計算機アーキテクチャ　125－23
（１９９７．８．２１）

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

小林　良太郎[†]　岩田　充晃[†]
安藤　秀樹[†]　島田　俊夫[†]

高性能なプロセッサの設計においては制御依存をいかに緩和するかということが重要である．近年，制御依存を緩和する技術として投機的実行が多くのプロセッサでよく用いられている．しかしより高い性能を達成するために制御依存解析，複数命令流実行の技術も不可欠となる．そこで本稿では制御依存性を緩和するこれら 3 つの技術（投機的実行，制御依存解析，複数命令流実行）を高効率に混成する方式を検討し，その方式の実現例を提案する．評価の結果，m88ksim（SPECint95）において 8 つのプロセッサを持つ場合で従来のスーパスカラの 2.3 倍の性能が得られることを確認した．

A Proposal and preliminary evaluation of a speculative execution mechanism with control dependence analysis and multiple instruction flow execution

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

It is important to relax control dependences in high performance processor designs. Recently speculative execution is often used in most processors to relax control dependences. Control dependence analysis and multiple instruction flow execution are also indispensable to significantly improve performance. In this paper, we propose an architecture that introduces these techniques effectively. Evaluation results show that our machine with eight processors achieves 2.3x speedup over superscalar machines for the m88ksim(SPECint95) benchmark.

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（プログラム並列性）を利用することで向上させることができる．プログラム並列性を制約する要素は，そのプログラム中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，この制御依存を解消する方法として，投機的実行が用いられている．投機的実行とは，分岐先が決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，投機的実行による並列性の向上には限界が見えてきた．Wall らの並列性の限界を調査した研究[8]によれば，投機的実行による並列度の限界は 4 ～ 6 である．現在，商用のスーパスカラ・プロセッサは約 2 の並列度を実現しているので，この限界に近づいているということができる．

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[3]を示した．その結果によれば，これらの導入によって並列度は 40 に達するとしている．このことは，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，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ることを意味している．

そこで本稿では，これら 3 つの技術を融合させたアーキテクチャ **SKY** を提案し，その予備的評価を行う．2 章では現在の投機的実行の問題点を述べ，3 章では SKY の概要について述べる．4 章でスレッド並列実行の例を示し，5 章で SKY のアーキテクチャ構成を述べる．6 章で評価を行なう．7 章で関連研究について述べ，8 章でまとめる．

2. 現在の投機的実行の問題点

制御依存の緩和に投機的実行のみを用いている現在のスーパスカラでは以下のような問題がある．

(1) 制御依存検出が単純であるために投機的実行が不効率
(2) 命令ウィンドウが単一であることによる並列性に関する制限

まず (1) に関して述べる．ハードウェアはフェッチした命令が分岐ならばそれ以降フェッチする命令はその分岐に制御依存していると見なす．これは単純でハードウェアに向いているが不正確である．例えば図 1 において基本ブロック 1（以下，基本ブロックを BB と略す）の分岐を左と予測し，BB2，BB4 の命令が投機的に実行されたとする．その後，BB1 からの分岐予測が誤っていたと分かった場合，ハードウェアは BB2 の命令だけでなく BB4 の命令まで無効化する．なぜなら BB4 も BB1 の分岐命令に制御依存していると見なすからである．しかし実際には BB4 は BB1 には制御依存していないので必ずしも無効化する必要はない．こうした理由により現在の投機的実行は不効率な部分があるといえる．

次に (2) に関して述べる．ハードウェアは動的な単一の制御流に従い逐次的に命令をフェッチしていく．従って，命令ウィンドウはある局所的な範囲でしか広がっていかない．このためこれを越えた部分に利用可能な並列性があったとしても抽出できない．図 2 を考えてみる．データ依存はないとする．ここで BB3 は他のどの部分にも制御依存の関係になく，BB2 のループとともに並列実行可能である．しかし単一プロセッサは

† 名古屋大学工学部
School of Engineering, Nagoya University

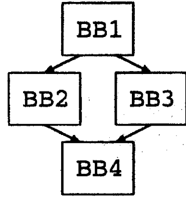

図1　制御フローグラフ1

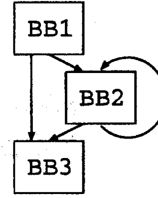

図2　制御フローグラフ2

ハードウェアの支援がない限り通常どの時点においても一度に一つの制御流しか追うことができない．つまり命令ウィンドウが単一であるためウインドウ内部はBB2の命令ばかりとなり，BB3が並列に実行できるにも関わらず，この並列性を抽出できない．

3. SKYの概要

まず設計の基本方針を示す．次に，コンパイラとハードウェアの特徴について述べる．

3.1 基本方針

我々は2章であげた問題をコンパイラとハードウェアを協力させることで解決する．すなわち，コンパイラは制御依存とデータ依存を正確に解析できる一方，ハードウェアは単純な制御依存検出に基づく投機的実行に優れているという点を利用する．具体的には，以下のように解決する．

- コンパイラは，制御依存とデータ依存解析を行い，互いに制御依存がなく並列実行によって性能向上が期待できる制御流（スレッド）の組を見つける．
- ハードウェアは，その情報をもとにそれらのスレッドを独立したプロセッサで同時に実行する．各プロセッサでは従来の投機的実行を行う．

この方式では，互いに制御依存のないブロック内の命令で並列実行可能なものは別のプロセッサで実行する．例えば，図1ではBB1，BB2，BB3をあるプロセッサで，BB4をそれとは別のプロセッサで並列に実行する．

命令ウィンドウは各プロセッサで独立である．このためBB1からの投機的実行が失敗しても，別の命令ウィンドウに入っているBB4の命令が無効化されることはない．従って，2章で述べたような投機的実行結果の不効率な無効化は生じない．

また，コンパイラは広い視野で制御依存のないブロックを発見でき，それらの間にデータ依存があるかどうかを検出できる．これにより，より広域的な並列性を利用することができる．従って図2のような場合でもBB2とBB3を並列に実行することが可能である．

以上のようなスレッドの並列実行により，スレッドにまたがる命令の並列性を利用し性能向上を図るには，異なるスレッドの命令間で生じるデータ依存による性能低下をできるだけ小さくする必要がある．具体的には，プロセッサ間の通信と同期によるオーバヘッドを最小にする必要がある．なぜならば，従来のマルチプロセッサで行われている粗粒度並列に比べて命令レベル並列では命令間の依存が非常に多く，通信と同期が頻繁に生じるからである．そこで我々は，レイテンシの極めて小さい(1クロックの）レジスタ・レベルでの通信と同期機構を実現することとした．

3.2 コンパイラの特徴

コンパイラは，スレッド分割とスレッド間通信に関する情報を作成し，ハードウェアに与える．

3.2.1 スレッド分割

スレッド分割に関する情報とは，現スレッドから新たなスレッド（以下，子スレッドと呼ぶ）を生成するフォーク点，子スレッドの開始点，および，現スレッドの終了点の組である．これらを求める方法を概説する．

まず最初に，現スレッドの開始点より到達するある基本ブロック BB_f に対し，制御依存のないブロック BB_c を見つける．より正確には，BB_f を後支配[5]するブロックを見つける．{ BB_f の先頭，BB_c の先頭 } を { フォーク点，子の開始点 } の候補とし，データ依存関係よりスレッド並列実行による性能利得を計算する．性能利得が十分ある場合，この候補を採用する．利得がなければ次の候補を探す．採用した { フォーク点，子の開始点 } の情報はプログラムに挿入する．具体的には，この情報をエンコードした専用の命令（FORK命令）を挿入する．

次に { フォーク点，子の開始点 } に対応するスレッドの終了点を求める．フォーク点より子の開始点に到達するパス上で，子の開始点の動的に直前の点がそのフォーク点に対応する終了点である．終了点も同様にこれを行う専用の命令（FINISH命令）を挿入する．

3.2.2 スレッド間通信

SKYは2つの通信機構を備える（詳細は後述）．1つは現スレッドの全レジスタ値を子スレッドのレジスタにコピーするレジスタ・ファイル・コピー機構である．もう1つは，値を命令（SEND命令）により個別に転送する機構である．フォーク点で既に定義されている値の子スレッドへの転送に関しては，レジスタ・ファイル・コピー機構を使用する．これはハードウェアがフォーク時に自動的に行う．従って，コンパイラは何も行わない．フォーク後に定義あるいは決定される値で，子の開始点でliveな値は，コンパイラはSEND命令を挿入して明示的に転送する．

受信するスレッド側では，送られてくる値を待ち合わせる必要がある．コンパイラは，子スレッドおいて待ち合わせるレジスタのリストを作成する．これを受信レジスタ・リスト（RRL : Receive Register List）と呼ぶ．RRLは，プログラム・テキストの中の固定の領域（**RRL セクション**と呼ぶ）に格納する．RRLセクションは複数のエントリからなり，各エントリに1つのスレッドに対するRRLを持たせる．ハードウェアはこれを用いて同期を実現する．

3.3 ハードウェア

図3にハードウェアの構成を示す．ハードウェアは，スーパスカラ方式の複数のプロセッサからなる．ただし命令キャッシュおよびデータ・キャッシュはハードウェア量が多いので，コストを著しく増加させないためにプロセッサ間で共有する．

単純なマルチプロセッサとSKYのハードウェアの大きな違いは，通信機構にある．レジスタ・レベルでの高速な通信機構を実現することにより，複数のスレッド同時実行において，命令レベルの並列性を利用することができる．実現した通信機構は，レジスタ・ファイル・コピーと命令による値の転送の2つである．以下説明を行う．

レジスタ・ファイル・コピー機構は，あるプロセッサの全てのレジスタを別のプロセッサのレジスタに1サイクルでコピーするものである．この機構はフォーク時に使用する．この機構を実現するためには，各プロセッサの対応するレジスタ間を結線する必要がある．従って，単純に実現しようとすると配線により非常に複雑になる．我々はこの問題を解決するために，集中レジスタ・ファイルと呼ぶ構造を提案する．集中レジスタ・ファイルは，各プロセッサのレジスタ・ファイルをビット・スライス（列）の単位で物理的に1箇所に集めた構造を持つ．

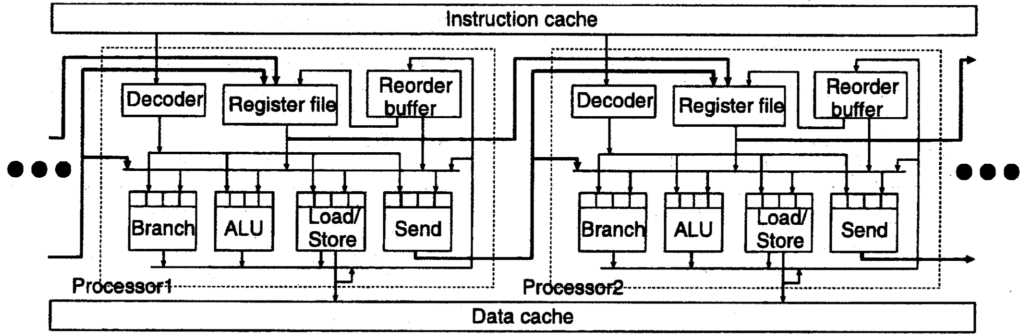

図3　SKY の構成

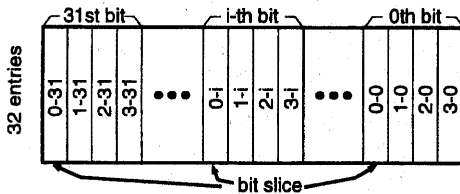

図4　集中レジスタ・ファイル

に対する 32 ビット 32 エントリの集中レジスタ・ファイルの構成を示す．集中レジスタ・ファイルの第 i 番目のビット・スライスは，プロセッサ 0 ～ 3 の各プロセッサの第 i 番目のビット・スライス（図 4 において，ビット・スライス $0-i$ ～ビット・スライス $3-i$）を集めたものである．この構造では，プロセッサ間のレジスタの転送は，ビット・スライス内の各プロセッサの対応する記憶セルを結線することで実現する．そのため配線を最小にすることができる☆．

以上のように集中レジスタ・ファイルはレジスタ間転送を 1 クロックで行なうことができるが，従来のレジスタ・ファイルに比べて読み出し時間が長くなる．理由は 2 つある．1 つはレジスタ・ファイル自体が大きくなるためワード線の駆動時間が長くなるためである．もう 1 つは，データパスから離れた位置に配置されることにより，読み出し値を転送する時間が長くなる．これら遅延時間の増加に対しては，レジスタ読み出しをパイプライン化することで対処する．パイプライン・レジスタの挿入位置としては，例えば，ワード線ドライバの直前やビット線センスアンプの直後などが考えられ，実現上の問題はない．パイプラインが深くなることで分岐予測ミスペナルティが大きくなるが，毎サイクル被るサイクル時間に対するペナルティの方が性能に与える影響より大きいと考えられるので，必要な措置である．

SKY では，レジスタ・ファイル・コピー以外に，前節で述べたように SEND 命令により他のプロセッサに値を転送する機構を持つ．SEND 命令はレジスタを読み出し，それを宛先のプロセッサのレジスタ・ファイルへ直接書き込むものである．フォーク以降に生成あるいは決定され，子の開始点で live な値を転送するために用いる．命令は専用の機能ユニット（図 3 の送信ユニット (Send)）を用いて実行する．

3.1 節で述べたように，通信ではレイテンシが極めて小さいことが要求される．つまり，SEND 命令の実行から，それにより転送される値をオペランドとする命令の実行開始までの時間を短くする必要がある．このために，転送値をレジスタを介さず，待ち合わせている命令にバイパスする機構を実現した．図 3 において，プロセッサ 1 の送信ユニットの出力を，プロセッサ 2 のリザベーション・ステーションへ転送するパスがあることに注意して欲しい．このバイパス機能により，SEND 命令が実行された次のサイクルに，転送値を使用する命令の実行を開始できる．5.3 節で詳細について述べる．

以上 2 つの通信機構は，プロセッサ間でレジスタ・レベルの通信を短いレイテンシで行うことができる可能性を持っている．しかし，依然として任意のプロセッサ間で通信を行おうとすると複雑化し，サイクル時間に悪い影響を与える．例えば，集中レジスタ・ファイルでは，ビット・スライス内の同一エントリの任意のセルを結線する必要がある．送信値のバイパスでは，全てのプロセッサに放送するためのバスが必要となる．

そこで，あるプロセッサから転送可能なプロセッサは 1 つに限定した．これにより集中レジスタ・ファイルのビット・スライス内の記憶セルは単純にリング状に結線すればよく，単純化できる．また，SEND 命令による転送値のバイパスは，隣接するプロセッサのリザベーション・ステーションに対してのみ行なえば良く，放送のためのバスは 1 つで良い．この制限によりあるスレッドから孫以降（2 世代以上先）のスレッドへの値の送信には，その間にあるスレッド（孫へ送信する場合子）を経由する必要がある．しかし，1 つの転送は 1 サイクルで行われるので，これによる性能低下は小さいと考える．

4. スレッド並列実行の例

SKY によるスレッド並列実行の例を示す．図 5 のプログラムで，考えられる並列実行のシナリオを 4 つあげて説明する．

シナリオ 1 :ブロック B からの分岐方向が予測困難な場合

B でフォークし，E または J を開始点とする子スレッドを生成する．即ち，B から始まるスレッドと E または J を開始点とするスレッドを同時実行する．従来の投機的実行では，B C E J と投機的実行を行った後，B の分岐予測が外れた場合，C 以降の実行が全て無効化される．しかし E を子の開始点とする場合，E 以降は無効化されない．J を開始点とした場合も同様に J 以降は無効化されない．E，J のどちら（あるいは図示していないが他の候補）を開始点とするか，あるいはどちらも子の開始点としないかは，データ依存解析に基づく性能利得計算によって決める．

シナリオ 2： F G H I よりなるループの繰り返し回数が多い場合

F でフォークし，I を開始点とする子スレッドを生成する．図 6 にスレッドの並列実行の様子を示す．ブロック名に付けた添字はイタレーションを表す．例えば，F_2 はブロック F の 2 回目のイタレーションでの実行を表す．まず，プロセッサ 0 で F_1 が実行され，プロセッサ 1 に I より始まるスレッドを生成する．これによりプロセッサ 0 では F_1 G_1 が，プロセッサ 1 では I_1 F_2 H_2 が実行される．プロセッサ 1 が F_2 を実行す

☆ 集中レジスタ・ファイルは，以上のように物理的構成を規定するものである．従って，論理的には図 3 に示すように，各プロセッサがレジスタ・ファイルを「分散」して所有する構成である．

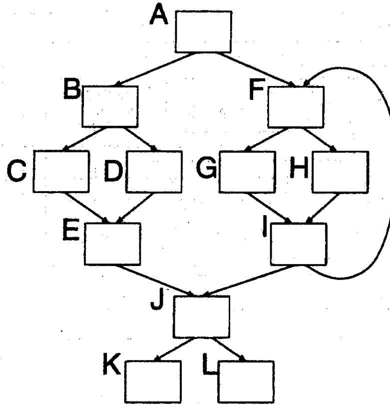

図5　プログラム例

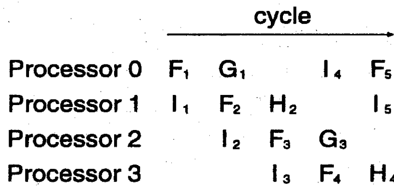

図6　ループの実行

る際，プロセッサ2に同様にIから始まるスレッドが生成される．以下同様にして，図6に示すように複数のイタレーションが同時に実行される．

この例では，次の2点によって実行が高速化されている．第1に，ブロックFからの分岐の予測精度に関係なく，複数のイタレーションの実行をオーバラップできる．従来の投機的実行では，例えば F_1 G_1 I_1 F_2 H_2 と実行された場合で F_1 からの分岐予測がはずれたら，G_1 以降の実行が無効化される．しかしSKYでは，I_1 F_2 H_2 の実行は F_1 の分岐によって無効化されることはない．

第2に，イタレーション間にデータ依存があった場合でも，プロセッサ間通信のレイテンシは1サイクルと非常に小さいので，スレッド間で命令レベルの並列性を利用することができる．即ち，各イタレーションの命令はイタレーション間にデータ依存があっても，そのデータ依存を満足する最も早いサイクルで実行を開始できる．これはちょうどソフトウェア・パイプライニングされたスケジューリングのようになり，ループを高速に処理できる．

シナリオ3：FGHIよりなるループの繰り返し回数が少ない場合

Fでフォークし，IまたはJを開始点とする子スレッドを生成する．これによる効果はシナリオ1の場合と同じである．

シナリオ4：シナリオ1～3のどの場合もスレッド並列実行による性能利得が小さい場合

あるブロックに対して後支配ブロックが存在し，複数のスレッドに分割可能としても，データ依存などでスレッド並列実行の性能利得が小さい場合がある．この場合，より大きな視野で性能利得の存在するスレッド分割を探索する．図5の例では，例えば，Aでフォークし，Jを開始点とするスレッドを生成する場合などがある．

これらのシナリオは，コンパイラがプロファイルをとるなどして，各基本ブロックの実行頻度や分岐方向などの情報を得ることにより予測する．それに応じて最も効果のあるスレッド分割方法を決定する．

5. アーキテクチャ

本章ではアーキテクチャの詳細について述べる．最初に，SKY専用命令について説明する．その後，スレッドの生成，終了，プロセッサ間通信の機構について順に述べる．

5.1 専用命令

SKYにはスレッドを生成するFORK，スレッドを終了させるFINISH，レジスタ値を子スレッドに送信するSENDの3つの専用命令がある．形式はそれぞれ，

```
FORK   tid,offset
FINISH tid
SEND   tid,reg
```

である．

各スレッドはスレッド識別子（TID：Thread ID）によって識別される．TIDはFORK命令実行によってスレッドが生成された場合，命令で指定された tid によって与えられる．ハードウェアには，現スレッドのTIDを保持する **TID レジスタ**と，FORK命令実行によって生成した子スレッドのTIDを保持する **CTID**（Child Thread ID）**レジスタ**を持つ．3.2節で述べたように，スレッド分割とスレッド間通信に関する情報はフォーク点に対応して存在する．この対応は命令コードの tid で認識する．

FORK命令は以下の操作を行う．

- 現プロセッサ（現スレッドを実行しているプロセッサ）のレジスタ・ファイルを子プロセッサ（子スレッドを実行するプロセッサ）のレジスタ・ファイルにコピーする．
- このFORK命令に対応するRRLを読み出し，受信レジスタ待ち合わせのための表である受信状態表（RST：Receive Status Table）（5.3節で述べる）をセットアップする．
- 子プロセッサのTIDレジスタと現プロセッサのCTIDに tid を書き込む．
- 子プロセッサのPCに，FORK命令のアドレスと offset を加えた値を書き込み，子スレッドの実行を開始する．

FINISH命令は，指定された tid が現プロセッサのTIDに等しい場合，現スレッドの実行を停止し，プロセッサを開放する．等しくなければ実行されない．

SEND命令は，指定された tid がCTIDと等しい場合，現スレッドのレジスタ reg の値を子プロセッサのレジスタ reg に書き込む．等しくなければ実行されない．

5.2 スレッド生成

スレッドの生成を行うFORK命令実行の詳細について説明する．FORK命令は操作内容が従来の逐次マシンに存在する命令とは異なり複雑である．特に，FORK命令の行うべき操作を投機的にどこまで行うかが問題となる．最も積極的な考え方は，FORK命令の全ての操作を投機的に行うことである．この方法はスレッドの実行を投機的に行うことにつながるが，複数のスレッドが投機的に生成されたとき，それらのコミットと無効化の論理が複雑になる．逆に，最も消極的な考え方は，FORK命令の全ての操作を投機的に行わないことである．この方法は，スレッド生成の時間，とくにRRLをメモリより読み出す時間が露出され，性能低下につながる．

そこで我々は，FORK命令による操作の一部，具体的には，RRLのロードと子の開始点の計算のみを投機的に行い，その他の操作は，このFORK命令が実際にコミットされたときに行うこととした．この方法では，時間のかかる操作を投機的に行うことにより隠蔽することができ，FORK命令の実行を高速化できる．RRLのロードと子の開始点の計算を隠蔽することができれば，FORK命令がコミットされたとき，子スレッドの生成を1サイクルで行うことができる．

FORK命令の投機的実行を支援するために，FORK命令実行の途中結果を格納するバッファを用意する．このバッファを，フォーク情報バッファ（FIB：Fork Information Buffer）と呼ぶ．FIBは複数のエントリよりなり，各エントリは，子のTID（CTID）と子の開始アドレス（CSA：Child Start Address）

および RRL からなる．

FORK 命令はデコード時に FIB のエントリを割り当てられる．FORK 命令は割り当てられた FIB エントリ番号を付加され，ロード・ユニットに発行される．ロード・ユニットでは，子の開始アドレスの計算と RRL のロードを行う．子の開始アドレスは，命令アドレスと FORK 命令で指定された `offset` を加算することにより得られる．

RRL は，プログラム・テキストの RRL セクションのスレッドに対応するエントリに格納されている．アドレスは，具体的には，RRL セクション先頭アドレスに（FORK 命令で指定される `tid` × sizeof(RRL)）を加算して得られる．

実行が終了したら，通常の命令と同じく割り当てられたリオーダ・バッファ（ROB : Reorder Buffer）のエントリに実行終了のフラグをセットする．同時に，子の開始アドレス，RRL，命令で指定される `tid` を割り当てられた FIB エントリに書き込む．

投機が成功し FORK 命令が ROB の先頭に達したら，割り当てられた FIB のエントリを読み出すとともに，以下の 2 つの条件を検査する．

(1) 子スレッドを割り当てる（隣の）プロセッサが空いているか?
(2) レジスタ・ファイル・コピーによって子プロセッサに送信するレジスタ値が全て受信されているか？

条件 (2) は，現スレッドの親から 2 世代以上先のスレッドへ値を送信する時に必要である．なぜならば，3.3 節の最後に述べたように，通信は各世代を経由して行われるからである．

上記 2 つの条件を満足してる場合，FORK 命令の残りの操作を行いスレッドを生成する．つまり，

- レジスタ・ファイルをコピーする．
- FIB より読み出した RRL を用いて，RST をセットアップする．
- FIB より読み出した CTID を子プロセッサの TID レジスタと現プロセッサの CTID レジスタに書き込む．
- FIB より読み出した CSA を子プロセッサの PC に書き込む．

上記フォーク条件を満足しなかった場合，このフォーク命令を無効化する．

5.3 スレッド終了

スレッドの終了を行う FINISH 命令は FORK 命令と 1 対 1 で対応している．従って，命令デコード時に，命令によって指定された `tid` と現スレッドの TID と比較し，一致していればスレッド終了の操作を開始する．一致していなければ，無効化する．スレッド終了の操作は，具体的には，まず命令フェッチを即座に停止し，FINISH 命令以降の命令を無効化する．その後，FINISH 命令に割り当てられたエントリが ROB の先頭に来るのを待つ．先頭に来たら，FINISH 命令に先行する命令の実行は完了しているので，現プロセッサを開放する．

5.4 SEND 命令による通信

SEND 命令の実行には，送信と受信の機構が必要である．以下それぞれについて説明する．

5.4.1 送　　信

SEND 命令は，指定された子スレッドへレジスタ値を送信する命令である．従って，生成された子スレッドに対応する SEND 命令だけを実行するように制御しなければならない．

SEND 命令デコード時点で，FORK 命令がすでにコミットされており CTID が定義されているれば，SEND 命令が指定する `tid` と CTID を比較する．異なればこの時点で無効化する．等しければ，SEND 命令をリザベーション・ステーションに発行する．通常の命令と違い SEND 命令は投機的に実行しない．そのため，リザベーション・ステーションでは，通常の命令の場合と同じくオペランドを待つが，通常の命令と異なり，制御依存が解消，即ち，依存する分岐方向が全て定まり正しく予測されたことが分かった後に実行する．具体的には，子プロセッサへ値とレジスタ番号を送信する．

デコード時点で CTID が定義されていない場合も，リザベーション・ステーションへ発行する．ただしそこで CTID が定義されるのを待つ．このため，送信ユニットのリザベーション・ステーションのエントリには，通常のフィールドの他に，SEND 命令が指定する `tid` を保持するフィールドを持つ．FORK 命令がコミットされ CTID が定義されたならば，それを送信ユニットのリザベーション・ステーションの全エントリに送る．各エントリでは，エントリ内の `tid` と送られて来た CTID を比較し，一致しないエントリを無効化する．一致したエントリは，後にオペランドが送られて来たらそれを受け取り，制御依存が解消すれば送信を行う．

5.4.2 受　　信

子プロセッサでの受信について説明する．受信を制御するため，受信状態表（RST : Receive Status Table）をハードウェアとして設ける．RST は，レジスタの数（整数レジスタ $32+$ 浮動小数点レジスタ $32=64$）のエントリを持つ表である．各エントリはレジスタに対応させ，タグと Wait，WEN（Write Enable）の 2 つのフラグと持つ．タグは子プロセッサにおいてオペランドを識別するためのものであり，スーパスカラにおけるオペランド・タグと同じである．Wait は，当該エントリに対応するレジスタの値の受信を待っていることを示す．このフラグは，スレッド間の命令の真のデータ依存を満足させるためのものである．WEN は，値を受信したときに，それをレジスタに書き込んで良いことを示す．これは，スレッド間の命令の出力依存を満足させるためのものである．以下詳細に説明する．

RST はスレッド生成時にセットアップする．具体的には，親プロセッサでの FORK 命令実行により得られる RRL から，待ち合わせるべきレジスタに対応するエントリの Wait，WEN の両フラグをセットし，タグを割り当てる．この他のエントリのフラグは全て，スレッド終了の際にリセットされているとする．

親プロセッサでの SEND 命令実行により，レジスタ値とレジスタ番号が送られて来た場合，現プロセッサは，そのレジスタ番号で RST を参照する．WEN がセットされていれば，指定されたレジスタに値を書き込み，受信が完了したことを示すため Wait をリセットする．また，受信値をオペランドとして待ち合わせている命令に値をバイパスするために，RST より読み出されたタグを値につけて，全リザベーション・ステーションに放送する．

親プロセッサから送信されて来る値は，レジスタ・ファイルに書き込みを行って良いとは限らない．なぜなら，この値は，現スレッドの開始点で live な変数の値であるというだけであるからである．即ち，現スレッドあるいはそれ以降のスレッドで参照される可能性があることを示しているにすぎない．実際に実行されるパスによっては，この変数は再定義され dead になる可能性がある．この場合，出力依存が生じる．これをフラグ WEN によって満足させる．現プロセッサの命令はコミットされた際（実現上，ROB からレジスタ・ファイルに書き込みを行う際），書き込みレジスタに対応する RST のエントリの WEN をリセットする．こうすることで，親プロセッサから送られて来た値が live な値を持つレジスタを上書きし出力依存に違反するということが起こらないようにする．

次に，受信値のバイパスに関して説明する．一般にスーパスカラ・マシンでは，命令の真のデータ依存関係をタグによって識別する．他のプロセッサから送信されて来る値を参照する命令では，プロセッサ内で定義される値へのデータ依存を識別するタグの他に，受信値へのデータ依存を識別するためにタグを付ける必要がある．このために，命令デコードの際，通常のスーパスカラ・マシンと同様にレジスタ・ファイルと ROB を参照する他，RST を参照する．以下の 3 つの場合が存在する．

(1) 参照した RST のエントリの Wait フラグがセットされていない場合

受信値に依存していないか，あるいは，既に受信が完了しているかのどちらかである．この場合，依存関係は現プロセッサの中だけに存在するので，通常のスーパスカラ・マシンと同様に，デコード命令はレジスタ・ファイルあるいは ROB から読み出された値をオペランドとして取るか，あるいは，そのオペランドに割り付けられたタグを取る．

(2) 参照した RST のエントリの Wait フラグがセットされており，かつ，ROB より値もタグも読み出されなかった場合

デコード命令のオペランドは受信する値である．この命令は，RST を参照して得られるタグを取り，リザベーション・ステーションで値が送信されて来るのを待つ．

(3) 参照した RST のエントリの Wait フラグがセットされているが，同時に ROB より値またはタグが読み出された場合

デコード命令のオペランドは現プロセッサにおいて定義された（未だ投機的な場合，定義される予定の）値である．即ち，受信値は dead であり，この命令のオペランドは，同一レジスタを再定義する命令の結果である．従って，ROB より読み出された値またはタグを取る．

なお，状態 (3) において親プロセッサより値を受信した場合，それはレジスタに書き込まれる．これは必要である．なぜならば，同一レジスタを再定義する命令の実行が投機の場合があり，受信値が dead になるのは投機に成功したときに限られるからである．つまり，投機に失敗したとき受信値は参照される可能性がある．

6. 評価

6.1 評価環境

トレース駆動型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 m88ksim を使用した．実行トレースは NEC EWS4800/360AD（MIPS R4400）上で Pixie[4] によって採取し，評価では 4 百万命令を使用した．

SKY の構成する 1 つのプロセッサは，8 命令を同時にフェッチ，32 エントリのリザベーション・ステーションを持つスーパスカラとした．8 履歴の PAp[9] 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，16 命令のバンド幅を持つとした．即ち，同時に 2 つのプロセッサに命令を供給できるとした．この他の資源，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，SKY アーキテクチャの持つ制御依存を緩和する能力に注目するために無限とした．

6.2 評価結果

プロセッサ数を 2，4，8 と変化させた時の IPC を測定した．比較のため，同一命令供給バンド幅（16 命令）で，SKY とリザベーション・ステーションの総エントリ数が等しい単一のスーパスカラの IPC も測定した．リザベーション・ステーションのエントリ数を同じにすることで，SKY と単一スーパスカラのスケジューリングにおける命令ウィンドウの大きさは等しい．

図 7 に測定結果を示す．縦軸は IPC，横軸はリザベーション・ステーションの総エントリ数である．単一のスーパスカラでは，リザベーション・ステーションのエントリ数を増加しても IPC はほとんど改善しない．例えば，32 エントリから 64 エントリに増やすことによる改善は 5 % しかない．これに対して，SKY では，プロセッサ数を増加させれば IPC は大幅に向上する．8 プロセッサを持つ SKY は，リザベーション・ステーションの総エントリ数が等しい単一のスーパスカラに対して 2.3 倍も高い性能を達成できることが分かった．

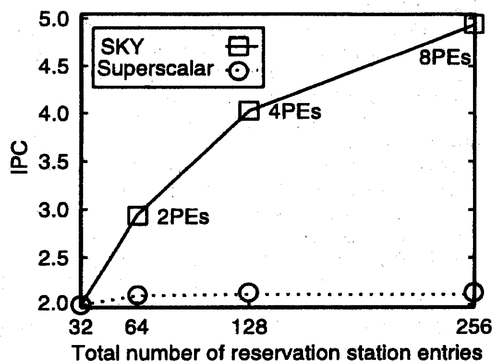

図 7　評価結果

7. 関連研究

単一プログラムの BB 間において制御依存がない，あるいは弱い部分でスレッドを生成する機構として，Multiscalar[6]，MUSCAT[7]，SPSM[1] をあげることができる．これらと我々の提案する SKY の最も大きな違いは，SKY がスレッド間の命令レベル並列を抽出するために，レジスタ・レベルでの高速な通信機構を実現したことにある．

8. まとめ

今回我々は制御依存を大幅に緩和するために，制御依存解析，複数命令流実行，そして投機的実行を融合したアーキテクチャ SKY を提案した．SKY は互いに制御依存がないスレッドの組を，高速な通信機構によって連結した複数のプロセッサで同時に実行する．評価の結果，m88ksim(SPECint95) において，プロセッサ数を 8 とした時の IPC は 4.93 となり，スーパスカラの 2.3 倍の性能向上を確認した．この結果は SKY がより多くのプログラム並列性を引き出したことを示している．

参考文献

1) P. K. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton, "Single-Program Speculative Multithreading(SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading," In *Proc. 1st Int. Conf. on Parallel Architectures and Compilation Techniques*, pp.109-121, September 1995.

2) M. Johnson, *SuperscalarMicroprocessorDesign*. Prentice Hall, Englewood Cliffs, NJ, 1991.

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. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.248-259, October 1992.

6) G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar Processor," In *Proc. 22nd Int. Symp. on Computer Architecture*, pp.414-425, June 1995.

7) 鳥居 淳，近藤 真己，木村 真人，西 直樹，小長谷 明彦，"On Chip Multiprocessor 指向 制御並列アーキテクチャMUSCAT の提案"，1997 年並列処理シンポジウム *JSPP'97*, pp.229-236, 1997 年 5 月．

8) D. W. Wall, "Limits of Instruciton-Level Parallelism, " In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.176-188, April 1991.

9) 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.