Vol.2013-OS-126 No.3
2013/7/31

# キャッシュポリシの動的変更による分散ファイルシステムのメタデータサーバー負荷低減手法

松野 雅也[1]　津邑 公暁[1]　齋藤 彰一[1]　松尾 啓志[1]

**概要**：スケーラブルなストレージを実現する分散ファイルシステムであるが，その構造上，メタデータを管理するメタデータサーバに処理が集中するという問題がある．この問題を回避するために、クライアント側でメタデータのキャッシュを保持する手法が広く用いられている．この手法を適用する場合には，クライアントの保持するキャッシュの一貫性を保証する必要があり，その一貫性制御手法として，サーバが主体となる手法とクライアントが主体となる手法の 2 つが存在する．しかし，どちらの手法においても，一貫性を保証するための制御メッセージのために，システムの性能が低下してしまうようなアクセス状況が存在する．そこで，メタデータへのアクセス状況に応じて，ファイルごとにキャッシュポリシを動的に設定・変更する手法を提案する．提案手法により，各キャッシュポリシで発生する制御メッセージを抑制し，メタデータサーバの負荷を軽減できることを確認した．

**キーワード**：分散ファイルシステム，メタデータ，キャシュポリシ

## 1. はじめに

インターネットの普及，ネットワークの高速化に伴い，処理要求されるデータ量が増大し続けている．このような状況において，単一のストレージによる管理は拡張性に乏しく，さらには単一障害点になるという問題がある．そのため，複数のノードに分散したストレージを構成する分散ファイルシステムが広く利用されている．分散ファイルシステムは，一般的に，ファイルの管理情報であるメタデータを管理するメタデータサーバとファイルデータの実体を管理するストレージノードから構成される．そして，メタデータを単一のメタデータサーバで集中的に管理することで，ストレージに対するアクセス並行性を高め，システム全体の性能向上を実現している．しかし，このような構成の分散ファイルシステムでは，ファイルに対する入出力操作は分散されるが，メタデータ操作はメタデータサーバに集中する．そのため，メタデータサーバがシステムのボトルネックとなるという問題が存在する．この問題を回避するために，クライアント側でメタデータのキャッシュを保持する手法が広く用いられている．この手法は，クライアントが一度アクセスしたファイルのメタデータを，自身のローカルメモリ上にキャッシュしておくことで，メタデータサーバへのアクセス回数を減らし，メタデータサーバの負荷を軽減するものである．しかし，この手法を用いる場合には，メタデータが更新された時に，クライアントが保持しているメタデータキャッシュの一貫性を保証しなければならない．メタデータキャッシュの一貫性を保証する方法としては主に，メタデータサーバが更新を通知する方法，クライアントが更新を確認する方法の 2 つがある．本稿は，前者をサーバベース，後者をクライアントベースと呼ぶ．サーバベース，クライアントベースはそれぞれ，一貫性を保証するための制御メッセージにより，システムの性能が低下してしまうようなアクセス状況が存在する．そこで本稿では，これら 2 つのキャッシュポリシを，ファイルを単位として設定し，そのアクセス状況に応じてキャッシュポリシを動的に変更する手法を提案する．さらに，提案手法を分散ファイルシステム Gfarm[1], [2] に実装することにより，各キャッシュポリシで発生する一貫性を保証するための制御メッセージを抑制し，メタデータサーバの負荷を軽減できることを確認する．

以降の章では，まず 2 章で一貫性制御手法であるサーバベース，クライアントベースの動作について説明する．次に，3 章でファイル単位でのキャッシュポリシの設定に関して述べ，4 章でキャッシュポリシの動的変更方法について説明する．そして，5 章で提案手法の評価を行い，6 章でまとめと今後の課題に関して述べる．最後に，7 章で本

[1] 名古屋工業大学
Nagoya Institute of Technology

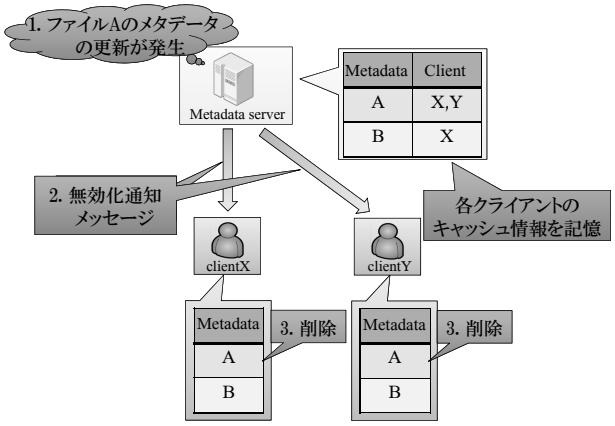


**図 1** サーバベースでの一貫性制御

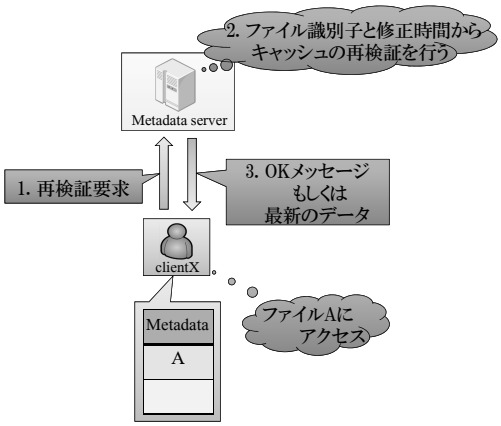


**図 2** クライアントベースでの一貫性制御

研究の関連研究に関して述べる．

## 2. 既存のキャッシュポリシ

本章では，分散ファイルシステムにおいてクライアントが保持するメタデータキャッシュの一貫性を保証する既存の手法である，サーバベースとクライアントベースの動作について説明する．

### 2.1 サーバベース

サーバベースにおける一貫性制御の様子を**図 1** に示す．サーバベースでは，メタデータサーバがどのクライアントが，どのファイルのメタデータをキャッシュしているかを管理する表を保持している．メタデータサーバにおいてメタデータの更新が発生すると，この表を参照し，更新されたメタデータをキャッシュしている全クライアントに対し，キャッシュが無効になったことを通知するメッセージ（以下，これを無効化通知メッセージと呼ぶ）を送信する．このメッセージを受信したクライアントは，メタデータが更新されたことを把握し，自身のローカルメモリ上から無効になったメタデータキャッシュを削除することで一貫性を保証する．このように，サーバベースではメタデータサーバ側が更新を通知することで，メタデータキャッシュの一貫性を保証している．

### 2.2 クライアントベース

クライアントベースにおける一貫性制御の様子を**図 2** に示す．クライアントベースでは，クライアントがメタデータキャッシュの利用時に，キャッシュの有効性を確認することで一貫性を保証している．つまり，クライアントはメタデータキャッシュを利用する前に，メタデータサーバに対して，そのキャッシュが更新されているかどうかを問い合わせるメッセージ（以下，これを再検証要求と呼ぶ）を送信する．メタデータサーバは再検証要求を受け取ると，ファイルの識別子と修正時間からクライアントが保持するメタデータキャッシュが最新のものであるかを検証し，最新のものであった場合には OK メッセージを，そうでない場合には最新のメタデータをクライアントに送信する．これにより，再検証要求後にはクライアントの保持するメタデータキャッシュを最新のものとすることができる．このように，クライアントベースではクライアント側が更新を確認することで，メタデータキャッシュの一貫性を保証している．

## 3. ファイル単位のキャッシュポリシ

メタデータキャッシュを利用する場合には，その一貫性を保証するために，制御メッセージが発生する．サーバベースの場合は無効化通知メッセージであり，クライアントベースの場合は再検証要求である．これらの制御メッセージが発生する条件は，キャッシュポリシ別に異なるため，メタデータへのアクセス状況に応じた適切なキャッシュポリシを，ファイル別に設定する必要があることが考えられる．本章では，サーバベース，クライアントベースにおける一貫性制御メッセージについて考察し，ファイル単位でキャッシュポリシを設定する手法を提案する．

### 3.1 サーバベースでの一貫性制御メッセージ

サーバベースのキャッシュポリシを適用した際に発生する一貫性制御メッセージは，クライアントにキャッシュの無効化を通知する無効化通知メッセージである．無効化通知メッセージは，メタデータサーバにおいてメタデータが更新された際に発生し，更新されたメタデータのキャッシュを保持する全クライアントに対して送信される．従って，メタデータの更新が発生しない限り，一貫性制御メッセージが発生することはなく，クライアントはメタデータキャッシュの有効性を確認することなく利用できることになる．しかし，メタデータの更新が集中して発生した場合には無効化通知メッセージが大量に発生する可能性がある．また，無効化通知メッセージを受信したクライアントはキャッシュを削除するため，メタデータの更新が集中した直後のアクセスにおいて，各クライアントでキャッシュ

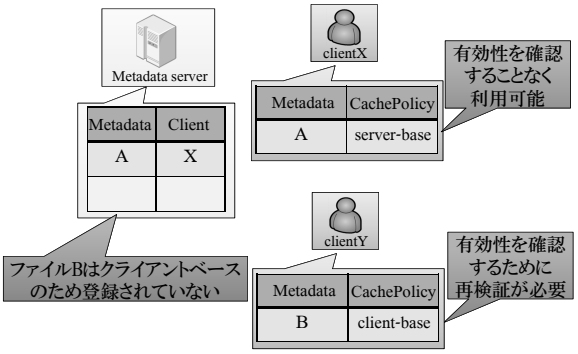


**図 3** ファイル単位でのキャッシュポリシの設定

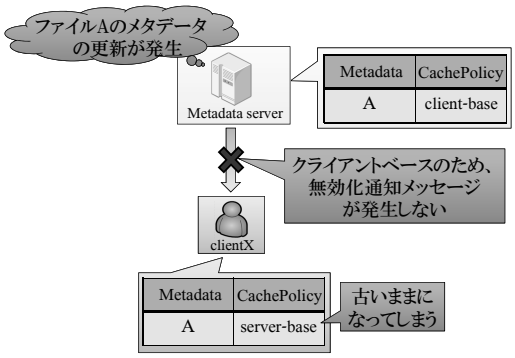


**図 4** キャッシュポリシ情報の一貫性

ミスが発生し，一時的にメタデータサーバに負荷がかかる恐れがある．

### 3.2 クライアントベースでの一貫性制御メッセージ

クライアントベースのキャッシュポリシを適用した際に発生する一貫性制御メッセージは，クライアントがメタデータキャッシュの有効性を確認する再検証要求である．再検証要求は，クライアントがメタデータキャッシュを利用する際に発生し，メタデータサーバはクライアントから受信したファイルの識別子とその修正時刻からキャッシュの再検証を行うことになる．この再検証処理は，クライアントからファイルの識別子を受け取ることで，ファイルのパス解決を行う必要はないため，処理は軽い．従って，メタデータの更新時に一貫性制御メッセージが発生することはなく，サーバベースのようにメタデータの更新が集中した場合にメタデータサーバの負荷が増大することはない．しかし，メタデータキャッシュを利用する際に必ず再検証要求が発生するため，メタデータが更新されていない場合には，再検証要求がオーバヘッドとなる恐れがある．

### 3.3 キャッシュポリシの設定

前節までの考察から，サーバベースで発生する無効化通知メッセージの増大はクライアントベースでは発生せず，逆にクライアントベースで発生する再検証要求によるオーバヘッドはサーバベースでは発生しないことが分かる．しかし，既存の手法ではすべてのファイルに対して同じキャッシュポリシを適用するため，上記 2 種類の一貫性制御メッセージによる影響を回避することはできない．

そこで，提案手法では，キャッシュポリシをファイル単位で適用することで，これら一貫性制御メッセージによる影響を軽減する手法を提案する．ファイル単位でのキャッシュポリシの設定は，通常のメタデータ情報にキャッシュポリシ情報を追加することで実現する．具体的には，クライアントは最初のメタデータ取得時にキャッシュポリシ情報が追加されたメタデータを受け取り，以降は，そのキャッシュポリシ (クライアントベースもしくはサーバベース) に基づいて，キャッシュしたメタデータの利用時における動作を判断する．つまり，**図 3** に示すように，サーバベースであった場合は，メタデータサーバが一貫性を保証するため，メタデータキャッシュの有効性を確認することなく利用でき，クライアントベースであった場合には，クライアントが一貫性を保証する必要があるため，メタデータキャッシュの有効性を確認するために再検証要求を行い，その有効性を確かめることになる．

## 4. キャッシュポリシの動的変更

ファイル単位でキャッシュポリシを設定することで，ファイルごとに適した一貫性制御を行うことが可能となる．しかし，ファイルへのアクセス状況の変化に伴い，適切なキャッシュポリシも変化することが考えられる．そこで，本章ではファイルへのアクセス状況を観測し，その変化に応じて適切なキャッシュポリシへ動的に変更する手法を提案する．

### 4.1 キャッシュポリシ情報の一貫性

キャッシュポリシを変更する場合には，メタデータキャッシュの一貫性と同様に，メタデータサーバとクライアント間でキャッシュポリシ情報の一貫性を保証する必要がある．その必要性を説明するため，**図 4** に示すように，あるメタデータのキャッシュポリシがメタデータサーバではクライアントベースであり，クライアントではサーバベースである状況を考える．この場合，メタデータが更新されるとクライアントが保持しているメタデータキャッシュは無効なキャッシュとなる．しかし，メタデータサーバでは，そのメタデータのキャッシュポリシがクライアントベースとなっているため，無効化通知メッセージが送信されない．その結果，クライアントのメタデータキャッシュは削除されず古いまま残ることになり，メタデータの一貫性が保たれなくなる．そのため，キャッシュポリシを変更する際には，何らかの方法でクライアントに対し，変更を通知する必要がある．

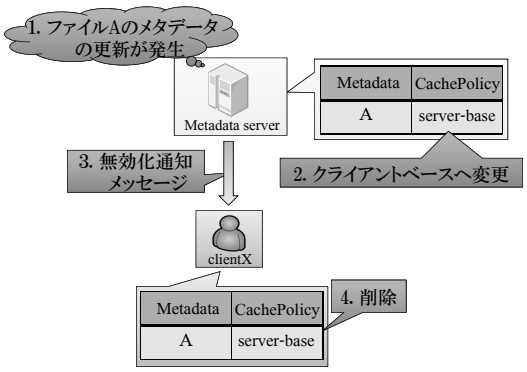


**図 5** サーバベースからクライアントベースへの変更方法 1

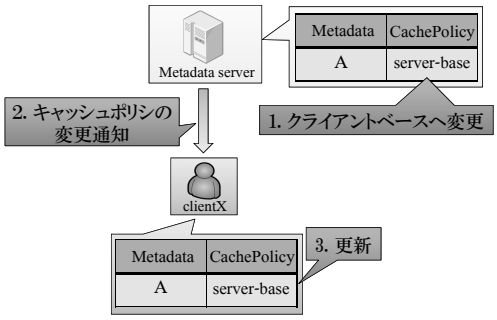


**図 6** サーバベースからクライアントベースへの変更方法 2

### 4.2 サーバベースからクライアントベースへの変更

サーバベースからクライアントベースへ変更する方法としては，2 つの方法が考えられる．一つは，キャッシュポリシの変更をメタデータの更新時に行う方法である．その変更の流れを**図 5** に示す．まず，メタデータサーバにおいてメタデータの更新が発生した時に合わせて，キャッシュポリシをサーバベースからクライアントベースへ変更する．このメタデータの更新時には，変更以前のキャッシュポリシがサーバベースであったため，無効化通知メッセージが発生する．そのため，キャッシュポリシを変更したファイルのメタデータは，クライアントのキャッシュ上から削除される．従って，クライアントはメタデータキャッシュが無効化された後，最新のメタデータを取得すると同時にキャッシュポリシ情報の変更も反映することができる．つまり，メタデータの一貫性を保証する場合と同様にしてキャッシュポリシ情報の一貫性も保証する．

もう一つは，**図 6** に示すように，キャッシュポリシの変更を明示的にクライアントへ通知する方法である．この方法では，前者の方法とは異なり，キャッシュポリシの変更をメタデータ更新時に限定されることなく，任意のタイミングで行うことが可能となる．しかし，キャッシュポリシの変更通知は無効化通知メッセージと同様に，メタデータキャッシュを保持する全クライアントに送信する必要がある．

従って，前者の方法に比べ，後者の方法はキャッシュポリシ変更のためのメッセージが余分に発生し，キャッシュポリシを変更する際のオーバヘッドとなることが考えられる．そのため，本提案手法では前者の，メタデータの更新時に変更する方法を用いる．

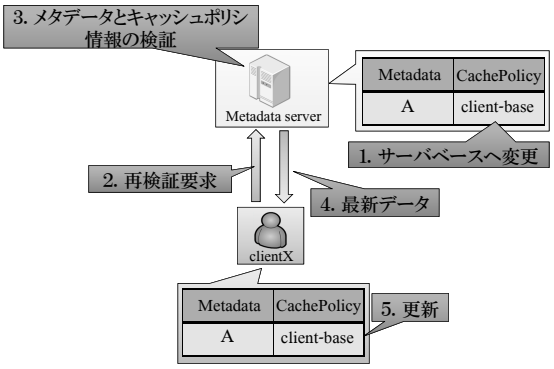


**図 7** クライアントベースからサーバベースへの変更方法

### 4.3 クライアントベースからサーバベースへの変更

クライアントベースからサーバベースへ変更する場合には，キャッシュポリシ情報を変更する際にメタデータサーバ側が変更を通知する必要はない．なぜなら，クライアントはメタデータキャッシュの利用時において，キャッシュの有効性をメタデータサーバに問い合わせるため，その時にキャッシュポリシ情報が変更されたことをクライアントのメタデータキャッシュに反映することができるからである．その変更の流れを**図 7** に示す．まず，メタデータサーバにおいてキャッシュポリシがクライアントベースからサーバベースに変更されたとする．この変更時には，メッセージは発生しない．その後，クライアントから，キャッシュポリシを変更したファイルに対する再検証要求が送信されたとする．この時，再検証要求を受信したメタデータサーバは，通常のメタデータの検証だけでなくキャッシュポリシ情報の検証も行うことで，キャッシュポリシ情報の変更も検出するようにする．そして，どちらかでも一致しない場合には，最新のメタデータを送信する．こうすることで，再検証要求後において，クライアント上のメタデータキャッシュにキャッシュポリシの変更を反映することができる．

従って，クライアントベースからサーバベースへの変更は，メタデータ更新時に限定されず，かつキャッシュポリシの変更時に別途メッセージを通知する必要もない．

### 4.4 キャッシュポリシの動的変更条件

キャッシュポリシを動的に変更する目的は，ファイルへのアクセス状況の変化に応じてキャッシュポリシを変更することで，一貫性制御メッセージによる影響を軽減することにある．つまり，サーバベースにおける無効化通知メッセージの増大を避け，かつクライアントベースにおける再検証要求によるオーバヘッドを軽減することである．この

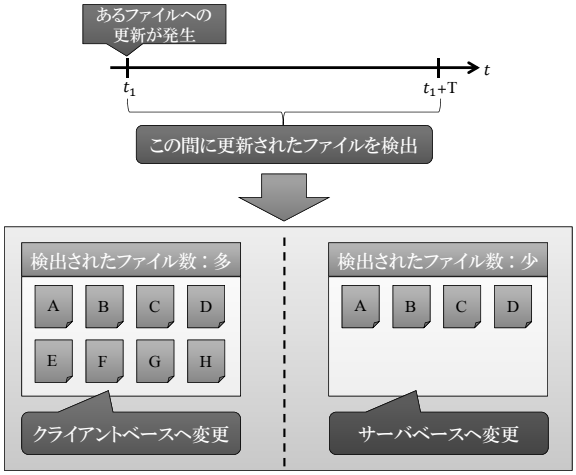


**図 8** キャッシュポリシの変更条件

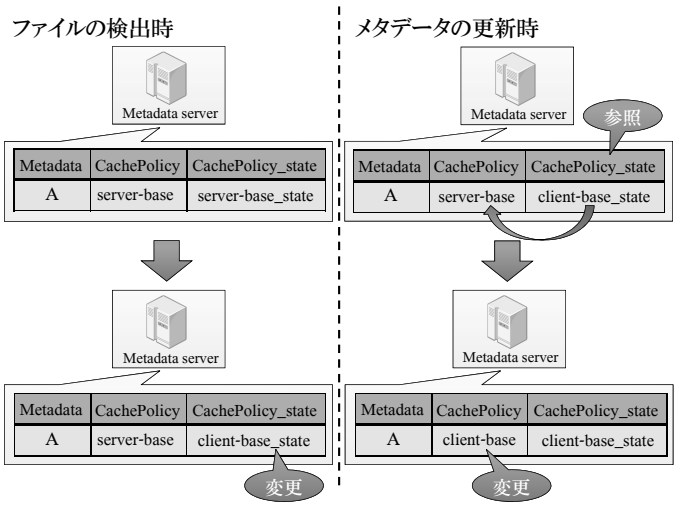


**図 9** サーバベースからクライアントベースへの動的変更

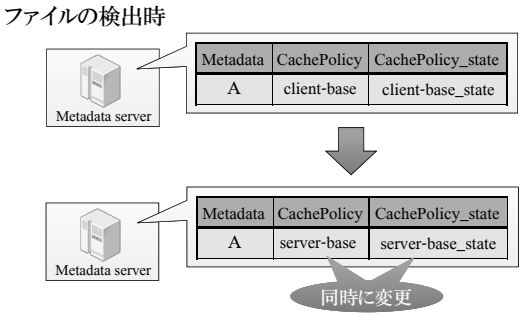


**図 10** クライアントベースからサーバベースへの動的変更

目的を達成するためには，無効化通知メッセージの増大につながるファイルのみをクライアントベースへ変更し，それ以外のファイルをサーバベースへ変更すれば良い．無効化通知メッセージが増大するのは，メタデータの更新が集中して発生した場合であるため，そのようなアクセス状況を検出する必要がある．本実装では，メタデータサーバにおいて各メタデータへの更新状況を監視することで，メタデータの更新が集中して発生したことを検出する．

具体的には，**図 8** に示すように，メタデータサーバにおいて，あるファイルのメタデータの更新が発生後，一定時間内にメタデータが更新されたファイルを検出する．そして，検出されたファイル数が多い場合にメタデータの更新が集中して発生したとみなし，無効化通知メッセージが増大すると考えられるため，その時に検出されたファイルすべてをクライアントベースへと変更する．逆に検出されたファイル数が少ない場合には，無効化通知メッセージが増大することはないと考えられるため，サーバベースへと変更する．しかし，4.1 節で述べたようにキャッシュポリシ情報の一貫性の問題があるため，サーバベースからクライアントベースへ変更する場合には，検出したファイルのキャッシュポリシをまとめて変更することはできない．

そこで，各ファイルのメタデータにキャッシュポリシ変更状態という，新たな変数を追加する．この変数は，サーバベースへ変更する状態，クライアントベースへ変更する状態のどちらか一つの状態を示す．そして，**図 9** に示すように，サーバベースからクライアントベースへ変更する場合には，まず検出されたファイルのキャッシュポリシ変更状態をクライアントベースへ変更する状態に遷移させる．この時は実際にキャッシュポリシが変更されているわけではなく，キャッシュポリシ変更状態が変わっただけとなる．実際のキャッシュポリシの変更は，各ファイルのメタデータ更新時に，自身のキャッシュポリシ変更状態を参照することで行うようにする．こうすることで，キャッシュポリシの変更をメタデータ更新時に行うことができる．

一方，クライアントベースからサーバベースへ変更する場合はキャッシュポリシを変更する際に，一貫性の問題は発生しない．そのため，**図 10** に示すように，ファイル検出時においてキャッシュポリシとキャッシュポリシ変更状態の変更を同時に行うようにすれば良い．

以上の手法により，アクセス状況に応じた，動的なキャッシュポリシの変更を行うことができる．

## 5. 実験

本章では，既存のキャッシュポリシであるサーバベースとクライアントベースにおける一貫性制御メッセージがシステムに及ぼす影響を確認する．そして，提案手法を適用した場合との比較を行い，提案手法の有効性を示す．

### 5.1 実験環境

既存のキャッシュポリシであるサーバベースとクライアントベースを分散ファイルシステムの一つである Gfarm 上にそれぞれ実装するとともに，さらに提案手法を追加実装した．実験環境は，表 1 に示す通りであり，以降，実験における Gfarm の構成はメタデータサーバを 1 台とし，クライアントノードは最大 10 台としている．各実験結果は，

5 回試行した中の最良の値を用いている．また，以降の評価では，提案手法においてキャッシュポリシの初期状態はサーバベースとしている．これは，メタデータの更新が集中して発生しない限りは，キャッシュ利用時にオーバヘッドが発生しないサーバベースで動作すべきであると考えられるためである．

**表 1** 実験環境

| | |
|---|---|
| Gfarm version | gfarm-2.5.5 |
| CPU | Core i5 750 / 2.67GHz |
| メモリ | 8GB |
| LAN | Ethernet(1000BASE-T) |

### 5.2 再検証要求による影響

まず，クライアントベースにおける再検証要求によるオーバヘッドの影響を確認するために，サーバベース，クライアントベース，提案手法それぞれの場合で，メタデータ操作の性能測定のためのベンチマークである mdtest[8] を実行し，スループット（op/sec）を測定した．mdtest では，合計 2000 個のファイルとディレクトリに対する create/stat/remove を，単一プロセスで実行している．

実験結果を**図 11** に示す．評価は，サーバベースにおけるスループットを 1 として正規化している．実験結果からクライアントベースはサーバベースと比較して，stat 操作，remove 操作においてスループットが低下していることが分かる．これは，stat 操作，remove 操作がメタデータキャッシュを利用する操作であり，サーバベースでは有効性を確認することなく利用できるのに対し，クライアントベースでは必ず再検証要求が発生するためであると考えられる．また，Dir-stat（ディレクトリに対する stat 操作）と File-stat（ファイルに対する stat 操作）で性能差に大きな違いがある．この原因を調査したところ，ディレクトリ生成後にはメタデータキャッシュが残るのに対し，ファイル生成後には強制的にメタデータキャッシュが削除される実装になっていたこと，及び 1 回の stat 操作でメタデータキャッシュを 2 回使用していたことが分かった．つまり，メタデータキャッシュ利用時において，Dir-stat では 2 回ともキャッシュヒットしていたが，File-stat ではキャッシュヒットとキャッシュミスがそれぞれ 1 回ずつ発生していた．このために，Dir-stat と File-stat で性能差に大きな違いが生じたと考えられる．一方，create 操作においては性能にほとんど差がないが，これはメタデータキャッシュを利用していない操作であるためだと考えられる．また，この実験では，メタデータの更新が集中する状況は発生せず，キャッシュポリシの変更が発生しないため，提案手法はサーバベースと同等の性能を示していると考えられる．

以上から，再検証要求によるオーバヘッドのために，クライアントベースでは性能が低下することが確認でき，サーバベースとして扱うことでこの問題を回避できることが分かる．

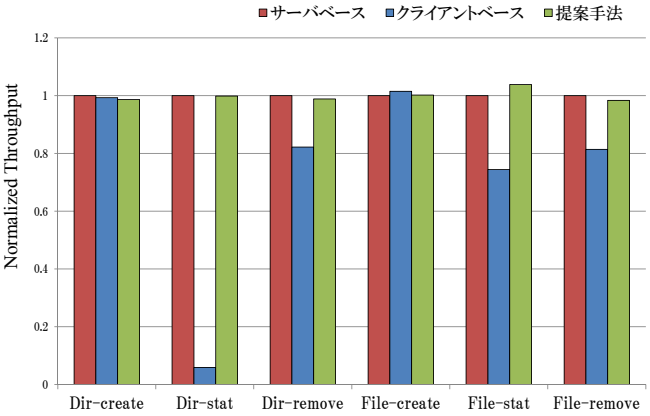


**図 11** 再検証要求によるオーバヘッドの影響

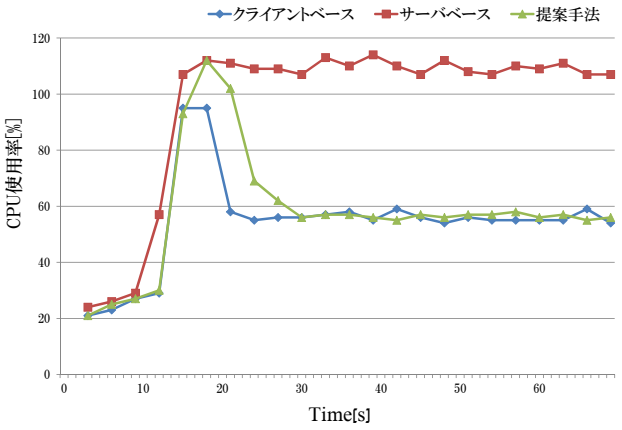


**図 12** メタデータサーバ上の CPU 使用率の推移

### 5.3 無効化通知メッセージによる影響

次に，サーバベースにおける無効化通知メッセージの影響について確認するために，メタデータの更新が集中して発生する状況を再現し，その負荷状況下における，メタデータサーバ上の CPU 使用率の推移をサーバベース，クライアントベース，提案手法それぞれの場合で測定した．メタデータが集中して更新される状況を再現するために，クライアントノード 9 台上に MPI を用いて 300 プロセスを生成し，メタデータの更新を伴う処理である touch 操作を実行する 5 プロセスと，メタデータの取得を伴う処理である stat 操作を実行する 295 プロセスに分け，特定の 100 ファイルに対して以下の操作を行った．

- touch 操作を実行する 5 プロセス
  各プロセスが 20 ファイルずつ，合計 100 ファイルに対して 1 秒間隔で，並列に touch 操作を実行
- stat 操作を実行する 295 プロセス
  100 ファイルに対して，0.01 秒ごとに stat 操作を実行

この負荷プログラムの実行時における，メタデータサーバ上の CPU 使用率の推移を，top コマンドを用いて 3 秒ご

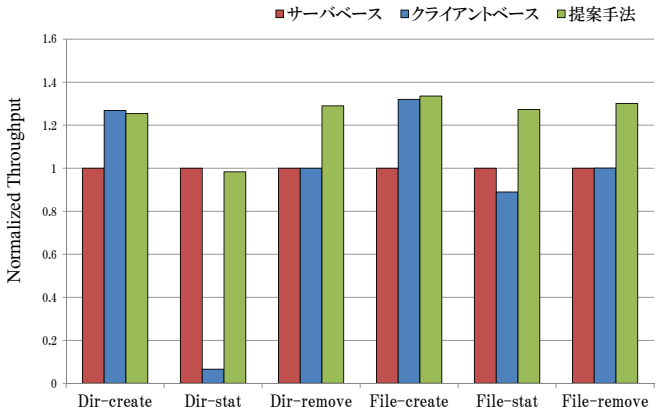


**図 13** 無効化通知メッセージによる影響

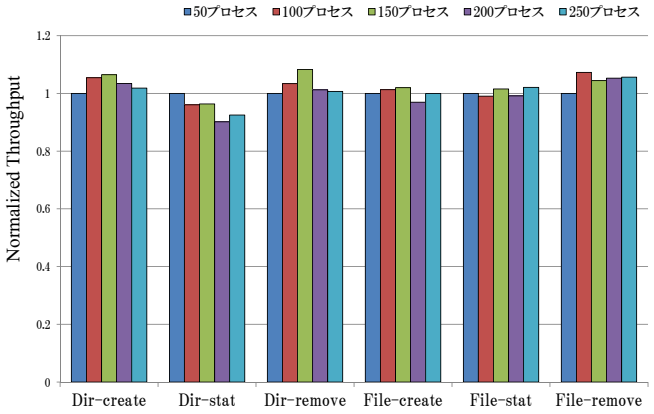


**図 14** クライアントベース：stat 操作を行うプロセス数による影響

とに計測した．

実験結果を**図 12** に示す．実験から，サーバベースはクライアントベースと比較して，CPU 使用率が極めて高いことが分かる．これは，サーバベースではメタデータの更新が集中して発生したことで無効化通知メッセージが増大したこと，及びその直後の stat 操作において，キャッシュミスが集中して発生したためであると考えられる．クライアントベースでは，stat 操作を行う各プロセスから再検証要求が発生することになるが，再検証の処理が軽いために，それほど負荷が高くならなかったと考えられる．一方，提案手法では，21 秒辺りまでは使用率が 100 %を超えているが，それ以降では使用率が低下し，クライアントベースとほぼ同等となっていることが分かる．これは，更新タイミングが重なったファイルを検出し，キャッシュポリシをサーバベースからクライアントベースへと動的に変更することで，無効化通知メッセージの影響を軽減したためであると考えられる．

続いて，この負荷がクライアントに与える影響を評価するために，上記の負荷状況下において，別のクライアントノード上から mdtest を実行し，サーバベース，クライアントベース，提案手法それぞれの場合でスループットを測定した．またこの時，一貫性制御メッセージによる影響を明確にするために，負荷プログラムとは別に，メタデータサーバ上で無限ループを実行している．また，mdtest では，再検証要求による影響を測定した場合と同様の動作を行っている．

実験結果を**図 13** に示す．評価はサーバベースにおけるスループットを 1 として正規化している．まず，サーバベースとクライアントベースに注目する．負荷を与える前の実験結果（図 11）と比較すると，create 操作においてサーバベースの方がクライアントベースよりもスループットが低下していることが分かる．create 操作はメタデータキャッシュを利用しないため，サーバベース，クライアントベースで同じ操作を行うことになる．従って，これは，メタデータの更新が集中して発生したことにより，サーバベースでは無効化通知メッセージが増大し，メタデータサーバに負荷がかかることで性能が低下したためであると考えられる．remove 操作においても，同様にサーバベースでは負荷が高くなったために，負荷を与える前に比べてサーバベースとクライアントベースで性能差がほとんどなくなった状態となっていると考えられる．一方，stat 操作では，負荷を与える前と比較してそれほど変化がないことが分かる．これは，サーバベースの場合，キャッシュヒットした際にメタデータサーバへアクセスが発生せず，メタデータサーバの負荷状態による影響がないためだと考えられる．ただし，先ほど述べたように，File-stat ではキャッシュミスが発生するため，サーバベースでは負荷による影響を受け，性能差が小さくなっていることが分かる．次に，提案手法に注目すると，サーバベース，クライアントベースと比較して，すべての操作において，同等か，それ以上の性能を示しているが確認できる．create 操作では，クライアントベースと同等の性能を示していることが分かる．これは，無効化通知メッセージの増大による影響を回避することで，メタデータサーバの負荷を軽減し，性能低下を防いだためであると考えられる．また，stat 操作，remove 操作についても同様のことが言えると考えられる．

以上から，提案手法では，アクセス状況に応じてキャッシュポリシを動的に変更していることが確認でき，無効化通知メッセージの増大による影響を回避することで，メタデータサーバの負荷を軽減できていることが分かる．

### 5.4 キャッシュを保持するクライアント数による影響

次に，先ほどの負荷状況下において，stat 操作を行うプロセス数を 50，100，150，200，250 と変化させた場合のスループットを，mdtest を用いて評価した．実験はクライアントベース，サーバベース，提案手法それぞれで行っている．

実験結果を**図 14**，**図 15**，**図 16** に示す．グラフは，各操作において stat 操作を行うプロセス数が 50 の場合のスループットを 1 として正規化している．クライアントベー

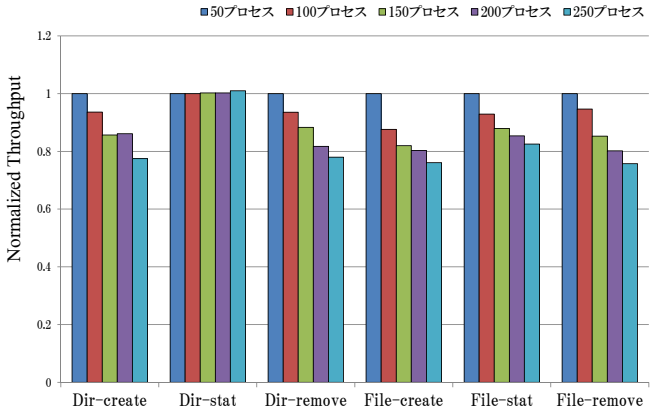


**図 15** サーバベース：stat 操作を行うプロセス数による影響

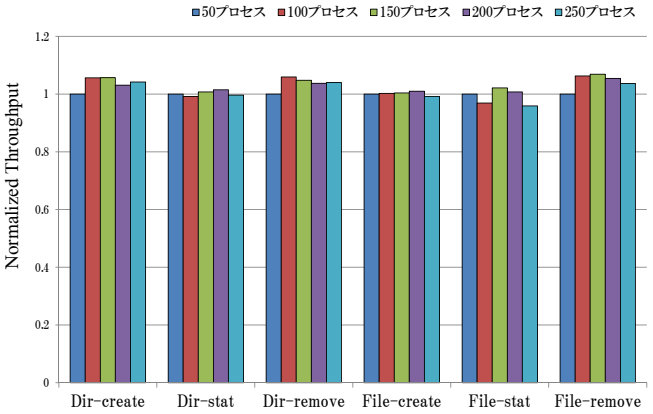


**図 16** 提案手法：stat 操作を行うプロセス数による影響

スの場合では，各操作において stat 操作を行うプロセス数が増えても，ほとんど性能に変化がないことが分かる．これは，メタデータキャッシュを保持するプロセス数が増加しても，再検証による処理が軽いためにメタデータサーバにあまり負荷がかからなかったためであると考えられる．サーバベースでは，Dir-stat を除いた各操作において，stat 操作を行うプロセス数の増加につれて性能が低下していることが分かる．これは，メタデータキャッシュを保持するプロセス数が増加することで，無効化通知メッセージによる影響がより大きくなったためだと考えられる．Dir-stat に関しては，先に述べたようにキャッシュにヒットし，メタデータサーバへアクセスが発生しないために負荷の影響を受けなかったためだと考えられる．一方，提案手法では，クライアントベースと同様に，stat 操作を行うプロセス数による変化はほとんどないことが分かる．これは，キャッシュポリシを変更することで無効化通知メッセージの増大による負荷を軽減したためであると考えられる．

以上から，メタデータをキャッシュしているプロセス数が多いほど，無効化通知メッセージの影響が大きいことが分かる．

## 6. おわりに

本稿では，分散ファイルシステムにおけるメタデータサーバの負荷を軽減する手法の一つである，クライアント側でのメタデータキャッシュに注目し，そのキャシュポリシであるサーバベースとクライアントベースをファイル単位で設定し，アクセス状況に応じて動的に変更する手法を提案した．メタデータの更新状況を観測し，更新タイミングが重なったファイルをサーバベースへ変更し，それ以外のファイルをクライアントベースへ変更することで，サーバベースにおける無効化通知メッセージの増大を回避しつつ，クライアントベースにおける再検証要求によるオーバヘッドを軽減した．mdtest ベンチマークを用いた実験の結果，前述した各キャッシュポリシの問題による性能低下を抑制し，メタデータサーバの負荷を軽減できていることを確認した．

今後の課題としては，ファイル単位でキャッシュポリシを設定するだけでなく，ディレクトリ単位でキャシュポリシを設定することが挙げられる．ファイル単位の場合では，ファイルが削除されてしまうとキャシュポリシの変更を行うことができないため，削除によりメタデータの更新が集中した場合には対応できなくなってしまう．しかし，ディレクトリ単位でキャシュポリシを設定することで，ファイル生成時にキャッシュポリシを設定できるため，同様にして一貫性制御による負荷を軽減することができると考えられる．また，ファイルへのアクセス状況を変化させた場合での提案手法の振る舞いについても調査する必要があると考えられる．

## 7. 関連研究

本研究では，分散ファイルシステムにおけるメタデータサーバの負荷を軽減するメタデータキャッシュの一貫性制御手法を提案した．

メタデータサーバの負荷を軽減する研究としてはこれまでに，クライアントにメタデータの管理を委譲する手法 [3] が提案されている．この手法は，メタデータサーバが，あるファイルのメタデータの管理を特定のクライアントに委譲することで，メタデータサーバの負荷を軽減するというものである．しかし，この手法はメタデータ処理自体を軽減するものではないため，あるメタデータにアクセスが集中した場合，クライアントに負荷がかかり，システムの性能が低下することが考えられる．また，メタデータキャッシュをクライアント間で取得可能とする手法 [4] も提案されている．この手法では，あるクライアント上に存在するメタデータキャッシュを，別のクライアントがメタデータサーバを介さず，直接取得できるようにするというものである．メタデータキャッシュを保持する任意のクライアン

トからメタデータを取得できるため，メタデータサーバへのアクセス回数が減少し，負荷を軽減することができる．しかし，この手法で行われている一貫性制御はクライアントベースであり，ファイルへのアクセス発生時に，クライアントはメタデータの更新を逐一確認する．このため，メタデータキャッシュの利用時においてオーバヘッドが発生することが考えられる．また，メタデータサーバを複数台で構成し，メタデータ操作を分散させて負荷を軽減する研究 [5] も行われている．この場合には，メタデータサーバ間でメタデータの一貫性を保証する必要があり，一般的には 2 層コミットが用いられている．この他にもメタデータ操作に関する様々な研究 [6], [7] が行われている．

本研究では，クライアント側でのメタデータキャッシュに注目し，サーバベースとクライアントベースをファイル別に設定することで，アクセス状況に応じた適切な一貫性制御をファイル単位で行う手法を提案した．

**謝辞** 本研究の一部は，科研費基盤研究 (C)24500113 による．

## 参考文献

[1] 建部修身，曽田哲之, 広域分散ファイルシステム Gfarm v2 の設計と実装, 情報処理学会研究報告：ハイパフォーマンスコンピューティング, 2004-HPC-099, pp. 145–150 (2004).
[2] 建部修身，曽田哲之, 広域分散ファイルシステム Gfarm v2 の実装と評価, 情報処理学会研究報告：ハイパフォーマンスコンピューティング, 2007-HPC-113, pp. 7–12 (2007).
[3] Vilobh Meshram,Xiangyong Ouyang and Dhabaleswar K. Panda: Minimizing Lookup RPCs in Lustre File System using Metadata Delegation at Client Side, The Ohio State University on Department of Computer Science and Engineering, (2011).
[4] Ermolinskly A. and Tewarl R.: C2Cfs: A Collective Caching Architecture for Distributed File Access, IEEE International Conference on High Performance Computing and Communications, pp. 642–647 (2009).
[5] Jin Xiong,Yiming Hu,Guojie Li,Rongfeng Tang and Zhihua Fan: Metadata Distribution and Consistency Techniques for Large-Scale Cluster File Systems, IEEE Transactions on Parallell and Distributed Systems, Vol. 22 pp. 803–816 (2011).
[6] Qian Zhang,Dan Feng and Fan Wang: Metadata Performance Optimization in Distributed File System, IEEE/ACIS International Conference on Computer and Information Science, pp. 476-481 (2012).
[7] Xiuqiao Li,Bin Dong,Limin Xiao,Li Ruan and Dongmei Liu: CEFLS: A cost-effective file lookup service in a distributed metadata file system, IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 25–32 (2012).
[8] mdtest, http://sourceforge.net/project/mdtest/.