2003－ARC－154　(19)
2003／8／6

# バイナリレベルマルチスレッド化への制御投機の導入とその評価

横田昌之[†]　佐藤智一[†]　大津金光[†]
横田隆史[†]　馬場敬信[†]

我々はシングルスレッドコードをマルチスレッドコードにバイナリレベルで変換するシステムを研究開発してきた．これまでの研究により数値計算系アプリケーションに対しては高い性能向上が得られているが，整数系アプリケーションにおいては高い性能向上が得られていない．この大きな原因は，アプリケーションコードの複雑な制御構造とそれによるスレッド間依存である．この問題は，制御投機スレッドを適用して不要な制御依存を除去することにより緩和できる．本稿では高速化が困難な整数系アプリケーションを対象として，制御投機マルチスレッド実行を行うコードをバイナリレベルで作成し，実行性能の評価を行う．評価により，プログラム中の実行頻度の高いパスに沿って投機的にマルチスレッド実行を行うことによって，速度向上が可能であることを示す．

# Control Speculation on Binary-Level Multithreading and its Evaluation

Masayuki YOKOTA,[†] Tomokazu SATOU,[†]
Kanemitsu OOTSU,[†] Takashi YOKOTA[†] and Takanobu BABA[†]

Currently, we are developing a binary translation system that can translate singlethread binary codes into multithreaded ones. Our previous studies show that the performance of floating-point applications can be highly improved. However, it is difficult to improve the performance of integer applications because of their complicated control-flow and inter-thread dependencies. We consider the problem can be relaxed by introducing the control speculation. In this paper, we evaluate the effectiveness of control speculation for binary-level multithreading using integer applications. The evaluation results show that the performance of integer applications can be improved by speculating the frequently executing paths.

## 1. はじめに

従来，高性能プロセッサシステムはシングルスレッド実行を前提にその性能を向上させてきたが，その高速化には限界が見えてきた．この問題の解決策としてマルチスレッド実行モデルが有効であるが，マルチスレッド実行を前提としたプログラミングは容易ではなく，自動的にマルチスレッド化を行うシステムが必須になる．しかしながら，高速化したいアプリケーションのソースコードが参照できない場合にはマルチスレッド化することは不可能であるという問題がある．そこで我々は，シングルスレッドコードをマルチスレッドコードへバイナリレベルで変換してアプリケーションの実行性能を向上させるシステムを提案し，実現に向けて研究を進めている[1),2)]．

これまで，数値計算系アプリケーションを対象とした場合にはマルチスレッド化によってスレッドユニット数に応じたスケーラブルな性能向上が達成可能であることが明らかになっているが，整数系アプリケーションにおいては複雑な制御依存とスレッド間依存のため高い性能向上は得られていない[1)~3)]．

マルチスレッド実行において，スレッド間に制御依存が存在する場合，スレッド間で同期を取るために先行するスレッドの制御の流れが確定するまで自スレッドの実行を停止させる必要がある．対象アプリケーションコードの制御構造が複雑であると，実際には実行されることのない命令によるスレッド間依存を解消するためにスレッドの実行を停止させることがあり，これが性能向上を妨げる原因になる．

この問題は，制御投機[4)]を用いてプログラム中の特定の制御フローに沿ってスレッドを生成し，マルチスレッド実行を進めることにより緩和できる．

本研究では，バイナリ変換システムを前提に，一度対象アプリケーションプログラムを実行して取得したプロファイル情報を基にプログラム中で実行頻度の高いパスを特定する．そして特定されたパスに沿って投

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University

機的にマルチスレッド実行を行うことにより，不要な制御依存を除去してアプリケーションプログラムの高速化を図る．本稿では高速化が困難な整数系アプリケーションを対象として，制御投機マルチスレッド実行を行うコードをバイナリレベルで作成し，その性能評価を行う．

## 2. 投機的マルチスレッド実行

### 2.1 マルチスレッド実行モデル

本研究では，スレッドパイプライニングモデル[5]をベースとしてマルチスレッド化を行う．図1にスレッドパイプライニングの概念を示す．スレッド間のデータ依存については，スレッド間で同期を取る機能を持つ**Memory Buffer**によって解消する．Memory Bufferは自スレッドのメモリアクセスについて常に監視を行い，適宜スレッドの実行を停止させる．

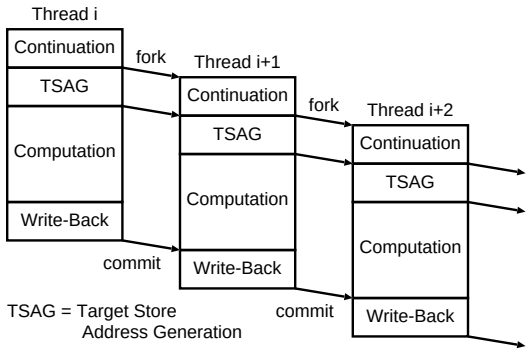



**図1** スレッドパイプライニングモデル

各スレッドは**Continuation**，**TSAG (Target Store Address Generation)**，**Computation**，**Write-Back**の4つのステージに分けられる．

Continuationステージでは，ループ変数の更新等，後続スレッドの実行開始時に必要となる計算を行う．その後，次のスレッドを起動しTSAGステージに入り，スレッド間でデータ依存の可能性があるメモリアクセスのアドレスをMemory Bufferに登録する．TSAGステージ終了後，Computationステージに入り計算本体のコードを実行する．この際に発生するメモリアクセスは，Memory Bufferによって監視され，スレッド間で依存があるメモリアクセスが発生した場合にはスレッド間で同期を取りながら処理を行う．最後にWrite-Backステージに入り，先行スレッドの実行終了を待ち，実行結果の書き戻しを行う．

### 2.2 マルチスレッド化方式

本研究ではループのイテレーションを単位として各スレッドに処理を割り当てる．また，マルチスレッドコードはアプリケーションプログラム実行前に作成する．図2（a）に示すシングルスレッドコードの点線で

```
for (i = 1; i < N; i++) {
  d = a[i-1] / 10;
  r = a[i-1] % 10;
  if (d == r) {
    b[i] = a[i];
  }
  else {
    a[i] = r + d;
    b[i] = 0;
  }
}
```

（a）オリジナルコード

```
b[i] = a[i];
```

（c）1スレッドの処理（最適化後）

```
d = a[i-1] / 10;
r = a[i-1] % 10;
if (d == r) {
  b[i] = a[i];
}
```

（b）1スレッドの処理（最適化前）

```
b[i] = a[i];

d = a[i] / 10;
r = a[i] % 10;
if (d != r) {
  後続スレッド破棄;
}
```

（d）1スレッドの処理（判定コード追加後）

**図2** コード変換例

囲まれたパス(以下，投機対象パス)に沿ってマルチスレッド化する場合を例に，マルチスレッドコードの生成方法を説明する．ここで，各スレッドにはループの1イテレーションの処理を割り当てるものとする．

まず，投機対象パスのコードを抜き出す(図2（b）)．

次に，次イテレーション以降も同じパスが実行されると仮定してDead Code Elimination[6]等の最適化を施す．図2（b）において3行目のif文は不要となるので削除する．さらに，このif文を削除することにより1行目と2行目も削除できる．この最適化の結果，同図（c）に変換される．

最後に，投機成功/失敗を判定するためのコードを追加する．各スレッドが自分自身の投機成功/失敗を判定して処理をやり直す場合，レジスタ及びメモリの内容を予め退避させておき，投機失敗時には投機実行中に上書きされたレジスタとメモリの内容を投機実行前の状態に戻す必要がある．この退避及び復元処理がオーバヘッドとなり，高速化の妨げとなる．

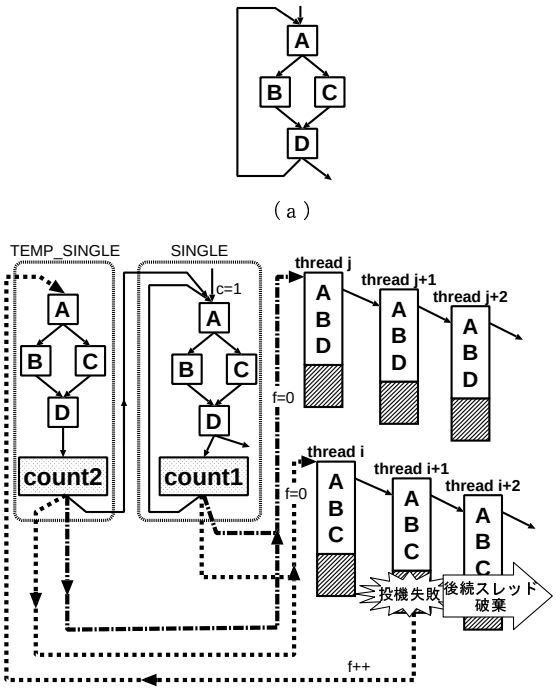
本研究ではこのオーバヘッドを軽減するため，各スレッドは自分自身ではなく次スレッドの投機の成功/失敗を判定し，次スレッドの投機が失敗したと判定された場合は後続スレッドを全て破棄することにより，レジスタとメモリの内容の退避/復元の必要が無くなるようにする．

そのために，図2（d）に示す通り，次イテレーションの分岐命令とその分岐命令で参照される値を計算する命令を同図（c）の後に追加する．

以上の方法で作成したコードを各スレッドが投機的に実行することによって速度向上を図る．

### 2.3 実行手順

図3（a）の制御フローグラフで表されるループを例に，本研究で適用した投機的マルチスレッド実行の手順を示す(図3（b）)．ここで，図中のA〜Dはプロ




(a)

(b)

**図 3** 制御投機実行

グラムの基本ブロックを表す.

(1) 最初はシングルスレッド実行を行う (図中 SINGEL).

(2) イテレーション毎にどのパスが実行されたかを調べる. 本研究では Efficient Path Profiling (EPP)[7] を利用して実行されたパスの特定を行う. EPP とは, プログラムの各基本ブロック間に, カウンタの操作コードを挿入することで各パスに番号付けを行うものであり, ソフトウェアによる正確なパスプロファイリングが可能である.

(3) 現在実行されたパスと, その 1 つ前のイテレーションで実行されたパスが等しければ c に 1 を加え, 異なっていれば c=1 とする (図中 count1). ここで, c は同じパスが連続して実行された回数を表す変数である.

(4) c の値が, 予め設定された閾値 $T_{to\_multi}$ 未満ならばそのままシングルスレッド実行を続ける.

(5) c の値が $T_{to\_multi}$ 以上ならば, 現在実行されたパス (以下, 投機対象パス) に沿ってマルチスレッド実行を開始する. ここでは ABC が投機対象パスになったとする.

(6) 各スレッドは 2.2節で説明した通り, 次スレッドの投機成功/失敗の判定を行う (図中 斜線部分).

(7) 次スレッドの投機が成功したと判定された場合, 自スレッドは実行を終了する.

(8) 次スレッドの投機が失敗したと判定された場合 (図中 thread i+1), 後続スレッドを全て破棄し, f に 1 を加え, 次の 1 イテレーションはシングルスレッド実行を行う (図中 TEMP_SINGLE). ここで f は投機が連続して失敗した回数を表す変数である.

(9) f の値と, 予め設定された閾値 $T_{to\_single}$ とを比較し (図中 count2), f の値が $T_{to\_single}$ 以上ならば (1) に戻る.

(10) f の値が $T_{to\_single}$ 未満ならば再び ABC に沿ってマルチスレッド実行を開始する.

上記 (9) によってマルチスレッド実行からシングルスレッド実行に移った場合, ABD が連続して $T_{to\_multi}$ 回実行されると, 今度は ABD が投機対象パスとなり, ABD に沿って投機的にマルチスレッド実行が行われる.

## 3. 評　　価

### 3.1 評価環境

前節で説明した制御投機マルチスレッド化手法の有効性を検証するため, 投機的マルチスレッド実行の実行性能の評価を行う.

評価は, 対象アプリケーションのマルチスレッド化対象ループ部分について, シングルスレッド実行サイクル数と投機的マルチスレッド実行サイクル数を計測し, 速度向上比 (= シングルスレッド実行サイクル数/投機的マルチスレッド実行サイクル数) を求めることで行う.

評価には, SimpleScalar をベースにした, スレッドパイプライニングモデルを実現するアーキテクチャシミュレータ **SIMCA**[8] を用い, 表1に示すシミュレーションパラメータを用いて評価を行った.

対象となるシングルスレッドバイナリコードは, SIMCA 用 gcc クロスコンパイラ (version 2.7.2.3 最適化オプション -O3) により生成した. 評価用アプリケーションとして SPECint95 の m88ksim, compress,

**表1** シミュレーションパラメータ

| | |
|---|---|
| スレッドユニット | 4 命令同時実行<br>out-of-order 実行 |
| 1 次キャッシュ<br>(命令) | direct-map<br>16KB(ラインサイズ 32B)<br>レイテンシ 1 クロック |
| 1 次キャッシュ<br>(データ) | 4-way set-associative<br>16KB(ラインサイズ 32B)<br>レイテンシ 1 クロック |
| 2 次キャッシュ<br>(命令・データ混在) | 4-way set-associative<br>256KB(ラインサイズ 64B)<br>レイテンシ 6 クロック |
| メモリ | レイテンシ 18 クロック |
| メモリバッファ | 512 エントリ |
| スレッドユニット間<br>通信ポート | レイテンシ 1 クロック |

**表2**　パスの実行頻度

| | 入力データセット | | | |
|---|---|---|---|---|
| | test | | train | |
| アプリケーション | path1 | path2 | path1 | path2 |
| m88ksim | 97.0% | 3.0% | 97.0% | 3.0% |
| compress | 50.6% | 43.0% | 54.5% | 22.4% |
| li | 51.1% | 48.9% | 80.7% | 19.3% |

**表3**　マルチスレッド実行開始・終了の条件

| アプリケーション | $T_{to\_multi}$ | $T_{to\_single}$ |
|---|---|---|
| m88ksim | 2 | 4 |
| compress | 4 | 4 |
| li | 8 | 8 |

liを用い，入力データはtestとtrainを使用した．

マルチスレッド化対象部分は，各アプリケーションについて頻繁に実行されるループをPC(プログラムカウンタ)サンプリング[9]により求めた結果を用いる．その結果，m88ksimは関数killtime()，compressは関数compress()，liは関数sweep()内のループをマルチスレッド化の対象とした．それらのループの中で実行頻度の高い2つのパスをマルチスレッド化の対象とした．各パスの実行頻度を表2に示す．ここでpath1は最も実行頻度の高いパスを表し，path2は2番目に実行頻度の高いパスを表す．

compressは1スレッドに1イテレーションの処理を割り当て，m88ksimとliについては，1イテレーションあたりの処理サイズが小さいため，スレッド制御のオーバヘッドを軽減するために1スレッドに8イテレーション分の処理を割り当てた．また，マルチスレッド実行を開始するための閾値とマルチスレッド実行を終了するための閾値(2.3で説明した$T_{to_multi}$と$T_{to_single}$)は表3に示す値に設定した．

### 3.2 評価結果

図4にマルチスレッド化対象ループ部分の速度向上比を，表4に投機の成功・失敗回数を示す．

m88ksimの速度向上比はスレッドユニット数4，8，16台において，testを入力とした時はそれぞれ0.96倍，0.94倍，0.94倍，trainを入力とした時はそれぞれ0.78倍，0.76倍，0.76倍と，約97%の確率で投機が成功するにも関わらず速度が低下してしまった．この理由を以下に説明する．

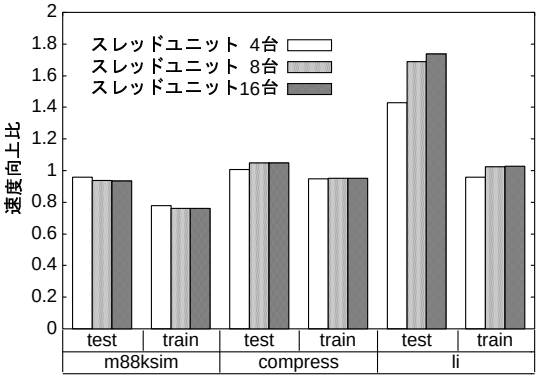
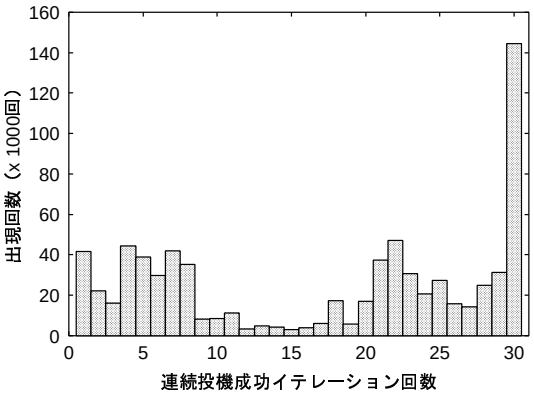


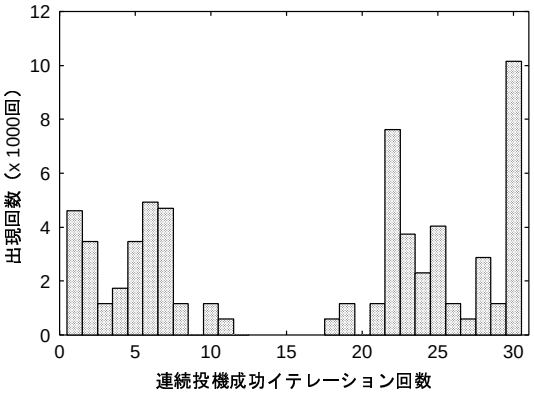

**図4**　マルチスレッド化対象部分の速度向上比

**表4**　投機成功率と成功・失敗回数

| アプリ | データ | 成功率 | 成功回数 | 失敗回数 |
|---|---|---|---|---|
| m88ksim | test | 96.8% | 13,124,679 | 431,531 |
| | train | 97.0% | 1,061,257 | 33,369 |
| compress | test | 79.3% | 4,411 | 1,149 |
| | train | 56.3% | 55,717 | 43,315 |
| li | test | 91.2% | 3,454,394 | 331,305 |
| | train | 86.7% | 985,679 | 151,616 |




（a）データセットtest




（b）データセットtrain

**図5**　m88ksimの連続投機成功回数

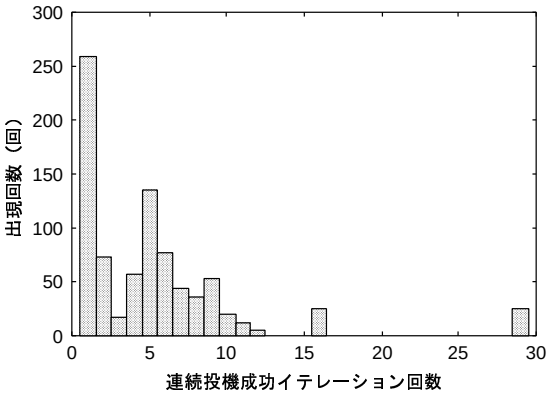
本稿の投機的マルチスレッド実行によってプログラムの性能が向上するためには，投機が連続して成功しなければならない．図5に，m88ksimにおいて投機が連続して成功した回数を示す．横軸が投機の連続成功回数を表し，縦軸はそれが何回あったかを表す．例えば横軸が20で縦軸が7の場合，20回連続で投機が成功したということが7,000回あったことを表す．

並列実行しているスレッドの数が3つ以下の部分の

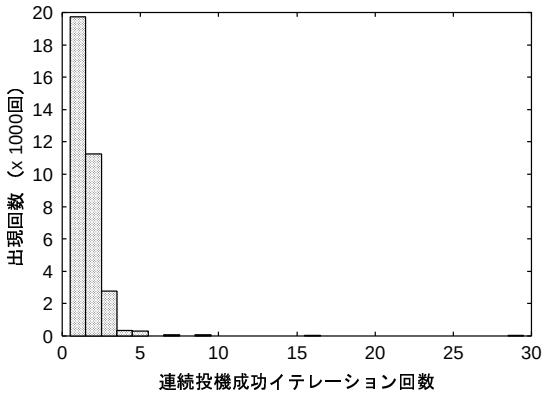
速度向上比は，入力がtest，trainのとき，それぞれ0.85倍，0.78倍であったが，4つのスレッドが並列に処理を進めている部分(図5中で横軸の値が30のところ) のみではそれぞれ1.21倍，1.22倍の速度向上が得られた．連続成功回数が8回以下のところでは，2つ以上のスレッドが並列実行することがなく1つのスレッドで実行を続けることになるが，シングルスレッドコードの計算本体の部分が11命令であるのに対し，パスの連続実行回数をカウントするためのコードとマルチスレッド実行を開始するかを判定するためのコードの合計が14命令ある．速度が低下したのはこの判定処理のオーバヘッドが大きいためである．

次にcompressの結果について述べる．本研究でマルチスレッド化の対象としたループは制御構造が複雑であり，スレッド間で依存する変数は数十個あるため，投機を用いずにマルチスレッド実行を行ったとしても高速化は望めない．しかしpath1のみに沿ってマルチスレッド化を行う場合，スレッド間依存は1つの変数のみになり，またpath2に沿った場合はスレッド間依存は完全になくなる．

図6に，compressにおける連続投機成功回数を示す．




（a）データセットtest




（b）データセットtrain

**図6** compressの連続投機成功回数

パスの実行頻度は表2よりpath1とpath2を合わせて，testで93.6%，trainで76.9%であるのに対し，投機成功率は表4よりtest，trainの場合でそれぞれ79.3%，56.3%と実行頻度に比べて低いことがわかる．これは，本研究ではパスの予測方式として連続して同じパスが実行されると予測しているが，compressの場合はpath1とpath2が交互に実行されることが多かったため，投機の成功と失敗を交互に繰り返していたことが原因である．

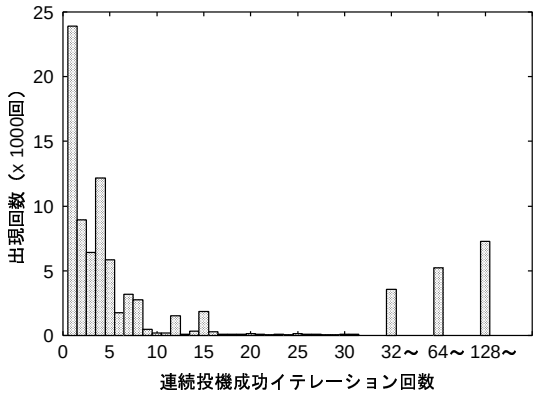
速度向上比はスレッドユニット数4，8，16台において，testを入力とした時はそれぞれ1.01倍，1.05倍，1.05倍となり速度が向上したが，trainを入力とした時はそれぞれ0.95倍，0.95倍，0.95倍と，速度が低下した．testを入力した場合において速度が向上した理由は，パスの連続回数や投機の連続失敗回数を数えてマルチスレッド実行の開始/終了を判定するオーバヘッドに比べて，連続して投機が成功したことによる速度向上の方が大きかったためである．一方trainでは，図6（b）に示す通り，投機を行ってもそのほとんどは1〜2回しか連続して成功せず成功と失敗を交互に繰り返すため，マルチスレッド実行による速度向上が，投機失敗回数を数えるオーバヘッドに打ち消されたことで速度が低下したと考えられる．

最後にliの結果について説明する．liでマルチスレッド化の対象としたループは，制御構造が複雑であり，かつ間接ジャンプ命令があるため，静的にスレッド間依存の解析を行うことができず，そのままではマルチスレッド化は不可能である．ところがpath1，path2それぞれに沿ってマルチスレッド化を行う場合，両方ともスレッド間依存となる変数は存在しなくなる．

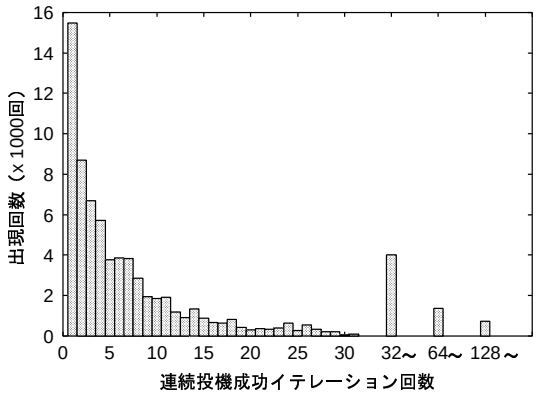
図7に，liにおける連続投機成功回数を示す．

liの速度向上比はスレッドユニット数4，8，16台において，testを入力とした時はそれぞれ1.43倍，1.69倍，1.74倍と，他のアプリケーションに比べて大きく性能向上した．これは，同時に複数のスレッドが実行されていない部分(図7（a）で横軸の値が8以下の部分)が多いが，投機が数十回以上(最大で992回)連続で成功する部分で大きく性能が向上したためである．

trainを入力とした時の速度向上比はスレッドユニット数4，8，16台において，それぞれ0.96倍，1.02倍，1.03倍となった．testを入力した場合と比べて速度向上比が小さくなった理由は，図7（b）に示す通り，投機の連続成功回数がtestのときより少ないためである．また，スレッドユニット数が4台のときに性能が低下しているが，8台，16台のときはシングルスレッド実行より性能が向上した理由は，スレッドユニット4台では投機成功による性能向上より投機を中断するかを判定する部分のオーバヘッドの方が大きかったが，8台，16台のときは逆に投機成功による性能向上が判定部分のオーバヘッドを打ち消したためである．




（a）データセット test




（b）データセット train

**図 7**　li の連続投機成功回数

## 4. おわりに

本稿では一度プログラムを実行して得られたプロファイル情報を基に制御投機を利用したマルチスレッド実行を行うコードをバイナリレベルで作成し，その評価を行った．マルチスレッド化対象部分を，PC サンプリングによって検出された，プログラム中で実行頻度の高いループとし，そのループ中で実行頻度の高いパスに沿って投機的にスレッドの実行を開始することにより，アプリケーションプログラムの速度向上を図った．

評価の結果，投機的マルチスレッド実行の開始および中断の判定をする部分のオーバヘッドを相殺できる程度に投機が連続して成功すれば，最大で 1.74 倍の速度向上が達成できることを確認した．

今後の課題として，本方式を他のアプリケーションに適用して評価を取ることが挙げられる．また，スレッド間のデータ依存を解消するためデータ投機を利用したマルチスレッド化手法を適用し，評価を取る必要がある．

さらに，本研究における投機を行うパスの予測方式は，連続して同じパスが実行されるという単純な予測方式であるため，投機の連続成功回数が少ない．そのため，さらに成功率の高い予測方式を適用することにより更なる性能向上が得られると考えられる．

**謝辞**　本研究は，一部日本学術振興会科学研究費補助金（基盤研究 (B)14380135，同 (C)14580362，若手研究 14780186）の援助による．

## 参 考 文 献

1) K. Ootsu, T. Ono, et al., “A Methodology of Binary-Level Multithreading and its Preliminary Evaluation,” Proc. 14th IASTED International Conference on Parallel and Distributed Computing and Systems, pp. 797-802, Nov. 2002.

2) K. Ootsu, T. Annou, et al., “Application of Binary-Level Multithreading to SPEC95 Benchmarks,” Proceedings of the Sixth Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC-6), pp.43-48, Nov. 2002.

3) 横田昌之，安濃隆弘，佐藤智一，大津金光，横田隆史，馬場敬信, “バイナリ変換による投機的マルチスレッド化方式の検討,” 情報処理学会第 65 回全国大会, 第 1 分冊 pp.133-134, 2003 年 3 月.

4) J. Tubella and A. González, “Control Speculation in Multithreaded Processors through Dynamic Loop Detection,” Proc. of 4th. Int. Symp. on High-Performance Computer Architecture (HPCA-4), pp.14-23, Feb. 1998.

5) J. Y. Tsai, J. Huang, et al., “The Superthreaded Processor Architecture,” IEEE Transactions on Computers, Vol. 48, No. 9, 1999.

6) D. F. Bacon, S. L. Graham and O. J. Sharp, “Compiler Transformations For High-Performance Computing,” ACM Computing Surverys, Vol. 26, No. 4, pp.345-420, Dec. 1994.

7) T. Ball and J. R. Laurs, “Efficient Path Profiling,” Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, pp.46-57, 1996.

8) J.Huang, “The SImulator for Multi-threaded Computer Architecture (Release 1.2),” http://www.cs.umn.edu/Research/Agassiz/Tools/SIMCA/simca.html.

9) 青木政人, 大津金光, 横田隆史, 馬場敬信, “マルチスレッド化のためのサンプリング情報に基づくホットループ検出,” 情報処理学会ハイパフォーマンスコンピューティング研究会 (HOKKE-2003), 2003.