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

2004－ARC－159　(10)
2004／7／30

ハードウェア統計情報を用いたプロセスの動的な最適スケジューリング手法

小川 周吾 †　　平木 敬 †

† 東京大学大学院情報理工学系研究科

要 旨

現在普及している主なオペレーティングシステムでは公平性を重視し，プロセス間でのプロセッサ資源の競合状況を把握した状態でプロセススケジューリングを行っていない．そのため近年用いられている 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では，同時実行されるスレッド間でのキャッシュなどの資源の競合を回避することが困難になり，実行性能が低下する．
本稿ではハードウェアから得られる統計情報を用いてプロセッサのキャッシュミス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する．具体的にはプロセッサの性能カウンタから得られる時間あたりのキャッシュミス回数の情報をもとに，プロセスの実行順序，プロセッサの割り当てを最適化する機能を Intel 社の Xeon プロセッサ上で動作する Linux のスケジューラ上で実装し，評価を行った．

Dynamic Optimization for Process Scheduler Using Hardware Statistics Information

Shugo Ogawa†　　Kei Hiraki†

†Graduate School of Information Science and Technology, University of Tokyo

Abstract

Major operating system schdule threads based on fairness rather than reducing resource conflicts between processes. This causes the performance fall on SMT architecture which runs more than one thread simultaneously in the same processors because operating system does not avoid cache conflicts.
In this paper, we propose the method which reduce cache misses on processors by using the statistics information from hardware in operating system scheduler. We implements the method which optimize the execution order and the processor assignment of processes by using the number of cache miss in a period from performance monitoring counter, and we evaluate this with Linux process scheduler running on Intel Xeon processor.

1　序論

現在普及している汎用的なオペレーティングシステムでは全てのプロセスに対して資源の占有時間または実行順序に関して，実行時の性能向上より公平性を重視したスケジューリングを行っている．

近年，Intel 社の HyperThreading[11] 等に代表される SMT[1] のような同一プロセッサ内で複数のスレッドを動作させるアーキテクチャを持ったプロセッサが実用化，あるいは提案されている．これらのアーキテクチャでは，同一プロセッサ内に存在する資源を複数のスレッドで共有するため，スレッド間で資源の競合が発生する．資源の競合を防ぐには，プログラムコードを生成する段階で資源競合が発生する可能性を抑えるようなコードを生成することで回避することが可能である．しかし上記のアーキテクチャを採用したシステムでは，複数のプロセスで資源を共有することになるため，同時に実行するプロセスの組み合わせ，プロセッサの割り当てが資源競合の回避にとって重要となる．

SMT プロセッサで同時実行するスレッドの組み合わせについて，Snavely ら [2],[3] はハードウェア資源を考慮して利用状況を考慮しプログラム同士の親和性を求め，これを基準に同時実行を行うプロセスの組み合わせを決定することで IPC が向上することを実験により示している．

SMT プロセッサが複数存在する場合は同時に実行するプロセスの組み合わせの問題に加えて資源競合を起こしやすいプロセス同士が同一物理プロセッサで実行されることで性能低下を引き起こす可能性が存在する．複数プロセッサ間でのキャッシュ親和性を考慮したプロセスのスケジューリングについては Torrellas ら [9] がキャッシュを考慮したプロセススケジューリングを行うことでプロセスのキャッシュミスを削減し，実行時間の短縮が可能であることを示している．加えて SMT プロセッサではキャッシュ親和性についてキャッシュを複数のスレッドで共有するため，直前に実行していたプロセスのキャッシュ状態が影響するだけでなく同時に実行されるプロセスとのキャッシュ競合が実行性能に影響するため，キャッシュ親和性の考慮が性能により大きな影響を与えると言える．

本稿ではこれらの問題を解決するため，プロセス，スレッドの実行順序，プロセッサの割り当てについてハードウェアから得られる統計情報を用いた割り当てを行うことでプロセッサのキャッシュミスに注目し，これを低減して高速化を行う方法を提案する．更に提案手法を Intel 社の Xeon プロセッサ上で動作する Linux スケジューラ上で実装を行う．

以降，2 章において本稿の提案手法，アルゴリズムについて述べ，続いて 3 章では Linux カーネルへの提案手法の実装方法について述べる．4 章では行列乗算アプリケーションで性能評価を行った結果について述べる．5 章では既存の関連研究ついて，6 章では本稿のまとめについて述べる．

2　提案手法

SMT プロセッサでの同時に実行するプロセスの組み合わせの違いによるプロセスの実行性能の低下を防ぐためには各プロセスのハードウェア資源の利用状況についての情報が必要となる．

この問題に対して我々はプロセッサの性能カウンタを用いてプロセスの実行切り替え毎にプロセスの動作性能の情報を収集し，その統計情報を用いて動的にプロセススケジューリング，プロセッサ割り当てを行う方法を提案する．

プロセッサの性能カウンタを用いることで，各タイムスライス内でのプロセッサでのキャッシュミスの回数を計測する．各タイムスライスの終了毎にカウンタの値を読み出してキャッシュミス回数を記録する．取得した値から実行していたプロセスのキャッシュミス回数の平均値を計算し，更にタイムスライス内でのキャッシュミス回数を求めた平均値を基準として評価し，その大小を同時に実行しているプロセスとのキャッシュ競合の頻度として記録する．この統計情報からキャッシュ競合がより減少する実行プロセスの組み合わせを探し出し，プロセスの実行性能を向上を図る．

2.1　アルゴリズム

本稿の目的である，プロセスの実行性能低下を回避するスケジューリングを行うために我々は以下の 2 つの処理を行うことを考え，そのアルゴリズムについて述べる．

1. 次に実行するプロセスについて同一プロセッサ内の別スレッドとの間のキャッシュ競合が少なくなる，あるいは実行命令数がより多くなるプロセスを選択
2. 資源競合を考慮したプロセッサ間の負荷分散

1 ではプロセッサ内でのスケジューリングの効率化，キャッシュの親和性についての評価を行い，2 ではその情報を用いたプロセッサの割り当てについて述べる．

2.1.1　プロセス実行順序の変更

まず 1 の処理を実現するために必要である同一プロセッサにおいて同時実行させた場合のキャッシュ競合の少ないプロセスの組み合わせを発見する手段を述べる．近年用いられているプロセッサにはプログラムの実行状況を観察し，その改善のためにプロセッサの動作状況を取得するために性能カウンタとしてモニタリング機能を持つものがある．例として，実行命令数や分岐予測のミス回数，メモリアクセス回数等の測定が可能である．本稿では最もメインメモリに近いレベルのキャッシュのミス回数及びプロセス内の実行命令数をプロセス切り替えの際に性能カウンタから取得する．これらの値を他のスレッドで実行中のプロセスと組み合わせて実行させた場合の実行性能の指標として用いる．

性能カウンタから取得した値を用いて各プロセス毎に，タイマの計測単位である tick あたりの平均実行命令数及び平均キャッシュミス回数を計算する．これらの値と性能カウンタの値から計算した tick あたりの実行命令数，キャッシュミス回数を比較しその大小をビットとして記録する．ビットとして記録することにより，後述する各プロセスの得点による評価を容易にする．

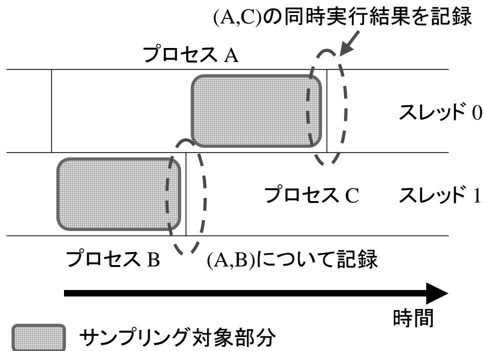

図 1: 性能カウンタのサンプリング箇所

性能カウンタの取得タイミングは図 1 に示したようになる．例えばプロセス A の切り替わり時にはプロセス C と同時実行した場合の性能を示す値としてサンプリングを行う．

性能カウンタから取得したデータはプロセスの切り替わり時にそれまで実行していたプロセスについてのみの資源利用情報として値を扱うのではなく，同時に実行していたプロセスと組み合わせて実行した場合のそのプロセスの実行性能値として扱う．そのためサンプリングと最適化のフェーズを明示的に分離せず，平行して処理を行うことが可能である．

この結果を履歴として記録し後に実行するプロセスを選択する際に，今まで蓄積した履歴を用いて同時実行した場合の性能が良いと記録されているものを可能な限り選択する．更に同時実行したプロセスが終了した際に再び測定結果をもとに実行性能を計算し，新しい結果を記録する．

2.1.2　統計情報によるプロセッサ間負荷分散

次に 2 のプロセッサ間でのプロセスの移動による負荷分散について述べる．一般的な負荷分散方法として各プロセッサに割り当てられる走行中のプロセス数が等しくなるような配分を行うことが考えられる．しかし，プロセス数のみから判断して配分を行うことで資源競合の発生しやすいプロセス同士が同一プロセッサに含まれて性能低下が発生することが考えられる．この問題を回避するための手法を提案する．

我々は 1 において同一プロセッサに属する各プロセス間の実行性能を求める方法を提案した．この結果を用いて各プロセッサ内で存在するプロセスについてその他のプロセスとの「親和度」を計算する．前述の性能カウンタによる評価で性能が高いと判定されたプロセス数と低いと判定されたプロセス数との差を「親和度」を表す得点とする．得点が高いものについては同一プロセッサで実行されている他のプロセスと比べ，相対的に競合による性能低下を引き起こしにくいプロセス可能な限り同一物理プロセッサに残す．逆に得点が低いものについては性能低下を引き起こしやすいプロセスと判断して他のプロセスと比較して優先的に別の物理プロセッサに移動させる．

3　提案手法の実装

本稿ではソースコードが公開されていてかつ広く用いられている Linux の kernel2.6 を用いて提案手法の実装を行う．

3.1　Linux カーネルスケジューラの概要

提案手法の実装説明に先立って Linux2.6 のスケジューラについて簡単に説明を行う．Linux スケジューラの構造を図 2 に示す．Linux2.6 では各プロセッサ毎に実行中プロセスのキュー（ランキュー）を持ち，SMT アーキテクチャを持つプロセッサでは各スレッド毎にキューが割り当てられる．キューは active, expired の 2 つの優先度別リストの配列からなり，通常はこの中から最高優先度を持つプロセスを active キューから選択して実行する．CPU 時間を使い果たしたプロセスは優先度を再計算され expired に移される．active キューが空になると active と expired が入れ替わり，再び実行が開始される．このキューが空になるまでの実行単位を epoch と呼ぶ．

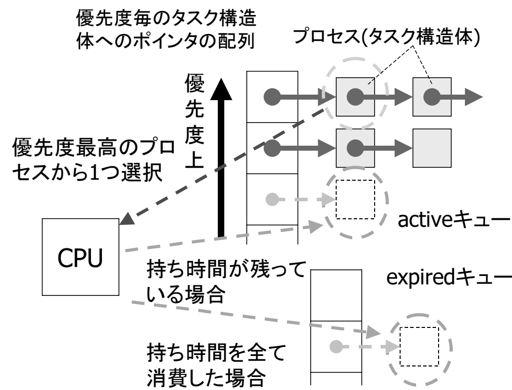

図 2: Linux2.6 のスケジューラの概念図

本稿の提案手法ではこの active キューに属するプロセスについてプロセスの実行性能を改善可能と判断した場合に，active キューに所属するプロセスから，最高優先度とは関係なく次に実行するプロセスを選択する．プロセスの優先度を無視したスケジューリングを行うことになるが，提案手法では同一 epoch の範囲内でのみプロセスの実行順序の入れ替えを行う．プロセス毎の CPU 時間を無視して実行を行う，あるいは epoch を越えた実行順序の入れ替えは行わない．よってプロセスの実行時間全体が単一 epoch より十分長い場合は，epoch の中では必ず優先度によって計算された同一の CPU 時間を割り当てられるため，特定のプロセスが偏って多くの時間プロセッサを占有することがない．

3.2　性能カウンタからの情報取得

本稿の提案を Linux カーネルへ実装するために，各物理プロセッサ毎にプロセッサ上で実行している各プロセスについて同時実行を行った場合の性能測

定結果，実行性能の判定基準となるキャッシュミス回数等の平均値を記録するテーブルをカーネルメモリ空間に用意する．各物理プロセッサ毎に別のテーブルを持つのは管理データへのアクセスによるロック回数を減らすためである．具体的な管理構造の図3に示したような形で各プロセッサ毎に履歴テーブルを持つ形になる．

各プロセスと同時実行した場合の競合の頻度はカウンタから取得した値をそのまま格納するのではなく，平均値を用いて比較した結果をビットデータで格納する．他の各プロセスと同時実行を行った場合の競合の頻度が大きい，小さいことを示すビットマップをそれぞれ用意する．これによって履歴の容量の削減と，履歴情報の走査の高速化を実現する．

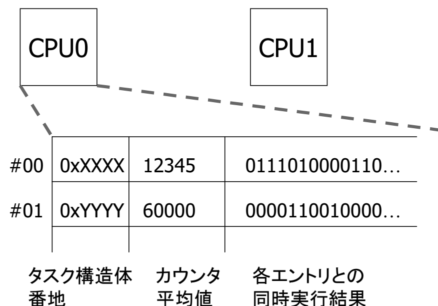

図 3: データ構造のイメージ

存在する全プロセスの組み合わせを扱うことは，プロセス数が増えた際に管理することが困難である [4]．そのため我々は履歴を記録するテーブルのエントリ数を固定サイズとし，プロセス間の資源競合関係はテーブルに記録されたプロセス同士のみについて処理を行い，その他のプロセスについては提案手法の適用をせずにスケジューラでは通常のスケジューリングを行う．

テーブルが有限であるため，プロセスの CPU 使用時間等からシステムに一定の負荷を与えるプロセスのみを検出して記録する．テーブルには履歴データ，キャッシュミス回数の平均値等のデータを記録する．履歴データは各エントリに対応するプロセスと他のプロセスを同時実行した際の結果がビットマップで記録されており，各ビットがそれぞれのプロセスと同時実行した際の結果に対応する．

これらのデータを用いて次に実行するプロセスを選択する手順を述べる．まず，テーブルに登録されているプロセスのうち現在ランキューに登録されたプロセスを表すビットマップを用意する．このデータと実行履歴のビットマップの論理積を取ることにより，実行可能かつキャッシュ競合が少ないと予測されるプロセスを選択することが可能である．

通常のスケジューラが示した次に実行するプロセスと比較してキャッシュミスを低減させることが可能である場合はこの候補の中から任意のプロセスを選択し，代わりに実行を行う．キャッシュミスの低減が不可能と判断すると，通常のスケジューラが選択したプロセスをそのまま実行する．

3.3 プロセッサ間負荷分散

アルゴリズムの章で我々が提案した得点を計算するには履歴データとして格納されているキャッシュミス頻度の大小を表すビットマップを用いる．ビットマップ中でセットされているビットの数を数えることで，同時実行した場合キャッシュミス頻度が大きくなるプロセスの組み合わせ数，小さくなる組み合わせの数を求めることが可能である．その差を計算することで各プロセス毎に得点を求め，プロセッサ間移動の基準とする．

また，Linux ではプロセスのプロセッサ間移動は CPU 間のプロセス数のバランスが崩れた場合に行われるため，本稿ではより競合の少ないプロセス配分を発見するために以下の処理を行うことで意図的にタスクキュー間のプロセス移動を発生させ，キャッシュ競合を減らす．各物理プロセッサで実行されているプロセスを前項で述べた得点の低いものから順番に別の物理プロセッサに所属するプロセスと交換を行う．その状態で短時間プロセスを走行させてプロセスの動作性能を測定する．その後得点を参照し，プロセス交換を行う以前より得点の変化がないか上昇した場合そのまま実行を続ける．一方低下を確認した場合は直ちに両方のプロセスを元のプロセッサに戻す処理を行う．

この操作を定期的に繰り返し行うことによって，競合の発生しやすいプロセス同士が同一プロセッサに同時に固定されることを防止する．

4 性能評価及び結果

我々の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プログラムの実行性能を比較する実験を行う．実験環境には以下の環境を使用する．

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

CPU は内部に 2 つのスレッドを持つ SMT プロセッサである．前述の実装を行った Linux カーネル及び通常の Linux カーネルの間でプログラムを実行し，提案手法について評価を行う．

キャッシュ競合の回避が行われていることを検証するため行列の乗算を行うプログラムを作成し，その実行時間を測定する．プログラムはブロック化された行列を単位としてマルチスレッドで動作し，結果をブロック化された形で格納するものである．プログラム全体のスレッドを 2 つのグループに分割し，同時にブロック化した 2 つの列ずつ同一数のスレッドを配分して計算を行う．この場合，同一列を計算するスレッド同士でキャッシュ競合がより少なくなる．

まず，4900×4900 の行列を 14×14 にブロック化して 2 列並列で乗算を行った場合について提案手法を用いた場合と用いなかった場合について計測を行う．

行列サイズはブロック化した行列が CPU のセカンドキャッシュの容量にほぼ等しくなるように設定している．ブロック化した同一列を同時に計算する場合，各スレッドの計算で頻繁にアクセスを要する

行列が同一のものになるため，ワーキングセットがキャッシュ内に収まりキャッシュ競合が少なくなる．一方，異なる列の計算を同一プロセッサで行った場合，頻繁にアクセスを行うブロック化行列が 2 つになりワーキングセットが同時にキャッシュに収まりにくくなることでキャッシュ競合が増加する．

それぞれ計算スレッド数を 8,12,16 スレッドとした場合について実験を行う．比較として手動で最適なプロセッサにスレッドの割り当てを行った場合についても計測する．

この行列計算を 8 スレッドで 10 回計算した結果は図 4 に示したようになる．また，実験データの分散は図に示したようになる．

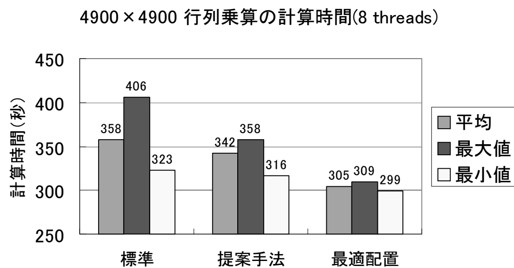

図 4: 行列乗算プログラムによるテスト結果

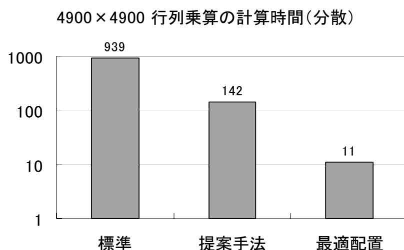

図 5: 行列計算プログラムによるテスト結果（分散）

標準のスケジューラでは最大と最小の実行時間差が提案手法よりも大きく開き，実行時間のばらつきが大きいことが分散のグラフから分かる．一方提案手法では標準のカーネルに比べ，処理時間の分散が削減されていることが分かる．実行結果の平均値，最大値の結果より最大処理時間は約 11%低減され，処理時間の増大する頻度が減少していることが分かる．また提案手法を用いた場合平均処理時間が 4%程度削減されており実行時間のばらつきを抑えるだけでなく，実行時間の高速化を達成できていることが分かる．これはキャッシュ競合の起こるスレッド同士の同時実行が提案手法を用いたスケジューラによって回避されているためであると言える．

我々の提案手法の利点として，上記の性能改善を達成するためにオペレーティングシステム以外のハードウェア，ソフトウェア共に改変する必要がなく，実行するプロセスについて前もって情報することが不要であることが挙げられる．また，使用資源が比較的少なくて済む（本稿の実験環境ではメモリを 16KBytes 使用している）という利点が挙げられる．

同じ処理をするプログラムにおいてスレッド数を 12, 16 に増加させて 10 回ずつ計測した実験の結果は図 6,7 に示したようになる．

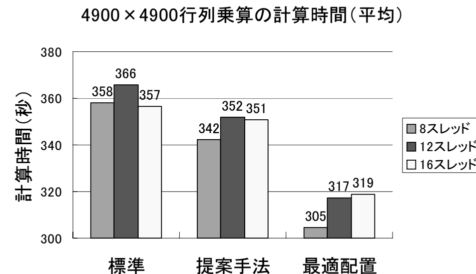

図 6: 行列乗算プログラムによるテスト（平均）

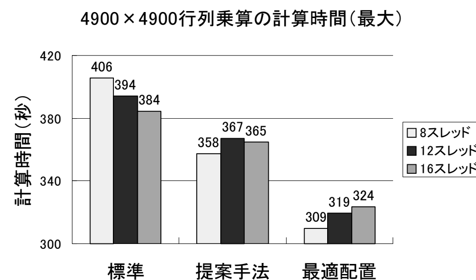

図 7: 行列乗算プログラムによるテスト（最大）

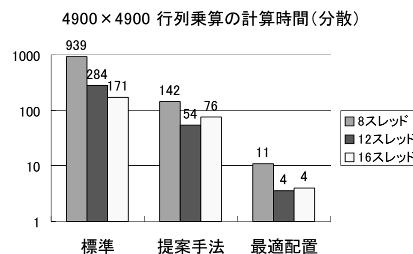

図 8: 行列乗算プログラムによるテスト（分散）

スレッド数が増加した場合においても，平均処理時間，最大処理時間の短縮幅が小さくなっているものの，高速化を達成していることが分かる．

スレッド数が増加した際に短縮幅が小さくなっている理由としては標準のスケジューラでテストした場合，図 8 に示したように分散が小さくなっていることから，標準のスケジューラでの実行時間のばらつきが低減し，相対的に提案手法の効果が小さくなっていることが原因であると言える．

5　関連研究

本稿と関連した研究について述べる．

Snavely ら [2] はマルチスレッドアーキテクチャにおける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，それを基準として同時にスレッドを動かすことが有益であるかの判断を行うことを提案している．そして 2 種類のベンチマークプログラムを同時に実行した場合の SMT プロセッサでの有益性の評価を実際に行っている．また [3] では，SMT プロセッサを持つシステムにおけるプロセッサで同時実行するプロセスの組み合わせについてハードウェア資源を考慮したスケジューリングを行うことの重要性を指摘している．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，プロセスの実行状況から同時に実行する最適なスレッドの決定を行う．その情報に従ってスケジューリングを行う手法を提案している．これらの研究では，性能カウンタによるプロセス実行状況のサンプリングと，最適化による同時実行プロセスの選択という段階に処理を分けている点が本稿での提案手法とは異なる．また，同時に実行するプロセスの最適化を行う際に全てのプロセスの組み合わせを想定している点が本稿での提案手法と異なる．さらに，シミュレータでの検証の際，同時に複数のスレッドでプロセスの切り替えが生じる仮定を行っている点が本稿の提案手法とは異なる．

Zaki ら [4] は同様に数プロセスでプロセッサ資源を共有することになる SMT プロセッサからなるシステムでは従来通りのプロセススケジューリング方式では性能低下か発生することを指摘している．そして性能カウンタで各資源の競合を回避するようにスケジューリングを行った場合の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通りのスケジューリング方式では性能低下が発生することを指摘している．そしてハードウェア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各プロセスの資源利用を考慮することの重要性を示している．

これらの研究と本稿での提案手法との違いとして挙げられる点は，本稿での我々の提案では演算資源のうちキャッシュミスのみに注目している点，具体的な履歴データの格納について提案し実装を行っている点，SMT プロセッサが複数存在する場合のプロセッサ間の負荷分散に履歴データを活用する点である．

6　まとめ

本稿では SMT アーキテクチャにおけ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た．それに対する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ハードウェア統計情報による履歴情報を用いたスケジューリングを行うことを提案した．またプロセスを同時実行した場合の性能情報履歴を用いて，プロセッサ間負荷分散に応用する手法を提案した．そしてこの提案手法を Linux2.6 上で実装し，プログラムの実行性能を評価した．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最大で約 4%削減され，互いにキャッシュ競合を引き起こすスレッドを同時に実行する際にプロセススケジューリングの最適化操作により実行速度の高速化を実現できることが示された．また，最大実行時間が最大で約 11%削減され，処理時間のばらつきを処理時間を増大させることなく抑えるために本稿の提案手法が有効であることが示された．

参考文献

[1] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, Teiji Nishizawa: An elementary processor architecture with simultaneous instruction issuing from multiple threads, Proceedings of the 19th annual International Symposium on Computer Architecture, pp.136-145, 1992

[2] Allan Snavely, Nick Mitchell, Larry Carter, Jeanne Ferrante, Dean Tullsen: Explorations in Symbiosis on two Multithreaded Architectures, Workshop on Multi-Threaded Execution, Architecture, and Compiler, 1999

[3] Allan Snavely, Dean M. Tullsen: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.234-244, 2000

[4] Omer Zaki, Matthew McCormick, Jonathan Ledlie: Adaptively Scheduling Processes on a Simultaneous Multithreading Processor, Technical report, University of Wisconsin - Madison, 2000

[5] Sujay Parekh, Susan Eggers, 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] James M. Barton, Nawaf Bitar, A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX, Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, pp.45-69, 1995

[8] Raj Vaswani, John Zahorjan: The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors, Proceedings of the 13th ACM symposium on Operatiog systems principles, pp. 26-40, 1991

[9] Josep Torrellas, Andrew Tucker, Anoop Gupta: Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors, Journal of Parallel and Distributed Computing, 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