分散共有メモリのキャッシュ一貫性制御プロトコルのカスタマイズ

上原 敬太郎† 猪原 茂和†† 益田 隆司†

本稿ではプロトコルカスタマイズシステム（PCS）の設計方針と実装について述べる．PCSは分散共有メモリの一貫性プロトコルをユーザの手で容易に拡張可能にするための枠組みである．PCSではプロトコルを記述する際の管理すべき状態数というものに着目し，これを削減するための抽象化技法を提案し，実装を行っている．具体的には（1）メッセージ到着順序の自動的な並べ換え，(2）カーネル-サーバ間の不必要な通信の隠蔽，(3）プロトコルの直線的な記述を可能にするためのメッセージ待ち状態の除去，という3種類の抽象化技術を使って状態数の削減を実現している．この抽象化によってwrite-invalidateプロトコルを記述するための状態数が18状態から3状態へと減った．また実験を行い，数十行の記述を付け加えただけで，約1/3のページフォールトと約10%の遠隔メッセージを削減できることが分かった．PCSを用いることで，アプリケーションの性質を利用したユーザの手による最適化が可能になり，効率的な分散システムの構築が可能になる．

A Customizable Cache Coherence Protocol for Distributed Shared Memory

KEITARO UEHARA,† SHIGEKAZU INOHARA†† and TAKASHI MASUDA†

This paper describes Protocol Customize System (PCS), which enables programmers to design and customize cache coherence protocols of distributed shared memories. PCS provides programmers with the abstracted model to reduce the number of states of protocols to manage. PCS allows programmers to optimize the behavior of protocols according to the properties of applications. As compared with a non-abstracted program, PCS reduces the number of states from 18 to just 3 in the standard write-invalidate protocol. A customized protocol reduces one-third of the number of page faults and about 10% of remote messages.

1. はじめに

分散共有メモリにおいて，分散した複製間の一貫性制御をどのように行うか（一貫性制御プロトコル）を決定することは非常に重要である．一貫性制御プロトコルの性質にはセマンティクスと効率という2つの側面があり，アプリケーションの要求するセマンティクスと実行効率によって適切な一貫性制御プロトコルを選ぶ必要がある．これまで行われてきた分散共有メモリの一貫性制御プロトコルについての研究は主に数値計算アプリケーションを高速化することが目的であった．数値計算アプリケーションにおける共有オブジェクトのアクセス方法はすでにかなりの研究がなされており，アクセスパターンに応じた効率の良い一貫性制御プロトコル[2),6)~8)]や，複数のプロトコルから適切なものを選択することができるシステムも提案されている[3)]．

一方で今後ネットワークが普及していくに従って，分散共有メモリの技術は，ネットワークを介して複数の人間またはプロセスが協調動作を行うアプリケーション（以下，分散協調アプリケーションと呼ぶ）へ応用されていくと考えられる．分散協調アプリケーションは数値計算アプリケーションと違い，まだ共有メモリのアクセスパターンも十分に判明しているとはいえない．一例として並行性制御の単位として使われるトランザクションがあげられる．従来のアトミックトランザクションモデルでは分散協調アプリケーションには対応できないため，さまざまな協調的トランザクションモデルが提案されている[1)]．しかしそれらのモデルのどれにもそれぞれの特徴があり，すべてのアプリケーションに対して効率良く処理できるようなトランザクションモデルはいまだ存在していない．すなわち，多様な分散協調アプリケーションに対応するため

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Graguate School of Science, University of Tokyo
†† 株式会社日立製作所中央研究所プロセッサシステム部
Central Research Laboratory, Hitachi Ltd.

には既存の一貫性制御プロトコルだけで効率的に処理できるとは限らない．そこでプロトコルそのものを記述することができるシステムが必要となる[4),5),10),13)]．このようなシステムではアプリケーションの性質に合わせたきめの細かいプロトコルの最適化が可能となる．

しかし一貫性制御プロトコルをユーザ自身の手で記述することは非常に難しい．その大きな理由として，プロトコルを記述する際に管理すべき状態数が非常に多くなってしまうということがあげられる．状態数の多いプロトコルの記述はメッセージの管理や状態間の複雑な遷移がプログラム上のあちこちに分散してしまうため，一通り正しいプロトコルを記述するのでさえ大変な手間を必要とする．そしてハードウェアによる分散共有メモリの分野で研究されているようなプロトコル記述可能なシステム[5),10)]では，ユーザが管理すべき状態数を軽減するような抽象化はほとんどなされていない．したがってユーザ自身の手によってアプリケーションの性質に合わせてプロトコルの一部をカスタマイズするといった目的に，このようなシステムを使用することは難しい．

これらのことから，より簡潔な記述によってカスタマイズできるような抽象的な枠組みを提供することが，ユーザによるプロトコル最適化のための有効な解決法になると考える．本稿では分散共有メモリのキャッシュ一貫性制御プロトコルを簡単な記述によってユーザがカスタマイズすることができるような枠組み（プロトコルカスタマイズシステム，以下PCS）を提案し，実装したプロトタイプによって有効性を検討する．PCSの基本的な設計方針は，「ユーザが管理すべき状態数を減らす」ということである．PCSでは記述性を向上させつつ汎用性を損なわないために，ユーザにとって不要な状態を抽象化によって隠蔽している．具体的には（1）メッセージ到着順序の自動的な並べ換え，(2) カーネル–サーバ間の不必要な通信の隠蔽，(3) プロトコルの直線的な記述を可能にするためのメッセージ待ち状態の除去，という3種類の抽象化を導入している．この3つの抽象化はそれぞれ，分散メッセージの抽象化，サーバの抽象化，スレッドの抽象化に対応している．これにより，簡潔な記述でプロトコルのカスタマイズが可能になる．PCSのインタフェースにはインタプリタ言語を採用し，プロトタイピングの用途に適したものとなっている．

本稿の構成は以下のようになっている．2章で従来のシステムの問題点と，それを解決するためのPCSの設計方針について述べる．3章でPCSの言語仕様および実装の詳細について述べる．4章では実際のプロトコル記述例と記述量および状態数の比較，カスタマイズを適用した実験結果について検討する．5章で他のシステムとの比較を行い，最後に6章で本稿をまとめる．

2. プロトコルカスタマイズシステム（PCS）の設計方針

この章では，分散共有メモリのプロトコルを記述する上での問題点と，それに対応するためのPCSの設計方針について述べる．

2.1 プロトコルの記述しやすさと管理状態数

ソフトウェアによる分散共有メモリプロトコルの記述システムとしてはMachの外部ページャー[12)]をあげることができる．しかし1章で述べたハードウェアによるプロトコル記述システムと同様に，Machの外部ページャーを用いたプログラミングでは抽象化が十分でないため，ユーザが分散共有メモリプロトコルをカスタマイズするのは非常に困難である．それでは外部ページャーを用いたプログラミングがなぜ難しいのか．ここでは一例としてMachの外部ページャーの例をあげたが，一般的にカスタマイズ可能な分散システムに共通する問題が含まれているのではないだろうか．問題点をより明確にするために，Machの外部ページャーのインタフェースに欠けていると思われる点をあげる．

(1) メッセージの送受信には単純なRPCのインタフェースのみが用意されている．そこには分散言語で見られるような保護条件（guard式）などを記述できないため，メッセージが到着すると必ず対応する処理ルーチンが起動される．処理を後回しにしたければ自分でキューを作って管理する必要がある．

(2) カーネルの機能の一部をページャーとして分離することができたが，すべての機能がページャーで行えるわけではない．そのことによってユーザはカーネルとページャーという2つの存在を考慮しなくてはならず，カーネルも含めた内部状態を管理しなくてはならない．

(3) Machではページャー間あるいはカーネルとページャー間の通信は非同期メッセージによって行う☆．このために「返事を待つ」という動作を直線的に記述するのが難しい．

以下では，記述の難しさをより定量的に表す指標と

☆ 並列性を最大限に高めるためのインタフェースであると推測する．

して「プロトコル記述のために管理すべき状態数」を考えることにする．そのうえで，上の3つの原因についてさらに深く考察し，いかにして管理すべき状態数が増えてしまうかを検討してみる．

2.1.1 分散メッセージの到着順序の非決定性

分散環境においてはメッセージはかならずしも期待どおりの順序で到着するとは限らない．また，たとえ任意の2点間のメッセージのFIFO性が保証されていたとしても，ある2点からのメッセージのどちらが先に到着するかは事前に予測することはできない．そのためにたとえば，あるメッセージAとBがどのような順番で到着するか分からない場合，Aを受けてからBを待つ状態とBを受けてからAを待つ状態の2つの中間遷移状態が必要になる．

より具体的な例としてStanford大学で開発されたハードウェアDSMであるDASH[11]のプロトコルにおけるwrite-requestの例をあげる．ある共有（shared）状態にあるページに対して書き込み要求が起こると，

(1) 書き込みが起こったホスト（writerホスト）がページの所有者であるホスト（ownerホスト）に対して書き込み要求（write request）を出す．
(2) ownerホストはそのページを共有している他のホスト（readerホスト）に対してページの無効化の要求（invalidate request）を出す．
(3) ownerホストはinvalidateの返事を待たずに，writerホストに対して要求されたデータを返す（write reply）．
(4) readerホストは無効化が終了したことをwriterホストへと直接通知する（invalidate ack）☆．

writerホストはownerホストからのデータの返信と，readerホストからのinvalidate ackをすべて受け取ってから書き込みを継続することになる（**図1** 参照）．この場合，writerホストはownerホストからのwrite reply (3) とinvalidate ack (4) のどちらが先に到着するか予測できない．したがって，これを正しく処理できるようにするためには，どちらが先に来た場合にでも遷移できるようにあらかじめ状態遷移を考えておく必要がある．

プロトコルの中にはもちろん到着順序によって処理を変えなくてはならないようなケースも存在するが，DSMのプロトコルではどちらが先に到着しても処理が変わらないようなケースが多い．しかしそのような場合にもユーザはあらゆる組合せを考えて状態遷移を用意しなくてはならないわけで，状態数が増えてしまう一因になっている．

☆ ここでinvalidate ackをownerホストではなくwriterホストへと直接送るところがDASHプロトコルの特徴である．

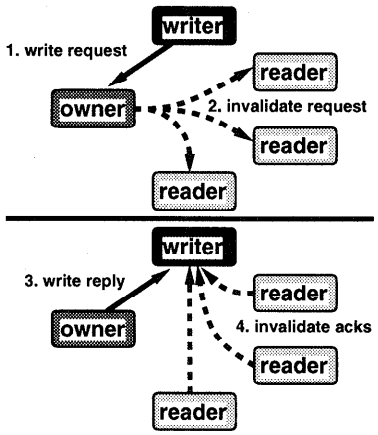


図1 DASHにおけるwrite-requestのプロトコル
Fig. 1 Write request protocol in DASH system.

2.1.2 カーネル–サーバ間の内部状態

Machの外部ページャー（以下，ページャー）ではページングのための処理をページャーとカーネルとに分離しているが，分離したことによってユーザは2つの存在を独立して意識する必要があり，プロトコル上で管理すべき状態数を増加させてしまうこともある．ページャーを用いて分散共有メモリを実現する場合，ページャーとカーネルは非同期メッセージによって通信を行う．たとえばページャーがカーネルからデータを回収する場合には，`memory_object_lock_request`という非同期的なシステムコールを用い，データの返答は`memory_object_data_return`，ロック変更の終了は`memory_object_lock_completed`という2種類の非同期メッセージ（upcall）によってカーネルからの返事を受け取る．

一方，カーネルはページャーからの要求によってのみページの再利用（reclaim）を行うわけではない．ページャーが回収することなしにカーネルにページを供給していった場合や，一定時間使用されないページがあった場合，カーネルはページャーへと自発的にページの返還を行う．この自発的なページのreclaimにより，外部から見た場合には1つに見える状態が，異なる2つの内部状態に分けられる．すなわち，カーネルがキャッシュを持っている場合と，ページャーがキャッシュを持っている場合である．たとえば外部よりキャッシュを捨てるような要求（flush request）が来たとする．このとき，実際にrequestを受け取ったページャーでは現在の内部状態に従って異なる手順を踏む必要がある（**図2** 参照）．もしページャー自身がキャッシュ

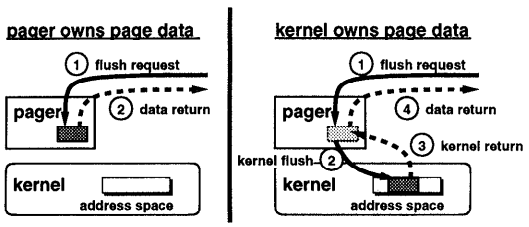


図2 カーネルの内部状態による遷移の違い
Fig. 2 Differences of the internal state during flush request.

を持っているのならば，flush request に対する返事をすぐに返すことができる（図2の左側参照）．そしてキャッシュがない状態へと遷移する．一方，カーネルがキャッシュを持っている場合には，ページャーはまずカーネルに対してキャッシュを捨てるよう要求を出し，それに対する返事が返ってきた後に返答をする必要がある（図2の右側参照）☆．状態遷移図を記述する場合には，このような内部状態も考慮して記述する必要がある．

内部状態による状態数の増加は必ずしも外部ページャーを用いたために生じたものではない．外部ページャーの使用できない OS 上で仮想記憶の保護機構を用いて分散共有メモリを実装する場合には，`mmap` や `mprotect` といったシステムコールを用いて保護機構を制御する必要があるが，この場合にはホスト上の各プロセスにサーバからの upcall を受け付けるためのルーチンが必要となる（ライブラリの中に含まれる）．するとやはりホストごとのグローバルな状態遷移とは別に，ホスト内の内部状態を別に管理する必要が出てくる．さらに，複数のサーバが1つの処理を行うことで処理効率や耐故障性を増すワークステーションクラスタやマルチサーバといった環境では，外部からは複数のサーバが1つのサーバとして見えるように動作する．この場合，サーバを形成する各サーバの状態は内部状態にほかならない．このような構造は分散システムでは頻繁に現れる例の1つである．

2.1.3 メッセージ待ちのための中間遷移状態

サーバ群はメッセージを交換しながら状態遷移をしていく．このときに問題となるのが，サーバがあるリクエストに対する返事（acknowledgment）を待つ場合である．この場合 RPC のように同期的なメッセージを使うことは難しい．なぜならば，分散共有メモリのサーバの場合，複数のサーバ間の調停を行う必要があるためすぐに返事を出すことが難しいうえに，サーバは「返事」のメッセージ以外のメッセージも受け付けなければならないからである．その一方で，非同期的なメッセージを使った場合には返事を待つための状態を新たに設けて，返事を受け付けた後の処理を別の部分に記述する必要がある．このためにプロトコルの記述がメッセージの受信の前後で分断され，直線的な記述ができないことになる．

同期的に待ちながら複数のメッセージを受け付けられるようにするために，サーバをマルチスレッドにするという解決法も考えられる．しかしページ単位でプロトコルの遷移を扱う分散共有メモリのサーバの場合，単純にメッセージ待ちの途中でスレッドを止めてメッセージを待つ方法ではページの数だけのスレッドが必要となる．スレッドの作成にともなう資源（メモリ）を考えた場合，これは現実的でない．

2.2 PCS による解決

PCS では Mach のようなシステムにおける上記の3つの問題点をそれぞれ次の方法で解決する．

(1) メッセージの到着順序を自動的に並べ換える．
(2) カーネル-サーバ間の不必要な通信をユーザから隠す．
(3) 与えられたソースコードを変形し，複数のスレッドがメッセージ待ちに陥ることを防ぐ．

以下順番に説明していく．

2.2.1 メッセージの到着順序の自動的な並べ換え

プロトコルを記述する際に，実際のメッセージの到着順序にかかわらず，処理したい順番にプロトコルを記述できれば，それにともなう状態数も減り，プロトコル記述者の負担は大幅に減る．

プロトコルを処理したい順番に記述しても期待どおりの動作するようにするためには，メッセージを到着したときではなく，到着していてかつプロトコルがそれを処理できる状態のときに「メッセージが到着した」と見なせばよい．こうすることでメッセージの到着順序はユーザから透明な形で並べ換えられる．このために，メッセージはまずキューに入れられ，プロトコルが処理できるメッセージから順に取り出されて，処理される（したがってキューといっても FIFO ではない）．

このような並べ換えを行ったうえで，2.1.1 項で述べた2種類のメッセージ（write reply と invalidate ack）を順不同で受け取る，というケースを考えてみる．この場合プロトコルの記述上は単に write reply を受け取ってから invalidate ack を受け取る，というように記述してあればよい．こうしておけば，たとえ

☆ 返事を待ってからでないと整列性（serializability）が崩れてしまう可能性がある．

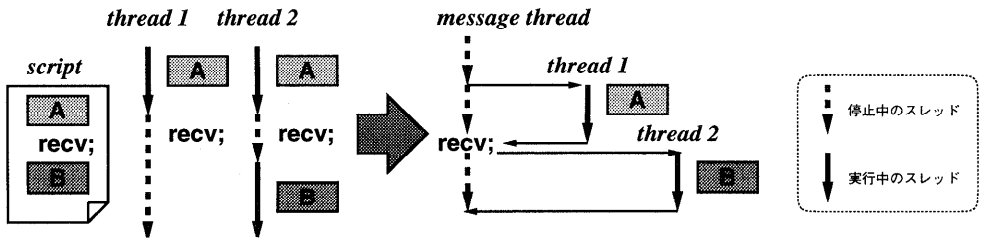


図3 多重スレッド待ちを防ぐ変形
Fig. 3 Transform of scripts for avoidance of multiple threads suspention.

invalidate ack が先に到着したような場合でもそれはキューの中に保存され，その後 write reply が到着し，処理されて invalidate ack が処理可能になったときに初めてキューの中から取り出されて「到着」したことになるからである．

もちろん，メッセージの到着する順序によって処理すべき内容を変えたいような場合には，そのように記述しておけばよい．このメッセージ順序の並べ換えは記述できるプロトコルになんら制限を加えるものではなく，頻繁に現れる処理を簡潔に記述するためのものである．

2.2.2 カーネル–サーバ間通信の隠蔽

ユーザがプロトコルを記述する場合，カーネルと外部ページャーの間のすべての通信を見せる必要はない．カーネルがページをページャーの方に返してくるのは，(1) ページャーがカーネルに対してページを返すように要求したとき，(2) カーネル内にあるページが一定数を超えた，あるいはページがしばらく使われなかったなどの理由で，自主的に返してきたとき，の 2 つの場合が考えられる．(1) は元々プロトコルからの要求によってページを返しているわけだが，(2) に関してはプロトコルでは明示的な処理を書かなくても処理することは可能である．すなわち，ページャー側でそのページをキャッシュしておき，カーネルから要求があった時点で再び供給すればよい．このようにプロトコルが明示的に処理を書かない場合には自動的に処理してやることで，内部状態をプロトコル記述者から隠蔽し，記述の負担を軽減することができる．

2.2.3 メッセージ待ち状態の除去

非同期メッセージを使ってプロトコルを記述した場合，メッセージ待ちのための中間遷移状態が必要となる．プロトコルを直線的に記述できるようにするためには，プロトコルの途中でメッセージを直接同期的に待てる必要がある．しかし実際に同期的な待ちを入れてしまうと，メッセージ待ちの数だけスレッドが必要になる．これによってスレッドの数が爆発的に増え，それにともなう資源が消費されることになる．

本システムでは，ユーザにプロトコルを同期的に記述することを許している．このとき，与えられたプロトコルのソースをメッセージ待ちの命令に従って変形することで，多くのスレッドが同期的なメッセージ待ちに陥らないようにしている（**図3** 参照）．図 3 の左の状態では複数のスレッドが同時にメッセージ待ちのために止まる可能性があるが，図 3 の右の状態ではメッセージ待ちのために止まるのはつねに 1 つのスレッドである．この変形を施すことによって，メッセージを待つ処理はすべて 1 つのメッセージサーバスレッドに任せることができ，メッセージ待ちのためにスレッドの数が爆発するという事態は避けられる．

3. PCS の実装

前章で述べたような設計方針に従い，PCS の実装を行った．この章では具体的な記述言語の仕様などを詳しく説明する．

3.1 プラットフォームと記述言語

PCS は DECstation 5000（MIPS R3000）の上の Mach 3.0 microkernel の上に実装されている．Mach を選んだのは主に外部ページャーの機能を使うためで，外部ページャーの機能さえ代用できれば他のプラットフォームへと移植することは容易である（PCS の機能の一部はすでに Sun OS 上に移植されている）．

PCS 用の記述言語としては，C 言語への組込みが容易であるという点で Tcl[14] を元にしたインタプリタ言語とした．インタプリタを選んだ理由は，プロトタイプとしての記述力に優れていることと，文字列と連想配列が使用できるためにメッセージの扱いが容易であるということである．その分速度の面での犠牲も払わなくてはならないが，分散環境での通信コストに比べれば十分無視できるであろうという予測を立てていた．インタプリタのオーバヘッドについては後で議論する．

PCS が走る各ホスト上には 2 種類のサーバを設け

表1 PCSで予約された属性
Table 1 System-defined attributes in PCS.

属性名	意味	使用例
event	イベント名	create_msg msg "DATA_REQUEST"
reply_to	送り主のポート	send $msg(reply_to) reply_msg
send_to	転送先のポート	set msg(send_to) $owner_port
dataID	データ識別子	set_data $pageID $msg(dataID)
dealloc	データを送り元から消す	set msg(dealloc) 0
prot	保護属性	set msg(prot) RW
pageID	ページ識別子	set msg(pageID) $pageID

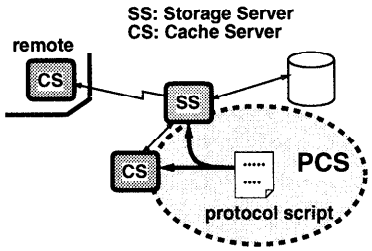


図4 PCSを含むシステム構成
Fig. 4 Servers and PCS.

ている[17]．キャッシュサーバ（CS）とストレージサーバ（SS）である（図4参照）．PCSは与えられたプロトコルスクリプトに従い，2つのサーバをコントロールする．CSはクライアントと直接通信を行い，SSはキャッシュのownerに相当する☆．

3.2 メッセージ：すべての通信の抽象化

PCSはユーザに抽象化されたプロトコル記述モデルを提供する．ここではその中でもプロトコル記述の要となる，通信を抽象化した概念，メッセージについて簡単に説明する．

メッセージはすべての通信を抽象化している．プロトコル中の明示的な送受信のほか，クライアントプログラムからのシステムコールや，システムにおける割込みやイベント（カーネルからのデータ要求や一定時間ごとの割込みなど）もすべて同じ“メッセージ”という枠組みで扱うことができる．各メッセージは必ずイベント名を持ち，メッセージの送受信はイベント名のマッチングによって行われる．

メッセージはイベント名以外にもさまざまな属性を持ち，伝えることができる．メッセージ中にはデータ（メモリイメージを抽象化したもの）や，ポート（メッセージの送信先を示す宛名を抽象化したもの）のようにシステムであらかじめ決められている属性もあるが（表1参照），そのほかにもユーザが属性を自由に定義することができるため，プロトコルで必要な付加的な情報を含めて通信することができる．

メッセージの送信は `send`，受信は `recv`，`mrecv`，`select` という各プリミティブによって行う．`send` は行き先のサーバのポートとメッセージ変数を指定する．`recv` には受け取るべきメッセージのイベント名とメッセージ変数を指定する．メッセージの内容は指定したメッセージ変数へと代入される．`select` ではいくつかのイベント名と，それに対する処理を書くことができる．`mrecv` は指定されたイベント名を持つメッセージを指定した数だけ受け取ることを意味する．

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する．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ほかに `msg(event)` のような連想配列がある．値は基本的にはすべて文字列で，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5はwrite-invalidationプロトコルの一部である．1行目の `CS_PAGE` はCS上のページ単位のプロトコルであることを示している☆☆．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2行目に書かれている `CLIENT_DATA_REQUEST` というイベントとして扱われる．3行目では `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．4〜5行目では `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．

☆ ここでは詳しくは述べないが，一般的にownerがつねに固定されている環境の方が，プロトコルの記述は容易である．

☆☆ ほかにファイル単位やホスト単位のプロトコルも記述できる．

```
 1: CS_PAGE: select msg {
 2:     CLIENT_DATA_REQUEST: {
 3:         create_msg rmsg DATA_REQUEST
 4:         set rmsg(prot) $msg(prot)
 5:         set rmsg(need_data) $msg(need_data)
 6:         send [SS_port $pageID] rmsg
 7:         recv DATA_REPLY dmsg
 8:         mrecv $dmsg(inv_cnt) INV_COMPLETED
 9:         if {$dmsg(should_return) == 1} {
10:             create_msg ack DATA_RECEIVED
11:             send [SS_port $pageID] ack
12:         }
13:         if {$rmsg(need_data) == 1} {
14:             set_data $pageID $dmsg(dataID)
15:         }
16:         set_prot $pageID $msg(prot)
17:     }
18:     ......
19: }
```

図5 プロトコル記述の例
Fig. 5 An example of protocol description.

6〜8行目の3行が2.1.1項で触れたDASHでのwrite requestプロトコルに対応した部分である．**send**文でSSに対して**rmsg**を送り，読み書き可能なページを要求する．次の**recv**文がSSからのリプライ（イベント名**DATA_REPLY**），その次の**mrecv**文は他のCSからのack（**INV_COMPLETED**）を待つ．送られてくるackの個数はSSからのリプライの中に含まれていて（**$dmsg(inv_cnt)**），実際にはリプライメッセージの到着がackよりも遅れる可能性もあるが，これだけの記述ですべてのメッセージを正しく受理することができる．Machの外部ページャーを使ってこれと同じ処理を記述すると，これだけの処理をするのにC言語にして50行以上の記述が必要であった．

9〜12行目は，返事を返さねばならないとき（**$dmsg(should_return)==1**）には**DATA_RECEIVED**というメッセージを作って返す処理である．13〜15行目でデータが欲しいときには**set_data**で送られてきたデータをセットし（**$dmsg(dataID)**），16行目の**set_prot**によって保護属性を換えて終了である．

4. 実　　験

この章では，実装されたPCSにおけるプロトコルの記述量，プロトコルのカスタマイズによる効果，実行時のオーバヘッドなどについて実験を行った結果を示す．

4.1 プロトコル記述量の比較

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た．ここでは（1）write-invalidate,（2）memory-mapped stream,（3）release consistency,（4）causal consistencyの4つのプロトコルと（5）lock,（6）barrierの2つの同期プリミティブを記述した[16),17),19)]．

表2 プロトコル記述行数の比較
Table 2 Comparison of numbers of protocol description.

lang	protocol	lines
C	write-invalidate	2286
PCS	write-invalidate	211
	memory mapped stream	88
	release consistency	111
	causal consistency	236
	lock	45
	barrier	18

write-invalidate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．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ほかに**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と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なしで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．lockとbarrierに関しては，直接分散共有メモリプロトコルではないが，分散システムの記述の例を示すために取り上げた．

上にあげた4種類のプロトコルと2種類のプリミティブをPCSで記述し，Machの上で書かれたCのプログラムと行数を比較したのが**表2**である．write-invalidateの場合，Machの外部ページャーとしてC言語で記述した場合に比べ，約1/10の量でプロトコルが記述できる．もちろんPCS自体もCで書かれており，その分のソースコード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-invalidate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-invalidateと比べても大分短く記述できる．causal consistencyはwrite-invalidateより長いが，記述のほとんどはwrite-invalidateをそのまま流

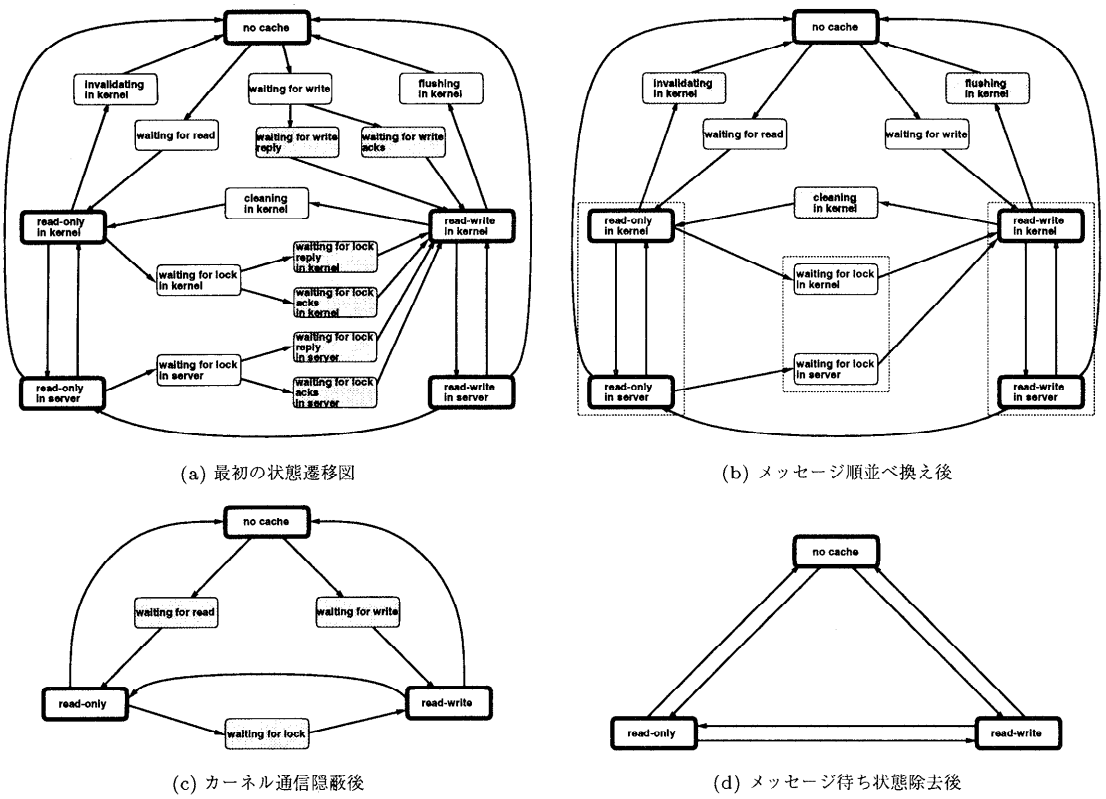


図6 write-invalidate プロトコルの状態遷移図
Fig. 6 State transition diagrams in write-invalidate protocol.

用できるため，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日のうちに記述から試験までを行うことができた．簡単なプロトコルならば数時間もあれば記述可能であり，プロトタイピングの目的にも十分使えると思われる．

4.2 抽象化による状態数の削減

ここでPCSによる抽象化によってどれくらい状態数が減少するかを比較する．比較するのは，write-invalidate プロトコルを Mach の外部ページャーを用いて実装したときの，クライアント側における状態数である（サーバ側ではまた別の状態遷移が必要となる）．

最初にC言語と外部ページャーを用いて記述した，まったく抽象化されていない状態での状態遷移図を**図6**(a)に示す．最初の状態遷移図は5つの基本状態（太線で囲まれているもの）と13の中間遷移状態の18状態，およびそれらの間の状態遷移からなっている．基本状態にはキャッシュがない（no cache），読み出し専用（read only），読み書き可能（read/write）の3状態に加え，キャッシュがカーネル側に存在するか（in kernel）サーバ側に存在するか（in server）によって後者の2つの状態がさらに2つずつに分かれるために5つとなる．5つの基本状態の間の遷移にはそれぞれメッセージ待ちのための中間遷移状態が存在するが，読み書き可能（read/write）へと遷移する場合にはreplyとacksを順序にかかわりなく受け取る必要があるために，2つの中間状態へと枝別れする遷移が存在する（2.1.1項のwrite requestプロトコル参照）．

ここに最初の抽象化であるメッセージの到着順序の並べ換えを施すことで，上で説明した2つの中間遷移状態が消去でき，結果として図6(a)において網目のかかった6つの中間遷移状態が除去できる．これによって図6(b)のように状態数は18から12へと減らすことができる．

2番目の抽象化である外部ページャーとカーネルとの通信の隠蔽によって，内部状態を隠蔽することができる．図6(b)において点線で囲まれた中にある2つの状態は，その内部状態だけが異なっている．したがって内部状態を隠蔽することで，これら2つの状態は同じ状態として扱えるようになる．また網目のかかった状態は，カーネルからの非同期的なメッセージを受け取るのための待ち状態であるが，これも除去すること

ができる．結果として図6(c)のように6つの状態だけが残されることになる．

最後の抽象化であるメッセージ待ち状態の除去により，図6(c)の3つの中間遷移状態も除去できる．最後に残された状態は図6(d)にあるように，読み書き禁止（no cache），書き込み禁止（read-only），読み書き可能（read/write），という基本的な3状態だけである．

4.3 PCSを用いたカスタマイズ

PCSを用いて実際にプロトコルをカスタマイズする例を示す．ここであげる例はSPLASH-2ベンチマークの中の基数ソート（radix sorting）を用いている．この基数ソートでは，各タスクが分配された配列に対する頻度を数え上げるフェーズ（以下，更新フェーズ）と，その頻度に従って新しい配列に並び替えるというフェーズ（以下，参照フェーズ）が存在する．この2つのフェーズではタスク間の「頻度表」の共有の仕方が異なることを利用して最適化を行う．

4.3.1 lock-adaptiveプロトコル

頻度を表すテーブル（以下，頻度表と呼ぶ）は各タスクから共有されている．この頻度表を管理するためのキャッシュ一貫性プロトコルにはwrite-invalidateプロトコルとmigratoryプロトコルが考えられる．write-invalidateプロトコルを用いた場合，参照フェーズでは効率的な共有ができる．しかし更新フェーズでは最初の読み込みの際にいったん共有状態（shared）になり，直後に占有状態（exclusive）に移行する，という無駄な遷移が生じる．一方，migratoryプロトコルを用いた場合には，読み込みが起こると占有状態（exclusive）に移行するため，更新フェーズでの無駄な遷移が生じない．しかし共有することができないため，参照フェーズでは非常に非効率的になってしまう．

そこでこの2つのプロトコルをフェーズごとに切り替えることを考える．プロトコルの切替えはアプリケーションに手を入れて明示的に切り替えさせることも可能であるが，今回の場合にはロックの取得を情報として利用することができる．すなわち，更新フェーズではロックをかけてからアクセスし，参照フェーズではロックをかけないことを利用する．このプロトコルをlock-adaptiveプロトコルと呼ぶことにする．lock-adaptiveプロトコルでは，ロックを持っていないときはwrite-invalidateと同じように，ロックを持っているときにはmigratoryプロトコルと同じように動作する（**表3**参照）．

4.3.2 prefetchの併用

さらにロックは複数のページ（セクションと呼ぶ）に対してかけられるため，ロックを取得した時点でセクション内のページに対する要求を（on demandではなく）発行してしまうことも考えられる．すなわちページの先読み（prefetch）である．prefetchは，要求していて実際にアクセスが起きなかったような場合には不利になるが，今回の場合のようにアクセスされることがほぼ確実に分かっている場合にはかなりの改善が見込める．セクション内のページの要求は1つのメッセージにまとめて（piggybackingして）送るため，遅延時間（latency）の削減だけでなく，メッセージの個数を減らす効果もある．

表3 基数ソートにおけるプロトコルの比較
Table 3 Comparison of 3 protocols in radix sorting.

プロトコル	更新フェーズ	参照フェーズ
write-invalidate	△（無駄な遷移）	○
migratory	○	×（共有不可）
lock-adaptive	○	○

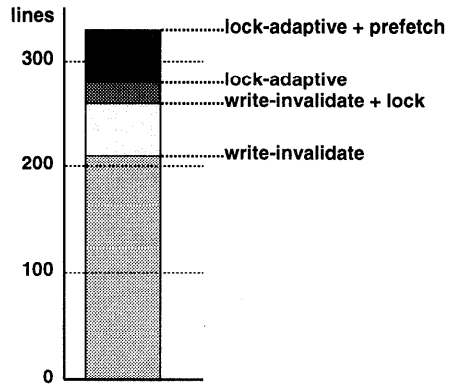


図7 lock-adaptive+prefetchの記述行数の内訳
Fig. 7 A breakdown of numbers of protocol description.

4.3.3 プロトコル記述行数と状態数の比較

これらのプロトコルの記述行数を比較してみたのが**図7**である．PCSを用いてlock-adaptiveを実現するには，write-invalidateの分とlockの分を合わせた行数に比べてわずか13行記述を追加すればよい．さらにprefetchを記述するためには約50行の追加が必要となるが，全体の行数から見ればわずか15%程度である．

lock-adaptiveプロトコルを，PCSによって抽象化されていないMach上の外部ページャーに組み込むことを考える．4.2節で見たように，抽象化されていないwrite-invalidateプロトコルでは全部で18の状態があった．非常に単純に考えれば，これらの状態のそれぞれにつきロックを取得しているかしていないか，という状態が考えられるので，状態数は2倍になる．

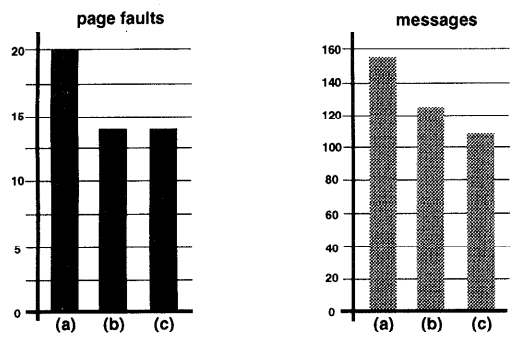


図8 ページフォールトと遠隔メッセージの個数．(a) write-invalidate のみ，(b) lock-adaptive，(c) lock-adaptive + prefetch
Fig. 8 Comparison of numbers of page faults and remote messages.

実際にはロックを指定している場合には migratory プロトコルになるため，共有状態はありえない．よって，読み書き可能に関連する 7 状態が増え，全部で 25 状態の状態遷移図になると考えられる．一方，PCS によって抽象化されたプロトコルでは，ロックがないときの 3 状態（write-invalidate プロトコルと同じ）に，ロックがあるときの 2 状態（ロックがあるときには migratory プロトコルと同じになるので「共有状態」がない）を加えて，全部で 5 状態になるだけである．

4.3.4 プロトコル最適化の効果

以上の結果を元に，(a) write-invalidate，(b) lock-adaptive，(c) lock-adaptive + prefetch の 3 種類についてページフォールトの回数と遠隔メッセージの個数をカウントした結果が**図 8** である．この結果から，lock-adaptive によって約 1/3 のページフォールトを削減できたことが分かる．またさらに prefetch を併用することによって，約 10%の遠隔メッセージの数を減らすことができていることが分かる☆．

4.4 オーバヘッド

この節では，PCS のオーバヘッドについて考察する．現在のバージョンの PCS はインタプリタを利用しているが，本システムを実装する前の段階で，インタプリタのオーバヘッドは遠隔メッセージ通信のコストに比べて十分小さいであろうとの予想を立てていた．

PCS は 3.1 節で述べたように，DECstation 5000（MIPS R3000，33 MHz と 25 MHz）上の Mach マイクロカーネルの上に外部ページャーの機能を使って実装されている．**表 4** にこの環境における基本的な実行の速度を示した．`mach_task_self()` システムコールはユーザモードとカーネルモードとの切替えのコストを示している．スレッドの切替えは cthread ライブラリの `condition_wait()` と `condition_signal()` を用いて行った．ページフォールトのコストはメモリアクセスが起き，復帰するまでの時間を計ったものである（ページャーは空白ページを提供する）．メッセージのコストはもう 1 台の DECstation 5000（MIPS R3000 20 MHz）との間を 10 Mbps の Ethernet でつないだ環境で計測した．

☆ prefetch によってページフォールトの回数が変わらないのは，要求されていないページを供給することができないという Mach の外部ページャーの制約によるもので，本質的にはフォールトを起こさないようにできる．

表 4 基本操作にかかるコスト
Table 4 Costs of primitive actions on the environment.

操作	コスト（msec）
mach_task_self()	0.031
スレッド切替え	0.018
ページフォールト（組込ページャー）	3.9
ページフォールト（外部ページャー）	1.6
空のメッセージ送信	7.42
空の遠隔手続き呼び出し（RPC）	19.0
n ページのデータを送信	$16.6 + 9.43n$

表 5 ページフォールト解決のコストの比較
Table 5 Comparison of costs of page faults.

	ローカル	リモート
(a) PCS なし（msec）	5.93	49.2
(b) PCS あり（msec）	18.2	62.9
オーバヘッド (b)−(a)	12.3	13.7
比 (b)/(a)	3.07	1.28

4.4.1 PCS のオーバヘッド

表 5 は，プロトコルとして実際に write-invalidate プロトコルを使用している場合にページフォールトの解決にかかる時間を，PCS を (a) 組み込む前と (b) 組み込んだ後で比較してみたものである．PCS を組み込んだ方がローカルの場合には約 3 倍，リモートの場合には 28%ほど大きくなっている．インタプリタ自身のコストは変わらないために，リモートの場合には通信のコストが大きい分，ローカルの場合に比べて相対的にインタプリタのオーバヘッドは小さくなる．

4.4.2 オーバヘッドと最適化の効果の兼ね合い

PCS を組み込んだことによるオーバヘッドと，最適化によるパフォーマンス向上の効果について検討する．全体の実行時間についての比較は，今回の最適化と直接関係のないフェーズにおけるフォールスシェアリングの影響が大きすぎるため，意味のある比較ができなかった．そこでページフォールトのコストを基準にして比較を行う．

図 8 より，最適化を行わない場合のページフォール

表6 Tclインタプリタの実行速度
Table 6 Execution speed of Tcl interpreter.

実行文	コスト（msec）
Tcl_Eval “set a 1”	0.052
Tcl_Eval “create_msg”	0.19
Tcl_Eval “send”	0.46

トの回数は20回，最適化することにより13回まで減らせる．一方，リモートでページフォールトを処理するコストは表5より，PCSを組み込んでいない場合は49.2 msec，組み込んだ場合は62.9 msecである．よって，PCSなしで最適化しない場合のページフォールト時間は984 msec，PCSを組み込んで最適化した場合には818 msecとなり，この部分だけで比較すればオーバヘッドの分を考慮しても最適化の効果が上回っていることが分かる．

ただしローカルのページフォールトを処理するコストは，ページフォールトの回数がリモートの場合と同じと仮定すると，PCSを組み込んでいない場合の119 msecに対して，組み込んでいる場合は237 msecとなる．ローカルなページフォールトとリモートのページフォールトの割合はアプリケーションの振舞いによるが，PCSを用いて最適化を実用的に行うためにはさらなるオーバヘッドの削減が必要であると考えられる．

4.4.3 オーバヘッドの原因

表6にTclインタプリタの実行速度を示す．同じ1行であっても命令によって速度はさまざまで，その中で`set a 1`は速い命令の代表，`send`は最も時間がかかる命令の代表として取り上げた．この表から，命令にもよるが，1行を解釈・実行するのにおよそ50 μsecから500 μsec程度かかることが分かる．この計測結果を元に，PCSプロトコルでページフォールトのハンドルに必要なコード（図5）を実行するのにかかるコストを計算すると，多くても5 msecくらいとなる．したがってオーバヘッドとなっている12〜13 msecのコストの残りの半分以上はインタプリタを起動するためのコストであると推測される．実際に計測してみると，Tclインタプリタでは変数のためのハッシュ表などを作るために，起動時に大きなコストがかかることが分かった．したがって，このコストは中間コード形式にコンパイルするなどの改良を加えることで，軽減できると考えている．

5. 他のシステムとの比較

この章ではプロトコルの選択を行うシステム，抽象化がなされていないプロトコル記述システム，宣言型によるプロトコル記述システムのそれぞれとPCSとの比較を行う．すでにいくつかのシステムで採り入れられている方式として，ユーザに適切なプロトコルを選択させるアプローチがある．しかしMunin[3]を始めとするプロトコルの選択が可能なシステムの多くは，共有のポリシーを選択するに留まり，具体的な動作のメカニズムにまでユーザの手が入れられるシステムは少ない．また同じポリシーを実現するプロトコルでも環境によって適切なメカニズムが異なる可能性があるため，PCSのようにメカニズムにまで手を入れられるシステムは，選択のみを許すシステムに比べて柔軟性が高い．さらにPCSではユーザが自由にサーバとのインタフェースを追加できるため，アプリケーションとより密な協調動作をさせることが可能となる．サーバを分散共有メモリとアプリケーションとの仲介役として使うことで，よりメッセージ通信に近いプロトコルを実現することが可能になる．

またPCSがターゲットとする分散協調アプリケーションの動作については未知数の部分が大きく，現在提案されているプロトコルだけで十分であるとはいえない．たとえば通常の分散共有メモリでは読み出しのみと読み書き可能の2種類のロックを用いるが，グループ指向トランザクションモデル[9]では5種類のロックを用いるため，このトランザクションモデルを分散で効率的に実装しようと思えば，従来のプロトコルを拡張する必要がある．もちろん有用であることが分かった時点で，それらのプロトコルをすべて選択可能にすればよいという考え方もあるが，1章でも述べたように協調的トランザクションモデルはいくつもあり，また次々に提案されている状況なので，それらすべてに対応するプロトコルを提供するというのはあまり現実的な解法ではない．

特に抽象化がなされていないプロトコル記述システム[5),10]とPCSとを比較した場合，抽象化による汎用性の低下が問題となる．しかしPCSの抽象化は特に記述できるプロトコルの汎用性を下げてはいない．たとえばメッセージの到着順序の入れ替えは到着の順序を気にしなくてよい場合の記述を簡潔に書けるようにするものであり，到着の順序によって動作を変えたい場合の記述を制限するものではない．またプロトコル記述のプロトタイピングのためにはPCSのような抽象度の高いカスタマイズの枠組みは有効である．

プロトコルの記述を可能にするシステムの場合でも，PCSのような手続き型言語による方法ではなく，状態とその間の遷移を表のような形で与える宣言型のアプローチが考えられる．この宣言型の利点はすべての

状態間の遷移を機械的にチェックすることができるため，プロトコルの自動検証や最適化と言った工夫がしやすいことである．しかしこの点に関しては，手続き型でもプロトコルの自動的な検証はある程度可能である．たとえばTeapot[4]では与えられたプロトコルのソースを，実行用のC言語とプロトコル検証用の言語の両方に翻訳できるようになっている．一方，宣言型によるアプローチの欠点としては，単純な拡張がしづらいという点がある．たとえば手続き型の場合には1つのフラグを追加し，1つのif文を追加するだけで解決するような拡張でも，宣言型の場合にはそのフラグの有無を考えた「状態」を考えてそれらの間の遷移を正しく記述する必要がある（最悪の場合，1つのフラグの追加で状態数が2倍になってしまう）．また，宣言型の場合には，手続きの途中でメッセージの返事を待つ，という記述はできず，メッセージ待ちのための状態を加えなければならない．今回のPCSの目標として「ユーザが管理すべき状態数を減らす」というものがあるため，あえて宣言的なアプローチはとらずに手続き型のインタフェースを用いた．もちろん，宣言型においても，与えられたわずかな状態遷移を実際の（多状態の）状態遷移へと自動的に変換するなどの処理を施すことで，同様の効果は得られるのではないかと考えられる．これは今後の研究課題である．

6. 結　　論

アプリケーションのアクセスパターンに応じて，分散共有メモリのプロトコルを記述・カスタマイズするためのプロトコルカスタマイズシステム（PCS）について，その設計方針に主眼をおいて述べた．本システムの特長は，ユーザが管理すべき状態数というものに着目し，カーネルの機能をユーザレベルに開放しているだけのシステム（Machの外部ページャーなど）に比べて，状態数を軽減するための抽象化を行っている点にある．状態数が減ることによってユーザは既存のプロトコルに対して容易に変更を加えることができ，また新しいプロトコルを記述することもたやすくなる．

インタプリタを組み込んだことによるオーバヘッドは，現在のところ実装に利用したTclインタプリタの起動コストが大きいために，特にローカルな通信の場合には無視できないほどになっているが，これらのオーバヘッドはインタプリタ部分の最適化によって十分に小さくできると考えられる．また，特に遠隔メッセージ通信を含むような場合には現状のインタプリタでも十分な性能が得られると考えられる．

このPCSの本来の目的は序文でも述べたように分散協調アプリケーションである．現在はまだ分散協調アプリケーションに対する十分なデータが揃っていないため，実験には数値計算アプリケーションを使用した．最終的にはPCSは分散協調トランザクションの記述システム[15),18)]の一部として組み込まれる予定である．

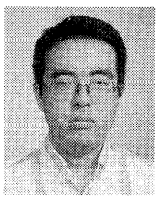
謝辞 研究に対して適切なアドバイスをくださった研究室のメンバ各位，および本稿の執筆にあたり有益な助言をくださった査読者の方々に感謝の意を表します．

参 考 文 献

1) Barghouti, N.S. and Kaiser, G.E.: Concurrency Control in Advanced Database Applications, *ACM Computing Surveys*, Vol.23, No.3, pp.269–317 (1991).
2) Bershad, B.N., Zekauskas, M.J. and Sawdon, W.A.: The Midway Distributed Shared Memory System, *Proc. 1993 IEEE CompCon Conference*, pp.528–537 (1993).
3) Carter, J.B., Bennett, J.K. and Zwaenepoel, W.: Implementation and Performance of Munin, *Proc. 13th ACM Symposium on Operating Systems Principles*, pp.152–164 (1991).
4) Chandra, S., Richards, B. and Larus, J.R.: Teapot: Language Support for Writing Memory Coherency Protocols, *Proc. 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.237–248 (1996).
5) Falsafi, B., Lebeck, A.R., Reinhardt, S.K., Schoinas, I., Hill, M.D., Larus, J.R., Rogers, A. and Wood, D.A.: Application-specific Protocols for User-level Shared Memory, Technical Report, TR 1239, Computer Sciences Department, University of Wisconsin, Madison, WI (1994).
6) Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors, *Proc. 17th Annual International Symposium on Computer Architecture*, IEEE, pp.15–26 (1990).
7) Johnson, K.L., Kaashoek, M.F. and Wallach, D.A.: CRL: High-performance All-software Distributed Shared Memory, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp.213–228 (1995).
8) Keleher, P., Cox, A.L. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *Proc. 19th Annual In-*

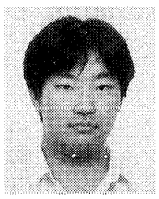
ternational Symposium on Computer Architecture, pp.13 21 (1992).
9) Klahold, P., Schlageter, G., Unland, R. and Wilkes, W.: A Transaction Model Supporting Complex Applications in Integrated Information Systems, *Proc. ACM SIGMOD International Conference on the Management of Data*, pp.388–401 (1985).
10) Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M. and Hennessy, J.: The Stanford FLASH Multiprocessor, *Proc. 21st Annual International Symposium on Computer Architecture*, ACM SIGARCH, pp.302–313 (1994).
11) Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M. and Lam, M.S.: The Stanford DASH Multiprocessor, *IEEE Computer*, Vol.25, No.3, pp.63–79 (1992).
12) Loepere, K.: *Mach 3 Server Writer's Guide*, Open Software Foundation and Carnegie Mellon University (1992).
13) 松本 尚，平木 敬：Memory-Based Processor による分散共有メモリ，並列処理シンポジウム JSPP '93 論文集，pp.245–252 (1993).
14) Ousterhout, J.K.: Tcl: An Embeddable Command Language, *Proc. 1990 Winter USENIX Conference*, pp.133–146 (1990).
15) Uehara, K., Inohara, S., Miyazawa, H., Yamamoto, K., Hara, M. and Masuda, T.: A Framework of Customizing Transaction in Persistent Object Management for Advanced Applications, *Proc. Fourth International Workshop on Object Orientation in Operating Systems*, pp.84–93 (1995).
16) Uehara, K., Inohara, S., Miyazawa, H., Yamamoto, K. and Masuda, T.: A Framework for Customizing Coherence Protocols of Distributed File Caches, *Proc. IEEE 16th International Conference on Distributed Computing Systems*, pp.83–90 (1996).
17) Uehara, K., Miyazawa, H., Yamamoto, K., Inohara, S. and Masuda, T.: A Framework for Customizing Coherence Protocols of Distributed File Caches in Lucas File System, Technical Report, 94-14, Department of Information Science, Faculty of Science, University of Tokyo (1994).
18) 上原敬太郎，猪原茂和，益田隆司：分散協調トランザクション記述のためのフレームワーク，第8回コンピュータシステムシンポジウム，情処シンポジウム論文集，Vol.96, No.7, pp.45–52 (1996).
19) 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 (1994).

(平成9年2月17日受付)
(平成9年7月 1 日採録)



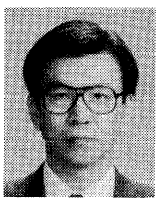
上原敬太郎（学生会員）

1970年生．1993年東京大学理学部情報科学科卒業．1995年同大学大学院理学系研究科情報科学専攻修士課程修了．現在，同専攻博士課程在籍中．分散ファイルシステム，分散共有メモリ，分散永続オブジェクト管理システム，分散協調トランザクションシステムなどのシステムソフトウェアに興味を持つ．



猪原 茂和（正会員）

1967年生．1993年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．1995年同専攻より学位取得．1996年より現在まで（株）日立製作所中央研究所．分散・並列オペレーティングシステムおよびミドルウェア，分散・並列データベース管理システム，トランザクション処理などのシステムソフトウェアに興味を持つ．理学博士．ACM，IEEE，USENIX各会員．



益田 隆司（正会員）

1939年生．1963年東京大学工学部応用物理学科卒業．1965年同大学大学院修士課程修了．同年（株）日立製作所入社．1977年から筑波大学，1988年3月から東京大学に勤務．現在，同大学大学院理学系研究科情報科学専攻教授．専門はオペレーティングシステム．