計算機アーキテクチャ 130－22
（1998. 8. 7）

実行時再構成方式テストベット Ocha-Pro の性能評価

玉造潤史　松本尚　平木敬

大きなチップ内資源を用いて構成されるオンチップシステムは今後の高性能アーキテクチャの一つである。そこで我々は、オンチップ MIMD マイクロプロセッサ内に付加したハードウエアによって逐次プログラムを並列化し投機的に実行することによって高速性能を得る MIMD プロセッサの“実行時再構成方式”の提案を行ってきた。実行時再構成方式はオンチップ資源を有効に用いて並列に実行しうるループ構造ブロックを投機実行する。投機実行として実現することで既にコンパイルされたプログラムを変更、再コンパイルする事なく並列に実行する“バイナリコンパチビリティ”を保持している。

本論文では実行時再構成方式のテストベットである OCHA-Pro(On-Chip mimd Architecture Processor) のクロックベースシミュレータを作成し、プログラムの実行効率を測定した。このシミュレータの要素プロセッサの並列度を変化することにより実行時再構成方式の実際の実行性能を測定した。

A performance evaluation of Run-time Restructuring architecture testbed “Ocha-Pro”

JUNJI TAMATSUKURI , TAKASHI MATSUMOTO and KEI HIRAKI

On-Chip system composed by large scale silicon resouces is a candidate of next genaration high performance architecture. We have already proposed “ Run-time restructuring ” MIMD architecture which can execute sequential binary programs on parallel and effectively by the specualtive execution and the hardware parallelization. Run-time restructuring executes a loop construct on parallel using available on-chip resources. The large scale speculative execution realizes non-recompiled,non-translated parallel execution. Therefore Run-time restructureing holds “binary compatibility”.

We use a new clock-base simulater for run-time restructuring testbed “OCHA-Pro(On-Chip mimd Architecture Processor)”. We examined the effects of run-time restructuring parallel execution. Then we measured the ILP & run-time restructuring perforamce according to variable element processor ILP.

1. はじめに

命令レベル並列性の限界が示されて以来、さらなる高速性を求めて様々な研究がなされてきた。これらの研究には既に単一のプログラムからの並列性抽出ではなく、いくつかのプログラムに含まれている並列性を抽出して実行するマルチスレッドアーキテクチャ[8]やマルチスレッドを機能ユニット単位で行う Simultaneous Multi Thread アーキテクチャ[1]などがある。また、分岐予測の精度を上げ、さらに複数の分岐予測を同時に行うことにより、実行ブロックを拡大して機能ユニットの実行効率を上げて実行するアーキテクチャ[16]も存在している。しかし、単一のプログラム中には並列性を抽出し高速に実行しうるループのような構造も存在する。そのようなプログラム構造のブロックを並列に投機実行するアーキテクチャ[7)5)12)2)]も近年多数提案されている。我々は、プログラム構造に並列性を明示的に記述しうるプログラムに対しては並列実行を可能としかつ既に存在するプログラム構造が逐次的に記述されているプログラムからは抽出しうる並列化要素を用いて並列実行する実行時プログラム再構成方式 (Run-time Restructuring) アーキテクチャ[4)6)10)3)14)13)]を開発してきた。

実行時再構成方式では、既存の技術である命令レベル並列性とループのような並列に実行可能なプログラム構造の含む並列性を組み合わせることにより、現在のマイクロプロセッサよりも高速化する方法である。新規のアーキテクチャの性能評価用のシミュレータは、既に命令トレースを用いたシミュレータや実際のプロセッサで実行するようなシミュレータが存在している。今回我々が、クロックベースのシミュレータを用いたのは、命令トレースを用いたシミュレータはメモリアーキテクチャやネットワークといった部分的なシミュレーションをするのに適しているが、複数台数のリオーダバッファの状態やメモリバッファの状態を合わせて知る必要があるよ

† 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

うな場合や、新たなプロセッサアーキテクチャのシミュレーションには適していない。また、現実のプロセッサで実行するシミュレータでは、実行速度も速く、実行時の設定パラメータも現実のものであるので、妥当ではあるが、実行時再構成方式のように次世代アーキテクチャを実験するには適しない。そこで、我々は、クロックベースのシミュレータを用いて現実には現在のテクノロジーでは実現不可能なオンチップシステム全体をシミュレートし各機能の正当性と実行効率の測定を行った。

2. 実行時再構成方式

実行時再構成方式 (Run-time Restructuring) によるオンチッププロセッサ構成法は文献[4)6)10)3)14)13)] 詳述したが、ここではその概要を示す。

実行時再構成方式はオンチップ MIMD 構成のマルチプロセッサで、逐次形式のバイナリプログラムを並列実行可能にするアーキテクチャである。逐次形式のバイナリプログラムの並列化を並列実行するループボディ全体を投機実行するような SSPMD プログラム (Speculative SPMD) を内部で作り出すことにより実現している。ループ構造の解析と並列化を行うハードウエア (Loop Anakyzer) によって生成された SSPMD プログラムは、並列実行時のループイテレーション間に跨って発生する依存のうちレジスタ上に存在するものを重複実行 (Duplicate Execution) という全イテレーション分実行する命令実行によって解決する 。近年のマイクロプロセッサは実行クロック数が数倍程度プロセッサ外部とのコネクションよりも高速である。そのため、実行時にレジスタ上の依存を通信によって解決するよりも逐次プログラムにおいてレジスタ上に存在するような高速性を必要とするデータ間依存は命令実行によって解決することは妥当である。また、メモリ上データに依存して発生する依存に関しては、バイナリ生成時より実行速度の制約を受けるため依存を解決せずに実行してしまうというアプローチを採っている。したがって、全てのメモリアクセスはメモリ投機アクセスバッファ (Speculative Access Buffer) での投機的に実行される。実行後、データ依存となるイテレーション間で RAW ハザードを検出した時に投機実行を失敗させ、実行の正しさを保証する。また、投機的に実行されたループブロックは、1イテレーション実行終了ごとに Elastic Barrier[15)] によってブロックされないリクエストで通知される。この管理は投機実行管理機構 (Speculative Execution Manager) によって行われる。このように SSPMD にはハードウエアにより動的な同期挿入が行われている。これらの動作により、実行時再構成方式には、次のような特徴がある。

(1) バイナリに改変を与えずに実行する (バイナリコンパチビリティを保っている)。

(2) プロセッサ間通信を極力せずに逐次プログラムを並列動作する。

(3) メモリアクセスは投機的に行われるため、実行時のメモリ上データに依存している。(一種の予測と同等である)

(4) Elastic Barrier の採用により、プロセッサ資源の限界まで先行してイテレーションの実行が可能である。また、同期により投機実行がブロックされない

これらの機能を付加したアーキテクチャの概略を図1に示す。

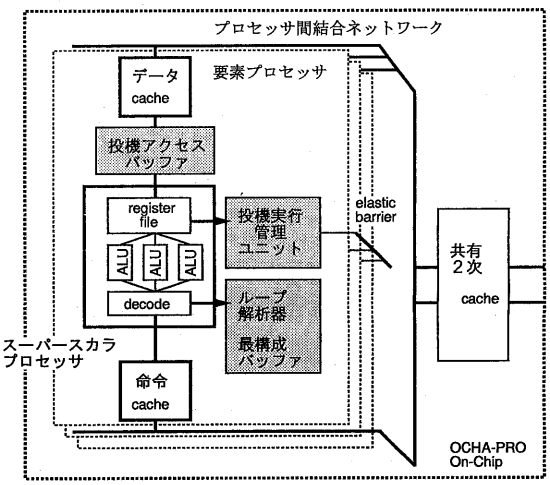

図1 Ocha-Pro のアーキテクチャ

3. Ocha-Pro シミュレータ

今回、新たにクロックベース OCHA-Pro シミュレータを作成した。このシミュレータはスーパースカラ度可変で命令セットは MIPS R1000[9)] と互換のものである。主な仕様は次の通りである。

- プロセッサは可変長リオーダバッファによるスーパースカラ実行を行う
- 命令並列度は可変だが、メモリアクセスに関してはプロセッサ内で逐次に実行される。ストア命令はノンブロッキング動作で実行するが、命令のリタイアは逐次的になる (リタイアは先行ロードを追い越せない)
- 命令のフェッチサイズも可変。ただし、分岐命令だけがアラインメントの制限を持ち、1命令フェッチのうち1つだけ分岐命令を含むことができる。
- 分岐予測は2ビット予測で、256エントリの分岐先ターゲットバッファを持っている。分岐予測失敗によるコストは、リタイア時に復旧するので不定だが、数〜十数クロックかかる。
- 投機実行にはリネーミングレジスタを用い、チェッ

表1　命令実行時間

命令種類	実行時間 (clock)
整数 mult	6(7)
整数 div	34(35)
その他の整数命令	1
浮動小数 mult	2
浮動小数 div	12
その他の浮動小数命令	2

クポイントによる回復は現在のところサポートしない。

- ALU は完全に対称に構成されている。(浮動小数点も現在のところ、乗算除算に関しても対称になっている)

各命令のレイテンシは R10000 に合わせて表 1 のように設定している。

3.1　シミュレータの構成

この要素プロセッサを用いて OCHA-Pro シミュレータを構成した。変更点は次の通りである。

- 命令デコードフェーズに Loop Analyzer と命令再構成バッファを付加する。
- リネーミングレジスタ (フューチャーファイルを含む) は 3 セットありループブロックの投機実行ごとに切り替える。
- 分岐予測は逐次プロセッサと同様に行われるが分岐情報は Loop Analyzer からもアクセスされる。
- 投機実行失敗時には実行の巻き戻し動作を次のように行っている。
  - (1)　現在実行しているイテレーションよりも手前の実行で loop exit をした場合には投機実行を一つ巻き戻して通常の実行を続ける
  - (2)　現在実行しているイテレーションより先の実行が先行して実行を終了した場合には現在の投機実行が終了した時点で通常の実行状態に戻る。

メモリアクセスはロードストアバッファにブロックされない場合、1 次キャッシュは 1 clock、メインメモリアクセスは最低で 10 clock かかるようにコストづけされている。現状では、プロセッサクロックが外部の 2 倍程度のクロックで実行されている。このコストは実際最新のマイクロプロセッサでは 4 倍から 5 倍違うことを考慮するともっと大きなものとする予定である。またこのシミュレータシステムは作成中のため、次のように動作に制限がある。

(1)　投機実行中のキャッシュメモリプロトコルは invalidate で全てのストア操作はライトスルーしている。

(2)　1 次キャッシュから先のネットワークアクセスには制限がない。(メモリアクセスは全てメモリに届くとしている)

これらの制限は今後、改善する予定である。

実行時再構成方式のアーキテクチャで付加される機構

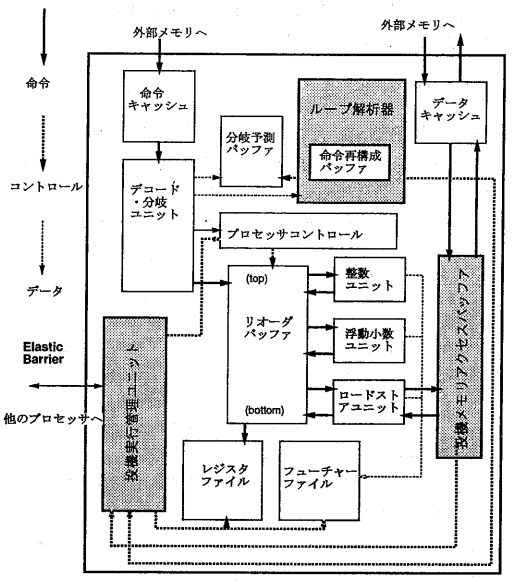

図2　シミュレータの構造

は次のものである。これらは、通常のシングルプロセッサの各ユニットに接続されるが、プロセッサの処理状態に特別なフェーズを追加する事なく、各ユニットの処理と並行して動作が可能である。

ループ解析器　ループの構造検出は、分岐ユニットのプログラムカウンタをアクセスする。ループ構造は分岐予測の結果分岐予測バッファに入れられるアドレスと分岐命令のプログラムカウンタで決定される。命令デコード時に作成されるオペランド情報を履歴情報として内部に持ち、ループ検出後のプログラム再構成に用いる。

命令再構成バッファ　オペランド情報から生成された再構成プログラムを格納する。サイズは、非常に小さく現在は 30 エントリ (命令) である。

投機実行管理ユニット　投機実行の実行の成功、失敗を検出する。他のプロセッサから Elastic Barrier によって伝えられる実行終了リクエストが先行するイテレーション分集まると、投機実行のリタイアを行う。リタイアは、メモリアクセスの実行と保持していた投機実行開始時のレジスタ情報を解放することである。

メモリ投機アクセスバッファ　全てのメモリアクセスは投機実行時にはブロックされる。ロードのみが実行されるが、その履歴は内部に残っている。投機実行失敗終了時には投機実行管理ユニットから失敗を通知されるとそのイテレーションのブロックしたエントリを削除する。

要素プロセッサの変更点はこれだけで、このプロセッ

サを MIMD 構成するにだけで実行再構成方式の オンチップ MIMD は構成される。

4. 性能評価

性能評価で用いたプログラムは現在のところ livermore kernel の次のようなループを含んだプログラムをテストプログラムとして実行した。

```
**************************************************
*    Kernel 1 -- hydro fragment
**************************************************
  DO 1 L = 1,Loop
    DO 1 k = 1,n
1         X(k)= Q + Y(k)*(R*ZX(k+10)
                        + T*ZX(k+11))
```

性能はこの 2 重ループの最内ループを 50 回外部ループを 1000 回 (外部ループの実行回数は固定) で実行したもの、最内ループの実行回数を 5 回にして実行したものについて実行を行った。これらの実験では重複実行のために整数ユニットの個数を増やしている (整数ユニット 3 個, 浮動小数ユニット 1 個) また、要素プロセッサ内の ALU の並列度を変更して実行した。

実行は、実機でバイナリプログラムの実行を測定するのと同じように行われる。測定に用いたバイナリプログラムは gcc-2.6.3 によって生成された通常の MIPS プロセッサでも実行可能なオブジェクトファイルである。ただし、開発環境はお茶の水 1 号[11] や MISC シミュレータと同様のものを用いているためオブジェクトのフォーマットは若干異なっている。(gcc が生成する部分に関しては変更していない) また、この環境では gnu の libc が使用可能で、今回の測定結果には並列化出来ない libc の初期化部分や終了部分の実行も含まれている。

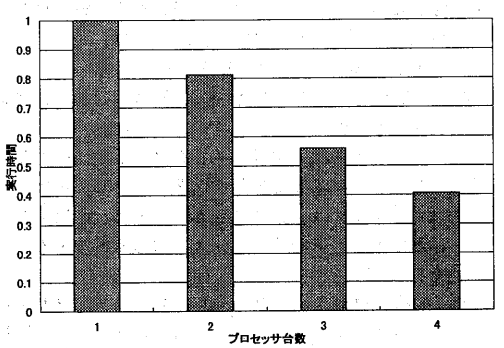

図3 並列性のあるループの実行結果

4.1 基本性能

最内ループを 50 回にして実行した結果が、図 3、表 2 の結果である。最内ループの並列性が大きいため、実行時再構成方式によって実行が大きく高速化している。プログラムの再構成にかかるのはループの実行一サイクル分で、2 イテレーション目から並列実行する。これは、レジスタ依存解析と並行してプログラムの構成が出来るためである。ただし、レジスタ上に冗長な (不必要なレジスタコピーや割り当てを行っている) ようなプログラムでは、2 サイクル目のレジスタ解析で 1 サイクル目の解析結果と異なるので、再構成を行うが、この場合は検出だけで再構成バッファに並行にプログラム再構成を行わない。(現在までに、この検査で再構成出来なかったプログラムはない) また Ocha-Pro シミュレータでは直前に並列実行したループと再構成バッファ中に入って要るプログラムのループが同じプログラムを用いる。ループへの突入は必ず分岐命令で起っているので、現在は Loop Analyzer は実行中のプログラムカウンタの変化を監視しており前回ループアドレスと同じアドレスを実行しようとすると解析を行わずに並列実行に突入する。この並列実行開始動作は各要素プロセッサ間では同期せずに行われる。このような並列化動作を改善することによって細粒度並列実行しなければならないループでも要素プロセッサ 4 台で 2 倍以上高速化する。

表2 並列性のあるループの実行結果

実行台数	1 台	2 台	3 台	4 台
速度向上	1	0.81	0.55	0.40

4.2 規模の小さなループの実行

規模が小さく実行回数の小さなループでは、並列実行からの復旧動作のために実際の有効実行の割合が小さくなってしまう。

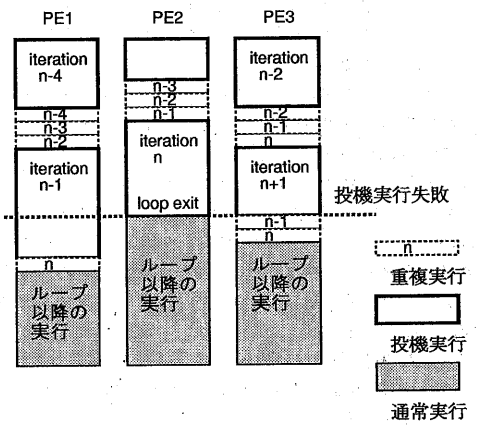

図4 投機実行からの復帰動作

図 4 に示すように 3 台で並列実行を行った場合には 3 台の要素プロセッサのうち、正しく実行を終了するものは 1 台である (図では PE2 である)。そのため、他のプ

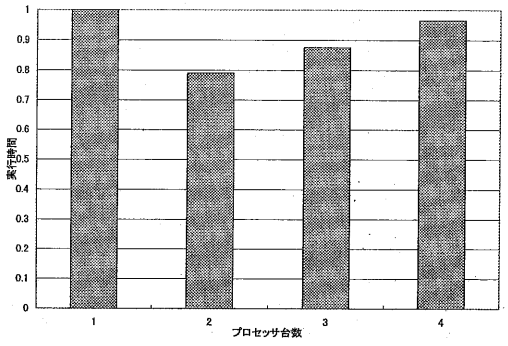

図5　並列実行回数の少ないループの実行結果

表3　並列実行回数の少ないループの実行結果

実行台数	1台	2台	3台	4台
速度向上	1	0.79	0.87	0.96

ロセッサは実行状態を修正する動作を行っている。先行してイテレーションを実行していない PE1 は次のイテレーションの重複動作を実行するだけで次の実行に入れるが、先行して実行しているイテレーションはその実行の開始時 (重複実行の動作開始時) に状態を戻して重複動作からやり直さなければならない。その結果、オーバラップして実行した並列実行の効率はあまり上昇しない。しかし、このような実行時再構成方式にとって不利な条件にも関わらず、並列実行によって速度が低下することはない。この結果が図 5、表 3 である。

4.3　命令レベル並列性との融合

実行時再構成方式による並列実行では重複実行によって並列実行するイテレーション間の依存を解決している。この重複実行のためにループを制御する変数を生成するプログラムサイズが増大している (図 6)

図のように、重複実行によって実行するプログラムはほとんど整数命令であり、かつこれらの命令はレジスタ数分の命令レベル並列性を持っているので、整数ユニットの数を増加することによって実行効率上昇することができる。そもそも、実行時再構成方式ではこのように、命令レベル並列性によって増加する命令による実行時間を隠蔽することを前提としている。並列に実行する整数ユニットと浮動小数ユニットの数を上昇するとこの実行時間は大きく減少する。この減少の割合は実際に作成する場合の指針となる。

この結果が図 7、表 4 である Int2fp1 は整数ユニット数 2、浮動小数ユニット数 1 の場合でそれぞれ整数ユニット数を 3 まで、浮動小数ユニットを 2 まで増加させた結果である。機能ユニットの増加にしたがって実行時間は減少しているが、並列実行時には整数ユニットの増加による減少が多い。また、このプログラムは浮動小数点演算を行うループの実行であるため浮動小数ユニット

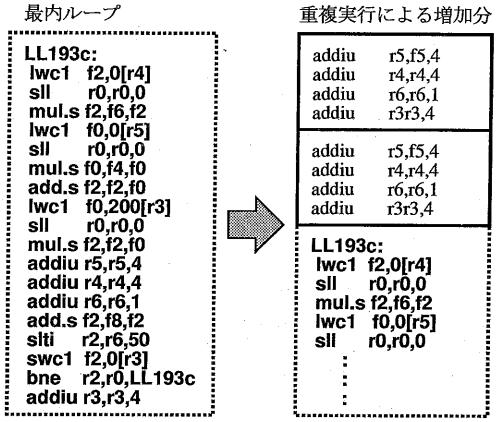

図6　再構成したプログラムの例

の増加により速度向上が期待できるアプリケーションであるが、浮動小数点ユニットの増加による速度向上よりも整数ユニットの増加の方が大きな影響を与えている。

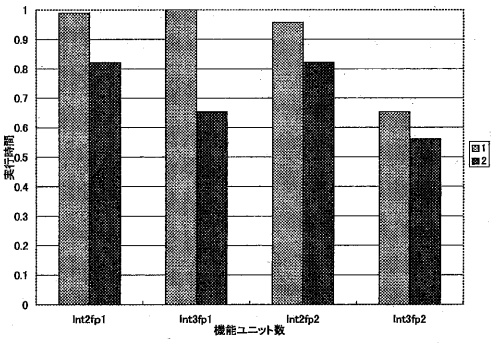

図7　整数ユニット数と浮動小数ユニット数の変化と実行時間

5. ま と め

実行時再構成方式のテストベットである Ocha-Pro のクロックベースシミュレータを作成し、 livermore kernel のループの実行を行った。実行の結果、ある程度以上の並列性の元では十分高速化すると言うことが分かった。また、小さな並列性しか無い場合でも遅くなることはないことも分かった。残念なことに、シミュレータの現状としては、メモリ階層が完全には出来ていない。(オンチップ内蔵 2 次キャッシュとアップデートプロトコルは現在のところシミュレートできない) 今後このシミュレータはキャッシュコヒーレンスプロトコルとして東大プロトコルを採用して階層キャッシュを作成する予定である。また、ベンチマークプログラムとしては

表4 整数ユニット数と浮動小数ユニット数の変化と実行時間

機能ユニット数	整数ユニット2台 浮動小数ユニット1台	整数ユニット3台 浮動小数ユニット1台	整数ユニット2台 浮動小数ユニット2台	整数ユニット3台 浮動小数ユニット2台
速度向上(1プロセッサ)	0.98	1	0.95	0.65
速度向上(2プロセッサ)	0.82	0.65	0.81	0.50

SPEC 全アプリケーションで動作を確認する予定である。

今回実験した方式で用いた方式では、ナイーブにイテレーション一つ一つを要素プロセッサに割り当てて実行している。しかし、実際キャッシュは数ワード単位で構成されているためそのワード数分のエントリが連続して用いられる可能性は大きい。そのため、1次キャッシュヒットを目的として、キャッシュラインサイズずつになるように並列実行するように変更することは可能である。しかし、この実行形態はメモリ投機アクセスがRAW ハザードを引き起こす頻度との問題を起こす可能性がある。ループが並列実行を行う際にメモリアクセスを投機実行しているが、問題となるのはスタックへのアクセスである。プロセッサアーキテクチャではスタックポインタに用いるレジスタが固定されておりこのレジスタの解析をすることでスタック上の依存も解決することができる。今後、全メモリデータを投機実行せずに、コンパイラによるコード生成によって用いられるメモリスタック上の依存に関しても解析を行う方法について研究を行う予定である。

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) D.Tullsen, S.Eggers and H.Levy: Simultaneous Multithreading:Maximizing On-Chip Parallelism, *proceedings of the 22nd Annual International Symposiumon Computer Architecture* (1995).
2) G.Sohi, S.Breach and T.Vijaykumar: Multiscalar Processors, *proceedings of the 22nd Annual International Symposium on Computer Architecture* (1995).
3) J.Tamatsukuri, T.Matsumoto and K.Hiraki: On-Chip Parallel Architecture for Run-time Loop Restructuring, (in English)04, University of Tokyo (1997).
4) J.Tamatsukuri, T.Matsumoto and K.Hiraki: Run-time Restructuring for On-Chip Multiprocessor, *proceedings of the international conffernce on Parallel Distributed Processing Technology and Applications'98* (1998).
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.Hiraki, J.Tamatsukuri and T.Matsumoto: Speculative execution model with duplicate execution, *proceedings of 1998 International Conference on Supercomputing* (1998).
7) 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).
8) M.Fillo, S.W.Keckler, W.J.Dally, N.P.Carter and A.Chang: The M-Machine multicomputer, *In proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture* (1995).
9) Silicon Graphics International: *R10000 Users Manual version 1.0* (1995).
10) 平木敬, 玉造潤史, 松本尚: プロセッサによる実行時ループ再構成方式, *proceedings of the HPCS'97* (1997).
11) 戸塚米太郎, 大津金光, 秋葉智弘, 松本尚, 平木敬: 汎用細粒度並列計算機：お茶の水1号, *proceedings of the Joint Symposium on Parallel Processing'94*, pp. pp73–80 (1994).
12) 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).
13) 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).
14) 玉造潤史, 松本尚, 平木敬: On Chip MIMD における大規模投機実行機構, 情報処理学会研究会報告計算機アーキテクチャ, Vol. 125, No. 24, pp. 139–144 (1997).
15) 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).
16) 安島雄一郎, 中村友洋, 吉瀬謙二, 辻秀典, 田中英彦: スーパスカラ・アーキテクチャのための複数パス実行機構の提案, *proceedings of the Joint Symposium on Parallel Processing'98*, pp. pp23–30 (1998).