システムソフトウェアと
オペレーティング・システム 65−13
（1994．7．21）

分散ファイルシステムにおける一貫性制御プロトコルのユーザレベルカスタマイズ機構

上原 敬太郎　猪原 茂和　宮澤 元　益田 隆司

東京大学 大学院 理学系研究科 情報科学専攻
〒113 東京都 文京区 本郷 7-3-1

要旨

Lucasオペレーティングシステムは64ビットの仮想空間上にメモリマップを用いることで分散協調作業のための効率の良いプラットフォームを提供する。本稿ではLucasファイルシステム上の分散キャッシュの一貫性プロトコルを簡潔に記述するためのプロトコル記述システムについて述べる。分散キャッシュの一貫性プロトコルは様々な種類があり、アプリケーションの性質を利用した最適化が可能であることから、Lucasではユーザに簡潔なモデルと記述言語を提供することでこれに対応する。実際にこのシステムを用いていくつかのプロトコルの記述を行い、実行速度やメモリ使用量の点から考察を試みる。

User-Level Customization of Cache Coherence Protocols in Lucas Distributed File System

Keitaro Uehara, Shigekazu Inohara, Hajime Miyazawa and Takashi Masuda

Department of Information Science, Graduate School of Science,
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113

Abstract

The Lucas Distributed File System (LucasFS), a platform for developing cooperative applications, provides an effective way of sharing complicated data structures through memory-mapped files. Consistency of memory-mapped files in different kinds of cooperative applications should be managed through different coherence protocols for efficient sharing of distributed data. This paper describes the Protocol Customize System (PCS) in LucasFS. Using PCS, users can define with brief descriptions cache coherence protocols that are adapted for particular applications. This paper examines the ability of PCS to describe different kinds of protocols, and investigate its run-time performance and memory usages.

1 はじめに

我々が研究開発を行っているLucasオペレーティングシステムは、64ビット仮想空間を用いることで複雑なデータ構造を分散サイト上の複数のプロセスが効率良く共有して使用することを大きな目標としている。Lucas上のファイルシステム(以下LucasFS)ではmemory-mapped fileを永続的にアドレス空間上の特定の位置に置くことで、複数ファイル間にまたがる参照関係をハードウェアが直接解釈可能なポインタとして表現可能になる[4]。

LucasFSでは複数のホスト上に分散するmemory-mapped fileのキャッシュの一貫性を、分散共有メモリの技術を用いて保持している。Lucasの目的とする応用である協調作業(グループCADやグループSDEなど)においては、キャッシュのアクセスパターンはアプリケーションの性質によって最適なものが存在すると考えられる。そこでLucasFSでは単一の一貫性プロトコルではなく、ユーザがアプリケーションの種類に応じて自由にキャッシュ一貫性プロトコルを記述することができる機構を、システムに組み込むことによってこの問題を解決する。本論文ではLucasFSにおける分散共有メモリプロトコル記述の機構であるプロトコルカスタマイズシステムについて述べる。実際にこのシステムを利用していくつかの代表的な一貫性プロトコルを記述し、その有用性を検討する。

2 背景

この章では後の章を理解するのに最低限必要なLucasFSのアーキテクチャと複数の一貫性プロトコルの必要性について述べる。

2.1 Lucasファイルシステムの概要

LucasFSは現在DECstation 5000上のMachマイクロカーネルの上にExternal Pagerのインタフェースを用いて実装されている。LucasFSでは、各サイト上のディスクを管理しているストレージサーバ(SS)と各サイト上でキャッシュを管理するキャッシュサーバ(CS)を置き、これらの間で分散共有メモリのプロトコルを用いてキャッシュの一貫性を保証している。各サイト上のクライアントプロセスは、ファイルを仮想空間にマップすることでそのサイト上のCSと通信をする。CSはクライアントのページフォールト時に、必要なページのキャッシュがサイト上にあればそれをクライアントのアドレス空間に供給し、なければそのファイルが存在するサイト上のSSと通信し、キャッシュを要求する(図1)。CSとSSを分離することでユーザプロセスとキャッシュとのローカルな関係とそのネットワーク上でのキャッシュのグローバルな状態(分散共有メモリでいうディレクトリ)を分けて管理することができる。ローカルサイト上のCSとSSの間の通信はメッセージを用いずに関数呼び出しとして実現されている。これによりローカルサイト内の通信のメッセージ数を減らしている。

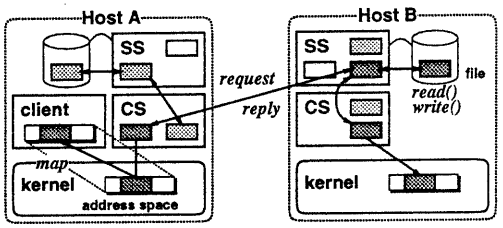

図1: LucasFSの構成

2.2 複数の一貫性プロトコルの必要性

従来の多くの分散ファイルシステムでは単一のプロトコルで分散キャッシュの管理がされていて、ファイルシステムとしての通常の使用に対しては十分な性能を引き出している。一方協調作業のための分散共有メモリプロトコルにはさまざまな種類のものがあり、これらはアプリケーションの性質やデータのアクセスパターンによって選ばれるべきものである。このような考察のもと、複数の一貫性プロトコルを予め用意し、ユーザが適切なプロトコルを選択できるようにしたMuninのような分散共有メモリシステムも研究されている[2]。協調作業ではアプリケーション毎の性質を活かして数多くの最適化ができると思われるので、LucasFSでは基本的なプロトコルを用意した上で、アプリケーションの性質に従ってプロトコルを柔軟に選択するために、簡潔な記述でプロトコルをカスタマイズできる機構を提供することで、この問題を解決することにした。

3 プロトコルカスタマイズシステム

実際にC言語でいくつかのプロトコルを記述してみた結果、いくつかの点でプロトコルを記述するのに注意すべき点、難しい点が存在することがわかった。LucasFSでは以下に述べる問題点を解決し、ユーザが簡単な記述によってプロトコルを記述できるようにするプロトコルカスタマイズシステム(以下PCS)を導入し、

ユーザレベルのプロトコルの記述を簡潔に行えるようにする。

3.1 設計の方針

プロトコルを記述する際に注意すべき点として主に次の2つの点が挙げられる。

第一はプロトコルの状態数を低く抑えることである。もともと多くのプロトコルには数種の定常状態と状態遷移のための一時的な状態があるが、Lucasではユーザレベルで仮想空間の管理を行うためにCSとカーネルとの間にデータのやりとりがあり、そのため実際に取り得る状態数が本質的な状態数以上に多くなってしまう。例えばカーネルからの要求はないが、プリフェッチプロトコルなどでCSにデータを供給した場合、同じread/write可能な状態でもカーネルが持っているかCSが持っているかの2通りの状態が存在することになる。従って同じメッセージに対する処理内容も内部では異なってくる(図2参照)。このため同じ処理の記述がソースコード上のあちこちに分散してしまい、プログラムの見通しが悪くなる。

第二は分散環境におけるメッセージの扱いである。分散環境においてはメッセージの到着する順が必ずしも期待通りになるとは限らない。また、メッセージ送信のタイミングによっては処理の内容が異なることもある。しかしメッセージの内容に従って関数が呼び出されるというRPC形式でこのような処理を記述するためには、ユーザがメッセージのキューを用意し、今受けとるべきでないメッセージに対応する関数が呼ばれた時にはキューに入れ、後で同じ処理を再現する必要がある。また複数のメッセージを受けとる場合には処理の流れが一元的でないために理解しづらいコードとなってしまう。

これらの問題点を解決するために、LucasFSでは簡単なインタプリタ型言語によって状態の変化やメッセージの送受信、プロテクションの変化を記述可能にする。C言語への組み込みが容易であるという理由でTclインタプリタ[5]をベースに用いることにした。

第一の点の解決法として、プロトコルを記述する上では、ユーザはカーネルやCSの細かい状態や通信を知る必要はないことに着目した。必要最低限のカーネルからの要求だけをイベントとしてユーザに見せ、ユーザはプロテクションの状態だけを記述することで適切な処理が行われるようにする。例えばユーザがあるページの属性をread/writeと指定したとする。もしカーネルがページを要求していればページの供給が行われるし、要求していなければCSがページを保持したまま"read/write"

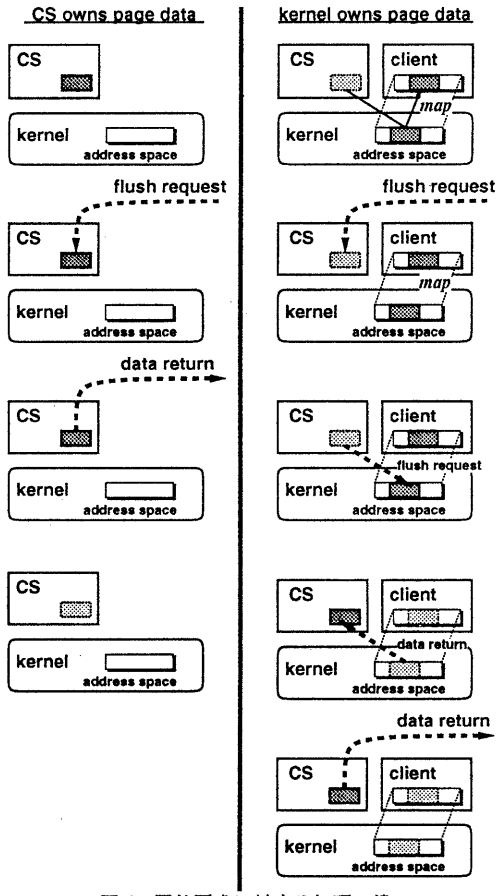

図 2: 同じ要求に対する処理の違い

左側はCSがデータを保持している状態。右側はカーネルがデータを保持している状態。データを破棄(フラッシュ)する要求が来た場合。左図ではすぐにデータを返すことができる。一方右図ではまずカーネルにフラッシュ要求を出し、そのデータが戻ってきてからデータを返さなくてはならない。

という状態だけが記憶される。将来的にカーネルから要求が起こった場合、ページの供給は即時に行われ、ユーザからこのカーネルの要求が見えることはない。逆にカーネルがページを返してきたとしても、それがユーザの要求によって起こったものでないならば、やはりユーザからは見えずにCSがページを保持する形となる。

第二の点の解決法として、イベントに対応する関数を用意するRPC形式でなく、プロトコルを処理の流れに従った形で記述できるようにした。このために各種のイベント(メッセージを含む)を受け取り先(ページやファイル)ごとにキューに入れ、その受け取り先が待ってい

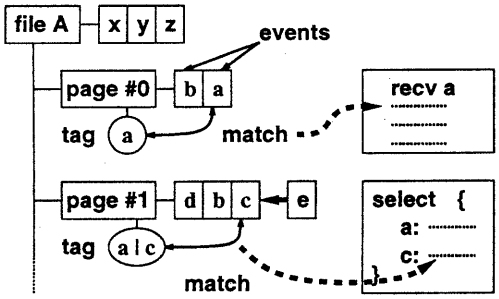

図 3: 行き先別のイベント処理

るイベントにマッチするものを選択し、実行を続けるという形を取る (図3参照)。

あるメッセージ A、B を受けとる場合には recv A ; recv B と記述することで、メッセージの到着順が入れ替わっても適切に処理が行われる。また到着するイベントが A か B かによって処理が変わる場合には select { A: イベント A の処理 ; B: イベント B の処理 } と記述することで 複数種類のイベントを待ち選択的に処理が行える。

この機能を単純に Tcl の関数として実装するとメモリや資源の使用量の点で問題が生ずるのだが詳細は 4 章で述べる。

3.2 プロトコル記述モデル

PCS でユーザが扱う概念にはページ、データ、ポート、メッセージがある。PCS ではこれらを取り扱うためのプリミティブを Tcl に付加している。

データ　データは実際に読み書きされる 1 ページ分のメモリバッファである。CS 上ではカーネルに、SS 上ではディスクに読み書きすることができる。Tcl 上ではデータ ID という値として扱われる。データはメッセージに詰めて送ることもできる。その際にデータを複製する (元に残す) か移動する (元は消す) かを選択することができる (dealloc フラグ)。実際にはローカルホスト内では CS と SS が同じデータを共有していることもあるが、ユーザはそれを意識する必要はない。

ポート　ポートはメッセージを送る先のホストを一意に指定するもので、Tcl 上ではポート ID という値で表される。各ページ (ファイル) は必ず 1 つの SS に属しているが、これは SS_port というコマンドで取得できる。ポートはメッセージに詰めて送ることもできる。送り元のポートはメッセージの reply_to を参照することで得られる。

タグ名	値	内容
event	イベント名	DATA_REQUEST など
reply_to	送信元ポート	送り元ポートが入る
send_to	送信先ポート	送り先ポートを指定する
dataID	データ	データを送る際に指定する
dealloc	破壊フラグ	データの参照を取り消す
prot	保護属性	保護属性を指定する
pageID	ページ	送り先ページを指定する

表 1: メッセージ変数の固定領域

ページ　ページは Tcl 上ではページ ID という値で表される。ページは必ず 1 つのファイルに属し、中にデータと保護属性を持つ。データの指定は set_data で行い、取得は get_data で行う。保護属性は N (none), R (read-only), RW (read/write) の 3 種類があり、set_prot で指定し、get_prot で取得する。これはカーネルの状態に関わらず、CS の外から見える属性で指定する。従って現在の状態が RW であって set_prot N を実行すると、カーネルがデータを持っているならばフラッシュが行われ、データが戻り、カーネルの保護属性の変更が行われるし、カーネルがデータを持っていないならば実際には何も行われないことになる。ユーザはこれらの違いを、プロトコル記述上は全く意識する必要がない。

メッセージ　ユーザはメッセージを用いてデータを他のサイトや他のページに送ることができる。通常、メッセージ通信は CS と SS との間で行われるが、CS が他の CS に直接メッセージを送ることも可能である。メッセージのやり取りにはメッセージ変数を使用する。メッセージ変数は create_msg によって作られ、表 3.2にあるような固定領域と、ユーザが任意に指定することのできる可変領域からなる。可変領域では任意の文字列を送ることができ、プロトコル間で使用することができる。メッセージの送信には send が使用される。send は送信先のポートとメッセージ変数を指定することで送られ、送信先でイベントが作られることになる。

3.3 具体的な記述例

具体的な記述例を説明するために、Tcl について簡単に説明を加える。Tcl は csh スクリプトのような構造を持っている。{ } はリストを指定する。変数は set によって作成・更新され、変数の値は $ によって参照される。変数は単純な変数の他に msg(event) のような連

```
CS_PAGE: select msg {
    CLIENT_DATA_REQUEST: {
        create_msg rmsg DATA_REQUEST
        set rmsg(prot) $msg(prot)
        set rmsg(need_data) $msg(need_data)
        send [SS_port $pageID] rmsg
        recv DATA_REPLY dmsg
        mrecv $dmsg(inv_cnt) INV_COMPLETED
        if {$dmsg(should_return) == 1} {
            create_msg ack DATA_RECEIVED
            send [SS_port $pageID] ack
        }
        if {$rmsg(need_data) == 1} {
            set_data $pageID $dmsg(dataID)
        }
        set_prot $pageID $msg(prot)
    }
    ......
}
```

図 4: プロトコル記述の例

想配列がある。値は基本的には全て文字列で、それを整数として計算したり比較したりすることも可能である。[ ] は置換で、csh のバッククォートと同じ働きをする。

図 4は write-invalidation プロトコルの一部である。最初の CS_PAGE は CS 上のページレベルのプロトコルであることを示している[1]。プロトコルは通常 1 つの大きな select 文から構成される。イベントが到着するとその内容が select 文の引数であるメッセージ変数 (ここでは msg) に代入され、マッチしたイベントへと処理が移る。カーネルからのページ要求は CLIENT_DATA_REQUEST というイベントとして扱われる。次の行では create_msg によって rmsg という名前の変数にイベント名 DATA_REQUEST からなるメッセージ変数が作られる。さらに rmsg に msg の内容の一部をコピーしている。$msg(need_data) はカーネルがデータを要求しているかどうかを示すフラグである。次の send 文でこのページの SS に対して rmsg が送られる。$pageID は常にこのページのページ ID が入っている。recv 文でイベント DATA_REPLY を待ち、そのメッセージの内容を変数 dmsg に受けている。次の mrecv 文は引数 $dmsg(inv_cnt) で指定された個数分のイベント (INV_COMPLETED) を受け取るまで待つ。返事を返さねばならない時 ($dmsg(should_return) == 1) には DATA_RECEIVED というメッセージを作って返す。データが欲しい時には set_data で送られてきたデータをセットし ($dmsg(dataID))、最後に set_prot によって保護属性を換えて終了である。

[1]他にファイルレベルのプロトコルも記述できる。

4 PCS の実装

LucasFS はメモリマップに基づく分散ファイルシステムであり、分散キャッシュとして仮想メモリを使用している。従ってキャッシュとして使用できるメモリの量がファイルシステムとしての性能に直接影響する。このためメモリを無駄に使用してしまうことは避けなければならない。

しかし記述したプロトコルをそのまま Tcl に解釈させてしまうと各ページ毎にスレッド及びそれに対応するスタックが必要となる点が問題となる。というのは select 文や recv 文ではメッセージを待つために処理を停止させる必要があるためである。Tcl ではユーザが定義する命令は対応する C の関数の呼び出しとなる。select や recv に対応する命令を C の関数として用意した場合、その関数内でメッセージを待たなくてはならない。関数をリターンさせてしまうとインタプリタはその命令の実行が終わったとみなし、次の行の実行に処理が移ってしまう。Tcl には計算を途中で保留できる (continuation を保存する) 機構がないため、あるページで待つ処理と他のページの処理を同時に行うためにマルチスレッドにする必要がある。

一方プロトコルの進行を考えると各ページやファイルは基本的には select によるループによって記述されている。従って最初に要求によって起動されてからは常に select や recv によるイベント待ちの状態にいることになり、各ページ毎にスレッドが必要となってしまう[2]各スレッドはスタックフレームを持つため、スタックフレームの大きさを 1 ページくらいとすると、処理するファイルの容量と同量のメモリを必要とすることになる。これはファイルシステムとしての性質上とても容認できない量である。

このことから select や recv を他の命令 (send や set_prot) と同様に Tcl の関数として実装することは現実的でないということになる。すなわち select と recv をインタプリタで解釈させてはならない。そこで予めプロトコルの記述を select と recv を含まない形に変形する必要がある。具体的には select, recv の直前までインタプリタで実行し、そこでインタプリタから戻ってくる。select, recv の実行は親スレッドが行い、イベントを受け取ったならば select の場合は対応するイベントの文を実行する。そしてその実行が終了した後、select, recv の直後の文へと実行を移す。実際には制御構造がネストしているため、if, while, for,

[2]SS と CS ではコンテクストが別なので、実際にはより多くなる。

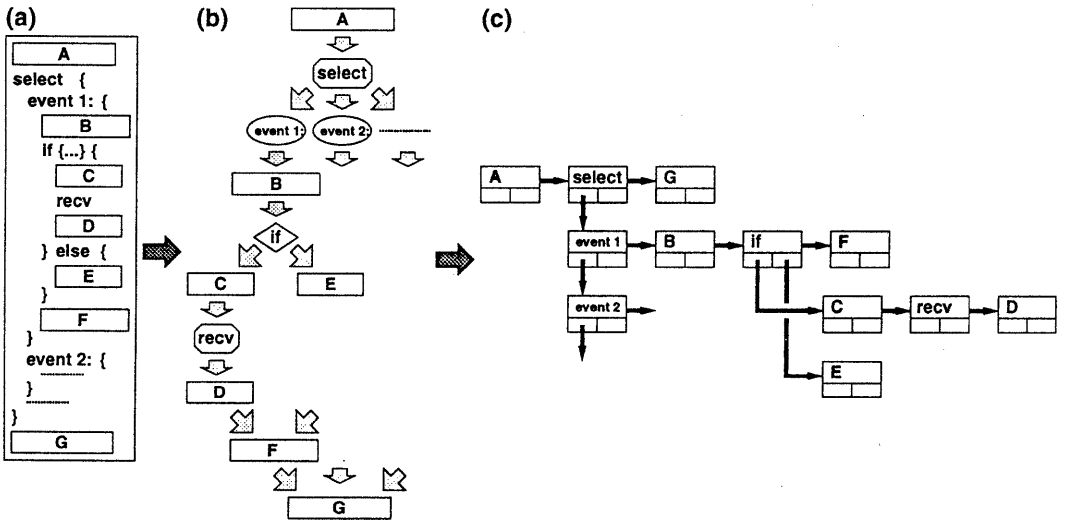

図 5: select, recv を含む文の変形
(a) PCS ソース (b) select と recv の切り出し (c) 構文木

foreach といった制御構造を決定する文も含めて変形を行う必要がある。PCS の内部には Tcl のソースを、select, recv の部分だけ Tcl から制御を戻し、それ以外の部分は Tcl に解釈させるような形に変形するパーザを持っている[3]。パーザは制御文をパーズし、構文木を作り上げていく (図 5参照)。制御文以外の部分 (図 5(C) のブロック **A**, **B** …) は字面のまま記憶しておき、実行時にそのまま Tcl インタプリタに解釈させる[4]。

5 考察

PCS を使って、性質の異なるいくつかのプロトコルを記述してみた。ここでは (1) write invalidation、(2) memory mapped stream、(3) release consistency、(4) causal consistency の 4 つのプロトコル [7] を記述してみる。

5.1 一貫性プロトコルの説明

write invalidation は厳密な一貫性を保証する分散システムで最も一般的に使われているプロトコルである。このプロトコルでは複数の writer や writer と reader が同時に参照を行うような場合には効率が落ちる。

[3]ML のコンパイルの中間形式で用いられる continuation passing style への変換 [1] の簡単な形であると見ることもできる。

[4]スレッドのスタック量を減らす研究としては Mach の continuation [3] があるが、この提案ではプログラマが明示的に continuation を意識したプログラミングを強いられるため処理の流れとプログラムの構造がかけ離れる傾向がある。

memory mapped stream は書き手と読み手が分かれているプロトコルである。書き手 (writer) が書いたページを send というプロトコルによって次々と読み手 (reader) に送ることで、要求されてから送るプロトコルに比べてアクセスの遅延時間 (latency) を小さくすることができる。

release consistency は通常のメモリアクセスの他に acquire と release という同期命令を用い、メモリアクセス単位での一貫性を取る代わりに同期命令単位での一貫性を保証することで通信の回数を減らすプロトコルである。

causal consistency はアクセス間の causality を保証するプロトコルである。同期命令などの拡張無しで read と write の間の causality が保証されるので、アプリケーションの種類によっては非常に有用なプロトコルであると思われる。

5.2 記述力の向上

上に挙げた 4 種類のプロトコルを PCS で記述し、行数を比較してみた (表 2)。write invalidation の場合、約 1/10 の量でプロトコルが記述できる。もちろん PCS を解釈するための C のソースの分があるため、単純な比較は意味がないが、どれくらい記述が簡潔になったかを示す指標にはなるであろう。write invalidation 以外のプロトコルは C では記述していないため比較はできないが、write invalidation と比べても大分短く記述できる。causal consistency は write invalidation より長い

protocol	lines
write invalidation in C	2286
write invalidation in PCS	195
memory mapped stream in PCS	79
release consistency in PCS	111
causal consistency in PCS	236

表 2: プロトコル記述の行数の比較

が、実はSSの記述は全くwrite invalidationと同じであるため、それを流用することで非常に短時間で記述できた。これらのプロトコルはだいたい1日で記述し、試すことができた。簡単なプロトコルならば数時間もあれば記述可能であると思われる。

またこれらのプロトコルのうち(2)と(3)はメモリアクセス以外に特別な同期命令を必要とする。ユーザプログラムの中に埋め込まれたこれらの同期命令は、プロトコル中ではイベントとして他のメッセージと同じように扱うことができる。RPCのようにエントリとなるべき関数を新たに書いたり、システムコールを追加することなく他のメッセージと同じように扱えるので、非常に簡潔な記述が可能となる。

5.3 実行速度

今回の実装は実際に複数のプロトコルを記述し、それらの有効性を検証するのが第一の目的であるが、インタプリタを組み込んだことで組み込まないものに比べどの程度のオーバヘッドがあるのかを調べることは、今後改良していく上でも非常に興味あるところである。

今回の実装をした環境はDECstation 5000 (33MHz)のMachカーネル上である。リモート通信を伴う実験はもう1台のDECstation 5000 (25MHz)との通信によって測っているため、必ずしも行きと帰りのメッセージのコストは同じではない。参考のために表3に実験を行ったマシン上での基本的な実行のコストを載せた。`mach_task_self()`はタスク自身のポートを得るシステムコールで、カーネルに入って戻ってくるまでのおおよそのコストを表していると思われる。threadのコンテクストスイッチは、`condition_wait()`と`condition_signal()`を明示的に用いてスイッチした場合のコストである。page faultのコストはマップしたページに対してすぐにページを供給した場合に、アクセスが再開されるまでの時間である。null RPCのうちsimple routineとは非同期のメッセージ通信によるもの、routineとは同期的メッセージ通信によるものである。

PCSを組み込む前のLucasFSサーバ(write invalidationのみ) [6]とPCSを組み込んだLucasFSサーバで最初のページアクセスにかかるコストを比較してみた(表4(a)(b)参照)。

action	cost (msec)
`mach_task_self()`	0.031
thread context switch	0.018
page fault (on default pager)	3.9
page fault (on external pager)	1.6
null RPC (simple routine)	7.42
null RPC (routine)	19.0
simple routine with n pages	$16.6 + 9.43 \times n$

表 3: 実験環境における諸基本操作のコスト

この表より(b)は(a)に比べ、ローカルアクセスの場合で約7倍、リモートアクセスの場合で約2倍のコストがかかっていることが分かる。リモートの場合、インタプリタのコスト自体はあまり変わらず通信のコストが大きくなるため、相対的に差は小さくなる。リモートの場合通信のコストに比べてインタプリタの実行速度は許容できる範囲であるとした仮定はそれほど外れていなかった。しかし思ったよりコストが大きかったというのもまた事実である。

インタプリタに関連するコストを調べてみた所、実はインタプリタ構造体の構築に大きなコストがかかることが判明した。Tclインタプリタ構造体(`Tcl_Interp`)を作る`Tcl_CreateInterp`の呼び出しには約5.28msecのコストがかかる(表5参照)。インタプリタ構造体はTclのインタプリタを呼び出す際に使われ、変数やスタックのフレームなどを保存している。PCSでは各ページ、各ファイル毎にCSとSSとを独立してプロトコルが記述可能であり、これら毎に別のインタプリタを用いるため、これだけでかなり大きなコストとなる。

このことを踏まえ、インタプリタ構造体を要求に応じて作るという最適化を行った所、表4の(c)にあるように(a)に比べてローカルで約5倍、リモートで約1.5倍

	local	remote
(a) PCS 組み込み前 (msec)	5.93	49.2
(b) PCS 組み込み後 (msec)	40.0	96.8
オーバヘッド (b) − (a) (msec)	34.1	47.6
速度比 (b) / (a)	6.75	1.97
(c) PCS 組み込み + 最適化 (msec)	32.3	77.1
オーバヘッド (c) − (a) (msec)	26.4	27.9
速度比 (c) / (a)	5.45	1.57

表 4: 1ページ当たりのページフォールトのコスト

action	cost (msec)
Tcl_CreateInterp	5.28
Tcl_Eval "set a 1"	0.052
Tcl_Eval "create_msg"	0.19
Tcl_Eval "send"	0.46

表 5: Tcl の実行に関連するコスト

程度のコストになり、かなりの性能の向上が見られた。

性能低下のもう1つの要因は消費するメモリ量であると考えられる。実際インタプリタ構造体自体はかなりのサイズを占めているようである。メモリの割り当てや解放のコストの他にも大量にインタプリタ構造体を作ることでメモリを大量に消費し、ページングを起こして性能低下を生み出すという原因にもなっている。ページ毎にスレッドの context を保持するのを防ぐために continuation を使って変形するという手段を取ったわけだが、それでもなおインタプリタの context を保持するために大量のメモリを消費している。現在の実装ではインタプリタ構造体は各ページにおけるローカル変数の値を保持するという目的のためだけに割り当てられている。従ってインタプリタ全体を保持することを止め、変数のフレームだけを各ページが管理し、それを差し替えながら解釈・実行していく、という形を取ることによって各ページ毎にインタプリタを保持する必要がなくなる。これによってメモリとスピードの両面でかなりの改善が見られると考えられる。この改良を行うことでオーバヘッドのうちさらに約 10msec は縮められると予想している。

一方インタプリタ自体の実行速度は Tcl の1行を解釈・実行するのに、50μsec から 500μsec かかる。ページフォールトの際に実行される PCS のコードは約 30 行で、そのオーバヘッドはローカルの場合で約 4msec 程度と推測され、このコストはインタプリタをやめ、中間コード形式にコンパイルすることでより小さくできると考えられる。

6 まとめと今後の課題

本論文では Lucas ファイルシステムの概要とキャッシュ一貫性プロトコルのカスタマイズ機構について述べた。この機構によってアプリケーションの性質に合わせて適切なプロトコルを組み込むことが可能になると考えられる。

将来的にいくつか課題がある。その1つは本文中でも述べたが、インタプリタの性能の向上である。Tcl の内部構造を最適化して性能向上を行う目処は立っている。さらにプロトコル記述の行数を減らすために、現在は複数行で行っている処理をより少ない行で実行できるようにするつもりである。

もう1つは複数のページをまとめて扱うための機能である。現在の仕様では1つのメッセージには単一のページあるいはファイルの情報のみしか入れることができない。効率的な先読みプロトコルなどを実装するためにはリモートメッセージの数を少なくできるこの機能が不可欠であると考える。

最後に同期機構との融合である。現在も release consistency などのメモリアクセス以外の同期命令を必要とするプロトコルのためにユーザプログラムからイベントへの窓口を設けてはあるが、あくまで実験的なものである。これをさらに推し進め、同期機構を分散共有メモリの管理と融合することでより効率的な環境を提供することができるのではないかと考えている。排他制御やトランザクションのような処理も PCS で記述することをめざす。

参考文献

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 152–164, Oct 1991.

[3] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 122–136, Oct 1991.

[4] 猪原茂和, 上原敬太郎, 宮澤元, 益田隆司. オペレーティングシステム Lucas における 64 ビットアドレス空間の管理. In *6th SWoPP*, Aug 1993.

[5] J. K. Ousterhout. Tcl: An Embeddable Command Language. In *1990 Winter USENIX Conference Proceedings*, pp. 133–146, Jan 1990.

[6] 上原敬太郎, 猪原茂和, 宮澤元, 益田隆司. Lucas オペレーティングシステムにおける分散ファイルシステムと分散共有メモリの融合. 第 47 回 (平成 5 年後期) 情報処理学会全国大会, 第 4 巻, pp. 21–22, Oct 1993.

[7] 上原敬太郎, 宮澤元, 猪原茂和, 益田隆司. 分散協調作業のための一貫性制御プロトコルに基づく分散ファイルシステム. 第 63 回システムソフトウェアとオペレーティングシステム研究会 研究報告, pp. 1–8, Mar 1994.