プロセスの実行時情報を用いたスケジューラによる高速化手法

小川周吾[†,☆] 平木敬[†]

近年用いられる SMT のような複数スレッドが同時に同一プロセッサで動作するアーキテクチャにおいて，現在のオペレーティングシステムでは同時実行されるスレッド間の資源競合を回避することが困難であり，性能が低下する．本稿ではプロセッサの性能カウンタから得られるプロセスの実行時情報を用いてプロセッサのキャッシュ競合を低減するプロセススケジューリングを行う高速化方法を提案し，Linux カーネルへの実装と評価を行う．また，提案手法を実装してプログラムを動作させた際のスケジューラとプロセスの動きについて調べ，資源競合による性能低下が発生する際のスケジューリングの状況，提案手法によって得られる効果，実行時情報を用いたスケジューリングにおいて解決すべき問題点を提示する．

A Speedup Technique with Scheduler Using Process Execution Information

Shugo Ogawa[†,☆] and Kei Hiraki[†]

Major operating system does not avoid resource conflicts between threads on SMT architecture which runs more than one thread simultaneously in the same processors, then the performance fall occurred. In this paper, we propose the method which reduce the frequency of cache conflicts on processors using execution information from performance counters and use it for scheduling. We implement this method to Linux kernel and evaluate it, and we experiment on the behavior of scheduler and processes. By this experiment, we tell the behavior of scheduler in performance fall from resource conflict, the effect of our method in scheduler, and the problem to be solved in process scheduling using execution information.

1. 序論

近年，SMT [1] アーキテクチャ，Intel 社の Hyper-Threading [11] のように同一プロセッサ内で複数のスレッドの動作が可能なアーキテクチャを持つプロセッサが実用化されている．これらのアーキテクチャではプロセッサ内の資源が複数のスレッドで共有されるため，スレッド間で資源の競合が発生した場合の性能低下が問題となる．同時に実行されるスレッド間の競合は同一プロセッサで実行されるスレッドの種類，実行中の箇所の違い等の実行時の環境により競合の内容，発生頻度が異なる．そのため同時にプロセッサで実行するプロセスの組合せ，プロセスを実行する順序が資源競合の頻度を決定する重要な要素である．

マルチスレッドプロセッサを複数持つシステムの場合はさらに問題が複雑である．複数プロセッサの場合，資源競合頻度を決定する要素として各プロセスのプロセッサへの割当てが加わる．プロセスどうしの資源競合は同一プロセッサ上で実行される場合に発生する．

本稿ではこれらの問題を解決するためにプロセス間の実行時情報を利用したスケジューリングによる高速化手法の提案を行う．実験により提案手法の性能評価を行い，提案手法が性能向上をもたらす根拠，提案手法のさらなる改善に必要な要素を明らかにする．

以降，2 章では本稿の提案手法およびアルゴリズム，3 章では Linux カーネルへの実装について述べる．4 章では提案手法に対する評価実験と結果，スケジューリングの改善点を述べる．5 章ではより一般的なプログラムによる性能評価を行う．さらに 6 章で提案手法の改善を行う場合に必要となる要素について考察を行う．7 章で関連研究ついて，8 章ではまとめを述べる．

2. 提案手法

本稿ではマルチスレッドプロセッサで同時に実行さ

† 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
☆ 現在，日本電気株式会社
Presently with NEC Corporation

れるプロセスの組合せの最適化による高速化手法を提案する．プロセスを同一プロセッサ上で同時に実行することにより，プロセス間で資源競合が発生する．資源競合が少ない場合に実行性能が向上し，資源競合の少なさをプロセス間の親和性として定義する．本稿ではキャッシュミスの実行命令数に対する割合を指標に用いる．キャッシュに注目した場合，同時に実行されるプロセス間の親和性に加えて連続して実行されるプロセス間の親和性が性能に関係する[8),9)]．しかし，連続して実行されるプロセス間のキャッシュ競合は主にプロセスの切替わり直後の状態に影響を与えるものであり，同一プロセッサ上で同時に実行されるプロセス間の競合に比べて影響が小さい．本稿では問題を単純化するため，より性能に影響を与える同時実行を行うプロセスの組合せにのみ着目をする．

本稿ではプロセス間のキャッシュ競合を，プロセスの実行時情報を用いて調べる方法を提案する．実行時情報を用いたスケジューラを以下のように構成する．

(1) プロセッサ上のプロセス間資源競合情報を記録
(2) 親和性の高い実行プロセスの組合せを選択
(3) 資源競合情報を用いたプロセッサ間の負荷分散

(1) ではプロセス間のキャッシュ親和性情報を性能カウンタで記録する．(2) ではプロセス間の親和性情報を参照し，競合による性能低下を回避するためにプロセスの実行順序変更を行う．(3) ではプロセス間の親和性情報を用いたプロセスのプロセッサ間移動を行う．

2.1 資源競合情報の記録

本稿ではプロセッサ上の資源競合情報を正確に取得するためにプロセッサ上に備えられた性能カウンタを用いる．性能カウンタを備えたプロセッサではカウンタからプロセッサの動作状況を正確に取得することが可能である．これによりキャッシュミス回数および実行命令数を実行プロセスを切り替えるごとに取得する．

情報の取得は例として**図 1** のように行う．任意のスレッドでプロセス切替えが発生するごとに各スレッドごとのキャッシュミス回数を記録する．この値から各プロセスでの単位時間あたりのキャッシュミス回数を計算する．キャッシュミス回数が過去の記録より小さいほどプロセス間のキャッシュ競合が少なく，親和性が高いと判断する．各スレッドでの実行命令数についても同時に取得する．キャッシュミス回数は実行命令数の多少により影響を受ける．そのため実行命令数あたりのキャッシュミス回数を測定し，キャッシュ競合頻度の多少を調べ，親和性を判断する．各物理プロセッサごとに同時実行したプロセス間の親和性情報を表すキャッシュミス回数等の測定結果，平均値を記録する．

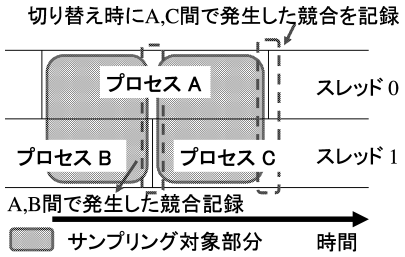

図 1 性能カウンタのサンプリング箇所
Fig. 1 Performance monitoring counter sampling points.

2.2 プロセスの実行順序変更

次のプロセスの選択時は記録された親和性情報を参照し，実行可能かつ現在同一プロセッサ内で実行中のプロセスとの間で資源競合の発生頻度の小さいプロセスを選択する．条件を満たすプロセスが存在する場合，優先度計算による実行順序を無視して次に実行する．

2.3 親和性情報を用いたプロセス移動

親和性の低いプロセスが同時に同一プロセッサ上で実行されないことでキャッシュ競合を低減する手法として実行順序を変更する以外に，同時に実行した結果得られた親和性情報を用いて親和性の低いプロセスが別のプロセッサ上で実行されるようにプロセスを配分する手法を提案する．各プロセッサで実行されるプロセス数が不均一になりプロセス移動の必要が生じた場合，提案手法によりプロセスの親和性情報を用いて各プロセッサ上のプロセスで他プロセスとの親和性が低いものを優先的に別プロセッサへ移動する．

しかし，新たにプロセスの生成，終了が発生せず，プロセッサの負荷が均一である場合はプロセス移動が発生しない．そこで本稿では定期的に他との親和性の低いプロセスから順に，プロセッサ間で 1 プロセスずつ能動的に入れ替えを行う手法を提案する．入れ替えを行い一定時間後に入れ替えたプロセスについて同一プロセッサで実行される他のプロセスとの親和性を求め，入れ替え前のものと比較を行う．その結果親和性が入れ替え前から低下した場合には入れ替えたプロセスを元に戻す処理を行う．これらの処理により各プロセッサで実行されるプロセスの組合せを改善する．

3. 提案手法の実装

本稿の提案手法を実装するために必要な事項について述べる．本稿では Linux 2.6 に対して実装を行う．

3.1 資源情報の記録

性能カウンタから得た情報を記録するテーブルを用意する．テーブルは物理プロセッサごとに確保し，テーブルへのアクセスによる lock 回数を抑える．テー

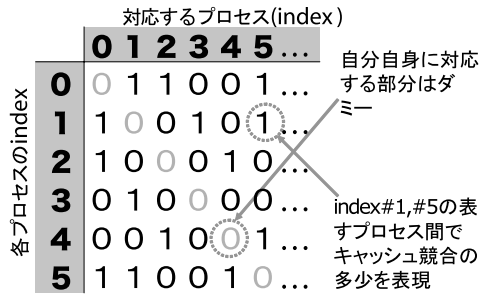

図 2 ビットマップによるプロセス間の親和性情報の表現例
Fig. 2 Example of bitmap which expresses process affinity.

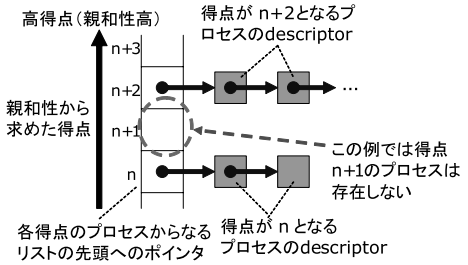

図 3 得点の低いプロセスを発見するためのデータ構造
Fig. 3 Data structure for finding low affinity process.

ブルには各プロセス間の親和性情報であるキャッシュミス回数を表すカウンタ値の平均値，プロセス情報を格納したタスク構造体へのポインタ等を記録する．プロセス間の親和性情報を扱うため，プロセス数が増大するとすべてのプロセスの情報を扱うことは困難である[4]．そこで情報取得対象のプロセスを限定し，テーブルのサイズは有限としてその範囲内で情報を扱う．本稿の実装環境ではテーブルサイズを 64 とする．

生成後短時間で終了するプロセスは情報収集の対象から外す．実行時間が短い場合，実行時情報による最適化の効果が低い．本稿ではプロセッサ占有時間が 3 秒以内のプロセスを対象外とする．またバックグラウンドプロセスのように長時間動作するが 1 回のプロセッサ占有時間の短いプロセスも実行時情報による最適化の効果が低いため情報収集の対象外とする．

情報収集対象のプロセスから次のプロセスに切り替わる際に親和性情報としてキャッシュミス回数を取得する．各プロセス間の情報は測定結果を直接格納せず，カウンタの値の平均値と大小を比較した結果のみをビットで格納する．具体的にはキャッシュミス回数と実行命令数の測定結果からなるプロセス間の親和性情報を**図 2** のとおりビットマップの形式でテーブル内の各プロセスを表す index に対応したビットに記録する．

大小判定の結果のみを格納することにより親和性が最大のプロセスの判定が不可能になる反面，以下の利点が存在する．第 1 に測定結果の格納に使用するメモリ量が削減される点があげられる．キャッシュミス回数の結果を 32 ビット整数で格納した場合と比較してビットマップで格納した場合は 32 分の 1 の容量となる．その結果提案方式の実装によるメモリ使用量が 48 K バイトから 16 K バイトに削減される．第 2 に親和性が最大のプロセスを求めるのではなく，親和性が高くなるプロセスの組合せのみを求めることで次に実行するプロセスの選択が高速化される点がある．親和性が最大のプロセスを求める場合，各プロセスとの親和性を表す測定値すべてを平均値と比較する必要がある．一方ビットマップ形式で結果を格納することで，これと現在の実行可能プロセスを表すビットマップとの間で論理積をとった結果真となるビットが親和性が高く実行可能なプロセスを表し，比較処理が不要となる．

よってカウンタ値を直接記録する場合と比較して履歴情報の使用メモリ量を削減し，参照が高速化される．

3.2 プロセスの実行順序変更

提案手法を用いる場合，先に標準のスケジューラで次に実行するプロセスを選択する．ここで選択したプロセスと同一プロセッサの別スレッドで実行中のプロセスとの親和性を調べ，両者の親和性が低い場合のみ提案手法を用いて次に実行するプロセスの選択を行う．

具体的には実行可能なプロセス中からプロセス間の親和性が高いものを選択する．そのために前項で述べた，親和性情報を記録したプロセスで実行可能なものを表すビットマップを用意する．一方各プロセス間の親和性情報についてもビットマップで格納されるので，両者の論理積を求めることで条件を満たすプロセスに対応するビットが真となることで結果が求められる．

3.3 親和性情報を用いたプロセス移動

プロセッサ内で他のプロセスと親和性の低いプロセスの優先移動を行うため，取得した親和性情報をもとに各プロセスに得点を算出する．プロセスごとにそのプロセスとの間の親和性が高いプロセスの数と親和性が低いプロセスの数を求め，その差をそのプロセスの得点とする．算出された得点から**図 3** のとおり各得点ごとに同一得点を持つプロセスのリストを作成し，得点からのプロセス検索を可能にする．プロセスをプロセッサ間で移動させる場合得点の低いプロセスを優先する．

3.4 スケジューリングの公平性

本稿の手法では資源競合の発生しにくいプロセスの組合せが可能な場合に実行順序の入れ替えられ，プロセスの優先度を無視したスケジューリングが行われる．

ここで Linux 2.6 のタスクキューに注目し，実行順序の変更が及ぼす影響について考察を行う．

Linux 2.6 のタスクキューとして各プロセッサごとに 2 つのキューが用意される．実行を待つプロセスは片方のキューに入れられ，これがタスクキューとして機能する．各プロセスにプロセッサ占有時間が割り当てられ，プロセスが割り当てられた時間を使いきるともう片方のキューに入れられる．これを繰り返し元のタスクキューが空になると，プロセスが移された先のキューが新たなタスクキューとなり，2 つのキューの役割が入れ替わる．この周期を epoch と呼び，本稿の手法ではタスクキュー内に存在するプロセス中に同時実行プロセス間の親和性をより高くするものが存在した場合にのみ順序を入れ替える．そのため同一 epoch の範囲でのみプロセスの実行順序の入れ替えが発生する．

以上により本稿の手法では単一 epoch に着目した場合には優先度の無視により実行順序が不公平になる．しかし単一 epoch より長い時間に着目した場合，各プロセスへのプロセッサ割当て時間は等しい．これにより単一 epoch より実行時間が長いプロセスの動作について，本稿の手法では公平性は問題にならない．

4. 性能評価および効果

提案手法のプロセス実行性能に対する効果を検証するために Linux カーネルに対する提案手法の実装の有無による実行性能の変化を調べる実験を行う．実験には以下の環境を使用する．CPU は内部に 2 つのスレッドを持つマルチスレッドプロセッサである．

- CPU：Intel Xeon2.80 GHz × 2（L2 Cache 512 KB）
- Memory：1 GBytes
- OS：Linux 2.6.3

4.1 行列乗算プログラム

提案手法の効果を検証するため行列乗算を行うプログラムを作成し，実行時間を測定する．プログラムでは行列をブロック化しマルチスレッドで計算を行う．

本稿の手法の効果の検証を容易にするためスレッド実行の最適化が可能な条件を設定し，以下の手順で計算を行う．プログラム全体の計算スレッドを 2 つのグループに分ける．それぞれのグループごとに**図 4** に示すとおり 1 つの列について計算を行う．各グループのスレッドは，キャッシュの共有が可能であり競合が少ない．また，行列のブロック化サイズはプロセッサのセカンドキャッシュの容量に近くなるように設定する．

実験では 4900 × 4900 の行列に対する乗算を行う．行列を 14 × 14 にブロック化し，各ブロックごとにスレッドに分割して計算を行う．プログラムを実行した際の実行時間を提案手法の使用の有無で比較する．計算スレッド数を 8 として各スケジューラに対して 10 回ずつテストを行う．また，比較用に各スレッドを最適なプロセッサに配置した場合の計測を行う．

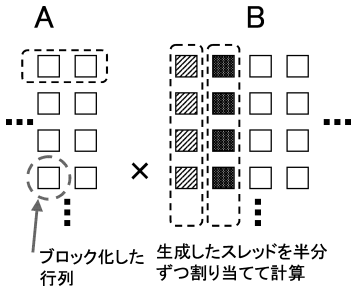

図 4 行列乗算プログラムの動作
Fig. 4 Behavior of matrix multiplication program.

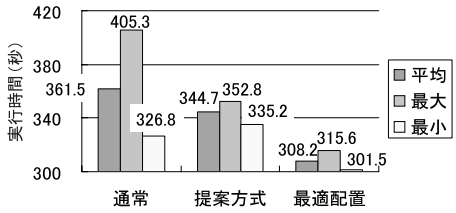

図 5 行列乗算プログラムの実行時間
Fig. 5 Execution time of matrix multiplication program.

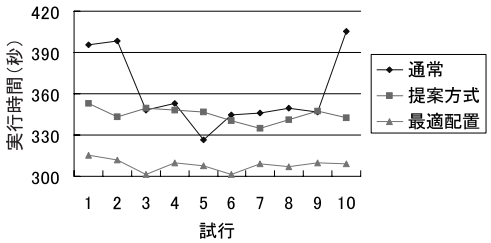

図 6 各テストでの行列乗算プログラムの実行時間
Fig. 6 Execution time of matrix program in each test.

4.2 実験結果

実験の結果，平均実行時間は**図 5** のとおりとなる．また，各試行における実行時間は**図 6** のとおりとなる．提案手法を用いた場合は用いなかった場合と比較して実行時間の差が減少している．特に最大実行時間について 50 秒以上と約 12%短縮され，処理時間の幅を抑えていることが分かる．また，提案手法によって平均実行時間についても短縮され，実行時間の幅を減少しつつ平均的な性能の向上を実現している．

また，プログラムのキャッシュミス回数についての結果は**図 7** に示したとおりである．提案手法を用い

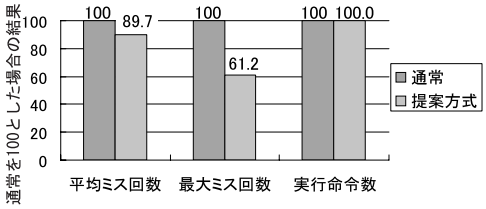

図 7 行列乗算プログラムのキャッシュミス回数
Fig. 7 Number of cache miss in matrix program.

	平均サイクル数	処理時間の割合
通常	3,920	0.0015%
提案手法	26,078	0.0096%

図 8 スケジューラのオーバヘッド
Fig. 8 Overhead of scheduler.

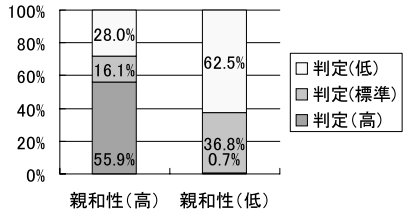

図 9 スレッド間の親和性の判定結果
Fig. 9 Accuracy in thread affinity judgement.

た場合，プログラム全体のキャッシュミス回数が平均10%，最大のキャッシュミス回数については約38%減少していることが分かる．また，実行命令数の提案手法の有無による差は存在しない．これにより提案手法が実際にキャッシュミスを低減する効果を有することが分かる．

実験時のスケジューラのオーバヘッドは**図 8** のとおりである．提案手法を用いた場合に，通常のスケジューラと比較して約6.6倍とより長い処理時間を要する．しかしスケジューラが占める処理時間は全体の0.01%に満たない．また実験での処理時間の短縮幅に比べてスケジューラにおける処理時間の増加は小さい．

スレッド間の親和性についてスケジューラが判定を行った結果については**図 9** のとおりである．提案手法ではスレッド間の親和性について高い/低い/標準の3段階で表現する．親和性が低いスレッドの組合せに対して判定を行った場合，親和性が高いと誤判定をした率は約0.7%であるのに対して親和性が高いスレッドの組合せに対して判定を行った場合は，28.0%が親和性が低いと誤って判定される．

4.3 実行時情報利用のスケジューリングへの効果

本稿の手法によるプロセススケジューリングの改善点をスケジューラの挙動から明らかにする．

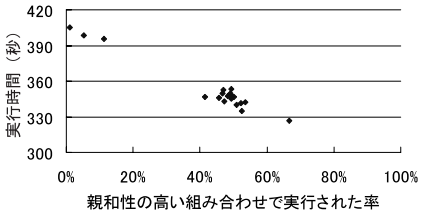

図 10 実行するスレッドの組合せと性能の関係
Fig. 10 Relation between performance and pair of running threads.

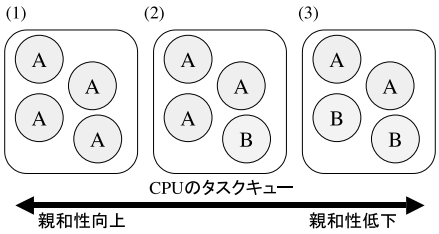

図 11 各物理プロセッサに割り当てられるスレッドの組合せ
Fig. 11 Combination of threads assigned to one CPU.

4.3.1 スレッドの配置と実行性能との関係の考察

提案手法がもたらす効果について述べる前に，スレッドの配置と性能の関係について明らかにする．実験で用いた行列乗算プログラムでは8つの演算スレッドを2グループに分け，それぞれのグループがブロック化された別の列を計算する．各グループのスレッドをA，Bとする．別グループのスレッドは参照領域が異なるためキャッシュ競合が発生し，親和性が低い．

本稿の環境ではプロセッサ内の2つの論理プロセッサ（スレッド）でスレッドを実行し，その間でキャッシュが共有されるため同時実行するスレッドの組合せが最も親和性に影響する．これを示すのが**図 10** である．プログラム中で親和性の高いスレッドが同時実行された時間の割合と実行時間の関係を示している．

親和性の高いスレッドの同時実行される率と実行時間が逆比例することが分かる．また両者の相関係数は−0.98となり相関関係が強い．したがって本稿の行列乗算プログラムではプロセッサ内で同時実行されるスレッドの組合せが性能に関係することが確認できる．

また，同時実行されるスレッドの組合せは複数のマルチスレッドプロセッサを持つシステムの場合，各物理プロセッサへのスレッドの配分に影響される．他に動作するプロセスが存在しない場合，プロセッサでは全スレッドが均等に実行される．各プロセッサ上のプロセス数が偏ると一定時間ごとに `load_balance()` によりプロセッサ間の負荷分散を行われプロセス数は均一に保たれる．本稿の実験では各プロセッサに配分されるスレッドの組合せは**図 11** で示される．このとき

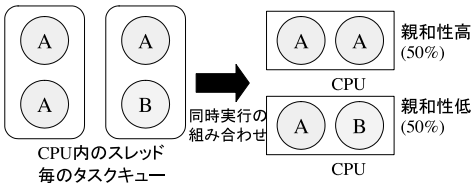

図 12 同一プロセッサ内で同時に実行されるスレッドの組合せ
Fig. 12 Pairs of threads running simultaneously on a CPU.

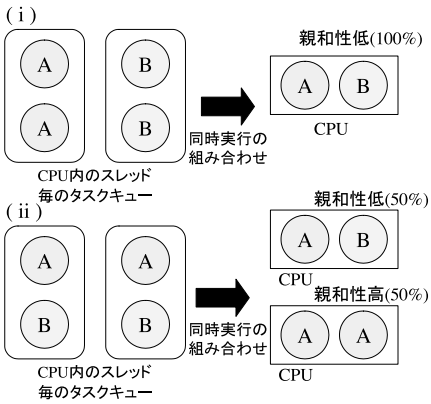

図 13 同一プロセッサ内で同時に実行されるスレッドの組合せ
Fig. 13 Pairs of threads running simultaneously on a CPU.

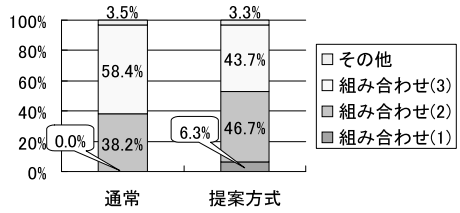

図 14 各プロセッサで実行されるスレッドの組合せ
Fig. 14 Rate of thread combination for each CPU.

	組合せ (i) の占める割合
通常	43.8%
提案手法	27.6%

図 15 組合せ (3) でのプロセッサ内のスレッド配分 (i) の比率
Fig. 15 Rate of the case (i) in thread combination (3).

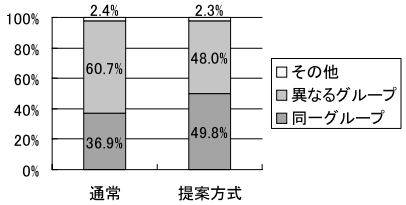

図 16 同時に実行されるスレッドの組合せ
Fig. 16 Rate of thread pairs on CPUs.

同一プロセッサ上で同時実行されるスレッドの組合せについて図 11 の配分に従って述べる．

(1) の組合せではプロセッサで実行されるスレッドがすべて同グループである．よってつねに親和性の高いスレッドどうしが同時実行されるため，最も性能が高い．

(2) の組合せでは必ず片方の論理プロセッサに親和性の低いスレッドが 1 個入る．そのため**図 12** のとおり同時実行されるスレッドの組合せは 2 通り存在し，親和性の高いスレッドの組合せで実行される確率は 50%であり，親和性の高いスレッドどうしが同時実行される時間の期待値は全時間の 50%である．

(3) の組合せでは，**図 13** のとおり，論理プロセッサへのスレッドの配置が 2 通り存在する．(i) の場合，各論理プロセッサで実行されるスレッドはつねに A，B であるためつねに親和性の低いスレッドの組合せで実行される．(ii) の場合は各論理プロセッサにおいて A，B ともに実行される可能性があるため，親和性の高いスレッドどうし，低いスレッドどうしの両方の組合せで実行される可能性がある．この場合親和性の高いまたは低いスレッドが同時に実行される率はそれぞれ 50%となる．よって組合せ (3) のうち (i) が最も性能が低くなる組合せであり，(ii) に関しては親和性の高いスレッドの同時実行時間の期待値が (2) の組合せと等しくなる．また (i) と (ii) の組合せについても各々同率で発生するため，親和性の高いスレッドが同時に実行される時間の期待値は全実行時間の 25%となり 3 種類の組合せの中で最も低い．

以上により，プロセッサ上で同時実行されるスレッド間の親和性が性能に影響し，さらに同一プロセッサで同時実行されるスレッドの組合せの決定に各物理プロセッサへのスレッド配分が影響する．

4.3.2 提案手法によって得られる効果

提案手法のスケジューリングへの効果について述べる．各プロセッサに割り当てられたスレッドの組合せの分布は**図 14** のとおりである．提案手法を用いた際の図 11 中の組合せ (1)，(2) の増加と (3) の減少が確認できる．一方組合せ (3) 中の (i)，(ii) の比率は**図 15** のとおりである．提案手法を用いた際にキャッシュ競合を多く発生させる (i) の減少が確認できる．

また，同時に実行されるスレッドの組合せについては**図 16** に示したとおりである．こちらも提案手法を用いた場合親和性の高いスレッドが同時に実行される率が上昇し，提案手法が有効であることを示している．

さらに 1 回のプログラム実行中でスレッドのプロ

	変動回数の平均値
通常	2.1
提案手法	34.8

図 17 各プロセッサで実行されるスレッドの組合せの変動回数
Fig. 17 Number of changing thread combination on CPUs.

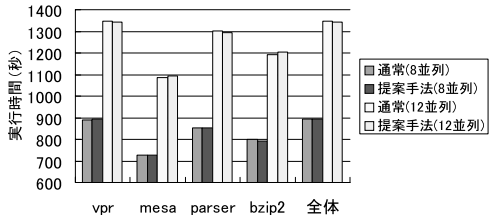

図 18 SPEC CPU2000 によるテスト結果
Fig. 18 Comparison of SPEC CPU2000 execution time.

セッサへの配置が変化した回数を示す．図 11 で示した組合せが変動した回数は**図 17** のとおりである．提案手法では 10 倍以上の変動が発生している．これは提案手法で，スレッドの最適配置を探索するためにプロセッサ間でのスレッドの移動を行うことによる．またこの移動で親和性の低い状態の持続が抑制され，性能が向上している．逆に最短実行時間については提案手法を用いない場合により優れた結果が得られているが，これは各スレッドに対して高い性能が得られる配置となった場合，通常のスケジューラではプロセスの移動が発生しないため，より性能が伸びやすいためである．

以上により本稿の提案手法は，スレッドの各プロセッサへの割当てを改善する効果，親和性の低い状態の持続の抑制効果を持ち性能向上に効果を持つ．

5. SPEC CPU2000 による性能評価

一般のアプリケーション実行した際の性能評価として SPEC CPU2000[12] を用いて実験を行う．実験では複数のシングルスレッドのテストを並列に動作させ実行時間を測定する．実験に用いるテストは並列動作時の性能を測定するために実行時間が比較的近いものとして，INT より 175.vpr，197.parser，256.bzip2 を，FP より 177.mesa を選択する．これらを同時に各 2，3 本ずつ，全体で 8 本および 12 本のテストを並列動作させて実行時間を測定する．この実験を 8 並列の場合と 12 並列の場合それぞれについて 10 回計測を行った結果，結果は**図 18** に示したとおりである．

並列数によらず，提案手法を用いた場合の全プログラム実行が完了する平均時間の短縮幅は 1%未満である．また各プログラムの提案手法の有無による実行時間差についても最大で 1%前後である．SPEC による実験において提案手法の有無にかかわらず性能が向上しない原因について考察を行う．まず SPEC の各ベンチマークでは行列乗算プログラムの各スレッド間のように共通のメモリ領域の参照が発生しない．そのためキャッシュの共有が発生せず，容量，ブロックの競合によるキャッシュミスが発生しやすくなり，同時実行するプロセスの選択によるキャッシュミスの低減が困難となる．そのため，キャッシュ競合頻度の低減の効果が行列乗算の実験と比較して低くなる．またプロセッサ上のスレッド間では，演算器等のキャッシュ以外のプロセッサ資源が共有されるため本稿の実装ではそれらの使用に関する競合の検出が不可能である．そのためキャッシュミス以外の要因で競合が発生してもそれを検出して回避するようにスケジューリングを行うことが不可能である．さらに，本稿で使用した 175.vpr は処理が 2 プロセスに分かれるため実行時間が短くなり，履歴情報を利用する本稿の手法では効果が低くなる．

この実験において提案手法は行列乗算プログラムほどの効果は示さなかった．しかし性能の低下も見られず，効果が得られる状況以外での性能低下は見られない．以上により本稿の手法は，特定の効果を示すプログラムの実行を効率化するために他のプログラムの性能を低下させずに実装が可能である．

6. 提案手法の問題点と改良

提案手法をさらに改善するため，行列乗算プログラムによる実験で判明した問題点について考察を行う．

まず提案手法では同時に実行されるスレッド間の親和性が高くなるスレッドの配分を，最適解となる配置の情報を持たずにスレッドを 1 つずつ移動させて性能変化を見ることで調べる．この方法は最適解を求める必要がなく，計算量が少なくなることが利点である．しかし必ずしもスレッド移動が性能向上に結びつかない欠点がある．また，スレッド配分が最適状態にある場合にそれを認識できないため，スレッド移動が発生する．性能向上に必要のないスレッド移動が発生することが問題点である．各スレッド割当ての状態についての平均移動回数は**図 19** のとおりである．組合せ (1) から (2) への変動が発生し，性能が低下する方向の状態変化が発生していることが問題である．

スレッドのプロセッサへの配分を最適化する別の方法としてスレッド間の親和性情報を用いて最適な配分を求め，それに従ってスレッドを移動させる方法があげられる．この方法の利点は最適解となるようにプロ

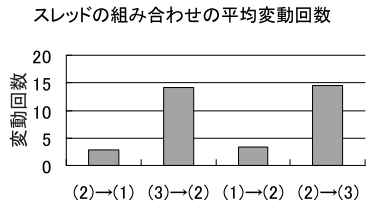

図 19 各プロセッサで実行されるスレッドの組合せの変動回数
Fig. 19 Number of changing thread combination on CPUs.

セスを直接移動可能なため，スレッドのプロセッサ間移動が少なくて済むことである．しかし本稿の実装環境である Linux 2.6 ではプロセス数に対して $O(1)$ の性能を実現しているため，システム中のプロセス数に比例した演算を行うことでスケジューラの性能低下を引き起こす．最適解を求めることによるプロセス数が増加した際のスケジューラの性能低下が問題点である．

もう 1 つの問題としてスレッド間の親和性の誤判定があげられる．親和性の誤判定の発生によって親和性の履歴に誤りが生じ，図 19 で示したようにスレッドの移動が不適切に行われるため，性能が低下する．資源競合を正しく検出するためにはより細かくプログラムの動作状況を把握する必要がある．キャッシュ以外の資源競合について同様にハードウェアから情報を取得することにより，実行時の状態をより正確に判断することが可能である．しかし，検出可能なすべての情報を用いると判定の精度は上がるものの，判定に必要とされる資源の使用量の増加，親和性判定条件の複雑化，取り扱うデータ量の増加，等の問題が発生する．

以上より本稿で提案した，プロセスの実行時情報をハードウェアで収集して利用するスケジューリングについて性能を改善するには，以下の点が必要である．

(1) 効率的に最適スレッド配置を行うアルゴリズム
(2) スレッド間親和性のより正確かつ効率的な判定

(1) については，最適配置に効率的に近づけるためのスレッドの移動アルゴリズム，または少ない計算量でスレッド配置の最適解または近似解を求めるアルゴリズムの提案が必要である．(2) については，性能カウンタで取得する情報の取捨選択と効率的に親和性を判定するアルゴリズム，データ構造の提案が必要である．

7. 関 連 研 究

Snavely ら[2] はマルチスレッドアーキテクチャにおいてスレッドを同時実行した際の性能の評価の基準として “Symbiosis” という基準を用いて，スレッドを同時に動かすかどうかの判断方法を提案している．そして 2 種類のベンチマークを SMT プロセッサで同時に実行した場合で評価を行っている．また文献 3) では，SMT プロセッサ上で同時実行するプロセスの組をハードウェア資源を考慮して決定することの有用性を指摘している．プロセッサの性能カウンタを用いてハードウェア資源の利用状況を調べ，同時に実行する最適なスレッドの決定を行う手法を提案している．これらの研究では，プロセス実行状況のサンプリングと，同時実行プロセスの最適化の 2 段階に処理を分ける点が本稿の手法と異なる．また，同時実行するプロセスの最適化を行う際に全プロセスの組合せを調べる点，プロセス切替えが全スレッドで同時に発生すると仮定する点が本稿の手法と異なる．

Zaki ら[4] は同様に SMT プロセッサでは複数プロセスで資源が共有され，従来のスケジューリングでは性能が低下することを指摘している．また，性能カウンタで各資源の競合を検出，回避するスケジューリングによる性能向上をシミュレーションで示している．Snavely ら[2] の研究と比較してこれらの問題に対してそれぞれ最後にサンプリングを行った時点での情報のみを利用してスケジューリングを行う，実行中プロセスと実行準備完了状態のプロセスから収集された情報を用いる改良を行い性能を測定している．

Parekh ら[5] も SMT プロセッサでは従来のスケジューリングでは性能が低下することを指摘している．そして資源の利用状況を考慮したスケジューリングとの性能差を示し，スケジューリングにおいて資源利用を考慮することの有用性を示している．

これらの研究と本稿の手法との違いは，本稿での提案では演算資源のうちキャッシュミスのみに注目することで単純化を図っている点，実装を行い評価を行っている点，SMT プロセッサを複数持つ環境でのプロセスの移動に履歴データを活用する点である．

8. ま と め

本稿ではマルチスレッドプロセッサに対する現在のオペレーティングシステムにおけるプロセススケジューリングの問題点を指摘し，その解決手法としてプロセッサの性能カウンタから得られるプロセスの実行時情報を用いたスケジューリングを提案した．また実行時情報から得たプロセス間の親和性情報をプロセスのプロセッサへの配分に応用する手法を提案した．

この提案手法を Linux2.6 上で実装し，プログラムで性能を評価した．その結果行列乗算を行うマルチスレッドプログラムでは平均実行時間が約 5%削減され，キャッシュ競合を発生させるスレッドが存在する場合

にプロセススケジューリングの最適化操作により性能向上が可能であることが示された．また，最大実行時間が最大で約 12%削減され処理時間の幅を平均処理時間を増大させずに抑えられることが示された．

SPEC CPU 2000 による性能測定では実行性能に向上は見られなかったものの提案手法による性能低下は発生しなかった．これにより本稿の提案手法が特定のプログラムの性能向上を目的として実装された場合に他の性能に悪影響を与えにくいことが示された．

プログラム実行時のプロセッサへのスレッドの割当てについて検証し，スレッド割当てと性能の関係について考察を行った．これにより本稿の提案手法はスレッドの各プロセッサへの割当ての改善による性能向上，親和性の低い状態の持続の抑制効果を持つことを示した．また提案手法に未解決の問題点が存在し，十分な効果が得られていないことを示した．さらに問題点を解決し，効果を高める方法を述べた．

以上により本稿の提案手法はマルチスレッドプロセッサを持つシステムのスレッド間の資源競合問題を解決するために有効であることを示した．

参 考 文 献

1) Hirata, H., Kimura, K., Nagamine, S., Mochizuki, Y., Nishimura, A., Nakase, Y. and Nishizawa, T.: An elementary processor architecture with simultaneous instruction issuing from multiple threads, *Proc. Annu. Int. Symp. on Computer Architecture*, pp.136–145 (1992).
2) Snavely, A., Mitchell, N., Carter, L., Ferrante, J. and Tullsen, D.: Explorations in Symbiosis on two Multithreaded Architectures, *Workshop on Multi-Threaded Execution, Architecture and Compiler* (1999).
3) Snavely, A. and Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.234–244 (2000).
4) Zaki, O., McCormick, M. and Ledlie, J.: Adaptively Scheduling Processes on a Simultaneous Multithreading Processor, Technical report, University of Wisconsin-Madison (2000).
5) Parekh, S. and Eggers, S.: Thread-Sensitive Scheduling for SMT Processors, Technical report, University of Washington (2000).
6) [announce] [patch] ultra-scalable O(1) SMP and UP scheduler. http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.0/0810.html
7) Barton, J.M. and Bitar, N.: A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX, *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, pp.45–69 (1995).
8) Vaswani, R. and Zahorjan, J.: The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors, *Proc. 13th ACM symposium on Operating systems principles*, pp.26–40 (1991).
9) Torrellas, J., Tucker, A. and Gupta, A.: Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors, *Journal of Parallel and Distributed Computing*, vol.24, pp.139–151 (1995).
10) The IA-32 Intel Architecture Software Developer's Manual. http://www.intel.com/design/Pentium4/documentation.htm
11) Hyper-Threading Technology Architecture and Microarchitecture. http://www.intel.com/technology/itj/2002/volume06issue01/art01_hyper/vol6iss1_art01.pdf
12) SPEC CPU2000. http://www.spec.org/cpu2000/

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

小川 周吾（正会員）

東京大学理学部情報科学科卒業，東京大学大学院情報理工学系研究科修士課程修了．現在，日本電気株式会社システムプラットフォーム研究所勤務．

平木 敬（正会員）

東京大学理学部物理学科卒業，東京大学大学院理学系研究科物理学専門課程博士課程退学，理学博士．工業技術院電子技術総合研究所，米国 IBM 社 T.J.Watson 研究センターを経て現在東京大学大学院情報理工学系研究科勤務．数式処理計算機 FLATS，データフロースーパコンピュータ SIGMA-1，大規模共有メモリ計算機 JUMP-1 等多くのコンピュータシステムの研究開発に従事，現在は超高速ネットワークを用いる遠隔データ共有システム Data Reservoir システムの研究，超高速計算システム GRAPE-DR の研究を行っている．