2007－ARC－174 (14)
2007／8／1

トラクションコントロール実行: CMP向け実行制御方式の検討

近藤正章[†] 佐々木広[†] 中村宏[†]

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向上に応用した結果，Fairnessを大きく改善できることがわかった．

Traction Control Execution: Optimizing Thread Execution in CMPs

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

Recently, a single chip multiprocessor (CMP) is becoming an attractive architecture due to its high throughput with low power consumption. 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 improvement and evaluate the effectiveness of it. The evaluation results reveal that TCE is very effective for improving fairness.

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ることなし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として期待されている．CMPではリソース有効活用の観点から，複数の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をアクセスし，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．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を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．これら従来手法により，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考えられるが，今後より多くのプロセッサコアが1チップに集積され，リソース競合の影響も深刻になると，競合による種々の問題に対し，より柔軟に，また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に，「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ーに従って行なわれる．TCEによる実行スピード制御により，比較的細粒度に共有リソースの使用率を調整することができるため，従来の手法に比べて柔軟に種々の最適化が可能となる．

本稿では，TCE手法の詳細について述べ，実機のCMPプラットフォーム上にTCEを実装し，主にFairness向上を目的とした最適化について評価を行なう．

† 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

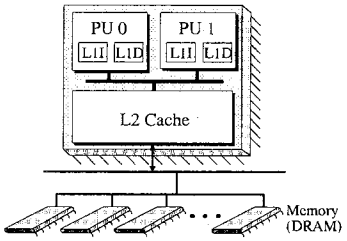

図 1 CMP の概要

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なった．Core2-QX6700 は，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，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 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 Multi-Programmed Speedup を示す．Multi-Programmed 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み合わせにおける Multi-Programmed 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，Multi-Programmed 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ると考えられる．キャッシュを共有した場合には，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．

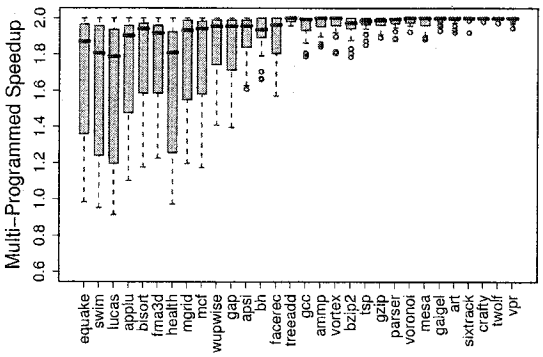

図 2 Core2-QX6700 における Multi-programmed Speedup

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．このリソース共有影響を緩和させるための手法について次節で述べる．

3. トラクションコントロール実行

本節では，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 が保たれない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．

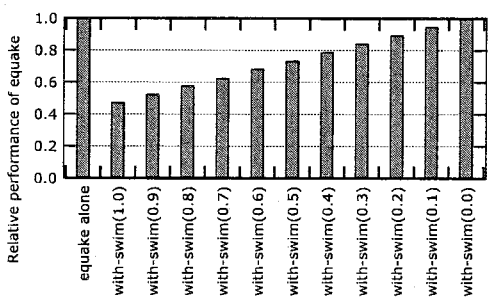

図 3　swim の実行スピードを変化させた場合の equake の性能

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，最も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も削減できるという利点もある．しかし，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，クロックを供給する時間 (duty-cycle) 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，duty-cycle を制限することで消費電力を低下させ，CPU のチップ温度を下げる目的で用いられる．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．

さらに，柔軟に実行スピードを制御するための方法として，オペレーティングシステム (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，ソフトウェアで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　TCE のための性能予測

TCE により各種最適化を行なう際には，プロセス毎に競合によりどれだけ性能が低下しているかを見積もる必要がある．通常，直接的にリソース競合の影響をモニタ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を考える．

まず，あらかじめ対象の CMP のマシン上で，様々なプログラムに対し，単一プロセス実行時，および複数プロセスを同時に実行した場合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，定期的に性能とパフォーマンスカウンタの値を取得する．次に，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とカウンタの値を重回帰分析により統計的にモデリングし，カウンタ情報から性能低下率を予測することができるようになる．

TCE を行なう場合には，実行時にパフォーマンスカウンタの値を取得しつつ，モデリング結果を参照してリソース競合による性能低下率を予測することで，未知のプログラムを実行した場合でも，動的に最適化を行なうことができるようになると考えられる．

なお，CMP におけるリソース競合に着目した性能のモデリングに関するさらなる詳細は，文献[8] で述べられている．

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 process-A と process-B が 2 つのコア (core0 と core1) 上で実行されている様子を示したものである．ある一定間隔 (TCE インターバル) 毎に，TCE のための Runtime-Controller が，その時点でのパフォーマ

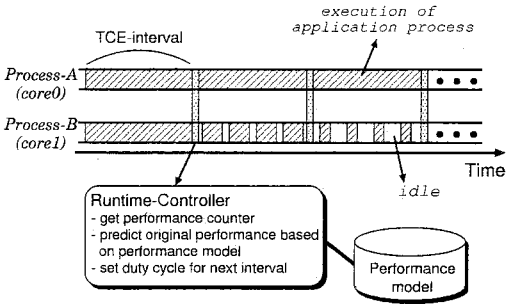

図 4 TCE による実行時最適化の概要

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への影響の予測を行なう．これにより，各プロセスが単独で実行した場合に比べて，どの程度性能が低下しているかを見積もることができる．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 TCE インターバルのための各プロセスの duty-cycle 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と考えられる．

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．図 5 に 2 プロセスを同時に実行させる際の Fairness 向上のためのポリシー，すなわち duty-cycle 調整のアルゴリズムを示す．

まず，Runtime-Controller は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する．ここで，すでに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，両プロセスが同じ時間動作した場合に得られるであろうカウンタ値に変換する必要がある．次に，取得したカウンタを用いて，あらかじめ学習した性能予測式に当てはめることで，図中 Pr_j (j はプロセス ID を意味する) で表わされている，プロセス毎に単独で実行した場合に比べての性能比を求める．

次に，Pr_j が大きい，すなわち性能低下が相対的に小さいプロセスの duty-cycle をどれだけ制限すれば性能低下率が等しくなるかを計算する．なお，他方のプロセスは常に 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，Fairness を向上することができると考えられる．

```
foreach i (Counter_0, Counter_1, ..., Counter_n) {
  C_{0_i} = get_perf_counter(0, i);
  C_{1_i} = get_perf_counter(1, i);
  C'_{0_i} = C_{0_i} × Pr_0 duty_1 / ((Pr_0 − 1) duty_1 + 1)
  C'_{1_i} = C_{1_i} × Pr_1 duty_0 / ((Pr_1 − 1) duty_0 + 1)
}
Pr_0 = get_perf_ratio(C'_{0_0}, ..., C'_{0_n}, C'_{1_0}, ..., C'_{1_n});
Pr_1 = get_perf_ratio(C'_{1_0}, ..., C'_{1_n}, C'_{0_0}, ..., C'_{0_n});

if (Pr_0 > Pr_1) {
  duty_1 = 1;
  duty_0 = 1 / (Pr_0 - Pr_1 +1);
}
else if (Pr_0 < Pr_1) {
  duty_1 = 1;
  duty_0 = 1 / (Pr_1 - Pr_0 +1);
}
else {
  duty_1 = 1;
  duty_0 = 1;
}
```

図 5 Fairness 向上のための TCE のポリシー

4. 評　価

4.1 評価環境

TCE 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，*AMD Athron 64X2 3800+* (以下，Athlon 64X2) 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．また Athlon-64X2 は，コア毎に 512KB の L2 キャッシュを持つデュアルコアプロセッサであり，メモリバス以下が共有リソースとなる．

表 1 評価に用いたマシンの仕様

CPU	AMD Athlon 64X2 3800+
- 周波数	2.0GHz
- L2 キャッシュサイズ	512KB × 2
- メモリバス	10.6GB/s
M/B	ASUS M2A-VM
主記憶	DDR2-SDRAM PC667
- サイズ	1GB

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] を用いる．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．文献[8] による CMP の競合のモデリングの結果，3.3 節で述べた性能予測のためのカウンタとして，「命令発行のなかったサイクル数」，「完了命令数」，「いずれかのリソース待ちによるストール数」，および「L1 データキャッシュアクセス回数」を用いることが有効であることがわかり☆，本稿ではそれらのカウンタを用いて競合による性能低下率の予測を行なう．

また，実行するプロセスは，SPEC および Olden ベンチマークのいくつかのプログラムを 2 つ同時に実行し，TCE を用いない場合の Fairness の値が幅広い組み合わせを選択する．ここで，プログラムによって実

☆ その際の寄与率は 0.69 であった．

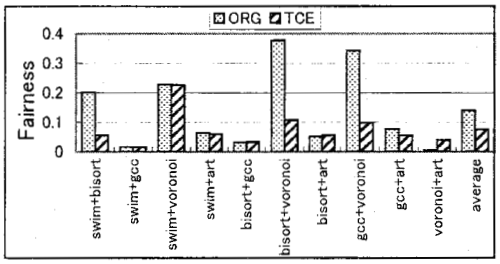

図 6 Fairness の値

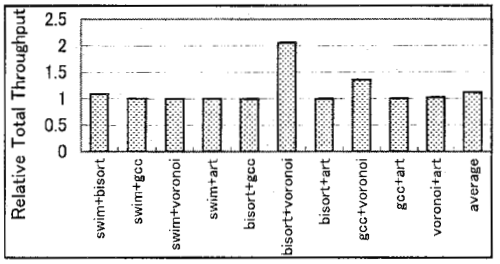

図 7 トータルスループットの相対値

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，両プロセスともが少なくとも3回の実行が終了するまでを評価対象とする．

4.2 評価手法

本稿では，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り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，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り当て制御を行なう．

まず，Runtime-Controllerは，3.4節で述べたTCEのための時間間隔よりも十分短い時間間隔 $T_{intrupt}$ でduty-cycleを制限するプロセスに対し，割り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，割り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) $\times T_{intrupt}$ 時間スリープする．その後に，再び割り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なうことが可能となる．

5. 評価結果

図6に，TCEを用いない場合 (ORG)，およびTCEを用いた場合 (TCE) におけるFairnessの値を示す．Fairnessの値は文献[2]を参考に，以下のように定義する．$Pded_i$ をリソース共有の影響がない場合の性能，$Pshr_i$ を複数のプロセスが動作している状況下でのプロセス i の性能として，プロセスaとプロセスbを同時に実行した場合のFairnessを，

$$Fair_{ab} = |X_a - X_b|, where X_i = \frac{Pshr_i}{Pded_i} \quad (1)$$

として求める．この，$Fair_{ab}$ の値が小さいほどFairnessは良いことになる．

図より，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わかる．特に，もともとのFairnessが悪い `swim+bisort` や `bisort+voronoi`，`gcc+voronoi` 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Fairnessが小さいものはTCE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，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 `bisort+art` および `voronoi+art` の場合は，TCEを用いることによりFairnessが悪化してしまっている．これらも，ORGで良いFairnessを達成しているものであり，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ううまく制御できず，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．

これらより，TCE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．

なお，一般的にFairnessを改善することで，全プロセス合計の単位時間あたりに実行できる命令数，すなわちトータルスループットを向上できると言われており[2]，本評価の場合にもそれが期待できる．そこで，トータルスループットに関しても評価を行なった．図7にORGに対するTCEのトータルスループットの相対値を示す．図7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．本稿ではTCEをFairness改善に応用したが，アプリケーションによってはトータルスループット向上にも効果があり，Fairnessを向上させることの重要性が伺える．

以上の結果から，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．

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．提

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[13),14)]．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[15)]．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[16)] で述べられている．

本稿で提案する TCE 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．

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に，トラクションコントロール実行を提案した．TCE は，各プロセスでの競合による性能への影響をモデリング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 向上に応用した結果，Fairness を大きく改善できることがわかった．

今後の課題としては，トータルスループット向上など，他の最適化のためのポリシーへ TCE を応用する他，4 コア上でのアルゴリズムの開発や評価を行なうことがあげられる．

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 PACT2006*, 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.Snavely and D.M.Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor", *In Proc. ASPLOS IX*, pp.234–244, Nov. 2000.
14) K.Luo, et al. "Balancing Throughput and Fairness in SMT Processors", *In Proc. ISPASS 2001*, pp.164–171, Nov. 2001.
15) R.Gabor, et al., "Fairness and Throughput in Swith on Event Multithreading", *In Proc. 39th MICRO*, pp.149–160, Dec. 2006.
16) T. Moseley, et al. "Methods for Modeling Resource Contention on Simultaneous Multithreading Processors", *In Proc. ICCD2005*, pp.373–380, Oct. 2005.