システムソフトウェアと
オペレーティング・システム
（1 9 9 9． 8． 2）

82－10

RMIにおけるオブジェクト整列化の実行時特化の枠組

河野 健二　益田 隆司

東京大学大学院 理学系研究科 情報科学専攻
E-mail: {kono, masuda}@is.s.u-tokyo.ac.jp

要旨

本稿では，遠隔メソッド起動 (RMI) におけるオブジェクト整列化の最適化手法を提案する．従来のオブジェクト整列化では，送信側でオブジェクトを正準表現に変換し，受信側で正準表現からオブジェクトを再構成していた．そのため，整列化のコストがRMIにおける通信時間の50%前後を占めることが指摘されている．本稿では，送信側の整列化ルーチンを，受信側の計算機環境にあわせて動的に特化する手法を提案する．動的に特化された整列化ルーチンは，受信側で直接利用できるレイアウトにオブジェクトを変換する．そのため，受信側でのオブジェクトの再構成が不要になり，整列化のコストを削減できる．この手法を実装したプロトタイプによる実験では，整列化ルーチンの動的生成は0.7msec程度で実現でき，整列化のコストを44%から67% 程度削減できた．

Dynamic Specialization of Object Serialization in RMI

Kenji Kono　Takashi Masuda

Department of Information Science,
Graduate School of Science,
University of Tokyo
E-mail: {kono, masuda}@is.s.u-tokyo.ac.jp

Abstract

This paper proposes an optimization method of object serialization in remote method invocation (RMI). In conventional approaches to object serialization, a sender transforms objects into the canonical representation and then the receiver reconstructs objects from the canonical representation. The cost of serialization is terrible; serialization takes the 50% time of the round-trip execution of one RMI. This paper presents dynamic specialization of serialization routines. The specialized code transforms objects directly into the memory layout used by the receiver, thereby eliminating the need of reconstruction. Experimental results show that the specialized code reduces the cost of serialization by 44% to 67%. The specialization itself takes only 0.7msec.

1 はじめに

遠隔メソッド起動 (RMI) とは，遠隔オブジェクトのメソッドをローカルなオブジェクトのメソッドと同じシンタックスで呼び出せるようにする技術であり，CORBA [15], DCOM [5], Java RMI [9] など多くの分散システムで実用的に用いられている．

RMI が広く普及した要因のひとつは，RMI が相互運用性 (interoperability) を保証しやすいという性質を持つことにある．相互運用性とは，オブジェクトを提供するシステムで用いられているプロセッサ，オペレーティングシステム (OS) およびプログラミング言語によらず，オブジェクト間でのメソッド起動が可能であることをいう．分散環境を構成する計算機システムが単一のプロセッサ，OS，言語によって構成されていることは稀であり，相互運用性は基盤ソフトウェアが提供すべき重要な性質のひとつである．

RMI の高い相互運用性は，オブジェクトを受け渡す際に，オブジェクトの整列化 (serialize) および再構成 (deserialize) を行なうことによって達成されている．オブジェクトの整列化とは，メモリ上でのオブジェクトの物理的な表現 (representation) を，ある一定の規約にしたがった正準表現 (canonical representation) に変換しつつ，通信用のバッファに詰め込む操作をいう．オブジェクトの再構成とは，正準表現にしたがって詰め込まれた通信用バッファから，メモリ上にオブジェクトを再構築する操作をいう．

しかしながら，多くの文献 [14, 13, 17] で指摘されているように，オブジェクトの整列化・再構成の実行時オーバーヘッドは極めて大きく，通信コストに比べても無視できない．実際，Sun RPC において整列化・再構成を行なう XDR ライブラリ [19] を用いた実験では，整列化・再構成のコストは RPC のラウンドトリップ時間の 47%から 57% を占めている．これは，(1) オブジェクトのメモリ・レイアウトから正準表現への変換と，(2) 正準表現からメモリ上のレイアウトへの変換という，二回の変換を常に行なっているためである．

本稿では，通信先の計算機環境に合わせて，実行時にオブジェクトの整列化ルーチンを特化する手法を提案する．この手法では，遠隔メソッドの名前解決時 (バインディング時) に，サーバの位置情報だけでなく，サーバにおけるオブジェクトのメモリ表現をクライアントに通知する．通知を受けたクライアントは，そのサーバに特化した整列化ルーチンを動的に生成する．特化された整列化ルーチンは，正準表現を介することなく，直接，サーバ上でのメモリ表現にオブジェクトを変換して送信する．これによって，サーバは通信バッファにあるオブジェクトを変換もコピーもすることもなく利用でき，再構成の処理をはぶくことができる．

この手法を用いたオブジェクトの整列化ルーチンと，XDR ライブラリを用いた整列化・再構成ルーチンとの比較実験によれば，本手法は約 1.8 倍から 3.0 倍程度の高速化を達成している．通信時間を含めても，約 1.3 から 1.7 倍程度の性能向上を達成している．なお，整列化ルーチンの動的生成には，IRISA で開発された Tempo [8] というシステムを用いており，そのコストはわずか 0.7msec 程度であった．

以下，2 章では，RMI のメカニズムとオブジェクトの整列化・再構成について説明する．3 章では，動的特化による整列化の最適化の手法を提案する．4 章では，提案方式による整列化・再構成と XDR ライブラリによる整列化・再構成のオーバヘッドを比較した実験の結果を示す．5 章では，関連研究についてまとめ，6 章で本稿をまとめる．

2 RMI のメカニズムと整列化

本章では，広く実用的に利用されている RMI のメカニズムについて概説し，従来の RMI で用いられているオブジェクトの整列化手法の利害得失について論じる．

2.1 プロキシとスケルトン

CORBA や Java RMI を含め，多くの RMI はプロキシとスケルトンと呼ばれるオブジェクトによって通信の詳細を隠蔽する方法をとっている．プロキシとは，遠隔のオブジェクトの代理を果たす通常のオブジェクトであり，遠隔オブジェクトと同一のインターフェースを提供する．スケルトンは，遠隔オブジェクトと同じアドレス空間上に存在し，遠隔メソッド起動の要求を受け付けて，要求されたメソッドを起動する．

遠隔オブジェクトのメソッドを起動するには，プロキシの対応するメソッドを起動すればよい．プロキシは，引数オブジェクトを整列化し，遠隔アドレス空間上のスケルトンにメソッド起動の要求を出す．要求を受け取ったスケルトンは，整列化されたオブジェクトから引数オブジェクトを再構成し，要求されたメソッ

ドを起動する．メソッドの返り値はスケルトンによって整列化され，プロキシに返される．プロキシは整列化された返り値からオブジェクトを再構成する．

クライアントによるプロキシの呼び出しは，通常のメソッド起動と全く同一であり，クライアントは遠隔メソッドをローカルなメソッドと同様に呼び出すことができる．また，スケルトンによるメソッド起動もローカルなメソッド起動と同一であり，結果として，遠隔メソッド自身をローカルなメソッドと同様に実装できる．

プロキシやスケルトンのコードは，遠隔オブジェクトのインターフェースから自動的に生成される．インターフェースからこれらのコードを生成するプログラムをIDLコンパイラと呼ぶ．

2.2 プロキシの生成と消去

スケルトンはサーバのコードの一部であり，サーバが起動されると同時に要求を受け付ける．プロキシは動的に生成・消去されるオブジェクトであり，実行時システムによって，必要に応じて生成・消去される．

プロキシが生成されるのは次の二つの場合である．ひとつは，遠隔オブジェクトにバインディングを行ったときである．ネーミング・サービスを用いてサービス名を解決すると，そのサービスを提供する遠隔オブジェクトの代理であるプロキシが生成される．プロキシが生成されるもう一方のケースは，遠隔メソッドの引数や返り値として，遠隔オブジェクトへの参照を受け取った場合である．その場合，参照されている遠隔オブジェクトの代理となるプロキシが作られ，そのプロキシへの参照を遠隔参照として用いることができる．

プロキシが消去されるのは，そのプロキシへの参照がなくなった場合である．CORBAの実装の一つであるOrbix[1]では，参照カウントを用いてプロキシの消去を行っている．

2.3 オブジェクトの整列化と再構成

分散環境における相互運用性を保証するために，プロキシとスケルトンでは，オブジェクトの整列化と再構成を行なう．遠隔メソッドを起動する場合，プロキシは引数オブジェクトを整列化してから転送し，スケルトンは引数オブジェクトの再構成を行う．

従来のRMIでは，相互運用性を保証するために，プロキシもスケルトンもお互いの計算機環境に関する情報は一切利用せずに整列化・再構成を行っている．プロキシはスケルトンが稼働している計算機環境にかかわらず，常にオブジェクトを正準表現へと変換ながら整列化する．これによって，スケルトンが受け取るオブジェクトは常に正準表現になっていることが保証され，スケルトンはプロキシの計算機環境にかかわらず，正準表現からオブジェクトの再構成を行えばよい．

2.4 オブジェクト整列化の問題点

上記の方式の利点は，次の二つである．ひとつは，どのような計算機環境であっても，正準表現とオブジェクトのメモリ上での表現とを相互に変換するルーチンさえ用意すれば，任意の計算機環境間でオブジェクトの送受信が可能となる点である．もう一点は，新しい計算機環境が追加された場合でも，その環境用の相互変換ルーチンさえ提供すれば，既存のシステムに何ら変更を加えることなしに，新しい計算機環境との相互通信が可能となる点である．

この方式の欠点は，実行時コストが極めて大きい点にある．これは，オブジェクトの正準表現への変換と，正準表現からメモリ上での表現への変換という二度の変換を常に行っていることに起因する．4章で示すように，この変換コストは通信時間全体に対しても47%から57%にも及ぶ．

3 動的特化による最適化

本章では，受信側の計算機環境に合わせて，オブジェクトの整列化ルーチンを動的に特化する手法を提案する．特化された整列化ルーチンは，送信側のオブジェクトを受信側のメモリ・レイアウトへと直接変換する．受信側では，通信バッファをそのままオブジェクトとして利用でき，オブジェクトの再構成の必要がない．新しい計算機環境が追加された場合でも，その環境に特化した整列化ルーチンが動的に生成される．そのため，既存のシステムに変更を加えることなしに，新しい環境との相互通信が可能となっている．

3.1 動的特化の枠組み

従来のRMIでは，サービスのバインディング時には，そのサービスを提供するサーバの位置情報のみが受け渡される．本稿で提案する手法では，サーバの位置情報に加え，そのサーバにおけるオブジェクトのレイアウト情報を受け渡すようにする．メモリ・レイアウ

トを表したこのデータ構造を型記述 (type description) と呼んでいる．型記述はコンパイラによって自動生成でき，文献 [11] で示されているように，既存のコンパイラを変更しなくとも，デバッガの生成する型情報から容易に得られる．

プロキシのコードを生成する IDL コンパイラは，型記述を引数にとる整列化ルーチンを生成する．この整列化ルーチンは，オブジェクトの受信側の型記述にしたがって，送信側のオブジェクトを受信側のメモリ・レイアウトに変換するルーチンである．たとえば，32bit の整数を変換する整列化ルーチンは次のようになる．

```
1: // bufptr は通信バッファへのポインタ
2: //    td は型記述
3: void encode_int32(char* bufptr,
4:                   int* data,
5:                   type_des_t td)
6: {
7:   // 受信側のアラインメントに揃える
8:   bufptr = align(bufptr, td.align);
9:   // 受信側とバイト順が同じならば
10:  if(is_same_endian(td.format)){
11:    // 受信側とアラインメントが合えば
12:    if(align_match(td.align)){
13:      // 直接バッファにコピー
14:      *(int*)bufptr = *data;
15:    } else {
16:      // アラインメントが合わなければ
17:      // バイト単位でコピー
18:      for(int i = 0; i < td.size; ++i){
19:        bufptr[i] = ((char*)data)[i];
20:      }
21:  } else {
22:    // 受信側とバイト順が違ったら
23:    // バイト順を変換する
24:    for(int i = 0; i < td.size; ++i){
25:      bufptr[i] = ((char*)data)[td.size-i-1];
26:    }
27:  }
28:  // 整数のサイズだけポインタを進める
29:  bufptr += td.size;
30: }
```

このルーチンは受信側のメモリ・レイアウトに応じて頻繁に条件分岐を行うため，効率のよいコードとは言えない．たとえば，4096 個の要素を持つ整数の配列を整列化する場合，最初に条件分岐を行えばよく，各要素を整列化するごとに条件分岐を行う必要はない．なお，簡明のため，このコードはエラー処理を省いている．

そこで，受信側のメモリ・レイアウトが分かった時点で，この整列化ルーチンを受信側に合わせて特化しておけば良い．受信側のメモリ・レイアウトが送信側と同じであれば，次のようなコードに特化できる．

```
1: void encode_int32(char* bufptr,
2:                   int* data)
3: {
4:   bufptr = align(bufptr, 4);
5:   *(int*)bufptr = *data;
6:   bufptr += 4;
7: }
```

受信側のメモリ・レイアウトを表す型記述は，サービスのバインド時に送信側に受け渡される．サービスをネームサーバに登録する際，サーバはその型記述を登録し，クライアントはサービスのバインド時にネームサーバからその型記述をもらう．バインド時にプロキシを生成するとき，クライアントは整列化ルーチンの特化を行う．整列化ルーチンの特化は，型記述の内容に従って，定数の伝搬とループの展開を行う．これによって，受信側のメモリ・レイアウトに応じた条件分岐やループのコストが削減され，整列化の効率を向上させることができる．

この方式では，プロキシごとに特化された整列化ルーチンを生成する．プロキシが開放されるとき，そのプロキシ用に生成された整列化ルーチンも一緒に開放される．これによって，使われることのなくなった整列化ルーチンがメモリを圧迫するのを防いでいる．

3.2 型記述

バインディング時に受け渡される型記述は，メソッドの引数や返り値として受け渡されるオブジェクトのメモリ・レイアウトを表す．型記述の表記方法は分散システム全体で合意をとる必要があり，この表記方法は従来の RMI における XDR や CORBA などの規格に相当する．ここでは，本方式を実装したプロトタイプで用いた型記述を示す．

ある型を τ で表すとし，その型記述を $\mathcal{D}(\tau)$ で表す．τ は次のように再帰的に定義される．

$$\tau ::= b \mid \mathit{array}(\tau) \mid \{l_1 : \tau_1, \cdots, l_n : \tau_n\} \mid \mathit{ref}(\tau)$$

ここで，b は整数などの基本データ型を表す．$\mathit{array}(\tau)$ は，τ 型のデータを要素に持つ配列であり，$\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ は各フィールド l_i が τ_i 型を持つオブジェクトを表す．$\mathit{ref}(\tau)$ は τ 型のデータへのリファレ

ンスを表す．現在のプロトタイプではオブジェクトの継承などは考えていない．これは本質的な制約ではなく，単に型記述の対象となる型システムを拡張すればよい．型記述 $\mathcal{D}(\tau)$ は τ の構造にしたがって，次のように再帰的に定義できる．

- $\tau \equiv b$ のとき．
  $\mathcal{D}(b) = (align(b), size(b), format(b))$.
  $align(b)$, $size(b)$, $format(b)$ はそれぞれ，基本データ型 b のアライメント，データ長，データフォーマット (バイト順や浮動小数点の表現方法など) を示す．
- $\tau \equiv array(\tau)$ のとき．
  $\mathcal{D}(array(\tau')) = \mathcal{D}(\tau')$.
- $\tau \equiv \{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ のとき．
  $\mathcal{D}(\{l_1 : \tau_1, \cdots, l_n : \tau_n\}) = [\mathcal{D}(\tau_1), \cdots, \mathcal{D}(\tau_n)]$.
  ここで，$[\cdots]$ は順序付きの集合を表す．
- $\tau \equiv ref(\tau')$ のとき．
  $\mathcal{D}(ref(\tau')) = [(align(r), size(r), format(r)), \mathcal{D}(\tau')]$.
  ここで， $align(r)$, $size(r)$, $format(r)$ はそれぞれ，リファレンス型のアライメント，データ長，データフォーマットを示す．

上記の型記述では，受信側のメモリ・レイアウトにゴミ集めのためのタグなど，型システムには現れない隠れたフィールドがあると，そのレイアウトを表現することができない．この問題に対処するには，オブジェクトの型記述を拡張し，各フィールドの前後にダミーの型記述を含めるようにすればよい．また，コンパイラによっては，オブジェクトのサイズを小さくするために，フィールドの順番を入れ替える場合がある．これにはフィールドごとにフィールド名のマッチングをとりながら整列化すればよい．このような処理を含めても，隠れたフィールドやフィールド順の入れ替えを必要としない場合には，そのような処理を行うコードは動的特化によって取り除かれ，実行時の性能には影響を与えない．

3.3 動的特化の実装

整列化ルーチンの動的な特化には，フランスのIRISAで開発されたC言語用の部分評価器であるTempo Specializer [8] を利用した．Tempoでは定数の畳み込み，ループの展開などを手続き間を越えて行い，元のプログラムと同等のプログラムへと自動的に変換する．Tempoはコンパイル時に部分評価を行うモードに加え，実行時に部分評価を行うモードを提供しており，本研究では後者のモードを用いて整列化ルーチンの動的特化を行った．

Tempoにおける動的特化のメカニズムは文献 [6] で提案されたものである．部分評価器の入力となるソースプログラムを静的に解析し，(1) そのプログラムに相当するアセンブラのテンプレートと，(2) そのテンプレートにパッチをあてるプログラムとを自動生成する．プログラムの実行時には，実行時に判明した引数 (整列化ルーチンの場合は型記述) を用いてテンプレートにパッチをあて，そのアセンブラをアセンブルしてアプリケーションにリンクする．

Tempoによる動的特化の効率は高く，本稿で示した整列化ルーチンの特化に要する時間は0.6msecから0.7msec程度であった．なお，この計測はSolaris2.6の稼働している Sun Enterprise250(UltraSPARC-II 300MHz × 2) 上で行った．

3.4 リファレンスの整列化

オブジェクトの整列化では，引数オブジェクトから参照されているオブジェクトは再帰的に整列化される．多くの言語において，オブジェクトへのリファレンスは仮想アドレスによって実装されている．そのため，整列化を行う時点では，送信側において，受信側のアドレス空間で有効な仮想アドレスを知ることができず，受信側で有効な仮想アドレスへとリファレンスを変換することはできない．

リファレンスの整列化では，仮想アドレスの値そのものには意味がなく，オブジェクト間での参照関係を保てばよい．そこで，オブジェクトを通信バッファへと整列化していく際に，リファレンスはその通信バッファ内でのオフセットへと変換する．整列化では，引数オブジェクトから参照できるすべてのオブジェクトを再帰的に通信バッファに詰めるので，すべてのリファレンスを通信バッファ内でのオフセットに置き換えることができる．

オブジェクトの受信側では，オフセットに置き換えられたリファレンスを仮想アドレスに変換する必要がある．整列化されたオブジェクトを通信バッファに受け取った時点では，すべてのリファレンスは受信側の

通信バッファ内でのオフセットになっている．そこで，すべてのリファレンスに通信バッファの先頭アドレスを加えてやればよい．これによってオフセットを有効な仮想アドレスに変換できる．

なお，引数オブジェクトは一般には任意のグラフ構造であり，二つ以上のリファレンスが同一のオブジェクトを参照していたり，サイクルを持つ場合がある．このような場合に対処するため，オブジェクトの整列化時にはリファレンスとオフセットの対応表を作り，グラフ構造を正しく保存するようにリファレンスとオフセットの置き換えを行っている．

4 実験

3章で示した手法によって，オブジェクト整列化の実行時間をどの程度削減できるかを知るために，簡単な実験を行った．この実験では，提案方式を用いてオブジェクトの整列化を行った場合と，正準表現の一つであるXDRを用いて整列化を行った場合との比較を行った．XDRと環境依存の表現の変換にはSunのXDRライブラリを用いた．

実験では，ノードの大きさが32バイトの完全二分木を用いた．完全二分木の高さは15で総ノード数は32,767個であり，総バイト数は1Mbyteである．各ノードは6個の32bitの整数と2個の32bitのリファレンスを持つ．この二分木をクライアントからサーバに転送し，整列化，転送，再構成にかかる時間を計測した．計測は，同機種間での転送の場合と異機種間での転送の場合で行った．1に実験に用いた計算機環境を示す．通信に用いたネットワークは100Mbpsのイーサネットであり，計算機間はスイッチング・ハブで接続されている．通信に用いたプロトコルはTCP/IPである．

図1に実験結果を示す．図1-(a)には同機種間(SPARC間)での計測結果を示し，図1-(b)には異機種間(SPARC, PentiumII間)での計測結果を示す．それぞれの場合で，(1)XDRライブラリを用いた場合，(2)動的特化をしていない整列化ルーチンの場合，(3)動的特化をした整列化ルーチンの場合，(4)動的特化の時間を(3)に含めた場合の，4つの場合の結果を示した．縦軸は処理にかかった時間で，単位はmsecである．

同機種間の場合も異機種間の場合も，XDRライブラリ版と受信側のメモリ・レイアウトに変換する方式の場合(上記の(2)から(4)の場合)とを比べると，オブジェクトの整列化・再構成にかかる処理時間が短縮されている．XDRライブラリを用いた場合は，クライアント側でオブジェクトのメモリ・イメージからXDRへの変換を行い，サーバ側でXDRからメモリ・イメージへの変換を行うため，オブジェクトの整列化にも再構成にも同程度の処理時間を要している．それに対し，受信側のメモリ・レイアウトに変換して送信する場合は，オブジェクトの再構成に要する処理時間が大きく短縮されている．これは，オブジェクトの再構成には，3.4節で述べたように，単にオフセットからリファレンスへの変換を行えばよいからである．

表 1: 実験環境

	クライアント	サーバ
同機種間	Sun Enterprise 250 UltraSPARCII 300MHz Solaris2.6	JU1/200 UltraSPARC 200MHz Solaris2.5.1
異機種間	同上	PC/AT 互換機 PentiumII 400MHz FreeBSD-2.7.2

表 2: 整列化の処理時間の向上率

	XDR	動的特化なし	動的特化あり
同機種間	1.0	2.2	3.0
異機種間	1.0	1.2	1.8

受信側のメモリ・レイアウトに変換して送信する整列化ルーチンは，3.1節に示したように，受信側のメモリ・レイアウトに応じて多くの条件分岐を必要とする．図1-(a)，図1-(b)のどちらの場合も，動的特化を行っていない整列化ルーチンは，XDRを用いた整列化ルーチンよりも，整列化に多くの処理時間を要している．整列化ルーチンを動的に特化した場合，整列化に要する時間は30%程度短縮され，XDRライブラリを用いた場合と同程度の処理時間で整列化が行えるようになっている．

受信側のメモリ・レイアウトに合わせて整列化する場合，図1-(a)と図1-(b)とを比べると，異機種間での整列化のほうが処理時間が多くかかっている．同機種間ではデータ変換の必要がなく，整数の`store`命令で

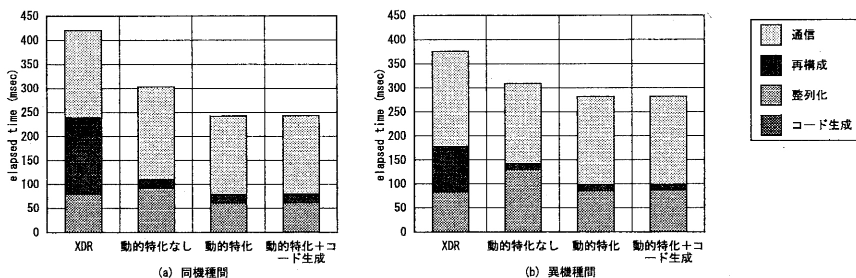

図 1: 提案方式と XDR の実行時間の比較

ノード内の各フィールドのコピーができる．それに対し，異機種間ではデータ変換を行うため，各フィールドをバイト単位でコピーしている．その結果として，異機種間での整列化により時間を要すると考えられる．

表 3: 通信時間を含めた処理時間の向上率

	XDR	動的特化なし	動的特化あり
同機種間	1.0	1.4	1.7
異機種間	1.0	1.2	1.3

動的特化そのものに要する時間は 0.7msec 程度であり，この実験ではほとんど無視できる．この実験では，送信するオブジェクトの総サイズが 1Mbyte 程度であるため，整列化に占めるコード生成の時間は極めて小さい．しかし，オブジェクトの総サイズが数十バイト程度の場合には，コード生成の時間が無視できないコストになることが予測される．しかしながら，動的特化はオブジェクトのバインディング時などに一度行えばよく，RMI を起動するたびに実行する必要はない．

表 2 に提案方式の整列化の処理時間の向上率を示す．XDR ライブラリを用いた場合を 1.0 として，それに対する向上率である．表 3 に通信時間も含めた性能向上率を示す．100Mbps 程度のネットワークでは，通信時間の占める割合が大きく，通信時間を含めた性能向上は 1.3 倍から 1.8 倍程度にとどまっている．ネットワークの性能は著しく向上しており，ギガビットのイーサネットや ATM, Myrinet などのより高性能のネットワークを用いた場合であれば，通信時間を含めても十分な性能向上が期待できる．

5 関連研究

RMI の性能向上のために，これまでにもさまざまな研究がなされている．RMI の基礎となる RPC は文献 [3] で提案され，Sun RPC[20] や OSF/DCE[16] などで広く実用的に用いられている．RMI は RPC をオブジェクト指向の文脈で捉えなおしたものであり，早い時期から Emerald[10]，分散 Smalltalk[2]，分散永続 C++[18] などの分散オブジェクト指向言語に採用されている．

これまでの RPC や R MI の最適化手法の研究は，引数として受け渡すデータがわずか数バイト程度の場合の最適化をねらったものが多い．これは，NFS の場合のように，RPC がプロトコルの実装に使われることが多かったためである．最近になって，CORBA や Java RMI などの普及によって，リファレンスによる参照構造をもったデータを受け渡すことが多くなり，整列化そのものの最適化が求められるようになってきた．

network object[4] では，これまでの RPC や RMI で用いられてきた整列化の手法を系統的に分析し，整列化のアルゴリズムの詳細を示している．また，RMI の往復時間を短縮するために，プロキシのコードを最適化する試みが文献 [7] に報告されている．これは汎用的なコンパイラのために開発されてきた最適化手法を，IDL コンパイラに適用し，より効率的なプロキシの生成を実現したものである．このプロキシの用いる整列化ルーチンに，本稿での提案方式を用いることも可能であると考えられる．

文献 [14] では，XDR ライブラリを静的に部分評価し，XDR ライブラリそのものの高速化を行ったものである．本稿で提案した方式と文献 [14] の方式の性能比較を行ってみると興味深い結果が得られるであろう．

Microsoft research では，文献 [13] に報告されているように，DCOM の整列化ルーチンの最適化を行っている．この方式は同機種の計算機だけで構成された環

境を想定しており，本稿で想定してる環境とは異なっている．また，Javaにおける整列化の最適化手法として，通信メッセージに詰め込むクラス名の短縮を行う方法が，文献 [17] で提案されている．

6 まとめ

本稿では，RMIにおけるオブジェクト整列化を最適化する一手法として，受信側の計算機環境で直接に利用できるメモリ・レイアウトに変換してから，オブジェクトを送信する手法を提案した．この手法では，受信側のメモリ・レイアウトを記述した型記述を実行時に獲得し，その型記述に特化した整列化ルーチンを動的に生成する．動的に特化された整列化ルーチンを用いた場合，従来の整列化ルーチンに比べて，1.8 倍から 3.0 倍の性能向上が得られた．

この手法は，従来の手法が正準表現とメモリ・レイアウトとの相互変換ルーチンさえ用意すればよいのと同様に，型記述から整列化ルーチンを生成するライブラリさえ用意すれば，任意の計算機環境間でのオブジェクトの送受信を可能にする．すなわち，新しい計算機環境がシステムに追加された場合でも，その環境上で動作する整列化ルーチン生成ライブラリさえ用意すれば，相互に通信が可能となる．この方式は，従来の整列化の利点を損なうことなしに，整列化のオーバヘッドを削減する手法と言える．

本稿で示した実装では，汎用的な C 言語を動的に部分評価する Tempo を用いた．整列化ルーチンの行う処理は定型の単純な処理だけであり，汎用の部分評価器を用いる必要はない．整列化ルーチンに用途を限定した言語を設計し，その言語の部分評価器を用いた方が，より効率のよいコードを短時間に生成できることが期待できる．今後，この方式を我々が開発している分散オブジェクトシステム [12, 21] に組み込む予定である．

参考文献

[1] Seán Baker. *CORBA Distributed Objects using Orbix.* ACM Press and Addison-Wesley, 1997.

[2] J. K. Bennett. The design and implementation of distributed smalltalk. In *ACM OOPSLA'87*, pages 318–330, 1987.

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

[4] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *14th ACM SOSP*, pages 217–230, 1993.

[5] D. Chappell. *Active X and OLE.* Microsoft Press, 1996.

[6] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proc. of ACM Symp. on POPL*, pages 145–156, 1996.

[7] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proc. of ACM Conf. on PLDI*, pages 44–56. 1997.

[8] IRISA/INRIA. Tempo specializer. available from `http://www.irisa.fr/compose/tempo/`.

[9] Java Soft. *Java Remote Method Invocation Specification*, 1997. available from http://www.javasoft.com/.

[10] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM TOCS*, 6(1):109–133, 1988.

[11] S. V. Kakkad, M. S. Johnstone, and P. R. Wilson. Portable run-time type description for conventional compilers. In *Proc. of ACM Int'l Symp. on Memory Management*, pages 146–153, 1998.

[12] K. Kono, K. Kato, T. Masuda, An implementation method of migratable distributed objects using an RPC technique integrated with virtual memory management, In *Proc. of ECOOP, LNCS 1098*, pages 295–315, 1996.

[13] L. Li, A. Forin, G. Hunt, and Yi-Min Wang. High-performance distributed objects over a system area network. Technical Report MSR-TR-98-68, Microsoft Research, 1998.

[14] G. Muller, R. Marlet, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proc. of IEEE ICDCS*, pages 240–249, 1998.

[15] OMG. The Comon Object Request Broker: Architecture and specification, 2.0ed. Technical report, OMG, July 1995.

[16] OSF. Remote procedure call in a distributed computing environment: A White Paper, 1991.

[17] L. Opyrchal and A. Prakash. Efficient object serialization in java. In *Proc. of IEEE ICDCS Workshop*, pages 96–101, 1999.

[18] M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In *Proc. of ECOOP)*, pages 191–204, 1989.

[19] Sun Microsystems Inc. *External Data Representation Standard:Protocol Specification*, March 1990.

[20] Sun Microsystems Inc. *rpcgen Programming Guide*, March 1990.

[21] 河野健二，加藤和彦，益田隆司. 自律協調システムのための分散オブジェクトの共有機構. In コンピュータソフトウェア, 16(2):51–55, 1999 年.