スレッド投機実行における命令スケジューリングの評価

田代大輔[†] バルリ ニコ デムス[††]
坂井修一[††] 田中英彦[††]

スレッド投機実行において、スレッド間のデータ依存は投機スレッドの実行を停止させ並列性を低下させる。本稿では非数値計算アプリケーションにおいて、命令スケジューリングによってスレッド間レジスタ依存による並列性の低下を削減する手法を検討し、単純な命令スケジューリングの実装と評価を行った。スケジューリングの効果はアプリケーションによって異なるが、ijpeg において 5%の速度向上を得た。高い性能向上を得るにはより高度なスケジューリングが必要である。

A Quantitative Evaluation of Instruction Scheduling for Speculative Multithreaded Executions

Daisuke Tashiro,[†] Niko Demus Barli,[††] Shuichi Sakai[††]
and Hidehiko Tanaka[††]

In Speculative Multithreading, inter-thread data dependencies stall the execution of speculative threads and degrade the performance that may be achieved. This paper discusses instruction scheduling to relax the inter-thread dependencies. We implement simple instruction scheduling and evaluate its impact for non-numerical applications. Evaluation results show that the effect of scheduling varies among the applications, with ijpeg show the largest gain of 5%. Overall, a more aggressive and intelligent scheduling is required to achieve optimal performance.

1. はじめに

スレッド投機実行は、制御投機、データ投機を行うことでスレッド間の依存制約を緩和し、依存が無いことが明確でないスレッドの並列実行を可能にする。そのためコンパイラによる静的な解析では並列化不可能なプログラムにおいてもマルチスレッド実行を行い並列性の向上を図ることができる。

スレッド投機実行を行い、スレッドレベル並列性を利用するのに有望と思われるアーキテクチャとして、一つのチップ上に複数のプロセッサを集積したチップマルチプロセッサ (CMP) である [1–5]。チップマルチプロセッサでは、プロセッサ間をレイテンシの小さい高速なネットワークで結合することが出来るため、スレッド実行やスレッド間通信のオーバヘッドが小さい。

スレッド投機実行アーキテクチャにおいては、シングルスレッドのプログラムをスレッド投機実行に適応したマルチスレッドプログラムに並列化し、スレッド投機実行の性能を引き出す最適化を行う必要がある。この並列化コンパイラの性能によって、アーキテクチャによって引き出される並列性は大きく左右される。

スレッド間のデータ依存制約は投機スレッドの巻き戻しやストールをひきおこし並列性を大きく制限する。これに対処する手法としては、シングルスレッドプログラムをマルチスレッドに分割する際にデータ依存を考慮し、スレッド間のデータ依存制約が緩和されるようにスレッド分割を行う手法や、命令スケジューリングやコードの変形を行ってスレッド間のデータ依存制約の緩和を図る手法などがあげられる。

そうしたスレッド間データ依存制約を緩和する並列化技術は、数値計算系アプリケーションを対象とするものについてはこれまでの多くの研究により高い成果が得られているが、非数値計算アプリケーションにおいては、複雑な制御構造とスレッド間の依存制約のために十分な成果は得られていない [6–8]。

本稿では、非数値計算アプリケーションにおいてスレッド間に依存をもつ命令を移動させる命令スケジューリングを行って、投機スレッドの停止時間を削減し、スレッドレベルの並列性を引き出す手法について検討を行い、予備的評価として単純な命令スケジューリン

† 東京大学大学院 工学系研究科
Graduate School of Engineering, The University of Tokyo
†† 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

グの実装と評価を行った。

以下、2 章では仮定するスレッド投機実行のモデルについて述べる。3 章ではスレッド間スケジューリングについて述べる。4 章で評価を行い、5 章でまとめを行い今後の展望を述べる。

2. スレッド投機実行モデル

2.1 仮定するアーキテクチャ

まず、本稿で仮定するアーキテクチャモデルについて述べる。我々はスレッド投機実行を行うアーキテクチャとして図 1 に示すようなチップマルチプロセッサを仮定している。このチップマルチプロセッサは 4 つのプロセッシングユニット (PU) によって構成され、個々の PU はアウトオブオーダ実行を行うパイプライン・スーパスカラプロセッサである。各 PU におけるスレッドの実行はスレッド制御ユニットによって集中的に管理される。

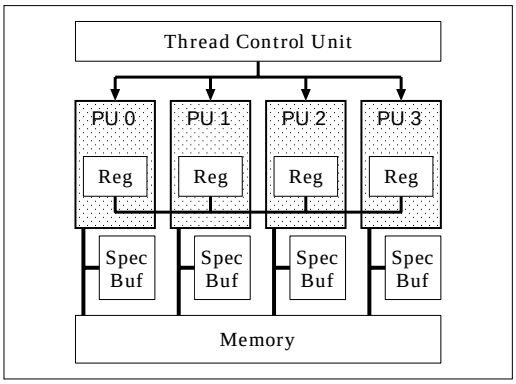

図 1 仮定 CMP アーキテクチャのイメージ

メモリアクセスはデータ投機実行を行い、メモリユニットによって投機スレッドにおけるストア命令のバッファリングやバイオレーションの検出とそれに伴う投機スレッドの破棄が適切に行われる。

各 PU はそれぞれ独立したレジスタセットを持ち、PU 間の高速なネットワークを介してレジスタの通信を行う。このレジスタ通信はメモリを介して PU 間の通信を行うよりも高速である。スレッド間のレジスタ依存はメモリ依存と異なりコンパイラによって全て静的に解析可能であるため、データ投機は行わずにスレッド間で同期を取って通信を行う。コンパイラは必要なレジスタ通信を解析し適切な指示命令をプログラムに挿入する必要がある。

2.2 スレッド分割手法

シングルスレッドのプログラムは、コンパイラによって静的に制御フローグラフの部分グラフであるスレッドに分割される。このスレッドはただひとつの入口をもち、その点がスレッドの開始点となる。また、プログラム中のあるスレッドに含まれる部分が他のスレッドに重複して含まれることはない。スレッドの開始点にはコンパイラによってスレッドの開始命令およびスレッドの情報がコンパイラによって挿入され、プロセッサは動的にスレッド開始命令を解釈し次のスレッド開始命令までを一つの動的なスレッドとして実行する。

シングルスレッドのスレッドへの分割は、文献 [9] で提案された手法により、関数の呼び出しおよび最内周ループのイテレーションの先頭をスレッドの開始点としてスレッドへの分割を行っている。このスレッド分割手法では、サイズの小さなスレッドが多量に生成され、スレッド生成のオーバヘッドやスレッドサイズの不均衡により並列性が低下するため、実行時に連続したサイズの小さなスレッドを連結し、動的にサイズの大きなスレッドとして実行する Dynamic Thread Extension を行っている [10]。

このスレッド分割手法は、プログラムの制御構造およびスレッドのサイズを考慮したもので、プログラムのデータ依存は考慮していない。

2.3 レジスタ通信モデル

先に述べたように、我々はスレッド間のレジスタ依存について、スレッド間でレジスタの通信を行う実行モデルを仮定している。ここでレジスタ通信のモデルを簡単に述べる。

各 PU は、それぞれ独立したレジスタセットをもち、PU 間の高速なネットワークを用いて必要なレジスタの値の送受信を行う。コンパイラはプログラムのレジスタ依存を解析し、スレッド開始命令とともに

- 先行スレッドから値を受信すべきレジスタ
- 後続スレッドに値を送信すべきレジスタ

の情報をプログラムに挿入する。また、送信すべきレジスタの値が確定し、後続スレッドへの送信が可能になる点に send 命令を挿入する。

PU は先行スレッドから受信すべきレジスタの値が先行スレッドを実行している PU から送信されてくるまで、そのレジスタを参照する命令の実行を行わない。また、送信すべきレジスタについて、send 命令によって送信する値が確定すると、可能な限り速やかに後続スレッドを実行している PU にその値を送信する。

本稿で仮定している実行モデルにおいて、先行スレッドからの受信を行う必要があるレジスタは、スレッド開始点において「生きている」すなわちそれ以降において参照される可能性のあるレジスタである。また、後続スレッドに送信を行うべきレジスタはスレッドの全ての出口において生きているレジスタの和集合となる。この場合、先行スレッドが複数の後続スレッド候補を持っている場合、必要ないレジスタが送信されてくる場合があるが、そのような場合にはその値は無視される。

送信すべきレジスタの値が確定する点は、その時点でのレジスタの値が必ず後続スレッドに到達する、す

なわちその点からスレッドの出口に至る全ての実行パスにおいてそのレジスタを再定義する命令が存在しない点である。このような点は、スレッドに含まれる基本ブロックの入り口またはそのレジスタの値を定義する命令の直後のいずれかになる (図 2)。

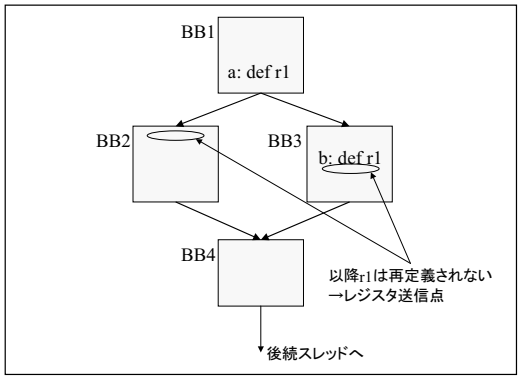

図 2 レジスタの通信確定点

3. スレッド間通信のスケジューリング

3.1 レジスタ依存によるスレッドの停止

プログラム中に存在するデータ依存のうち、ひとつのスレッド内に収まっているデータ依存は、そのスレッドを実行している PU によって従来のプロセッサと同じように適切に処理される。一方データの依存がスレッド間にまたがって存在する場合、それが制約となって投機スレッドの実行を制限し、並列性を低下させる。

スレッド間のレジスタデータ依存が投機スレッドの実行を制限するのは送信されるレジスタの値の確定点 (send 命令) と投機スレッドでその値を参照する命令 (consumer) がそれぞれのスレッドにおいてどちらが先に実行されるかによる。送信が consumer のフェッチよりも早く行われれば、consumer の実行は滞り無く行われるが、送信が consumer のフェッチよりも後に行われた場合、他に実行可能な命令が投機スレッドの命令ウィンドウに存在しなければ同期の待ち合わせによって投機スレッドの実行が停止し、並列性が低下する (図 3)。

こうしたレジスタ依存による性能低下を抑制する解決策として

- スレッド間のレジスタ依存を削減する。
- 送信点と consumer の位置関係を改善し、スレッドの停止を削減する。

のふたつがあげられる。これらを実現するには、スレッド分割を行う際にデータ依存を考慮したスレッド分割を行う方式と、スレッド分割を行った後に命令スケジューリングやコードの変形を行う方式の二つのアプローチがある。前者のアプローチはスレッド投機実行のための並列化コンパイラの多くで行われているが、それだけですべてのスレッド間の依存関係を解決することはできない。そのため後者のアプローチが並列性を得るためには必要になる。本稿では、後者のアプローチについて検討を行う。

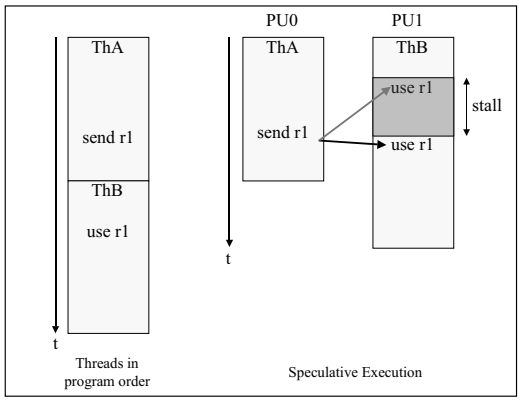

図 3 レジスタ依存によるスレッドの停止

3.2 命令スケジューリング

レジスタ通信の同期を削減するための命令スケジューリングの基本的方針は、図 4 で表わされる。通信が必要なレジスタの値を定義する命令 (producer) とそれが依存している命令をスレッドの上方に移動する。また consumer とそれに依存している命令をスレッドの下方に移動させる。この二つの命令移動を行うことで投機スレッドで consumer が実行可能になる前に producer を実行してレジスタ通信が行われるようにすることができる。ここで、各スレッドを実行する PU がアウトオブオーダ実行を行う場合は、consumer を下方に移動させるスケジューリングは producer の上方への移動に比べ重要性は低いと考える。その理由はアウトオブオーダ実行を行う場合には、必要な値がそろっている命令が他に存在する場合には、その命令が先に実行される。したがって値がそろっていない命令は後に実行され、静的にスケジューリングを行った場合と同様の効果を動的に得ることが期待できるからである。

一方で、複数の命令が実行に必要な値が揃っている場合には、通常はプログラムで記述された順序で実行される。そのため、producer が優先的に実行されるようにあらかじめ静的にスケジューリングを行う必要がある。

命令スケジューリングを行う上での問題は、移動させる命令の選択と命令をどれだけ移動させればよいかの決定を適切に行うことである。ある命令を移動させることでスレッド間の依存が緩和されるか、また緩和されるとしてどれだけ移動させれば良いのかを静的に見積もることは難しい問題である。多くの場合、それ

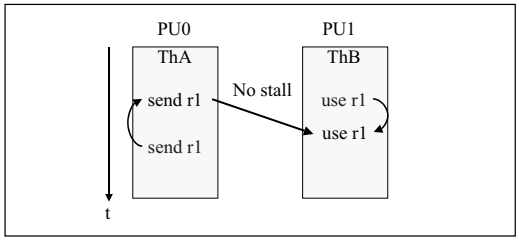

図 4 命令スケジューリング

ぞれの命令についてスレッド開始点からの距離を計算し、それを比較することでスケジューリングを行うことになる。

3.3 コード移動

スレッドは複数の基本ブロックから成り立っている。そのためコード移動によってスレッド間の依存を緩和するには基本ブロックを越えたコード移動を行わなければ、レジスタの送信点を大幅に引き上げることができない。コード移動においては、

- プログラムの意味を変えない。
- 本来発生しないはずの例外を発生させない。

という前提がある。そのため以下の様な制限が生じる。

- プログラムの正しさを保つため、移動の対象命令が依存している命令を追い越して移動することはできない。この依存による制限は RAW、WAW、WAR のいずれの依存関係においても成立する。このような場合、依存している命令をさらに移動させることができれば、コード移動の余地を広げることができる。
- メモリアクセスは、コンパイラによって依存が存在しないことを完全に解析することはできない。そのため依存の有無が明確でないストア命令は他のメモリアクセスと順序を入れ替えることができない。この制限はポインタ解析を行うことで依存関係が無いことが明らかになれば解消される。

さらに、基本ブロックを越えたコード移動を行う場合には、以下のような状況下ではコード移動を行うことができない。

(1) 先行する基本ブロックにおいて、移動させる命令が定義するレジスタが生きている。
(2) 移動する命令がメモリアクセスであり、先行する基本ブロックが分岐命令を含んでいる。

(1) は 2.3 節で示した図 2 の様な場合である。ここで基本ブロック BB3 の $r1$ を定義する命令 b を先行ブロックへ移動させると、BB1 で定義される $r1$ の値が生きているためにレジスタの生存区間が干渉する。このため命令 a を BB1 への移動することができない。この命令を移動させない限り、レジスタ $r1$ の送信点を引き上げることはできないため、スケジューリングが不可能になる。

(2) についてはメモリアクセスが分岐を越えて移動すると例外を発生させる可能性があるためである。分岐命令を越えてメモリアクセスを移動させるには、メモリアクセスと分岐命令の間に制御依存がないことを保証しなければならない。

4. 命令スケジューリングの評価

4.1 命令スケジューリングの実装

命令スケジューリングの評価として、単純な命令スケジューリングを最適化コンパイラ newcc [11] の中間コードに対して実装した。

実装した命令スケジューリングは、以下の手順に従って行われる。

(1) 通信が行われるレジスタを定義する命令 (producer) を解析し、集合 P に入れる。
(2) P に含まれる命令が依存している命令を P に加える。これを P が変化しなくなるまで繰り返す。
(3) P に含まれる命令をそれぞれ可能な限り上方に移動させる。すべての命令が移動不可能になった時点でスケジューリングを止める。

この命令スケジューリングは、producer をできるだけ上方に移動し、可能な限り早く実行することでレジスタ通信の最適化を図る。consumer については全く考慮せず、PU のアウトオブオーダ実行によって動的にスケジューリングと同等の効果が得られることを期待している。

コード移動については、すべての producer とそれが依存している命令を対象とし、命令に優先順位をつけるようなことは行っていない。また移動の余地が残されていても、移動によって追い越す範囲に producer でない命令が存在しない場合はそれ以上の移動を行わない。また、ポインタ解析によるメモリアクセスの解析を行っていないため、すべてのメモリアクセス間には依存の可能性があると仮定している。そのためにストア命令は他のメモリアクセスと順序を入れ替えることができない。そのためにメモリ命令のコード移動が制限される。

今回の実装では基本ブロックを越えたコード移動に強い制限があるため、基本ブロックを越えたコード移動はほとんど行われず、コード移動の大半は基本ブロック内の移動にとどまっている。

4.2 評　　価

実装した命令スケジューリングを SPECCint95 の 8 つのプログラムに対して適用し、シミュレータ上で実行し評価を行った。シミュレータのパラメータを表 1 に示す。PU 間のレジスタ通信は無限個のレジスタを 1 サイクルで通信可能な理想的モデルを仮定している。それぞれのプログラムは 2.2 節で述べた手法によりスレッドに分割されたうえで、命令スケジューリングと send 命令の挿入を行っている。

表 1 シミュレータのパラメータ

No. of PUs	4 Processing Units
PU parameters	6-stage out-of-order superscalar 4 functional units 2 load/store units 4-instruction fetch width 64-entry instruction window 32-entry speculative buffer
Inst. Latency	2 cycles for Load/Store 1 cycle for other instruction
Delays	1 cycle thread start/stop ovh. 1 cycle communication delay 1 cycle restart ovh
Idealized conditions	Perfect memory Perfect next thread prediction

スケジューリングによるサイクルあたりの実行命令数 (IPC) の向上を図 5 に示す。ijpeg がおよそ 5%の向上が得られている他はわずかな性能向上しか得られず、m88ksim, perl, vortex においては逆に IPC が低下した。

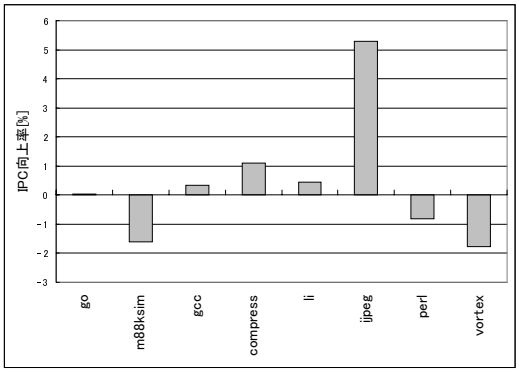

図 5 命令スケジューリングの効果

速度向上が得られなかった理由としては、今回実装したコード移動が先行基本ブロックへのコード移動を行わず、基本ブロック内での移動にとどまっているため、レジスタ送信点を押し上げる効果が小さかったことがもっとも大きな理由である。今回用いたプログラムは基本ブロックのサイズが小さいため、基本ブロック内での移動では、十分なコード移動を行うことができない。また、多くのプログラムで producer がスレッドの出口となる基本ブロックに集中して存在するものが多く見られた、このことは投機スレッドを長時間にわたって停止させるとともに、コード移動の自由度を下げている。

ijpeg が他のプログラムと異なりある程度の IPC 向上を示しているのは、他のプログラムに比べてサイズの大きな基本ブロックを持っているために、並列性を向上させるのに十分なコード移動ができたためと考えられる。

スレッド間の依存制約緩和のための命令スケジューリングが性能に与える影響としては、以下のような点があげられる。

(1) レジスタ通信の同期によるスレッド停止時間の削減
スケジューリングによってスレッド間のレジスタ通信の同期によるスレッド停止時間が削減され、スレッドレベル並列性が向上する。

(2) スレッド実行サイクルの変化
スケジューリングによって個々のスレッドにおいて命令の順序が並び替えられるため、スレッドの実行に要するサイクル数が変化する。スレッド間の依存緩和のためのスケジューリングは一般的な命令スケジューリングと相反する場合があるため、個々のスレッドの実行効率を考慮しない場合はスレッドの実行に要するサイクル数が大きくなる可能性がある。表 2 はスーパスカラプロセッサにおけるスケジューリングによる IPC の変化をシミュレータにより測定した結果である。わずかな変化ではあるが、IPC の向上または低下がみられる。命令スケジューリングが通常のプロセッサにおける実行効率について、アプリケーションによって全く異なる影響を与えることがわかる。

(3) メモリバイオレーションの増加
ロード命令が producer であるような場合、スケジューリングによって投機的ロードが早いタイミングで実行され、メモリバイオレーションを引き起こしやすくなる。また、スケジューリングによって投機スレッドの停止が削減されると、投機的なロードの実行が増加し、メモリバイオレーションを引き起こす可能性が増加する。メモリバイオレーションの発生回数が増加すると、投機スレッドの破棄によって性能が低下する。表 3 はシミュレーションにおけるコミットされたスレッドの数とメモリバイオレーションの発生回数を示したものである。IPC の低下がみられた m88ksim と perl は総スレッド数に対するメモリバイオレーションの発生回数が大きく、スケジューリングによって 4%ほどバイオレーションが増加している。m88ksim と perl における IPC 低下の原因は、このメモリバイオレーションの増加によるものと考えられる。

5. まとめと今後の課題

5.1 ま と め

本稿ではスレッドレベル並列性を低下させるスレッド間レジスタ依存を緩和するための命令スケジューリングの検討および簡単な実装と評価を行った。その結果アプリケーションによるものの、命令スケジューリ

表 2 命令スケジューリングによるスーパースカラプロセッサでの実行効率の変化

	go	m88ksim	gcc	compress95	li	ijpeg	perl	vortex
IPC 変化率	-0.24%	-0.57%	0.03%	0.63%	-0.96%	0.84%	0.03%	0.06%

表 3 コミットしたスレッド数およびバイオレーションの発生回数

Application	コミットした	バイオレーション (スケジューリング無し)		バイオレーション (スケジューリング有り)	
	スレッド数	回数	スレッド数との比	回数	スレッド数との比
go	7256286	864906	83.9 : 1	865125	83.9 : 1
m88ksim	8196554	4984228	1.64 : 1	5203995	1.58 : 1
gcc	7895169	1225520	6.44 : 1	1235852	6.39 : 1
compress95	6360722	918243	6.93 : 1	1076619	5.91 : 1
li	16135597	3137664	5.14 : 1	3053727	5.28 : 1
ijpeg	5685929	254773	22.3 : 1	685496	8.29 : 1
perl	6475028	2676905	2.42 : 1	2794160	2.32 : 1
vortex	5255189	525967	9.99 : 1	567377	9.26 : 1

ングによって性能向上が得られることを確認した。

5.2 今後の課題

今後の課題はより良い命令スケジューリング手法の検討と実装を行うことである。今回実行した命令スケジューリングは全ての producer を可能な限り移動させるというもので、移動させる命令の選択や、効果の予測を全く行っていない。また、非数値計算アプリケーションでは、先に述べたコード移動を制限する要因が多く存在する。そのために、今回用いた個々の命令を移動させることを繰り返すことで、レジスタの送信点を引き上げる手法は、コード移動が厳しく制限されてしまうので、大きな効果を引き出すことは難しい。そこで、非数値計算アプリケーションに適応した命令スケジューリング手法を構築する必要がある。

もうひとつの課題として、Dynamic Thread Extension(DTE) への対応がある。DTE によって複数のスレッドが連結され一つのスレッドとして実行されると、DTE で結合されたスレッドとその後続スレッドとの間のレジスタ通信は結合されたスレッドの終わりの方でしか行われないため、並列性が低下する。この問題を解決するようなレジスタ通信の機構や、DTE を考慮した命令スケジューリングを検討する必要がある。

6. 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)(2) 50291290 の支援により行った。

参 考 文 献

1) Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 414–425, June 22–24, 1995.

2) Lance Hammond, Mark Willey, and Kunle Olukotun, *Data Speculation Support for a Chip Multiprocessor*, Proc. 8th ASPLOS, pp. 58-69, San Jose CA, 1998

3) 小林、岩田、安藤、島田. 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、並列処理シンポジウム JSPP'98、pp.87-94(1998).

4) 鳥居、近藤、本村、西、小長谷. On Chip Multiprocessor 指向 制御並列アーキテクチャMUSCAT の提案、並列処理シンポジウム JSPP'97、pp.229-236(1997).

5) V. Krishnan, J. Torellas, *A Chip-Multiprocessor Architecture with Speculative Multithreading*, IEEE Transactions on Computers, Vol. 48, No. 9, Sept 1999

6) T.N. Vijaykumar and G.S. Sohi, *Task Selection for a Multiscalar Processor*, Proc. 31st MICRO, Nov-Dec 1998

7) T. N. Vijaykumar. Compiling for the Multiscalar Architecture. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, Jan. 1998.

8) Kunle Olukotun, Lance Hammond, and Mark Willey. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. Proceedings of the 1999 ACM International Conference on Supercomputing, Rhodes, Greece, June 1999.

9) Niko D. Barli, Hiroshi Mine, Shuichi Sakai, and Hidehiko Tanaka, *A Thread Partitioning Algorithm using Structural Analysis*, 情報処理学会 計算機アーキテクチャ研究会, ARC-2000-139 Vol. 2000, No. 24, pp. 37-42, Aug 2000

10) Niko D. Barli, Tashiro Daisuke, Shuichi Sakai, and Hidehiko Tanaka, *Dynamic Thread Extension for Speculative Multithreading Architectures*, 情報処理学会 計算機アーキテクチャ研究会, ARC-2001-144 Vol. 2001, Jul 2001

11) 飯塚 大介 他. C コンパイラによるループ最適化の検討. 99-HPC-77, pp. 65-70, 1999.