システムソフトウェアと
オペレーティング・システム 85−11
（２０００．８．４）

*Java RMI*のための効率の良いオブジェクトシリアライゼーション

前田 俊行[1]　河野 健二[2,3]　益田 隆司[2]

[1] 東京大学 大学院理学系研究科 情報科学専攻
[2] 電気通信大学 情報工学科
[3] 科学技術振興事業団 さきがけ研究 21

E-mail: tosh@is.s.u-tokyo.ac.jp, {kono, masuda}@cs.uec.ac.jp

要旨

オブジェクトシリアライゼーションはJava RMIでのオブジェクトの受け渡しに用いられるためその性能はJava RMIの性能に影響する．しかし現在のオブジェクトシリアライゼーションの方法は性能の点から見て２つ問題がある．第１の問題点は異機種間での相互運用性を保つためにオブジェクトを１度正規表現に変換している点である．第２の問題点はオブジェクトのクラス多相性を保つために実行時のクラス情報の解析が多く行われている点である．我々はこれらの問題を解決する手法として，受信側のメモリ表現にもとづく動的特化とクラス情報にもとづく動的特化の２つの動的特化を用いる手法を提案する．我々の提案する手法による実装は従来の実装に比べてシリアライゼーションに要する時間を約12%から17%短縮した．

Efficient Object Serialization for Java RMI

Toshiyuki Maeda[1]　Kenji Kono[2,3]　Takashi Masuda[2]

[1]Department of Information Science, Faculty of Science, University of Tokyo
[2]Department of Computer Science, University of Electro-Communications
[3]PRESTO, Japan Science and Technology Corporation

E-mail: tosh@is.s.u-tokyo.ac.jp, {kono, masuda}@cs.uec.ac.jp

Abstract

In Java RMI, object parameters are passed by object serialization. It is known that the serialization performance dominates total performance of Java RMI. However, the conventional approach of object serialization has two problems about efficiency. First, in order to preserve interoperability between heterogenous platforms, it converts object parameters to canonical representations and restores them from canonical representations. Second, in order to preserve class polymorphism of Java objects, it analyses class information at runtime more than necessary. We solve these problems by two-step dynamic specializations. The first step specializes serialization routines according to a receiver's memory representations and the second step specializes the serialization routines according to class information. The implementation of our approach was about 12%-17% faster than the implementation of the conventional approach.

1 序論

Java RMI [1] とは，異なる Java 仮想機械に存在す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ことを可能にする技術であり，Enterprise Java Beans [2] や Jini [3] などの Java による分散システムの基盤として広く用いられている．

分散システムの実現に Java RMI を用いる利点は，Java RMI が相互運用性を持ち，異なるプラットフォーム間でもオブジェクトの受け渡しを可能にしている点にある．Java 仮想機械の仕様 [4] では，オブジェクトのメモリ上でのフォーマットは規定されておらず，それぞれのプラットフォームに特化されたメモリ・フォーマットを用いることができる．例えば，バイト順やパディングなど，最適なフォーマットがプロセッサによって規定される場合，そのプロセッサに最適なものを用いてよい．

このように，Java 仮想機械ごとにオブジェクトのフォーマットが異なるため，RMI の相互運用性を保証するためには，それぞれの仮想機械にあわせて，RMI の引数となるオブジェクトのフォーマットを変換しなければならない．この変換のことをシリアライズといい，Java はシリアライズのための汎用的なライブラリ [5] を備えている．RMI の相互運用性を保証するため，Java RMI はこの汎用的なライブラリを用いて実装されている．

しかしながら，シリアライズ処理は多くのメモリ参照を必要とし，その実行時コストは極めて大きい．Maassen ら [6] の実験によると，RMI に要する時間の内，シリアライズ処理に要する時間は約 27%から 37%を占めている．このようにシリアライズのオーバーヘッドが極めて大きくなる理由は次の 2 点にある．

第 1 点は，異機種間での相互運用性を保証するためのオブジェクト・フォーマットの変換によるオーバーヘッドが大きい点にある．これは，送信側ではオブジェクトを送信側のメモリ・フォーマットから，プラットフォームに依存しない正規表現に変換し，受信側ではオブジェクトを正規表現から受信側のメモリ・フォーマットに変換しており，オブジェクトを送受信するたびに，2 回の変換が行われているからである．

第 2 点は，Java RMI がクラス多相性を持つことにある．クラス多相性とは，クラス間の継承関係を許すことにより，オブジェクトが複数の異なるクラスを持つことである．そのため，オブジェクトのシリアライズ時には，オブジェクトの実際の型を知ることができず，シリアライズ時にオブジェクトのクラス情報の解析を行う必要がある．従来の手法では，オブジェクトをシリアライズする度にクラス情報の解析を行っており，そのオーバーヘッドが無視できないものになっている．

本稿では，動的コード特化の手法を用いることによりこれらの問題を回避し，Java RMI を高速化できることを示す．動的コード特化とは，実行時に得られる情報を用いて動的にコードの最適化を行うことを言う．動的特化を用いると，次のように RMI の最適化を行うことができる．第 1 の問題点であるメモリ・フォーマット変換によるオーバーヘッドを削減するには，受信側のメモリ・フォーマットに合わせて送信側のシリアライズ・ルーチンの動的特化を行えばよい．送信側のメモリ表現を直接に受信側のメモリ表現に変換し，正規表現を介することなくメモリ・フォーマットの変換を行うことができる．そのため，オブジェクトのフォーマット変換は送信側での 1 回のみで済み，RMI の実行時オーバーヘッドを削減することができる．この方法は既に著者らが文献 [7] [8] で提案した手法を Java に適用したものである．

第 2 の問題点であるシリアライズ時のクラス情報解析によるオーバヘッドを削減するために，オブジェクトのクラス情報にもとづいてシリアライズおよびデシリアライズ・ルーチンを動的特化することにより，クラス情報の解析を 1 度だけ行えばよいようにした．これによって，実行時のクラス情報解析のオーバーヘッドを削減することができた．

この提案手法を，UltraSPARC 上で動作する Java 仮想機械 Sun JDK1.2.1 に対して，Java 仮想機械を変更しないという方針でネイティブコードを用いて実装を行った．提案手法は従来の手法と比べ，シリアライズに要する時間を約 12%から 17%短縮できることを示した．

以下，第 2 章では Java における従来のシリアライズ手法を述べる．第 3 章では提案方式を述べ，第 4 章では提案方式の実装について述べる．第 5 章では性能比較実験の結果を報告する．第 6 章では関連研究について述べ，最後に第 7 章で結論を述べる．

2 Javaにおける従来のシリアライズ手法

2.1 メモリ・フォーマットの変換

Java仮想機械の仕様 [4] ではオブジェクトのメモリ上でのフォーマットは規定されておらず，プラットフォームに特化したフォーマットを用いることができるため，異なるプラットフォーム上のJava仮想機械間ではバイト順やパディングなどのメモリ表現の違いによって，オブジェクトのフォーマットが異なる．

このため異なるプラットフォーム間では，送信側が直接メモリ上のオブジェクトを送信することはできない．何故ならフォーマットが受信側のオブジェクトのフォーマットと異なるためである．

このため従来のシリアライズ手法では，プラットフォーム独立なメモリ表現である**正規表現**を用いる．送信側は，オブジェクトのクラスの情報を用いて，オブジェクトの各フィールドを正規表現に変換し，受信側では逆に正規表現から受信側のメモリフォーマットに変換する．

このように従来の方法では，正規表現との2回の相互変換を行うことにより異なるプラットフォーム間での相互運用性を実現している．

2.2 シリアライズ時のクラス情報の解析

2.1節で述べたように，シリアライズを行うためにはオブジェクトのクラスの情報にもとづいてオブジェクトの各フィールドを正規表現に変換しなければならない．

しかしながら，Javaのオブジェクトはクラス多相性を持つため，オブジェクトのクラスはオブジェクトが実際にシリアライズされるまでわからない．なお，クラス多相性とは，クラス間の継承関係を許すことによりオブジェクトが複数の異なるクラスを持つことである．

このため従来のオブジェクトシリアライズ手法では，例えば図1に示したように，シリアライズされるオブジェクトについて，オブジェクトの型情報を解析しながらシリアライズを行っている．図1では，2行めと4行めでオブジェクトのクラスの情報を取得し，6行めから18行めで，オブジェクトの各フィールドの型を調べ，その型に応じて別々のシリアライズルーチンを呼び出している．8行目にあるように，もしboolean型であればboolean型用のシリアライズルーチンを呼び出し，12行めにあるようにdouble型ならdouble型用のシリアライズルーチンを呼び出す．このように，シリアライズ時にオブジェクトのクラス情報を解析し，オブジェクトのクラス多相性を実現しているため，このクラス解析に要する実行時オーバーヘッドが大きい．

```
1: void serialize_object(Object object) {
2:    Class clazz = get_class(object);
3:
4:    FieldInfo[] fields = get_fields(clazz);
5:
6:    for (int i = 0; i < fields.length; i++) {
7:       if (fields[i] is boolean type) {
8:          serialize fields[i] as boolean
9:       } else
11:      .....
12:      if (fields[i] is double type) {
13:         serialize fields[i] as double
14:      } else
15:      if (fields[i] is Object type) {
16:         serialize fields[i] as Object
17:      }
18:   }
19:}
```

図 1: Javaにおけるシリアライズルーチン

3 提案方式

本節では，前節で述べた従来方式の2つの問題点をそれぞれ解決する方法を提案する．提案方式の鍵は2段階でシリアライズルーチンを動的特化し，シリアライズのオーバーヘッドを削減しているところにある．

第1段階のシリアライズでは，受信側のメモリ表現に特化したシリアライズルーチンを生成する．生成されたシリアライズルーチンは，送信側のメモリ表現を受信側のメモリ表現に直接に変換する．これにより，送信側から受信側へのメモリ表現の直接変換を実現し，正規表現との相互変換によるオーバーヘッドが削減できる．

第2段階のシリアライズでは，シリアライズおよびデシリアライズルーチンをクラス情報にもとづいて動的特化を行う．動的特化されたシリアライズルーチンはあるクラスに特化されており，そのクラスのシリアライズに必要なクラス情報解析を行う必要の

ないものになっている．これにより実行時のクラス情報解析のオーバーヘッドを削減できる．

これらの動的特化はそれぞれ，クライアントと遠隔オブジェクトがバインドされたとき，実際にオブジェクトシリアライゼーションが行われているとき，に行われる．以下ではそれぞれの場合について述べる．

3.1 第1段階の動的特化

第1段階の動的特化では，RMIのクライアントと遠隔オブジェクトがバインドされたとき，受信側のメモリ表現に合わせて送信側のシリアライズルーチンを動的に特化する．バインド時に送信側と受信側の仮想機械で使われているメモリ表現を記述したメモリ表現記述子を受け渡ようにする．メモリ表現記述子は，それぞれの仮想機械で用いられているバイト順やアラインメント規則を記述したバイト列である．このメモリ表現記述子を受け取ると，そのメモリ表現記述子の内容に従ってシリアライズルーチンを動的に特化する．その結果生成されるシリアライズルーチンは，送信側のメモリ表現を受信側のメモリ表現に直接変換するルーチンとなっている．なお，提案方式におけるバインド時の処理を図2に示す．

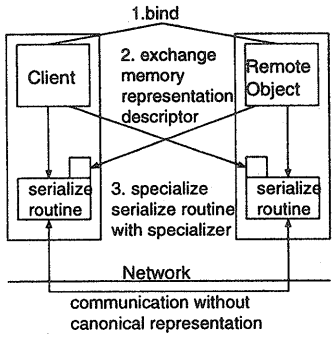

図2: バインド時の処理

動的特化前のシリアライズルーチンは，例えば，int型フィールドのシリアライズルーチンは図3のようになり，まず3，4行めのように，シリアライズするint型の値と受信側のメモリ表現を引数として受取り，9，10行めの**isSameByteOrder**と**isSameAlign**で送信側と受信側のメモリ表現が等しいかどうかを調べ，もし等しければ11行めのように，プラットフォーム固有のCPU命令を用いてバッファに値を直接書き込む．もし送信側と受信側でバイトオーダーが異なっていたら，14行めから17行めのように，値を逆順にバイトごとにバッファに書き込み，バイトアラインが異なっていたら，20行めから23行めのように，値をバイトごとにバッファに書き込む．このようにして受信側のメモリ表現にあわせてint型の値をシリアライズする．

```
1: char* buffer;
2: void serialize_int
3: (int data,
4: MemoryRepresentationDesc mrd)
5: {
6:     /* Align by receiver's align */
7:     buffer = ALIGN(buffer, mrd);
8:
9:     if ( isSameByteOrder(mrd) &&
10:         isSameAlign(mrd) ) {
11:        *(int*)buffer = data;
12:    } else
13:    if ( !isSameByteOrder(mrd) ) {
14:        buffer[0] = ((char*)&data)[3];
15:        buffer[1] = ((char*)&data)[2];
16:        buffer[2] = ((char*)&data)[1];
17:        buffer[3] = ((char*)&data)[0];
18:    } else
19:    if ( !isSameAlign(mrd) ) {
20:        buffer[0] = ((char*)&data)[0];
21:        buffer[1] = ((char*)&data)[1];
22:        buffer[2] = ((char*)&data)[2];
23:        buffer[3] = ((char*)&data)[3];
24:    }
25:
26:    buffer += 4;
27: }
```

図3: 動的特化前のint型用シリアライズルーチン

ところが，このシリアライズルーチンは効率が悪い．何故ならシリアライズごとに受信側のメモリ表現をチェックしているからである．たとえば，4096個のint型配列をシリアライズする場合，4096回のメモリ表現記述子のチェックが行われてしまう．

メモリ表現記述子のチェックを削除するため，シリアライズルーチンを受信側のメモリ表現記述子にもとづいて部分評価を行う．この部分評価によって，メモリ表現記述子のチェックを省くことができる，例えば，図3のシリアライズルーチンは，受信側のメモリ表現が送信側と同じであれば，動的特化によってメモリ表現のチェック（**isSameByteOrder**や**isSameAlign**）が削除でき，図4に示したコードに特化できる．

図4のシリアライズルーチンではメモリ表現のチ

```
char* buffer;
void serialize_int(int data)
{
    /* Align by receiver's align */
    buffer = ALIGN(buffer);

    *(int*)buffer = data;

    buffer += 4;
}
```

図 4: 動的特化後の int 型用シリアライズルーチン

ェックが行われていないため，図 4 のシリアライズルーチンは図 3 のシリアライズルーチンよりも効率がよい．このように，受信側のメモリ表現にもとづく動的特化により，正規表現との相互変換を削除し，より効率的なシリアライズルーチンを得ることができる．

3.2 第 2 段階の動的特化

オブジェクトのシリアライズ時に必要となるクラス情報の解析を避けるため，提案方式では各クラスに特化したシリアライズルーチンを動的に生成している．これにより，各クラスについて初出時にのみクラス情報解析を行えばよく，実行時のクラス情報解析が削減できる．これは，以下のような手順で行われる．

1. シリアライズされるオブジェクトのクラスが初出であるかどうかを調べる．
2. 初出のクラスであれば，クラス情報の解析を行い，そのクラスに特化したシリアライズルーチンを動的に生成する．
3. 既出のクラスであれば，そのクラス用に特化済のシリアライズルーチンを用いる．

なお，上記の手法はデシリアライズルーチンについても同様に適用できる．

従来のシリアライズルーチンでは，図 5 に示したクラスのオブジェクトをシリアライズする場合であっても，図 1 に示したシリアライズルーチンによって，シリアライズされるごとにクラス情報の解析が必要となる．

このシリアライズルーチンを，あるクラスのクラス情報に基づいて，部分評価すると，そのクラスに特化したシリアライズルーチンを得ることができる．たとえば，図 5 に示したクラスのクラス情報に基づいて特化を行う場合，クラス Sample が 2 つのフィールドを持つため，図 1 中のループは 2 回まわればよく，各ループでは int フィールドと double フィールドをシリアライズすればよいことがわかる．そのため，動的特化されたコードは，図 6 に示したようになる．不要な条件分岐やループが削除されているため，図 6 のシリアライズルーチンは図 1 のシリアライズルーチンよりも効率がよい．

```
public class Sample {
    private int       intField;
    private double doubleField;
}
```

図 5: シリアライズされるクラスの例 (サンプルクラス)

```
void serialize_Sample(Object obj) {

    serialize obj.intField       as int
    serialize obj.doubleField as double

}
```

図 6: サンプルクラスに動的特化されたシリアライズルーチン

図 6 に示したシリアライズルーチンを用いると，Sample クラスのオブジェクトのシリアライズ時にはクラス情報の解析が省略できる．従って，クラス情報による動的特化によって，実行時のクラス情報の解析を削減することができる．

4 実装

我々は提案手法を Java 仮想機械を一切変更しないという方針で実装を行った．

図 7 に実装の概要図を示す．

Java では，メモリ・アクセスがバイト単位でしか行えないという制約があるため，メモリを int 型や long 型等としてプラットフォーム固有の CPU 命令を用いて直接操作する必要がある我々の提案手法は実現できない．そのため，Java のラッパー部分を除

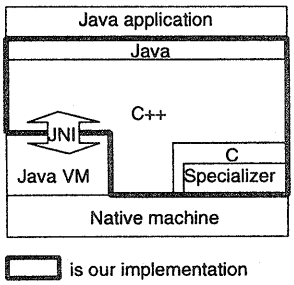

図 7: 実装の概要図

いて全てネイティブコードC++で実装を行った．ただし動的特化器とのインターフェイス部分はCで書いた．

またネイティブコードからJavaのオブジェクトのフィールドへアクセスする方法として，JNI(Java NativeInterface) [9] を用いた．既に文献 [10] などで指摘されているように，JNIは間接関数呼び出しで実現されており，実行時のオーバーヘッドが大きい．しかしながら，JNIはネイティブコードからJava仮想機械を操作するための現在唯一の標準的な方法であり，我々もこれを利用した．Javaのオブジェクトのフィールドを直接操作できれば，提案方式の効率を更に改善できると考えられる．

また，動的特化を行う動的特化器には，INRIAで開発されたC言語用の特化器であるTempo specializer [11] を用いた．ただし，Tempoが生成する動的特化器はJavaのネイティブメソッドとして利用することができなかったため，現在の実装では，実行前にルーチンを特化し，それを実行時に呼び出すことにし，動的特化を仮想的に実現している．これは本質的な制限ではなく，単にJavaのネイティブメソッドが共有オブジェクト形式でなければならないにも関わらず，Tempoの生成する動的特化器が共有オブジェクト形式でなかっただけのことである．

なお実装を行ったJava仮想機械，利用したコンパイラ，動的特化器は表1の通りである．

Java仮想機械	Sun JDK1.2.1 Production Release
C++コンパイラ	g++-2.95.2
Cコンパイラ	gcc-2.95.2
動的特化器	Tempo-1.194

表 1: 実装に用いたツール

5 実験

前節で述べた我々の提案する手法の実装と従来の実装の性能を比較するためシリアライゼーションに要する時間を測定する実験を行った．

比較対象とする従来の実装としてはJDKそのものではなく，従来のJDKのアルゴリズムをネイティブコードで実装したものを用意した．これは我々の実装がネイティブコードで書かれていることに起因する性能改善の影響を排除するためである．(JDKの実装の大部分はJavaで書かれている)

また実験では，シリアライゼーションの対象とするオブジェクトとしてツリー構造のオブジェクトを用いた．ツリーの各ノードはintフィールドを2つとdoubleフィールドを2つを持ち，また子ノードへの参照を2つ持つようにした．ノードオブジェクト1つのサイズは32バイトとなる．

実験では高さ1から11までのツリー構造のオブジェクト (32バイトから65504バイトまで) について，シリアライズ側ではオブジェクトをメモリバッファに書き込むまでの時間，デシリアライズ側ではメモリバッファからオブジェクトを復元するまでの時間を計測し，それらを合計した値をシリアライゼーションに要した時間とした．

また，実験を行った環境はシリアライズ側，デシリアライズ側ともに表2の通りである．つまり今回は送信側と受信側のメモリ表現が同じ場合についてのみ実験した．

CPU	UltraSPARC 168MHz x 14
Memory size	1.2 Gbytes
Operating System	Solaris 2.6(SunOS 5.6)
Java仮想機械	Sun JDK1.2.1 Production Release

表 2: 実験環境

実験の結果はサイズが32バイトから992バイトまでのオブジェクトについては図8，2016バイトから65504バイトまでのオブジェクトについては図9のようになった．図8，図9において**JDK**は従来のJDKのアルゴリズムをネイティブコードで実装した

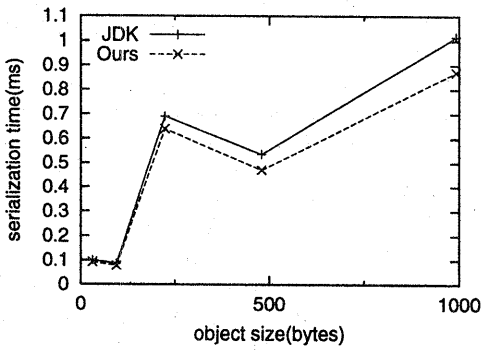

図 8: 性能比較実験結果 (サイズ 32 バイトから 992 バイト)

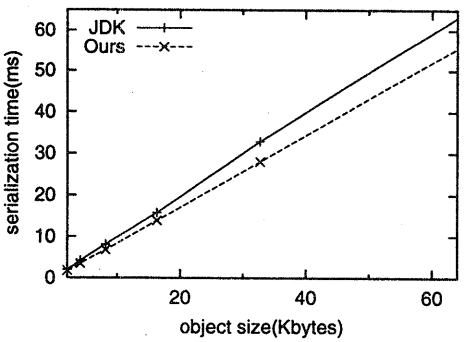

図 9: 性能比較実験結果 (サイズ 2016 バイトから 65504 バイト)

もの，Ours は我々の手法の実装を意味している．

なお，図 8 においてオブジェクトのサイズが 32 バイトと 224 バイトの場合にシリアライゼーションに要した時間が他の場合にくらべ多くなっているが，この原因は不明である．

実験の結果より，我々の手法の実装は従来のアルゴリズムの実装に比べてシリアライゼーションに要する時間を約 12%から 17%短縮することがわかった．

ところで，我々は既に [7] で，Sun RPC や CORBA [12] など，C 言語をベースとした RMI におけるオブジェクトシリアライゼーションに関して，動的特化を用いて正規表現との相互変換を削除する手法を実装し，従来の実装との比較実験を行っている．その結果によると，シリアライゼーションに要する時間は約 47%から 67%短縮できた．

その短縮率と比較すると本研究での短縮率 (約 12%から 17%) はあまりよくない．これは本研究の実装において，オブジェクトのフィールドの操作やオブジェクトのクラスを得る操作などに用いている JNI が大きなオーバーヘッドとなっているためと考えられる．

6 関連研究

Manta [6] は RMI に特化した独自のネイティブコードを用いて高速な RMI，オブジェクトシリアライゼーションを実現した Java 仮想機械である．しかしこの方法は Java 仮想機械をつくり直さなければならず，また独自のネイティブコードに大きく依存している．

Opyrchal [13] らは，我々が変換操作の効率化を行ったのに対し，オブジェクトのサイズを減少させることによりシリアライゼーションを効率よく行った．彼らは，シリアライズされたオブジェクトに含まれるクラス情報を削減することでオブジェクトのサイズを約 50%削減した．

Jaguar project [10] では，正規表現に変換したオブジェクト (これを pre-serialized object と呼ぶ) を Java 仮想機械から直接操作できるように仮想機械を改造した．これによりシリアライゼーション時の変換を完全に省略することができた．しかしこの方法は Java 仮想機械をつくり直す必要があり，また正規表現との相互変換が通常の実行時に行われるので通常の Java プログラムの実行効率を悪くする可能性が

ある.

Braux [14] らは動的特化の手法を用いて，Java の reflection 操作 (オブジェクトのクラスを得るなどの操作) を動的特化し，実行時のクラス情報の解析を削減することによって，オブジェクトのシリアライズに要する時間を約 20%から約 38%短縮した.

彼らの方法は，実行時のクラス情報の解析を削減したのみで，シリアライズにおけるオブジェクト変換は，従来の手法と変わらず正規表現を介して行われる.

正規表現を介しているにも関わらず，シリアライズ効率の改善の割合が我々の手法よりも彼らの手法の方が大きいのは，我々が変換操作に関係する実行時クラス情報解析のみを削減したのに対し，彼らはシリアライゼーション全体における実行時クラス解析を削減したためである.

7 結論

我々は Java RMI におけるオブジェクトシリアライゼーションを効率よく行う方法として動的特化を用いる手法を提案した.

動的特化の手法を用いることによりオブジェクトの送信者はオブジェクトを受信者のメモリ表現へ直接変換することが可能となりオブジェクトの正規表現への変換を省略することができた. また動的特化の手法により実行時のクラス情報の解析を削減することができた.

我々の提案した手法の実装は従来のアルゴリズムの実装と比べてシリアライゼーションに要する時間を約 12%から 17%短縮した. これにより我々の提案した手法の有効性が確認できた.

また更なるシリアライゼーション効率の改善のためには JNI によるオーバーヘッドを削減することが必要であることがわかった.

参考文献

[1] Sun Microsystems, Inc: *Java Remote Method Invocation Specification*. http://java.sun.com/.

[2] Sun Microsystems, Inc: *Enterprise Java Beans specification*. http://java.sun.com/.

[3] Sun Microsystems, Inc.: *Jini specification*. http://java.sun.com/.

[4] Sun Microsystems, Inc: *Java Virtual Machine Specification*. http://java.sun.com/.

[5] Sun Microsystems, Inc: *Java Object Serialization Specification*. http://java.sun.com/.

[6] Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H. E. and Plaat, A.: An efficient implementations of Java's Remote Method Invocation, *Proc. of 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 173–182 (1999).

[7] Kono, K. and Masuda, T.: Efficient RMI : Dynamic Specialization of Object Serialization, *Proc. of IEEE 20th International Conference on Distributed Computing Systems*, pp. 308–315 (2000).

[8] 河野健二, 益田隆司: 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. 投稿中.

[9] Sun Microsystems, Inc: *Java Native Interface Specification*. http://java.sun.com/.

[10] Welsh, M. and Culler, D.: Jaguar: Enabling Efficient Communication and I/O in Java. http://www.cs.berkeley.edu/ mdw/proj/jaguar/.

[11] IRISA / INRIA - University of Rennes: *Tempo specializer - a partial evaluator for C*. http://www.irisa.fr/compose/tempo/.

[12] Object Management Group: *The Common Object Request Broker Architecture Specification*. http://www.omg.org/.

[13] Opyrchal, L. and Prakash, A.: Efficient object serialization in Java, *Proc. of IEEE 19th International Conference on Distributed Computing Systems Workshop*, pp. 96–101 (1999).

[14] Braux, M. and Noye, J.: Towards Partially Evaluating Reflection in Java, *Proc. of 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)* (2000).