Vol.2014-OS-130 No.6
2014/7/28

分散ファイルシステムGfarmの協調キャッシュ機構の検討

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：大量のデータを読み書きするデータインテンシブなアプリケーションを高速化するため，単一ノード内で完結しない，ネットワークを介した他のマシンとの協調キャッシュ戦略を導入する必要がある．本論文では分散ファイルシステム Gfarm 上での Cooperative Caching (協調キャッシュ) 機構について議論した後，現実的なデータインテンシブワークフローである Montage ワークフローを用いて，どの程度ページヒット率が見込めるかについての調査を行った．調査の結果，かなりの数のページを再利用することができ，Gfarm に Cooperative Caching を導入した時，リードの性能向上の可能性が示唆された．

A Study of Cooperative Caching on Distributed File System Gfarm

Abstract: In the field of data-intensive science that reads and writes a huge amount of data, we have to consider the cache strategy over network instead of single node cache strategy for speed up In this paper, first we proposed a cooperative cache mechanism for the distributed file system Gfarm. Next, we investigate how many pages as cache pages exist in one of real world applications (e.g. Montage) in our proposed method. Our results show that a large of amount pages become re-usable on our mechanism. The results show that our method is probably effective way.

1. はじめに

10-gigabit Ethernet や InfiniBand といった高速なインタフェイスで構築されたネットワーク上でのデータ転送速度は，最新鋭のディスクドライブと比べると，極めて高速である．ハイパフォーマンス・コンピューティングで利用される分散ファイルシステムを構成するノードはこれらのインタフェイスで相互接続されることが多い．

そのため，これらの分散ファイルシステム上で，大量のデータを読み書きするデータインテンシブなアプリケーションの高速化のためには，単一ノード内で完結しないキャッシュ戦略，単一ノード内で完結しない，ネットワークを介した他のマシンとの協調キャッシュ戦略を導入する必要がある．

例えばノード A とノード B で分散ファイルシステムが構築され，ノード A 上のローカルドライブ上にデータ X が存在するとする．そのデータ X が，ノード B によって読み出され，自身のメモリ上にロードされキャッシュされているとする．ノード A がそのデータを読むときは，ノード B のキャッシュをネットワーク経由で転送したほうが，自身が持つ 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 *Cooperative Caching* [1] (協調キャッシュ) と呼ばれ，古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を減らす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，Dahlin らの研究が行われた時と比べて，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，Cooperative Caching の重要性は今後ますます高まると考えられる．

われわれは，分散ファイルシステムである Gfarm [2] 上に Cooperative Caching のシミュレータを実装した．Gfarm にはこれまで協調キャッシュの仕組みが実装されていないため，実装案を提示することは大きな新規性である．また，Gfarm に対して協調キャッシュを実装する手法の議論は，最新鋭の分散ファイルシステム上にどのように

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

© 2014 Information Processing Society of Japan

Cooperative Caching を実現するのかという問いに対する一つの答えとなり，これも重要な知見になる．

Gfarm 上で Cooperative Caching を実現するためには，いくつかの手法が存在するが，今回はキャッシュが移動するたびに，キャッシュ管理マネージャに対してキャッシュの移動を報告する中央集権的な手法ではなく，Gfarm を構成する各々のノードが独立して保持するキャッシュ情報を元にしてキャッシュ保持の問い合わせを行う手法を仮定してシミュレーションを行った．これは Gfarm 上で実行されるデータインテンシブなアプリケーションを考えると妥当な設計であるといえる．データインテンシブなアプリケーションでは大量のデータが読み書きされるため，その都度逐一中央集権的なキャッシュマネージャにキャッシュの情報を送ることはコストが高いからである．

この設計を仮定した上でシミュレーションを行い，現実的なデータインテンシブアプリケーションである Montage ワークフロー [3] をその上で走らせ，Cooperative Caching でどの程度のキャッシュヒット率が見込めるかの評価を行った．評価の結果，Montage ではかなりの量のページヒット率があることがわかった．

2. Gfarm のアーキテクチャ

Gfarm 分散ファイルシステム上の Cooperative Caching を論じるためには，Gfarm のアーキテクチャに関する知識と Cooperative Caching に関する知識が必要になる．そのため，まずはじめに本章にて Gfarm のアーキテクチャについて述べたあとで，3 章にて既存の Cooperative Caching について解説する．

Gfarm 分散ファイルシステムは，Lustre ファイルシステム [4] などとは異なり，**計算を行うアプリケーションと分散ファイルシステムのデータを保存するサーバを同一のノードで動作させることにより，アプリケーションがデータに効率的にアクセスできるという利点を持つ分散ファイルシステムである**．そのため，Gfarm では，クライアントとディスクを束ねるノードを同一にすれば，それらの間を接続する高速なインタフェイスを用意する必要がないと言った利点が存在する．

図 1 に Gfarm のアーキテクチャを示す．Gfarm は，ファイルの open 状態や，どの I/O server にファイルが実際にいるかと言ったメタデータを管理するサーバ (gfmd: 図 1 の Gfarm Metadata Server) とファイルを実際に格納する I/O server (gfsd) と，gfarm ファイルシステムを利用するクライアントの三つから構成される．gfarm は専用の Gfarm API で `open`, `close`, `read`, `write`, `seek`, `stat`, `rename`, `unlink` と言ったファイル操作を行うことができる．また，`gfarm2fs` と呼ばれるツールを用いることで，FUSE [5] 経由でマウントを行い，通常のアプリケーションで Gfarm を利用することも可能である．

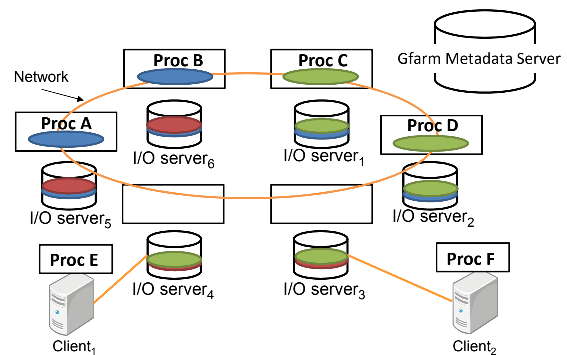

図 1 Gfarm のアーキテクチャ
Fig. 1 Gfarm architecture

クライアントがファイルを open する時は，まずはじめに gfmd にたいして `open` リクエストが送られる．この時，open 対象のファイルがどの gfsd に存在しているかが gfmd から送られてくる．次に，gfmd が返してきた gfsd に対して改めて open を行う．gfmd にアクセスするのは `open` と `close` 時のみである．`read`, `write` 時には gfsd へのアクセスのみが発生し，gfmd へのアクセスは発生しない．そのため，ファイルを実際に読んでいる時には gfmd への負荷はかからない．

3. Cooperative Caching

本節ではまずはじめに，Cooperative Caching の概要を説明した後，これまで研究されてきた基本的なアルゴリズムである N-Chance と Hint-based Cooperative Caching について解説する．

3.1 Cooperative Caching の概要

Cooperative Caching とはクライアントとサーバの間にまたがるキャッシュ機構である．クライアントのページキャッシュやストレージキャッシュの容量が不足したり，キャッシュが存在しなかった時に他のクライアントに対してキャッシュを転送したり，他のクライアントに存在するキャッシュ利用するものである．このように，Cooperative Caching は単一のクライアント内に閉じたクライアントキャッシュとは異なり，ネットワークを通してクライアント間をまたいだキャッシュであることが他のキャッシュ機構と異なる点である．

図 2 に示すように，Cooperative Caching と呼ばれる機構には，本質的に，三つの論理的な登場人物が存在する．これは 2 節で説明した Gfarm のアーキテクチャの解説とは異なり，一般論として，Cooperative Caching にはどのような登場人物が存在するかの説明である．

Cooperative Caching には，クライアントとサーバとキャッシュマネージャが存在する．クライアントは，自分のメモリ内にキャッシュが存在しない場合，他のクライアントにキャッシュを保持しているかどうかを聞きに行く．

© 2014 Information Processing Society of Japan

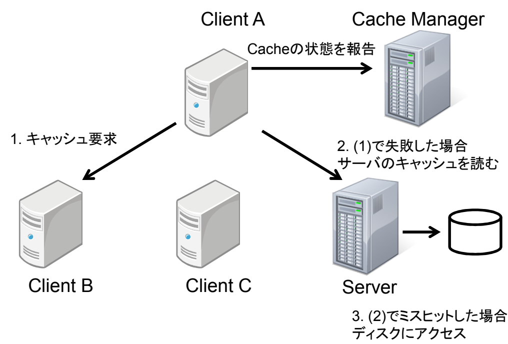

図 2　Cooperative Caching のイメージ図
Fig. 2　The image of cooperative cache

もし，問い合わせに行ったどのクライアントにもキャッシュが存在しない場合，サーバに問い合わせを行う．サーバは自身のキャッシュにそれが存在するか確認し，あればそれを転送する，なければ，ディスクからデータを読む．キャッシュマネージャはキャッシュの位置を管理し，どのキャッシュがどのクライアントに存在しており，破棄されたかを管理する．これを実現する方法は例えば，クライアントがキャッシュを移動したり，破棄した時にキャッシュマネージャに逐一それを報告するという手法がある．これが基本的な Cooperative Caching の流れである．

3.2 N-Chance

Dahlin らによって行われた研究 [1] では Cooperative Caching を実現するためのいくつかの手法について論じられており，その中でも *N-Chance* と呼ばれる手法が最も効果的であると結論づけている．

N-Chance においてキャッシュの置き換えポリシーは，すべて LRU で管理される．つまり，クライアントは参照が少ないキャッシュを順に破棄してゆく．しかし，特定の *Singlet* と呼ばれるキャッシュは特別扱いされる．Singlet とは，あるクライアントしか保持していない（全クライアント中の最後のコピー）キャッシュである．キャッシュマネージャはシステム全体のグローバルなキャッシュ情報を管理しており，すべてのクライアントが持つすべてのキャッシュの情報を保持しているため，Singlet を特定するのは容易である．

クライアントはキャッシュを破棄するとき，それが Singlet であった場合，そのキャッシュのリサイクルカウンタを N にセットする．そして，ランダムなクライアントを選んでそのキャッシュを送り，キャッシュが移動したことをキャッシュマネージャに報告する．キャッシュを受け取ったクライアントはそれがあたかも最も最近参照されたものであるかのように扱い，クライアントの LRU リストの最も上に持ってくる．またそのキャッシュブロックがクライアントの LRU リストの最後に達した時は，カウンタ N の値を 1 減らして，ランダムに選んだまた別のクライアントに転送する．これは N が 0 になるまで繰り返されることになる．つまり，N-Chance は特別扱いされるキャッシュが存在する LRU ということができる．この Singlet 付きの LRU はクライアント間で分散共有メモリ空間を実現する GMS [6], [7] でも同じものが使用されているが，GMS はメモリページに対して常に Singlet か否かのフラグが付く点が Cooperative Caching の N-Chance とは異なる点である．

また，キャッシュの場所を特定する *Block Lookup* を行うときは，キャッシュマネージャに問い合わせを行なえば，キャッシュを保持しているクライアントを特定することができる．N-Chance の方式においては，クライアントが，キャッシュの移動，破棄を行った場合，それはすべて中央集権的なキャッシュマネージャに報告されるからである．

なお，この N-Chance は xFS [8] や zFS [9], [10] といったファイルシステムに導入されている．

3.3 Hint-based Cooperative Caching

Hint-based Cooperative Caching [11] とは Cache Manager がシステム全体でグローバルなキャッシュ管理を行わないタイプの Cooperative Caching である．N-Chance の場合では，図 2 の図にある，キャッシュマネージャが，どのクライアントがキャッシュを破棄したのか，どのクライアントがキャッシュを他のクライアントに転送したのかというキャッシュの動きをすべてトレースして管理していた．つまり，キャッシュマネージャはシステムのグローバルなキャッシュ情報を把握していることになる．しかし，この手法は当然すべてのキャッシュの移動を逐一トレースする必要が有るためキャッシュマネージャの負荷が増大するという欠点があった．

Hint-based Cooperative Caching はこのキャッシュマネージャの負荷を低減することを目標に提案されたものである．クライアントはマネージャによるシステム全体のグローバルなキャッシュ情報ではなく，クライアント各自のローカルな情報を元にキャッシュの場所を特定する．そのため，従来の N-Chance とは異なり，キャッシュの状態をすべて中央集権的にトレースする必要がなく，マネージャの負荷を軽減することができる．

Hint-based Cooperative Caching を理解するためには，マスターコピーと非マスターコピーの概念について理解する必要がある．**図 3** が示すように，Hint-based Cooperative Caching では，データを二次記憶内に格納しているサーバから直接ダウンロードしたものがマスターコピー，そして，マスターコピーを得たクライアントからコピーされたものは非マスターコピーといった具合に区別される．

サーバに対してファイルをオープンしたクライアントは，キャッシュマネージャから，そのファイルを構成するブロックのマスターコピーを誰が保持しているかに関す

© 2014 Information Processing Society of Japan

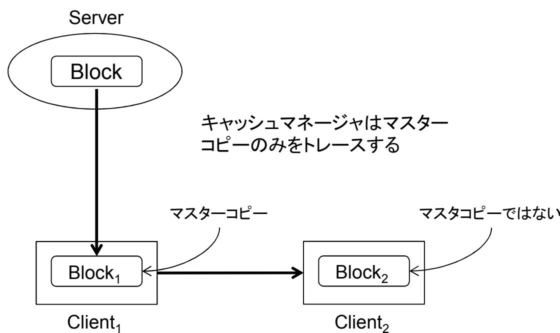

図 3 Hint-based Cooperative Caching におけるマスターコピーと非マスターコピー

Fig. 3 Master copy and non-master copy on hint-based cooperative caching

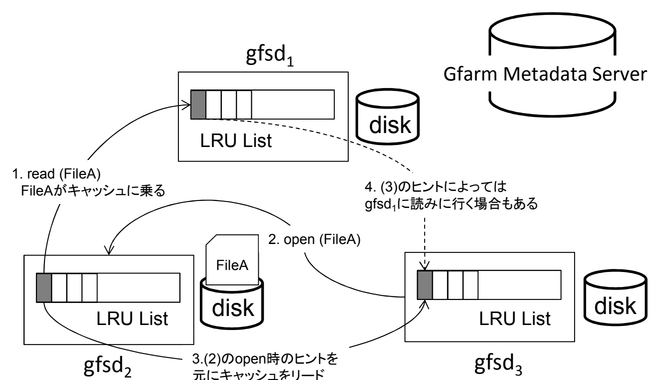

図 4 Gfarm での Hint-based Cooperative Caching

Fig. 4 Hint-based Cooperative Caching on Gfarm

るヒントの集合をキャッシュマネージャから得る．キャッシュマネージャはこのヒントを最後にファイルをオープンしたクライアントから得る．最後にファイルをオープンしたクライアントは最も正しいヒントを保持している可能性が高いからである．キャッシュの置き換え（フォワーディング）については後に詳しく述べるが，もし，あるクライアントがキャッシュを別のクライアントにフォワーディングする場合は，フォワーディングした後に両方のヒントを更新する．

この手法は，キャッシュマネージャへの負荷を低減させることが可能であるが，その一方で，キャッシュマネージャから得る「どこにキャッシュが存在するのか？」という情報が正確ではない可能性が出てくる．システム全体でグローバルなキャッシュ情報を管理する N-Chance では図 2 が示すキャッシュマネージャに問い合わせれば，必ず，正確な，事実 (fact) に基づくキャッシュの位置が返ってくる，その一方で，Hint-based Cooperative Caching は fact ではなくひょっとしたら間違っているかもしれないヒントをキャッシュマネージャから得る．したがって Block lookup の際には，間違ったヒントを受け取る可能性も存在する．つまり，クライアントにキャッシュを要求したものの，そのキャッシュをクライアントが持っていない可能性も存在するのである．この時は，キャッシュミスとして改めてサーバ側に問い合わせればよい．

Hint-based Cooperative Caching でのキャッシュの置き換えは *Best-guess Replacement* と呼ばれるポリシーにしたがって行われる．Cooperative Caching 上のクライアントはそれぞれ *oldest block list* と呼ばれるリスト構造を一つ持つ．この oldest block list とは各クライアントが保持するブロックの中で最も古いブロックをリストアップしたものである．各ブロックには世代番号があり，oldest block list はその世代番号を元にソートされている．サーバから得たマスターコピーはこの oldest block list から最も古いものを保持しているクライアント，つまり，oldest block list の最も先頭にあるブロックを保持するクライアントに転送される．この手法は多くの場合において Global LRU に非常に似た性能を出すことが可能であると Hint-based Cooperative Caching の開発者らは主張している．

4. Gfarm における Cooperative Caching 機構の設計と実装

3 章にて，Cooperative Caching を実現するために何が必要なものかを論じた．本章では，3 章での議論を元に，分散ファイルシステム Gfarm 上に Cooperative Caching を実現する手法について検討した後，Gfarm 上に実装したシミュレータについて解説する．

Gfarm における Cooperative Caching を実現するためにはキャッシュがどこにあるかを検索する Block lookup と，各ノードが持つキャッシュをどのようなポリシーで破棄，入れ替えを行うのかの replacement policy をそれぞれ設計する必要がある．本章ではこれらを順に解説してゆく．

4.1 Block lookup

3.2 節で述べたように，Dahlin らによって提案された Cooperative Caching では，キャッシュの場所はすべて中央集権的なキャッシュマネージャによって管理される．キャッシュを探すときは，このキャッシュマネージャに問い合わせればキャッシュブロックの位置を正確に特定することができる．

一方で，われわれは，中央集権的なキャッシュマネージャの存在は仮定せず 3.3 節で述べた Hint-based Cooperative Caching で提案されている手法に近い手法を取る．すなわち，グローバルなキャッシュステータスを管理するのではなく，ローカルのキャッシュステータスを元に，fact（事実）ではなく間違っているかもしれない hint ベースの Block lookup を行う．

われわれは，Gfarm の gfsd 上にキャッシュマネージャを配置する設計を行った．2 章で述べたとおり，Gfarm はメタデータサーバである gfmd，データを格納する gfsd，gfsd に接続してファイルを利用する Client の三つからなる．た

だし，Gfarm は Lustre ファイルシステムなどとは異なり，計算を行うノードと，データを保存するノードを同一にできるという利点を持つ分散ファイルシステムである．この場合，Client と gfsd は同一ノードで動作することになる．gfmd ではなく，gfsd 上に配置した理由は以下のとおりである．まず，2 章で述べたように gfsd は，その gfsd に保存されているファイルに対するすべての `read`, `write` のログを取ることができる．したがって，gfsd にアクセスするクライアントがどのようにデータを呼んだのかの詳細を得ることができ，高い精度でクライアントがキャッシュを保持しているかどうかのヒントを与えることができるからである．

図 5 が示すように，Gfarm 上での Cooperative Caching でもマスタコピーと非マスターコピーの概念が存在する．gfsd が保持するファイルに対して直接リードしたものだけがマスターコピーであり，それ以外はすべて非マスターコピーである．3.3 節で述べたものとは異なり，gfsd（サーバ）には他の gfsd（クライアントと gfsd が同一ノードで動いていた場合）がファイルを読んだり，また，クライアントが gfsd に読みに行ったりする際にマスターコピーが作られる．Gfarm では gfsd 上にジョブがデプロイされ，そのジョブが他の gfsd に保存されているファイルにアクセスすることもあるからである．そのため，gfsd（サーバ）上にもマスターコピーが作られることがある．

図 4 に，Gfarm 上での Block lookup の流れを示した．まず，gfmd に対してあるファイル A の `open` 要求がいくと，ファイル A が格納されている gfsd（図中 $gfsd_2$）が返ってくるので，その gfsd に対してファイルの `open` の要求がいく．この時，この $gfsd_2$ はそのファイルを過去に read したことのある他の gfsd（図中の $gfsd_1$）を，自身の LRU リストで管理しているため，そのファイルのキャッシュを持っていそうな gfsd をリストからランダムに選び，これをヒントとして返す．ヒントを受け取ったらそのヒントを元に $gfsd_3$ に問い合わせを行う．この時，キャッシュのブロックは $gfsd_3$ の LRU List には入らない．なぜなら，図 5 で示したとおり，$gfsd_3$ が取得したキャッシュのコピーは非マスターコピーに相当するためである．この非マスターコピーは LRU List に乗らないという手法は，3.3 節で述べた Hint-based Cooperative Caching の手法を踏襲した．

4.2 キャッシュの置き換えポリシーと破棄

次にキャッシュの置き換えポリシーについて述べる．3.2 節で述べたとおり，N-Chance の場合は Singlet がある LRU にてキャッシュの置き換えを行っている．Hint-based Cooperative Caching の場合はサーバからマスターコピーを得るとき，Best-guess Replacement に基づいてキャッシュの置き換え（フォワーディング）を行う．

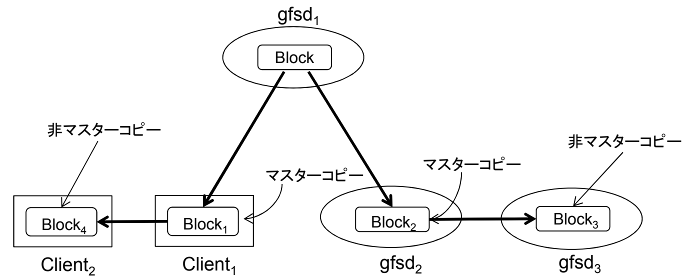

図 5 Gfarm での Cooperative Caching におけるマスターコピーと非マスターコピー

Fig. 5 Master copies and Non-master copies of Cooperative Caching on Gfarm

一方で，Gfarm 上での Cooperative Caching の場合，図 4 で示すようなアーキテクチャになる．各々の gfsd が独自の LRU リストを持ち gfsd が読んだファイルのブロックはここに格納される．われわれは N-chance や Hint-based Cooperative Caching の Best-guess Replacement と行ったような，能動的なキャッシュブロックの転送（フォワーディング）は一切行わない．このこの理由として，われわれの目指すデータインテンシブサイエンスにおいては，扱うファイルが巨大であり，ファイルのキャッシュの転送はノード間通信に負荷をかける恐れがあるからである．

また，既存の Cooperative Caching は何らかの手法でキャッシュの重複を排除して，なるべく多くのキャッシュが Cooperative Caching 上に乗るようにしている．しかし，われわれが今回提案する手法では，この重複排除も行わず，各々の gfsd が LRU にしたがってキャッシュを保持するだけにする．この理由としては重複排除を行ってしまうと，仮に，多くのクライアントが必要とするファイルにアクセスが集中した場合，特定の gfsd にアクセスが集中してしまうおそれがあるためである．したがって，Hint-based Cooperative Caching のように非マスターコピーが破棄されることはなく，各々の gfsd 上にある LRU Cache List に保持されることになる．

4.3 シミュレータの実装

4 章で述べた設計を元にシミュレータを実装した．**図 6** にシミュレータの内部構成を示す．

各 gfsd は内部に二つのテーブルを持つ．一つ目のテーブルは gfsd list である．これは，gfsd に今までファイルを要求してきた gfsd（もしくはクライアント）の一覧が格納されているテーブルである．図にあるように，`open` を行った段階で，表に記録される．次に，Reads table であるが，これはその gfsd に接続して来たすべてのクライアントがどのファイルのどの部分を読んだかがページ単位ですべて記録されている．図が示すように，`read` した時，そのリードしたファイルの gfarm 上での i ノード番号，世代番号，ページ番号が記録される LRU テーブルである．なお，世代番

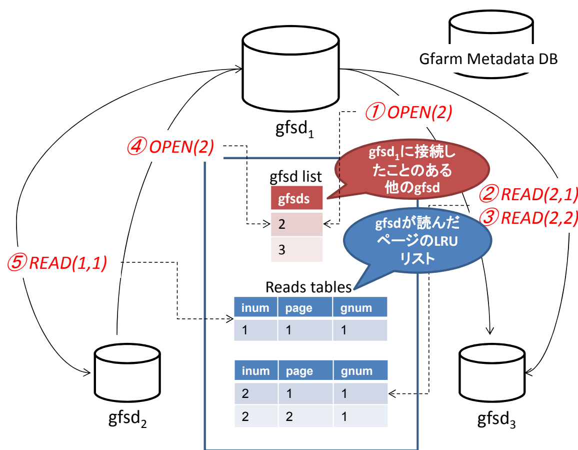

図 6 Gfarm 上での Cooperative Caching シミュレータのイメージ図
Fig. 6 Image of our simulator for Cooperative Caching on Gfarm

号は i ノード番号が再利用された時，write 後に close が走った時にインクリメントされる番号である．この二つのテーブルは Gfarm に sqlite3 を組み込むことで実現した．

ワークフローが走った時，read 要求が来るたびにこの reads table を参照し，テーブル内にエントリが存在していれがキャッシュヒット．そうでなければキャッシュミスにして，さらに，LRU ポリシーでエントリを追加する．なお，現在はランダムに他の gfsd に読みに行くという実装はなされておらず，他の gfsd によって読まれたものが LRU リストに乗ったものをキャッシュとして利用するという形になっている．

5. シミュレーション結果

われわれは Montage [3] ワークフローを利用して 4.3 節で論じたシステムの評価を行った．Montage ワークフローとは天文分野で使われるもので，バラバラに撮影された星雲の写真を，重なっている部分等を除いて一枚の巨大な写真に仕上げるワークフローである．評価の目的は，4.3 節で論じたシステムにおいて，どれくらいの Cooperative Caching が効くのかを確かめるものである．Montage ワークフローは Pwrake [12] を使用して 8 台（うち一台はジョブ投入用のフロントエンド）のクラスタマシン（jupiter 1 から Jupiter 8）上で実行を行った．Pwrake のバージョンは 0.9.9, rake のバージョンは 10.0.0 を利用した．またシミュレーション実装に使った gfarm はバージョン 2.5.7.2 であり，gfarm2fs は 1.2.9.5 を改造した．使用したクラスタマシンの詳しいスペックは表 1 に示す．

表 2 と表 3 にシミュレーション結果を示す．この表は jupiter1 から jupiter 8 上に Pwrake によってデプロイされたジョブがどれだけのページ（ページは 1MB のラージページ単位）を読み，そのうちどれだけのページがキャッシュヒットしたかを示したものである．

表 1 実験に使用したクラスタマシン（Jupiter）のスペック
Table 1 Experiment Environment (jupiter)

CPU	Intel(R) Xeon(R) CPU E5645 @ 2.40GHz
Memory	49 GBytes
HDD	SAS 600GB
OS	CentOS release 6.3
Network	InfiniBand Mellanox Technologies MT26428 QDR

表 2 を見ればわかるが，まず，最もキャッシュのヒット率が高いのが jupiter3 上にデプロイされたジョブで 8197 ページリード中 6269 ページがヒットしている．次点は，jupiter4 にデプロイされたジョブで 9631 ページリード中 7865 ページがヒットしている．リモートノードからリードされたページでヒット率が最も高いのは jupiter4 にアクセスしている jupiter3 でこれは 5942 ページリード中 5853 ページがキャッシュヒットしているのがわかる．

次に，表 3 を見ると，最もキャッシュのヒット率が高いのは jupiter5 上でデプロイされたジョブで 9374 ページリード中 8516 ページがヒットしている．次点は，jupiter7 上にデプロイされたもので 7999 ページリード中 7112 ページリード中ヒットしている．また，リモートノードからリードされたページでヒット率が高いのは jupiter5 が jupiter8 にたいして 4870 ページリードしてそのうち 4502 ページがヒットしている．

このように，Montage ワークフローに限って言えば，われわれの Cooperative Caching の手法において，ページヒットが十分に存在する（つまり，他のマシンが呼んだデータを再び誰かが読む，高速化できる可能性）があることが示唆された．

6. 関連研究

xFS [8] は N-Chance を改良したアルゴリズムを搭載したサーバーレスの分散ファイルシステムである．xFS と Gfarm の実装の違いは，xFS が “First Writer Policy” と呼ばれるメカニズムをとっているところにある．xFS の場合，最初にファイルに書き込んだクライアントがキャッシュマネージャとなり，単一のキャッシュマネージャに負荷がかからないようになっている．一方で，われわれが今回設計した Gfarm では，当該ファイルを保持する gfsd がキャッシュマネージャになる．そのため，xFS とは異なる．

zFS [9], [10] も，xFS と同様に，N-Chance Forwarding を実装した分散ファイルシステムである．zFS には open 時に関連付けられる FMGR と呼ばれるファイルマネージャによってキャッシュを管理する．クライアント A がファイル X をリードする場合はこの FMGR が他にキャッシュを保持していないかどうかをチェックし，他にキャッシュを保持していない場合はそのキャッシュを singlet に，そうでない場合は（例えばクライアント B がキャッシュを保持

© 2014 Information Processing Society of Japan

表 2　シミュレータ上で Montage を動かした時の再利用可能ページの数．Jupiter2 から Jupiter4

Table 2　Amount of cache hit pages of Montage on the simulator. From Jupiter2 to Jupiter 4.

jupiter2 にアクセスしたノード	hits	read
jupiter1	0	0
jupiter7	977	1312
jupiter5	1661	2034
jupiter8	940	1255
jupiter6	2243	2649
jupiter3	2815	3709
jupiter2 (local)	8240	9676
jupiter4	2749	4786
jupiter3 にアクセスしたノード	hits	read
jupiter1	7	12
jupiter5	1140	1519
jupiter7	1273	1676
jupiter8	3091	3494
jupiter6	1515	2032
jupiter3 (local)	6269	8197
jupiter2	1548	2135
jupiter4	1845	2483
jupiter4 にアクセスしたノード	hits	read
jupiter1	8	16
jupiter7	4041	4087
jupiter5	4687	4732
jupiter8	3939	3990
jupiter3	5853	5942
jupiter2	4487	4661
jupiter4 (local)	7865	9631
jupiter6	4073	4116

していた場合）クライアント A と B が replicated としてマークされる．なお，キャッシュのフォワーディングや回収は Linux の kswapd を改良して行われている．われわれが今回設計したものは N-chance ではないため，本研究とは異なる

GMS [6], [7] とはネットワークを介した分散メモリである．各ページは age base のキャッシュ管理ポリシーを使ってキャッシュを管理している．GMS はファイルシステムではないため，Gfarm 上に Cooperative Caching を実装した本研究とは異なる．

HyCache+ [13] は RAM と Disk の間にグローバルなキャッシュ空間をノード間に，ネットワークを介して構築する研究である．実装は FUSE [5] を用いて構築されているため，アプリケーション側には完全な POSIX API が提供される．本研究は，バックエンドにある GPFS [14] と協調動作を行わず，独自の Key-value ストアのような動作を行う．そのため，GPFS とは別に，また HyCache+独自のメタデータが分散ハッシュテーブル (ZHT [15]) で管理される．そのため，Gfarm そのものに組み込まれ，メタデータ自体が統一的に管理される本研究とは異なる．

表 3　シミュレータ上で Montage を動かした時の再利用可能ページの数．Jupiter 5 から Jupiter 8

Table 3　Amount of cache hit pages of Montage on the simulator. From Jupiter 5 to Jupiter 8

jupiter5 にアクセスしたノード	hits	read
jupiter1	0	0
jupiter7	3153	3430
jupiter8	1727	2085
jupiter3	664	1000
jupiter6	1796	2076
jupiter4	1264	1466
jupiter2	834	1131
jupiter5 (local)	8516	9374
jupiter6 にアクセスしたノード	hits	read
jupiter1	1	2
jupiter8	2521	2829
jupiter3	1634	2371
jupiter7	515	731
jupiter2	4023	4630
jupiter5	2878	3308
jupiter4	863	1251
jupiter6 (local)	4027	4364
jupiter7 にアクセスしたノード	hits	read
jupiter1	0	0
jupiter5	1339	1593
jupiter6	2284	2581
jupiter8	3229	3541
jupiter2	3212	3556
jupiter3	996	1385
jupiter7 (local)	7112	7999
jupiter4	2774	3000
jupiter8 にアクセスしたノード	hits	read
jupiter1	0	0
jupiter7	2999	3392
jupiter5	4502	4870
jupiter8 (local)	6288	6677
jupiter3	893	1313
jupiter2	677	902
jupiter4	1649	2012
jupiter6	2785	3134

7. おわりに

本論文では分散ファイルシステムである Gfarm 上に Cooperative Caching のシミュレータを実装した．Gfarm 上で Cooperative Caching を実現するためには，いくつかの手法が存在するが，今回ではキャッシュのトレースを行わず，中央集権的なキャッシュ管理マネージャが存在しない手法を仮定してシミュレーションを行った．これは Gfarm 上で実行されるジョブを考えて妥当な設計であるといえる．この設計を仮定した上でシミュレーションを行い，現実的なデータインテンシブアプリケーションであ

る Montage ワークフローをその上で走らせ，Cooperative Caching でどの程度のキャッシュヒット率が見込めるかの評価を行った．

評価の結果，最もキャッシュのヒット率が高いのが最もキャッシュの再利用率が高いのは jupiter5 上でデプロイされたジョブで，90%以上のページが再利用できていることがわかる．次点で，jupiter3 上にデプロイされたジョブで，再利用率は 76%であることがわかった．全体的に数 GB 単位でのページ再利用が観測されたため，本提案手法による Cooperative Caching における高速化の可能性は高いことが示された．

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] Montage Project: Montage. `http://montage.ipac.caltech.edu/`

[4] Schwan, P.: Lustre: Building a File System for 1,000-node Clusters, *Proceedings of the Linux Symposium*, p. 9 (2003).

[5] FUSE Project: FUSE. `http://fuse.sourceforge.net/`

[6] Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M. and Thekkath, C. A.: Implementing Global Memory Management in a Workstation Cluster, *SIGOPS Oper. Syst. Rev.*, Vol. 29, No. 5, pp. 201–212 (online), DOI: 10.1145/224057.224072 (1995).

[7] Feeley, M. J., Morgan, W. E., Pighin, E. P., Karlin, A. R., Levy, H. M. and Thekkath, C. A.: Implementing Global Memory Management in a Workstation Cluster, *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, New York, NY, USA, ACM, pp. 201–212 (online), DOI: 10.1145/224056.224072 (1995).

[8] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S. and Wang, R. Y.: Serverless Network File Systems, *ACM Transactions on Computer Systems*, pp. 109–126 (1995).

[9] Rodeh, O. and Teperman, A.: zFS - A Scalable Distributed File System Using Object Disks, *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, MSS '03, Washington, DC, USA, IEEE Computer Society, pp. 207–218 (online), available from ⟨http://dl.acm.org/citation.cfm?id=824467.824995⟩ (2003).

[10] Teperman, A. and Weit, A.: Improving Performance of a Distributed File System Using OSDs and Cooperative Cache.

[11] Sarkar, P. and Hartman, J. H.: Hint-based Cooperative Caching, *ACM Trans. Comput. Syst.*, Vol. 18, No. 4, pp. 387–419 (online), DOI: 10.1145/362670.362675 (2000).

[12] Tanaka, M. and Tatebe, O.: Pwrake: A Parallel and Distributed Flexible Workflow Management Tool for Wide-area Data Intensive Computing, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, New York, NY, USA, ACM, pp. 356–359 (online), DOI: 10.1145/1851476.1851529 (2010).

[13] Zhao, D., Qiao, K. and Raicu, I.: HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems, *IEEE/ACM CCGrid* (2014).

[14] Schmuck, F. and Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters, *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, USENIX Association, (online), available from ⟨http://dl.acm.org/citation.cfm?id=1083323.1083349⟩ (2002).

[15] Li, T., Zhou, X., Brandstatter, K., Zhao, D., Wang, K., Rajendran, A., Zhang, Z. and Raicu, I.: ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table, *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, Washington, DC, USA, IEEE Computer Society, pp. 775–787 (online), DOI: 10.1109/IPDPS.2013.110 (2013).

© 2014 Information Processing Society of Japan