修士学位論文

マルチコアプロセッサのための
スレッドスケジューリングに関する研究

平成 19 年度

(平成 20 年 1 月 31 日提出)

東北大学大学院情報科学研究科

情報基礎科学専攻

船矢祐介

Thread Scheduling for Multi-Core Processors

Yusuke Funaya

Abstract

A multicore processor of simultaneous multi-threading (SMT) cores has attracted much attention as a promising architecture to effectively use a lot of transistors on a chip. Until recently, the operating systems (OS) was not able to take into account the difference between physical and logical cores for thread scheduling. The thread scheduling strategies of the latest OSes are designed to handle extreme load unbalance among cores, and hence passive for thread migration. However, the selection of threads simultaneously running on an SMT core is an important factor that significantly affects the effective performance. Therefore, it is necessary to establish an active thread scheduling methodology that can take advantage of the features of both threads and the architecture.

The objective of this thesis is to extract the potential of a multicore processor by the thread scheduling. To this end, thread scheduling needs to consider the characteristic of each thread for the appropriate core allocation. This thesis proposes a new metric for characterizing each thread, named a SMT priority, and thread scheduling algorithms based on the metric. The SMT priority can estimate the performance degradation caused by conflicts of threads on shared resources such as arithmetic units. It can be obtained by runnning each thread alone, and therefore the number of cycles for thread profiling to estimate is in proportion to the number of threads, not to the number of their combinations. It can also estimate the conflicts among threads on each resource only from the instruction queue length.

This thesis first proposes the SMT priority and then a static thread scheduling method, which calculates the SMT priority once in advance and schedules threads at the beginning of the execution. Moreover, this thesis proposes a dynamic thread scheduling method,

which performs thread scheduling at fixed intervals to reflect the dynamic changes in behaviors of threads.

The static thread scheduling method is evaluated by scheduling four threads for a multiprocesor of three SMT cores. The results show that the static thread scheduling method can find the best combination for 53 threads out of 70 threads. The superiority of the proposed method is clearly demonstrated because it can find an appropriate combination of threads especially if the performance greatly depends on their combination.

The results also show that the dynamic thread scheduling method has a potential to achieve higher performance than the static thread scheduling method, because of its capability to adapt to the time varying behaviors of a problem.

目次

図目次

表目次

第 1 章
緒論

1.1 研究の背景

近年，半導体加工技術の進歩により，1 つのマイクロプロセッサのチップ上に非常に多くのリソースが搭載されるようになった．潤沢なリソースをいかに効率的に利用するかが，これからのマイクロプロセッサの性能向上の鍵である．

従来のマイクロプロセッサは，クロック周波数の向上と，命令レベル並列性（ILP）を利用したシングルスレッドのスループット向上により性能向上を実現してきた．クロック周波数の向上は，半導体加工技術の微細化やパイプライン段数の増加によって実現され，より高い ILP の抽出は，スーパースカラや VLIW に代表されるアーキテクチャ上の工夫やコンパイラ技術の向上により実現された [1]．しかし，チップの高集積化・高周波数化に伴い，消費電力と発熱量の増大が深刻な問題となり，これ以上の周波数上昇は困難になってきている [2, 3]．さらに，従来から指摘されている ILP 抽出の限界 [4] により，投資したハードウェアコストに対して得られる性能向上が大きく低下してきている．

従来のプロセッサが抱えるこれらの問題を解決することが期待されている代表的なアプローチとして，比較的低い周波数で動作する複数のコアを 1 つのチップ上に集積し，チップの総合性能を向上させる Chip Multi-Processor（CMP）[5] や，1 つのコアで複数のスレッドを同時実行可能にし，チップ上の資源の効率的な活用，およびメモリレイテンシの効果的な隠蔽が可能な Simultaneous Multi-Threading（SMT）[6, 7, 8] がある．CMP はマルチコアプロセッサとも呼ばれ，多くの場合 L2 キャッシュもしくは L3 キャッシュのみを複数のスレッド間で共有する．CMP の例として IBM Power4 [9]，Intel Core 2 Duo [10] がある．一方，SMT で

は CMP よりも多くのリソース（命令キューのエントリや物理レジスタ，演算ユニットなど）を複数のスレッド間で共有するため，より効率的なリソースの使用が可能となる．シングルコアに SMT を用いた例として Intel Pentium4 [11] がある．今後，マイクロプロセッサアーキテクチャの多くが，SMT 技術を利用するであろうと言われている [12].

現在，SMT コアで構成されたマルチコアプロセッサが注目されている．例として，2 コンテキストの SMT コアを 2 つ搭載した IBM Power5 [13] と Intel Dual-Core Xeon [14] がある．本論文ではこのアーキテクチャに焦点を当て，プロセッサのリソースを無駄なく利用して過剰なリソース投入を行うことなくプロセッサの性能向上を狙う．

SMT では同時実行するスレッド間でプロセッサリソースのほとんどを共有するのに対し，CMP ではキャッシュメモリ以外のリソースは共有しない．そこで，このようなプロセッサにおけるリソース競合は，どのような特徴を持ったスレッドが同じ SMT コア上にスケジューリングされるかに大きく依存する．Moseley ら [15] によると，Intel Pentium4 2.53GHz で SPEC CPU2000 ベンチマークを SMT を利用して 2 つ同時実行した場合，2 つを順次実行する場合と比較して 30% 以上速度向上する組み合わせもあるが，逆に 30% 速度低下する組み合わせもあることが報告されている．これは，SMT コアの実効性能は同時実行するスレッドの組み合わせによって大きく変動することを示している．現在，マイクロプロセッサに対するスレッドスケジューリングは，主にオペレーティングシステム（OS）によって行われている．しかし，これまでの多くの OS は，SMT による同一コアのスレッドコンテキストと CMP による物理的に異なるコアのスレッドコンテキストを区別したスレッドスケジューリングを行ってこなかった．最近では SMT に対応したスケジューリング [16] が行われるようになったが，コア間の極端な負荷の偏りを分散することを目的とした非常に消極的なスレッドスケジューリングしか行われていない．SMT コアで構成されたマルチコアプロセッサの潜在能力を最大限引き出すためには，スレッドの特徴と，アーキテクチャの特徴を考慮した積極的なスレッドスケジューリングが必要不可欠である．そのためには，プロセッサリソースの大部分をスレッド間で共有している SMT の特性を考慮し，各スレッド固有な特徴に応じて最適なコア内・コア間スレッドスケジューリングを行う必要がある．

1.2 研究の目的

プロセッサのコア数より実行するスレッド数が多い場合，複数のスレッドを組み合わせてSMT を利用して実行することになる．そこでまず，SMT で実行するのに適したスレッドの組み合わせを見つける事が重要となる．SMT での実行に適したスレッドの組み合わせを見つけるためには，組み合わせて実行した場合に発生する，キャッシュ，演算器，命令キューなどのリソース競合が，実効性能にどの程度影響を及ぼすのかを知る必要がある．その方法として，実際に複数のスレッドを組み合わせて実行し，リソースの競合とその影響を調査する方法が考えられるが，この方法では実行するスレッド数が多い場合，調査すべき組み合わせが膨大な数になるという問題点がある．この問題点を解決するには，スレッドを単独で実行して得られる統計量から，他のスレッドと組み合わせて実行した場合の実効性能への影響を予測し，スレッド数が増えた場合にも，少ない調査でスレッドスケジューリングを行う必要がある．そこで本論文では，各スレッドが持つ固有の特徴を調査し，実効性能に大きな影響を及ぼす統計量をそのスレッドの特徴量として定量的に評価する．そして，その特徴量を基に，あるスレッドを他のスレッドと組み合わせて実行した場合に発生するリソース競合とその実効性能への影響を予測し，最適な組み合わせを決定する．

本論文の目的は，SMT コアで構成されたマルチコアプロセッサ上で実行される複数のスレッドを，スレッドの特徴を考慮して適切なコアに割り当てることで，高いスループットを達成することである．そのために，各スレッドのスレッド特徴量として SMT 優先度を定義し，SMT 優先度に基づくスレッドスケジューリング手法を提案する．評価はシミュレーションによって行う．

1.3 本論文の構成

本論文の構成を以下に示す．1 章は緒論である．2 章では，現在のスレッドスケジューリングの問題点に触れ，スレッドスケジューリングの必要性を述べる．さらにスレッドスケジューリングの関連研究について概説し，従来手法の問題点を指摘する．3 章では，スレッド特徴量として SMT 優先度を定義し，SMT 優先度に基づく静的および動的スレッドスケジューリング手法を提案する．4 章では，提案手法の有効性をシミュレーションによって評価する．静

的および動的スレッドスケジューリング手法の評価の後，SMT 優先度の有効性についても評価・考察する．5 章では，結論として本研究によって得られた知見と，今後の課題について述べる．

第 2 章
スレッドスケジューリングの必要性

2.1 緒言

本章では，はじめに現在のマルチコアプロセッサにおけるスレッドスケジューリングとその問題点について述べ，新しいスレッドスケジューリング手法の必要性を示す．その後，現在行われているスレッドスケジューリングの関連研究を挙げ，その手法を概説し問題点を指摘する．

2.2 現在のスレッドスケジューリングと問題点

マルチスレッド処理による高いスループットの実現は，ソフトウェア，コンパイラ，ハードウェアの各分野で広く研究が行われている．中でもソフトウェアのマルチスレッド化は古くから研究されており，現在ではウェブサーバアプリケーションからテキストエディタに至るまで，多くのアプリケーションで実現されている．また，コンパイラもソフトウェアと共にマルチスレッド化への対応が進み，現在では GNU Compiler Collection（GCC）や Intel Compiler といった主なコンパイラの多くがマルチスレッドをサポートしている．さらに，シーケンシャルなコードから積極的にスレッドを生成し，投機的に実行することで実効性能を向上させるコンパイラも提案されている [17]．一方，ハードウェア（ここではプロセッサを指す）のマルチスレッド処理能力は，従来のタイムスライスマルチスレッディング技術に加え，SMT や CMP の実現により飛躍的に向上してきている．

プロセッサ上で，あるスレッドを実行するのに必要な一連のデータ（プログラムカウンタやレジスタの値など）のことをスレッドコンテキストという．SMT プロセッサや CMP では，

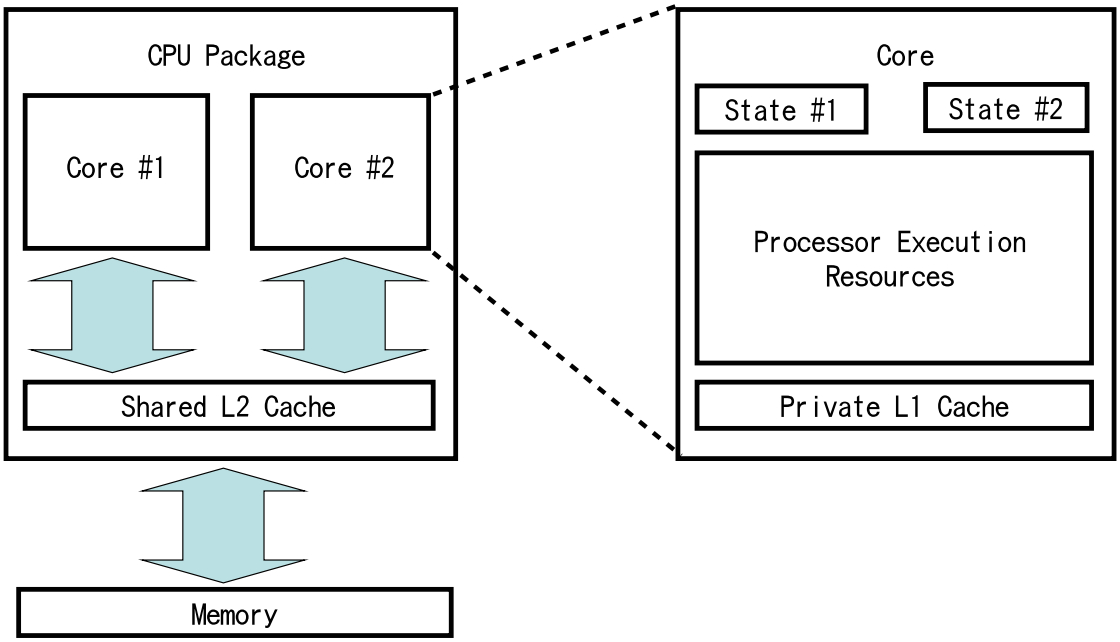

図 2.1　SMT コアで構成された CMP

スレッドコンテキストを同時に複数保持することで，複数のスレッドを停止させることなく実行可能となる．SMT コアで構成された CMP のアーキテクチャを図 2.1 に示す．実行中のスレッドのスレッドコンテキストが保存される部分を State で示した．物理的なコアが 2 つあり，そのそれぞれが SMT によって 2 つのスレッドコンテキストを保持できるため，合計で 4 つのスレッドを同時実行できる．スレッドコンテキストを複数保持可能なこれらのプロセッサにおいて，実際に複数のスレッドを実行する場合，どのスレッドをどのスレッドコンテキストで実行するかを決定するスレッドスケジューリングが必要となる．

現在の汎用プロセッサにおけるスレッドスケジューリングは一般的に OS が行う．従来の OS は，SMT コア上の複数のスレッドコンテキストを，物理的に異なるコアとして認識していた．Linux を例に挙げると，2004 年 6 月にリリースされた kernel 2.6.7 までは，スケジューリングの際に SMT による論理的なコアと物理的なコアの違いを考慮したスケジューリングは行われてこなかった．これに対し，kernel 2.6.7 で導入された負荷均衡化アルゴリズムでは，物理的なコアと論理的なコアを区別し，物理的なコア間の負荷が極端に異なる場合に，いくつかのプロセスを物理的なコア間で移動し負荷分散を試みている [16]．しかし，プロセス移動の条件は非常に消極的である．次のいずれかの条件を満たさない限り，プロセスの移動はされない．

- 移動先の CPU のすべての論理プロセッサが待機状態である場合.
- 負荷に大きな偏りがあるにもかかわらずプロセスの移動に何度も失敗している場合.
- 移動するプロセスがしばらく実行されていない場合.

よって，1 つ以上の物理コアが完全に待機状態になるか，もしくはコア間の負荷が極端に偏り，いくつかのプロセスが長時間実行の機会を失うような事態にならない限り，一度アサインされたプロセスは同一のコアで実行され続けることになる.

ハードウェア資源を最大限に活用し，複数のスレッドを高いスループットで実行するためには，負荷バランスだけではなく，スレッドの相性に基づいた積極的なスレッドスケジューリングが必要不可欠である．そこで，本論文ではスレッドの相性を各スレッドの特徴から予測し，プロセッサの能力を最大限に引き出すためのスレッドスケジューリング手法を提案する.

2.3 静的スレッドスケジューリングと動的スレッドスケジューリング

事前にスレッドの特徴を評価し各コアへの割り当てを決定し，それに従い実行開始時に一度だけスレッドスケジューリングを行い，その後実行終了まで再スケジューリングを行わない方法を，**静的スレッドスケジューリング**という．図 2.2 に静的スレッドスケジューリングの概要を示す．このアプローチは，事前にスレッドの特徴量を評価できることを前提としているため，時間をかけて多くの命令をサンプリングすることで，スレッドの平均的な挙動を比較的正確に捉えることができる．また，スケジューリングは最初の 1 回のみであるため，実行中にスケジューリングのためのコストが掛からない．特に，規則的な計算が多いアプリケーション，例えば，行列計算の繰り返しがほとんどを占める流体計算アプリケーションなどは，スレッドの挙動が予測しやすく，静的スレッドスケジューリングに向いているといえる.

しかし，静的スレッドスケジューリングが適さない場合もある．一般に，アプリケーションの挙動は実行の進捗に伴って大きく変化するため，プログラム全体の特徴を一意に決定することは難しいことが知られている [18, 19]．特に挙動の変化が大きいスレッドの場合，事前に得られるスレッド特徴量と，1 回だけのスケジューリングでは充分に対応できない可能性がある．なぜなら，スレッド特徴量が示す特徴では，スレッドの挙動が変化する前と変化した後のそれぞれの特徴が平均化されてしまうため，どちらの挙動も正確に予測することができないからで

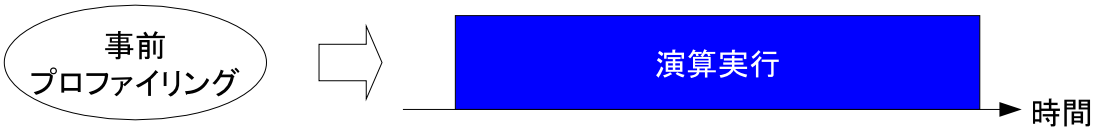

図 2.2　静的スレッドスケジューリング

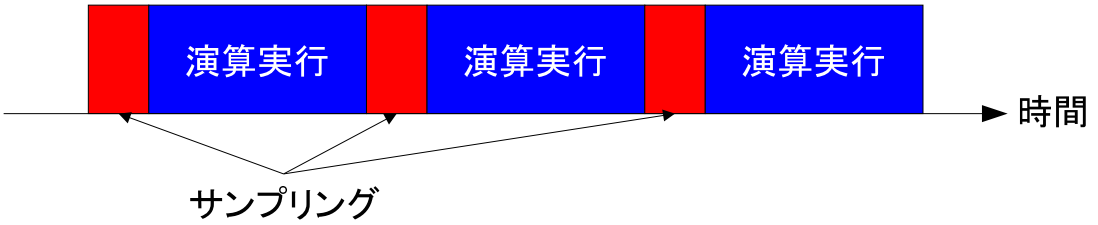

図 2.3　動的スレッドスケジューリング

ある．さらに，1 回だけのスケジューリングでは，実行中のスレッドの 1 つが終了したり，新たにスレッドの実行が開始された場合に対応できない．

そこで，より多様なスレッドの特徴に対応するために，実行中に再スケジューリングを繰り返すアプローチが必要となる．これを**動的スレッドスケジューリング**という．図 2.3 に動的スレッドスケジューリングの概要を示す．動的スレッドスケジューリングにより，スレッドの挙動の変化が大きい場合，または途中でスレッドが終了，もしくは開始された場合にも適切なスケジューリングが可能となる．一方で，実行中に何度もサンプリングとスケジューリングを行うため，スレッド実行中に発生するスケジューリングのためのオーバーヘッドを考慮しなければならない．

2.4　関連研究

SMT プロセッサに対するスレッドスケジューリングは，主にどのような特徴を持ったスレッドを組み合わせて同時実行するかという点に着目して研究が行われている．

Snavely ら [20, 21, 22] は，Sample Optimize Symbios（SOS）と呼ばれる SMT 用のスケジューラを提案している．SOS では，スレッド特徴量の計測のためにスレッドの実行を一時

中断し，サンプリングフェーズと呼ばれる専用のフェーズ用いてサンプリングを行う．まず，サンプリングフェーズでそれぞれのスレッドの組み合わせを一定時間実行し，演算ユニットや命令キューでの競合を評価する．続くフェーズでその結果を基に最適なスレッドの組み合わせを決定する．最後に，OS によってスケジュールされているジョブの優先度を変更し，最適なスレッドの組み合わせが相対的に長い時間実行されるようにする．SOS では，サンプリングフェーズを用いることによる時間的なオーバーヘッドがあるにも関わらず，SOS を実装しない場合に比べて，システムのスループット向上と，ターンアラウンドタイム短縮の両方を達成している．しかし，この手法はサンプリングをそれぞれのスレッドの組み合わせ数行うため，今後スレッド数が多くなった場合，サンプリングのコストが増大してしまう恐れがある．

Parekh ら [23] は，SMT におけるスレッドスケジューリング手法の提案と評価を行い，IPC-based thread-sensitive スケジューリングアルゴリズムと呼ばれる，最も IPC が高いスレッド同士を組み合わせる手法が最適であるとしている．リソースを効率的に使うスレッド同士を一緒にすることで，レイテンシが長い命令（主にキャッシュミスを引き起こす命令）で多くのリソースを占有する遅いスレッドによって，速いスレッドが妨害されるのを避けることができるためである．スレッド毎に設けたハードウェアカウンタによって，実行中のスレッドの整数演算ユニットおよび浮動小数点演算ユニットのそれぞれの IPC を計測し，その値を基に次のタイムスライスで実行するのに最適な組み合わせを OS にフィードバックする．しかし，他のスレッドと同時実行しながら計測される IPC は，どのスレッドと一緒に実行されているかに依存して大きく変化するため，正確にスレッド固有の特徴を予測できない．

El-Moursy ら [24] は，Ready Inflight Ratio Scheduling （RIRS）というスレッド特徴量に基づく動的スレッドスケジューリング手法を提案している．RIRS では，命令発行キューの中にある発行準備ができている命令数と，発行ステージからライトバックステージの間を実行中の命令数の割合（RIR）を用いて，そのスレッドがどの程度リソースの変化に敏感であるかを予測する．RIR が高いスレッドは，プロセッサリソースを大量に消費しながら高いスループットで演算が進んでいる敏感なスレッドである．このようなスレッドは，利用可能なリソースが減少すると実効性能が大きく低下する恐れがあるため，リソースをあまり必要としていないスレッドと組み合わせる．RIRS では，スレッド特徴量の計測のためにスレッド実行を中断する必要はなく，2 つのスレッドを SMT コア上で同時実行しながら，ハードウェアカウンタを用いて必要な値を計測し，その値を基に次のインターバルのスケジューリングを行う．10 組のワークロードミックスを用いて性能評価を行い，静的スレッドスケジューリングで見られ

る大きな性能低下が起こらなかったという結果を得ている一方，すべての組み合わせの中から最も実効性能が高い組み合わせを選んだ理想的な静的スレッドスケジューリングの結果に対して，優位な結果は得られていない．

Cazorla ら [12] は，SMT プロセッサにおける QoS 問題に取り組んでいる．実行中の複数のスレッドの中で最も優先度の高いスレッドが，単独実行時の 70% 以上の性能が発揮できることを保証する．彼らの手法も，SOS と同様にスレッド特徴量の計測のためにスレッドの実行を一時中断する必要がある．一定時間ごとにサンプリングフェーズを設け，そこで最優先のスレッドを単独で実行し，そのスレッドの単独実行時の性能を予測する．その情報を基にそのスレッドへのリソースの配分を決定する．筆者らは，論文中で，直後の 100 万サイクルの実効性能を予測するのに，100 万サイクルの 1% にあたる 1 万サイクルをサンプリングすれば，予測が充分可能であるとしている．

すでに同時実行されている複数のスレッドが，SMT プロセッサの共有リソースを効率的に利用するために，フェッチポリシによってスレッド間のリソース割り当てを調整する手法も提案されている．代表的な手法に，前回発行命令数が少ないスレッドから優先して命令をフェッチする ICOUNT [7]，ICOUNT を拡張し，片方のスレッドが L2 キャッシュミスでストールしている間はもう片方のスレッドをフェッチしつづける STALL [25]，STALL を改良し L2 キャッシュミスの発見精度を高めた FLUSH [25]，さらに，スレッドのキャッシュミスの振る舞いによって STALL と FLASH を切り替える FLUSH++ [26] などがある．

現在の問題点は，正確な予測のためには，サンプリングに多くのコストがかかってしまうことである．スレッドを組み合わせてサンプリングを行う SOS では，スレッド数が増加した場合にはサンプリングのコストが膨大になってしまう．IPC-based thread-sensitive スケジューリングと RIRS では，サンプリングにかかる時間を削減するために実行とサンプリングを同時に行っているが，その結果正確なスレッド特徴量を取得できていない．

この問題を解決するためには，サンプリングするスレッド数を少なくすることに加えて，SMT によって共有している様々なリソースでの競合を，シンプルな特徴量から正確に予測することが必要である．そこで，本論文では単独のスレッドを実行したときの命令キューから得られるシンプルな特徴量を用いたスレッドスケジューリング手法を提案する．単独のスレッドから得られる特徴量を用いることでサンプリングの回数を実行中のスレッド数に抑え，将来のスレッド数増加にも対応できるようにする．さらに，実行とサンプリングを分けて行い正確な

スレッド特徴量の取得を可能にさせる．また，フェッチポリシによるリソースの割り当てはスレッドスケジューリングと共存することが可能であるが，スレッドのリソース利用量の予測を困難にしてしまう可能性があるため，本論文では特に断りが無い限りフェッチポリシは最もシンプルなラウンドロビンを用いる事とする．

2.5 結言

SMT コアで構成されたマルチコアプロセッサにおけるスレッドスケジューリングでは，どのスレッドを SMT で実行するのかがスケジューリングの鍵となる．将来，実行するスレッドが増加していくことを考慮すると，重要なことは，いかに少ないコストで適切な組み合わせを選ぶかということである．

以上，スレッドスケジューリングの概要を示し，その関連研究について概説した．3 章では，提案手法である SMT 優先度に基づくスレッドスケジューリング手法について述べる．

第 3 章
SMT 優先度に基づくスレッドスケジューリング

3.1 緒言

スレッドスケジューリングには，静的スケジューリングと動的スケジューリングがあることは第 2 章で述べた．どちらの手法が優れているのかは，実行環境や実行するスレッドの特性によって異なるため一概に決めることは困難である．そこで本論文では，静的・動的両方のスレッドスケジューリング手法を提案する．

本章では，まず SMT での実効効率を評価する指標として，SMT 効率を定義する．次に，スレッドスケジューリングのためのスレッド特徴量として SMT 優先度を定義し，SMT 優先度に基づく静的スレッドスケジューリング手法を提案する．その後，より多様なスレッドに対応するために，動的スレッドスケジューリング手法を提案する．

3.2 SMT 優先度

適切なスレッドスケジューリングのためには，あるスレッドの組み合わせが SMT を利用して同じコアで実行するのにどの程度適切であるかを表すスレッド特徴量が必要である．そこでまず，SMT の実効効率を定義する．2 つのスレッドを 1 コアで SMT を利用して実行した場合と，2 つのスレッドにそれぞれ 1 コアずつ割り当てて 2 コアを使って実行した場合の実効性能を IPC（Instructions Per Cycle）で計測する．後者の，スレッドを 2 コアで実行したときの性能を最大性能と考え，それに対する SMT 実行時の性能の比率を式 (3.1) に示す SMT 効

率として定義する．ただし IPC_{1core}，IPC_{2cores} はそれぞれ 1 コア実行時の IPC，2 コア実行時の IPC を表す．

$$\text{SMT 効率} = \frac{\mathrm{IPC}_{1core}}{\mathrm{IPC}_{2cores}} \tag{3.1}$$

これによって，SMT 効率が高い程その 2 つのスレッドの組み合わせは SMT に適していると定義できる．しかし，SMT 効率は 2 つのスレッドの組み合わせに対しての定義であり，個別のスレッドからは得られない．

そこで，単独のスレッドから SMT 効率を予測する特徴量を定義する．SMT 効率が低下するのは，同じコアで実行する 2 つのスレッドが競合し，共有する演算ユニットをうまく使えない場合である．原因は以下の 2 つが考えられる．

■演算ユニット　演算ユニットの利用率が高く演算ユニットの空きが少ない場合である．この場合，あるスレッドが演算ユニットを使用している間もう一方のスレッドはその演算ユニットを利用できないからである．

■命令キュー　演算ユニットへ直接命令を発行する各命令ユニットのキューに空きがなく，命令発行がストールしてしまう場合である．各命令ユニットのキューとは整数演算ユニット・浮動小数点演算ユニットのリザベーションステーションおよびロードユニット・ストアユニットのバッファを指し，これらをまとめて命令キューと呼ぶことにする．あるスレッドの命令が命令キューの 1 つを占有した場合，たとえその演算ユニットが空いていたとしても，もう一方のスレッドはその演算ユニットを使うことができない．実際，Intel のハイパースレッディングテクノロジでは，片方のスレッドが利用可能な命令キューのエントリ数に上限を設定し，両方のスレッドが完全にストールするという最悪の事態を防止するようにしている [11]．しかしその上限はハードウェアのキューサイズによって決められる固定されたものであり，スレッドが必要としているエントリ数とは無関係である．エントリを多く必要とするスレッドはエントリが不足して実効性能が低下し，一方でエントリをあまり必要としないスレッドは，過剰に割り当てられたエントリを使用しないで遊ばせることになる．この問題を解決するためには，ハードウェアによる固定的な資源制約を与えなくとも適切な資源配分が可能なスレッドスケジューリング技術が必要不可欠である．

以上の 2 種類の原因によって引き起こされたリソース競合を定量的に求めるために，予備実

験にて事前に 1 つのスレッドを 1 つのコアで実行し，その結果を分析する．まず全ての命令が滞りなく実行される理想の場合を考える．この時，複数同時発行のプロセッサにおける理論的な最大処理能力を

$$\mathrm{C}_{ideal} = \text{実行にかかるサイクル数} \times \text{命令発行幅} \tag{3.2}$$

とする．これを用いて演算ユニットの利用率を次の式で定義する．

$$\mathrm{C}_{busy} = \frac{\text{実際に発行された命令数}}{\mathrm{C}_{ideal}} \tag{3.3}$$

次に，あるサイクルで 1 つの命令が発行されなかった原因を，命令キューの場合（instQ）とそれ以外（other）に分けると，それぞれの割合 C_{instQ}，C_{other} と C_{busy} の間には次式が成立する．

$$\mathrm{C}_{busy} + \mathrm{C}_{instQ} + \mathrm{C}_{other} = 1 \tag{3.4}$$

前述の考察より，あるスレッドが SMT に適しているということは C_{busy} と C_{instQ} が小さい，つまり C_{other} が大きいということである．よって C_{other} を SMT 優先度として式 (3.5) で定義する．

$$\text{SMT 優先度} = 1 - \mathrm{C}_{busy} - \mathrm{C}_{instQ} \tag{3.5}$$

3.3 静的スレッドスケジューリング手法

SMT 優先度に基づく静的スレッドスケジューリング手法を提案する．以下にスケジューリングの手順を示し，さらに図 3.1 にスケジューリングのフローチャートを示す．スケジューリングは事前に 1 回のみ行い，その後はスケジューリングされたスレッドの組み合わせを実行する．また，同時に実行するスレッド数を N_t，プロセッサのコア数を N_c とし，$N_t \leq N_c$ の場合は，スループットを最大にするためには SMT を利用する必要がないので，1 コアに 1 スレッドずつ割り当てることにする．そこでここでは $N_t > N_c$ であると仮定する．

1. 事前にスレッドを単独で実行し，SMT 優先度に必要な値を計測する．そして，その値を基に各スレッドの SMT 優先度を計算する．

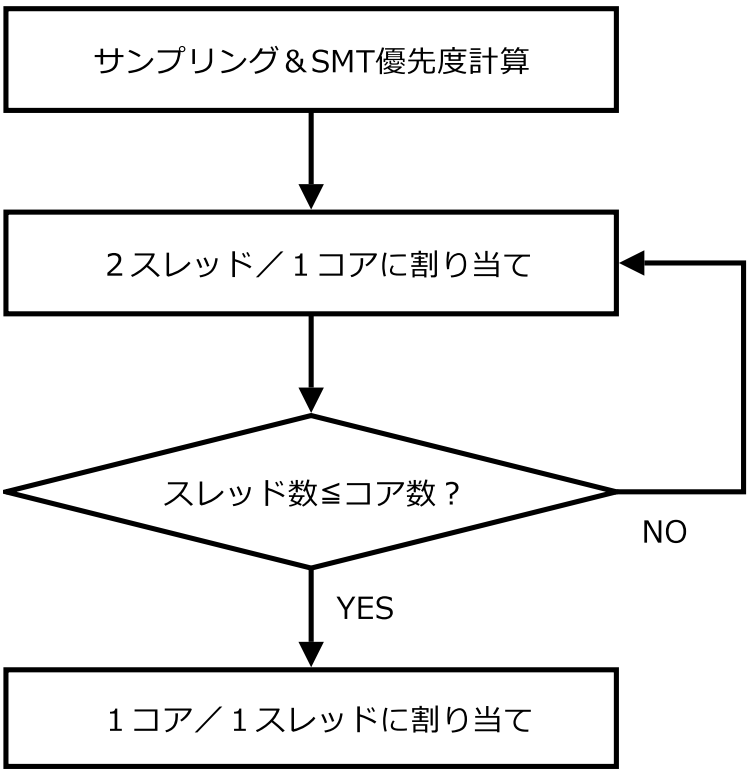

図 3.1　静的スレッドスケジューリングのフローチャート

2. SMT 優先度の和が最大となるスレッドの組み合わせを選択し，1 コアに割り当てる．割り当てを n 回行った後の，どこのコアにも割り当てられていないスレッドは $N_t - 2n$ 個，1 つもスレッドが割り当てられていないコアは $N_c - n$ 個となる．この割り当てを $N_t - 2n \leq N_c - n$ になるまで行う．
3. 残りのスレッドを残りのコアに 1 つずつ割り当てる．

3.4　動的スレッドスケジューリング手法

静的スレッドスケジューリングを動的スレッドスケジューリングに拡張するにあたって，特徴量の取得方法と，スケジューリングの間隔について議論する必要がある．

3.4.1 特徴量の取得方法

プロセッサ上で複数のスレッドが実行されている場合，そのスレッドの特徴量を取得するには次の 2 つの方法が考えられる．1 つは，一度実行を止め一定時間サンプリングを行う方法である．この方法では，特徴量取得のためのオーバーヘッドが大きくなる恐れがあるが，専用のサンプリングフェーズを用いることで，正確な特徴量が取得できる．もう 1 つは，実行しながらサンプリングを行う方法である．この方法では，特徴量取得のためのオーバーヘッドは小さくてすむが，特徴量取得の時に他のスレッドの影響を受けるため，正確な特徴量が取得できない可能性がある．

El-Moursy ら [24] の提案する RIRS では，実行しながらサンプリングを行っているが，静的スレッドスケジューリングに対する優位性が示されていない．一方，Snavely ら [22] の提案する SOS では，一度実行を止めてサンプリングを行っているが，そのオーバーヘッドを考慮しても，動的スレッドスケジューリングによるスループット向上を達成している．しかしながら，SOS では，すべての組み合わせを調査しなければならないという問題がある．これに対して，本論文ではスレッド特徴量として単独のスレッドから得られる SMT 優先度を用いるため，サンプリングする回数はすべての組み合わせ数ではなく実行中のスレッド数で済む．以上の理由から，本論文では，実行を一度止めてサンプリングを行う方法で動的スレッドスケジューリングを実現する．

3.4.2 スケジューリング間隔とサンプリング長さ

再スケジューリングの結果，スレッドの組み合わせに変更が生じた場合，実行中のスレッドを他のコアに移動する必要がある．これをスレッドマイグレーションという．動的スレッドスケジューリングでは，スケジューリングの間隔が短い程スレッドの挙動の変化に対してより適切なスケジューリングができる．一方で，サンプリングとスレッドマイグレーションによるオーバーヘッドが増大するというトレードオフがある．また，サンプリングの長さにも，長い程正確な予測ができる一方で，オーバーヘッドが増大するというトレードオフがある．これらのトレードオフについての一般的な最適値というのはなく，様々な論文で様々な議論がなされている．

スケジューリングの間隔は，提案するスケジューラを OS のスケジューラの一部として実

装するか，独立したスケジューラとして実装するかによって異なってくる．OS のスケジューラとして実装する場合，スケジューリング間隔は OS のタイムスライスインターバルとなる [27, 12]．しかし独立したスケジューラとして実装する場合でも，多くは OS のタイムスライスインターバルと同じか，もしくはそれよりも短い間隔である [28, 24]．動的スレッドスケジューリングでは，スケジューリング間隔が短くなるほどスレッドの挙動の変化を的確に捉えることができる一方，相対的にサンプリングのオーバーヘッドが増大する．そこで，ある程度の間隔の長さが必要となるが，このトレードオフに対して，上に示した関連研究から，OS のタイムスライスインターバルである 100 万サイクルは充分に長い間隔であるといえる．また，スケジューリングの間隔を OS のタイムスライスインターバルの間隔にすることで，将来，提案手法を OS のスケジューラと協調させることも容易になる．以上の理由により，本論文ではスケジューリングの間隔を 100 万サイクルとする．

サンプリングの長さは，100 万サイクル後までのスレッドの挙動を予測でき，かつなるべく短いことが要求される．N サイクル後までのスレッドの挙動を予測するために必要なサンプリングの長さは，使用するアーキテクチャと実行するスレッド，そしてスケジューリングの手法によって異なるため，論文によって異なる値が用いられている．本論文と同じ SMT コアで構成されたマルチコアプロセッサをターゲットにしている Cazorla ら [12] は，ある 1 スレッドの，サンプリング直後の 100 万サイクル後までの性能を予測するのに，100 万サイクルの 1% にあたる 1 万サイクルのサンプリングで充分であると報告している．さらに短いサンプリングでのスケジューリングも報告されている [21] ため，わずか 1% でも性能予測には充分であると考えられる．以上の理由により，本論文ではサンプリングの長さをスケジューリング間隔の 1% にあたる 1 万サイクルとする．

3.4.3 スレッドスケジューリング手法

以上の議論を基に，SMT 優先度を用いた動的スレッドスケジューリング手法を提案する．以下にスケジューリングの手順を示し，さらに図 3.2 にスケジューリングのフローチャートを示す．ただし，スケジューリング間隔を L サイクル，サンプリングフェーズの長さを l サイクルとする．スケジューリング完了後 L サイクル演算を実行し，またスケジューリングを行うことを繰り返す．また，$N_t \leq N_c$ の場合は，スループットを最大にするためには SMT を利用する必要がないので，1 コアに 1 スレッドずつ割り当てることにする．そこでここでは $N_t > N_c$

であると仮定する．

1. 実行中のスレッドのうちの 1 つを選び，残りのスレッドの実行を停止する．
2. 1 つのスレッドを l サイクル実行し，SMT 優先度に必要な値を計測し，その値を基にそのスレッドの SMT 優先度を計算する．
3. まだサンプリングしていないスレッドがあれば，そのうちの 1 つにコンテキストスイッチし，そのスレッドのサンプリングを行う．全てのスレッドのサンプリングが完了したら，次の手順へ進む．
4. SMT 優先度の和が最大となるスレッドの組み合わせを選択し，1 コアに割り当てる．この割り当てを $N_t - 2n \leq N_c - n$ になるまで n 回行う．
5. 残りのスレッドを残りのコアに 1 つずつ割り当てる．

3.5 結言

本章では，スレッド特徴量として，スレッド実行時の演算ユニットの利用率と命令発行キューの利用率に着目した SMT 優先度を提案し，SMT 優先度に基づくスレッドスケジューリング手法を提案した．4 章では，本手法の有効性をシミュレーションによって評価する．また，SMT 優先度の評価・考察も行う．

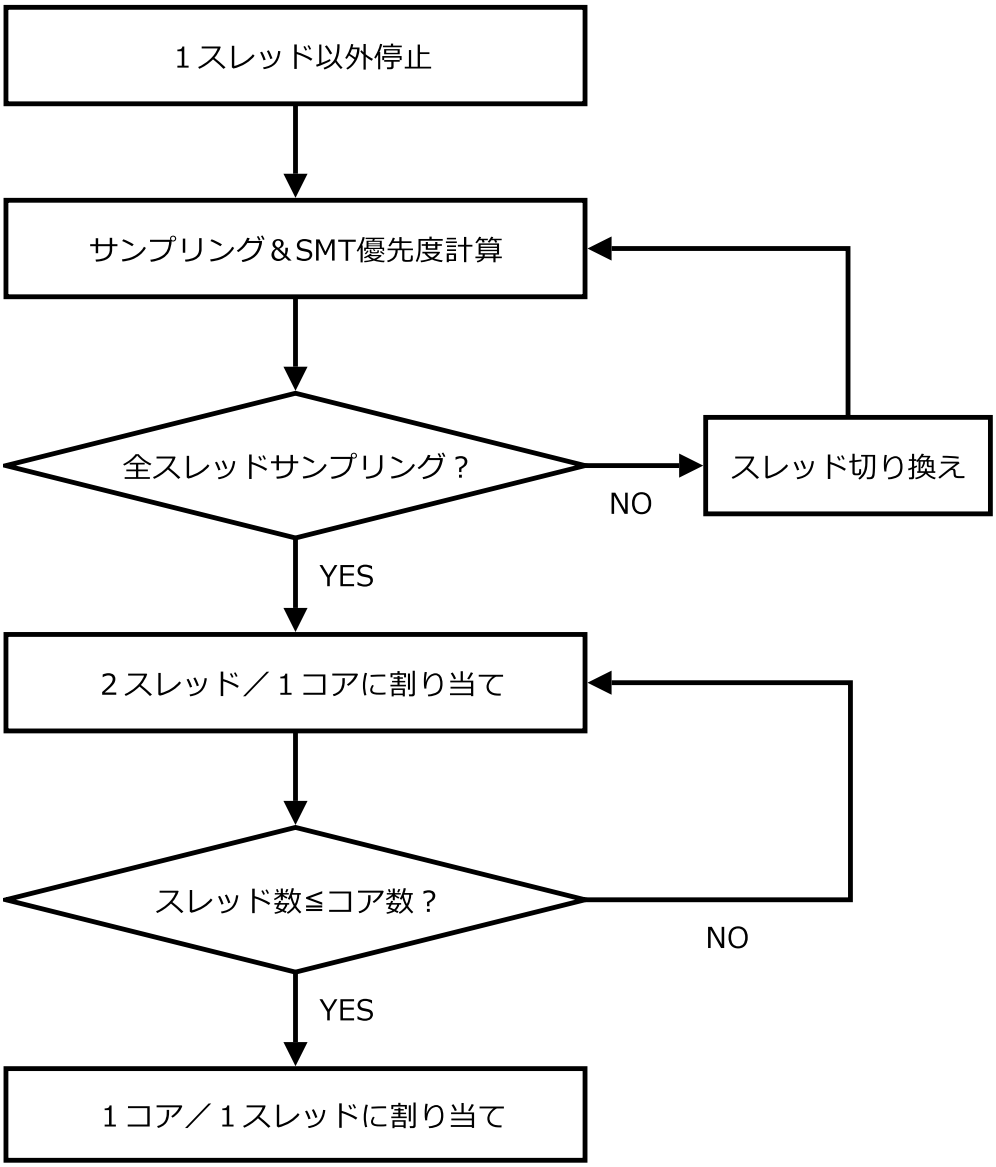

図 3.2　動的スレッドスケジューリングのフローチャート

第 4 章
性能評価

4.1 緒言

本章では，3 章で提案した SMT 優先度に基づくスレッドスケジューリング手法の効果を，シミュレーションによって評価する．まず，静的スレッドスケジューリングの有効性を評価し，次に動的スレッドスケジューリングの有効性を評価する．最後に，スレッド特徴量である SMT 優先度の検証および考察を行う．

4.2 静的スレッドスケジューリング手法

4.2.1 SMT 優先度の評価

最初に，静的スレッドスケジューリングにおける SMT 優先度の SMT 効率予測精度をシミュレーションによって評価する．まず，マルチコアプロセッサにおいて，2 スレッドを 1 コアで SMT を利用して実行する場合と，2 スレッドにそれぞれ 1 コアずつ割り当てて 2 コアを使って実行する場合の，プロセッサ全体の IPC および必要なパラメータを計測する．次に，計測した値を基に，2 つのスレッドの組み合わせに対する SMT 効率と，それぞれのスレッドの SMT 優先度を求め，2 つのスレッドの SMT 優先度の和と SMT 効率の関係を求める．SMT 優先度と SMT 効率に相関関係が存在すれば，2 つのスレッドの SMT 優先度の和が高い程，そのスレッドの組み合わせは SMT に適していると推測することができる．シミュレーションにはマルチコアシミュレータ M5 [29] を使用し，アプリケーションは，SPEC CPU2000 [30] に含まれている Bzip2，Gcc，Gzip，Mcf，Parser，Twolf，Ammp，Applu，

Apsi，Art，Equake，Mgrid，Swim，Wupwise の 14 種類を用いる．本実験では，1 つのアプリケーションを 1 つの独立したスレッドとして実行する．M5 は，スーパースカラのプロセッサコアを複数持つマルチコアプロセッサの動作をシミュレーションする．また，SMT に対応しており，複数のスレッドの命令を同時に発行することができる．本実験で用いるプロセッサの構成を表 4.1 に示す．SMT コンテキストは 1 コアで同時実行できるスレッドの数を示す．使用したベンチマークアプリケーションの概要を表 4.2 に示す．本章では，以後特に断りがない限りこれらの値を用いることとする．

本節では，静的スレッドスケジューリングにおいて SMT 優先度が有効に機能するアプリケーションと機能しないアプリケーションに分類し，その分類に基づいて議論する．まず表 4.2 に示す 14 種類のアプリケーションを以下の手順で分類する．ここで，グループ A は静的スレッドスケジューリングにおいて SMT 優先度が有効に機能するアプリケーションのグループ，グループ B は有効に機能しないアプリケーションのグループである．

- まず全てのアプリケーションをグループ A とする．
- 次に，グループ A の 1 つのアプリケーションを除外して，残りのアプリケーションのみでの相関係数を計算する．
- それをグループ A の全てのアプリケーションに関して行い，相関係数が最も高い場合に除外していたアプリケーションを，グループ B とする．
- これを，一般的に強い相関関係が認められると判断される相関係数 0.8 以上になるまで

表 4.1　プロセッサの構成

SMT コンテキスト	2
フェッチ命令数	8
発行命令数及びリタイア命令数	4
ロードユニット及びストアユニット数	2
整数演算ユニット数	3
浮動小数点演算ユニット数	2
L1 データキャッシュ	32KB
L2 キャッシュ	1MB

繰り返す．

このようにして得られたグループ A とグループ B の内訳を表 4.3 に示す．またそれぞれのグループの SMT 効率と SMT 優先度の和のグラフを図 4.1 と図 4.2 に示す．縦軸が SMT 効率（SMT Efficiency, SMTE），横軸が SMT 優先度（SMT Priority, SMTP）の和である．SMT 効率は 0 以上 1 以下の値となり，各スレッドの SMT 優先度は 0 以上 1 以下の値となるために SMT 優先度の和は 0 以上 2 以下の値となる．図 4.1 と図 4.2 より，グループ A のアプリケーションのみを用いた場合は相関が高く，グループ B のアプリケーションのみを用いた場合は相関が低くなっていることが確認できる．相関係数は，グループ A が 0.89，グループ B が 0.17 であった．次節より，各グループそれぞれのスレッドスケジューリングの性能を評価する．

表 4.2　ベンチマークアプリケーションの概要

名前	種類	概要
Bzip2	INT	データ圧縮
Gcc	INT	C コンパイラ
Gzip	INT	データ圧縮
Mcf	INT	ネットワークフロー最適化
Parser	INT	自然言語の情報処理
Twolf	INT	マイクロチップの配置と配線
Ammp	FP	化学計算
Applu	FP	放物型／楕円型偏微分方程式
Apsi	FP	温度，風，重力，汚染拡大の計算
Art	FP	ネットワークシミュレーション（適応共鳴法）
Equake	FP	有限要素シミュレーション（地震モデル）
Mgrid	FP	三次元ポテンシャル場のマルチグリッド計算
Swim	FP	浅瀬水域のモデリング
Wupwise	FP	量子色力学計算

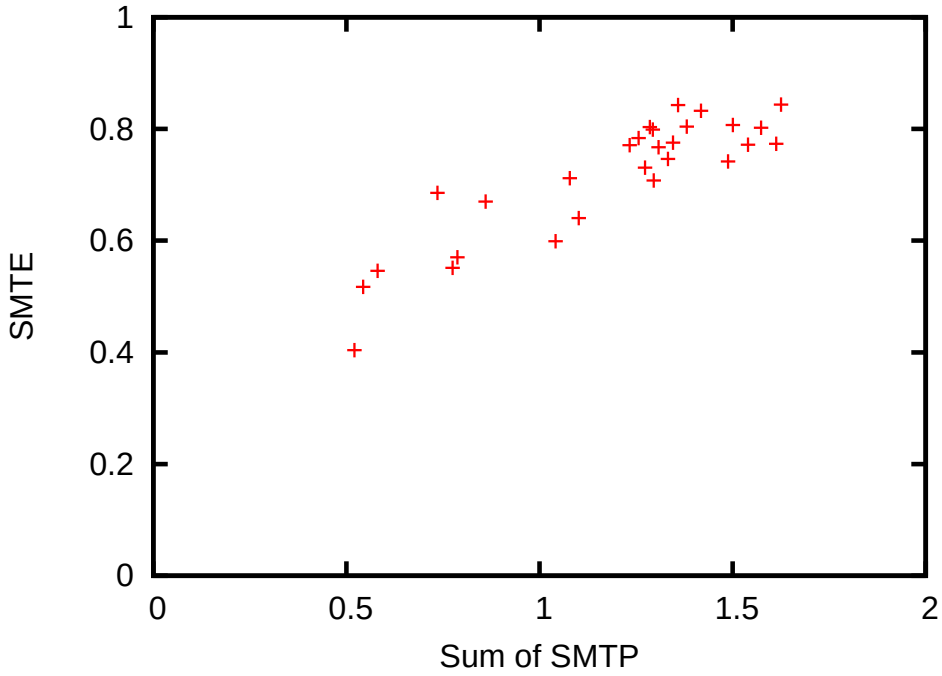

図 4.1　グループ A（相関あり）の SMT 優先度の和と SMT 効率の相関関係

4.2.2　スレッドスケジューリング手法の評価

提案手法の有効性をシミュレーションによって検証する．シミュレーションにはマルチコアシミュレータ M5 を使用する．プロセッサの構成は表 4.1 と同じである．シミュレーションでは 3 コアのマルチコアプロセッサで 4 スレッドを同時に実行する．これによって，2 つのスレッドが 1 コアで SMT を用いて実行され，残りの 2 つのスレッドがそれぞれ 1 コアで単独で実行されることになる．14 個のベンチマークを，前節で示したグループ A のベンチマーク 8 個と，グループ B のベンチマーク 6 個に分け，グループごとに実験を行った．

グループ A の 8 種類のベンチマークからスレッドとして実行する 4 つを選ぶ組み合わせは全部で 70 組，グループ B の 6 種類のベンチマークからスレッドとして実行する 4 つを選ぶ組

表 4.3　SMT 優先度と SMT 効率の相関係数による分類

グループ A	Bzip2	Gcc	Gzip	Twolf	Apsi	Mgrid	Swim	Wupwise
グループ B	Mcf	Parser	Ammp	Applu	Art	Equake		

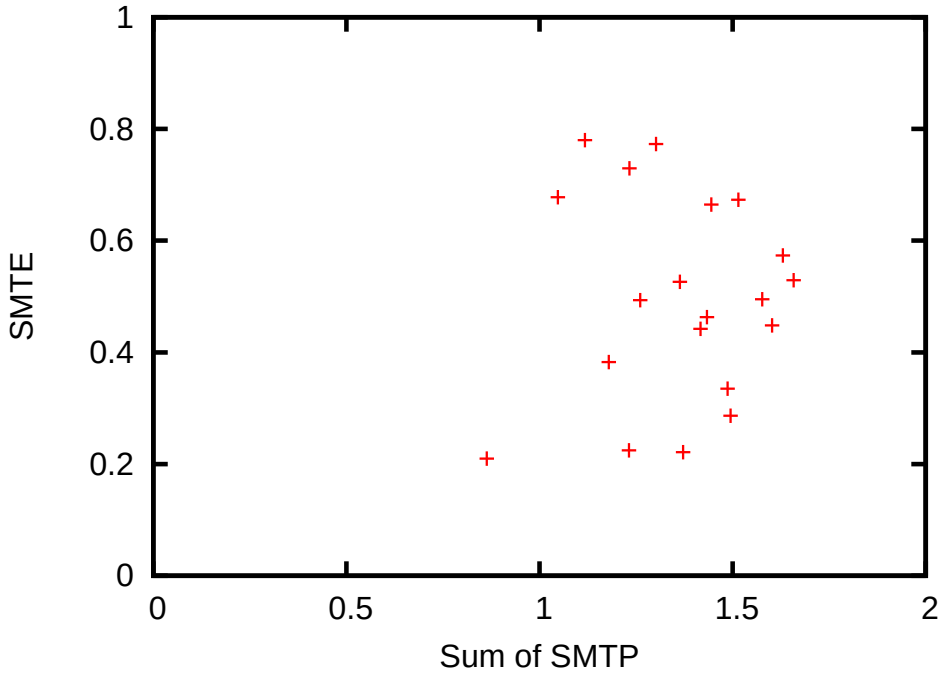

図 4.2　グループ B（相関なし）の SMT 優先度の和と SMT 効率の相関関係

み合わせは全部で 15 組あり，選んだ 4 スレッドを 3 コアに静的にスケジューリングする組み合わせは 6 種類あるので，その組み合わせすべてに対してシミュレーションを行い，提案手法の有効性を順位によって評価する．

グループ A，グループ B において，とり得るすべての組み合わせの中で何番目に高い IPC が得られたかを表 4.4 に示す．まずグループ A について考察し，その後グループ B について考察する．

表 4.4　各グループの順位

順位	1 位	2 位	3 位	4 位	5 位	6 位
グループ A（全 70 組）	53	14	2	1	0	0
グループ B（全 15 組）	5	6	3	1	0	0

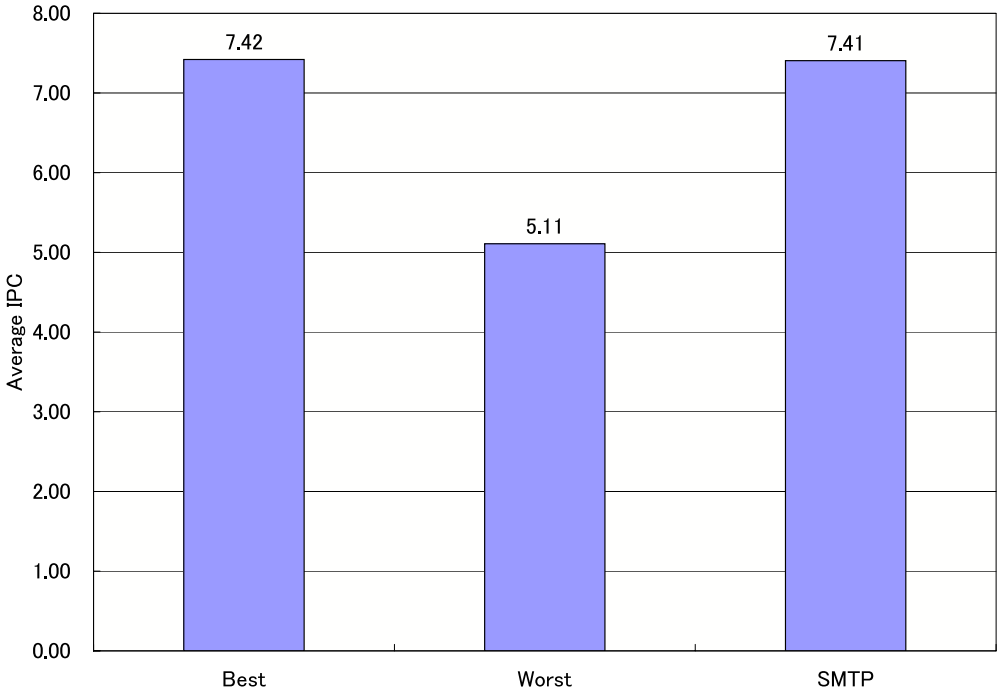

図 4.3　グループ A におけるプロセッサ全体の実効性能（IPC）の比較

グループ A

グループ A は 70 組中およそ 75.7% にあたる 53 組で最も性能がよく，全体のおよそ 95.7% にあたる 67 組が 2 番目までに高い性能のスケジューリングを選ぶことができた．さらに 5 番目，6 番目になったことはなかった．

次に 70 組のシミュレーション結果の平均値を図 4.3 に示す．それぞれの組で最も性能のよかったスレッドの組み合わせを性能の上限値として「Best」で示し，逆にそれぞれの組で最も性能の悪かったスレッドの組み合わせを性能の下限値として「Worst」で示している．提案手法は「Best」にはわずかに及ばないものの，「Best」の 99.8% の実効性能を達成している．表 4.4 で示した通り，53 組で最も性能のよい組み合わせを選択できただけでなく，2 番目以下の性能の組み合わせを選んだ 17 組でも，その性能は最もよい組み合わせに非常に近かったためであることがわかる．

グループ A の Apsi，Swim，Gcc，Gzip を同時実行する場合に，提案手法は 4 番目に良い組み合わせを選択した．前述のように，この 4 つのアプリケーションを 3 コアにスケジューリングする組み合わせは 6 通りある．6 通りの組み合わせそれぞれの IPC を図 4.4 に示す．こ

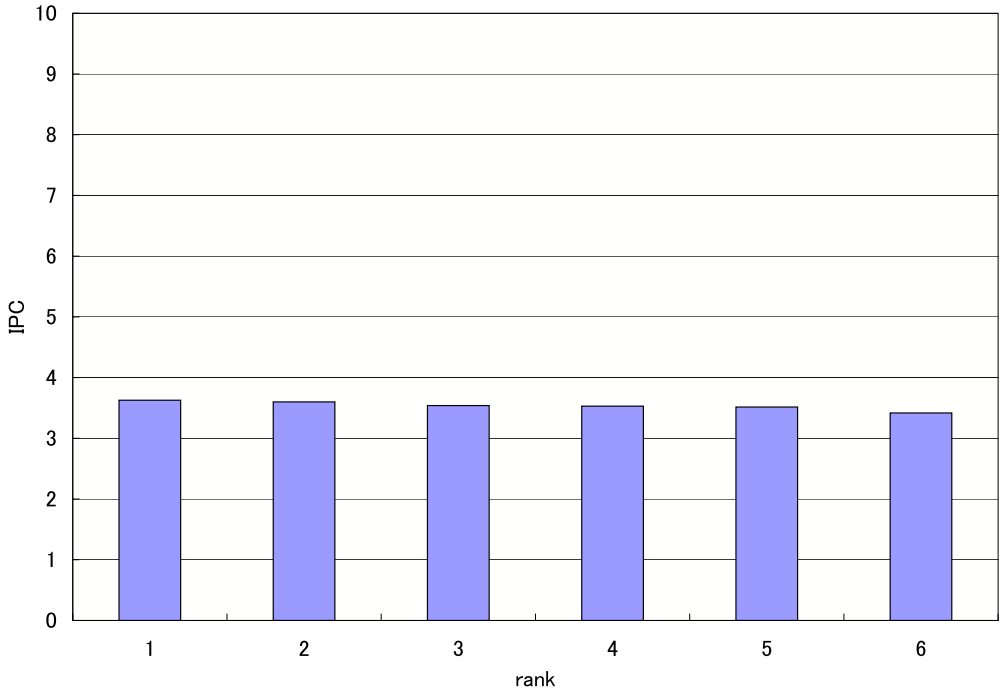

図 4.4　Apsi・Swim・Gcc・Gzip（グループ A で 4 位）の全組み合わせ 6 通りの IPC

の組み合わせにおいて，最適なスケジューリングが行われた場合に得られる 1 位の実効性能（IPC）は 3.63 であり，SMT 優先度に基づくスレッドスケジューリングにより選択した 4 位の実効性能は 3.53 である．この差はわずか 0.10 であり，この選択による実効性能の低下は極めて小さい．次に，グループ A において提案手法が 1 位を選択した 53 組のうちの一つとして，Twolf，Swim，Gzip，Wupwise の組み合わせにおける 6 通りの IPC を図 4.5 に示す．この組み合わせでは，SMT 優先度に基づくスレッドスケジューリングにより選択した 1 位の実効性能（IPC）は 9.53 であるが，4 位を選択した場合の実効性能は 7.54 であり，その差は 1.99 もある．もしこの組で 1 位ではなく 4 位を選択した場合，その損失は非常に大きい．また，他の組み合わせでも同様の傾向が見られた．

したがって，提案手法は，スケジューリングによって実効性能が大きく変動する組み合わせに対して最適な組み合わせが選択できたために，グループ A のどのような組み合わせに対しても，最大もしくは最大に非常に近いスループットを達成することができたと言える．これにより SMT 優先度がうまく機能するグループ A に属するアプリケーションに対して，提案手法による静的スレッドスケジューリングが非常に有効であることが明らかになった．

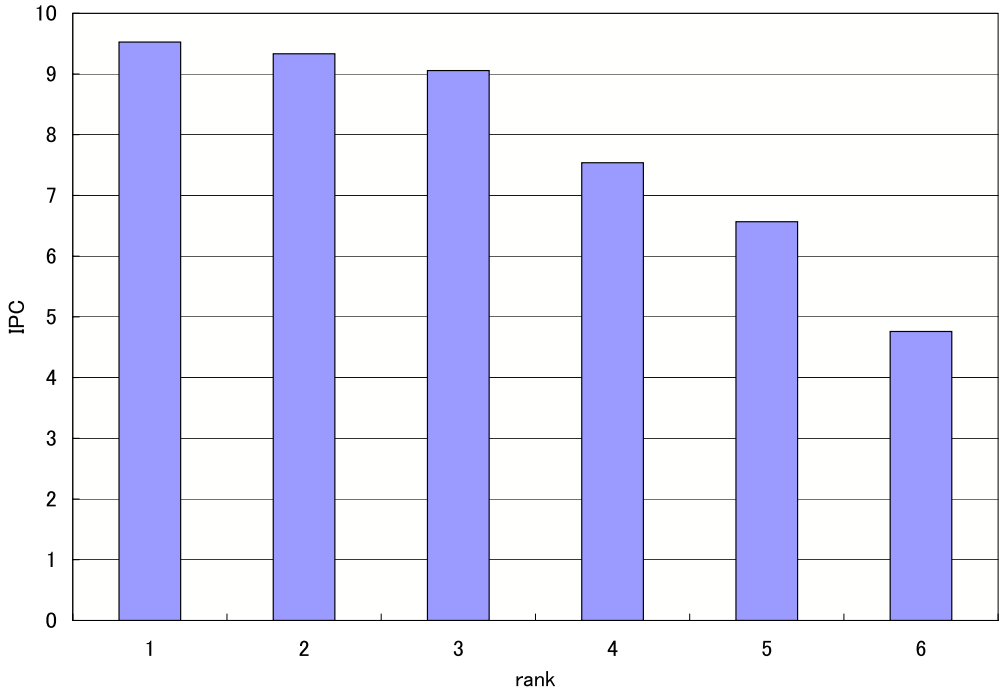

図 4.5　Twolf・Swim・Gzip・Wupwise（グループ A で 1 位）の全組み合わせ 6 通りの IPC

グループ B

グループ B は 15 組中およそ 33% にあたる 5 組で最も性能がよく，全体のおよそ 73% にあたる 11 組が 2 番目までに高いスケジューリングを選んだ．グループ A 同様，グループ B でも 5 番目，6 番目になったことはなかった．

次に，15 組のシミュレーション結果の平均値を図 4.6 に示す．それぞれの組で最も性能のよかったスレッドの組み合わせを性能の上限値として「Best」で示し，逆にそれぞれの組で最も性能の悪かったスレッドの組み合わせを性能の下限値として「Worst」で示している．提案手法は「Best」には及ばないものの，「Best」の 96.6% の実効性能を達成している．しかし，グループ A と比較すると優れたスケジューリング結果とは言えない．

グループ B において提案手法が 4 位を選択した，Mcf, Equqke, Parser, Applu の組み合わせにおける 6 通りの IPC を図 4.7 に示す．最適なスケジューリングが行われた場合に得られる 1 位の実効性能と，SMT 優先度に基づくスレッドスケジューリングにより選択した 4 位の実効性能との差は 0.59 と大きく，この組では SMT に不適切な組み合わせを選択してしまったと言える．これは，グループ B では SMT 優先度によって正しい性能予測ができず，適切な

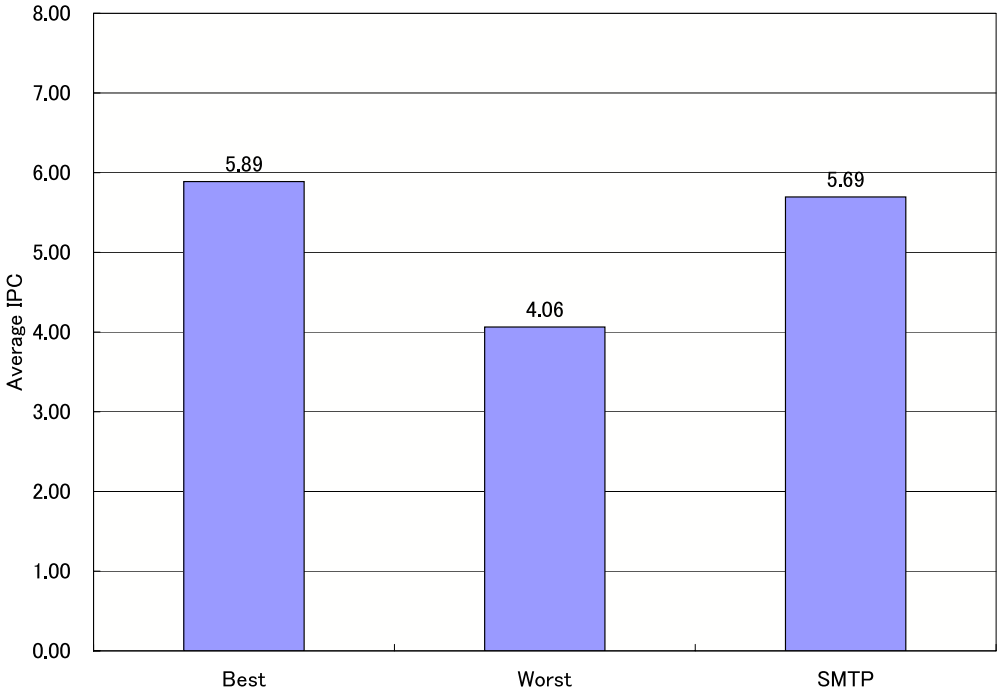

図 4.6　グループ B におけるプロセッサ全体の実効性能（IPC）の比較

組み合わせを選択できなかったためである．

ここで，SMT 優先度によって正しい性能予測ができない場合がある理由を考察する．一般に，アプリケーションの実効性能は，時間が進むにつれて変化する．一方，静的スレッドスケジューリングでは，実行結果全体の平均値から SMT 優先度を計算する．このことから，アプリケーションの時間変化を，静的スレッドスケジューリングでは考慮できないことが理由の一つとして考えられる．例として，Parser の実行中の IPC の変化を図 4.8 に示す．Parser は，アプリケーションの実行中の IPC の変動が大きく，400 万命令付近までは IPC が高いが，それ以降は IPC が低くなる．このようなアプリケーションに対し，同じく実行の前半部分で IPC が高くなるようなアプリケーションが組み合わされた場合，予測以上の競合が発生して，性能が低下する恐れがある．この問題を解決するには，動的スレッドスケジューリングによって，アプリケーションの実効性能の変化に応じてスケジューリングを行う必要がある．次節では，SMT 優先度と SMT 効率を一定時間ごとに計算し，動的スレッドスケジューリングによってこの問題が解決できる可能性を示す．

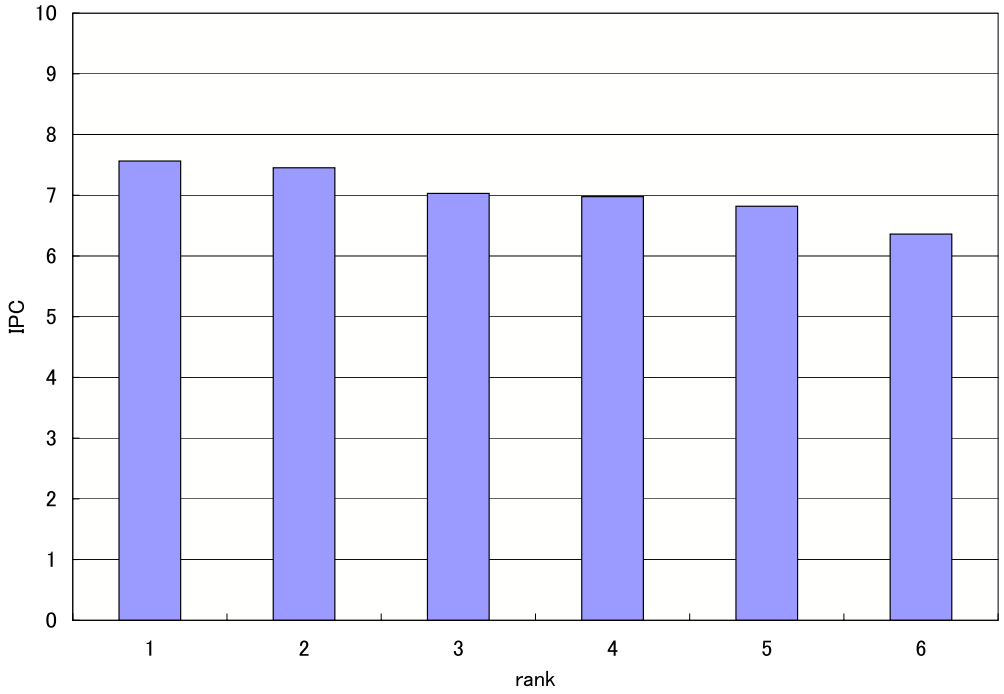

図 4.7　Mcf・Equake・Parser・Applu（グループ B で 4 位）の全組み合わせ 6 通りの IPC

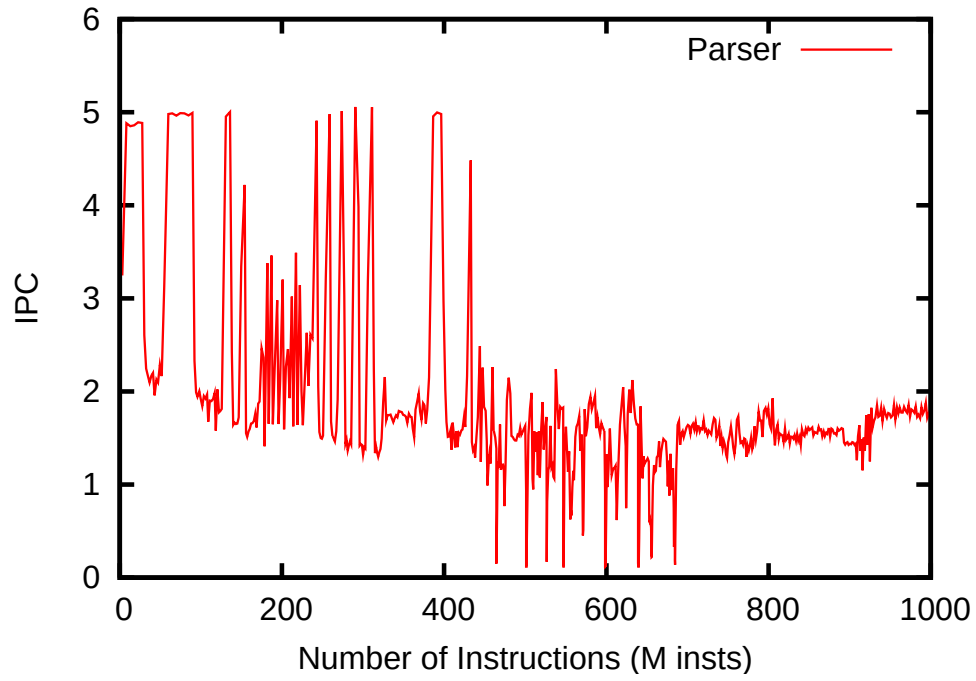

図 4.8　Parser の IPC の時間変化

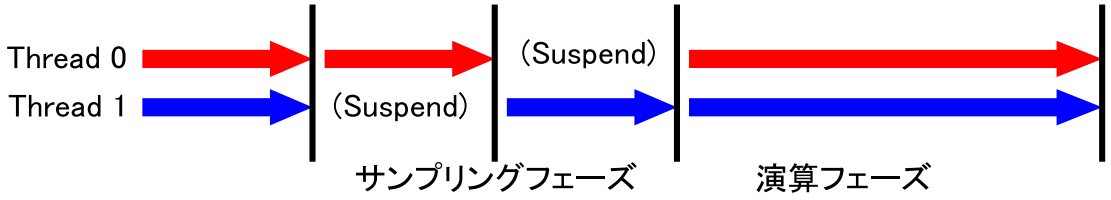

図 4.9　動的スレッドスケジューリングにおける SMT 優先度の評価方法

4.3　動的スレッドスケジューリング手法

本節では，4 つの組み合わせを例にして，3 章で提案した動的スレッドスケジューリング手法における SMT 効率予測性を詳細に評価する．評価方法は以下の通りである．また，評価方法の概要を図 4.9 に示す．

- 1 コアの SMT プロセッサで 2 スレッドを同時実行する．
- 演算 100 万サイクルごとにサンプリングフェーズに入り，そこでは片方のスレッドが停止して，もう片方のスレッドが単独で実行された状態になる．
- サンプリングフェーズは 1 万サイクルで，その間に SMT 優先度の計算に必要な値を取得する．
- 2 スレッドのサンプリングが完了したら，すぐに両方のスレッドの演算を再開し，そこから再び演算フェーズを 100 万サイクル行い，その間の SMT 効率を計測する．

表 4.3 のグループ A に含まれる Swim と Wupwise の結果を図 4.10 に示す．横軸は時間（×100 万サイクル），縦軸は SMT 効率もしくは SMT 優先度の値である．SMT 効率を緑線で，SMT 優先度の和を赤線で示した．緑線が時間経過に従って大きく変動していれば，そのアプリケーションの組み合わせは，動的に性能が変動することを示している．さらに，理想的な SMT 優先度の和と SMT 効率は比例関係にあるので，緑線の変化に赤線の変化が追従していれば，それは動的スレッドスケジューリングの各演算フェーズで，適切なスレッドの組み合わせを選ぶことが可能であることを示している．

Swim と Wupwise はともにグループ A に属するアプリケーションであり，静的に性能が予測可能であった組み合わせである．図 4.10 より，SMT 効率の時間変化がとても小さいこと

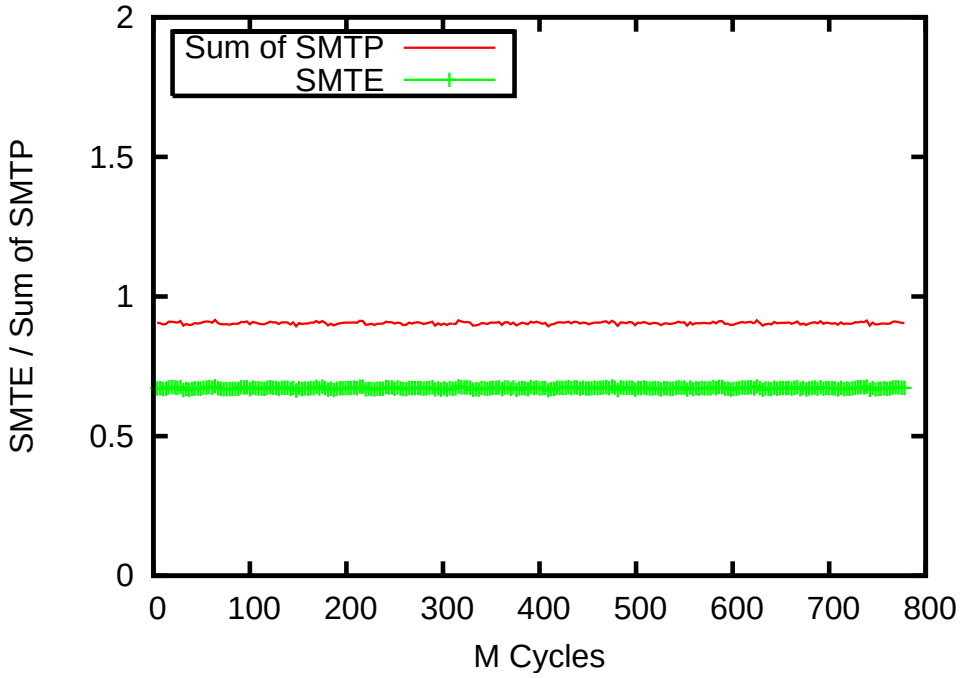

図 4.10　Swim-Wupwise（グループ A）の SMT 効率と SMT 優先度の和の時間変化

がわかる．このため，静的に SMT 効率を予測できたものと考えられる．SMT 優先度の和も SMT 効率と同様に時間変化がほどんどない．このことから，動的に予測する場合にも SMT 優先度の和によって SMT 効率の変化を適切に予測できており，各演算フェーズでの適切な組み合わせ選択を期待できる組み合わせであるといえる．

次に，静的に SMT 効率を予測困難なグループ B に属する Ammp と Mcf を組み合わせた結果を図 4.11 に示す．この組み合わせでは，SMT 効率が時間によって全く異なる振る舞いをしている．0M サイクルから 600M サイクルまでは SMT 効率が一定の振幅で振動するように変化し，600M サイクル付近からは SMT 効率がおよそ 0.8 で一定となっている．さらに 1000M サイクル付近からは急激に SMT 効率が下がり，そのまま 2400M サイクル付近まで徐々に SMT 効率が下がっている．この振る舞いを平均化した一つの指標で示すことは非常に困難であり，このような不規則かつ大きな性能の変化が，この組み合わせに対する静的スレッドスケジューリングでの SMT 優先度の和と SMT 効率の相関が低い理由であると考えられる．一方 SMT 優先度の変化を見ると，600M サイクルから 1000M サイクルの部分で，SMT 効率が高くなっているのに対して，SMT 優先度の和が低くなっている．

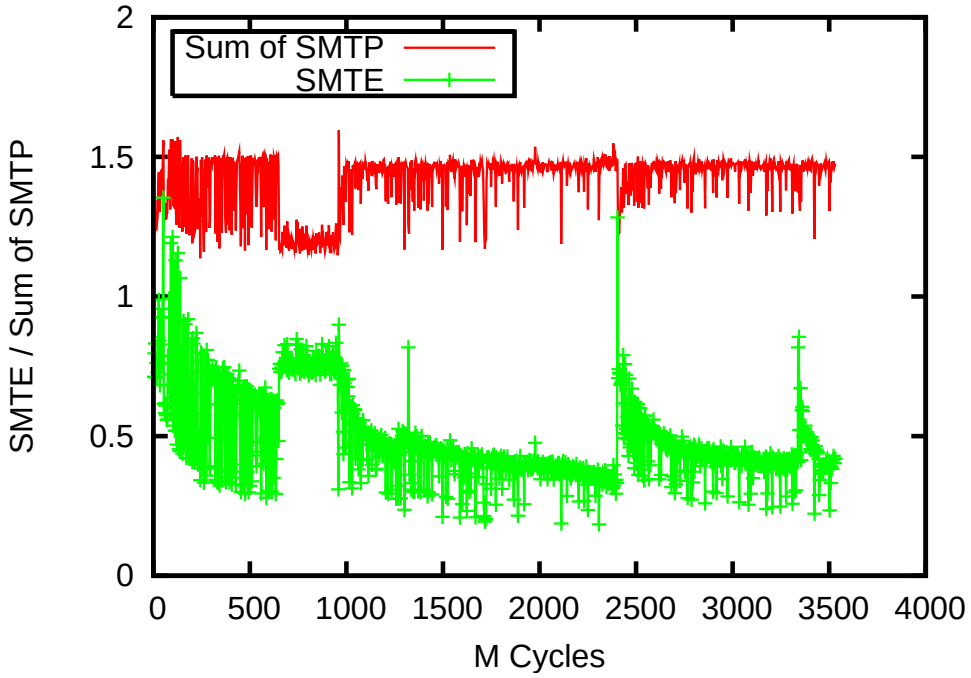

図 4.11　Ammp-Mcf（グループ B）の SMT 効率と SMT 優先度の和の時間変化

次に，この部分で性能予測が大きく外れている原因を考察する．まず，組み合わせた時の性能に大きな影響を及ぼしているアプリケーションを特定するために，Ammp を単独で実行した時の実効性能（IPC）の時間変化を図 4.12 に，Mcf を単独で実行した時の実効性能（IPC）の時間変化を図 4.13 に示す．横軸が実行命令数，縦軸が実効性能（IPC）である．図 4.13 および図 4.12 より，Mcf では実効性能の時間変化が少なく Ammp では実効性能の時間変動が大きいことが分かる．図 4.11 に示した，Ammp と Mcf を組み合わせた時の SMT 効率の時間変動が，図 4.12 に示す Ammp 単独の時間変動と非常に似ていることから，Ammp が組み合わせたときの実効性能に大きな影響を及ぼしていると考えられる．さらに，図 4.11 における 600M サイクルから 1000M サイクルの SMT 効率が高くなっている部分は，図 4.12 における 200M 命令から 400M 命令の IPC が高くなっている部分に対応していると考えられる．

次に，600M サイクルから 1000M サイクルの部分における SMT 優先度の変化を検証する．図 4.14 に Ammp を単独で実行した時の L1 データキャッシュミス率の時間変化を示す．横軸が実行命令数，縦軸が L1 データキャッシュのキャッシュミス率である．また，図 4.15 に組み合わせて実行したときの Ammp の C_{busy} と C_{instQ} の時間変化を示す．C_{busy} と C_{instQ} は SMT 優先度に用いている特徴量であり，C_{busy} が大きいほど発行命令数が多く，C_{instQ} が大

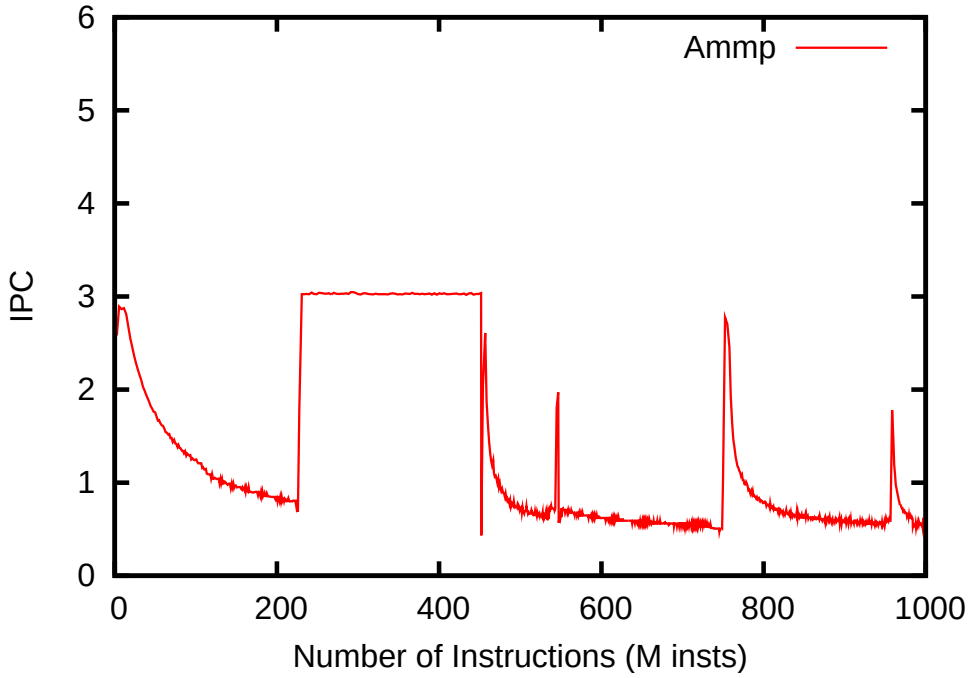

図 4.12　Ammp の実効性能（IPC）の時間変化

きいほど命令キューでの待ち時間が長いことを示す．図 4.14 より，注目する部分では Ammp の L1 データキャッシュミス率が低下したため，Ammp の発行命令数が増加したと考えられる．よって図 4.15 が示す通り C_{busy} の値が増加し，その結果 SMT 優先度が低下した．発行命令数が多いスレッドは，演算器での他のスレッドとの競合が予想されるので SMT に向いていないと考えれることから，この部分で SMT 優先度が低下したことは期待した通りの変化であると言える．

しかし，実際には SMT 効率は低下するという予想に反して上昇した．これは，注目する部分では Ammp の発行命令数が増加したにも関わらず，Mcf との演算器での競合が少なかったために，実効性能が低下しなかったことを示している．演算器での競合が発生しにくい要因として，Mcf は整数演算型のベンチマークであり Ammp は浮動小数点演算型のベンチマークであるため，2 つのアプリケーションが頻繁に利用する演算器の種類が異なっていたという可能性がある．この問題に対応するには，整数演算器，浮動小数点演算器を区別してそれぞれの競合を予測するような指標の検討が必要である．

最後に，600M サイクルから 1000M サイクル以外の部分における SMT 優先度の変化を検証する．C_{instQ} の割り合いが高い箇所として，図 4.11 の 500M サイクルから 600M サイクル

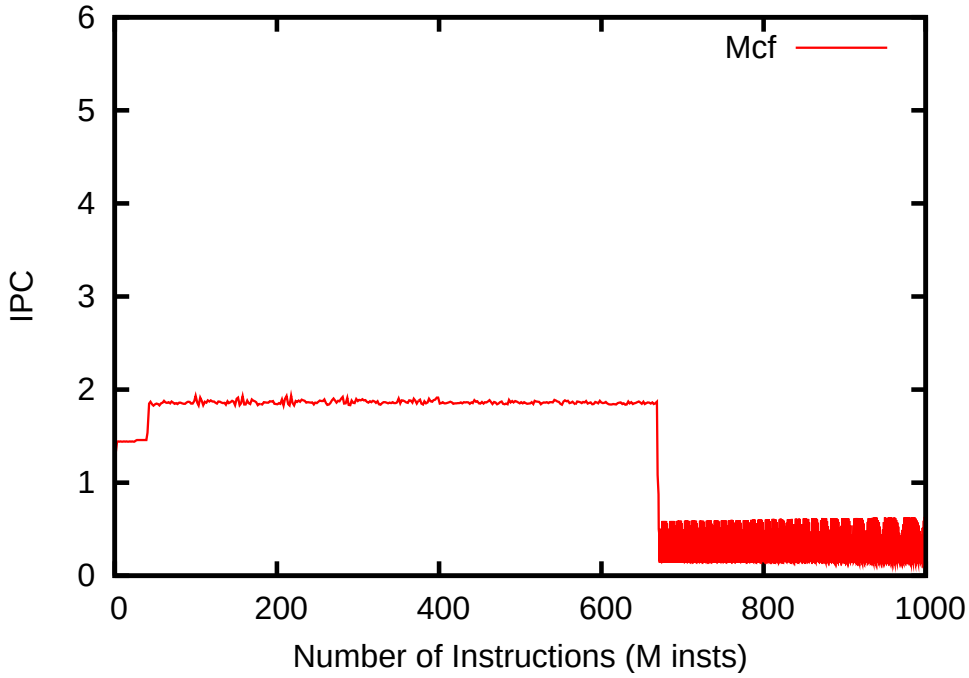

図 4.13　Mcf の実効性能（IPC）の時間変化

と 1000M サイクルから 1100M サイクルの部分を拡大した図を，図 4.16 と図 4.17 に示す．図 4.16 および図 4.17 共に，この部分では SMT 優先度の和が SMT 効率を追従していると言える．このことから C_{instQ} として考慮している命令キューでの競合による性能予測はうまく機能していると考えることができる．

以上の考察をまとめる．動的スレッドスケジューリングを行った場合に，各演算フェーズで適切なスレッドの組み合わせが選択できる可能性を，サンプリングした SMT 優先度と直後のインターバルの SMT 効率との相関から評価した．SMT 優先度が有効に機能するアプリケーションでは，提案手法のスレッドスケジューリングによって高い実効性能を達成できることが図 4.3 から明らかであり，動的スレッドスケジューリングによって，静的スレッドスケジューリングと同等以上の高い実効性能が達成できることを期待できる．一方，動的な変動を考慮しても SMT 優先度では性能を正確に予測できない組も存在する．そのようなアプリケーションでは，時間変化に対応するために細かく時間を区切っただけでは不十分であり，正確な性能予測のためには他の要素も考慮する必要あると考えられる．次節でそれを考察する．

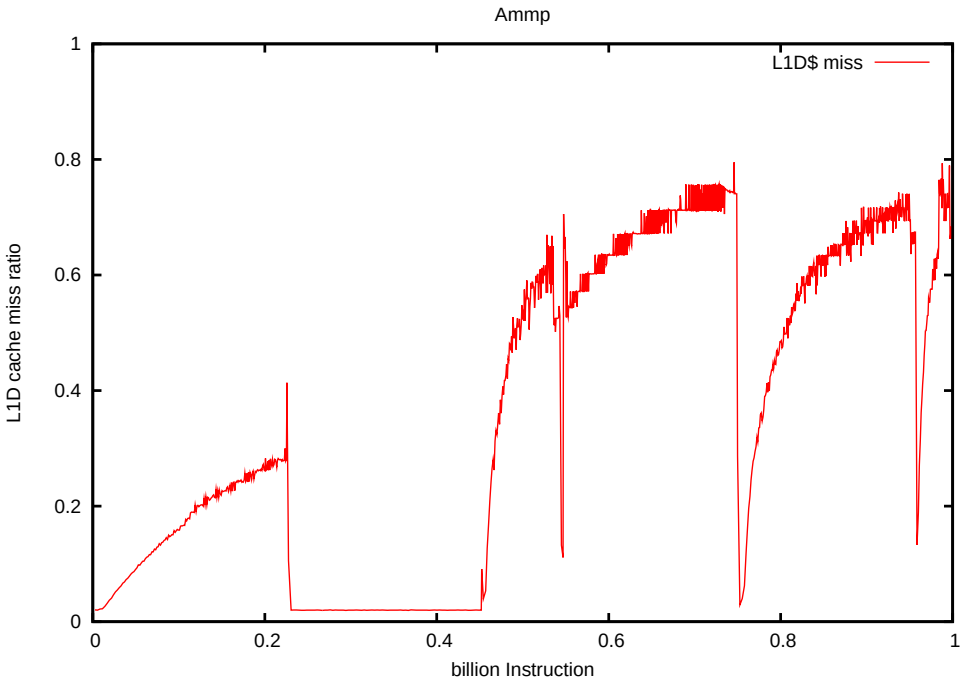

図 4.14　Ammp の L1 データキャッシュミス率の時間変化

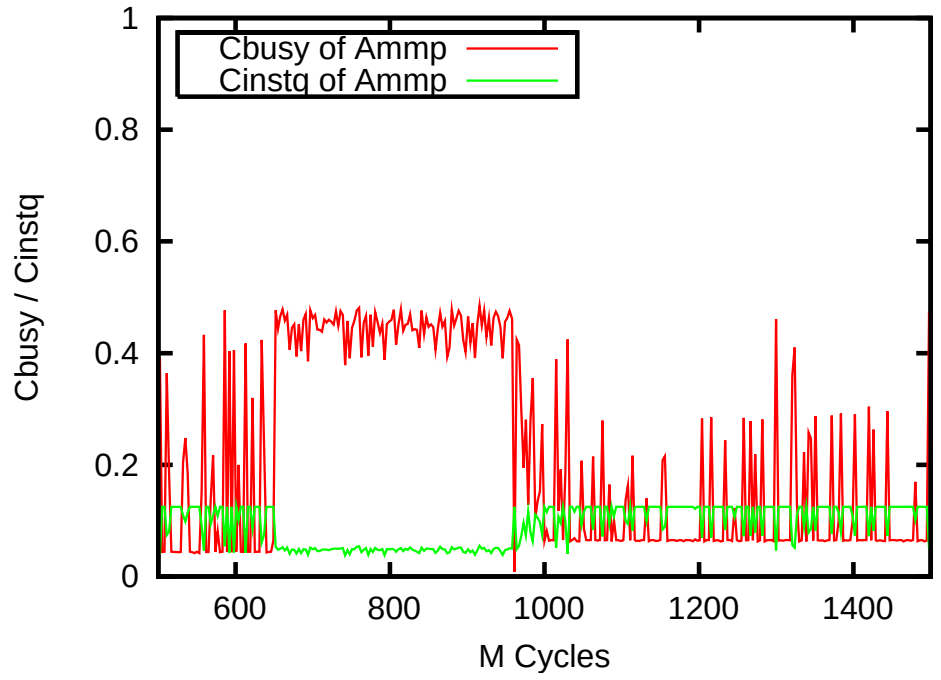

図 4.15　Ammp-Mcf における Ammp の C_{busy} と C_{instQ} の時間変化

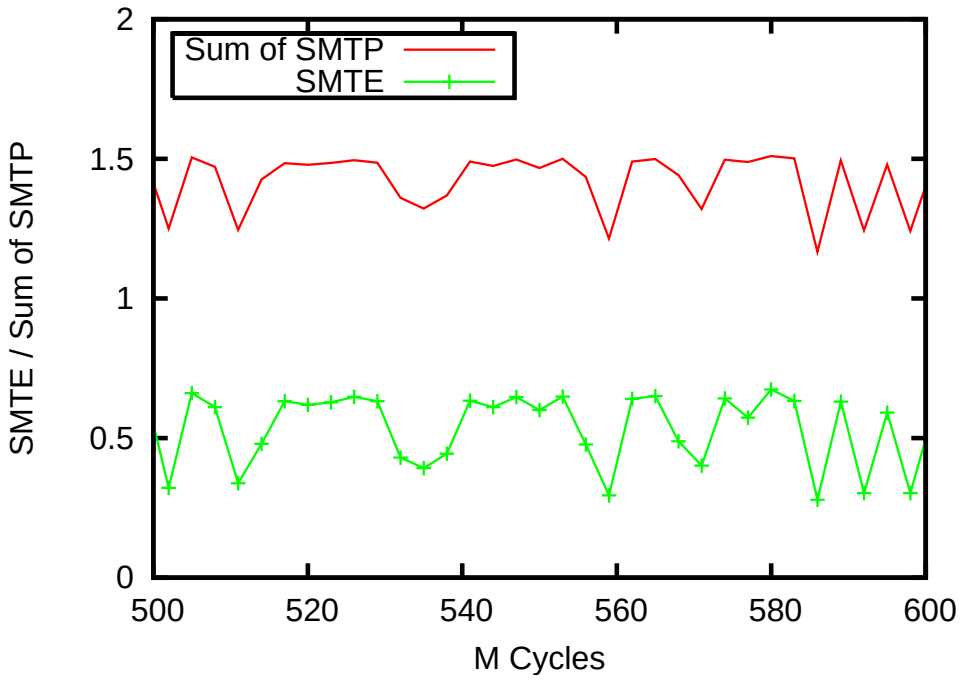

図 4.16　Ammp-Mcf の SMT 効率と SMT 優先度の和の時間変化（拡大図 1）

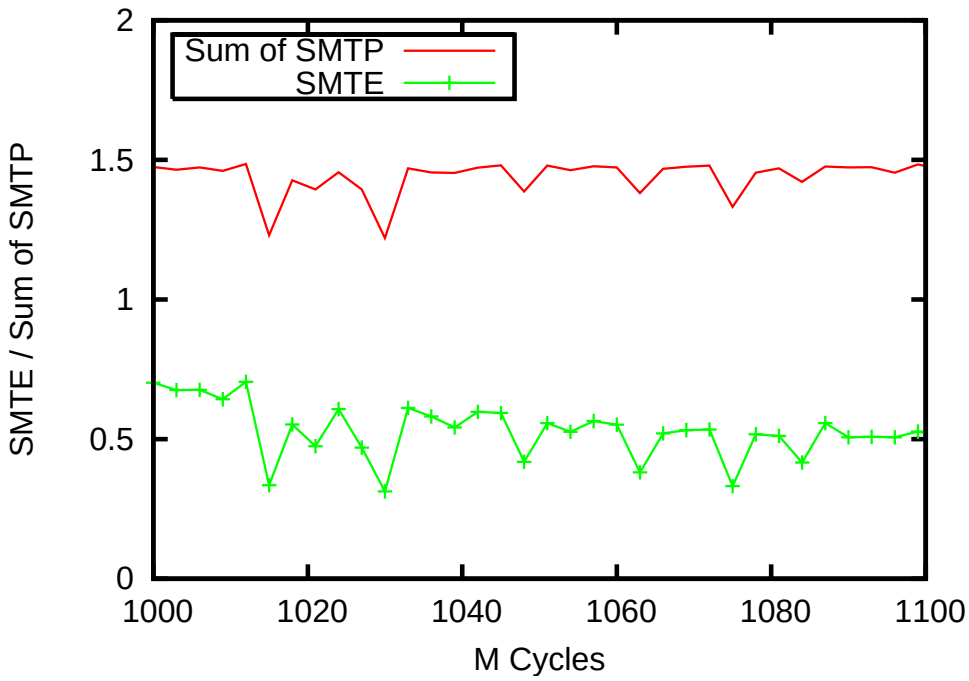

図 4.17　Ammp-Mcf の SMT 効率と SMT 優先度の和の時間変化（拡大図 2）

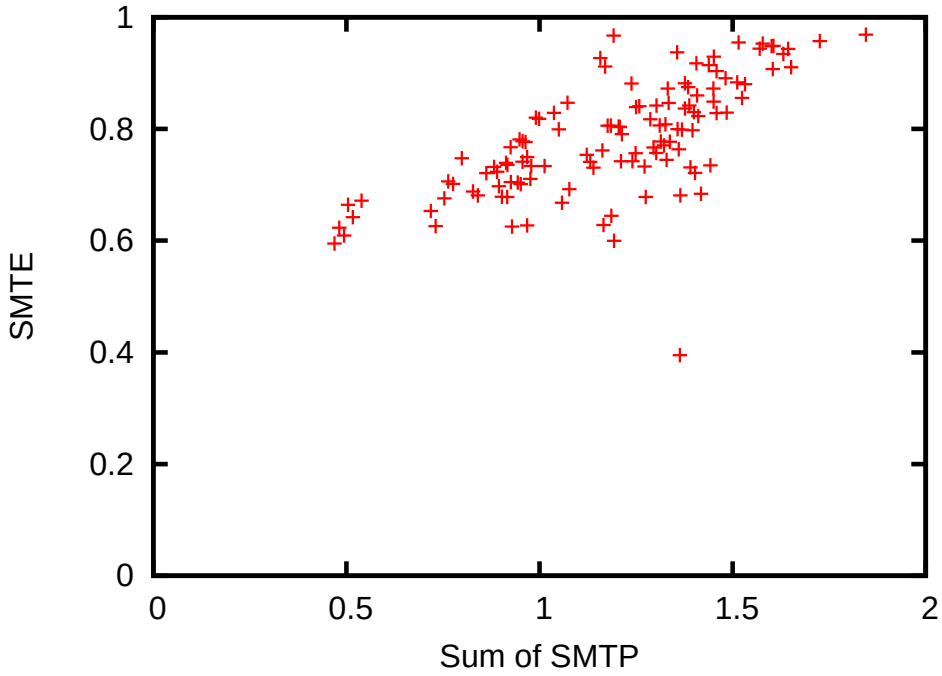

図 4.18　L1 および L2 キャッシュのサイズをそれぞれ 2 倍にした場合の SMT 優先度の和と SMT 効率の相関関係

4.4　考察

SMT 優先度の和と SMT 効率の相関が低いアプリケーションが，どのような特徴を持っているのか調査するために，プロセッサの構成を変えてシミュレーションを行った．SMT 優先度では，スレッド特徴量としてアプリケーションの演算器の利用率を基礎としている．ロード命令・ストア命令といったメモリ操作の命令については，ロード・ストアバッファを見ることで考慮しているが，メモリ操作の結果を直接表すパラメータ，例えばキャッシュアクセス数やキャッシュミス率などは用いていない．そこでメモリアクセスが SMT 優先度に及ぼす影響を調査するために，L1 データキャッシュのサイズと L2 キャッシュのサイズを，表 4.1 に示した構成の 2 倍にしてシミュレーションを行った．

結果を図 4.18 に示す．表 4.2 中の 14 アプリケーションすべての組み合わせにおける相関係数は 0.67 であり，表 4.1 の構成時よりも高い相関を示している．実際の相関係数も 0.67 と最初に比べて高くなっている．キャッシュミスが減ったことで，相関が低かった一部のアプリ

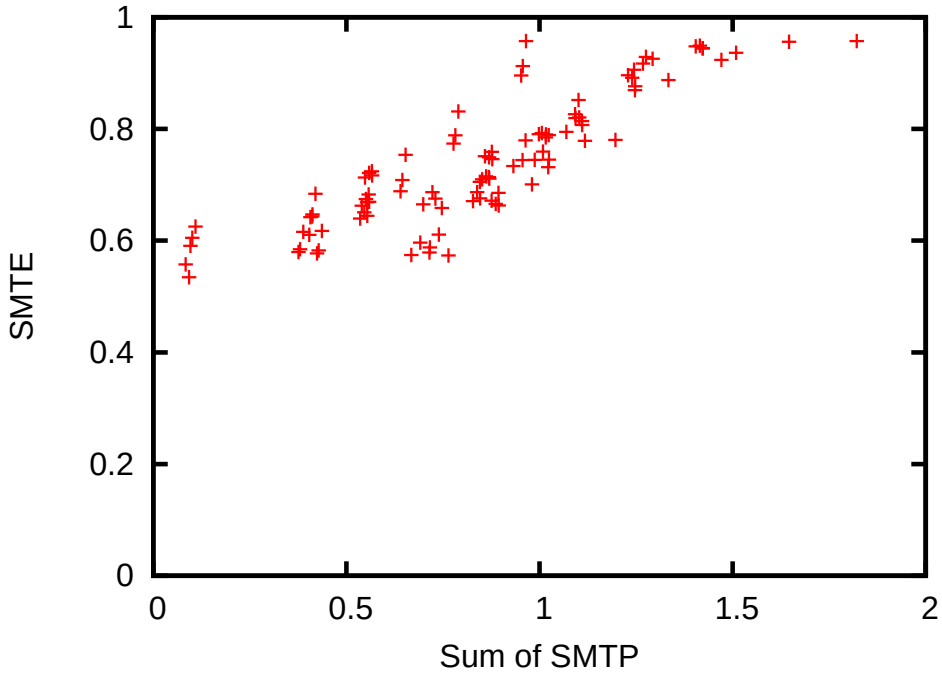

図 4.19　L1 キャッシュミスを排除した場合の SMT 優先度の和と SMT 効率の相関関係

ケーションの相関が高くなったと考えられる．このことをさらに明確にするために，次に L1 データキャッシュにミスのない完全キャッシュを仮定してシミュレーションを行った．ただし，シミュレータの構成上，L1 キャッシュの容量を無限大にはできないため，L1 キャッシュの次の階層にすぐメインメモリを接続し，かつメインメモリへのアクセスにかかるレイテンシを 1 サイクルとして実験を行った．結果を図 4.19 に示す．図 4.18 よりもさらに相関が高い．14 アプリケーションでの相関係数は 0.86 であり，これは充分に高い相関であると言える．

ここまでの考察と追加実験により，次の事が明らかになった．SMT 優先度から SMT 効率を予測できないアプリケーションにおいても，プロセッサの L1 キャッシュが完全キャッシュであれば予測可能となる．この結果より，SMT 優先度では，2 つのスレッドを組み合わせて実行する場合に増加するキャッシュミスレイテンシの影響が，SMT 優先度の値に反映されない場合があり，そのために SMT 優先度の予測精度が低下している可能性がある．さらに，SMT 優先度が有効に機能するグループ A と，有効に機能しないグループ B は，キャッシュミスレイテンシの特徴によって分類できると推測される．

以上のことを検証するため，各アプリケーションを単独で実行した場合のキャッシュミスによるレイテンシを比較する．ここで，キャッシュミスによるレイテンシとは，キャッシュア

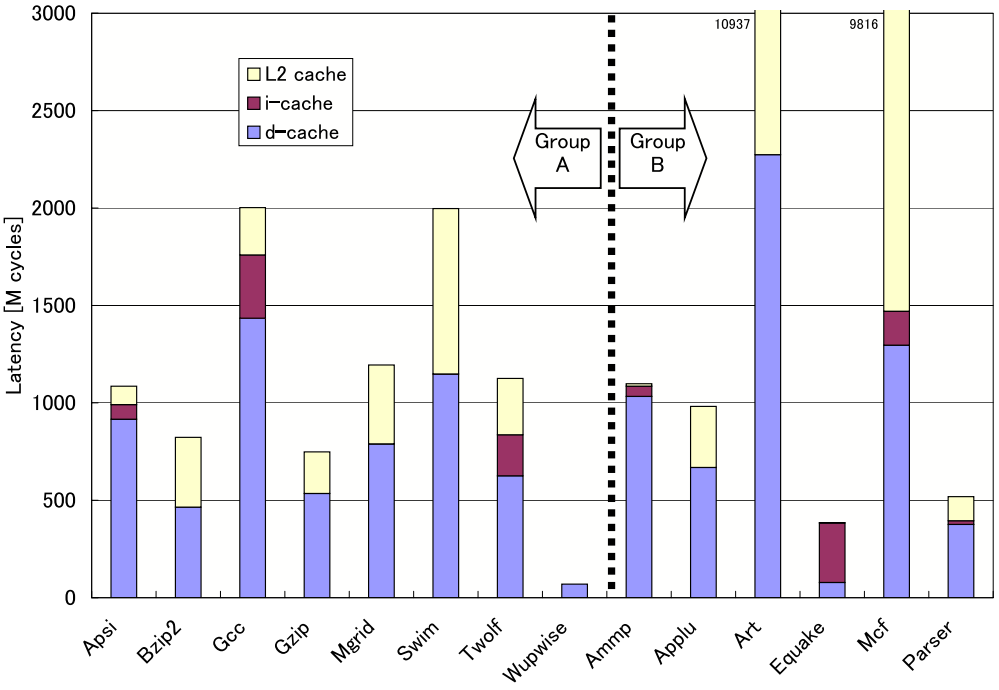

図 4.20　各ベンチマークのメモリアクセスによるレイテンシ

クセス回数（Memory access），キャッシュミス率（Miss rate），キャッシュミスペナルティ（Miss penalty）を用いて以下の式で計算される．

$$\text{Latency} = \text{Momory access} \times \text{Miss rate} \times \text{Miss penalty} \tag{4.1}$$

各アプリケーションのキャッシュミスレイテンシを図 4.20 に示す．破線の左側がグループ A，右側がグループ B のアプリケーションである．図 4.20 より，Art と Mcf はミスレイテンシが非常に大きく，SMT 優先度がこれを充分に考慮しきれなかったために，相関が低くなったと考えられる．しかし，Ammp，Applu，Equake，Parser は，グループ A のアプリケーションと比較して特別レイテンシが大きいとは言えない．キャッシュミスによる性能低下示す他の指標の検討が必要である．

キャッシュミスによる性能低下の評価法のさらなる検討と，その影響を適切に表現できるように SMT 優先度を改良することが今後の課題である．

4.5 結言

本章では，SMT 優先度の評価と，提案手法である SMT 優先度に基づくスレッドスケジューリング手法の評価を行った．まず，静的スレッドスケジューリングでは，SMT 優先度が有効に機能しない一部のアプリケーションを除いて，提案手法の SMT 優先度に基づくスレッドスケジューリングが有効であることを示した．次に，動的スレッドスケジューリングを行うことで，静的な場合と同等以上の性能向上が見込めることを示した．最後に，現在 SMT 優先度によって充分な性能予測ができない一部のアプリケーションについて，その特徴を明らかにし，SMT 優先度をさらに多くのアプリケーションに対応させるために必要な課題を示した．

第 5 章
結論

5.1 本論文のまとめ

プロセッサコア上の潤沢なリソースを有効に利用するアーキテクチャとして，SMT コアで構成されたマルチコアプロセッサが注目されている．プロセッサに対するスレッドスケジューリングを行うのは OS であるが，OS が物理的なコアと論理的なコアを区別したスケジューリングを行うようになったのはごく最近である．また，現在の OS のスレッドスケジューリングは，コア間の極端な負荷の偏りを解消することを目的としており，プロセスの移動には非常に消極的である．一方，SMT において同時に実行するスレッドの組み合わせは，実効性能に大きな影響をあたえる大事な要因である．よって，スレッドの特徴とアーキテクチャの特徴を考慮した積極的なスレッドスケジューリングを行うことが必要不可欠である．

2 章では，OS が行っている現在のスレッドスケジューリングについて述べ，その問題点を指摘した．次に，事前のプロファイリング情報を基に一度だけスケジューリングを行う静的スレッドスケジューリングと，事前プロファイリングなしで，実行の途中でサンプリングとスケジューリングを繰り返す動的スレッドスケジューリングについて述べ，それぞれの長所と短所を示した．最後にスレッドスケジューリングの関連研究について概説しその問題点を述べた．その結果，高効率なスレッドスケジューリングを実現するためには，単独のスレッドから得られるシンプルで正確なスレッド特徴量が必要であることがわかった．

3 章では，SMT 優先度に基づく静的および動的スレッドスケジューリング手法を提案した．SMT 優先度は，複数のスレッドを 1 つの SMT コアで実行した時に，演算器などのリソース競合によって起きる性能低下を予測する特徴量である．スレッドを単独で実行しプロファイリ

ングすることで得られるため，サンプリングの回数はスレッド数分でよいという利点と，各リソースでの競合を命令発行キューのみから予測できるという利点がある．

4 章では，静的スレッドスケジューリングと動的スレッドスケジューリングの有効性を評価した．3 コアの SMT コアを持つマルチコアプロセッサに対し，4 スレッドを静的にスケジューリングする実験を行い，その結果，70 組のスレッドの組み合わせのうち，53 組で理想的な静的スレッドスケジューリングを実現した．特に組み合わせによって性能の変動が大きいスレッドに対して適切なスケジューリングが行われており，本手法の有効性が示された．さらに，静的スレッドスケジューリングでは適切なスケジューリングが不可能なアプリケーションを動的スレッドスケジューリングにより解決できる可能性があることを明らかにした．最後に，より多くのアプリケーションに対応することを目指して，本論文では性能予測が不十分であったアプリケーションについて考察を行い，今後の課題を明らかにした．

5.2 今後の課題

本論文では，今後のスレッドスケジューリングの研究にあたって，2 つの重要な課題が明らかになった．

一つは，メモリアクセスによる性能低下を的確に予測できる指標の必要性と，その予測に基づいて SMT 優先度の適用範囲をさらに拡大することである．実験の結果，SMT 優先度で予測可能なアプリケーションに関しては，提案スケジューリング手法によって性能向上を達成できることが示されていることから，SMT 優先度がさらに多くのアプリケーションを予測可能となれば，提案スケジューリング手法はさらに広範囲に適用可能となり，その有用性が高まることは明らかである．

もう一つは，動的スレッドスケジューリング手法の実装と，その評価を行うことである．現段階では，動的にサンプリングを行い，各インターバルにおける最適な組み合わせを決定することができている．後はその情報を基にスレッドの移動を行うことが必要となる．その方法には，OS のコンテキストスイッチを利用することも可能であるし，ハードウェアでスレッドの移動をサポートすることも可能である．

参考文献

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Moregan kaufmann Publishers, fourth edition, 2007.

[2] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, Vol. 19, No. 4, pp. 23–29, 1999.

[3] David J. Frank. Power-constrained CMOS scaling limits. *IBM Journal of Research and Development*, Vol. 46, No. 2–3, pp. 235–344, 2002.

[4] David W. Wall. Limits of Instruction-Level Parallelism. In *ASPLOS-IV: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, 1991.

[5] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, Vol. 30, No. 9, pp. 79–85, 1997.

[6] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[7] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *ISCA '96: Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[8] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, Vol. 17, No. 5, pp. 12–19, 1997.

[9] Douglas C. Bossen, Joel M. Tendler, and Kevin Reick. Power4 System Design for

High Reliability. *IEEE Micro*, Vol. 22, No. 2, pp. 16–24, 2002.

[10] Ofli Wechsler. Inside Intel Core Microarchitecture: Setting New Standards for Energy-efficient Performance, 2006. Technology@Intel Magazine.

[11] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A.Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Jornal*, Vol. 6, No. 1, pp. 4–15, 2002.

[12] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, and Alex Ramirez. Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, Vol. 55, No. 7, pp. 785–799, 2006.

[13] Balaram Sinharoy, Ronald N. Kalla, Joel M. Tendler, Richard J. Eickemeyer, and Jody B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, Vol. 49, No. 4–5, pp. 505–522, 2005.

[14] Stefan Rusu, Simon Tam, Harry Muljono, David Ayers, Jonathan Chang, Brian Cherkauer, Jason Stinson, John Benoit, Raj Varada, Justin Leung, Rahul Dilip Limaye, and Sujal Vora. A 65-nm Dual-Core Multithreaded Xeon Processor With 16-MB L3 Cache. *IEEE Journal of Solid-State Circuits*, Vol. 42, No. 1, pp. 17–25, 2007.

[15] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Matthew Iyer, Dan Fay, David Hodgdon, Joshua L. Kihm, Alex Settle, Dirk Grunwald, and Daniel A. Connors. Dynamic Run-time Architecture Techniques for Enabling Continuous Optimization. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pp. 211–220, 2005.

[16] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, third edition, 2003.

[17] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 158–167, 2006.

[18] T. Sherwood and B. Calder. Time Varying Behavior of Programs. *Technical Report UCSD-CS99-630, University of California, San Diego*, 1999.

[19] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pp. 45–57, 2002.

[20] Allan Snavely and Dean M. Tullsen. Explorations in Symbiosis on two Multithreaded Architectures. In *M-TEAC '99: Proceedings of the Workshop on Multithreaded Execution, Architecture, and Compilation*, 1999.

[21] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS-IX: Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*, pp. 234–244, 2000.

[22] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. *ACM SIGMETRICS Performance Evaluation Review*, Vol. 30, No. 1, pp. 66–76, 2002.

[23] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-Sensitive Scheduling for SMT Processors. *Technical report, Department of Computer Science and Engineering, University of Washington*, 2000.

[24] Ali El-Moursy, Rajeev Garg, David H. Albonesi, and Sandhya Dwarkadas. Compatible Phase Co-Scheduling on a CMP of Multi-Threaded Processors. *IPDPS 2006: 20th International Parallel and Distributed Processing Symposium*, 2006.

[25] Dean M. Tullsen and Jeffery A. Brown. Handling Long-latency Loads in a Simultaneous Multithreading Processor. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 318–327, 2001.

[26] Francisco J. Cazorla, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Improving Memory Latency Aware Fetch Policies for SMT Processors. In *ISHPC '03: Proceedings of the 5th International Symposium on High Performance Computing*, pp. 70–85, 2003.

[27] Michael D. Powell, Mohamed Gomaa, and T. N. Vijaykumar. Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In *ASPLOS 2004: Proceedings of the 11th International Conference on Architectural*

Support for Programming Languages and Operating Systems, pp. 260–270, 2004.

[28] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 81–92, 2003.

[29] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, Vol. 26, No. 4, pp. 52–60, 2006.

[30] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, Vol. 33, No. 7, pp. 28–35, 2000.

発表論文リスト

[1] 船矢祐介, 小寺功, 滝沢寛之, 小林広明. スレッド特徴量に基づくマルチコアプロセッサスケジューリング. FIT2006: 第 5 回情報科学技術フォーラム 情報科学技術レターズ, pp. 37–40, 2006.

[2] 佐藤雅之, 船矢祐介, 小寺功, 滝沢寛之, 小林広明. SMT プロセッサの実行時性能予測のためのハードウェアリソース競合解析. FIT2007: 第 6 回情報科学技術フォーラム 情報科学技術レターズ, pp. 67–70, 2007.

謝辞

本研究を進めるにあたり，懇切な御指導・御鞭撻を賜わりました，小林広明教授に厚く御礼申し上げます.

本論文の審査にあたり，有意義な御指導・御助言を賜わりました，情報科学研究科　堀口進教授，青木孝文教授に厚く御礼申し上げます.

本研究を進めるにあたり，有益な御指導・御協力を頂きました，後藤英昭准教授に深く感謝致します.

本研究を進めるにあたり，懇切な御指導・御指摘を頂きました，滝沢寛之講師に心より感謝致します.

本研究を取りまとめる上で，多大なる御意見・御助言を頂きました，江川隆輔助教に深く感謝致します.

加えて，日頃から公私にわたり私を支えてくださった，本研究室の皆様に深く感謝致します.

最後に，私を支えてくれた家族，友人に心より感謝致します.

平成 20 年 1 月 31 日　船矢祐介