ブリッジ・メソッド生成によるJava実行環境上の言語処理系高速化

牧野 祐己　竹内 幹雄　堀井 洋　河内谷 清久仁　小野寺 民也

Accelerating Java-based Programming Languages by Generating Bridge Methods

Yuki Makino, Mikio Takeuchi, Hiroshi Horii, Kiyokuni Kawachiya, and Tamiya Onodera

Java™ の実行環境上でほかのプログラミング言語の処理系を実装することが増えている．このようなシステムにおいて，他の言語の基本データ型を Java の型に対応させる場合，Integer のような「オブジェクト型」ではなくint のような「スカラー型」をなるべく使った方が性能がよい．また，ほかの言語の総称型を表現する際に Java の総称型をそのまま活用できれば，効率的な実装が可能となる．しかし，Java の総称型はスカラー型を用いて実体化することができないため，基本データ型をスカラー型に対応付けられない状況が存在する．本論文では，この問題を「ブリッジ・メソッド」を必要個所に生成することで解決し，基本データ型を可能な限りスカラー型で表現するための実装手法を提案する．提案した手法を用いることでスカラー型をより多用することができるようになり，Java 上の言語処理系の高速化が可能となる．この手法により，Java 上の X10 言語処理系において，現実的な数値計算アプリケーションの性能を 2.2 倍向上することができた．

It has become increasingly popular to implement other programming languages in Java. To improve the performance of such implementations, basic data types of these languages should be mapped to Java scalar types (such as int) rather than object types (such as Integer). It is also better to map generic types to Java generics. However, since scalar types cannot be used as type parameters in Java generics, there are situations in which basic data types cannot be mapped to Java scalar types. This paper proposes a technique for mapping basic data types to scalar types as much as possible, by generating "bridge methods" in necessary places. By using this technique, we achieved up to 2.2 times improvement in performance for the implementation of X10 in Java.

Key Words & Phrases ：Java, X10, プログラミング言語処理系, 総称型, コード生成, ブリッジ・メソッド
Java, X10, programming language processor, generics, code generation, bridge method

1. はじめに

コンピュータの性能向上はクロックの高速化やパイプラインの改善によって達成されてきた．しかし，シングルプロセッサの性能向上は頭打ちとなったために，最新のプロセッサでは，複数の実行コアを備えたマルチコア構成をとるものが普通となっている．また，グラフィックス処理用チップの多数の演算ユニットを汎用計算処理に用いる General-Purpose computation on Graphics Processing Units（GPGPU）も身近になってきている．近い将来には，これらを複数搭載したマルチプロセッサー・マシンが高速ネットワークで相互接続された大規模並列分散環境が一般的なものになると予想される．

そのような環境では実行コアの総数は数千～数十万個にもなるため，それらを活用できるソフトウェアを開発することは容易ではない．そのためには，ハードウェア環境の変革に合わせ，プログラミング環境も変革を遂げることが必要となる．具体的な要件としては，以下のものが挙げられる．

- 大量の非同期タスクを軽量に生成・消滅できること.また，デッドロックなどに悩まされずにタスク間の同期が行えること.
- 共有メモリー環境と分散メモリー環境を単一のプログラムで扱えること.また，異なるアーキテクチャーの実行環境をサポートできること.
- 複数マシン間の複雑な通信処理を記述しなくてよいこと.しかし，並列処理や分散処理をある程度明示的に制御できること.
- 開発をサポートするツール群が用意されていること.

これらの要件を満たすべくIBM 基礎研究部門（IBM Research）が開発中の新しいプログラミング言語が，X10

提出日：2010年9月6日　再提出日：2011年3月3日

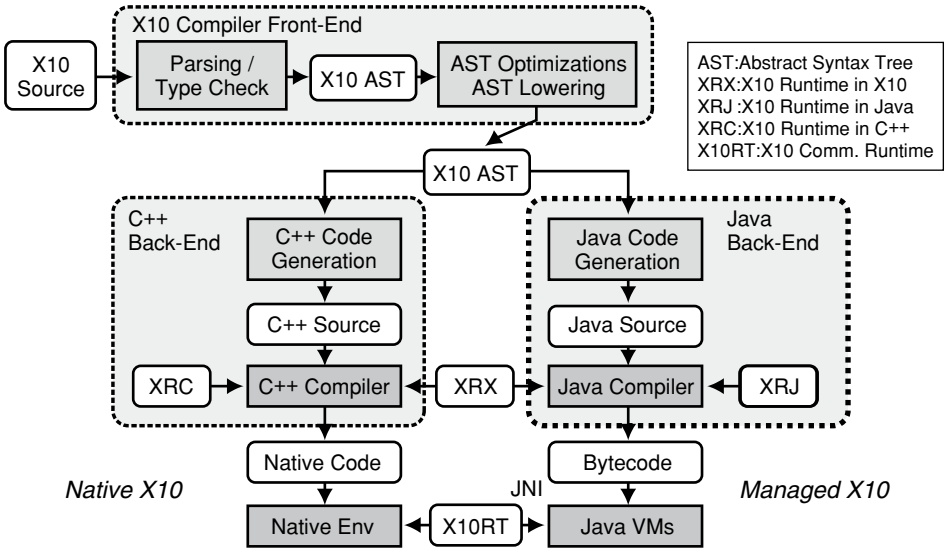


図1. X10コンパイラーの構成

（エックステン）である［1］［2］.

X10は，米 Defense Advanced Research Projects Agency（DARPA）の High Productivity Computing Systems（HPCS）プログラムに基づくIBM の PERCS プロジェクトの一部である. Productive Easy-to-use Reliable Computer Systems（PERCS）プロジェクトは，先進的なチップ技術とアーキテクチャー，OS，プログラミング言語などを統合し，ハードウェアとソフトウェアを総合的にデザインすることで，並列アプリケーションの生産性を向上させることを目標としている. そのために，X10 は新しいプログラミングモデルと開発ツール群を提供する. X10 の開発は 2004 年に開始され，オープン・ソース・プロジェクトとして行われている.

図1に示すように，X10 にはソース･コードを Java のソース･コードにコンパイルし Java 処理系上で実行する「Java バック・エンド」が存在する［3］. Java バック・エンドは，既存の Java の資産を活用しつつ X10 が持つ新しい機能を利用するために開発されている.

最近では Scala［4］，JRuby［5］，Jython［6］など X10 と同様に，Java［7］のプラットホームをほかの言語処理系の実行環境として活用する JVM 言語（JVM languages）が増えている. こうした言語は，Java バイトコードにコンパイルされ Java 仮想マシン上で実行される. Java 処理系上で実行できれば，既存の Java で書かれた資産を再利用することができ，新しい言語でガベージ・コレクターなど Java が持つ特徴を有効活用することができる.

このような JVM 言語の性能向上のためには，その言語の基本データ型を Java のスカラー型（int など）に

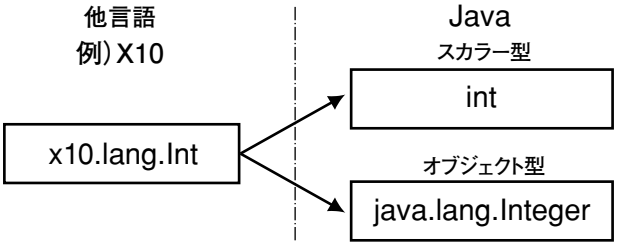


図2. X10の基本データ型とJavaの型の対応の例

うまく対応付けることが必要である. しかし，Java の総称型には型パラメーターとしてオブジェクト型（Java.lang.Integer など）しか使えないという制限があり，この対応付けが困難になる場合がある.

本論文では，JVM 言語の総称型の実装に Java の総称型を活用した場合に生じるスカラー型とそのオブジェクト型の間の変換オーバーヘッドを，「ブリッジ・メソッド」の生成によって低減する手法を提案する. これにより，Java との相互運用性を損なわず可能な限りスカラー型に対応させて処理を高速化することができる . 本手法を Java 上の X10 言語処理系の実装に用いることで，現実的な数値計算アプリケーションの性能を 2.2 倍向上することができた.

2. JVM言語における課題

JVM 言語における実装上の課題の 1 つが，言語独自の基本データ型をどのように Java の型に対応付けるかである. 図2のように，Java では数値，文字データ，論理値などの基本データの値を表現する「スカラー型」が存

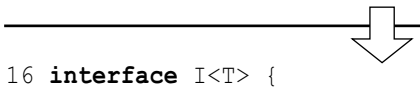
在し，「オブジェクト型」と区別される．それぞれのスカラー型には，対応するオブジェクト型が存在し，それらの間では，例えば int と java.lang.Integer のように暗黙的な型変換が行われる．スカラー型に比べオブジェクト型は高機能であるが，値はヒープ上に生成されメモリー管理の対象となるなどのオーバーヘッドがある．そのため，Java 処理系上で動くプログラミング言語を実現するとき，その言語の基本データ型を可能な限り Java のスカラー型に対応付けることが，性能上重要となる．

Java には HashMap<K,V> のように，型をパラメーター（K や V）として持つ「総称型」※1 が用意されている［8］．Java の総称型はコンパイル時に型パラメーターの情報を消去し実行時に残さない方法（型消去）によって実現されている．他言語の総称型の機能を Java の総称型を使って実現できれば，効率的に他言語の総称型が実装でき，Java との相互運用性の観点からも都合が良い．しかし，Java の型パラメーターはコンパイル時にオブジェクト型へと変換されてしまい，スカラー型で実体化することができない．

基本データ型で型パラメーターを実体化することができる他言語の総称型を実現するために Java の総称型を活用した場合，型パラメーターを実体化している基本データ型はオブジェクト型として扱う必要があり，スカラー型に対応させようとすると食い違いが生じてしまう．例えば，メソッドの基本データ型の引数をスカラー型に対応させてしまうと，そのメソッドが，実体化した型パラメーターを引数に持つメソッドをオーバーライドできないという問題が生じる．例えば図3では，クラス D のメソッド m をスカラー型で宣言することができない（26 行目）．

Dragos らは，総称型の型パラメーターを特定のスカラー型で特化したクラスをそれぞれ生成する手法を提案している［9］．この手法により，型パラメーターを引数に持つメソッドであってもスカラー型の引数で呼ぶことが可能となる．しかし，特化されたクラス間の継承関係をうまく表現できないため，適用範囲は部分的である．

```
X10
 1 interface I[T] {
 2   def m(a:T, b:T):Int;
 3 }
 4
 5 class C implements I[Int] {
 6   public def m(a:Int, b:Int) {…}
 7 }
 8
 9 class D {
10   public def m(a:Int, b:Int) {
11     return a - b;
12   }
13 }
14
15 class E extends D implements I[Int] {}
```



```
Java
16 interface I<T> {
17   Integer m(T a, T b);
18 }
19
20 class C implements I<Integer> {
21   public Integer m(Integer a, Integer b)
22   {…}
23 }
24
25 class D {
26   public Integer m(Integer a, Integer b){
27     return a - b; // Boxing & Unboxing!
28   }
29 }
30
31 class E extends D implements I<Integer> {}
```

図3. スカラー型とオブジェクト型の食い違いにより,スカラー型が使用できない例

3. ブリッジ・メソッド生成方法の提案

本節では，スカラー型とオブジェクト型の食い違いによって生じる問題点を，X10 を例に具体的に説明し，その解決手法を提案する．

3.1 基本データ型のJavaの型への対応付け

すでに述べたように，他言語の基本データ型を Java 上で効率的に実現するためには，それらをできるだけ Java のスカラー型に対応させることが望ましい．そのため X10 では，整数を表す基本データ型 x10.lang.Int を図2のように Java のスカラー型の int へとコンパイルし，やむを得ない場合のみオブジェクト型の java.lang.Integer にコンパイルしている．

※1　総称型は，抽象的に型を総称するパラメーター型を導入し，定義時に具体的なデータ型に依存せず型安全で汎用的なプログラムが書けるようにする仕組みのことである．

```
1 class C implements I<Integer> {
2   public int m(int a, int b) {…}
3
4   // generated bridge method
5   public Integer m(Integer a, Integer b) {
6     return m((int) a, (int) b);
7   }
8 }
```

図4. スカラー型とオブジェクト型をブリッジするメソッド

```
1 class D {
2   public int m(int a, int b) {
3     return a - b;
4   }
5   // generated many bridge methods!
6   public Integer m$G(int a, int b) {…}
7   public Integer m$G(Ingeger a, int b) {…}
8   public int      m(Integer a, int b){…}
9   public Integer m$G(int a, Integer b) {…}
10  public int      m(int a, Integer b){…}
11  public Integer m$G(Integer a, Integer b) {…}
12  public int      m(Integer a, Integer b) {…}
13 }
```

図5. 無関係なクラスに複数のブリッジ・メソッドが必要な例

x10.lang.Int の算術演算には，Java の int の算術演算をそのまま利用しているが，その結果がスカラー型であっても，オブジェクト型で受け取るごとに Java による暗黙の型変換が行われてしまう．暗黙の型変換では，「ボクシング」と呼ばれるオブジェクトへの変換や，逆にオブジェクトからスカラー値を取り出す「アンボクシング」と呼ばれる処理が行われ，性能を低下させる一因となる．

そのため，算術演算結果を変数に受け取る部分やメソッドの引数として渡す部分では，可能な限りスカラー型を使用し，オブジェクト型を用いないようにコンパイルすることが性能上重要となる．

3.2 Javaの総称型とその問題点

Java の総称型は，コンパイル時に型消去と呼ばれる処理によって，型パラメーターの部分がその上限の型に変換され，実行時には型パラメーターの情報を持たないという実装になっている．

型パラメーターが変換された結果の型はオブジェクト型となる．そのため，Java の総称型の型パラメーターをスカラー型で実体化することはできない．

例えば，I<T> という総称型があった場合，I<Integer> というように，オブジェクト型である Integer による実体化は可能であるが，スカラー型である int を使用して I<int> というようには実体化できない．このため Java では，スカラー型を引数に持つメソッドが総称型の型パラメーターを引数に持つメソッドをオーバーライドや実装することができず，JVM 言語のコード生成に性能上の問題が発生してしまう．

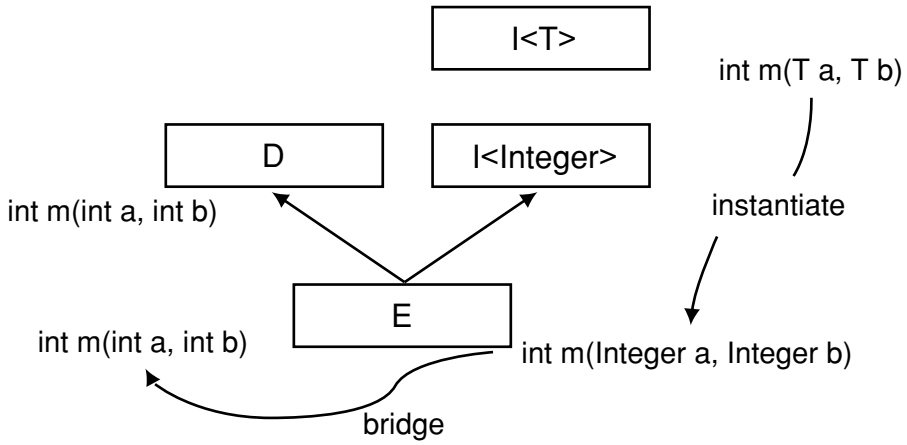
例えば図3の X10 コードでは，総称型のインターフェース I［T］とそれを具体的な型 Int で実体化し実装したクラス C，総称型ではないクラス D とそれを拡張したクラス E が定義されている．クラス E は I［Int］も実装している．X10 コードから生成された Java のクラス C では，それが実装している I<Integer> インターフェースのメソッド m（17 行目）を実装している．しかし，そのためにはすべての基本データ型をオブジェクト型で表現したメソッドを用意する必要があり，スカラー型を使用できなくなっている（21 行目）．

この問題は，クラス D のようにそのクラスが I<Integer> のメソッドを直接実装しない場合でも発生する．これは，クラスEのように D を継承したクラスが，型パラメーターで宣言されているメソッドを実装する可能性があるためである（31 行目）．このためすべてのクラスにおいて，メソッドの宣言の部分に使用する型をスカラー型にすることは単純にはできない（26 行目）．

3.3 スカラー型とオブジェクト型のブリッジ

なるべくスカラー型を使用するため，図4のようにスカラー型を使用したメソッドとオブジェクト型を使用したメソッドの両方を用意して，その間をブリッジする方法を考案した．オブジェクト型で実装した「ブリッジ・メソッド」からは，スカラー型で実装したメソッドを呼び出すようにする（6 行目）．これにより，型パラメーターで宣言されているメソッドのオーバーライドや実装を可能にしつつ，引数の型が確定している状況ではスカラー型のメソッドを直接利用することができるようになる．

しかしながら，この方法では総称型と無関係なクラスでも，基本データ型を使用している各メソッドに対してブリッジ・メソッドを生成する必要がある．特に，複数の基本データ型が出現する場合は，図5のようにそれぞれに対しスカラー型のものとオブジェクト型のものを生成する必要がある．その場合，一つのメソッドあたり生成すべきブリッジ・メソッドの数が，「2 の “引数の型として基本データ型が使用されている数” 乗 － 1」個になってしまう（6-12 行目）．つまり，この手法では生成される Java コードのサイズが




```
1 interface I<T> {
2   int m(T a, T b);
3 }
4
5 class D {
6   public int m(int a, int b) {
7     return a - b;
8 }
9
10 class E extends D implements I<Integer> {
11   // generated bridge method to the instantiating class
12   public int m(Integer a, Integer b) {
13     return m((int)a, (int)b);
14   }
15 }
```

図6. 実体化するクラスにブリッジ・メソッド生成

非常に大きくなってしまい，現実的とはいえない．

3.4 ブリッジ・メソッド生成手法の改良

われわれはさらに，上述のコード爆発問題を避けつつブリッジ・メソッドを使用する新しいコード生成手法を考案した．この手法のポイントは，型パラメーターを実体化するクラス側にブリッジ・メソッドを生成する点である．これによって，無関係なクラスに複数のブリッジ・メソッドを常時生成することを避け，コードサイズの増加を最小限にとどめている．また，ブリッジ・メソッド方式の利点である可能な限りスカラー型を利用した高速化も実現できている．

あるクラスが型パラメーターを実体化する場合，その実体化した型パラメーターを使用しているメソッドをオーバーライドするブリッジ・メソッドを生成する．ブリッジ・メソッドからは，スカラー型で実装されたメソッドを呼び出す．そのクラスが型パラメーターを実体化しない場合はこの処理は行われないので，無関係なクラスに対してブリッジ・メソッドは生成されない．

例えば，図6のクラスDのメソッドm（6行目）は型パラメーターを実体化しないので，int m（int a, int b）のようにスカラー型としてコンパイルする．そして，クラスDを継承し，パラメーター化されたインターフェースI<T>を具体的なクラスIntegerで実体化するクラスEにブリッジ・メソッドを生成する（12-14行目）．

ブリッジ・メソッドは，そのクラスが継承により型パラメーターを実体化する場合に，型パラメーターを含んでいるメソッドの型が確定する時に生成する．クラスEはインターフェースI<T>のint m（T a, T b）（2行目）のTをIntegerで実体化しint m（Integer a, Integer b）とするので，クラスEにint m（int a, int b）を呼び出すint m（Integer a, Integer b）をブリッジ・メソッドとして生成する．

メソッドの呼び出し側はそのメソッドの宣言を見て，型パラメーターが確定していればオブジェクト型でなくスカラー型で呼び出すようにすればよい．

3.5 特殊なケースへの対応

インターフェースはメソッドを実装できないので，インターフェースが型パラメーターを実体化していても，そこにブリッジ・メソッドを生成することはできない．そのため，あるクラスがそのようなインターフェースを実装した時点でそのインターフェースのすべての親を調べてブリッジ・メソッドを生成する．

例えば図7のように，インターフェースJSup<T>がint m（T a, T b）というメソッドを宣言し（2行目），インターフェー

X10

```
1′ interface JSup[T] {
2′   def m(a:T,b:T):Int;
3′ }
4′
5′ interface J extends JSup[Int]{
6′   def m(a:Int, b:Int):Int;
7′   // no bridge
8′ }
9′
10′ class F implements J {
11′   public def m(a:Int, b:Int):Int {…}
12′ }
13′
14′ abstract class GSup[T] {
15′   abstract def abst(a:T):void;
16′   def sup(a:T):void {…}
17′ }
18′
19′ abstract G extends GSup[Int]{
20′   abstract def abst(a:Int):void;
21′ }
22′
23′ interface K[T] {
24′   def n():T;
25′ }
26′
27′ class H implements K[Int] {
28′   def n():Int {…}
29′ }
```

Java

```
1  interface JSup<T> {
2    int m(T a, T b);
3  }
4
5  interface J extends JSup<Integer>{
6    int m(int a, int b);
7    // no bridge
8  }
9
10 class F implements J {
11    public int m(int a, int b) {…}
12    // bridge method
13    public int m(Integer a, Integer b) {
14       return m((int)a, (int)b);
15    }
16 }
17
18 abstract class GSup<T> {
19   abstract void abst(T a);
20   void sup(T a){…}
21 }
22
23 abstract G extends GSup<Integer> {
24   abstract void abst(int a);
25   // bridge method
26   void abst(Integer a) {
27     abst((int)a);
28   }
29
30   // bridge method
31   void sup(int a){super.sup((Integer)a);}
32 }
33
34 interface K<T> {
35   T n$G(); // renamed
36 }
37
38 class H implements K<Integer> {
39   int n() {…}
40   // bridge method
41   Integer n$G() {return n();}
42 }
```

図7. インターフェース間継承,クラス間継承,メソッド名リネームの例

スJがJSup<Integer>を継承していても，Jにはブリッジ・メソッドを生成せずint m（int a, int b）を宣言しておき（6行目），Jを実装するクラス側でブリッジ・メソッドを生成する（13-15行目）.

このような方法を取ることで，インターフェースを宣言する時に基本データ型として確定している型パラメーターをJavaのスカラー型として扱うことができる.

抽象メソッドで用いられる型パラメーターを実体化するが実装は行われない場合に対しても，ブリッジ・メソッドを生成する．例えば，親クラスからabstract void abs（T a）を継承し実装しない場合もabstract void abs（int a）;というような型パラメーターを実体化するメソッド（24行目）とvoid abs（Integer a）{abs（（int）a）;}というブリッジ・メソッドを生成する（26-28行目）.

親クラスが型パラメーターを用いてメソッドを実装している場合は，型パラメーターが確定した子クラスではスカラー型を用いたメソッドを生成し，そこから親クラスのメソッドを呼ぶようにする．例えば，void sm（T a）というメソッドを親クラスGSup<T>が実装している場合（20行目），GSup<Integer>を継承する子クラスGにブリッジ・メソッドvoid sm（int a）を生成し，super.sm（（Integer）a）を呼ぶようにする（26-28行目）.

このようにすることで，型パラメーターが実体化されているメソッドに対して，一貫してスカラー型で宣言されたメソッドを用意することができる.

戻り値が型パラメーターで宣言されているメソッドはすべてリネームする．これは，戻り値の型が確定したことによって生成されるブリッジ・メソッドが，元のメソッドと区別

できないためである．例えばT n（）というメソッドをint n（）で実体化した場合に，Integer n（）というブリッジ・メソッドを生成したいが，双方のメソッドのシグネチャー（Signature）が同じであるので区別できない．これを解決するため,戻り値が型パラメーターのメソッドはT n$G（）のようにすべてリネームする必要がある（35，41行目）．このリネームは，呼び出す側でも行う必要がある．

以上の手順により，ブリッジ・メソッドを必要な部分にのみ生成し，可能な場合にはスカラー型でメソッドを呼び出すことができるようになる．

4. 実装とベンチマークによる評価

前節で提案した手法の有効性を評価するため，X10のコンパイラーに変更を加えた．測定環境はSimultaneous Multithreading（SMT）付きの6コアの3.3GHz Intel® Xeon®と16GBのメモリーを搭載したPCで，OSは64bit版のRed Hat Enterprise Linux 5.5，Java処理系はIBM 64-bit SDK for Linux, Java Technology Edition, Version 6 SR9である．JITコンパイラーはデフォルト設定で使用している．

K - 平均法（K-means algorithm）と呼ばれる，データをK個のクラスタにクラスタリングする計算を行う，現実的な数値計算アプリケーションをベンチマークとして実験を行った．メソッド宣言のすべての型がオブジェクト型でコンパイルされたものと，ブリッジ・メソッドを生成しメソッド宣言が可能な限りスカラー型でコンパイルされたものとを比較した．

結果を図8に示す．縦軸は相対的な性能（実行時間の逆数）で，提案手法により2.2倍の高速化が達成できたことがわかる．メソッドの宣言がスカラー型になったことによって，型変換のコストが抑えられたと考えられる．

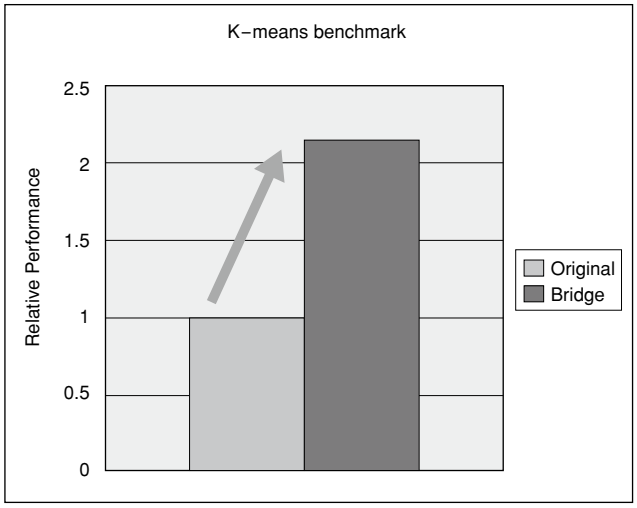


図8. K-平均法の計算を行うベンチマークによる比較

5. おわりに

本論文では，Javaの実行環境上で実現される他の言語処理系の高速化を図るためにブリッジ・メソッドを生成する手法を提案した．この手法では，Javaの総称型の型パラメーターを実体化するクラスに，ブリッジ・メソッドを生成することによって，コンパイル時に基本データ型を使用して宣言されているメソッドを，スカラー型を使用するようにコンパイルすることによって，ボクシング，アンボクシングがメソッドの境界で毎回起こることを防ぐ．これにより，コンパイル時に分かる部分では基本データ型がスカラー型で表現され，基本データに関する処理が高速化できる．X10のコンパイラーを変更し生成したコードによる比較では，現実的な数値計算アプリケーションにおいて2.2倍の高速化を達成できた．提案手法はX10の処理系（バージョン2.0.4以降）に採用されている．

本論文の手法は，クラスやメソッドを基本データ型で宣言した場合に，可能な限りJavaのスカラー型を使用することで高速化するものであるが，宣言時には型パラメーターを使用し実行時に実体化するようなものに関してもスカラー型を活用し高速化できる可能性がある．今後はこのような状況に対しても応用できる手法を研究したい．

謝辞

提案手法についてアドバイスをいただいたIBM Research - WatsonのVijay Saraswat氏,David Grove氏,Igor Peshansky氏に深謝したい．

参考文献

［1］ X10 Home, http://x10-lang.org/

［2］ 河内谷清久仁: マルチコア時代のプログラミング言語「X10」, 情報処理, Vol. 52, No 3, pp. 342-356, (2011).

［3］ Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera: Compiling X10 to Java, In *Proceedings of the ACM SIGPLAN 2011 X10 Workshop,* to appear, (2011).

［4］ The Scala Programming language. http://www.scala-lang.org/

［5］ JRuby.org. JRuby: 100% Pure-Java Implementation of the Ruby Programming Language. http://jruby.org/

［6］ The Jython Project: Jython: Python for the Java Platform. http://www.jython.org/

［7］ James Gosling, Bill Joy, Guy Steele, and Gilad Bracha: *The Java Language Specification, Third Edition,* Addison Wesley, (2005).

［8］ Gilad Bracha: *Generics in the Java Programming Language,* 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

［9］ Iulian Dragos and Martin Odersky: Compiling Generics Through User-Directed Type Specialization, In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems,* pp 42-47, (2009).

日本アイ・ビー・エム株式会社
大和ソフトウェア開発研究所
ソフトウェア・エンジニア

牧野 祐己 Yuki Makino

［プロフィール］
2009 年 , 日本 IBM 入社. 同社大和ソフトウェア開発研究所にてインメモリ型分散データベースの開発を経て, プログラム言語 X10 の コンパイラの研究に従事.
makino@jp.ibm.com

日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア
アドバイザリー・リサーチャー

竹内 幹雄 Mikio Takeuchi

［プロフィール］
1993 年 , 日本 IBM 入社. 東京基礎研究所にて Java JIT コンパイラの高速化, キー・バリュー型データストアを用いたトランザクション処理の高速化の研究の後, 大和ソフトウェア開発研究所にてインメモリ型分散データベースの開発を経て, 現在同基礎研究所にてプログラム言語 X10 のコンパイラの研究に従事. ACM および情報処理学会会員.
http://www.trl.ibm.com/people/takeuchi/

日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア
スタッフ・リサーチャー

堀井 洋 Hiroshi Horii

［プロフィール］
2004 年 , 日本 IBM 入社. 以来, 同社東京基礎研究所にて, トランザクション処理システムおよび分散処理システムの研究に従事. 工学博士. 情報処理学会会員.
http://www.trl.ibm.com/people/horii/

日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア
シニア・リサーチャー

河内谷 清久仁 Kiyokuni Kawachiya

［プロフィール］
1987 年 , 日本 IBM 入社. 以来, 同社東京基礎研究所にて, オペレーティ ング・システムやプログラミング言語に関する研究に従事. Java に関しては, Linux® 用 JIT コンパイラ開発, 同期処理の高速化, 使用メモリ解析と削減, JVM 言語の高速化等の研究を行っている. ACM シニ ア・メンバー, 情報処理学会, 日本ソフトウェア科学会会員. 博士.
http://www.trl.ibm.com/people/kawatiya/

日本アイ・ビー・エム株式会社
東京基礎研究所
インフラストラクチャー・ソフトウェア担当
シニア・テクニカル・スタッフ・メンバー

小野寺 民也 Tamiya Onodera

［プロフィール］
1988 年 , 日本 IBM 入社. 以来, 同社東京基礎研究所にて, オブジェクト指向言語の設計および実装の研究等に従事し, 2008 年より同研究所インフラストラクチャー・ソフトウェア担当部長.同社シニア・テクニカル・スタッフ・メンバー. 理学博士. ACM Senior Member. 情報処理学会, 日本ソフトウェア科学会, 各会員.
http://www.trl.ibm.com/people/onodera/