

制御等価を利用したスレッド分割技法

岩田 充晃[†] 小林 良太郎[†]
安藤 秀樹[†] 島田 俊夫[†]

我々は、スレッド間命令レベル並列を利用するプロセッサ・アーキテクチャ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、このSKYのためのコンパイラのスレッド分割技法について述べる。これは制御等価な基本ブロックに着目し、必要なレジスタ通信を求め、利得計算することによってプログラムをスレッド分割するものである。この分割技法でSPECint95のプログラムをスレッド分割することにより、8個のプロセッサ要素を持つSKYはスーパスカラに対して1.42倍の性能を達成することができた。

A Thread Partitioning Technique that Utilizes Control Equivalence

MITSUAKI IWATA,[†] RYOTARO KOBAYASHI,[†] HIDEKI ANDO[†]
and TOSHIO SHIMADA[†]

We have proposed the processor architecture SKY, which exploits interthread instruction-level parallelism. 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 describes a thread partitioning technique of the compiler for SKY. It partitions a program into threads by finding control equivalent basic blocks, generating register communication information and calculating performance gain. We partitioned the SPECint95 benchmark programs into threads by means of this technique and executed them with the SKY simulator. Our evaluation results show that SKY with eight processor elements achieves 1.42x speedup over superscalar machines.

1. はじめに

プロセッサの処理能力は、プログラム内に存在する並列性を利用することで向上させることができる。並列性を制約する要因は、そのプログラムに存在する依存関係である。依存関係にはデータ依存、制御依存などがある。これらの依存の中で、並列性を制限する要因としては制御依存が最も大きい。

現在、商用の主なスーパスカラ・プロセッサは、この制御依存を解消する方法として投機的実行を用いている。これは分岐先が確定する前に分岐方向を予測し、その方向の命令を実行する技術である。しかし、Wallらの研究結果¹⁾は、現在のスーパスカラの投機的実行による並列度は限界に近付いているということを示している。

これに対してLamらは、投機的実行に加えて、制御依存解析と複数命令流実行を導入すれば、制御依存を大幅に緩和できるという研究結果を示した³⁾。このことは、現在のスーパスカラに代表されるような投機的実行だけを備えたプロセッサが限界に達したとき、制御依存解析と複数命令流実行の技術も利用したプロセッサが重要になってくることを意味している。

そこで、我々はこれら3つの技術を利用したアーキテクチャSKYを提案した²⁾。SKYは、複数の独立したプロセッサ要素(PE)を持ち、各PE間に高速なレジスタ通信機構を備えたアーキテクチャである。各PEでスレッドを実行することにより複数命令流実行を実現する。スレッドとは、コンパイラによって分割されたプログラムの断片である。コンパイラは、プログラムの制御依存解析などを行なうことによってプログラムを複数

のスレッドに分割する。従来の投機的実行は各PEによって行なわれる。このようにしてSKYは投機的実行、制御依存解析、複数命令流実行という3つの技術を融合し、性能向上を図っている。

SKYにより大きな性能向上を得るためには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。なぜなら、コンパイラが並列性を見つけ出せずスレッド分割ができなければ、SKYは1つのPEしか動かず並列実行できない。また、たとえスレッド分割ができたとしてもスレッド間のデータ依存による通信のオーバーヘッドが多ければスレッド並列実行の効果は得られない。したがって、通信のオーバーヘッドが少なく、並列実行による利得が得られると期待されるようなスレッドに分割することがコンパイラに要求される。

我々はこのような要求を満たすため、プログラム内の制御等価な部分に着目したスレッド分割技法を開発した。本稿ではこのスレッド分割技法の詳細について述べ、スレッド分割されたプログラムをSKYで実行し、評価を行なう。2章ではSKYの概要について述べる。3章ではスレッド分割アルゴリズムを説明する。4章で評価を行ない、5章でまとめる。

2. SKYの概要

この章ではSKYの概要について述べる。まずSKYの構成を述べた後、スレッドについて説明する。次にフォーク、通信というSKYの動作を説明することによって、SKYでスレッドがどのように並列実行されるかを述べる。詳細は文献[2]を参照されたい。

[†]名古屋大学工学部

School of Engineering, Nagoya University

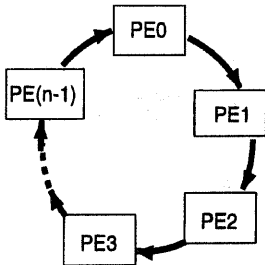


図1 各PEの接続

2.1 SKYの構成

SKYは n 個のPEを備え、それぞれのPEでスレッドを並列実行することにより性能向上をはかるアーキテクチャである。各PEは、図1に示すようにリング状に結合される。各PEは従来のスーパーカスカラ方式のプロセッサとなっており、スレッド内の命令レベル並列性を引き出す。機能ユニット、リザベーションステーション、レジスタファイルなどはそれぞれ独立して持っている。メモリは各PEで共有する。命令キャッシュ、データキャッシュも共有である。

2.2 スレッド

スレッドとはコンパイラによって分割されたプログラムの断片である。図2(a)の太い縦の線はプログラムの動的な命令列を表す。コンパイラは、図2(b)に示すように動的な命令列が先頭から順に区切られているようにプログラムを分割する。このように分割することにより、前方のスレッドへの依存は存在する可能性があるが、後方への依存は存在しない。

例えば図2(b)に示すように命令列の先頭から順に各スレッドに $Th_0, Th_1, Th_2, \dots$ と名前をつける。元の逐次プログラムにおいて Th_1 は Th_0 よりも後に、 Th_2 は Th_1 よりも後に実行される部分である。この場合、一般に Th_i, Th_j ($i < j$)において、 Th_i から Th_j への依存は存在する可能性があるが Th_j から Th_i への依存は存在しない。

2.3 フォーク

SKYは、コンパイラによって作られたスレッドをリング上に接続したPEで順に実行していくことによりスレッドを並列実行する。すなわち、図2(c)に示すように Th_m を実行している PE_m が、その実行途中で $Th(m+1)$ を $PE((m+1) \bmod n)$ に割り当て起動するということを繰り返す。このように新たにスレッドを起動することにより、フォークという。スレッドのフォークはSKYの専用命令であるFORKによって行なう。フォークによって各スレッドはオーバーラップして実行され、スレッド並列実行が実現する。

スレッドの実行の停止は専用命令FINISHによって行なう。FINISH命令によって PE_m は実行を終え、待機状態になる。待機状態とは隣のPEからのフォークを待っている状態である。プログラムを実行し始めたときはPE0以外のPEはすべて待機状態である。

待機状態でないPEに対してフォークを行なうことはできない。例えば図2(c)において $PE(n-1)$ はPE0に対して Th_n をフォークしているが、このときPE0が待機状態でなく、まだ Th_0 を実行中であったとすると Th_n をフォークできない。この場合 Th_n は $PE(n-1)$ において $Th(n-1)$ の実行終了後に実行される。 Th_n から $Th(n-1)$ への依存はないので、これによってプログラムの意味が変わることはない。その後、 Th_n を実行している $PE(n-1)$ はPE0に対して $Th(n+1)$ のフォークを試みる。失敗すれば $PE(n-1)$ において実行する。

2.4 通信

データ依存には2種類ある。レジスタに関するものとメモリに関するものである。

レジスタのデータ依存はSKYのレジスタ通信機構を利用することにより解消する。この機構を利用する命令がSEND命令

である。 PE_m でSEND命令が実行されると、指定されたレジスタが $PE((m+1) \bmod n)$ のリザベーションステーションとレジスタファイルに送られる。レジスタ値をリザベーションステーションに直接転送するパスがあるため、この通信のレイテンシは小さい。 PE_m は $PE((m+1) \bmod n)$ に対してのみ、つまり一方にしかレジスタを転送できない。しかし $Th(m+1)$ から Th_m への依存はないようにコンパイラがスレッド分割を行うので逆方向の通信が必要になることはない。

メモリのデータ依存はハードウェアで暗黙的に解消しているが、その機構については省略する。

2.5 フォークと通信の制限

以上述べてきたように、フォークとレジスタ通信は隣りの1つのPEに対してしか行えない。つまり PE_m は $PE((m+1) \bmod n)$ に対してのみフォークやレジスタ通信を行えるということである。各PEの接続は図1の矢印が示すように一方通行のリングとなる。フォークや通信の自由度は小さくなるが、通信機構等に要するハードウェアがかなり簡略化されるという利点がある。

3. プログラムのスレッド分割

前節でSKYにおいてどのようにプログラムが実行されるか、その概略を説明した。このことからSKYのコンパイラには、スレッドの並列実行による性能利得が大きくなるようにプログラムの分割を行ってFORK, FINISH命令を挿入し、通信すべきレジスタを求めてSEND命令を挿入する作業を行う必要がある。この章ではこれらの作業をいかに行うか、ということについて述べる。

3.1 定義

まず、これ以降の説明のために記号と言葉の定義を行う。

プログラム中のある関数の制御フローグラフ G を $G = (N, E, s, t)$ と表す。ここで、 (N, E) は有向グラフで N は節(基本ブロック)の集合、 $E \subseteq N^2$ は辺の集合、 s は開始節(関数の入口)で、 t は終了節(関数の出口)である。一般に関数の出口は1つとは限らないが、複数ある場合でもダミーの節を作り、関数の各出口からこの節への辺を作って終了節を1つにする。 $n \in N$ の先行節の集合 $pred(n)$ と $n \in N$ の後続節の集合 $succ(n)$ を次のように定義する。

$$pred(n) = \{x \in N | (x, n) \in E\}$$

$$succ(n) = \{x \in N | (n, x) \in E\}$$

$n \in N$ を支配¹⁾する節(支配節)の集合 $dom(n)$ は開始節 s から n に至るすべての経路で必ず通る節の集合である。 $n \in N$ を後支配²⁾する節(後支配節)の集合 $pdom(n)$ は n から終了節 t に至るすべての経路で必ず通る節の集合である。すなわち、 $x \in N$ から $y \in N$ へ至るある経路をその経路が通る節の集合 P として表し、 x から y へ至るすべての経路 P の集合を $path(x, y)$ と書くとき、

$$dom(n) = \{x \in N | (\forall P \in path(s, n)) x \in P\}$$

$$pdom(n) = \{x \in N | (\forall P \in path(n, t)) x \in P\}$$

で定義される集合である。 $x \in N$ が $y \in N$ を支配し、 y が x を後支配するとき x と y は制御等価⁵⁾であるという。制御等価な基本ブロックの組の集合 C は次のように書ける。

$$C = \{(x, y) \in N^2 | x \in dom(y) \text{ かつ } y \in pdom(x)\}$$

あるスレッド Th_m が $Th(m+1)$ をフォークしたとする。このとき Th_m を $Th(m+1)$ の親スレッドといい、 $Th(m+1)$ を Th_m の子スレッドという。親スレッドが子スレッドをフォークする位置をフォーク点といい、子スレッドの先頭を子の開始点という。

ある命令 B が命令 A の結果に依存しているとき、命令 A をこの依存の依存元、命令 B をこの依存の依存先という。

命令 A と命令 B の距離とは制御が命令 A から命令 B に到達するまでの時間を見積もったものである。フォーク点から子の開始点までの距離をフォーク距離という。依存元と依存先の

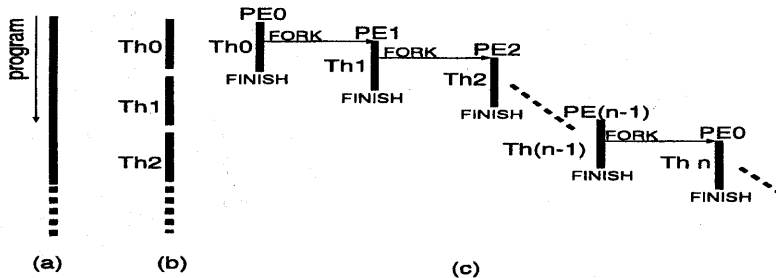


図2 スレッドの分割と実行

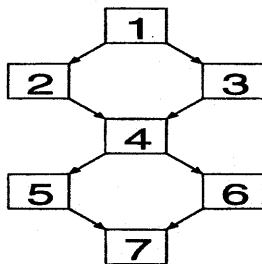


図3 制御フローグラフ

距離を依存距離という。

3.2 概要

3.2.1 基本方針

制御等価な基本ブロックの組に着目してスレッド分割を行うことを基本方針とする。ここではその理由を述べる。

基本ブロック x のある点をフォーク点とし、基本ブロック y のある点を子の開始点とすることを考える。すなわち y の子の開始点より前の部分を親スレッドとし、子の開始点以降の部分を子スレッドとして分割する場合である。

このとき元の逐次プログラムにおいて、 x を実行する時点で y がいずれ実行されることが確定していなければならない。言い換えれば $y \in pdom(x)$ である必要がある。なぜなら、いったん子スレッドがフォークされると、それは親スレッドがたどる制御フローとは関係なく実行が進んでしまうからである。

このような理由で一般にはすべての(被後支配節, 後支配節)の組に着目して分割すればよい。具体的には被後支配節にフォーク点を、後支配節に子の開始点を設け、この分割による利得を計算し、利得の多いものを残す。しかしこれは現実的ではない。なぜなら、(被後支配節, 後支配節)の組の数はプログラムが大きくなると爆発的に増え、分割の際の計算量が膨大なものになってしまうからである。

そこで我々は被後支配節の中でも後支配節を支配するもの、すなわち制御等価な基本ブロックの組にのみ着目して分割することにする。これにより計算量はかなり削減される。

3.2.2 分割処理の流れ

スレッド分割をするときの全体の大まかな処理の流れについて述べる。まずその流れを以下に示す。

前処理 (3.3節);

各関数について {

制御等価な基本ブロックの組の集合 C を求める;

C の各要素について {

f, c を求める (3.4.1節);

$Comm$ を求める (3.4.2節);

利得計算を行う (3.4.3節, 3.4.4節);

利得が 0 でないなら F に加える;

}
 F の中から選択する (3.5節);
 命令挿入 (3.6節);
 }

最初にスレッド分割に必要な各種の情報を得るために前処理を行う。次に実際に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
- レジスタ通信集合 $Comm$
- この分割による利得値 g

これら値及び集合の組 $(f, c, Comm, g)$ を分割情報と呼ぶ。分割情報の集合を F とする。

ある関数の中で制御等価な基本ブロックをすべて見つける。すべての制御等価な基本ブロックの組に着目して分割し、 f, c を求める。その f, c からレジスタ通信集合 $Comm$ を求める。これは f から c にフォークするした場合に必要なとなるレジスタ通信の情報の集合である。

次にこの分割でどれだけの性能利得が見込めるかを計算する。性能利得があらかじめ定めた値以下で非常に小さい場合、利得計算ルーチンは利得値として 0 を返す。利得値が 0 となる分割は分割候補にはせず、破棄する。利得値が 0 でないなら、分割情報 $(f, c, Comm, g)$ を分割候補 F に加える。

このようにして分割候補 F を作ったあと、 F の中から利得が最大になるような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 の各命令を挿入する。

3.2.3 具体例

図 3 に示す制御フローグラフを持つ関数があるとする。前処理をしたのち、制御等価な基本ブロックの組の集合 C を求める。この例では $C = \{(1, 4), (4, 7), (1, 7)\}$ である。

まず、(1, 4) に着目すると、1 から 4 へのフォークが可能であることが分かる。したがって、 $\{1, 2, 3\}$ と $\{4, 5, 6, 7\}$ という二つのスレッドに分割し 1 の中のある点をフォーク点、4 の中のある点を子の開始点とする。次にこの分割によって必要となるレジスタ通信集合を求める。この場合は 1, 2, 3 のどこかを依存元とし、4, 5, 6, 7 のどこかを依存先とするレジスタ依存が、通信すべきものとなる。そのあと利得値を計算する。この値を g_{14} とする。この分割による分割情報を $info_{14}$ と書く。

(4, 7) に着目すれば $\{4, 5, 6, 7\}$ はさらに $\{4, 5, 6\}$ と $\{7\}$ という二つのスレッドに分けられる。この場合フォーク点は 4 の中で、子の開始点は 7 の中となる。同じようにレジスタ通信を求め、利得を計算し利得値 g_{47} を得る。この分割による分割情報を $info_{47}$ と書く。結局 (1, 4) と (4, 7) に着目することにより $\{1, 2, 3\}, \{4, 5, 6\}, \{7\}$ の 3 つのスレッドに分割できた。

次に (1, 7) に着目する。この場合は $\{1, 2, 3, 4, 5, 6\}$ と $\{7\}$ の 2 つのスレッドにわかれ、1 から 7 へフォークすることになる。この分割の利得値を g_{17} とする。分割情報を $info_{17}$ と書く。

$\{(1, 2, 3), \{4, 5, 6\}, \{7\})$ とスレッド分割するか、それとも

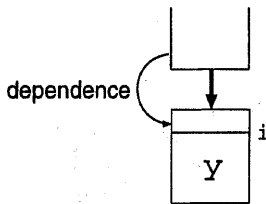


図4 子の開始点

{{1,2,3,4,5,6},{7}}とスレッド分割するかどちらが良いかは後の3.5節で述べる「選択」によって判断する。この場合は $g_{14} + g_{47}$ と g_{17} を比較する。 $g_{14} + g_{47} \geq g_{17}$ であれば {{1,2,3},{4,5,6},{7}}とスレッド分割するほうがよいと判断して $info_{14}$ と $info_{47}$ を選択し、 $info_{17}$ を破棄する。 $g_{14} + g_{47} < g_{17}$ であれば {{1,2,3,4,5,6},{7}}とスレッド分割するほうがよいと判断して $info_{17}$ を選択し、 $info_{14}$ と $info_{47}$ を破棄する。

このようにして最終的な分割情報の集合を求める。

3.3 前処理

スレッド分割を行なうには、分割の対象となるプログラムのさまざまな情報を取得しておく必要がある。この情報を求める作業が前処理である。具体的には以下の作業である。

- プログラムを基本ブロックに分割する
- 制御フローグラフを作る
- 支配節及び後支配節を求める
- レジスタの依存を解析する
- メモリの依存を解析する
- プロファイルをとる

最初の4つは静的にプログラムを解析することによって行なう。プログラム内の関数ごとにこれらの情報を求める。

最後の2つは少し説明を要する。まずメモリの依存は命令のトレースから解析する。命令のトレースでは、ロード命令、ストア命令のアクセスするアドレスが確定しているのて容易に解析できる。命令トレースから解析しているので、解析結果はそのトレースをとったときのプログラムの入力に依存している。実際は依存している可能性があるのに、その入力ではたまたま依存がなかったため依存なしと判定されるかもしれない。しかしある入力のトレースは概ね全ての場合を代表していると考えられる。また、メモリ依存解析の結果は後述するように利得計算を行なうのに用いているだけなので、一部で正しくない解析があったとしてもプログラムの意味を変えてしまうことはない。

プロファイル情報も命令トレースから取得する。具体的には基本ブロックの実行回数、関数が呼び出された回数などの情報を取得する。これらの情報はメモリ依存解析結果と同様に利得計算に使用する。

3.4 分割候補

3.4.1 フォーク点と子の開始点

制御等価な基本ブロックの組 $(x, y) \in C$ が与えられると、フォーク点 f を x の中のある点、子の開始点 c を y の中のある点に定めなければならない。問題は x のどこをフォーク点とし y のどこを子の開始点にすればよいかということである。

簡単な方法の一つとして、 x の先頭をフォーク点、 y の先頭を子の開始点とする方法が挙げられる。しかし、この方法は最大の並列性を抽出するという点で最速でない。

例えば図4で示すように親スレッドの終了直前を依存元とし y の先頭になり近い点を依存先とするレジスタ依存があったとする。子の開始点を y の先頭に定めるとする。この場合、依存のためレジスタを通信する必要性が生じる。子スレッドはフォークされるとまもなくレジスタが送られてくるのを待ってスタートする。そのレジスタは親スレッドの終了間際で送信される。そのため親スレッドと子スレッドが並列実行されている時間はほとんどなく、逐次的にスレッドを実行しているのとあまり変

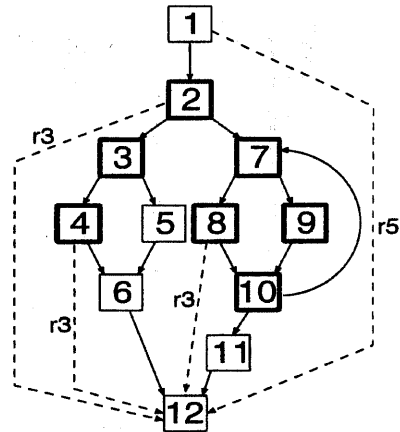


図5 制御フローグラフ

わらなくなってしまう。

このことは子の開始点を依存先以降の i の位置にずらすことによって回避できる。なぜなら子の開始点を i にすれば、依存先が親スレッドに含まれることになり、子スレッドにレジスタを送る必要がなくなるからである。

このようにできるだけスレッドが並列に実行できるようにフォーク点、子の開始点を定める。

3.4.2 レジスタ通信

レジスタ通信を求める作業は、フォーク点 f と子の開始点 c を入力とし、レジスタ通信集合 $Comm$ を出力とする。レジスタ通信集合の各要素は (r, a, b) という3つの値の組である。 r は通信すべきレジスタの番号、 a は SEND 命令を挿入する位置を示す。 b は送られたレジスタを使う子スレッドの命令の位置^{*}で、SEND 命令の挿入には関係ないが利得計算で用いる。

図5に示す制御フローグラフを持つ関数があるとすると。この図では制御フローを実線の矢印で示し、レジスタのデータ依存は点線の矢印で示す。点線の矢印の横に付いている $r3$ などの記号は、そのデータ依存がどのレジスタに関するものであるかを示している。

いま、(2,12) という制御等価な基本ブロックの組に着目して分割を試みて行なうとする。フォーク点 f は2の先頭に、子の開始点 c は12の先頭に定めたとする。このとき通信しなければならないレジスタは、フォーク前に1において定義される $r5$ と、フォーク後に2,4,8において定義される $r3$ である。

フォーク前に定義され、子スレッド以降で使われるレジスタ(この例では $r5$)はフォーク直後に通信を行なえばよい。したがって SEND 命令挿入位置を表す a はフォーク点 f の直後にする。 b はこの依存の依存先とする。したがってフォーク前に定義されるレジスタの通信では (r, a, b) の組は容易に求められる。

これに対しフォーク後に定義されて、子スレッド以降で使われるレジスタの通信では SEND 命令挿入位置を求めるのが難しい。以下、どのように求めるかについて説明する。

図5の例では $r3$ は3つの異なる場所 2,4,8 で定義される。4で行なわれる定義は子スレッドに到達する。したがって4で行なわれる定義の直後は SEND 命令挿入位置の一つとなる。

これに対し2で行なわれる $r3$ の定義は子スレッドに到達するとは限らない。なぜなら2の定義が行なわれたあと、4や8で再定義される可能性があるからである。制御の流れが5に行ったとき、4,8には制御が行かないことが確定する。すなわち制御が5に流れたときは子スレッドに到達する $r3$ の定義は4や8で行なわれる定義ではなく、2で行なわれる定義であること

^{*} 正確にはレジスタ依存解析によって求められた依存先の位置である。依存解析の控え目な推定¹⁾によりそのレジスタを実際に使用する命令の位置とは異なる可能性がある。

が決定する。したがって5の先頭も SEND 命令挿入位置となる。
8で行なわれる定義は子スレッドに到達するとは限らない。なぜならループを回って再び8で違う値が定義されるかもしれないからである。この場合はループの出口である11で、8の定義が子スレッドに到達することが決定するので、11の先頭が SEND 命令挿入位置となる。

まとめると、子スレッドに確実に到達する定義の直後の点か、どの定義が到達するかが決定する点を SEND 命令挿入位置とすればよい。

これを求めるには次のようにする。まずあるレジスタ r に着目して、 r に関する依存でその依存元が f 以降 c 以前 (c は含まない) にあり依存先が c 以降にあるものを挙げる。そしてその依存元が属する基本ブロックの集合を E とする。図5の例では $E = \{2, 4, 8\}$ となる。

次に集合 D を決める。集合 D とは f を通過してから E に属する基本ブロックに至るすべてのパス上に存在する基本ブロックの集合である。この例では $D = \{2, 3, 4, 7, 8, 9, 10\}$ となる。図5では太線の基本ブロックで示してある。 D を求めた後、集合 R と集合 S を求める。

集合 R は、 D に属するある基本ブロックの後続節の一部が D に属し、一部が D に属していないとき、属していないほうの後続節の集合である。すなわち、

$$R = \{x \in N | (\exists d \in D) [(\exists s \in \text{succ}(d)) s \in D \text{ かつ } (x \in \text{succ}(d) \text{ かつ } x \notin D)] \}$$

である。この例では $R = \{5, 11\}$ となる。制御の流れが R に属する基本ブロックに入ったとき、レジスタ r の値が c に至るまで変更されないことが決定する。したがって、SEND 命令は R に属する基本ブロックの先頭に挿入すればよい。

集合 S は、 D に属する基本ブロックの中でその後続節がすべて D に属さないものの集合である。すなわち、

$$S = \{x \in N | (\forall s \in \text{succ}(x)) s \notin D \text{ かつ } x \in D\}$$

である。この例では $S = \{4\}$ となる。プログラムの流れが S に属する基本ブロックに入ったとき、その基本ブロックでレジスタ r が定義され、その値が c に至るまで生存するということが決定する。したがって、 S に属する基本ブロックの中でレジスタ r が定義される点の直後に SEND 命令を挿入すればよい。 $S \subseteq E$ なので S に属する基本ブロックの中には必ずレジスタ r の定義がある*。

以上のような作業をすべてのレジスタに対して行えば、レジスタ通信集合が求められる。

3.4.3 利得計算

利得計算はフォーク点 f 、子の開始点 c 、レジスタ通信情報の集合 $Comm$ を入力とし、その予想利得値を出力する作業である。利得計算の中心となるのは距離計算である。距離計算の詳細については3.4.4節において述べる。命令 i_1 から命令 i_2 までの距離を $\text{dist}(i_1, i_2)$ と書く。

利得計算は3つの要素からなる。それはフォーク距離、レジスタに関する依存の依存距離(レジスタの依存距離)、メモリに関する依存の依存距離(メモリの依存距離)である。これらの距離のうち、最も小さい距離だけスレッドがオーバーラップして実行される。したがって3つの距離の最小値を利得値とする。以下、3つの距離の計算方法について説明する。

フォーク距離 fd は、単純に f と c の距離そのものであり、

$$fd = \text{dist}(f, c)$$

である。

レジスタ依存距離 rd は SEND 命令とそれによって送られるレジスタを使う命令までの距離である。同じレジスタに関して依存距離の平均をとり、異なるレジスタの間でその平均の最小値をとった値を rd とする。すなわち、 $(r, a_{r,j}, b_{r,k}) \in Comm$ すると

$$rd = \min_{0 \leq r \leq r_{max}} \frac{\sum_{j=1}^{j_{max}} \sum_{k=1}^{k_{max}} \text{dist}(a_{r,j}, b_{r,k})}{j_{max} k_{max}}$$

と表される。 $r_{max}, j_{max}, k_{max}$ はそれぞれ r, j, k の最大値である。

レジスタ依存からレジスタ通信集合を作ったのと同じようにメモリ依存からもメモリ通信集合を作る。ただし、SEND 命令が挿入されるわけではなく依存距離の計算に使うだけである。メモリ依存集合の各要素は (d_j, u) で表す。 d_j は、あるアドレスの値が定義または決定される点、 u はそれを使用する点を表す。メモリ依存距離 md は、同じ依存先に関して依存距離の平均をとり、異なる依存先の間ではその平均の最小値をとった値とする。すなわち、

$$md = \min_u \frac{\sum_{j=1}^{j_{max}} \text{dist}(d_j, u)}{j_{max}}$$

である。

fd, rd, md にはそれぞれ下限の閾値を定め、すべてがこの閾値を上回らなければ、この分割を性能向上に寄与しない分割であるとみなし、利得値として0を返す。なぜならこれらの値があまりにも小さいときは、スレッドの実行がオーバーラップしている時間よりもフォークのオーバーヘッドの方が大きいと考えられるからである。

fd には上限値も定め、これを越えると利得値として0を返す。 fd が大きいとスレッドがあまりにも長く PE を占有してしまい、その PE に対する新たなフォークができなくなる可能性がある。その場合、フォークに失敗した PE において本来フォークできたはずのスレッドが逐次的に実行され性能低下を招く。

また $\frac{fd}{rd}, \frac{fd}{md}$ にも上限値を定める。フォーク距離と依存距離の差が大きすぎると、子スレッドが長時間ストールしてしまう。このような子スレッドをフォークするよりもストールしないような他の子スレッドをフォークする方が良いと考えられる。よって $\frac{fd}{rd}, \frac{fd}{md}$ がこの上限値を越える場合は利得値として0を返す。上記のいずれでもないときは利得値として

$$\min(fd, rd, md)$$

を返す。

3.4.4 距離計算

距離計算はプログラム上の2つの点 (i_1, i_2) を入力とし、距離値 $\text{dist}(i_1, i_2)$ を出力する作業である。

$\text{dist}(i_1, i_2)$ はプログラムを逐次実行した時に i_1 から i_2 に到達するまでの時間の見積りであるので、正確に求めるのは難しい。現段階の実装では単純に i_1 から i_2 までの各バスの平均命令数を距離値としている。より正確にするには、 i_1 から i_2 までのデータフローグラフを作りその高さを距離値とすることが考えられる。

各バスの命令数の平均をとるときはプロファイル情報によってどちらのバスがより頻繁に通るかという情報を得て、重み付きの平均をとる。

i_1 から i_2 に至るまでの間に関数呼び出し命令がある場合は、プロファイル情報を利用して関数が呼び出されてから戻ってくるまでに何命令実行したか、という情報を得て、その命令数を加算する。

$(x, y) \in E$ で $y \in \text{dom}(x)$ であるとき (x, y) は後向きの辺であるという。後向きの辺で構成されるループが自然なループ¹⁾である。 i_1 から i_2 に至るまでの自然なループがある場合、プロファイル情報によってループ回数を取得、ループ部分の命令数をループ回数倍する。

3.5 選択

選択は分割候補 F の中から必要な分割情報だけを選ぶ作業で、スレッド分割の最終段階である。

2.5節で述べたように、1つのスレッドからは1回のフォークしかできないという SKY のアーキテクチャ上の制限がある。

* $(\exists x \in S) x \notin E$ であると仮定する。 $(\forall s \in \text{succ}(x)) s \notin D$ でありかつ $x \notin E$ であるので $x \notin D$ 。しかしこれは $x \in S$ に矛盾する。したがって $(\forall x \in S) x \in E$ 。よって $S \subseteq E$ 。

この制限によって分割候補 F に含まれる2つの分割のうちのどちらか1つの分割しかできないとき、この2つの分割は互いに排他的であるという。あるいは、この2つの分割によるフォークが互いに排他的であるという。

3.2.3節の例では、(1,4)の制御等価な基本ブロックの組に着目して分割することにより1から4のフォークが作られる。同じようにして(1,7)からは1から7へのフォークが、(4,7)からは4から7へのフォークが作られる。

1から4へのフォークと1から7へのフォークは互いに排他的である。なぜなら、この2つのフォークをともに行なうとすると、親スレッドは2回のフォークをしなければならないからである。SKYアーキテクチャは親スレッドは1回のフォークしか許さないで、これは不可能である。4から7へのフォークと1から7へのフォークも互いに排他的である。したがって、この場合は(1,7)のフォークか、(1,4),(4,7)と2回連続のフォークか、どちらがよいかを判断し選択しなければならない。

この選択は排他的でない分割情報の組合せのうち、どれが最大の合計利得値を持つかということを決めることによって行う。排他的でない組合せが仮に n 個あるとすると、 $F_1, F_2, \dots, F_n$ ($F_i \subseteq F$ ($1 \leq i \leq n$)) となる。すべての F_i について、含まれる分割情報の利得値の合計を計算する。ただし単純な合計ではなく異なるパスにあるフォークはその利得値の平均(プロファイルを使った重みつき平均)をとって加算していくようにする。このようにして最大の合計利得値を持つ F_i が選ばれる。

3.6 命令挿入

選択が終わると分割が最終的に決定するので、決定した分割情報にしたがって命令を挿入していく。フォーク点には FORK 命令を挿入し、子の開始点の直前には FINISH 命令を挿入する。レジスタ通信集合の各要素 (r, a, b) の a の位置に SEND 命令を挿入し、レジスタを送信するようにする。

4. 評価

4.1 評価環境

SKYのシミュレータを作成し評価した。ベンチマーク・プログラムには、SPECint95の中から gcc, go, li, m88ksim, perl, vortex を使用した。

SKYのシミュレータはトレース駆動である。実行トレースは NEC EWS4800/360AD(MIPS R4400) 上で pixie⁴⁾ によって採取した。評価では100,000,000命令を使用した。スレッド分割は、同じく NEC EWS4800/360AD(MIPS R4400) 上でコンパイルしたベンチマーク・プログラムをディスアセンブルし、それを解析することにより行った。メモリ依存、プロファイル情報はトレースから取得した。

SKYを構成する1つのPEは、8命令フェッチ幅、32エントリのリザーベーション・ステーションを持つスーパースカラとした。8履歴のPAP⁷⁾予測器を持つ2048エントリのBTBを用いて分岐予測を行なう。命令キャッシュは2ポートで、全部で16命令のバンド幅を持つとした。命令キャッシュは、各PEに1つだけポートを割り当てる。即ち、同時に2つのPEに命令を供給できるとした。同期/通信能力を決定するPE間の通信バス幅は8オペランドとした。その他の資源、即ち、機能ユニット、命令キャッシュの容量、データ・キャッシュの容量とポート数等は、無限とした。SKYのPE数は8とした。比較のためSKYと同一の命令供給バンド幅(16命令)とリザーベーション・ステーションの総エントリ数が等しい(32×8=256エントリ)スーパースカラ(SS)のIPCも測定した。

4.2 評価結果

IPCを測定した結果を図6に示す。どのベンチマークでもSSと比べて性能が向上した。特に m88ksim においてはSSに比べて238%の性能向上を達成した。これは m88ksim の最も頻繁に実行されるループの中がうまくスレッド分割できたからである。gcc, li, vortex の向上率は16~20%で、go は35%向上した。性能向上が最も小さいのは perl でSSに比べて8%程度の向上にとどまった。perl はフォークできる機会が少なく、7個以上のPEが同時に動くことがなかった。SKYのハードウェア

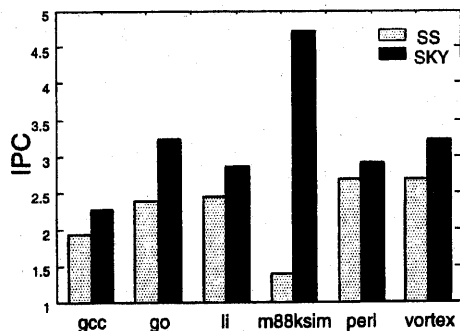


図6 結果

ア資源が活かしきれなかったといえる。

各ベンチマークのIPCの調和平均はSSが2.14、SKYが3.05であった。調和平均ではSKYはSSと比較して42%の性能向上があった。

5. まとめ

スレッド間命令レベル並列を利用するプロセッサ・アーキテクチャSKYのコンパイラのスレッド分割技法について述べた。この技法は制御等価な基本ブロックに着目し、依存解析の結果を利用してフォーク点、子の開始点、レジスタ通信集合を求める。そしてプロファイルを利用して利得計算を行い、最適なフォークをするようにする。この技法を用いてSPECint95のプログラム(gcc, go, li, m88ksim, perl, vortex)をスレッド分割し、SKYのシミュレータで実行して評価したところ各IPCの調和平均でスーパースカラに比べ42%の性能向上を達成した。

謝 辞

本研究の一部は、文部省科学研究費補助金基盤研究(B)「マルチスレッド型並列計算機方式の研究」の支援による。

参考文献

- 1) A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.
- 2) 小林良太郎, 岩田充晃, 安藤秀樹, 島田俊夫, “制御依存解析と複数命令流実行を導入した投機的実行機構の提案と予備的評価,” 情報処理学会研究報告 97-ARC-125, pp.133-138, 1997年8月.
- 3) M. S. Lam and R. P. Wilson, “Limits of Control Flow on Parallelism,” In *Proc. 19th Int. Symp. on Computer Architecture*, pp.46-57, June 1992.
- 4) M. D. Smith, “Tracing with Pixie,” Technical Report CSL-TR-91-497, Stanford University, November 1991.
- 5) M. D. Smith, M. A. Horowitz, and M. S. Lam, “Efficient Superscalar Performance Through Boosting,” In *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.248-259, October 1992.
- 6) D. W. Wall, “Limits of Instruction-Level Parallelism,” In *Proc. Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.176-188, April 1991.
- 7) T. Y. Yeh and Y. N. Patt, “A Comparison of Dynamic Branch Predictors that use Two Level of Branch History,” In *Proc. 20th Int. Symp. on Computer Architecture*, pp.257-266, May 1993.