オブジェクト整列化の動的特化による効率的なRMIの実現

河野健二[†,††] 益田隆司[†]

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．オブジェクト整列化は，異機種のプラットフォーム間での通信を可能にする技術であり，その高速化はRMIの性能向上に大きく貢献する．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メモリ表現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特化は0.6 msecで実現でき，SunのXDRライブラリを用いた従来の整列化ルーチンより1.9倍から3.0倍程度の性能向上を得ることができた．

Efficient Object Serialization by Dynamic Specialization in RMI

Kenji Kono[†,††] and Takashi Masuda[†]

This paper describes a novel approach to object serialization in remote method invocation (RMI). Object serialization enables an object to be passed between heterogeneous platforms. Efficient serialization is a primary concern in RMI because the conventional approaches incur large runtime overheads. The approach described here specializes a serializing routine dynamically according to the receiver's platform. The specialized routine converts the sender's in-memory representations of objects directly into the receiver's in-memory representations. The receiver can thus access the passed objects without any data copies and data conversions. Experimental results show that our prototype implementation is 1.9–3.0 times faster than Sun XDR, and the time needed for specializing a serializing routine is only 0.6 msec.

1. はじめに

遠隔メソッド起動（RMI）は遠隔手続き呼び出し（RPC）の技術をオブジェクト指向の文脈でとらえなおしたものであり，遠隔のアドレス空間上にある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す技術である．RMIの有用性は広く認識されており，CORBA[11]，DCOM[2]，Java RMI[8]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1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などのプラットフォームによらず，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同一のプラットフォームのみ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1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）☆を行うことによって達成されている[1),6),13)]．オブジェクトの整列化とは，オブジェクトのメモリ表現（in-memory representation）を，ある一定の規約に従った正準表現（canonical representation）に変換しつつ，送信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，受信バッファに詰め込まれた正準表現のオブジェクトをメモリ上に再構成する操作をいう．

正準表現を用いることの利点は2つある．第1点

† 電気通信大学情報工学科
Department of Computer Science, University of Electro-Communications
†† 科学技術振興事業団さきがけ研究21
PRESTO, Japan Science and Technology Corporation

☆ 整列化・再構成をmarshaling・unmarshalingと呼ぶ流儀やpickling・unpicklingと呼ぶ流儀もある．

は，オブジェクトの送信側と受信側とが異なるメモリ表現を用いていても，相互にオブジェクトを受け渡すことができる点である．第 2 点は，新しいプラットフォームが追加された場合であっても，既存のプラットフォームを変更することなしに，相互にオブジェクトを受け渡すことができる点である．これは，新しいプラットフォームが，そのプラットフォーム上でのメモリ表現と正準表現の変換方法のみを知っていればよいからである．

しかしながら，多くの文献[4),9),10)]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ヘッドは大きく，通信コストに比べても無視できない．実際，本稿で示すように，Sun RPC においてオブジェクトの整列化・再構成を行う XDR ライブラリ[14)]を用いると，RMI の往復時間の約 50%がオブジェクトの整列化・再構成に費やされてしまう．

本稿では，プログラム・コードの動的特化の技術を利用し，オブジェクトの整列化・再構成を最適化する手法を提案する．提案手法では，受信側のプラットフォームにあわせて整列化ルーチンを動的に特化する．動的に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，送信側のオブジェクトを受信側のメモリ表現に直接変換する．そのため，受信側では受信バッファ内のオブジェクトをコピーしたり変換したりすることなしに利用でき，再構成のオーバヘッドを削減できる．

また，受信側のプラットフォームに特化された整列化ルーチンは，必要に応じて動的に生成されるため，送信側のプロセスは通信相手となりうるプラットフォームをあらかじめすべて知っている必要はない．そのため，新しいプラットフォームを追加した場合であっても，既存のプラットフォームに変更を加えることなく相互運用性を保証できる．なぜなら，新しいプラットフォームに特化した整列化ルーチンの生成法を既存のプラットフォームに通知するだけでよいからである．

この手法を用いたオブジェクト整列化ルーチンと，XDR ライブラリを用いた整列化ルーチンとの比較実験では，本手法が約 1.9 倍から 3.0 倍の高速化を実現することが示された．なお整列化ルーチンの動的生成には，IRISA で開発された Tempo[7)]という部分評価器を用いており，整列化ルーチンの生成にはわずか 0.6 msec しかかからなかった．

以下，2 章では RMI のメカニズムについて述べ，3 章では動的コード特化を用いた整列化の最適化手法を提案する．4 章では実験結果を報告する．5 章では関連研究について述べ，6 章で本稿をまとめる．

2. RMI のメカニズム

本章では，広く実用的に用いられている RMI のモデルとメカニズムについて説明し，以下の議論で必要となる用語を定義する．

2.1 プロキシ，スケルトン，遠隔オブジェクト

遠隔のアドレス空間上からメソッドを起動できるオブジェクトを**遠隔オブジェクト**と呼ぶ．RMI の技術を用いると，ローカルなオブジェクトのメソッド起動と同じシンタックスで，遠隔オブジェクトのメソッドを起動することができる．遠隔オブジェクトに関しては，遠隔のアドレス空間から起動できるメソッドの名前，引数の型，および返り値の型をインタフェースとしてあらかじめ定義しておく必要がある．このインタフェースを定義する言語をインタフェース記述言語（IDL）と呼ぶ．

CORBA や Java-RMI を含め，RMI ではプロキシとスケルトンと呼ばれる 1 組のオブジェクトによって通信の詳細を隠蔽する．プロキシおよびスケルトンのコードは遠隔オブジェクトのインタフェース記述から静的に生成される．インタフェース記述からこれらのコードを生成する処理系を IDL コンパイラと呼ぶ．**図 1** に示すように，スケルトンは遠隔オブジェクトの存在するアドレス空間と同じアドレス空間上にあり，プロキシはスケルトンとは異なる（多くの場合，遠隔サイト上の）アドレス空間上にある．プロキシは，遠隔オブジェクトの代理を果たすオブジェクトであり，遠隔オブジェクトの持つインタフェースと同一のインタフェースを持つ．遠隔オブジェクトのメソッドを呼び出すには，その遠隔オブジェクトの代理であるプロキシのメソッドを呼び出す．プロキシは引数オブジェクトを整列化し，整列化されたオブジェクトをスケルトンに送る．それを受け取ったスケルトンは，メモリ

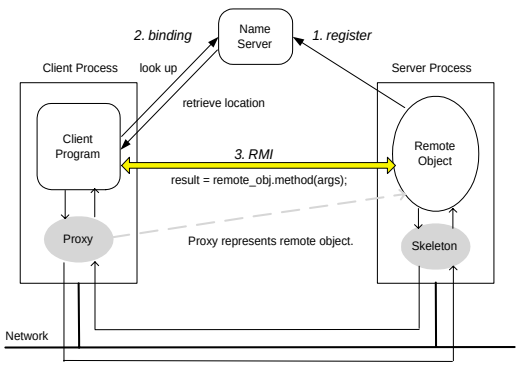


図 1 RMI のメカニズム
Fig. 1 Mechanism of RMI.

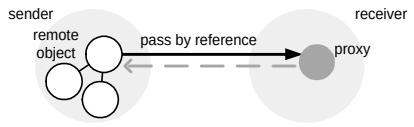
上にオブジェクトを再構成し，遠隔オブジェクトのメソッドを呼び出す．遠隔オブジェクトのメソッドの実行結果も同じように呼び出し側に返される．

遠隔オブジェクトを生成（instantiate）すると，スケルトンもいっしょに生成され，生成された遠隔オブジェクトの名前と位置がネームサーバに登録される．遠隔オブジェクトのメソッドを呼び出す前に，クライアントはバインディングと呼ばれる名前解決の処理を行う．クライアントは，呼び出したい遠隔オブジェクトの位置をネームサーバに問い合わせ，その遠隔オブジェクトを呼び出すためのプロキシを生成する．その後，クライアントはプロキシを経由して遠隔オブジェクトのメソッドを呼び出せるようになる．

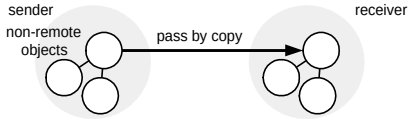
2.2 パラメータ受け渡し

遠隔オブジェクトのメソッドのパラメータ受け渡しのセマンティクスは，参照受け渡し（pass by reference）またはコピー受け渡し（pass by copy）のどちらかである．遠隔オブジェクトは参照受け渡しで受け渡され，遠隔オブジェクトでない通常のオブジェクトはコピー受け渡しで受け渡される．遠隔メソッドの引数や返り値に遠隔オブジェクトが現れると，受信側のプロセスはその遠隔オブジェクトの代理となるプロキシを生成する（**図 2** (a) を参照）．

遠隔オブジェクトでない通常のオブジェクトが遠隔メソッドの引数や返り値に現れると，送信側のプロセスはそのオブジェクトを整列化し，受信側に送信する．受信側ではメモリ上にオブジェクトを再構成する．コピー受け渡しで受け渡されるオブジェクトが他の通常の（遠隔オブジェクトでない）オブジェクトを指すリファレンスを持っていた場合，そのオブジェクトも一緒に整列化される．すなわち，引数または返り値に現れた通常のオブジェクトは再帰的に整列化され，受信側に送信される．この様子を図 2 (b) に示す．なお，整列化されるオブジェクトがプロキシを指すリファレンスを持つ場合，そのプロキシが表す遠隔オブジェクトのプロキシを受信側に生成する．



(a) pass by reference.



(b) pass by copy.

図 2 RMI における引数渡し
Fig. 2 Parameter-passing in RMI.

2.3 オブジェクトの整列化と再構成

2.2 節で述べたように，遠隔メソッドの引数や返り値を整列化するとき，遠隔オブジェクトでない通常のオブジェクトは再帰的に整列化され，送信バッファに詰め込まれる．本節では，Birrell ら[1)]によって提案された整列化の手法を簡単に説明する．この手法では，整列化または再構成されるオブジェクトの型が静的に決まっていることを仮定している．

遠隔メソッドの引数または返り値となっているオブジェクトを整列化する場合，そのオブジェクトの各フィールド（インスタンス変数）を順次，正準表現に変換しながら送信バッファに詰める．オブジェクトを再構成するには，そのオブジェクトを格納するメモリ領域を割り当て，通信バッファからその領域にオブジェクトをコピーする．このとき，オブジェクトの各フィールドを正準表現から受信側のメモリ表現に変換する．

引数または返り値のオブジェクトを再帰的に整列化する場合，1 つのオブジェクトを 2 回以上整列化しないように注意する必要がある．なぜなら，サイクルのあるグラフ構造や木構造ではないグラフ構造のオブジェクトを整列化する場合，1 つのオブジェクトが 2 つ以上のリファレンスによって指されている場合があるからである．

1 つのオブジェクトを 2 回以上整列化することを避けるために，リファレンスの値をキーとするハッシュテーブルを用意する．オブジェクトを再帰的にたどりながら整列化を行うとき，出現したリファレンスに 0 から順に整数値を割り当てていき，ハッシュテーブルにはリファレンスとその整数値との対応を記録しておく．整列化の過程で初めて出現したリファレンスを整列化する場合，この整数値をリファレンスの正準表現として送信バッファに書き込み，そのリファレンスが指すオブジェクトを再帰的に整列化する．このリファレンスが整列化の過程で 2 回以上出現した場合，そのリファレンスの値はすでにハッシュテーブルに登録されている．この場合には，リファレンスに対応する整数値を送信バッファに書き込むだけで，リファーされているオブジェクトを整列化する必要はない．

オブジェクトの再構成を行うルーチンでは，リファレンスの値を格納する配列を用意する．オブジェクトを再構成している過程で，リファレンスの正準表現である整数値に出会うと，そのリファレンスの指すオブ

ジェクトを格納するメモリ領域を割り当て，出会った整数値をインデックスとする配列のエントリにその領域を指すリファレンスの値を格納する．再構成の過程でこの整数値に 2 回以上出会ったときは，その整数値をインデックスとして配列を参照すれば，対応するリファレンスの値を得ることができる．以上の方法によって，サイクルを持つ構造や木ではないグラフ構造であっても正しく整列化・再構成を行うことができる．

3. オブジェクトの整列化の動的特化

本稿で提案する手法では，送信側のメモリ表現を受信側のメモリ表現へと**直接**に変換する．これによって，RMI におけるオブジェクトの整列化・再構成を大幅に簡略化でき，整列化・再構成のオーバヘッドを削減することができる．なぜなら，受信バッファ内のオブジェクトはすでに受信側のメモリ表現になっており，受信側でのオブジェクトの再構成が不要になるからである．

3.1 システム・アーキテクチャ

本稿で提案する方式の特徴は，受信側のメモリ表現に応じて整列化ルーチンを**動的**に特化している点にある．RMI を行う前に，クライアント側とサーバ側とで**レイアウト記述**と呼ばれるデータを交換しておく．レイアウト記述とは，クライアントまたはサーバ上のメモリ表現を記述したデータであり，クライアントはサーバ上のメモリ表現を記述したレイアウト記述を受け取り，サーバはクライアント上のメモリ表現を記述したレイアウト記述を受け取る．

RMI を行うたびにレイアウト記述を交換することを避けるため，遠隔オブジェクトのバインディング時にレイアウト記述を交換するようにしておく．遠隔オブジェクトをネームサーバに登録するとき，遠隔オブジェクトの存在するサーバの位置だけでなく，サーバ上のメモリ表現のレイアウト記述も登録しておく．クライアントがバインディングの処理を行うと，クライアントはサーバの位置に加えてサーバ上のレイアウト記述を受け取る．

バインディングの処理を行った後，受け取ったレイアウト記述に従って，クライアントは RMI の引数をサーバ上のメモリ表現に変換するようにプロキシを動的特化する．その後，クライアント上のメモリ表現を記述したレイアウト記述をサーバに送信する．クライアント上のレイアウト記述を受け取ったサーバは，それに従って，RMI の返り値をクライアント上のメモリ表現に変換するようにスケルトンを動的特化する．**図 3** に提案方式のアーキテクチャを示す．

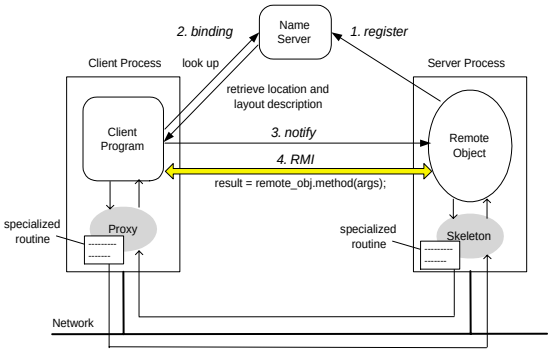


図 3 システム構成
Fig. 3 System architecture.

3.2 レイアウト記述

レイアウト記述とはオブジェクトのメモリ表現を記述したデータ構造である．本稿で提案する方式を実用化するには，XDR [14] が標準化されているのと同様にレイアウト記述を標準化する必要がある．以下，我々のプロトタイプ・システムで用いたレイアウト記述の定義を述べる．

レイアウト記述は RMI の引数や返り値に出現するデータ型ごとに用意する．これらのレイアウト記述はインタフェースの定義から IDL コンパイラによって自動的に生成される．レイアウト記述は言語レベルでのデータ型に応じたネスト構造を持ち，データの型名とレイアウト記述のペアがネストした構造をしている．ネスト構造のベースは**表現子**（representor）と呼ばれるデータであり，ハードウェアレベルでのデータ表現を表す．

ハードウェアでサポートされている整数の表現子は，整数のバイト長，アラインメント，バイト順のタプルである．たとえば，Sparc プロセッサにおけるワードは `(4, 4, BIG)` という表現子で表され，これは 1 ワードが 4 バイト長，4 バイトでアライン，バイト順はビッグエンディアン（big endian）であることを意味する．ハードウェアでサポートされる浮動小数点数の表現子は，浮動小数点数の標準フォーマットのうちどれを用いるのかを指定する．そのため，標準化されていない表現方法を用いたプロセッサでは利用できない．しかしながら，多くのプロセッサでは標準化されたフォーマットを用いており，実用上は特に問題は生じない．

言語レベルでの型を表すレイアウト記述は，型の構造を反映した再帰構造を持つ．整数や浮動小数点数などの，ハードウェアによって直接サポートされている基本データ型の場合，そのレイアウト記述は言語レベルでの型名とその型の表現子のペアで表される．たとえ

ば，Sparc における 32 ビット幅の整数は，[int, (4, 4, BIG)] というレイアウト記述によって表される．

配列型の場合，レイアウト記述は配列の要素のレイアウト記述と要素数とからなる．たとえば，Sparc における要素数 4096 の整数型配列は，[int[4096], [int, (4, 4, BIG)]*4096] によって表される．可変長配列の場合，LENGTH というマクロが配列の要素数に置き換えられる．たとえば，バイトデータの可変長配列は，[char[], [char, (1, 1, -)]*LENGTH] というレイアウト記述によって表される．なお，バイトデータの表現子に現れる '-' は，バイトデータではバイト順は問題にならないことを示している．

オブジェクト型の場合，レイアウト記述はオブジェクトの各フィールドのレイアウト記述を並べたものになる．たとえば，**図 4** に示した Account 型のレイアウト記述は，[Account, person:[Person*, (4, 4, BIG)], number:[int, (4, 4, BIG)], balance:[int, (4, 4, BIG)]] となる．各フィールドのレイアウト記述の前にはそのフィールドの名前が付けられている．レイアウト記述にフィールド名を含めることによって，フィールドの順番がクライアントとサーバで異なる場合にも対応できる．

文字列型の場合，レイアウト記述は文字列の内部構造を明示的に記述する必要がある．**図 5** に C および Pascal における文字列のレイアウト記述を示す．C でも Pascal でも文字列は可変長の配列で実現されているが，文字列の終端を示す方法が異なる．C ではヌル文字によって文字列の終端を示し，Pascal では文字列の長さを表す整数を冒頭に置く．図 5 における記号 '$' は，その直後の値でその直前のフィールドを初期化することを表す．したがって，C の文字列における [char (1, 1, -)]$0 はヌル文字を表し，文字列の最後にヌル文字を置くことを表す．Pascal では [int, (4, 4, BIG)]$LENGTH が文字列の長さを表し，冒頭に文字列長を置くことを表す．

```
1: class Account {
2:   Person *person;
3:   int number, balance;
4: };
5:
6: class Person {
7:   string name, address;
8:   int age;
9: };
```

図 4 例で用いるデータ型
Fig. 4 Example data type.

3.3 動的特化

遠隔オブジェクトのバインディング時にレイアウト記述を受け取ると，そのレイアウト記述に従って整列化ルーチンの動的特化を行う．この動的特化には部分評価と呼ばれる手法を用いた．部分評価とは汎用的なプログラムを入力としてとり，特別な場合に特化されたコードを自動的に生成するプログラム変換である．このような部分評価を行うプログラムを部分評価器と呼ぶ．部分評価器は 2 つの引数をとるプログラム $\mathcal{PE}$ であり，1 つめの引数は特化の対象となる汎用的なプログラム P であり，もう 1 つの引数はプログラム P の引数のうちその値が分かっているもの A である．$\mathcal{PE}$ はプログラム P を引数 A に特化したプログラム P_A を自動的に生成する．すなわち，$\mathcal{PE}(P, A) = P_A$ であり，P の引数のうち A 以外の残りの引数を B とすると，$P(A, B) = P_A(B)$ という関係が成り立つ．

本研究では部分評価器として IRISA で開発された Tempo[7] を用いた．Tempo に付属のドキュメントによれば，部分評価は定数の畳み込み，インライン展開，ループの展開，手続き間での定数伝播をすべてのデータ型（ポインタ，構造体，配列を含む）に対して行うものである．

Tempo は文献 3) によって提案された動的特化の機能を備えている．この方式では，コンパイル時に (1) 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し，実行時の動的特化ではそのテンプレートにパッチをあててアセンブルを行う．4 章で述べる実験では，動的コード生成はわずか 0.6 msec で行うことができた．

本稿で提案する RMI システムでは，次のステップを経て RMI が実行される．最初に，遠隔オブジェクトのインタフェースを IDL を用いて記述する．次に，IDL コンパイラを用いてインタフェース記述をコンパイルし，**汎用的な整列化ルーチン**を生成する．汎用的な整列化ルーチンは，受信側のメモリ表現を表すレイアウト記述を引数にとり，送信側のメモリ表現を受信側の

```
C string:      [string, [char[], [char, (1, 1, -)]*LENGTH], [char, (1, 1, -)]$0]]
Pascal string: [string, [int, (4, 4, BIG)]$LENGTH, [char[], [char, (1, 1, -)]*LENGTH]]
```

図 5 文字列のレイアウト記述の例
Fig. 5 Layout descriptions of strings.

```
1:  struct Account_ld {
2:    struct ref_ld person;
3:    struct int_ld number, balance;
4:  };
5:
6:  struct Person_ld {
7:    struct string_ld name, address;
8:    struct int_ld age;
9:  };
10:
11: serialize_Account(char *buf,
12:     class Account account,
13:     struct Account_ld account_ld,
14:     struct Person_ld person_ld)
15: {
16:   serialize_ref_Person(buf,
17:               account.person,
18:               account_ld.person,
19:               person_ld);
20:   serialize_int(buf,
21:               account.number,
22:               account_ld.number);
23:   serialize_int(buf,
24:               account.balance,
25:               account_ld.balance);
26: }
```

図 6 汎用的な整列化ルーチンの例（Account 型）
Fig. 6 Example of a generic serializer (for Account).

メモリ表現に変換する．この整列化ルーチンは，レイアウト記述さえ変更すれば，様々なメモリ表現にオブジェクトを変換できるという意味で，汎用的なルーチンである．本稿で提案する方式では，部分評価によって，汎用的な整列化ルーチンを受信側のメモリ表現に特化したルーチンへと動的に変換する．

汎用的な整列化ルーチンが部分評価によって動的に特化される様子を示すため，**図 6** に汎用的な整列化ルーチンの例を示す．この汎用的な整列化ルーチンは，図 4 に示した Account 型のオブジェクトを整列化するルーチンである．IDL コンパイラによって Account 型と Person 型のレイアウト記述は，それぞれ Account_ld，Person_ld という構造体にエンコードされる．_ld という接尾子は，その構造体がレイアウト記述（layout description）であることを意味する．serialize_Account() は Account 型の各フィールドを整列化する．このとき，各フィールドを整列化する整列化ルーチンにレイアウト記述を受け渡している．serialize_ref_Person() は，(1) リファレンスそのもののレイアウト記述と，(2) リファーされているオブジェクトのレイアウト記述を引数にとっている（18 行目と 19 行目）．これは，リファレンスを整列化する場合は，リファーされているオブジェクトも整列化されるからである．なお，説明のためエラー処理などは省いた．

```
1:  serialize_int(char* buf,
2:                int data, int_ld ld)
3:  {
4:    buf = align(buf, ld.align);
5:    if(is_same_endian(ld.endian)
6:         && align_match(ld.align)){
7:    /* sender and receiver
8:       use the same layout */
9:       *(int*)buf = data;
10:   } else if(is_same(ld.endian)){
11:   /* receiver uses the same ordering
12:      but different alignment */
13:     for(i = 0; i < ld.size; ++i){
14:       buf[i] = ((char*)&data)[i];
15:     }
16:   } else {
17:   /* receiver uses
18:      different ordering */
19:     for(i = 0; i < ld.size; ++i){
20:       buf[i]=((char*)&data)[ld.size-i-1];
21:     }
22:   }
23:   buf += ld.size;
24: }
```

図 7 汎用的な整列化ルーチンの例（integer 型）
Fig. 7 Example of a generic serializer (for integer).

図 7 に 32 ビットの整数を変換する汎用的な整列化ルーチンを示す．serialize_int() では，引数として渡されたレイアウト記述に従って，受信側のメモリ表現にあわせて 32 ビット整数を変換している．もし送信側と受信側が同じメモリ表現を使用しているならば，9 行目に示したように，何の変換もせずに整数の値をバッファにコピーする．これは機械命令の load と store で実現でき，実行時のオーバヘッドが小さい．もし，送信側と受信側が異なるメモリ表現を用いているならば，13 行目から 15 行目あるいは 19 行目から 21 行目のように，必要に応じてバイト順を変換しながらバイト単位でのコピーを行う．図 7 に示したルーチンは，呼び出されるたびにレイアウト記述を参照しており，効率の良いコードとはいえない．たとえば，要素数 4096 の整数型配列を整列化する場合，同じレイアウト記述を 4096 回参照する．

Tempo を使用して，汎用的な整列化ルーチンを受信側のメモリ表現に特化した整列化ルーチンに変換できる．受信側のレイアウト記述を受け取った時点で，汎用的な整列化ルーチンを部分評価し，レイアウト記述を参照しながら分岐していくコードを削除し，バイトごとにコピーを行うループを展開できる．たとえば，図 7 に示した汎用的な整列化ルーチンは**図 8** に示したコードに動的に特化される．なお，この例では送信

```
1: void serialize_int(char *buf, int data){
2:   buf = align(buf, 4);
3:   buf[0] = ((char*)&data)[3];
4:   buf[1] = ((char*)&data)[2];
5:   buf[2] = ((char*)&data)[1];
6:   buf[3] = ((char*)&data)[0];
7:   buf += 4;
8: }
```

図 8 動的特化されたコードの例
Fig. 8 An example of specialized code.

側と受信側のプラットフォームでは異なったバイト順を用いているものとした．

3.4 リファレンスの変換

本稿で提案する方式では，すべてのオブジェクトが送信側のメモリ表現から受信側のメモリ表現に変換されて送信される．送信されるオブジェクトが他のオブジェクトを指すリファレンスを持つ場合，そのリファレンスも受信側で意味のあるリファレンスに変換する必要がある．多くの場合，リファレンスは仮想アドレスによって実現されているため，送信側の整列化ルーチンは，リファーされているオブジェクトの受信側での仮想アドレスを知っている必要がある．しかしながら，オブジェクトの仮想アドレスはオブジェクトの再構成時に決まるのが普通であり，整列化時にはその仮想アドレスは決まっていない．

この問題は次のようにして解決できる．送信側では，リファレンスを受信側の仮想アドレスに変換するのではなく，リファーされているオブジェクトの送信バッファ内でのオフセットに変換する．リファレンスを整列化するとき，そのリファレンスが指すオブジェクトを詰め込む領域を，送信バッファ内に予約しておく．そして，その領域のオフセットにリファレンスを変換する．オブジェクトの再構成を行うときには，リファレンスに相当するオフセットに受信バッファの先頭アドレスを足しこむ．これによって，リファレンスに相当するオフセットは正しい仮想アドレスに変換され，結果として，送信側でのみ意味のあるリファレンスが受信側でも意味のあるリファレンスに変換される．

リファレンスの整列化・再構成は，次のように実装できる．整列化ルーチンはデータ構造として，(1) 送信バッファ内の空き領域を指すポインタ，(2) リファレンスをキーとして，そのリファレンスに相当するオフセットを返すハッシュテーブルを持つ．これらのデータ構造を用いて，リファレンスの整列化を次のように行う．整列化の過程で最初に出会ったリファレンスの場合，そのリファレンスをバッファ内の空き領域のオフセットに変換し，リファレンスとオフセットの対応をハッシュテーブルに登録する．その後，空き領域を指すポインタを，そのリファレンスの指すオブジェクトのサイズ分だけ進める．整列化の過程で 2 回以上出会ったリファレンスの場合，そのリファレンスに対応するオフセットはすでにハッシュテーブルに登録されている．なお，1 つのオブジェクトを 2 回以上整列化してしまうことを避けるため，すでにハッシュテーブルに登録されているリファレンスはたどらない．

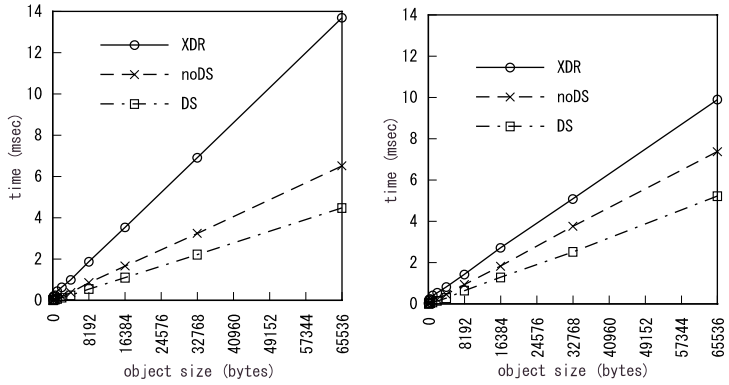
3.5 見えないフィールドの整列化

言語処理系によっては，オブジェクトの中にプログラマからは見えないフィールドを埋め込む場合がある．たとえば，ごみ集めのタグや C++における仮想関数テーブルを指すポインタなどである．多くの場合，オブジェクトに埋め込まれたフィールドの値は，受信側の実行環境に依存しており，送信側の整列化ルーチンでその値を決めることはできない．このような見えないフィールドについては，そのフィールドのサイズ分だけ送信バッファ内の領域をあけておき，受信側でフィールドの値を設定するようにしている．そのため，3.2 節で示したレイアウト記述には，適当な大きさの領域をあけておくことを示すプレース・ホルダを用意している．

4. 実　　験

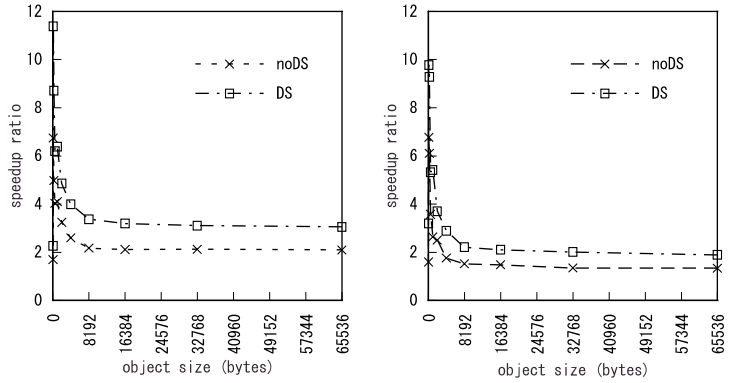
4.1 実験環境

提案方式の有効性を示すため，(1) Sun の XDR ライブラリ[14]を用いた整列化ルーチン（以下，`XDR`），(2) 部分評価による動的特化を行わなかった汎用的な整列化ルーチン（以下，`noDS`），(3) 動的特化を行った整列化ルーチン（以下，`DS`）の 3 つの比較を行った．これらの整列化ルーチンを用いて，同機種間および異機種間での整列化のコストを計測した．同機種間での実験では，サーバ計算機に UltraSparc（200 MHz）のプロセッサ，Solaris 2.5.1 の稼動している計算機を用い，クライアント計算機に UltraSparcII（300 MHz）のプロセッサ，Solaris 2.6 の稼動している計算機を用いた．異機種間での実験では，サーバ計算機に PentiumII（400 MHz）のプロセッサ，FreeBSD-2.2.7 の稼動している計算機を用い，クライアント計算機に UltraSparcII（300 MHz）のプロセッサ，Solaris2.6 の稼動している計算機を用いた．PentiumII と UltraSparcII では，バイト順やアライメントの規則が異なっているため，同じオブジェクトであってもメモリ表現は互いに異なる．同機種環境の場合も異機種環境の場合も，クライアント計算機とサーバ計算機は 100 Mbps のスイッチング・イーサネットで接続し，通信プロトコルには TCP/IP を用いた．



(a) 同機種の環境 (b) 異機種の環境

図 9 整列化および再構成の処理時間
Fig. 9 Time spent in serialization and deserialization.



(a) Homogeneous server and client (b) Heterogeneous server and client

図 10 整列化および再構成の性能向上率（XDR と比較した場合）
Fig. 10 Speedup ratio of serialization and deserialization (speedup ratios are comparisons with XDR).

4.2 ベンチマーク

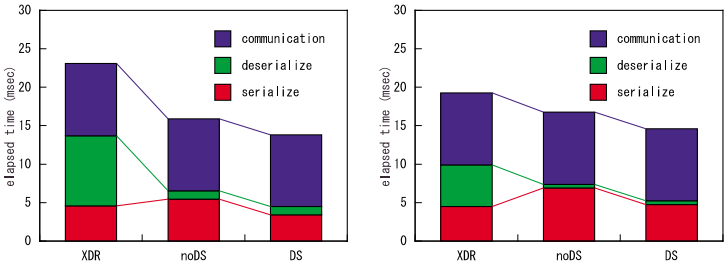
ベンチマークに用いた題材では，クライアント側で完全二分木を整列化しサーバに送る．サーバ側では，受け取った二分木を再構成し，クライアント側に制御を戻す．完全二分木の高さを変えながら，この一連の処理にかかる時間を計測した．この計測時間に動的特化にかかる時間は含まれていない．二分木の各ノードは 4 バイトの整数 6 個と，子のノードを指すリファレンス 2 つからなる．ベンチマークのプログラムは gcc（バージョン 2.7.2）を用いてコンパイルした．最適化オプションは-O2 である．

4.3 実験結果

図 9 に示したように，同機種間の場合も異機種間の場合も，noDS と DS は XDR より高い性能を示している．noDS の処理時間と XDR の処理時間とを比べると，クライアントのメモリ表現を直接サーバのメモリ表現に変換することによって，性能向上が得られたことが分かる．noDS の処理時間と DS の処理時間とを比べると，汎用的な整列化ルーチンを動的特化することによって，さらに性能向上が得られたことが分かる．

図 10 に XDR に対する処理時間の向上率を示す．この図から，noDS は同機種環境では平均 2.0 倍，異機種間では平均 1.35 倍の性能向上が得られたことが分かる．また，DS は同機種間で平均 3.0 倍，異機種間では平均 1.9 倍の性能向上が得られたことが分かる．

1 回の RMI にかかる時間のうち，整列化，再構成および通信にかかる時間を**図 11** に示す．この図に示した処理時間は，高さ 11 の完全二分木の場合である．同機種間の場合も異機種間の場合も，noDS および DS では，再構成にかかる処理時間が XDR に比べて大幅に減少していることが分かる．これは，再構成において，3.4 節で述べたリファレンスの変換以外の処理を



(a) Homogeneous server and client (b) Heterogeneous server and client

図 11 RMI の通信時間のブレークダウン
Fig. 11 Breakdown of time spent in one round-trip RMI.

行う必要がないためである．整列化にかかる時間をみると，同機種の場合も異機種の場合も，noDS は XDR と比べて多くの処理時間がかかっている．これは，noDS で使われている汎用的な整列化ルーチンが，毎回レイアウト記述を参照し，分岐などの処理を行っているためである．noDS の処理時間と DS の処理時間とを比べると，DS の方が整列化の処理時間が短くなっている．これは，動的特化によってレイアウト記述を参照するオーバヘッドが削減されたためである．

5. 関 連 研 究

オブジェクトの整列化の問題は，文献 6) において最初に議論された．Birrell らの network object [1)]は文献 6) の仕事を拡張したものといってよい．network object では，整列化時にはメモリ表現を変換することなく通信バッファにオブジェクトを詰め込み，再構成時に受信側にあわせてオブジェクトのメモリ表現を変換する．このアプローチでは，送信側で 1 回，受信側で 1 回の計 2 回のコピーと，1 回のメモリ表現の変換が必要である．それに対し，本稿で提案する方式では，送信側で 1 回のコピーと 1 回のメモリ表現の変換を行うだけでよい．

USC [12)]は，ユーザ定義された 2 つのメモリ表現間での変換を行う．USC はプロトコルのヘッダを変換することを目的として設計されており，ヘッダの変換に特化した最適化を行うことで，Sun RPC より高速な整列化・再構成ルーチンを生成する．しかしながら，プロトコルのヘッダの変換に特化しているため，浮動小数点数やリファレンスを変換することはできない．

Flick [5)]は，汎用的なプログラミング言語の最適化技術を IDL コンパイラに適用し，高い柔軟性と高度な最適化を行えるようにした IDL コンパイラである．Flick は階層的なコンポーネント群から構成されており，様々な IDL，データのエンコード方式，トランスポート層に柔軟に対応できるようになっている．また，通信バッファの管理の最適化，データコピーの最適化，インライン展開などの最適化を行っている．

USC や Flick で用いられている最適化手法は，本稿の提案方式における汎用的な整列化ルーチンに適用できるものである．これらの最適化を行った汎用的な整列化ルーチンを動的特化すれば，さらに性能の高い整列化ルーチンが生成できると期待される．

文献 10) では，商用の Sun の XDR ライブラリを**静的**に部分評価しても，現在の部分評価技術によって性能向上が得られることが報告されている．この手法では，正準表現を介したデータ変換の最適化を行うだけであり，依然として，整列化・再構成には 2 回のコピーと 2 回のデータ変換を必要としている．

6. ま と め

本稿では，コードの動的特化の手法を利用した新たな整列化手法の提案を行った．本稿の提案方式では，受信側のメモリ表現にあわせて送信側の整列化ルーチンを動的特化し，送信側の整列化ルーチンが受信側のメモリ表現へと直接にオブジェクトを変換する．そのため，送信側で 1 回のコピーと 1 回のデータ変換を必要とするだけであり，整列化・再構成のオーバヘッドを低くすることができる．実験の結果によれば，Sun の XDR ライブラリに比べ，約 1.3 倍から 3.0 倍の性能向上が得られた．また，コードの動的特化は約 0.6 msec で実現できた．

本稿で述べた実装では，動的特化を行うために C 言語用の汎用的な部分評価器である Tempo を用いた．こうした汎用的な部分評価器を用いる代わりに，オブジェクトの整列化に特化された言語を設計し，その言語用の部分評価器を用いれば，高い最適化の行われた整列化ルーチンをより短時間に生成できると期待できる．

参 考 文 献

1) Birrell, A., Nelson, G., Owicki, S. and Wobber, E.: Network Objects, *Software Practice and Expericence*, Vol.25, No.S4, pp.87–103 (1995).
2) Chappell, D.: *Active X and OLE*, Microsoft Press (1996).
3) Consel, C. and Noel, F.: A general approach for runtime specialization and its application to C, *Proc. ACM Symposium on Principles of Programming Languages*, pp.145–156 (1996).
4) da Silva, M.M., Atokinson, M. and Black, A.P.: Semantics for Parameter Passing in a Type-Complete Persistent RPC, *Proc. IEEE 16th Int. Conf. on Distributed Computing Systems*, pp.411–418 (1996).
5) Eide, E., Frei, K., Ford, B., Lepreau, J. and Lindstrom, G.: Flick: A Flexible, Optimizing IDL Compiler, *Proc. ACM Conf. on Programming Language Design and Implementations*, pp.44–56, ACM (1997).
6) Herlihy, M. and Liskov, B.: A Value Transmission Method for Abstract Data Types, *ACM Trans. Programming Languages and Systems*, Vol.4, No.4, pp.527–551 (1982).
7) IRISA/INRIA: Tempo Specializer. available from `http://www.irisa.fr/compose/tempo/`.
8) Java Soft: *Java Remote Method Invocation Specification* (1997). available from http://www.javasoft.com/.
9) Li, L., Forin, A., Hunt, G. and Wang, Y.-M.: High-Performance Distributed Objects over a System Area Network, Technical Report MSR-TR-98-68, Microsoft Research, Microsoft Corporation (1998).
10) Muller, G., Marlet, R., Pu, C. and Goel, A.: Fast, Optimized Sun RPC Using Automatic Program Specialization, *Proc. IEEE 18th Int. Conf. on Distributed Computing Systems*, pp.240–249 (1998).
11) Object Management Group: The Comon Object Request Broker: Architecture and specification, 2.0ed., Technical Report, Object Management Group (1995).
12) O'Malley, S., Proebsting, T. and Montz, A.B.: USC: A Universal Stub Compiler, *Proc. ACM Conference on Communications Architectures, Protocols, and Applications* (*SIGCOMM*), pp.295–306 (1994).
13) Queinnec, C.: Marshaling/unmarshaling as a compilation/interpretation process, Technical Report, Research Report LIP6/1998/049, LIP6 (1998).
14) Sun Microsystems Inc: *External Data Representation Standard: Protocol Specification* (1990).

(平成 12 年 1 月 27 日受付)
(平成 12 年 9 月 7 日採録)

河野 健二（正会員）

1970 年生．1997 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．2000 年 1 月から電気通信大学情報工学科助手，現在に至る．1999 年 10 月から科学技術振興事業団さきがけ研究 21「情報と知」領域・研究員を兼務．オペレーティングシステム等のシステムソフトウェア，分散および並列処理に興味を持つ．理学博士．平成 11 年度情報処理学会論文賞受賞．

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．2000 年 4 月から電気通信大学情報工学科教授．専門はオペレーティングシステム．