Vol.2013-OS-126 No.1
2013/7/31

ページキャッシュ最適化手法における ベイズ的アプローチの検討

高橋 一志[1,2,a)]　佐々木 慎[1,b)]　松宮 遼[1,c)]　大山 恵弘[1,2,d)]

概要：本稿では，アプリケーションのディスクリードパターンを解析し，各々のアプリケーションのリード特性に応じてキャッシュを配置することで，キャッシュヒット率向上を狙う手法について報告する．ネットワークファイルシステムにおいてキャッシュを最大限に利用することは，高速にアプリケーションを実行するために重要である．安価で大容量かつ高速な RAM や，超高速なネットワーク機器が普及するにつれ，二次記憶装置へのアクセスはネットワークファイルシステムのボトルネックとして無視できないものになりつつある．可能な限り二次記憶装置へのアクセスを低減するために，ネットワーク越しのノード間上にてキャッシュを融通し合う Co-operative Caching と呼ばれる機構をうまく活用してゆくことが，今後のネットワークファイルシステム高速化の鍵であるとわれわれは考えている．本稿では，アプリケーションのリードパターンをベイズ統計の手法で解析し，得た情報を元に，似たようなリードパターンが発生するアプリケーション同士をクラスタリングすることで，ネットワーク上に存在する多数のノードの中から，目的のキャッシュを持つノードを素早く発見する手法について検討する．

A cache node look-up method using the Bayesian approach

Abstract: In this paper, we proposed a means to improve cache hit rate by the cache block searching that is decided by analysing disk reading pattern of applications. It is important to take full advantage of file cache in network file system to run application faster. As inexpensive fast RAM and high-speed network device widely spreading, bottleneck of storage device is real and substantive problem. To reduce number of storage device access, we consider Co-operative Caching that provides cache control among nodes on network would be a key technique to achieve high speed network file system. We propose a Bayesian cache node look-up method for "Co-operative Caching" by discovering read pattern of application similarity among several nodes on network.

1. はじめに

10-gigabit Ethernet や InfiniBand といった高速なネットワーク環境が普及している．これらの高速なインタフェイスで構築された LAN 上でのデータ転送速度は，最新鋭のディスクドライブとくらべて二桁以上高速である．近い将来には，このような高速インタフェイスがより普及し，より多くのクラウド・コンピューティング環境や分散ファイルシステムを構成するノードがこれらのインタフェイスで接続されると予想される．

これらの分散システム上で動作する I/O intensive なアプリケーションの高速化を考えるためには，単一ノード内で完結しないキャッシュ戦略が重要である．不特定多数の人々が利用する分散システムの場合，例えばノード A がデータ X を読み出そうとした場合，そのデータ X が，それ以前に他のノード B によって読み出され，メモリ上にロードされキャッシュされている可能性も十分考えられる．ノード A はそのキャッシュを高速ネットワーク経由で転送したほうが，自身のノードが持つ HDD へとデータを読みに行くよりもはるかに高速にデータ X をフェッチすることができる．結果，システム全体のディスクアクセス回数は減少し，アプリケーションの高速化を達成することが可能である．

このような仕組みは *Co-operative Cache*[1] と呼ばれ，古

[1] 電気通信大学
The University Electro-Communications
[2] 独立行政法人科学技術振興機構, CREST
JST, CREST
[a)] kazushi@inf.uec.ac.jp
[b)] sasashin@ol.inf.uec.ac.jp
[c)] r.matsumiya@ol.inf.uec.ac.jp
[d)] oyama@inf.uec.ac.jp

くからネットワークファイルシステム上における全体のディスクアクセスの減少を行うための手法として開発されてきた．上述したように，10-gigabit Ethernet や InfiniBand といった，10 年前とは比較にならないほど高速なネットワークが普及しつつあることを鑑みると，Co-operative Cache の重要性は今後ますます高まると推測される．

Co-operative Cache を実現するための重要な機構の一つに，多数存在するノードの中から目的のキャッシュを持つノードを見つけ出すための *block locating* と呼ばれる機構がある．分散システムの場合，システムは多数のノード（計算機）から構成されている．そのため，これら多数のノードから目的のキャッシュを探し出す必要があり，この block-locating は Co-operative Cache を実現する上で無くてはならないものである．

本稿では，Co-operative Cache における新たな block locating の手法を提案する．**図 1** に今回提案する手法についての概念図を示す．これは 6 つのノード $n_{1...6}$ を持つ分散ファイルシステムである．それぞれのノードは独立した計算機でもあり，ハードドライブ，メモリや CPU を持つ．図 1 に示すように n_4 上にプロセス **Proc E** がデプロイされたとする．Proc E はデータ（図中の緑の円盤）を読み出したいとする．この時 n_4 上ではディスクを読む前に（あるいは同時に）n_4 のオンメモリ上にある表（図中左）を参照する．この表には n_4 からみて他のノードがどれくらい自分と同じキャッシュブロックを読んでいるかを示す指標値（類似度）が記録される．この値は距離として捉えることもできる．そのため，0 に近ければ近いほど，自分と同じデータを読んでいる．類似度が高い，すなわち，自分が欲するデータがキャッシュに乗っている可能性が高いノードである．この表は定期的に更新されることになる．図中では n_2 に続いて n_1 がもっとも自分に近しい存在であると観測されている．そのため，この二つのノードがキャッシュを持っている可能性が高いと判断し，これらのノードに問い合わせを行う．これが本稿で提案する手法である．

まとめると，今回提案する手法は，類似度の高いノードを探し出し，類似度が高い（距離が小さい）ノードに対してキャッシュブロックの問い合わせを行う手法である．

類似度の判定にはいくつかの手法が考えられるが，今回はベイズ統計学的手法を用いた類似度を提案する．ノード上でアプリケーションがどのようなデータをどのようなパターンで読んでいるかをベイズ統計の手法を用いて解析し，そのノードと比較してよく似たデータを読んでいるノードを探しだす．ベイズの定理を式 1 に示す．

$$Pr(\theta|X) = \frac{Pr(X|\theta)Pr(\theta)}{Pr(X)} \qquad (1)$$

具体的に言うと，$\theta \in [0,1]$ であり，これは特定のキャッシュブロックが，特定のノード内部で呼ばれる確率である．

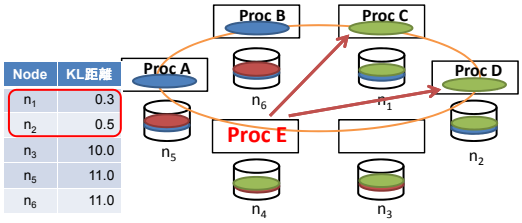


図 1 システムの概要図
Fig. 1 System overview

ノード N_j 上で，どのキャッシュブロックが何回読まれたのかを X_j と定める．すると，$Pr(\theta|X_j)$ は，ノード N_j 上で X_j が観測された後の事後分布となる．この時，すべてのノードで読まれるキャッシュブロックに SHA-1 のような暗号学的ハッシュ関数をかけ，出てきたハッシュ値を辞書順に並べた上で 1...k の番号を振ってゆく．つまり，あるノードが k 個あるデータブロックをディスクから読んでメモリにロードするという事象を，人間が k 面を持つサイコロをふるという行為であると捉え直す．すると，キャッシュリードという事象を，k 面のサイコロを n 回ふるという独立試行において，k 面サイコロの各々の目が特定の回数出たという事象が起きる確率はいかほどになるかという数学的な問題に置き換えることができる．すると，尤度関数として多項分布が仮定でき，そこからベイズの定理を適応し，事後分布 $Pr(\theta|X_j)$ は単純な共役事前分布の計算でディリクレ分布となる．$Pr(X|\theta) \sim Multinomial_k(n,\theta)$ かつ $Pr(\theta) \sim Dir(\alpha)$ の時，$Pr(\theta|X)$ はディリクレ分布となる．得られた事後分布 $Pr(\theta|X_j)$ には，ノード N_j のキャッシュブロックのリードを観測していた観測者の，特定のキャッシュブロックがどの程度読まれるのかについての信念 (*belief*) がここにエンコードされているとみなすことができる．

各ノードは 1 つずつ事後分布 $Pr(\theta|X)$ を持つことになる．そして，各ノードは，自身の事後分布と他のノードの事後分布の値の距離（類似度）を算出する．相対的に最も 0 に近いノードが，同じようなデータを同じようなパターンで読んでいると判定する．確率分布同士の距離を算出するためには Kullback-Leibler 情報量（KL 情報量, KL 距離）[*1] を選択するのが自然である．

最終的には図 1 の左に示されたような，自身から見た他のノードまでの距離の表を得ることができる．後は，上述したとおり，相対的に見て 0 に近いノードが自分と似たようなデータを似たようなパターンでリードを行なっているノードであると判断し，キャッシュの問い合わせを行う．

VM（仮想マシン）を利用して簡単な実験環境を構築し本手法の評価を行った．5 つの異なるリードパタンでデータを読む簡単なベンチマークプログラムを用意しそれらを

[*1] KL 距離は対称性と三角不等式を満たさないため，われわれが日常的に使用する「距離」の概念とはやや異なった性質を示す

5 つの VM 上で走らせた．その上で，VMM（仮想マシンモニタ）側から VM のページリードを観測して KL 情報量を算出した．新しいキャッシュのチャンクが入ってくるたびに，後ろのキャッシュを追い出すキューで管理していると仮定した時，キャッシュがヒットしそうな，似たようなデータを似たようなパターンで読んでいる VM の KL 距離が，全てではないが，近くなったことを確かめることが出来た．

2. アルゴリズムの詳細と想定環境

本研究は，Co-operative Cache 機構において，多数のノードの中から目的のキャッシュを持つノードを見つけ出す *block locating* と呼ばれる機構に関する新たな提案を行うものである．キャッシュのノードを見つけたその後はキャッシュをフェッチする機構も当然必要であるが，キャッシュのフェッチは本研究では扱わず対象外である．本研究はあくまで，新たな block locating アルゴリズムの提案に注力している．

本稿にて提案するアルゴリズムは分散ファイルシステム Gfarm[2] と IaaS 型（DaaS 型）クラウド基盤に対して有効に活用できるのではないかと分析している，しかし，それら二つの想定環境について詳しく述べる前に，まずは，提案アルゴリズムを一般化して詳しく述べる．その後，提案したアルゴリズムが Gfarm や IaaS（DaaS）型クラウド基盤に対してどのように適用できるのかについて詳しく論じてゆく．

2.1 提案アルゴリズムの詳細

はじめに定義を明確にする．本手法を実現する上で必要な登場人物は以下のとおりである．コンダクター : $\mathcal{C}$, プローブ : p_j, そしてノード : N_j，ノード上に配置されるディスクリードを行うアプリケーション job_j, 存在するキャッシュの識別子がすべてが格納されているグローバルな辞書 : U と，ノード N_j ごとにある時間 V の間に読まれたキャッシュの識別子を記録する辞書 : X_j が存在する．U はひとつしか存在しないが，X_j はノードの数だけ存在する．U に含まれる識別子の数と X_j に含まれる識別子の数は同一であると定義する．つまり，すべての j に対して必ず $|U| = |X_j|$ である．U に存在する識別子は必ずすべての X_j の中に存在しており，逆に，X_j の中に存在する識別子も必ず U に存在するものとする．なお，ここで変数 k を定め $k = |U| = |X_j|$ とする．キャッシュはすべて固定長のデータチャンクとして扱われる．キャッシュの識別子は，存在するすべてのチャンクを入力として，SHA-1 といった暗号学的ハッシュ関数で生成されたハッシュ値を辞書順にならべ，番号を振ったものとする．U は辞書であるため同一の識別子が二つ以上存在することはない．U に含まれる識別子はすべてユニークである．プローブ p_i はノード N_j に 1 つずつ配置される．N_j は計算機でありディスクとメモリを持ち，この上でアプリケーションである job_j が動作する．p_i はノード N_j 上のすべてのディスクリードを監視している．p_j は区間 V の間に N_j が読んだデータ（チャンク単位）に SHA-1 のハッシュを計算した上で X_j を参照し，ノードごとにどの識別子のチャンクが何回読まれたかを記録する．

本稿で提案するアルゴリズムは，ベイズ統計の手法をつかってノード N_j を分析し，あるノード N_j からみた時，他のノードがどの程度似通っているのか，どの程度同じデータを読んでいるのかを距離として算出する．自分から見て他のノードまでの距離をすべて算出した上で比較を行い，最も 0 に近い値を持つノードが，現存するすべてのノードの中で，似たようなデータのチャンク列を，似たようなパターンで読でいるということがわかる．そのため，自分の欲するデータがキャッシュとして保持されている可能性が高いと判断して，キャッシュデータを読みに行く．

ベイズ統計を使って分析を行い最終的にノード間の距離を算出するためには，まず尤度関数を定める必要がある．そのためには，ディスクをリードしてキャッシュにのせるという事象を何らかの数学的な問題，数学的なモデルに置き換えて考える必要がある．

ここではノードのディスクを読むという事象を表す尤度関数として，多項分布を利用することを提案する．つまり，ノードがディスクを読むという事象を，人間が k 面のサイコロをふるという事象に置き換えるのである．すべての j に対して $k = |X_j| = |U|$ であるため，各ノードが独立してディスクを読んでいるという事象を，人々が独立して k 面サイコロを転がしているという事象に読み替えるのである．すると，尤度関数として多項分布を仮定し，k 面のサイコロを転がし，出た目を記録した上でサイコロの各目が出る事後確率を推定するというベイズ統計の一般的な問題に帰着させることができる．ここでの目的は事後確率を推定することではなく，得られた事後分布の距離を利用するわけであるが，考え方は同じである．

一定の期間内 V に，ノード j が k 面の目を持つサイコロを n 回振り，X_j をどの目が何回出たかという一次元ベクトルの確率変数と読み替えた時．定義 2.1 が成り立つ．

定義 2.1. $X_j = (x_{j1}, x_{j2}, ..., x_{jk})$ とした時，確率変数 X_j は n と θ をパラメタとする多項分布 $Multinomial_k(n, \theta)$ に従うとする．この多項分布の確率密度関数は

$$Pr(x_{j1}, x_{j2}, ..., x_{jk}|n, \theta) = \frac{n!}{x_{j1}!, x_{j2}!...x_{jk}!}\theta_1^{x_{j1}}...\theta_k^{x_{jk}}$$

と定義できる．

つまり

$$X_j \sim Pr(X_j|\theta) = Multinomial_k(n, \theta),\ \theta \in [0, 1] \quad (2)$$

である．

ここで，ベイズの定理を適応すると，事後分布 $Pr(\theta|X_j)$ は単純な共役事前分布の計算でディリクレ分布となる．つまり，$Pr(\theta) \sim Dir(\alpha)$ の時，$Pr(\theta|X)$ もまたディリクレ分布となる．

Proof. もし，$Pr(X|\theta) \sim Multinomial_k(n,\theta)$ かつ $Pr(\theta) \sim Dir(\alpha)$ の時，$Pr(\theta|X)$ もディリクレ分布となる．

$$\begin{aligned} Pr(\theta|X) &= \gamma Pr(X|\theta)Pr(\theta) \\ &= \gamma\left(\frac{n!}{x_1!x_2!...x_k!}\prod_{i=1}^{k}\theta_i^{x_i}\right) \\ &\left(\frac{\Gamma(a_1+...+a_k)}{\prod_{i=1}^{k}\Gamma(\alpha_i)}\prod_{i=1}^{k}\theta_i^{\alpha_i-1}\right) \\ &= \tilde{\gamma}\prod_{i=1}^{k}\theta_i^{\alpha_i+x_i-1} \\ &= Dir(\alpha+x) \end{aligned}$$ □

したがって，$Pr(\theta|X) \sim Dir(\alpha+x)$

α は事前分布のディリクレ分布を表すパラメタである．ベイズ統計では事前分布としてどのような分布を仮定するのかがとても重要であり，要である．本稿では共役事前分布を使用しているため，一様分布を仮定する．すなわち $\alpha = 1$ と定める．

定義 2.2. ディリクレ分布同士における KL 情報量の定義 $q(\pi) = Dir(\pi;\lambda_q)$, $p(\pi) = Dir(\pi;\lambda_p)$ として

$$\begin{aligned} KL_{Dir}(\lambda_q;\lambda_p) &= \log\frac{\Gamma(\lambda_{qt})}{\Gamma(\lambda_{pt})} + \sum_{s=1}^{m}\log\frac{\Gamma(\lambda_p(s))}{\Gamma(\lambda_q(s))} \\ &+ \sum_{s=1}^{m}[\lambda_q(s)-\lambda_p(s)][\Psi(\lambda_q(s))-\Psi(\lambda_{qt})] \end{aligned}$$

この時

$$\lambda_{qt} = \sum_{s=1}^{m}\lambda_q(s),\quad \lambda_{pt} = \sum_{s=1}^{m}\lambda_p(s)$$
$$\Psi(x) = \frac{d}{dx}\log\Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

事後分布 $Pr(\theta|X)$ はすべてのノードに一つ割り当てられ，ノード間同士の距離は事後分布間の距離として求められる．確率分布同士の距離として KL 情報量を選択するのは自然な選択である．一定区間 V の間に，U に存在するどのデータチャンクが何回呼ばれたかを X_j 上に記録してゆく．観測が終わると，プローブ p_j は X_j をコンダクター $\mathcal{C}$ に送信する．$\mathcal{C}$ は存在するすべてのノードのペアを作り，ノードペア間の KL 情報量（距離を）を算出する．ディリクレ分布同士の KL 情報量の定義は定義 2.2 に示した．

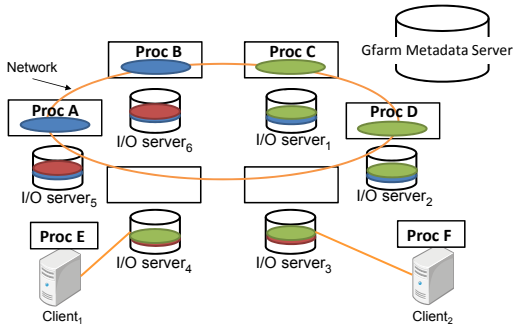


図 2 Gfarm アーキテクチャ
Fig. 2 Gfarm Architecture

2.2 想定環境

本節では 2.1 節にて述べたアルゴリズムの実環境への適応について論じる．われわれは現在，Gfarm と呼ばれる分散ファイルシステムと，IaaS 型クラウド基盤に本手法を適応することができると分析している．それぞれ詳細について以下に述べる．

本手法を適用するためには Co-operative Cache が有効であることが大前提であるため，はじめに，これから挙げる二つの想定環境において Co-operative Cache をどのように適応すれば高速化が見込めるかについて議論する必要がある．そのため，2.2.1 節と 2.2.2 節で取り上げる二つの例の冒頭部では，これらの環境に対して Co-operative Cache はどのように有効なのかを議論した後，2.1 節にて論じた提案手法をどのようにして適用するかについて論じる．

2.2.1 Gfarm への適応

Gfarm は大容量，高性能な分散ファイルシステムである．高エネルギー物理学，天文学，生物学といった科学技術分野から，Web サーバ，メールサーバといった一般的な利用まで幅広く利用範囲を想定して設計されている．

Gfarm のアーキテクチャを**図 2** に示す．Gfarm はジョブが投入される**クライアント**，データを蓄積し，クライアントからの `read()`/`write()` 要求に答える **I/O server**，そして，データがどの I/O server に存在するのか，ファイルの属性情報などのメタデータを管理する **Gfarm Metadata Server** からなる．

Gfarm の最も特徴的な点は計算ノードと I/O server を統合して扱うこともできる点である．そのため，アプリケーションが実行できる計算環境と大規模なデータストレージを同一環境上で一度に提供することが可能である．もちろん，図 2 に示したように，計算ノードをクライアントとして I/O server から分離することも可能である．

Gfarm 上で Co-operative Cache を実装することで，全体のディスクリードの回数を減らしてアプリケーションの高速化を図ることは可能であると考える．Gfarm に限った話ではないが，分散ファイルシステムというものは多くの情報を蓄えて，それを多くの人々が同時に利用できるよう

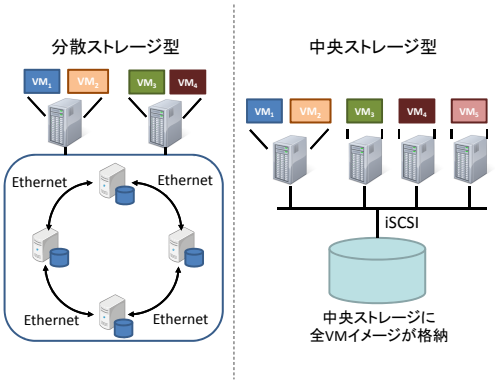


図 3 IaaS 型クラウド基盤のアーキテクチャ
Fig. 3 IaaS Cloud Architecture

にするものである．つまり，あるプロセス A がデータ X を欲するとき，それはすでに他のプロセス B によってすでにディスクから読み出されてメモリ上にキャッシュされているという状況が十分に考えられるのである．

これは，科学技術分野では顕著である．例えば，素粒子実験が行われ，その膨大な実験データが Gfarm 上に格納されていたとする．科学者たちは複数の異なる手法（図 2 中の Proc A, B, C, D）でこの実験データを同時に分析したいと考えるかもしれない．そのため，これら 4 つのジョブは複数の科学者によって I/O server へと投入される．これら 4 つのプロセスは全く同じデータを利用するため，Proc A が Proc B が必要とするデータ（図 2 中の青い部分）をすでにディスクから読み込んでメモリ上にキャッシュしているということは十分に有り得る．この時，Co-operative Cache があれば，遅いディスクドライブを読みに行く必要がなく，高速な InfiniBand 経由でデータをフェッチすることが可能である．

Gfarm に今回提案した手法を適応するためには以下のようにすればよい．ここで N_j は I/O server と Client がそれに相当する．$\mathcal{C}$ と U は Gfarm Metadata Server 上に置くことも可能だし，新たに別の Metadata Server を用意しておいてもよい．N_j 上の p_j が定期的に $\mathcal{C}$ に対して X_j のデータを送る．すると，$\mathcal{C}$ からはノード間距離（KL 情報量）の表が送られてくるので．次回の I/O リクエストがあった時はその表から最も値が小さいノードを幾つか選んでキャッシュの問い合わせを行えば良い．問い合わせのキーは SHA-1 であり，戻り値はキャッシュチャンクとなる．そのため，すべての N_j は SHA-1 とキャッシュチャンクの表をオンメモリで保持する必要がある．

2.2.2 IaaS 型クラウド基盤への適応

さらに，提案するアルゴリズムは Iaas 型のクラウド基盤における VM 間のページキャッシュヒット率の向上にも適用することができる．

VM のディスクイメージを格納するストレージと，VM を動作させる計算機ノードの関係に着目すると，現在の IaaS 型クラウド基盤は大きく分けて**図 3** に示すような二種類のアーキテクチャに分類される．図左の分散ストレージ型は，インターネットを経由して不特定多数のユーザに対して同時に VM を提供するシステムに見られる．Amazon EC2 や Windows Azure といった IaaS 型クラウド基盤はこれの代表例であろう．Amazon EC2 の場合 Amazon S3 と呼ばれる分散ストレージが存在している．また，Windows Azure の場合，Windows Azure Storage と呼ばれる分散ストレージが存在する．一方，図右の中央ストレージ型は，NAS や SAN 上に VM ストレージを保存しておき，複数人のユーザが iSCSI のようなプロトコルでリモートでマウントして VM を利用する形式である．当然この手法は，人数が増えれば中央ストレージに負荷がかかるため，VM の台数に対してスケールはしない．

この IaaS 型クラウド基盤においても，Co-operative Cache の概念は有用である．分散ストレージ型，中央ストレージ型いずれの場合にせよ，仮想マシンは物理的な計算機ノードに集約される．このとき，同じようなデータを読んでいる VM 同士は，同一の計算機ノードにまとめたほうが，他の VM が読んで VMM 上にキャッシュしたデータを有効活用できる機会が増える．不特定多数の人間が利用する IaaS 型クラウド基盤であれば，似通った動作，似通ったデータを利用している VM というものはある程度の数存在するのではないかと考えている．わかりやすい例が VM の *Boot storm* である．Boot storm とは，朝，従業員が出勤して一斉に各自の VM を立ち上げた時，OS のブートシーケンスが始まり，多数のランダムアクセスが中央ストレージに集中する現象である．これは特定組織内で構成員に対して提供される中央ストレージ型の IaaS（もしくは DaaS）型のクラウド基盤において特に問題となる現象 [3] である．この時，同じ OS 同士，同一の計算機ノードにまとめておけば，OS のブートシーケンスにおいて VM が読むデータはほとんど同一であるため，中央ストレージに読みに行くことなく，他の VM が読んだキャッシュデータを利用できる．結果的に，ネットワーク帯域や中央ストレージにかかる負荷を低減することができる．

なお，これを実現するためには，単一の計算ノード内において内容が同一のファイルが存在してた場合，それらを同一のものであるとみなしてキャッシュを融通する機構が必要である．現在の実用的なすべての OS はディスクキャッシュはファイルシステムの inode 番号を元に作用するものである．つまり，たとえ内容が同一のものであったとしてもファイルが違えば，いいかえると，inode 番号が違えばキャッシュは働かない．そのため，従来の inode 番号を元にしたキャッシュではなく，ファイルの内容に着目した Content-base のキャッシュ機構を考える必要がある．これは CBRC (Content-Based Read Cache) などと呼ばれ先行研究が存在 [4], [5], [6] しており，これを利用すること

が可能であると考えている．

この同一計算機ノード上に存在する VM 間でのキャッシュは，従来提案されてきたネットワーク共有ファイルシステムにおける Co-operative Cache の概念とは異なっている．しかし，システム全体で協調処理を行なってシステム上のディスクリード回数を減らし，その結果としてパフォーマンス向上を狙うという観点からして見れば，これも広義の Co-operative Cache であるとみなすことができる．

このクラウド基盤にも今回提案した手法は適応可能である．ここで N_j はクラウド上の VM に相当する．$\mathcal{C}$ と U はクラウド基盤を支える複数の VMM がインストールされたデータセンタに対して一つ存在する．一定区間内に各 VM:N_j がどんなデータを読んだのかを p_j が監視して X_j を作り，それを $\mathcal{C}$ に送る．$\mathcal{C}$ はそれを元に VM 間の距離（KL 情報量）の表を作る．VM ごとに表を参照してゆき，KL 情報量が近い者同士をグループ分けし同一物理ノード上に再配置する．これは VM を無停止で移動することができる VM ライブマイグレーション [7], [8] を用いることで実現する．この時 KL 情報量は対称性を満たさないことに注意しなければならない．つまり，VM_1 からみて VM_2 が近くても，VM_2 からみると VM_1 はとても遠いと判定される可能性もある．この際，どのようにグループ分けするべきかは結論が出ておらず今後検討する必要がある．

3. 実験

似たようなリードのパターンを持つノード同士の KL 距離が相対的に近づくのかどうかについて確認するための実験を行った．

実験のための簡単な環境を構築した．**図 4** に作成した実験装置の概要図を示す．VM（Virtual Machine）を用意し，その上で自作のプログラムを走らせた．プログラムは 100MB のデータを用意した後 5 つのリードパタンで読むプログラムである．100MB はページ単位（4KB）ごとに分割され，ページ毎に `rand() & 0xff` で生成した乱数値で埋められている．VM からディスクのリード要求があった場合，読んだ部分を VM のディスクページから 4KB ページ単位で取得し SHA-1 を計算した上で MySQL データベースへ区間 V の間に何回当該ページを読んだかの記録をとる改造を施した KVM[9] 上でこの VM を動作させた．ディリクレ分布同士の KL 情報量の計算には LingPipe API[10] を利用している．なお，V は 5 分とした．

図 5 にベンチマークプログラムのリードパタンを示す．一つ目のパターンは 100MB のファイルをすべてシーケンシャルに 4KB のづつ読み込んで行くものである．二つ目のパターンは特定のページのみを 100 回読み込む．三つ目のパターンは特定のページのみを 200 回読み込む．四つめのパターンは偶数ページのみを読み，最後の五つ目のパターンは奇数ページのみを読み込むものである．

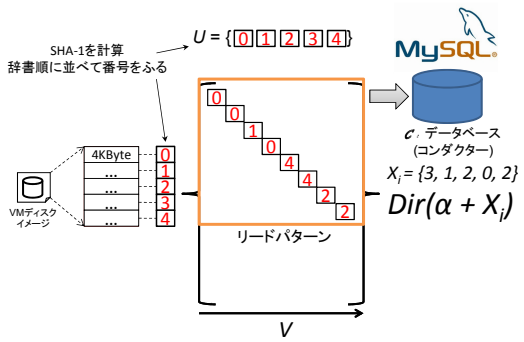


図 4 実験装置の概要図
Fig. 4 Overview of the experiment device

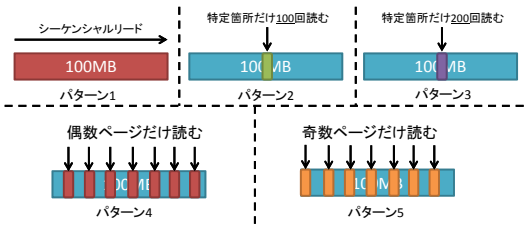
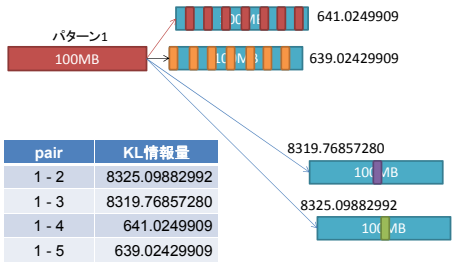


図 5 ベンチマークプログラムの 5 つのリードパタン
Fig. 5 5 patterns by the benchmark program



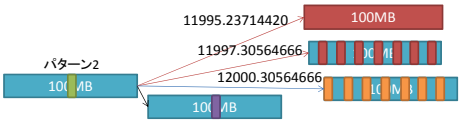
pair	KL情報量
1 - 2	8325.09882992
1 - 3	8319.76857280
1 - 4	641.0249909
1 - 5	639.02429909

図 6 実験結果 1
Fig. 6 Experiment result 1

これらのパターンを一回ずつ VM 上で走らせた．VM のディスクイメージは 7GBytes であり，RAM に 1GB 割り当てた．ゲスト OS は CentOS 6.4 の 32 ビット版である．

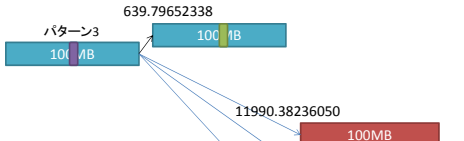
3.1 実験結果：パターン 1 からみた距離

パターン 1 を基点として残り 4 つのパターンまでの距離を**図 6** に示す．パターン 1 はすべてのページをリードするパターンである．この時，ある程度の長さを持つキューを仮定し，読んだ時系列順に古いキャッシュから捨てていくような単純なキャッシュアルゴリズムを想定する．すると，パターン 1 のような読み方をするアプリケーションがキャッシュの保持先として問い合わせに行くべきはパターン 4 と 5 である．パターン 2,3 はパターン 4,5 と比べるとキャッシュのヒット率が低いであろうことは容易にわかる．KL 情報量を見ると，パターン 2,3 が最も近く，この直感に反しない距離が算出されていることがわかる．



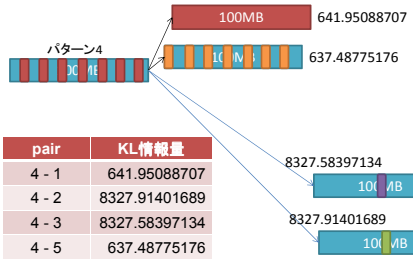
pair	KL情報量
2 - 1	11995.23714420
2 - 3	639.17879806
2 - 4	11997.30564666
2 - 5	12000.30564666

図 7 実験結果 2
Fig. 7 Experiment result 2



pair	KL情報量
3 - 1	11990.38236050
3 - 2	639.79652338
3 - 4	11997.45113609
3 - 5	11994.45113608

図 8 実験結果 3
Fig. 8 Experiment result 3



pair	KL情報量
4 - 1	641.95088707
4 - 2	8327.91401689
4 - 3	8327.58397134
4 - 5	637.48775176

図 9 実験結果 4
Fig. 9 Experiment result 4

3.2 実験結果：パターン 2 からみた距離

つぎに，パターン 2 を基点として他の 4 つのパターンまでの距離を図 6 に示す．パターン 2 は特定箇所を 100 回リードするパターンである．3.1 節で論じたようなキャッシュ戦略が働いているとして，パターン 2 が最も先に問い合わせに行くべきはパターン 3 であることがわかる．キャッシュを保持するキューの長さに完全に依存するが，パターン 1 に対して問い合わせに行なってもよいかもしれない，しかし，キューの長さが短ければ追い出されてしまっている可能性もある．パターン 4, 5 に問い合わせにいくのは危険な賭けである．なぜなら，この二つのいずれかどちらかはパターン 2 が欲するデータを全く読んでいないことになるからである．結果を見ると，最も近いパターンはパターン 3 となり，つぎに，パターン 1, 4, 5 と続いている．そのため，今回の結果も直感には反しないものであることがわかる．

3.3 実験結果：パターン 3 からみた距離

そして，パターン 3 を基点として他の 4 つのパターンまでの距離を**図 7** に示す．パターン 3 もパターン 2 と同じ箇所を読んでいるが，読んだ回数が違う．同様にキャッシュ戦略にキューを仮定すると，真っ先に問い合わせに行くべきはパターン 2 である．これは 3.1 節で議論したのと同様，キューの長さに依存するが，パターン 1 に対して問い合わせを行なってもよいかもしれない．パターン 4,5 に問い合わせに行くのは危険な賭けになる．この二つのどちらかはパターン 3 が欲するデータを全く読んでいないのは明らかである．算出された距離を見ると，この直感に反しない結果がでていることがわかる．

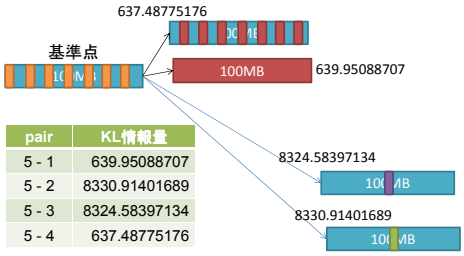
3.4 実験結果：パターン 4 からみた距離

パターン 4 を基点として他の 4 つのパターンまでの距離を図 6 に示す．パターン 4 は偶数ページのみを読み取るものである．パターン 4 がまっさきに問い合わせに行くべきは，パターン 1 である．同様にキューのようなキャッシュ戦略を考えると，この中で最もキャッシュヒット率が期待できるのはパターン 1 以外ありえない．パターン 5 は奇数ページのみを読んでいるため，パターン 4 が欲するデータを全く持っていないことになる．パターン 2,3 は持っているかもしれないが，キャッシュヒット率が著しく悪くなるであろうことは想像に難くない．

今回の結果をみるとかなり直感に反した結果が出ていることがわかる．パターン 5 が最も近い存在であると判定されているが，このパターンはパターン 4 が欲するデータを全く持っていないはずである．パターン 5 は最も遠い存在であるべきだが最も近い存在であると判定されている．

これにはいくつかの原因が考えられる．はじめに考えたのは Linux カーネルの Read Ahead が影響である．つまり，アプリケーションは実際は偶数ページしか読んでいないのだが，Linux カーネルが先読みを行い，奇数ページも読み込み，実質的にシーケンシャルリードになっているのではないかという可能性である．しかしそれであるならば，最も近しいのはパターン 1 になるべきだし，そのようになっていない以上この可能性は低い．

次に考えられる可能性は，今回実験に使用した 100MB のファイルに含まれるページの中にまったく同一の内容を持つページが存在しており，それがこの結果を引き起こしているという可能性である．3 章の冒頭で述べたように，ページごとに `rand() & 0xff` を利用してページの中を埋めている．`rand()` は散らばりがわるいことで知られているので，同一の内容を持つページが複数存在しており，図示したような横断歩道的なリードパタンがうまく X_i に反映できていないのかもしれない．そのため，これはキューの長さに完全に依存するが，実際はパターン 5 が最もキャッシュを保持しているとみなしている可能性も十分に考えられる．今後これを検証する必要があると考えている．



pair	KL情報量
5 - 1	639.95088707
5 - 2	8330.91401689
5 - 3	8324.58397134
5 - 4	637.48775176

図 10 実験結果 2
Fig. 10 Experiment result 5

3.5 実験結果：パターン 5 からみた距離

最後に，パターン 5 を基点として他の 4 つのパターンまでの距離を図 6 に示す．パターン 5 は奇数ページのみを読むパターンである．読みに行くべきはパターン 1 であり，パターン 4 はパターン 5 が持つデータを全く読んでいないことになるので問い合わせに行くのは無駄である．残りのパターンはキャッシュを保持している可能性はあるものの，キャッシュヒット率は極端に悪い容易に想像がつく．

結果を見ると，やはり 3.4 節で述べたのと同様，直感に反する結果が出ていることがわかる．この原因の仮説については先ほど述べたとおりである．

今回の実験結果をまとめると，一部さらなる検証が必要な結果が出たものの，概ね直感に反しない，満足出来る結果が出ていると言える．

4. 議論

今回提案した統計モデルは愚直な手法であり，SHA-1 で番号を割り振られたキャッシュブロックの数 k が予め定められた固定長となっている．当然，ディスクは刻一刻とか書き換わってゆくわけであり，k が固定であると定めるのは非現実的であるし，とても高い次元になってしまう．

これに対して有効であると思われるアプローチは LDA : Latent Dirichlet Allocation[11] である．楽譜，画像，文章分類等に幅広く使用されているモデル化の手法である．このモデルは文章が複数のトピック（話題）からなると考え，同一のトピックであれば，同じ単語が出現しやすいと仮定したモデルである．ここで，単語は SHA-1 で番号付けられたキャッシュブロックに，文章をノード N_j と読み替え，トピックをページディレクトリエントリと読みかえればこのモデルを利用することが可能なのではないかと考えている．

なお，今回の実装と評価では評価に使う VM とベンチマークプログラムを予め走らせておいて，k が十分大きくなった時点で改めて評価用にもう一度 VM とベンチマークを走らせるという手法をとっている．

5. 関連研究

今までの提案されてきた Co-operative Caching における block locating の手法 [1] と比較する．

最も単純な手法は *Direct Client Cooperation* で，これはアイドル状態のノード B をスレーブとして扱い，そこに対して一方的にプライマリのノード A から溢れたキャッシュを押し付け，その後必要になった場合はスレーブノード B に対して問い合わせを行うものである．この手法と比較すると，今回提案した手法は，完全に独立して動作するノード同士を統計的に分析した上で，似たもの同士をグループ分けするものである．そのため，過去にノード A がディスクから読んだことがないデータすら，B に対して期待を持って問い合わせに行くべきか否かの判断基準を与えることが可能であり，ノード間におけるキャッシュデータそのものの転送も発生しないという点が異なる．

Greedy Forwarding は中央サーバが，ノードが持つすべてのキャッシュデータのリストを管理する手法である．キャッシュが欲しい時，中央サーバに問い合わせる，サーバがそのキャッシュを持っていなければ，サーバ内に存在する，欲しいキャッシュとそれを保持するクライアントのリストを参照して読みに行く．一方で，提案手法では，キャッシュを持っているかもしれないという期待としてノード間距離の概念を利用している．そのため，提案手法のほうがノード数とキャッシュのバッファ数に対する空間効率が高いという点が異なる．大規模な分散ファイルシステムでは，提案手法のこの特徴は魅力的である．

Centrally Coordinated Caching は，ノード自身が持つローカルキャッシュ空間とノード同士が協調して作る分散キャッシュ空間の二つのキャッシュ空間を持つ手法である．まず，自分がもつローカルキャッシュを読んだ上で，ヒットしなければサーバに問い合わせに行きサーバにキャッシュされているかどうかを確かめる．それでもヒットしなければ，クライアント同士で構成される分散キャッシュ空間を探す，UDP のブロードキャスト，もしくは逐次問い合わせで，分散キャッシュを構成するすべてのノードに対して問い合わせを行うものと推測される．すべてのノードにキャッシュがヒットしなければ，サーバは諦めてディスクを読むという手法である．われわれの手法は Centrally Coordinated Caching に分類されるものであると考えられる．分散キャッシュ空間が提案手法におけるノード間距離の表であり，Centrally Coordinated Caching の分散キャッシュ空間のけるベイズ的統計学的手法を用いたキャッシュ look-up 機構の提案であると言える．

6. おわりに

本稿ではベイズ統計を用いた Co-operative Cache における新たな block locating の手法を提案した．類似度の高いノードを探し出し，類似度が高い（距離が小さい）ノードに対してキャッシュブロックの問い合わせを行う手法である．提案した手法を一般化し，本稿にて提案するアルゴ

リズムは分散ファイルシステム Gfarm と IaaS 型（DaaS 型）クラウド基盤に対して有効に活用できるのではないかと分析した上で，どのようにすれば適応可能かについての議論をおこなった．

VM（仮想マシン）を利用して簡単な実験環境を構築し本手法の評価を行った．5 つの異なるリードパタンでデータを読む簡単なベンチマークプログラムを用意しそれらを 5 つの VM 上で走らせ，その上で，VMM（仮想マシンモニタ）側から VM のページリードを観測して KL 情報量を算出した．新しいキャッシュのチャンクが入ってくるたびに，後ろのキャッシュを追い出すキューで管理されたいると仮定した時，キャッシュがヒットしそうな，似たようなデータを似たようなパタンで読んでいる VM の KL 距離が，すべてのパターンではないが，近くなったことを確かめることが出来た．しかし，一部納得しがたい結果が出たため，原因の仮説をいくつか提示した．今後はこの仮説を実験を通して具体的に検証してゆく予定である．

今後の課題としては以下の様なものがあげられる．今回はベイズ統計的なアプローチをとったが，類似度を求めるためのアルゴリズムは他に多数存在している．よく知られたものとしてコサイン尺度（コサイン類似度），ピアソン相関があげられる．これらの類似度指標との比較も必要であると考えている．

謝辞

本研究を行うにあたり，有益な助言を頂いた筑波大学建部修見准教授と建部研究室の方々に深く感謝する．また，本研究は，科学振興機構戦略的創造研究事業 (JST CREST) の研究課題「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」の支援を受けている．

参考文献

[1] Dahlin, M. D., Wang, R. Y., Anderson, T. E. and Patterson, D. A.: Cooperative caching: using remote client memory to improve file system performance, *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, USENIX Association, (online), available from ⟨http://dl.acm.org/citation.cfm?id=1267638.1267657⟩ (1994).

[2] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (online), DOI: 10.1007/s00354-009-0089-5 (2010).

[3] Hansen, J. G. and Jul, E.: Lithium: virtual machine storage for the cloud, *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, New York, NY, USA, ACM, pp. 15–26 (online), DOI: http://doi.acm.org/10.1145/1807128.1807134 (2010).

[4] Morrey, C. B. and Grunwald, D.: Content-Based Block Caching, *23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST2006)* (2006).

[5] Govindankutty, S.: View Storage Accelerator in VMware Viewreport (2012).

[6] Koller, R. and Rangaswami, R.: I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance, *FAST*, pp. 211–224 (2010).

[7] Nelson, M., Lim, B.-H. and Hutchins, G.: Fast transparent migration for virtual machines, *Proceedings of the USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, USENIX Association, pp. 25–25 (online), available from ⟨http://portal.acm.org/citation.cfm?id=1247360.1247385⟩ (2005).

[8] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live migration of virtual machines, *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, Berkeley, CA, USA, USENIX Association, pp. 273–286 (online), available from ⟨http://portal.acm.org/citation.cfm?id=1251203.1251223⟩ (2005).

[9] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, *Proceedings of the Linux Symposium*, pp. 225–230 (2007).

[10] Alias-i, Inc.: LingPipe, `http://alias-i.com/lingpipe/`.

[11] Blei, D. M., Ng, A. Y. and Jordan, M. I.: Latent dirichlet allocation, *J. Mach. Learn. Res.*, Vol. 3, pp. 993–1022 (online), available from ⟨http://dl.acm.org/citation.cfm?id=944919.944937⟩ (2003).