GfarmファイルシステムへのCooperative Cachingの実装

佐々木　慎[1]　松宮　遼[1]　高橋　一志[1,2]　大山　恵弘[1,2]

概要：高性能科学計算の分野では，大規模なデータを高速に処理するために分散ファイルシステムの性能向上が求められている．メモリへのアクセスはディスクへのアクセスに比べて高速なため，メモリキャッシュを有効に利用することで性能向上が期待できる．しかしながら，分散ファイルシステムにおいて，データを保存するストレージノードのメモリ容量には限界がある．この問題を解決する手法として cooperative caching がある．これは，分散ファイルシステムのクライアントが，他クライアントのメモリキャッシュを利用することを可能とする手法である．本研究では，高性能科学計算に利用される分散ファイルシステムである Gfarm ファイルシステムに cooperative caching を実現する機構を提案する．提案機構は Linux カーネルが提供するページキャッシュを利用する．本論文では，提案機構の設計と予備評価について示す．

Implementing Cooperative Caching for Gfarm File System

Shin Sasaki[1]　Ryo Matsumiya[1]　Kazushi Takahashi[1,2]　Yoshihiro Oyama[1,2]

Abstract: In the field of high performance computing, there is a demand for improving the performance of distributed file systems to deal with large data more efficiently. A memory cache is important to achieve high performance in file systems because fetching data from memory is much faster than fetching data from disks. However, the amount of memory in storage nodes is limited. Cooperative caching is a method that can solve this problem. This method enables distributed file system clients to read the memory cache of other clients. We propose a cooperative caching mechanism for Gfarm distributed file system. The proposed mechanism makes use of page caches provided by Linux kernel. In this paper, we show the design of the mechanism and report the results of preliminary evaluation.

1. はじめに

高性能科学計算の分野において，大規模なデータを扱うために分散ファイルシステムが広く用いられている．分散ファイルシステムは，ネットワークでつながれた 1 つ以上のノードのストレージにデータを配置し，それらを複数のノードから 1 つのストレージのように扱うことを可能にするファイルシステムである．分散ファイルシステムは一般的に，ストレージを提供するストレージノードとメタデータを管理するメタデータサーバから構成される．また，分散ファイルシステムを利用するノードはクライアントと呼ばれる．分散ファイルシステムでは，大規模なデータを扱うことができる一方で，ネットワークの遅延やメタデータ管理のオーバヘッドによる性能低下が問題となっている．大規模データを高速に処理する必要のある高性能科学計算において，分散ファイルシステムの性能向上は重要な課題である．

多くの分散ファイルシステム [1)2)3)4)5)] では，複数のストレージノードを組み合わせてストレージを構成し，データはそれらのノードのローカルディスクに配置される．クライアントがデータにアクセスする際，ストレージノードのメモリキャッシュにデータが存在する場合は高速にデータを読むことが可能である．しかしながら，ストレージノードのメモリ容量には限りがあるため，メモリにキャッシュできるデータの量には限界がある．メモリキャッシュにデータが存在しない場合は，ディスクからデータを読むが，ディスクへのアクセスにはメモリへのアクセスに比べて時間がかかる．例えば，2013 年に Seagate から発売

1 電気通信大学
The University of Electro-Communications
2 独立行政法人科学技術振興機構，CREST
JST，CREST

された HDD である ST4000DM000 の平均データ転送速度は 146MB/sec であり，InfiniBand (1xDDR) の論理スループットは 4Gbit/sec である．このように，ディスクはネットワークに比べて非常に低速である．高性能科学計算が行われるような高速なネットワーク環境においては，ディスクアクセスにかかる時間は大きなボトルネックになることが予想される．

本研究において我々は，次のような環境を想定する．分散ファイルシステムを構成するノードとクライアントノードはローカルディスクとして HDD を搭載している．ノード間は InfiniBand のような高速なネットワークによって接続され，どのノード間の通信遅延も極端に大きくない．分散ファイルシステムを利用するクライアントは 1 万以上である．

分散ファイルシステムの性能を向上させる手法として cooperative caching [6] がある．Cooperative caching とは，分散ファイルシステムを利用するクライアントのメモリにデータをキャッシュし，それを他クライアントが利用することを可能にする手法である．Cooperative caching に関しては，キャッシュのロケーションを検索する負荷を分散するためのヒントベースのアルゴリズム [7] や，ローカリティを考慮したクライアント間でのキャッシュの配置アルゴリズム [8] などが存在する．

Cooperative caching により，分散ファイルシステム全体で大容量のメモリキャッシュを構成することが可能となる．cooperative caching については多くの研究が存在する．しかし, InfiniBand のような高速なネットワークが環境において，分散ファイルシステムに cooperative caching を導入したときの性能については明らかにされていない．本研究では，分散ファイルシステム Gfarm [1] に cooperative caching を実装し，性能の向上を図る．対象とする OS は Linux である．将来的には InfiniBand で通信する環境で実験を行う予定であるが，その前段階として，Gigabit Ethernet で通信する環境で予備実験を行ったので，本論文ではその結果を報告する．

本論文の構成は以下のとおりである．2 章で Gfarm について述べる．3 章で提案機構の設計と実装について説明し，4 章でその予備評価について述べる．5 章で関連研究について述べ，6 章で本論文のまとめと今後の課題について述べる．

2. Gfarm

2.1 概要

Gfarm ファイルシステムはメタデータサーバと I/O サーバから構成される．メタデータサーバはファイルの inode 番号，サイズ，アクセス権などのメタデータや I/O サーバの情報，ユーザアカウント，ファイルのロケーションなどを管理する．I/O サーバは Gfarm ファイルシステムに複数存在し，ファイルをローカルのストレージに保存する．ファイルの保存にはローカルファイルシステムが利用される．Gfarm ファイルシステムでは，クライアントが I/O サーバを兼ねることが可能であり，ローカリティを考慮したデータ配置を行うことで計算性能の向上を図っている．クライアントは，専用のライブラリ API を用いて Gfarm ファイルシステム上のデータにアクセスする．クライアントによる Gfarm ファイルシステム上のファイルへのアクセスは**図 1** のように行われる．クライアントはファイルを open するために，まずメタデータサーバに問い合わせを行う．メタデータサーバは当該ファイルを保持する I/O サーバの情報やファイルのメタデータをクライアントに返す．次に，クライアントは I/O サーバに対して open を発行する．I/O サーバはメタデータサーバに問い合わせてクライアントの認証を行う．ファイルの open 後は，I/O サーバに対して read や write 等の処理を直接要求する．ファイルの open 後は，メタデータサーバを介さず I/O サーバと直接通信を行うことで，メタデータサーバへの負荷を軽減している．

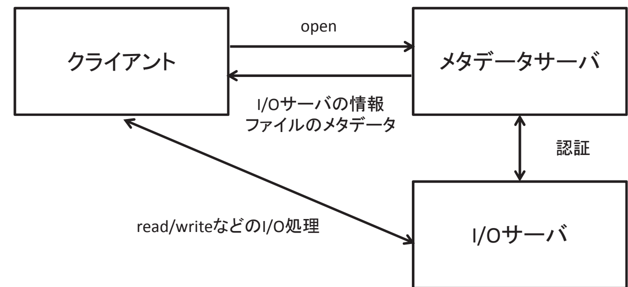

図 1 Gfarm ファイルシステムのファイルアクセス
Fig. 1 File access in Gfarm file system

2.2 gfarm2fs

クライアントは gfarm2fs を用いて Gfarm ファイルシステムをローカルファイルシステムにマウントすることが可能である．gfarm2fs はユーザレベルファイルシステムを構成するためのフレームワークである FUSE [9][10] を利用して実装されている．gfarm2fs を用いて Gfarm ファイルシステムへアクセスする場合の構成を**図 2** に示す．gfarm2fs はクライアントのユーザ空間でデーモンとして動作する．アプリケーションプロセスがローカルファイルシステムにマウントされた Gfarm ファイルシステムに対して I/O 処理を発行すると，カーネル空間で動作する FUSE モジュールへ VFS を介して処理が渡る．FUSE モジュールは gfarm2fs に Gfarm ファイルシステムとの通信を依頼する．gfarm2fs はメタデータサーバや I/O サーバと通信し，結果を FUSE モジュールに返す．FUSE モジュールは，VFS を介してアプリケーションプロセスに結果を返す．

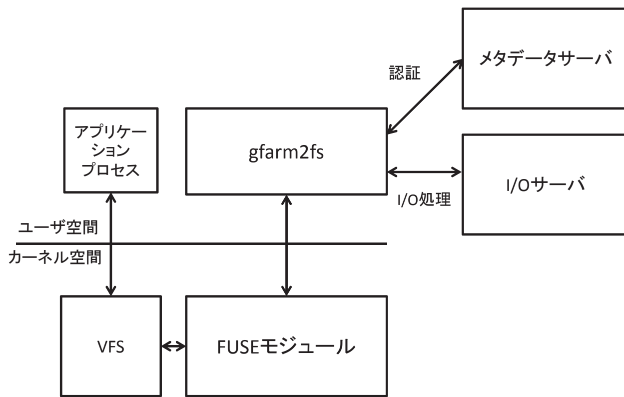

図 2 gfarms2fs を用いた場合の Gfarm の構成
Fig. 2 The architecture of Gfarm with gfarms2fs

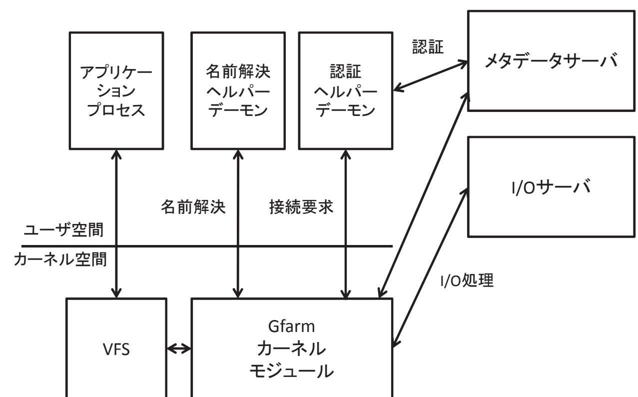

図 3 カーネルモジュールを用いた場合の Gfarm の構成
Fig. 3 The architecture of Gfarm with kernel module

2.3 Gfarm カーネルモジュール

クライアントはカーネルモジュールを用いて Gfarm ファイルシステムをカーネルにファイルシステムとして登録し，VFS から直接アクセスすることも可能である．カーネルモジュールは Gfarm ファイルシステムのクライアントにロードされる．カーネルモジュールを用いて Gfarm ファイルシステムへアクセスする場合の構成を**図 3** に示す．カーネルモジュールは，クライアントがカーネル空間から Gfarm ファイルシステムにアクセスすることを可能にする．クライアントがカーネルモジュールをロードすると，Gfarm がカーネルにファイルシステムとして登録される．それと同時に，ユーザ空間で動作するヘルパーデーモンが起動される．ヘルパーデーモンには，メタデータサーバとの接続認証を担うものと，ユーザ名，グループ名，ホスト名の名前解決を担うものが存在する．カーネルモジュールを用いて Gfarm ファイルシステムにアクセスすることで，gfarm2fs を用いるよりもユーザ空間とカーネル空間の間のコンテキストスイッチとメモリコピーの回数が減少し，高速な I/O 処理が可能となる．

カーネルモジュールでは，クライアントが I/O サーバを兼ねている場合には，ユーザ空間とカーネル空間の間でバッファのコピーが発生しないので gfarm2fs に比べてオーバヘッドが小さい．それに対して gfarm2fs では，クライアントが I/O サーバを兼ねている場合であっても，ユーザ空間で動作する gfarm2fs とカーネル空間で動作する FUSE モジュールの間でバッファのコピーが発生する．

3. 提案機構

3.1 設計

Gfarm ファイルシステムからファイルを read したクライアントのメモリには，OS により当該ファイルのキャッシュが作られる．そこで，提案機構はクライアントが Gfarm ファイルシステムから read するとき，他クライアントが保持しているキャッシュからの read を可能にする．ページを単位として作られるファイルのキャッシュを，Linux ではページキャッシュと呼ぶ．ページキャッシュはファイルの inode 番号とページインデックスにより指定される．ページインデックスは，ファイルをページ単位に区切ったときの各ページのインデックスである．ページサイズはすべて 4KB であるとする．

提案機構では，cooperative caching を実現するためにページキャッシュの検索および要求されたページキャッシュを他クライアントに転送するデーモンを導入する．このデーモンは，クライアント上で動作する．また，カーネルモジュールを拡張し，I/O サーバへの I/O アクセスの前に，他クライアントへのページキャッシュの要求処理および要求したページキャッシュの受信処理を追加する．提案機構導入後のファイルの read 処理の概要を**図 4** に示す．各クライアントと I/O サーバは，inode 番号とそれに対応するファイルをキャッシュしている可能性のあるクライアントとの対応が書かれたリストを保持している．このリストをキャッシュ配置リストと呼ぶことにする．ページキャッシュ単位で，クライアントとの対応が書かれたリストを保持する設計も考えられるが，管理するリストのサイズが大きくなり，リストに対する処理にかかるコストが増えるため採用しない．

デーモンはファイルを open するためにメタデータサーバに問い合わせを行い，I/O サーバの情報や当該ファイルのメタデータを得る．このとき，メタデータにはファイルの inode 番号が含まれている．デーモンは自ノードのキャッシュ配置リストを参照し，当該ファイルをキャッシュしている可能性のあるクライアントを探す．当該ファイルをキャッシュしている可能性のあるクライアントが存在する場合には，そのクライアントに inode 番号とページインデックスで指定されるページキャッシュの転送を要求する．キャッシュ配置リストに，当該ファイルをキャッシュしている可能性のあるクライアントが複数存在する場合には，あらかじめ決められた閾値数のクライアントに並列に要求する．要求を受けたクライアントは，inode 番号

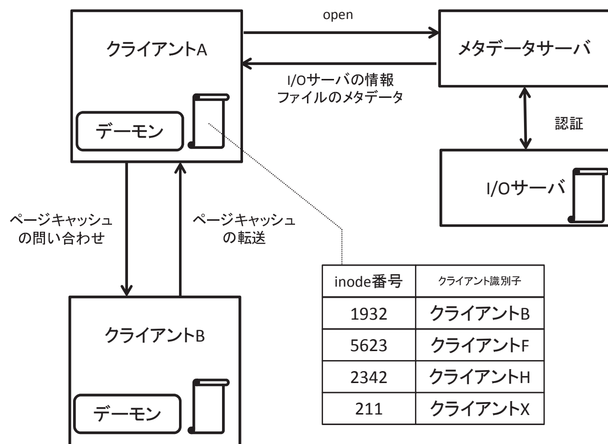

図 4 提案機構導入後の read 処理
Fig. 4 Read access using our mechanism

とページインデックスに対応するページキャッシュを検索する．ページキャッシュが存在しページの読み込みが完了している場合には，そのページキャッシュのデータを転送する．ページキャッシュが存在するがページの読み込みが完了してない場合には，ページキャッシュのデータは転送せず，要求元のクライアントにはデータを転送しないことを通知する．ページキャッシュが存在しない場合も同様に，データを転送しないことを通知する．当該ファイルをキャッシュしている可能性のある他クライアントが存在しない場合には，従来通り I/O サーバから read する．当該ファイルをキャッシュしている可能性があるクライアントが自ノードである場合にも，同様の処理を行う．この場合ページキャッシュの転送はネットワークを介さず，自ノードのページキャッシュから直接コピーすることになる．

write の処理ではコンシステンシを維持するために，書き込みのあったファイルのキャッシュを無効化する必要がある．しかし，Gfarm ファイルシステムは close-to-open セマンティクスであるため，書き込みのあったファイルのキャッシュをすぐに無効化する必要はない．あるクライアントがファイルを open し，write して close したとする．別のクライアントが同じファイルを open したときに，メタデータサーバはそのクライアントに対して，当該ファイルのキャッシュは無効であることを通知する．当該ファイルのキャッシュが無効であることを通知されたクライアントは，他クライアントにページキャッシュの転送を要求せずに，直接 I/O サーバから read する．write の処理に関する詳細な設計は今後行う．

3.2 キャッシュ配置リストの更新

提案機構において，各クライアントと I/O サーバが保持しているキャッシュ配置リストは次のような手法で更新する．はじめ，各クライアントと I/O サーバは空のリストを保持している．クライアントがファイルの read 処理のために I/O サーバにアクセスした際，I/O サーバは自ノードが保持するキャッシュ配置リストにクライアントがアクセスしたファイルの inode 番号とクライアントの対応を追加する．その後，I/O サーバはアクセス元クライアントに自ノードが持つリストを転送する．キャッシュ配置リストを受け取ったクライアントは，自ノードのキャッシュ配置リストとマージすることでキャッシュ配置リストを更新する．キャッシュの要求処理のために，クライアント間で通信する際にも，互いに自ノードのキャッシュ配置リストを転送し合いマージすることでキャッシュ配置リストを更新する．この手法では，クライアントのページキャッシュの情報を管理するサーバが存在しないため負荷が分散される．そのため，クライアントの台数に対して性能がスケールすると考えられる．しかし，この手法では各ノードでリストの送受信やマージにかかる処理のオーバヘッドは大きくなる．今後，リストのデータ構造やマージのアルゴリズムを工夫し，最適化を行っていく．

3.3 実装

デーモンによるページキャッシュの検索および他クライアントへの転送を行うデーモンの実装と，カーネルモジュールの拡張について述べる．これらのうち，デーモンによるページキャッシュの検索とクライアント間でのページキャッシュの転送の処理は実装済みであり，キャッシュ配置リストからクライアントを探す処理やリストを更新する処理については未実装である．実装済みの I/O 処理は read のみであり，write には対応していない．

3.3.1 Gfarm カーネルモジュールの拡張

Gfarm カーネルモジュールをロードすると Gfarm がファイルシステムとしてカーネルに登録されるため，Gfarm ファイルシステムのファイルへの I/O 処理に対しては，対応する Gfarm ファイルシステムのファイル操作関数群が VFS により呼び出される．ファイル操作関数群は，`struct file_operations` 構造体にカーネルモジュールのロード時に登録されている．Gfarm ファイルシステムが保持するファイルへの read に対してはカーネルが提供する標準の read 関数である `do_sync_read()` 関数が呼ばれ，最終的には `do_generic_file_read()` 関数が呼ばれ read を処理するようになっている．

`do_generic_file_read()` 関数に，他クライアントへのページキャッシュ転送要求の処理を加えることで，Gfarm ファイルシステムが保持するファイルへの read 時に，他クライアントへのページキャッシュの転送要求が行われるようにした．

3.3.2 ページキャッシュの検索および転送

ページキャッシュの検索とは，他クライアントからの要求を受けたデーモンが inode 番号とページインデックスを受け取り，対応するページキャッシュを得る処理である．

クライアントはinode番号とページインデックスを指定し，ページキャッシュの転送を要求する．この処理を実現するためにデーモンは，OSがページキャッシュ管理に用いているデータ構造を得る必要がある．Linuxでは，ページキャッシュは`struct page`構造体により管理されている．クライアントから要求されたinode番号とファイル内オフセットに対応する`struct page`構造体を得るために，辿っていく構造体について説明する．`struct page`構造体を得るために構造体を辿る流れを**図5**に示す．図5では，例として辿る構造体を，灰色に塗られた四角形で表している．

Linuxではファイルはinodeとして定義され，`struct inode`構造体により表現される．ページキャッシュはinode単位に作られ，`struct address_space`構造体により表現される．`struct inode`構造体は，対応する`struct address_space`構造体へのポインタを`i_mapping`メンバに，inode番号を`i_ino`メンバにそれぞれ保持している．ファイルシステムは`struct file_system_type`構造体で定義され，マウントポイントごとに`struct super_block`構造体に表現されるスーパーブロックが作られる．`struct file_system_type`構造体は，ファイルシステムの登録名を`name`メンバに，`struct super_block`構造体のリストへのポインタを`fs_supers`メンバに保持している．また，`struct super_block`構造体は，`struct inode`構造体のリストを`s_inodes`メンバに保持している．`get_fs_type()`関数の引数に，Gfarmファイルシステムの登録名である“gfarm”を与えることでGfarmファイルシステムを定義する`struct file_system_type`構造体を取得する．

`find_get_page()`関数の第1引数に，取得した`struct inode`構造体が保持する`address_space`構造体を，第2引数にページインデックスを与えることで，対応するページキャッシュを表す`struct page`構造体を得る．ここで，対応するページキャッシュが存在しない場合には`find_get_page()`関数はNULLを返す．取得したページキャッシュの読み込みが完了している場合，要求を受けた他クライアントへ要求されたページキャッシュのデータを転送する．取得したページキャッシュの読み込みが完了しているかの検査は，`PageUptodate()`関数により行う．`PageUptodate()`関数は，`struct page`構造体を引数に与えると，引数に与えられたページの読み込みが完了していれば1を，そうでなければ0を返す関数である．

4. 予備評価

4.1 ページキャッシュ転送性能の評価

他クライアントからのページキャッシュ転送の性能を評価した．実験環境を**表1**に示す．Gigabit Ethernetのネットワーク環境で測定を行った．実験に用いた計算機は同一LAN内にある．

以下の各条件で1GBのファイルをreadするのに要する

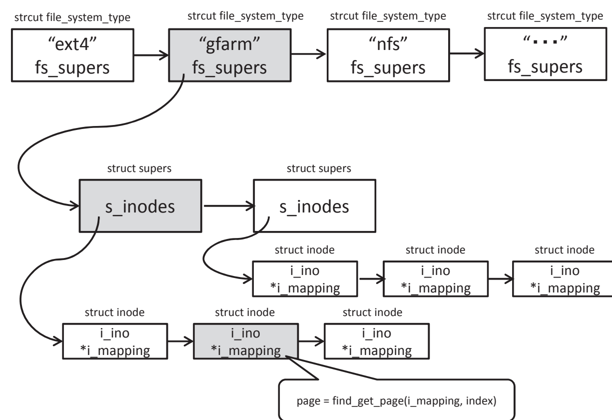

図5　ページキャッシュの検索の流れ
Fig. 5　The flow of page cache lookup

時間を測定した．

(a) gfarm2fsで，リモートのI/Oサーバにキャッシュが存在しないファイルをバッファサイズ1MBでread
(b) gfarm2fsで，リモートのI/Oサーバにキャッシュが存在するファイルをバッファサイズ1MBでread
(c) gfarm2fsで，リモートのI/Oサーバにキャッシュが存在しないファイルをバッファサイズ4KBでread
(d) gfarm2fsで，リモートのI/Oサーバにキャッシュが存在するファイルをバッファサイズを4KBでread
(e) Gfarmカーネルモジュールで，リモートのI/Oサーバにキャッシュが存在しないファイルをread
(f) Gfarmカーネルモジュールで，リモートのI/Oサーバにキャッシュが存在するファイルをread
(g) 変更を加えたGfarmカーネルモジュールで，他クライアントのページキャッシュからread

(a)，(b)，(c)，(d)，(e)，(f)ではメタデータサーバと同一ノード上で動作するクライアント(ノードA)が，別ノードのI/Oサーバ(ノードB)からファイルをreadしている．(g)ではメタデータサーバと同一ノード上で動作するクライアント(ノードA)が，別ノードのクライアント(ノードC)のページキャッシュをreadしている．

gfarm2fsがリモートのI/Oサーバからファイルをreadするときのバッファサイズは，デフォルトで1MBである．バッファサイズはGfarmファイルシステムの設定ファイル`gfarm2.conf`に`client file bufsize`の設定を書き加えることで変更できる．カーネルモジュールがリモートのI/Oサーバからファイルをreadするときのバッファサイズはページサイズ(4KB)であり．現在のところ変更はできない．

catコマンドで1GBのファイルをreadし出力を/dev/nullにリダイレクトするのに要した時間からread処理のスループットを求めた．catコマンドが発行するreadシステムコールのバッファサイズは8KBである．各測定の前に，I/Oサーバのキャッシュをクリアしている．I/Oサーバに

表 1 実験環境
Table 1 Experimental environment

	ノード A	ノード B・C
CPU	Intel Core i7-3930K 3.80GHz (6core)	Intel Core i7-870 2.93GHz (4core)
メモリ	32GB	16GB
ストレージ	SSD 120GB 6Gbps	HDD 1.5TB 5400rpm
OS	CentOS 6.2 64bit	CentOS 6.2 64bit

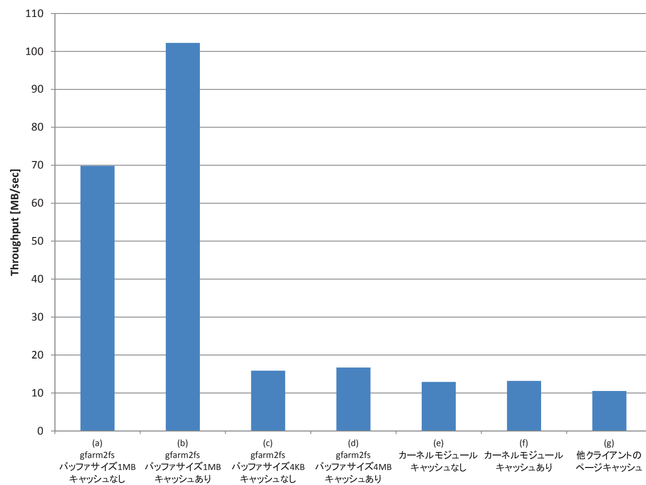

図 6 1GB のファイルの read スループット
Fig. 6 Throughput of reading a 1GB file

キャッシュが存在する場合の測定については，同一ファイルに対して cat コマンドを 2 回連続して実行した．1 回目の実行で可能な限りファイル全体をキャッシュに載せ，2 回目の実行に要する時間を測定した．

4.2 議論

測定結果を**図 6** に示す．(a)，(b) のスループットに比べ，それ以外のスループットが低くなっている．バッファサイズが (a)，(b) では 1MB であり，それ以外では 4KB であることが影響している．これより，4KB はリモートの I/O サーバに対する read のバッファサイズとして小さすぎると考えられる．

(a) より (b)，(c) より (d)，(e) より (f) のスループットが高くなっている．これは，I/O サーバ上のメモリキャッシュの効果を示している．

(c) のスループットが (e)，(f) に比べてやや低くなっているのは，他クライアントからの要求を受けてページキャッシュを検索するオーバヘッドのためである．現在の実装では，他クライアントからのページキャッシュ要求のたびに inode 番号から対応する `struct inode` 構造体を検索する処理を行っているが，inode 番号と `struct inode` 構造体の対応リストをあらかじめ作っておくなどの工夫をすることで検索の高速化は可能であると考える．また，提案機構ではページキャッシュの検索および他クライアントへの転送を行うデーモンがクライアント上で動作しているが，他クライアントからのページキャッシュ要求の受付やページキャッシュの転送などの通信はユーザレベルで行われている．そのため，ページキャッシュを他クライアントへ送信する際に，ユーザ空間とカーネル空間の間のコンテキストスイッチのオーバヘッドやカーネル空間のバッファからユーザ空間のバッファに要求されたページキャッシュをコピーするオーバヘッドが生じる．現在デーモンが行っているページキャッシュの送信処理の一部をカーネル内で実行するように実装を変更することにより，これらのオーバヘッドは削減することが可能であると考える．

5. 関連研究

Shark [11] は，クライアントが互いに信用できない環境における cooperative caching の手法を実装した分散ファイルシステムである．Shark ではファイルを可変長のチャンクに分割し，ファイルシステム全体でインデキシングしている．ファイルアクセスはチャンク単位で行われ，複数の他クライアントに並列してアクセスすることが可能である．性能評価を行い cooperative caching による分散ファイルシステムの性能向上を示している．しかし，ローカルエリアネットワークにおける性能評価では，ネットワークのスループットが 11.14MB/sec と低く，我々の想定する環境とは異なる．

NFS-cc [12] は，NFS に cooperative caching を実装した分散ファイルシステムである．NFS-cc では．単一のサーバがキャッシュの位置を管理している．性能評価を行い，NFS に比べて read のスループットが高いことを示している．単一のサーバがキャッシュの位置を管理している点で，我々の研究とは異なる．また，実験で使用しているクライアントノードの CPU が 400MHz，メモリが 64MB であり，我々の想定する環境よりも性能が低い．

C2Cfs [13] は，分散ファイルシステムのクライアント間でファイルのキャッシュを共有するためのシステムである．C2Cfs は NFS を対象にしており，また，実験は 100Mbps で通信する環境で行われている．すなわち，この研究は高性能科学計算を行う環境を対象にしていない．本研究は，メタデータサーバと複数の I/O サーバを組み合わせてストレージを構成する分散ファイルシステム，および，InfiniBand などの高速なネットワーク環境を対象にしている点で，この研究と異なる．

Jiang らによる研究 [8] では，ローカリティを考慮したクライアント間でのキャッシュの配置アルゴリズム LAC を提案し，評価を行い有効性を示している．しかし，評価はシミュレーションによるもののみである．

以上のように，既存研究においても分散ファイルシステムに cooperative caching を実装し性能評価を行い，coop-

© 2013 Information Processing Society of Japan

erative caching が分散ファイルシステムの性能向上に効果があることが示されている．我々は，これらを参考にネットワークやマシン性能が高い非広域環境での cooperative caching について研究している．

6. おわりに

本論文では，Gfarm ファイルシステムのために cooperative caching を実現する機構について提案した．また，他クライアントのページキャッシュを read する機構を Gfarm ファイルシステムに実装し，性能について予備評価を行った．今後の課題としては，まず，write 処理において，データのコンシステンシを維持する機構の実装が挙げられる．また，高性能科学計算で用いられる環境とアプリケーションを用いて実験を行うことも今後の課題である．

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，科学技術振興機構戦略的創造研究推進事業 (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) Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (2010).
2) Babu, A.: GlusterFS, `http://www.gluster.org/`.
3) Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. and Maltzahn, C.: Ceph: A Scalable, High-performance Distributed File System, *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 307–320 (2006).
4) Donovan, S., Huizenga, G., Hutton, A., Ross, C., Petersen, M. and Schwan, P.: Lustre: Building a File System for 1,000-node Clusters, *Proceedings of the Linux Symposium* (2003).
5) Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google File System, *Proceedings of the ACM Symposium on Operating System Principles (SOSP'03)*, ACM, pp. 29–43 (2003).
6) 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 Symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 267–280 (1994).
7) Sarkar, P. and Hartman, J.: Efficient Cooperative Caching Using Hints, *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pp. 35–46 (1996).
8) Jiang, S., Petrini, F., Ding, X. and Zhang, X.: A Locality-aware Cooperative Cache Management Protocol to Improve Network File System Performance, *Proceedings of the 26th International Conference on Distributed Computing Systems*, IEEE, pp. 42–42 (2006).
9) Szeredi, E.: FUSE: Filesystem in Userspace, `http://fuse.sourceforge.net/`.
10) Rajgarhia, A. and Gehani, A.: Performance and Extension of User Space File Systems, *Proceedings of the 2010 ACM Symposium on Applied Computing*, ACM, pp. 206–213 (2010).
11) Annapureddy, S., Freedman, M. J. and Mazieres, D.: Shark: Scaling File Servers via Cooperative Caching, *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, USENIX Association, pp. 129–142 (2005).
12) Xu, Y. and Fleisch, B. D.: NFS-cc: Tuning NFS for Concurrent Read Sharing, *International Journal of High Performance Computing and Networking*, Vol. 1, No. 4, pp. 203–213 (2004).
13) Ermolinskiy, A. and Tewari, R.: C2Cfs: A Collective Caching Architecture for Distributed File Access, *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications*, IEEE, pp. 642–647 (2009).

© 2013 Information Processing Society of Japan