情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 2　111–123 (Aug. 2008)

トラクションコントロール実行：CMP 向けプロセス実行制御方式の提案

近藤 正章[†1,*1]　佐々木 広[†2,*2]　中村 宏[†1,†3]

近年では，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，複数のプロセスが L2 キャッシュやチップ・メモリ間バスなどのリソースを共有するが，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性能が低下する，あるいは各プロセスの性能低下の影響の公平さ（Fairness）が保たれないなどの問題が生じる．本論文では，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に，トラクションコントロール実行を提案する．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の向上に応用した結果，Fairness を大きく改善でき，またトータルスループットが向上するアプリケーションも多いことが分かった．

Traction Control Execution: Optimizing Concurrent Process Execution for CMPs

Masaaki Kondo,[†1,*1] Hiroshi Sasaki[†2,*2] and Hiroshi Nakamura[†1,†3]

Recently, a single chip multiprocessor (CMP) is becoming an attractive architecture due to its advantage of achieving high throughput and low power. In CMPs, multiple processor cores share several hardware resources such as cache memories, memory buses, and main memory banks. Performance degrades significantly if resource contention occurs. In this paper, we propose Traction Control Execution (TCE) which controls execution speed of threads running on multiple cores to optimize shared resource utilization. We apply TCE to fairness and total throughput improvement. The evaluation results reveal that TCE is very effective for improving fairness and total throughput in many applications.

1. はじめに

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（Chip MultiProcessor: CMP）が主流となっている．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らずに高性能化を達成できるため，性能あたりの消費電力効率に優れるアーキテクチャであると考えられている．CMP ではリソース有効活用の観点から，複数のプロセッサユニット（PU）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 のバンクを共有するのが一般的である．しかし，複数の PU が同時に共有リソースへアクセスした際に競合が発生すると性能が低下する恐れがある．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い．

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，この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．たとえば，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，ある程度はリソース競合の影響を緩和させることができると考えられるが，将来的により多くのプロセッサコアが 1 チップに集積されリソース競合の影響も深刻になると，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．

そこで本論文では，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

†1 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
†2 東京大学大学院工学系研究科
School of Engineering, The University of Tokyo
†3 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
∗1 現在，電気通信大学大学院情報システム学研究科
Presently with Graduate School of Information Systems, The University of Electro-Communications
∗2 現在，東京大学先端科学技術研究センター
Presently with Research Center for Advanced Science and Technology, The University of Tokyo

グラム実行環境を提供することを目的として,「トラクションコントロール実行（Traction Control Execution: TCE)」を提案する．TCE 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で，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に従ってオペレーティングシステム（OS）により行われる．TCE による実行スピード制御によりハードウェアの追加・拡張なしに競合の影響を調節できるため，従来の手法に比べて柔軟に種々の最適化が可能となる．本論文では，TCE 手法の詳細について述べ，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うことでその有効性を明らかにする．

本論文の構成は以下のとおりである．次章では CMP におけるリソース共有の影響について述べる．3 章では TCE を提案し，その際に必要となる統計的モデリングによる競合の影響予測について 4 章で述べる．5 章では評価環境および評価条件について述べ，6 章に評価結果を示す．7 章で関連研究について述べ，最後に 8 章でまとめと今後の課題を記す．

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 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った．Core2-QX6700 は，実際には 4 MB 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 4 コアを持つ CMP であるが，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った．なお，どの 2 コア上でプロセスを実行するかにより，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合せにおける，2 プロセスを同時に実行した場合の Multiprogrammed Speedup を示す．Multiprogrammed Speedup は，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合せにおける Multiprogrammed Speedup を，“箱ひげグラフ（box-and-whisker plot)” で表しており，対象プログラムごとに，箱部分が *IQR*（*Inter-Quartile Range*）と呼ばれる中央 50%範囲内のデータを，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．その範囲外のデータは，はずれ値としてそのデータポイントが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，対象プログラムのミス率が高い場合には Multiprogrammed Speedup が大き

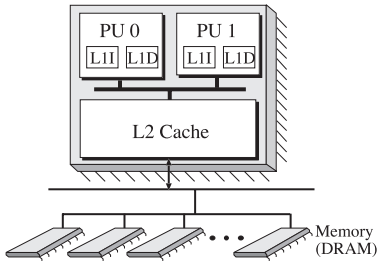

図 1　CMP の概要
Fig. 1　Overview of a CMP.

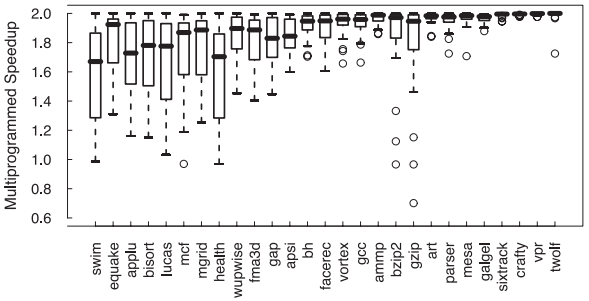

図 2　Core2-QX6700 における Multiprogrammed Speedup
Fig. 2　Multiprogrammed Speedup on Core2-QX6700.

く低下してしまうことが分かる．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．次章では，このリソース競合の影響を緩和させるための手法について述べる．

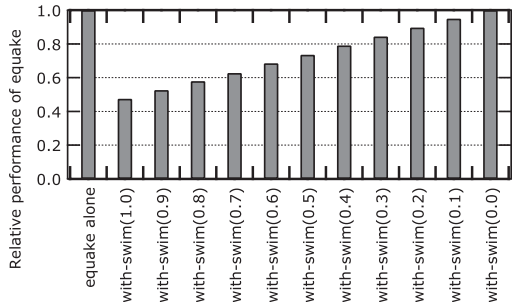

図 3　swim の実行スピードを変化させた場合の equake の性能
Fig. 3　Performance of equake vs. execution speed of swim.

3. トラクションコントロール実行

本章では，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トラクションコントロール実行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 が保たれなかったりするといった問題が生じる．

TCE 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，前章と同じく 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った．**図 3** に，前章と同じく 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従って，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分かる．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．

なお，本論文では各コアではある 1 つのプロセスが定常的に動作する，すなわちアクティブに動作するプロセスはコア数以下であり，頻繁にプロセス・スイッチは行われないようなシステムの状況を想定する．これは，バッチジョブシステムやマスタ・ワーカ方式での並列処理の各ノードのように，通常アクティブに動作するプロセスがコア（CPU）数を超えないように制御されているシステムがその対象となる．特にそのようなシステムでは Fairness やスループットが重要な指標であることが多く，それらの低下を防ぎつつ効率的な実行を提供できる TCE 手法の価値は大きいと考えられる．

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，時間的に最も細粒度にスピード制御を行うことができ，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も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（たとえば Core2-QX6700 では 2.4 GHz，2.0 GHz，1.6 GHz の 3 通り），コアごとに個別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も考えられる．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，割込みなどのオーバヘッドがあるため短くても数マイクロ秒程度と時間的には粒度が粗く，実行スピード制限を行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えるという利点を持つ．本論文では，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　TCE のための性能予測

TCE により各種最適化を行う際には，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把握する必要がある．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，実行時に動的に予測する必要がある．ここで，プラットフォームごと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，本論文では論文 7) を基にした統計的学習手法により性能低下を予測することにする．この統計的学習手法は，パフォーマンスカウンタと性能低下率の関係をあらかじめ統計的にモデル化しておき，実行時にはカウンタの値を参照しつつ性能低下を予測するものである．なお，本手法の詳細は 4 章で説明する．

3.4　TCE による実行時最適化

図 4 に TCE 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．ある時間間隔（TCE インターバルと呼ぶ）ごとに，OS 上の Runtime-Controller 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統計的学習に基づく手法を用いて競合による性能低下率の予測を行う．次に，最適化すべき目的に応じて設定されたポリシに基づいて性能低下率を調整するべく，Runtime-Controller は次の TCE 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 TCE インターバル経過後にも同様に性能低下率の予測，およびポリシに基づいた duty-cycle の制御を行う．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．

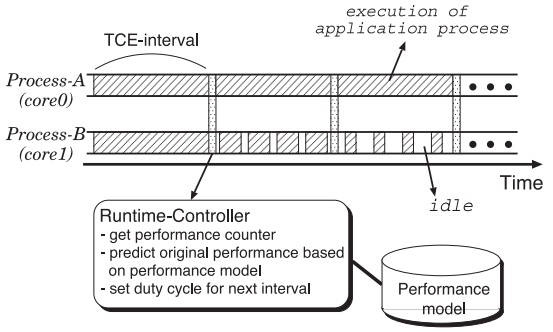

図 4　TCE による実行時最適化の概要
Fig. 4　Dynamic optimization with TCE.

3.4.1　Fairness 改善への応用

ここでは，Fairness を改善させるための duty-cycle 制御方法を説明する．Algorithm 1 に Fairness 改善のための duty-cycle 調整のポリシを示す．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，単独実行時に対する複数プロセス実行時の性能低下率（PR）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（PR_{avg}）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 より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，当該プロセスの duty-cycle をあらかじめ設定したパラメータである LOW_DUTY に設定する．LOW_DUTY はたとえば 0.5 などの 1 よりも低い値となる．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われるため，Fairness を改善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を Algorithm 2 に示す．まず，Fairness 改善ポリシと同様に性能低下率の予測を行う．次に，種々の duty-cycle の組合せ（$Dvec$）

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 TCE-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 ⇐ LOW_DUTY
    else
      Duty_i ⇐ 1.0
    end if
  end for
end for
```

を想定し，仮にそれらの duty-cycle で実行した場合に各プロセスの IPC（Instruction Per Cycle）がどうなるかを予測する．具体的には，まずプロセスごとに，現在の IPC とそのプロセスの duty-cycle（$Dvec_{k,i}$: k 番目の duty-cycle の組合せにおける，i 番目のプロセスの duty-cycle を意味する）をかけることで，自身のスピードが制御された場合の IPC を求める．さらに，他のプロセスの実行スピードを低下させた場合は自身の性能が向上するので，その向上分を計算する必要がある．このために，現在の IPC に対して PR の逆数をかけることで競合がなかった場合の IPC を求め，そこから現在の IPC を引くことで増分の最大値を求める（アルゴリズム中の ($\frac{IPC_i}{PR_i} - IPC_i$) に該当）．それに対し，他のプロセスの速度低下分（$1 - Dvec_{k,j}$）と他プロセスが動作するコアが自プロセスのコアに対して，どれだけの影響を及ぼすのかの度合い（$Eff_{i,j}$: i 番目のコアに対する j 番目のコアの影響の

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
Dvec: Possible Duty cycle combination Vector

for each end of TCE-interval do
  for i = 0 to NPU − 1 do
    PR_i = GetPerfRatio(i, Duty_0 .. Duty_NPU−1)
  end for

  for all k in possible Duty combination do
    P_IPC_k = 0
    for i = 0 to NPU − 1 do
      P_IPC_k += IPC_i × Dvec_k,i
                 + Σ_{j=0}^{NPU−1} {(IPC_i/PR_i − IPC_i)
                 × (1 − Dvec_k,j) × Eff_i,j}
    end for
  end for
  p = SelectMAXIPC(P_IPC)
  SetDuty(Dvec_p)
end for
```

度合いを意味する）をかけ，自身の性能の向上分を求めることができる．なお，$Eff_{i,j}$ は CMP のアーキテクチャなどに依存する値であり，いくつかのプログラムを実行してスピードを調整しつつ，各コアの性能にどの程度影響があるかを実験的に求めることでプラットフォームごとに取得するパラメータである．そして，全コア分の IPC と向上分を合計することで，$Dvec_k$ における IPC を予測する．最終的に，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．

4. 統計的モデリングによる競合の影響の予測

2 章でも述べたように，CMP はその構成や同時に実行するプログラムの組合せによって単独で実行した場合に比べて性能が大きく異なり，その振舞いは複雑である．したがって，リソース競合が性能に与える影響をプログラムの特徴や CMP の構成などから定性的に予

測することは難しいと考えられる．我々はこのような定性的に振舞いを理解することが困難な事象に対して，重回帰分析を用いた統計的な学習を用いることによってモデリングを行う手法を提案しており[7]，本論文ではこれを応用することで競合による性能低下を予測する．

4.1　オフラインでの統計的モデリング

4.1.1　重回帰分析

重回帰分析は多変量解析の 1 つで，従属変数（y）と独立変数（x_i）との関係を調べ，関係式を導くことによって独立変数から従属変数の予測を行う手法であり，その関係式は一般的に式 (1) で表される．ここで，y は従属変数，x_i は独立変数，β_0 は定数項，β_i はそれぞれの独立変数にかける重み（偏回帰係数），e は誤差項である．

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + e \tag{1}$$

本論文でのモデリング手法では，単独で実行した際に比べて複数プロセスで同時に実行した際の性能比を従属変数，それぞれのコアで実行しているプログラムの振舞いを示すパフォーマンスカウンタの値を独立変数とする．ここで，CMP において複数のコアで同時にプロセスを実行している際にはリソース競合が起こり，その影響によって性能が変化することは述べたが，同時にそれぞれのカウンタ値にも影響を与えることが考えられる．たとえば，あるコアで L2 キャッシュミスが頻発した場合に，他のコアは競合によって性能が低下し，その結果として L2 キャッシュミス回数にも変化が出てくるような場合がそれにあたる．通常の重回帰分析ではある独立変数 x_1 の値が変化しようがしまいが他の独立変数 x_2 と従属変数 y の関係は一定であると仮定されているため，上記のような事象を表すことができない．これは独立変数間の交互作用の問題と呼ばれ，この交互作用をモデルに組み込むためには，独立変数同士の積を新たな独立変数 $x_3 = x_1 x_2$ として投入する手法が用いられることが多く，本手法でもコア間の競合による影響をモデル化するためにこの手法を用いる．以下に交互作用付きの線形回帰モデルを示す．

$$y = \beta_0 + \sum_{i=1}^{n-1} \beta_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \beta_{i,j} x_i x_j + e \tag{2}$$

4.1.2　学習のためのデータサンプリング

本項では，統計的な学習による性能のモデリングを行うために必要なデータのプロファイリング手法について述べる．対象のプラットフォームにおいて，様々な未知のアプリケーションを同時に実行した場合における性能の低下率を正確に予測するため，学習には様々な特徴を持つアプリケーションを広く用い，大量のサンプルデータを取得する必要がある．このように，様々な種類のアプリケーションを用いてモデリングを行うことで，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．

本モデリングにおいては，従属変数（y）を単独で実行した際に比べて複数プロセスで同時に実行した際の性能比，独立変数（x_i）をそれぞれのコアにおけるパフォーマンスカウンタの値としている．多くのアプリケーションは振舞いが最初から最後までで一定でなく，様々なフェーズに分かれている．このようなプログラム内の細粒度の振舞いを学習に用いてモデルに反映させるために，それぞれのコアでパフォーマンスカウンタの値を一定間隔ごとにサンプリングし，それぞれを異なる学習用のサンプルデータとする．本論文では，サンプリングの際には 2 つのプログラムを同時に異なるコアで実行しはじめ，30 ms ごとにハードウェアのタイマ割込みによってシグナルを送り，そのつどパフォーマンスカウンタの値を取得することにする．これらのアプリケーションはそれぞれ長さが異なっており，特定のアプリケーションの影響が大きくなることを防ぐため，それぞれの組合せにおいて 400 サンプルを取得した時点（約 12 秒）でサンプリングを終了とする．また，性能の低下率を算出するために，単独で実行した際の性能が必要となるため，同じように 30 ms 間隔でサンプリングを行う．

4.1.3　モデリング

前項の手法でサンプリングしたデータを基に，重回帰分析を用いてモデリングを行う．モデリングに際して，実行時に動的に性能の予測を行うことを想定した場合，用いるカウンタの個数は対象とするプラットフォームで各コアが同時に取得できるカウンタの個数に制限される．たとえば，2 つのカウンタを用いる場合のモデルは $x_{(i,\ c)}$ をコア i におけるカウンタ c として，以下の式 (3) となる．なお，本モデルではそれぞれのコアで同じ種類のカウンタを用いることとしている．また，前節の TCE による最適化アルゴリズム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は必ず含まれる必要がある．なお，性能の低下率は，単独で実行したときと複数のプロセスを実行したときで，命令数を揃えつつかかったサイクル数の比率を計算して求めることができる．

$$\begin{aligned} y = \beta_0 &+ \beta_1 x_{(0,\ 1)} + \beta_2 x_{(0,\ 2)} \\ &+ \beta_3 x_{(1,\ 1)} + \beta_4 x_{(1,\ 2)} \\ &+ \beta_5 x_{(0,\ 1)} x_{(1,\ 1)} + \beta_6 x_{(0,\ 1)} x_{(1,\ 2)} \end{aligned}$$

$$+\beta_7 x_{(0,\,2)}x_{(1,\,1)}+\beta_8 x_{(0,\,2)}x_{(1,\,2)} \quad (3)$$

最終的にモデルに用いるカウンタは，すべてのカウンタ値を用いて制限にかからない全通りのモデリングを行い，寄与率の最も高いカウンタの組合せを選択する．なお，6.1 節においては，種々のプラットフォームにおいて上記の統計的学習手法を適用し，選択されたカウンタとその際の寄与率を評価する．

4.2 オンラインでの適用

統計的学習で得られた予測モデルに基づき，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，3.4 節で述べた TCE インターバルごとにカウンタの値を取得し，それを予測式にあてはめることで各プロセスごとにその時点の性能低下率を求めることができる．

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，本論文ではさらに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 TCE インターバルのうち，数回に 1 回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．この単独で実行させる間隔をフィードバックインターバルと呼ぶ．その単独での性能と，それまで複数プロセスで実行していたフィードバック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，同じフィードバックインターバル中の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か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われ，予測精度が向上すると考えられる．

5. 評　　価

5.1 評価環境

TCE の効果を調べるために，本論文では CMP を搭載した実機のマシンとして，2 章でも用いた Core2-QX6700，および *AMD Athlon 64X2 3800+*（以下，Athlon-64X2）を搭載した PC を用いて評価を行う．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，本論文では以下に示す 4 つの環境で評価を行う．

- Core2-SHR：Core2-QX6700 において L2 キャッシュを共有する 2 つのコアを用いて評価

表 1　評価に用いたマシンの仕様
Table 1　Evaluated platforms.

	Core2-QX6700	Athlon-64X2
CPU	Intel Core2 Extreme QX6700	AMD Athlon64 X2 3800+
- CPU 周波数	2.66 GHz	2.0 GHz
- L2 キャッシュ	8-Way, 4 MB×2	16-Way, 512 KB×2
- メモリバス	FSB 1066 MHz	5.3 GB/s
M/B	Intel D975XBX2	ASUS M2A-VM
主記憶	DDR2-SDRAM	DDR2-SDRAM
- 周波数	667 MHz (PC667)	667 MHz (PC667)
- サイズ	1 GB	512 MB

- Core2-DED：Core2-QX6700 において L2 キャッシュを共有しない 2 つのコアを用いて評価
- Core2-QUAD：Core2-QX6700 において 4 つのコアを用いて評価
- Athlon：Athlon-64X2 において 2 コアを用いて評価

OS は Linux-2.6.18 であり，カウンタ値の取得には PAPI（Performance Application Programming Interface）[9] を用いた．Core2-QX6700 は 33 種類のパフォーマンスカウンタを持ち，コアあたり同時に 2 つのカウンタ値を取得可能である．また，Athlon-64X2 は 29 種類のパフォーマンスカウンタを持ち，コアあたり同時に 4 つのカウンタ値を取得可能である．

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，art，lucas，mgrid，equake，mcf を選択し，その全組合せを実行した．ここで，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わり 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 1 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，および Olden ベンチマークを用いた．これは，未知のプログラムが実行された場合でも 4 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，TCE による最適化が有効に働くことを示すためである．

5.2 評価手法

本論文では，実行スピードの制御手法として OS によるタイムスライスの割当て制御をベースとするが，本評価では実際に OS を拡張せずに，以下に述べるように sleep() を用い

て CPU リソースの使用時間を制限することで，仮想的にタイムスライスの割当て制御を行い評価を行う．

まず，Runtime-Controller は，3.4 節で述べた TCE インターバルよりも短い時間間隔 $T_{interrupt}$ で duty-cycle を制限するプロセスに対し，割込みのためのシグナルを送る．当該プロセスはシグナルを受け取ると割込み処理のための関数に移り，その中で sleep() を用いて (1 - duty-cylce) $\times T_{interrupt}$ 時間スリープする．その後に，再び割込み処理前の状態から実行を再開する．これにより，図 4 とほぼ同様の動作を行うことが可能となる．

5.3　評価の仮定

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．

- TCE インターバル：120 ms
- TH：0.05
- LOW_DUTY：0.5
- フィードバックインターバル：TCE インターバル × 50 回
- $T_{interrupt}$：120 ms（= TCE インターバル）

なお，トータルスループットを向上させるためのポリシにおける $Eff_{i,j}$ の値は，2 コアの評価プラットフォームは各コアで 1，Core2-QUAD の場合は L2 キャッシュを共有するコアは 0.75，非共有のコアは 0.125 として評価を行う．

6. 評 価 結 果

6.1　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最も寄与率が高い組合せとして選択されたカウンタと，そのときの寄与率を示す．なお，4.1 節でも述べたように，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った結果である．表中のその他のカウンタは，*L1D accesses* が L1 データキャッシュへのアクセス回数，*Cycles stalled on any resource* はリソース待ちによるストールサイクル数，*Cycles with no inst. issue* は命令発行のなかったサイクル数を意味する．

表 2　性能低下率予測のために選択されたカウンタ
Table 2　Selected counters for predicting performance degradation.

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
Core2-QUAD	Inst. completed, L1D accesses	0.425
Athlon	Inst. completed, L1D accesses Cycles stalled on any resource Cycles with no inst. issue	0.690

表より，全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分かる．直感的には L2 キャッシュミス回数が適切と思われるが，たとえば L2 キャッシュミス回数が少ない場合に，競合により自コアの性能が低下した結果，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．なお，Athlon では，リソース待ちによるストール数というリソース競合による性能低下を予測するうえで非常に重要となりうるカウンタの値が選択されており，妥当な結果である．次節以降の評価はすべて表 2 に示すカウンタを用いて予測を行ったものである．

6.2　Fairness 改善ポリシの評価結果

図 5 に，TCE を用いない場合（ORG），および TCE を用いた場合（TCE）の 2 コアのプラットフォームにおける，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．この差が小さいほど Fairness は良いことになる．なお，各プログラムごとに他の 1 つのプログラムを同時に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 6** は，4 コアを持つ C2-QUAD プラットフォームにおける，各プロセスの性能低下率の差の最大値を示したものである．なお，この場合は評価した各プロセスの組合せを個別に表しており，S は swim，L は lucas，G は mgrid，E は equake，M は mcf を表している．

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 TCE により Fairness を改善できていることが分かる．特に，元々の Fairness が悪い swim や mgrid との組合せの場合では 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た結果，後者のプロセスが共有リソースに円滑にアクセスでき，性能低下率がバランスしたためである．

一方，ORG において性能低下率の差が小さいものは TCE の効果が小さい．これは，元々

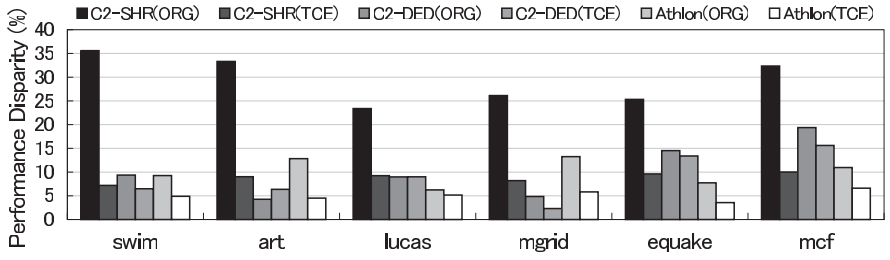

図 5　各プロセスの性能低下率の差（2 コア）
Fig. 5　Disparity in performance degradation (2-core).

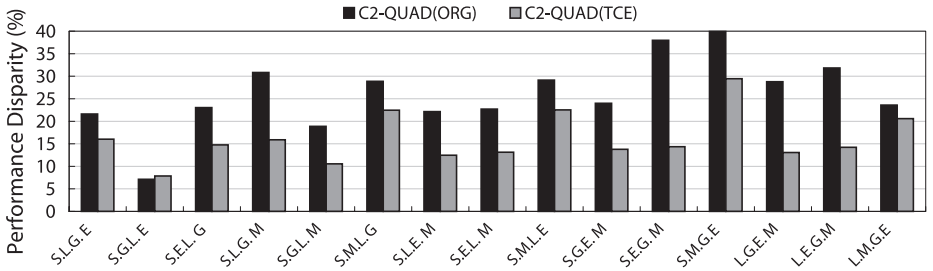

図 6　各プロセスの性能低下率の差（4 コア）
Fig. 6　Disparity in performance degradation (4-core).

ORG でも良い Fairness が達成されている場合には，TCE 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．さらに C2-DED の art では TCE を用いることにより Fairness が悪化してしまっている．これも，ORG で良い Fairness を達成しており，TCE により実行スピードを変更した際に性能低下率が等しくなるよう制御できず，かえって Fairness を悪化させてしまったことが原因である．ただし，これらの場合にも悪化率はそれほど大きくない．

次に，図 6 の C2-QUAD 上での結果を議論する．図より，2 コアの場合に比べて性能差が大きいことが分かる．これは 4 コアの方が共有リソースへのアクセスが多くなり，相対的に競合の影響が大きいためである．この場合，TCE を用いることで S.G.L.E を除くすべての場合において Fairness が改善しており，提案手法の有効性が高いと考えられる．なお，S.G.L.E で Fairness が悪化しているのは，2 コアの場合と同じく元々ORG の Fairness が良く，TCE により性能低下率が等しくなるようううまく制御できなかったためである．

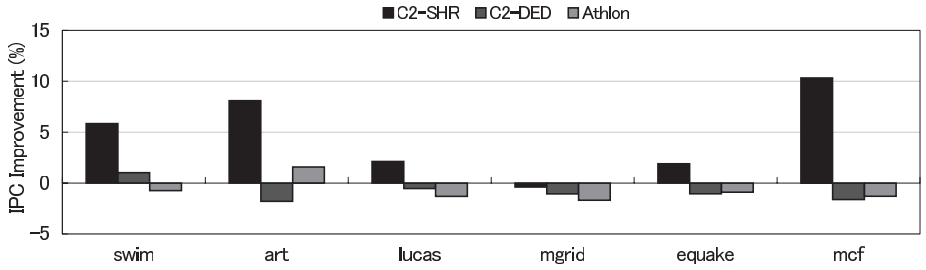

図 7　トータルスループットの向上率（2 コア）
Fig. 7　Improvement of total throughput (2-core).

これらより，TCE を Fairness 改善に応用することで，実際に Fainress 改善の効果があることが分かった．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ったものであるが，期待されたとおりに Fairness が改善され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 TCE 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．

6.3　トータルスループット向上ポリシの評価結果

図 7，**図 9** に 2 コアおよび 4 コア（C2-QUAD）のプラットフォームにおける，ORG に対する TCE のトータルスループットの向上率を示す．前節同様，2 コアの場合には各プログラムごとに他の全プロセスと同時に実行した場合の平均の値を，4 コアの場合は，評価した各プロセスの組合せの結果を個別に示している．

図より，いくつかのプログラムの組合せで，実際にトータルスループットが改善されていることが分かる．特に，C2-SHR や C2-QUAD 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，それらのプラットフォームで L2 キャッシュを共有しており，ORG ではリソース競合による性能低下の影響が大きく，TCE により性能向上の余地が大きかったためである．一方，元々競合の影響が少ないプラットフォームでは TCE 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ヘッドのためである．しかし低下率は大きくなく，TCE の有効性は高いと考えられる．

以上の結果より，TCE 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分かり，TCE 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．

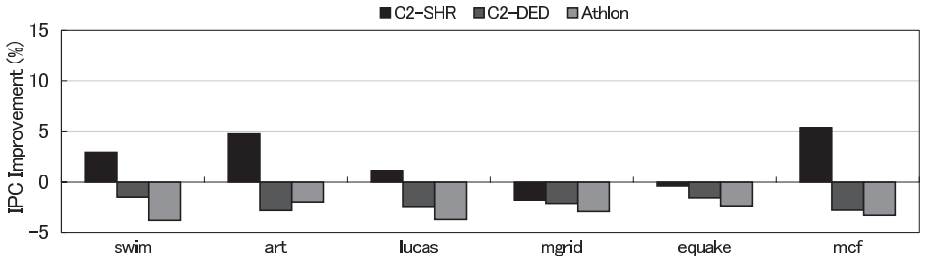

図 8　トータルスループットの向上率（2 コア，性能予測の微調整なし）
Fig. 8　Improvement of total throughput (2-core, without on-line feedback).

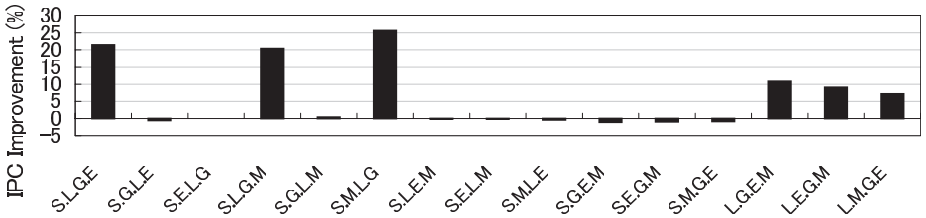

図 9　トータルスループットの向上率（4 コア）
Fig. 9　Improvement of total throughput (4-core).

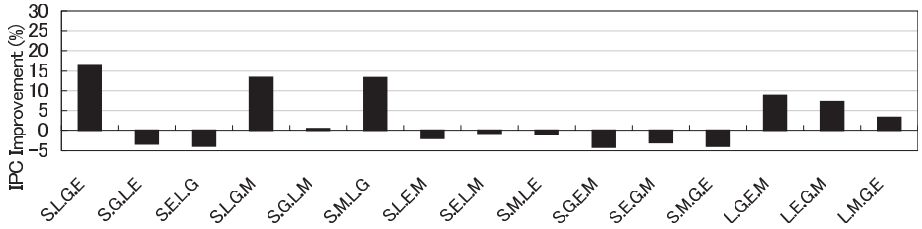

図 10　トータルスループットの向上率（4 コア，性能予測の微調整なし）
Fig. 10　Improvement of total throughput (4-core, without on-line feedback).

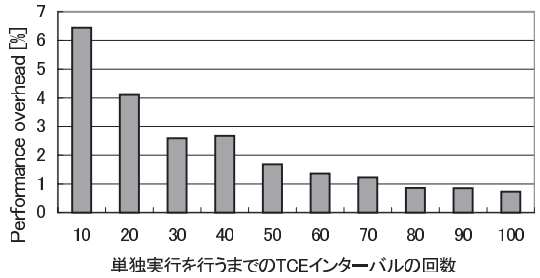

図 11　オンラインフィードバックのオーバヘッド
Fig. 11　Performance overhead of on-line feedback mechanism.

また，4.2 節で述べたオンラインでの性能予測の微調整の効果を調べるため，微調整を行わなかった場合の 2 コアおよび 4 コアのプラットフォームにおける，ORG に対する TCE のトータルスループットの向上率を**図 8**，**図 10** に示す．図 7，図 9 と比較すると，多くのプログラムにおいて微調整を行わなかった場合にトータルスループットの向上率が低下してしまっている．さらには，スループットが ORG に対して悪くなってしまう場合も増えている．これは，微調整を行わなかったことで競合による性能低下の予測が正確に行えず，必要な場合にスピードの制御ができない，あるいは間違った制御をしてしまったことが原因である．このことから，本論文で提案するように，統計的学習による性能低下の予測に対してオンラインでの微調整を行うことが重要であり，この微調整の結果正確に性能低下を予測できるようになることで，TCE の最適化にとって非常に有効であることが分かる．

6.4　オンラインフィードバックのオーバヘッド

4.2 節で述べたように，統計的学習結果へのオンラインでの性能予測の微調整を行ううえで，TCE インターバルの何回かに 1 回各プロセスを単独で実行する必要がある．この単独実行により，ある程度性能へのオーバヘッドが発生する．本節ではこのオーバヘッドの影響を調べるために，競合を起こさないプログラムどうし（具体的には，L2 キャッシュミス率の非常に小さい vpr と twolf）を Core2-DED 上で実行し評価を行い，オンラインでの微調整をする場合としない場合で比較を行った．なお，競合がほとんど生じないため実行スピードの調節は行われず，TCE の影響はない．

図 11 は，TCE インターバルの何回に 1 回，各プロセスを単独実行させるかの回数を変化させた場合の，オンラインの微調整を行わなかった場合に対する性能（スループット）低下率を示したものである．図より，単独実行を行うまでの TCE インターバルの回数が少ない場合頻繁に単独実行が行われるため，微調整の精度は改善されると考えられるが，オーバヘッドが大きくなり性能が大きく低下してしまう．逆に TCE インターバルの回数を増やすと，単独実行を行うまでの期間が長くなるため，微調整の精度は悪くなるがオーバヘッドは

小さくなる．したがって，この回数は求められる精度とオーバヘッドとのトレードオフで決める必要がある．前節の評価で，フィードバックをした場合にもトータルスループットが向上したのは TCE の効果が性能低下分を上回ったためであり，TCE インターバル 50 回程度の間隔でフィードバックを行うのは適切であったと思われる．しかし，競合が発生しないような場合ではオーバヘッドがそのまま現れてしまうため，それを検出して性能低下を避けるような手法が必要となる．この点に関しては今後の課題である．

7. 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，性能向上や Fairness 改善を狙う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 の改善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について検討が行われている．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 を改善させるものである．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 TCE 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替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 改善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．

本論文で提案する TCE は，キャッシュだけでなく，メモリバスなどを含めた一般的な共有リソースに対して，プロセスの実行スピードを調整することで，競合による性能低下の影響を柔軟に制御することを目的としている．現在でも，共有リソースを持つ SMP（Symmetric Multi Processor）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 TCE 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，本論文の大きな貢献である．今後，CMP が広く普及することが予想されるが，CMP では SMP に比べ共有するリソースも増え，また共有の度合いも強くなるため，リソース競合の影響を緩和し効率的な実行環境を提供できる OS の必要性が増すと考えられる．したがって，TCE は将来的に非常に有望な手法であると考えられる．

8. まとめと今後の課題

本論文では，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に，トラクションコントロール実行（TCE）を提案した．TCE は，各プロセスの競合による性能への影響を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に従って各プロセスの実行のスピードを調整し，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 改善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，また多くのプログラムの組合せでトータルスループットが向上することが分かった．

今後の課題としては，他のプラットフォームで評価を行うほか，実際に OS に TCE を実装して評価を行うことで TCE の有効性を検討していくことがあげられる．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 TCE 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どうしの状況だけでなく，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行を提供できる TCE 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) Suh, G.E., Devadas, S. and Rudolph, L.: A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning, *Proc. 8th Intl. Symp. on High-Performance Computer Architecture*, pp.117–128 (2002).
2) Kim, S., Chandra, D. and Solihin, Y.: Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture, *Proc. 13th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.111–122 (2004).
3) Qureshi, M.K. and Patt, Y.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *Proc. 39th Intl. Symp. on Microarchitecture*, pp.423–432 (2006).
4) Hsu, L.R., Reinhardt, S.K., Iyer, R.K. and Makineni, S.: Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource, *Proc. 15th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.13–22 (2006).
5) Nesbit, K.J., Aggarwal, N., Laudon, J. and Smith, J.E.: Fair Queuing Memory System, *Proc. 39th Intl. Symp. on Microarchitecture*, pp.208–219 (2006).
6) 近藤正章，中村　宏：CMP 向け動的電源電圧・周波数制御手法，*Proc. SACSIS2007*, pp.103–110 (2007).
7) 佐々木広，浅井雅司，池田佳路，近藤正章，中村　宏：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：コンピューティングシステム，Vol.47, No.SIG18(ACS16), pp.80–91 (2006).
8) 佐々木広，近藤正章，中村　宏：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, pp.85–90 (2007).
9) Browne, S., Dongarra, J., Garner, N., London, K. and Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters, *Proc. Supercomputing 2000* (2000).
10) Rafique, N., Lim, W.-T. and Thottethodi, M.: Architectural Support for Operating System-Driven CMP Cache Management, *Proc. 15th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.2–12 (2006).
11) Chandra, D., Guo, F., Kim, S. and Solihin, Y.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture, *Proc. 11th Intl. Symp. on High-Performance Computer Architecture*, pp.340–351 (2005).
12) Iyer, R.: CQoS: A Framework For Enabling QoS in Shared Caches of CMP Platforms, *Proc. 18th International Conference on Supercomputing*, pp.257–266 (2004).
13) Fedorova, A., Seltzer, M. and Smith, M.D.: Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler, *Proc. 16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.25–38 (2007).
14) Snavely, A. and Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems IX*, pp.234–244 (2000).
15) Luo, K., Gummaraju, J. and Franklin, M.: Balancing Throughput and Fairness in SMT Processors, *Proc. 2001 Intl. Symp. on Performance Analysis of Systems and Software*, pp.164–171 (2001).
16) Gabor, R., Weiss, S. and Mendelson, A.: Fairness and Throughput in Swith on Event Multithreading, *Proc. 39th Intl. Symp. on Microarchitecture*, pp.149–160 (2006).
17) Moseley, T., Grunwald, D., Kihm, J.L. and Connors, D.A.: Methods for Modeling Resource Contention on Simultaneous Multithreading Processors, *Proc. International Conference of Computer Design 2005*, pp.373–380 (2005).

(平成 20 年 1 月 29 日受付)
(平成 20 年 5 月 14 日採録)

近藤　正章（正会員）

1998 年筑波大学第三学群情報学類卒業．2000 年同大学大学院工学研究科博士前期課程修了．2003 年東京大学大学院工学系研究科先端学際工学専攻修了．博士（工学）．独立行政法人科学技術振興機構戦略的創造研究推進事業 CREST 研究員，2004 年東京大学先端科学技術研究センター特任助手，2007 年同特任准教授を経て，現在電気通信大学大学院情報システム学研究科准教授．計算機アーキテクチャ，ハイパフォーマンスコンピューティング，ディペンダブルコンピューティングの研究に従事．電子情報通信学会，IEEE，ACM 各会員．

佐々木　広（正会員）

2003 年東京大学工学部計数工学科卒業．2005 年同大学大学院情報理工学系研究科修士課程修了．2008 年同大学院工学系研究科博士課程修了．博士（工学）．2008 年より，東京大学先端科学技術研究センター特任助教．

中村　　宏（正会員）

1985 年東京大学工学部電子工学科卒業．1990 年同大学大学院工学系研究科電気工学専攻博士課程修了．工学博士．同年筑波大学電子・情報工学系助手．同講師，同助教授，1996 年東京大学先端科学技術研究センター助教授，2008 年より東京大学大学院情報理工学系研究科准教授．この間，1996〜1997 年カリフォルニア大学アーバイン校客員助教授．高性能・低消費電力プロセッサのアーキテクチャ，ハイパフォーマンスコンピューティング，ディペンダブルコンピューティング，ディジタルシステムの設計支援の研究に従事．情報処理学会より論文賞（平成 5 年度），山下記念研究賞（平成 6 年度），坂井記念特別賞（平成 13 年度）各受賞．IEICE，IEEE，ACM 各会員．