システムソフトウェアと
オペレーティング・システム 65−14
（１９９４．７．２１）

TRaP-RPC: 分散透明な遠隔ポインタの効率的な実現法について

河野 健二† 加藤 和彦†† 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

††筑波大学 電子・情報工学系
〒305 茨城県つくば市天王台 1-1-1

要旨

遠隔手続き呼び出し (RPC) は、分散システムを構築するために実用的に広く用いられている方法のひとつである。しかし従来の RPC には明らかな制限があった。プログラマが明示的にコードを記述しない限り、遠隔手続きの引数としてポインタ (アドレス) を利用することができない。本論文ではこの制限を取り除くための方法を述べる。この方法を用いると RPC において遠隔ポインタを分散透明に扱うことが可能になる。また、この方法に基づいて TRaP-RPC システムを実装し、実行性能の測定を行なってシステムの有効性を実証した。

TRaP-RPC: Implementation Technique for Transparent Remote Pointers

Kenji Kono†, Kazuhiko Kato†† and Takashi Masuda†

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

††Institute of Information Sciences
and Electronics
University of Tsukuba
Tsukuba, Ibaraki 305, Japan

Abstract

Remote procedure call (RPC) systems have been proven to be a practical basis for building distributed applications. Compared with an ordinary procedure call, however, the conventional RPC technique has one evident restriction; pointers (addresses) cannot be passed to remote procedures without the explicit and nontrivial programming effort. This paper presents a method that eliminates this restriction. The method enables transparent treatment of pointers in RPC. The experiments performed using an implementation of the method show that the method provides performance that is scalable to the access ratio of the remotely referenced data.

1 はじめに

遠隔手続き呼び出し(RPC)は、分散システムを構築するための重要な技術のひとつである。RPCとは、異なる計算機上で定義された手続きを、あたかも通常の手続きであるかのように呼び出す技術である。多くのRPCシステムでは、スタブと呼ばれるいくつかの手続きを生成するコード生成系を提供している。実行時に必要となるネットワークを介した通信は、これらのスタブ中に隠蔽されている。これによって、遠隔手続きを通常の手続きと同じように記述したり利用することが可能となる。RPCを用いると、異なるアーキテクチャを持った計算機同士を比較的容易に接続することができる。遠隔手続きの引数をネットワークのメッセージに詰め込むときにデータ表現の正準形に変換し、メッセージから引数を取り出すときに正準形からそれぞれのデータの内部表現を得ればよい。データの内部表現と正準形との変換を行なうコードをスタブ中に含めることによって、異機種分散環境であってもデータ型を保存してRPCを実現することができる。

この方法の実用性は広く認識されており、多くの実験的なシステムや商用のシステムで利用されている。しかし従来のRPCには明らかな制限がある。遠隔手続きの引数にポインタ型のデータや高階関数を利用することができない。通常のプログラミング言語では、ポインタ(アドレス)を手続きの引数として利用することがしばしばであるにもかかわらず、従来のRPCシステムではプログラマ自身が明示的にコードを記述しない限り、ポインタ型の引数を用いることができなかった。

本論文で述べるTRaP-RPCシステムでは、遠隔手続きの引数としてポインタ型のデータを自由に利用できる。本論文では、TRaP-RPCにおける遠隔ポインタの実現方法[6]について述べる。TRaP-RPCではプログラミング言語による支援なしに、通常のポインタと全く同様に遠隔ポインタを扱うことができ、あたかもアドレス空間の境界が存在しないかのように自由にポインタを利用できる。ソース・コード上で透明に遠隔ポインタを扱うことができるだけではない。ひとたび遠隔サイト上のデータを参照すると、そのデータはローカルにキャッシュされ、以後の参照ではローカルな通常のデータ参照と全く同じコストでそのデータを参照できる。ただし、キャッシュされたデータを更新した場合には、一貫性維持のためのコストが必要になる。遠隔ポインタを分散透明に実現するには、1)遠隔のデータの参照を効率的に検出する方法、2)遠隔ポインタを通常のポインタと同様に扱うための方法、3)キャッシュされたデータと元のデータの間の一貫性を保証するプロトコル、という三点が問題となる。これらの問題を解決するために、1)仮想記憶の機構の利用、2)ポインタ変換(pointer swizzling)、3)一貫性維持のためのプロトコル、という三つの技術をRPCシステムに統合した。仮想記憶の機構ではMemory Management Unit(MMU)のハードウェアを利用している。このハードウェアを利用して、遠隔データへの最初のアクセス要求を効率的にかつ透明に検出することが可能になった。ポインタ変換とは、永続オブジェクトや永続プログラミング言語[7, 4]において利用されている技術である。この技術を分散システムに応用することによって、遠隔ポインタと通常のポインタを全く同一に扱うことが可能となった。仮想記憶の機構とポインタ変換を組み合せるというアイデアは、P. Wilson[9]によって初めて提案された。Wilsonの方法は、非分散の環境でワード・サイズよりも大きいアドレス空間を効率的に提供するための方法として提案された。TRaP-RPCでは、Wilsonの方法を異機種分散環境にまで拡張して利用している。

キャッシュされたデータに更新を行なった場合、その更新を元のデータに反映する必要がある。そこで、キャッシュと元のデータの間の一貫性維持のためのプロトコルを設計した。このプロトコルはRPCの同期的な特性を利用し、簡潔で軽量なプロトコルとなっている。

2 問題点

TRaP-RPCでの遠隔ポインタの実現方法を述べる前に、従来のRPCではどのようにポインタを扱っていたかを説明し、問題となる点を考察する。

2.1 Eagerな方法とLazyな方法

遠隔手続きにポインタを渡すための従来の方法は、ポインタによって参照されているデータを*eager*に転送する方法と*lazy*に転送する方法とに分類される。

まずeagerな方法を説明する。eagerな方法は、遠隔手続きに引数として渡されるポインタの推移的閉包全体を、RPCの引数として一度に転送する方法である。この方法は、ポインタをたどって参照され得るすべてのデータを、遠隔手続きから要求される前に転送するとい

う点で *eager* である。この方法はポインタによって参照されているデータが比較的小さい場合には有効である。しかし、ポインタによって巨大なデータが参照されており、そのデータの一部分のみが遠隔手続きの実行中に必要とされる場合には、この方法は適していない。なぜなら、巨大なデータ全体をネットワークのメッセージに変換しそれを転送するというオーバーヘッドによって、実行効率が著しく低下するからである。

次に lazy な方法であるが、これは callback と呼ばれる機構を利用する方法である。callback とは、RPC で呼び出された側のプロセスが、逆に呼び出し側のプロセスを呼び出すことをいう。なお、手続きを呼び出す側のプロセスを *caller* と呼び、呼び出された側のプロセスを *callee* と呼ぶ。遠隔手続きの実行中に、遠隔ポインタが参照しているデータが必要になると caller に callback をし、遠隔ポインタが参照しているデータを転送するように要求する。こうして、ポインタが参照しているデータを必要に応じて *lazy* に転送する。この方法は、巨大なデータの一部分のみを遠隔手続きから参照する場合に適している。しかし、ポインタを参照する回数が増えると callback の回数も増え、実行効率が低下する。また、単純な実装では一度転送されたデータでも参照される度に callback が起きてしまう。

上で述べた問題を解決する方法として、遠隔のデータを“キャッシュ”するという方法が考えられる。遠隔のデータを初めて参照したときに、そのデータをローカルにキャッシュしておき、以後、そのデータを参照するときにはキャッシュを参照する。こうすれば、遠隔のデータへの参照が必要最小限に押えられるので実行効率が向上する。しかし、この方法を実現しようとするとさまざまな技術的な問題が生じる。1) 遠隔データの最初の参照と、それ以後の参照を効率的に区別する方法、2) ソース・コード上の透明性を確保する方法、3) キャッシュに更新があった場合に、キャッシュと元のデータの一貫性維持の方法、を確立する必要がある。

3 TRaP-RPC の実現機構

本節では TRaP-RPC の遠隔ポインタの実現方法を述べる。TRaP-RPC は lazy な方法と eager な方法の利点をあわせ持つ。

3.1 RPC のモデルと用語の定義

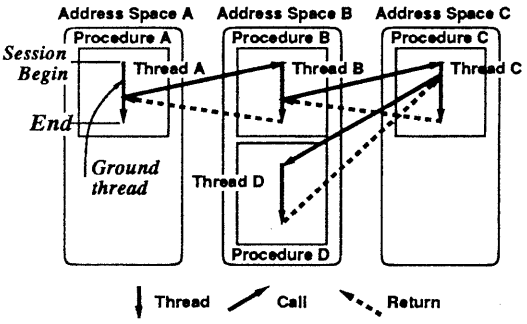

図 1: RPC の実行モデル

TRaP-RPC で仮定されている RPC のモデルを説明する (図 1を参照)。プログラムはアドレス空間中のスレッドによって実行される。スレッドが遠隔手続きを呼び出すと、遠隔手続きから返り値が返されるまで、そのスレッドの実行は止められる。次に、呼び出されたサイト上で新しいスレッドが作られ、遠隔手続きが実行される。遠隔手続きの実行が終了すると、その遠隔手続きを実行するために作られたスレッドも終了する。RPC によって起動されたスレッドでないスレッドを基底スレッドと呼ぶ。多くの場合、基底スレッドはユーザによって起動されたスレッドである。図 1では、手続き A を実行しているスレッドが基底スレッドである。基底スレッドは RPC の“セッション”の開始と終了を宣言する必要がある。RPC のセッション中でのみ遠隔データの一貫性が維持され、遠隔ポインタが意味を持つ。

RPC はネストできる。図 1ではひとつのセッション内で三つの RPC がネストしている。callback も可能である。また、RPC の同期的な特性からひとつのセッション内ではただひとつのスレッドが active である。

3.2 基本機構

TRaP-RPC は eager な方法と lazy な方法とキャッシングの技術とを RPC に融合したものであり、その機構は単純ではない。説明を見通しよく行なうために、TRaP-RPC の lazy な部分を先に説明し、次に eager な部分を説明する。また遠隔ポインタの扱いに集中するため、通常の RPC で利用されるスタブ生成の技術は周知のものとする (説明を要する読者は [2] や [8] を参照)。

はじめにいくつかの概念を定義する。一般に、ポインタはそのポインタが定義されたアドレス空間内でのみ意味をもつ。アドレス空間を越えてポインタを渡せるよう

にするには、分散システム全体で意味を持つようにポインタの定義を拡張する必要がある。そこで“長い”ポインタを定義する。長いポインタに対し、通常のポインタを“短い”ポインタと呼ぶ。長いポインタは、1) 分散システム全体で一意に定義されたアドレス空間識別子、2) アドレス空間識別子で指定されたアドレス空間で有効なアドレス、3) 長いポインタが参照しているデータのデータ型を表すデータ型識別子、という三つの要素からなる。分散環境でデータの位置を特定するためには、アドレス空間識別子とそのアドレス空間内で有効なアドレスが必要である。また異機種分散環境では、異なるアーキテクチャの間でのデータ表現の変換のためにデータの型情報が必要である。

通常のハードウェアでは短いポインタしか扱うことができない。そのため、少なくともハードウェアが長いポインタを利用する前までに、長いポインタを短いポインタに変換する必要がある。長いポインタから短いポインタへの変換を *pointer swizzling* と呼び、その逆の変換を *pointer unswizzling* と呼ぶ。

遠隔ポインタを遠隔手続きに渡すとき、caller 側で遠隔ポインタを unswizzling し短いポインタから長いポインタを得る。caller から callee にその長いポインタが渡される。callee はその長いポインタを swizzle して短いポインタを得る。ここで問題が生じる。swizzle して得られる短いポインタは、どのアドレスを参照すればいいのだろうか。callee のアドレス空間内の短いポインタは、そのアドレス空間内のアドレスしか参照できない。しかし、この時点では参照すべきデータは caller のアドレス空間にしか存在しない。この問題は次のように解決される。callee は長いポインタを受け取ると、そのポインタが参照しているデータを格納するためのメモリ領域を、読み出しからも書き込みからも保護されたページに確保する。このメモリ領域は、長いポインタが参照しているデータがコピーされる“予定の”領域である。長いポインタを swizzle して得られる短いポインタはこのメモリ領域を参照する。この時点では、保護されたページにはひとつもデータが入っていないことに注意。

以上の様子は図 2に示されている。二つのポインタ A と B が caller から callee に渡されている。callee 側では保護されたページが確保され、ポインタ A と B がそのページ内の領域を参照している。

保護されたページに確保されたデータへのアクセスは MMU によって検出され、メモリ保護例外が発生する。

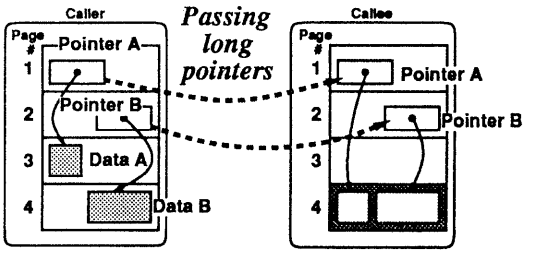

図 2: ポインタ A と B が callee 側で swizzle された直後の様子

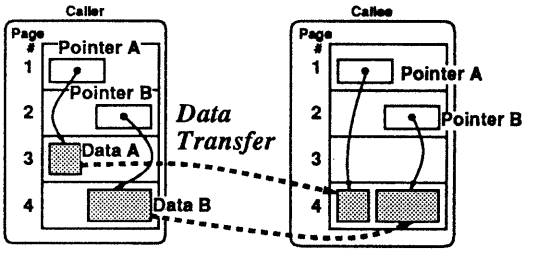

図 3: 遠隔のデータが転送される様子

例外が発生したアドレスは TRaP-RPC の実行時システムに知らされる。この時点で、アクセス要求のあったデータが caller 側から callee 側に転送される。データの転送が終了するとページの保護が解放される。例外が発生したページに割り付けられていたすべてのデータがこの時点で転送される。なぜなら、ひとたびページの保護を解放してしまうと、そのページに確保されていた他のデータへの最初のアクセス要求を検出できなくなるからである。最後に、メモリ保護例外を起こしたスレッドが再開される。図 3はデータ転送の様子を示している。以上の手続きによって、転送されたデータがキャッシュされるので、以後のデータ参照のコストは通常のメモリ参照と全く同等である。この方法は RPC がネストした場合でも正しく動作する。

TRaP-RPC の実行時システムはデータ割り当てテーブルを保持しており、どのようなデータが遠隔サイトから転送されるべきかを記録している。テーブルのエントリは、ページ番号、そのページ内での offset、長いポインタの三つである。図 2の例では表 1 のようになる。

3.3 Eager 性の導入

この節では TRaP-RPC の eager な部分を説明する。TRaP-RPC ではポインタを遠隔のアドレス空間に転送

表 1: データ割り当てテーブル

page #	offset within the page	long pointer
4	$offset_1$	A
4	$offset_2$	B

するとき、そのポインタの、ある一定の深さの推移的閉包をいっしょに転送する。ある深さの推移的閉包をとるアルゴリズムにはさまざまなものがある。TRaP-RPC の現在の実装では、ユーザが指定した大きさの推移的閉包を幅優先探索でとってくる。ユーザが指定する推移的閉包の大きさを *closure size* と呼ぶ。このパラメータを調整すると TRaP-RPC での "eager 性" を調整できる。たとえば、このパラメータを 0 に設定すると TRaP-RPC は 2節で述べた lazy な方法に近くなり、無限大に設定すると eager な方法に近くなる。

3.4 一貫性維持のプロトコル

一般に、キャッシングを利用した技術は元のデータとその複製の間の一貫性を維持する必要がある。TRaP-RPC の一貫性維持のプロトコルは RPC の同期的な特性を利用している。RPC のセッションの間では、たとえ RPC がネストしていても active なスレッドはひとつなので、active なスレッドに関してのみ一貫性が保証されていればよい。

TRaP-RPC の一貫性維持のプロトコルでは、スレッドの activity がアドレス空間の境界を越えて移動する時 (つまり、遠隔手続きを呼び出す時と遠隔手続きから戻る時)、キャッシュ内のすべての更新されたデータを転送する。あるスレッドが RPC を発行したとき、キャッシュに "汚れた" データがあればプロトコルによってすべての汚れたデータが callee に転送される。このプロトコルが正しく機能するのは、このプロトコルが 3.3節で述べた eager な転送方法とみなすことができるからである。更新されたデータは分散計算における working set と考えられるので、これらのデータは eager に転送されたほうがよい。

データの更新は MMU を利用して効率的に検出する。データの更新はページ単位で検出される。遠隔のデータが保護されたページに転送された時、ページの保護モードを read-only に設定しておく。これによって、データを更新しようとするとメモリ保護の例外が発生し、どのページに更新が行なわれようとしたのかを実行時システムが知ることができる。

RPC がネストした場合、すべての汚れたページがアドレス空間の間で転送される。ある時点でこれらのページがもとのアドレス空間に書き戻されない限り、転送される汚れたページの量は単調に増加し、システムの実行性能が低下する。これを避けるために、基底スレッドによってセッションの終了が宣言されたとき、汚れたページの書き戻しを行なう。なお、キャッシュされたデータの解放も同時に行なわれる。

TRaP-RPC は遠隔サイト上のメモリの確保や解放のためのプリミティブも提供している。これについては論文 [6] を参照のこと。

4 実験

これまで述べてきたアルゴリズムの有用性を実証するために、TRaP-RPC を用いていくつかの実験を行なった。TRaP-RPC は Sun SPARC(28.5 MIPS) ワークステーション上に実装されている。OS は SunOS 4.1.1 である。各ワークステーションは 10Mbps の Ethernet で接続されている。通信のプロトコルには TPC/IP を用いた。また、小さなパケットでもバッファリングなしで転送されるように、ソケットに `TCP_NODELAY` というオプションを設定した。データの正準表現には XDR(eXternal Data Representation) を使用し、データ表現の変換のためには SunOS に付属の XDR ライブラリを利用した。現在の TRaP-RPC は数台の SPARC station から構成されているが、システム自身は異機種分散環境でも動作するような配慮のもとで実装されている。本節で示す実験結果には、データ表現の変換などの異機種分散環境で必要となるオーバーヘッドも含んでいる。

4.1 三つの方法の比較

TRaP-RPC で用いられている方法と 2.1節で述べた eager な方法と lazy な方法とを比較する。実験の題材として完全二分木の渡り歩きを用いた。二分木のノードのサイズは 16 バイトである。はじめに caller 側に 32,767 個のノードを持つ完全二分木を作り、この二分木の探索を callee 側から遠隔に行なった。この探索を行なう遠隔手続きの実行時間を、訪問するノードの個数を変えながら測定した。探索は深さ優先探索で行ない、ノードの総数 (32,767) に対する訪問したノードの数の比が、図 4中の x 軸で示された比に到達した時点で遠隔手続きから

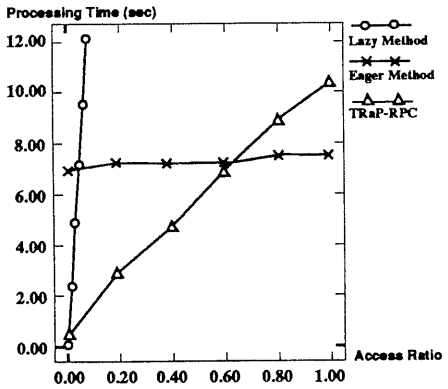

図 4: 三つの方法の比較 x 軸: (訪問したノードの数)/(ノードの総数); y 軸: 実行時間 (秒)

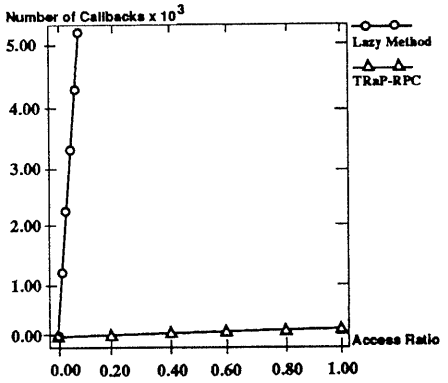

図 5: callback の回数の比較 x 軸: (訪問したノードの数)/(ノードの総数); y 軸: callback の回数

戻る。この遠隔手続きの呼び出しを次の三つの方法で行なった。ひとつは、2.1節で述べた完全に eager な方法である。この方法では遠隔手続きを呼び出すときに、二分木全体 (524,272bytes) を callee に一度に転送する。もうひとつは、2.1節で述べた完全に lazy な方法である。この方法では遠隔手続きを呼び出すときには、二分木の根へのポインタだけを転送する。callee がポインタを参照するたびに callback が行なわれポインタの中身が転送される。最後に TRaP-RPC での方法である。遠隔手続きを呼び出すときに closure size で指定した大きさの推移的閉包をシステムが転送する。callee に転送された閉包はローカルに参照され、ポインタの参照がページフォールトを起こすと、そのポインタの推移的閉包が新たに転送される。この実験では closure size を 8Kbytes に設定した。

実験結果を図 4に示す。Eager な方法では遠隔手続きを呼び出すときにデータ全体を一度に転送するので、実行時間はほぼ一定である。

この実験の場合、lazy な方法は実行効率が悪い。これは callback の回数が増えて、性能が劣化するからである。図 5は y 軸が callback の回数を表している点以外は、図 4と同じである。この図から実行時間の多くが callback に費やされていることがわかる。この実験ではデータ転送の粒度が小さすぎてネットワークのバンド幅を有効に活用できない。Lazy な方法は、ハッシュテーブルの参照のように大きなデータの一部分のみを参照する場合に優れた性能を示す。

TRaP-RPC はアクセス率が 0.0 から 0.6 の間に最も

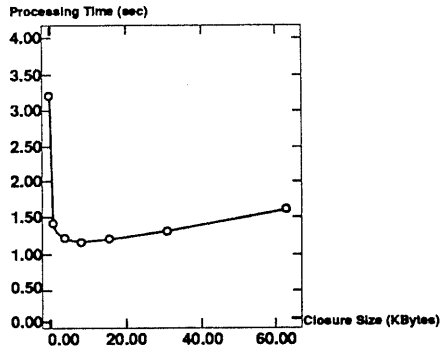

図 6: Closure Size と実行時間の関係

よい性能を示している。これは TRaP-RPC が lazy な側面と eager な側面を合わせ持つことによる。TRaP-RPC が完全に lazy な方法よりも優れた性能を示すのは、eager な側面によって callback の回数が少なく押さえられるからである (図 5を参照)。また eager な方法にくらべ実行効率が優れているのは、TRaP-RPC が lazy な側面を持つので、データの転送量が比較的少なく押さえらるからである。データのアクセス率が 0.6 をこえると callback の回数が増えたために、完全に eager な方法よりも性能が劣る。

4.2 Closure Size の効果

3.3節でも述べたように TRaP-RPC では closure size が重要な役割を果たす。このパラメータが実行効率に与える影響を調べた。実験の題材には前節で用いたものとほぼ同様なものを用いた。まず caller 側に完全二分木を

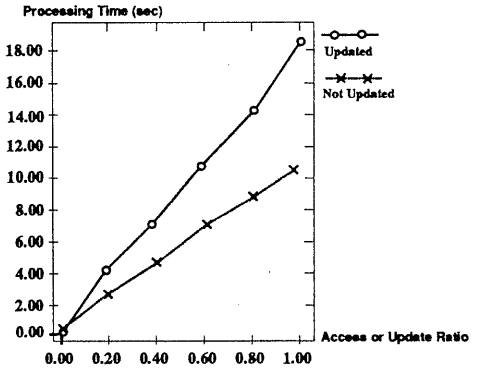

図 7: 更新時のシステムの性能 x 軸: (訪問したノードの数)/(ノードの総数); y 軸: 実行時間 (秒)

作る。その二分木を遠隔に callee 側で根から葉までランダムに 10 回たどり、その実行時間を測定した。図 6 に実験結果を示す。Closure size が小さすぎると実行効率は著しく悪い。これは十分な大きさの closure が転送されないために、callback の回数が多くなるためである。Closure size が大きすぎると参照されないデータを転送するオーバーヘッドが大きくなり、かえって性能が劣化する。この実験では closure size が 8Kbyte のときに最善となっている。

4.3 Update による影響

キャッシュされたデータに更新を行なった場合、システムの性能にどの程度の影響が生じるかを調べた。実験に用いた題材は 4.1節で用いたものと同じである。Closure size は 8Kbyte に設定した。図 7 に実験結果を示す。丸印でプロットされたグラフは更新を行なった場合のグラフであり、訪問したすべてのノードに更新を行なった。バツ印でプロットされたグラフは、更新を行なわずにノードを参照しただけの場合のグラフである。この実験結果は二つの点で妥当である。ひとつは、更新を行なった場合の実行時間が、x 軸に示された更新率に比例して増加している点である。もう一点は、更新した場合の実行時間が、更新をしなかった場合の実行時間のちょうど二倍程度である点である。更新を行なった場合には、データを転送するために一回、更新されたデータを書き戻すために一回の、計二回の通信が行なわれるので、この結果は妥当である。

5 関連研究

本節では本研究の関連研究について述べる。RPC で遠隔ポインタを利用するための従来の方法については 2節で述べたので、その他の方法について述べる。

5.1 プログラミング言語の支援によるアプローチ

分散プログラミング言語の中には、遠隔ポインタに相当する機能を備えたものがある。Orca[1] などの言語は、データ間の論理的な“リンク”を表すためのデータ型を導入している。この論理的なリンクは、CPU によって直接解釈できるポインタとは全く異なり、プログラマはポインタとリンクを完全に区別して利用しなければならない。

CLAM[3] や分散 C 言語 [5] などの言語では、それぞれ C++ や C 言語に RPC の機能を追加し、さらに遠隔ポインタを利用できるようにしている。これらの言語では遠隔ポインタと通常のポインタをプログラマが意識して区別する必要があるが、TRaP-RPC ではプログラマがそのような区別する必要はない。加えて、TRaP-RPC では遠隔のデータを自動的にキャッシュするので、より優れた実行効率が期待できる。

5.2 異機種分散共有メモリ

本研究で述べた方法は分散共有メモリ (DSM) と共通した側面を持つ。両者とも仮想記憶の機構を利用し、遠隔のデータがローカルにあるかのように扱うことができる。多くの DSM は同一のアーキテクチャを持った計算機のみで構成された環境を想定しているが、Mermaid[10] のように異機種分散環境で動作することを想定した DSM もある。異機種 DSM と TRaP-RPC には、異機種分散環境でのデータの共有を可能にするという共通点がある。TRaP-RPC は異機種 DSM と比較して次のような特徴がある。

まず、RPC の同期的な特性により TRaP-RPC の一貫性維持のプロトコルは、DSM システムのそれに比べ軽量である。よって RPC による通信で記述できるアプリケーションであれば、TRaP-RPC を用いたほうが実行効率が優れている。

もう一点は、異機種 DSM で対処できる異機種性には制限があるという点である。Mermaid の場合には、1) すべてのプロセッサが同じアラインメントを使用しなけ

ればならない、2) すべてのコンパイラが同じデータ・フォーマットを用いなければならない、という制限がある。異機種 DSM のこれらの制限はページ単位で分散メモリを共有することから生じる。それに対し TRaP-RPC ではデータの論理的なデータ型を共有している。そのため TRaP-RPC にこうした制約はない。実際に異機種分散システムを構築しようとすると、異機種 DSM のこれらの制限は大きな障害となる。というのも、分散システムを構成するすべての計算機のアーキテクチャを事前に知る必要があるからである。

6 まとめと今後の課題

遠隔手続き呼び出しにおいて分散透明に遠隔ポインタを扱う方法を述べた。この方法は 1) 仮想記憶の機構、2) ポインタ変換 (pointer swizzling)、3) 一貫性維持のプロトコル、という三つの技術を統合している。

仮想記憶の機構を用いて、ポインタによって遠隔から参照されているデータへのアクセス要求を効率的に検出することが可能になった。加えて、遠隔から転送されたデータを自動的にキャッシュすることも可能になった。ポインタ変換の技術によって、遠隔ポインタを通常のポインタと全く同様に扱うことが可能になった。これによって、プログラマはアドレス空間の境界を意識することなしに、自由にポインタを遠隔手続きに渡すことができる。また、キャッシュされたデータに更新を行なうことができるように、データ間の一貫性を維持するためのプロトコルを開発した。このプロトコルは仮想記憶の機構を用いて効率的にデータの更新を検出する。また、RPC の同期的な特性を利用することによって、単純で軽いプロトコルとなった。

現在の TRaP-RPC システムでは、遠隔手続きの引数として高階関数を利用することはできない。最近、大堀と加藤 [8] によって、遠隔手続きの引数として多相型の高階関数を自由に利用することが可能となるような、スタブ生成の系統的な方法が提案された。彼らの方法は、非分散の通常のプログラミング言語を分散対応に拡張し、その拡張された言語から元の言語への系統的な変換アルゴリズムを与えている。この方法では、この変換を行なうためのトランスレータを拡張された言語ごとに作成する必要がある。このトランスレータを生成するためのトランスレータ生成系の実現方法が、すでに論文 [11] で述べられている。以上の方法と本稿で述べた方法を統合し、できる限り制約を取り除き、かつ多くのプログラミング言語に適用可能な RPC システムを実現することを予定している。

また、TRaP-RPC の機構を分散オブジェクトの実装に応用することも考えている。

謝辞

本研究を進めるにあたって、有益な助言を下さった京都大学数理解析研究所の大堀淳先生に感謝いたします。

参考文献

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems, March 1992.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, Feb. 1984.

[3] L. A. Call, D. L. Cohrs, and B. P. Miller. CLAM — an open system for graphical user interfaces. In *ACM OOPSLA '87*, pages 277–286, 1987.

[4] K. Kato, A. Narita, S. Inohara, and T. Masuda. Distributed shared repository: A unified approach to distribution and persistency. In *Proc. IEEE 13th Int. Conf. on Distributed Computing Systems*, pages 20–29, May 1993.

[5] K. Kato, A. Ohori, T. Murakami, and T. Masuda. Distributed C language based on higher-order remote procedure call techniques. *JSSST Advances in Software Science and Technology*, 5:65–82, 1992.

[6] K. Kono, K. Kato, and T. Masuda. Smart remote procedure calls: Transparent treatment of remote pointers. In *Proc. IEEE 14th Int. Conf. on Distributed Computing Systems*, 1994.

[7] J. E. B. Moss. Working with persistent objects: to swizzle or not to swizzle. *IEEE Trans. Software Engineering*, 18(8):657–673, Aug. 1992.

[8] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pages 99–112, Jan. 1993.

[9] P. Wilson. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. *ACM Computer Architecture News*, pages 6–13, Jun. 1991.

[10] S. Zhou, S. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, September 1992.

[11] 河野健二, 加藤和彦, and 益田隆司. 分散透明性を与えるプログラム変換系の生成システムの実現. In 日本ソフトウェア科学会 第 *10* 回大会論文集, pages 301–304, 1993.