2004－ARC－156　(1)
2004／2／2

単一チップ・マルチプロセッサ SKY における データフローを考慮したスレッド分割技法

山口　武[†]　小林　良太郎[†]
安藤　秀樹[†]　島田　俊夫[†]

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 において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．非数値計算プログラムでは，粒度の小さなスレッドレベル並列性（TLP: Thread-Level Parallelism）を利用することが不可欠である．ハードウェアでは SKY において細粒度の TLP を効率よく利用することができるようになったが，従来のコンパイラではこのハードウェアを十分に活用するようなスレッドに分割しているとはいえなかった．本論文では，SKY が備えるノンブロッキングな同期／通信機構を最大限に利用して性能を向上させるスレッドに分割できるよう，データフローを考慮したスレッド分割技法を提案する．その結果，従来の評価方法を用いて選択したスレッド分割に対して，最大 17.2% ポイント，平均 6.3%ポイントの性能向上を得ることができた．

A Dataflow-based Thread Partitioning Technique in a Single-Chip Multiprocessor SKY

Takeshi Yamaguchi,[†] Ryotaro Kobayashi,[†] Hideki Ando[†]
and Toshio Shimada[†]

We have proposed a multi-processor architecture, called SKY, which efficiently executes multiple threads in parallel for non-numerical programs. To achieve significant performance improvements in the SKY architecture, the compiler is required to partition a program into threads with a large amount of parallelism. It is vital for non-numerical programs to exploit fine-grain thread-level parallelism (TLP). Although the SKY hardware has ability to exploit fine-grain TLP efficiently, the conventional compiler cannot partition a program into threads which fully utilize this ability. We propose a dataflow-based thread partitioning technique to partition a program into threads which improve performance by using the non-blocking synchronization mechanism of SKY. Our results show that the proposed technique achieves a maximum of 17.2% points or an average of 6.4% points performance improvements over a conventional thread partitioning technique without consideration of the non-blocking synchronization mechanism.

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，近年，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 (**CMP**: Chip MultiProcessor) が注目を集めている．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なことがある．発表されている商用の CMP [4–6, 11, 12] では，複数のプログラムの同時実行や，特定の処理の高速化 [2] を目的としている．

これに対し，1 つのプログラムをスレッドに分割し並列実行することで性能を向上させる研究が行われている [7, 9, 13, 15, 16]．CMP ではプロセッサ間通信のレイテンシを減少させることができるため，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．このため，粒度の大きな TLP が少ない非数値計算プログラムの高速化が期待できる．

CMP の同期／通信機構の中でも，レジスタ値を直接通信し，同期を行う機構は，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 [15, 16]．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，非数値計算プログラム向けの CMP アーキテクチャとして **SKY** を提案した [10]．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にない後続命令の実行を停止させることはない．これを**ノンブロッキング**な同期という．この機構により，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．

マルチプロセッサで大きな性能向上を得るには，コンパイラはプログラムを並列性が大きいスレッドに分割しなければならない．非数値計算プログラムは制御構造が複雑でデータ依存関係が多いので，スレッドに分割した際にスレッ

† 名古屋大学大学院 工学研究科
Graduate School of Engineering, Nagoya University

ド間でデータ依存が多く生じる．スレッド間データ依存を解決するための待ち合わせは性能に大きな影響を与えると考えられる．そこで，岩田ら [8] は，スレッド間データ依存が性能に与える影響を**距離**という基準で評価し，性能向上が見込めるスレッドに分割する手法を提案した．距離とは，ある 2 命令の間に存在する命令数の期待値である．

岩田らの手法はスレッド間データ依存のみに着目しているので，SKY の特徴であるノンブロッキングな同期を考慮できているとはいえない．ノンブロッキングな同期が性能向上に寄与するのは，スレッド間データ依存で待ち合わせをする命令と依存関係にない後続命令を実行可能な点である．スレッド間データ依存を考慮しただけでは，依存関係にない後続命令がどの程度存在するのかがわからない．実行を妨げられない命令が数多くあるならば，たとえスレッド間データ依存による待ち合わせがあったとしても並列性は高いといえる．この点で岩田らの手法は SKY の特徴を十分に活かしたスレッドに分割できていないといえる．

本論文では，スレッド間データ依存の他に，スレッド内のデータフローも詳細に解析し，ハードウェアを有効利用して性能を向上させるスレッドに分割することを目的とする．SKY におけるスレッド分割の処理は，各スレッド分割候補の性能向上を見積もり，性能向上が最も大きくなると期待されるスレッド分割候補の組合せを選ぶというものである．見積もりが正確でなければ，性能向上に寄与するスレッド分割を選択できない．そこで，スレッド内のデータフローを解析して性能向上の見積もりを正確にする手法を提案する．

2 章では SKY の概要について述べる．3 章では従来の研究におけるスレッド分割技法について述べる．4 章では本論文における提案手法である，データフローを考慮した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 のマルチスレッド・モデル，ハードウェア構成 [10] およびコンパイラ [8] について述べる．はじめにマルチスレッド・モデルについて述べる．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，**Multiscalar** [15] 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，各スレッドに T_0，T_1，T_2 と逐次実行順に名前をつける．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．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．

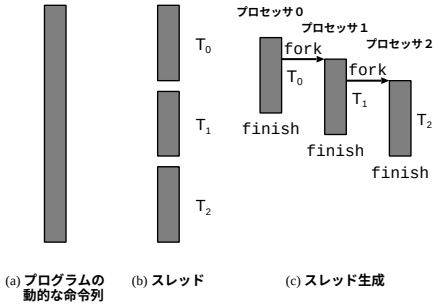

図 1 SKY のマルチスレッド・モデル

次にハードウェア構成について述べる．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 を利用するため，`send` と呼ぶ専用命令を使用してプロセッサ間でレジスタ値を直接通信し，同期／通信のオーバヘッドを 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り，ノンブロッキングな同期を実現している．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 を効率よく利用することができる．

最後にコンパイラについて述べる．コンパイラが行うスレッド分割の処理について，全体の流れを図 2 に示す．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，フォーク点 f，子の開始点 c，レジスタ通信集合 $Comm$，**利得** g を求めることである．

これらの値及び集合の組 $(f,c,Comm,g)$ を**分割情報**と呼び，その集合を F とする．以下に F を求める手順を説明する．

まず，ある関数の中で**制御等価** [14] な基本ブロックの組の集合 B を抽出する．次に抽出されたすべ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** [1] ことが決定する点とする．

最後に利得 g を求める．これは，注目しているスレッド分割を選択することによって期待される性能向上の見積もりを示す．あるスレッド分割の利得とは，フォーク点から子の開始点を経由し，関数の出口に至るまでのすべての経路について，それぞれの経路でオーバラップ実行される命令数をその経路を通る確率で重み付けした期待値である．利得があらかじめ定められた閾値より大きい場合，分割情報 $(f,c,Comm,g)$ を F に加える．

このようにして F を作った後，F の中から SKY のマルチスレッド・モデルにおいて利得が最大となるような分割情報の組合せを選ぶ．その後，選択された分割情報にしたがって `fork`，`finish`，`send` の各命令を挿入する．

SKY におけるスレッドへの分割方法は，性能向上の見積もりである利得が重要である．なぜならば，利得を正確に計算しなければ，性能向上に寄与する分割情報を選択できないためである．そこで本論文では，利得を計算する手法に着目した．3 章では，従来のスレッド間データ依存を考慮

```
前処理;
各関数について{
  基本ブロックの組の集合 B を求める;
  B の各要素について{
    フォーク点 f, 子の開始点 c を求める;
    レジスタ通信集合 Comm を求める;
    利得 g を求める;
    利得 g があるならば分割情報の集合 F に加える;
  }
  分割情報の集合 F の中から選択する;
}
命令挿入;
```

図 2 コンパイラの処理の流れ

したスレッド分割における利得計算手法を説明し，その問題点を述べる．

3. スレッド間データ依存を考慮したスレッド分割

本章では，従来の研究におけるスレッド分割技法とその問題点について述べる．

非数値計算プログラムではデータ依存関係が多いため，スレッド間のデータ依存関係が多く存在し，これが TLP に大きな影響を与える．この点に注目して，スレッド間のデータ依存関係を考慮してスレッド分割を行う技法が提案された [8, 17]．Vijaykumar ら [17] の技法は，発生回数の多いデータ依存に着目し，そのようなデータ依存関係にある命令は同じスレッド内に含まれるようにスレッド分割を行う．その分割ができない場合は，スレッド間データ依存による待ち合わせが少なくなるように，値を定義する命令が属する基本ブロックをスレッドの開始点とする．しかし，データ依存関係が多いので，スレッド間データ依存をなくすことはほとんどできない．また，依存の発生回数のみに着目しており，どのように分割したら性能向上が最も大きいかを正確に考えていないという問題点がある．

一方，岩田ら [8] は，データ依存関係にある命令間の距離に着目して，並列実行による性能向上が大きくなるようなスレッドに分割する手法を提案した．命令 `a` と命令 `b` の距離とは，命令 `a` から命令 `b` に至るまでのすべての経路について，それぞれの経路に存在する命令数をその経路を通る確率で重み付けした期待値である．この手法では，スレッド間でデータ依存関係にある命令間の距離の最小値をスレッドの並列実行による利得とする．そして，その利得が大きいスレッドを選択する．このように利得を見積もることにより，スレッド間のデータ依存関係が TLP に与える影響を考慮する．

岩田らの手法は，データ依存関係にある命令間の距離に着目しているため，同期の成立しなかった命令が後続命令の実行を妨げる，インオーダ・1 命令発行のプロセッサで構成されるマルチプロセッサでは，利得を正確に見積もることができる．しかし，SKY のようにスレッド内の ILP を利用し，ノンブロッキングな同期を行うマルチプロセッサに対しては，利得を正確に見積もることができないという問題がある．これを図 3 を用いて説明する．

図 3(a) に，スレッドに分割する前のプログラムの命令列を示す．破線は命令間にデータ依存関係があることを示す．この命令列に，命令 `i1` の前に `fork` 命令を，命令 `i4` の前に `finish` 命令を挿入するスレッド分割の利得を計算することを考える．図 3(b) に，インオーダ・1 命令発行のプロセッサで構成されるマルチプロセッサにおける実行例を示す．岩田らの距離に着目した方法では，上で述べたように図 3(b) のような状態を想定していることになる．その結果，命令 `i3`，`i5` 間で生じるスレッド間データ依存が並列実行を制限し，利得はほとんどないと計算される．このため，このスレッド分割は選択されない．これに対し，図 3(c) に，SKY における実行例を示す．ノンブロッキングな同期により，命令 `i5` と依存関係にない命令 `i6-i9` までは命令 `i5` よりも早く実行可能である．また，スレッド内の ILP を利用できる．これから，岩田らの距離に着目した利得計算では利得が少ないと計算され選択されないスレッド分割の中には，実際には性能向上に寄与するものが存在するといえる．

このように，岩田らの手法では，スレッド内の ILP を利用し，ノンブロッキングな同期を行うマルチプロセッサに対して，利得を正確に計算することができず，性能向上の大きいスレッドへの分割が十分ではないといえる．この問題に対して我々は，スレッド間とスレッド内のデータフローを考慮してスレッド分割を行う手法を提案する．

スレッド内のデータフローを解析することで，図 3(c) の命令 `i6-i9` のような，SKY でのスレッド並列実行においてオーバラップ実行される命令を検出することができる．なぜならば，ILP およびノンブロッキングな同期の影響は，スレッド内のデータ依存関係で決まるためである．よって，データフローを考慮することで利得計算が正確になり，その結果として性能向上に寄与するスレッドに分割することが可能となる．

4. データフローを考慮したスレッド分割

本章では，はじめに提案手法であるデータフローを考慮したスレッド分割技法について述べる．これは，データフローを考慮して利得計算を行うものである．次に利得計算をより正確にする方法について述べる．

4.1 提 案 手 法

3 章で述べたように，岩田らの手法ではノンブロッキングな同期とスレッド内の ILP を利用している点を考慮できない．提案手法では，これらを正確に利得に反映できるようにする．

利得はオーバラップ実行される命令数から計算している．オーバラップ実行される命令数は，`fork` 命令によって生成されたスレッド（子スレッド）の命令のうち，親スレッドの実行が終了するまでに実行される命令数のことである．そこで，親スレッドの実行終了サイクルと子スレッドの各命令の実行サイクルを見積もることで利得計算を行う．実行サイクルを見積もることで図 3(c) のように実行されることを利得に反映することができる．

実行サイクルの見積もりには**リスト・スケジューリング** [3] を用いる．リスト・スケジューリングによる実行サイクルの計算は，命令に対応するノードと，命令の実行順序を決定する制約（これを**先行制約**と呼ぶ）を表すエッジからなる先行制約グラフ（**PCG**: Precedent Constraint Graph）を使用して行うものである．一般的な先行制約はデータ依存であるが，本手法では命令の実行サイクルを正確に計算することが目的であるので，データ依存の他にプロセッサの構成要素であるリオーダ・バッファ（ROB: ReOrder Buffer）とロード／ストア・キュー（LSQ: Load/Store Queue）に関する制約を追加した（詳細は 4.2 節で述べる）．ノードは先行制約がなくなり次第実行可能となり，機能ユニットが使用可能な最も早いサイクルにおいて実行されると計算する．このようにして各命令の実行サイクルを見積もる．

利得計算の手順を以下に示す．以下の説明において，親スレッドとは利得計算対象のスレッド分割のフォーク点から子の開始点まで，子スレッドとは子の開始点以降のことを表す．

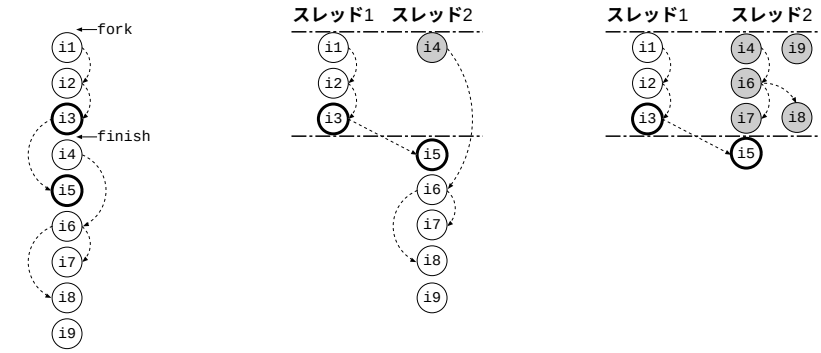

図 3 距離を用いた利得計算手法の問題点

2 章で述べたように，利得計算は各スレッド分割のフォーク点から子の開始点を経由し，関数の出口に至るまでの経路毎にオーバラップ実行される命令数の計算を行う．計算対象の経路について，親スレッドの PCG を作成し，親スレッドの実行終了サイクル，子スレッドにレジスタ値を送る send 命令の実行サイクルとストア命令がデータをメモリに書き込むサイクルを計算する．send 命令の実行サイクルとストア命令の書き込みサイクルは，スレッド間データ依存を満足するよう子スレッドの命令をスケジューリングするために使用する．次に，子スレッドの PCG を作成し，子スレッドの各命令の実行サイクルを計算する．このとき，スレッド間レジスタ依存がある命令は，当該レジスタが親スレッドの send 命令によって送信され，子スレッドで受信するサイクルまで実行できないとすることで，スレッド間レジスタ依存を考慮する．また，スレッド間メモリ依存がある命令は，親スレッドの依存関係にあるストア命令がメモリにデータを書き込むサイクルまで実行できないとすることで，スレッド間メモリ依存を考慮する．計算した実行サイクルが親スレッドの終了サイクルよりも早い場合，オーバラップ実行されるとする．子スレッドの命令の実行サイクルが親スレッドの終了サイクルを超えたら計算を終了し，それまでにオーバラップ実行された命令数を数える．これをすべての経路について行い，各経路を通る確率で重み付けした期待値をそのスレッド分割の利得とする．

4.2 利得計算をより正確にするための先行制約の追加

命令の実行サイクルを決定する要因はデータ依存と機能ユニットだけではない．プロセッサの他の構成要素も先行制約として加味することで，利得を実際にオーバラップ実行される命令数にさらに近づけることができると考えられる．本論文では，実行を大きく制限していると考えられる ROB と LSQ について，命令の実行サイクルの計算に反映させた．これらをそれぞれ，ROB の制約と LSQ の制約と呼ぶことにする．

命令は ROB のエントリが割り当てられないと実行不可能であることから，ROB を考慮した先行制約を追加した．ROB のエントリはプログラム順に割り当て，実行が終了した命令からプログラム順に解放する．ROB のエントリが割り当て不可能な場合とは，ROB に新たな命令を割り当て可能なエントリが残っていない場合である．これは，その時点で ROB のエントリが割り当てられている命令の中で，プログラム順で最も古い命令がリタイアできない場合である．すなわち，あるノードに対する ROB の制約とは，すでに ROB に割り当てられている命令の中でプログラム順で最も古い命令がリタイアするまで実行できないというものである．ある命令に ROB のエントリを割り当てることができない状況でのプログラム順で最も古い命令は ROB のエントリ数分前の命令であるので，全命令に対しプログラム順に番号をつけることでこの制約は簡単に追加することができる．実行サイクルの計算時に，命令のリタイアを管理することで ROB の制約を考慮することができる．

LSQ を考慮した先行制約は，ロード／ストア命令をアドレス計算部とメモリ・アクセス部とに分け別々にスケジューリングすること，またロード命令のメモリ・アクセス部のスケジューリングに影響があることから追加した．ロード／ストア命令を PCG に追加するには，まず，アドレス計算部とメモリ・アクセス部の 2 つのノードに分ける．これによりアドレス計算部とメモリ・アクセス部を別々にスケジューリングすることを考慮できる．また，ロード命令は先行するストア命令の実効アドレスがすべて判明するまで実行できないことから，ロード命令のメモリ・アクセス部に対応するノードの先行制約として，真のデータ依存の他に，先行ストア命令のアドレス計算部が実行完了するまで実行できないという制約を加える．上記 2 点の変更によって，LSQ の制約を考慮することができる．

5. 評　　価

本章ではまず，評価環境について述べる．次に，データフローを考慮した利得計算を用いてスレッド分割を行った場合の評価を行う．

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．使用したベンチマーク・プログラムとその入力セットを表 1 に示す．この表において，プロファイル用の入力はコンパイラがスレッド分割時に使用したものを，シミュレーション用の入力は評価に使用したものを示す．ベンチマーク・プログラムのバイナリは，GNU GCC Version 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タを用いて行った．トレースは SimpleScalar Tool Set Version 3.0 を利用して採取した．

表 2 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 1 つのプロセッサとした．表 3 に，各ベンチマークにおける基準プロセッサの IPC を示す．SKY のプロセッサ数は 8 とした．

スレッド分割を生成する際に使用した利得計算手法が，距離を基準としたものを DIST，データフローを基準としたものを DF とした．また，DF において ROB の制約を考慮したものを DF w/ R，ROB と LSQ の制約を考慮したものを

表 1 使用したベンチマークと入力セット

ベンチマーク	プロファイル用	シミュレーション用
compress95	train/test.in	30000 e 2231
gcc	ref/regclass.i	ref/genoutput.i
go	11×11 board level 4, ref/null.in	9×9 board level 6, train/2stone9.in
ijpeg	train/vigo.ppm, 68 × 48 pixcels	ref/specmun.ppm
li	test/test.lsp	train/train.lsp
m88ksim	test/ctl.in, test/dhry	train/ctl.in, train/dcrand
perl	train/jumble.pl train/jumble.in,	train/scrabbl.pl, train/scrabbl.in (add three words)
vortex	ref/persons.1k, ref/vortex.in (reduced iterations)	ref/persons.1k, ref/vortex.in

表 2 SKY の基本モデル

(a) プロセッサ

命令フェッチ幅	8 命令
命令デコード幅	8 命令
命令発行幅	8 命令
命令コミット幅	8 命令
命令ウィンドウ	64 エントリ
ROB	128 エントリ
LSQ	128 エントリ

(b) 共有資源

分岐予測機構	1024 エントリ，履歴長 4 の PAp 分岐予測ミスペナルティ4 サイクル
命令キャッシュ	完全
データキャッシュ	完全
メモリ曖昧性検出機構	理想

表 3 基準プロセッサの IPC

ベンチマーク	IPC
compress95	3.51
gcc	2.82
go	2.36
ijpeg	4.82
li	3.48
m88ksim	4.39
perl	3.75
vortex	4.20
GM	3.59

DF w/ R,L とした．以後の図のベンチマーク名の表記において，compress95，m88ksim，vortex をそれぞれ comp.，m88k.，vort. と表す．また，GM は全ベンチマークの幾何平均を表す．

5.2 データフローを考慮したスレッド分割技法の評価

はじめに，利得計算方法を変更したことにより選択されたスレッド分割がどのように変化したかについて述べる．表 4 に利得計算が DIST，DF w/ R,L における静的なスレッド分割数を示す．

表からわかるように，利得計算を変更したことにより，選択されたスレッド分割数が増加した．これは，DIST ではスレッド間データ依存のために利得が小さいと判断されたスレッド分割が，DF w/ R,L ではノンブロッキングの効果により利得が大きいと判断されたためである．これから，DF w/ R,L では距離を用いた利得計算では見つけることのできなかったスレッド分割を選択できたといえる．

次に，利得計算を提案手法にした場合の評価結果について図 4 に示す．この図において，縦軸は基準プロセッサに

表 4 静的スレッド分割数

ベンチマーク	DIST	DF w/ R,L
compress95	8	11
gcc	293	888
go	533	580
ijpeg	95	162
li	2	12
m88ksim	14	48
perl	57	169
vortex	132	543

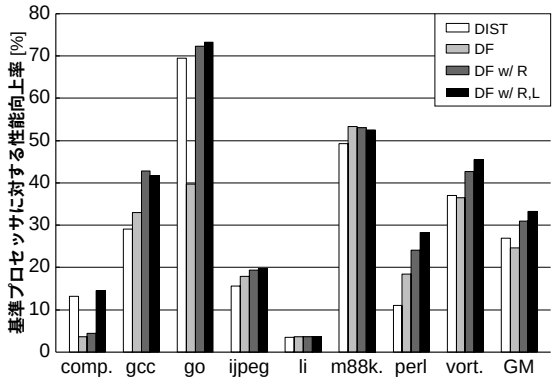

図 4 各利得計算手法による性能向上率

対する性能向上率を，横軸はベンチマーク名を表す．各ベンチマークに 4 本の棒グラフがあり，左から利得計算方法を DIST，DF，DF w/ R，DF w/ R,L として生成したスレッド分割による性能向上を示す．

図からわかるように，DIST に対して DF は性能を向上させることができなかった．最大で 7.4%の性能向上，平均では 2.3%ポイントの性能低下であった．これは，データフローと機能ユニットを考慮しただけでは命令の実行サイクルの計算の誤差が大きく，利得計算の精度が低いためと考えられる．

これに対して，ROB の制約を考慮することで，多くのベンチマークで性能が向上した．DF w/ R は DF に対して最大で 32.6%ポイント，平均では 6.3%ポイントの性能向上となった．さらに LSQ の制約も考慮することで性能は向上し，DF w/ R,L は DF w/ R に対して最大 10.2%ポイント，平均では 2.3%ポイントの性能向上となった．また，DIST に対し，最大で 17.2%ポイント，平均では 6.3% ポイントの性能向上となった．以上より，データフローに加えて，プロセッサの資源を考慮することは実行サイクルの計算に有効であったといえる．ROB ／ LSQ の制約は共に効果があったが，中でも ROB の制約の効果が大きいことがわかった．これは，ROB がプロセッサにおける命令の実行に大きな影響を及ぼしているためと考えられる．

以上の結果から，データフローとプロセッサの資源を考慮することによって，DIST と比べて正確に性能向上を計算することができるようになり，実際に性能向上に寄与するスレッド分割を選択できるようになったといえる．

図 4 において，DF w/ R,L と DIST を比較すると，性能向上の差が大きいプログラムと，ほとんど差がないプログラムとがある．これは，距離を用いた利得計算で十分正確に利得を計算できているプログラムでは，提案手法の有効性が低くなるためと考えられる．そこで，コンパイラが計算した利得から，コンパイラの見積もった性能向上を *espeedup* として以下の式で計算し，DIST と DF w/ R,L とを比較した．

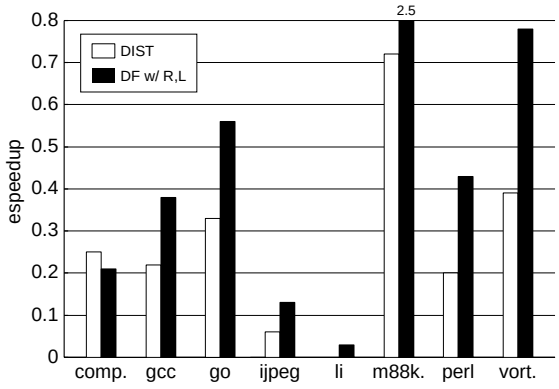

図 5 コンパイラにおける性能向上の見積もり

$$espeedup = \alpha \frac{\sum_{t \in F} e(t)g(t)}{N_{inst} - \sum_{t \in F} e(t)g(t)}$$

ここで，N_{inst} は全動的命令数，$e(t)$，$g(t)$ はそれぞれスレッド分割 t の実行回数および利得である．また，α はマルチスレッド実行により生じるオーバヘッドを表す係数で，$0 < \alpha < 1$ である．N_{inst}，$e(t)$，α は測定により求めた．計算結果を図 5 に示す．この図において，縦軸は *espeedup*，横軸はベンチマーク名を表す．各ベンチマークに 2 本の棒グラフがあり，利得計算が左は DIST，右は DF w/ R,L である．

図 4 において DIST に対して DF w/ R,L の性能向上が大きかった gcc，perl は，*espeedup* でも差が大きい．よって，これらは提案手法が有効に働いているといえる．また，図 4 において DIST に対して性能が向上しなかった compress95，li は *espeedup* でも差がないので，提案手法が有効に働かないベンチマークであるといえる．go，ijpeg，m88ksim，vortex は *espeedup* では DIST に対して性能向上が見込めるが，実際には見積もりほど性能の差はない．この原因は子スレッドの生成のタイミングをコンパイラでは正確に計算できないためと考えられる．なぜならば，SKY では，子スレッドを生成する `fork` 命令は制御が確定するまで実行できないとしているが，コンパイラでは制御が確定するタイミングがわからないためである．

図 5 からは，コンパイラは性能が向上するようなスレッド分割を選択しているといえる．しかし，実際の性能向上はコンパイラの見積もりよりも小さいものが多い．より高い性能向上を得るためには，コンパイラがハードウェアの動作をより正確に利得計算に反映させるか，ハードウェアをコンパイラの想定に近い動作をするように改良することが必要であると考えられる．

6. ま と め

細粒度の TLP を効率よく利用するハードウェアに適したスレッドに分割するため，データフローを考慮したスレッド分割技法を提案した．SKY の特徴であるノンブロッキングな同期を考慮したことにより，命令間の距離を用いた従来のスレッド分割と比較して，データフローを用いたスレッド分割は，最大 17.2%ポイント，平均 6.3%ポイントの性能向上を達成した．

謝辞 本研究の一部は，文部科学省科学研究費補助金基盤研究（C）課題番号 15500036，文部科学省 21 世紀 COE プログラム，財団法人栢森情報科学振興財団研究助成の支援により行った．

参 考 文 献

[1] A. V. Aho et al., *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.

[2] L. A. Barroso et al., “Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,” In *Proc. ISCA-27*, pp.282-293, June 2000.

[3] Jr. E. G. Coffman, *Computer and Job-Shop Scheduling Theory,* John Wiley and Sons, New York, 1976.

[4] K. Diefendorff, “Power4 Focuses on Memory Bandwidth: IBM Confronts IA-64, Says ISA Not Important,” In *Microprocessor Report*, Vol. 13, No. 13, Oct. 1999.

[5] P. N. Glaskowsky, “IBM Previews Power5,” In *Microprocessor Report*, 9/8/03-02, Sep. 2003.

[6] P. N. Glaskowsky, “IBM Raises Curtain on Power5: More Details Disclosed at Microprocessor Forum,” In *Microprocessor Report*, 10/14/03-01, Oct. 2003.

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

[8] 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.

[9] 木村ほか, “シングルチップマルチプロセッサ上での近細粒度並列処理,” 情報処理学会論文誌, Vol.40, No.5, pp.1924-1933, 1999 年 5 月.

[10] 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 2 月.

[11] K. Krewell, “UltraSPARC IV Mirrors Predecessor: Sun Builds Dual-Core Chip in 130nm,” In *Microprocessor Report*, 11/10/03-02, Nov. 2003.

[12] K. Krewell, “Fujitsu Makes SPARC See Double: SPARC64 VI Uses Process Shrink to Double Cores,” In *Microprocessor Report*, 11/24/03-01, Nov. 2003.

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

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

[15] G. S. Sohi et al., “Multiscalar Processors,” In *Proc. ISCA-22*, pp.414-425, June 1995.

[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] T. N. Vijaykumar et al., “Task Selection for a Multiscalar Processor,” In *Proc. MICRO-31*, pp.81-92, Dec. 1998.