情報処理学会論文誌　コンピューティングシステム　Vol.6　No.1　31–44 (Jan. 2013)

仮想資源管理基盤におけるキャッシュ機構の導入

杉木 章義[1,a)]　加藤 和彦[1]

受付日 2012年7月4日, 採録日 2012年10月30日

概要：近年，クラウドコンピューティングが注目されている．クラウドを支えるデータセンタにおいて，仮想計算機，ネットワーク，ストレージなどのさまざまな資源が用いられており，これらを効率的かつ適切に管理することは重要である．本論文では，これらを管理する仮想資源管理基盤ソフトウェアにキャッシュ機構を導入する．本機構では，さまざまな資源を分散オブジェクトとして統一的に扱うため，分散オブジェクトのキャッシュ技術をそのまま適用することができる．また，キャッシュポリシによる資源ごとのキャッシュ期間の指定や，インバリデーションによるキャッシュの無効化も行う．実験では，個別の操作ごとの性能改善効果やオーバヘッドを示すとともに，多数の仮想計算機を管理する場合の効率性を改善できることを確認した．

キーワード：並列分散システム，資源管理，キャッシュ機構，分散オブジェクト，クラウド・コンピューティング

Maintaining Performance and Consistency in Virtual Resource Management

Akiyoshi Sugiki[1,a)]　Kazuhiko Kato[1]

Received: July 4, 2012, Accepted: October 30, 2012

Abstract: In recent years, cloud computing has been received much attention. Resources in such clouds are managed in a large-scale data center which consists of thousands of machines, network switches, and storages. Virtualization technology at each layer also makes management ever difficult. In this paper, we present a caching technique for such virtual infrastructure. By taking advantage of reference locality, it achieves high efficiency as well as maintaining consistency. Our approach is based on classical caching techniques for distributed objects since resources in the target platform are represented as distributed objects. Consistency is also managed by timeouts and invalidation protocol following the given policy. In the experiments, we have shown that overhead and performance improvement for each operation and demonstrated its effectiveness in virtual machine management.

Keywords: parallel and distributed systems, resource management, cache mechanism, distributed objects, and cloud computing

1. はじめに

近年，クラウドコンピューティングが注目されている．クラウドは，計算資源を自身で所有する必要がなく，また需要の変動にも対応しやすいなどの利点から広く利用されている．利用者からはクラウドの運用形態を意識する必要はないが，実際にはクラウドを支える計算資源は大規模なデータセンタで運用されている [1]．

これらのデータセンタにおいて，仮想計算機，ネットワーク，ストレージなどのそれぞれの資源の情報を効率的かつ，適切に収集することは重要である．これらの情報は，性能の維持や障害の監視，およびそもそもの機器構成や設定内容などの把握に役立つ．しかも，近年では，VM（virtual machine）技術，ネットワーク仮想化技術，ストレージ仮想化技術などのさまざまな仮想化技術が用いられ，データ

[1] 筑波大学大学院システム情報工学研究科
Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] sugiki@cc.tsukuba.ac.jp

センタの複雑性が高まっており，これらの状態の把握がより難しくなっている．

本論文で想定する IaaS（Infrastructure as a Service）環境では，これらの資源を管理するために VMware Infrastructure [2]，OpenNebula [3] などをはじめとする**仮想資源管理基盤**（virtual infrastructure）[4] が用いられている．また，利用者ポータルまでを備えた Eucalyptus [5]，OpenStack [6]，CloudStack [7] などの完全なクラウド基盤ソフトウェアも，中核部分に，仮想資源管理基盤の機能を備えている．これらの仮想資源管理基盤は，VM，仮想ネットワーク，ストレージなどの資源の管理に特化した機能を提供し，この機能をもとに多くの事業者が利用者ポータルなどをカスタマイズし，実際の IaaS 型のクラウドサービスを提供している．

本論文では，これらの仮想資源管理基盤に対するキャッシュ機構を提案する．キャッシュ機構の導入により，さまざまな資源を監視する場合の性能改善効果と一貫性の両立を目指す．

本研究のアプローチは大きく 2 つである．まず，さまざまな資源を扱う必要性から，すべての資源を分散オブジェクトとして抽象化し，統一的に扱う．これにより，資源間の差異を吸収し，キャッシュ機構などの共通性の高い機構の導入を容易にする．次に，伝統的な分散オブジェクトのキャッシュ技術 [8], [9], [10], [11] をこれらのオブジェクトに適用し，仮想資源基盤へのキャッシュ機構の導入を達成する．

また，想定される利用環境の分析から，キャッシュの一貫性の管理にタイムアウト方式と無効化（invalidation）方式を採用する．一般的なキャッシュ機構と同様に，これらのタイミングをキャッシュポリシで資源ごとに制御可能とする．

実験では，キャッシュ機構の有効性をマイクロベンチマークとマクロベンチマークにより示した．まず，操作ごとの性能改善効果やオーバヘッドを示し，多数の VM を操作する場合での性能改善効果を示した．

改めて本論文の貢献は次のようにまとめることができる．

- 本論文では，クラウドの仮想資源管理基盤におけるそれぞれの資源によらない統一的なキャッシュ機構の導入可能性について分析している．
- 本論文では，キャッシュ機構の効果的な実装方法について示す．本論文が対象とする仮想資源管理基盤は各種資源を分散オブジェクトとして抽象化するため，従来の分散オブジェクトを対象としたキャッシュ機構の手法をそのまま適用することができる．
- 本論文では，仮想資源管理において重要性の高いセキュリティ上のアクセス制御にキャッシュ機構を活用し，性能改善効果を実験的に確認している．

本論文の構成は次のとおりである．まず，2 章で仮想資源管理基盤へのキャッシュ機構導入可能性について議論する．次に，3 章で今回，キャッシュ機構の導入を行う仮想資源管理基盤 Kumoi について説明し，4 章で提案するキャッシュ機構の方式について記述する．5 章で実装を示し，6 章でセキュリティ機構との連携について説明する．7 章で実験結果を示し，8 章で関連研究について述べ，最後に 9 章で本論文をまとめる．

2. キャッシュ機構の導入可能性

まず，本章ではクラウドの仮想資源管理基盤におけるキャッシュ機構の導入可能性について議論する．次に，導入によって期待される性能改善効果についても議論する．

2.1 仮想資源管理基盤の想定

本論文では，**図 1** のような仮想資源管理基盤 [4] を想定する．具体的なミドルウェアとしては，VMware Infrastructure [2]，OpenNebula [3] などのソフトウェアが対応し，Eucalytpus [5]，OpenStack [6]，CloudStack [7] などの利用者ポータル部分を取り除いた中核部分のソフトウェアにも相当する．これらの仮想資源管理基盤では，物理計算機および VM，ネットワーク，ストレージなどの計算資源を管理する．

これらの仮想資源管理基盤では，それぞれの資源からつねに情報を収集し，フィードバックとなる操作を繰り返し，データセンタを適切に維持していくことが中核的な役割となる．また，クラウドの利用者からの要求に基づいて，資源を適切に割り当て，サービスを適切に提供することも必要である．このうち，キャッシュが特に有効なのは前者の情報取得の部分であり，取得の対象となる情報には次のようなものが含まれる．また，**表 1** にも具体例を示す．

- **インベントリ情報**：CPU，メモリ，ディスクなどのハードウェアの構成情報やソフトウェアのインストール情報などを取得する．データセンタでは，運用コストを下げるため，できるだけ均一な構成を目指すが，数世代にわたって段階的な拡張や置き換えが行われるため，結果としてヘテロジニアスな構成となり，これらの情報の把握が必要である．
- **設定情報**：設定ファイルやレジストリに記載された設

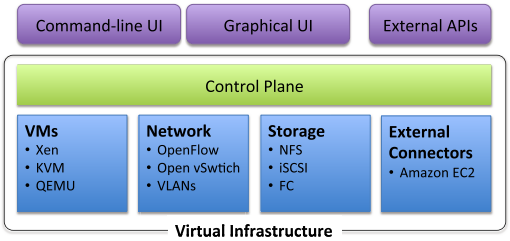


図 1　仮想資源管理基盤
Fig. 1　A virtual resource management platform.

表 1　仮想資源管理基盤で収集する情報
Table 1　Information and statistics collected in virtual resource management.

情報の種類	具体的な例
インベントリ情報	CPU のコア数・周波数，メモリ・ディスク容量， OS のバージョンやライセンス情報，仮想マシンモニタなどのインストール状況など
設定情報	VM に割り当てる仮想 CPU コア数，メモリ・ディスク容量など
実行状態	稼働している VM 数，各 VM の実行 CPU 状態など
性能情報	CPU 使用率，メモリ使用率，ネットワーク転送量，送受信エラー量など

定（configuration）情報を取得する．性能パラメータの値や，特定の機能の有効や無効などの値が対応する．上記のインベントリ情報ほど変わらない情報ではないが，管理者の必要に応じて更新される．

- **実行状態：**クラウドのサービスを構成するコンポーネントが正しく機能しているか確認する．VM が適切に起動しているかどうかの情報や各ネットワークスイッチの状態，ストレージ領域が確保されているかなどの情報が該当する．これらの機能性が確保されているか把握しなければ，サービスを適切に提供することはできない．
- **性能情報：**上記に加えて，各コンポーネントが，十分な性能を提供しているかどうか確認する．これらの把握は SLA（Service-Level Agreement）を満たすために重要であり，システムの状況に応じて動的に変動する．

これらの情報は，仮想資源管理基盤においていずれも重要な情報であり，上から下に向かうほど更新頻度が頻繁になる．そのため，上の情報ほど長時間のキャッシュが可能であり，下になるほど情報の特性に応じたキャッシュ時間の調整が必要になる．

2.2　期待されるキャッシュ効果

仮想資源管理基盤にキャッシュを導入した場合の性能改善効果は，主に以下の 2 つの理由によると期待される．

- **通信コストの削減：**仮想資源基盤はデータセンタ内に分散した機器を管理するため，必然的に分散システムとなる．その際，各機器との通信コストが問題となる．キャッシュ機構の導入によって，これらの通信コストを削減できれば，効率的に監視することができる．
- **計算コストの削減：**通信コストに加えて，各資源での情報を取得する際の計算コストが問題となる．対象となる情報の取得にオーバヘッドが大きくかかるようであれば，特にキャッシュが有効となる．

また，一般的にキャッシュ機構が有効に機能するのは，**参照局所性**（locality of reference）がある場合である．仮想資源管理基盤においても同様であり，これらについて議論する．

- **時間局所性：**同じ情報が時間的に繰り返し参照される場合，キャッシュは有効に機能する．仮想資源管理基盤の場合には，VM のスケジューリングアルゴリズムなど，反復（iterative）計算を行う場合が該当する．
- **空間局所性：**空間的に近い領域が参照される場合も，キャッシュしておけばよい情報が小さくなるため，有効に機能することが期待される．仮想資源管理基盤では，データセンタ全体を 1 度に管理することは少なく，いくつかの領域に分割して管理される．たとえば，サーバが搭載されたラックごとや，ラックの列を単位として管理を行う場合が該当する．この場合，キャッシュが有効に機能することが期待される．
- **逐次局所性：**逐次処理を行う場合も先読み（prefetch）が可能となり，キャッシュが有効に機能する．データセンタ内の計算機から逐次的に情報を取得し，処理を進める場合には，先読みが有効に機能することが期待される．ただし，並列に操作を行う場合も多いため，この場合は空間的局所性に注目した方が効果的である．

以上の参照局所性の議論から，仮想資源管理基盤においてもキャッシュが有効に機能することが期待される．逐次局所性の効果は並列操作によって限定的となる可能性もあることから，本研究では，まず，時間局所性と空間局所性に注目した手法を提案する．

3.　導入対象の基盤ソフトウェア

本研究では，仮想資源管理基盤ソフトウェアの 1 つである Kumoi [12], [13] にキャッシュ機構を導入する．本論文で必要な範囲について，本章で説明する．

3.1　概要

Kumoi の概要を**図 2** に示す．Kumoi では，クラウドに要求される俊敏性への対応から，高度なカスタマイズ性と拡張性を目標としており，中核となる機能を最小化し，その他の多くの機能を周辺機能としてスクリプトで実現するというアプローチをとっている．

そのため，ミドルウェアの機能をまず，カーネルとシェルに分割し，カーネルはデータセンタ内の各計算機で動作させ，シェルは管理者の計算機などさまざまな場所で動作するように設計している．

カーネルでは，Java Remote Method Invocation（RMI）と Scala の Actor を組み合わせた通信機構のみをマイクロ

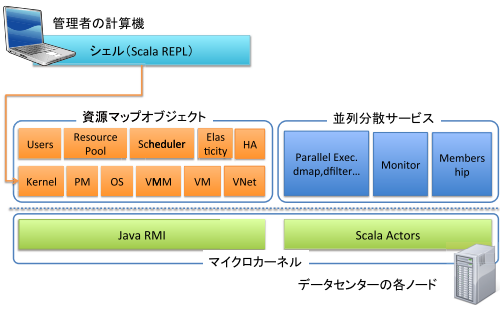


図 2　Kumoi の概要
Fig. 2　Overview of Kumoi.

```
kumoi> pms.dmap(_.vms).flatten
```

図 3　具体的なスクリプト記述例
Fig. 3　Example of small script.

カーネルとして最下層に配置している．そのうえで，物理計算機や VM などの各資源を分散オブジェクトとして抽象化し，統一的なインタフェースを提供している．Kumoi では，これらのオブジェクトを**資源マップオブジェクト**と呼んでいる．一方で，スケーラビリティを向上させるために，`dmap()` や `dfilter()` などの並列スケルトンや，メンバシップ機能などの並列分散サービスを提供している．

クラウドの管理者や開発者は，これらの 2 つの機能を組み合わせ，実現したい作業をスクリプトとしてシェル環境で記述していく．本記述は Scala [14] をベースとしている．

図 3 は具体的なスクリプト記述例である．`pms` は現在，システムに参加している物理計算機に対応する資源マップオブジェクトのリストである．一方で，`vms` はそれぞれの計算機で稼働している VM に対応する資源マップオブジェクトのリストである．本スクリプトでは，並列版の `map` 関数である `dmap()` を使用し，最後に `flatten` で現在稼働している VM の一覧のリストとしている．

3.2　クラウド環境の構築

Kumoi を使用してクラウド環境を構築した場合の例を**図 4** に示す．前節で述べたように，Kumoi のカーネルはサーバ資源プールの各計算機で動作する．一方で，シェル環境はクラウドのフロントエンドとなる計算機で動作させ，また必要に応じて管理者の計算機で起動することもできる．スクリプティングによって，さまざまな利用者インタフェースを柔軟に構築し，クラウドの利用者に提供することができる．なお，シェル環境はあくまで利用者インタフェースの構築に利用することを想定しており，クラウドの利用者が直接，シェル環境を操作することは想定していない．

Kumoi では，このほかにもさまざまな計算機を利用して，クラウド環境を構築する．まず，資源プールなどの大域的な機能の高可用性を実現するために，ZooKeeper を利用する．次に，利用者認証を LDAP サーバを通じて行い，その認証情報をもとに，さまざまな資源へのアクセス認可を行うデータベースを MySQL で構築している．また，クラウド環境の提供には欠かせない課金情報の記録や監査ログの記録なども MySQL で行っている．最後に，VM のライブマイグレーションを実現するために，VM 用のストレージも利用しており，現在は NFS サーバで実現している．

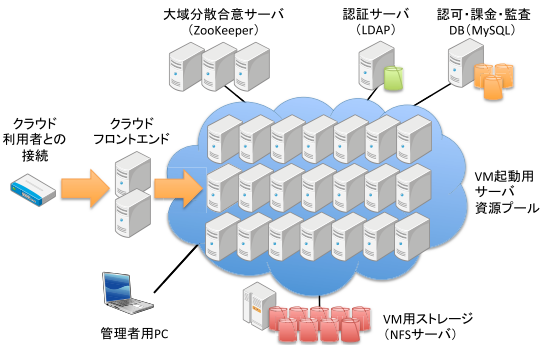


図 4　Kumoi を使用したクラウド環境
Fig. 4　A cloud environment with Kumoi.

```
val coldPool = global.createPool
coldPool.name = "gold pool"
global.add(coldPool)

val pool = global.pools(0)
pool.add(pms.filter(_.cpus >= 8 &&
                    _.memory >= 16 * 1024 * 1024 * 1024))
pool.add(pool.createElastic(vm))
```

図 5　クラウド API の使用例
Fig. 5　Example usage of cloud API.

3.3　2 種類のシェル環境 API

Kumoi はカスタマイズ性と拡張性を特徴としているため，必要に応じて 2 種類の API（Application Programming Interface）から選択して，シェル環境で利用することができる [13]．

- **ダイレクト API**：3.1 節の図 3 の記述例に対応する API で，データセンタ上の各資源をオブジェクトとして直接操作する API である．仮想資源管理基盤ソフトウェアの本質的な部分のみを抽出できるため，省電力化を目指した VM 配置アルゴリズムなどの検証に利用できる．
- **クラウド API**：3.2 節の図 4 のようなクラウド環境を構築するための API であり，資源プールや VM インスタンス数の増減による伸縮，ハイアベイラビリティなどの高水準な機能を提供する．**図 5** に本 API の記述例を示す．この例では，gold pool という名前の資源プールを作成し，8 コア以上の CPU と 16 GB 以上のメモリを持つ物理計算機を資源プールに割り当てて

る．最後に，VM を伸縮機能付きで資源プールに追加し，稼働させている．以上のように，クラウド API もさまざまな要素を分散オブジェクトとして表現し，従来からのダイレクト API の自然な拡張となっている．

本論文で提案するキャッシュ機構は，2 つの API のどちらにも共通して適用可能な手法となっている．

3.4　役割の分離

Kumoi の設計では，耐障害性を向上させるために，**状態の最小化（stateless）**というアプローチをとっている．その詳細を次に示す．

Kumoi のカーネルでは，状態の一貫性管理や障害対策を考慮して，状態を極力持たないように設計している．そのため，スクリプトから資源オブジェクトに操作を行うと，その内容をただちに実際の資源に反映する．また，資源オブジェクトの状態をスクリプトから取得しようとすると，実際の資源から状態を直接取得し，返却する．

この方式はシステムの堅牢性を高めており，万が一，Kumoi のカーネルに障害が発生しても，Kumoi カーネルのみを再起動すれば，動作中の VM などを含めた現在の計算機の状態から，資源マップオブジェクトを正しく復元することができる．

また，この設計には別の意図もあり，資源オブジェクトと RMI 実装との通信をフックすることで，性能を向上させる機構，セキュリティ機構や耐障害性機構など，さまざまな機能を追加していくことを想定している．この**役割の分離**は，システムの設計を単純化するとともに，特定の資源に依存しない共通機能の導入を可能とする．本論文では，この機構を利用してキャッシュ機構を導入する．

4.　キャッシュ機構

本論文では，IaaS 環境を想定した仮想資源管理基盤ソフトウェアにおけるキャッシュ機構を提案する．データセンタ内の情報取得の参照局所性を利用し，キャッシュ機構による性能改善効果と一貫性の両立を目指す．

具体的なキャッシュ方式は，要求する**性能改善効果**，**一貫性**の水準，および**実装の複雑さ**の 3 つの観点から適切に選択されなければならない．本論文は，これら 3 つの観点を考慮し，1 つの実装法を提案する．

4.1　概要

本論文のキャッシュ機構の概要を**図 6** に示す．説明における便宜上，RMI 通信の呼び出し側を**クライアント**，要求を処理する側を**サーバ**と呼ぶ．Kumoi における各計算機は，基本的に対等に扱われることから，どの計算機もクライアントやサーバになることができる．また，クライアントはクラウド利用者の計算機とも異なる点に注意が必要である．

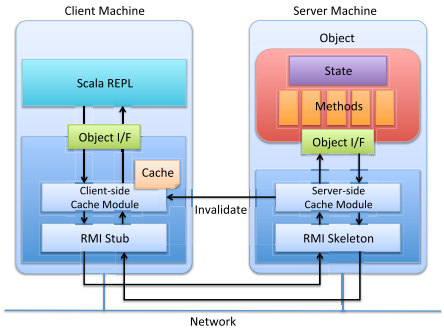


図 6　キャッシュ機構の概要
Fig. 6　Cache and RMI implementations.

図 6 のシステムは基本的に通常の Java RMI の機構と同一であるが，2 つの点で異なる．1 つ目は，クライアント側，サーバ側ともにキャッシュ機構のためのモジュールが挿入されている点である．これらのモジュールはそれぞれの RMI スタブおよびスケルトン実装の直上に位置し，通信をフックして，対応する．2 つ目は，サーバ側におけるオブジェクト実装の状態は実資源の状態に対応している点である．そのため，メモリページを利用した分散オブジェクトのキャッシュ機構 [10], [15] は実現できない．

本論文では，資源オブジェクトのキャッシュを次のような流れで実現する．

(1) まず，クライアント側のシェル環境でサーバ側の資源オブジェクトのメソッド呼び出しを開始する．
(2) 次に，クライアント側のキャッシュモジュールが，キャッシュ表のエントリを確認し，メソッド呼び出しの結果が格納されているか確認する．
(3) もし格納されていれば，キャッシュされた戻り値をシェル環境にただちに返却する．一方で，キャッシュされていない場合は，通常通りスタブを経由してリモートメソッド呼び出しを行う．
(4) サーバ側のスケルトンがメソッド呼び出しを受付けた後，サーバ側のキャッシュモジュールがその呼び出しを中継し，資源マップオブジェクトの呼び出しを行う．
(5) 戻り値を返却する際，サーバ側のキャッシュモジュールは，クライアント側にメソッド呼び出しの戻り値を中継するとともに，必要に応じてサーバ側からキャッシュエントリのインバリデーションを行う．
(6) クライアント側のキャッシュモジュールでは，呼び出しの結果をキャッシュの有効期間とともにキャッシュ表に格納し，最終的にシェル環境に呼び出し結果を返却する．

それぞれの方式の詳細について，キャッシュポリシとともに次の節から議論する．

4.2　キャッシュポリシ

本論文の提案では，通常のキャッシュ機構と同様に機構

(mechanism) とポリシを分離する．これらの分離によって，VM，ネットワーク，ストレージなどのそれぞれの資源ごとにキャッシュの性能改善効果や一貫性のバランスの調整を可能にする．また，これらのポリシは基本的に仮想資源管理基盤ソフトウェアの開発者が記述することを想定しており，クラウド管理者が記述する必要はない．

また，本論文では，キャッシュの管理に厳密な一貫性 (strict consistency) ではなく，緩やかな一貫性管理手法を採用する．他の多くの分散システムと同様に，厳密な一貫性を担保することよりも，性能改善効果を優先する．なお，一貫性が問題となる場合には，インバリデーションによりただちに最新の状態を反映することもできる．

4.3 キャッシュの登録

本論文でキャッシュの対象とするのは，**図 7** のような資源マップオブジェクトである．図 7 は物理計算機の場合の例を示している．各資源マップオブジェクトには，マシン名の取得を行う `name()` などのさまざまな情報の取得を行うメソッドを提供している．一方で，停止を行う `shutdown()` のようにそれらの情報をもとにしたフィードバック操作を行うメソッドも提供している．

資源マップオブジェクトをキャッシュする場合，さまざまな粒度でのキャッシュ方式が考えられる．それぞれの利点や欠点について，以下で議論する．

- **オブジェクト全体をキャッシュ：**オブジェクト全体をキャッシュする．キャッシュの登録や一貫性の管理に大きなコストを要するが，いったんキャッシュされてしまえば，オブジェクト内のすべての情報を効率的に取得できる．
- **オブジェクトの一部をキャッシュ：**オブジェクトで使用頻度の高い一部の情報のみをキャッシュする．キャッシュによる性能改善効果を維持しつつ，キャッシュの登録や一貫性管理のコストを軽減する．
- **メソッドごとにキャッシュ：**上記の極端な場合で，それぞれのメソッドごとに情報をキャッシュする．キャッシュ表のエントリに記載しなければならない情報は大きくなるが，キャッシュの登録や一貫性の管理コストは小さい．

```
@remote trait HotPhysicalMachine {
  @persistcache def name(): String
  @persistcache def addr(): InetAddress
  @persistcache def os(): HotOS

  @persistcache def info(): List[PMInfo]
  @cache def stats(): List[PMStat]

  @cache(3000) def cpuRatio(): Double
  @cache(3000) def cpuAvailable(): Double
  @cache(3000) def memory(): Long
  @cache(3000) def freeMemory(): Long
  @persistcache def maxMemory(): Long

  // some shortcuts
  @persistcache def vmm(): VMM
  @cache def vms(): List[HotVM]

  // update (feedback) operations
  @invalidate @nocache def shutdown()
  @invalidate @nocache def restart()
  ...
}
```

図 7　資源マップオブジェクトの例
Fig. 7　Example definition of a physical machine object.

本論文では，メソッド単位でのキャッシュ方式を採用する．その理由として，VM 配置のスケジューリングなどのデータセンタ管理のアルゴリズムでは，CPU 使用率などの特定の情報のみに関心がある場合が多いからである．また，実装上の都合もある．4.1 節で説明したように資源マップオブジェクトの状態はメモリ上にあるのではなく，実際の資源上にある．そのため，オブジェクトの状態をすべてキャッシュするためには，それぞれのメソッドを通じて実際の資源にアクセスすることが必要である．

実際のキャッシュエントリには，次のような情報を格納している．オブジェクトの参照，メソッド名，型シグネチャ（引数，戻り値の型），実引数の 4 つの組がキーであり，メソッド呼び出しの結果を値として格納している．

$$(obj, method, signature, args) \rightarrow res \tag{1}$$

さらに，現在の実装では，すべてのキャッシュ可能なメソッドが 1 度でも呼び出された段階でキャッシュに登録している．一方で，一般的なキャッシュでは，キャッシュ表の大きさに制限があり，キャッシュの登録や置き換えのアルゴリズムが問題となる．これは，タイムアウトによってエントリが削除できること，メモリの制限がそれほど厳しくないことに由来している．一般的にキャッシュ表の大きさは，空間コストと一貫性の維持コストによって決まるが，必要ならば LRU（Least Recently Used）などの置き換えアルゴリズムを導入することも可能である．

4.4 キャッシュの持続期間

キャッシュの一貫性が問題となるのは，2 つの場合である．(1) 1 つは刻一刻と資源の情報が変化する場合，(2) もう 1 つは一貫性に影響を与えるような操作を行う場合である．前者への対応を本節で説明し，後者についてを次節で説明する．

本論文では，基本的にキャッシュの一貫性管理にタイムアウト方式を用いる．タイムアウト方式を使用すると，クライアントとサーバ間でキャッシュの一貫性維持管理のための特別なプロトコルを使用する必要がなく，簡易的に実装できる．その反面，厳密に一貫性を管理することは難

しい．

キャッシュのタイムアウト時間は，仮想資源管理基盤の開発者がアノテーションを使用し，オブジェクトのメソッドごとに与えることにしている．提案機構では，以下のようなアノテーションを提供している．図 7 の例では，情報の種類に応じて，アノテーションをすでに付与している．

- `@nocache`：キャッシュを許可しない．
- `@cache(timeout)`：指定した時間だけキャッシュすることを許可する．また，キャッシュ時間はデフォルト値を使用することもできる．
- `@persistcache`：永続的にキャッシュすることを許可する．ただし，現在は 30 秒の上限を設けている．

提案機構では，キャッシュは**読み込み専用キャッシュ**として機能し，データセンタ環境に対して何らかの更新操作を行う場合にはキャッシュせず，サーバに直接操作を行う．これは，分散ファイルシステムや分散共有メモリのキャッシュと異なり，それぞれの資源の更新操作の結果を予測することが難しいためである．そのため，write-back 方式のようなキャッシュに対して操作を行い，後々，サーバ側にも反映するような他のシステムで頻繁に行われる最適化は採用できない．また，更新操作を一定期間バッファし，バッチ適用することも可能であるが，スクリプトの記述者はただちに環境に反映されることを期待していると考えられるため，こちらも行わない．

4.5　キャッシュの無効化

キャッシュでは，**時間的な順序関係**（timed order）や**因果関係**（causal order）が大きく問題となることがある．そのため，本論文ではキャッシュの一貫性を保つために，無効化（invalidation）方式を併用する．

仮想資源管理基盤で，依存関係が大きく問題になる場合として，次のような例があげられる．

- **VM を起動した直後の場合：**VM を起動すると，ただちに起動中の VM のリストに反映されることを期待するが，キャッシュのタイムアウト時間が長くとられているとなかなか反映されない状況が発生する．
- **セキュリティ上のアクセス権限を変更した場合：**アクセス権限を剥奪したものの，ただちに反映されない場合には，しばらくの間，権限を持たないものの資源にアクセスできる状況が発生する．
- **障害が発生した場合：**障害が発生した場合には，タイムアウト時間までの間，障害の発生に気がつかない．

また，キャッシュの無効化を効率的に行うために，現在，以下の 4 つの範囲での無効化を提供している．

- `InvalidateSubObject(obj, methods)`：オブジェクトの特定のメソッドのみを無効化する．たとえば，**図 8** の例のように，VM の名前を変更した場合に，`name()` メソッドや `toString()` メソッドを無効化する場合などに使用する．なお，図 8 は説明の都合上，実際のコードを単純化している．
- `InvalidateObject(obj)`：オブジェクト全体を無効化する．VM を起動する場合に，VM モニタの起動中の VM の一覧リストを更新する場合などに使用する．
- `InvalidateMulti(multi)`：複数のオブジェクトを無効化する．たとえば，図 8 の例のように，VM を移送した場合に使用する．VM の移送では，移送する VM 自身，および移送元と移送先の物理計算機および VM モニタの 5 つを無効化する．
- `InvalidateGlobal`：クライアントのキャッシュ全体を無効化する．個別に指定したのでは，状態管理が煩雑になる場合に使用する．

```
abstract class LibvirtHotVM
       extends UnicastRemoteObject with HotVM {
  def name_=(n: String) {
    // change its name to n
    invalidateSubObject(this, List("name", "toString"))
  }

  def migrateTo(dest: HotPhysicalMachine) {
    // do some migration work
    invalidateMulti(List(this, vmm, local,
                         dest.vmm, dest))
  }
}
```

図 8　キャッシュの無効化
Fig. 8　Example code of cache invalidation.

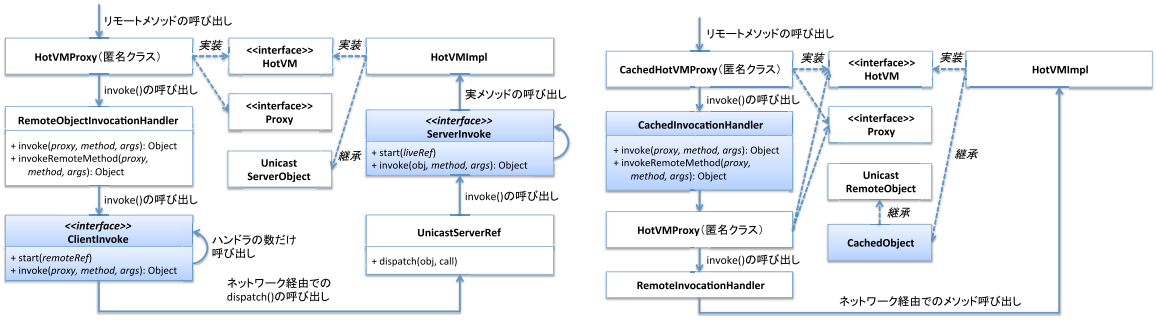
以上の 4 つの無効化は資源マップオブジェクトの実装コード中で自由に制御することができる．また，図 7 のようにアノテーションとしてこれらの情報を付与し，自動的に行わせることもできるが，1 番目と 4 番目の場合に制限される．これは，アノテーションで Java のプリミティブ型しか指定できないことに由来している．

4.6　無効化プロトコル

無効化はサーバ主導で，クライアントに向けて行う．また，複数のクライアントのキャッシュを同時に無効化することもありうる．よって，無効化を実現するためには，サーバ側でどのクライアントが情報をキャッシュしているか把握しなければいけない．

本研究では，サーバの管理情報を削減するため，サーバ側ではクライアントの RMI 通信の接続情報のみを管理し，過去に接続したすべてのクライアントに無効化メッセージを送信する．よって，無効化が該当するかどうかのフィルタリングはクライアント側で行う．しかしながら，クライアント側でキャッシュ表を探索するコストは，ハッシュ表のため小さい．結果として，無効化のコストは大きくメッセージ数に依存する．

また別の問題として，一般的に分散オブジェクトではオ



（a）Java RMI 内部に実装　　（b）Proxy 機能を利用した実装

図 9　キャッシュ機構の実装（VM オブジェクトの場合）

Fig. 9　Cache implementations.

ブジェクトの参照が別の計算機に渡され，複数の計算機から同じオブジェクトが参照されることがありうる．結果として，キャッシュの無効化は，サーバの直接のクライアントだけではなく，これらのすべての計算機において行われなければならない．

本機構では，無効化メッセージを転送させることで，この問題に対処する．現在の実装では，Time-To-Live（TTL）を使用した簡易な転送方法を使用している．TTL を使用した場合，メッセージの重複がありうるが，Kumoi では，API の設計により，オブジェクトの参照は 2 段までに限られることから，大きな問題とはならない．また，キャッシュの無効化は idempotent な操作であるため，複数回行われてもよい．最後に，これらの転送方法は，効率的なマルチキャスト方式の採用によって改善できる．

4.7　障害への対応

キャッシュは，障害によっても無効化する必要がある．障害による無効化では，下記の 2 つの場合がありうる．

- **サーバ側での無効化：**サーバ側の計算機で，資源に何らかの障害を検出した場合に無効化メッセージを送信する．
- **クライアント側の無効化：**クライアント側でも，自発的に障害を検出し，キャッシュを破棄することが必要である．これは，サーバ計算機自体が障害で停止した場合などが該当する．キャッシュ表を確認し，資源を所有していたサーバのエントリを破棄する．

上記からも分かるとおり，本研究では，複数のクライアントが独立にキャッシュの管理を行う．つまり，本研究では，複数のクライアントでのキャッシュの *coherence* は特別に管理しない．キャッシュの coherence は，タイムアウト機構と無効化機構によって一定の範囲内に保たれる．以上によって，あるクライアントが古い情報をもとに操作を行い，他のクライアントに影響を与えることも想定されるが，キャッシュを導入しない場合も通信の遅延によってこれらの問題は起こりうる．

5.　実装

本論文の実装は，Java RMI の一部を変更することで行った．現在，**図 9** の 2 つの実装方式を提供している．

最初の方式は，Java RMI 内部に実装する方式である．Java Research License に基づき，Java RMI の実装を Scala に移植し，RMI のスタブ機構，およびスケルトン機構にフックを追加した．このフック機構は，Kumoi 内部でさまざまな目的に使用しているが，本論文ではキャッシュ機構の追加に使用した．

本実装方式を，RMI の処理の流れに従って，順に説明していくと，まず，クライアント側でのオブジェクトの参照は `Proxy` で実現されている[*1]．リモートメソッドの呼び出しが開始されると，`Proxy` は `RemoteObjectInvocationHandler` の `invoke()` メソッドを呼び出す．実際のリモート呼び出しは，`invokeRemoteMethod()` が行っており，本研究では，このメソッドに `ClientInvoke` と呼ぶフックを追加し，クライアント側の実装としている．一方で，サーバ側では，`UnicastServerRef` の `dispatch()` メソッドが，実オブジェクトのメソッドを呼び出す．そこで，この関数にフックを追加し，`ServerInvoke` でサーバ側の実装を行っている．

また，別の方式として，Java RMI 内部を極力改変しない方式も実装した．`Proxy` はデザインパターンの Decorator パターンのように何段にも重ねることができることから，キャッシュ専用の `CachedInvocationHandler` を作成し，クライアント側のキャッシュ実装を追加している．また，サーバ側では `UnicastRemoteObject` を継承し，無効化機能などを追加した `CachedObject` を作成した．

双方の実装方式には，どちらも利点と欠点があり，前者は実装が複雑な反面，透過的に利用できるという特徴がある．また，後者は実装が簡単な反面，キャッシュの実装には問題がないが，サーバ側でのフックができないなど，フックできる機能に制約があるという欠点がある．以降の

[*1] RMI コンパイラ（`rmic`）は Java 5 以降，使用されていない．

実験では，前者の Java RMI 内部方式を使用している．

なお，キャッシュ機構は概念上は簡単ではあるが，内部では複雑な実装を必要とする．まず，フック機構の内部では，さまざまな情報が間接的にしか取得できないことから，リフレクション機能や Java RMI の内部構造に関する知識を活用し，メソッドの情報や RMI の接続端点などの情報を取得している．また，Java RMI はそもそも拡張するように設計されておらず，多くのクラスが `final` であり，メソッドも `private` として宣言されている．改変が必要な部分のみを Scala に移植し，元々の実装を利用できる部分についてはリフレクションでアクセス権を変更し，メソッドを呼び出すことで，必要な実装量を抑えている．Kumoi 全体のソースコード 36,276 行中の 2,017 行が RMI とキャッシュに関連した実装になっている．

キャッシュポリシのアノテーションについては，ランタイム時まで保持するよう指定することで，実行時にリフレクションを使用し，動的にチェックを行った．

6. セキュリティ機構との連携

本論文のキャッシュ方式は，Kumoi のセキュリティ機構と親和性が高く機能するように設計してある．その理由について，本章で説明する．

Kumoi のセキュリティ機能は，3.2 節の図 4 に示したように，LDAP と MySQL を使用して実装している．利用者は，まず Java の JAAS（Java Authentication and Authorization Service）機能を使用し，LDAP サーバで認証を行う．次に，その認証情報をもとに，資源マップオブジェクトのメソッド呼び出しごとにアクセス権限があるかどうか認可を行う．現在の実装では，ACL（Access Control List）を基礎としたアクセス権限の確認を行っており，MySQL サーバ上でアクセス権限があるかどうか確認し，Java のサンドボックス機能で制御している．

Kumoi の資源マップオブジェクトの階層構造は，実際の資源の階層構造を反映しており，資源の参照をたどるたびにアクセス権限のチェックを行う．**図 10** に具体例をあげて説明する．Kumoi では，ある物理計算機で現在稼働している VM の一覧を取得する場合，次のようなメソッド呼び出しを内部的に必要とする．

```
kernel.local().vmm().vms()
```

この例では，まず，`kernel` の `local()` メソッドを呼び出している．Kumoi では，カーネル自身も分散オブジェクトとして抽象化されており，`kernel` がこのオブジェクトに該当する．次に，`local()` メソッドで，シェル環境が現在接続している物理計算機に対応するオブジェクトを取得している．さらに，物理計算機の `vmm()` メソッドを呼び出すことで，物理計算機上で動作している VM モニタに対応するオブジェクトを取得している．最後に，VM モニタの

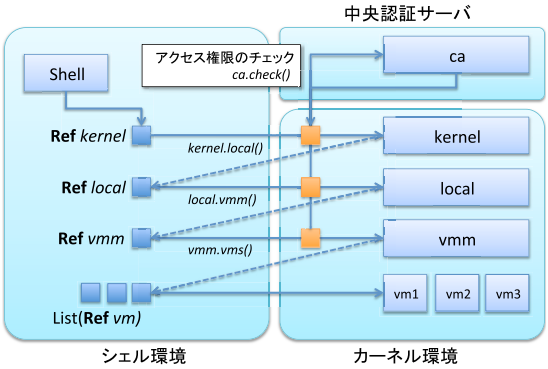


図 10　RMI と認可処理の関係
Fig. 10　RMI and authorization.

```
ca.check('Read, Kernel, kernel, "local", auth)
ca.check('Read, HotPhysicalMachine, local,
         "vmm", auth)
ca.check('Index, VMM, vmm, "vms", auth)
```

図 11　Kumoi 実装内部で行われている認可処理
Fig. 11　Internal process of authorization.

`vms()` を呼び出すことで，ようやく稼働している VM の一覧をリストで取得することができる．

以上から，Kumoi の資源マップオブジェクトも実資源の階層構造を反映していることから，VM の一覧を取得するだけでも 3 回の RMI 呼び出しを必要とする．また，アクセス権限のチェックもメソッドごとに行うことから，同様に 3 回の認可処理を必要とする．認可処理は，具体的には**図 11** のような呼び出しを 3 回に分けて行っている．

結果として，上記のような状況で，キャッシュは 2 つの理由で有効に機能することが分かる．まず，資源の階層構造は頻繁には変わらないため，オブジェクトの参照関係をキャッシュしておけば，大きく性能が改善される．次に，アクセス権限のチェックもキャッシュがあれば不要となる．アクセス権限もそれほど頻繁には変更されず，もし変更される場合にも，無効化によって最新のアクセス権限の状態をただちに反映することができる．

7. 実験

本論文の実験では，まず，キャッシュ機構の導入による操作ごとのオーバヘッドおよび性能改善効果について示す．次に，具体的な利用例を示し，スクリプト実行の中での性能改善効果について測定する．

7.1 実験環境

実験はクラスタ計算機のうち 16 台で行った．各ノードはデュアル Xeon 3.60 GHz CPU，2 GB メモリ，32 GB の SCSI ディスクで構成されている．ソフトウェアは 32 bit 版の CentOS 5.8（Linux 2.6.18），Xen 3.0.3，Sun JDK 1.6.0，Libvirt 0.8.2，Scala 2.9.2 を使用した．

表 2　操作ごとの処理時間
Table 2　Execution time per operation.

条件	1 回目	2 回目	3 回目	タイムアウト時	[ms]
キャッシュなし（ローカル）	8.08	1.48	1.50	1.64	
キャッシュなし（リモート）	14.12	2.03	2.51	4.22	
キャッシュあり（ローカル）	7.87	0.12	0.06	1.84	
キャッシュあり（リモート）	12.90	0.11	0.07	4.92	
キャッシュ手動（ローカル）	9.16	0.00	0.00	0.00	
キャッシュ手動（リモート）	12.85	0.00	0.00	0.00	

また，NFS，MySQL，LDAP のサーバとして，デュアル Xeon E5620 2.40 GHz CPU，24 GB メモリ，RAID5 ディスク（SAS 接続 600 GB を 4 台搭載）のサーバ 1 台を使用した．このサーバは，64 bit 版の CentOS 5.8，OpenLDAP 2.3.43，MySQL 5.0.95 を使用した．なお，ZooKeeper は今回の実験の範囲では不要のため，使用していない．以上の計算機はすべて，1000BASE-T で接続されている．

7.2　実験結果：操作ごとの性能改善効果

本研究では，まず，メソッド呼び出しごとのオーバヘッドや性能改善効果を測定するため，実験を行った．その結果を，**表 2** に示す．表 2 では，オーバヘッドを識別しやすくするため，できるだけ軽量な操作を選択し，カーネルが接続に使用するポート番号の取得を行っている．また，それぞれの測定はオペレーティングシステム（OS）のキャッシュができるだけ暖まった状態で測定し，各 5 回の測定結果の平均時間を示している．

表 2 では，提案機構のキャッシュを導入しない場合，提案機構を導入した場合の 2 つについて比較している．また，それぞれ，ローカルな RMI 呼び出しの場合，別の計算機に対するリモート呼び出しの場合の 2 つを示している．

キャッシュなしの場合を見ると，初回の呼び出しは呼び出し時間が大きく，2 回目以降は大きく変わらないことが分かる．また，ローカル呼び出しに比べて，リモート呼び出しの方が時間を要する．さらに，キャッシュを行わない場合にも，1 回目から 2 回目にかけて呼び出し時間が短縮されているが，関連するクラスのロード時間，Just-In-Time（JIT）コンパイラの影響や OS のキャッシュの影響などの複合的な要因が考えられる．これらは一体となり効果として現れるため，どの要因であるか特定することは難しい．

次に提案機構のキャッシュありの場合を見ると，初回の呼び出し時はキャッシュなしの場合と比べて，ほとんど変わらない時間を要する．2 回目以降は，キャッシュの効果により呼び出し時間が劇的に短縮されていることが分かる．3 回目の測定では，2 回目よりさらに短縮され，キャッシュヒット時の実行パスが最適化されていることが分かる．4 回目以降は 3 回目の測定結果とほとんど変わらないため，表 2 には掲載していない．最後に，キャッシュのタイムアウト時間が経過し，再度サーバから情報取得を行わなければならない場合には，キャッシュなしの場合の 2 回目以降の呼び出し時間と変わらない時間を要する．ただし，この場合もオーバヘッドがわずかながら存在する．

表 2 の最後に示してあるキャッシュ手動は，本機構を使用せず，下記のように利用者がスクリプト上で値を変数に束縛し，手動でキャッシュを行った場合である．

```
val p = local.port
```

この場合についても，初回はほかと変わらない処理時間を要する．2 回目以降は変数から値を取得するだけで済むため，非常に短い時間で完了する．また，提案方式にあるようなリフレクションも必要としない．ただし，手動でキャッシュを行う場合には，システム側でキャッシュの一貫性を管理することができないため，利用者が手動で管理する必要がある．

7.3　実験結果：アクセス認可確認を有効にした場合

これまでの Kumoi では，大きなオーバヘッドを要するため，セキュリティ上の認可機能を標準で有効にしてこなかった．しかしながら，キャッシュ機構を導入することで，これらの機能を現実的に利用することができるようになる．

表 3 に認可機能を有効にした場合の実験結果を示す．この実験も，表 2 の実験と同様にカーネルが使用するポート番号を取得する場合の結果を示している．

表 3 を見ると，キャッシュなしの場合には，メソッド呼び出しにかかる時間が大幅に増加している．これは，アクセス認可の確認にかかる時間が大きく占めているからである．一方で提案機構を有効にした場合は，2 回目以降，認可機能を有効にする前と変わらない所要時間を示している．これは，キャッシュが有効に機能し，認可機能の導入による性能低下を抑えているからである．そのため，次節以降の実験では，認可機構を有効にした状態で実験を行う．なお，手動キャッシュを行った場合については，初回はほかと同じような時間を要し，2 回目以降は最も短い時間で情報取得を行うことができる．ただし，この場合も利用者が一貫性の管理を行う必要がある．

なお，この場合についてもキャッシュなしの 1 回目から

表 3　アクセス認可確認を有効にした場合の処理時間
Table 3　Execution times when authorization was enabled.

条件	1 回目	2 回目	3 回目	タイムアウト時 [ms]
キャッシュなし（ローカル）	91.12	53.56	47.51	39.89
キャッシュなし（リモート）	208.55	61.17	61.31	56.70
キャッシュあり（ローカル）	108.55	0.11	0.06	58.38
キャッシュあり（リモート）	200.54	0.10	0.06	78.79
キャッシュ手動（ローカル）	110.83	0.00	0.00	0.00
キャッシュ手動（リモート）	212.63	0.00	0.00	0.00

表 4　各モジュールごとの処理時間
Table 4　Execution time for each component.

モジュール		1 回目	2 回目	3 回目	タイムアウト
Client	(1) Precall	2.28	0.04	0.03	0.05
	(2) Cache	0.17	1.70	0.05	0.08
	(3) Security	0.01	–	–	0.00
	(4) RMI	4.14	–	–	1.93
Server	(5) Cache	0.01	–	–	0.01
	(6) Security	65.30	–	–	44.42
	(7) Impl.	0.03	–	–	0.03
	(8) Security	16.86	–	–	15.17
	(9) Cache	0.00	–	–	0.00
	(10) RMI	0.43	–	–	0.34
Client	(11) Security	0.00	–	–	0.00
	(12) Cache	1.69	–	–	0.11
	(13) Postcall	0.02	0.01	0.00	0.00
合計		90.96	1.74	0.09	62.13 [ms]

3 回目にかけて，キャッシュありの 2 回目から 3 回目にかけて性能が改善されている．これも，7.2 節の場合と同様に，OS キャッシュなどの複合的な要因が考えられる．

7.4　実験結果：所要時間の分解

表 4 に前節のカーネルの接続ポートを取得する場合の処理時間の内訳を示す．認可機能およびキャッシュ機能の双方を有効にし，時間計測の関係からローカル呼び出しの場合のみで実験を行っている．表 4 では，メソッドを呼び出してから，呼び出し元に戻ってくるまでの時間を，上から下に各段階ごとに示している．

表 4 を見ると，まだキャッシュされていない 1 回目の場合では，まず，最初の（1）呼び出し前処理で時間を要していることが分かる．これは，ロード処理，JIT コンパイラ，OS キャッシュなどの複合的な要因が考えられる．また，サーバ側の（6），（8）セキュリティに関する処理が多くの時間を占めている．これは，（7）実メソッドの呼び出し前では，認可処理，呼び出し後は監査処理を行っているからである．呼び出しの戻りでの（12）キャッシュ処理では，キャッシュ表に登録する処理のため，少し時間を要する．

2 回目と 3 回目では，キャッシュの効果により，多くの処理を省略できていることが分かる．また，（2）キャッシュ処理の時間が 2 回目より 3 回目の方が短くなっている．この要因について調査するのは容易ではないが，7.2 節などと同様に OS キャッシュなどの外部要因が考えられる．

なお，タイムアウトの場合では，すべての処理を必要とするが，各段階での実行パスが最適化されているため，1 回目と比べて短い時間で完了する．

```
1:  def compact(pms: List[HotPhysicalMachine]) {
2:    def firstFit(v: HotVM,
3:                 rest: List[HotPhysicalMachine]) {
4:      rest match {
5:        case h :: rs if h.cpuAvailable > v.cpuRatio &&
6:          (v.maxMemory + h.memory) < h.maxMemory =>
7:            v.migrateTo(h)
8:        case h :: rs => firstFit(v, rs)
9:        case List() =>
10:     }
11:   }
12:   def compacti(pms: List[HotPhysicalMachine]) {
13:     pms match {
14:       case h :: rest =>
15:         h.vms.foreach(v =>
16:              firstFit(v, rest.reverse))
17:         compacti(rest)
18:       case List() =>
19:     }
20:   }
21:   compacti(pms.reverse)
22: }
```

図 12　VM 集約スクリプトの例
Fig. 12　Example script of VM compaction.

7.5　実験結果：スクリプトでのキャッシュ効果

これまでの実験結果では，キャッシュの導入によって操作ごとの性能改善効果が見られることを確認してきた．本節では，実際のスクリプト実行時におけるキャッシュ効果を確認する．また，キャッシュの無効化を利用することで，一貫性が適切に維持されることを確認する．

本実験では，**図 12** のようなスクリプトを使用する．このスクリプトは，VM の配置を圧縮するスクリプトであり，VM を 1 台ずつ移送可能か確認しながら，少ない計算機台数に VM を First-Fit 方式で集約していく．

図 13 は，本スクリプトを実行した場合の `firstFirst()` 関数の各呼び出しごとの処理時間を示している．この関数

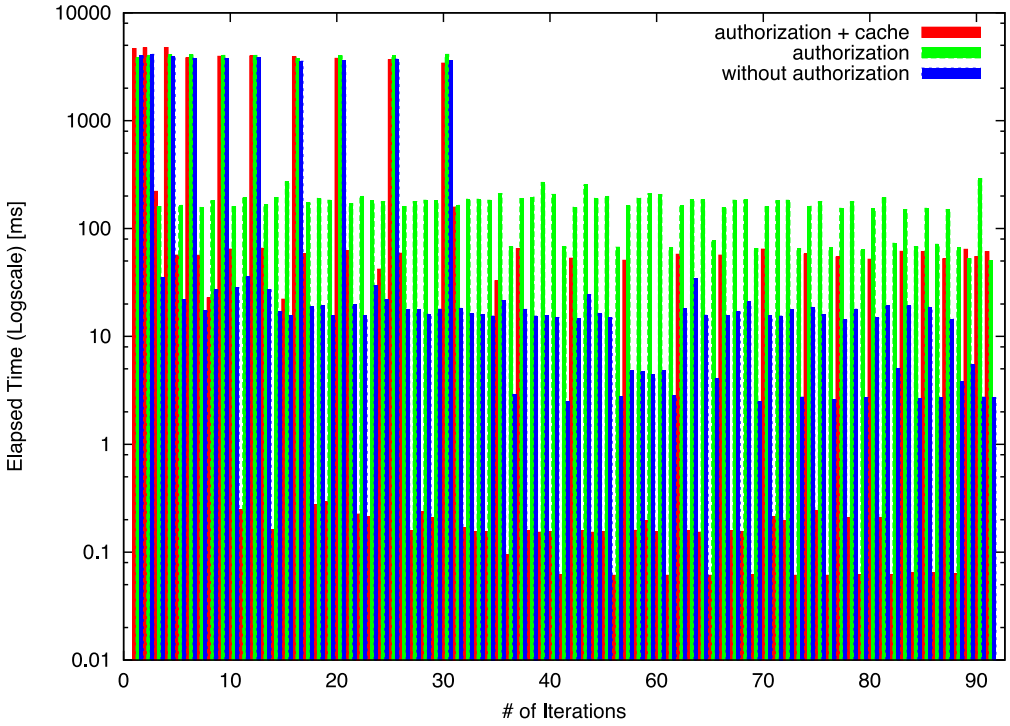


図 13　スクリプト実行時のキャッシュ効果
Fig. 13　Cache effects in script execution.

```
kumoi> compact(pms)
kumoi> pms.map(p => p.name -> p.vms)
res7: List[(String, List[kumoi.shell.vm.HotVM])] = List((ibm1,List(centos6-ibm1, centos6-ibm16, centos6-ibm15)),
(ibm2,List(centos6-ibm2, centos6-ibm14, centos6-ibm13)), (ibm3,List(centos6-ibm3, centos6-ibm12, centos6-ibm11)),
(ibm4,List(centos6-ibm4, centos6-ibm10, centos6-ibm9)), (ibm5,List(centos6-ibm5, centos6-ibm8, centos6-ibm7)),
(ibm6,List(centos6-ibm6)), (ibm7,List()), (ibm8,List()), (ibm9,List()), (ibm10,List()), (ibm11,List()),
(ibm12,List()), (ibm13,List()), (ibm14,List()), (ibm15,List()), (ibm16,List()))
```

図 14　キャッシュの無効化ありの場合
Fig. 14　Result of VM compaction (Invalidation was enabled).

は合計 81 回呼び出されており，横軸は呼び出し回数，縦軸は処理時間を示している．なお，さまざまな要因による影響を分かりやすくするため，縦軸は対数表示としている．

図 13 では，(a) キャッシュと認可の双方を有効にした場合と，比較のため (b) 認可のみを有効にした場合，(c) キャッシュと認可の双方を無効にした場合の 3 つの結果を示している．まず，どの方式も 4 秒程度の大きな立ち上がりがあることが分かる．これは，VM 移送を行った場合であり，本実験では合計 10 回の VM 移送が行われている．次に，認可機能のみを有効にした場合には，それ以外の場合で 100 [ms] 前後の時間を要していることが分かる．(c) の双方を無効にした場合も，RMI 通信により，(b) ほどではないが，数十 [ms] 程度の時間を毎回要していることが分かる．一方で，(c) の提案方式のキャッシュを行う場合には，キャッシュが無効化された場合や VM 移送を行う場合のみ時間を要するが，それ以外の場合ではキャッシュが有効に機能し，処理時間が短くなっていることが分かる．

次に，キャッシュの無効化の有効性を確認する．無効化を行った場合を**図 14**，無効化を行わない場合の結果を**図 15** に示す．無効化を行った場合には，VM が 3 台ずつ適切に集約されていることが分かる．本実験の VM は CPU をほとんど使用しないため，メモリの使用量のみで配置が決まる．一方で，無効化を行わなかった場合には，古い移送前のメモリ使用量の情報をもとに移送先を決めてしまい，VM が集約されすぎ，途中で失敗していることが分かる．図 12 のスクリプトで無効化は，`v.migrateTo(h)` の VM 移送のタイミングで行われており，無効化が必要であることが分かる．

現在の無効化プロトコルは最適化されたものではないが，本実験の状況では次のように通信が行われる．ある計算機で無効化を行う場合，残りの 15 台の計算機とは直接接続されており，シェル環境のみが無効化プロトコルの 2 段目に位置するため，転送を必要とする．また，現在の実装ではクライアント側の障害を考えて，無効化の完了を待

```
kumoi> compact(pms)
java.rmi.UnexpectedException: unexpected exception; nested exception is:
   ...
Caused by: org.libvirt.LibvirtException: POST operation failed: xend_post: error from xen daemon:
(xend.err '/usr/lib/xen/bin/xc_save 19 28 0 0 1 failed')
   at org.libvirt.ErrorHandler.processError(Unknown Source)
    ...
kumoi> pms.map(p => p.name -> p.vms)
res6: List[(String, List[kumoi.shell.vm.HotVM])] = List((ibm1,List(centos6-ibm1, centos6-ibm16, centos6-ibm15,
centos6-ibm14, centos6-ibm13, centos6-ibm12)), (ibm2,List(centos6-ibm2)), (ibm3,List(centos6-ibm3)),
(ibm4,List(centos6-ibm4)), (ibm5,List(centos6-ibm5)), (ibm6,List(centos6-ibm6)), (ibm7,List(centos6-ibm7)),
(ibm8,List(centos6-ibm8)), (ibm9,List(centos6-ibm9)), (ibm10,List(centos6-ibm10)), (ibm11,List(centos6-ibm11)),
(ibm12,List()), (ibm13,List()), (ibm14,List()), (ibm15,List()), (ibm16,List()))
```

図 15　キャッシュの無効化なしの場合
Fig. 15　Result of VM compaction (Invalidation was disabled).

たないようにしているため，無効化を行うサーバ側から見た場合，4.05 [ms] で無効化の要求処理が完了する．

8. 関連研究

8.1 キャッシュの実現方式

キャッシュは，Domain Name System などでの利用を代表例として，さまざまな分散システムで用いられており，古くから研究が進められている [16]．キャッシュは，性能を向上させる代わりに一貫性の管理コストや可用性が問題となることから，さまざまな方式が提案されている．また，キャッシュは複製（replication）の特殊な場合だと考えられ，クライアント主導で行われる点が複製と異なる．

分散オブジェクトのキャッシュ方式も古くから研究が行われており [17], [18]，特に Java RMI では文献 [8], [9], [10], [11], [19] などの研究がある．本論文の資源の分散オブジェクト化のアプローチを用いれば，基本的にこれらの研究成果を仮想資源管理基盤の環境に適用することができる．ただし，対象とする仮想資源管理基盤環境の特性に応じてキャッシュ方式を取捨選択している点が，これらの研究との違いである．特に，資源マップオブジェクトの状態を持たず，実資源の状態に直接対応させているため，従来の研究で見られた，メモリページを活用した実装の最適化 [10], [15] は用いることができない．また，クライアント側のキャッシュを読み書き可能とし，後にサーバ側のオブジェクトに反映する方式 [15] も，それぞれの資源のメソッド呼び出しの結果が予測しにくいことから，適用しにくいと考えられる．

また，別のアプローチとして，キャッシュなどの機構をアスペクト指向プログラミングの手法を用いて導入することもできる．ただし，本論文の対象とする環境では，資源の抽象化によって，すでに統一的なインタフェースが与えられているため，アスペクト指向の研究で取り扱われている問題と比べて，些細な問題であるといえる．

8.2 仮想資源管理基盤へのキャッシュの導入

仮想資源管理基盤に限らなくても，以前より，データセンタでは，SNMP（Simple Network Management Protocol）などを使用した情報収集が行われている．いくつかの SNMP のエージェント実装でも，キャッシュの導入によって，情報を収集する際の効率性を高めている．本研究の仮想資源管理基盤が収集する情報も，これらの機構によって集められる情報と重複している部分もある．一方で，本研究では，これらの情報をもとにフィードバック操作を繰り返すことを想定しており，より動的な利用を想定している．

また，仮想資源管理基盤におけるキャッシュ手法は，VMware Infrastructure [2], Eucalyptus [5], OpenNebula [3], OpenStack [6], CloudStack [7] などの他の仮想資源管理基盤に対しても導入することが可能である．ただし，これらの基盤ソフトウェアでは本研究ほど資源の抽象化を徹底して行っていないと考えられることから，キャッシュ機構の導入には，より多くのコストを要する．

最後に，本研究で使用している個別の手法自身は特に目新しくないが，仮想資源管理基盤の利用形態を想定し，必要な機能を組み合わせて実現している点で従来の研究と異なると考えられる．

9. まとめ

本論文では，クラウド環境を想定した仮想資源管理基盤におけるキャッシュ機構を提案した．他の一般的なキャッシュ機構と同様に参照の局所性を利用し，管理操作における情報取得の効率性と一貫性の両立を目指す．本機構が対象とする仮想資源管理基盤ソフトウェアは，さまざまな資源を分散オブジェクトとして抽象化し，統一的に扱うため，分散オブジェクトのキャッシュ機構をもとに実装を行うことができる．また，キャッシュ時のポリシとしてタイムアウトによる保持期間の指定と，インバリデーションによる無効化を行った．

今後の予定として．実際に近い環境で運用を行うことによって，さらなるキャッシュ機構の有効性を確認する．その際に，過去の文脈で行われたさまざまなキャッシュ方式が導入可能であるが，実装の複雑さと性能および一貫性のトレードオフから，どこまでの方式を導入していくべきかという検討が必要である．

謝辞　本研究の一部は科研費（2230006，22700023）の支援を受けている．

参考文献

[1] Barroso, L.A. and Hölzle, U.: *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan & Claypool (2009).

[2] VMware Inc.: VMware Infrastructure (1998), available from ⟨http://www.vmware.com/products/vi/⟩.

[3] OpenNebula Project Leads: OpenNebula: The Open Source Toolkit for Cloud Computing (2008), available from ⟨http://www.opennebula.org/⟩.

[4] Sotomayor, B., Montero, R.S., Llorente, I.M. and Foster, I.: An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds, *IEEE Internet Computing*, Vol.13, No.5, pp.14–22 (2009).

[5] Nurmi, D. et al.: The Eucalyptus Open-Source Cloud-Computing System, *IEEE/ACM CCGrid'09*, pp.18–21 (2009).

[6] Rackspace Cloud Computing: OpenStack: The Open Source, Open Standards Cloud (2010), available from ⟨http://openstack.org/⟩.

[7] Apache Software Foundation: CloudStack (2012), available from ⟨http://cloudstack.org/software.html⟩.

[8] Eberhard, J. and Tripathi, A.: Mechanisms for Object Caching in Distributed Applications using Java RMI, *Software: Practice and Experience*, Vol.37, No.8, pp.799–831 (2007).

[9] Krishnaswamy, V. et al.: Efficient Implementation of Java Remote Method Invocation (RMI), *4th USENIX COOTS* (1998).

[10] Aridor, Y. et al.: A High Performance Cluster JVM Presenting a Pure Single System Image, *ACM Java Grande Conf. '00*, pp.168–177 (2000).

[11] Lipkind, I., Pechtchanski, I. and Karamcheti, V.: Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations, *14th ACM SIGPALAN OOPSLA* (1999).

[12] Sugiki, A. et al.: Kumoi: A High-Level Scripting Environment for Collective Virtual Machines, *IEEE ICPADS 2010*, pp.322–329 (2010).

[13] Sugiki, A. and Kato, K.: An Extensible Cloud Platform Inspired by Operating Systems, *IEEE/ACM UCC 2011* (*Short paper*), pp.306–311 (2011).

[14] Odersky, M.: The Scala Programming Language (2003), available from ⟨http://www.scala-lang.org/⟩.

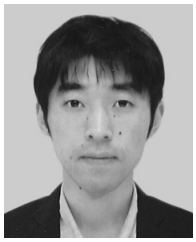
[15] Liskov, B., Castro, M., Shrira, L. and Adya, A.: Providing Persistent Objects in Distributed Systems, *13th ECOOP*, pp.15–35 (1999).

[16] Tanenbaum, A.S. and Steen, M.V.: *Distributed Systems: Principles and Paradigms*, Prentice Hall (2006).

[17] Tari, Z., Hamidjaja, H. and Lin, Q.T.: Cache Management in CORBA Distributed Object System, *IEEE Concurrency*, Vol.8, No.3, pp.48–55 (2000).

[18] Chockler, G.V., Dolev, D., Friedman, R. and Vitenberg, R.: Implementing a Cache Service for Distributed CORBA Objects, *IFIP/ACM Middleware '00*, pp.1–23 (2000).

[19] Krishnaswamy, V. et al.: Distributed Object Implementations for Interactive Applications, *IFIP/ACM Middleware '00*, pp.45–70 (2000).



杉木 章義　（正会員）

2002 年電気通信大学情報工学科卒業．2004 年同大学大学院情報工学専攻博士前期課程修了．2007 年同博士後期課程修了．博士（工学）．2007 年科学技術振興機構 CREST 研究員，2009 年筑波大学システム情報工学研究科助教．分散システム，オペレーティングシステムの研究に従事．2009 年度山下記念賞受賞．日本ソフトウェア科学会，ACM，IEEE-CS，USENIX 各会員．

加藤 和彦　（正会員）

1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，2004 年筑波大学大学院システム情報工学研究科教授，現在に至る．1992 年博士（理学）（東京大学）．オペレーティングシステム，分散システム，仮想計算環境，セキュリティに興味を持つ．1990 年情報処理学会学術奨励賞，1992 年同研究賞，2005 年同論文賞，2004 年日本ソフトウェア科学会論文賞，各受賞．