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

2008－OS－108 (11)
2008／4／23

リソース競合を考慮したチップマルチプロセッサ向けプロセススケジューリング

佐々木 広[†] 小藤 健太郎[†]
近藤 正章[†] 中村 宏[†]

近年，複数のプロセッサコアを１チップに搭載するチップマルチプロセッサ（CMP）が汎用マイクロプロセッサにおける主流となりつつある．CMP では共有リソース上で競合が発生するとチップ全体のトータルのスループットが低下する，あるいは各プロセスの性能低下の公平さ（Fairness）が保たれないなどの問題が生じる．本稿では，CMP において各プロセスの実行のスピードを調整することで，リソース競合の影響を柔軟に制御し，効率的なプログラム実行環境を提供することを目的としたプロセススケジューラを提案し，Linux への実装を行った．

Optimizing Multiprogramming Performance on CMPs via an OS Scheduler

Hiroshi Sasaki ,[†] Kentaro Kofuji ,[†] Masaaki Kondo [†]
and Hiroshi Nakamura [†]

Recently, chip multiprocessors (CMPs) are becoming an attractive architecture due to its high throughput with low power consumption. In CMPs, performance degrades significantly if resource contention occurs. In this paper, we propose a process scheduler which controls the execution speed of each threads running on CMPs so as to optimize the shared resource utilization. We implemeneted the proposed scheduler on Linux and evaluated the effectiveness of it.

1. はじめに

近年，プロセッサの構成として，複数のプロセッサコアを１チップに搭載するチップマルチプロセッサ（chip multiprocessor: CMP）が主流となってきている．CMP は，シングルタスクの並列処理，あるいは複数タスクの並行処理を行うことでクロック周波数の向上に頼ることなく高性能（高スループット）化を達成できるため，消費電力あたりの性能に優れるアーキテクチャとして期待されている．一般的に CMP ではリソース有効活用の観点から，複数のプロセッサコアが，あるメモリ階層以下にあるキャッシュやバス，主記憶である DRAM のバンクなどを共有することが多い．しかし，これらの共有リソースに対して複数のプロセッサコアが同時にアクセスし競合が発生すると，性能が低下することがある．特に，プロセッサと主記憶の性能差が拡大している近年においては，メモリ階層において競合が発生すると性能への影響が非常に大きく，深刻な問題となっている．

上述したようなリソース競合の発生により，(1) トータルスループットの低下，(2) *Fairness*（各スレッドの競合による性能低下の公平さ）の低下，(3) 性能予測性の悪化，といった問題が生じる．これまでにも，リソース競合の影響の緩和を目的として，いくつかの手法が提案されている．例えば，キャッシュ上での競合を防ぐために，キャッシュを論理的なパーティションに分割する手法[1)~4)]，主記憶 SDRAM のバンク上での競合に対処するためのメモリアクセススケジューリング手法[5)]，メモリバス上での競合の影響の緩和を目的とした動的電源電圧・周波数制御手法[6)] などがある．従来手法により，ある程度リソース競合の影響を緩和させることができると思われるが，今後より多くのプロセッサコアが１チップに集積されると競合による影響はさらに深刻になるため，今まで以上に柔軟かつ効果的にリソース競合の影響を制御できる手法が必要になると考えられる．

そこで本稿では，CMP においてリソース競合の影響を柔軟に制御し，効率的なプログラム実行環境を提供することを目的とした，オペレーティングシステム（OS）によるプロセススケジューリング手法を提案する．提案手法は各コアで実行される各プロセスの実行スピードを調整することによって，競合の影響を制御するものである．実行スピードの調整は，各プロセスにおける競合による性能への影響をあらかじめ統計的にモデリングしておくことで予測しつつ，最適化すべき目的に応じて設定されたポリシーに従って行われる．提案手法による実行スピード制御により，ハードウェアの追加や拡張をすることなく競合の影響を調節できるため，従来の手法と比較して柔軟に種々の最適化が可能となる．本稿では，提案手法の詳細および

† 東京大学 先端科学技術研究センター
Research Center for Advanced Science and Technology, The University of Tokyo

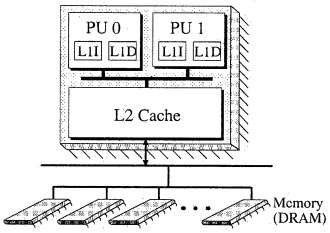

図 1　CMP の概要

Linux システムへの実装について述べ，最適化の例として Fairness の改善，およびトータルスループットの向上に着目し，CMP を搭載した実プラットフォーム上で評価を行い，その有効性を明らかにする．

2. CMP におけるリソース共有の影響

CMP では，図 1 に示すように，複数のプロセッサコアがメモリバスや主記憶を共有するのが一般的である．また，図のようにチップ内の L2 キャッシュを共有する場合も多い．このように，複数の PU でリソースを共有する場合，各コア上で動作するプロセスの性能は，共有リソース上での競合の状況に大きく依存する．

リソース競合が性能に与える影響を調べるため，*Intel Core2 Extream QX6700*（以下 Core2-QX6700）を搭載した実機のマシンにより評価を行なった．Core2-QX6700 は，4 コアを持つ CMP であるが，実際には 4MB の L2 キャッシュを共有するデュアルコアプロセッサを 1 パッケージ内に 2 個搭載したものであり，今回は 2 プロセスを同時に実行した場合について評価を行なった．なお，どの 2 コア上でプロセスを実行するするかにより，L2 キャッシュを共有する場合と非共有の場合の両者を評価することが可能であるが，今回はキャッシュ以外の共有リソース上での競合も影響が大きいことを示すために，L2 キャッシュを共有しない場合について評価を行なった．したがって，メモリバス（FSB）以下のメモリ階層が共有リソースとなる．

図 2 に，SPEC2000 ベンチマーク，および Olden ベンチマークのいくつかのプログラムの組み合わせにおける，2 プロセスを同時に実行した場合の Multiprogrammed Speedup を示す．Multiprogrammed Speedup は，CMP における評価指標の一つであり，それぞれのプロセスを単独で実行した場合に対して，複数プロセスで同時に実行した際の各プロセスの性能比を，全プロセスで合計したものである．図は，対象プログラム（横軸に示すプログラム）と，全プログラムとの組み合わせにおける Multiprogrammed Speedup を，“箱ひげグラフ（box-and-whisker plot）”で表わしており，対象プログラム毎に，箱部分が *IQR (Inter-Quartile Range)* と呼ばれる中央 50%範囲内のデータを，またひげ部分の線が，箱部分の上下それぞれから $1.5 \times IRQ$ 範囲内のデータを示している．その範囲外

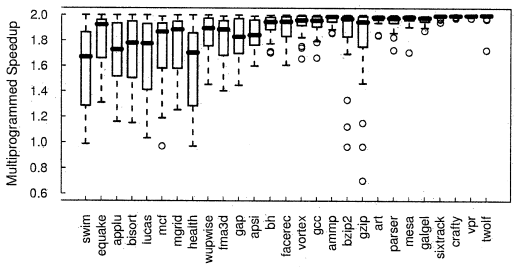

図 2　Core2-QX6700 における Multiprogrammed Speedup

のデータは，はずれ値としてそのデータポイントがそれぞれプロットされている．また，箱内部の横線は中央値を示している．なお，対象プログラムは，左から単独で実行した場合の L2 キャッシュミス率の高い順に並んでいる．

図 2 より，対象プログラムの L2 キャッシュミス率が高い場合には，Multiprogrammed Speedup が大きく低下してしまうことがわかる．本評価では，独立に実行可能な 2 つのプログラムを実行しているため，この性能低下はリソース競合によるものである．また，この評価では L2 キャッシュを共有していない 2 コア上で対象プログラムを実行しているため，これらの性能低下はメモリバス，およびメモリバンクの競合により生じたものである．したがって，キャッシュを共有していない場合でも競合の影響は深刻であり，キャッシュを共有する場合には，キャッシュ上での競合も発生することから性能への影響はさらに大きくなる．

このように，CMP ではリソース競合の影響で，実行しているプロセスの性質に依存して性能が大きく低下してしまう場合がある．次節では，このリソース競合の影響を緩和させるための提案手法について述べる．

3. リソース競合を考慮したプロセススケジューリング

本節では提案手法である，CMP においてリソース競合の影響を制御し，効率的なプログラムの実行を目指すプロセススケジューリング手法について述べる。

3.1 概　　要

あるコア上で実行しているプロセスの L2 キャッシュミス率が相対的に高く，共有リソースへのアクセス率が高い場合，当該プロセスの性能低下に比べ，他のコア上で実行されているプロセスの性能が競合により大きく低下する可能性がある．この場合，もともと高い命令スループットを達成できるプロセスの性能低下が大きいと，トータルスループットが大きく低下してしまったり，Fairness が保たれないといった問題が生じる．

提案手法は，各コアで動作するプロセスの実行スピードを制御し，共有リソースのアクセス率を調整す

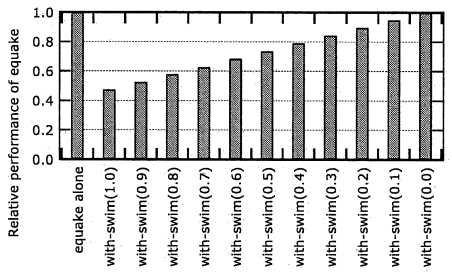

図 3　swim の実行スピードを変化させた場合の equake の性能

ることで競合の発生を制御し，リソース競合における種々の問題の解決を狙うものである．共有リソースアクセス率の高いプロセスの実行スピードを遅くすれば競合の発生が抑制され，他のプロセスの性能低下を改善することができる．

実行スピードの調整による性能への影響を調べるため，前節と同じく Core2-QX6700 を用い，SPEC2000 の **equake** と **swim** を同時に実行させた際の equake の性能について，swim の実行スピードを変化させつつ評価を行なった．図 3 に，前節と同じく L2 キャッシュ非共有の場合の結果を示す．図中 with-swim(X) は，フルスピードに対する swim の実行スピードの割合を X とした場合を意味している．また，equake-alone は equake のみを 1 つのコアで実行した場合である．なお，実行スピードの制御手法については，3.2 節で述べる．

図 3 より，swim がフルスピードで動作した with-swim(1.0) の場合には，equake の性能は単独実行時に比べて半分程度まで低下してしまうが，swim の実行スピードを抑えるにしたがって，単位時間あたりの swim による共有リソースアクセスが減少し，equake が共有リソースへ円滑にアクセスできるようになるため，equake の性能が向上していくのがわかる．このことより，TCE により実行スピードを調整することで競合の影響を制御でき，ひいては各コアで実行されているプロセスの性能を制御可能であると考えられる．

3.2　実行スピードの制御手法

プログラム実行のスピードを制御する方法はいくつか存在する．最も一般的で効率的な手法は，dynamic voltage and frequency scaling (DVFS) 手法により，周波数を制御するものである．DVFS により時間的に細粒度にスピード制御を行なうことができ，また周波数と同時に電源電圧も下げることで，消費電力および消費エネルギーが削減できるという利点もある．ただし，近年では多くのプロセッサが DVFS の機能を有してはいるが，利用可能な周波数が限られていたり（例えば Core2-QX6700 では 2.4GHz，2.0GHz，1.6GHz の 3 通り），コア毎に独立に周波数を変更できないなど，制約も多い．

また，他の方法として Clock Modulation と呼ばれるクロックを供給する時間を制限する方法も考えられる．これは，Intel Pentium 4 ベースのプロセッサにおいて，*Thermal Throttling* として実装されている．Thermal Throttling は，クロック供給時間を制限することで消費電力を低下させ，チップ温度を下げるための機構である．この場合，理論的には任意の実行スピードを選択できるという利点がある一方，利用可能なプロセッサが多くないという問題がある．

より柔軟に実行スピードを制御するための方法として，OS のプロセススケジューリングに基づく手法が考えられる．通常，OS は各プロセスに対して，優先度に基づいて計算処理のためのタイムスライスを割り当てるが，この割り当てるべきタイムスライスの量（duty-cycle と呼ぶ）を制御することで，実行スピードを仮想的に制御することが可能となる．OS により管理できるタイムスライスは，割り込みなどのオーバヘッドがあるため，短くても数百マイクロ秒程度と時間的には粒度が荒く，実行スピード制限を行なった場合に効率が悪くなる可能性があるが，ハードウェアによる制約なしに柔軟に実行スピード制御を行なえるという利点を持つ．本稿では，この OS のプロセススケジューリングに基づく手法を用いる．

3.3　統計的モデリングに基づく性能予測

提案手法により各種最適化を行なう際には，競合により各プロセスがどれだけ性能低下しているかを見積もる必要がある．通常，同時に実行されるプロセスあるいはプロセス中のフェーズは実行時でないと分からないため，性能低下の見積もりは実行時に動的に行う必要がある．ここで，プラットフォーム毎に共有リソースの構成は異なり，直接的にリソース競合の影響をモニタする機構を仮定することは難しいため，本稿では競合による性能低下を論文[7] を基にした統計的学習手法により予測することにする．この予測手法はオフラインでの統計的学習と，学習結果を用いたオンラインでの適用からなる．

3.3.1　オフラインでの統計的学習

まず，あらかじめ対象の CMP のマシン上で，SPEC などのベンチマークを用い，様々なプログラムに対して，単一プロセス実行時，および複数プロセス実行時の両方についてプロファイリングのための実行を行ない，定期的に性能と全パフォーマンスカウンタの値を取得する．これにより，プロセス毎の各実行フェーズにおける，単一プロセスで実行した場合に対する複数プロセス実行時の性能低下率を求めることができる．

この性能低下率は，いくつかのパフォーマンスカウンタの値と強い相関があり，複数プロセス実行時に取得したカウンタ値から，性能低下率を推定できると考えられる．この，性能低下率とカウンタの値の関係を明らかにするため，本稿では重回帰分析を用いる．回

帰分析を行う際は，通常単純な線形モデルを用いることが多いが，競合による各プロセスの性能低下は同時に実行している他のプロセスの状況にも依存するため，自身のパフォーマンスカウンタだけでなく，同時に実行している他のプロセスのスカウンタからの作用も考慮する必要がある．そこで本稿では，式（1）に示す，交互作用付きの線形回帰モデルを用いる．

$$y = \beta_0 + \sum_{i=1}^{n-1} \beta_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \beta_{i,j} x_i x_j + e \quad (1)$$

ここで，y は従属変数，x_i と x_j は独立変数，β_0 が定数項，β_i や $\beta_{i,j}$ はそれぞれの独立変数にかける重み(偏回帰係数)，e は誤差項を意味する．

この交互作用付き線形回帰モデルに対し，従属変数として予測対象プロセスの性能低下率を，説明変数 x_i として予測対象プロセスのパフォーマンスカウンタ値を，また x_j として他のプロセスのカウンタの値を当てはめることで回帰分析を行い，性能低下予測式を導出する．この際，全カウンタの組み合わせで回帰分析を行い，最も高い寄与率を示すカウンタの組を性能予測に用いるカウンタとする．これにより，プロセス毎に性能低下率，すなわち競合による性能への影響を予測することが可能となる．なお，この予測モデルはプラットフォーム毎に別々に構築する必要がある．この統計的学習に関するさらなる詳細については，文献[8]で述べられている．

3.3.2 オンラインでの適用

統計的学習結果で得られた予測式に基づき，実行時にパフォーマンスカウンタの値を参照しつつ動的に性能低下率を予測する．具体的には，あるインターバル毎にカウンタの値を取得し，それを予測式に当てはめることで，各プロセス毎にその時点の性能低下率を求める．なお，学習の際に様々な種類のアプリケーションを用いて回帰分析を行っておけば，広範囲のアプリケーションに適応できるモデルを構築できるため，未知のアプリケーションが実行された場合でも性能低下率を予測することが可能であると考えられる．

この統計的学習に基づく手法のみでもある程度の予測精度は得られるが[8]，より高い予測精度を得るべく本稿ではオンラインで実際の性能低下率を測定しつつ，それを学習結果にフィードバックして微調整をする手法を提案する．

上記で述べたインターバルのうち，何回かに一回（例えば 50 回に一回程度），各プロセスを単独で実行させることで，単独実行時の性能を求める．その性能と，それまで複数プロセスで実行していた数回のインターバル中の平均性能を比較し，性能低下率のサンプルを取得する．さらに，同じ数回のインターバル中の平均平均予測性能低下率とを比較することで，サンプルの性能低下率と予測性能低下率の比，すなわち予測に対する補正値を求める．次のインターバルより，予測された性能低下率に対し補正値を掛けることで，予測値を補正する．これを繰り返すことで，フィードバックにより予測の微調整が行なわれ，予測制度が向上すると考えられる．

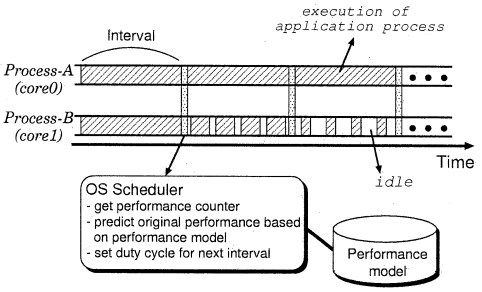

図 4　提案手法による実行時最適化の概要

3.4 提案手法による実行時最適化

図 4 に提案手法による実行時最適化の概要を示す．図はプロセス A とプロセス B が 2 つのコア（core0 と core1）上で実行される様子を示したものである．インターバル毎に，OS のプロセススケジューラが，その時点でのパフォーマンスカウンタの情報を基に，前節で述べた手法により競合による性能低下率の予測を行なう．次に，最適化すべき目的に応じて設定されたポリシーに基づいて性能低下率を調整するべく，Runtime-Controller は次のインターバルのための各プロセスの duty-cycle（$0 \leq$ duty-cycle ≤ 1）を決定する．そして，次のインターバル経過後にも同様に性能低下率の予測，およびポリシーに基づいた duty-cycle の制御を行なう．これを繰り返すことで，最終的に最適化すべき目的が達成される．

3.4.1 Fairness 向上への応用

ここでは，Fairness を向上させるための duty-cycle 制御方法を説明する．Algorithm 1 に Fairness 向上のための duty-cycle 調整のポリシーを示す．まず，Runtime-Controller は GetPerfRatio() により実行中のプロセスのパフォーマンスカウンタの値を取得し，性能低下率を予測する．ここで，前のインターバル中で実行スピードが調整されている場合には，各プロセスで実行に割り当てられた時間，すなわちタイムスライスが異なるため，直前のインターバルでの各プロセスの duty-cycle($Duty_i$) を考慮してカウンタ値を調整してから，性能低下を予測する必要がある．

次に，全プロセスでの PR の平均を求め，各プロセスの PR と平均との差を $Diff_i$（i はプロセスの ID を意味する）に毎インターバル足していく．ここで，$Diff_i$ が閾値 TH 以上大きくなった場合は，プロセス i の性能低下率は他のプロセスよりも小さく，他のプロセスに対して悪影響を及ぼしていることになるため，

Algorithm 1 Fairness Policy

NPU: Number of PUs
TH: Threashold for Speed Control
$Duty_i$: Duty cycle for PU_i
$Diff_i$: Perf. disparity for PU_i

for each end of interval **do**
  for $i = 0$ to $NPU - 1$ **do**
    $PR_i = GetPerfRatio(i, Duty_0 .. Duty_{NPU-1})$
  end for

  $PR_{sum} = 0$
  for $i = 0$ to $NPU - 1$ **do**
    PR_{sum} += PR_i
  end for
  $PR_{avg} = PR_{sum}/NPU$

  for $i = 0$ to $NPU - 1$ **do**
    $Diff_i$ += $PR_i - PR_{avg}$
    if $Diff_i > TH$ **then**
      $Duty_i \Leftarrow LOW\_DUTY$
    else
      $Duty_i \Leftarrow 1.0$
    end if
  end for
end for

Algorithm 2 Throughput Policy

NPU: Number of PUs
PR_i: Performance ratio for PU_i
$Duty_i$: Duty cycle for PU_i
IPC_i: Current IPC for PU_i

for each end of interval **do**
  for $i = 0$ to $NPU - 1$ **do**
    $PR_i = GetPerfRatio(i, Duty_0 .. Duty_{NPU-1})$
  end for

  for all j in possible Duty combination **do**
    $P\_IPC_j = 0$
    for $i = 0$ to $NPU - 1$ **do**
      P_IPC_j += $PredictIPC(i, PR, IPC, Dvec_j)$
    end for
  end for
  $p = SelectMAXIPC(Pred\_IPC)$
  $SetDuty(Dvec_j)$
end for

当該プロセスの duty-cycle をあらかじめ設定した低い値（例えば 0.5）に設定する．それ以外のプロセスは duty-cycle を 1 に設定しフルスピードで動作させる．このアルゴリズムにより，性能低下率を均等化するようにスピードの調整が行なわれるため，Fairness を向上することができると考えられる．

3.4.2 トータルスループット向上への応用

トータルスループットを向上させるためのポリシーを Algorithm 2 に示す．まず，Fairness 向上ポリシーと同様に性能低下率の予測を行なう．次に，種々の duty-cycle の組み合わせを想定し，それらの duty-cycle で実行した場合に，各プロセスの IPC がどうなるかを PredictIPC() により予測する．これは，現在の PR，IPC と仮定する duty-cycle の組み合わせにより計算される．全プロセスの IPC の合計が一番高くなる場合の duty-cycle の組を選択し，次のインターバルの duty-cycle として SetDuty() により設定する．

3.5 実　　装

以上で述べた設計に基づき，Linux Kernel 2.6.18 をベースとし，スケジューラに機能追加を行った．ターゲットとなるアーキテクチャは，本稿で評価に用いたプロセッサが Core2-QX6700 であるため x86 とした．既存のカーネルソースに対するコード変更量は，OS に対するタイマ割り込み毎に呼ばれる/kernel/sched.c 内の scheduler_tick() 関数に追加した 10 行程度と非常に少なく，その他の追加部分は新規ファイルにまとめてある．提案手法を実装したスケジューラに関するソースコードの総行数は 1800 行程度である．また，本スケジューラにおける追加部分によるオーバーヘッドは 0.1%以下と十分小さい．なお，本章で述べてきたインターバルはカーネルに対する OS のグローバルタイマ割り込みの間隔としており，その都度実行しているプロセスの性能予測を行い，実行速度を制御する．

また，Linux カーネルではアーキテクチャ依存部は/arch ディレクトリ以下に格納されており，異なるアーキテクチャへの移植が容易な構造となっている．本スケジューラにおいて，パフォーマンスカウンタの制御部がアーキテクチャ依存であるため，これらのコードは/arch 以下に分離し，移植が容易な構造となっている．

なお，本稿では Core2-QX6700 を実装のターゲットとしているが，近年ではほとんど全てのプロセッサにおいてパフォーマンスカウンタが搭載されており，これらを利用することによって他のプロセッサにおいても同様に提案手法を用いたスケジューラを開発することが可能である．また，プロセッサのパフォーマンスカウンタだけでなく，I/O に関する情報や，OS から得られる sleep_avg などの情報を用いることによってさらに精度の高い予測を行うことも可能となると考えられるが，これらについては今後の課題である．

また，以下に評価で用いた，提案手法のアルゴリズムに関するパラメータの値を示す．

- インターバル：4 ms（OS のタイマ割り込み）
- TH: 0.05
- LOW_DUTY: 0.5
- フィードバックインターバル: インターバル × 50 回

4. 評　　価

4.1 評価環境

提案手法の効果を調べるために，本稿ではCMPを搭載した実機のマシンとして，2章でも用いたCore2-QX6700を搭載したPCを用いて評価を行なう．評価に用いたマシンの仕様を表1に示す．なお，Core2-QX6700はどのコアでプロセスを実行するかにより種々の状況で評価可能であり，本稿では以下に示す2つの環境で評価を行なう．

- Core2-SHR:
  L2キャッシュを共有する2コアを用いて評価
- Core2-DED:
  L2キャッシュを共有しない2コアを用いて評価

表1　評価に用いたマシンの仕様

CPU	Intel Core2 Extreme QX6700
- CPU 周波数	2.66 GHz
- L2 キャッシュ	8-Way,4MB×2
- メモリバス	FSB 1066 MHz
M/B	Intel D975XBX2
主記憶	DDR2-SDRAM
- 周波数	667 MHz (PC667)
- サイズ	1GB

実行するプロセスは，SPEC2000ベンチマークよりL2キャッシュミス率の高い，すなわち共有リソースアクセス率の高い`swim`，`applu`，`lucas`，`art`を選択し，その全組み合わせを実行した．プログラムによって実行時間が異なるため，片方のプロセスが先に終り，1プロセスのみが実行されてしまうことがある．そこで本評価では，プログラムの実行が終了し次第，すぐに同じプログラムを同一コアで実行し，各プロセスで少なくとも3回の実行が終了するまでを評価対象とする．

また学習には，SPEC2000ベンチマーク中，上記で示した評価に用いるプログラム以外の全プログラムを用いた．これは，未知のプログラムが実行された場合でも前章で述べた統計的学習手法で競合による性能低下が予測可能であり，提案手法による最適化が有効に働くことを示すためである．

5. 評価結果

5.1 統計的学習の結果

表2に，各評価プラットフォームでの学習結果により，もっとも寄与率が高い組み合わせとして選択されたカウンタと，その時の寄与率を示している．なお，アルゴリズム中や評価において，実行命令数を知る必要があるため，実行命令数取得のためのカウンタである*Inst. completed*は必ず含まれるようにして学習を行なった結果である．また，*L1D accesses*はL1データキャッシュへのアクセス回数を意味する．

表より，両プラットフォームにおいて，L1データキャッシュアクセスが選択されていることがわかる．直感的にはL2キャッシュミス回数が適切と思われるが，例えばL2キャッシュミス回数が少ない場合に，競合により自コアが性能低下した結果，プログラムの進行が遅くなり少なくなっているのか，それともL2キャッシュミスが元々少なかったのかの判断は難しく，その結果L2キャッシュミス回数が性能低下率の予測に適切なカウンタにはならなかったのが原因だと考えられる．一方，L1データキャッシュアクセスはロード命令の頻度を表し，このカウンタを用いた方が，多くの場合において共有リソースへのアクセス率を直接的に把握することができ，より妥当なカウンタとして寄与率が高くなったものと考えられる．

表2　性能低下率予測のために選択されたカウンタ

Platform	選択されたカウンタ	寄与率
Core2-SHR	Inst. completed, L1D accesses	0.523
Core2-DED	Inst. completed, L1D accesses	0.748

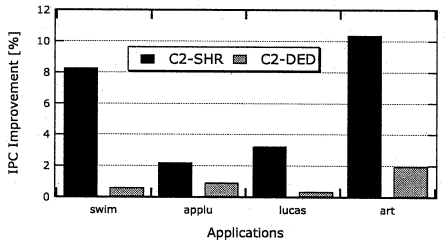

図6　トータルスループットの向上率

5.2 Fairness向上ポリシーの評価結果

図5に，提案手法を用いない場合（ORG），および提案手法を用いた場合（Pro）における，単独で実行した場合に対する，複数プロセスで実行した場合の各プロセスの性能低下率の差を示す．したがって，この差が小さいほどFairnessは良いことになる．なお，各プログラム毎に他のプログラムとを同時に2つ実行する場合を全通り評価し，その平均の値を示している．

図5より，評価したプログラムのほとんどの場合でORGに比べ提案手法によりFairnessを改善できていることがわかる．特に，もともとのFairnessが悪い`swim`や`mgrid`との組み合わせの場合では，Fairness改善の効果が大きい．これは，頻繁に共有リソースへアクセスするプロセスが，他方のプロセスの共有リソースアクセスを阻害することで性能低下率に不均衡が生じた場合に，TCEにより前者のプロセスの実行スピードを抑え，後者のプロセスが共有リソースへ円滑にアクセスできるように制御することで，性能低下率がバランスした結果である．

一方，ORGにおいて性能低下率の差が小さいものは提案手法の効果が小さい．これは，もともとORGでも良いFairnessが達成されている場合には，提案

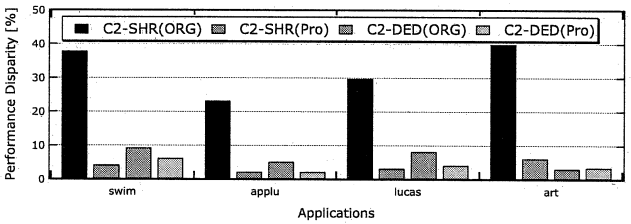

図 5 各プロセスの性能低下率の差

手法による実行スピード制御の余地がほとんどなかったためである．

これらより，提案手法を Fairness 向上に応用することで，実際に Fainress 向上の効果があることがわかった．また，実行スピード制御は統計的モデリングを用いたリソース競合による性能低下率の予測を基に行なったものであるが，期待された通りに Fairness が向上しているプログラムが多いことから，予測の精度は非常に高いと考えられる．したがって，他の最適化に対して提案手法を適用した場合にも，効果的な実行スピード制御ができると思われる．

5.3 トータルスループット向上ポリシーの評価結果

図 6 に，ORG に対する提案手法のトータルスループットの向上率を示す．前節同様，各プログラム毎に他の全プロセスと同時に実行した場合の平均の値を示している．

図より，いくつかのプログラムの組み合わせで，実際にトータルスループットが改善されていることがわかる．特に，C2-SHR の場合で大きく向上しているプログラムが存在する．これは，L2 キャッシュを共有しているためリソース競合による性能低下の影響が ORG では大きく，TCE により性能向上の余地が大きかったためである．一方，もともと競合の影響が少ない場合，提案手法により若干性能が低下してしまっている．これは，スピード制御によるオーバーヘッドのためである．しかし，低下率はとても小さく，提案手法の有効性は高いと考えられる．

以上の結果より，提案手法は実際にトータルスループット向上にも効果があることがわかり，提案手法は CMP において効率的な実行を提供する手段として非常に有効であると結論付けることができる．

6. 関連研究

従来より，共有キャッシュのリソース競合を削減するためにキャッシュを分割し，性能や Fairness を向上させる手法が提案されている．文献[1),2)] では，共有 L2 キャッシュを分割することでキャッシュミスの削減，あるいは Fairness の向上を狙う手法が提案されている．文献[3)] では動的にキャッシュを分割し，utility に基づいて各アプリケーションに領域を割り当てる手法を提案している．また，文献[4)] では，キャッシュを分割する際の種々のポリシーについて検討が行なわれている．共用キャッシュを OS から制御するための拡張について文献[10)] で提案されている．さらに，文献[11)] において，キャッシュ競合の影響を予測するモデルが提案されている．また，文献[12)] では，QoS を導入した共有キャッシュの制御手法について述べられている．

文献[13)] は，キャッシュ競合の影響を考慮した OS によるプロセススケジューリング手法を提案している．これは，コア数以上のアクティブに動作するプロセスに対し，各プロセスのタイムスライス量を性能低下率にあわせて変化させ，Fairness を向上させるものである．この手法は，コア数以上のプロセスが動作している場合しかタイムスライス量の調整ができないため，アクティブに動作しているプロセス数がコア数以下である場合には効果がない．一方提案手法では，プロセスの実行スピードを仮想的に制御することでそのような場合に対応でき，この点で文献[13)] とは異なるものである．

文献[5)] では，CMP の主記憶 DRAM アクセスのスケジューリングにおいて，トータルスループットを向上させるための QoS 制御手法が提案されている．また，文献[6)] では，メモリバス上での競合による Fairness の悪化を緩和させるための動的電源電圧・周波数制御手法が提案されている．

メモリ階層だけでなく，演算器などのリソースも共有する simultaneous multi-threading (SMT) プロセッサでは，リソースの使用率を考慮した最適化の研究が活発に行なわれている[14),15)]．また，メモリアクセスなど長いレーテンシのストールが発生した際に，異なるスレッドに切り変えて実行することでスループット向上を狙う switch on event (SOE) 型のマルチスレッドプロセッサにおける Fairness 向上手法も提案されている[16)]．また，SMT 上での共有リソース競合のモデリング手法が文献[17)] で述べられている．

本稿で提案するプロセススケジューリング手法は，CMP において各プロセスの実行スピードを制御することで，リソース競合の影響を柔軟に制御し，効率的なプログラム実行環境を提供するものであり，この点が従来の手法と比べての大きな違いである．現在でも，共有リソースを持つ SMP (symmetric multi proces-

sor）やCMPに対応したOSカーネルはいくつか存在するが，提案手法のように広範囲の共有リソース競合の影響を制御できるようなプロセススケジューラは存在せず，この点で提案手法の新規性を強調することができる．さらには，統計的な学習により競合による性能低下を動的に予測することも，本稿の大きな貢献である．

7. まとめと今後の課題

本稿では，CMPにおいて効率的なプログラム実行環境を提供することを目的とした、プロセススケジューリング手法の提案および設計・Linuxへの実装を行った。提案手法は，各プロセスでの競合による性能への影響をオフラインで統計的学習に基づく手法により予測することで，最適化すべき目的に応じて設定されたポリシーに従って各プロセスの実行のスピードをオンラインで調整し，リソース競合の影響を柔軟に制御するものである．実機のCMPマシンに対し，提案手法をFairness向上，およびトータルスループット向上に応用した結果，Fairnessを大きく改善でき，またいくつかのプログラムの組み合わせで実際にトータルスループットが向上することがわかった．

今後の課題としては，他のプラットフォームで評価を行なうことがあげられる．また，本論文では，アクティブに動作するプロセスがコア数以下である場合を想定して提案手法手法を提案した．しかし，コア数以上のプロセスが同時に実行されることも多く，そのような場合にはある瞬間に同時に実行されているプロセス同士の状況だけでなく，並行的に実行される全プロセスの状況を総合的に判断しつつ，スピードをコントロールをする必要がある．このような，コア数以上のプロセスが同時に実行される状況においても効率的な実効を提供できる提案手法手法を構築することも今後の課題である．

謝辞 本研究の一部は，科学技術振興機構・戦略的創造研究推進事業（CREST）の研究プロジェクト「革新的電源制御による超低電力高性能システムLSIの研究」，および文部科学省科学研究費補助金（基盤研究(A) No.18200002）の支援によって行われた．

参 考 文 献

1) G.E.Suh, et al., "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning", *In Proc. 8th HPCA*, pp.117–128, Feb. 2002.
2) S.Kim, et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", *In Proc. 13th PACT*, pp.111–122, Oct. 2004.
3) M.K. Qureshi and Y. Patt,"Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", *In Proc. 39th MICRO*, pp.423–432, Dec. 2006.
4) L.R. Hsu, S.K. Reinhardt, R.K. Iyer, S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource", *In Proc. 15th PACT*, pp.13–22. Sep. 2006.
5) K.J.Nesbit, et al., "Fair Queuing Memory System", *In Proc. 39th MICRO*, pp.208–219, Dec. 2006.
6) 近藤, 中村, "CMP向け動的電源電圧・周波数制御手法", *In Proc. SACSIS2007*, pp.103–110, May 2007.
7) 佐々木, 浅井, 池田, 近藤, 中村, "統計情報に基づく動的電源電圧制御手法", 情報処理学会論文誌, Vol.47, No.SIG18 (ACS16), 2006年11月.
8) 佐々木, 近藤, 中村, "CMPにおけるリソース競合に着目した性能の解析とモデリング", 情報処理学会研究報告, ARC-174, 2007年8月.
9) S.Browne, et al. "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters" *In proc Supercomputing 2000*, Nov. 2000.
10) N.Rafique, et al., "Architectural Support for Operating System-Driven CMP Cache Management", *In Proc. 15th PACT*, pp.2–12, Sep. 2006.
11) D.Chandra, et al., "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture", *In Proc. 11th HPCA*, pp.340–351, Feb. 2005.
12) R.R.Iyer, "CQoS: A Framework For Enabling QoS in Shared Caches of CMP Platforms", *In Proc. ICS2004*, pp.257–266, June 2004.
13) A. Fedorova, M. Seltzer, and M.D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler", *In Proc. 16th PACT*, pp.25–38, Sep. 2007.
14) A.Snavely and D.M.Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor", *In Proc. ASPLOS IX*, pp.234–244, Nov. 2000.
15) K. Luo, et al. "Balancing Throughput and Fairness in SMT Processors", *In Proc. ISPASS 2001*, pp.164–171, Nov. 2001.
16) R.Gabor, et al., "Fairness and Throughput in Swith on Event Multithreading", *In Proc. 39th MICRO*, pp.149–160, Dec. 2006.
17) T. Moseley, et al. "Methods for Modeling Resource Contention on Simultaneous Multithreading Processors", *In Proc. ICCD2005*, pp.373–380, Oct. 2005.