スレッド投機実行におけるエッジに着目したスレッド分割手法

田 代 大 輔[†]　バルリ ニコ デムス[†]
坂 井 修 一[†]　田 中 英 彦[†]

スレッド投機実行技術は、チップマルチプロセッサにおいて逐次プログラムからスレッドレベル並列性を抽出し、性能向上を図ることができるが、そのためには適切なスレッド分割を行う必要がある。本稿では、性能を低下させる要因となる微小スレッドの生成を抑制し、十分な粒度のスレッドを生成するために、複数スレッドによる CFG ノードの共有を可能とするスレッドモデルが必要であることを示し、スレッド分割手法として、CFG のエッジ上にスレッド境界を置くことで不要なスレッド分割を回避し、ノード共有を実現する手法を提案し、評価を行う。その結果、提案手法によってスレッドサイズの拡大と速度向上が得ることができた。

An Edge-Based Thread Partitioning Method for Speculative Multithreading

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

Speculative multithreading increases performance of a single thread program on a Chip Multiprocessor by exploiting thread-level parallelism. However, a appropriate thread partitioning is required to achieve optimal performance. In this paper, we present that a node-shareable thread model is required to reduce small threads that leads to performance loss and generate threads with sufficient granularity, and propose an edge-based thread partitioning method that reduces needless thread boundaries by putting thread boundaries on edges of CFG. Preliminary evaluation shows that proposed thread partitioning method achieves enlarge the size of threads and performance improvement.

1. はじめに

一つのチップ上に複数のプロセッサコアを集積したチップマルチプロセッサ（CMP）は、並列化されたマルチスレッド、マルチプロセスのアプリケーションが持つスレッドレベル並列性（TLP）を引き出すことができ、将来のプロセッサアーキテクチャとして期待されている。

しかしながら、CMP がその性能を発揮するのは並列化されたアプリケーションであり、現在、また将来においても多数用いられる逐次のアプリケーションにおいては個々のプロセッサコアの能力しか引き出すことができず、CMP の有効性を高めるには、そうした逐次プログラムにおいても CMP の利点である TLP の抽出を可能とし、高い性能を引き出す必要がある。

逐次のプログラムから TLP を抽出する手法として、スレッド投機実行と呼ばれる技術がある。スレッド投機実行は逐次のプログラムをスレッドに分割し、それらを投機的に並列実行することで TLP を抽出する。投機的に並列実行を行うことでスレッド間の制御依存、データ依存の制約が緩和されるため、並列化を妨げていた依存関係の有無があいまいなプログラムにおいても TLP を抽出することが可能になる。CMP は、プロセッサコア間の距離が近く、スレッドの実行管理やスレッド間の通信のオーバヘッドを小さくすることができるため、スレッド投機実行を行うのに適したアーキテクチャであり、様々な研究が行われている [1–3, 6, 8, 9]。

スレッド投機実行を行うためには、まず逐次プログラムを効率よく投機実行できるようにスレッドへの分割が適切に行われる必要があり、スレッド分割手法がスレッド投機実行の性能を大きく左右する。

我々は、非数値計算プログラムをプログラム全体にわたってスレッドに分割する手法を提案している [4]。しかしながら、その手法にはスレッド投機実行に適さない微小なスレッドが多数生成されてしまうという問題点があり、この微小なスレッドの存在により投機実行の性能が低下してしまう。本稿では、この微小なスレッドが生成される問題を解決するため、十分なスレッドサイズを確保しうるスレッド分割手法を提案し、そ

† 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

の評価を行う。

以下、2 章では本稿で想定するスレッド投機実行モデル及びそれをサポートする CMP アーキテクチャについて述べる。3 章では従来のスレッド分割手法及びその問題点について述べる。4 章で本稿で提案するスレッド分割手法について述べ、5 章で評価を行い、6 章でまとめる。

2. スレッド投機実行モデル

まず、本稿で想定するスレッド投機実行モデル、および投機実行を行う CMP アーキテクチャについて述べる。

図 1 に本稿で想定するスレッド投機実行をサポートする CMP アーキテクチャの構成を示す。この CMP は 4 つのプロセッシングユニット (PU) で構成され、個々の PU はアウトオブオーダ実行を行うパイプライン・スーパースカラプロセッサである。各 PU は、それぞれが独立したレジスタファイルを持つ。スレッド投機実行を行う場合は、PU 間に設ける遅延の小さいネットワークを用い、レジスタの値の通信を PU 間で行うことでスレッド間のレジスタを介したデータ依存を解決する。また、メモリについては 1 次キャッシュは各 PU で独立であり、2 次キャッシュを共有する。また、スレッド投機実行においてスレッドの実行管理を集中的に処理する Thread Control Unit（TCU）を持つ。TCU はスレッド投機実行において、投機スレッドの実行を管理し、割り当て可能な PU が存在すると次に実行されるべきスレッドの予測を動的に行い、ラウンドロビン方式で投機スレッドとして割り当てて実行する。また、制御投機、メモリ投機の失敗時に必要なスレッドの破棄を行う。

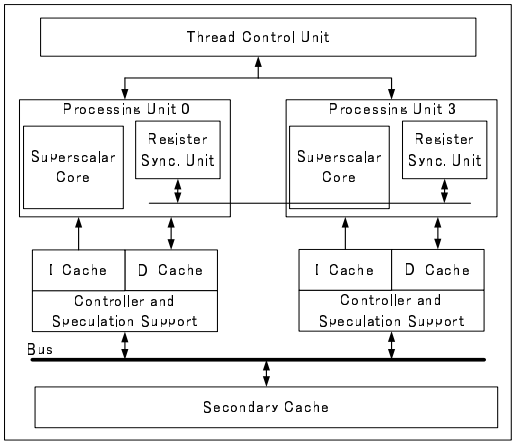

図 1 想定する CMP アーキテクチャ

スレッド投機実行を行う場合、まず逐次のプログラムがコンパイラによって静的にスレッドに分割される。このとき、生成される実行コードにはスレッド投機実行の制御に必要な命令が挿入される。また、スレッド間には制御依存、データ依存が存在することが許されるが、依存関係は常に先行するスレッドから後続するスレッドへの一方向のみ存在することが許される。これは依存の方向を限定することで、スレッドの実行管理と依存関係の解決を簡単にするためである。

投機実行の制御命令としては、スレッドの開始点を表すスレッド開始命令と、スレッド間のレジスタ通信の指示命令、スレッドの制御投機を支援するための命令等がある。スレッド分割の手法については後に詳しく述べる。PU は実行時にスレッド投機実行に関する命令を解釈し、スレッド開始命令から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を行う。

スレッド間に存在するレジスタを介したデータ依存は、コンパイラによって静的に解析され、実行コードに挿入されるレジスタ通信命令に従って PU 間でレジスタ値の同期、通信を行い依存関係を解決する。一方メモリを介したデータ依存は静的な依存解析が困難であり、データ投機実行を行う。メモリアクセスの投機実行を行うには投機状態の保持と投機失敗の検出及び巻き戻しを行う機構が必要である。我々は 1 次キャッシュのコヒーレンスプロトコルを拡張し、1 次キャッシュ上でメモリ投機を処理する [7]。

3. 従来のスレッド分割手法

スレッド投機実行においては、スレッドの分割手法がその性能を大きく左右する。その要因としては、(1) スレッドの粒度、(2) スレッド間の制御依存、(3) スレッド間のデータ依存、の 3 つが上げられる。本稿では、スレッド分割手法を特にスレッドの粒度の点で論じる。

一般的に、並列処理を行う場合には、並列に実行されるタスクの粒度が適切かつ揃った大きさを持つことが望ましい。スレッド投機実行においても、生成されるスレッドは十分な大きさを持ち、並列に実行されるスレッドと同程度の大きさを持っていることが望まれる。

我々は、Structural Analysis を用いてプログラムの制御フローグラフを階層的に解析し、それに基づいてスレッド分割を行う手法を提案している [4]。この手法では、スレッドを制御フローグラフ（CFG）の部分グラフ、つまりノードの集合として定義する。各スレッドはただ一つの開始点と 1 つ以上の終了点を持ち、かつ CFG の各ノードは唯一つのスレッドにのみ属するものとしている。そして、スレッド分割においては逐次プログラムの CFG の階層構造を解析し、そこから経験則によってスレッドの候補となるノード集合を選び出す。ここでは、最内周ループ及び関数呼び出しをスレッドとして選ぶ経験則を用いている。その

後、スレッドの定義に従い残った CFG のノードを集めてスレッドを生成することで、プログラム全体にわたってスレッド分割を行う。

しかし、この分割手法では、その手法からくる制約により、適切でない箇所においてスレッドの分割が行われ、きわめて微小なスレッドが多く生成される。こうした微小なスレッドは、スレッド投機実行の性能を低下させる。

微小なスレッドが性能を低下させる要因としては、次の 2 つがあげられる。

(1) スレッドが小さくなることで、スレッド開始、終了のオーバヘッドが相対的に大きくなる。また、スレッド開始時には割り当てられた PU のパイプラインにはまだ命令が入っていない状態になるため、実行が開始してから実際に命令がフェッチされ、パイプラインを流れて実行されるまでの時間がスレッド開始時のオーバヘッドに加わる。この相対的なオーバヘッドの増加は、PU のパイプライン段数が深く、あるいは PU が引き出せる命令レベル並列性が大きくなるほど顕著に現れる。

(2) 微小なスレッドの前後のスレッドとの間で、粒度の不均衡が生じ、PU が有効に利用できなくなる。我々の想定する実行モデルでは、先行するスレッドが全て完了するまで投機スレッドは完了せず、割り当てられた PU を開放しないため、粒度の不均衡があると無駄に PU が占有されてしまう。

従来のスレッド分割手法において、微小なスレッドが生成される要因は、スレッド分割において、スレッドの入り口を唯一つに限定し、CFG の各ノードを複数のスレッドによって共有することを許していないことにある。複数のスレッドがノードを共有することができない場合、図 2 のように、複数のスレッドが合流する点が存在すると、ノード C,D をスレッド 1,2 で共有できないため、C,D からなる新たなスレッド 3 を生成しなければならなくなる。このように、適切でない箇所においてスレッドの分割が行われるため、スレッドのサイズを十分に確保することができないばかりでなく、非数値計算プログラムにおいては、こうした不必要に分割されるスレッドの多くが高々数命令のきわめて微小な、スレッド投機実行に適さないものとなってしまう。

例えば、if〜then〜else〜構文において、then,else の一方のパスでのみ関数呼び出しが行われるような場合、関数呼び出しのために一方のパスでスレッドが分割され、もう一方のパスとの合流点で不必要なスレッド分割が再度行われてしまうことになる。

この微小なスレッドの問題を解決するために、我々は微小なスレッドを動的に連結し、同一の PU 上で連続して実行することで等価的にスレッドサイズを拡大する手法である Dynamic Thread Extension (DTE) を提案している [5]。DTE はスレッドサイズの拡大と

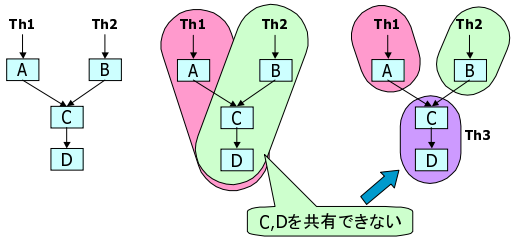

図 2 合流点におけるスレッドの分割

いう点では効果的であるが、実際の管理機構をどのようにするかという点や、連結される後続スレッド全てを予測する必要があるなど課題も多い。

また、Vijaykumar は Multiscalar において、合流によってスレッドが分割されることを避けるため、合流点以降のコードを複製し合流を排除することで余計なスレッド分割を回避している [6]。しかし、コード複製による手法はコードサイズの肥大化を招くため、その効果には限界がある。別のアプローチとして、投機スレッドを fork し、非投機スレッドで join する形のスレッドモデルを用いることでもこの問題を解決できる [9]。だがこの場合はスレッドの実行管理が複雑化してしまうという問題があるため、そうしたスレッドモデルは適切でないと考えられる。

4. エッジに着目したスレッド分割手法の提案

4.1 ノード共有可能なスレッドモデル

非数値計算プログラムにおいてスレッド投機実行を行うには、ノードの共有が可能なスレッドモデルならびにスレッド分割手法が有効であると考えられる。ここで、第 2 節で述べた我々の想定する CMP アーキテクチャ及びスレッド投機実行モデルは、複数のスレッドがあるノードの実行コードを共有することが可能なモデルになっており、ノードの共有を認めないという制約はスレッド分割手法によってのみ存在している。従って、コンパイラのスレッド分割手法を変更することで、複数のスレッドによるノード共有を実現することができる。

逐次プログラムがスレッドに分割される際に、コンパイラによって静的に行われるスレッド投機実行に関する命令の挿入は、挿入する点から到達する全てのスレッド境界（スレッド終端）までの間に存在する命令列及び、後続するスレッド候補の情報に従って行われる。言い換えれば、どのようなパスを経てその点に至ったかには全く依存しない。従って、CFG のあるノードにおいて、そこから到達するスレッドの終端が全て同一であるようなスレッド群は、そのノードを共有することが可能であり、そのノードに挿入されたスレッド投機実行命令は、そのノードを共有するどのスレッドに対しても全く同一の命令が挿入され、正しく

動作することが保障される。例えば、図 2 のような CFG において、ノード C に挿入されるレジスタ通信の指示命令は、ノード C,D に存在する命令列と、後続スレッド候補でアクセスされるレジスタに従って決定され、スレッド 1（A,C,D）においても、スレッド 2（B,C,D）においても同一になる。

逆に言えば、投機実行を制御する命令の挿入が、その命令に至る実行パスに依存して決定されるようなスレッド実行モデルをとる場合、つまり図 2 において、ノード C,D にスレッド 1 とスレッド 2 で異なる命令を挿入しなければならないような場合には、そうした命令が挿入されるノードを複数のスレッドで共有することが不可能になる。

4.2 ノードを共有させるスレッド分割手法

従来の静的なスレッド分割手法 [4,6,9] は、スレッドをただ一つの入り口を持つノード集合としてとらえ、スレッドの開始点（スレッド境界）をノードの先頭に置くことでスレッド分割を行っていた。そのため、そのノードに入るすべての実行パスにおいて新たなスレッドが開始されてしまい、不必要なスレッドの分割が行われている。

そこで、スレッドの境界をノードにではなく、CFG のエッジ上に配置することで、特定の実行パスにおいてのみ新たなスレッドが開始されるようにスレッドの分割手法を変更する。これにより、図 3 のように特定のエッジを経てノードに入る場合のみ、そのノードがスレッドの開始点となるようにすることができ、複数のスレッドが CFG の各ノードを共有するようなスレッド分割を行うことができる。

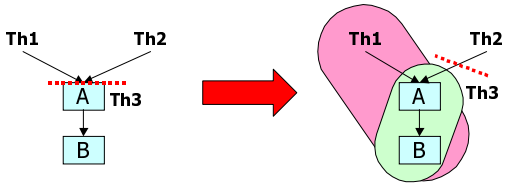

図 3 エッジベースのスレッド分割

エッジベースのスレッド分割およびコード生成は、以下の手順で行われる。

(1) 前処理として、関数呼び出し命令で基本ブロックを分割する。これは、関数呼び出しがノードの末端に位置するようにするためである。
(2) CFG を構築し、スレッドの境界とするエッジを選択する。このとき、任意の選択アルゴリズムを用いることができ、このアルゴリズムの選択がスレッド分割の性能を左右することになる。
(3) 選択したスレッド境界に基づいて、スレッド間のレジスタ依存等の解析を行う。
(4) スレッド境界として選択したエッジにノードを新たに挿入し、そのノードにスレッドのヘッダ（スレッド開始命令及びスレッド情報）を挿入する。

4.3 スレッド境界の選択アルゴリズム

提案するスレッド分割手法では、スレッド境界とする CFG のエッジを選択するアルゴリズムが重要となる。本稿では、以下の 2 つの選択アルゴリズムを実装し、評価を行った。

- ループの戻り辺と関数呼び出し、関数からの復帰をスレッド境界として選択するアルゴリズム（LOOP-FUNC）
- 従来手法でスレッド候補を選択し、微小なスレッドを連結するアルゴリズム（SA+）

本稿で実装した 2 つのアルゴリズムは、プログラムの制御構造だけを見てスレッド分割を行うもので、スレッド間の制御依存関係、データ依存関係については全く考慮していない。

4.3.1 LOOP-FUNC アルゴリズム

LOOP-FUNC アルゴリズムは、ループの戻り辺と関数の呼び出し、関数から復帰する点でスレッドを分割するアルゴリズムで、ループレベル及び関数レベルの並列性を抽出しようとする分割手法 [1,8] と同様のものである。

4.3.2 SA+アルゴリズム

SA+アルゴリズムは、まず [4] で提案した従来のスレッド分割手法を用いてスレッド分割を行い、暫定のスレッド候補を得て、スレッド間を結ぶ全てのエッジをスレッド境界として選択する。その上で、各スレッドの推定サイズが閾値以下のものについて、その後続スレッドへのエッジをスレッド境界から除去する。これにより、閾値以下のスレッドはその後続スレッドと連結され、従来手法で行われていた不必要なスレッド分割を取り除く。スレッドのサイズの推定は、そのスレッドの全ての実行パスの命令数を単純に平均することで行う。

5. 評　　価

提案したスレッド分割手法を SPECCint95 の 8 つのベンチマークに対して適用し、トレースベースの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。各ベンチマークの入力は実行命令数が 1〜3 億命令となるように調節した。

スレッド分割手法には、従来の Structural Analysis に基づくノード共有を行わない手法（SA）、関数呼び出しとループの戻りエッジのみをスレッドの境界として分割を行う手法 (LOOP-FUNC アルゴリズム)、従来手法で一度スレッド分割を行い、15 命令以下のスレッドを連結する手法 (SA+アルゴリズム、閾値 15)、の 3 つの手法を用いた。

まず、図 4 に実行時の平均スレッドサイズを、図 5 に go におけるスレッドサイズの分布を横軸の各数値

表 1 シミュレータのパラメータ

PU 数	4
パイプライン段数	10 段
機能ユニット	ALU × 4、 Load/Store × 2
命令フェッチ幅	4 命令
命令発行幅	4 命令
命令ウィンドウ	20 エントリ
リオーダバッファ	64 エントリ
命令キャッシュ	32KB (2048 エントリ) 2 ポート、 アクセスレイテンシ 2 サイクル
1 次データキャッシュ	32KB (512 エントリ) 2 ポート、 アクセスレイテンシ 2 サイクル
2 次データキャッシュ	4 ポート、 アクセスレイテンシ 6 サイクル 常にヒット
スレッド内分岐予測	BTB(1024 エントリ、連想度 2) Bimodal Predictor(4096 エントリ)
PU 間レジスタ通信	通信テイテンシ 1 サイクル 1 サイクルで 1 つのレジスタ値を転送
スレッド予測機構	理想化（常にヒット）
スレッド開始・終了 オーバヘッド	1 サイクル

n に対して、スレッドサイズが $n-10$〜n 命令であったスレッドの割合で示す。8 つのベンチマークのうち、compress95 を除き全てのベンチマークで平均スレッドサイズが拡大した。また、スレッドサイズの分布を見ると、従来手法 (SA) で多く存在していたサイズが 10 命令以下の微小なスレッドが大きく減少しており、提案したスレッド分割手法によって、性能低下の要因となる微小なスレッドを大幅に削減できることがわかる。

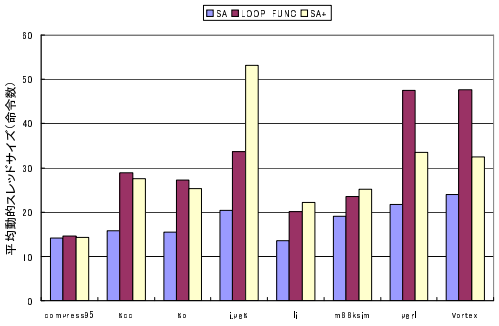

図 4 平均動的スレッドサイズの増加

次に、図 6 に実行されたスレッド数を示す。提案した分割手法によって不要なスレッドの分割が抑制されたことにより、compress95 を除き実行スレッド数が 30〜70%減少した。これは全体の実行時間に対するスレッドの開始、終了のオーバヘッドの削減につながり、性能の向上につながる。

図 7 に各ベンチマークの従来手法 (SA) を 1 とした相対実行サイクルを示す。提案したスレッド分割手法

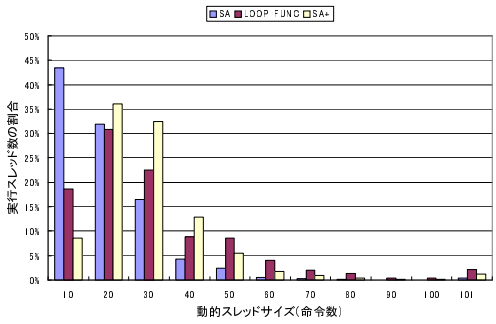

図 5 実行スレッド数の分布（go）

のうち、SA+アルゴリズムによる分割を行うことで、ijpeg で 15%、gcc,go,li,m88ksim 等で 8〜10%の性能向上が得られた。

ijpeg で大きな性能向上が得られた理由としては、ループのイテレーションが閾値である 15 命令以下のループにおいて、各イテレーションをスレッドとして並列に実行されていたものが、全てのイテレーションが連結されループ全体が一つのスレッドとして扱われるようになったことがあると考えられる。これより、最内周のループにおいて、イテレーションの小さな、かつループ回数の少ないようなループについてはループ全体を一つのスレッドとし、より上位の階層で TLP を抽出することが有効であると考えられる。

一方で、ループ、関数に着目してスレッド分割を行った場合は、微小なスレッドが削減できているにもかかわらず性能向上は小さく、いくつかのベンチマークではかえって性能が低下している。

これは、従来手法で抽出できていたループ、関数以外の部分におけるスレッドレベル並列性が抽出できなくなったことが一因と見られる。また、図 5 を見ると、LOOP-FUNC による分割は、SA+による分割に比べ、スレッドサイズの分散が大きくなっており、スレッドサイズにばらつきがある。この点や、前述したループ全体を一つのスレッドとするような分割が行われないといった点なども性能向上が小さい要因となっていると見られる。

これらのことから、今回用いたベンチマークにおいては、単純にループ及び関数のみに着目してスレッド分割を行うだけではスレッドレベルの並列性を引き出すことは難しく、それ以外の部分からも TLP を抽出するような分割手法が必要であるといえる。

6. おわりに

本稿では、スレッドレベル投機実行を行う場合に性能低下の要因となる極めて微小なスレッドの生成を抑制するスレッド分割手法を提案し、評価を行い提案手法によってスレッドサイズが拡大できること、ベンチ

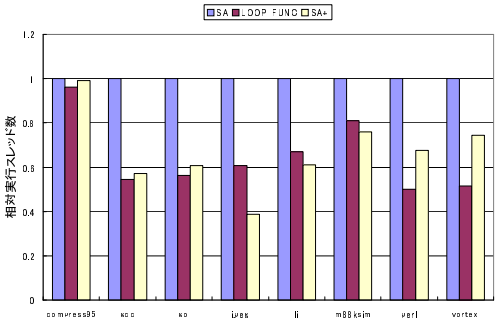

図 6 実行スレッド数の削減

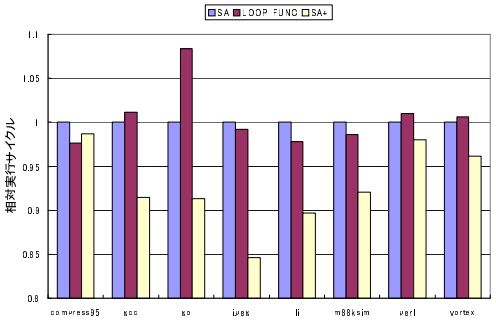

図 7 相対実行サイクル

マークによって 10〜15%の速度向上が得られることを示した。また、スレッド分割においては、ループ、関数のみに着目した分割では不十分であることがわかった。

本稿で提案したスレッド分割手法は、複数のスレッドが CFG のノードを共有するようにスレッド分割を行うもので、ノードの共有が可能なスレッドモデルであることが必要である。

今後の課題として、第一にスレッド境界エッジの選択アルゴリズムの改良があげられる。特にループの扱いについては更なる検討が必要と考えられる。また、スレッド間のデータ依存を考慮することも必要になる。更に、m提案したスレッド分割手法によってスレッドサイズが拡大したことにより、スレッド間の制御依存、データ依存に関する最適化の余地が大きくなることが期待できる。

次に、提案したスレッド分割手法によって生成されたスレッドの性質の評価を詳細に行う必要がある。特にスレッド予測に対しては大きな影響を及ぼしている可能性が高く、その影響を評価し、場合によっては予測機構の改良を行う必要がある。

謝　　辞

本稿の研究の一部は、文部科学省科学研究費補助金 基盤研究 B(2)13480077 及び半導体理工学研究センター、科学技術振興事業団 CREST プロジェクト、日本学術振興会 21 世紀 COE プログラムの支援により行われた。

参　考　文　献

[1] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In Proc. of the 31st MICRO, pages 226236, 1998.

[2] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing, 2000.

[3] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, Vol. 48, No. 9, pp. 866–880, 1999.

[4] Niko D. Barli, Hiroshi Mine, Shuichi Sakai and Hidehiko Tanaka. A thread partitioning algorithm using structural analysis. 情報処理学会研究報告 計算機アーキテクチャ研究会, ARC-139, August 2000.

[5] Niko D. Barli, Tashiro Daisuke,Shuichi Sakai and Hidehiko Tanaka. Dynamic thread extension for speculative multithreading architectures. 情報処理学会研究報告 計算機アーキテクチャ研究会, ARC-144, July 2001.

[6] T. N. Vijaykumar. Compiling for the multiscalar architecture. Technical Report CS-TR-1998-1370, 1998.

[7] Yoshimitsu Yanagawa, LuongDinh Hung, Chitaka Iwama, Niko Demus Barli, Shuichi Sakai, and Hidehiko Tanaka. Complexity analysis of a cache controller for speculative multithreading chip multiprocessors. 情報処理学会研究報告 計算機アーキテクチャ研究会, ARC-152, March 2003.

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

[9] Anasua Bhowmik and Manoj Franklin, A General Compiler Framework for Speculative Multithreading, Proceedings of the 14th Symposium on Parallelism in Algorithms and Architectures, August 2002.