Javaバイトコード変換による細粒度CPU資源管理

速 水 雄 太[†] 田浦 健次朗[†] 米 澤 明 憲[†]

JVMは型安全性やクラスローダにより単一アドレス空間内で複数のプログラムを安全に分離する．しかし，アプレットなどの信用できないプログラムを実行するためには，プログラムの分離だけでなく，資源割当ての制御が必要でありJVMはそれを欠いている．たとえばあるアプレットがCPU資源を独占して，他のプログラムの実行を妨害することが起りうる．本研究ではJVMでCPU使用量をユーザレベルできめ細かに制御する手法について述べる．基本的な方針は，バイトコードに直接カウンタをインクリメントするコードを挿入し，各スレッドの実行命令数を動的にカウントするというものである．命令数を正確に数えながらオーバヘッドを小さくするアルゴリズムを考案した．まず，対象プログラムを，基本ブロックをノード，ブロック間の潜在的な制御の移動をエッジ，基本ブロックの実行頻度を重さをとする重みつきグラフでモデル化した．コード挿入のコストは，カウンタが挿入された命令の重みの総和で与えられる．この定式化のもとコストを低く抑えるアルゴリズムを考案した．このアルゴリズムは任意のコントロールフローグラフを対象とするので，goto文や例外処理などの複雑な制御構造も統一的に扱うことができる．また，潜在的な制御の移動をエッジとするので，高階関数や動的メソッド呼び出しに対してはコントロールフロー解析などの結果を用いて，より良いコード挿入を達成できる．さらに，短いサイクルをunrollingすることにより，オーバヘッドを軽減することについても考察する．

Java Bytecode Transformation for Fine Grain CPU Resource Management

Yuta Hayami,[†] Kenjiro Taura[†] and Akinori Yonezawa[†]

JVM safely separates several programs within a single address space using bytecode verifier and classloader. However, in order to execute untrusted code such as applet it requires not only separating programs but also accounting resources, which current JVM lacks. For instance an applet may use too much CPU resource and disturb the execution of other programs. This work presents an efficient method which provides fine grain CPU resource management at user level. We insert instructions into given byte code, which increment counters to dynamically count the number of instructions executed by each thread. We designed an algorithm which precisely counts the number of instructions at a low overhead. We modeled an object program as a weighted graph, where a node corresponds to a basic block ,an edge to a potential transfer of control, and the weight of a node to the execution frequency of the node. The cost of insertion is given as the weighted sum of nodes to which the counting instruction is inserted. We designed an algorithm keeping the cost at low level. This algorithm applies to arbitrary CFG. Therefore it uniformly deals with complex control structures such as goto statements and exceptions. Because an edge corresponds to a potential transfer of control, we can improve the insertion of code for higher order functions and dynamic method invocations using the result of control flow analysis. Finally, we will investigate an unrolling strategy for reducing overhead.

1. はじめに

インターネットの発展にともないプログラミング言語Javaが使用される機会・用途ともに増大している．現在，大部分のwebブラウザはJava Virtual Machine（JVM）を搭載しており，アプレットやサーブレットなどのプログラムをネットワーク経由でダウンロードして実行することが可能である．

このようにJavaが身の回りに浸透するにつれ，必然的に他人がプログラムしたいわゆる「信頼できないコード」を実行する必要性も出てくる．既存のJavaのセキュリティ機能には主なものに，ベリファイア，クラスローダ，セキュリティマネージャなどがあげられる．これらの保護機構は信頼できないコードの動作を実行前に検証するとともに，プログラムの実行をつね

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo

に監視することによって，ローカルファイルシステムの破壊，個人情報の漏洩，あるいはネイティブコードの実行を禁止するといった最低限のセキュリティ機構を提供している．

しかし，OS では必須である「計算機資源の管理」についてみると，Java 処理系はまだまだ不十分である．その欠点を補うためにこれまでにも，Java 処理系に資源管理機能を付加する研究がなされている[1)~3)]．ここで意図している計算機資源とは CPU サイクルやヒープメモリ，ネットワークの送受信などを指しており，基本的に Java 処理系はこれらの資源消費量について制限を設けない．そのため，ただ単に無限ループを実行するだけのプログラムや，オブジェクトを無意味に作成し続けるプログラムを実行させることにより，同一の JVM で実行される他のプログラムの実行を妨害し，ひいてはホストマシンの計算機資源を浪費させることが可能である．

本研究では計算機資源の中でも特に重要な CPU 資源の管理を扱った．管理を行うためには，第 1 に各プログラムが消費している CPU 資源の量を正確に把握する必要がある．アプローチとして考えられる方法を列挙すると以下のようになる．

- OS の資源管理機構を利用する．
- JVM を拡張する．
- 管理対象のプログラムを変換する．

OS の機能を利用する方法では，JVM の下にある OS の資源管理機構を積極的に利用する．この場合，たとえばネイティブコードを用いて各 Java プログラムの CPU 使用状況をクエリするという方法があり，JRes [1)] では CPU 資源管理にこの方針を用いる．しかし，この場合管理機能の大部分は OS に依存しており，管理の粒度に関しても数十 msec といった粗いものになりがちである．そもそも，機種・OS によらずにプログラムを実行できるという Java の利点が失われてしまう．

次に JVM を拡張する方法を検討すると，他のアプローチに比べて直接的で分かりやすいという利点がある．しかし，現在の JVM は実行を高速化するための数々の最適化機構を備えており，なかでも Just In Time コンパイラ（JIT）の有無は，プログラムの実行時間に大きな影響を与える．したがって，JVM を改造あるいは拡張する際は，これらの最適化機構もあわせて組み込まなければ実用的な JVM にはならない．

以上の点をふまえて，本研究では管理対象となるプログラムを変換する方針を採用した．具体的には，プログラムのバイトコード中に適切なコード断片を挿入し，プログラムの実行命令数をバイトコードレベルで正確にカウントする．そしてこの情報をもとにプログラムが消費する CPU 資源を管理することで実行命令数の平等性をバイトコードレベルで保証する．

バイトコード変換の利点としては，Java の長所であるポータビリティの良さをまったく損なわないことがあげられる．また，JVM を改変する場合と異なり JIT をはじめとする数々の最適化機構を利用でき，OS を利用する場合では困難だったより細かい粒度の資源制御が可能である．文献 3) も資源管理にバイトコード変換を利用しているが，資源管理のきめに関する考察はなく，その点で細粒度の管理を目的とする本研究とは異なる．

バイトコード変換の欠点としてプログラム中に命令列を挿入する都合上，実行時にオーバヘッドが生じることは避けられない．実行時に資源管理のオーバヘッドがかかることは他の手法も同様だが，バイトコード変換の場合は対象プログラムの性質によってもかかるオーバヘッドが変わってくる．コード挿入アルゴリズムは，管理のきめと命令数の正確なカウントを保証したうえでいかにオーバヘッドを削減できるかが重要であり，以上の要請を満足するアルゴリズムを考案することが本研究の主な目的の 1 つである．

この目標を達成するため，我々は対象プログラムを重み付き Control Flow Graph（CFG）でモデル化し，コード挿入コストを定義した．そして，基本ブロックの重みを反映してコストを低く抑えるアルゴリズムを考案し，単純なアルゴリズムとのコストの比較を行った．さらに，我々は提案したアルゴリズムを実際に実装し，いくつかのサンプルプログラムで実験を行った．この実装はすべて Java で記述されており，クラスローダ機構を利用することによりバイトコード変換をロード時に行う．実験の結果，変換を行わないオリジナルのプログラムと比べて，変換後のプログラムは 11%〜62%程度のオーバヘッドがかかることが確認された．

以降の論文の構成は次のようになっている．2 章では実行時のオーバヘッドを低く抑えながら，なおかつ正確に実行命令数をカウントするためのコード挿入アルゴリズムを提案し，単純なアルゴリズムとの比較・評価を行う．3 章では 2 章で提案するアルゴリズムを実装し，いくつかのベンチマークプログラムに適用した結果について考察を行う．4 章では資源管理，コード挿入アルゴリズムなどに関する関連研究について説明し，5 章がまとめと今後の課題となっている．

2. コード挿入アルゴリズム

2.1 目　　標

本章では CPU 資源管理を行うための目標と，その目標を達成するための問題点を整理し問題の適切な定式化を行う．しかる後その問題を解くアルゴリズムを提案し評価を行う．そして最後に，適切な CFG の重み付けを行うために利用した手法を説明する．

最初に我々が達成すべき資源管理の目標を述べると，「ミリ秒以下の細かい粒度で，各スレッドの実行命令数を制御する」ということになる．具体的な制御の例として最も単純なものは,「各スレッドにほぼ同じだけの命令数を実行させる」というものだが，ほかにも，「毎 10 ms ごとにある決められた命令数だけ実行する」（ソフトリアルタイム応用),「アプリケーションスレッドが 100,000 命令実行するごとに，なんらかのバックグラウンドタスク（たとえば GC）が 1,000 命令実行する」などの様々な制御が考えられる．そのような制御を，なるべく小さなオーバヘッドで実現するのが本研究の目的である．

さて，本研究ではそれを，実行命令数を数えるコードを挿入することによって実現するが，実行命令数の制御を行う方法には実行命令数を計測するほかにも，実行「時間」の制御を行うことにより，近似的に実行命令数の制御をすることも可能である．

このアプローチをとった場合，OS によって提供されるインターバルタイマを用いて，タイマの expire 時にユーザレベルスレッドを切り替えて実行時間を管理する方法と，各スレッドが定期的に高精度の計時関数を呼び出しながら，実行時間を管理する方法とが考えられる．しかし，アラームシグナルの精度はタイマ割込みの頻度以下にはなりえず，一般的な OS では 10 ミリ秒程度の間隔であるため，前者の方法では，ミリ秒以下の細粒度の管理は不可能である．一方後者の場合，我々のアプローチと同様，計時関数を呼び出すオーバヘッドが問題となる．コード挿入方式であれば実行命令数を検査するはずの場所で，実行命令数の代わりに経過時間を検査をすることになり，この検査のオーバヘッドは実行命令数の検査に比べれば大きい．一方で，この方式を用いれば実行命令数を数えるための加算命令は必要なくなるが，それが検査のオーバヘッドを打ち消すほどのものであるかどうかは明らかでない．

本研究では精度の高さ，オーバヘッドの少なさ，OS への依存度の低さなどの点を評価して，以降はコード挿入方式について考察する．

逆に実行時間の平等性を保証したい場合には，バイトコード命令の複雑さに応じてカウンタを増加させることにより，修正を加えた実行命令数と計時関数を用いて得た CPU 占有時間との比を一定に保つことは十分可能である．以下では簡単のためすべてのバイトコードの複雑度を 1 として扱う．

実行命令数を正確に数える点ではプログラムのプロファイリングと共通するところがあるが，資源管理とプロファイリングとでは管理の「きめ」において決定的に異なる．プロファイリングでは実行終了時のカウンタの値が総実行命令数と等しくなっていることが重要であるのに対し，資源管理においてはプログラム実行中の任意の時点で，真の実行命令数とカウンタの値の差が一定の範囲内に収まっていることが重要である．ゆえにコード挿入アルゴリズムはカウンタのインクリメントが定期的に起こることを保証しなければならない．しかし，単純に 1 命令実行するたびにカウンタを増やすような変換を行うと，実行時に非現実的なオーバヘッドを被るはめになり目的に反してしまう．そこで以上の目的を達成するコード挿入アルゴリズムを開発する準備として，対象プログラムのモデル化とコード挿入コストの定義を行う．

2.2 準　　備

まず，対象プログラムを重み付き CFG によってモデル化する．重み付き CFG の定義は以下のようになる．

定義 1　重み付き CFG $G = (V, E)$

- V： 頂点集合．各頂点 $v \in V$ はプログラム中の基本ブロックに相当する．さらに各頂点 $v \in V$ は対応する基本ブロックの長さと実行頻度に応じて，長さ l_v と，重み w_v を持つとする．
- $E \subset \{(i, j)|i, j \in V\}$ 有向枝集合．頂点間の潜在的な制御の移動を表現．

定義 1 を補足する．頂点 $v(\in V)$ には入力辺を持たない *entry node* と出力辺を持たない *exit node* を持つ．実行時には *entry node* から実行を開始し，*exit node* へと制御が移動していく．*entry node*, *exit node* ともに複数あってもよい．そのほかに末尾でメソッド呼出を行うノードを *call node* といい，メソッドから復帰するノードに出力辺をはることにする．

あるノード v について l_v，w_v が与えられたとき，l_v，w_v にはいくつかの制約がある．l_v は v の長さという定義から l_v は正の整数である．また，CFG において $w_v(\in$ 実数$)$ は流量保存則を満たす必要がある．すなわち $freq(i, j)$ を枝 (i, j) の実行頻度として，

$$w_v = \sum_{(u,v)\in E} freq_{(u,v)} = \sum_{(v,u)\in E} freq_{(v,u)} \quad (1)$$

が成立していなければならない．**図 1** は重み付き CFG

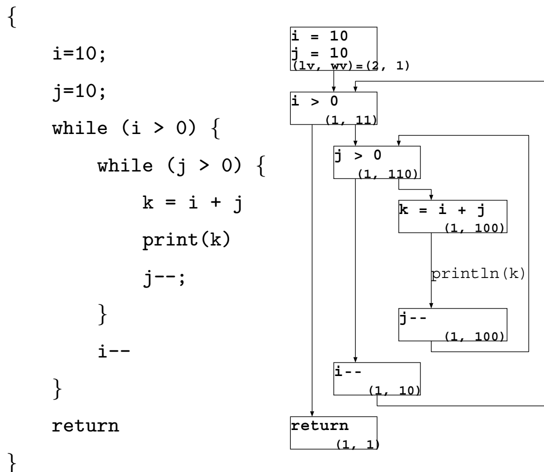

図 1　サンプルプログラムとその CFG
Fig. 1　A sample program and its CFG.

の一例である．

次にカウンタが定期的にインクリメントされるための条件を定式化する．そこで任意の時点での真の実行命令数とカウンタの値との差 ε とおく．すなわち

$$\varepsilon = (\text{真の実行命令数}) - (\text{カウンタの示す値}) \quad (2)$$

としたとき，ε がつねに一定の正の整数 $L_{\max}$ 未満であることを保証すればよい．そこで次のような制約条件（以下 $L_{\max}$ 制約）を導入する．

$L_{\max}$ 制約

$$0 \leq \varepsilon < L_{\max} \quad (3)$$

ここでパラメータ $L_{\max}$ を調節することにより管理の粒度を自由に変えることができる．$L_{\max}$ の値を大きくすれば実行時のオーバヘッドは低下することが期待できるが管理の粒度は粗くなる．反対に小さくすればきめ細かな管理が可能であるがそれなりのオーバヘッドを被ることが予想される．

では次にコード挿入のコストを定義する．基本的にはアカウンティングコードを挿入したノードの重みの総和で与えられる．ノードの重みは基本ブロックの実行頻度に対応しているのでこの定義は自然である．コード挿入全体のコスト C_{total} は各ノード $v (\in V)$ における挿入コストを C_v とおいたときの C_v の総和であるから，

$$C_{total} = \sum_{v \in V} C_v \quad (4)$$

となる．ところで実行命令数を正確にカウントするだけならば単純には各ノード v の末尾で l_v だけカウンタの値を増加させれば十分であり，各ノードへのコード挿入は 1 カ所だけで済む．

しかし，たとえば l_v が大きい場合などでは $L_{\max}$ 制約を満たすために 1 つのノードに複数のコード挿入が必要になる場合もでてくる．そこでノード $v (\in V)$ に挿入したコードの数を n_v と表記することにすると，C_v は以下で与えられる．

$$C_v = w_v n_v \quad (n_v \text{ は非負整数}) \quad (5)$$

式 (4) と式 (5) からコード挿入コスト C_{total} が定義できたが，式 (3) の $L_{\max}$ 制約が C_{total} のなかに陽に現れるように n_v をさらに形式的に表現する．まず新たに $2N (N = |V|)$ 個の変数

$$0 \leq \varepsilon_{v_1}^{\rm in}, \varepsilon_{v_1}^{\rm out}, \cdots, \varepsilon_{v_N}^{\rm in}, \varepsilon_{v_N}^{\rm out} < L_{\max}$$

を導入する．これらの変数が意図するところはノード v の先頭（in）と末尾（out）における真の命令数との差をそれぞれ $\varepsilon_v^{\rm in}$，$\varepsilon_v^{\rm out}$ で表したものである．当然，CFG において枝 $(i, j) \in E$ $(i \in V, j \in V)$ で結ばれているノード i，j に対応する $\varepsilon_i^{\rm out}$，$\varepsilon_j^{\rm in}$ 間には依存関係があり，

$$\varepsilon_i^{\rm out} = \varepsilon_j^{\rm in} \quad ((i, j) \in E) \quad (6)$$

を満たさなければならない．

さて，$\varepsilon_v^{\rm in}$，$\varepsilon_v^{\rm out}$ を用いてノード v の先頭と末尾における ε の値を指定すると，ノード v に最低限必要なコード挿入の個数 n_v は一意に定まり式 (7) で表される．

$$n_v = \left\lfloor \left| \frac{l_v + \varepsilon_v^{\rm in} - \varepsilon_v^{\rm out}}{L_{\max}} \right| \right\rfloor \quad (7)$$

ここで式 (7) で与えられる n_v を満たすようなコード挿入は確かに存在し，挿入位置決定アルゴリズムとして**図 2** に示すとおりである．図 2 の position_and_count は $L_{\max}$ 制約を満たし，なおかつ，ちょうど式 (7) で与えられる n_v と等しい数のコード挿入を行うアルゴリズムである．n_v の最小性については図 2 のアルゴリズムの挿入法からほぼ明らかである．

結局，式 (7) の導入によりコード挿入アルゴリズムは $2N$ 個の変数を解く部分（制約解消アルゴリズム）と，得られた $2N$ 個の変数から具体的なノード内部のコード挿入位置を計算する部分（挿入位置決定アルゴリズム）の 2 つに分割することができた．後者については図 2 で示したとおりであるから，あとは制約解消アルゴリズムが問題となる．

2.3　コード挿入アルゴリズム

本節では対象プログラムの CFG を入力とし，式 (6) の制約のもとで前節で導入した $2N$ 個の変数の値を求める制約解消アルゴリズムを提案する．ただし制約そのものはそれほど強くないので，制約を満たす解は複数存在する．その中からできるだけコスト C_{total} の小さいものを求めることが目的である．

```
Input  : ε_v^in, ε_v^out, l_v
Output :(コード挿入位置 p, インクリメントする値 c)
    のリスト {(p,c)} (0 < p ≤ l_v, 0 < c ≤ L_max)
Algorithm  position_and_count(ε_v^in, ε_v^out, l_v))
{
    l'_v = ε_v^in + l_v − ε_v^out
    if (l'_v < 0)
        return {(l_v, ε_v^in + l_v − ε_v^out)}
    else if (l'_v = 0)
        return {}
    else if (0 < l'_v ≤ L_max)
        if (ε_v^in + l_v < L_max)
            return {(l_v, ε_v^in + l_v − ε_v^out)}
        else
            return {(L_max − ε_v^in, ε_v^in + l_v − ε_v^out)}
    else
        return {(L_max − ε_v^in, L_max)}
          +position_and_count(0, ε_v^out, l_v−(L_max−
        ε_v^in))
}
```

図2 挿入位置決定アルゴリズム
Fig. 2 An algorithm for determining insertion points.

そもそも変数 $\varepsilon_v^{\mathrm{in}}$，$\varepsilon_v^{\mathrm{out}}$ のとりうる値は有限個の可能性しかないので，全通り探索すれば必ず最適解を計算することが可能である．しかし式 (7) に示した n_v は $\varepsilon_v^{\mathrm{in}}$，$\varepsilon_v^{\mathrm{out}}$ について線型ではないので，手軽に最適解を計算するのは容易ではない．

一方，単純なアルゴリズムとして EachBlock がある．これは単に，

$$\varepsilon_{v_1}^{\mathrm{in}} =, \cdots, = \varepsilon_{v_N}^{\mathrm{in}} = 0$$
$$\varepsilon_{v_1}^{\mathrm{out}} =, \cdots, = \varepsilon_{v_N}^{\mathrm{out}} = 0$$

としてしまう方法である．これは一応制約を満たしているので確かに解の1つになっている．しかし必ずしも最適解とは限らず，かなり無駄なコストを払う可能性が高い．その原因は EachBlock が $L_{\max}$ より小さなノードにも必ずコードを挿入することに起因する．

図3 が我々の提案するアルゴリズム WeightFirst である．WeightFirst の概要を説明する．WeightFirst には重みの大きいノードの $\varepsilon_v^{\mathrm{in}}$ と $\varepsilon_v^{\mathrm{out}}$ の値を優先的に決定することにより，重みの小さいノードでコードの挿入が増えたとしても全体のコスト C_{total} を小さくする意図がある．アルゴリズムの内容を詳しく見てみると，まず $\varepsilon_v^{\mathrm{in}}$，$\varepsilon_v^{\mathrm{out}}$ のいずれかが undef であるノードのリストから重み最大のノードを1つ取り出す．

```
Input  : CFG G = (V, E), 重み W = {w_v | v ∈ V}
Output :各 node の先頭と末尾のにおける誤差の値
    {ε_v^in | v ∈ V} ⋃ {ε_v^out | v ∈ V}
Algorithm  WeightFrist(G, W)
{
    // 初期化 出入口，メソッド呼び出し前では誤差 0,
    // その他 undef
    ε_v^in := 0 (v : entry_node)
    ε_v^out := 0 (v : exit_nodes or call_nodes)
    ε_v^in, ε_v^out := undef (v : その他のノード)
    node_list := V
    // すべての ε が決まるまで反復
    while(node_list ≠ null)
        // sort: 1. w_v の降順
        // 2. ε_v^in = undef 優先,
        // 3. ε_v^out = undef 優先，あとは任意.
        sort(node_list);
        // first: リストの先頭の要素
        max = first(node_list)
        if (ε_max^in = undef and ε_max^out = undef)
            ε_max^in := 0
        else if (ε_max^in = undef)
            ε_max^in :=
            ε_max^out − L_max   (mod L_max)
        else
            ε_max^out :=
            ε_max^in + L_max   (mod L_max)
        // update: コストの決定したノードは削除
        update(node_list)
    return {ε_v^in | v ∈ V} ⋃ {ε_v^out | v ∈ V}
}
```

図3 制約解消アルゴリズム（Weight First）
Fig. 3 An algorithm for solving the constraints (Weight First).

重み最大のノードが複数あった場合はそのうちのどのノードを選択してもかまわない．取り出したノードは $\varepsilon_v^{\mathrm{in}}$，$\varepsilon_v^{\mathrm{out}}$ の値によって場合分けを行い，一方の変数の値が分かっているときは position_and_count を適用したとき n_v が最小になるように他方の値を決定する．もし，$\varepsilon_v^{\mathrm{in}}$，$\varepsilon_v^{\mathrm{out}}$ の両方が undef の場合，$\varepsilon_v^{\mathrm{in}}$（あるいは $\varepsilon_v^{\mathrm{out}}$）の値を適切に設定する必要がある．現時点では単に $\varepsilon_v^{\mathrm{in}} = 0$ と置いているが，実験的に最適な $\varepsilon_v^{\mathrm{in}}$ の値を決定することも可能である．

以下ではある例題に EachBlock と WeightFirst を

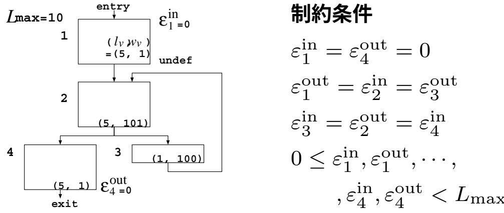

図 4 コード挿入の例題
Fig. 4 A sample problem.

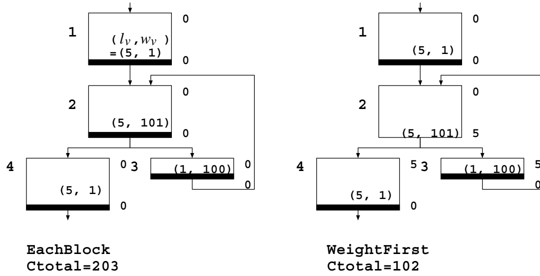

図 5 コード挿入アルゴリズムの適用結果（EachBlock and WeightFirst）
Fig. 5 After code insertion (EachBlock and WeightFirst).

適用した場合の具体例をあげる．今，**図 4** のような CFG $G = (V, E)$ が与えられたとする．$|V| = 4$ であり，各ノードは 1〜4 の番号で識別される．特に 1 は *entry node*，4 は *exit node* であり，それを反映して $\varepsilon_1^{in} = \varepsilon_4^{out} = 0$ に初期化する．この例題に EachBlock と WeightFirst を適用した結果が**図 5** である．挿入したコードの数は EachBlock が 4，WeightFirst が 3 で 1 つしか違わないが，WeightFirst では最大重みのノード 2 にコードが挿入されておらず，全体のコストは両者で大きく異なっていることがわかる．

2.4 コストの評価

以下ではコード挿入アルゴリズムに EachBlock と WeightFirst を用いたときのコストの評価を行う．

コード挿入アルゴリズムに EachBlock と WeightFirst を用いたときのコストをそれぞれ C_{each}，C_{WF} とおき，C_{opt} を最適解とする．まず C_{each} と C_{opt} の関係は次式で与えられる．

$$C_{each} = \sum_{v \in V} \left\lfloor \left| \frac{l_v + \varepsilon_v^{in} - \varepsilon_v^{out}}{L_{\max}} \right| \right\rfloor w_v$$

(すべての $v \in V$ について $\varepsilon_v^{in} = \varepsilon_v^{out} = 0$ より)

$$= \sum_{v \in V} \left\lfloor \frac{l_v}{L_{\max}} \right\rfloor w_v$$

ゆえに，

$$\begin{aligned} & C_{each} - C_{opt} \\ &= \sum_{v \in V} \left\{ \left\lfloor \frac{l_v}{L_{\max}} \right\rfloor - \left\lfloor \left| \frac{l_v + \varepsilon_v^{in} - \varepsilon_v^{out}}{L_{\max}} \right| \right\rfloor \right\} w_v \\ &\leq \sum_{v \in V} \left\{ \left(\frac{l_v}{L_{\max}} + 1 \right) - \left| \frac{l_v + \varepsilon_v^{in} - \varepsilon_v^{out}}{L_{\max}} \right| \right\} w_v \\ &\leq \sum_{v \in V} \left\{ \left(\frac{l_v}{L_{\max}} + 1 \right) - \frac{l_v + \varepsilon_v^{in} - \varepsilon_v^{out}}{L_{\max}} \right\} w_v \\ &= \sum_{v \in V} w_v + \frac{1}{L_{\max}} \sum_{v \in V} (\varepsilon_v^{in} - \varepsilon_v^{out}) w_v \\ &= \sum_{v \in V} w_v \end{aligned} \tag{8}$$

となる．なお，最後の変形では各ノードが流量保存則を満足するように重み付けされていることを利用した．式 (8) は C_{each} のコストがたかだか C_{opt} と全ノードの重みの総和で抑えることができることを示している．

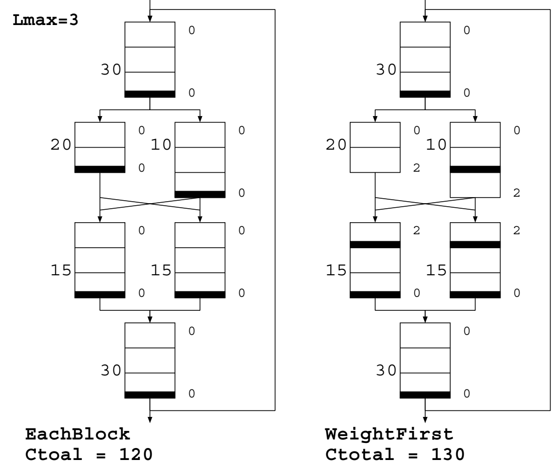

図 6 $C_{each} < C_{WF}$ となる例
Fig. 6 An example where $C_{each} < C_{WF}$ holds.

次に，C_{WF} と C_{each} の関係について考察する．C_{WF} は重みの大きなノードから優先的に ε_v^{in}，ε_v^{out} を決定することにより，挿入コストの高いノードへの挿入をできるだけ回避するように工夫している．そのため，C_{each} よりコストが小さくなることが期待される．しかし，**図 6** に示すように $C_{WF} > C_{each}$ であるような例が実際に存在する．これは重みの大きなノード（図 6 では重み 20 のノード）を優先した結果，他のノードのコストがかさんだ例である．

ただし，実際のプログラムでは 3 章で述べるように，多くの場合において $C_{WF} \leq C_{each}$ が成り立っていることが確認できる．

2.5 バイトコード変換の注意点

以上に述べたバイトコード変換アルゴリズムは本研究の目的であるきめの細かさと実行命令数の平等性

を保証している．ただし，実行命令数ではなく実行時間の平等性を保証したい場合には，計測した実行命令数から実行時間を推定することは可能ではあるが必ずしも容易なことではない．たとえば，今日の実用的な JVM の多くは実行時に JIT コンパイラがネイティブコードを生成することによって実行の高速化を図っている．JIT コンパイルされるタイミングや，実際に生成されるネイティブコードは JVM の実装に依存するので，バイトコードレベルの実行命令数が単純に実行時間に比例するわけではない．よって同じバイトコードでも実行頻度およびプログラムの文脈によりコストの見積りを変えるなどの工夫の余地がある．

またバイトコード変換による資源管理の限界として，native コードで実装されている native メソッドは管理対象に含まれない．

2.6　CFG の重み付け

今までは CFG のノードの重みは与えられるものとしてコード挿入アルゴリズムを構築してきた．しかし，元々のプログラムには重み情報は付加されておらず，なんらかの方法で重み情報を得なければ `WeightFirst` アルゴリズムは適用できない．

ここでノードの重みとは対応する基本ブロックの実行頻度であった．我々は CFG を構築する際に，静的な解析によって実行頻度を予測した．静的な予測に限らずプロファイリングも各基本ブロックの実行頻度を与える．しかし，プロファイリングは対象プログラムをいったん実行する必要があり，資源管理という視点からは静的な解析の方が望ましい．

実行頻度予測の関連研究に静的な分岐予測[4),5)]がある．これらの概要を説明すると，まず CFG 中のループ構造を検出し，諸々の経験則に基づき各々の条件分岐について分岐先の確率分布を求める．次のステップでは計算した確率分布を元にコントロール・フローを伝搬させて各エッジの実行頻度を見積もる．ノードの実行頻度はエッジの実行頻度から容易に求まる．

本研究では文献 4)，5) に列挙してある経験則の中から，ループに関する経験則 Loop branch heuristic（LBH）と Loop exit heuristic（LEH）の 2 つを用いて分岐予測を行った．LBH は分岐先がループの先頭に戻る枝の方に分岐しやすいと予測し，LEH はループ内の分岐はループを抜ける枝には分岐しにくいと予測する．このほかの経験則と組み合わせて予測を行うことで実行頻度予測の解析精度向上が期待できる．

3.　実装・実験

我々は 2 章で述べたアルゴリズムに従ってバイトコード変換を行うプログラムを実装した．この変換プログラムは Java で記述されているため JVM がインストールされている環境ならどこでも容易に実行可能である．

さらに我々の実装は Java のクラスローダの機能[6)]を利用することにより，クラスファイルが JVM にロードされたときに変換を適用する．変換後のプログラムは直接 JVM に渡されるので，管理の対象となるプログラムの余計な複製をファイルシステムに保存する必要がない．また JVM が，本当に必要になった時点ではじめてクラスファイルをロードする lazy loading 機能を備えている場合，実際に利用されるクラスファイルのみを変換すればいいので，無駄な変換を防止できる利点がある．反面，実行時に動的にバイトコード変換を行うので，カウンタを増加するコードを実行する時間に加えて，バイトコード変換を行う時間が実行時のオーバヘッドに加算される．しかし，実際には後に示すようにプログラム変換にかかる時間は長くて数秒であり，多くの場合では問題にならない．

ただし，クラスローダに関する制約[6)]から標準パッケージ（`java.lang`，`java.util` など）に含まれるクラスをロードできるのは，JVM に組込みの bootstrap クラスローダのみであり，標準パッケージ中のクラスはロード時に変換を施すことができない．そのためこれらのクラスに関しては前もって変換しておく必要があるが，単純に変換するとバイトコード変換プログラムまで資源管理の対象に含まれてしまうので都合が悪い．対処法としては，文献 3) にあるようにクラス中の各メソッドについて，元々の実装と資源管理を行う実装の 2 種類を用意するといった方法が考えられるが，現時点では標準パッケージについてはバイトコード変換を行っておらず，そのため標準パッケージ中のコードを実行しても実行命令数にはカウントされない．

バイトコード変換ツールは JOIE[7)] を利用した．以下ではバイトコード変換プログラムを複数のサンプルプログラムで実行し，

- プログラム変換前後の実行時間，
- $L_{\max}$ の値を変化させたときの実行時間の増減，
- プログラム変換前後のクラスファイルサイズ，

について調査した．さらにコード挿入アルゴリズムには `EachBlock` と `WeightFirst` を用いて比較を行った．

実験環境は次のとおりである．

- CPU：Intel Pentium III 500 MHz,
- メインメモリ：128 MB,
- Java 実行環境：Java2 SDK1.3.

資源管理の対象とするサンプルプログラムは

SPECjvm98 ベンチマーク集に含まれるプログラムで，それぞれのプログラムの説明は**表 1** にまとめた．

表 2 は $L_{\max} = 100$ のときのコード挿入を行わないプログラム `Normal` と，コード挿入アルゴリズムに `EachBlock` と `WeightFirst` を用いた場合の実行時間をまとめたものである．実行時間はクラスをロード・変換するのにかかった時間（Load）と，総実行時間（Total）について調べた．このときの総実行時間をグラフ化したものが**図 7** である．db と jack では `EachBlock` と `WeightFirst` に違いが見られず，compress ははっきり `WeightFirst` の優位性を示しており，jess ではわずかに `WeightFirst` が有利との結果が得られている．

表 3 は $L_{\max}$ を 3〜300 まで段階的に変化させたときの実行時間の変化の様子を調べたものである．コード挿入アルゴリズムには `WeightFirst` を用いた．**図 8** では `Normal` の実行時間を 100 として表 3 の実行時間をグラフ化した．compress と jess は $L_{\max}$ が増加するとともに実行時間が大きく減少しているが，150 を下回ることはなかった．一方 db と jack は実行時間は減少はしているものの，はじめから 110 程度の数値から大きな変化は見られない．いずれの場合も $L_{\max} = 100$ で下げ止まっているのが分かる．ここで $L_{\max} = 3$ でも db と jack の実行時間が予測より明らかに短い理由の 1 つとして，現時点での実装は対象

表 1 サンプルプログラム
Table 1 Sample programs.

プログラム	説明
compress	ファイルの圧縮と解凍
jess	Java expert shell system でパズルを解く
db	database system. データ：1 MB，クエリ：19 KB
jack	A Java parser generator

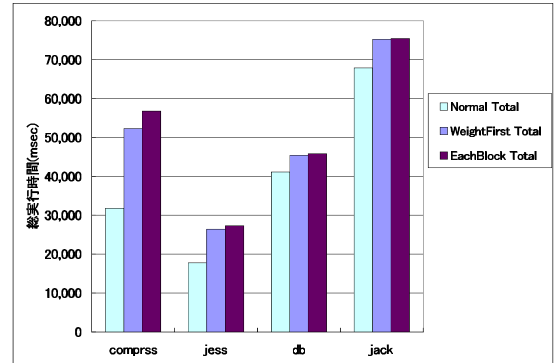

図 7 $L_{\max} = 100$ の時の総実行時間（ミリ秒）
Fig. 7 Total execution time (msec) at $L_{\max} = 100$.

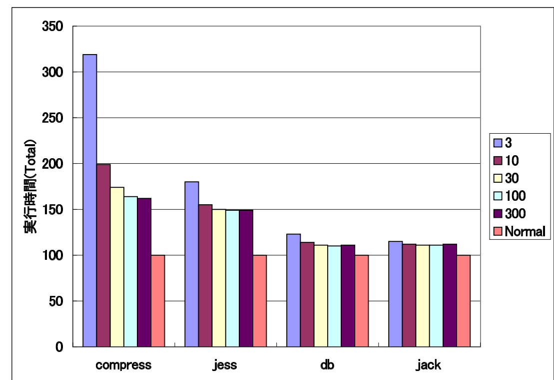

図 8 Normal の実行を 100 とし，`WeightFirst` で $L_{\max} = 3 \sim 300$ に設定した場合の総実行時間
Fig. 8 Total execution time relative to Normal (`WeightFirst`, $L_{\max} = 3 \sim 300$).

表 2 $L_{\max} = 100$ のときの実行時間（ミリ秒）
Table 2 Exec time (msec) at $L_{\max} = 100$.

	Normal		WeightFirst		EachBlock	
	Load	Total	Load	Total	Load	Total
comprss	620	31,800	1,510	52,320	1,470	56,800
jess	1,210	17,740	3,820	26,390	3,250	27,300
db	470	41,100	1,630	45,400	1,430	45,830
jack	660	67,890	4,250	75,250	3,540	75,410

表 3 `WeightFirst` で $L_{\max}$ を変化させたときの実行時間（ミリ秒）
Table 3 Exec time (msec) at $L_{\max} = 3 \sim 300$ (`WeightFirst`).

Lmax	3		10		30		100		300	
	Load	Total	Load	Total	Load	Total	Load	Total	Load	Total
compress	1,950	101,580	1,680	63,220	1,480	55,230	1,510	52,320	1,560	51,490
jess	4,160	31,890	4,250	27,460	4,270	26,580	3,820	26,390	3,910	26,450
db	1,660	50,615	1,770	46,750	1,480	45,430	1,630	45,400	1,490	45,670
jack	4,730	78,130	4,290	75,800	4,420	75,360	4,250	75,250	4,250	75,770

表 4 ファイルサイズ（byte）
Table 4 filesize (byte).

	#class	Normal	EachBlock	WeightFirst
compress	22	77,232	84,208	83,990
jess	158	449,294	485,657	485,897
db	14	71,240	78,779	78,683
jack	66	191,881	218,495	216,907

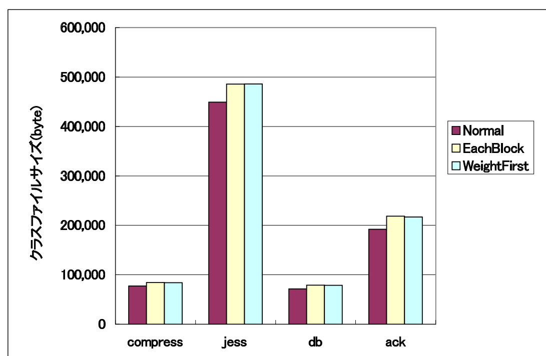

図 9 変換後のファイルサイズ（byte）（$L_{\max} = 100$）
Fig. 9 File size (byte) after conversion ($L_{\max} = 100$).

プログラムのユーザコードのみを対象としており，標準的なパッケージ（java.lang, io, util, $\cdots$）のコードは管理対象としていないことがあげられる．そのためユーザコードと標準パッケージのコードの割合の相違によっても資源管理のオーバヘッドが変わってくる．将来的には標準パッケージのコードも管理対象に含める予定である．

表 4，**図 9** では変換前後のクラスファイルサイズに関して調べた．クラスファイルサイズは JVM 内部でのメモリ消費量と相関があるので，変換によってファイルサイズが極端に大きくなるのは望ましくない．この実験では，挿入するコードは `increment` 1 カ所につきバイトコードで 2 命令であり，`EachBlock` においてもクラスファイルサイズの増大はさほど大きくなかった．

4. 関連研究

Java で資源管理を行うものに JRes [1] がある．JRes では CPU，ヒープメモリ，ネットワークの送受信が管理の対象であり，各々の資源に関して別々のアカウンティング方法を用いる．具体的には CPU 資源の消費量はネイティブコードを通じて OS の管理機能を利用しており，ヒープメモリとネットワーク消費量に関しては本研究と同様にバイトコード変換を利用している．ここで CPU 資源管理にバイトコード変換を用いないのは，オーバヘッドが非現実的な値になるというのがその理由である．しかし，本研究が示したように問題を適切に定式化し，解析の結果に基づいてバイトコード変換を施せば実行時のオーバヘッドは現実的な範囲に収まる．また，本研究では JRes と異なり JVM の実行命令数を正確にカウントするので，たとえば数千命令単位といった細かい粒度での資源管理が可能である．

J-SEAL2 は Java のモバイルエージェント向けに開発されたシステムであり，階層的な保護ドメインを構成する．文献 3) は個々の保護ドメインが使用する CPU，メモリ資源を管理する研究である．文献 3) も CPU，メモリ資源管理をバイトコード変換によって実現しており，本研究では現時点で管理の対象外であるシステムクラス（`java.lang.`，`java.util` など）も管理対象に含めている．ただし，J-SEAL2 における資源管理は，本研究の資源管理と違い，管理の精度に関する保証や，考察を行っていない．

J-SEAL2 における資源管理では，対象プログラム中に実行命令数に応じてカウンタを増加するコードのみを挿入する．資源を管理するためには実際にカウンタの値を参照し，実行命令数があらかじめ設定した上限を超えていないかチェックしなければならないが，それは独立したスレッドが定期的にチェックを行うことになっており，チェックを行う間隔については特に問題としていない．一方，本研究の資源管理は，対象プログラム中にカウンタを増加するコードとカウンタの値をチェックするコードの両方を挿入することで実現する．2 章のバイトコード変換アルゴリズムでは，カウンタを増加するコードの挿入位置についての説明のみで，チェックコードの挿入位置については特に規定していないが，適切にチェックコードを挿入することにより非常に細かい粒度での資源管理が可能である．また，カウンタを増加するためのバイトコード変換アルゴリズムを比較すると，J-SEAL2 では原則的には基本ブロックの先頭に，ブロックのサイズだけカウンタを増加させるコードを挿入し，加えていくつかの最適化を施している．一方，本研究では各基本ブロックの実行頻度の予測からブロックの重み付けを行い，その重みを考慮してコード挿入を行うため，オーバヘッ

ドの削減が期待できる．

プログラム中にコードの断片を挿入することにより，プログラムの総実行命令数や各々の関数の実行頻度やトレースを求めることはプロファイリングで普通に用いられる手法である[8),9)]．CFG を構築してノードあるいはエッジの重みをもとにコード挿入を行うなど共通点も多い．しかし，プロファイリングと比べると本研究では管理のきめが重要であり $L_{\max}$ 制約があるため，プロファイリングでのコード挿入でよく利用される Maximum Spanning Tree を構築して characteristic edge にコードを挿入するアプローチをそのまま適用することはできない．その点，非同期に発生するイベントを定期的に検査するポーリングコード挿入アルゴリズムに関する研究[10),11)]は，検査が行われる間隔に上限を設けており，より本研究との関連が深い．文献 10) の Balanced Polling の特徴は関数呼び出しへの対応である．具体的には単純なアルゴリズムではすべての関数にコード挿入が必要であるが，Balanced Polling では一定の命令数以下の関数にはできる限りコードを挿入しないように工夫している．文献 11) の Punctural Polling は Balanced Polling を拡張したアルゴリズムであり，実行命令数の正確なカウントや，ポーリング間隔の下限を保証といった特徴を備えており，実験でもオーバヘッドが低く抑えられている．我々のアルゴリズムではノードの実行頻度を活用している点が新しいが，ループの展開や interprocedural な解析への拡張など共通する課題も残されている．

5. まとめ・今後の課題

本研究では Java でユーザレベルで細粒度の CPU 資源管理を行う手法を開発した．この手法はバイトコード変換によって管理を実現するものであり，具体的には実行命令数をカウントするコードの断片をプログラム中に挿入することによって，バイトコードレベルの実行命令数を正確にカウントする．OS や JVM にはいっさい依存しないため Java のポータビリティを最大限に発揮することができる．そして実行時のオーバヘッドを軽減するために対象プログラムの構造を CFG でモデル化し，静的な分岐予測の結果を利用してノードに実行頻度に対応する重み付けを行い，この重み付き CFG 上でコード挿入のコストを定義した．この定義の下，コストを低く抑えるアルゴリズムを構築した．さらに提案したアルゴリズムに従って変換を行うプログラム変換器を実際に Java で実装した．この実装はクラスローダの機能を用いてロード時にバイトコードの変換を行っている．いくつかのサンプルプログラムで実験を行い実行時間を計測することにより，提案したアルゴリズムの有用性を確認した．その結果，バイトコード変換を施さなかった場合と比較して 11〜62%のオーバヘッドがかかることが確認された．

今後の課題にはバイトコード変換アルゴリズムの改良がある．さらなるオーバヘッドの削減に向けていくつか見直す点が考えられる．現在は intraprocedural な解析しか行っていないが，Java のようなオブジェクト指向言語では非常に小さなメソッドが頻繁に呼ばれることが少なくない．解析の結果呼び出されるメソッドを特定できれば，小さなメソッド中へのコード挿入を省略できる可能性がある．また，管理の粒度を保証しようとすると小さなループにも最低限 1 カ所はコードを挿入しなければならず，パラメータ $L_{\max}$ を大きくしてもオーバヘッドが思うように下がらない要因となっている．そのような小さなループの展開は試す価値がある．最後に CPU 資源管理以外への応用も視野にいれて研究を進める．直接的な応用は非同期なイベントの polling があり，そのほかにもスレッドを細粒度で切り替えることによるシングル CPU 上での並列プログラムのエミュレーションなどが考えられる．

謝辞 多くの有益な助言をいただいた東京大学の遠藤敏夫氏に心から感謝いたします．

参 考 文 献

1) Czajkowski, G. and von Eicken, T.: JRes: A Resource Accounting Interface for Java, OOPSLA'98, Vancouver, BC, October (1998).
2) Czajkowski, G., Chang, C., Hu, D. and von Eicken, T.: *Resource Management for Extensible Internet Servers* (1998).
3) Binder, W., Hulaas, J.G. and Villazon, A.: Resource Control in J-SEAL2, Technical Report, Cahier du CUI No.124, University of Geneva (2000).
4) Ball, T. and Larus, J.R.: Branch Prediction for Free, Technical Report, Computer Sciences Department, University of Wisconsin, Madison (1993).
5) Wu, Y. and Larus, J.R.: Static Branch Frequency and Program Profile Analysis, Technical Report CS-TR-1994-1248, Computer Sciences Department, University of Wisconsin (1994).
6) Liang, S. and Bracha, G.: Dynamic Class Loading in the Java Virtual Machine, *OOPSLA'98*, pp.36–44 (1998).
7) Cohen, G., Chase, J. and Kaminsky, D.: Automatic Program Transformation with JOIE,

1998 USENIX Annual Technical Symposium, pp.167–178 (1998).

8) Ball, T. and R.Larus, J.: *Optimally Profiling and Tracing Programs* (1992).

9) Knuth, D.E. and Stevenson, F.R.: Optimal Measurement Points for Program Frequency Counts, *BIT*, Vol.13, No.3, pp.313–322 (1973).

10) Feeley, M.: Polling Efficiently on Stock Hardware, *Functional Programming Languages and Computer Architecture*, pp.179–190 (1993).

11) 田中義純，田浦健次朗，米澤明憲：定期的なポーリングを保証するアルゴリズム，並列処理シンポジウムJSPP2001，pp.229–236 (2001).

(平成13年7月11日受付)
(平成13年12月28日採録)

速水　雄太

1978年生．2001年東京大学理学部情報科学科卒業．現在同大学大学院情報理工学系研究科コンピュータ科学専攻修士課程に在学中．主に並列分散計算の研究に従事．

田浦健次朗（正会員）

1969年生．1997年東京大学大学院理学博士（情報科学専攻）．1996年より同大学院理学系研究科情報科学専攻助手．2001年より同大学院情報理工学系研究科電子情報学専攻講師．並列・分散処理，プログラム言語に興味を持つ．ACM，IEEE，ソフトウェア科学会会員．

米澤　明憲（正会員）

1947年生．1977年Ph.D. in Computer Science（MIT）．2000年より東京大学大学院情報学環教授．超並列・分散ソフトウェアアーキテクチャ等に興味を持つ．共著書「モデルと表現」等（岩波書店），編著書「ABCL」（MIT Press）等がある．1992〜1996年ドイツ国立情報処理研究所（GMD）科学顧問，ACM Transaction on Programming Languages and Systems副編集長，IEEE Parallel & Distributed TechnologyおよびComputer編集委員等を歴任，元日本ソフトウェア科学会理事長，総合規制改革会議委員，ACM Fellow.