単一チップマルチプロセッサ・アーキテクチャSKY における スレッド分割技法の評価

川梅 慶紀　　小林 良太郎　　安藤 秀樹　　島田 俊夫

名古屋大学大学院 工学研究科

我々は非数値計算プログラムに対し、マルチスレッド実行により性能を向上させる単一チップマルチプロセッサ SKY を提案してきた。SKY の機構を最大限に利用し大きな性能向上を達成するには、コンパイラはプログラムを並列性の高いスレッドに分割する必要がある。本稿では、ループや関数呼び出しなどの制御構造に着目した様々なスレッド分割技法について評価を行う。また、新たな投機的スレッド分割技法を導入し、その評価を行う。その結果、ループや関数呼び出しに着目してスレッド分割を行えばある程度の性能向上が得られ 、制御等価な基本ブロックに着目すれば更に高い性能向上が得られることが分かった。また今回導入した投機的スレッド分割技法は、投機を行わない技法に比べてほとんど性能向上が得られなかった。

Evaluation of a Thread Partitioning Techniques in a Single-Chip Multiprocessor Architecture SKY

Yoshinori Kawaume　Ryotaro Kobayashi　Hideki Ando　Toshio Shimada

Graduate School of Engineering, Nagoya University

We have proposed a multi-processor architecture, called SKY, which efficiently executes multiple threads in parallel for non-numerical programs. To achieve significant performance improvement with best utilizing the SKY architecture, the compiler is required to partition a program into threads with a large amount of parallelism. This paper shows evaluation results of various thread partitioning techniques based on control structures, such as loops, functions, and basic blocks. We also introduce a new speculative thread partitioning technique, and evaluate it. Our results show a certain amount of improvement is achieved by a technique focusing on loops and functions, and a technique focusing on control equivalent basic blocks achieves higher performance. We also show our new speculative thread partitioning technique which we introduce achieves little improvement compared with a non-speculative one.

1　はじめに

スーパスカラ・プロセッサに代わるアプローチとして、最近、複数のプロセッサを単一チップに集積するマルチプロセッサの研究が行われている [1, 4, 9, 11, 17, 18]。その背景には、単一制御流において利用可能な命令レベル並列性 (**ILP**:Instruction-Level Parallelism) が限界に近づきつつあるなど、スーパスカラ・プロセッサに対する悲観的観測に加え、半導体回路技術の進歩にともない、複数のプロセッサの単一チップへの集積化が実現可能となっていることがある。単一チップ・マルチプロセッサには、通信レイテンシを減少させることができるという利点がある。これにより、粒度の小さなスレッドレベル並列性 (**TLP**:Thread-Level Parallelism) を利用することができる。中でも、レジスタ値を直接通信し、同期を行う機構 [5, 17] は、メモリを介して同期／通信を行う機構に比べてオーバヘッドを大幅に減少させることができる [5]。

しかし、この同期／通信機構には、まだ改善の余地がある。なぜならば、同期が成立しなかった命令が現れたとき、その命令が受信待ちで停止するのに加え、それに後続する命令の実行も停止するという問題があるからである。

そこで我々は、**SKY** と呼ぶ単一チップ・マルチプロセッサのアーキテクチャを提案した [6]。SKY は、同期が成立しなかった命令が現れても、その命令とデータ依存関係がない後続命令の実行を停止することはない。このため、スレッド間に存在する並列性を十分に引き出すことができる。

マルチプロセッサにより大きな性能向上を得るため、コンパイラはプログラムの中の並列性を見つけ出し、スレッドに分割しなければならない。並列性を見つけ出すためには、スレッド間データ依存による待ち合わせの頻度が低く、スレッド並列実行による利得が得られるような箇所を探せばよい。このような箇所を発見し、スレッドに分割することがコンパイラに期待される。

数値計算分野におけるスレッド分割技法について様々な研究が行われている [8, 10] 。これらは、ループや関数呼び出しに着目してスレッド分割を行うことにより、スレッド間に存在する並列性を引き出している。

非数値計算分野におけるスレッド分割技法についても、様々な研究が行われている。これらの多くは制御構造に着目している。例えば、ループに着目する技法 [1, 12, 19]、関数呼び出しに着目する技法 [1, 9]、**制御等価** [15] に着目する技法 [3] などがある。しかし、これらのスレッド分割技法のうち、どれが非数値計算プログラムに有効で SKY の持つ特徴を活かすことができるか分かっていない。

本論文では、SKY において、制御構造に着目した様々

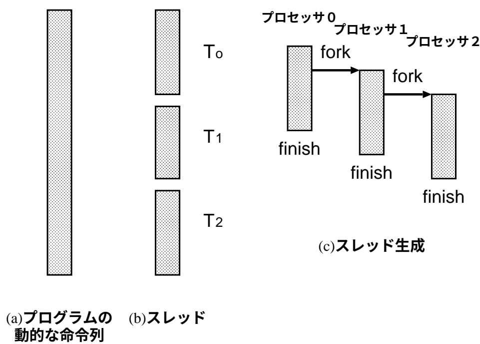

図 1: SKY のマルチスレッド・モデル

なスレッド分割技法について評価する。さらに投機的スレッド分割技法について評価する。

2 章では SKY の概要について述べる。3 章では SKY におけるスレッド分割技法の概要について述べる。4 章では本論文において評価の対象とするスレッド分割技法について述べる。5 章で評価を行い、6 章でまとめる。

2 SKY の概要

本章では SKY のアーキテクチャ[6] について述べる。SKY は、リング・バスで結合された複数のスーパスカラ・プロセッサからなる。細粒度の TLP 利用するため、プロセッサ間でレジスタ値を直接送受信し、同期/通信のオーバーヘッドを 2 サイクルまで減少させている。また、**命令ウィンドウ・ベース**の同期と呼ぶ同期機構を導入している。この機構は、命令ウィンドウで受信値に対応するタグを用いてレジスタに関する同期をとる機構である。この機構により、同期により後続命令の実行がブロッキングされることはなく、プロセッサは ILP を効率良く利用することができる。

スレッド並列実行のためのオーバヘッドを小さくするため、SKY のマルチスレッド・モデルは通常のマルチスレッド・モデルと比べて次に示す制約を課している。これは、**マルチスカラ・プロセッサ** [17] や **MUSCAT**[16] と同様のモデルである。

- 各スレッドは、逐次実行における動的に連続する部分で構成される。
- 各スレッドは、逐次実行の順において自分の直後のスレッドを生成する。

図 1(a) に逐次実行命令列を、図 1(b) にこれに対する SKY におけるスレッド分割の様子を示す。同図に示すように、SKY における各スレッドは、動的な命令列における単一の連続した部分からなり、異なる複数の部分からは構成されない。したがって、スレッドに結合はなく、制御に関する同期は必要ない。

図 1(b) に示したスレッドを並列に実行する様子を、図 1(c) に示す。図 1(b) において、各スレッド T_0、T_1、T_2 と逐次実行の順に名前をつける。図 1(c) に示すように、スレッド T_i は実行途中で、スレッド T_{i+1} を生成するという逐次生成を繰り返す。各スレッドは実行中には高々 1 回しか新しいスレッドを生成しない。実行のできるだけ早い時期に新しいスレッドを生成することにより、スレッドの並列実行を実現する。スレッドの生成は、`fork` と呼ぶ専用の命令を用い、終了は `finish` と呼ぶ専用の命令を用いて行う。以下、T_{i+1} を T_i の**子スレッド**と呼び、T_i を T_{i+1} の**親スレッド**と呼ぶ。また `fork` 命令を挿入する位置を**フォーク点**、`finish` 命令を挿入する位置を**子の開始点**と呼ぶ。

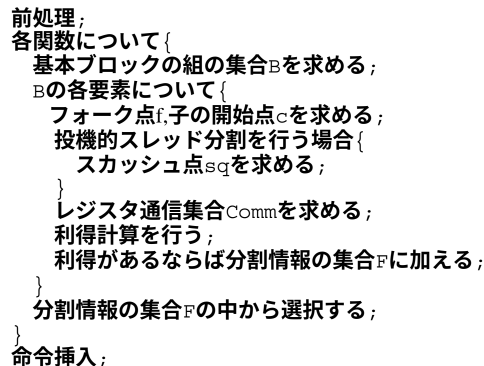

図 2: 分割処理の流れ

SKY の同期／通信機構は命令レベルで行う。通信には `send` 命令と呼ぶ専用の命令を用いることで、親スレッドから子スレッドへデータを送信する。

SKY は、制御依存のない部分をスレッドに分割することを前提としているため、投機的スレッド分割をサポートしていない。今回は 4 章で述べるように、投機的スレッド分割技法を含めた様々なスレッド分割技法について評価を行うため、投機的スレッド分割をサポートする必要がある。このために、新たに `squash` 命令を追加する。これは、投機的に生成した子スレッドとそれ以降の全てのスレッドを無効化する命令である。この命令は、スレッドの投機的生成の失敗が判明する点 (これを**スカッシュ点**と呼ぶことにする) に挿入される。

3 SKY におけるスレッド分割技法

本章では SKY におけるスレッド分割技法の概要を述べる。この技法は、文献 [3] に述べられているものに一部変更を加えたものである。変更を加えたのは、以下の 2 点である。

- スレッド分割対象の変更
  従来のものはスレッド分割対象を制御等価な基本ブロックの組とすることを前提としている。これを変更してループや関数などの構造に着目してスレッド分割対象を選ぶことがができるようにした。
- 投機的スレッド分割のサポート
  従来のものは制御依存のない部分をスレッドに分割することを前提にしていたため、投機的スレッド分割をサポートしていない。そこで、スカッシュ点を求める部分の追加、及び利得計算の変更などを行い、投機的スレッド分割をサポートした。

図 2 に全体の大まかな処理の流れを示す。最初にスレッド分割に必要な、分岐プロファイル等の情報を得るために前処理を行う。次に実際のスレッド分割に入る。分割は関数ごとに行う。スレッド分割は具体的には、以下に示すものを求めることである。

- フォーク点 f
- 子の開始点 c

- スカッシュ点の集合 sq
- レジスタ通信集合 $Comm$
- 利得値 g

これらの値及び集合の組 (f,c,sq,$Comm$,g) を**分割情報**と呼ぶ。分割情報の集合を F とする。以下にスレッド分割情報を求める手順を説明する。

まず、ある関数の中でスレッド分割の対象とする基本ブロックの組の集合 B を抽出する。すでに述べたように、抽出する対象として様々なものを選ぶことができるが、その詳細は 4 章で述べる。次に抽出された全ての基本ブロックの組について、フォーク点 f、子の開始点 c を求める。f、c は、基本ブロックの先頭とする。次にその f、c からスカッシュ点の集合 sq を求める。

次にレジスタ通信集合 $Comm$ を求める。各レジスタに関する `send` 命令の挿入位置は、送るべきレジスタ値が子の開始点に**到達する** [2] ことが決定する点とする。

次に利得 g を求める。これは、スレッド並列実行によってオーバラップ実行される命令数を見積もることによって求める。利得があらかじめ定められた閾値より低い場合、この分割は分割候補にせず、廃棄する。そうでない場合、分割情報 (f,c,sq,$Comm$,g) を分割候補の集合 F に加える。我々は以前に文献 [3] において利得計算手法を示したが、それは投機的でないスレッド分割を行うことを前提としていた。今回は、これに変更を加え、投機が成功する確率を考慮することにより、投機的なスレッド分割に対応させた。

このようにして分割候補の集合 F を作った後、F の中から SKY のマルチスレッドモデルにとって利得が最大となるように分割の組合せを選ぶ。そのあと、選択された分割情報にしたがって `fork`、`finish`、`send`、`squash` の各命令を挿入する。

4 スレッド分割モデルの説明

本論文の目的は、様々なスレッド分割技法について評価することである。そのため、前章で説明したスレッド分割技法をもとに、以下のスレッド分割技法を実装し、評価を行うことにする。

- LOOP モデル
  ループ・レベルのスレッド分割技法 (4.1 節)
- ELOOP(entire loop) モデル
  ループ全体をスレッドとして分割する技法 (4.2 節)
- FUNC モデル
  関数レベルのスレッド分割技法 (4.3 節)
- CE(control-equivalence) モデル
  制御等価に着目して分割する技法 (4.4 節)
- ACE(approximate control-equivalence) モデル
  本論文で新たに導入する投機的スレッド分割技法 (4.5 節)

以下、それぞれのモデルについて簡単に説明する。以下の説明では、図 3 の制御フローグラフを用いる。基本ブロック 2 は関数呼び出し命令を含むものとする。破線で示す辺に沿って制御が遷移する確率は、その始点を同じくする別の辺に比べて十分に低いものとする。

4.1 LOOP モデル

LOOP モデルは、ループ・レベルのスレッド分割を行うものである。具体的には、図 3 において、フォーク点、

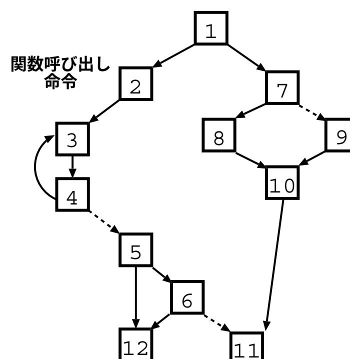

図 3: 制御フローグラフ

子の開始点をともに基本ブロック 3 の中に設けるようなスレッド分割を行う。

ループ・レベルの並列性に着目するスレッド分割技法については、様々な研究が行われている。Oplinger らは、積極的なコード変換を適用しない限り、ループ・レベルの並列性を利用することは難しいと述べている [13]。Olukotun らは **Hydra** と呼ぶマルチプロセッサにおいて、ループ・レベルの並列性を利用する技法を提案した [12]。しかし、この研究では各プロセッサにおいて ILP を利用しない場合の測定となっている。これに対し、我々は各プロセッサにおいて ILP を十分に利用できる場合について測定を行う。

4.2 ELOOP モデル

ELOOP モデルは、あるループの入口の直前の基本ブロックの先頭にフォーク点、そのループの出口の直後の基本ブロックの先頭に子の開始点を設けるようなスレッド分割を行うものである。具体的には、図 3 において、フォーク点を基本ブロック 2 の先頭、子の開始点を基本ブロック 5 の先頭に設けるようなスレッド分割を行う。

ループ全体をスレッドとして分割すれば多くの並列性が引き出せると言われており [7]、現実的なプロセッサ上での評価も行われている [14]。しかし、この技法に関してはまだ十分な調査がなされていない。

4.3 FUNC モデル

FUNC モデルは、関数呼び出し命令を含む基本ブロックの先頭にフォーク点、その直後の基本ブロックの先頭に子の開始点を設けるようなスレッド分割を行うものである。具体的には、図 3 において、フォーク点を基本ブロック 2 の先頭、子の開始点を基本ブロック 3 の先頭に設けるようなスレッド分割を行う。

Hydra[9]、**DMT**[1] では関数レベルのスレッド分割を行っている。関数に着目してスレッド分割を行えば、ループに着目する場合に比べて多くの並列性を引き出すことが可能であることが知られており [7, 13]、DMT では実際に多くの並列性を引き出している。

4.4 CE モデル

CE モデルは、制御等価な基本ブロックの組の**支配節** [2] の先頭にフォーク点、**後支配節** [2] の先頭に子の開始点を設けるようなスレッド分割を行うものである。具体的には、図 3 において、フォーク点を基本ブロック 7 の

先頭、子の開始点を基本ブロック 10 の先頭に設けるようなスレッド分割を行う。このようにすれば、ループや関数に着目する場合に比べて広い範囲を分割対象とすることができるため、より多くの並列性を見つけ出す可能性がある。

我々は過去に制御等価に着目したスレッド分割技法を提案した [3]。しかし、このスレッド分割技法の他の技法に対する有効性はまだ確かめられていない。

4.5 ACE モデル

ACE モデルは、本論文で新たに導入する投機的スレッド分割技法である。これは、制御の遷移確率の低い辺を削除した制御フローグラフにおいて、制御等価な基本ブロックの組の支配節の先頭にフォーク点、後支配節の先頭に子の開始点を設けるようなスレッド分割を行うものである。我々は、制御の遷移確率の低い辺を削除した制御フローグラフにおいて、ある基本ブロックの組が制御等価であるとき、その組は**近似制御等価 (approximate control-equivalence)** であるという。具体的には、図 3 において、フォーク点を基本ブロック 5 の先頭、子の開始点を基本ブロック 12 の先頭に設けるようなスレッド分割を行う。これは、辺 $6 \rightarrow 11$ に沿って制御が遷移する確率が低く、この辺が削除され、基本ブロックの組 $(5, 12)$ が新たに制御等価な関係となるためである。このようにすれば、投機的な分割のうち、投機が成功しやすいものを効率良く探すことができると考えられる。

辺を削除する際に注意すべきことがある。それは、辺を削除することにより、もともと制御等価な関係であった基本ブロックの組が制御等価な関係ではなくなってしまう状況が存在することである。以下に 2 つの具体例を示す。辺 $4 \rightarrow 5$ はループの出口の辺であるが、この辺を削除しても新たに制御等価な関係はできない。逆に、基本ブロックの組 $(2, 5)$ が制御等価な関係ではなくなる。同様に辺 $7 \rightarrow 9$ を削除しても、やはり新たに制御等価な関係はできず、逆に基本ブロックの組 $(2, 5)$ が制御等価な関係ではなくなる。このような状況を防ぐために、以下のようにする。

- **後ろ向きの辺** [2] の始点になっている基本ブロックを始点とする辺は削除しない。具体的には、図 3 において、辺 $4 \rightarrow 3$ 及び $4 \rightarrow 5$ は削除しない。
- 制御の遷移確率が低いために削除する辺の始点を A、削除する辺の終点を X、A の直後の基本ブロックの集合から X を除いたものを S、S から到達可能な基本ブロックの集合に S を加えたものを T、X から到達可能な基本ブロックの集合に X を加え、T を除いたものを Y とする。このとき始点が集合 Y に含まれ、かつ終点が集合 T に含まれる辺を削除する。具体的には、図 3 において辺 $7 \rightarrow 9$ を削除するとき、$A = 7$、$X = 9$、$S = \{8\}$、$T = \{8, 10, \cdots\}$、$Y = \{9\}$ とし、辺 $9 \rightarrow 10$ を削除する。

このようにすれば、CE モデルにおいて分割対象となっていた部分のほとんどは ACE モデルにおいても分割対象とすることができると考えられる。

5 評価

本章ではまず、評価環境について述べる。次に、各スレッド分割技法による性能を評価する。そのあと、今回導入した投機的スレッド分割技法について評価する。

表 1: SKY の基本モデル

(a) プロセッサ

項目	値
命令発行幅	8 命令
命令フェッチ幅	8 命令
命令ウインドウ	64 エントリ
リオーダバッファ	128 エントリ
機能ユニット	Int × 8、Load/Store × 4、Branch × 2、FP ALU × 8、Send × 4
分岐予測ミスペナルティ	4 サイクル

(b) 共有資源

項目	値
命令キャッシュ	2 ポート、各ポート 8 命令、常にヒット
データキャッシュ	4 ポート、各ポート 1 つのデータアクセス、常にヒット
メモリのデータ依存	ハードウェアで理想的に解消
分岐予測機構	1024 エントリ 4 履歴の PAp 予測 1024 エントリ、連想度 2 の BTB

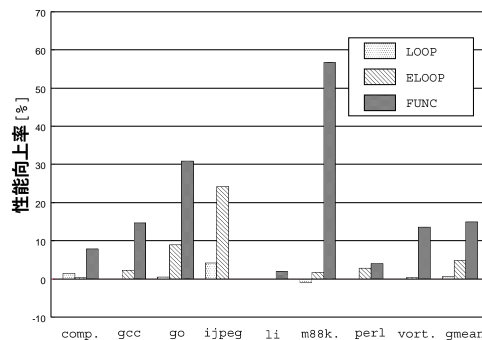

図 4: 各スレッド分割技法における性能向上率 (1)

5.1 評価環境

ベンチマーク・プログラムとして、SPECint95 の全 8 種を使用した。ベンチマーク・プログラムのバイナリは、GNU GCC 2.7.2.3(コンパイルオプション:-O6 -funroll-loops) を用いて作成した。評価はトレース駆動シミュレーションにより行った。トレースは SimpleScalar Tool Set Version 3.0a を利用して採取した。

表 1 に SKY の基本モデルを示す。性能比較における基準プロセッサは、SKY を構成する一つのプロセッサとする。SKY のプロセッサ数を 4 として評価を行った。測定するスレッド分割技法は、4 章で述べた 5 つのモデルの他に、LOOP-FUNC モデル (LOOP モデル、ELOOP モデル、FUNC モデルを組み合わせたもの) とした。以後のグラフでのベンチマーク名の表記において、compress95、m88ksim、vortex は、それぞれ comp.、m88k.、vort. と表すこととする。

5.2 各スレッド分割技法における性能

図 4 及び図 5 に性能の評価結果を示す。縦軸は、基準プロセッサに対する性能向上率である。gmean は全ベンチマークによる幾何平均を示す。各ベンチマークにつき 3 つの棒グラフがある。図 4 では左から LOOP モデルに

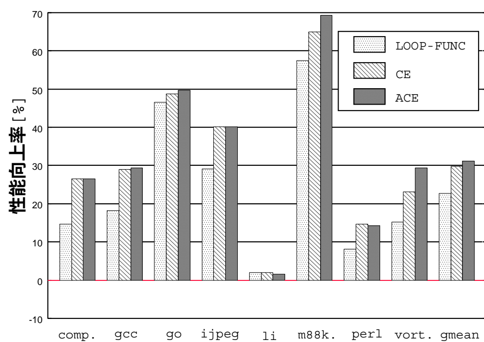

図 5: 各スレッド分割技法における性能向上率 (2)

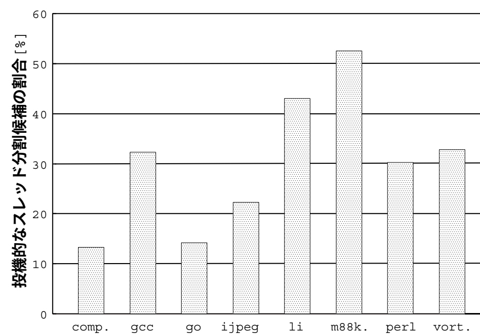

図 6: 投機的なスレッド分割候補の割合

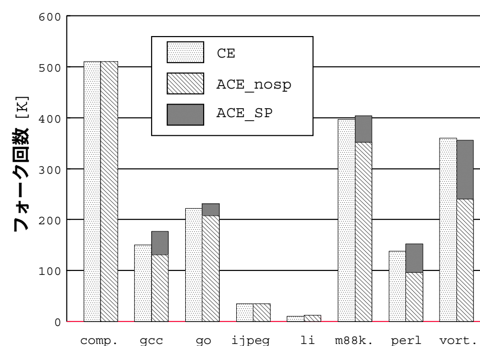

図 7: フォークの実行回数

おける性能向上率、ELOOP モデルにおける性能向上率、FUNC モデルにおける性能向上率を示す。図 5 では左から LOOP-FUNC モデルにおける性能向上率、CE モデルにおける性能向上率、ACE モデルにおける性能向上率を示す。

LOOP モデルでは、性能向上率が僅か 0.6%にとどまった。その理由は、4 章で述べた通り、ループ・レベルの並列性がほとんど存在しないためであると考えられる。

ELOOP モデルでは、平均で 4.8%の性能向上率であった。ijpeg において 24.2%、go において 8.9%の性能向上率を達成しているが、他のベンチマークではほとんど性能が向上していない。

FUNC モデルでは平均で 15.0%の性能向上率を達成している。特に m88ksim では 56.8%の性能向上を達成している。その理由は、やはり 4 章で述べた通り、関数レベルの並列性が比較的多く存在するためであると考えられる。

LOOP-FUNC モデルでは、平均で 22.7%の性能向上率を達成している。ループや関数に着目すれば、ある程度の並列性を引き出すことができる。しかし、CE モデルなどに比べると、十分に並列性が引き出せているとは言えない。

CE モデルでは、平均で 29.8%の性能向上率を達成している。LOOP-FUNC モデルよりも性能が高く、ループや関数以外の部分にも並列性が存在していることが分かる。

ACE モデルでは、平均で 31.1%の性能向上率を達成している。一部のベンチマークを除き、性能向上率は CE モデルとほぼ等しい。すなわち、今回導入した投機的スレッド分割技法では、制御等価に着目する場合に比べて性能はほとんど向上しない。

ACE モデルでは、CE モデルに比べてより広い範囲を分割対象とすることができるにも関わらず、ほとんど性能が向上していない。この原因を調べるため、CE モデル及び ACE モデルについてより詳細な評価を行う。

まず、ACE モデルにおいて投機的なスレッド分割候補がどの程度存在するのかを調べる。ここでスレッド分割候補とは、スレッド分割時に求める全ての分割情報のうち、利得があらかじめ定められた閾値以上であるものの集合のことである。図 6 に評価結果を示す。縦軸は、分割候補のうち、投機的なものが全体に占める割合を示す。

compress95、go、ijpeg では投機的な分割候補の割合は比較的少ないが、他のベンチマークでは 3 割以上が投機的な分割候補であり、少ないわけではないことが分かる。

次に、CE モデルと ACE モデルにおけるフォークの実行回数を調べ、実際にどの程度スレッド分割の機会が増加しているかを調べる。図 7 に評価結果を示す。各ベンチマークにつき 2 つの棒グラフがあり、左側は CE モデル、右側は ACE モデルにおけるフォークの実行回数を示す。右側の棒グラフは 2 つの部分からなり、下側は投機的でないフォークの実行回数、上側は投機的なフォークの実行回数を示す。

評価結果から、ACE モデルについて以下のことが分かる。

- CE モデルに対してフォークの実行回数がほとんど増加していない。
- 投機的でないフォークの実行回数が CE モデルに比べて少ない。
- 実行されているフォークのうち、投機的なものが占める割合は少ない。

投機的な分割候補の数が少なくないにも関わらず投機的なフォークの実行頻度が低い理由として、投機的な分割による利得は、投機的でない分割による利得よりも少ないため、スレッド分割時に選択されにくいためであると考えられる。

次に、ACE モデルにおいて投機的なフォークと投機的でないフォークがそれぞれ性能向上に寄与している度合を調べ、CE モデルにおいてフォークが性能向上に寄与している度合と比較する。

ある分割の集合が性能向上にどの程度寄与しているかを示す指標として、性能向上率の見積もり $espeed_up$ を導入する。これを以下のように定義する。

$$espeed_up = \frac{Dy_inst}{Dy_inst - egain}$$

ここで、Dy_inst は動的実行命令数を示す。$egain$ は利得の見積もりで、以下のように定義する。

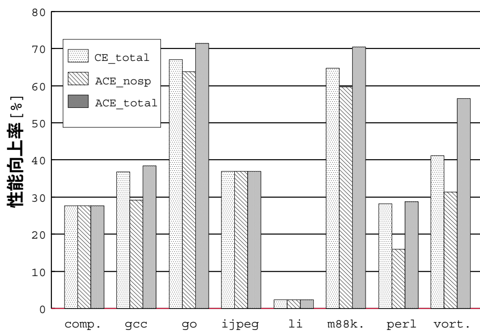

図 8: CE モデル及び ACE モデルにおける性能向上率の見積もり

$$egain = \sum_{t \in F} invoke(t) \cdot gain(t)$$

ここで、F は分割情報の集合、$invoke(t)$ は、シミュレーション時に分割情報 t のフォーク命令が実行された回数、$gain(t)$ は、分割情報 t の利得を示す。

まず、性能向上率の見積もりとシミュレーションによる性能向上率を比較することにより、この見積もりが妥当であるかどうかを調べた。その結果、go、perl、vortex では性能向上率の見積もりはシミュレーションによる性能向上率に比べて 15〜30%ポイント程度大きくなっているものの、他のベンチマークでは、性能向上率の差は 10%ポイント以内にとどまっている。このことから、利得計算が正確であるとは言えないが、性能向上への寄与を示す指標として用いても問題はないと言える。

図 8 に性能向上率の見積もりを示す。各ベンチマークにつき 3 本の棒グラフがあり、左から CE モデルにおける全体の性能向上率、ACE モデルにおける投機的でないフォークによる性能向上率、ACE モデルにおける全体の性能向上率を示す。ACE モデルにおいて、全体の性能向上率から投機的でないフォークによる性能向上率を引いた値は、投機的なフォークによる性能向上への寄与を示す。

go、m88ksim、vortex において、ACE モデルは CE モデルに対し性能向上率の見積もりが増加しているが、他のベンチマークではほとんど増加していない。また go や vortex においては利得計算の誤差が大きく、見積もられたほどには性能が向上していない。

以上の評価結果より、以下のことが言える。

- ループのみに着目する技法では、ほとんど性能が向上しない。
- ループや関数に着目する技法では、ある程度の性能向上が得られる。
- 制御等価な基本ブロックに着目する技法では、ループや関数に着目する技法に比べ、高い性能向上が得られる。
- 近似制御等価な基本ブロックに着目する技法では、制御等価な基本ブロックに着目する技法に比べ、ほとんど性能が向上しない。

6 まとめ

単一チップマルチプロセッサ・アーキテクチャSKY において、専用のコンパイラをもとに様々なスレッド分割技法を実装し、評価した。その結果、ループや関数に着目する技法に比べ、制御等価に着目する技法の方が高い性能を達成できることが分かった。また、今回導入した投機的スレッド分割技法では、制御等価に着目する技法に対してほとんど性能が向上しないことが分かった。

謝辞

本研究の一部は、文部省科学研究費補助金基盤研究(C)(課題番号 11680351) 及び財団法人大川情報通信基金の支援により行った。

参考文献

[1] H. Akkray, et al.,“A Dynamic Multithreading Processor,” In *Proc. MICRO-31*, pp.226-236, Nov. 1998.

[2] A. V. Aho, et al., *Compilers: Principles, Technique and Tools*, Addison-Wesly Publishing Company, 1986.

[3] 岩田充晃ほか,“制御等価を利用したスレッド分割技法,” 情報処理学会研究報告, 98-ARC-128, pp.127-132, 1998 年 3 月.

[4] 木村啓二ほか,“近細粒度並列処理用シングルチップマルチプロセッサにおけるプロセッサコアの評価,” 情報処理学会論文誌, Vol.42, No.4, pp.692-703, 2001 年 4 月.

[5] S. W. Keckler, et al.,“Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor,” In *Proc. ISCA-25*, pp.306-317, June 1998.

[6] 小林良太郎ほか,“非数値計算応用向けスレッド・レベル並列処理マルチプロセッサ・アーキテクチャ SKY,” 情報処理学会論文誌, Vol.42, No.2, pp.349-366, 2001 年 9 月.

[7] B. Kreaseck, et al.,“Limits of Task-based Parallelism in Irregular Applications,” In *Proc. HPC-III*, pp.43-58, Oct. 2000.

[8] M. W. Hall, et al.,“Maximizing Multiprocessor Performance with the SUIF Compiler,” *IEEE Computer*, Vol.29, No.12, pp.84-89, Dec. 1996.

[9] L. Hammond, et al.,“Data Speculation Support for a Chip Multiprocessor,” In *Proc. ASPLOS-VIII*, pp.58-69, Oct. 1998.

[10] C. Plychronopoulos, et al.,“Parafrase-2: An Environment for Parallelizing, Partitioning, Syncrinizing, and Schedulling Programs on Multiprocessors,” In *Proc. ICPP-IX*, pp.39-48, Aug. 1989.

[11] K. Olukotun, et al.,“The Case for a Single-Chip Multiprocessor,” In *Proc. ASPLOS-VII*, pp.2-11, Oct. 1996.

[12] K. Olukotun, et al.,“Improving the Performance of Speculatively Parallel Applications on the Hydra CMP,” In *Proc. Int. Conf. on Supercomputing*, pp.21-30, June 1999.

[13] J. T. Oplinger, et al.,“In Search of Speculative Thread-Level Parallelism,” In *Proc. PACT'99*, pp.303-313, Oct. 1999.

[14] E. Rotenberg, et al.,“Control Independence in Trace Processors,” In *Proc. MICRO-32*, pp.4-15, Nov. 1999.

[15] M. D. Smith, et al.,“Efficient Superscalar Performance Through Boosting,” In *Proc. ASPLOS-V*, pp.248-259, Oct. 1992.

[16] 鳥居淳ほか,“オンチップ制御並列プロセッサ MUSCAT の提案,” 情報処理学会論文誌, Vol.39, No.6, pp.1622-1631, 1998 年 6 月.

[17] G. S. Sohi, et al.,“Multiscalar Processor,” In *Proc. ISCA-20*, pp.414-425, June 1995.

[18] J. Y. Tsai, et al.,“The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation,” In *Proc. PACT'96*, pp.35-46, Oct. 1996.

[19] J. Y. Tsai, et al.,“Compiler Techniques for the Superthreaded Architectures,” *Int. Journal of Parallel Programing Special Issue on Languages and Compilers for Parallel Computing*, June 1998.