ければ多数のスレッドが少数のノード上に詰め込まれて実行されることになる．

このように，スレッドマイグレーション型モデルには欠点もあるが，Amazon EC2 と Google App Engine の中間的存在として魅力的な利点を備えている．以降では，スレッドマイグレーション型モデルを実現するための要素技術として，カーネルスレッドマイグレーションと高速なメモリマイグレーションの手法について検討していく．

3　カーネルスレッドマイグレーション

3.1　ポインタ無効化の問題点と対処策

スレッドマイグレーション [6, 7, 12, 17, 10, 11] とは，実行中のスレッドをあるノードから別のノードに移動させることである．以降では特にカーネルスレッドを考え，各ノードでは同一プロセス内に複数のカーネルスレッドが実行されており，かつこれらのカーネルスレッドはメモリアクセスしか行わないことを仮定する．つまり，ファイルアクセスやネットワーク通信などは考慮しない．カーネルスレッドの実体は CPU レジスタとメモリ領域（スタック領域 + ヒープ領域）から構成されており，メモリ領域内には，メモリ領域内のアドレスを指し示すポインタが複数含まれている．

以上のような状況で，単純にノード S 上のカーネルスレッド T をノード D に移動させるとポインタ無効化の問題が起きる．特に何の対策も行わなければ，移動元のノード S でカーネルスレッド T が使用していたアドレス領域 a が，移動先のノード D 上のプロセスにおいて空いている保証はない．よって，一般には，ノード D では空いている適当なアドレス領域 b にカーネルスレッド T のメモリ領域を配置してカーネルスレッド T を復帰させることになるが，このときメモリ領域内に含まれるポインタはメモリ領域がアドレス領域 a に配置されていることを仮定した値になっているため，カーネルスレッド T は正しく動作しない [6, 7, 10, 11]．

このポインタ無効化の問題には大きく分けて 2 つの解決策が提案されている．第一の解決策は，移動先のノード D において，メモリ領域に含まれる全ポインタを，アドレス領域 b に合わせて完全に正しく更新する手法である．この手法を取る場合，ポインタは単なる整数値に過ぎないため，整数値とポインタを見分ける手段が必要となる．手法 [10] では，ポインタ登録用の関数を提供することで，どれがポインタなのかをプログラマに明示的に指示させるアプローチを取っている．しかし，複雑なプログラムにおいて全てのポインタをプログラマに漏れなく登録させるのは困難な上，各種ライブラリ呼び出しの内部などでプログラマからは暗黙的に作られてしまうポインタには対処できない．一方，手法 [11] は，コンパイルの段階においてソースコードレベルでポインタを検出して，実行バイナリにポインタ検査のための命令を仕込むアプローチを取っている．また，C 言語におけるポインタ型から整数値型へのキャスト，共用体内に含まれるポインタなど，ポインタと整数値の区別が難しいいくつかの場合に対する対処策も提示されている．しかし，この手法でも，ソースコードが与えられないライブラリ呼び出し中のポインタは検出できない．以上をまとめると，ポインタと整数値が本質的に同じものである以上，どうしても両者を区別できない場合が存在するため，移動先のノード D においてポインタを完全に正しく更新することは困難だと言える．

第二の解決策は，あるカーネルスレッドが使用しているアドレス領域が他のいかなるカーネルスレッドによっても使用されないことを保証することで，移動元と移動先で常に同一アドレス領域にカーネルスレッドのメモリ領域を配置できることを保証する手法である．これは最も単純には，各カーネルスレッドが使用するアドレス領域が重複しないように，各カーネルスレッドが使用するアドレス領域を静的に決め打つことで実現できるが，これではカーネルスレッドの動的な増減に対応できない．そこで，動的なメモリ領域確保に対しても，割り当てられるアドレス領域の一意性を保証する手法が必要になる．これは最も単純には，

(1) メモリ領域確保時に，空いているアドレス領域を全カーネルスレッドに問い合わせる．
(2) 全カーネルスレッドの空いているアドレス領域の共通部分を取り，そのアドレス領域にメモリを確保する．

という手順を踏むことで実現可能だが，メモリ確保の度に全ノードとの通信を行うのは明らかに非効率である．よって，次節では，これをより効率的に実現するアルゴリズムとして Iso-address [6, 7] を紹介する．

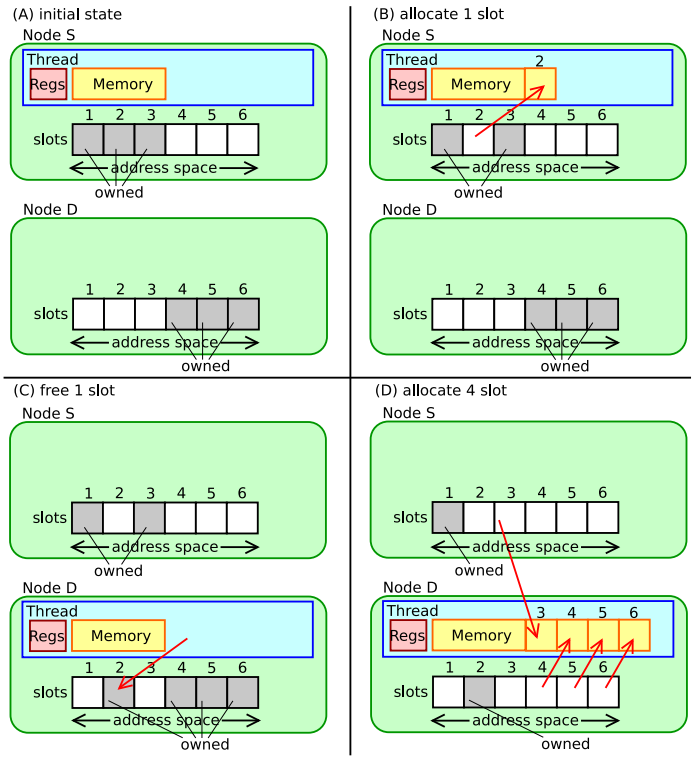

Fig.3　Memory allocation/deallocation based on Iso-address.

3.2　Iso-address

Iso-address では，アドレス領域全体をスロットと呼ばれる複数の小領域に分割し，初期的に，全スロットを全ノードに分散配置する．最適なスロットサイズはアプリケーション依存であるが，論文 [6] では 64KB に設定されている．以下では，アドレス領域を 384KB，スロットサイズを 64KB，ノード数を 2 個とし，この 6 スロットのうち，初期状態として，スロット 1〜3 をノード S に，スロット 4〜6 をノード D に配置した場合を例にして説明する（Fig.3(A)）．

第一に，カーネルスレッド T がメモリ領域を確保する際には，カーネルスレッド T が所属するノードのスロットを取得し，そのスロットが示すアドレス領域にメモリ領域を確保する．たとえば，ノード S 上のカーネルスレッド T が 50KB のメモリ領域を確保する際には，たとえばスロット 2 を取得してメモリ領域を割り当てる（Fig.3(B)）．この操作はノード内で完結するためノード間通信は生じない．第二に，カーネルスレッド T がノード S からノード D に移動する場合を考えると，スロットは初期的に全ノードに重複なく分散配置されているため，移動先でスロット 2 が使用されていないことが保証される．これにより，移動元と移動先で常に同一のアドレス領域にメモリ領域を割り当てられることが保証できるため，ポインタ無効化の問題が発生しない．第三に，メモリ領域を解放する際には，その時点でカーネルスレッド T が所属するノードに対してスロットの返却が行われる（Fig.3(C)）．この操作もノード内で完結するためノード間通信は生じない．このように，カーネルスレッドの移動に伴って，全ノードを通じたスロット配置が変化する．

しかし，以上の手順だけだと，ノード内に十分なスロットが存在しない場合にメモリ領域の確保が失敗するという問題が生じる．たとえば Fig.3(C) の状態では，カーネルスレッド T は連続する 200KB のメモリ領域を確保できない．このような場合には，ノード間通信を行い，適当なノードからスロットを奪ってくることで対処する（Fig.3(D)）．ただし，このようなノード間通信が多発するとアプリケーションの性能が劣化するため，スロットサイズや初期的なスロット配置を適切に選択することが重要になる．

4　高速なメモリマイグレーション

4.1　動機付け

前述の Iso-address などのカーネルスレッドマイグレーション技術を利用すれば，ひとまず以下の手順でノード S からノード D へカーネルスレッド T を移動できる：

(1) ノード S でカーネルスレッド T を停止する．
(2) CPU レジスタとメモリ領域をノード S からノード D に移動する．
(3) ノード D でカーネルスレッド T を復帰する．

しかし，この手順では，CPU レジスタとメモリ領域の転送時間がカーネルスレッド T の停止時間になるため，カーネルスレッド T が使用しているメモリ領域が大きい場合には停止時間が長くなってしまう．そこで，本節では，メモリ領域の転送を工夫することで停止時間を短縮化する手法として，Pre-copy[9] と Post-copy[16] のアプローチについて紹介する．要約すれば，Pre-copy はカーネルスレッド T の移動前にメモリ領域を移動させる手法であり，Post-copy はカーネルスレッド T の移動後にメモリ領域を移動させる手法である．

4.2　Pre-copy

4.2.1　アルゴリズム

Pre-copy は複数回のラウンドから構成される．まず，ラウンド 1 では，OS のページ単位でのメモリ保護機構を利用して，カーネルスレッド T のメモリ領域の全ページのアクセス保護属性をライトアクセス禁止に設定する．そして，カーネルスレッド T とは別のバックグラウンドプロセスを使って全メモリ領域をノード S からノード D に移動する．この間，カーネルスレッド T はメモリマイグレーションとは関係なくノード S で実行を継続できる．また，全ページをライトアクセス禁止に設定していることを利用して，カーネルスレッド T によってライトアクセスが行われたページを全て検出して記録しておく．バックグラウンドプロセスによる全メモリ領域のコピーが完了した時点でラウンド 1 が完了する．次に，ラウンド 2 では，再びカーネルスレッド T のメモリ領域の全ページのアクセス属性をライトアクセス禁止に設定した上で，バックグラウンドプロセスを使って，ラウンド 1 でライトアクセスが行われたページのみをノード S からノード D に移動する．つまり，

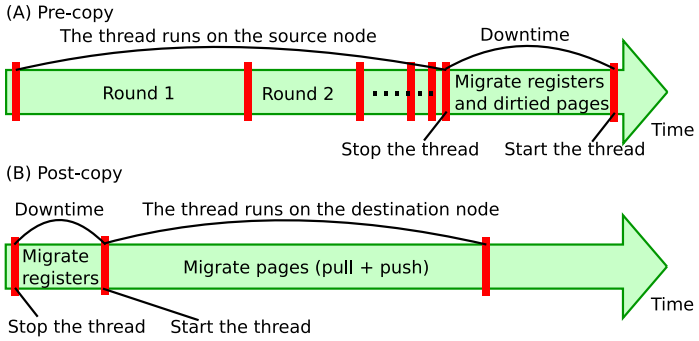

Fig.4　Timelines of Pre-copy and Post-copy.

ページの最新状態がまだノード D に伝えられていないページのみ，ノード S からノード D に移動する．また，このメモリマイグレーションの間に，ノード S 上のカーネルスレッド T によってライトアクセスが行われたページを全て検出して記録しておく．

以降，一般にラウンド n では，カーネルスレッド T のメモリ領域の全ページのアクセス属性をライトアクセス禁止に設定した上で，バックグラウンドプロセスを使って，ラウンド $n-1$ でライトアクセスが行われたページのみをノード S からノード D に移動する．このようなラウンドを，直前のラウンドにおいてライトアクセスが行われたページ数が十分に小さくなるか，もしくはラウンド回数が一定数に達するまで繰り返す．最後に，ノード S 上のカーネルスレッド T を停止させ，CPU レジスタと最終ラウンドにおいてライトアクセスが行われたページ全てをノード S からノード D に移動する．その後，ノード D 上でカーネルスレッド T を復帰させることで，カーネルスレッドマイグレーションが完了する．以上の Pre-copy のタイムラインを Fig.4(A) に示す．

この Pre-copy の欠点は，ライトアクセスが行われたページが何度もノード S からノード D に移動される可能性があることだが，これに関しては，メモリアクセスの時間的局所性に基づき余分なページを重複転送しないための改善策が提案されている [9]．具体的には，過去のラウンドにおいてライトアクセスが度々発生しているページは今後のラウンドにおいても再びライトアクセスされる可能性が高いため，ラウンド n では，ラウンド $n-1$ でライトアクセスが行われたページ全てを移動するのではなく，ラウンド $n-1$ でライトアクセスが行われたページのうち過去のラウンドであまりライトアクセスが行われていないページのみを移動することで，ライトアクセスが多発するページの重複転送を低減できる．

4.2.2　特徴

Pre-copy の利点は，メモリマイグレーションが完全にバックグラウンドプロセスによって実行されるため，アプリケーションの性能劣化が小さい点である．一方，第一の欠点として，ライトアクセスが行われたページが重複転送される可能性があり，実際に使用しているページ数以上のページ数の移動が行われるため，ネットワークバンド幅が浪費されてしまう．この影響は，リードアクセス中心のアプリケーションでは無視できるがライトアクセス中心のアプリケーションでは顕著になる．よって，第二の欠点として，メモリマイグレーションに要するコストがアプリケーション依存になるという点が指摘できる．また，第三の欠点として，特にライトアクセス中心のアプリケーションでは停止時間中に移動させなければならないページ数が多くなるため，停止時間が長くなってしまう．

Tab.2　Characteristics comparison between Pre-copy and Post-copy （移動ページ数に関する「同じ」「ほぼ同じ」の比較対象は，実際に使用されているページ数である）．

手法	Pre-copy	Post-copy
リード中心アプリの移動ページ数	ほぼ同じ	同じ
ライト中心アプリの移動ページ数	多い	同じ
多様なアプリに対する安定性	低い	高い
停止時間	長い	短い
アプリの性能劣化	小さい	大きい

4.3　Post-copy

4.3.1　アルゴリズム

まず基本アイディアから述べる．Post-copy では，まず，ノード S 上でカーネルスレッド T を停止させ，CPU レジスタのみをノード S からノード D に移動する．次に，ノード D 上で全ページの保護属性をリードアクセス禁止かつライトアクセス禁止に設定した上で，ノード D 上でカーネルスレッド T を復帰させる．当然，この状態では，カーネルスレッド T による全てのリードアクセスとライトアクセスはメモリ保護違反を引き起こすが，このメモリ保護違反を契機として，ノード S に該当ページを要求してノード D に移動することによって，カーネルスレッド T の実行を継続させる．すなわち，カーネルスレッド T のアクセス違反を契機として，demand-driven にノード S からノード D へメモリマイグレーションを行う．

以上が Post-copy の基本アイディアであるが，この基本アイディアには問題がある．第一の問題として，アクセス違反の度にノード S に対してページ移動の要求を発行していては，アプリケーションの性能劣化が著しい．第二の問題として，ノード D 上のカーネルスレッド T によってアクセスされないページは永久にノード S に残存するため，いつまで経っても移動元ノードとの依存関係が解消できない．これらの問題は，Post-copy が “pull 型” でしかメモリマイグレーションを行わないという点に起因しているため，問題を解決するためには “push 型” の因子を追加すれば良い．すなわち，カーネルスレッド T とは別のバックグラウンドプロセスを用意して，ノード S からノード D に強制的なメモリマイグレーションを行えば良い．

さらに，このバックグラウンドプロセスによるメモリマイグレーションは，カーネルスレッド T のメモリアクセスの時間的

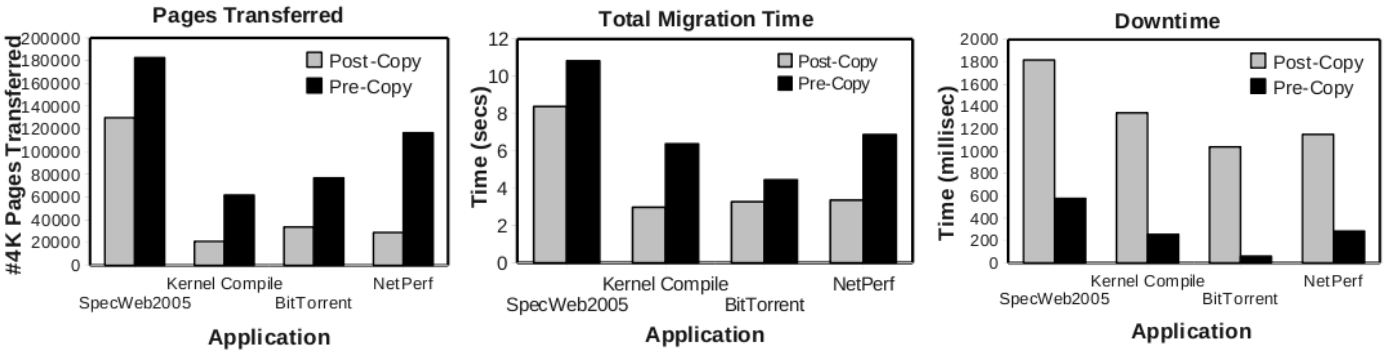

Fig.6 Performance comparison between Pre-copy and Post-copy in VM migration[16].

var $N \leftarrow$ the total number of pages
var $bitmap[0..N-1] \leftarrow \{0, 0, \ldots, 0\}$
var $pivot \leftarrow 0$
var $bubble \leftarrow 0$

A background procedure :
  while $bubble < \max(pivot, N - pivot)$ **do**
    $left \leftarrow \max(0, pivot - bubble)$
    $right \leftarrow \min(N-1, pivot + bubble)$
    if $bitmap[left] = 0$ **then**
      $bitmap[left] \leftarrow 1$
      queue the page *left* for transmission
    endif
    if $bitmap[right] = 0$ **then**
      $bitmap[right] \leftarrow 1$
      queue the page *right* for transmission
    endif
    $bubble \leftarrow bubble + 1$
  endwhile

When a page fault on a page p occurred :
  if $bitmap[p] = 0$ **then**
    $bitmap[p] \leftarrow 1$
    transmit the page p immediately
    discard pending queue
    $pivot \leftarrow p$
    $bubble \leftarrow 1$
  endif

Fig.5 An algorithm which migrates pages from a source node to a destination node, considering temporal access locality.

局所性に基づいた順序で行われることが望ましい．このようなアルゴリズムとして，論文[16]では Fig.5 に示すアルゴリズムが示されている．このアルゴリズムでは，ノード S に *pivot* という変数を用意し，最も直近にノード D 上のカーネルスレッド T においてアクセス違反のあったページを管理させる．そして，ノード S 上のバックグラウンドプロセスを利用して，*pivot* の左右両側に広がる順序で，ページをノード S からノード D に移動する．ノード D 上のカーネルスレッド T が別のページに対してアクセス違反を引き起こした場合，その瞬間に *pivot* の値はそのページに更新され，今度はその *pivot* の左右両側に広がる順序で，ノード S からノード D へのページ移動が進行していく．要するに，このアルゴリズムでは，カーネルスレッド T がページ p でアクセス違反を引き起こした場合に，近い将来にページ p の周辺でアクセス違反が起きるだろうという予測を根拠として，最も直近にアクセス違反があったページの周辺から優先的にページ移動を行う．以上の Post-copy のタイムラインを Fig.4(B) に示す．

4.3.2 特徴

Post-copy の第一の利点として，停止時間中には CPU レジスタのみを移動すれば良いため，停止時間が短い．第二の利点としては，Post-copy では，移動されるページ数が実際に使用されているページ数と一致するため，ネットワークバンド幅を余分に消費することがない．これは，リードアクセス中心のアプリケーションでもライトアクセス中心のアプリケーションでもメモリマイグレーションに要するコストが等しいことを意味するため，第三の利点として，さまざまなアプリケーションに対する挙動の安定性が指摘できる．一方で，欠点として，アクセス違反発生の度にアプリケーションの性能が劣化する．

4.4 Pre-copy vs Post-copy

Pre-copy と Post-copy の特徴を Tab.2 に比較する．また，Pre-copy と Post-copy を（スレッドマイグレーションではなく）VM マイグレーション[16, 9, 13]の手段として適用し，SpecWeb2005, Kernel Compile, BitTorrent, NetPerf の 4 つのアプリケーションを実行中に VM を移動させた場合における，移動ページ数，VM マイグレーションの所要時間，VM の停止時間を Fig.6 に示す．Fig.6 において，Post-copy の停止時間が Pre-copy より長いのは前述の考察と異なるが，これは，実装を簡略化するために，Post-copy の停止時間の中で，移動元ノードに存在する全ページを一度仮想的なデバイスにスワップアウトさせる処理を挟んでいるためである．アプリケーションや実験条件の詳細は論文[16]を参照されたい．

一般に，Pre-copy と Post-copy のいずれが優れているかは目的に依存するが，クラウドコンピューティングサービスにおけるスレッドマイグレーション型モデルを実現する上では Post-copy の方が適している．なぜなら，2.4 節で述べたように，スレッドマイグレーション型モデルでは負荷変動に対して高速に適応できるような資源のスケジューリングが重要になるため，移動元のノードで長らくスレッドが実行され続ける Pre-copy よりも，移動の必要が生じた場合には直ちに移動元

のノードからスレッドが退去する Post-copy の方が望ましいためである．

5 結論

多数の計算資源を要求する並列分散アプリケーションの増加に伴い，計算資源を必要なときに必要な量だけ，従量制課金のサービスとして利用できるクラウドコンピューティングの重要性が高まっている．クラウドコンピューティングには多様な形態が存在するが，

- 負荷の増減に応じて柔軟に計算規模を拡張/縮小できること
- （何らかのポリシーに基づいて）共有資源を利用者間でスケジューリングできること

が最低限の共通の要請として課せられる．本稿では，この 2 つの要請に着眼しつつ，Amazon EC2 と Google App Engine について分析した上で，その中間的存在として，スレッドマイグレーション型モデルに基づくクラウドコンピューティングサービスの形態を提案した．さらに，スレッドマイグレーション型モデルを効率的に実現するための要素技術として，カーネルスレッドマイグレーションのための Iso-address，高速なメモリマイグレーションのための Pre-copy と Post-copy を紹介した．特に，Pre-copy と Post-copy の特徴を比較し，スレッドマイグレーション型モデルに対しては Post-copy の方が適していることを指摘した．

参考文献

1) Amazon EC2. http://aws.amazon.com/ec2/.
2) Google App Engine. http://code.google.com/intl/appengine/.
3) Google Docs. http://docs.google.com/.
4) Salesforce.com. http://www.salesforce.com/.
5) Windows Azure. http://www.microsoft.com/windowsazure/.
6) Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System. *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pp. 496–510, 1999.
7) Gabriel Antoniu and Christian Perez. Using Preemptive Thread Migration to Load-Balance Data-Parallel Applications. *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pp. 117–124, 1999.
8) Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, Vol. 25, pp. 599–616, 12 2008.
9) Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansenf, Eric Julf, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation*, Vol. 2, pp. 273–286, 2005.
10) David Cronk, Matthew Haines, and Piyush Mehrotra. Thread Migration in the Presence of Pointers. *Proceedings of the 30th Hawaii International Conference on System Sciences: Software Technology and Architecture*, Vol. 1, pp. 292–302, 1997.
11) Hai Jiang and Vipin Chaudhary. On Improving Thread Migration: Safety and Performance. *Proceedings of the 9th International Conference on High Performance Computing*, pp. 474–484, 2002.
12) K.Thitikamol and P.Keleher. Thread migration and communication minimization in DSM systems. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, Vol. 87, pp. 487–497, 3 1999.
13) H.Andres Lagar-Cavilla, Joseph A.Whitney, Adin Scannell, Philip Patchin, Stephen M.Rumble, Eyal de Lara, Michael Brudno, and M.Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. *Proceedings of the 4th ACM European conference on Computer systems*, pp. 1–12, 2009.
14) L.Vaquero, L.Rodero-Marino, J.Caceres, and M.Lindner. A Break in the Clouds : Towards a Cloud Definition. *SIGCOMM Computer Communication Review*, pp. 137–150, 2009.
15) Armbrust M., A.Fox, R.Griffith, A.D.Joseph, R.Katz, A.Konwinski, G.Lee, D.A.Patterson, A.Rabkin, I.Stoica, and M.Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2 2009.
16) Michael R.Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 51–60, 2009.
17) Wenzhang Zhu, Cho-Li Wang, and Lau F.C.M. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. *Fourth IEEE International Conference on Cluster Computing*, pp. 381–388, 2002.
18) 田浦健次朗. クラウド時代の基盤ソフトウェア，ツール，プログラミングシステム. 東京大学 情報理工学系研究科 講演会, 10 2009.