Vol. 46 No. SIG 12(ACS 11) 情報処理学会論文誌：コンピューティングシステム Aug. 2005

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[†]

SMT（Simultaneous MultiThreading）アーキテクチャプロセッサは，複数のスレッドを並列実行することによりプロセッサ性能の向上を目指している．ところが，キャッシュメモリを共有しているため，キャッシュミスが増加し，性能が低下する．そこで，本論文では，スレッド相性の概念を提案し，スレッド相性によるスレッドスケジューラを開発した．個々のスレッド間の実行性能を監視し，それに応じてより良いスケジューリングを施すことで，性能向上を試みた．さらに，キャッシュヒット率の低下に対して，適した実行スレッド数を決める方式も同時に開発した．評価の結果，SPLASH-2 ベンチマークの RADIX ソートプログラムが，スレッドスケジューラによって最大 1.96 倍の速度向上を達成した．

Development of a Thread Scheduler for SMT Processor Architecture

Kaname Uchikura,[†] Koichi Sasada,[†] Mikiko Sato,[†]
Norito Kato,[†] Masanori Yamato,[†] Hironori Nakajo[†]
and Mitaro Namiki[†]

An SMT (Simultaneous MultiThreading) architecture processor aims to progress processor performance by executing parallel threads. However, the increasing cache misses caused by sharing the cache memory bring performance degradation. In this paper, we have developed a thread scheduler based on a concept of thread affinity. Our proposed system observes performance of concerning threads with common cache and reschedules them. In addition, we have developed a strategy to choose the suitable thread number according to decreasing cache hit ratio. As experimental results, the system with our developed thread scheduler performs up to 1.96 times higher with benchmark programs of RADIX sort in SPLASH-2.

1. はじめに

近年，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さかんに研究されている．

マルチスレッドアーキテクチャの 1 つとして，SMT（Simultaneous MultiThreading 以下 SMT）プロセッサがある[3)]．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．SMT プロセッサでは，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する．そのため，スーパスカラアーキテクチャ以上の性能向上が SMT プロセッサでは可能である．

ところが，SMT プロセッサには短所がある．SMT プロセッサではそれぞれのスレッドが独立に並列に実行され，それぞれのスレッドは 1 つのチップ上のキャッシュメモリや演算器を共有する．すると，スレッド間でのキャッシュメモリの競合や，演算器の枯渇が起こる．結果として，性能が低下してしまうことがある．

本論文では，スレッドスケジューラによる SMT プロセッサの性能向上を図る．性能低下の原因として，SMT プロセッサのキャッシュメモリの競合に着目する．そこで，キャッシュメモリの競合を改善するための 2 つの方式を提案する．

なお，本論文ではマルチスレッドアーキテクチャにおいて，1 つの命令流を処理する単位を**実スレッド（物**

† 東京農工大学大学院工学教育部
Graduate School of Technology, Tokyo University of Agriculture and Technology
†† 現在，ソニー株式会社
Presently with Sony Corporation
††† 現在，株式会社東芝
Presently with TOSHIBA CORPORATION

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 Thread*）として定義する．

次章では，本論文で使用する SMT プロセッサの予備実験を行う．3 章では，本論文の目標を述べる．4 章で，システム全体の構成を述べ，5 章でスレッドスケジューラの設計を行う．6 章では評価，7 章では関連研究との比較を行う．

2. SMT プロセッサアーキテクチャとその問題点

本章では，本論文で使用する SMT プロセッサの全体構成について述べる．本 SMT プロセッサの構成を使って，予備実験を行い，キャッシュメモリの競合を調査する．

2.1 OChiMuS PE

OChiMuS PE とは，筆者らが研究，開発を進めているオンチップマルチ SMT（On Chip Multi SMT: OChiMuS）プロセッサである．OChiMuS PE は，MIPS アーキテクチャをベースとした SMT アーキテクチャのプロセッサであり，マルチスレッドを提供するためのモジュールや命令が新たに拡張されている．本研究では，SMT プロセッサとして OChiMuS PE を使用する．この SMT プロセッサの詳細については文献 7) を参照されたい．

2.2 予 備 実 験

SMT プロセッサアーキテクチャでは，各スレッドがハードウェアリソースを共有するため，それらの利用競合が起こる．従来の研究でもハードウェアリソース競合による SMT プロセッサアーキテクチャの性能低下が指摘され，性能低下改善の方式が提案されている[5),6)]．本研究ではスレッドが共有するキャッシュメモリにおける性能低下の問題に注目する．

本実験では，SMT プロセッサアーキテクチャに OChiMuS PE を使用し，キャッシュサイズと実スレッド数の違いによる影響を実験によって調査する．OChiMuS PE が搭載するキャッシュのキャッシュサイズを変更し，実スレッド数 1，2，4，および，8 での実行結果を求めた．設定したキャッシュメモリのサイズの組合せを**表 1** に示す．

キャッシュメモリは，1 次キャッシュと 2 次キャッシュがあり，各実スレッドが共有する．また，1 次キャッシュは 1 次命令キャッシュと 1 次データキャッシュに分かれるが，1 次命令キャッシュについてはベンチマークに対して十分な容量を確保した．また，1 次データキャッシュのデータ格納構造はダイレクトマップ方式とし，2 次キャッシュは，4 ウェイセットアソシアティビティとした．

表 1 キャッシュのサイズと組合せ（KB）
Table 1 The size and combination of cache memory (KB).

	A	B	C	D	E	F
1DCache	4	4	4	8	16	32
2Cache	256	512	1,024	1,024	1,024	1,024

*1DCache：1 次データキャッシュ
**2Cache：2 次データキャッシュ

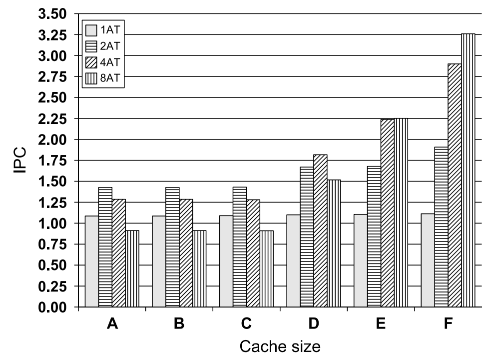


図 1 キャッシュサイズの違いによる IPC 比較
Fig. 1 The comparison of IPC by cache size.

実験には SPLASH-2 ベンチマークより LU 分解を実行した．論理スレッドを 16 個生成し，スレッドスケジューラは FIFO により交互にスレッドを割り当てる．詳細な計算方法は SPLASH-2 の論文を参照されたい[11)]．

実行結果を**図 1** に示す．表 1 に対応した実行結果である．それぞれ，実スレッド数 1，2，4，および，8 の場合である．

A，B，C では 1 次データキャッシュサイズを 4 KB とし，2 次データキャッシュのサイズの違いにより実行した．実スレッド数 4，8 では実スレッド数 2 の場合よりも IPC が低下した[☆]．ところが，1 次データキャッシュサイズが大きくなる D，E，F では，実スレッド数 4，8 の IPC が向上し，F では，実スレッド数 8 が最も高い IPC となった．

そこで，それぞれの実行における 1 次データキャッシュのキャッシュヒット率を求めた．1 次データキャッシュのヒット率を**図 2** に示す．

1 次データキャッシュサイズ 4 KB であった A，B，C では，実スレッド数 4，8 での 1 次データキャッシュヒット率が低下した．逆に，1 次データキャッシュサイ

☆ IPC:Instruction per cycle

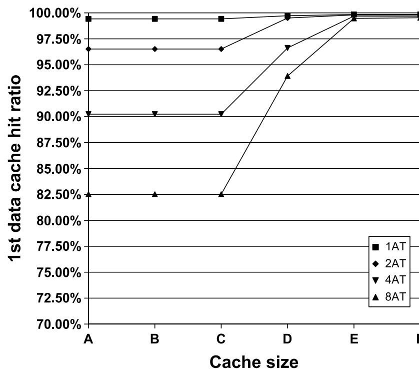


図 2 キャッシュサイズによる 1 次データキャッシュヒット率比較
Fig. 2 The comparison of L1 data cache hit ratio by cache size.

ズが大きくなるに従って，実スレッド数 4，8 のキャッシュヒット率が上昇し，F では 99%以上となった．

スレッドごとのワーキングセットが小さいために，2 次データキャッシュのサイズによるヒット率の影響は現れなかった．1 次データキャッシュについての影響だけを調査し，本研究での 2 次データキャッシュサイズは十分な量を確保することとする．

1 次データキャッシュヒット率の高さが IPC の結果に密接に関係しており，1 次データキャッシュのキャッシュヒット率を高めれば，SMT プロセッサアーキテクチャの性能は向上する．

2.3 SMT プロセッサアーキテクチャの問題点

予備実験の結果から，キャッシュミスについて分析する．Hennessy らによると，キャッシュミスの原因は 3 つある[2)]．

(1) 最初にキャッシュメモリにアクセスする際のキャッシュミス（Compulsory misses）
(2) キャッシュサイズの許容を超えるワーキングセットによるキャッシュミス（Capacity misses）
(3) キャッシュメモリのあるインデックスについて異なるメモリアドレスによるアクセスが集中して，キャッシュライン競合が起こるキャッシュミス（Conflict misses）

以上のキャッシュミスの 3 つの原因を SMT プロセッサアーキテクチャにおけるキャッシュミスの問題として考察する．

(1) が原因によるキャッシュミスは不可避である．

(2) の原因による性能低下は，キャッシュメモリが許容できる範囲を超えた実スレッド数が同時に実行されたことである．シングルスレッドであれば十分であるキャッシュサイズも，複数の実スレッドが並列実行するため，そのキャッシュサイズは不十分である．予備実験では，あるスレッドはそのスレッド自身が使用していたキャッシュライン対して，キャッシュライン競合を起こすキャッシュミスが多発した．実行中は，プログラムの進度とともに，各スレッドがキャッシュに保持しておきたいデータとデータの量は，たえず変化する．その変化に応じた適正な実スレッド数で実行することが必要である．このようなキャッシュミスを減少させれば，性能低下を改善できるであろう．

(3) の原因による性能低下は，各スレッドがあるキャッシュラインに対してアクセスを集中させるためである．SMT プロセッサアーキテクチャでは，各実スレッドはキャッシュを共有するため，(3) は多発する．そこで，無数にある論理スレッドから，スレッド間のデータ共有が多いスレッドを見つけ出し，それらのスレッドを同時に実行すれば，キャッシュの共有が増える．また，スレッド間で，アクセスしているキャッシュラインが違う場合，つまり，必要とするキャッシュラインが重ならないスレッドを同時に実行した場合についても (3) の問題は解消される．どちらの場合においても，キャッシュヒット率が向上し，性能が向上する．スレッドスケジューラは，キャッシュラインのデータ共有が多く，かつ，キャッシュスラッシングが少ないスレッドを見つけ出し，それらのスレッドを同時に実行することで，性能向上が見込める．

以上のように (2)，(3) のキャッシュミスの原因に注目し，それぞれに対し，解決できる方式を提案する．

3. 目 標

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を開発することが目標となる．

SMT プロセッサの性能が低下する原因として，1 次データキャッシュのヒット率の低下があげられる．キャッシュのヒット率の低下は，さらに 2 つの原因に分けることができ，1 つは，Capacity misses であり，他方は Conflict misses である．

前者の原因に対しては，キャッシュのヒット率全体を見て，スレッドスケジューラが毎回最適なスレッド数にすることで対処する．

後者の原因に対しては，実行中に個々のスレッドが他のスレッドとのキャッシュライン共有を見て，共有の高いスレッドどうしを選択する．ハードウェア上で実行中の全スレッドどうしのキャッシュライン共有を把握するのは物理的にレジスタ数が足りないため困難である．そこで，1 対 1 のスレッドのキャッシュライン共有と全体のキャッシュヒット率の比較により共有が多いスレッドどうしを選択する．

以上の 2 つの方式を実現し，OChiMuS PE 全体の性能向上を目指す．

スレッドスケジューラはユーザレベルで実行され，実スレッドへ論理スレッドの割当てを行う．そのため，OChiMuS PE と同様に，SMT アーキテクチャのプロセッサであれば，本方式を適用可能である．さらに，スレッドスケジューリング以外にもプロセススケジューリングが必要な CMP（Chip Multi Processor）などにも応用が可能である．

4. 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 3** に示す．

4.1 OS Future

OS *Future* とは SMT アーキテクチャ向けのために開発された OS で，PE を 1 つの CPU 資源として仮想化し，プロセスへの割当てを行う．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 SMT プロセッサアーキテクチャ向けに見直す方針とした．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．詳細は文献 7)〜9) を参照されたい．

4.2 スレッドライブラリ MULiTh

MULiTh（*Thread Library for Multithreaded Architecture*）とは筆者らが SMT アーキテクチャ向けに研究開発しているユーザレベルスレッドライブラリである．POSIX Thread（Pthread）仕様に準拠したスレッド関数呼び出しによって機能する．プロセッサ内部の**論理スレッド番号**（*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．詳細は文献 10) を参照されたい．

4.3 スレッドスケジューラの概要

スレッドスケジューラは，図 3 にあるようにユーザレベルに位置する．これは MULiTh の持つユーザレベルのスレッド制御機構をそのまま利用するためである．スレッドスケジューラは論理スレッドの実行順序や，並列実行の組合せを決定し，MULiTh を利用して，実スレッドへの論理スレッドの割当てを行う．また，ユーザレベルにあることで SMT アーキテクチャの各種命令を直接利用できる．

本スレッドスケジューラは，OS Future のプロセス切替え時のプロセス実行再開直後に稼動する．スレッドスケジューラの呼び出し周期は実行プログラムなど

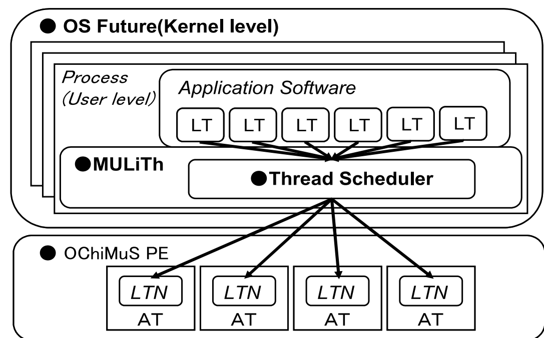


図 3　システムの全体構成
Fig. 3　The overview of the system.

の特定の条件に依存するため，スレッドスケジューラでは最適な呼び出し周期を設定する必要がある．予備実験では数通りの呼び出し周期を試行することで，スレッドスケジューラの呼び出し周期を設定した．100 万サイクル以下の短い周期では，1 桁から 2 桁程度のヒット回数やミス回数が得られるが，部分的なキャッシュヒット率となり，全体の実行結果とかけ離れた不安定なキャッシュヒット率となることから採用しなかった．逆に，200 万，1,000 万サイクルという長い周期では，スレッドスケジューラが働かず，実行内容が変化するスレッドには即応できなかった．ゆえに，予備実験で最も高い性能を示した 100 万サイクルを本論文での最適なスレッドスケジューラの周期とした．100 万サイクル周期では，他の周期に比べて 10〜20%の実行速度向上を示した．

スレッドスケジューリングの詳細な設計を次章で説明する．

5. スレッドスケジューラの設計

予備実験から，キャッシュミスの原因を 2 つに分けることができた．1 つは各実スレッドのワーキングセットのサイズがキャッシュサイズを超えている Capacity misses であった．他方は，キャッシュラインの競合による Conflict misses であった．

そこで，それぞれの原因を解決するため，2 つの方式をスレッドスケジューラに組み込んだ．前者の原因に対するスケジューラとして，**スレッド数決定スケジューラ，** *DT*（*The method to decide on the thread number*）を提案する．後者の原因に対しては，**スレッド相性スケジューラ，** *SA*（*Scheduling with Thread Affinity*）を提案する．

5.1 DT（スレッド数決定）スケジューラ

スレッドスケジューラが呼び出された際に，次の実行スレッド数を決定するスレッドスケジューラである．

```
(a) プロセッサから全体のヒット，ミス回数を得る
(b) 式 1 の計算
(c) X = 前回決定したスレッド数;
(d) /*表 2 による場合分け*/
 if(Hit_ratio >= 90){
    X = X + 1;
 }
 else if(Hit_ratio >= 80){
    X = X * 3 / 4;
 }
 else if(Hit_ratio < 80){
    X = X * 2 / 4;
 }
```

図 4　スレッド数決定スケジューラの概要
Fig. 4　The algorithm of DT scheduler.

この方式を**スレッド数決定スケジューラ，DT（The method to decide on the thread number）**とする．

DT では，**図 4** に示す条件に従って次に実行する実スレッド数が決定する．図中 (a) から (d) についてそれぞれ説明する．

(a) では，1 次データキャッシュへのヒット回数とミス回数を得る．プロセッサ内でヒット回数とミス回数をカウントし，スレッドスケジューラはその値を取り出す．なお，キャッシュのヒット回数とミス回数の計測は，OChiMuS PE に追加した．実用化されている SMT プロセッサの Xeon でもパフォーマンスモニタを持っていることから，スレッドスケジューラを適用可能である．

(b) では，以下に示す，式 (1) によって，キャッシュヒット率を導出する．

Hit_Ratio : キャッシュヒット率
Hit_num : キャッシュヒット回数
$Miss_num$: キャッシュミス回数

$$Hit_Ratio = \frac{Hit_num}{Hit_num + Miss_num} \times 100 \quad (1)$$

(c) では，1 回前のスケジューリングで決定した実行スレッド数を得る．さらに，(d) で次の実行スレッド数を決定する．スレッド数決定の指標を**表 2** に示す．予備実験によると，キャッシュのサイズにかかわらず，実スレッド数 1，2 では安定した実行性能であった．性能低下防止の閾値を求めるために 95〜80%間の数値を閾値とし予備実験を行った．閾値を 85〜80%とすると，実行全体の平均キャッシュヒット率は，90%とならず，実行サイクル数は増加し，実行速度は低下した．閾値を 95%とすると，95%以上のキャッシュヒット率となるように，実行スレッド数を減らし，キャッシュヒット率 95%以上の実行を維持しようと，シングルスレッド実行の時間が増加する．そのため，並列実行の利点が失われ，閾値 90%の場合に比べると，性能はやや低下した．性能低下防止のための閾値を 90%とすることが，本研究では最も良い条件であった．そこで，全体のキャッシュヒット率 90%以上を性能低下防止の閾値とした．

表 2　実スレッド停止の条件と停止される数
Table 2　The assumption to stop AT and the number of stopped ATs.

キャッシュヒット率	80%以下	80〜90%	90%以上
スレッド数 (X)	$X - X/2$	$X - X/4$	$X + 1$

この閾値とした 90%を下回ると，実スレッド数 4 では 1 個，実スレッド数 8 では 2 個の実スレッドを停止させる．さらに，80%を下回ると実スレッド数 4 では 2 個，実スレッド数 8 では 4 個の実スレッドを停止させるとした．90%以上になれば，毎回，1 個だけ停止していたスレッドの実行を再開する．実スレッドの実行を再開して，再び，キャッシュヒット率が 90%を下回れば，実スレッドの実行を停止させる．閾値となる 90%の値や，停止させる実スレッド数が影響するキャッシュヒット率の間隔は，いくつかのパラメータを与えて試行を行ったが，今回の数値設定が妥当な結果であった．実行スレッド数の上限は，実スレッド数であり，下限は 1 である．この設定では，キャッシュヒット率が 90%以下では必ず実スレッド数を停止させる．そのため，実行中は 90%以上のキャッシュヒット率を見込める．

しかしながら，この設定はアプリケーションによって変わる可能性もある．今回の数値設定は一例であって，アプリケーションに見合った数値設定が必要である．

5.2　スレッド相性スケジューラ

スレッド相性を各論理スレッドに用意し，そのスレッド相性の良さで次回に実行するスレッドを選択する．この方式を**スレッド相性スケジューラ，***SA*（*Scheduling with Thread Affinity*）とする．

5.2.1　スレッド相性の定義

スレッド相性（*Thread Affinity*）とは，2 つのスレッド間の互いの実行性能の指標である．あるスレッドとそのスレッドのターゲットとなるスレッドとのスレッド相性を計測する．このターゲットとなるスレッドを**ターゲットスレッド**とする．

スレッド A，B があり，スレッド A がターゲットスレッド B とのスレッド相性を見るとすると，スレッド相性を以下の式によって定義する．share(A,B) はスレッド B がデータを置いていたキャッシュラインにスレッド A がアクセスした際，そのキャッシュラインを

共有した場合にカウントされる．exclude(A,B) は，スレッド B が置いていたキャッシュラインにスレッド A がアクセスした際，そのキャッシュラインのデータを追い出した場合にカウントされる．share と exclude を使った式 (2) によりスレッド A がターゲットスレッド B に対するスレッド相性を求める．式 (2) では，共有が多いか，または，キャッシュスラッシングが少ないスレッドどうしのスレッド相性が高い数値となる．

A : スレッド A
B : スレッド A のターゲットスレッドであるスレッド B
$share(A,B)$: B のキャッシュラインに A がアクセスし，共有した回数
$exclude(A,B)$: B のキャッシュラインに A がアクセスし，データ入れ替えを行った回数

$$TA(A,B) = \frac{share(A,B)}{share(A,B) + exclude(A,B)} \times 100 \quad (2)$$

$*TA : Thread\ Affinity$

よって，スレッド A がターゲットスレッド B に対するスレッド相性，TA(A,B) が求まる．さらに，前節の式 (1) で得られるプロセッサ全体の実行中のキャッシュヒット率との比較により，スレッド相性の良い悪いを以下で定義する．

$TA(A,B) \geqq Hit_Ratio$: スレッド相性が良い (3)
$TA(A,B) < Hit_Ratio$: スレッド相性が悪い (4)

ある実行期間中で，プロセッサ全体のキャッシュアクセス対して，全スレッドが任意のキャッシュラインをアクセスし，共有した割合が Hit_Ratio であり，2 つのスレッド間のキャッシュライン共有率が TA である．

share，*exclude*，*target* はハードウェア上のスレッドコンテキストがそれぞれ持つ．実スレッド数 4 でのモデルを**図 5** に示す．スレッドスケジューラが実スレッドへ論理スレッドを割り当て，実行を再開する際に，ターゲットスレッドを各論理スレッドに設定する．スレッドとターゲットスレッドとの関係は循環的になっており，たとえば，あるスレッド A のターゲットスレッドは，スレッド A をターゲットスレッドとすることはなく，別なスレッドをターゲットスレッドとする．また，各スレッドがスレッド相性を測るためのターゲットスレッドの数は 1 つしか設定できない．実験では，スレッド間のキャッシュライン競合に，さらに，別なスレッドが入り込むことはきわめて稀であることから，TA(A,B) と TA(B,A) はほぼ等しいとした．その場合，実スレッド数 4 ならば 6 通り，実スレッド数 8 ならば 28 通りのスレッド相性組合せがあり☆，すべてのスレッド相性を知ることが望ましい．しかし，スレッド相性を計測するために SMT アーキテクチャの各実スレッドが持てるレジスタは限られているため，最小限の組合せである 1 つのターゲットスレッドとのスレッド相性を計測する方式とした．すべてのスレッド相性を見るための機能をプロセッサに実装することは非現実的である．仮にすべての実行中スレッドとのスレッド相性が得られたとすると，その中で最もスレッド相性が良いスレッドを選択することができるため，スレッドスケジューリングの方式としては最大限の効果を発揮するはずである．逆に，スレッド相性の最も悪いスレッドを取り出して実行するとした場合，性能は低下し，スレッドスケジューリングの方式としては最悪となる．本方式 SA は，ターゲットスレッドを 1 つと限定したため，スレッドすべてのスレッド相性を求めることはできず，スレッド相性がより良いスレッドを見つけ出すことができるかどうかは，ターゲットスレッドの選択に依存する．ターゲットスレッドは，無作為に並んだスレッドから決定するため，得られるスレッド相性が良いか悪いかは状況によって変化し，全体的に見るとターゲットスレッドとのスレッド相性が良いか悪いかは平均的に得られると考えられる．そこで，ターゲットスレッドとのスレッド相性が良い場合は，並列実行とし，スレッド相性が悪い場合は，どちらか一方を実行，もう一方を待ち行列に格納し，同時に実行させせない方式により，並列実行のためのスレッドの選択を行う．

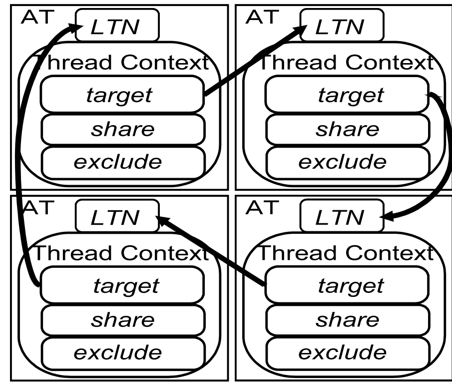


図 5　スレッドとターゲットスレッドとの関係
Fig. 5　The relation between a thread and a target thread.

5.2.2　スレッド相性によるスレッドの組合せ

従来の MULiTh のスレッドスケジューラは FIFO

☆ ${}_4C_2 = 6$，${}_8C_2 = 28$ の計算による．

```
実行していたスレッド t1,t2,t3,t4 があるとする.
do{
  A = スレッド Ta;      /*t1,t2,t3,t4 の順に*/
  B = A のターゲットスレッド Tb;  /*t1 なら t4*/
  if(TA(A,B) >= Hit_num){
     if(B が待ち行列へ格納されていた){
         A を待ち行列へ格納;
     }
     else{  /*B が実スレッドに割り当てられていた*/
         A を実スレッドへ割り当て;
     }
 }
  else{
     if(B が待ち行列へ格納されていた){
         A を実スレッドへ割り当て;
     }
     else{  /*B が実スレッドに割り当てられていた*/
         A を待ち行列へ格納;
     }
 }
}while(未判定のスレッドがある);
キューから必要な数のスレッドの取り出し;
ターゲットスレッドの設定;
```

図 6 スレッド相性スケジューラの概要
Fig. 6 The algorithm of SA scheduler.

により実スレッドへ割り当てるスレッドの入れ替えを行っていた．本研究では，スレッド相性スケジューラではスレッド相性の良し悪しによって，そのスレッドが，再び実スレッドに割り当てられて実行するか，待ち行列に格納するかを選択する．スレッド相性スケジューラの概要を**図 6** に示す．

4 つの論理スレッドを実行し，スレッドスケジューラが呼び出されたときを例にとって説明する．実行していた論理スレッド t1，t2，t3，t4 があるとする．さらに，t1 は t4，t2 は t1，t3 は t2，t4 は t3 とのスレッド相性を測っていたとする．

スレッドスケジューラが呼び出されると，最初に t1 を取り出す．t1 と t4 のスレッド相性が良い場合，t1 を再び実スレッドへ割り当てる．スレッド相性が悪い場合，t1 は待ち行列へ送る．1 つ目のスレッドは，スレッド相性の良し悪しだけで，再び実行するのか，実行待ち行列の最後尾に送るのかを選択する．

次に，t2 と t1 のスレッド相性を見る．t1 とのスレッド相性が良く，かつ，t1 が実スレッドへ割り当てられていた場合は t2 も実スレッドへ割り当てる．t1 を待ち行列へ格納していた場合は，t2 も待ち行列へ格納する．逆に，t1 とのスレッド相性が悪い場合で，かつ，t1 が実スレッドへ割り当てられていた場合は，t2 を待ち行列へ格納する．t1 が待ち行列へ格納していた場合は，t2 を実スレッドへ割り当てる．t2 以降も同様に，スレッド相性と，ターゲットスレッドが実スレッドへ割り当てられたか，待ち行列へ格納されたかによって t3，t4 の挙動が決まる．

スレッド相性が良い場合はそれぞれのスレッドが同時に実行されるか，あるいは，同時に待ち行列へ格納される．逆に，スレッド相性が悪いスレッドどうしは，再び同時に実行されないように，あるスレッドは実スレッドへ割り当てられ，あるスレッドは待ち行列へ格納される．すると，毎回のスレッドスケジューリングにより相性の良いスレッドどうしが同時に実行される機会が増し，キャッシュライン共有が増す．その結果，キャッシュミスの低下を防ぐことにつながる．

すべての判定後，スレッドが割り当てられなかった実スレッドには待ち行列から新たなスレッドを割り当てる．実行すべきスレッドが決定すると，ターゲットスレッドを設定し，並列実行に入る．たとえば t1，t4 が再実行となり，新たなスレッド t5，t6 を待ち行列から取り出した場合，t1 は t6，t4 は t1，t5 は t4，t6 は t5 がターゲットスレッドとなる．

以上のように，スレッドスケジューラはスレッド相性によってスレッドは再び実行するのか，待ち行列に格納するのかを選択する．各スレッドともターゲットスレッドを 1 つしか持つことができないため，この選択によってスレッド相性の良いスレッドが同時に実行されるようにする．

スレッド相性の悪いスレッドがある場合，スレッドの入れ替えが起こるが，すべての実行スレッドのスレッド相性が良い場合，それらは毎回，実スレッドに割り当てられるようにスケジューリングされ，同じスレッドがプロセッサを占有する．そのため，スレッド全体がプロセッサ資源を利用できるというスレッドの公平性に欠ける．しかし，本論文では性能向上を最も重要な課題と考えたことから，公平性は今後の課題とした．この問題を回避しようとした場合，長く占有しているスレッドを強制的に入れ替える方式や数回おきに実行中の全スレッドを入れ替える方式が有効と考えられる．

5.3 キャッシュの監視

プロセッサ全体のキャッシュヒット回数，キャッシュミス回数はプロセッサ内のスレッドコンテキストが計測しており，それらの数値をスレッドスケジューラがまとめ，プロセッサ全体のキャッシュヒット率を得る．

同様に，スレッド相性を得るには，プロセッサ上でスレッドコンテキストが持つ *share*，*exclude* で回数を計測する．あるスレッドがキャッシュラインにアクセスした際，タグ比較と同時に，そのキャッシュラインを使用している実スレッドナンバを比較する．実スレッドナンバは LTN と対応しているため，ターゲットスレッドが使用していて，共有かデータ入れ替えか

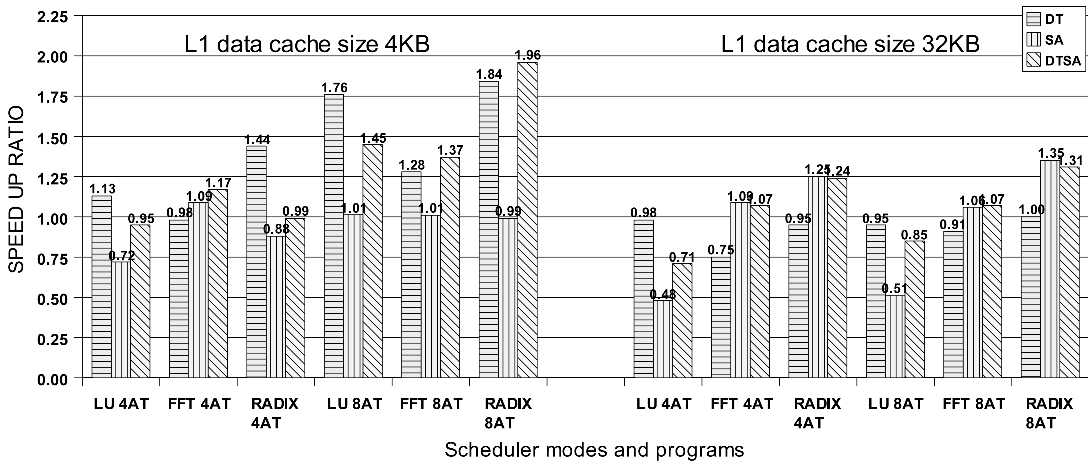


図 7 スレッドスケジューラによる速度向上率
Fig. 7 The speedup ratio by scheduler mode.

で，自身の *share*，*exclude* が 1 増える．

スレッドスケジューラはこれらのプロセッサで計測されたデータを得て，キャッシュ利用状況を監視する．

5.4 実 装

スレッドスケジューラをマルチスレッドライブラリ MULiTh 内に実装した．スレッドスケジューラは MULiTh のスレッド管理機構を利用し，論理スレッドの選択，実スレッドへの割当て，実スレッドの停止を行う．プログラムの作成は binutils-2.13☆，gcc-3.2☆☆，newlib1.9.0 を用いた．スレッドスケジューラが呼び出される周期 100 万サイクル中でのスレッドスケジューラのオーバヘッドはサイクル数で見ると 1%未満であった．

6. 評 価

評価には，SPLASH-2 ベンチマークより，LU 分解，FFT（高速フーリエ変換），および，RADIX ソートを用いた．SPLASH-2 のベンチマークは，それぞれ並列化マクロによって並列化した．すべてのプログラムは，それぞれ 16 個の論理スレッドを生成する．

評価に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*）[7] を用いた．MUTHASI は OChiMuS PE をシミュレートし，プロセッサのパラメータが設定可能である．予備実験により，4，8 実スレッドでの実行性能が大幅に低下した 1 次データキャッシュサイズ 4 KB と，逆に高い性能を発揮した 1 次データキャッシュサイズ 32 KB でのスレッドスケジューラの効果を評価する．スレッドスケジューラは，DT のみ，SA のみ，および DT+SA(DTSA) を適用し，実スレッド数 4，8 で実行した．

6.1 実 行 結 果

それぞれの実スレッド数，プログラムでスレッドスケジューラ適用前の実行と比較した．**図 7** ではスケジューラ適用前の IPC をそれぞれ 1 として，スレッドスケジューラ適用後の速度向上率を求めた．

LU 分解は，キャッシュのサイズにかかわらず，SA と DTSA の性能は向上しなかった．DT は，実スレッド数 8 キャッシュサイズ 4 KB で 1.76 倍の性能向上があった．

実スレッド数 FFT，実スレッド数 4，8，RADIX ソートは，キャッシュサイズ 4 KB では，DT の性能が高く，キャッシュサイズ 32 KB では，SA の効果があった．最大で，実スレッド数 8，RADIX ソートが DTSA で 1.96 倍の性能向上を示した．

6.2 考 察

全実行の IPC，速度向上率，および，1 次データキャッシュヒット率を**表 3** に示す．

1 次データキャッシュサイズ 4 KB では，DT の効果が強く現れ，目標である SMT プロセッサアーキテクチャの性能改善が達成できた．予備実験では，実スレッド数 4，8 ではキャッシュヒット率が低下し，それにともない IPC も低下した．DT 適用後は実スレッド数 4，8 の LU 分解では IPC が 1.60 にまで改善され，実スレッド数 2 の IPC1.43 以上の性能向上となった．これは，キャッシュサイズが小さく Capacity misses が多く発生したため，それに対応した DT の方式が性能改善につながった．DT ではつねにプロセッサ全体

☆ 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．

表 3 すべての実行結果
Table 3 The result of all experiments.

		LU			FFT			RADIX		
		IPC	SUR	1CHR	IPC	SUR	1CHR	IPC	SUR	1CHR
L1 cache size 4 KB	4AT	1.41	1.00	89.24%	1.05	1.00	92.80%	1.93	1.00	96.49%
	4AT DT	1.60	1.13	95.36%	1.03	0.98	96.62%	2.77	1.44	99.73%
	4AT SA	1.01	0.72	90.86%	1.14	1.09	95.21%	1.70	0.88	93.23%
	4AT DTSA	1.34	0.95	95.08%	1.23	1.17	96.92%	1.92	0.99	97.18%
	8AT	0.91	1.00	79.27%	0.75	1.00	83.90%	0.81	1.00	79.54%
	8AT DT	1.60	1.76	94.53%	0.96	1.28	95.37%	1.49	1.84	95.65%
	8AT SA	0.92	1.01	82.14%	0.76	1.01	84.63%	0.80	0.99	79.54%
	8AT DTSA	1.32	1.45	93.37%	1.03	1.37	94.12%	1.59	1.96	95.12%
L1 cache size 32 KB	4AT	2.90	1.00	99.68%	1.51	1.00	99.75%	2.99	1.00	99.96%
	4AT DT	2.85	0.98	99.66%	1.14	0.75	99.72%	2.85	0.95	99.95%
	4AT SA	1.38	0.48	99.71%	1.64	1.09	99.80%	3.74	1.25	99.96%
	4AT DTSA	2.07	0.71	99.73%	1.62	1.07	99.82%	3.70	1.24	99.96%
	8AT	3.26	1.00	99.53%	1.51	1.00	99.56%	3.83	1.00	99.94%
	8AT DT	3.09	0.95	99.07%	1.37	0.91	99.68%	3.84	1.00	99.93%
	8AT SA	1.66	0.51	99.57%	1.60	1.06	99.75%	5.18	1.35	99.95%
	8AT DTSA	2.78	0.85	99.58%	1.61	1.07	99.73%	5.03	1.31	99.94%

*SUR: Speedup ratio **1CHR:L1 cache hit ratio

のキャッシュヒット率が 90%以上となるような設計とした．その結果，キャッシュヒット率 90%以上を維持した実行ができた．結論として，複数のスレッドを多く並列に実行するよりも，キャッシュヒット率の低下を防ぐことが，SMT プロセッサアーキテクチャの性能向上において必要である．

また，1 次データキャッシュサイズ 32 KB では，SA の効果が強く現れた．キャッシュにデータを置くための領域が十分にあり，キャッシュミスがほとんど起こらない場合は，それぞれの実行結果で，1.50 から 3.90 近くまでの高い IPC となった．さらに，SA を適用後は，それ以上に性能が向上した．特に SA の効果が現れた FFT，RADIX ソートにおいて，SA は複数のスレッドの中からスレッド相性を見つけ出して，スレッド相性の良いスレッドを組み合わせることに成功した．キャッシュのデータの多くを共有するスレッドを組み合わせて実行すれば，キャッシュのデータ入れ替えなどの性能低下となる要因が減少し，SMT プロセッサ全体の性能向上につながった．ところが，LU 分解のように，SA の効果が現れない結果もあった．LU 分解ではスレッド 1 つ 1 つのデータ共有が少なかったためで，スレッド相性の良し悪しがあまり現れず，性能向上には至らなかった．

本スレッドスケジューラによって，SMT プロセッサはキャッシュの大小にかかわらず従来以上の性能向上を示した．論理スレッド割当てを制御できる DT の適用により，実スレッド数 8 以上の場合でも，最適な実スレッド数を決定して，可能な限りの高い性能を示すことができる．また，スレッド間のスレッド相性が良ければ，それを見い出し，組み合わせて実行する SA によって，従来以上の性能向上が見込める．

スレッドスケジューラの効果について考察したが，次の点は今後検討すべき内容である．スレッドスケジューラの呼び出し周期ついては，予備実験により今回 100 万サイクルが妥当であるとしたが，呼び出し周期はプログラムのコードやアーキテクチャに依存しているため，それぞれのコードに合った呼び出し周期が必要である．スケジューリング周期を設定できる方式，または，実行中に適正な周期を自動判別する方式が有効と考えられる．

また，性能低下防止のための適正な閾値を 90%に固定し，それぞれのプログラムで効果を示した．しかし，プログラムの内容やキャッシュへのアクセスパターンによっては，閾値を固定するのではなく，実行中に最適な閾値を見つけ出せれば，より性能が向上すると考えられる．あらかじめ閾値を設定できる方式や，実行中にキャッシュへのアクセスパターンによって閾値が自動的に変化する方式が有効と考えられる．

7. 関 連 研 究

関連研究に，Snavely らによるつねに最適なスレッドの組合せを求めるスレッドスケジューラ[4)]や，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)]がある．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．しかし，スレッド単体ごとの性能の関係には踏み込んでおらず，本研究ではスレッド相性を定義してスレッドどうしの関係からスレッドの組合せを求める方式を導入した．

また，大河原らの研究では，SMT プロセッサ上で，サンプリング期間中に可能なすべてのスレッド並列度を試行する．その結果から最適な実行スレッド数を決定する[6)]．本研究では，つねにサンプリングを行い続

け，毎回のキャッシュヒット率を得る．キャッシュヒット率と指標とを比較してスレッド実行数を決定するため，スレッド数決定のための試行期間は必要ない．また，スレッド相性，および，スレッド相性スケジューラを組み合わせることにより，それ以上の性能向上という結果が得られた．

8. 終わりに

一般的な SMT プロセッサアーキテクチャはキャッシュミスが性能低下の原因であり，SMT プロセッサアーキテクチャOChiMuS についても，キャッシュミスが性能低下の原因であった．

本研究では，キャッシュミスに対応する 2 つのスケジューラ，DT，SA，および，それらを組み合わせた DTSA を実現した．結果として，実スレッド数 8，RADIX ソートで，DTSA がスレッドスケジューラ適用前に比べて最大 1.96 倍の速度向上を示した．他のプログラムにおいても，DT，および，DTSA の適用によって，高い速度向上率を得た．

今後の課題としては，他の SMT プロセッサアーキテクチャで本スケジューラを適用することである．また，今回使用したベンチマーク以外のアプリケーションを使用して，スレッドスケジューラの性能評価を行い，効果的なスレッドスケジューラについて検討，考察する．

参 考 文 献

1) 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).
2) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, 3rd Edition, Morgan Kaufmann Publishers (2002).
3) Tullsen, D.M., Eggers, S. and Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.392–403 (1995).
4) Snavely, A. and Tullsen, D.M.: Symbiotic jobscheduling for a simultaneous multithreading processor, *Architectural Support for Programming Languages and Operating Systems*, pp.234–244 (2000).
5) Parekh, S., Eggers, S.J., Levy, H.M. and Lo, J.: Thread-sensitive scheduling for SMT processors, Technical report, Dept. of Computer Science, University of Washington (2000).
6) 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，情報処理学会研究報告，Vol.2002, No.112, pp.17–22 (2002).
7) 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).
8) 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).
9) 佐藤未来子，笹田耕一，加藤義人，大和仁典，河原章二，中條拓伯，並木美太郎：マルチスレッドアーキテクチャ向け OS「Future」におけるプロセス管理，情報処理学会論文誌：コンピューティングシステム，Vol.45, No.SIG3(ACS5), pp.38–49 (2004).
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：コンピューティングシステム，Vol.44, No.SIG11(ACS3), pp.215–225 (2003).
11) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.24–36 (1995).

(平成 17 年 1 月 24 日受付)
(平成 17 年 5 月 30 日採録)

内倉 要（学生会員）

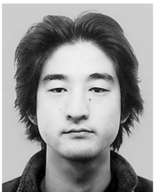
1981 年生まれ．2004 年東京農工大学工学部情報コミュニケーション工学科卒業．現在，同大学大学院工学教育部博士前期課程情報コミュニケーション工学専攻在学中．オペレーティングシステム，システムソフトウェアに興味を持つ．

笹田　耕一（学生会員）

1979 年生まれ．2004 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．現在，同大学院工学教育部博士後期課程電子情報工学専攻在籍．オペレーティングシステムやシステムソフトウェア，並列処理システム，言語処理系，プログラミング言語に関する研究に興味を持つ．

佐藤未来子（学生会員）

1966 年生．1990 年東京農工大学大学院工学研究科修了．同年（株）日立製作所入社，サーバシステムの設計・性能評価等に従事．2002 年より東京農工大学大学院工学研究科博士課程に在学中．オンチップマルチスレッドアーキテクチャプロセッサ，オペレーティングに関する研究に興味を持つ．



加藤　義人

2005 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．現在，ソニー株式会社に所属．プロセッサアークテクチャに興味を持つ．

大和　仁典

2005 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．現在，株式会社東芝に所属．オンチップマルチスレッドアークテクチャ，メモリアーキテクチャに関する研究に興味を持つ．

中條　拓伯（正会員）

1961 年生まれ．1985 年神戸大学工学部電気工学科卒業．1987 年同大学大学院工学研究科修了．1989 年神戸大学工学部助手の後，1998 年より 1 年間 Illinois 大学 Urbana-Champaign 校 Center for Supercomputing Research and Development（CSRD）にて Visiting Research Assistant Professor を経て，現在東京農工大学大学院共生科学技術研究部助教授．プロセッサアーキテクチャ，並列処理，クラスタコンピューティング，高速ネットワークインタフェースに関する研究に従事．電子情報通信学会，IEEE CS 各会員．博士（工学）．

並木美太郎（正会員）

1984 年東京農工大学工学部数理情報工学科卒業．1986 年同大学大学院修士課程修了．同年 4 月（株）日立製作所基礎研究所入社．1988 年東京農工大学工学部数理情報工学科助手．1989 年電子情報工学科助手．1993 年 11 月電子情報工学科助教授．1998 年 4 月情報コミュニケーション工学科助教授．博士（工学）．オペレーティングシステム，言語処理系，ウィンドウシステム等のシステムソフトウェア，並列処理，コンピュータネットワークおよびテキスト処理の研究・開発・教育に従事．ACM，IEEE 各会員．