社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－OS－96 (20)
2004／6／18

SMTプロセッサにおけるスレッドスケジューラの開発

内倉 要 † 笹田 耕一 † 佐藤 未来子 †
加藤 義人 † 大和 仁典 †
中條 拓伯 † 並木 美太郎 †

近年，SMT(Simultaneous MultiThreading) などのマルチスレッドプロセッサアーキテクチャが新たなプロセッサ性能向上の可能性として注目されている．これは，プロセッサに送られる命令流を一つのスレッドとして並列に実行する方式である．演算器やキャッシュを共有し実行することで処理の性能を高めるが，逆にこれらの共有はスレッドごとの競合をももたらすため，欠点となることも予想される．予備実験によりSMTプロセッサが性能低下を起こす原因を調査した結果，キャッシュミスであることが明らかになった．そのため，スレッドの実行を制御したスレッド切り替えによって，キャッシュミスによる性能低下を防ぐためのスレッドスケジューラを開発した．また，実行結果を分析し，スレッドスケジューラがキャッシュミスの問題を改善したことを示す．評価の結果，SPLASH-2ベンチマークのLU分解プログラムが，スレッドスケジューラによって最大1.7倍の速度向上率を達成することができた．

Development of a thread scheduler for SMT processor architecture

KANAME UCHIKURA ,† MIKIKO SATO ,† NORITO KATO ,†
MASANORI YAMATO ,† HIRONORI NAKAJO † and MITARO NAMIKI†

Recently, multithreaded processor architecture is remarked as new possibility of processor performance progressing, for example SMT(Simultaneous Multi Threading) and so on. This is a parallel processing system that regards a sent instruction stream as one of thread. By sharing and using execution units and caches, SMT heighten processor performance. By contraries, we suppose that these sharing also are defect with a conflict each thread. We examined cause decreasing SMT processor performance by pilot test. This problem is cache miss. Therefore, we developed a thread scheduler that switches some threads with control thread processing to keep what decreased performance by cache miss. And analyzing result, we show that a thread scheduler improves cache miss. As a result of evaluation, LU solution of SPLASH-2 benchmarks up to 1.7 times higher performance has been gained by thread scheduler.

1. はじめに

一つのチップ上で複数のスレッドを同時に処理することでプロセッサの性能向上を図るスレッドレベル並列性 (TLP) を主体としたマルチスレッドアーキテクチャが提案されている．これまで，命令レベル並列性 (ILP) に着目することにより，高性能化を図るスーパスカラアーキテクチャが主流であったが，ILP抽出の困難さから，性能向上の限界に達したと考えられている．これに対し，マルチスレッドアーキテクチャでは，複数の実行命令流を1つのスレッドとして複数のスレッドを1チップ上で同時に実行することにより，さらなるプロセッサアーキテクチャ性能の向上が期待できる．マルチスレッドアーキテクチャの一つとして，SMT(Simultaneouse MultiThreading 以下 SMT) プロセッサがある．[6] これは，1チップで複数のプログラムカウンタ，レジスタコンテキストを持ち，命令実行ユニット，キャッシュメモリなどのハードウェアリソースを共有することで，複数の命令を高速に処理することができる．スーパスカラアーキテクチャでは利用しきれなかった複数の命令実行ユニットやキャッシュメモリを有効に活用し，プロセッサ性能の向上が期待される．

マルチスレッドアーキテクチャにおいて，送られてくる実行命令流の単位を**実スレッド**(物理スレッド，AT:Architecture Thread，AThread) と定義する．また、マルチスレッドプログラミングやコンパイラによって生成されるスレッドを**論理スレッド**(LT:Logical

† 東京農工大学大学院工学教育部
Graduate School Technology, Tokyo University of Agriculture and Technology

Thread) として定義し，本研究ではアーキテクチャにおけるスレッドとソフトウェア上でのスレッドを明確に分けることにする．つまり，マルチスレッドアーキテクチャでは複数の実スレッドを持ち，マルチスレッドプログラミングやコンパイラによって生成された論理スレッドを並列に処理する．

SMT プロセッサでは，各実スレッドが独立して処理を行なうため，共有している演算器で，競合を引き起こすことが予想される．同様に，キャッシュメモリを共有していることから，各々の実スレッドで必要とされるデータの競合が起こらないような方式が必要となる．演算気の競合やキャッシュミスの発生は，各々の実スレッドの処理の停滞につながり，結果としてプロセッサ全体の性能を低下させる．SMT プロセッサが持つ演算器の数やキャッシュメモリのサイズにより，競合を引き起こさないための，実スレッドに割り当てる論理スレッドの数や，命令自体が他の論理スレッドと，データが比較的共有されているかどうかと言った関係性が重要となる．本研究では，SMT プロセッサにおける性能低下について調査し，その結果，得られた原因を解決し，最大限にハードウェアリソースを有効に利用するためのスレッドスケジューラを開発した．具体的に性能低下の原因とは，キャッシュミスによるもので，このスレッドスケジューラはプロセッサ内で起こるキャッシュのミス率を動的に監視し，性能が低下したと判断した場合に，実スレッドの並列度を下げる．また，キャッシュへの参照の局所性を高めるために，適宜待ち行列にある論理スレッドをソートし，順序を決めて処理を行なう方式をとった．また，これらの方式がどの程度有効であるかを分析した．

次章では、本研究で使用する SMT プロセッサの概要と、SMT プロセッサが持つ課題について述べる．3 章では課題に対して，本論文の目標について述べ，スレッドスケジューラを提案する．4 章では，目標に従った設計について述べる．5 章では，スレッドスケジューラの性能の評価について述べ，6 章では，5 章での結果について考察したことを述べる．

2. SMT プロセッサの持つ課題

本章では，本研究で使用する SMT プロセッサのシステム全体の構成について述べる．また，SMT プロセッサを使用した実験をもとに，SMT プロセッサの持つ課題について考察する．

2.1 OChiMuS PE

OChiMuS PE とは，筆者らが研究，開発を進めている SMT プロセッサである．この SMT プロセッサの詳細については[8] を参照されたい．OChiMuS PE は，MIPS アーキテクチャをベースとした SMT アーキテクチャプロセッサであり，マルチスレッドをサポートするためのモジュールや命令を新たに拡張したものである．将来的には，プロセッサを一つのエレメント (PE:Processing Element) とし，この PE を同一チップ上に複数搭載したオンチップマルチ SMT(OChiMuS) アーキテクチャとすることを検討している．このため，ここで示すアーキテクチャによるプロセッサを OChiMuS PE と称し，本研究で使用する．

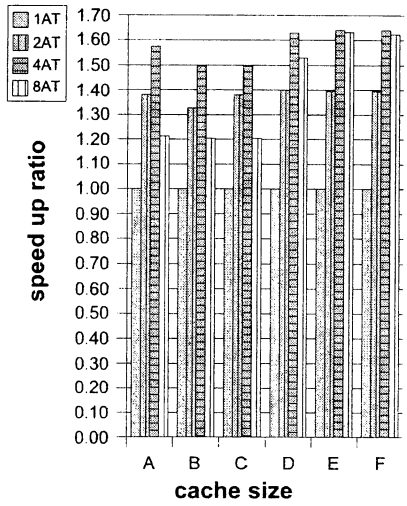

図 1 キャッシュサイズの違いによる実行速度比較
Fig. 1 The relation of processing speed by cache size

表 1 キャッシュのサイズと組み合わせ (KB)
Table 1 The size and combination of cache memory(KB)

	A	B	C	D	E	F
D1Cache	4	4	4	8	16	32
U2Cache	128	256	512	512	512	512

OChiMuS PE では，複数の実スレッドを持ち，各実スレッドに対して動的に論理スレッド番号(LTN) を設定し動作する．ソフトウェアで生成された論理スレッドは，論理スレッド番号によって実スレッドを操作する．この仕様により，ユーザレベルにおいて実スレッドの管理，操作が可能となる．

2.2 並列実行の予備実験

SMT プロセッサが並列実行を行ない，どのような挙動を示すのかを実験により調査した．OChiMuS PE を対象とした OS Future，および、マルチスレッドライブラリ MuLiTh が開発されており，それらのシステムソフトウェア上で SPLASH-2[11] ベンチマークより LU 分解を実行した．OS Future，および、マルチスレッドアーキテクチャ向けスレッドライブラリ MuLiTh については後述する．

実験では，プロセッサに搭載されている，キャッシュメモリを様々なサイズで実行した．表 1 に実験で設定

表 2 1 次データキャッシュヒット率
Table 2 The level 1 data cache hit-ratio

	A	B	C	D	E	F
1	99.71%	99.71%	99.71%	99.88%	99.93%	99.97%
2	99.60%	99.60%	99.60%	99.85%	99.93%	99.97%
4	96.54%	97.67%	97.63%	99.58%	99.90%	99.97%
8	92.73%	92.75%	93.11%	98.10%	99.77%	99.96%

した，メモリコンフィグレーションの組み合わせの種類とそのサイズを示した．D1Cache とは，1 次データキャッシュで各スレッドが利用する．U2Cache とは，統一 2 次キャッシュのことで，1 次データキャッシュと 1 次命令キャッシュ共に扱う．1 次命令キャッシュは全ての場合で 4KB と固定した．

実行結果を図 1 に示す．それぞれのメモリコンフィグレーションにより，4 本の棒グラフが並んでいるが，それぞれ実スレッド数を示している．シングルスレッド実行を基準として、多スレッドでの実行がどの程度高速化されたかを示している．全体では，キャッシュサイズの増加と，実スレッド数の増加に伴って，最大 1.8 倍の速度上昇が示された．しかし，キャッシュサイズが小さいと 8 実スレッドの性能は大幅に低下した．結果として，1.2 倍以下の速度向上率となり，2 実スレッドよりも速度向上率は低下した．1 次データキャッシュが実行性能に関係があることから，1 次キャッシュヒット率を求めた．結果を表 2 に示す．キャッシュサイズが小さくなると，4 実スレッド，8 実スレッドではキャッシュヒット率が低下し，95.00%を下回っている．これは，同時に実行するスレッドの数が多いために，キャッシュの競合が多発したからである．キャッシュが大きい場合のキャッシュヒット率はほぼ 100%でキャッシュがヒットすれば，実スレッド数に見合った性能の向上が見込まれる．

全体として，1 実スレッドと 2 実スレッドの速度向上率の関係は常に一定で，キャッシュヒット率も常に高く，これらは処理性能の限界に達している．対して，4 実スレッド，8 実スレッドでは，キャッシュのサイズが結果に左右されることがわかった．したがって，キャッシュサイズによる性能低下を解決することで，SMT プロセッサの持つ欠点を克服し，さらなる性能向上が期待される．

2.3 課　　題

このように，SMT プロセッサにおいて実スレッドの数が大きくなり並列度が高くなると，キャッシュが小さい場合に，性能低下を引き起こす．

一つの原因は，複数のスレッドがキャッシュメモリを取り合う場合に起こるキャッシュミスである．もう一つの原因として，他のスレッドとのメモリ空間の共有が少ないことである．前者の原因に対しては，一時的にスレッドの並列度を下げて実行することでこの問題を解決する．後者は，メモリ空間を多く共有する，もしくは近いスレッド同士で実行ができるようにする．そのためにはスレッド操作が必要であり，スレッドスケジューラを提案する．

また，このような課題に対し，これまでスレッドスケジューラとして Snavely，Tullsen らによる常に最適なスレッドの組み合わせを求めるスレッドスケジューラ[3),4)] や，Sujay Parekh らの性能が低下しているスレッドを入れ替えるスケジューラ[5)] などが挙げられており，演算器の使用率や，キャッシュミス率をスレッドスケジューラの指標として，スレッド切り替えによりプロセッサの性能を維持する方式が提案されている．

3. 目　　標

本論文の目標はキャッシュミスを防ぎ，プロセッサの性能向上を図るスレッドスケジューラを開発し，スレッドスケジューラによりスレッドの実行を制御することで，キャッシュミスの上昇を抑制するための方式を提案する．キャッシュミスの上昇となる原因として，多くのスレッドが同時に実行することによるキャッシュミスの増加が挙げられる．この原因に対しては，一部の実スレッドの実行を停止する．他の原因としてはキャッシュを共有していないスレッド同士の並列実行によるキャッシュミスがの増加が挙げられる．これは，キャッシュを利用している場所が近いスレッド同士を選択して並列実行できるようにする．

スレッドスケジューラはユーザレベルにおき，様々なプログラムに対処できるように，動的にプロセッサの性能を判断し実行する．

4. 設　　計

前章での目標を元に，どのようにスレッドスケジューラを設計，実装したかについて述べる．

4.1 システムの全体構成

ここでは，OChiMuS プロセッサアーキテクチャを制御しているシステムソフトウェア全体の構成について述べる．システム全体の構成を図 2 に示す．

ハードウェアを制御するために，カーネルレベル，ユーザレベルそれぞれの階層でシステムソフトウェアが開発され，アプリケーションプログラムはこのシステム上で動作する．ユーザレベル上でスレッドライブラリがスレッドの管理を行なっているため，スレッドスケジューラはユーザレベルに置き，スレッド実行の制御を行なう．

4.1.1 スレッドライブラリ MuLiTh

マルチスレッドアーキテクチャ上でスレッドを操作するためのスレッドライブラリについて述べる．MULiTh(Thread Library for Multithreaded Architecture 以下 MULiTh) は筆者らが研究を進めている OChiMuS PE 用に作られたスレッドライブラリである．アプリケーションソフトウェアである POSIX Thread(Pthread) 仕様に準拠したスレッド関数呼び出

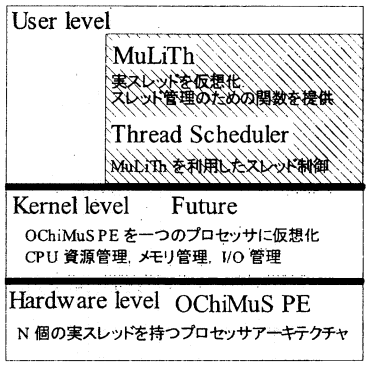

図 2 システムの全体構成
Fig. 2 The over view of a system

しによって機能する．プロセッサ内部の LTN を利用することで，実スレッドを仮想化し，スレッドの生成，消滅，および，同期などのプロセッサの持つスレッド制御命令を容易に利用することを可能にしている．詳細は[10] を参照されたい．

4.1.2 OS Future

プロセッサを管理するために必要な OS アーキテクチャについて述べる．マルチスレッドアーキテクチャ向け OS Future は筆者らが研究を進めている OChiMuS PE 用に開発されたもので，PE を一つのプロセスとして仮想化し，マルチスレッドアーキテクチャに適したプロセスモデルをサポートしている．前節で述べた．MULiTh と協調動作することで，互いの性能を発揮している．

Future では UNIX の従来の API を可能な限り維持しつつ，カーネルの内部構造を OChiMuS PE プロセッサ向けに見直す方針とする．また，できる限りカーネルのオーバヘッドの軽量化を図ることを目標としている．つまり，CPU 資源管理，メモリ管理，I/O 管理において，個々の制御機能をできる限りシンプルに設計し，カーネルで行うべき処理，ランタイムライブラリ，スレッドライブラリで行うべき処理を切り分け，そしてカーネル内処理のオーバヘッドを最小限に止められる小型・軽量な OS を目指している．詳細は[8],[9] を参照されたい．

4.2 スレッドスケジューラの概要

スレッドスケジューラはユーザレベルに位置し，スレッド管理を行なっているマルチスレッドライブラリ MuLiTh，および，OS Future と連携して，論理スレッドを入れ替えることができる．MuLiTh では，スレッドスケジューラが利用するための LTN やスレッド待ち行列の設定，スレッドの切り替えなど，スレッドに関する全ての制御を利用することができる．Future では，プロセス切り替え時に全スレッドの物理メモリへの退避、復帰を行なっている．スレッドの復帰時にスレッドスケジューラを呼び出すため，プロセス切り替えが起こる周期でスレッドスケジューラが呼び出され実行する．

```
(1)OS からスケジューラが呼び出される.
(2) スレッドを待ち行列に格納.
(3)if(待ちスレッドが実スレッドより多い)
{
  プロセッサからキャッシュヒットと
  ミスの回数を得る.

(4)if(キャッシュミス率が上昇して性能低下)
  {
    スレッド制御の性能低下フラグを立てる.
  }
}
(5)else
{
  待ち行列のスレッドは，実スレッド以下に
  なったため，性能低下フラグをリセットし，
  スレッド制御をやめる.
}
(6)if(性能低下フラグが立っている)
{
  本研究では,
  (A) 実スレッドの一部実行停止.
  (B) キャッシュ共有を高めるための LTN 整列.
  を行なう.
}
(7) 実行開始
```

図 3 スケジューラ実行の概観
Fig. 3 The scheduler processing view

スレッドスケジューラ全体のアルゴリズムを図 3 に示す．

スレッドスケジューラは，(1)OS のプロセス切り替えに同調したある周期的なサイクルによって呼び出され，一つの実スレッドがスケジューリングを行なう．(2) 他のスレッドは一時的にその間，待ち行列に格納され sleep 状態となる．(3) 待ち行列にある実行待ちスレッドが実スレッドより多いかどうかを判断し，多ければプロセッサのキャッシュ利用状況を得る．(4) スレッドスケジューラはプロセッサが性能低下であるかどうかを判断し，性能低下を検知すると性能低下フラグを立てる．(5) 待ち行列にある実行待ちスレッドが実スレッドより少ない場合は，一旦，性能低下フラグをリセットする．(6) 性能低下フラグが立っていると (A)，(B) の方針に従った実スレッドの割り当てを行い，最後は自らも次の処理に移る．

性能低下フラグは，プロセッサの実行性能の状態によって決まる．スレッドスケジューラは現在のプロセッサの性能をキャッシュの状態から判断し，キャッシュミスが多ければ性能低下と判定する．キャッシュの情報は，cp2，コプロセッサ 2 を利用して直接プロセッサからキャッシュの利用状況が得られる．得られたキャッ

シュ利用状況のデータから，キャッシュのミス率を求め，性能低下を判定する．具体的な性能低下の判定方法と，性能低下時に行なうスレッド制御については次項以降で述べる．

4.3 性能低下の判断

スレッドスケジューラは動的にプロセッサの性能が低下しているかどうかを判断する．

プロセッサでは，キャッシュに対して読み込みと書き込みを何度も行なっている．読み込みと書き込みそれぞれの回数と，ミスの回数をカウントし，スレッドスケジューラはそれを取り出すことができる．スレッドスケジューラは全体のアクセス回数とミスの回数から，キャッシュミス率を得る．

実行開始時からの累積による総キャッシュミス率と，周期的なサイクル間に得られた部分キャッシュミス率を比較することで，性能低下を判断する．具体的には以下の式で表される．この比較には総キャッシュミス率に補正値を与えて，妥当な比較式にする必要があるが，今回は特に補正は行なわないで単純な比較判定の式とした．総キャッシュミス率は実行開始時からのキャッシュミス率であり，時間の経過によって平均的な値に近づく．部分キャッシュミス率は，100 万から 200 万サイクルでの一定区間でのキャッシュミス率である．100 万サイクル以下であると，瞬間的なミス率の上昇を検知するので誤った制御の元となる．また，200 万サイクル以上では冗長になりすぎて区間値としての意味がなくなる．100 万サイクルから 200 万サイクルが妥当であるとした．

$$PART\_MISS\_RATE \geq TOTAL\_MISS\_RATE \quad (1)$$

上記の比較式から実行開始時からの平均的なキャッシュミス率を上回るキャッシュミス率であった場合，そのプロセッサは性能低下を引き起こしていると判断する．

また，この性能低下の判断は，図 3 の (3)，(5) で示しているように，実スレッド数以上に待ち行列に論理スレッドがある場合に実行される．これは，実スレッド数より論理スレッドの数が小さい場合は，プロセッサに実スレッドのあまりが出るため、基本的には性能は低下しないと考えているからである．さらに，性能低下の判断が一度スレッドスケジューラで行なわれると，待ち行列の論理スレッドが，実スレッド数より小さくなるまで判断は覆らない．スレッドが同期や消滅により，待ち行列からスレッドが無くなるまでは，処理内容にほとんど変化がなく，キャッシュ利用の変化はほとんどないためである．スレッド生成や同期後に待ち行列のスレッドが実スレッドを上回ると，再びプロセッサでの性能低下を監視する．

4.4 スレッドの選択

プロセッサの性能低下と判定した場合，スレッドスケジューラは次の並列実行でのスレッドをなんらかの方式で選択し実行する．性能低下が起こらずスレッドスケジューラが何もしない場合は、これまで実行していた論理スレッドを待ち行列に置き，待ち行列にたまった論理スレッドを順番に実スレッドに割り当て実行する．このように，通常時では FIFO(First In First Out) として論理スレッドを入れ替え，全スレッドを平均的に実行している．

今回提案する方法として，(A) 実スレッドの一部を停止する方式，(B) 割り当てる論理スレッドの持つ論理スレッド番号 (LTN) を整列し割り当てる方式、の二つを行なうとした．これらの二つの方式について述べる．

4.4.1 実スレッドの一部を停止する方式

並列実行スレッドの並列度が高いと，それだけ必要とするキャッシュメモリの量が大きくなる．しかし，キャッシュメモリのサイズが小さいと，キャッシュミスが多発し演算処理に移れない問題が生じる．その結果，プロセッサの性能は低下する．そのため，性能低下時に一部の実スレッドの実行を停止し，実際の実スレッド数より少ないスレッド数で実行を行なう．実際には実スレッドの半分を止める．実スレッドの一部を停止することで，残りの実スレッドが円滑にキャッシュを利用することができるため，結果として全体の処理は速くなる．実スレッドの半分を停止すると，単純に二倍のキャッシュが使えるようになる．この方式を **Reducing Scheduler**，**RS** とする．

実行する際の並列度を変化させる方式は，大河原らによる最適実行多重度に基づく SMT プロセッサのジョブスケジューリング[7] で提案されている．この方式では，サンプリング期間を設けて，もっとも適切なスレッド並列度を求めてその並列度で実行を行なう方式を繰り返すものであるが，サンプリング期間の長さにはトレードオフが発生し，最適なサンプリング期間と，並列度決定後の処理期間を設定するのは容易ではない．さらに，このトレードオフによる最適なサンプリング期間はプログラムの内容によって一意に決まることはなく，様々なプログラムに対して高い性能を発揮するための汎用性に欠けている．本論文で提案する方式では，動的に性能低下を判断し，並列度を下げて実行することで性能を改善しているため，様々なプログラムに対応することができる．

4.4.2 LTN を整列する方式

従来のスレッド切り替えでは，スレッド切り替え時に次に実行されるスレッドは予測不可能であるため，次の並列実行が相性の良いスレッド同士で実行されるかわからない．相性が悪いと，キャッシュ共有の割合が低くなり，キャッシュミスが増える．マルチスレッドプログラミングではスレッドの生成順に，処理を割り当てるため，生成順に近いスレッド同士を実行するとキャッシュにヒットすることが予想される．

表 3 実行結果 (キャッシュサイズ小)
Table 3 The prcessing result on small cache

	cycle	IPC	L1Dcache hit rate
LU 4AT	3.02×10^8	1.74	92.60%
LU 4AT RS	3.16×10^8	1.67	94.83%
LU 4AT RS+SS	2.65×10^8	1.98	98.13%
LU 8AT	5.50×10^8	0.96	80.21%
LU 8AT RS	3.61×10^8	1.46	89.92%
LU 8AT RS+SS	2.82×10^8	1.87	94.78%
FFT 4AT	4.98×10^7	1.16	93.39%
FFT 4AT RS	5.00×10^7	1.16	98.23%
FFT 4AT RS+SS	4.27×10^7	1.36	98.77%
FFT 8AT	7.84×10^7	0.74	82.56%
FFT 8AT RS	5.92×10^7	0.98	93.00%
FFT 8AT RS+SS	4.48×10^7	1.30	96.97%

表 4 実行結果 (キャッシュサイズ大)
Table 4 The prcessing result on large cache

	cycle	IPC	L1Dcache hit rate
LU 4AT	1.65×10^8	3.18	99.78%
LU 4AT RS	2.16×10^8	2.43	99.70%
LU 4AT RS+SS	2.15×10^8	2.45	99.83%
LU 8AT	1.57×10^8	3.35	99.62%
LU 8AT RS	1.72×10^8	3.06	99.49%
LU 8AT RS+SS	1.79×10^8	2.96	99.75%
FFT 4AT	3.44×10^7	1.67	99.79%
FFT 4AT RS	4.63×10^7	1.25	99.71%
FFT 4AT RS+SS	3.94×10^7	1.47	99.78%
FFT 8AT	3.58×10^7	1.61	99.57%
FFT 8AT RS	4.15×10^7	1.40	99.63%
FFT 8AT RS+SS	3.72×10^7	1.56	99.65%

スレッドが生成されるときに論理スレッドにはそのスレッドを識別するための LTN が与えられる．基本的には，この LTN の近いスレッド同士を実スレッドに割り当てることで，キャッシュ利用の近いスレッド同士で実行を行なう．このため，スレッド切り替え時に，待ち行列にあるスレッドを一度整列し，そのスレッドを順番に割り当てる．LTN の整列にはバブルソートを使用した．この方式を **Sorting Scheduler**，**SS** とする．

4.5 実　　装

スレッドスケジューラはマルチスレッドライブラリ MuLiTh 内に実装した．MuLiTh ではスレッドの生成やスレッド切り替えなど，スレッドの管理を行なっているため，スレッド操作を非常に容易に行なうことができる．実スレッドへの論理スレッド割り当て，待ち行列のスレッド整列などの操作は，全て MuLiTh の関数を利用した．

プログラムの作成は binutils-2.13☆，gcc-3.2☆☆，newlib1.9.0 を用いた．

☆ OChiMuS PE のスレッド制御命令を利用可能にしたもの．
☆☆ 最適化オプションは -O3 を設定した．

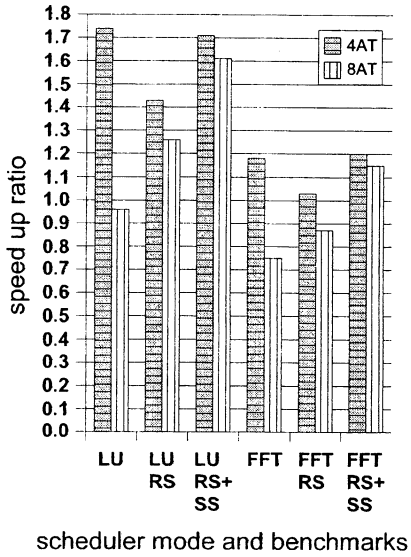

図 4 スケジューラによる速度向上率 (キャッシュサイズ小)
Fig. 4 The speed up ratio by scheduler mode on small cache

表 5 キャッシュヒット向上率 (キャッシュサイズ小)
Table 5 The progress ratio of cache hit on small cache

	LU RS	LU RS+SS	FFT RS	FFT RS+SS
4	1.02	1.06	1.05	1.06
8	1.12	1.18	1.13	1.17

5. 評　　価

ここではスレッドスケジューラの実行速度について述べる．

本評価は筆者らが作成している実行駆動型シミュレータ **MUTHASI**(MUltiTHreaded Architecture SImulator)[8] を用いて行った．MUTHASI は OChiMuS PE をシミュレートし，プロセッサのパラメータについて容易に設定可能となっている．また，実スレッド数を 1 に設定すると，通常の MIPS プロセッサの挙動を示す．MUTHASI の実行環境は，L1Dcache4KB，L2Ucache512KB をキャッシュサイズ小として，L1Dcache32KB，L2Ucache512KB をキャッシュサイズ大とし，二通りのキャッシュの組み合わせで処理を行なった．特に 1 次キャッシュサイズの大小によるスレッドスケジューラの影響を調査した．

評価には，SPLASH-2 ベンチマークより，行列演算を行なう LU 分解，FFT，高速フーリエ変換を行なった．実スレッド数は 4，8 である．全ての結果を表 3，表 4 に示す．

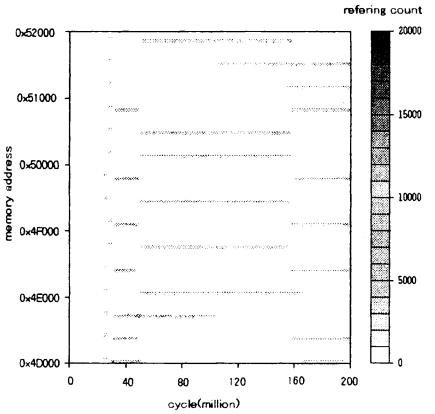

図 5 従来方式のメモリアクセスパターン
Fig. 5 The memory access pattern in conventional

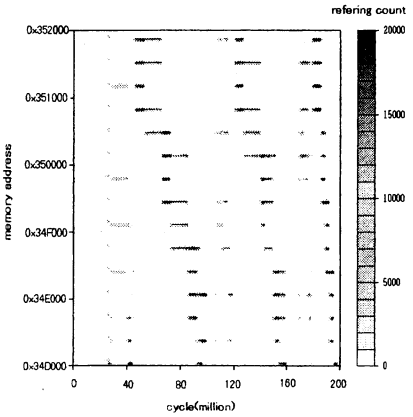

図 6 RS+SS のメモリアクセスパターン
Fig. 6 The memory access pattern in RS+SS

様々な試行の結果，RS で行なった場合と，RS と SS のスケジューラ方式を組み合わせた場合を示す．キャッシュサイズ小では，8 実スレッドにおいて，スレッドスケジューラがない状態から RS，RS+SS とキャッシュヒット率，IPC を増加させ，IPC は最大で 1.87 まで引き上げることができた．4AT においては，RS では IPC が落ち込むが、RS+SS では IPC を最大 1.98 まで引き上げることができた．しかし，キャッシュ大では RS，RS+SS 共に，IPC が低下した．この原因としては 1 次キャッシュサイズが大きい場合，スレッドスケジューラがなくとも処理性能は高く，RS や RS+SS が持つスレッドスケジューラが速度を低下させたのは OS 処理のオーバヘッドが影響したためである．キャッシュヒット率は，わずかな差ではあるが RS+SS が高く，スレッドスケジューラはキャッシュサイズが大きい場合でも，キャッシュの共有が高まるようなスレッド実行となっている．

設計で示したスケジューラを実装しない状態でシングルスレッドでの実行に対する実行速度向上率を求めた．結果を図 4 に示す．

LU では IPC が RS+SS で 4 実スレッドでは 1.71 倍，8 実スレッドでは 1.61 倍となった．また，FFT では RS+SS で 4 実スレッドでは 1.20 倍，8 実スレッドでは 1.15 倍となった．全体としては，キャッシュサイズが小さいプロセッサで，キャッシュミスが起こる問題を解決し，性能の改善を行なうことができた．

また，キャッシュヒットの向上率を表 5 に示す．表より，8 実スレッドでは 1.1 倍以上の向上を示しており，キャッシュヒット率が 10%以上向上した場合もあった．RS ではキャッシュを利用するスレッドを減らし，SS ではキャッシュを共有させるようにすることで，プロセッサ全体のキャッシュヒットが向上した．

6. 考　　察

今回提案したスケジューラ方式が性能低下を改善したことについて考察した．

キャッシュを共有することで起こるキャッシュミスを解決するための方法として，一つのスレッドあたりに利用できるキャッシュのサイズを増やす方法と，キャッシュの共有率が高いと考えられるスレッド同士，つまり，LTN の近いスレッドを同時に実行する方法があった．これらの方式を実装した RS，RS+SS では性能低下が改善され，高速な処理が実現できた．特に RS+SS が全体の処理にどのような影響をもたらしたかについて，サイクル数とメモリアドレスにおけるメモリアクセスパターンを調べた．従来方式の実行の結果を図 5 に,RS+SS による方式での実行を図 6 に示す．これらの図はスレッドがスタックに対してメモリアクセスを行なっている部分であり，横軸と平行に並んでいるメモリアクセスのラインそれぞれが一つのスレッドの挙動を示している．

従来方式では，並列実行を行なっているスレッドが競合を起こしているため，それぞれのメモリアクセスの回数が少なく一つの処理を終えるのに大幅な時間を要している．しかし，RS+SS 方式では，性能低下と判断した直後に，一部実スレッドを止め，並列実行スレッドの数を下げている．同時に，LTN の整列によりスレッド切り替え後は必ず最も近いアドレスを利用しているスレッドが選択されており，結果として，一度にメモリへアクセスする回数が非常に高くなっている．従来方式では 5000 回未満のメモリアクセスが長く続いているのに対して，RS+SS 方式では短い間隔で 15000 回以上のメモリアクセスを行なっている．

以上により，RS+SS の方式は SMT プロセッサが抱えるキャッシュの競合問題を解消し，各スレッドが

高いキャッシュヒット率によって高速な処理を実現していることが確認された.

7. まとめ

本論文ではSMTプロセッサにおけるスレッドスケジューラについて考察, 実現しそれを評価し分析した結果を示した.

評価の結果, 1次データキャッシュ4KByteの実行環境で, RS+SS方式でLU分解では最大1.7倍, FFTでは最大1.2倍の速度向上率を達成した. また, キャッシュの大小によらず, 高いキャッシュヒット率により並列実行され, 性能低下となる原因を改善することができた. 分析により, キャッシュのミスを抑え, 効率の良い実行の実現を確認した.

本論文では, OChiMuS PEを対象としたスレッドスケジューラの実装であったが, 実スレッドの管理がユーザレベルで可能であれば, ハードウェアアーキテクチャに依存しない. そのため, 提案したスレッドスケジューラはXeonなどの実用化されたプロセッサアーキテクチャに対しての適用が可能である.

本論文で提案した方式では, キャッシュの状態を基にする方式であったが, 他に演算器の利用状態やIPCついてもプロセッサの性能判断の指標となる. 今後は, それらの指標を加えて, より精度の高い性能の判断と, スレッド選択の新たの方式として利用する. また, 今回使用したベンチマーク以外のソフトなどを利用して, より汎用性の高いスレッドスケジューラとしての開発評価を行なっていく.

参考文献

1) Intel.Hyper-Threading Technology: http://www.intel.com/technology/hyperthread/
2) Intel. White Paper:Hyper-Threading Technology on the Intel Xeon Processor Family for Servers Offering increased server performance through on-processor thread-level parallelism, 2002.
3) Allan Snavely and Dean M. Tullsen: Symbiotic jobscheduling for a simultaneous multithreading processor, In *Architectural Support for Programming Languages and Operating Systems*, pp. 234–244, 2000.
4) Allan Snavely, Dean M.Tullsen, Geoff Voelker: Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor, *2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp66-76, 2002.
5) Sujay PAREKH, Susan Eggers, Henry LEVY, and, Jack Lo.: Thread-sensitive scheduling for SMT processors,*Technical report, Dept. of Computer Science, University of Washington (2000).*
6) Dean M. Tullsen, Susan Eggers, and Henry M. Levy.: Simultaneous multithreading: Maximizing on-chip parallelism, In *Proceedings of the22th Annual International Symposium on Computer Architecture*, pp. 392–403, 1995.
7) 大河原英喜、安里彰: 最適化実行多重度に基づくSMTプロセッサのジョブスケジューリング方式, 情処研報(2002-ARC-150),Vol.2002,No.112,pp.17-22(2002)
8) 河原章二, 佐藤未来子, 並木美太郎, 中條拓伯: システムソフトウェアとの協調を目指すオンチップマルチスレッドアーキテクチャの構想, コンピュータシステムシンポジウム, Vol. 2002, No. 18, pp. 1–8 (2002).
9) 佐藤未来子, 河原章二, 中條拓伯, 並木美太郎: SOC時代に向けたSMT用OSの構想, システムソフトウェアとオペレーティング・システム, No. 91-5, pp. 31–38 (2002).
10) 笹田耕一, 佐藤未来子, 河原章二, 加藤義人, 大和仁典, 中條拓伯, 並木美太郎: マルチスレッドアーキテクチャにおけるスレッドライブラリの実現と評価, 情報処理学会論文誌, Vol.44, No.SIG11(ACS3), pp.215-225 (2003.9).
11) Woo, S. C. et al: The SPLASH-2 Programs:Characterization and Methodological Consid-erations, *ISCA-22*,pp.24-36,1995.