消費 CPU 資源を細粒度に計測するアルゴリズム

速水雄太[†]　田浦健次朗[†]　米澤明憲[†]

現在の JVM の仕様では Java プログラムの資源管理が不十分である。本研究では Java プログラムの消費 CPU 資源をユーザレベルで細粒度 (1 ~ 数千命令) に計測する．対象プログラムのバイトコード命令列に，カウンタを増加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．我々の提案する変換アルゴリズムは，重みつきの制御流グラフの解析に基づき，実行命令数を正確に数えながらオーバヘッドを小さく抑制する．また，小さなサイクルの展開や，メソッドをまたがった解析という最適化を行った。本アルゴリズムは一般的な制御流グラフを対象としているため，Java 以外のプログラムの変換にも適用できる．評価を行った結果，最適化後は 1 回あたりの平均増加回数は 2–30 命令改善し，オーバヘッドは 8%–56%程度であった．

An Algorithm for Fine-Grain CPU Resource Accounting

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

The JVM specification lacks an appropriate resource management functionality. We presents an efficient method which provides fine grain CPU resource accounting 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, with the result of a control flow graph analysis, precisely counts the number of instructions at a low overhead. We also applied some optimizations such as loop unrolling and an interprocedural flow analysis. This algorithm applies to arbitrary CFG. Therefore its use is not restricted in instrumenting Java class files. As experimental results, with the optimizations, the average number of instructions per increment improved by 2–30 instructions, and the runtime overhead was about 8–56%.

1. はじめに

JVM はクラスローダ，バイトコード検証器，セキュリティマネージャを備えており，単一アドレス空間で複数のプログラムを安全に分離する．しかし，OS が提供するような資源管理機構がなく，プログラムが互いに干渉する恐れがあることが指摘されている[1)~4)]．

計算機資源を正しく管理するためには，プログラムの消費資源量を正しく計測する必要がある．多くの OS は標準的な資源管理機構を備えており、Unix 等では getrusage などのシステムコールを呼ぶことにより手軽に実行時間を知ることができる．しかし，この場合の管理の粒度は OS 依存であり，多くの場合 10 ミリ秒単位の比較的粗粒度な管理になる．また，一般的な OS では定期的なサンプリングによって実行時間を計算しているため，サンプリング周期に同期して停止と実行を繰り返すことで管理を欺くことができる場合がある．

本研究では，対象プログラムのバイトコード命令列中にカウンタを増加するコードを挿入することにより，各スレッドの実行バイトコード命令数を正確に計測する．コードの挿入間隔を調整することで 1 ~ 数千命令の細かい粒度で CPU 消費量を計測できる．また、サンプリングのように計測が不正確になることもない．一方、JVM を改変して実行命令数の計測する方法も考えられる．インタプリタ実行であれば 1 命令ずつ計測しても問題ないが，通常は JIT コンパイルするので，中間コードあるいは実コードの段階でカウンタを増加するコードの挿入が必要になる．その場合でも，我々のコード挿入アルゴリズムは一般的な制御流グラフを対象としているので，バイトコード以外の変換にも適用できる。さらに，命令の複雑さに応じた重み付けを行うことで，実行時間を近似する拡張は容易であるが，簡単のためこの論文では述べない．

なお，実際の資源管理ではカウンタを増加するコードに加えて，得られた実行命令数をもとに管理を行うコードも挿入する．

† 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

たとえば，消費量がある一定の上限を越えたらそのスレッドを破棄するようなコード挿入は容易に実現できる．だが，本研究の特徴である細粒度な計測をより活用できるのは，非同期に発生するイベントの定期的なポーリングである．並列プログラムにおける負荷分散のメッセージ到着などを頻繁に検査する必要がある場合，プログラム中にポーリングを挿入して自発的に検査するのが有効である．実行命令数を正確に計測することによりポーリングの間隔を保証しつつ過剰なポーリングを抑制することが可能である[5]．

ただし，コード挿入は実行時のオーバヘッドの増大をともなう．以前，我々は正確なカウントと細粒度なカウントを保証しつつ，オーバヘッドを抑制するアルゴリズム WeightFirst を提案した[6]．このアルゴリズムは，対象プログラムの重みつき制御流グラフを解析することにより冗長なコード挿入を削減する．ただし，このアルゴリズムには以下のような制限があった．1) コードの挿入は頂点のみ．制御流の合流点では辺にコードを挿入したほうが低コストの場合も多いが，頂点にしか挿入できない．2) メソッド内の解析限定．制御流グラフは各メソッドで独立しているため，他のメソッドの解析結果を組み合わせることができない．3) サイクルの展開を行わない．サイクルには最低限一カ所はコードを挿入する必要があるため，小さなサイクルほどカウンタを増加する頻度が高くなる．

今回，我々は制御流グラフを改変することでこれらの欠点を改善する．まず，メソッド中の最も内側のサイクルを検出し適切な段数だけ展開する．次に，動的なメソッド呼出などのコード挿入が困難な辺以外は辺を分割して中間点を導入する．辺へのコード挿入はこの中間点へのコード挿入として実現できる．最後にメソッド呼出命令を含む頂点から，呼び出される可能性のあるメソッドに辺を張る．以上のように改変した制御流グラフを WeightFirst の入力として与えることにより，WeightFirst そのものを改変することなく自然にアルゴリズムを拡張する．

我々は拡張したアルゴリズムを実装し，評価を行った．いくつかのベンチマークで動的なカウント数を計測した結果，最適化によって平均実行命令数が 2–最大 30 命令改善した．また，実行時間を計測した結果，実行時のオーバヘッドは 8–56% であった．

2. バイトコード変換

2.1 概　　要

我々は図 1 に示す手順でバイトコードを変換する．Java では java.lang.ClassLoader を拡張することで

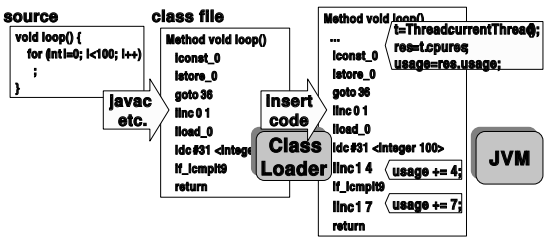


図 1　バイトコード変換変換例

```
class java.lang.Thread {
    CPURes cpures; // カウンタオブジェクト
    ..  // コンストラクタで cpures を初期化
}
class CPURes{
    int usage; // 現在の総実行命令数
}
```

図 2　java.lang.Tread へのカウンタ付加

ユーザ定義のクラスローダを作成できる．我々はバイトコード変換器をクラスローダに組み込み，管理の対象となるユーザプログラムが JVM にロードされた時点で変換する．ロードしたバイトコードはメソッド単位で解析を行ない，資源管理コードを挿入する．変換後のバイトコードは，defineClass() メソッドによって Class クラスのインスタンスに変換され，JVM で実行される．

java.lang パッケージや java.io パッケージに含まれるシステムクラスについても同様な変換を行なうが，これらのクラスは JVM 組み込みの bootstrap クラスローダによってロードされるべきものであり，ユーザ定義のクラスローダでロード時に変換することはできない．したがって，これらのクラスについては予め変換を施しておく．

2.2 カ ウ ン タ

本研究では各スレッドの実行命令数を正確にカウントする．そのためは実行命令数を保持するカウンタが各スレッドに 1 つ必要になる．そこで java.lang.Thread クラスにカウンタ CPURes への参照を付加した（図 2）．CPURes には int 型のフィールド usage があり，ここに実行命令数を記録する．また，Thread クラスのコンストラクタにもコードを挿入して，スレッドの初期化時にカウンタオブジェクトも同時に初期化する．なお，Thread クラスもシステムクラスに含まれているため，プログラム実行以前にあらかじめこれらの変換を施しておく．

2.3 対象プログラムの変換

資源管理の対象となるプログラムには，フィールド usage を増加するコードを挿入する．変換はメソッド単位であり，まず，メソッドの先頭で，現在実行中のスレッドのカウンタをローカル変数にキャッシュする．カウンタの増加は，$-127 \sim +128$ までなら iinc 1 命令で実行できる．また，メソッド呼出しから復帰した後は，再び実行命令数をローカル変数にキャッシュし，メソッド呼出しの直前とメソッドからの復帰前にはローカル変数から usage に実行命令数を書き戻す．

単純には 1 命令おきに iinc 命令を挿入すれば正確なカウントは可能である．しかし，この方法では実行時のオーバヘッドが膨大になり非現実的である．本研究では，オーバヘッドを削減するため，ある程度まとめてカウンタを増加する．我々は、メソッドを重みつき制御流グラフでモデル化する．単純には基本ブロック単位で増加する方法が考えられるが，すべての基本ブロックにコードを挿入するのはコストが高い．我々は静的な分岐予測で基本ブロックの重みを計算し，その重みに基づいてよりよいコード挿入を行なう．これらの変換ついては 3 章で詳しく述べる．

2.4 メソッドのオーバーロード

メソッド呼出の先頭で毎回 currentThread() を実行するのは重い処理である．しかし，カウンタオブジェクトへの参照を直接メソッドの引数として渡せば currentThread() の呼び出しは不要である．そこで，すべてのメソッドをオーバーロードし，オーバーロードしたメソッドのシグニチャは引数を一つ増やしてカウンタオブジェクトへの参照を付加する．

オーバーロードされた本来のメソッドは，オーバーロードしたメソッドに実行を委譲するように書き換える．この場合は，メソッドの先頭で currentThread() を実行する必要がある．また，安全のためもともとのメソッドのシグニチャは削除しない．native コードは変換の対象外であるため，java.lang.Runnable.run() のように native メソッドから呼び出される可能性があるメソッドは，呼び出し元の native コード中でカウンタオブジェクトへの参照を渡すことができないからである．

基本的な手順は以上の通りだが，メソッドのアクセス修飾子に応じて若干の相違がある．abstract なメソッドはコードを持たないため，オーバーロードはするがコードの書き換えは不要である．また，native なメソッドはコードを変更することができないが，オーバーロードは必要である．オーバーロードしたメソッドからは native 修飾子を取り除き，もともとの native なメソッドに実行を委譲するコードを挿入する．

3. コード挿入アルゴリズム

3.1 目　　標

我々は各 java スレッドの実行命令数をバイトコードレベルで正確に計測する．単純には，1 命令実行したらカウンタを 1 だけ増加するようなコード挿入を行えば，カウンタの値を常に正確に保つことは可能である．ただしこれは非常に重い処理であり，バイトコード変換で実現するにはコストがかかりすぎる．そこで，条件を緩和して，ある程度まとめてカウンタを増加する．

しかし，まとめてカウンタを増加することはカウンタの更新頻度が落ちることを意味しており，カウンタは誤差を含むものになる．この誤差が無制限に大きくなるのは都合が悪いので，カウンタの値に関して制約を設ける．準備として，実行命令数を I_e，カウンタの示す値を I_c とおく．I_e はプログラムの実行中常に 1 ずつ増加するのにたいして，I_c は明示的にカウンタを増加するまで更新されない．定期的なカウンタの更新を保証するために，カウンタの誤差 $I_e - I_c$ に上限を設定し，ある任意の非負定数 $L_{\max}$ までとする．以降，この制約を $L_{\max}$ 制約とよぶ．

$L_{\max}$ 制約

$$0 \leq I_e - I_c < L_{\max}$$

理想的には，$L_{\max}$ 命令に一度カウンタを $L_{\max}$ だけ増加すればオーバヘッドは最小になる．しかし，現実のプログラムはにおいては，一回あたりのの増加は $L_{\max}$ 未満になることが大半である．たとえば長さが $L_{\max}$ 未満のサイクルでは，一度に増加できる値の最大値はそのサイクルの長さまでに制限される．

コード挿入による実行命令数の計測は，従来のプロファイリングでも研究されており Knuth[7] や，Ball[8] らは効率的に実行命令数を計測するアルゴリズムについて考察している．しかし，本研究では正確な計測に加えて $L_{\max}$ 制約があるという点でこれらの研究とは異なる．我々の提案するコード挿入アルゴリズムは入力として対象プログラムと $L_{\max}$ の値を受取り，正確なカウントと $L_{\max}$ 制約を満たすコード挿入を実現する．その上で，できるだけ実行時のオーバヘッドが小さく抑える挿入を実現することが目標である．

3.2 準　　備

プログラム中にコードを挿入するための準備として対象プログラムを重み付き制御流グラフでモデル化する．

定義

重み付き制御流グラフ $G = (V, E)$

- 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．
- $E \subset \{(i,j)|i,j \in V\}$ 有向辺集合．頂点間の潜在的な制御の移動を表現．

頂点の重みは制御流グラフの構築時に，静的な解析によって実行頻度を予測する．基本的には，まず制御流グラフ中のループ構造を検出し，いくつかのの経験則に基づき条件分岐があったらそれぞれの分岐先の確率分布を求める．次に，計算した確率分布を元に制御流を伝搬させて各辺の実行頻度を見積る．頂点の実行頻度は辺の実行頻度から計算できる．

本研究では文献 9)，10) 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辺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．

次に，挿入されたコードにかかるコストを定義する．コストはカウンタを増加するコードを挿入したノードの重みの総和で定義する．コード全体のコスト C_{total} は各ノード $v(\in V)$ におけるコストを C_v とおいたときの C_v の総和であり，

$$C_{total} = \sum_{v \in V} C_v$$

となる．$L_{\max}$ 制約があるため，$L_{\max}$ よりも長いノードには複数個所のコード挿入が必要である．そのため $C_v = w_v$ が常になりたつとは限らない．C_{total} は実行時のオーバヘッドに対応しており，正確なカウントと $L_{\max}$ 制約を満たしつつ，C_{total} を下げることが目標である．

3.3 WeightFirst

正確な計測を満たすだけの単純な変換アルゴリズムであれば，各基本ブロックの末尾にそのブロックの長さだけカウンタを増加するコードを挿入すればよい．ただし，長さが $L_{\max}$ 以上のブロックに関しては $L_{\max}$ 制約を満たすためには複数個所にコードを挿入することが必要である．この場合のコストは

$$C_{total} \geq \sum_{v \in V} w_v \qquad (1)$$

となり，$\sum_{v \in V} w_v$ がコストの下限となる．

しかし，すべての基本ブロックに一律にコードを挿入するのは明らかに無駄である．実際の制御流グラフでは，コードの挿入個所と，カウンタの増加量を調節することで，コード挿入が不要になる頂点も少なくない．そうすればコストを式 1 の下限より下げることが可能である．我々は以前，このような冗長なコード挿入を削減するアルゴリズム WeightFirst を提案した[6]．このアルゴリズムは頂点を重みの順に整列し，重みの大きい頂点からコードの挿入の有無を決定する．

WeightFirst は，頂点へのコード挿入に関する以下の性質を利用している．ある頂点 v の先頭と末尾におけるカウントの誤差をそれぞれ $\varepsilon_v^{\text{in}}, \varepsilon_v^{\text{out}}$ とおくと，その頂点へのコード挿入コスト C_v は $\varepsilon_v^{\text{in}}, \varepsilon_v^{\text{out}}, L_{\max}$ と頂点の長さ l_v の関数で表される．逆に $\varepsilon_v^{\text{in}}$ あるいは $\varepsilon_v^{\text{out}}$ の値の一方が決まれば，C_v を最小にするように他方の値を決定することで，頂点 v へのコード挿入コストを最小限に抑えることができる．

WeightFirst は重い頂点から順に $\varepsilon_{\cdot\cdot}^{\text{in}}, \varepsilon_{\cdot\cdot}^{\text{out}}$ の値を決定していき，最後に，すべての頂点について $\varepsilon_{\cdot\cdot}^{\text{in}}, \varepsilon_{\cdot\cdot}^{\text{out}}$ の値が決定した時点で停止する．このように，重い頂点へのコード挿入をできるだけ避けるように変換することで，たとえ重みの小さい頂点へのコード挿入個所が増えたとしても，全体のコストを削減できる．

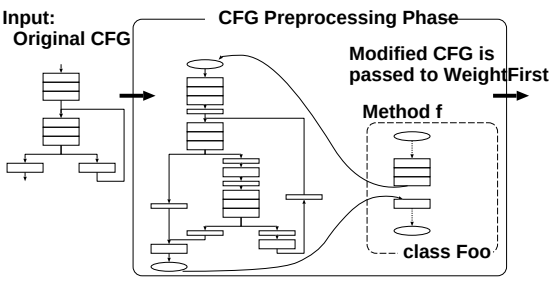
3.4 WeightFirst の問題点

WeightFirst は単純な挿入アルゴリズムよりは優れているが，いくつかの限界があった．第 1 にコードの挿入は頂点限定であり，辺には挿入しない．これはコードを挿入するのが困難な辺が存在するためである．辺はもともとのプログラムでは命令 I_s から I_t への潜在的な制御の移動であった．ここで I_s は分岐やメソッド呼出しあるいは例外の発生など，制御流の変更に関わる命令である．辺 (I_s, I_t) へのコード挿入は、まず命令列中の適切な個所にそのコードを挿入し、いったん I_s から挿入したコードに実行を移し、その後で I_t に制御を戻すことで実現できる．I_s が条件分岐の場合は分岐先の変更は容易であるが、挿入が難しいのは I_s が動的なメソッド呼出しや例外を発生する命令の場合である．これらの命令では制御の移動先である I_t が明示されておらず，どの I_t にとぶかは実行時のスタックをもとに JVM が動的に決定する．これらの辺へのコード挿入をバイトコード変換で実現するためには，instanceof 命令と条件分岐を組み合わせた非常に煩雑なものになる．しかも，プログラムの実行とともに新しいクラスがロードされると，新しい辺を付け加える必要もでてくる．辺の追加はバイトコード変換のやり直しが必要であるが，すでにロードされたクラスへの変更は不可能である．いったんクラスをアンロードしてロードしなおすにしても，そのクラスが到達可能である限り自由にアンロードすることは許されない．

第 2 の制限はサイクルを展開しないことである．$L_{\max}$ 制約はグラフ中のすべてのサイクルに最低限一カ所はコードを挿入することを要求する．当然，あるサイクルが実行されている間，まとめて増加できる命令数はそのサイクルの長さまでに制限される．ゆえに、短いサイクルではカウンタの増加頻度が大きくなり、深刻なオーバヘッドを被ってしまう．この問題にはサイクルの展開が有効である．短いサイクルを適当な段数だけ展開することによりオーバヘッドの改善が期待できる．しかし，サイクルを無制限に展開してはコード量が急増し，消費メモリや命令キャッシュのミスの増大などを招き、かえって逆効果になりかねない。展開するサイクルの選択と，展開段数は十分な考察が必要である．

第 3 の制限は，制御流グラフがメソッド単位で独立していることである．本来ならばメソッドの呼出元と呼出先を，メソッドをまたがった辺でつなぐ必要がある．さもないとカウントが不正確になる可能性がある．もし，プログラム全体で 1 つの巨大な制御流グラフを構築するのであれば，容易にメソッドをまたがった辺を張ることができ，そのグラフに対して WeightFirst を適用すればよい．しかし，本研究では JVM がロードしたクラスを順次変換しており，一度に変換できるのは 1 つのクラスに限られる．そのため，将来ロードされるクラスのメソッドには直接，辺を張ることができない．メソッド呼出でカウントが不正確になるのを避けるため，メソッド呼出しの前後に一カ所づつコードを挿入していた．これは単純なフィボナッチのように，メソッド呼出しが頻発するプログラムではオーバヘッド増大の大きな要因となる．このように，将来ロードされるクラスのメソッドに直接は辺をはることはできないが，間接的に辺を張ることは可能である．この辺を利用すれば，あるメソッドの解析結果を将来ロードされるメソッドの解析に利用できる．

これらの問題はいずれも制御流グラフの変換により対処できる．サイクルの展開は制御流グラフ上のサイクル展開そのものであり，辺へのコード挿入は辺の分割で実現できる．メソッド間解析は新たに入口点 (entry node) と出口点 (exit node) という特殊な頂点を介してメソッドの呼出元と呼出先に辺を張る．変換した制御流グラフを WeightFirst の入力として与えることで，WeightFirst 本体は改変せずに変換アルゴリズム全体を拡張する．次節以降ではこれらの変換の詳細について説明する．



1.Cycle Unrolling
- **Find inner most cycle.**
- **Unroll it *n* times.**
  (size of cycle after unrolling) < Lmax

2. Enable Insertion to Edge
- **Split edges so that WeightFirst can insert a code fragment into an edge.**
  (except for Method Invocation and Exceptions)

3. Interprocedural Analysis
- **Connect a call node with the entry node.**

図 3　制御流グラフの拡張

4. WeightFirst の拡張

4.1 概　　要

我々は制御流グラフを変更することでこれらの問題に対処する．静的な分岐予測により重み付けしたあと，図 3 に示すように制御流グラフを変換する．まず最初にサイクルを展開する．グラフ中にサイクルが存在する場合，長さ $L_{\max}$ を越えない程度に展開する．もし複数のサイクルが存在した場合は，もっとも内側のサイクルを展開する．次に，WeightFirst が辺へもコードを挿入できるように，辺を分割する．分割した辺には長さ 0 の中間点を挿入することにより，頂点にしかコードを挿入できない WeightFirst でも辺へのコード挿入が可能になる．これらの変換と同時に，メソッドをまたがる辺を張ることによりメソッド間解析を行う．クラスローダに組み込んでロード時に変換を行う都合上，メソッドをまたがる辺を直接張ることはできないが，入口点と出口点を介して間接的に辺を張る．この解析で WeightFirst はメソッド呼び出し前後に必ず挿入していたコードを削減できる．

以上の変換は WeightFirst に制御流グラフを入力する前の前処理にあたり，変換後のグラフを WeightFirst に入力することで最終的なコード挿入結果を得る．

4.2 辺へのコード挿入

動的なメソッド呼出や例外の発生に関する辺はコード挿入が困難であり，WeightFirst は辺へのコード挿入を行なわなかった．しかし，辺へのコード挿入を一律に禁止するのではなく，条件分岐令のようにコード挿入が比較的容易な辺にはむしろ積極的にコードを挿

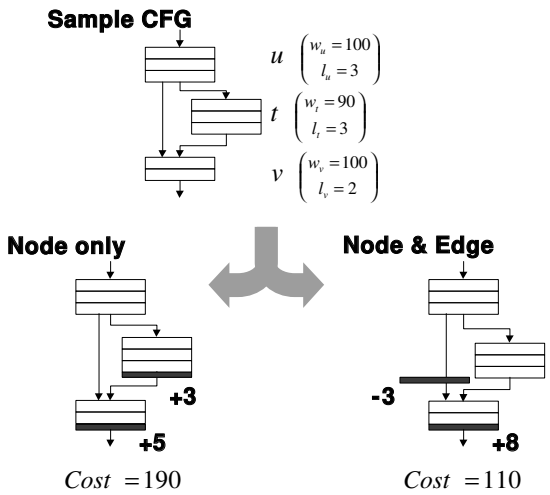


図 4　辺へのコード挿入が必要な例

入すべきである．図 4 は辺へのコード挿入により，コストを削減できる制御流グラフの一例である．簡単のためこの例では $L_{\max}$ は十分大きいとする．グラフは u, t, v の 3 つの頂点からなり，それぞれに重みと長さが定義されている．図 4 左の Node only はコード挿入を頂点に限定した場合，右の Node & Edge は辺にもコード挿入を許した場合，それぞれ最適な変換結果を示した．どちらの変換でも静的なコード挿入は 2 個所であるが，Node & Edge では重み 90 の頂点 t のかわりに重みが 10 の辺 (u, v) にコードを挿入できるため，その分コストが低く抑えられている．

上の例が示すように辺へのコード挿入は重要である．そこで我々は辺にもコードを挿入するように制御流グラフを変換した．具体的には，コードを挿入してもよい辺を分割して間に中間点を作る．一方，挿入が困難な辺は分割しない．新たに作成した中間点は，辺の重みを引き継ぎ，長さ 0 として扱う．この時点では，実際に中間点に対応するダミーコードを挿入したり，辺の張り替えは行わない．

そして，変換した制御流グラフを WeightFirst の入力として与える．WeightFirst はこれらの中間点と通常の頂点を区別せずにコード挿入を行う．もし中間点に挿入しないのであれば，何もする必要はない．挿入するのであれば，命令列の末尾に挿入したいコードを付け加え，辺の張り替えを行う．

現在，コード挿入が容易である条件分岐とスイッチに関してはすべての辺を分割している．しかし，本質的に分割する必要がある辺は，複数の出力辺をもつ頂点と複数の入力辺をもつ頂点を結ぶ，いわゆるクリティカルな辺のみである．それ以外の辺は先行点の末尾，あるいは後続点の先頭へのコード挿入に置き換えることができる．ゆえに，はじめからクリティカルな辺のみを分割すれば，より少ない解析時間で同一のコード挿入達成できると考えられる．

4.3　サイクルの展開

実行時のオーバヘッドを増大させている主要な原因の一つは，制御流グラフにおけるサイクルである．$L_{\max}$ 制約を満たし，かつ実行命令数を正確に計測するためには，どんなに小さなサイクルにも最低限一つはコード挿入が必要である．そのため，小さなサイクルが頻繁に実行される場合は，$L_{\max}$ の値をいくら大きくしても動的なカウンタの増加回数はほとんど減少しない．

この問題にはサイクルの展開が有効である．小さなサイクルを適切な回数だけ展開することによりサイクルのサイズが増大し，カウンタ増加の頻度を削減できる．ただし，サイクルの展開はコードサイズの増大を招くため，むやみに多用するわけにはいかない．そこで以下では，展開の対象となるサイクルの選択及び，サイクルの展開手順，展開段数に関する戦略について検討する．

4.3.1　展開するサイクルの選択

サイクルの展開手順自体は，グラフ理論で定義される狭義のサイクルから，for 文や while 文に対応するループでも同様に展開できるので，これらをまとめてサイクルとよぶ．実際は，制御流グラフを構築した後の静的な分岐予測の段階で natural loop[9] を計算しており，この natural loop 単位で展開する．natural loop はサイクルを構成する頂点集合と，すべての頂点集合を支配 (dominate) するループヘッドからなる．狭義のサイクル単位で展開すると，中に if 文を含むような for 文を複数のサイクルとして展開してしまう．natural loop ではこれらのサイクルをひとまとまりのサイクルとしてまとめて展開できる．

展開するサイクルの選択基準にはサイクルの包含関係，重み，長さなどが考えられる．サイクルの包含関係とはサイクルのネストに関する関係であり，いわゆる多重ループのように，ネストしたサイクルが存在する時は，それらを外側のサイクルと内側のサイクルに分類できる．natural loop の包含関係は頂点間の支配関係を調べることでわかる．サイクルの重みとはサイクルの (予想) 実行頻度である．そして，サイクルの長さはもっとも長い閉路の経路長とする．

展開するのが望ましいサイクルは，実行頻度が大きく，なおかつ短いサイクルである．頻繁に実行されるループを展開するのは，プログラム全体のオーバヘッ

ド削減に貢献するからであり，小さなサイクルを展開するのは，小さなサイクルほど展開の効果が顕著に現れるからである．

本研究では，メソッドがサイクルを一つしか含まない場合はそのサイクルを展開するが，複数のサイクルを含んでいる場合はもっとも内側のサイクルを一つだけ展開する．それは，内側のサイクルほど上に述べた条件を満たしているからである．ただし，サイクル間に包含関係が存在しない場合もあり，そのときはより重いサイクルを展開する．

4.3.2 展開の手順

サイクル展開は，プログラムの意味を変えずにサイクルの長さを整数倍にする．m 段展開した後のサイクルの長さは展開前の $m+1$ 倍になる．ただし，もとになる小さなサイクルをつなぎ合わせるには，どこか一つの辺でサイクルを切断してそこからつなぎ合わせる必要がある．切断する辺には任意性があるが，サイクルを破壊する辺でなければならず，バックエッジが適当である．

展開手順は，頂点の複製段階と，枝張り段階からなる．頂点の複製段階では対応する基本ブロック製を複製し，メソッドの末尾に付け加える．ただし，try による例外処理が設定されている場合は，あたらに追加したコードが設定範囲からはずれてしまう．ゆえに，もとのサイクルと隣接するように複製を挿入する，あるいは例外処理のエントリを追加しなければならないが，まだ実装できていない．

次の枝張り段階では，まず，サイクルをバックエッジで切断し，それを次の複製のループヘッドにつなぎかえることで一つの大きなサイクルを作る．最後に，プログラムの意味を変えないように，複製された頂点から，複製以前から存在していた頂点への枝を適宜付け加える．

4.3.3 展開段数に関する考察

展開後対象のサイクルが決定したら，次に，サイクルの展開段数を決める．もともと短いサイクルでは，数段展開した後でも長さが $L_{\max}$ よりはるかに小さく，展開の効果が不十分である．しかし，長さ $L_{\max}$ を越える展開は，コードサイズの増大に見合ったコスト削減は困難であると予想できる．以下ではコードサイズと予測されるコストから，最適な展開段数について考察する．

図 5 に示すサイクルは，途中に分岐を含まない単純なサイクルである．このサイクルの長さを l，実行頻度を n とする．$L_{\max}$ 制約と正確なカウントという条件から，このサイクルには最低限 $\left\lceil \frac{l}{L_{\max}} \right\rceil$ 個所のコード挿入が必要である．ゆえに展開前の予想コスト C_0 は次のようになる．

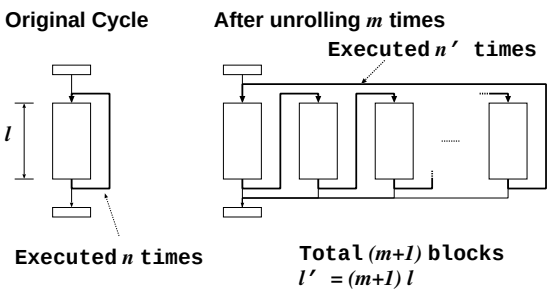


図 5 サイクルの展開例

$$C_0 = n \left\lceil \frac{l}{L_{\max}} \right\rceil$$

では次に，サイクルを m 段展開した場合のコスト C_m を見積もる．展開後のサイクルの長さを l'，実行頻度 n' とおくと，m 段展開するので $l' = (m+1)l$．また，実行命令数は展開前後で等しいので，$ln = l'n'$ が成り立つ．ゆえに，

$$\begin{aligned} n' &= \frac{ln}{l'} = \frac{ln}{(m+1)l} = \frac{n}{(m+1)} \\ C_m &= n' \left\lceil \frac{l'}{L_{\max}} \right\rceil \\ &= \frac{n}{(m+1)} \left\lceil \frac{(m+1)l}{L_{\max}} \right\rceil \\ &= \frac{nN}{(m+1)} \\ &\quad \left(\begin{array}{l} \dfrac{(N-1)L_{\max}}{l} < m+1 \leq \dfrac{NL_{\max}}{l}, \\ N = 1, 2, .. \end{array} \right) \\ \therefore \frac{nl}{L_{\max}} &\leq C_m < \frac{nl}{L_{\max}} \cdot \frac{N}{(N-1)} \quad (N = 1, 2, ..) \end{aligned}$$

C_m が最小となるのは，

$$m+1 = \left\lfloor \frac{NL_{\max}}{l} \right\rfloor$$

のときであり，このとき展開後のサイクルの長さは，

$$l' = \left\lfloor \frac{NL_{\max}}{l} \right\rfloor l$$

となる．これは展開後のサイクルの長さが，$L_{\max}$ の整数倍を越えない範囲で，最大限に展開したときコストが極小となることを示している．コードサイズの増加との兼ね合いから，展開後の長さが $L_{\max}$ を越える展開は行わない．

以上サイクル展開をまとめると，メソッド中の最も内側のサイクルを 1 つ，$L_{\max}$ を越えない範囲で最大

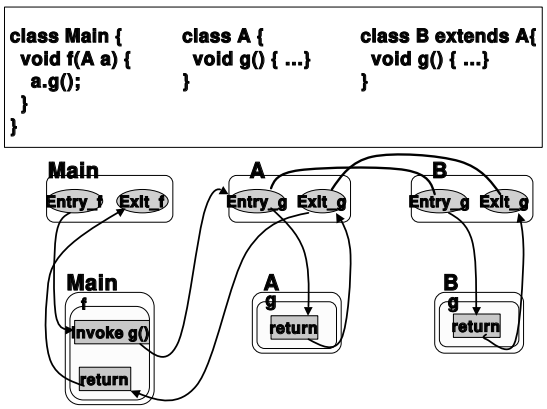


図 **6** 入口点と出口点

限に展開する．ただし，$L_{\max}$ が数千のオーダーになると，展開後の長さも $L_{\max}$ 程度になってしまうため，巨大な $L_{\max}$ については現実的な範囲内で展開を打ち切る必要があるだろう．

4.4 メソッド間解析

サイクルと並んで重要なのがメソッド呼出しである．これまで WeightFirst がメソッド内の解析にとどまっていたのは，制御流グラフが各メソッドで独立していたのが原因であった．そこで，メソッド呼出命令と，呼び出される可能性のあるメソッドを辺で結合することによりメソッドをまたがった解析を行なう．

しかし，プログラムを変換できるのはバイトコードが JVM にロードされてから，defineClass() メソッドで Class クラスのインスタンスが生成されるまでであり，一度に 1 つのクラスしか変換できない．そのため，将来ロードされるクラスのメソッドへ直接，辺を張ることは不可能である．また，バイトコード変換レベルでは，一度変換したメソッドを再度変換することも許されない．

我々は各メソッドに入口点と出口点を用意することで，間接的にメソッドをまたがった辺を張ることで，正しくコードを挿入する．以降，これらの特別な頂点と辺の張り方について説明する．

4.4.1 入口点と出口点

入口点と出口点は，文字通りそれぞれメソッドの入口と出口に対応し，長さ 0，コードは挿入できない特別な頂点として扱う．入口点からメソッドの先頭の頂点へ有向辺を張る．同様に，復帰命令から出口点についても同様に有向辺を張る．複数の復帰命令が含まれている場合は辺が複数存在してもよい．

さらに，メソッド呼出し命令を含む頂点を呼出点，メソッド呼出からの戻り番地に相当する命令を含む頂点を復帰点と呼ぶことにする．メソッド呼出命令には静的なものと動的なものがあるが，いずれも呼び出すメソッドのクラス名とシグニチャを指定する．静的なメソッド呼出では，クラス名とシグニチャで呼び出すメソッドを一意に特定できるので，呼出点からそのメソッドの入口点へ辺を張り，同様に出口点から復帰点へも辺を張る．一方，動的なメソッド呼出の場合は指定したクラスのサブクラスのメソッドが呼ばれる可能性もあるため，一般には複数のメソッドの入口点に辺を張る必要がある．しかし，この辺を張る時点では，将来ロードされるクラスの情報はわからないので必要な辺をすべて張ることはできない．

具体的には，図 6 のような場合が考えられる．この図はクラス Main から実行され，クラス A，B の順にロードされたときの制御流グラフを表している．Main クラスのメソッド f を変換している段階ではクラス B はロードされておらず，B のメソッド g の入口点に辺を張ることができない．

このままでは，本来必要な辺を欠いているため，計測が不正確になる可能性がある．そこで，入口点と入口点，出口点と出口点を結合する辺を導入する．あるクラスのメソッド m が基本クラスのメソッド m_0 を拡張しているとき，m と m_0 の入口点同士と出口点同士をそれぞれ辺で結合する．図では，クラス B のメソッド g がクラス A のメソッド g を拡張しているので，これら 2 つのメソッドの入口点と出口点をそれぞれ結合している．一方，動的なメソッド呼出では可能性のあるすべての入口点へ辺を張るのではなく，指定されているクラスの，指定されているメソッドの入口点にのみ辺を張る．たとえ指定されているメソッドが abstract であってもかまわない．重要なのは，呼出点から入口点を何回かたどることにより，実際に呼び出されるメソッドの入口点に到達できることである．

WeightFirst は呼出先メソッドを単一の頂点であるかのように扱う．すなわち，メソッドの入口点における誤差 $\varepsilon^{\text{in}}_{entry}$ とメソッドの出口点における誤差 $\varepsilon^{\text{out}}_{exit}$ のみが重要であり，呼出先メソッドの制御流グラフの解析までは立ち入らない．メソッドをまたがる辺は，呼び出される可能性のあるメソッドの入口点と出口点で $\varepsilon^{\text{in}}_{entry}$ と $\varepsilon^{\text{out}}_{exit}$ の値を共有するために用いる．呼出先メソッドの制御流グラフを解析しないのは，次のような理由による．呼出先のメソッドがすでに解析・変換済であれば，既に決定している $\varepsilon^{\text{in}}_{entry}$ と $\varepsilon^{\text{out}}_{exit}$ の値を呼出元の解析に利用するよりなく，また，未解析であれば，将来実行されるかどうか未確定のメソッドを再帰的に解析するよりも，呼出元の解析結果を $\varepsilon^{\text{in}}_{entry}$

名称	内容
fib	naive な再帰で fib(40) を計算
linpack	行列で連立一次方程式を解く
compress	ファイルの圧縮と解凍

表 1　ベンチマークの種類

と $\varepsilon_{exit}^{\mathrm{out}}$ に書き込んで解析を切り上げるだけで，十分メソッド間解析の効果が得られる．

これらの辺をはる必要があるのは，新しいクラスがロードされたときと，メソッド呼出命令を解析するときである．手順を簡単に説明する．クラスが JVM にロードされると，メソッドを変換する前に，そのクラスのスーパークラスとスーパーインターフェースを参照する．もし，ある基本となるメソッドを拡張しているのであればそのメソッドの入口点と出口点同士を結ぶ．また，メソッド呼出で指定されているクラスがまだロードされていない場合は，クラスのロード時と同様に，そのクラスのスーパークラスとスーパーインターフェースを参照する．そして指定されているメソッドの基本となるメソッドがみつかれば辺をはる．

5. 実　　験

我々は 3, 4 節で述べた変換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をもちいて評価を行なった．それぞれのプログラムの簡単な説明を表 1 に示す．実験に用いた計算機環境は以下のとおりである．

IBM Netfinity 7100
CPU: Pentium III Xeon 700MHz x 4
MEM: 4GB
OS: Debian GNU/Linux 3.0

なお，Java の処理系は SunJDK1.3.1 のものを用いた．

まだ実装に不完全な部分もあり，以降の実験では java.lang などのシステムクラスにはカウンタを増加するコードを挿入していない．

5.1 平均増加量

最初の実験では総実行命令数と総カウンタ増加回数を計測する．この両者の比から，一回あたりの平均カウンタ増加量を得る．この値が大きいほどカウンタ増加の頻度が下がることになる．ただし，$L_{\max}$ 制約があるため，この値は $L_{\max}$ を越えることはないが，理想的なプログラムではほぼ $L_{\max}$ に近い値が得られる．

本研究では実行命令数を正確に計測するので，総実行命令数の計測は容易である．どの変換アルゴリズムを用いても $L_{\max}$ の誤差をのぞいて一致する．一方，カウンタの動的な増加回数は変換アルゴリズムによって増減する．今回の実験では $L_{\max} = 100$ のもと，EachBlock から WF(All) の 6 種類の変換アルゴリズムについて実験を行った．EachBlock は各基本ブロックの末尾でカウンタを増加する単純なアルゴリズムである．また，WF(basic) は 4 章で述べた各種最適化を行う以前の WeightFirst である．その他は WeightFirst に最適化を適用したものであり，Edge は辺へのコード挿入，Cycle はサイクル展開，InterP はメソッド間解析，All はこれらすべてを用いたことを意味する．表 2 に計測結果をまとめた．

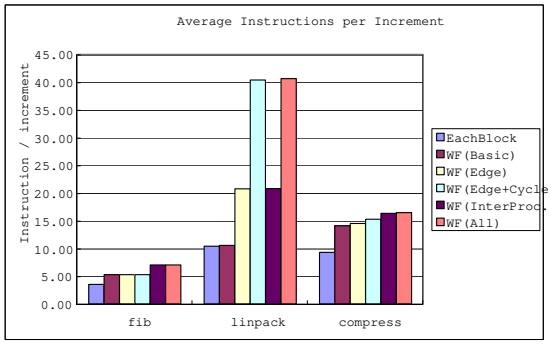


図 7　一回あたりの平均増加量 ($L_{\max} = 100$)

図 7 は得られた平均増加量をプロットしたものである．まず，fib の結果をみると WF(Basic) と WF(InterP) で値が改善している．変換後のクラスファイルを調査すると，それぞれ挿入コードを静的に 1 つ削減していることが確かめられた．また，$L_{\max} = 100$ にも関わらず，全体的に値が小さい．我々の現在行なっているメソッド間解析は，主にメソッド呼びだし前後のコード挿入を削減するが，fib のように小さなメソッドが再帰的に呼ばれるプログラムでは効果が薄い．fib の平均増加量をさらに改善する方法に，メソッドのインライン化などが考えられるが，これはさらにコード量が増えるという問題がある．

次に linpack の結果をみる．linpack はサイクルの実行回数が多いため，サイクル展開による改善が期待できる．結果をみると，たしかにサイクル展開が効いていることがわかる．サイクル展開後の平均増加量は 40.3 であり，展開前の WF(Basic) の約 4 倍である．また，サイクルの展開以外の最適化でもそれぞれ平均増加量が改善した．

最後に compress の結果をみる．compress はもともと比較的長いサイクルがよく実行される傾向があり，最適化のない WF(Basic) でも他のベンチマークよりも良好な結果が得られている．それでも，最適化を加えるといずれの場合も平均増加量の改善が見られた．linpack のようにサイクル展開によって劇的な改善が

	T	EachBlock		WF (Basic)		WF(Edge)		WF (Edge+Cycle)		WF(Edge+InterP)		WF(All)	
		C	T/C	C	T/C	C	T/C	C	T/C	C	T/C	C	T/C
fib	229,863	65,677	3.50	43,786	5.25	43,786	5.25	43,786	5.25	32,839	7.00	32,839	7.00
linpack	7,189,120,738	692,248,356	10.39	681,714,832	10.55	346,875,577	20.73	178,086,328	40.37	345,868,552	20.79	177,079,301	40.60
compress	13,403,736,376	1,446,074,371	9.27	952,359,701	14.07	925,565,272	14.48	878,692,597	15.25	821,665,261	16.31	815,861,876	16.43

表 2　$L_{\max} = 100$ のもとでの，総実行命令数 (T) と動的な増加回数 (C)，及び一回あたりの平均増加量 (T/C)

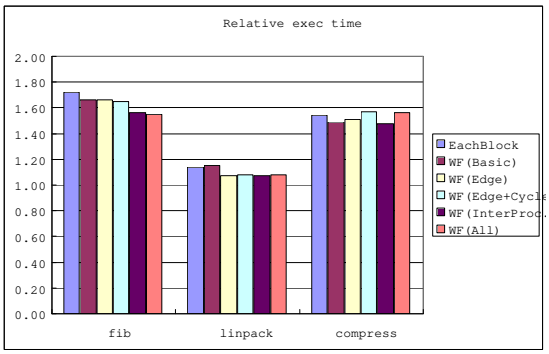


図 8　無変換の時を 1 としたときの実行時間 ($L_{\max} = 100$)

見られない原因の一つとして，メソッド中にサイクルが複数存在しても，最も内側のサイクル 1 つしか展開しないことが考えられる．

5.2　実 行 時 間

次の実験では，プログラム変換前後の実行時間を比較する．ここではアルゴリズムの性能を正確に比較するため，変換に要した時間は除いた．計測結果は表 3 のとおりである．Original は変換以前のもともとの実行時間である．さらに，先ほどと同一の 6 種類のアルゴリズムにしたがって変換した後の実行時間を計測し，Original の実行時間を 1 としたときの実行時間を求めた．結果は表 8 にプロットした．

得られた値自体は WF(All) で 1.08 ~ 1.56 程度である．fib の結果をみると，WF(Basic) と WF(InterP) で実行時間が改善しており，これは平均増加量の結果と一致する．次に，linpack の実行時間をみると WF(Edge) で実行時間が改善しているのは予想通りである．しかし，平均増加量の計測では，サイクル展開で顕著な改善がみられたにも関わらず，実行時間にはサイクル展開による改善が見られない．compress でも同様な傾向が見られる．サイクル展開後の実行時間が EachBlock よりも悪いのは明らかに予想された結果と異なる．

考えられる原因の一つは，サイクル展開そのもののコストである．複製したコード片をつなぎ合わせる際に，適宜 goto 文を付け加えている．これらのコードは JIT コンパイルされる際に最適化される可能性も高いが，コードサイズの増加による影響も考えられる．その他にも，実行する JVM，JIT によっても実行時間は大きく影響される．今回用いたもの以外の Java 処理系においての比較検討が必要であろう．

6. 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としている．CPU 資源はネイティブコードを通じて OS の管理情報を利用しており，ヒープメモリとネットワーク消費量に関してはバイトコード変換を利用している．ただし，OS の提供する情報はサンプリングに基づいているため，プログラムによっては正確に計測することが困難である．本研究では実行命令数を正確にカウントする．また，1 ~ 数千命令単位の細かい粒度での資源計測が可能である．

J-SEAL2[3] 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．CPU，メモリ資源の管理をバイトコード変換によって実現しており，保護ドメイン間の資源割り当てに関しても考察している．J-SEAL2 では本研究と同じく，消費 CPU 資源を実行命令数を計測することにより管理する．さらに，バイトコード変換で実現する点も同様である．J-SEAL2 のコード挿入アルゴリズムは，特に計測の粒度については問題としておらず，本研究の $L_{\max}$ 制約に相当する制限を設けていない．また，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する．それに加えていくつかの最適化を施している．一方，本研究で用いた WeightFirst では各基本ブロックの予測実行頻度を利用しており，さらにサイクル展開やメソッドをまたがった解析を行っているためよりよいコード挿入が期待できる．

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[5),11)] は，検査が行われる間隔に上限を設けており，本研究との関連が深い．Balanced Polling[11] はメソッドをまたがった解析を行わないが，メソッド呼出しの前後で適切な仮定を設けることで，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．Punctural Polling[5] は Balanced Polling を拡張

	Original	EachBlock		WF (Basic)		WF(Edge)		WF (Edge+Cycle)		WF(Edge+InterP)		WF(All)	
	T	t	t/T	t	t/T	t	t/T	t	t/T	t	t/T	t	t/T
fib	7,647	13,152	1.72	12,661	1.66	12,674	1.66	12,641	1.65	11,934	1.56	11,887	1.55
linpack	19,982	22,691	1.14	22,989	1.15	21,313	1.07	21,617	1.08	21,312	1.07	21,597	1.08
compress	16,845	25,984	1.54	25,103	1.49	25,387	1.51	26,418	1.57	25,007	1.48	26,237	1.56

表 3 $L_{\max} = 100$ での実行時間の比較．無変換のときの実行時間 (T) と実行時間 (t)，及び無変換の時を 1 としたときの実行時間 (t/T)

したアルゴリズムであり，実行命令数の正確な計測や，ポーリング間隔の下限を保証している．本研究で提案したアルゴリズムは Punctural Polling で用いているコード挿入アルゴリズムをさらに拡張したものであり，特別な変更を行うことなく Punctual Polling と組み合わせることが可能である．

7. まとめと今後の課題

本研究では Java プログラムの消費 CPU 資源を細粒度に計測する手法を実現した．バイトコード命令列中の適切な個所に，カウンタを増加するコードを挿入することにより，スレッドごとに実行命令数を正確に計測する．また，カウントの絶対誤差を $L_{\max}$ 命令以内におさえるという条件 ($L_{\max}$ 制約) を保証しており，1〜 数千命令の細粒度の計測が可能である．

今回，我々は先に考案したコード挿入アルゴリズム WeightFirst を拡張し，制御流グラフの辺へのコード挿入，サイクル展開，メソッドをまたがった解析に対応した．この拡張は制御流グラフの拡張によって実現しており，WeightFirst 本体とは独立している．そのため，頂点にのみコード挿入を行うアルゴリズムであれば，WeightFirst を置き換えることも可能である．

実験ではいくつかのベンチマークプログラムで，拡張したアルゴリズムの評価を行なった．最初の実験では，実行命令数とカウンタの増加回数の計測から，一回あたりの平均増加量を算出した．その結果，サイクル展開とメソッド間解析を行う前後で，平均増加量が数命令から最大で 30 命令増加改善した．次にの実験では実行時間を計測し，コード挿入によるオーバヘッドを観察した．その結果，平均増加量の改善から予想されるほど，サイクル展開によるオーバヘッドの減少が認められなかった．その原因は，サイクル展開にともなうコード量の増加，および辺の張り替えによる goto 文の挿入などが考えられる．

今後の課題にオーバヘッドが期待通りにに低下しない原因の究明がある．JMV や JIT の相違は実行時間に大きな影響を与えるので，他の Java 実行環境で実行時間を計測するといった実験が必要であろう．また，現在メソッド内に複数のサイクルが存在した場合，最も内側のサイクルを一つしか展開していない．コードサイズの爆発を防ぎつつ複数のサイクルを展開する戦略が必要である．さらに，管理対象にヒープメモリも含めるといった拡張も考えられる．

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) Back, G., Hsieh, W. C. and Lepreau, J.: Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java, *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, USENIX (2000).
5) 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).
6) 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, pp. 41–51 (2002).
7) Knuth, D. E. and Stevenson, F. R.: Optimal Measurement Points for Program Frequency Counts, *BIT*, Vol.13, No.3, pp.313–322 (1973).
8) Ball, T. and R.Larus, J.: Optimally Profiling and Tracing Programs (1992).
9) Ball, T. and Larus, J. R.: Branch Prediction for Free, Technical report, Computer Sciences Department, University of Wisconsin - Madison (1993).
10) Wu, Y. and Larus, J. R.: Static Branch Frequency and Program Profile Analysis, Technical Report CS-TR-1994-1248, University of Wisconsin, Computer Sciences Department (1994).
11) Feeley, M.: Polling Efficiently on Stock Hardware, *Functional Programming Languages and Computer Architecture*, pp. 179–190 (1993).