クラスファイル変換によるJavaプログラムの実行制御

丸山一貴[†] 寺田実[††]

関数呼び出しやループなど，プログラムの制御がジャンプするときに，あるカウンタ（タイムスタンプ）をインクリメントすると，ソースコードの行番号とタイムスタンプの値の組によってプログラム実行系列における1点（プログラム実行点）を識別可能になる．これはデバッグなどにおけるプログラムの実行制御に応用できる．この仕組みをJavaプログラムに対して実装した．コンパイル済みのクラスファイルを変換し，制御をジャンプさせるバイトコードの直前にタイムスタンプを更新するバイトコードを挿入することで実現している．本論文ではこのプログラム実行点の応用について述べ，実装に関する詳細と，付加コードによるオーバヘッドの計測の結果について述べる．

Execution Control of Java Programs by Class File Transformation

Kazutaka Maruyama[†] and Minoru Terada[††]

We propose an idea of identifying a point in a program execution trace by a pair of "line number in source files" and a counter, "timestamp". The counter increases when a control point in a program jumps such as function calls and loops. We call the point in a trace "position" and apply it to the execution control of programs for debugging, and so on. We implemented the timestamp system for Java programs. Transforming Java class files, we inserted bytecodes for updating timestamp just before the control point jumps. In this paper, we describe the applications and the implementation details of "position". Overhead measurement of added codes is also included.

1. はじめに

プログラムの実行制御はデバッグやプログラム理解など，実行時のプログラム状態を知る必要のあるタスクにおいて重要な役割を果たす．ここでいう実行制御とは，プログラムの実行系列（実行されたソースコード行の時系列）におけるある時点に制御点を移動することを指す．現在プログラマが利用できる実行制御は，ステップ実行やブレークポイント，ウォッチポイントといった，デバッガの持つ低レベルな（抽象度の低い）実行制御だけである．

本論文ではカウンタと行番号を用いて実行状態を特定する，より高度な実行制御のための基礎技術とその応用とを提案する．そのカウンタ（タイムスタンプ）は関数呼び出しやループなど，制御がジャンプする命令でインクリメントされるので，行番号と組にすると小さい記録量で実行状態を特定できる☆．Javaプログラムを対象とし，クラスファイルを変換することでタイムスタンプをインクリメントするコードを挿入する．

本論文で提案する技術の構成を**図1**に示す．以下，2章で新たな概念である実行点を定義する．これが，タイムスタンプと呼ぶカウンタと行番号との組で表されることを述べる．また，実行点を従来のブレークポイントと組み合わせることで，実行点の単純な応用である動的ブレークポイントを構築する．3章では，実行点を用いた実行制御の自動化のさらなる応用として，実行点の印づけ，逆向きウォッチポイント，実行系列中の2分探索について述べる．4章ではタイムスタンプ更新コードをJavaクラスファイル変換を用いて挿入する手法を説明する．5章でタイムスタンプ更新コードによるオーバヘッドの測定結果を示し，6章で関連研究との比較を行い，7章でまとめと今後の課題を述べる．

† 東京大学大学院情報理工学系研究科知能機械情報学専攻
Department of Mechano-Informatics, Graduate School of Information Science and Technology, The University of Tokyo
†† 電気通信大学電気通信学部情報通信工学科
Department of Information and Communication Engineering, Faculty of Electro-Communications, The University of Electro-Communications

☆ その実行状態を再現するためにはデバッガなどを用いて先頭から再実行する必要があるため，応用の局面では対象プログラムに再現性が要求される．再現性については7章で述べる．

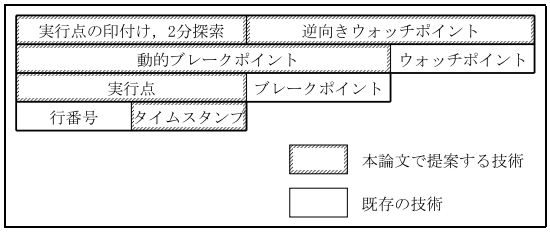

図 1 提案する技術の構成
Fig. 1 Structure of our proposal.

2. タイムスタンプによる実行制御

実行制御によってプログラムをある目的の状態に遷移させるためには，プログラムの実行系列における 1 行（プログラム実行点と呼ぶ）を特定できなくてはならない．

Boothe [1] はステップ実行のカウンタを用意して，プログラムの先頭からのステップ回数を用いて実行制御を行っている．同様のことは，我々の提案する「行番号」と「制御がジャンプする命令で更新されるカウンタ（タイムスタンプ）」の組を用いても実現可能であり，これらの差異は応用の方向性によっている．Boothe の目的は任意のデバッガコマンドの逆実行版を実装することである．我々は，デバッガコマンドが提供する低レベルな機能をプログラマが直接利用するだけでなく，それらの組合せで実現される，より高度な実行制御を自動化することを目的としている．したがって，ステップ実行を単位とするような更新頻度の高い（オーバヘッドの大きい）カウンタは不要である．

以下，本章ではタイムスタンプを用いてプログラム実行点を特定する手法について述べる．

2.1 タイムスタンプと実行点

旧来の実行制御はソースコードの行番号といった静的な情報を用いるため，実行状態を表すことができなかった．それというのも制御のジャンプによって同一の行を複数回実行する可能性があるからである．この「同一行の複数回実行」を区別するために，制御が上向きにジャンプするたびにインクリメントされるカウンタとして「タイムスタンプ」を導入し，行番号とタイムスタンプ値の組によって動的な実行状態を一意に表すことができる．

ループ構造を持つコードの例を**図 2** に示す．コードの実行系列は（1）から（3）が繰り返される形で表され，同一の行が繰り返し登場することになる（**図 3**）．行番号のみで表す静的な実行位置を location と呼ぶことにする．

一方，ループ本体が実行されるたびにタイムスタンプはインクリメントされるので，行番号とタイムスタンプ値の組は動的な位置である position（実行点）を表す．タイムスタンプを用いることで，実行系列中に複数回登場する location を区別することができる．

```
        :
(1) while(i < a){
(2)   i += b;
(3) }
        :
```

図 2 ループ構造を持つコード
Fig. 2 Code with a loop structure.

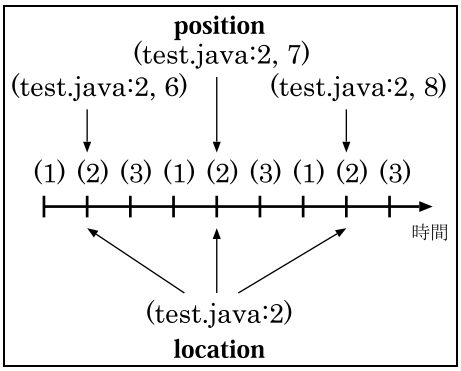

図 3 実行系列における location と position
Fig. 3 Location and position in execution path.

実行点をタイムスタンプを用いずに，デバッガのブレークポイントのヒット回数を用いて表すことも不可能ではない．しかしプログラムを実行中のどの時点でも実行点を知るためには，実行の最初からあらゆる行に対してブレークポイントを設定しておかなくてはならず，現実的な手段たりえない．したがって，タイムスタンプは必須である．

2.2 タイムスタンプの更新

タイムスタンプをインクリメントする対象となるのは制御が上向きにジャンプする命令である．これは，端的にはプログラムカウンタが減少する場合にあたるが，Java ソースコードとの対応としては以下がそれに該当しうる．

- メソッド呼び出しの入口と出口
- ループ本体の繰返し
- 例外

これらに該当する命令の直前でタイムスタンプ更新のためのメソッド（`Timestamp.inc()`）を呼び出せばよい．現在の `Timestamp` クラスの実装は**図 4** のようになっている．

2.3 動的ブレークポイントの実装

実行点を表現する仕組みが確立したので，移動の目

```
public final class Timestamp{
  private static long ts, ref;

  static{
    ts = 0;
    ref = 0;
  }

  public static void inc(){
    ts++;
    if(ts == ref){
      brake();
    }
  }

  private static void brake(){
    // empty
  }
}
```

図 4 Timestamp クラス
Fig. 4 Timestamp class.

標となる実行点の情報（タイムスタンプ値と行番号）が与えられたときにデバッギの制御点をそこに移動するという応用が可能になる☆．これは実行点にブレークポイントを設定していると考えられるので動的ブレークポイントと呼び，location に対して設定する通常のブレークポイントを静的ブレークポイントと呼ぶ．動的ブレークポイントを実装するにはデバッガなどのサポートが必要であり，Java VM の持つ Java Platform Debugger Architecture（JPDA）[2] を利用することになる．JDK に含まれるデバッガ jdb はこの API を利用している．

動的ブレークポイントの単純な実装は，デバッガの条件付きブレークポイントを利用することである．jdb は条件付きブレークポイントを持たないが，この仕組みを持つ GDB[3] の書式で表すと，図 2 のプログラムで（test.java:2, 8）という実行点が目標の場合は以下のようになる．

```
break test.java:2 if Timestamp.ts = 8
```

移動先の実行点が実行系列の中で現在よりも古い場合は，上記を行ったうえで，デバッグ対象プログラム（デバッギ）を先頭から再実行させる．ある実行状態にあるときにそのときの実行点情報を記録しておけば，動的ブレークポイントを用いてその状態に遷移することができる．

条件付きブレークポイントを用いた実装は容易ではあるものの，通常はデバッガによって目標の行番号（location）に静的ブレークポイントが設定され，それにヒットするたびに条件の成立の有無が評価されることになり実行速度が大きく低下する．図 2 のプログラムの場合は，**図 5** 中の △ 印に到達するたびにデバッギが停止することになる．この速度低下を避けるために，以下のような手順によって効率的に実装している．

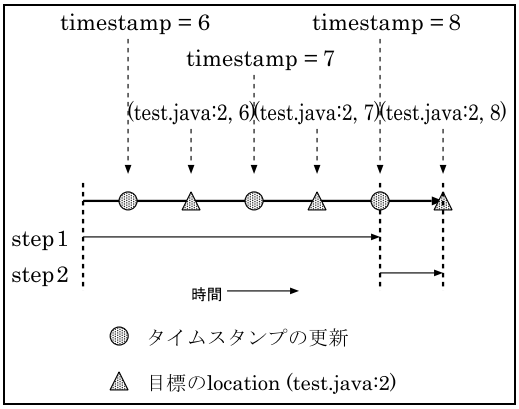

図 5 動的ブレークポイントの実行手順
Fig. 5 Procedure of dynamic breakpoint.

Step 1 実行を開始する前に，brake() メソッドに静的ブレークポイントを設定し，目標のタイムスタンプをデバッギ中の変数 Timestamp.ref に設定する．実行を開始するとタイムスタンプ値が Timestamp.ref の値に到達し，ブレークポイントで停止する．

Step 2 新たな静的ブレークポイントを目標の行番号（location）に設定して実行を再開する．

3. タイムスタンプによる実行制御の応用

タイムスタンプによって実行系列中の任意の 1 点（実行点）を特定できるようになった．これを用いると手動では行うことが困難な，抽象度の高い実行制御を実現できる．これらはデバッギを何度も再実行させるひどく乱暴な手法に見えるかもしれないが，近年の高速な計算機を利用することで，人間が手動で行うよりもずっと楽にデバッグを行うための有用な手法である．

なお，マルチスレッドプログラムが対象となる場合にはスレッド切替えに関する履歴を保存するなどの仕組みが必要になる．これについては 7 章で述べる．

3.1 実行点の印づけ

通常のデバッグにおいてバグの原因である場所が曖昧にしか分かっていないとき，その場所の手前に（静的）ブレークポイントを設定して制御を移動し，ステップ実行を繰り返し行う，という方法で実行点を動

☆ 目標の実行点情報は，デバッギが正常動作しない実行をするときにデバッガなどを用いて取得しておき，あとから利用するという状況を想定している．

かす．大規模なプログラムの中をこの方法で移動していると，うっかりコマンドを間違って問題の場所を通り過ぎてしまうことがある．このようなときには，そこへ至る経路を覚えておくか書きとめるかしておき，デバッギを先頭から再実行してその経路を手動で追いかけなくてはならない．

こうした面倒を避けるためには，実行点に印づけをすることが有効である．デバッギの振舞が「ここまでは正しい」というときに動的ブレークポイントを用いて印をつけ，後になってやり直したいようなことがあったらその実行点まで簡単に戻ってくることができる．

3.2 逆向きウォッチポイント

意図せずに変数の値が変更されてしまうようなバグは，破壊よりもずっと後に発覚するので原因を特定しにくい．通常はデバッガのウォッチポイントで破壊した操作を特定する．これはある変数に対応するメモリ領域を監視して，アクセスがあったときにトラップを発生してプログラマに知らせる仕組みである．一般にある変数への書き込みは何度も起こるので，デバッグ中のプログラマが手動で行うのは時間がかかりすぎ困難である．しかし，その多数の書き込みの中でも，発覚の直前に書き込んだ操作が原因であることが多いので☆，この直前の書き込みの時点に自動的に制御を移す仕組みとして,「逆向きウォッチポイント」を提案する．

この機能は実行点を利用した動的ブレークポイントと既存のデバッガを用いて，2 回の再実行で実現できる．手続きは次のようになる．

(1) 逆向きウォッチポイント（revwatch）が指定されたときの実行点に動的ブレークポイントを設定する（**図 6** の **S**）.

(2) Pass 1：
  (a) 通常のウォッチポイントを監視対象の変数に設定し，デバッギを先頭から再実行する．ウォッチポイントのトラップによって停止するたびにそのときの実行点の情報を記録する（図 6 の **W1** から **Wn**）.
  (b) これを，制御が実行点 **S** に到達するまで繰り返す☆☆.

(3) Pass 2：
  (a) もう一度デバッギを再実行し，新たな動的ブレークポイントを実行点 **Wn** に設定して実行を再開する．

☆ その変数に不正な値を書き込んだ操作であることは確かである.
☆☆ 最初に設定した動的ブレークポイントによって **S** に到達したことを知ることができる.

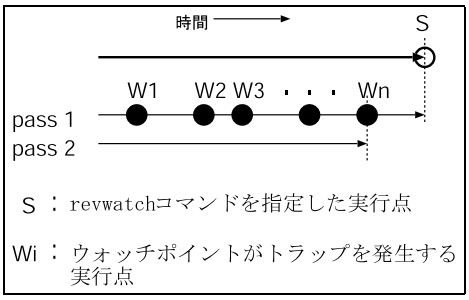

図 6 逆向きウォッチポイントの実行手順
Fig. 6 Procedure of reverse watchpoint.

  (b) デバッギは実行点 **Wn** で停止する.

3.3 タイムスタンプを用いた実行系列中の 2 分探索

タイムスタンプ値をキーとして実行系列中を 2 分探索し，何らかの条件が初めて成立しなくなる実行点を見つけ出すことができる．たとえば双方向リストのデータ構造を扱うプログラムで，その構造の妥当性を検査するアサーションを用意しておけばその条件が不成立になる（データ構造が不当になる）ときのタイムスタンプ値を見つけられる．これは次のような手順で実現できる．

(1) タイムスタンプ値が実行系列の左端と右端の中間値に達するまで制御を進める．探索の初期状態では，左端はデバッギの実行開始時点に相当するのでタイムスタンプ値は 1．右端はデバッギの実行終了時点に相当する．

(2) 条件式を評価し，
  - 真であれば現在の実行点を新たな左端とし，タイムスタンプ値が新たな左端と右端の中間値になるまで制御を進める．
  - 偽であれば現在の実行点を新たな右端とし，タイムスタンプ値が左端と新たな右端の中間値になるところまで，再実行によって制御を戻す.

(3) 上記を繰り返す.

タイムスタンプ値の最大値が n であれば評価回数は $\log n$ となるので，タイムスタンプが 32 ビットであれば条件評価の回数も再実行回数もたかだか 32 回である．

`AssertionError` クラスを用いても同様の処理は実現可能だが，2 分法では手動で挿入する必要がなく，条件評価の回数も最小限で済み，より複雑な条件が指定できる．

この手法は Tolmach ら[4] によってすでに提案されているが，我々はこれを関数型言語の領域から手続き型言語の領域に持ち込むことを可能にした．

4. タイムスタンプ機構の実装

適切な場所でタイムスタンプをインクリメントするために，与えられた Java プログラムの変換を行う．我々はすでに C プログラムに対して同様のシステムを実装済みであり[5]，コンパイラが中間コードを生成する段階でタイムスタンプ更新のためのコードを追加した．その経験に基づいて，Java プログラムに対してはコンパイラが生成したクラスファイル（バイトコード）を変換することでコードの追加を実現した．本章では C に実装した際の検討を述べたうえで，Java への実装の詳細について述べる．

4.1 C への実装の経験

C プログラムの変換方法として以下の 3 つのレベルを検討した．

(1) ソースコードレベル
(2) 中間コードレベル
(3) アセンブリコードレベル

第 1 はソースコードそのものを変換の対象とするので，処理系や OS，プラットフォームを選ばないという利点があるが，デバッグ中にプログラマから見えるコードは変換後のコードになってしまうという欠点がある．また，変換処理はプリプロセッサを通してからとなり，プリプロセッサの出力を解釈するのはそれほど容易ではないと考えた．

第 2 はコンパイラの中間コードのレベルでタイムスタンプ機構を挿入する．中間コードレベルで挿入すればターゲットアーキテクチャに対する移植性が失われることはないが，特定の処理系に依存することになる．

第 3 はコンパイラの出力するアセンブリコードを別途実装したフィルタによって変換する．これは実装が最も容易な方法だが，ターゲットアーキテクチャごとに実装する必要があるため移植性を著しく欠く．

以上の検討より第 2 の中間コードレベルで実装することが，実装の容易さと移植性とを両立させる最良の手法である．対象とする処理系は様々なプラットフォームで広く利用されていることを考慮して GNU C Compiler（GCC）[6] を選んだ．GCC では Register Transfer Language（RTL）と呼ばれる中間コードを内部で生成しており，ソースコードから RTL を生成する段階でタイムスタンプ更新コードを挿入するように GCC の一部を改変した．

4.2 Java への実装

前述の経験をふまえ，Java へ実装する場合の選択肢として以下の 4 つのケースを検討した．

(1) コンパイラ（ソースコードレベル）
(2) クラスファイル（バイトコードレベル）
(3) VM（バイトコード実行レベル）
(4) JPDA（デバッグインタフェースレベル）

第 1 と第 3 は C における検討と同様の理由で候補から除外される．第 4 は移植性に優れるが，Java プログラムの実行に細かく干渉する形になると考えられ，実行速度の大幅な低下が懸念されるためこれも除外される．

バイトコードは C におけるアセンブリコードと同質のものであるが，その形式は仕様によって定められており，ターゲットアーキテクチャに依存することがない．そこで第 2 を採用することが最良である．

4.3 タイムスタンプ更新対象のバイトコード

2.2 節ではタイムスタンプ更新場所の例として Java ソースコードの場合を示したが，4.2 節から対象をバイトコードとしたので，該当するバイトコードを改めて検討する．なお，前述のとおりタイムスタンプの更新は該当するバイトコードの直前で行われる．

メソッド呼び出しの入口と出口 入口に対応するバイトコードはないが，メソッド本体の先頭がこれにあたる．出口のうち，`void` 型のメソッド定義の末尾には `return` バイトコード（命令コード：177）が，値を返すメソッドでは `ireturn`（命令コード：172）から `areturn`（命令コード：176）までの 5 つが該当する．

メソッド本体の先頭ではなく，`invokevirtual` のような，メソッドを起動するバイトコードを対象とする方法もある．しかしこの方法には追加するコード量がより大きくなるという欠点がある．また，変換していないクラスファイルのメソッド呼び出し（たとえばシステムライブラリの呼び出しなど）までがタイムスタンプ更新の対象となり，オーバヘッドを増加させてしまうという欠点もある．

条件分岐 バイトコードの仕様として上向きのジャンプが起こりうるので，以下の 16 命令を対象に含める．`ifeq`（命令コード：153）から `if_acmpne`（命令コード：166）までの 14 命令と，`ifnull`（命令コード：198）と `ifnonnull`（命令コード：199）が該当．

無条件ジャンプ `goto`（命令コード：167）と `goto_w`（命令コード：200）．

その他の制御ジャンプ `jsr`（命令コード：168）と `ret`（命令コード：169），`jsr_w`（命令コード：201）が該当．

例外 例外に関するバイトコードには `athrow`（命令

表 1 コンスタントプールに追加した要素
Table 1 Added constant pool entries.

Index	Type	Contents	
$n+1$	Methodref	$n+2$	$n+3$
$n+2$	Class	$n+4$	
$n+3$	NameAndType	$n+5$	$n+6$
$n+4$	Utf8	9	“Timestamp”
$n+5$	Utf8	3	“inc”
$n+6$	Utf8	3	“()V”

表 2 オフセットの再計算が必要なバイトコード
Table 2 Bytecodes whose offsets are to be corrected.

Opcode	153	···	168
Mnemonic	`ifeq`	···	`jsr`
Opcode	170		171
Mnemonic	`tableswitch`		`lookupswitch`
Opcode	198	···	201
Mnemonic	`ifnull`	···	`jsr_w`

コード：191）があるが，これだけでは NullPointerException のような，ユーザのコードが明示的に `throw` しない例外に対処できない．よって，例外を発生（throw）する場所ではなく，例外を捕捉（catch）する場所を対象としなければならない．

`catch` ブロックの位置情報はクラスファイル中のメソッド定義に付随する例外表に含まれているので，これに基づいてタイムスタンプ更新コードを挿入する．

これらのうち，条件分岐と無条件ジャンプ，その他の制御ジャンプでは，各バイトコードはジャンプ先の番地をオフセットとして引数に保持している．これが負である場合（上向きジャンプの場合）に限って，`invokestatic` バイトコードを挿入する．挿入はこの 1 命令のみで，コンスタントプール中で `Timestamp.inc()` を表す Methodref タグのインデクス番号がその引数となる．

4.4 クラスファイルのその他の修正

ここではクラスファイル変換に関する，その他の処理について述べる．

- コンスタントプールに `Timestamp.inc()` メソッドのための 6 つのエントリを追加し，コンスタントプールの要素数を修正する．追加したエントリは**表 1** のとおりである．
- 制御をジャンプさせるバイトコードでは，ジャンプ先をそのバイトコードからのオフセットで表す．タイムスタンプ更新コードを挿入することでそれらオフセットの再計算が必要になる．再計算の対象となるバイトコードは**表 2** のとおりである．
- 例外表が保持する `catch` ブロックなどの位置もメソッドの先頭からのオフセットによって表されているので，同様に再計算を行う．

4.5 クラスファイル変換プログラム

クラスファイルを変換するプログラムは Java で記述し，コード量は 1,000 行程度である．この変換プログラムの大部分を作成してからバイトコード解析ツール BCEL[7] を知ったため，解析ツールを利用せずにスクラッチからコーディングした．

ユーザは，タイムスタンプによる実行制御の対象としたいコードを含むクラスファイルを，事前にこの変換プログラムで変換しておく．実行制御の対象としない場合は変換しなくてよく，その分，実行時のオーバヘッドを減らすことができる．

5. オーバヘッド測定

ここでは前述の方法でタイムスタンプ機構を組み込んだ Java プログラムのオーバヘッドを計測し，異なる `Timestamp.inc()` の挿入方法を行った以下の場合について比較する．

- 変換前（Original Code）
- すべての制御ジャンプに挿入（All Jumps）☆
- 下向きの制御ジャンプを除くすべての制御ジャンプに挿入（Backward Jumps）

対象のプログラムには，オーバヘッドが最大になる空ループのみを持つプログラム（empty loop）と SPEC JVM98[8] のベンチマークプログラムを用いる．また，計測を行った環境は Pentium-III 733 MHz，640 MB メモリ，Linux-2.4.7，JDK-1.4.0 である．

5.1 実行時間

実行時間の計測は各ベンチマークを 7 回連続して実行し，最短と最長の結果を除外した 5 回分の平均を求めた．結果を**表 3** に示す．

空ループの場合では，ループを 1 回実行するたびにもともとは存在しないメソッド呼び出しが発生するため，かなりの速度低下を起こしている．現在の実装では最悪の場合，実行時間が 4 倍程度になる☆☆．しかしおおむね 1.5 倍から 2 倍程度であり，実用上の問題はないと判断する．

☆ 4.3 節で述べた，ジャンプ先を示すオフセットが正であるバイトコード（下向きジャンプ）にも挿入した場合について，参考までに掲載する．

☆☆ このベンチマークは空ループを 100 万回実行したものであり，All Jumps と Backward Jumps における更新コード実行回数はそれぞれ 1,000,004 回と 1,000,003 回であり，1 回分の違いしか存在しない．2 者の間の実行時間の相違には特に意味はないと考える．

表 3 オーバヘッド測定結果
Table 3 Measurement results of the overhead.

Benchmark	Original Code		All Jumps		Backward Jumps	
empty loop	12.774	(1.00)	53.278	(4.17)	56.416	(4.42)
_201_compress	1.4006	(1.00)	4.33	(3.09)	2.7754	(1.98)
_202_jess	0.2126	(1.00)	0.3024	(1.42)	0.271	(1.27)
_209_db	0.412	(1.00)	0.464	(1.13)	0.433	(1.05)
_222_mpegaudio	0.1826	(1.00)	0.375	(2.05)	0.2792	(1.52)
_227_mtrt	0.363	(1.00)	1.0094	(2.78)	0.9038	(2.49)
_228_jack	0.603	(1.00)	0.791	(1.31)	0.7212	(1.19)

単位は秒（括弧内は比）

表 4 ファイルサイズの変化
Table 4 Increment of file size.

Benchmark	Original Code		All Jumps		Backward Jumps	
empty loop	276	(1.00)	331	(1.20)	328	(1.19)
_201_compress	14,443	(1.00)	19,105	(1.32)	18,640	(1.29)
_202_jess	396,536	(1.00)	410,368	(1.03)	407,240	(1.03)
_209_db	10,156	(1.00)	10,910	(1.07)	10,588	(1.04)
_222_mpegaudio	120,182	(1.00)	125,515	(1.04)	124,438	(1.04)
_227_mtrt	859	(1.00)	929	(1.08)	920	(1.07)
_228_jack	132,516	(1.00)	140,642	(1.06)	138,109	(1.04)

単位はバイト（括弧内は比）

また，All Jumps と Backward Jumps との実行時間の増分を比べると，ほとんど変化がないものと増分がほぼ半減するものとに分かれた．これは各ベンチマークプログラムのループの構成によるものと考えられる．

なお，_227_mtrt はマルチスレッドであるため，図 4 で示した inc メソッドに synchronized をつけたうえで計測した．

5.2 クラスファイルの大きさ

クラスファイルの大きさの合計を比べたものを**表 4** に示す．

元のファイルが極端に小さい空ループの場合と _201_compress の場合に増し分がやや顕著になるものの，多くの場合においてほとんど変化がなく実用上の問題はないと考える．

6. 関連研究

本論文に関係する我々の先行研究として文献 9)，5) がある．

文献 9) では C プログラムを対象として，「現在実行中の関数」を呼び出した関数を親関数と呼び，再実行を用いてその親関数の入口に制御を戻すという実行制御の実現を目的とし，関数呼び出しの入口と出口のみにタイムスタンプ更新コードをアセンブリコードレベルで挿入する方法を述べた．本論文とは目的が異なっており，また実装方式に移植性を欠く点も異なる．

文献 5) は 4.1 節で述べた中間コードレベルでの実装に関する報告である．

Binder ら[10] は J-SEAL2 と呼ばれるモバイルエージェントシステムに CPU とメモリの資源管理機構を組み込んでおり，完全な移植性を維持するためにバイトコード変換を用いている．特に CPU 資源の管理のためにスレッドごとにカウンタが用意されており，スレッドは実行中に基本ブロック単位でそのカウンタを更新する．また，その更新オーバヘッドを最適化するために制御フローグラフを用いた解析を行い，更新頻度を削減する．オーバヘッドは最大で 1.41 倍に抑えられている．

速水ら[11] も同様の CPU 資源管理のために，バイトコード変換を用いてコードを挿入している．より粒度の高い資源管理のためにオーバヘッドがやや大きくなり，最大となる _201_compress で 1.63 倍となっている．

彼らが我々より大きな機構を組み込んでいるにもかかわらずオーバヘッドが小さいのは，我々の現在の実装ではタイムスタンプ更新のためにメソッド呼び出しを行っており，そのオーバヘッドが大きいためであると考えている．

7. まとめと課題

プログラムの実行制御のために，制御がジャンプするごとにインクリメントされるカウンタ（タイムスタ

ンプ）を用いて，実行系列における 1 点（実行点）を特定する方法について述べ，その応用を紹介した．次に，C を対象として GCC に実装した経験をもとに Java に対する実装をバイトコード変換によるものとし，具体的な変換手法について述べた．そして変換後のクラスファイルについて，ファイルサイズと実行時間のオーバヘッドについて計測した結果を示した．

今後の課題には未対応のクラスファイルへ対応する点があげられる．現状ではメソッドに含まれるバイトコード長が `Integer` の最大値を超える場合に対応できない可能性があるので，この点を修正する．また，タイムスタンプ更新コードを挿入した結果として `goto` バイトコードなどが 2 バイトのオフセットで不足した場合に，4 バイトオフセットの `goto_w` に切り替えることも必要である．バイトコードの長さが変わることでオフセットのさらなる再計算が必要になり，コード長の最適化を考えると NP 完全になる[12]が，そのような場合には無条件にすべての命令を 4 バイトオフセット型に変換すればよい．こうした対応にはバイトコード変換プログラムを修正するか BCEL などのバイトコード解析ツールで実装しなおすかのどちらかとなる☆．

また，タイムスタンプの更新に関しては問題が起きないが，プログラムを先頭から再実行して実行制御に応用する場合にはマルチスレッドを用いたプログラムの再現性の維持を考慮する必要がある．マルチスレッドの Java プログラムの再現は Choi ら[13]が DejaVu に実装済みであり，そうした技術を利用することで解決できる．そのうえで，3 章で述べた応用の実装を JPDA を用いて記述することになる．

参 考 文 献

1) Boothe, B.: Efficient Algorithms for Bidirectional Debugging, *Proc. ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pp.299–310 (2000).
2) Sun Microsystems, Inc.: *Java Platform Debugger Architecture*. http://java.sun.com/products/jpda/
3) Stallman, R., Pesch, R., Shebs, S., et al.: *Debugging with GDB — The GNU Source-Level Debugger — Eighth Edition* (2000). (included in GDB source tree).
4) Tolmach, A. and Appel, A.W.: A Debugger for Standard ML, *Journal of Functional Programming*, Vol.5, No.2, pp.155–200 (1995).
5) 丸山一貴，寺田 実：タイムスタンプを用いたプログラム実行点のポジショニング，情報処理学会第 61 回全国大会講演論文集 (2000).
6) Stallman, R.M.: *Using and Porting the GNU Compiler Collection*, Free Software Foundation (1999). (included in GCC source tree).
7) Dahm, M.: Byte Code Engineering, *Java-Information — Tage 1999* (*JIT'99*) (1999). http://bcel.sourceforge.net/
8) The Standard Performance Evaluation Corporation: *SPEC JVM98 Benchmarks*. http://www.spec.org/osg/jvm98/
9) 丸山一貴，寺田 実：関数単位疑似逆実行の高速化，情報処理学会論文誌：プログラミング，Vol.41, No.SIG9(PRO8), pp.1–7 (2000).
10) Binder, W., Hulaas, J.G. and Villazón, A.: Portable Resource Control in Java, *Proc. OOPSLA'01 the Conference on Object Oriented Programming Systems Languages and Applications*, pp.139–155 (2001).
11) 速水雄太，田浦健次朗，米澤明憲：Java バイトコード変換による細粒度 CPU 資源管理，情報処理学会論文誌：プログラミング，Vol.43, No.SIG3(PRO14), pp.41–51 (2002).
12) Szymanski, T.G.: Assembling Code for Machines with Span-Dependent Instructions, *Comm. ACM*, Vol.21, No.4, pp.300–308 (1978).
13) Choi, J.-D. and Srinivasan, H.: Deterministic Replay of Java Multithreaded Applications, *Proc. SIGMETRICS Symposium on Parallel and Distributed Tools*, pp.48–59 (1998).

(平成 14 年 12 月 24 日受付)
(平成 15 年 7 月 22 日採録)

丸山 一貴（学生会員）

1975 年生．1999 年東京大学工学部機械情報工学科卒業，同年同大学院工学系研究科情報工学専攻修士課程入学．2001 年同修士課程修了，同年同大学院情報理工学系研究科知能機械情報学専攻博士課程入学．現在，同博士課程在学中．プログラム言語処理系やユーザインタフェース，ネットワークプログラムに興味を持つ．ACM，IEEE 各学生会員．

☆ 2003 年 3 月現在，BCEL を利用してのコーディングが完了している．

寺田　　実（正会員）

1959 年生．1981 年東京大学工学部計数工学科卒業．1983 年同大学院工学系研究科情報工学修士課程修了．同大学計数工学科助手，電気通信大学電子情報学科助手を経て，1991 年東京大学工学部機械情報工学科講師，1992 年同大学助教授．2002 年電気通信大学電気通信学部情報通信工学科助教授，現在に至る．工学博士．主な研究分野はプログラム言語処理系，プログラミング環境など．ソフトウェア科学会，ACM 各会員．