DBSJ Journal Vol. 12, No. 1
June 2013

大規模データベースにおけるアクセス局所性を利用したVMライブマイグレーションスケジューリング手法の提案と評価

Runtime Load Balancing for Data Intensive Application Exploiting Access Locality

石田 渉♡ 横山 大作♢ 中野 美由紀♢ 豊田 正史♢
喜連川 優♢

Wataru ISHIDA, Daisaku YOKOYAMA,
Miyuki NAKANO, Masashi TOYODA,
Masaru KITSUREGAWA

本論文では大規模な OLTP を行うアプリケーションのエラスティシティを効率良く実現する手法を提案する。この手法ではアプリケーション性能のボトルネックとなりうるハイパバイザーとデータベースの負荷分散を協調して行う. 特にデータベースの負荷分散に関してはアプリケーションのアクセス分布とデータベースのキャッシュの類似度を考慮することによってデータベースのキャッシュミスによるアプリケーションの性能劣化を防ぎつつエラスティシティを実現する. 評価実験では実際にクラウドコンピューティング環境を模した実環境を構築し, TPC-C ベンチマークを利用したアプリケーションを実行し, 提案手法によりアプリケーションの性能劣化を防ぎつつハイパバイザーとデータベースの負荷分散が実行できることを確認した.

In this paper, we are going to show controlling the elasticity of application servers and databases separetly can cause performance degradation of client applications. To solve this problem, we propose the scheduling method of live migration which utilizes VMs' access locality. Then, we clear up the framework of the execution environment of applications which use large scale database. Furthermore, we evaluate the proposed method with OLTP in real environment.

1 はじめに

近年 Amazon EC2[2], Rackspace[4], Microsoft Azure[3], Google Compute Engine[5] といった広く利用されているクラウドコンピューティングサービスの大きな特色は計算資源を動的に増減できるエラスティシティ(伸縮性) にある. e コマースやオンラインゲームなど, 利用者数やアクセスされるデータ量の変動が大きく, 処理負荷が容易に予測できないウェブサービスアプリケーションにおいてエラスティシティの重要性は高い. 一方, これらのアプリケーションでは OLTP(オンライントランザクション処理) が必要不可欠であり, クライアントアプリケーションのエラスティシティを実現するにあたりデータベース等のミドルウェアとの連携が強く望まれている.

♡ 東京大学大学院 情報理工学系研究科

♢ 東京大学生産技術研究所

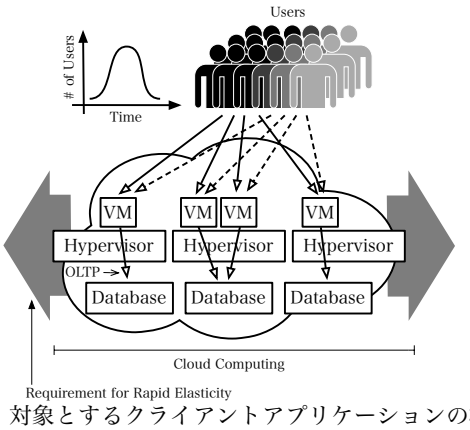

図 1: 対象とするクライアントアプリケーションの構成

本論文では大規模な OLTP 処理を行うアプリケーションのエラスティシティを効率良く実現する手法を検討する. クラウドコンピューティングでは Xen[6] や KVM[7] などの仮想化技術を利用して, エラスティシティの実現手法として VM ライブマイグレーション [1] が利用可能である. またデータベースに関しても Albatross[10], Zephyer[9] といったエラスティシティを実現するための手法が提案されている. しかしより効率的にエラスティシティを実現するためにはアプリケーションとデータベースのエラスティシティを別々に考慮するのではなく協調させることが不可欠である. そこで, 本論文ではクライアントアプリケーションのアクセス局所性を利用した負荷分散スケジューリング手法を提案し, 提案手法を実機により評価する.

本論文の構成は以下のようになっている. 2 でアプリケーションサーバとデータベースサーバに関してエラスティシティを実現するための基本技術を説明する. 3 でアプリケーションとデータベースを協調させた負荷分散を実現する際の課題を指摘し, 4 でアクセス局所性を利用した負荷分散スケジューリング手法を提案した後, 5 で実装方式について説明する. さらに 6 で提案手法の有用性を実機により評価し, 最後に 7 で結論を述べる.

2 クライアントアプリケーションのエラスティシティ実現のための基本技術

図 1 に本論文が対象とするクライアントアプリケーションの構成を示す. 対象とするアプリケーションは, 仮想化技術を利用して VM(ヴァーチャルマシン) として HV(ハイパーバイザー) 上に実装されるアプリケーションサーバ群と, データを管理する DB(データベース) サーバ群から構成される. 本節ではクライアントアプリケーションのエラスティシティを実現するための技術として仮想化技術と VM ライブマイグレーション, そしてマルチテナントなデータベースで利用可能な Albatross[10] と Zephyer[9] を説明する.

2.1 仮想化技術と VM ライブマイグレーション

仮想化技術は物理マシン上で稼働する HV がエミュレートした仮想的な物理環境上で VM を稼働させるもので, VM は自らが仮想化されていることを意識せず, HV を物理マシンと認識し稼働する. クラウドプロバイダは仮想化技術を利用しクライアントに提供する計算資源を VM として扱うことで, 物理的な設定などを必要とせず, クライアントに迅速に計算資源を提供できる. またクライアントはクラウドプロバイダが持つ物理インフラを意識せずに自分専用の計算資源として VM を利用することができる. 代表的な HV の実装として KVM[7] と Xen[6] がある.

アプリケーションサーバのエラスティシティを実現する方法として VM ライブマイグレーションを紹介する. VM ライブマイグレーションとは HV 上で稼働している VM を停止させることなく, 異なる HV 上に移す技術で Clark らによって提案された [1]. VM

のメモリと CPU ステート, ハードウェアのステートを VM を止めずに段階的にマイグレート先の HV に転送することで行う. VM の動作中はメモリの書き換えが生じるため多くの場合メモリの再転送が必要となり, VM ライブマイグレーションにおいてはこのメモリ転送量の削減がマイグレーションのオーバーヘッドを減らす要となっている. VM ライブマイグレーションにおけるメモリの転送量の削減に関しては多くの研究がなされており, 代表的なものに [11], [12], [13] がある. VM ライブマイグレーションにより, クライアントの VM の HV への集約, 分散が可能になり VM が必要とする計算資源と HV が提供できる計算資源のバランスを考慮し VM をライブマイグレーションにより再配置することでエラスティックな計算資源をクライアントに提供することができる.

2.2 マルチテナントデータベースのエラスティシティ

マルチテナントな DB のエラスティシティに対しては DB テナントのライブマイグレーション手法としては Sudipto らによる Zephyr[9] と Aaron らによる Albatross[10] が提案されている. マルチテナントな DB とはそれぞれ独立した契約ポリシーを持つ複数クライアントに物理インフラを共有させる DB のことである. Zephyr はマイグレーション元とマイグレーション先は DB のテーブルを含め何も共有していない状況を前提とし, マイグレーションの前後を通してトランザクション処理の中断を生じさせないためにマイグレーションが始まる時点で既に実行が始まっているトランザクションはそのままマイグレーション元で行い, ライブマイグレーション中に新たに発生したトランザクション処理はマイグレーション先で行う. マイグレーション先で行うトランザクションに必要なデータはオンデマンドにマイグレーション元から読み込む. Albatross も Zephyr と同様マルチテナントな DB のライブマイグレーションであるが, Zephyr とは異なり物理ノードがストレージを共有しているシステムを前提としている. Albatross では VM ライブマイグレーションのメモリ転送と同様, トランザクションテーブルをイテレーティブに転送することでライブマイグレーションの前後を通してトランザクション処理の中断を避ける.

3 クライアントアプリケーションのエラスティシティを実現する際の課題

本節ではアプリケーションサーバである VM, それをホストする HV, そして DB から構成される仮想化環境 (2) でクライアントアプリケーションのエラスティシティを実現するための課題を考察する. 議論を簡単にするため VM 群は $VM1$〜$VM4$, HV を $HV1$, $HV2$, DB を $DB1$, $DB2$ とする. $DB1$ と $DB2$ はレプリケーションにより, どちらを参照しても同様の結果が得られるとする. はじめすべての VM 群は $HV1$ 上で稼働し, $DB1$ を参照している. ここで VM 群の負荷が上昇し, $HV1$ が十分な計算資源を提供できなくなった場合を考える (図 2 ①部). VM 群のうち $VM1$ と $VM2$ を $HV2$ にライブマイグレートすることで $HV1$ の負荷分散をすることができる (図 2 ②部). 一方負荷分散後, VM 群はより計算資源を得る, すなわち $DB1$ への負荷が増加する (図 2 ③部). ここで $DB1$ の計算資源が足りないと, アプリケーション全体としてはスループットが上がらない. そこで $VM1$, $VM2$ の接続する DB サーバを $DB1$ から $DB2$ に切り替えると, DB サーバの負荷も分散する (図 2 ④部). しかし, $VM1$, $VM2$ の利用するデータが $DB2$ のキャッシュにない場合, キャッシュミスが発生し, $DB1$ へリクエストし続ける場合と比較し, クライアントアプリケーションの性能が劣化する (図 2 ⑤部). ここでは HV が先にボトルネックとなる状況を想定したが, 実環境では HV と DB のどちらが先にボトルネックになるかは環境と負荷に依存し, 一方の負荷分散が他方の負荷増大を招く恐れがある.

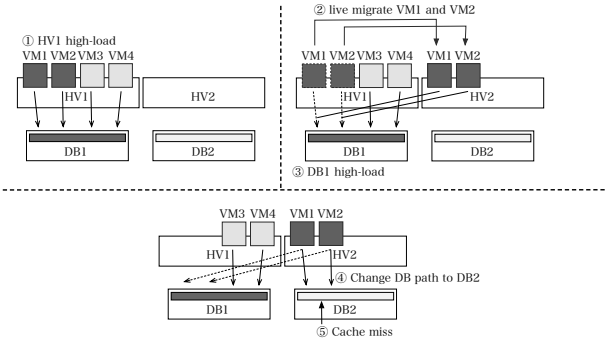

図 2: 想定環境

3.1 キャッシュミスによるアプリケーション性能劣化の検証

実際に VM ライブマイグレーションと DB の接続変更をエラスティシティの実現のために実行した際, キャッシュが温まっていない DB に接続変更をしてしまうことでアプリケーションの性能が劣化することを確かめた. VM 内で稼働させるアプリケーションとして TPC-C を一部改変したものを用いた (アプリケーションの詳細については後述する). VM の負荷変化は DB への接続数の変化によって実現する. 検証環境を図 3 に示す. まず $HV1$, $HV2$ 上で合計 24 台の VM を稼働させ, 上述のアプリケーションを接続数 4 として稼働させる. はじめ全ての VM は $DB1$ を参照し, $DB1$ のキャッシュは温まっている. 全ての VM の接続数を 50 秒毎に 4 ずつ接続数が 16 になるまで上昇させ負荷を増大させていく (図 3 ①部). 負荷上昇の後, HV がボトルネックとなるため, 2 台の HV からそれぞれ 4 台, 合計 8 台の VM を $HV3$ にマイグレートし HV の負荷分散を行う (図 3 ②部). マイグレーションの後 3 台の HV 上でそれぞれ 8 台の VM が稼働する状態となる. HV の負荷分散によって VM に与えられる計算資源が増加するため, VM の $DB1$ へのクエリ頻度が高まり今度は $DB1$ の負荷が上昇する. そこで 24 台の VM のうち半数の接続する DB を $DB2$ に変更し, DB の負荷分散を行う (図 3 ③部). このとき $DB2$ のキャッシュが温まっている場合と温まっていない場合の 2 通りで計測を行う.

$DB2$ のキャッシュが温まっている場合と温まっていない場合での, 全 VM のアプリケーションの TPM(transaction per minites) と平均クエリ処理時間 (delay) の平均の時間変化の様子を図 4 に示す. 横軸は計測を開始してからの時間である. 0 から 200 秒までは VM の負荷を上昇させている様子, 200 秒から 270 秒までは VM のマイグレーションを行った後の様子, 270 秒から 320 秒までが DB の接続先変更を行った後の様子となっている. グラフから, キャッシュが温まっていない場合, $DB2$ に接続変更をした後, キャッシュが温まっている場合に比べて TPM は約 40%減少し, 平均クエリ処理時間は約 3 倍増加することがわかる. また 100 秒から 200 秒の間, 接続数は増加しているにも関わらず TPM が 50 秒から 100 秒の間よりも減少している. これは VM の負荷増大により HV がボトルネックとなり VM に十分な計算資源を提供できていない事による.

以上より, アプリケーションのエラスティシティを実現するために VM ライブマイグレーションと DB の接続先変更を単純に利用しただけではアプリケーション性能を大幅に劣化させてしまうことがわかる. このため HV と DB を協調させることでアプリケーションの性能を維持しつつ, エラスティシティを実現するフレームワークが必要である.

4 エラスティシティ実現のための負荷分散スケジューリング手法

本論文ではアプリケーションの性能を維持しつつエラスティシティを実現するためのアプリケーションのアクセス局所性を考慮した実行時負荷分散フレームワークを提案する. 負荷分散フレー

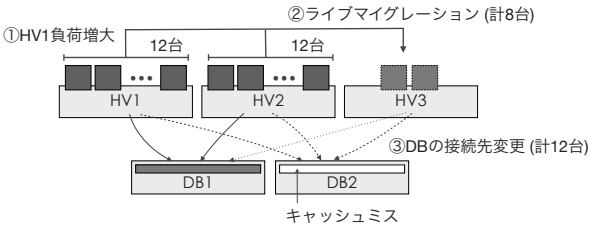

図 3: 計測環境

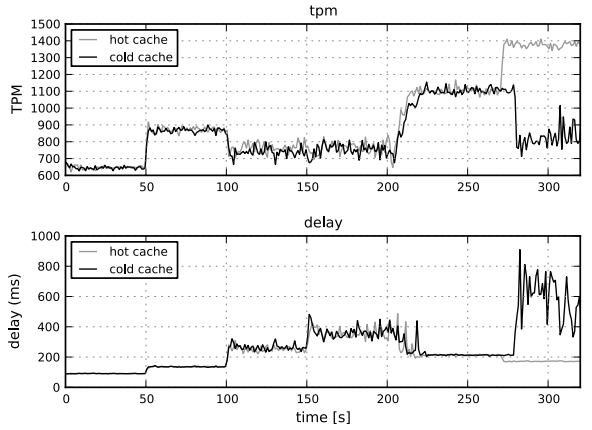

図 4: キャッシュミスによる性能劣化の様子

ムワークを考えるに当たって, 負荷分散をトリガーする基準を決定しなければいけない. クライアントとクラウドサービスのプロバイダとの間にはサービスレベル契約 (SLA) が結ばれることが一般的であるため, 負荷分散のトリガーは SLA が遵守されるように行われる必要がある.

負荷分散アルゴリズムは次の 2 つの指針に基づいて設計する.

1. なるべく少ない HV と DB で全ての VM の SLA を守る
2. ボトルネックがはっきりしないときは SLA を守れなくても負荷分散を実行しない.

1 つ目の指針は VM のスケールインを想定したものである. 本論文では負荷分散によるスケールアウトのみを扱っているが, クラウドコンピューティングではスケールインも同時に考慮しなければいけない. VM のアプリケーション性能を最大化するには, システム内の全 HV 上の VM の台数が均一になるように VM を HV 上に配置し, DB もキャッシュが温まっていることを前提にすれば, システム内の全 DB に接続する VM の台数が均一になるようにすればいい. しかしクラウドコンピューティングにおいて, 金銭的なコストを抑えるため, SLA を満たせる範囲で利用するクラウドコンピューティングの計算資源を最小化することが望ましい. そのため負荷分散においてもなるべく少ない HV と DB で全ての VM の SLA を守るように設計する.

2 つ目の指針はライブマイグレーションと接続先変更の頻発を防ぐためである. ライブマイグレーションと接続先変更にもコストがあるため, 負荷分散することによってアプリケーション性能が向上することが確かな場合以外は負荷分散を避けるべきである. 具体的には本提案手法が想定していないボトルネックによりアプリケーション性能が SLA を守れていないとき負荷分散を行なってもアプリケーション性能が向上する保証がないため本手法の適用範囲外とし負荷分散を行わない.

クラウドコンピューティングの特徴にもあるように, クラウドコンピューティングにおいてクライアントの使用する計算リソースはモニタリングされており, OLTP では各 VM 内の発行するクエリの発行レートとクエリの処理にかかった時間を DB を通して取得することができるため, 本手法では予め設定した SLA を遵守するための閾値 *sla_threshold* を VM のアプリケーション性能が下回った場合に負荷分散をトリガーする.

負荷分散がトリガーされた場合, HV と DB のどちらがボトルネックとなっているかを判断する. HV と DB のどちらが VM の性能のボトルネックになっているかは HV と DB のリソース使用率が基準値 *hv_heavy_threshold*, *db_heavy_threshold* を上回っているか否かで判定する. リソース使用率は例えば CPU, メモリ, ネットワークなどの利用率から算出することを想定しているが, アプリケーションによってボトルネックの原因となるリソースは異なるため, 具体的な算出方法は 4 章でアプリケーションの詳細とともに議論する. HV と DB のリソース使用率が基準値 *hv_heavy_threshold*, *db_heavy_threshold* を下回っているにもかかわらず, アプリケーション性能が *sla_threshold* を下回っている場合は 2 つ目の指針に基づき, 負荷分散を行わない.

ボトルネックが HV の場合は VM ライブマイグレーションを, DB の場合は VM の DB 接続先を変更する. 負荷分散スケジューリングフレームワークのアルゴリズムをアルゴリズム 1 に示す.

Algorithm 1 負荷分散スケジューリングフレームワーク

```
//threshold for load balancing trigger
sla_threshold
//threshold to judge HV is the bottleneck.
hv_heavy_threashold
//threshold to judge DB is the bottleneck.
db_heavy_threshold
// array of virtual machine in the system vms
loop
  for all vm in vms do
    if vm.performance < sla_threshold then
      hv = get_hv(vm)
      db = get_db(vm)
      if hv.load() > hv_heavy_threshold then
        hv_load_distribution(vm)
      if db.load() > db_heavy_threshold then
        db_load_distribution(vm)
```

4.1 HV の負荷分散

HV がボトルネックと判定された場合, VM のライブマイグレーションを検討する. マイグレーション先の決定アルゴリズムをアルゴリズム 2 に示す. マイグレーション先の決定のためまず全 HV に対し現在のリソース使用率とマイグレートする VM が消費すると予想されるリソース使用量を足した *estimated_resource_usage* を算出する. *hv_heavy_threashold* を下回る *estimated_resource_usage* を持つ HV の中で *estimated_resource_usage* が最大となるものをマイグレーション先とする. *estimated_resource_usage* が *hv_heavy_threashold* を下回る HV をマイグレーション先の候補とするのは, マイグレート先でも HV のリソース不足が生じることでアプリケーション性能が *sla_threashold* を下回りライブマイグレーションが連鎖するのを防ぐためである. また *hv_heavy_threashold* を下回った中で *estimated_resource_usage* が最大となる HV をマイグレート先として選択するのは, 全 VM のアプリケーション性能を *sla_threashold* 以上にするために稼働させる必要がある HV の台数を最小化するためである.

マイグレーション先が見つかり, かつマイグレーションのコストが *migration_threshold* が下回っている場合マイグレーションを行う. マイグレート先が見つからなかった場合またはマイグレーションのコストが高すぎる場合はシステム全体でアプリケーションの要求に応えられるだけの計算資源がないとして負荷分散を行わない.

Algorithm 2 HV の負荷分散アルゴリズム

```
//array of hypervisor in the system
hvs
dst_candidates = []
for all hv in hvs do
  //estimate the hv's load when vm is migrated
  estimated_resource_usage = hv.load() + vm.load()
  if estimated_resource_usage < hv_heavy_threshold then
    add hv to dst_candidates
//choose the highest loaded hv.
//destination can be NULL if dst_candidates is empty
destination = highest_load(dst_candidates)
if destination and migration_cost < migration_threshold then
  vm.live_migrate(destination)
else
  //give up hv load distribution
```

4.2 DB の負荷分散

DB がボトルネックと判定された場合, DB の接続変更を検討する. DB の接続変更先の決定アルゴリズムをアルゴリズム 3 に示す. DB の接続変更先の決定はリソース使用率と VM のアクセス範囲と DB のキャッシュの類似度の 2 つの指標から決定される. どちらをどれだけ重視するかはパラメータ α ($0 \leq \alpha \leq 1$) によって設定する.

全 DB に対し現在のリソース使用率とマイグレートする VM が消費すると予想されるリソース使用率を足した値 (*estimated_resource_usage*) と VM のアクセス範囲と DB のキャッシュの類似度を算出する (*access_similarity*). *access_similarity* もリソース使用率と同様正規化し 0 以上 1 以下を満たすようにする. 類似度の具体的な算出方法については後述する. *estimated_resource_usage* が *db_heavy_threshold* を上回る DB と *access_similarity* が *db_similarity_threshold* を下回る DB を除外して残った DB の中で $\alpha * estimated\_resource\_usage + (1 - \alpha) * access\_similarity$ の値が最大となる DB を接続変更先とする. α を小さくするとリソース使用率よりも VM のアクセス範囲と DB のキャッシュの類似度を重視して DB の接続変更先を決定する. α を大きくした場合は VM のアクセス範囲と DB のキャッシュの類似度よりもリソース使用率を重視して DB の接続変更先を決定する. 接続変更先が見つかり, かつ接続変更のコストが *change_db_threshold* が下回っている場合接続変更を行う. 接続変更先が見つからなかった場合または接続変更のコストが高すぎる場合はシステム全体でアプリケーションの要求に応えられるだけの計算資源がないとして負荷分散を行わない.

Algorithm 3 DB の負荷分散アルゴリズム

```
//array of database in the system
dbs
dst_candidates = []
for all db in dbs do
  //estimate the db's load when vm is migrated
  estimated_resource_usage = db.load() + vm.load()
  //calculate access similarity
  access_similarity = similarity(db, vm)
  if estimated_resource_usage < hv_heavy_threshold and
  access_similarity > db_similarity_threshold then
    add db to dst_candidates
//choose the highest scored db
//destination can be NULL if dst_candidates is empty
//score is calculated by
//score = α*estimated_resource_usage+(1−α)*access_similarity
//destination can be NULL if dst_candidates is empty
destination = highest_score(dst_candidates)
if destination and change_db_cost < change_db_threshold then
  vm.change_db(destination)
else
  //give up db load distribution
```

5 提案手法の実装

5.1 OLTP を行うアプリケーションの実装

提案フレームワークの実装と評価のため VM 内で稼働させるアプリケーションとして TPC-C を一部改変したアプリケーションを実装し利用した. TPC-C は OLTP システムの評価を目的としたベンチマークであり, 商取引とサプライチェーン管理を模擬したアプリケーションとなっている. 注文, 支払, 注文確認, 発送, 在庫確認の 5 種類のクエリを規定の割合に従いデータベースに対して発行し, 注文クエリが 1 分間に何回処理できるか (TPMC) で OLTP システムの評価を行う. 5 種類のクエリのうち, 注文, 支払, 発送はデータベースの更新を伴うクエリ, 注文確認, 在庫確認は参照のみを行うクエリであるが, TPC-C の規定値では更新を行うクエリが全体の約 95%を占めている. 提案フレームワークにおいてデータベースの負荷分散はレプリケーションによって行う. レプリケーションは参照クエリの負荷分散を行うもので, 更新クエリはマスターノードで処理された後, 全てのスレーブノードにログが転送され再実行されるため, 負荷分散されない. そのため TPC-C を規定値通り実行した場合, データベースのレプリケーションによる負荷分散の効果を期待できない. そこでデータベースのレプリケーションが効果を持つアプリケーションを模擬するため TPC-C の 5 種類のクエリのうち参照クエリのみを実行するアプリケーションとした.

TPC-C ではデータベースに warehouse テーブルを W 個分構築し, クライアントは発行するクエリ毎にランダムに W 個の中から warehouse を選び, その warehouse テーブルに対してクエリを発行する. そのため複数の TPC-C クライアントを実行した場合, 各クライアントのアクセス範囲は全ての warehouse に対して均一なものとなる. 本論文では各クライアントは同一のアプリケーションを実行するが, アクセス範囲は異なり局所性を持つクラウドコンピューティング環境を対象としているため, クライアントに与えるパラメータとしてクエリを発行する warehouse を W 個の warehouse の中から指定できるようにすることでクライアントによるアクセス範囲の違いを実現した. クライアントはパラメータとして与えられた warehouse からランダムに選んだ warehouse に対してクエリを発行する.

負荷変化に関しては TPC-C クライアントにパラメータとして与えられる *connection* を変化させることで実現する. *connection* は TPC-C クライアントが DB に対して張る接続数で, TPC-C クライアント内で *connection* 個のスレッドを立て, その全てで DB に対してクエリを発行する. 実装したアプリケーションでは *connection* と接続先の DB を動的に外部から変更できるようにした.

5.2 計測環境

上述のアプリケーションを稼働させるシステムとして, クラウドコンピューティング環境を模擬した実環境を構築した (図 5). システム内に HV と DB はそれぞれ 8 台存在し, VM のディスクイメージは 128 台分用意した. VM のディスクイメージの保管場所として glusterfs を用いて 8 台のファイルサーバで分散ファイルシステムを構築し, 各 HV はそれをマウントすることで任意の HV 上に任意の VM を起動できる環境を構築した. またコントローラを介して全ての VM の起動, 停止の一元管理を可能にすると伴に, ライブマイグレーションも行うことができるようにした. さらに VM 内で稼働させるアプリケーションの起動, 停止, アプリケーション性能の監視, 接続数の変更, 接続 DB の変更もコントローラを介して実行できる. このコントローラ上に提案フレームワークを実装

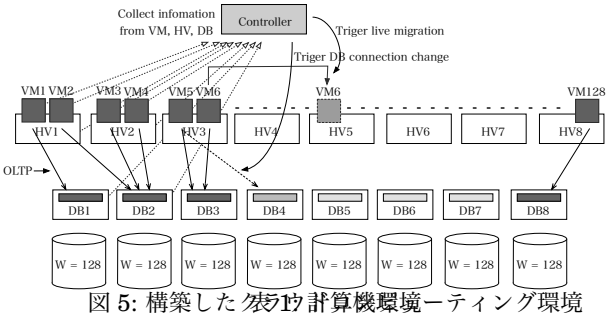

図 5: 構築した 7-17 計算機環境ーティング環境

HV(*8 台)	CPU	Intel Xeon E5530 2. 4Ghz
	memory	24GB
	OS	Debian 6. 0. 4 Squeeze
	kernel	Linux 2. 6. 32
	qemu	qemu-kvm-1. 0. 1
	network	Intel 82599EB 10Gbit
VM	CPU	2 virtual CPU
	memory	1GB
	OS	Ubuntu 11. 04
	kernel	Linux 3. 0. 0
	blk driver	virtio-blk
	network	virtio-net
DB(*8 台)	CPU	Intel Xeon E5530 2. 4Ghz
	memory	24GB
	OS	Debian 6. 0. 5 Squeeze
	kernel	Linux 2. 6. 32
	DB	MySQL 5.5
	network	Intel 82599EB 10Gbit

し, VM 群の負荷上昇に合わせて自動的に HV と DB の負荷分散を実行するようにする. 計算機環境を表 1 に示す. HV, DB はスイッチを介して 10Gbit イーサネットで接続されている.

5.3 各閾値の決定

負荷分散を検討するアプリケーション性能の閾値 *sla_threshold* は平均クエリ処理時間 500ms とした. 構築した環境で HV がボトルネックとなる状況と DB がボトルネックとなる状況のもとで事前評価を行い実装したアプリケーション性能のボトルネックは HV と DB の CPU 使用率から判定できることが確かめられた. よって提案アルゴリズムにおいて HV と DB の負荷状況は CPU 使用率で判定する. HV がボトルネックとなる状況での計測結果で平均クエリ処理時間が 500ms を上回るデータ点のうち HV の CPU 使用率が最小のものは 88.6%, DB がボトルネックとなる状況での計測結果で平均クエリ処理時間が 500ms を上回るデータ点のうち DB の CPU 使用率が最小のものは 80.1%であったため, リソース使用率は 0 以上 1 以下となるよう $hv\_heavy\_threshold = 0.88$, $db\_heavy\_threshold = 0.80$ とした. また *estimated_resource_usage* の算出は計測環境内で VM1 台のみを稼働させた時の結果を利用する. 例えばリソース使用率 0.5 の HV に接続数 4 の VM をマイグレートするとき, VM1 台での計測の接続数 4 のときの HV の CPU 使用率 0.12 より, $estimated\_resource\_usage = 0.5 + 0.12 = 0.62$ となる. DB の接続変更先の決定に VM のアクセス範囲と DB キャッシュの類似度のどちらを優先するかを示す指標である α はどちらも同等に評価する 0.5 と設定した.

5.4 DB キャッシュと VM アクセス範囲の類似度と DB の接続先変更を行う際の閾値の決定

提案フレームワークでは DB キャッシュの情報は DB から取得することを想定しているが, 本論文では実装を簡便に済ませるためアクセス単位を warehouse として VM のアクセス範囲の情報と接続先 DB の情報よりコントローラが DB キャッシュの状態を実行時にシミュレートする.

シミュレーションのアルゴリズムをアルゴリズム 4 に示す. まず各 DB についてキャッシュの状態を更新する. DB に接続する VM を取得し, 接続する VM がアクセスする warehouse を取得する. DB のキャッシュ情報はコントローラ内で warehouse の ID をキーとするハッシュとして実装され 0 から 1 の値をとる. 0 は VM からアクセスされる可能性があるが, DB キャッシュにそのデータが存在しない可能性が高いことを示し, 1 は DB キャッシュにデータ存在する可能性が高いことを示す. 取得した warehouse, w がキーとして存在しない場合, キーを w, 値は 0 としてキャッシュ情報を更新する. w がキーとして存在する場合, w に対応する値が 1 未満の場合に限りキャッシュ情報の更新をする. 1 以上の場合は w のデータはキャッシュ上に存在すると判断しさらなる更新は行わない. キャッシュ情報の更新のため, まず DB のディスクアクセススループットを調べる. スループットが予め定めた disk_access_threshold を下回っている場合, キャッシュ上にデータがあるためにディスクが発生していないと判断し, 値に 1 とし更新する. スループットが disk_access_threshold を上回る場合は, 予め定めた cache_increment を VM がアクセスする warehouse の総数で割り, キーに対応する値に足す. VM がアクセスする warehouse の総数で割るのは, アクセスする warehouse の数が増えると w に対するクエリ発行量が相対的に減少するためキャッシュにデータがのるまでに時間がかかると予想されるためである.

disk_access_threshold と cache_increment を決定するため, キャッシュをクリアした状態で事前計測を行った. その結果, アクセスする warehouse 数に比例してアプリケーション性能が収束するまでの時間が長くなることがわかり, 収束時にはディスク読み込み帯域は 1.0MB/sec 以下となり, warehouse1 つに対応するデータがキャッシュにのるまでに 17 秒かかることがわかった. そのため disk_access_threshold は 1.0MB/sec, cache_increment は 1/17 とした. また DB キャッシュと VM アクセス範囲の類似度は VM アクセス範囲に含まれる warehouse の DB キャッシュでのカバー率とし, 次の式で計算する.

$$\begin{aligned} \mathbb{V} &= \text{warehouses which a VM accesses} \\ \mathbb{C} &= \text{warehouses in DB cache} \\ similarity &= \frac{\mathbb{V} \cap \mathbb{C}}{|\mathbb{V}|} \end{aligned}$$

本論文では $db\_similarity\_threshold = 0.5$ とし VM のアクセスする warehouse の半分以上がキャッシュミスを起こすことが予測される場合, DB の接続先変更を中止するようにした.

Algorithm 4 DB キャッシュのシミュレーション

```
//array of database in the system
dbs
loop
  for all db in dbs do
    vms = select vms connected to db
    for all vm in vms do
      warehouses = get warehouses accessed by vm
      for all w in warehouses do
        if db cache doesn't include w then
          db.cache[w] = 0
        else
          if db.cache[w] < 1.0 then
            if db.diskread < disk_access_threshold then
              db.cache[w] = 1.0
            else
              db.cache[w]+= cache_increment
  sleep 1 second
```

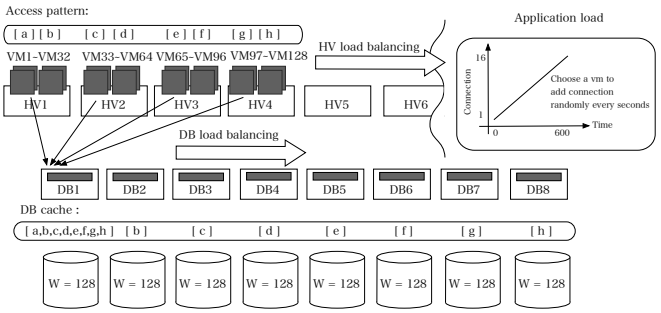

図 6: 大規模環境における実負荷を想定した計測環境

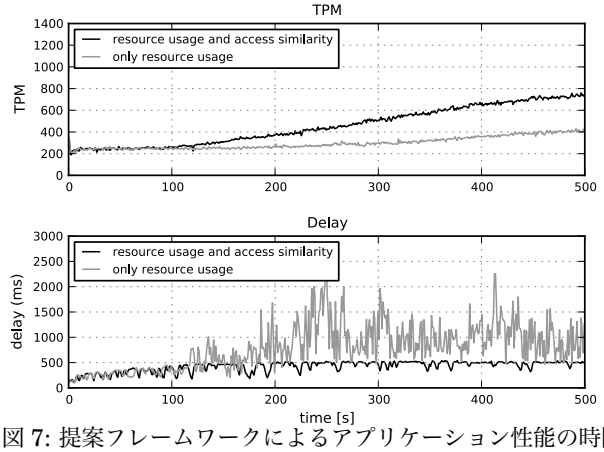

図 7: 提案フレームワークによるアプリケーション性能の時間変化

6 大規模環境における実負荷を想定した提案手法の評価

大規模環境において実負荷を想定した負荷変動をアプリケーションに加えた場合, 提案フレームワークがアプリケーション性能の劣化を防ぎつつエラスティシティを実現できるか評価した. 計測状況を図 6 に示す. VM は 4 台の HV 上に 32 台ずつ, 合計 128 台稼働させ, 全ての VM 内でアプリケーションを実行した. はじめ全ての VM は *DB*1 に接続するようにした. VM のアクセス分布は 8 通りのアクセスパターン (a, b, c, d, e, f, g, h) を用意し, VM1 16 は a, VM17 32 は b...VM113 128 は h と VM16 台をグループとしてアクセスパターンを割り当てた. それぞれのアクセスパターンは 8 つ warehouse を含み, それぞれ共通部分はない. 各 DB は図 6 の DB cache 部に示したアクセスパターンに含まれる warehouse のデータがキャッシュに存在するように温めた. 負荷の上昇は毎秒 128 台の VM の中からランダムに 1 台選び接続数を 1 増やすようにした. また比較対象としてアプリケーションのアクセス分布と DB キャッシュの類似性を考慮せず, DB のリソース利用率のみを指標として DB の接続変更先を決定した場合での計測も行った.

計測の結果を図 7 に示す. 横軸は計測を開始してから時間で上から全 VM の平均 TPM と平均クエリ処理時間を表す. グラフより提案手法においてはアプリケーションのアクセス分布と DB キャッシュの類似性を考慮して負荷分散を行うことで大規模環境においても平均クエリ処理時間を 500ms 以下に抑えられていることがわかる. また TPM に関してもリソース使用率のみを指標として負荷分散をした場合に比べて最大 1.8 倍となり, 性能劣化を抑えてエラスティシティを実現できていることが確認できた.

7 まとめ

本論文ではクラウドコンピューティングにおいてデータインテンシブなアプリケーションのエラスティシティを実現するために, アプリケーションのアクセス局所性を考慮した実行時負荷分散手法を提案した. データインテンシブなアプリケーションでは利用者数の変動, アクセスされるデータ量の変動が大きく, 処理負荷が容易に予測できないため, 負荷の上昇に合わせて実行時に負荷分散を行うことが重要である. クラウドコンピューティングにおいて HV と DB の両者がアプリケーション性能のボトルネックになる可能性があり, それぞれの負荷分散に関する研究は多くなされている. しかしより効率的な負荷分散のためには HV と DB の両者を協調させた負荷分散手法が必要となる. そこで HV と DB を協調させたアプリケーションのアクセス局所性を考慮した実行時負荷分散手法を設計した.

提案手法では, HV と DB の負荷分散を協調して行い, DB の負荷分散の際にアプリケーションのアクセス分布と DB キャッシュの類似度を考慮することでキャッシュミスによるアプリケーションの性能劣化を防ぎつつ負荷分散を行う. 評価実験により提案する手法によってアプリケーションの性能劣化を防ぎつつ負荷分散が行えるか検証を行った. その結果, 性能目標を維持しつつ HV と DB の負荷分散が実行できることが確認された. さらに大規模環境を想定した 128 台の VM が稼働する環境下で VM の負荷を上昇させたときの提案手法の有効性も確認された.

最後に今後の課題をまとめる. 本論文では実装を簡便に済ますため DB のキャッシュ情報を DB から直接得るのではなく, 負荷分散を行うコントローラが VM のアクセス範囲を利用し, 推定するよう実装したがより正確な DB キャッシュの情報を取得するため DB から直接キャッシュ情報を得るようにしたい. またエラスティシティの実現方法として負荷分散にのみ着目したが, アプリケーションの負荷が減少した場合は HV と DB の負荷を集約することでクラウド環境のリソース利用率を下げられると考えられる. アプリケーションの増減に対して適切に計算資源を割り当てる手法についても今後検討して行きたい.

[文献]

[1] C. Clark, K. Fraser, Steven Hand, et al. *Live Migration of Virtual Machines* NSDI, 2005.

[2] Amazon Elastic Compute Cloud(Amazon EC2). http://aws.amazon.com/ec2/.

[3] Microsoft azure. http://www.microsoft.com/azure

[4] rackspace http://www.rackspace.com/

[5] Google Compute Engine. https://cloud.google.com/products/compute-engine

[6] xen http://xen.org/

[7] KVM http://www.linux-kvm.org/page/Main_Page

[8] TPC-C http://www.tpc.org/tpcc/

[9] Aaron J.Elmore, Sudipto Das, Divyakant Agrawal, Amr El Abbadi *Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms* SIGMOD, 2011.

[10] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, Amr El Abbadi *Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration* VLDB, 2011.

[11] Jie Zheng, T.S.Eugene Ng and Kunwadee Sripanidkulchai. ￥em Workload-Aware Live Storage Migration for Clouds. VEE, March 2011.

[12] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar and Matei Ripeanu. ￥em VMFlock: Virtual Machine Co-Migration for the Cloud HPDC, June 2011.

[13] Umesh Deshpande, Xiaoshuang Wang and Kartik Gopalan. ￥em Live Gang Migration of Virtual Machines HPDC, June 2011.

[14] T. Hirofuchi, H. Ogawa, H. Nakada, et al. ￥em A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services on Clouds CCGrid, 2009.