計算機アーキテクチャ 125－24
（1997. 8. 21）

On Chip MIMD における大規模投機実行機構

玉造潤史　松本尚　平木敬

大規模集積回路の集積度の上昇により得られる大きなハードウエア資源を活用するため、現在の命令レベルでの投機実行よりもさらに大きな粒度であるループブロックの投機実行が注目を集めている。ループレベルの並列性は命令レベルの並列性の活用よりも大きな資源を必要とするが、よりハイパフォーマンスを獲得することが可能である。我々は、既に On Chip MIMD アーキテクチャをベースとした並列マイクロプロセッサにおいて逐次形式で生成されたバイナリプログラムを実行時に解析し、再構成することによって、重複実行よるループレベルの投機実行を行なうことが可能であることを示した。

本稿では、我々のプログラム再構築による並列化スレッドでのループレベル投機実行の実行形態と他のループレベル、ブロックレベル投機実行との比較を行なう。また、制御構造の依存を解決するハードウエアについても述べる。

Large-scale speculative parallel execution mechanism on On-Chip MIMD

JUNJI TAMATSUKURI , TAKASHI MATSUMOTO and KEI HIRAKI

For exploiting large hardware resources given by the increase of integrated transisters on one VLSI chip, we attract much larger gulanuarity parallelism of the loop-level speculative execution than the one of current instruction- level parallelism (ILP). The loop-level parallelism needs more resources than ILP but is able to accomplish the higher performance. We have already proposed a parallel microprocessor architecture based on On-Chip MIMD. The architecture can execute a current binary program compiled for a single sequential microprocessor and analyze the program on run-time and restruct it for parallel execution. The restructured program can be executed by a duplicate speculative execution. By these parallel execution ,we have showed the ability of the binary compatible parallel microprocessor.

We'll show a comparison of differences among our way of loop-level speculative execution which element processors execute restructed programs and other way such as it forks speculative thread continually like a pipeline or it products a control thread for speculation in this paper. And we show a resoluble way of control structure contained in the most-inner loop.

1. はじめに

近年、VLSIの集積度の上昇を背景としてさらなる高速化を目指した様々なアーキテクチャが提案されている。その中でも On-Chip MIMD をベースとしたマルチプロセッサ[6)5)2)8)] 上で基本ブロックを越えた投機実行を用いた並列実行を目指すものが多数提案されている。これらの多くは投機的な並列スレッドを fork によって生成するもの[5)]、コントロールスレッドを生成し、実行部分を従属スレッドとして実現するもの[8)]、そして基本ブロック毎にパイプライン的に実行するものに大別される。最後に挙げたパイプライン的に実行するものはスケーラビリティが他の方式よりも圧倒的に低くなるので議論から外すこととする。

我々は、既にループの構造を実行時に解析し、再構成(restructuring)する方法[4)9)]を提案している。プログラムの再構成は、現在の逐次実行を行なう形式のプログラムを要素プロセッサで実行する際に発見したループ構造のイテレーション間解析を少量のハードウエアで実現可能なレジスタ解析によって行ない各イテレーションが独立して実行可能なイテレーションの制御構造と計算実行部分とに分離する。多くの場合制御構造の部分は非常に小さい。この分離された制御構造を数回実行を繰り返すこと(duplication)により、必要とするイテレーションのプロセッサ状態を生成することが出来る。この制御結果を作り出すために数回繰り返す重複実行のほとんどは各要素プロセッサの命令レベル並列実行によってイテレーションの実行にオーバーラップし隠蔽される。この性質に着目し再構成と重複実行によって並列スレッドを動的に生成する。したがって、要素プロセッサで実行さ

† 東京大学 大学院 理学系研究科 情報科学専攻
Department of Information Science,Faculty of Science
University of Tokyo

れる並列スレッドは全てのイテレーションを実行する。そのため、本方式で再構成されたスレッドは静的に並列化されたスレッドによる並列実行よりもわずかに実行ステップが増加するが他のスレッドとの明示的な通信やレジスタコピーを必要としない。スレッド間の通信は共有メモリ上で行なわれ、先に示した他の投機実行同様にメモリアクセスは投機的に実行されるため、ループブロック内のプログラム制御構造に対する投機が失敗しない場合には先行して実行可能である。

また、本方式は動作モデルは SPMD 形式のプログラム並列実行と同様のスレッド間の自由なオーバーラップを可能とする実行モデルを採用している。SPMD 形式では、順序性が必要なプログラムの制御では lock や同期を用いる。しかし、我々のループレベルの投機実行では同期は実行の終了が先に実行を完了した他のスレッドと制御とメモリアクセスの実行結果が一致した場合に同期を取ってリタイアする。このような実行モデルで疎粒度の並列実行を基本ブロックを越えたブロックを単位に投機的に行なうアーキテクチャである。また、コンパイラによる静的な投機実行を SPMD 形式で行なう研究[3]も存在する。これらの点が、先に述べた投機的なスレッドの生成法とは全く異なる実行モデルとなっている。

自由にオーバーラップ可能なブロックレベルの投機実行を実現するアーキテクチャにおいて必要となるハードウエアが備えなければならない要件は、次の 3 点である。

(1) 投機実行の開始時にプロセッサ状態を巻き戻せること
投機の失敗が検出された時に、レジスタの内容、メモリの内容を投機実行開始状態に戻せなくてはいけない。

(2) 投機実行中のメモリアクセスを実メモリ上に反映してはいけない
メモリへ投機実行の副作用として、不必要な変更を加えないハードウエアが必要。書き込みを行なってしまう場合には元に書き戻すハードウエアが必要。また、投機的に行なった読み出しの正当性も確認する必要がある。

(3) ブロック内のプログラム構造の動作を投機実行し、正当性を保証できる。
投機実行を基本ブロックを越えて行なう場合に、そのブロックを越えた制御構造をどのように検出し他のスレッドの制御結果と一致しているか判定可能であるようにプログラムの制御機構を構成する必要がある。

これら 3 点のうち (1),(2) に関しては他のブロックレベルの投機実行の実現に用いられている投機実行の条件である。特に、(2) のメモリへの投機実行をサポートするメモリシステムは多数提案[1] されている。しかし、実現法の違いはあるが、投機実行のメモリアクセスを外部メモリに反映しないという機能に関する差はない。

本稿では、まず第一に、他のブロックレベルを越えた投機実行のモデルであるフォーク、制御スレッドを持つ並列実行と我々の再構成と重複実行を用いた方法との実行形態の基本的な違いについて述べる。最も大きな差は我々の方法が動的な並列化を行なうという点であるが、実行形態としてもそれぞれのスレッドが並列に実行しながら自由にオーバーラップ可能であり、投機の失敗による巻き戻し以外の同期による逐次化は行なわれない。また、ループレベル投機実行の本質として実行の待ち合わせは投機実行の失敗時にしか起こらないはずである。この特徴に着目した制御構造の投機実行の実現法を示し、我々の方式においてプログラム制御構造の投機実行の実現法について述べる。

2. Ocha-Pro

我々は動的な並列化プログラム再構成とブロックレベル投機実行が可能な On-Chip の基本アーキテクチャとして Ocha-Pro(On-CHip mimd Architecture PROcessor) というプロトタイプを想定している。

このプロトタイプでは、

- スーパースカラマイクロプロセッサを要素プロセッサとする MIMD アーキテクチャ
- 逐次のバイナリプログラムを実行時に再構成し、重複動作により並列実行可能ななスレッドを実行する
- 並列動作のはループをブロックとした投機実行で実現し、メモリアクセスも投機の影響を外部に及ぼさないバッファ (Speculative Access Buffer) により投機的に実行
- 並列投機実行の実行状態と資源の管理は要素プロセッサに分散して備える投機実行管理機構 (Speculative Execution Manager) で行なう

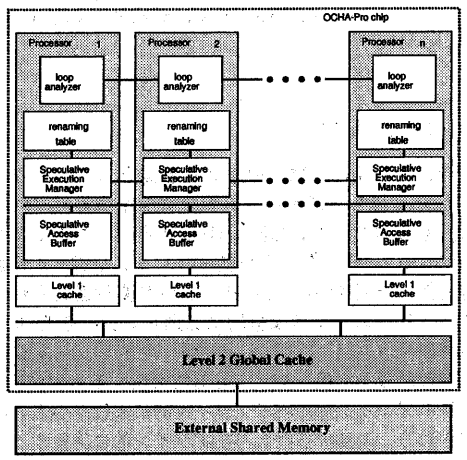


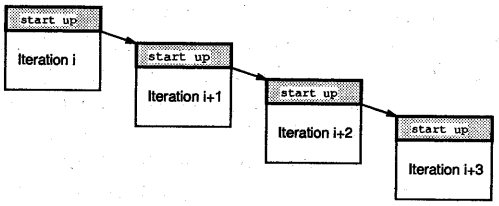
図1 Ocha-Pro の構成図

3. 投機による並列実行モデル

投機実行を実現する方法は多数存在するがSPMDモデル以外のものは、フォークによる投機実行かコントロールスレッドと従属スレッドによるものである。これらの実現する実行モデルは次のような形態である。

フォーク型での投機実行は前イテレーションからの分岐によって開始される。したがって、前イテレーションの実行結果から次の制御構造の投機を行なうことが出来る。また、コントロールスレッドは制御構造を判定するマスタースレッドであるので、投機的に実行される従属スレッドの制御依存を解決している。後者はコントロールスレッドのために並列実行できる要素プロセッサ数が減少する。

◎フォーク型の実行モデル



◎コントロールスレッド、従属スレッド型の実行モデル

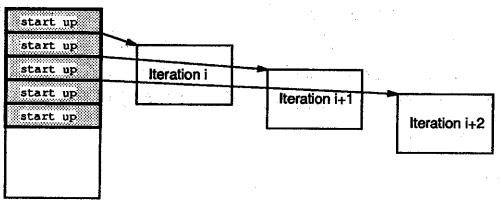


図2 フォーク型、コントロール型の投機実行モデル

しかし、これらの投機実行の形態は次に挙げる問題を持っている。

- イテレーション間の通信が必要。
  イテレーションを跨る依存はイテレーションがフォークする際に引き渡すことで簡単に解決できる。ただし、レジスタ及びプロセッサ状態の引き渡しのための手段を設ける必要がある。

この問題を解決する要素プロセッサ間の通信の実現は通信の手段として、他のプロセッサからのレジスタアクセスを許すことによって実現するか、他のプロセッサのスタック (またはメモリ領域) へアクセスすることでハードウエアによるサポートなしに実現できる。しかし、これらのアクセスは通常のレジスタアクセスよりも大きなレイテンシを必要とする。

一方、我々のプログラム再構成重複実行による方法ではそれぞれのスレッドが完全にオーバーラップして実行が可能となる。各スレッド、イテレーションがオーバーラップして実行可能となるためには、メモリアクセスが投機実行可能である他にループのイテレーションの制御構造 (主にループの実行条件とループボディ内部の条件分岐) の投機実行が可能でなければならない。

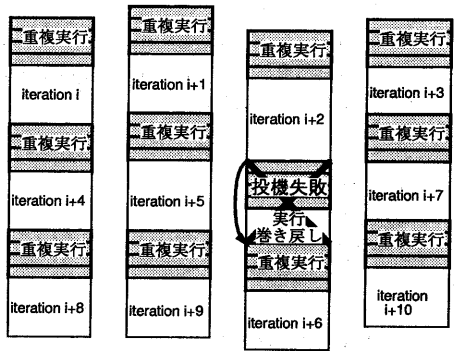


図3 再構成法による実行モデル

それぞれの再構成されたスレッドは全イテレーションで必要となる制御部分を重複実行により必要回数繰り返す。図の例ではプロセッサ数が4であるので、各スレッドは重複実行を3回ずつ行ない、4回目にはループの実行をそのまま実行する。条件分岐などの制御命令は同期信号を受けとることで確認する。各ループイテレーションの最後に同期信号が出力されると他の要素プロセッサとの関係が

- 先行する全てのイテレーションの実行が終了している。
- 先行する全てのイテレーションの投機的メモリアクセスが終了している。
- 先行する全てのイテレーションのプログラムの制御と制御の結果が一致した。

となっているのを確認し、実行状態を投機状態から確定状態へと移し、巻き戻しのために保持しているレジスタ内容と必要なメモリへの書き出しを行なう。この動作は、スーパースカラプロセッサのリタイア動作と同じである。投機実行では I/O の様に出力の逐次性を必要とする場合以外は実行の終了 (リタイア) だけが逐次的に行なわれればよい[7]。

また重複実行により全イテレーションの実行状態のうち計算を必要とするメモリ上のデータ以外の実行資源は全ての要素プロセッサで生成されている。このため並列投機実行を行なっている全てのスレッドは完全に独立して実行可能となっている。つまり、投機実行は失敗の検出を行ない、ループブロックの投機実行終了確認だけが逐次的に行なわれるという、ブロックレベル投機実行での特徴を引き出している。

プログラム再構成重複実行モデルの例として簡単なループを示す。(livermore kernel loop No.24 に制御

構造の依存を持つような変更を加えた)

```
* find location of first minimum in array

for ( k=1 ; k<n ; k++ ) {
    if ( x[k] < x[m] ) m = k;
}
```

このプログラムの逐次形式でコンパイルされたアセンブルコードは図4のようになる。

```
$L5:
        sll $2,$16,2
        addu $2,$2,$5
        l.s $f2,0($4)
        l.s $f0,0($2)
        #nop
        c.lt.s $f2,$f0
        #nop
        addu $4,$4,4
        bc1f $L4
        move $16,$3
$L4:
        addu $3,$3,1
        slt $2,$3,$17
        bne $2,$0,$L5
```

図4 制御構造を持った最内ループ

このループのバイナリプログラムの再構成を行なうと重複実行を行なうプログラムは図5のようになる。プロセッサ状態とプログラムの実行状態を制御する部分とが図4のバイナリプログラム中から抽出される。再構成されたプログラム中には制御構造の遷移全てが含まれている。

```
$L5:    sll $2,$16,2
        addu $2,$2,$5
        bc1f $L4
        move $16,$3
$L4:    addu $3,$3,1
        slt $2,$3,$17
        bne $2,$0,$L5
```

図5 最構成したプログラム

4. 制御構造投機実行の実現

プログラム中の条件分岐のような制御構造の投機を実現するのは命令レベルの投機を活用し、かつ他のスレッドの実行状況とも関係を持つため困難である。しかし、制御構造を投機的に扱うことを可能にすることで次のような性質を得ることが出来る。

- ループを単位としたブロックを独立して実行可能
- ループ内に条件分岐を含んだプログラムを投機実行可能

これらの性質のうち一番目のものはループの並列実行モデルに不可欠であることを示している。これは、一般にループの成立条件は条件分岐によって実現されているためである。先に述べたように分岐命令のようなプロセッサ状態の遷移を投機実行するの困難を避けるためにそのようなループの終了条件を投機的にせずに他の何らかの終了条件を挿入することで回避することも可能である。しかし、二番目は制御構造の投機実行を実現することで条件分岐を含んだ一般的な最内ループの並列実行を可能にすることが出来る。したがって、プロセッサ状態の制御を行なう分岐命令を投機実行可能にすると、投機実行によって高速化できる対象が多くなる。

また、ループの並列実行をフォーク型やコントロールスレッドを用いて実現する場合にはループの終了条件は次のスレッド生成をしないように設定する。それによって、プログラム中の制御命令 (分岐命令) の投機実行という問題から回避しているが、この場合、投機実行を行なうスレッド中に制御構造を含むことは不可能である。しかし、ループを構成する条件分岐の制御構造だけを投機実行とする方法を用いてしまうと発見したループが単純なループでないと投機実行が不可能となる。

ここで議論するループは一般的な単一ループでありしたがって、ループの投機実行の終了条件を投機実行の対象の条件分岐の分岐の方向がループ外に選択されるということとして扱うことにする。この仮定にしたがって、最内ループの含む条件分岐を投機的に実行する機構について述べる。

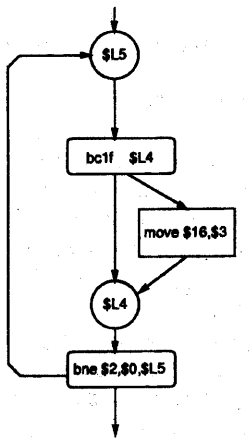


図6 イテレーション間依存の条件分岐を持ったループの例

この分岐命令を含んだループを並列化して実行する。簡単な方法として、実行の状態遷移を行なう制御構造も

メモリの投機実行同様、状態遷移のシーケンスを確認する機構を備えて可能とする。しかも、外部で実行の確認を行なうだけであるので要素プロセッサが複数の分岐命令を同時に行なう機構を備えていても問題ない。

4.1 並列実行としての実装

我々は投機実行管理機構 (speculative execution manager) に制御構造の投機を実現する機構を付加する。投機実行管理機構は各要素プロセッサに分散して装備される。投機実行管理機構は要素プロセッサ内部の資源管理も行なっている。管理しなければならない対象はレジスタセットである。このレジスタセットは投機実行が1段先に進むごとに1セットずつ使用されるためにこのレジスタセットの総数が先行出来る投機実行の総数を決定している。レジスタセットが不足した場合には資源の枯渇のため投機実行のリタイアが行なわれ、資源の回収が行なわれるまで実行は停止される。

この投機実行管理機構に制御構造の状態遷移を記録するキューを付加する。付加するキューは重複実行用のものも用意する必要があるので、総数はレジスタセット × プロセッサ数 だけ必要である。しかし、このキューは高々1ビットのエントリを2個ずつもつだけであるので、大きな資源を必要としない。図7はそのキューを模式的に表したものである。

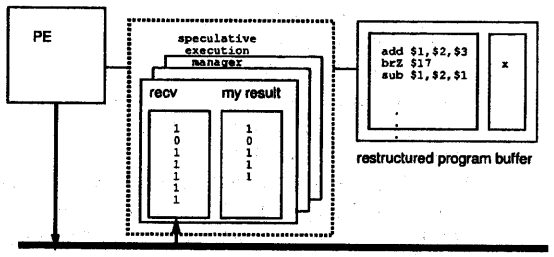


図7 制御構造の投機を管理する機構

キューの各エントリはそれぞれの制御命令 (ここでは分岐命令) が対応している。投機実行を行なっている状態では、要素プロセッサは制御命令を実行する度にその実行の制御を移した方向を my result の方のキューへと記録する。割り当てられたイテレーションの実行時には結果を全要素プロセッサにブロードキャストする。受けとった要素プロセッサは recv のエントリに記録し、投機実行時の結果と比較する。比較した際に、一致していれば投機実行は成功であり、投機実行は継続される。一致していない場合には投機実行は失敗であるので、そのエントリの実行の前から投機実行を再開する。また、イテレーションを投機実行する順序は実行時にしか決定されないので my result と recv が書き込まれる順序づけはなされていない。しかし、制御のパスは必ず同一となるので、先の同期信号を送る際に制御の方向 (1ビット) を付加して送出するだけで良い。同期の待ち合わせ (実際には待たないが) の回数は固定で Elastic Barrier[10] のカウンタによる実装が適している。

したがって、これらのエントリは投機実行の成功 (失敗を検出しないで逐次的にリタイア出来た) まで、保存される。システム上では、メモリ上の依存と制御構造の依存関係が正しく保たれている場合を検出しながら投機実行は行なわれてゆく。

先に示した for ループ (図4) を再構築し性能を測定した。この再構成されたループを複数回実行することで、求めるイテレーションの実行状態を作り出すことが出来る。また全て整数演算と条件分岐ばかりであり、分岐は予測だけに従って実行する。実行時にはループ内の分岐命令とループを構成する分岐命令が実行時の状態遷移を同期情報に付加して送られる。分岐予測には、2ビットの分岐予測を用いた。結果は図8のグラフで、同一回数 (1000回) のループを実行した場合、分岐予測が外れた場合の影響が並列数 (プロセッサ数) の多い場合に大きく出てしまう。しかし、巻き戻しの際に発生するオーバーヘッドは大きくなく、要素プロセッサが4台の場合 2.5 倍の性能を得られた。

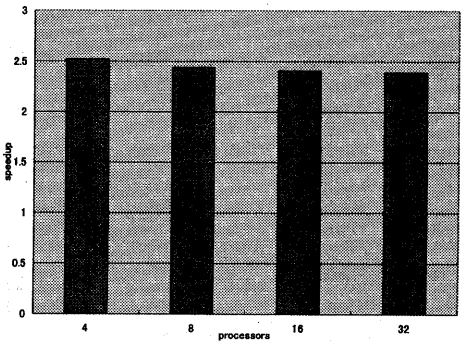


図8 プログラム制御依存のある場合

4.2 並列実行と逐次実行の融合

先の例の結果のように条件分岐の結果が後続のイテレーションに影響するような場合は多くない。しかも、その性能がスケーラブルにならない。これは条件分岐の予測機構の限界を示している。

そのため、条件分岐の初めの予測にだけ従い、失敗したイテレーションの実行時は逐次的に実行するという方法を取ることも出来る。この方法による実装の方がハードウエアの支援も必要とせず、簡単である。

図9のように分岐の最初のパスが、太線で表されているパスであるとする。このパスは逐次実行時に要素プロセッサの branch target buffer のエントリに記録される。解析終了後、重複実行による並列化実行では、再構成されたプログラムによりこの結果が用いられ、このパスが選択されるイテレーションが続く限り並列実行を行なう。分岐がこの branch target buffer を外れた

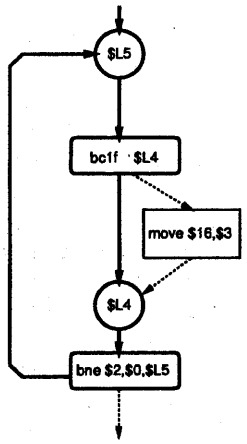


図9 分岐制御のマーキング

場合は後続のイテレーションの実行は全て失敗し、このイテレーションだけが逐次的に処理される。その次のイテレーションからは、また元のように並列実行を再開する。

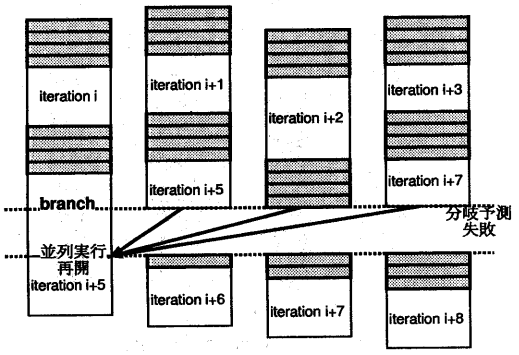


図10 分岐失敗時の動作状況

この実行の様子が、図10である。一時的に逐次化されるものの。先の結果が示すように依存がイテレーション間を渡る場合、大きな並列性を得ることは出来ない。そのため、このような逐次化によって、処理するという方法も先の例と同様の結果を得ることが出来る。

5. ま と め

我々は先に示したプログラム再構成による並列実行における、プログラム制御構造の投機実行の方式について示し、また性能の評価を行なった。これによって、イテレーションを渡るような依存を持った分岐命令を並列実行として実行するのではなく、逐次化による問題解決の方向があることも発見した。これについては今後検討を加えたいところである。また、SPMDモデルに基づいた重複実行による並列化と投機実行とを効率的に組み合わせた実行モデルの詳細について述べ、バイナリコンパチビリティを保持した方法である再構成重複実行による方法が優れている点を示した。本研究ではメンターグラフィックス社とシノプシス社のUniversity Programを用いました。両社に深く感謝します。

参 考 文 献

1) Franklin, M. and Sohi, G. S.: ARB: A Hardware Mechanism for Dynamic Reordering of Memory References, *IEEE Transactions on Computers* (1996).
2) G.Sohi, S.Breach and T.Vijaykumar: Multiscalar Processors, *proceedings of the 22nd Annual International Symposium on Computer Architecture* (1995).
3) J.Oplinger, D.Heine, S.W.Liao, B.A.Nayfeh, M.S.Lam and K.Olukotun: Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor, Technical report, Stanford University (1997).
4) J.Tamatsukuri, T.Matsumoto and K.Hiraki: On-Chip Parallel Architecture for Run-time Loop Restructuring, (in English) 04, University of Tokyo (1997).
5) J.Y.Tsai and P.C.Yew: The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation, *proceedings of Internatinal Conference on Parallel Architectures and Compilation Techniques* (1996).
6) K.Olukotun, B.Nayfeh, L.Hammond, K.Wilson and K.Y.Chang: The Case for a Single-Chip Multiprocessor, *proceedings of the 7th International Symposium on Architectural Support for Parallel Languages and Operating Systems* (1996).
7) R.M.Tomasulo: An efficient algorithm for exploring multiple arithmetic units, *IBM Journal of Research and Development*, Vol. 11, No. 1, pp. 25–33 (1967).
8) 鳥居淳, 近藤真巳, 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, *proceedings of joint symposium on parallel processoin 1997* (1997).
9) 玉造潤史, 松本尚, 平木敬: Loop を並列実行するアーキテクチャ, 情報処理学会研究会報告計算機アーキテクチャ, Vol. 119, No. 11, pp. 61–66 (1996).
10) 松本尚: Elastic Barrier: 一般化されたバリア型同期機構, 情処学会論文誌, Vol. 32, No. 7, pp. 886–896 (1991).