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

小林 良太郎[†] 小川 行宏[†,☆] 岩田 充晃[†,☆☆]
安藤 秀樹[†] 島田 俊夫[†]

近年のマイクロプロセッサは，スーパスカラ・アーキテクチャにより，より多くの命令レベル並列（ILP: Instruction-Level Parallelism）をプログラムより引き出し高性能化を図ってきた．しかし，この方法は，スーパスカラ・プロセッサが引き出すことのできる命令レベル並列の限界や，ハードウェアの複雑さの増加により，限界が見え始めてきた．これを解決する1つの方法は，ILPに加えスレッド・レベル並列（TLP: Thread-Level Parallelism）を利用することである．本論文では，レジスタ値の同期/通信機能を備え，複数のスレッドを並列に実行するSKYと呼ぶマルチプロセッサ・アーキテクチャを提案する．SKYは，非数値計算応用で高い性能を達成することを目的としている．このためには，細粒度のTLPを低オーバヘッドで利用することが要求され，SKYでは，命令ウィンドウ・ベースの同期/通信機構と呼ぶ機構を新たに導入した．この機構は，従来のレジスタ・ベースの同期/通信機構と異なり，受信待ちの命令に後続する命令の実行を可能にするノンブロッキング同期を実現している．これにより，TLPとILPを同時に最大限利用することを可能とする．SPECint95を用いた評価により，8命令発行の2つのスーパスカラ・プロセッサにより構成したSKYは，16命令発行のスーパスカラ・プロセッサに対して，最大46.1%，平均21.8%の高い性能を達成できることを確認した．

A Multiprocessor Architecture SKY that Exploits Thread-Level Parallelism in Non-Numerical Applications

Ryotaro Kobayashi,[†] Yukihiro Ogawa,[†,☆] Mitsuaki Iwata,[†,☆☆]
Hideki Ando[†] and Toshio Shimada[†]

Current microprocessors have improved performance by exploiting more amount of instruction-level parallelism (ILP) from a program through superscalar architectures. This approach, however, is reaching its limit because of the limited ILP available to superscalar processors and the growth of their hardware complexity. Another approach that solves those problems is to exploit thread-level parallelism (TLP) in addition to ILP. This paper proposes a multiprocessor architecture, called SKY, which executes multiple threads in parallel with a register-value communication and synchronization mechanism. The objective of SKY is to achieve high performance in non-numerical applications. For this purpose, it is required to exploit fine-grain TLP with low overhead. To meet this requirement, SKY introduces an instruction-window-based communication and synchronization mechanism. This mechanism allows subsequent instructions to waiting instructions for receiving registers to be executed unlike previously proposed register-based mechanisms. This capability enables fully exploiting both TLP and ILP. The evaluation results show that SKY with two eight-issue superscalar processors achieves a speedup of up to 46.1% or an average of 21.8% over a 16-issue superscalar processor.

1. はじめに

近年のマイクロプロセッサは，スーパスカラ・プロセッサが主流となっている．スーパスカラ・プロセッサは，複数の命令を同時に発行し，プログラム中に存在する**命令レベルの並列性**（ILP: Instruction-Level Parallelism）を利用する．これまでスーパスカラ・プ

† 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University
☆ 現在，岐阜県製品技術研究所
Presently with Gifu Prefectural Research Institute of Industrial Products
☆☆ 現在，三菱重工業株式会社
Presently with Mitsubishi Heavy Industries Ltd.

ロセッサは，命令の発行幅を広げ，より多くの並列性を利用することで，性能向上を図ってきた．しかし今後さらに命令発行幅を広げても，それによって得られる性能向上は小さいと考えられる．この理由は以下の2点である．

- 命令発行幅を広げるとハードウェアの複雑度が増加し，サイクル時間に大きな悪影響を与える[4),15)]．
- 現在のスーパスカラ・プロセッサが引き出している ILP は，単一制御流において利用可能な ILP の限界に近づきつつある[11),24)]．

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[7),14),19)~23)]．その背景には，上記スーパスカラ・プロセッサに対する悲観的観測に加え，半導体集積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現実的になりつつあることがある．単一チップ・マルチプロセッサの長所として，以下の3点をあげることができる．

- ILP に加えて，**スレッド・レベルの並列性**（TLP: Thread-Level Parallelism）を利用することができる．
- 命令幅を広げるなどプロセッサの複雑度を増加させる必要がないので，サイクル時間への悪影響を抑えることができる．
- チップ上に相互結合網を構成することで，従来のマルチプロセッサに比べ通信レイテンシを減少させることができる．

これらにより，単一チップ・マルチプロセッサはスーパスカラ・プロセッサを超える性能を達成できる可能性がある．

これまでマルチプロセッサの応用として，多くの場合，数値計算プログラムが想定されてきた．この理由の1つは，複数のプロセッサを単一チップに集積することがこれまで不可能であったことから，マルチプロセッサが非常に多くの計算量を必要とするが，コスト対性能比に寛容な分野に適するという点である．これは一般に，数値計算の分野である．もう1つの理由は，数値計算プログラムは一般に，豊富な並列性を内在しており，マルチプロセッサを比較的容易に利用することができる点である．

これに対して，マイクロプロセッサの多くの応用は，非数値計算プログラムである．我々は，その非数値計算プログラムにおいて，従来のスーパスカラ・プロセッサでは達成することができない高い性能を発揮するマルチプロセッサの実現に向けて研究を行っている．非数値計算プログラムでは，数値計算プログラムと異なり，不規則な制御構造と複雑なデータ依存関係が数多く存在するため，粒度の大きな TLP を利用することは難しく，粒度の小さな TLP を利用することが不可欠である．細粒度の TLP では，スレッドあたりの計算量が少ないので，スレッドを並列に実行するためのオーバヘッドを小さくすることが強く求められる．そのオーバヘッドのなかでも同期/通信のオーバヘッドの削減は，特に重要である．

一般にマルチプロセッサでは，同期/通信はメモリを介して行う．これを**メモリ・ベースの同期/通信**と呼ぶこととする．これに対して，単一チップ・マルチプロセッサでは，集積化という利点を生かし，レジスタ値を直接通信し，同期を行う機構が提案されている[2),7),19)]．これを**レジスタ・ベースの同期/通信**と呼ぶこととする．この機構はメモリ・アクセス回数を削減することができるので，メモリ・ベースの同期/通信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[7)]．

これまで提案されたレジスタ・ベースの同期/通信機構は，低オーバヘッドを実現しているものの，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．このことを我々は，**同期による命令実行のブロッキング**と呼ぶ．後続命令の中には，受信待ちの命令とは独立の命令がある可能性がある．このような命令は，受信待ち命令とは無関係に実行可能である．したがって，ブロッキングは非効率である．ブロッキングが生じると，スレッド内の ILP の利用が妨げられ，性能が低下する．非数値計算プログラムにおいては，スレッド内の ILP は並列性の重要な源なので，命令実行のブロッキングは性能に対して大きな影響を与えると考えられる．

本論文では，スレッド内の ILP 利用を阻害することなく細粒度の TLP を利用することを目的とした **SKY**[8)~10)]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する．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，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**と呼ぶ新しい同期機構を提案し，これを導入している．この機構は，命令ウィンドウ上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，同期による命令実行のブロッキングが生じることはなく，プロセッサは ILP を最大限利用することができる．

以下，2章ではこの研究の背景について述べる．3章でSKYアーキテクチャの提案を行う．4章で性能を評価する．5章で関連研究について述べる．最後に6章で本論文をまとめる．

2. 背　　景

プログラムに内在する並列性を調べた研究が数多く存在する（たとえば，文献3)，11)，24)）．その中でも，Lamらの研究[11]は複数のスレッド実行から得られる並列性について重要なことを示唆している．この研究によれば，並列性が低いと考えられている非数値計算プログラムにおいても，基本ブロック・レベルで制御依存関係を詳細に解析し，複数の制御流の命令を同時に実行すれば，単一制御流の命令だけを実行する場合に比べて，多くの並列性を引き出すことができるとしている．これは，スレッド内のILPとともに細粒度のTLPを利用すれば，そのようなマシンの性能はスーパスカラ・プロセッサの限界を大幅に超える可能性があることを示唆している．

TLPを利用する一般的な方法は，マルチプロセッサによるスレッド並列実行である．単一チップ・マルチプロセッサとスーパスカラ・プロセッサの性能を比較した研究として，Olukotunらの研究がある[14]．この研究では，命令発行幅が狭く小規模なスーパスカラ・プロセッサを複数結合したマルチプロセッサと，それとほぼ同一のチップ面積で，命令発行幅が広く規模の大きな単一のスーパスカラ・プロセッサの性能を比較している．マルチプロセッサの同期/通信機構は，従来のメモリ・ベースのものである．この研究によれば，彼らが評価に用いたプログラムの中で，並列化困難な非数値計算プログラムにおいては，単一のスーパスカラ・プロセッサに比べて単一チップ・マルチプロセッサは，10〜30%性能が低いという測定結果を得ている．しかし彼らは，マルチプロセッサは単純なプロセッサにより構成されているので，高いクロック速度での動作が期待でき，その結果，この程度の性能差はなくなると推測している．

単一チップ集積の利点を生かし，レジスタ・ベースの同期/通信機構を提案した研究として，**マルチスカラ・プロセッサ**[2),19)]と**マルチALUプロセッサ**（MAP）[7)]の研究がある．これらで提案された機構は，プロセッサ間でレジスタ値を直接通信し，full/emptyビット[16)]を用いてレジスタに関する同期をとるものである☆．full/emptyビットはレジスタごとに存在し，対応するレジスタに格納された値が利用可能かどうかを示す．Kecklerらは，共有のオンチップ・キャッシュを持つ3プロセッサ構成のMAPアーキテクチャにおいて，レジスタ・ベースの同期/通信機構を用いれば，メモリ・ベースの機構を用いる場合に比べて，オーバヘッドを大幅に小さくできることを示した．

これらの研究において提案されたレジスタ・ベースの同期/通信機構は，オーバヘッドを小さくできるという長所を持つが，1章で述べたように，同期による命令実行のブロッキングという欠点を持つ．つまり，デコードした命令のソース・オペランドに対応するfull/emptyビットがemptyであった場合，その命令はfull/emptyビットがfullになるまで後続の命令のパイプライン処理を停止させてしまうという欠点を持っている．

☆ マルチスカラ・プロセッサでは，accumマスクとrecvマスクという2種類のビットで実現されているが，機能はfull/emptyビットと同じである．

3. SKYアーキテクチャ

本章では，SKYアーキテクチャについて述べる．最初に，マルチスレッド・モデルについて述べる．次に，ハードウェアの構成について述べる．その後，SKYの特徴である同期/通信機構について説明する．さらに，SKY用のコンパイラについて述べ，最後にSKY専用命令の詳細について述べる．

3.1 マルチスレッド・モデル

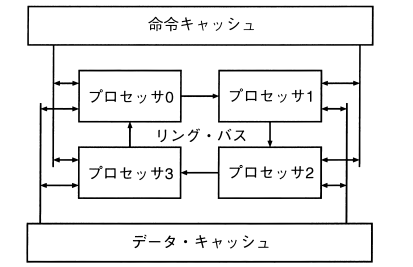
スレッド並列実行のためのオーバヘッドを小さくするため，SKYにおけるマルチスレッド・モデルは通常のモデルと比べて，次に示す制約を課している．これは，同じく非数値計算プログラムのTLPを利用する目的を持つアーキテクチャである，マルチスカラや**MUSCAT**[20),21)]と同様のモデルである．

- 1つのスレッドは，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，逐次実行の順における自身の直後のスレッドを逐次生成する．

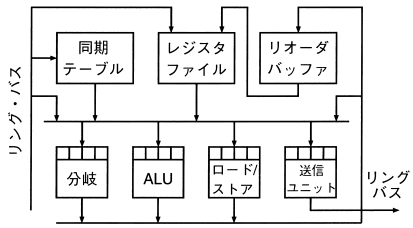
3.2 ハードウェア構成

図1 (a) に示すように，SKYは複数のスーパスカラ・プロセッサからなる．プロセッサは単方向のリング・バスで結合する．同図に示すように，プロセッサiはプロセッサ$(i+1)\%n$にのみデータを送り，プロセッサ$(i-1)\%n$からのみデータを受け取る（nはプロセッサ台数で，%はモジュロ演算子を示す）．以下，プロセッサ$(i+1)\%n$をプロセッサiの**後続プロセッサ**と呼び，プロセッサ$(i-1)\%n$をプロセッサiの**先行プロセッサ**と呼ぶ．

スレッドは実行を開始してから終了するまで，1つ



(a) 構成



(b) プロセッサ

図 1 SKY のアーキテクチャ
Fig. 1 SKY architecture.

のプロセッサを占有する．あるプロセッサ上のスレッドは `fork` 命令を実行することによって，後続プロセッサに子スレッドを生成する．ただし，後続プロセッサが他のスレッドによって占有されているなら，`fork` 命令は無効化され，子スレッドは生成されない．3.1 節で示した SKY のスレッドの性質より，`fork` 命令を無効化しても，スレッドに分割され実行されるべきところが逐次に実行されるだけで，プログラムの意味は保持される．

キャッシュはプロセッサ間で共有する．SKY は少ない数のプロセッサで構成することを想定しているため，キャッシュのポート数は実現上容認できると考える．

図 1 (b) にプロセッサ内部の構成を示す．通常のスーパスカラ・プロセッサと異なる点は，レジスタの同期/通信を支援する機構を持つ点である．レジスタの送信は，`send` と呼ぶ専用の命令を送信ユニットで実行することにより行う．`send` 命令は，コンパイラによって挿入される．これについては 3.4 節で述べる．

一方，受信はハードウェアによって暗黙的に行う．このハードウェアの中心をなすものが，**同期テーブル**と呼ぶテーブルである．このテーブルの各エントリはレジスタに対応し，当該プロセッサがそのレジスタの値を受信すべきかどうかについての情報と，受信値を識別するタグを保持する．命令はレジスタ・フェッチの際，同期テーブルを参照し，参照オペランドがこれから受信する値かどうかを判断する．受信値がオペランドの場合，同期テーブルからその受信値に対応するタグを読み出し，命令ウィンドウで受信を待ち合わせる．一方，先行プロセッサからレジスタ値が送られてくると，プロセッサはそれをレジスタ・ファイルに書き込むとともに，同期テーブルより対応するタグを読み出し，命令ウィンドウに放送する．命令ウィンドウで受信を待ち合わせている命令は，放送されたタグと自身の持つタグが一致する場合，その受信値を受け取り，同期が完了する．

上記は，最も典型的な受信の動作を示したものであるが，実際には種々の場合に対応し動作が異なる．次節では，この説明を含め，同期/通信の機構の詳細について述べる．

3.3 同期/通信の機構

SKY の同期/通信機構は，以下の 2 つの特徴を持っている．

- 同期/通信のオーバヘッドを 2 サイクルにまで減少させている．
- ノンブロッキングの同期を実現している．

これらの特徴は，従来の単一チップ・マルチプロセッサの研究で提案あるいは論じられている同期/通信の高速化，すなわち，

- 単一チップ化による物理的な高速化[12)]
- メモリを介さずレジスタ値を直接送受信することによるアーキテクチャ上の高速化[7),19)]

といった事項に加え，新しく命令ウィンドウ上でレジスタに対する同期を行う機構を導入したことによるものである．この機構を我々は，**命令ウィンドウ・ベースの同期機構**と呼ぶ．

以下，命令ウィンドウ・ベースの同期機構について説明する．3.3.1 項では，まず，本機構の説明を容易にするために，投機的実行を行わないプロセッサにおいて，本機構の基本的動作を説明する．次に 3.3.2 項において，投機的実行を行うプロセッサに対応できるように，機構を修正する．

3.3.1 非投機的実行プロセッサにおける実現

最初に機構について説明し，その後，簡単な例を用いて動作を説明する．

[機構]

同期テーブルについて説明する．同期テーブルの各エントリはレジスタに対応し，受信値を識別するタグと **R フラグ**（receive フラグの略）と呼ぶフラグを保持する．R フラグは，当該エントリに対応するレジスタ値を受信するかどうかを示す．R フラグが 1 ならば受信し，0 ならば受信しない．

先行プロセッサよりレジスタ値が送られてきた場合，そのレジスタに対応する同期テーブルのエントリを読

み出す．読み出されたエントリの R フラグが 1 ならば，当該プロセッサはそのレジスタ値を受信する．受信においては，送られてきた値をレジスタ・ファイルに書き込むとともに，同期テーブルより読み出されたタグを値に付けて命令ウィンドウの全エントリに放送する．これにより，命令ウィンドウで受信値を待ち合わせている命令に対し，レジスタ・ファイルを介さず値を渡すことができる．

読み出されたエントリの R フラグが 0 ならば，受信動作は行わず，送信されてきた値は捨てられる．ここで，R フラグが 0 であっても，先行プロセッサから値が送信されてくる場合があることを注意しておく．前述のようにレジスタの送信はコンパイラがプログラム中に send 命令を挿入することによって実現しているが，静的解析の性質上コンパイラは，スレッド間の可能性のあるすべてのデータ依存について send 命令を挿入するからである．実行時には，静的に存在するデータ依存の一部がオペランドの定義と参照の関係として現れるので，受信する必要のないレジスタの送信が行われることがある．

同期テーブルの R フラグは，スレッドがプロセッサに生成されたときに，すべて 1 にセットする．あるエントリの R フラグは，そのエントリに対応するレジスタの値を受信するか，あるいは，スレッド内で値を定義する命令がデコード・ステージにきたら 0 にリセットされる．R フラグを参照することによって，命令が参照するオペランドについて，次の 3 つの場合を判断することができる．

(1) これから受信する値
(2) すでに受信された値
(3) 自身のプロセッサにおいて定義された値

(2) または (3) の場合，通常のスーパスカラ・プロセッサと同様の方法で，値またはそのタグを得，命令は命令ウィンドウに発行される．(1) の場合，同期テーブルよりタグを得，同じく命令ウィンドウに発行される．従来の full/empty ビットによる同期機構では，受信待ちの命令はレジスタ・フェッチのステージで停止し，後続の命令の実行をブロックするが，我々の機構では，命令は命令ウィンドウで受信値を待ち合わせるので，後続の命令の実行をブロックすることはない．

[簡単な例]

同期/通信における典型的な動作を例を用いて説明し，我々の機構では同期/通信のオーバヘッドが 2 サイクルであること，また，同期がノンブロッキングであることを示す．

2 つのプロセッサにおける次の場合を考える．プロセッサ 1 では，命令 i1 と i2 を実行する．i1 はレジスタ r1 を生成する演算命令であり，i2 は i1 の実行結果 r1 をプロセッサ 2 に送信する send 命令である．プロセッサ 2 では，命令 i3 と i4 を実行する．i3 は i1 の実行結果 r1 を参照する命令であり，i4 はどの命令にも依存しない独立な演算命令とする．

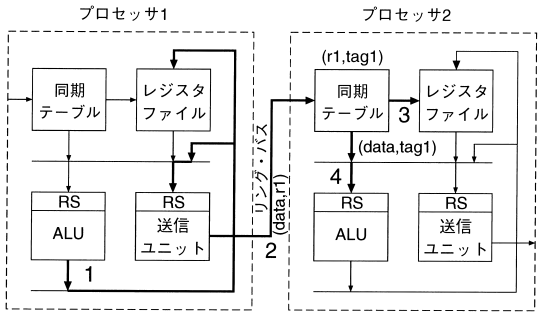


図 2 同期/通信の例
Fig. 2 Example of communication and synchronization.

プロセッサのパイプラインは，命令フェッチ，命令デコード，実行，書き戻しの 4 ステージで構成されているとする．**図 2** を用いて実行の過程をサイクルごとに説明する．図において，点線で囲んだ部分は 1 つのプロセッサを表す．プロセッサの内部は簡略化し，ここでの説明に関係する部分のみ示す．RS はリザベーション・ステーションを表す．

初期状態：図においてプロセッサ 2 の同期テーブルは，レジスタ r1 に対応するエントリの R フラグがセットされており，タグとして tag1 を保持しているとする．これ以外のエントリの R フラグは，プロセッサ 1 の同期テーブルを含めてすべてリセットされているとする．このサイクルでは，プロセッサ 1 は命令 i1 と i2 を，プロセッサ 2 は命令 i3 と i4 をフェッチしている．

第 1 サイクル：プロセッサ 1 において，命令 i1 と i2 が同期テーブルを参照し，ソース・レジスタに対応するエントリを読み出す．読み出されたエントリの R フラグはすべて 0 なので，どのソース・レジスタも受信する値ではないことが分かる．この場合，通常のスーパスカラ・プロセッサと同一の手順で，命令はリザベーション・ステーションに発行される．

プロセッサ 2 においても，命令 i3 と i4 が同期テーブルを参照し，ソース・レジスタに対応するエントリを読み出す．命令 i3 のソース・レジスタ r1 に対応する R フラグは 1 なので，r1 は受信する値であることが分かる．この場合，同期テーブルより得られるタグ tag1 を r1 に付け，命令 i3 はリザベーション・ステーションに発行される．命令 i4 については，ソース・レ

ジスタに対応するRフラグは0なので，通常のスーパスカラ・プロセッサと同一の手順で発行される．

第2サイクル：プロセッサ1において，命令i1が実行される．その結果は，レジスタ・ファイルに送られると同時に，送信ユニットのリザベーション・ステーションで待っているsend命令i2にフォワーディングされる（図2の1）．同様にプロセッサ2においては，命令i4が実行される．

第3サイクル：プロセッサ1において，send命令i2が送信ユニットで実行され，レジスタr1の値とレジスタ番号が，リング・バスを介してプロセッサ2に送信される（図2の2）．プロセッサ2では，送信されてきたレジスタ番号をインデクスとして同期テーブルを参照し，対応するRフラグとタグを読み出す．

第4サイクル：プロセッサ2は，前サイクルで読み出したRフラグの値が1なので，受信動作を行う．まず，送信されてきたレジスタr1の値をレジスタ・ファイルに書き込み，Rフラグをリセットする（図2の3）．同時に，r1の値と読み出したタグtag1を，リザベーション・ステーションのすべてのエントリに放送する．リザベーション・ステーションで待ち合わせていた命令i3は，自身の保持するタグと放送されたタグが一致するので，値を受け取る（図2の4）．

第5サイクル：命令i3はオペランドがそろったので実行される．

以上から分かるように，命令i1とi3の間に発生する同期/通信のオーバヘッドは，第3サイクルと第4サイクルの2サイクルである．また，同期が成立していない命令i3は，リザベーション・ステーションで受信値を待ち合わせることにより，後続する命令i4の実行を停止させることはない．つまり，ノンブロッキングの同期を実現している．

3.3.2 投機的実行プロセッサにおける実現

投機的に命令を実行するプロセッサでは投機に失敗した場合，一般に，投機的実行結果をすべて取り消しプロセッサ状態を戻したうえで，分岐予測を誤った点からプログラムの実行を再開する．前項で説明した機構では，投機を失敗したとき，プロセッサ状態に影響を与える動作で，取り消すことのできないものが存在する．それは，受信に関する動作である．この動作には，送信値を受け取る動作と受け取らない破棄という動作がある．前者については，受信により書き込まれたレジスタを再実行時に上書きするので，プログラムの意味は保たれ，問題ない．これに対して破棄の場合は，プログラムの意味を保つことができなくなる．これを**図3**に示す例で説明する．

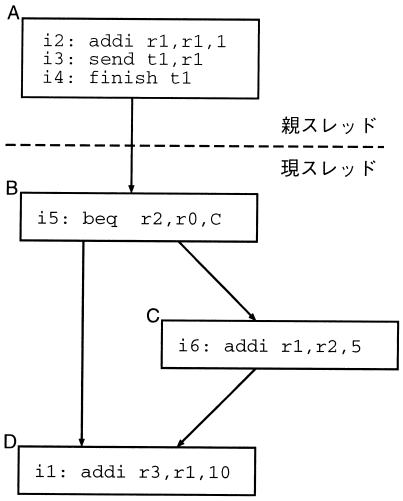


図3 プログラム例
Fig. 3 Program example.

図3において，ブロックDにある命令i1に注目する．命令i1を含むスレッド（以下，現スレッドと呼ぶ）は，命令i5から始まるスレッドであり，その親スレッドは命令i4で終了するスレッドである．命令i1のソース・レジスタr1は，親スレッドの命令i2，あるいは，現スレッドのi6によって定義される．どちらの定義を参照するかは，実行時のブロックBからの制御移行に依存する．もしも命令i5の分岐が成立せず，制御がブロックCを経由せずDに移行すれば，命令i1のレジスタr1は，命令i2によって定義された値である．命令i5の分岐が成立し，制御がBからCを経由してDに移行すれば，i6によって定義された値である．

ここで，命令i5の分岐は成立と予測され，命令i6が投機的に実行された場合を考える．この実行により，レジスタr1に対応するRフラグはリセットされる．これにより，命令i1はr1に関して先行プロセッサから送られてくる値を待つのではなく，命令i6の実行結果をオペランドとすることを認識する．このとき，先行プロセッサで命令i3が実行され，命令i2が定義したレジスタr1の値が送信されてきたとする．r1に対応するRフラグはリセットされているので，この値は受信されず破棄される．

ここで投機が失敗したとする．命令i6とi1の投機的実行結果は破棄され，ブロックBから正しい方向（ブロックDの方向）に向かって実行が再開される．これにより命令i1が再実行される．この場合，命令i1が参照すべきソース・レジスタr1のオペランドは，命令i2が定義した値である．しかしこの値は，投機

的実行時に送信されてきたが，受信されず破棄されている．そのため命令 i1 はこれを得ることができない．

この問題に対処するために，オペランド・フェッチと受信の動作において，次の 2 点を変更する．

(1) R フラグは，対応するレジスタ値を受信したときか，あるいは，そのレジスタを定義した命令がリオーダ・バッファからリタイアするときにリセットする．

(2) オペランド・フェッチにおいては，リオーダ・バッファ検索の結果，ソース・レジスタを定義する命令が見つかったときは，そのソース・レジスタの値またはタグを，R フラグの値にかかわらずフェッチする．

第 1 の修正は，投機の失敗時においても，親スレッドからの送信値を使用することができるようにするための措置である．定義時，つまり，実行終了したときではなく，リオーダ・バッファからリタイアするときに R フラグをリセットする点が，非投機的実行プロセッサの場合の実現方法と異なる．第 2 の修正は，第 1 の修正に対応して，オペランドあるいはそのタグを正しくフェッチするための措置である．R フラグの参照に加えて，リオーダ・バッファの検索結果を判断に加えた点が異なる．

以上の修正により，オペランド・フェッチと受信が投機的実行を行うプロセッサにおいても正しく行われることを説明する．いま，ある命令 i1 がソース・レジスタ r1 をフェッチしようとしている場合について考える（図 3 の例ではなく一般的な場合について考えていることに注意）．r1 に対応する R フラグとリオーダ・バッファの参照結果に応じて，以下の 3 つ場合が存在する．

(1) R フラグが 0 の場合

この場合，r1 をすでに受信したか，あるいは，r1 を定義する命令が i1 の前に少なくとも 1 つ存在し，すでにリタイアしていることを示している．いずれの場合も，必要とする r1 の値あるいはそのタグは，レジスタ・ファイルかリオーダ・バッファの中に存在する．これは，通常のスーパスカラ・プロセッサで行われている手順によって得ることができる．よって，R フラグが 0 の場合のオペランド・フェッチ動作は正しい．

また，r1 を定義する命令のリタイアによって R フラグがリセットされた場合において，先行プロセッサから r1 の値が送られてくることがあるが，この場合，R フラグは 0 なので，それは破棄される．したがって，r1 が不正に上書きされることはなく，プログラムの意味は保たれる．

(2) R フラグが 1 であり，かつ，リオーダ・バッファに r1 を定義する命令が存在しない場合

R フラグが 1 ということより，r1 は受信しておらず，かつ，r1 を定義する命令があったとしても，まだリタイアしていないことが分かる．後者はこの場合あてはまらないので，結局，r1 を受信していない場合である．したがってこの場合，同期テーブルより r1 に対応するタグを得て，リザベーション・ステーションで受信を待つ必要があり，オペランド・フェッチ動作は正しい．

リオーダ・バッファ内の命令はプログラム順にリタイアし，現在リオーダ・バッファに r1 を定義する命令はないから，r1 に対応する R フラグは，r1 の値を受信するまでリセットされることはない．つまり，送信されてくる値を i1 が受け取ることは保証されている．

(3) R フラグが 1 であり，かつ，リオーダ・バッファに r1 を定義する命令が存在する場合

リオーダ・バッファに r1 を定義する命令が存在するので，命令 i1 が参照すべき値は，リオーダ・バッファ内の r1 に関する最新の定義である．これは，リオーダ・バッファより得られ，この場合のオペランド・フェッチ動作は正しい．

R フラグは 1 なので，リオーダ・バッファ内の r1 を定義する命令がリタイアするまでに，先行プロセッサから値が送られてくると，これを受信し，レジスタ・ファイルに書き込まれる．これがプログラムの意味を変えることはないことを，次の 3 点によって示す．

- r1 に対応する R フラグが 1 であるということは，現スレッドの命令によってレジスタ r1 に値が書き込まれたことは，1 度もないことを示している．つまりレジスタ r1 は現スレッドにおいては「空」である．
- 先行プロセッサから送られてくる r1 の値は，現スレッドに先行するいずれかのスレッドにおいて，r1 に対する最後の定義値である（詳細は 3.4 節で述べる）．
- 現プロセッサで空のレジスタが過去のスレッドでの最終値を持つようになることは，SKY のスレッドが逐次的実行における 1 部分であり，スレッドは逐次に生成されるという性質より正当である．

リオーダ・バッファ内の r1 を定義する命令は，まだ投機的状態である場合とすでにコミットされている状態（制御依存がすべて解消されている状態）という 2 つの場合がある．すでにコミットされている場合，受信したレジスタは，リオーダ・バッファ内命令によって上書きされ，プログラムの意味は保たれる．投機的

状態であったが，投機に成功しコミットされた場合も同様である．一方，投機に失敗した場合，図3で示したように，命令 i1 が参照すべき値が受信値となることがある．受信値はレジスタ・ファイルに存在するので，再実行の際にはこれを得ることができ，プログラムの意味は保たれる．

以上，オペランド・フェッチと受信が，投機的実行を行うプロセッサにおいても正しく行われることを示した．

ここで述べた方法以外の解決法として，投機的実行時の送信記録を残し，先行プロセッサに送信を要求する方法が考えられる．この方法では，あるプロセッサで送信するレジスタは，後続のプロセッサの投機が完了し，送信を要求されることがないと分かるまで，別の命令による上書きは許されず，保持しておかなければならない．このような制御は複雑であるばかりでなく，新たに逆依存を生じさせる結果となり，性能が低下する．別の方法として，投機的実行時に送られてきた値を，別途バッファを用意し保持する方法が考えられる．この方法は実現は簡単であるが，そのようなバッファを必要とせず，制御も同程度に簡単な本機構の方が優れている．

3.4 コンパイラの概要

コンパイラはスレッド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を分割した後，スレッド間で送信すべきレジスタを求め，それを送信する send 命令をプログラムに挿入する[5),13)]．SKY においてプログラム分割とは，**フォーク点**と**子の開始点**を定めることである．フォーク点とは，スレッドを新たに生成するプログラム上の位置であり，子の開始点とは子スレッドが実行を開始する位置である．フォーク点に fork 命令を挿入し，子の開始点の直前に finish 命令を挿入する．

以下にコンパイラの特徴を述べる．なお，詳細は本論文の範囲を超えるので，文献 5)，13) を参照されたい．

(1) **制御等価**[18)]な部分に着目してプログラム分割の候補を決定する．一般に，制御フロー・グラフ[1)]において，基本ブロック X から出口ノードに向かうどのパスも基本ブロック Y を通るとき，Y は X を**後支配**[18)]するといい，入口ノードから Y へ向かうどのパスも X を通るとき，X が Y を**支配**[1)]するという．また X が Y を支配し，同時に Y が X を後支配するとき，組 (X,Y) は制御等価であるという．フォーク点と子の開始点を含む基本ブロックをそれぞれ X，Y とする．SKY の実行モデルでは，Y は X を後支配しなければならない．しかし，Y が X を後支配する組は数が非常に多いので，現実的な計算量で分割するために，制御等価な組をプログラム分割の候補とすることとした．

(2) 各候補についてスレッド並列実行による**性能利得**をプロファイルを用いて計算し，利得が大きいものを採用する．あるスレッドとその子スレッドの並列実行による性能利得を，2つのスレッドの実行がオーバラップする間に子スレッドで実行された命令の数と定義する．コンパイラは，フォーク点から子の開始点までの**距離**，および，データ依存関係にある命令間の距離を求め，その最小値を性能利得の推定値とする．ここで点 p_1 から点 p_2 までの距離とは，p_1 から p_2 に至る各パス上の命令数の，パスの実行確率による重み付き平均値とする．パスの実行確率は，分岐のプロファイルから得る．

(3) あるスレッドにおいて送信すべきレジスタ値が子の開始点に**到達**[1)]することが決定する点に send 命令を挿入する．ここで，レジスタ値 r が点 p に到達するとは，r が p または p の後方のどこかで参照される可能性があることをいう．SKY の実行モデルでは，スレッドは子スレッドの開始点で**生きている**（**live**）[1)]レジスタを送信しなければならない．送るべきレジスタは，次の2つに大別できる．生成順にスレッドに番号を付け，$i < j$ のとき，T_i は T_j より後方にあるといい，T_j は T_i より前方にあるということにする．

- **真の送信レジスタ**：現スレッドで定義され，その前方のいずれかのスレッドで使用される可能性のあるレジスタ
- **転送レジスタ**：現スレッドの後方のいずれかのスレッドで定義され，その前方のいずれかのスレッドで使用されるレジスタ

これらの送信のため，値の到達が決定する点に send 命令を挿入する．ただし，転送レジスタにおいて，値の到達はフォーク点においてすでに決定しているので，フォーク点の直後に send 命令を挿入している．

3.5 SKY 専用命令の詳細

SKY 専用命令はすでに述べたように，fork 命令，finish 命令，send 命令の3つである．各スレッドはその親スレッドの制御に依存してはならないので，これらの専用命令は投機的に実行せず，スレッド内における制御依存が解消した後に実行する．各命令の形式は以下のとおりである．

```
fork    tid,offset
finish  tid
```

表1 ベンチマーク・プログラム
Table 1 Benchmark programs.

ベンチマーク	プロファイル用トレース		シミュレーション用トレース	
	入力セット	命令数	入力セット	命令数
compress95	train/test.in	42M	ref/bigtest.in (reduced to 30K elements)	113M
gcc	ref/regclass.i	141M	ref/genoutput.i	99M
go	11x11 board, level 4, ref/null.in	106M	9x9 board, level 6, train/2stone9.in	75M
ijpeg	train/vigo.ppm, 68x48 pixcels	92M	ref/specmun.ppm	437M
li	test/test.lsp	474K	train/train.lsp	187M
m88ksim	test/dhry	499M	train/dcrand	121M
perl	train/jumble.pl, train/jumble.in, train/dictionary	61M	train/scrabbl.pl, train/scrabbl.in (add three words), train/dictionary	89M
vortex	ref/persons.1k, ref/vortex.in (reduced iterations)	136M	ref/persons.1k, ref/vortex.in	89M

`send tid,reg`

以下，各命令について説明する．

3.5.1 fork 命令

`fork` 命令は，後続プロセッサが空いていれば，そこに子スレッドを生成する．具体的には，自身の命令アドレスと命令にエンコードされている `offset` を加え，後続プロセッサの PC に書き込む．同時に，自身のプロセッサの **TID レジスタ**と呼ぶ専用のレジスタに，`tid` を書き込む．レイテンシは 2 サイクルである．最初のサイクルで子スレッドの開始アドレスを計算し，次のサイクルで PC と TID への書き込みを行う．

TID（Thread Identifier）は，子スレッドに与えられる識別子である．あるスレッドに対しコンパイラが挿入する `send` 命令と `finish` 命令は，そのスレッドから生成されるある特定の子スレッドに対応して挿入されるものである．一方，子スレッドが生成されるかどうかは，前述のように，`fork` 命令が実行される際の後続プロセッサの空き状況による．したがって，実行中に現れたすべての `send` 命令と `finish` 命令を実行してよいわけではなく，実際に実行された `fork` 命令によって生成された子スレッドに対応する `send` 命令と `finish` 命令だけを実行する必要がある．TID はこれを認識するためのものである．

3.5.2 finish 命令

コンパイラは `finish` 命令挿入の際，対応するスレッドの `TID` を命令にエンコードする．実行時は，TID レジスタの内容と `tid` を比較し，一致すれば現スレッドを停止する．そうでなければ無効化される．

`finish` 命令は，命令フェッチを停止し，後続の命令をすべて無効化する．そして，その `finish` 命令がリオーダ・バッファの先頭にくるまで待った後，現プロセッサを解放する．リオーダ・バッファ内で待つことによって，`finish` 命令に先行するすべての命令がリタイアすることを保証する．`finish` 命令のレイテンシは，命令フェッチを停止し，後続の命令をすべて無効化するための 1 サイクルとリオーダ・バッファ内で待機していたサイクルを加えたものである．

3.5.3 send 命令

`finish` 命令と同様に，コンパイラは `send` 命令挿入の際に，対応するスレッドの TID を命令にエンコードする．実行時は，TID の内容と `tid` を比較し，一致すれば `reg` の値と番号を後続プロセッサに送る．そうでなければ無効化される．レイテンシは，3.3 節で述べたように 2 サイクルである．

4. 評 価

本章ではまず，評価環境について述べる．次に，プロセッサ数を変化させたときの SKY の性能を評価し，何が性能に大きな影響を与えているについて議論する．また，ほぼ同一規模のスーパスカラ・プロセッサとの性能比較もあわせて行う．次に，命令ウィンドウ・ベースの同期機構の有効性を評価し，その後，同期/通信オーバヘッドの性能への影響を評価する．最後に，専用命令の挿入によるコードの増加について述べる．

4.1 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．**表1** に各ベンチマークにおける入力セットを示す．同表に示すように，異なる 2 つの入力セットを用いた．1 つは，プログラムのプロファイルを採取するためのものであり，もう 1 つは，シミュ

表 2 SKY の基本モデル
Table 2 Baseline model of SKY.
(a) プロセッサ

フェッチ幅	8命令
発行幅	8命令
命令ウィンドウ	64エントリ
リオーダ・バッファ	128エントリ
機能ユニット	8つの整数演算ALU, 8つの浮動小数点演算ALU, 4つのロード/ストア・ユニット, 2つの分岐ユニット, 4つの送信ユニット
分岐予測機構	深さ32のリターン・アドレス・スタック

(b) 共有資源

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
メモリあいまい性検出機構	理想
分岐予測機構	1024エントリ, 連想度2のBTB, 1024エントリ, 履歴長4のPAp[25]
分岐予測ミスペナルティ	4サイクル

レーションを行うためのものである．ベンチマーク・プログラムは NEC EWS 4800/360AD（MIPS R4400）のコンパイラで最適化し，MIPS R2000[6]用のコードを生成し，これを測定に用いた．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 pixie[17]を用いて採取した．

表 2 に，SKY の基本モデルを示す．以下特に断らない限り，評価においてはこの基本モデルを使用する．なお，本論文で理想としているメモリ曖昧性検出機構は，SKY の性能を決定する重要な要素の 1 つであり，今後の課題である．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，つまり 8 命令発行の単一のスーパスカラ・プロセッサとする．

以下に，SKY を構成する 1 つのプロセッサの命令発行幅を 8 にした理由を示す．**表 3** に命令発行幅を変化させた場合の単一のスーパスカラ・プロセッサの性能を示す．表の IPC は全ベンチマークによる平均値である．表より分かるように，命令発行幅を 2 倍に増加させることによる性能向上率は，命令発行幅を 8 にするまでは 30%前後と大きいが，それ以降は急速に低下する．SKY は非数値計算応用で高い性能を達成することを目的としているので，並列性の重要な源である ILP を最大限引き出す必要がある．命令発行幅を広げるとハードウェアの複雑度が増加することを考慮すると，SKY を構成する 1 つのプロセッサは，大きな性能向上が望める最大の命令発行幅である 8 が妥当と考えた．**表 4** に，各ベンチマーク・プログラムにおける 8 命令発行のスーパスカラ・プロセッサの IPC を示す．

表 3 単一スーパスカラ・プロセッサの性能
Table 3 Single superscalar processor performance.

命令発行幅	IPC
2	1.39
4	1.86
8	2.41
16	2.60
32	2.64

表 4 8 命令発行スーパスカラ・プロセッサの性能
Table 4 Eight-issue superscalar performance.

ベンチマーク	IPC
compress95	2.73
gcc	2.07
go	1.84
ijpeg	3.63
li	2.55
m88ksim	2.04
perl	2.25
vortex	2.57

4.2 性　　能

図 4 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．プロセッサ数が 2

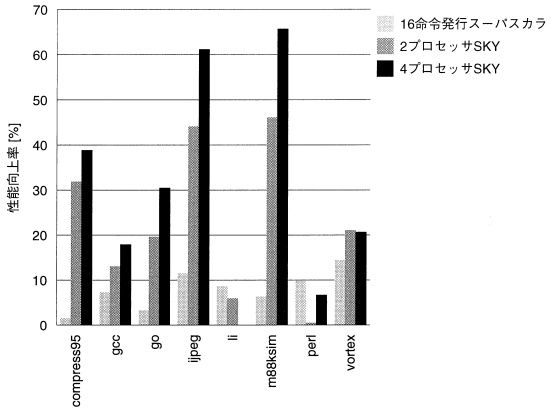


図 4 基準スーパスカラ・プロセッサに対する性能向上
Fig. 4 Speedup over the base superscalar processor.

と 4 の場合について性能を測定した．図には，比較のため，表 3 の 16 命令発行のスーパスカラ・プロセッサの性能向上率についても示す．このプロセッサのハードウェア量は，2 プロセッサ構成の SKY のハードウェア量と同程度である．

4.2.1 2 プロセッサ構成の SKY

2 プロセッサ構成の SKY は，最大で 46.1%，平均で 21.8%の性能向上率を達成している．特に，compress95，ijpeg，m88ksim では，性能向上率は 30%以上と大きい．この性能向上率は，同程度のハードウェア量を投入した 16 命令発行のスーパスカラ・プロセッサより最大で 37.3%，平均で 13.0%高い．SKY を構成する 8 命令発行のスーパスカラ・プロセッサは，16 命令発行のものより大幅に単純なので，実現可能なサイクル時間には大きな差が生じる[15]．したがって，実際の性能差は IPC の差よりさらに大きく広がると考えられる．

SKY は以上述べたように，大きな性能向上を達成しているものの，プログラムによって性能向上率に大きな違いが見られる．ijpeg，m88ksim，compress95 では大きな性能向上を達成しているものの，逆に li と perl ではほとんど性能が向上していない．これらでは，16 命令発行のスーパスカラ・プロセッサより性能が低いという結果を得ている．

解析の結果，このような性能差が生じる原因には次の 2 つがあることが分かった．1 つは，コンパイラがプログラムから多くの TLP を抽出できるかどうかという点である．抽出できればスレッド並列実行により性能は向上し，そうでなければ，スレッドが並列に実行される状況が少なく，性能向上は抑えられる．もう 1 つの原因は，スレッド並列実行がスレッド内の ILP 利用に悪影響を与えるかどうかという点である．SKY は，同期については，命令ウィンドウ・ベースの機構により ILP 利用が妨げられないよう工夫を施しているが，このほかにも ILP 利用に大きな影響を及ぼす事項があることが評価により分かった．以下この 2 点について考察する．

［TLP の抽出］

まず，コンパイラによる TLP の抽出について考察する．このために，次の 2 つの原因により，1 サイクルあたりに命令発行スロットが平均でいくつ「空き」となったかを測定した．

- スレッド不在
- スレッド間データ依存

ここで，**空きスロット**とは，次のようなものである．1 つのプロセッサの同時発行命令数は最大 8 である．これを，「このプロセッサは 8 つの発行スロットがある」ということとする．2 プロセッサ構成の SKY の同時発行命令数は全部で最大 16 であるから，スロット数は 16 である．これらのスロットは実行時においては，すべて有効な命令によって埋められるわけではなく，データ依存など種々の原因で埋められないことがある．命令を埋められなかったスロットのことを，空きスロットと呼ぶ．

上記 2 つの原因のうち，スレッド不在で多くの空きスロット数が生じるのは，コンパイラがプログラムよりプロセッサ数に見合うだけの十分な TLP を抽出することができなかった場合である．スレッド間データ依存で多くの空きスロットが生じるのは，コンパイラが何らかの理由で誤って TLP の少ないスレッド分割を行ってしまった場合である．

図 5 に評価結果を示す．縦軸は，上記 2 つの原因によるサイクルあたりの平均空きスロット数である．各棒グラフの下の部分は，スレッド不在による空きスロット数であり，上の部分は，スレッド間依存によるものである．同図よりまず分かることは，スレッド間依存による空きスロット数は非常に少ないということである．このことよりコンパイラは，TLP の少ないスレッドへの分割は行っていないことが分かる．

次に分かることは，perl ではスレッド不在による空きスロットが非常に多いということである．perl では，全スロットのほぼ半数が，スレッド不在で空いている．つまり，ほとんどの時間 2 つのプロセッサの一方でしかスレッドは実行されていない．これが，perl で性能向上がほとんどない原因である．逆に，ijpeg ではスレッド不在による空きスロットが 2 程度と少なく，大きな性能向上を達成している原因であることが分かる．

コンパイラが十分な TLP を抽出できなかった理由は，3 つ考えられる．1 つは，我々の現在のコンパイ

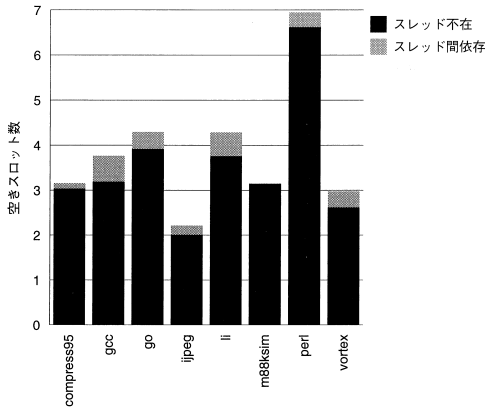


図 5 スレッド不在とスレッド間依存によるサイクルあたりの空きスロット数（2 プロセッサ SKY）
Fig. 5 Number of empty slots per cycle due to thread nonexistence and interthread dependences (2-processor SKY).

ラの能力不足である．たとえば，次の点において能力に不足がある．

- コンパイラは，制御等価なブロックの組に着目している．これは制御依存がないブロックの組の一部であり，解析範囲が制限されている．
- 関数ごとに解析を行っている．このため，フォーク点を含む関数の外に子スレッドの開始点を設けることはできない．

前者は，論理的には後支配の組に対し解析を行うことで解決できる．ただし，3.4 節で述べたように，後支配の組は制御等価の組に比べて非常に多いので，コンパイラを実用的なものにするには，計算時間を短縮する工夫が必要である．後者は，関数を展開することで解決できる．ただし，コード量が増加するので，このトレードオフを考慮した方法が必要である．これらについては，今後検討していく予定である．

コンパイラが十分な TLP を抽出できなかった 2 つめの理由は，3.1 節で述べた SKY のマルチスレッド実行における制約である．特に，スレッドの逐次生成の制約は単純化に貢献しているが，TLP の抽出には大きな制約を課している．たとえば，次のようなコードを考える．

```
        fork  t1, L1;
        foo();
L1:     ...
```

このコードは，関数 `foo` と `L1` 以下のコードの間に存在する TLP を利用するものであるが，`foo` 内に豊富な TLP が存在しても，それを利用することができない．制約を緩和すればこの問題は解消できるが，マルチスレッド実行の複雑さが増加し，そのオーバヘッドとのトレードオフが存在する．

3 つめの理由は，プログラム中に内在する TLP がそもそも十分には存在しないということである．存在しない TLP をコンパイラは引き出すことはできない．これを証明することは非常に難しいが，今後調査の必要がある．

［マルチスレッド実行による ILP 利用への悪影響］

perl と ijpeg 以外のプログラムにおいては，スレッド不在による空きスロット数は比較的近い値を示している．しかし，図 4 に示したように性能向上率に大きな差が見られる．これは調査の結果，マルチスレッド実行が ILP 利用に対して悪影響を与えており，その影響の大きさによることが分かった．マルチスレッド実行が原因で ILP の利用が妨げられたとき，これを ILP を損失したということとする．ILP 損失の原因として，次の 2 つがある．

- スレッドの実行開始と終了時の命令不足
- 分岐予測精度の低下

スレッドの実行開始と終了時には，十分な命令がプロセッサに供給されないために，ILP の量が定常状態に比べて低下する．SKY では，スレッドの最初の命令がフェッチされ実行を開始するまで 3 サイクルかかり，この間プロセッサは実行する命令がない．また，`finish` 命令が発行された後，リオーダ・バッファからリタイアする十数サイクルの間発行する命令がない．このような期間は単一のスーパスカラ・プロセッサにはなく，マルチスレッド実行により生じたものであるから，ILP 損失ということができる．これらの期間のサイクル数はあまり大きくないように感じられるが，TLP の少ない非数値計算プログラムでは性能に対して大きな影響を与える．

また，マルチスレッド実行によって分岐予測精度が低下することも分かった．SKY では，分岐予測機構はプロセッサ間で共有している．静的に 1 つの分岐が異なるスレッドで同時に実行される場合，PAp 予測器の**分岐履歴表**（BHT: Branch History Table）には，逐次実行とは異なる順で履歴が記録される．これはたとえば，ループの各イタレーションをスレッドとし，これらを同時に実行する場合，そのループに含まれる分岐に対して生じる．このような場合，逐次実行において存在した履歴パターンと将来の分岐方向の間の相関を，PAp 予測器はとらえることができなくなり，予測を誤る確率が高まる．予測を誤る確率が高まると，パイプラインへの命令供給が阻害され，ILP の利用が妨げられる．

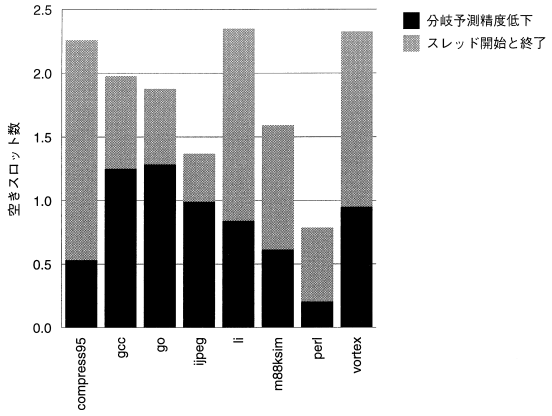


図 6 マルチスレッド実行によるサイクルあたりの空きスロット数（2 プロセッサ SKY）
Fig. 6 Number of empty slots per cycle due to multithread execution (2-processor SKY).

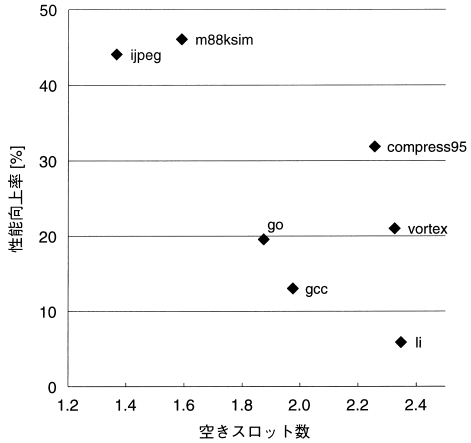


図 7 マルチスレッド実行による ILP 損失と性能向上率の相関（2 プロセッサ SKY）
Fig. 7 Correlation between ILP loss due to multithreaded execution and speedup (2-processor SKY).

図 6 に，これら 2 つの原因によって生じたサイクルあたりの平均空きスロット数を示す．各棒グラフの下の部分は，分岐予測精度の低下により生じた空きスロット数であり，上の部分は，スレッドの実行開始と終了により生じた空きスロットである．同図より分かるように，TLP の利用が少ない perl を除いて，1.5〜2.5 ものスロットがマルチスレッド実行によって空いてしまっている．一般に，スレッド・サイズ（後に表 6 で示す）が小さいプログラムほど，スレッドの開始と終了による空きスロット数が大きい．これは，そういったプログラムほどスレッドの開始と終了の頻度が高いからである．一方，分岐予測精度の低下は，プログラムをスレッドにどのように分割するかによる．この解析はプログラムが大きく複雑なため，プログラムとの明確な関係をいうことは難しい．

図 7 に，マルチスレッド実行による ILP 損失と性能向上の間の相関を示す．横軸は ILP 損失による空きスロット数であり，縦軸は性能向上率である．TLP 利用の少ない perl については，この議論の対象外なのでプロットしていない．同図より，空きスロット数が多いほど性能向上率が低いという相関があることが分かる．ベンチマーク別に見ると，図 5 より li は TLP を比較的多く利用しているといえるが，ILP 損失が大きく，その結果，性能が向上していない．逆に go は，li と同程度 TLP を利用しているが，ILP 損失が少ないので，li より高い性能向上率を達成している．また，li と同程度の ILP 損失であっても，compress95 と vortex はより多くの TLP を利用しているので，li より高い性能向上率を達成している．m88ksim と ijpeg は，TLP 利用量が大きく，かつ，ILP 損失が少ないので，他のプログラムに比べ大きな性能向上を達成している．

スレッド実行開始と終了による ILP 損失を抑える 1 つの方法として，1 つのプロセッサ内において，`fork` 命令による新しいスレッドの開始と `finish` 命令によるスレッドの終了をオーバラップさせることが考えられる．現在プロセッサの解放は，`finish` 命令がリオーダ・バッファからリタイアするまで行われないが，これを `finish` 命令実行完了時に解放するようにすれば，新しいスレッドをそのプロセッサで開始することができる．このようにスレッドの開始と終了をオーバラップさせることができれば，開始と終了時の命令供給不足をある程度回避することができると思われる．これを実現するには，何らかのハードウェア支援が必要であり，今後の課題である．

一方，マルチスレッド実行による分岐予測精度の低下を抑えるのは，かなり困難な問題である．なぜならば，現在知られている分岐予測機構はすべて，逐次実行における分岐の過去の履歴と将来の分岐方向との間の相関を利用しているが，マルチスレッド実行では，分岐が逐次に実行されないので，この相関を利用することができないからである．

4.2.2 4 プロセッサ構成の SKY

4 プロセッサ構成の SKY は基準プロセッサに対して，最大で 65.7%，平均で 21.8%の性能向上率を達成している（図 4）．しかし，2 プロセッサ構成の SKY に対して最大でも 13.4%しか性能が向上しておらず，コスト性能比が良いとはいえない．これはスレッド不在が主因である．**図 8** に，スレッド不在とスレッド間依存によるサイクルあたりの空きスロット数を示す．どの

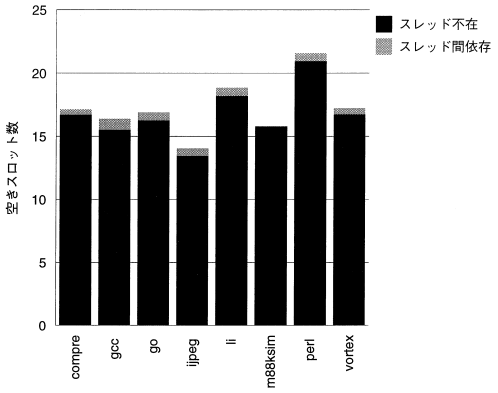


図 8 スレッド不在とスレッド間依存によるサイクルあたりの空きスロット数（4プロセッサ SKY）
Fig. 8 Number of empty slots per cycle due to thread nonexistence and interthread dependences (4-processor SKY).

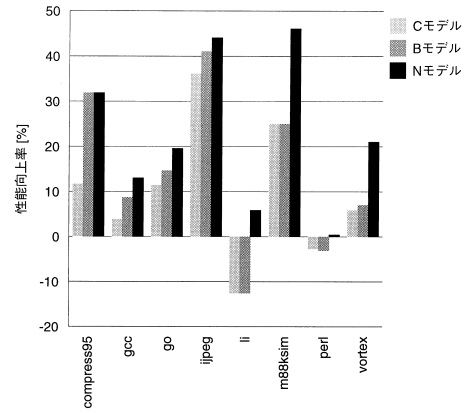


図 9 同期機構の評価結果（2プロセッサ SKY）
Fig. 9 Evaluation results of synchronization mechanisms (2-processor SKY).

プログラムでも，2プロセッサ分のスロットである 16 に近いスロットが，スレッド不在で空きとなっている．2プロセッサを超えるプロセッサを効率良く利用するには，TLP 抽出に関してさらなる検討が必要である．

4.3 同期機構

本節では，次の 2 つの有効性に関して同期機構を評価する．

- 同期の粒度が全レジスタでなく 1 レジスタであること
- ノンブロッキング同期

2プロセッサ構成の SKY を基本とし，以下の 3 つの同期機構を実現したモデルについて評価した．

C モデル（coarse-grain model）：このモデルでは，子スレッドの開始点に到達するレジスタ値が現スレッドにおいてすべて定義されるまで子スレッドを生成しない．子スレッドを生成するときに，レジスタ・ファイルの内容をすべて後続プロセッサにコピーする．レジスタ転送のバンド幅は十分にあるとし，コピーには 1 サイクルしかかからないと仮定した．これにより，いったんスレッドが生成されると，レジスタに関する同期は必要ない．このため，個々のレジスタについての同期/通信機構が必要なく単純である．しかし，通信データの粒度を全レジスタにまで粗くしたため，同期に無駄なオーバヘッドが生じている．

B モデル（blocking model）：このモデルでは，SKY と同様，個々のレジスタについての同期/通信機構を持つが，受信待ち命令に後続する命令の実行がブロックされる．このモデルは，full/empty ビットによる既存の同期モデル[2),7),19)]である．

N モデル（non-blocking model）：命令ウィンドウ・ベースの同期機構を有し，ブロッキングの生じない同期を実現する SKY のモデルである．

図 9 に評価結果を示す．縦軸は基準プロセッサに対する性能向上率である．同図より分かるように，C モデルは，どのベンチマーク・プログラムにおいても N モデルより性能が低い．基準プロセッサに対する平均の性能向上率は，わずか 8.9%である．このことより，レジスタ通信バンド幅がたとえ十分にあったとしても，全レジスタという粗い粒度での同期は性能低下の大きな要因となることが分かる．

B モデルの性能向上率も，平均で 12.9%と低い．compress95 では N モデルに近い性能を示しているが，そのほかでは N モデルに劣る．特に，m88ksim と vortex において大きな違いを示している．N モデルは B モデルより最大で 16.9%，平均でも 7.8%性能が高い．このことより，ノンブロッキング同期機構によりスレッド内 ILP の利用を妨げないことは，性能向上にとって重要であることが分かる．

4.4 同期/通信のオーバヘッド

同期/通信のオーバヘッドが性能に与える影響を評価する．**図 10** に，同期/通信のオーバヘッドが 2 サイクル場合を基準とし，オーバヘッドを 4，8，16 サイクルと増加させた場合の 2 プロセッサ構成の SKY での性能低下率を示す．

図 10 より分かるように，ほとんどの場合，オーバヘッドの増加に従い，性能が低下していく．オーバヘッドが 4 サイクルの場合，性能低下率は最大で 1.9%であり，この程度は許容できることが分かる．一方，オーバヘッドが 8 サイクルの場合，性能低下率は，平均では 4.4%とあまり大きくないが，プログラムによっては 12.2%もの大きな性能低下が生じている．また，オーバヘッドが

16サイクルの場合，平均で12.6%，最大では25.9%にもなり，性能を著しく低下させることが分かる．

なおvortexでは，オーバヘッドが4サイクルのときの方が2サイクルのときより，わずかに（0.6%）性能が向上している．SKYでは，fork命令が実行されスレッドが生成されるかどうかは，後続プロセッサの空き状況によるので，同期/通信のオーバヘッドが変われば，生成されるスレッドも変わる．この効果が性能に現れたと思われる．

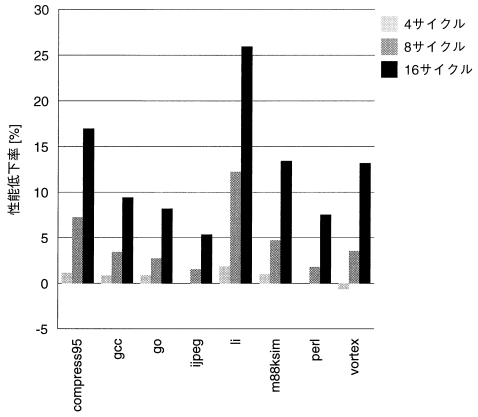


図10 同期/通信のオーバヘッドによる性能低下（2プロセッサSKY）
Fig. 10 Performance degradation due to comunication and synchronization overhead (2-processor SKY).

4.5 コード量と命令の分布

SKYが実行するプログラムには，通常の逐次プログラムに対して，SKY専用の命令を追加しなければならない．命令を追加することによりコード量が増加し，メモリ・システムの性能を低下させる可能性がある．**表5**に，SKYのプログラムにおける専用命令の静的分布を示す．`send`命令についてはさらに，3.4節で述べた真の送信と転送の2つに分類している．同表より，コード量の増加率は，平均でわずか4.8%，最大でも10.5%と小さい．したがって，メモリ・システムに与える影響はほとんどないと考えられる．

表6に，2プロセッサ構成のSKYにおける，1スレッドあたりの動的命令数の平均を示す．命令は，専用命令を除く元のプログラムにおける命令と，3つのSKY専用命令に分類している．`send`命令に関しては

表5 SKY専用命令の静的分布
Table 5 Static distribution of SKY special instructions.

ベンチマーク	fork	finish	send		合計
			真の送信	転送	
compress95	1.01%	1.01%	7.02%	1.50%	10.53%
gcc	0.72%	0.72%	5.64%	1.02%	8.10%
go	0.66%	0.66%	5.53%	1.25%	8.11%
ijpeg	0.29%	0.29%	2.34%	0.36%	3.27%
li	0.31%	0.31%	1.78%	0.38%	2.78%
m88ksim	0.02%	0.02%	0.12%	0.01%	0.15%
perl	0.32%	0.32%	2.36%	0.34%	3.35%
vortex	0.27%	0.27%	1.35%	0.24%	2.12%

表6 スレッドあたりの動的命令数
Table 6 Dynamic instruction count per thread.

ベンチマーク	元のコード	fork	finish	send		合計
				真の送信	転送	
compress95	193.3	1.0	1.0	11.2	1.3	207.8
gcc	256.0	1.0	1.0	7.7	4.1	269.8
go	263.6	1.0	1.0	9.4	3.0	278.0
ijpeg	1218.1	1.0	1.0	12.2	2.7	1235.0
li	128.2	1.0	1.0	7.8	1.0	139.0
m88ksim	234.7	1.0	1.0	11.9	0.9	249.5
perl	292.9	1.0	1.0	8.8	1.7	305.4
vortex	250.6	1.0	1.0	3.8	1.2	257.6

さらに，表5と同様，真の送信と転送に分類している．同表に示すように，スレッドあたりの動的命令数は約100〜1000，平均で368である．この数は，従来の粗粒度並列における命令数より非常に小さく，非数値計算に求められる細粒度TLPをSKYが利用していることが分かる．また，SKY専用命令が全命令数に占める割合は，平均4.9%，最大でも7.8%と多くない．また，挿入された専用命令のほとんどはsend命令であるが，これらは互いに独立な命令なので並列に実行され，これらの命令による性能低下はほとんどないと考えられる．さらに，1スレッドあたりの転送用sendの命令の数は，1〜4と非常に小さいことが分かる．このことは，複数のスレッドをまたいで生きるレジスタはほとんどなく，プロセッサが命令実行によりレジスタ転送を行うことは，性能上ほとんど問題がないことを示している．

5. 関連研究

Nayfehらは，マルチプロセッサをチップに集積する場合において，種々のメモリ・システムの性能について詳細な測定を行った[12]．彼らの測定結果によると，L1キャッシュを共有するマルチプロセッサは，共有することにより（具体的には，キャッシュとプロセッサをつなぐクロスバー・スイッチによる），L1キャッシュ・アクセス時間が延びるにもかかわらず，細粒度並列アプリケーションではもちろん，計算量に対してより通信量の少ないアプリケーションにおいても，L2キャッシュ共有や主記憶共有のマルチプロセッサより高い性能を示すことを示した．彼らの研究は，メモリ・システムに関する研究であり，プロセッサに関する我々の研究とは直交するものである．

鳥居らは，スレッド生成時にすべてのレジスタの内容を後続のプロセッサのレジスタに，非常に少ないサイクル数でコピーする特別なレジスタ・ファイルを持つMUSCATと呼ぶアーキテクチャを提案した[20),21)]．レジスタ・ファイル間に高いバンド幅を実現することにより，スレッド間通信の時間を小さくした．しかし，スレッド生成後に生成され後続スレッドに送信しなければならないレジスタ値については，メモリを介しての同期/通信の必要性があり，このオーバヘッドは削減されていない．

Tsaiらは，プロセッサごとに用意した特別なバッファを介して同期/通信を行うスーパスレッド・プロセッサと呼ぶ方式を提案した[22]．この方式では同期による命令実行のブロッキングが発生しないよう命令を静的にスケジューリングする．しかし，現実的には静的スケジューリングは実行時に通りうるすべてのパスについて十分に最適化することはできないので，同期による命令実行のブロッキングを完全に防ぐことは困難であると考えられる．

Tullenらは，単一のスーパスカラ・プロセッサにおいて，命令レベルで複数のスレッドを実行するSMT（Simultaneous Multithreading）と呼ぶ方式を提案した[23]．SMTは，豊富な機能ユニットを持つスーパスカラ・プロセッサにおいて，複数のプログラムを命令レベルで同時に実行することにより，プロセッサのスループットを向上させることができる．しかし，SMTだけでは，1つのプログラムの実行時間を短縮することはできない．

6. まとめ

本論文では，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SKYと呼ぶマルチプロセッサ・アーキテクチャを提案し，詳細な評価を行った．SKYの最大の特徴は，ノンブロッキングのレジスタ同期を行う命令ウィンドウ・ベースの同期機構である．本機構は，TLP利用において，同期がILPに悪影響を与えることを抑制することができ，マルチプロセッサを構成するプロセッサとしてスーパスカラ・プロセッサを用いることに適している．

SKY用にコンパイラを作成し，シミュレーションにより評価を行った．単一のスーパスカラ・プロセッサに対して，2プロセッサ構成のSKYは最大で46.1%，平均で21.8%の性能向上率を達成した．2プロセッサ構成のSKYと同等のハードウェア量を持つ命令発行幅の広い単一のスーパスカラ・プロセッサと性能を比較すると，SKYは最大で37.3%，平均で13.0%高い性能が得られることを確認した．SKYを構成する命令発行幅の狭いスーパスカラ・プロセッサは，命令発行幅の広いスーパスカラ・プロセッサに比べて高速に動作するので，性能差はさらに広がると考えられる．また，ブロッキングの同期に対して，我々の提案するノンブロッキングの同期は，最大で16.9%，平均で7.8%の性能向上率を示した．

SKYは上述のように高い性能と従来の機構に対する優位性を示したものの，まだ多くの課題を残している．1つはコンパイラを改良し，プログラムに内在するTLPをより多く引き出すことが必要である．また，同期以外にTLP利用がILP利用を阻害する問題が大きいことが明らかとなった．この問題を解決する方法を見いだす必要がある．これらについて今後も検討していく予定である．

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）「広域命令レベル並列によるマイクロプロセッサの高性能化に関する研究」（課題番号 1068034），文部省科学研究費補助金基盤研究（C）「分岐予測と投機的実行に関する研究」（課題番号 11680351）および財団法人堀情報科学振興財団助成「広域命令レベル並列性を利用するコンピュータ・アーキテクチャとコンパイラに関する研究」の支援により行った．

参 考 文 献

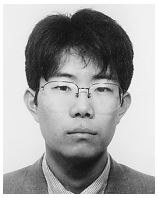
1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company (1986).
2) Breach, S.E., Vijaykumar, T.N. and Sohi, G.S.: The Anatomy of the Register File in a Multiscalar Processor, *Proc. MICRO-27*, pp.181–190 (1994).
3) Butler, M., Yeh, T-Y. and Patt, Y.: Single Instruction Stream Parallelism Is Greater than Two, *Proc. 18th Int. Symp. on Computer Architecture*, pp.276–286 (1991).
4) Hara, T., Ando, H., Nakanishi, C. and Nakaya, M.: Performance Comparison of ILP Machines with Cycle Time Evaluation, *Proc. 23rd Int. Symp. on Computer Architecture*, pp.213–224 (1996).
5) 岩田充晃，小林良太郎，安藤秀樹，島田俊夫：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，97-ARC-128, pp.127–132 (1998).
6) Kane, G.: *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, New Jersey (1988).
7) Keckler, S.W., Dally, W.J., Maskit, D., Carter, N.P., Chang, A. and Lee, W.S.: Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor, *Proc. 25th Int. Symp. on Computer Architecture*, pp.306–317 (1998).
8) 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).
9) 小林良太郎，岩田充晃，安藤秀樹，島田俊夫：非数値計算プログラムの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，1998 年並列処理シンポジウム JSPP'98，pp.87–94 (1998).
10) Kobayashi, R., Iwata, M., Ogawa, Y., Ando, H. and Shimada, T.: An On-Chip Multiprocessor Architecture with a Non-Blocking Synchronization Mechanism, *Proc. 25th Euromicro Conf.*, pp.432–440 (1999).
11) Lam, M.S. and Wilson, R.P.: Limits of Control Flow on Parallelism, *Proc. 19th Int. Symp. on Computer Architecture*, pp.46–57 (1992).
12) Nayfeh, B.A., Hammond, L. and Olukotun, K.: Evaluation of Design Alternatives for a Multiprocessor, *Proc. 23th Int. Symp. on Computer Architecture*, pp.22–24 (1996).
13) 小川行宏：マルチスレッド計算機 SKY のコンパイラにおける命令移動に関する研究，名古屋大学電気学科，電子工学科，および電子情報学科卒業研究論文 (1997).
14) Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K. and Chang, K.: The Case for a Single-Chip Multiprocessor, *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.2–11 (1996).
15) Palacharla, S., Jouppi, N.P. and Smith, J.E.: Complexity-Effective Superscalar Processors, *Proc. 24th Int. Symp. on Computer Architecture*, pp.206–218 (1997).
16) Smith, B.J.: Architecture and Applications of the HEP Multiprocessor Computer System, *Society of Photo-optical Instumentation Engineers*, Vol.298, pp.241–248 (1981).
17) Smith, M.D.: Tracing with Pixie, Technical Report CSL-TR-91-497, Stanford University (1991).
18) Smith, M.D., Horowitz, M.A. and Lam, M.S.: Efficient Superscalar Performance Through Boosting, *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.248–259 (1992).
19) Sohi, G.S., Breach, S.E. and Vijaykumar, T.N.: Multiscalar Processor, *Proc. 22nd Int. Symp. on Computer Architecture*, pp.414–425 (1995).
20) 鳥居 淳，近藤真己，本村真人，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).
21) 鳥居 淳，近藤真己，本村真人，池野晃久，小長谷明彦，西 直樹：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).
22) Tsai, J.Y., Jiang, Z., Ness, E. and Yew, P.C.: Performance Study of a Concurrent Multithreaded Processor, *Proc. 4th Int. Conf. on High-Performance Computer Architecture*, pp.24–35 (1998).
23) Tullsen, D., Eggers, S. and Levy, H.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proc. 22nd Int. Symp. on Computer Architecture*, pp.392–403 (1995).
24) Wall, D.W.: Limits of Instruction-Level Parallelism, *Proc. 4th Int. Conf. on Architectural*

Support for Programming Languages and Operating Systems, pp.177–188 (1991).
25) Yeh, T-Y. and Patt, Y.N.: Alternative Implementation of Two-Level Adaptive Branch Prediction, *Proc. 19th Int. Symp. on Computer Architecture*, pp.124–134 (1992).

(平成 11 年 10 月 29 日受付)
(平成 12 年 12 月 1 日採録)

小林良太郎（正会員）

1995 年名古屋大学工学部電子情報学科卒業．1997 年名古屋大学大学院工学研究科電子情報学専攻博士課程前期課程修了．2000 年同大学院工学研究科電子情報学専攻博士課程後期課程満了．同年名古屋大学大学院工学研究科電子情報学専攻助手．1999 年情報処理学会山下記念研究賞受賞．計算機アーキテクチャの研究に従事．電子情報通信学会会員．



小川　行宏（正会員）

1998 年名古屋大学工学部電子情報学科卒業．2000 年名古屋大学大学院工学研究科電子情報学専攻博士課程前期課程修了．同年岐阜県製品技術研究所入所．

岩田　充晃（正会員）

1996 年名古屋大学工学部電子情報学科卒業．1998 年名古屋大学大学院工学研究科電子情報学専攻博士課程前期課程修了．同年三菱重工業（株）に入社．

安藤　秀樹（正会員）

1959 年生．1981 年大阪大学工学部電子工学科卒業．1983 年大阪大学大学院修士課程修了．京都大学工学博士．1983 年三菱電機（株）LSI 研究所．ISDN 用ディジタル信号処理 LSI，第 5 世代コンピュータ・プロジェクトの推論マシン用プロセッサの設計に従事．1991 年 Stanford 大学客員研究員．1997 年名古屋大学大学院工学研究科電子情報学専攻講師．1998 年同大学助教授．1998 年東京大学大学院理学系研究科助教授併任．1998 年情報処理学会論文賞受賞．計算機アーキテクチャ，コンパイラの研究に従事．

島田　俊夫（正会員）

1968 年東京大学工学部計数工学科卒業．1970 年東京大学大学院修士課程修了．同年電子技術総合研究所入所．1993 年より名古屋大学大学院工学研究科電子情報学専攻教授．人工知能向き言語，LISP マシン，データフロー計算機の研究に従事．最近はマイクロプロセッサのアーキテクチャやチップ内並列処理の研究を行っている．1988 年度市村賞，1994 年度情報処理学会論文賞，1995 年度注目発明賞受賞．工学博士．