修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院情報システム学研究科　情報システム基盤学専攻　博士前期課程		
氏　　　名	久米　正人	学籍番号	0853009
論 文 題 目	共有資源の競合に着目した CMP 向け実行フェーズスケジューリング手法の研究		

要　　旨

近年主流となっているプロセッサのアーキテクチャの 1 つに，複数の CPU コアを 1 チップに搭載したチップマルチプロセッサ（CMP）がある．CMP に搭載された複数のコアで並列処理あるいは並行処理を行うことにより，クロック周波数の向上に頼らず高い演算性能を達成することができる．

CMP では，キャッシュやバスなどの資源は複数のコアで有効に活用するため共有されることが多い．しかし，各コアが同時にそれらの共有資源へアクセスした場合，競合が生じる．共有資源に対するコア間の競合が生じると，トータルスループットや，各コアの性能低下の公平さ（Fairness）が低下するなどの問題が発生し，CMP が本来持つ演算性能が発揮されなくなる．特に，現在ではプロセッサ―主記憶間の性能の格差が拡大するメモリウォール問題が深刻化しているため，メモリアクセスに関連する資源において競合が生じた場合には CMP の性能低下は著しい．そのため，できる限り共有資源の競合を回避することが課題となる．

本論文では，プロセスよりもさらに細粒度な実行フェーズ単位でプロセススケジューリングを行い共有資源の競合を緩和させる手法を提案する．本提案手法は，実行フェーズ単位で各プロセスの共有資源の要求率を予測し，各コアで共有資源の要求率のバランスが取れたプロセススケジューリングを可能にする．これにより競合が緩和させることができると考えられる．本提案手法により，トータルスループットおよび Fairness の向上を目指す．

提案手法を CMP を搭載した 2 種類の実機を用いて評価したところ，2 つのコアを用いる場合には，提案手法を用いない通常のスケジューラに対して Fairness を改善でき，またトータルスループットが向上する例も見られたが，4 つのコアを用いる場合には，Fairness やトータルスループットが逆に低下してしまう例も多く見られた．これは，4 つのコア全てを用いるとスケジューリングを工夫しても共有資源の競合が発生してしまうことなどが原因と考えられる．

修士論文

共有資源の競合に着目したCMP向け実行フェーズスケジューリング手法の研究

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻

0853009 久米 正人

指導教員 近藤 正章 准教授
本多 弘樹 教授
星 守 教授

2010年 1月 28日 提出

目 次

1 はじめに 1

1.1 研究の背景 1

1.2 研究の目的 2

1.3 本論文の構成 3

2 CMP における共有資源の競合の影響 5

2.1 CMP のアーキテクチャ 5

2.1.1 デュアルコア CMP 5

2.1.2 クアッドコア CMP 7

2.2 共有資源の競合による性能低下 8

2.3 競合の程度を表す指標 10

3 実行フェーズスケジューリング手法の提案 12

3.1 従来のプロセススケジューラ 12

3.2 提案手法 13

3.2.1 プロセスの制御手法 13

3.2.2 提案手法の概要 14

3.2.3 提案手法のスケジューリングアルゴリズム 16

3.2.4 スケジューラの具体的な動作 16

3.3 実装 18

3.3.1 スケジューラの実装手法 18

3.3.2 アフィニティ設定 20

4 評価 22

4.1 評価環境 22

4.2 評価のための指標 22

4.3 2 コアを用いる場合の評価 26

4.3.1 各ベンチマークの組み合わせでの評価結果 26

4.3.2 N を変化させた場合の評価 26

4.3.3 I を変化させた場合の評価 29

4.3.4 カウンタごとの評価 32

4.4 4 コアを用いる場合の評価 37

4.4.1 ベンチマークの組み合わせでの評価結果 37

4.4.2 N を変化させた場合の評価 37
4.4.3 I を変化させた場合の評価 40
4.4.4 カウンタごとの評価 45
4.5 結果のまとめ 45

5 考察 48

6 関連研究 50

7 おわりに 52
7.1 まとめ 52
7.2 今後の課題 52

謝辞 54

参考文献 55

図 目 次

1 プロセッサ–主記憶間の性能差 2
2 CMP の概要 6
3 デュアルコア CMP のアーキテクチャの例 6
4 クアッドコア CMP のアーキテクチャの例 7
5 プログラム a のソースコード 9
6 各ケースの模式図 10
7 スケジューラ内の RUN キュー 13
8 プロセスの状態遷移 14
9 実行フェーズ 14
10 2 コア 4 プロセスにおける提案手法の概要 15
11 2 コア 4 プロセスにおける実行プロセス選択の概要 15
12 2 コアで各コアの RUN キューに 4 プロセス存在する場合の例 16
13 スケジューラの動作 19
14 Manipulator が子プロセスにシグナルを送る例 20
15 2 コアに 4 プロセスがアフィニティ設定されている例 21
16 各コアの利用方法 24
17 2 コア，ベンチマークの組み合わせごとの Fairness 評価 27
18 2 コア，ベンチマークの組み合わせごとのトータルスループット評価 . 28
19 2 コア，N の変化に関する Fairness 評価 30
20 2 コア，N の変化に関するトータルスループット評価 31
21 2 コア，I の変化に関する Fairness 評価 33
22 2 コア，I の変化に関するトータルスループット評価 34
23 2 コア，カウンタごとの Fairness 評価 35
24 2 コア，カウンタごとのトータルスループット評価 36
25 4 コア，ベンチマークの組み合わせごとの Fairness 評価 38
26 4 コア，ベンチマークの組み合わせごとのトータルスループット評価 . 39
27 4 コア，N の変化に関する Fairness 評価 41
28 4 コア，N の変化に関するトータルスループット評価 42
29 4 コア，I の変化に関する Fairness 評価 43
30 4 コア，I の変化に関するトータルスループット評価 44
31 4 コア，カウンタごとの Fairness 評価 46
32 4 コア，カウンタごとのトータルスループット評価 47

33 不適切なアフィニティ設定をマイグレーションにより解決する例 . . . 50

表 目 次

1 計測結果 (Intel Core 2 Quad Q8400) 9
2 評価に用いたマシンの仕様 23
3 評価環境 23
4 カウンタ名 23
5 2 コア時におけるベンチマークの組み合わせ 24
6 4 コア時におけるベンチマークの組み合わせ 25
7 2 つのコアを用いる場合の結果のまとめ 48
8 4 つのコアを用いる場合の結果のまとめ 48

1 はじめに

1.1 研究の背景

プロセッサのクロック周波数を高くするとプロセッサが高速で動作するようになる．結果としてプロセッサは高性能なものとなる．クロック周波数を高くするための主な方法には，半導体の製造プロセス微細化やプロセッサ内部の命令パイプライン細分化がある．

半導体の製造プロセスを微細化することにより，トランジスタのゲート長が短くなり，トランジスタのスイッチング遅延が小さくなる．そのためトランジスタの動作速度が向上する．結果としてクロック周波数を向上することができる．

プロセッサ内部の命令パイプラインを細分化することによりパイプラインステージ中の遅延時間が短くなり，プロセッサのクロック周波数を高くすることが可能となる．2000 年代半ばまでは，各プロセッサベンダはこの方法を推し進めることによりプロセッサを高性能化していった．命令パイプラインを細分化することにより高性能化を目指したマイクロアーキテクチャの中では，特に米国 Intel 社の開発した NetBurst マイクロアーキテクチャ[9] が有名である．NetBurst マイクロアーキテクチャを採用したプロセッサには Pentium 4 [3] などがある．Pentium 4 は，内部に約 30 段ものパイプラインを搭載したシングルコアプロセッサである．

プロセッサのクロック周波数を高めると，消費電力や発熱量は増加する．以前は，半導体製造プロセスの微細化によりトランジスタの駆動電圧を低くすることで消費電力や発熱量の増加をある程度抑えてきた．

しかし，トランジスタのスレッショルド電圧が低くなるとリーク電流が増大し，それによる発熱や消費電力が無視できなくなる．よって半導体の製造プロセスを微細化してもトランジスタの駆動電圧を低くすることができず消費電力量や発熱量が削減できなくなった．さらに，「消費電力量が 2 倍になっても性能向上は $\sqrt{2}$ 倍しか得られない」というポラックの法則 [13] も予見されるようになり，クロック周波数の向上によるプロセッサの高性能化は非常に困難であることがわかった．

この問題を解決し，さらなる高性能化を実現するために提案され，近年主流となっているプロセッサのアーキテクチャの 1 つに，複数のプロセッサコアを 1 つのチップ上に搭載したチップマルチプロセッサ (Chip Multi-Processor: CMP) がある [2]．CMP に搭載された複数のコアで並列処理あるいは並行処理を行うことによって，クロック周波数の向上に頼らず高い演算性能を達成することができるため，演算性能/消費電力比の優れたプロセッサを設計することができる．

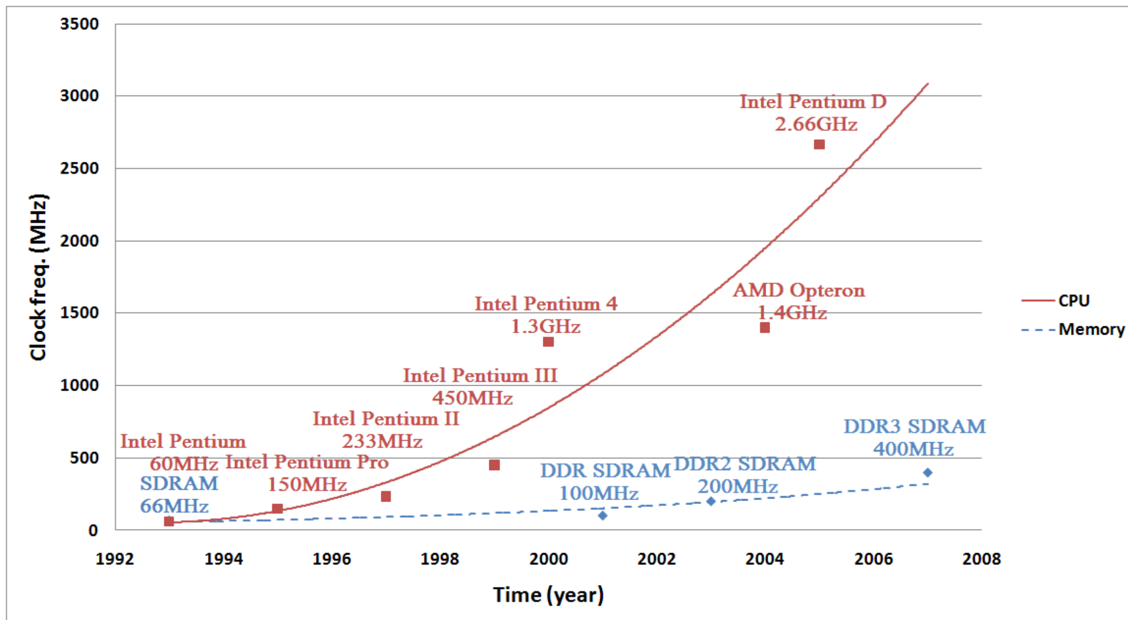

図 1: プロセッサ–主記憶間の性能差

CMP では，さらなる演算性能の向上を目指し，1 つのチップ上に搭載されるコア数は年々増加傾向にある．コアの数が増加すれば，チップ全体の演算性能は向上する．ただし，キャッシュやバスなどの資源は複数のコアで有効に活用するために共有することが一般的であり，多くのコアを搭載した CMP においては，これら共有資源に対するコア間の競合が起こりやすい．共有資源に対するコア間の競合が生じると，チップ全体の性能が低下し，期待される演算性能が十分に得られない．特に，現在では図 1 に示すようにプロセッサ–主記憶間の性能の格差が拡大するメモリウォール問題が深刻化しているため，主記憶のアクセスに関連する競合が生じるとチップ全体の性能低下は著しいものとなる．このため，CMP の持つ演算性能を最大限に引き出すためには，共有資源に対するコア間の競合をできる限り回避することが重要である．

1.2 研究の目的

共有資源の競合が CMP に与える影響は大きい．このことは CMP が実用化される以前から予見されており，マルチプロセッサにおける共有資源の競合を統計的にモデリングする研究が行われている [12]．競合が引き起こす問題には主に以下の 2 つがある．

- **トータルスループットの低下**
  競合により，共有資源へのアクセスが遅延させられるコアが出てくる．このような状況下ではそのコアには十分なスループットでデータが供給されず演算処理がストールし，結果として CMP が本来持つ演算性能が発揮されなくなる．

- ***Fairness*（各コアの性能低下の公平さ）の低下**
  競合による影響を受けやすいプロセスを実行しているコアは，受けにくいプロセスを実行しているコアに比べてスループットが低下し，コア間のスループット低下の程度の格差が大きくなる．OS のスケジューラは競合による影響は全てのコアで一様であると仮定しているため，競合によってスループット低下の程度の格差が生じている状態においては，OS が優先度の高いプロセスに長いタイムスライスを割り当ててもスループットが上がらず優先度が逆転することも起こり得る．Fairness 低下による問題については，文献 [7, 8] において議論されている．

共有資源の競合により CMP の性能が著しく低下してしまうことは，現在においても依然として大きな問題であり，共有資源の競合に関する研究は近年でも多く行われている．例えば，トータルスループットや Fairness を保つためにそれぞれのコアの電源電圧や周波数を動的に制御する手法 [8, 20]，プロセスの実行スピードを制御する手法 [18, 19] がある．ただし，これらの手法は全てプロセスを単位として何らかの制御を行うものである．

本論文では，1 つのプロセス内でも共有資源の競合の程度は変化することを考慮し，競合の程度の変化へより細かく対応するためにプロセスよりもさらに細粒度な実行フェーズ単位でスケジューリングを行うことで共有資源の競合を緩和させる手法を提案する．本提案手法は，各プロセスを単位時間ごとに分割した実行フェーズ単位で共有資源の要求率を予測し，それに従い動的にプロセスのスケジューリングを行う．要求率を予測し，共有資源のバンド幅を超えないようなスケジューリングを行うことで共有資源の競合を緩和させることができると考えられる．提案手法により，トータルスループットおよび Fairness の向上を目指す．なお，本研究においては共有資源は各コアで共有となる共有キャッシュや，メモリウォール問題の重大性からメモリバスに焦点を当てることとする．

1.3 本論文の構成

本論文の構成を以下に示す．

第 2 章で CMP における共有資源の競合の影響について述べる．第 3 章で本研究で提案する実行フェーズスケジューリング手法の詳細について述べる．第 4 章で提案手法を評価するための環境，仮定および評価結果を示す．第 6 章で関連研究について記し，第 7 章でまとめや今後の課題について述べる．

2　CMP における共有資源の競合の影響

2.1　CMP のアーキテクチャ

図 2 は L2 キャッシュが 2 つのコアで共有されている CMP の例である．Intel Core 2 Duo などはこの構造をとっている．このように，プロセッサから最も離れたキャッシュであるラストレベルキャッシュ（Intel Core 2 Duo における L2 キャッシュ，AMD Phenom における L3 キャッシュなど）は全てのコアで共有されることが多い．

複数のプロセッサコアが搭載されることによりチップ全体の演算性能はシングルコアのそれに比べて高いものとなる．ただし全てのコアの実効性能を実際に維持するためには，データ供給を担うバスなどに増えたコアが要求する分のバンド幅がさらに必要とされる．しかし実際には種々の理由によりそれが満たされず，バンド幅が不足して競合が生じ，性能面において様々な悪影響を受ける．

実際の商用プロセッサは，様々なアーキテクチャをとっている．本節では一般的な CMP のアーキテクチャについて紹介する．

2.1.1　デュアルコア CMP

消費電力量を削減したモバイル用途向けシングルコア CPU（Intel Atom など）を除き，現在市場に流通している商用プロセッサのほとんどは，デュアルコア CMP など，複数コアを搭載するアーキテクチャをとっている．図 3 は，2 つのコアを搭載したデュアルコア CMP の一般的なアーキテクチャを図で示したものである．図に示されるキャッシュは全てラストレベルキャッシュである．

図 3(a) は，各コアでラストレベルキャッシュを個別に搭載した CMP を表している．この構造の CMP では，ラストレベルキャッシュへ各コアが自由にアクセスできるため，キャッシュの階層では競合は起こらない．しかし，あるコアのラストレベルキャッシュへ一方のコアはアクセスできないため，メモリ上の同じアドレスの同一データをこれら 2 つのキャッシュで保持することがあり，効率的でない．さらにはキャッシュのコヒーレンシを保つためのコストが大きくなると考えられる．この種のアーキテクチャでは，各コアで共有するメモリバス以下の階層において競合が生じる．NetBurst マイクロアーキテクチャを採用した Intel Xeon の一部などがこの種のアーキテクチャをとる CMP として挙げられる [9]．これらの CMP は，L1 キャッシュ，L2 キャッシュ双方とも各コアごとに個別に搭載され，共有キャッシュは持たない．

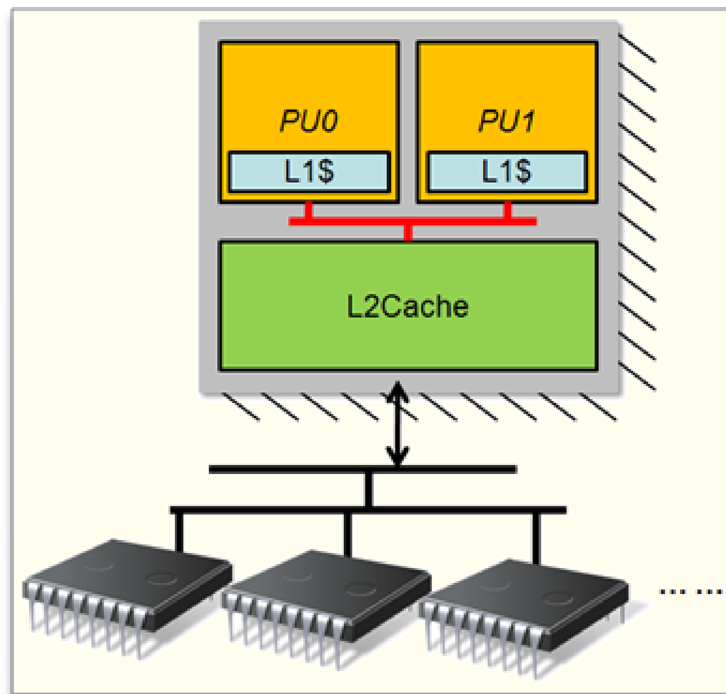

図 2: CMP の概要

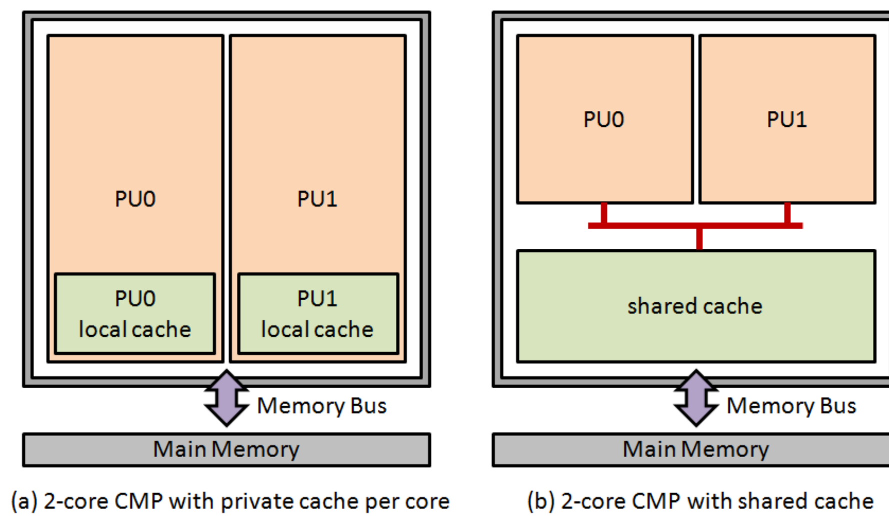

図 3: デュアルコア CMP のアーキテクチャの例

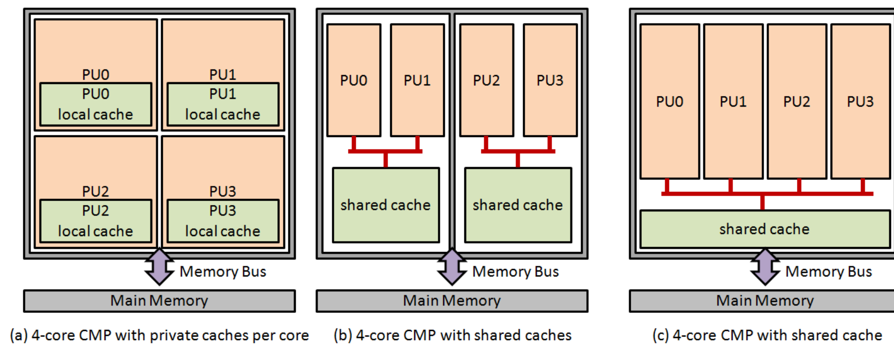

図 4: クアッドコア CMP のアーキテクチャの例

図 3(b) は，ラストレベルキャッシュを 2 つのコアで共有する CMP の例である．この構造の CMP では，ラストレベルキャッシュ以下の階層が共有資源となる．ラストレベルキャッシュが 1 つだけのため，それぞれのコアにラストレベルキャッシュを搭載する場合よりもハードウェアコストが削減される．さらに図 3(a) のアーキテクチャのようにメモリ上の同一データを複数保持する必要がなくなる．L1 キャッシュを各コアで個別に持ち，ラストレベルキャッシュである L2 キャッシュを各コアで共有する構造の CMP には，Intel Core 2 Duo などがある [6]．また，L1，L2 キャッシュを各コアで個別に持ち，ラストレベルキャッシュである L3 キャッシュを各コアで共有する構造の CMP には，K10 マイクロアーキテクチャを採用した AMD Athlon X2 などがある．

2.1.2 クアッドコア CMP

図 4 は，4 つのコアを搭載したクアッドコア CMP の一般的なアーキテクチャを図で示したものである．クアッドコアの CMP は，共有資源に対してコアの数が多くなっていることからデュアルコアの CMP に比べて競合が生じやすくなる．

図 4(a) は，図 3(a) のアーキテクチャのコアをそのまま 4 コアに拡張したようなアーキテクチャとなっている．この構造の CMP では，図 3(a) のアーキテクチャのようにラストレベルキャッシュへ各コアが自由にアクセスできるため，キャッシュの階層では競合は起こらない．しかし，キャッシュの個数も増えているため図 3(a) のアーキテクチャよりも，同一データ保持やコヒーレンシの問題などでさらに非効率になると考えられる．

図 4(b) は，図 3(b) のタイプの CMP が 2 つ，同チップ上に搭載されているような構造であり，2 つのラストレベルキャッシュが 2 つのコアごとに搭載され共有されている．そのため図 3(b) の構造をとる CMP を製造するためのチップをそのまま組み

合わせて異なる種類の CMP を製造できることから，プロセッサを開発するベンダにとっては異なる CMP のために異なるアーキテクチャを設計するコストが削減できる．ただしユーザの視点からみると，ラストレベルキャッシュを多く要求するようなプログラムが複数存在する場合，ラストレベルキャッシュを共有するコア同士で実行されてしまうと性能が低下するなどの問題も残る．Intel Core 2 Quad はこのアーキテクチャである [6]．

図 4(c) は，ラストレベルキャッシュを全コアで共有する CMP の例である．このアーキテクチャは 2009 年現在で最新の CMP に多くみられる．AMD Opteron，Intel Core i7 などがこれにあたる [5, 1]．これらの CMP は L1 キャッシュ，L2 キャッシュを各コアで個別に持ち，ラストレベルキャッシュとして L3 キャッシュを全コアで共有する．

いずれのアーキテクチャをとる場合においても，少なくともメモリバス以下の階層は全てのコアで共有される．よって主記憶へのアクセスに関連する競合が生じる可能性が高い．しかし，メモリウォール問題が深刻化している現在では，メモリバスや主記憶の競合によるペナルティが非常に大きくなる．つまり，CMP では競合によって性能大きく左右される．

2.2 共有資源の競合による性能低下

共有資源の競合による性能低下を調べるため，*Intel Core 2 Quad Q8400* を搭載したマシンを用いた計測を行った．このマシンに搭載されたメモリは DDR2-800（シングルチャンネル，2 GB）である．Intel Core 2 Quad Q8400 は，2 つのコアごとに L2 キャッシュを共有するアーキテクチャとなっているため，使用するコアの選択の仕方により L2 キャッシュを共有する場合と共有しない場合について計測を行うことが可能である．本計測ではメモリウォール問題を考慮し，メモリバス以下のメモリ階層における競合による性能低下を調べるために L2 キャッシュを共有しない 2 コアを用いる．

キャッシュミス時に共有資源である主記憶へのアクセスが発生することから，キャッシュヒット部とキャッシュミス部を交互に実行するプログラムを作成し，実行完了までの時間を計測した．キャッシュヒット部とキャッシュミス部はそれぞれ 10 秒程度になるよう設定し，それぞれが 10 回ずつ繰り返される．そのプログラムをプログラム a として図 5 に示す．また，ミス部とヒット部の順番を逆に，すなわちプログラム a の 3〜6 行目と 7〜10 行目を交換したものをプログラム b として用意する．さらに，ヒット部，ミス部を実行するのにかかる時間を同じになるようにした．

program_a.c（細部省略）

```
1   char a = 0, *U;
2   U = (char *)malloc(268435456 * sizeof(char));
3   for(k = 0; k < 10; k++){
4       for(i = 0; i < 422; i++)
5           for(j = 0; j < 268435456; j += 128)
6               a += U[j];
7       for(m = 0; m < 50; m++)
8           for(i = 0; i < 536870912; i++)
9               for(j = 0; j < 1073741824; j++)
10                  a += 1;
11  }
12  free(U);
```

図 5: プログラム a のソースコード

表 1: 計測結果 (Intel Core 2 Quad Q8400)

	実行時間 (秒)	実行時間/ケース C の実行時間
ケース A	301.01	1.49
ケース B	206.28	1.02
ケース C	201.69	-

- **競合が生じるケース (ケース** A):
  プログラム a を 2 個それぞれのコアで同時に実行
- **競合が生じないケース (ケース** B):
  プログラム a とプログラム b を同時に実行
- **競合のないケース (ケース** C):
  プログラム a を単独で実行

上記 3 種類のケース A，B，C を図示したものを図 6 に示す．これらのケース A，B，C で実行時間を計測した．なお，ケース A，B においては後に終了するプログラムまでの実行時間とした．計測結果を表 1 に示す．

競合が生じないケース B であれば，単独実行時と比べて約 1.02 倍程度の実行時間となったが，競合が生じるケース A では約 1.49 倍となり，ケース A，B では結果的

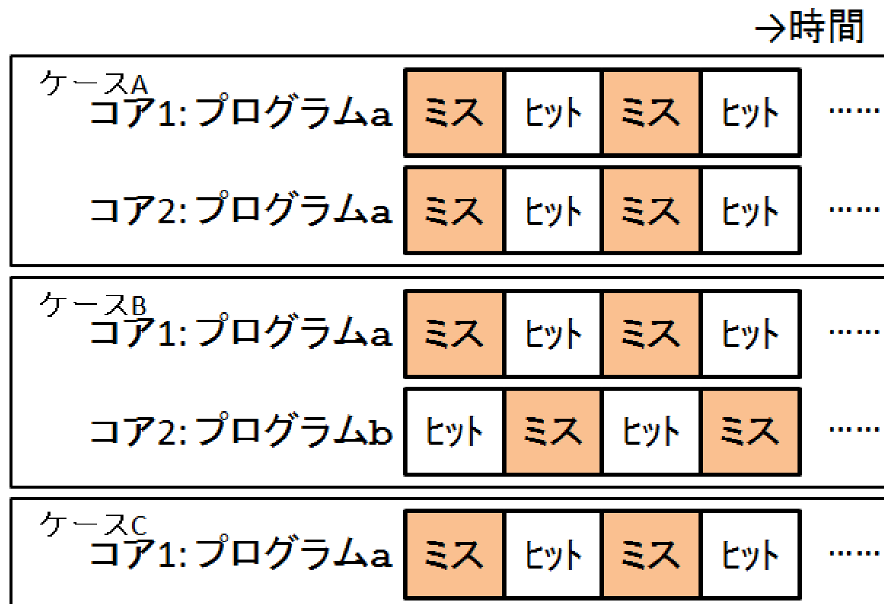

図 6: 各ケースの模式図

に同じ量の演算処理を行ったのにもかかわらず，競合により大きく実行時間が延び性能が著しく低下した．

また，文献 [21] において，様々な 2 つのプログラムを同時に実行させ性能を測定した例が示されている．この文献によると，キャッシュミス率が高いベンチマーク同士を 2 つのコアで同時に実行させると，逐次的に 2 つのベンチマークを実行する場合よりも性能が低くなる場合があることが示されている．

これらの事象より，多くのコアを持つ CMP においては共有資源の競合が生じやすく，競合による影響は非常に深刻であり，それに伴い CMP の持つ性能は著しく低下してしまうということがいえる．

2.3　競合の程度を表す指標

本提案手法の実現にあたり，どの程度競合が生じているかを定量的に表す指標が必要となる．各プロセスの性能を低下させる要因は共有資源の競合であると仮定し，この節で競合の程度を表す指標となる性能比 (Performance Ratio: PR)[19] を採用する．

プロセッサの性能を表す単位として IPC (Instructions Per Cycle) がよく用いられる．これは，1 サイクルあたりの実行命令数である．あるプロセス a の IPC $IPC(\mathrm{a})$ は，プロセス a の総実行命令数を $Inst(\mathrm{a})$，実行に要した総サイクル数を $Cyc(\mathrm{a})$ と

して，式 4 で求めることができる．

$$IPC(\mathtt{a}) = \frac{Inst(\mathtt{a})}{Cyc(\mathtt{a})} \tag{1}$$

この指標を用いてプロセッサの絶対的な性能を表すことができる．しかし，この指標では競合によってどの程度の性能低下が起こったかを知ることはできない．競合なくプロセスを実行させたときの性能に対する，競合が生じている状態での性能を相対的に把握する必要となる．そのため，IPC を基にした新たな指標 PR を用いる．PR は，プロセスの単独実行時に対する複数プロセス実行時の性能低下率である．複数プロセスを同時に実行すると，単独実行時に比べて性能が低下する場合があるが，これは複数のプロセスが資源を共有し，競合を生じさせているためである．つまり競合が生じない場合に対する競合が生じた場合のプロセスの性能低下率である．よって PR は，競合の程度を表す指標となり得る．あるプロセス a を単独実行したときの IPC を $IPC_s(\mathtt{a})$，他のプロセスと複数で同時に実行したときのプロセス a の IPC を $IPC_m(\mathtt{a})$ とし，プロセス a の PR $PR(\mathtt{a})$ $(0 \leq PR(\mathtt{a}) \leq 1)$ は式 2 で表せる．

$$PR(\mathtt{a}) = \frac{IPC_m(\mathtt{a})}{IPC_s(\mathtt{a})} \tag{2}$$

3　実行フェーズスケジューリング手法の提案

本章では，本研究で提案する実行フェーズスケジューリング手法について述べる．本提案手法は，プロセスを単位時間ごとに分割した実行フェーズごとの共有資源の競合の程度に着目し，競合を回避するよう各プロセスをスケジューリングすることで，性能の向上を狙うものである．なお本提案手法では，CMP のコア数よりもプロセスの数が多い状況を前提とする．多くのサーバアプリケーションを動かす必要のあるサーバマシンなどがそれにあたる．この場合はユーザへのサービスの提供に不公平が生じる状況が好ましくないため，Fairness の維持も重要である．本提案手法は，Fairness の向上も目指しているために有用であると考えられる．

3.1　従来のプロセススケジューラ

本節では，提案するスケジューラが前提とする従来の OS のプロセススケジューラについて説明する．

図 7 に示すように，Linux などで用いられている一般的なプロセススケジューラは RUN キューと呼ばれる実行可能プロセスのリストを持つ．スケジューラは，RUN キュー内のプロセスの中から実行するプロセスを選択する．実行するプロセスを選択する方法は，アルゴリズムによって異なる．広く用いられているスケジューリングアルゴリズムには以下のものがある．

優先度を用いたアルゴリズム

各プロセスに優先度を付け，その優先度に基づいてスケジューリングするアルゴリズムである．スケジューラは，RUN キュー内から優先度の高いプロセスを選択し，実行させる．そのプロセスの実行が終了したのち，次に優先度の高いプロセスを選択し実行させることを繰り返す．このアルゴリズムにおいては，優先度の低いプロセスがコアに割り当てられず実行されなくなるスタベーションが起こる．スタベーションを起こさないように改良された優先度スケジューリングアルゴリズムが Linux カーネルのスケジューラで採用されている．

ラウンドロビンスケジューリング

ラウンドロビンスケジューリングは，実行プロセスを一定時間ごとに交換するスケジューリングアルゴリズムである．このアルゴリズムで実装されたスケジューラ

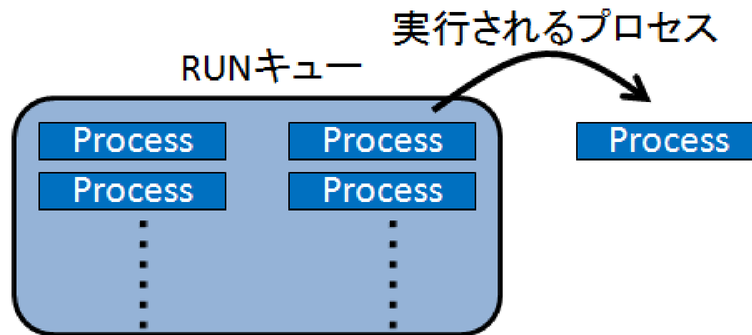

図 7: スケジューラ内の RUN キュー

は，まず RUN キューからプロセスを取り出し，実行させる．一定時間が経過したのち，実行中のプロセスを一旦停止させ，RUN キューの最後に戻す．その後また RUN キューからプロセスを取り出し，一定時間実行させ RUN キューの最後に戻すことを繰り返す．このアルゴリズムでは，一定時間で次々に実行プロセスを切り替えるため全てのプロセスが平等に実行されスタベーションは起こらない．

3.2 提案手法

3.2.1 プロセスの制御手法

プロセスの制御には，OS のプロセススケジューリングに基づく手法を利用する場合と，プロセススケジューリングを行うプロセスを持つ場合の 2 通りが考えられるが，OS のプロセススケジューリングに基づく手法を利用する場合には直接的に実行プロセスを制御できるためオーバヘッドが少なくなるという利点がある．また，OS 上での制御のためハードウェアによる制約を受けず，余計なハードウェアコストなしに実現できる．そのため本研究では，OS のプロセススケジューリングに基づく手法を用いてプロセスの制御を行う．

OS のスケジューラによって RUN キュー上に存在する各プロセスを Ready 状態からディスパッチによって Running 状態へ，Running 状態からプリエンプションによって Ready 状態へと繰り返し遷移させることにより実行するプロセスを選択することができる．この様子を図 8 に示す．

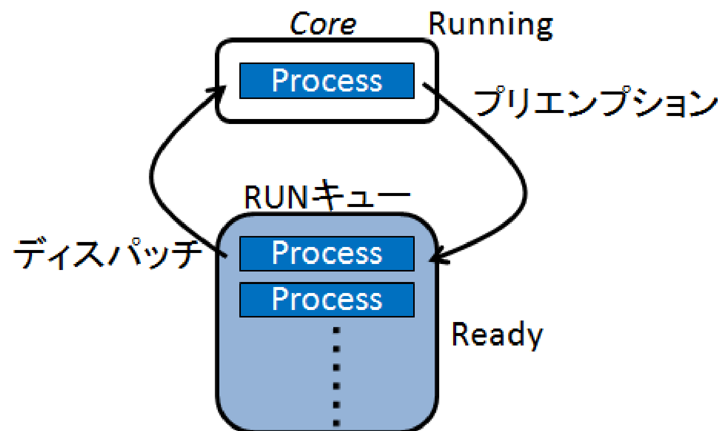

図 8: プロセスの状態遷移

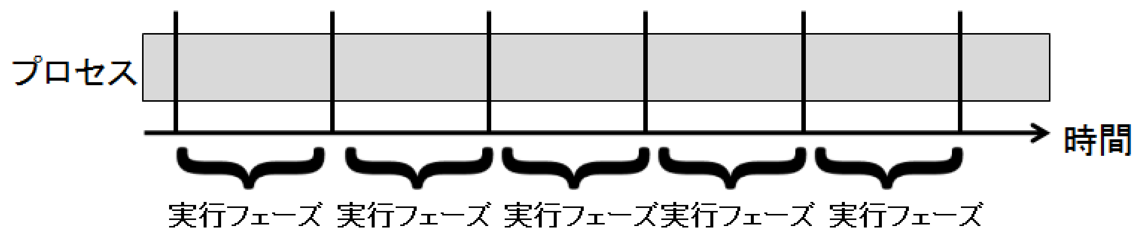

図 9: 実行フェーズ

3.2.2　提案手法の概要

本研究で提案する実行フェーズスケジューリング手法は，各プロセスの共有資源の要求率を予測し，あるコアでは要求率の大きいプロセス，他のコアでは要求率の小さいプロセスを選択することで，要求率をバランスさせる．これによりトータルスループットおよび Fairness の向上を目的とするものである．ここで実行フェーズとは，プロセスを単位時間ごとに分割した単位である．図 9 にその概略を示す．

図 10 に実行フェーズスケジューリング手法の概要を示す．本提案手法が組み込まれた OS 上のスケジューラが，ある一定間隔 (PollingInterval) ごとにその時点でのパフォーマンスカウンタのデータを取得し，それに基づいて RUN キュー内の各プロセスの共有資源の要求率を予測する．このスケジューラには，パフォーマンスカウンタから要求率を予測する計算式があらかじめ組み込まれており，予測が可能となる．

次に，その予測値により次の実行フェーズでの実行プロセスを RUN キューより選択する．図 11 に 2 コアの CMP 上で 4 プロセスをスケジューリングする際におけ

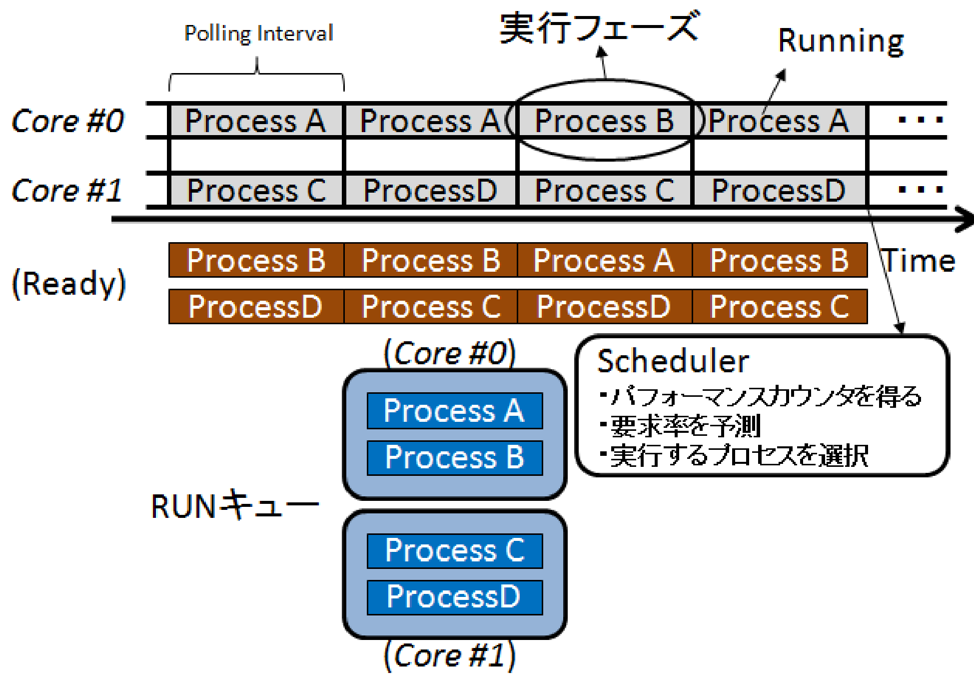

図 10: 2 コア 4 プロセスにおける提案手法の概要

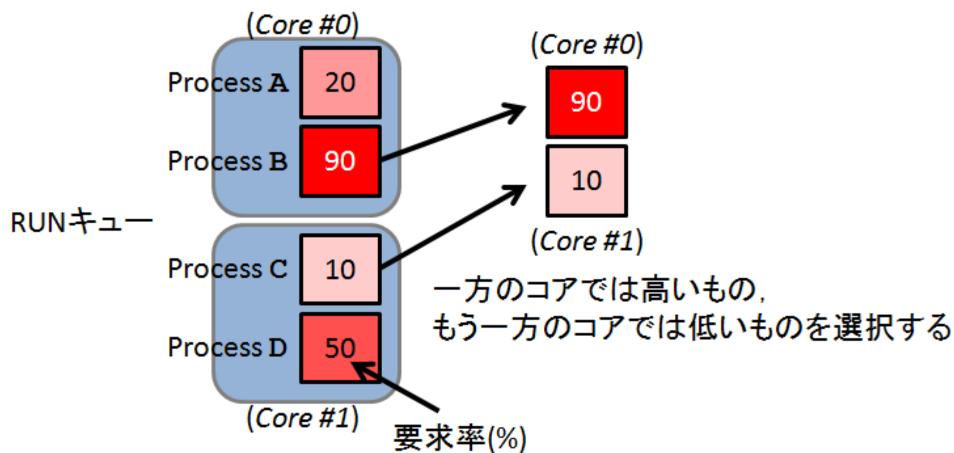

図 11: 2 コア 4 プロセスにおける実行プロセス選択の概要

る，実行プロセスを選択する過程の概要を示す．プロセス選択のアルゴリズムについては第 3.2.3 節で詳しく述べる．

さらに次の実行フェーズでも，要求率の予測，実行プロセスの選択を行う．これを繰り返し行い，提案手法を実現する．

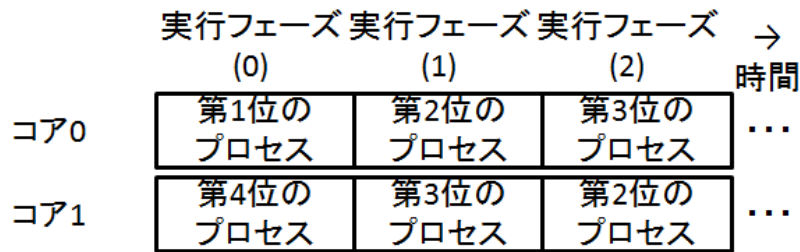

図 12: 2 コアで各コアの RUN キューに 4 プロセス存在する場合の例

3.2.3 提案手法のスケジューリングアルゴリズム

本提案手法のスケジューリングアルゴリズムをアルゴリズム 1 に示す．ここで，NPU 個のコアが存在するものとし，各コアの名前をそれぞれコア 0，コア 1，⋯，コア $NPU-1$ とする．

まず，スケジューラは GetPerfCounter() によりそれぞれのプロセスのパフォーマンスカウンタ値を取得し，Predict() で最小二乗法によるパフォーマンスカウンタ値の予測を行う．

次に，この予測値に従い，次の実行フェーズで動かすプロセスを決定する．コア 0，2，4，⋯では予測値の最も大きいもの，コア 1，3，5，⋯では予測値の最も小さいものが選ばれる．その次の実行フェーズでは，コア 0，2，4，⋯では 2 番目に大きいもの，コア 1，3，5，⋯では 2 番目に小さいもの，というようにそれぞれ互いに逆順にローテーション方式で実行するプロセスを選択する．図 12 に，2 コアで各コアに 4 つのプロセスが RUN キューに存在する場合の例を示す．つまりこれは合計で 8 プロセスをスケジューリングする場合の例である．この場合，n 番目の実行フェーズ（$n = 0, 1, 2, \cdots$）ではコア 0 で第 $((n \mod 4) + 1)$ 位，コア 1 で第 $(5 - ((n \mod 4) + 1))$ 位のプロセスが選択される．

3.2.4 スケジューラの具体的な動作

2 つのコアを用い，8 つのプロセスをスケジューリングする場合を考える．コア 0 の RUN キューに A, B, C, D，コア 1 の RUN キューに E, F, G, H が存在する場合の，スケジューラの動作を具体的に示し，概略を図 13 に示す．

1. スケジューラを動かすと，コア 0 では A, B, C, D，コア 1 では E, F, G, H が実行される．スケジューラは計 N 回，I 秒 (PollingInterval) ごとにそれぞれのプログラムのカウンタ値を読む．

アルゴリズム 1: 提案手法のスケジューリングアルゴリズム

NPU: 全コア数
$P_{i,j}$: コア i の RUN キューにあるプロセス j
NPR_i: コア i の RUN キューにあるプロセス数
$PC_{i,j}$: $P_{i,j}$ のパフォーマンスカウンタ値
$PH_{i,j}$: $PC_{i,j}$ の履歴
$PV_{i,j}$: $P_{i,j}$ のパフォーマンスカウンタの予測値
RP_i: コア i において次の実行フェーズで動かすプロセス

GetPerfCounter($P_{i,j}$): $P_{i,j}$ のパフォーマンスカウンタ値を取得する
Predict($PC_{i,j}, PH_{i,j}$): カウンタ $PC_{i,j}$ とカウンタ履歴 $PH_{i,j}$ からカウンタ値を予測する
ProcessRank$\{R, A_0, \cdots, A_n\}$: 予測値 $A_0, \cdots, A_n$ から第 $R+1$ 位のプロセスを選択する
Stop(A): プロセス A を止める
Run(A): プロセス A を動かす

$n = 0$
for each end of PollingInterval **do**
  Stop($P_{0,0}, \cdots, P_{0,NPR_0-1}, P_{1,0}, \cdots, P_{1,NPR_1-1}, \cdots, P_{NPU-1,0}, \cdots, P_{NPU-1,NPR_{NPU-1}-1}$)
  for $i = 0$ to $NPU-1$ **do**
    for $j = 0$ to $NPR_i - 1$ **do**
      $PC_{i,j}$ = GetPerfCounter($P_{i,j}$)
      $PV_{i,j}$ = Predict($PC_{i,j}, PH_{i,j}$)
    end for
  end for
  for $i = 0$ to $NPU-1$ **do**
    if $i \mod 2 = 0$ **then**
      RP_i = ProcessRank$\{(n \mod NPR_i), PV_{i,0}, \cdots, PV_{i,NPR_i-1}\}$
    else
      RP_i = ProcessRank$\{(NPR_i - 1 - (n \mod NPR_i)), PV_{i,0}, \cdots, PV_{i,NPR_i-1}\}$
    end if
  end for
  Run($RP_0, RP_1, \cdots, RP_{NPR-1}$)
  $n = n + 1$
end for

2. スケジューラは読んだカウンタ値から最小二乗法を用いて次の実行フェーズでのカウンタ値を予測する．

3. コア 0 で実行されている A, B, C, D について，予測値が第 1 位のもの（たとえば B）を選択しコア 0 に割り当てる．コア 1 で実行されている E, F, G, H について，予測値が第 4 位のもの（たとえば H）を選択しコア 1 に割り当てる．その他のプロセス（A, C, D, E, F, G）は停止させる．

4. I 秒経過し，スケジューラはそれぞれのプロセスのカウンタ値を読む．そのカウンタ値を含む過去 N 個のカウンタ値から最小二乗法を用いて次の実行フェーズでのカウンタ値を予測する．

5. コア 0 の RUN キュー内に存在する A, B, C, D について，予測値が第 2 位のもの（たとえば D）を選択しコア 0 に割り当てる．コア 1 の RUN キュー内に存在する E, F, G, H について，予測値が第 3 位のもの（たとえば E）を選択しコア 1 に割り当てる．その他のプロセス（A, B, C, F, G, H）は停止させる．

6. 次のフェーズでコア 0 に関しては第 3 位，コア 1 に関しては第 2 位を選択する．その次のフェーズではコア 0 に関しては第 4 位，コア 1 に関しては第 1 位を選択する．選択はこの順番で繰り返す．

3.3 実装

3.3.1 スケジューラの実装手法

実際に OS のスケジューラを拡張して提案手法のスケジューラを組み込むことは，OS の実装に関する専門的な知識が必要とされるため容易ではない．そこで本研究では，提案手法を簡便に実装するためにユーザプロセスよりスケジューリングを実現する仮想プロセススケジューラを実装した．本項では，本提案手法を仮想的に実現するスケジューラをプロセスとして実装する手法について述べる．

スケジューラとしての役割を果たすプロセスを Manipulator と呼ぶこととする．あらかじめ実行するアプリケーションプロセスには，パフォーマンスカウンタ値を Manipulator に通知する関数を，あるシグナル（本実装では SIGUSR2 を用いた）のシグナルハンドラとして定義しておく．アプリケーションプロセスは，Manipulator から fork()，exec() システムコールを用いて子プロセスとして実行される．Manipulator へのパフォーマンスカウンタ値の受け渡しには，共有メモリを用いた．

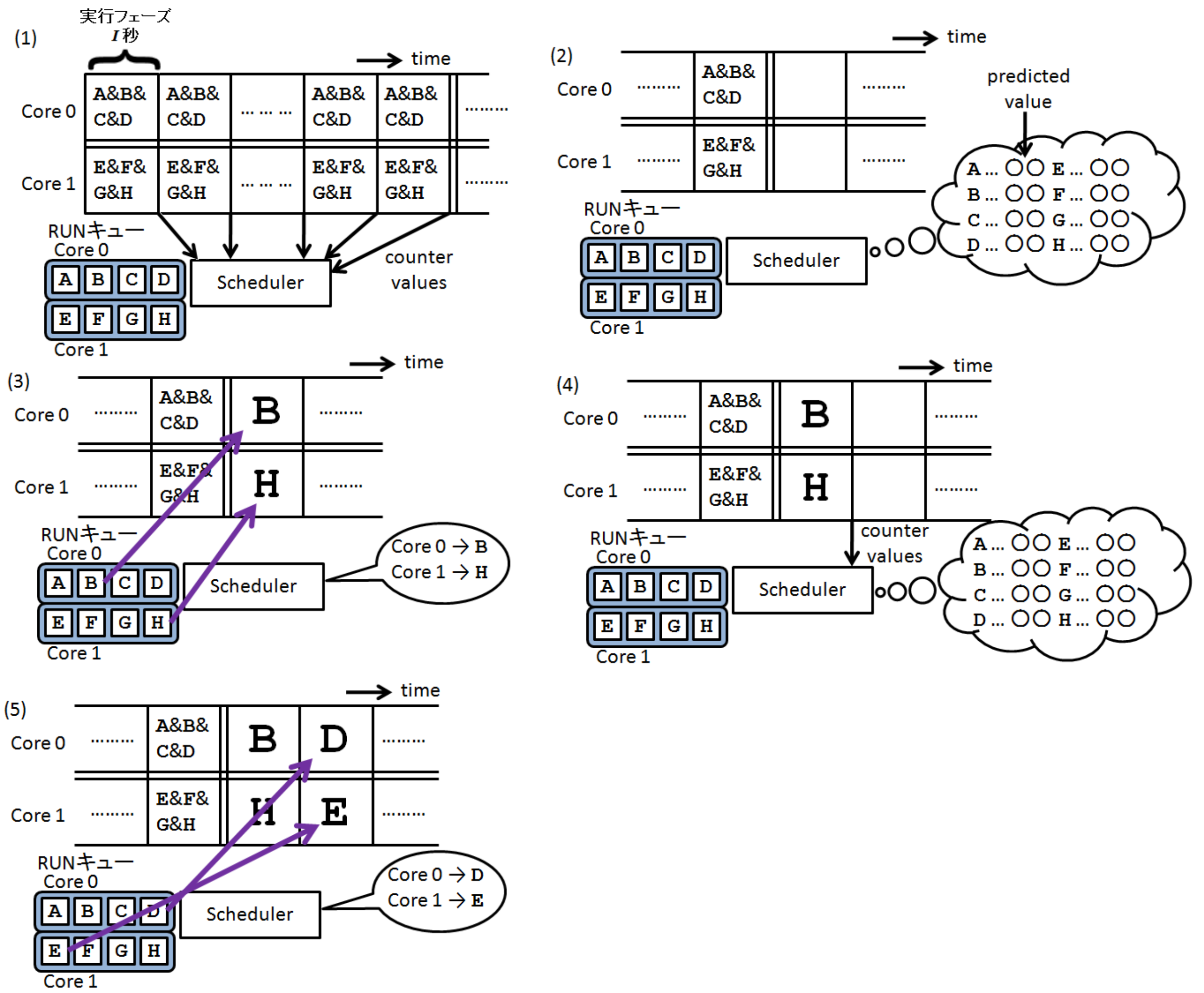

図 13: スケジューラの動作

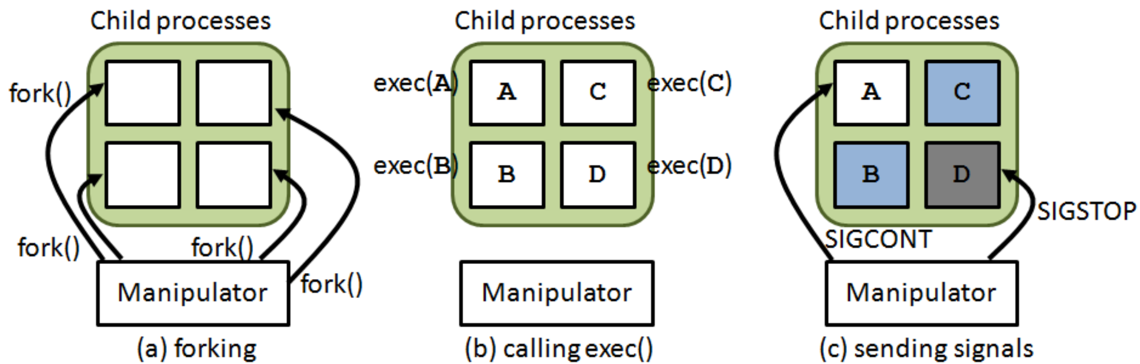

図 14: Manipulator が子プロセスにシグナルを送る例

Manipulator は，I 秒 (PollingInterval) の間 sleep() 関数でスリープした後，子プロセスに対して SIGUSR2 シグナルを送る．子プロセスが SIGUSR2 シグナルを受けることにより，パフォーマンスカウンタ値を共有メモリに書き込む．それを Manipulator が読むことにより，Manipulator に対してのパフォーマンスカウンタ値通知を実現する．この 1 回が 1 つの実行フェーズに相当する．これを N 回繰り返す．

Manipulator は，プロセスごとに N 個のデータを得た後，それらを使って最小二乗法を適用しカウンタ値がどのように変化するかを予測する．

予測を行った後，提案手法のスケジューリングアルゴリズムに基づいて次の実行フェーズで実行させるプロセスを選択する．Manipulator は，選択したプロセスに対して SIGCONT シグナルを送る．これは，RUN キューよりプロセスを選択しコアに割り当てるディスパッチを仮想的に実現するものである．また，それ以外のプロセスには SIGSTOP シグナルを送る．これは，コアに割り当てられていたプロセスを RUN キューへ戻るプリエンプションを仮想的に実現するものである．

これを繰り返すことで本提案手法によるスケジューリングを行う．Manipulator が fork() システムコールで子プロセスを生成した後，子プロセスに対してシグナルを送る様子を図 14 に示す．

3.3.2 アフィニティ設定

あるプロセス a が，コア 0 内の RUN キュー中に存在しているとする．この状態から，a がコア 1 内の RUN キュー中へ移動し，コア 1 で実行されるようになることがある．このように，プロセスが他のコアへ移っていくことをマイグレーションという．これによって 1 つのコアに負荷が集中しないようになっているが，マイグレーションが起こると，プロセスを移動させる分のオーバーヘッドが発生し性能低下が

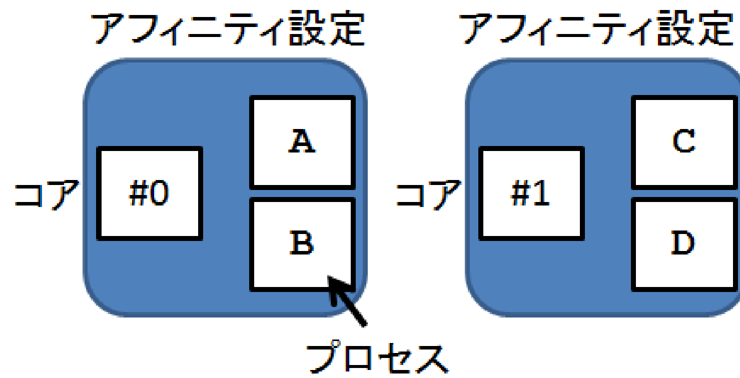

図 15: 2 コアに 4 プロセスがアフィニティ設定されている例

起こる．その性能低下を防ぐために，本研究では各プロセスをコアにアフィニティ設定する．アフィニティ設定されたプロセスは，実行されるコアが固定されるようになる．実行されるプロセスは，各コアに同じ数だけアフィニティ設定されるものとする．図 15 に，2 コアに 4 プロセスがアフィニティ設定された例を示す．コア 0 にプロセス A と B，コア 1 にプロセス C と D というようにそれぞれのコアに 2 つずつプロセスがアフィニティ設定される．

4 評価

4.1 評価環境

本提案手法の効果を調べるため，本論文ではクアッドコア CMP である Intel Core 2 Quad Q8400，Intel Core i7 940 を搭載した 2 種類の実機のマシンを用いて評価を行う．マシンの名前をそれぞれ C2Q，Ci7 とする．それぞれの CMP のアーキテクチャについては第 2.1.2 節で説明した通りである．この 2 つのマシンを用いることにより，資源の共有の仕方が異なる 2 種類の CMP を評価することができる．表 2 に評価に用いたマシンの仕様を示す.

本論文では，デュアルコア環境を再現するために 4 つのコアのうち 2 つのコアを用いる場合と，4 つのコア全てを用いる場合の環境で，第 3.3.1 項に述べた Manipulator を用いる方法により評価を行う．評価環境については表 3 に示す．各コアの利用方法については図 16 に示す.

また，パフォーマンスカウンタ値の取得には，文献 [16] で使用されている perfmon を用いた．本評価においてプロセス実行中に得るカウンタ値は，本提案手法を実現するために必要な情報を得るためのカウンタと，性能測定のための実行命令数とする．

評価では，使用したカウンタや PollingInterval (I)，最小二乗法を適用する際のデータの個数 (N) を変えて評価を行う．使用したカウンタについては表 4 に示す.

2 コア使用時は表 5，4 コア使用時は表 6 に示す 10 種類の SPEC2000 ベンチマークの組み合わせを実行し評価した.

ただし，各ベンチマークの実行時間が異なるため，あるベンチマークが実行終了しても残りのベンチマークは終了せず実行され続ける．そのような状況では同時に実行されているベンチマークの数が減り，正しい評価ができなくなる．そのため，本評価ではベンチマークの実行が終了し次第，すぐに同一ベンチマークを実行し，実行している全ベンチマークが少なくとも 1 回実行が終了するまでの間を評価対象とする．本評価で表す ORG は，提案手法を用いない通常の Linux スケジューラでの結果を表す.

4.2 評価のための指標

本評価では，以下に示す Fairness とトータルスループットの評価指標を用いる．

表 2: 評価に用いたマシンの仕様

Machine name	C2Q	Ci7
CPU	Intel Core 2 Quad Q8400	Intel Core i7 940
CPU frequency	2.66 GHz	2.93 GHz
L2 cache	Shared, 16-Way, 2 MB×2	Private, 8-Way, 256 KB×4
L3 cache	-	Shared, 16-Way, 8 MB
Memory bus	FSB, 1333 MHz (Bandwidth: 10.7 GB/s)	QPI, 4.8 GT/s (Bandwidth: 23.4 GB/s)
M/B	GIGABYTE GA-EG41MF-US2H	Intel DX58SO
Main memory	Single Channel, DDR2-800, 2 GB (Bandwidth: 6.4 GB/s)	
OS	Linux-2.6.28 with perfmon	

表 3: 評価環境

使用コア数	環境名	使用マシン	使用コア
2	C2Q-SHR	C2Q	L2 キャッシュを共有する 2 つのコアを使用
	Ci7-DUAL	Ci7	2 つのコアを使用
4	C2Q-QUAD	C2Q	4 つのコアを使用
	Ci7-QUAD	Ci7	4 つのコアを使用

表 4: カウンタ名

	カウンタ名	詳細	利用可能マシン
1	`L1D_CACHE_LD:MESI`	L1 データキャッシュアクセス	C2Q, Ci7
2	`L2_RQSTS:SELF:I_STATE`	L2 キャッシュミス	C2Q
3	`L2_RQSTS:LD_MISS`	L2 キャッシュミス	Ci7

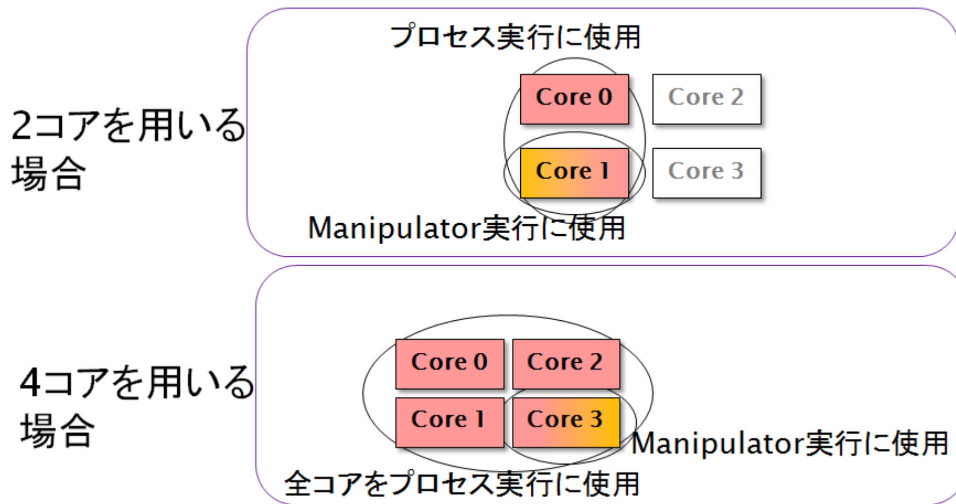

図 16: 各コアの利用方法

表 5: 2 コア時におけるベンチマークの組み合わせ

組み合わせ	Core 0	Core 1
i	bzip2, gzip, art, mesa	twolf, equake, mcf, gap
ii	art, gzip, gzip, twolf	bzip2, gap, mcf, equake
iii	equake, art, mcf, gzip	equake, twolf, bzip2, equake
iv	gap, bzip2, equake, mesa	gap, mcf, gap, twolf
v	gap, twolf, gap, mcf	equake, bzip2, equake, mesa
vi	equake, gap, gzip, gzip	twolf, gzip, equake, gzip
vii	art, gzip, equake, gap	bzip2, gzip, bzip2, gap
viii	mcf, twolf, twolf, art	gap, bzip2, twolf, mcf
ix	mesa, equake, gap, bzip2	bzip2, mesa, mcf, equake
x	mcf, bzip2, gzip, gzip	twolf, mesa, gzip, equake

表 6: 4 コア時におけるベンチマークの組み合わせ

組み合わせ	Core 0	Core 1	Core 2	Core 3
i	bzip2, gzip	art, mesa	twolf, equake	mcf, twolf
ii	art, gzip	gzip, twolf	bzip2, bzip2	mcf, equake
iii	equake, art	mcf, gzip	equake, twolf	bzip2, equake
iv	mesa, bzip2	equake, mesa	art, mcf	equake, twolf
v	gzip, twolf	mcf, mcf	equake, bzip2	equake, mesa
vi	equake, mesa	gzip, gzip	twolf, gzip	equake, gzip
vii	art, gzip	equake, gzip	bzip2, gzip	bzip2, mcf
viii	mcf, twolf	twolf, art	bzip2, bzip2	twolf, mcf
ix	mesa, equake	twolf, bzip2	bzip2, mesa	mcf, equake
x	mcf, bzip2	gzip, gzip	twolf, mesa	gzip, equake

Fairness (Performance Disparity)

本評価で用いる Fairness の指標は，文献 [7] を参考にして Performance Disparity (PD) とする．PD は，単独実行した場合に対する複数ベンチマークで実行した場合の各ベンチマークの PR の差，つまり実行したプロセス中で最も PR の高いものと最も PR の低いものの差である．すなわち，PD は実行したベンチマーク間の性能低下の最大較差を表している．この値が小さいほど，Fairness は良いと考えることができる．実行するプロセスをそれぞれ $P_0, P_1, \cdots, P_{NPR-1}$ として，PD は式 3 によって計算される．

$$\begin{aligned} PD\big(P_0, P_1, \cdots, P_{NPR-1}\big) &= \max_{a,b=0,1,\cdots,NPR-1} \big|PR(P_a) - PR(P_b)\big| \\ &= \max\Big(PR\big(P_0\big), PR\big(P_1\big), \cdots, PR\big(P_{NPR-1}\big)\Big) \\ &\quad - \min\Big(PR\big(P_0\big), PR\big(P_1\big), \cdots, PR\big(P_{NPR-1}\big)\Big) \end{aligned} \quad (3)$$

トータルスループット (IPC Improvement)

IPC Improvement は，IPC で見た場合の，提案手法を用いない通常の Linux スケジューラ (ORG) に対しての性能向上率を表す．なお，マイナスの値の場合 ORG に対しての性能低下率を示す．C の条件のもとで実行したときの全プロセスの IPC の合計を

$TotalIPC$(C)，ORG で実行したときの全プロセスの IPC の合計を $TotalIPC$(ORG) とする場合の IPC Improvement は，式 4 で計算される．

$$IPC_Improvement(\mathrm{C}) = \frac{TotalIPC(\mathrm{C})}{TotalIPC(\mathrm{ORG})} - 1 \tag{4}$$

4.3　2 コアを用いる場合の評価

4.3.1　各ベンチマークの組み合わせでの評価結果

図 17，18 は，ベンチマークの組み合わせごとに見た評価の結果を示している．各パラメータは，$N = 25$，$I = 200$ ms，使用するカウンタはカウンタ 1（L1 キャッシュアクセス回数）とした．

図 17 は，Fairness の評価結果を示している．C2Q-SHR に関しては，viii を除く全ての組み合わせで ORG に対して Fairness が向上した．全ての組み合わせを平均しても ORG に対して約 1.19 ポイントの Fairness の向上が見られた．一方 Ci7-DUAL に関しては，i，iii，vi，x の組み合わせで ORG に対して Fairness が向上した．平均すると ORG よりも Fairness が約 0.51 ポイント悪化した．これは，iv，v，ix において Fairness の悪化が大きいためである．

図 18 は，トータルスループットの評価を示している．C2Q-SHR に関しては，i，iii，iv，viii，x でトータルスループットが悪化したものの，他の組み合わせでは ORG に対してトータルスループットの向上が見られた．平均すると ORG に対して約 0.48%トータルスループットが向上した．ここで iii においてトータルスループットの悪化が大きいが，この場合でも Fairness は ORG に対して上回っている組み合わせであった．Ci7-DUAL に関しては，全ての組み合わせで ORG に対してトータルスループットが向上した．平均すると ORG に対して約 1.48%トータルスループットが向上している．特に iii においてトータルスループットの向上率が高かった．この点は C2Q-SHR とは対照的である．この他に vi，ix も向上率が大きい．なお，iii は，Fairness も大きく改善できている．

これらの結果より，Ci7-DUAL の Fairness は悪化してしまったが，本提案手法が概ね有効であるということが確認できた．

4.3.2　N を変化させた場合の評価

図 19，20 は，最小二乗法を適用する際のデータの個数である N を変化させた場合の評価の結果を示している．これらの結果に関しては，$I = 200$ ms，使用したカ

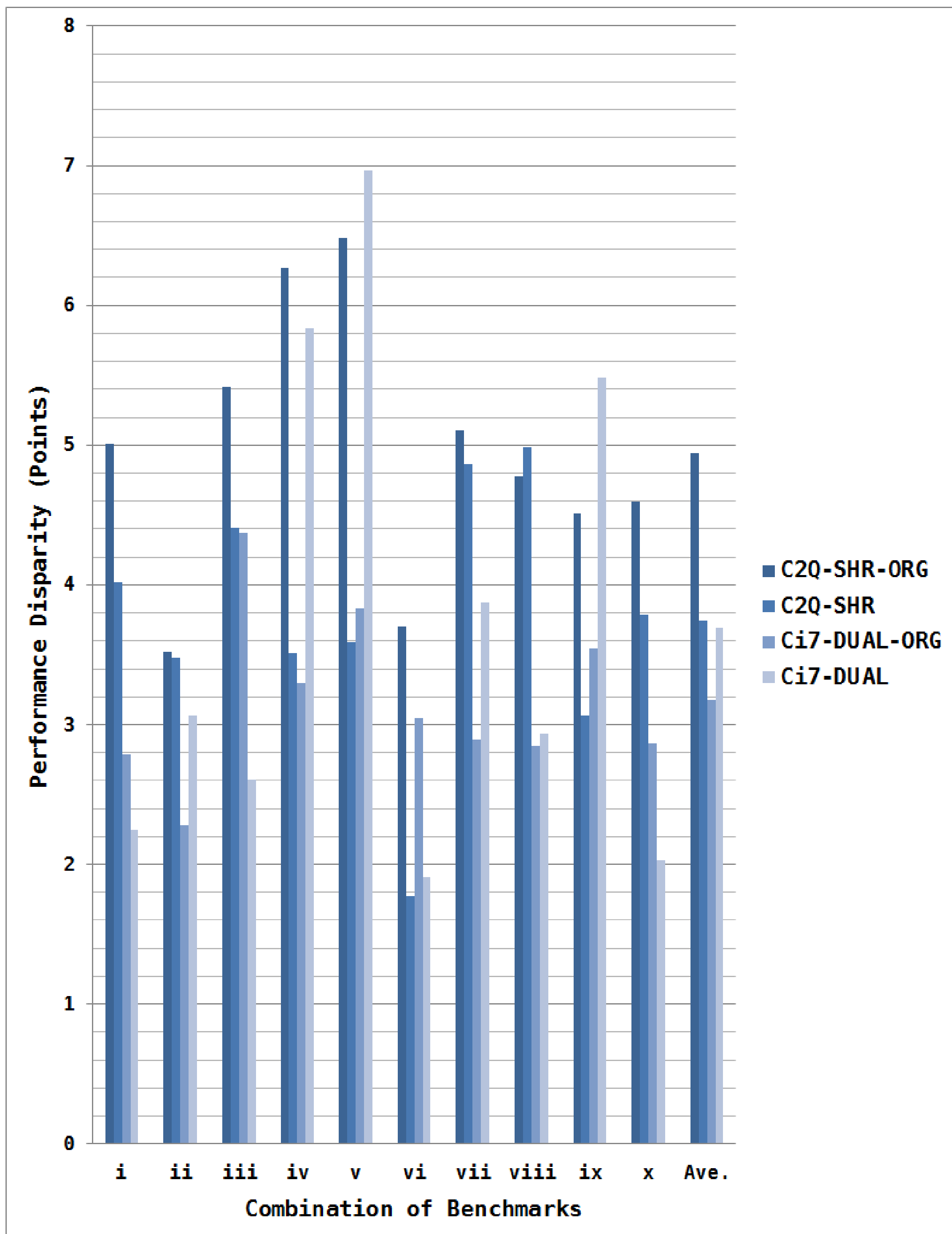

図 17: 2 コア，ベンチマークの組み合わせごとの Fairness 評価

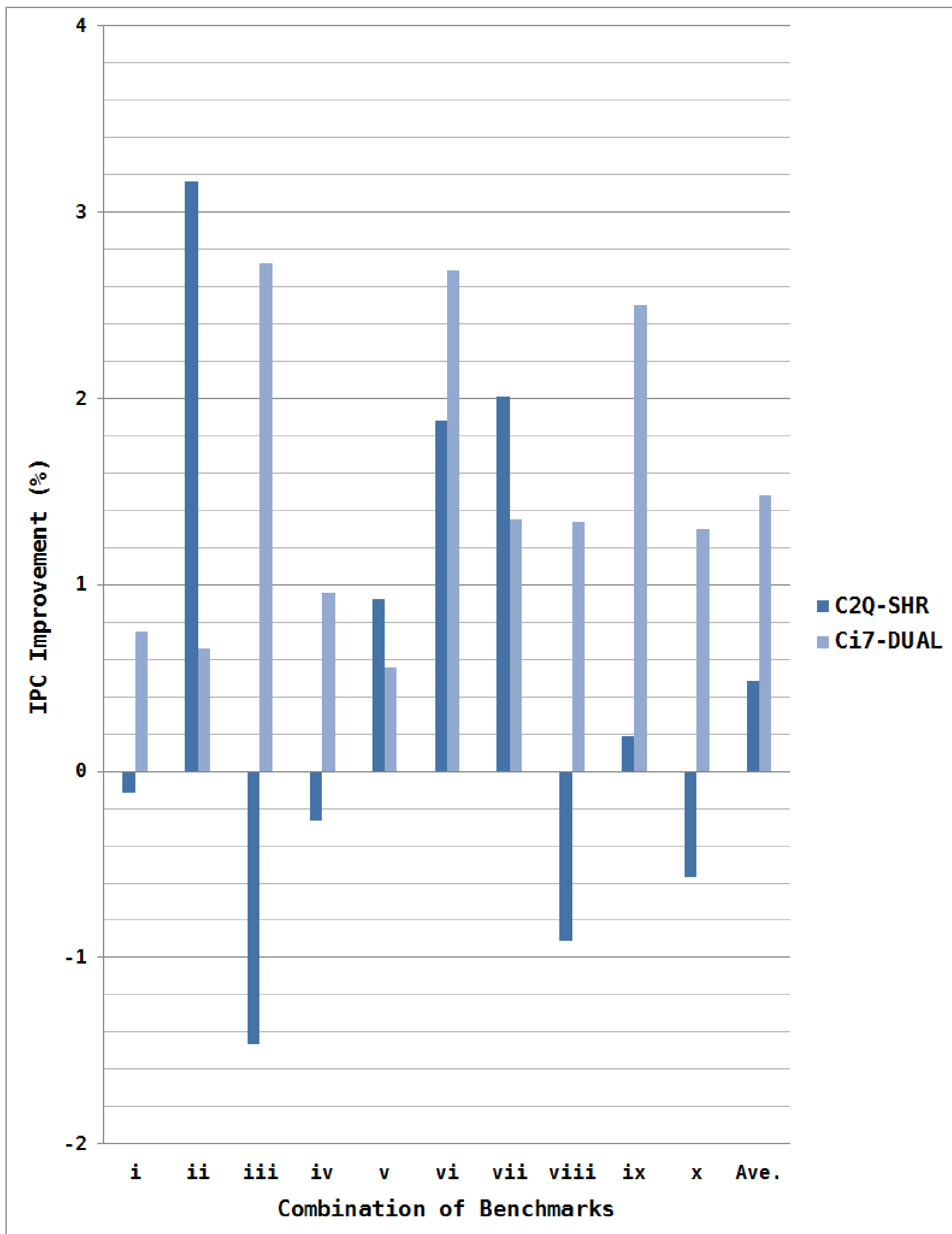

図 18: 2 コア，ベンチマークの組み合わせごとのトータルスループット評価

ウンタはカウンタ1（L1キャッシュアクセス回数），全ベンチマークの組み合わせの平均とした．

図19は，Fairnessの評価結果を示している．C2Q-SHRに関しては，$N = 5$の場合でORGよりも約0.65ポイントFairnessが悪化しているが，$N = 10$，$N = 25$と，Nが増加するに従って，ORGよりもFairnessが改善されている．また$N = 25$の場合の方がORGに対して約1.19ポイントとFairnessの改善の度合いが大きい．Ci7-DUALに関しては，$N = 5$，$N = 10$，$N = 25$のいずれの場合でもORGよりもFairnessが悪化している（ORGに対してそれぞれ約4.63，1.40，0.51ポイントの悪化）．Ci7-DUALの場合にはFairnessの改善があまり見られなかった．この結果より，Nを少なくとも$N = 25$まで大きくするとよいと考えられる．

図20は，トータルスループットの評価結果を示している．C2Q-SHRに関しては，$N = 5$に比べて$N = 10$，$N = 10$に比べて$N = 25$の場合がトータルスループットが低下した（ORGに対してそれぞれ約0.73，0.55，0.48%の向上）．Ci7-DUALに関しては，C2Q-SHRとは逆に$N = 5$に比べて$N = 10$，$N = 10$に比べて$N = 25$の場合がトータルスループットが向上した（ORGに対してそれぞれ約1.28，1.41，1.48%の向上）．C2Q-SHRとは対照的に$N = 25$の場合がORGに対して約2.10%と最もトータルスループットは高かった．C2Q-SHR，Ci7-DUAL双方ともにトータルスループット面で平均してORGに対してそれぞれ約0.59，1.39%と，ORGに上回ることができた．

Nの変化とトータルスループットの向上率との間の関係は環境によって異なり，最適なNの値は簡単には定まらない．しかしORGに対してどのNもトータルスループットで上回っているため，Fairnessとの関連を考えると$N = 25$が適切だと考えられる．

4.3.3　Iを変化させた場合の評価

図21，22は，PollingIntervalであるIを変化させた場合の評価の結果を示している．これらの結果に関しては，$N = 25$，使用したカウンタはカウンタ1（L1キャッシュアクセス回数），全ベンチマークの組み合わせの平均とした．

図21は，Fairnessの評価結果を示す．C2Q-SHRに関しては，$I = 200$ msをピークとして（ORGに対して約1.20ポイントの向上）それよりもIを長くしても短くしてもFairnessは悪化した．Ci7-DUALに関しては，どの場合でもFairnessはORGに比べ悪化している．この結果では，Iの変化によるFairnessの変化の相関は見受けられなかった．以上の結果より，$I = 200$ msがFairnessに関して最高の性能が得

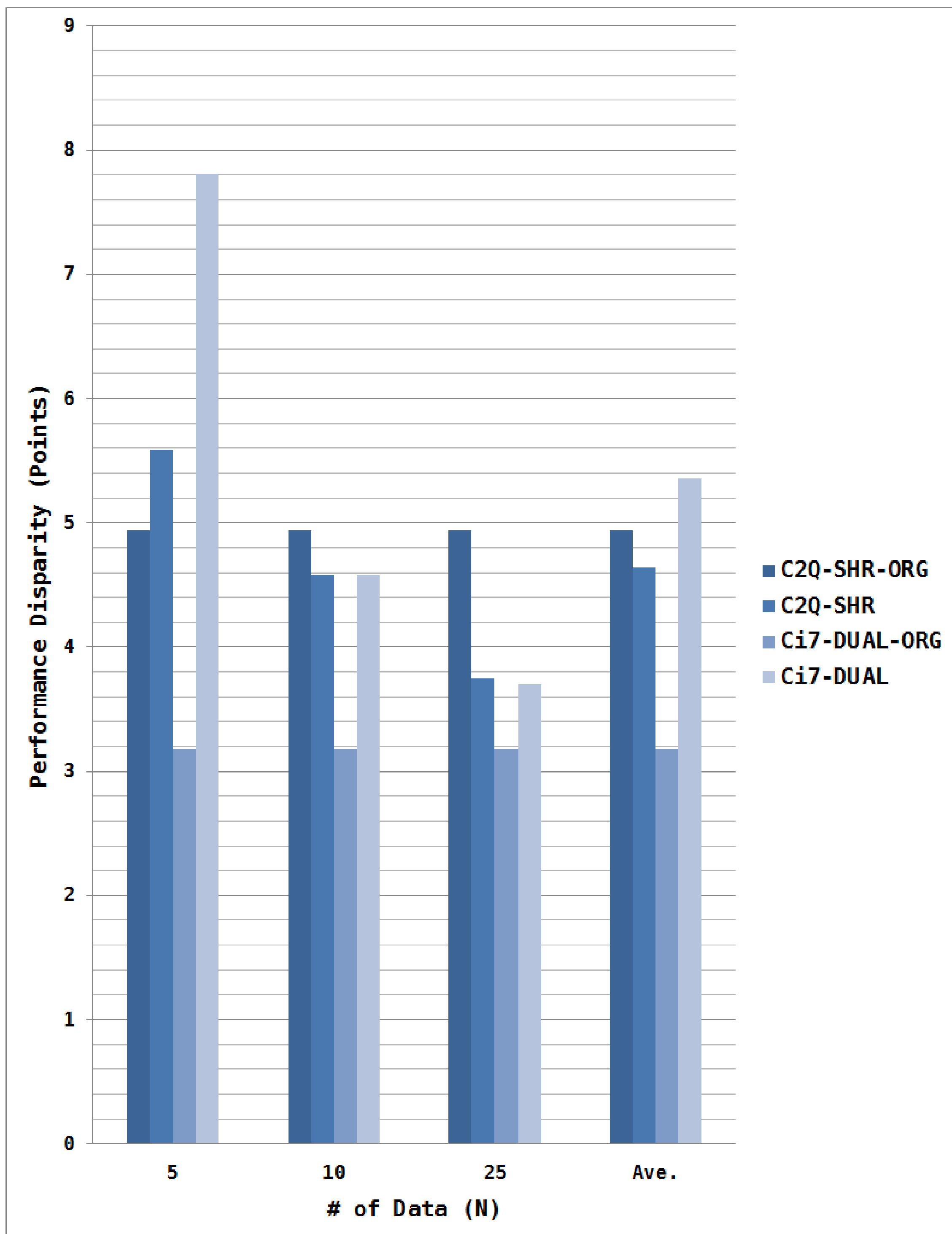

図 19: 2コア，N の変化に関する Fairness 評価

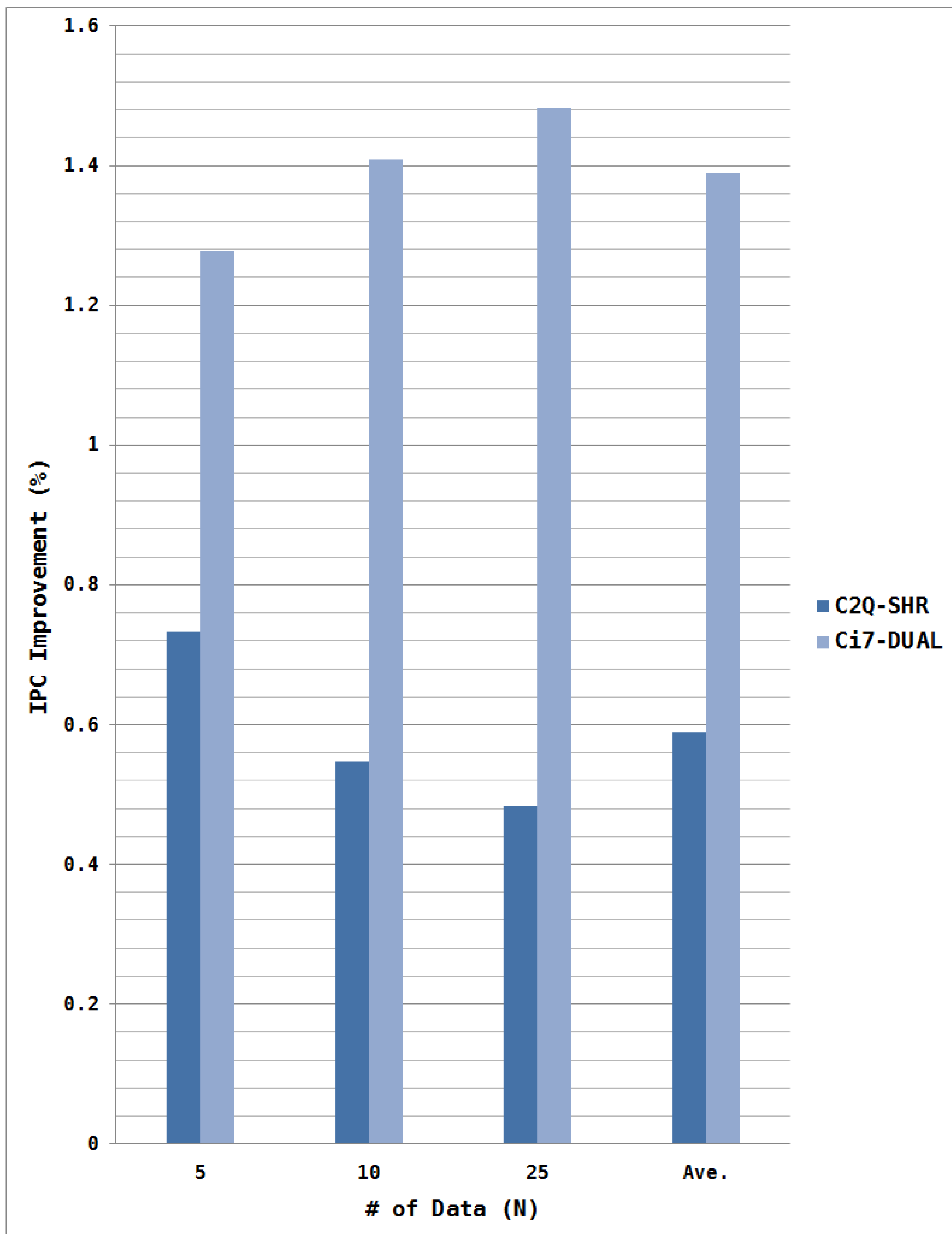

図 20: 2 コア，N の変化に関するトータルスループット評価

られると考えられる．

図 22 は，トータルスループットの評価結果を示している．C2Q-SHR に関しては，I を長くするほどトータルスループットは概ね向上する傾向が見られた．しかし $I = 150$ ms や $I = 200$ ms の場合はそれに当てはまらない（ORG に対してそれぞれ約 0.46，0.48%の向上）．Ci7-DUAL に関しては，I を $I = 180$ ms まで長くしていくとトータルスループットは向上していくが（ORG に対して約 1.48%の向上），それよりも長くすると逆にトータルスループットは低下に転じた．

以上の結果より，$I = 180, 200$ ms あたりがトータルスループットに関して最高の性能が得られると考えられる．

4.3.4 カウンタごとの評価

図23，24 は，カウンタごとの評価の結果である．これらの結果に関しては，パラメータは $N = 25$，$I = 200$ ms，全ベンチマークの組み合わせの平均とした．なお，カウンタ 1 が L1 キャッシュアクセス回数，カウンタ 2 および 3 が L2 キャッシュミス回数である．

図 23 は，Fairness の評価結果を示している．C2Q-SHR に関しては，カウンタ 1，2 のどちらを用いた場合にも ORG と比較して Fairness が改善できている（ORG に対してそれぞれ約 1.19，0.63 ポイントの向上）．ただし，カウンタ 1 を用いた場合がカウンタ 2 を用いた場合に比べて若干 Fairness が向上した．Ci7-DUAL に関しても，カウンタ 1，3 のどちらを用いた場合にも ORG と比較して Fairness が悪化した（ORG に対してそれぞれ約 0.51，0.41 ポイントの悪化）．こちらの場合は，カウンタ 3 を用いた場合がカウンタ 1 を用いた場合に比べて若干 Fairness が向上している．

図 24 は，トータルスループットの評価結果を示している．C2Q-SHR に関しては，カウンタ 1，2 のどちらを用いた場合にも ORG と比較してトータルスループットが向上している（ORG に対してそれぞれ約 0.48，1.39%の向上）．ただし，カウンタ 2 を用いた場合の方が，カウンタ 1 を用いた場合よりもトータルスループットが向上している．Ci7-DUAL に関しても，カウンタ 1，3 のどちらを用いた場合にも ORG と比較してトータルスループットが向上していることがわかる（ORG に対してそれぞれ約 1.48，1.47%の向上）．こちらは，C2Q-SHR とは逆にカウンタ 1 を用いる場合の方がトータルスループットの向上率が高い．Ci7-DUAL においては，カウンタによるトータルスループットの差が C2Q-SHR よりも小さい．

C2Q-SHR の場合はカウンタ 2，Ci7-DUAL の場合はカウンタ 1 を選択しても本提案手法による効果が得られると考えられる．

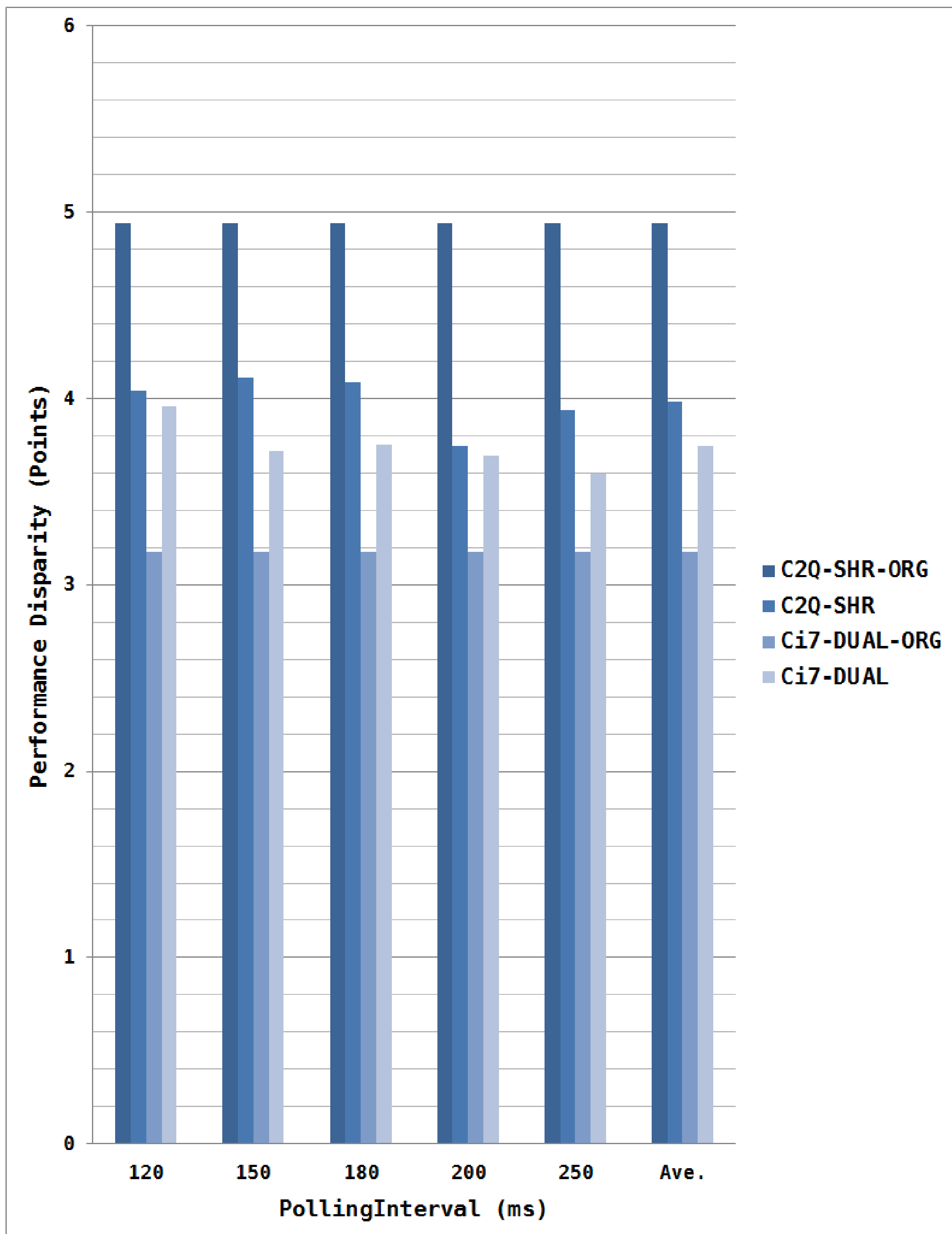

図 21: 2 コア，I の変化に関する Fairness 評価

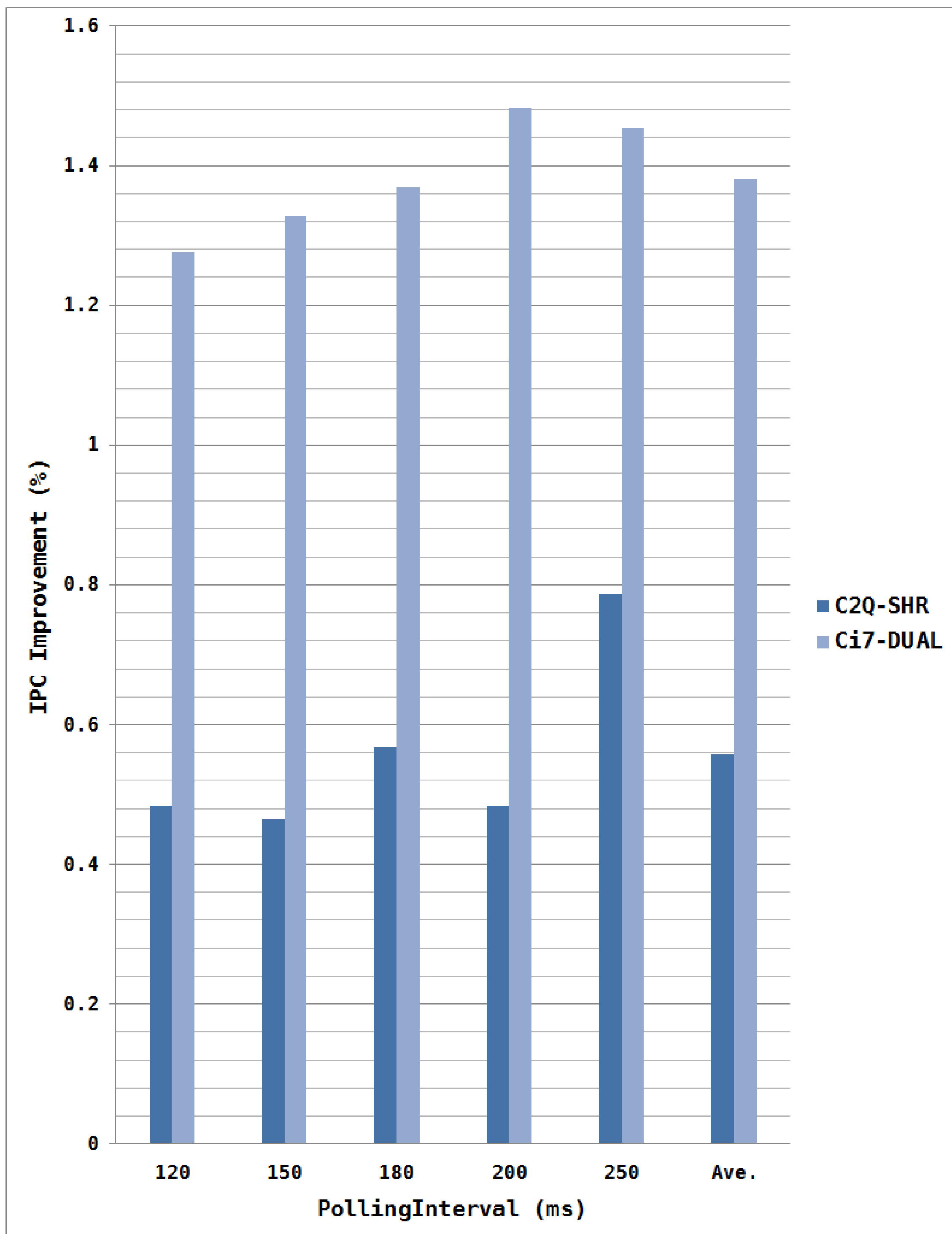

図 22: 2 コア，I の変化に関するトータルスループット評価

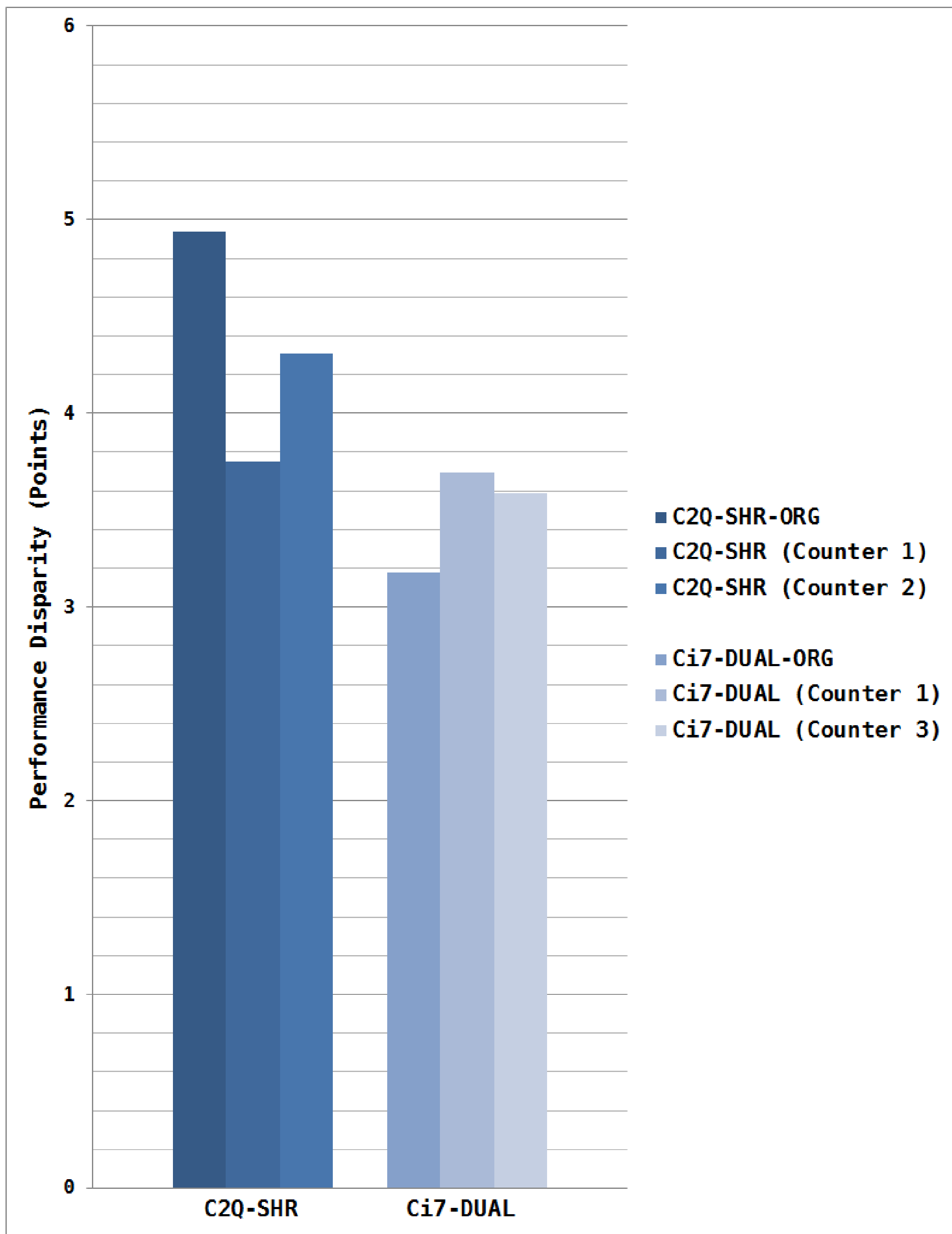

図 23: 2 コア，カウンタごとの Fairness 評価

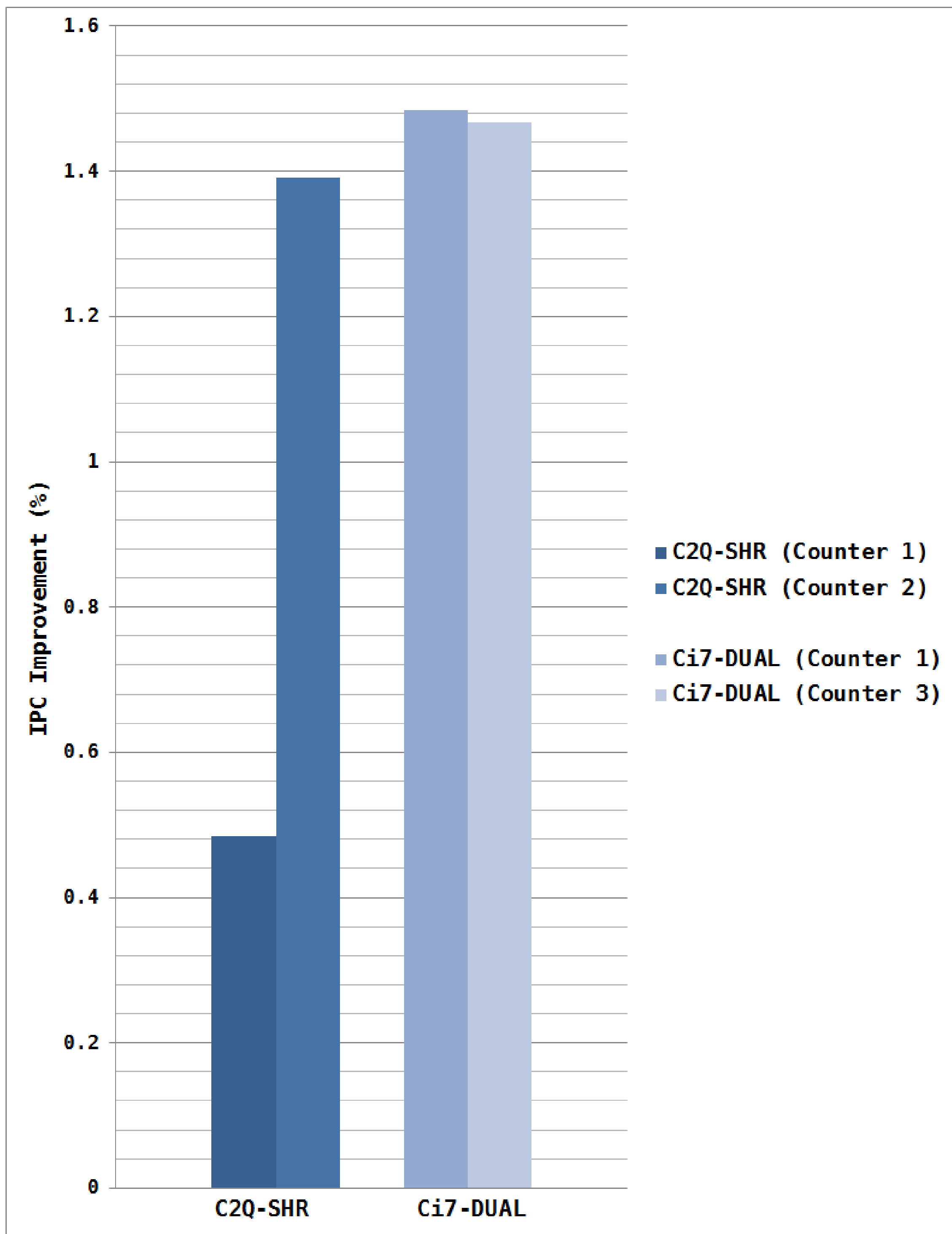

図 24: 2 コア，カウンタごとのトータルスループット評価

4.4　4 コアを用いる場合の評価

4.4.1　ベンチマークの組み合わせでの評価結果

図 25，26 は，ベンチマークの組み合わせごとの評価の結果を示している．各パラメータは，$N = 25$，$I = 200$ ms，使用するカウンタはカウンタ 1（L1 キャッシュアクセス回数）とした．

図 25 は，Fairness の評価結果を示している．C2Q-QUAD に関しては，ORG に対して i，v，vii，ix では Fairness が改善され，ii，iii，iv，vi，viii，x では Fairness が悪化した．Ci7-DUAL に関しては，ORG に対して i，ii，iv，v，x では Fairness が改善され，iii，vi，vii，viii，ix では Fairness が悪化した．いずれの場合においても平均すると ORG よりも Fairness が悪化した（ORG に対してそれぞれ約 2.13，0.64 ポイントの悪化）．

図 26 は，トータルスループットの評価結果を示している．C2Q-QUAD に関しては，特に i において向上率が大きく，ix において低下率が大きい．平均すると ORG に対して約 1.73%低下している．Ci7-QUAD に関しては，ORG に対して vi，ix 以外の組み合わせでトータルスループット向上という結果を得た．平均しても ORG よりもトータルスループットが約 0.35%向上している．

C2Q-QUAD においては，Ci7-QUAD に対してベンチマークの組み合わせによるトータルスループットの変動が大きかった．このことは，2 つのラストレベルキャッシュを 2 つのコアごとに共有している C2Q-QUAD では，全てのコアでラストレベルキャッシュを共有する Ci7-QUAD に比べてどのコアでどのベンチマークが実行されるかにより大きく性能に左右するためだと考えられる．

4.4.2　N を変化させた場合の評価

図 27，28 は，最小二乗法を適用する際のデータの個数である N を変化させた場合の評価の結果を示している．これらの結果に関しては，$I = 200$ ms，使用したカウンタはカウンタ 1（L1 キャッシュアクセス回数），全ベンチマークの組み合わせの平均とした．

図 27 は，Fairness の評価結果を示している．C2Q-QUAD，Ci7-QUAD 双方に関して，N の変化による Fairness の差はほとんどなかった．しかし，C2Q-QUAD，Ci7-QUAD どちらにおいても Fairness は ORG に比べて悪化している（ORG に対してそれぞれ約 0.73，0.64 ポイントの悪化）．

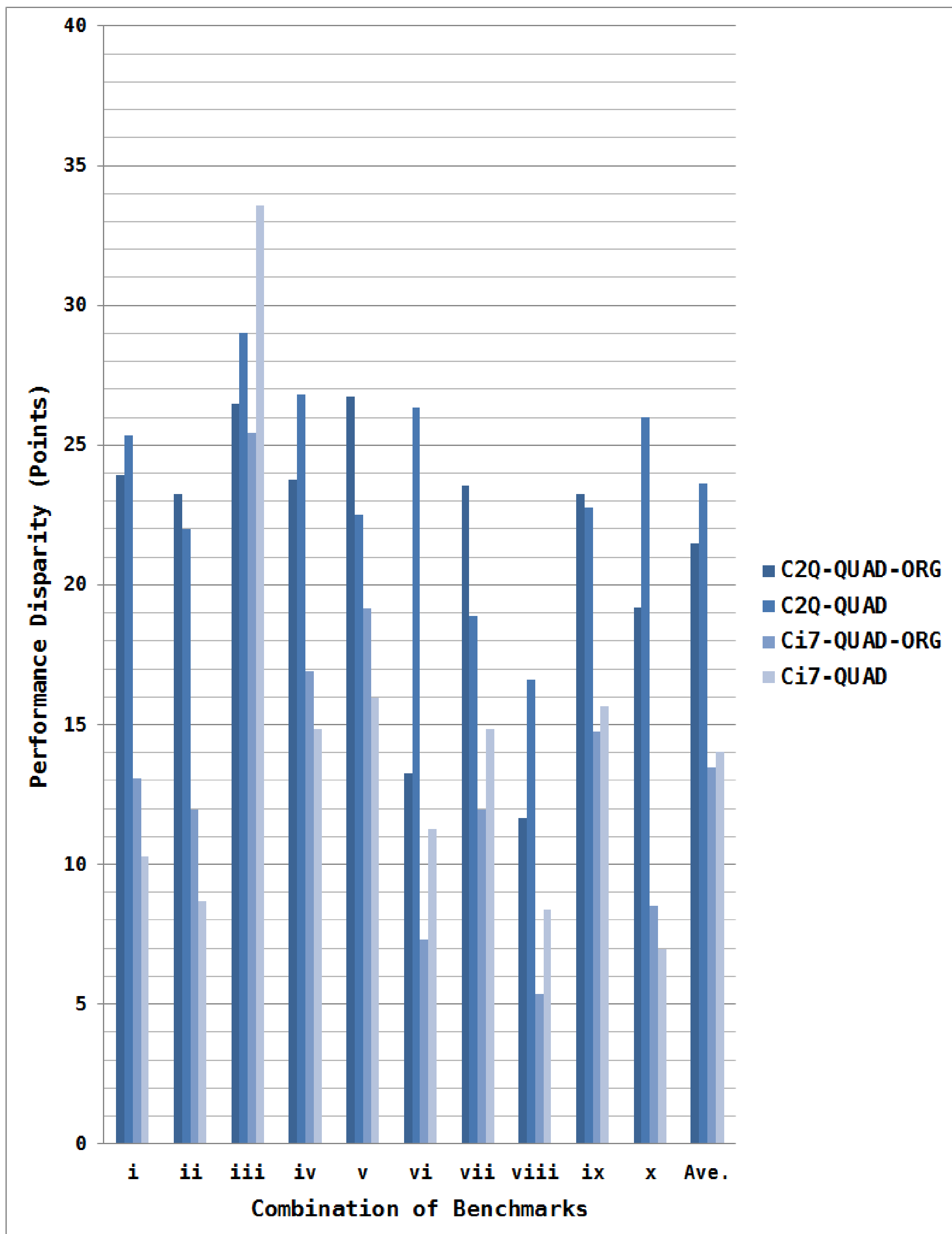

図 25: 4 コア，ベンチマークの組み合わせごとの Fairness 評価

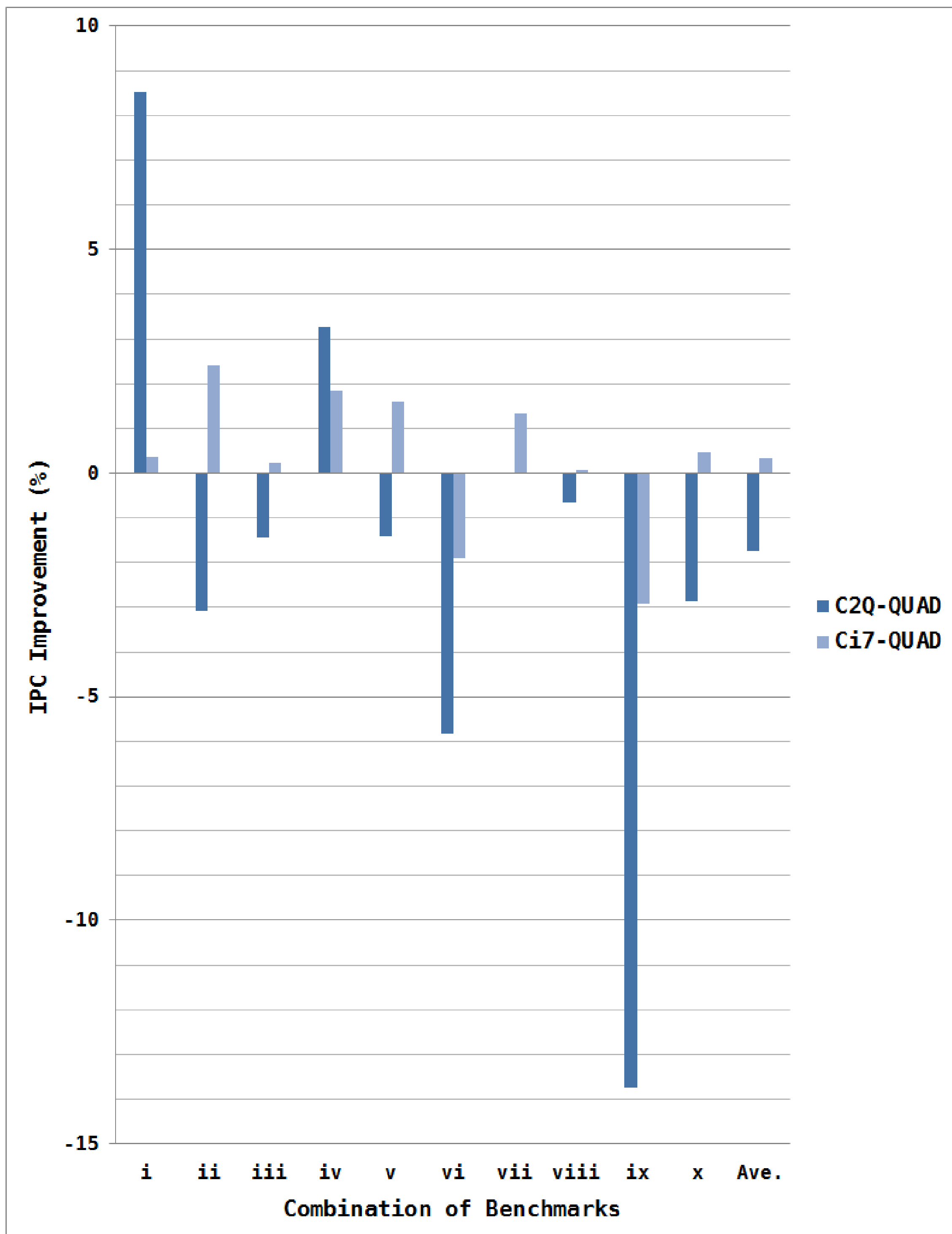

図 26: 4 コア，ベンチマークの組み合わせごとのトータルスループット評価

図 28 は，トータルスループットの評価結果を示している．C2Q-QUAD に関しては，$N = 5$ から $N = 10$ に大きくするとトータルスループットは大きく低下するが，$N = 10$ から $N = 25$ にさらに大きくすると若干向上した．Ci7-QUAD に関しては，N の値を大きくしていくとトータルスループットは向上する傾向が見られた．C2Q-QUAD では ORG に対してのトータルスループット向上は見られなかったが（ORG に対して約 1.43%の悪化），Ci7-QUAD では ORG に対して約 0.17%のトータルスループット向上があった．

この場合では，N の値は Fairness に影響を与えない．そのため最小二乗法によるオーバヘッドを減らすため $N = 5$ を選択するとトータルスループットが向上すると考えられる．実際に C2Q-QUAD においては，$N = 10, 25$ よりもトータルスループットが向上している．Ci7-QUAD はこれには当てはまらないが，$N = 5$ の場合には N の値が小さすぎて正確な予測ができていないためだと考えられる．

4.4.3　I を変化させた場合の評価

図 29，30 は，PollingInterval である I を変化させた場合の評価の結果を示している．これらの結果に関しては，$N = 25$，使用したカウンタはカウンタ 1（L1 キャッシュアクセス回数），全ベンチマークの組み合わせの平均とした．

図 29 は，Fairness の評価結果を示している．C2Q-QUAD，C2Q-QUAD 双方に関して，ORG よりも Fairness で上回ることができなかった．ただし，I を長くするにつれて若干ではあるが Fairness が改善する方向に向かうが，$I = 200$ ms を超えると Fairness が悪化する傾向にあった．

図 30 は，トータルスループットの評価結果を示している．C2Q-QUAD に関しては，ORG よりもトータルスループットで上回ることができなかった（ORG に対して約 2.18 ポイントの悪化）．ただし，I を長くするにつれてトータルスループットが向上する傾向にあるが，$I = 250$ ms としたときには $I = 200$ ms に比べてトータルスループットは低下した．Ci7-QUAD に関しては，ORG よりもトータルスループットが約 0.27%向上した．C2Q-QUAD 同様に，I を長くするにつれてトータルスループットが向上する傾向にあるが，$I = 250$ ms としたときには $I = 200$ ms に比べてトータルスループットは低下した．

I の変化による Fairness の変化はほとんどなく，トータルスループットは大きく変動することから，トータルスループットだけを見て本提案手法の効果が最も表れる $I = 200$ ms を選択するとよいと思われる．

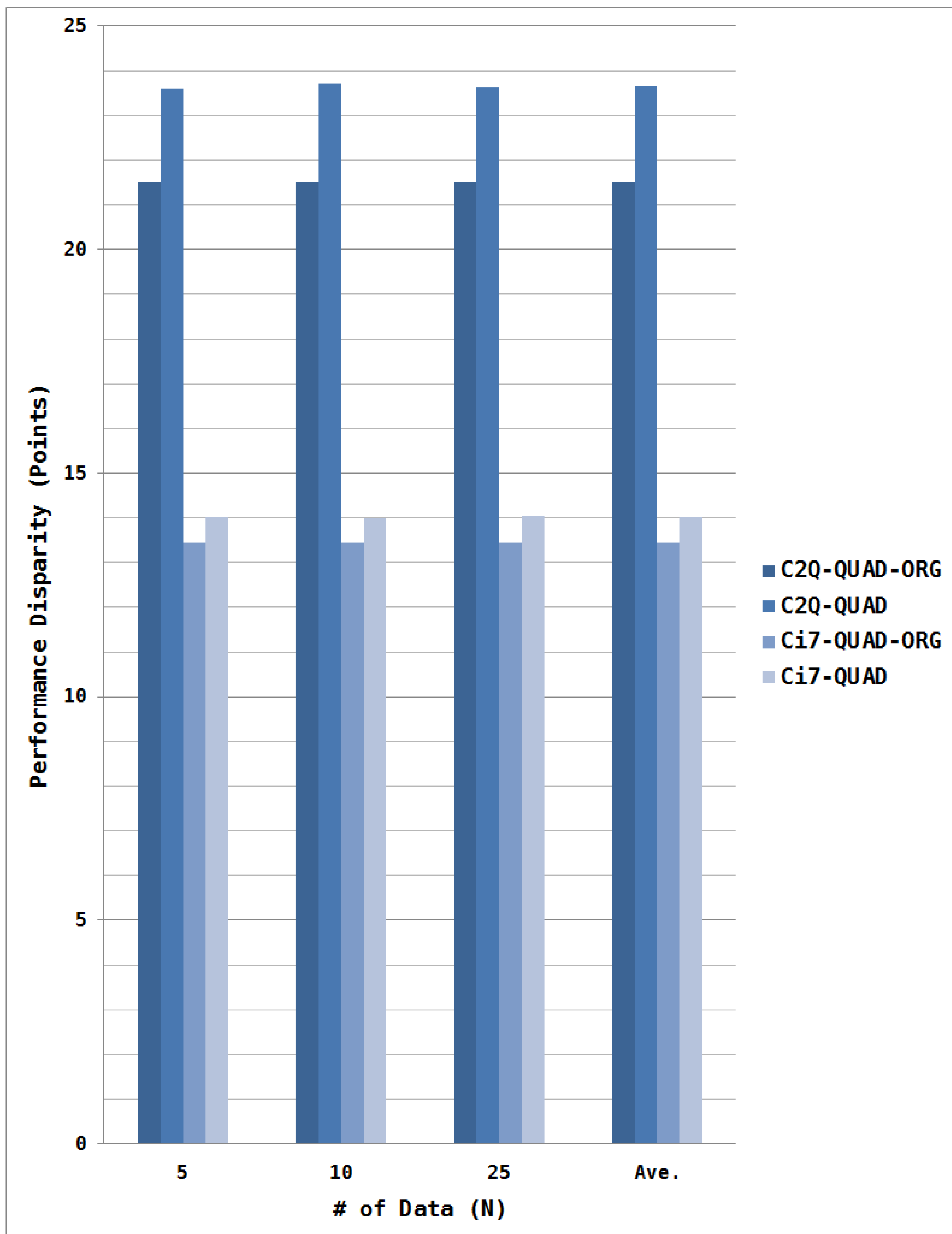

図 27: 4 コア，N の変化に関する Fairness 評価

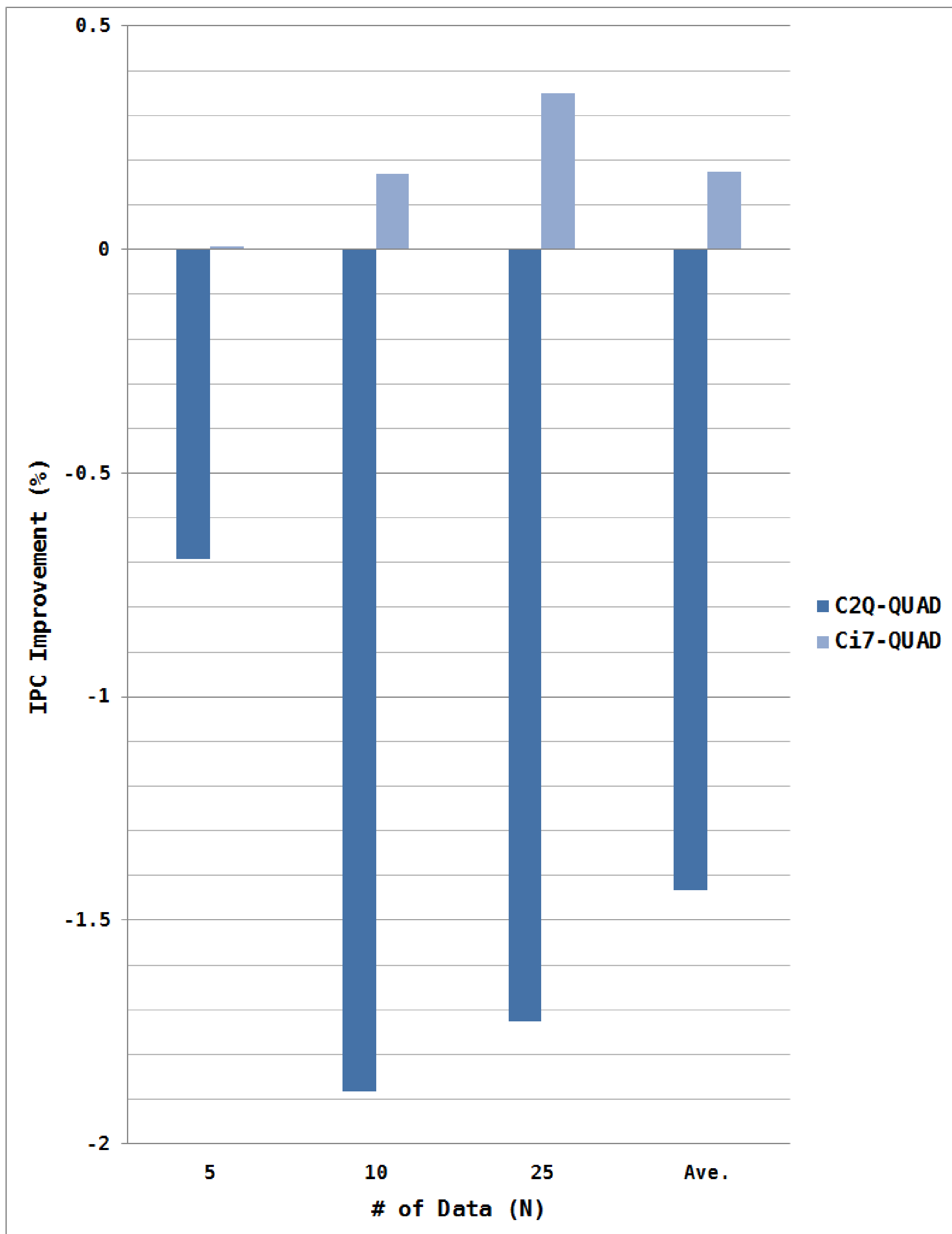

図 28: 4 コア，N の変化に関するトータルスループット評価

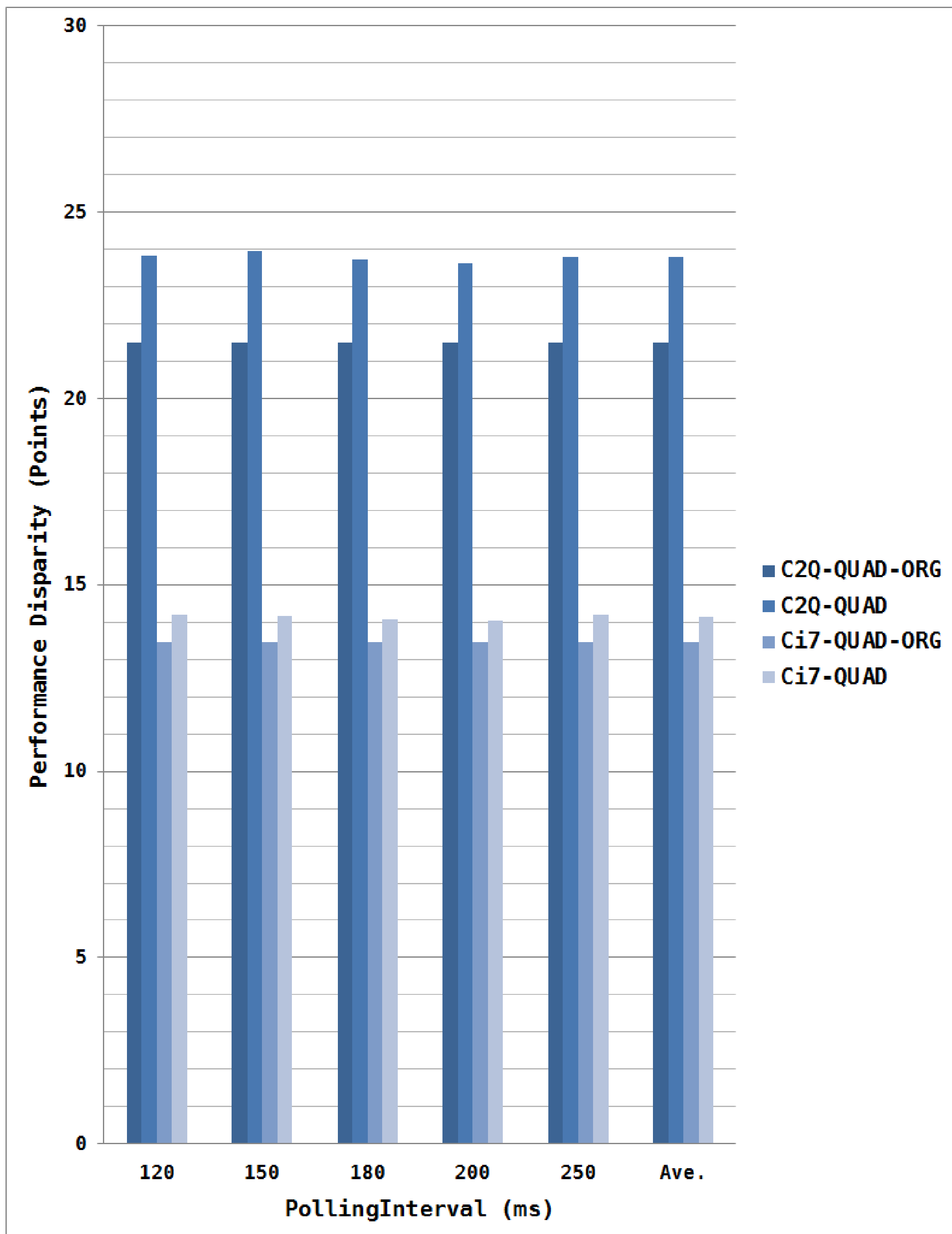

図 29: 4 コア，I の変化に関する Fairness 評価

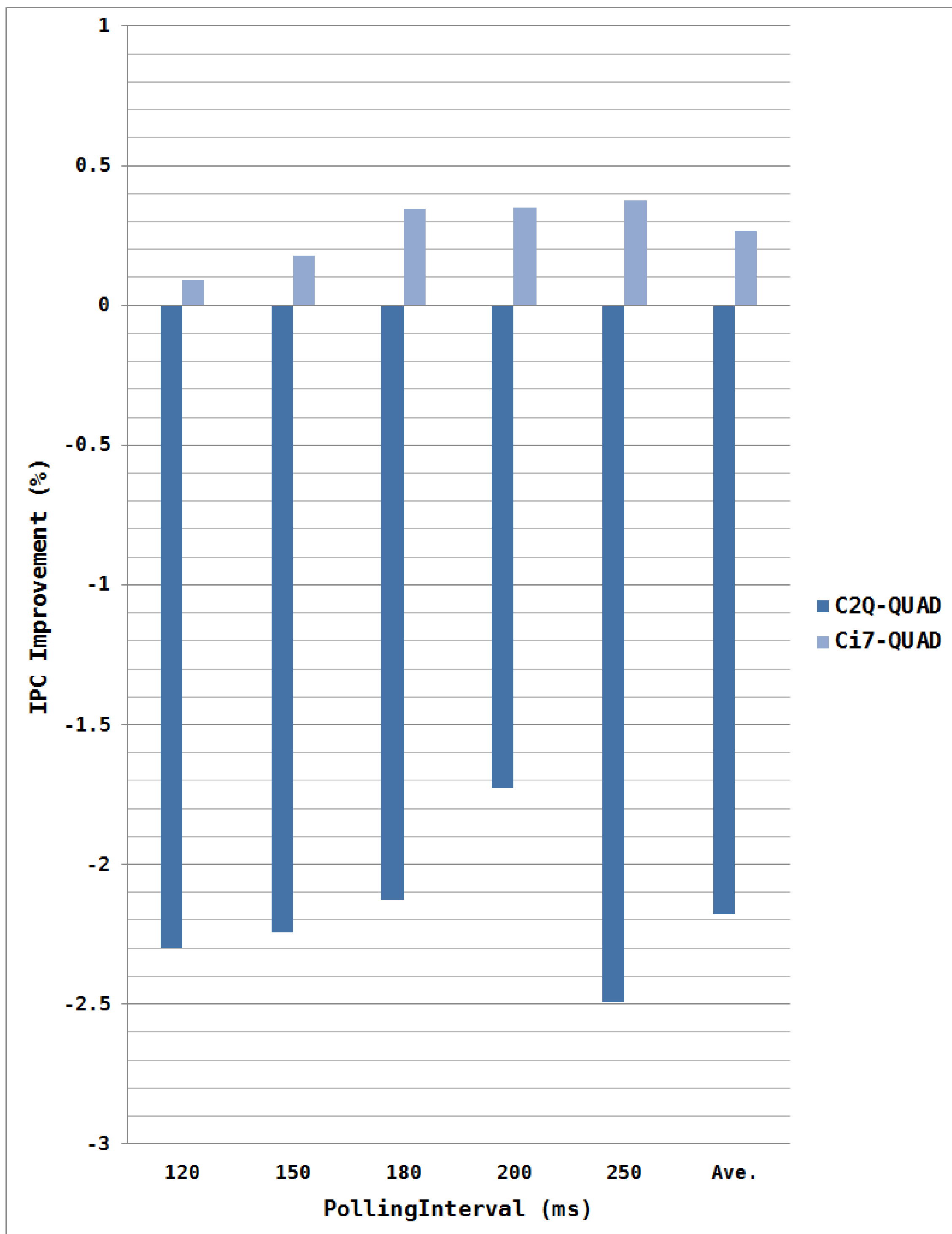

図 30: 4 コア，I の変化に関するトータルスループット評価

4.4.4　カウンタごとの評価

図 31，32 は，カウンタごとの評価の結果である．これらの結果に関しては，パラメータは $N = 25$，$I = 200$ ms，全ベンチマークの組み合わせの平均とした．なお，カウンタ 1 が L1 キャッシュアクセス回数，カウンタ 2 および 3 が L2 キャッシュミス回数である．

図 31 は，Fairness の評価結果を示している．C2Q-QUAD に関しては，カウンタ 1，2 のどちらを用いた場合にも ORG と比較して Fairness が悪化した（ORG に対してそれぞれ約 2.13，2.24 ポイントの悪化）．カウンタ 1 の場合は，カウンタ 2 比べて若干 Fairness が良い．Ci7-QUAD に関しても，カウンタ 1，3 のどちらを用いた場合にも ORG と比較して Fairness が悪化した．（ORG に対してそれぞれ約 0.60，3.04 ポイントの悪化）こちらの場合は，カウンタ 3 を用いた方がカウンタ 1 に比べて Fairness 悪化の程度が大きい．

図 32 は，トータルスループットの評価結果を示している．C2Q-QUAD に関しては，カウンタ 1 を用いた場合にはトータルスループットが ORG に比べて約 1.73%低下しているが，カウンタ 2 を用いる場合にはトータルスループットは ORG に対して約 0.04%向上している．Ci7-QUAD に関しては，C2Q-QUAD とは逆にカウンタ 1 を用いた場合にはトータルスループットは ORG に比べて約 0.35%向上しているが，カウンタ 3 用いる場合には ORG に比べてトータルスループットが約 0.03%悪化した．

カウンタの選択の仕方により，C2Q-QUAD と Ci7-QUAD ではかなり異なる傾向の結果が得られた．これは，キャッシュ階層の構成の違いにより，どのカウンタが共有資源の要求率をよく表しているかがそれぞれ異なるためであると考えられる．

4.5　結果のまとめ

各結果を Fairness，トータルスループットごとにまとめ，表 7，8 に示した．各結果は，$N = 25$，$I = 200$ ms，カウンタ 1 を使用した時の，全ベンチマークの組み合わせの平均である．ORG に対して提案手法が優れている結果に関しては太字で示した．

各結果より，2 つのコアを用いる場合に関しては Ci7-DUAL の Fairness 以外の結果全てで ORG に対して優れた結果となった．一方で，4 つのコアを用いる場合に関しての結果は，Ci7-QUAD でのトータルスループットの以外で全て ORG に対して下回った．

よって，2 つのコアを用いる場合には ORG に対しての優位性が認められたが，4 つのコアを用いる場合には提案手法が有効に機能していない結果となった．

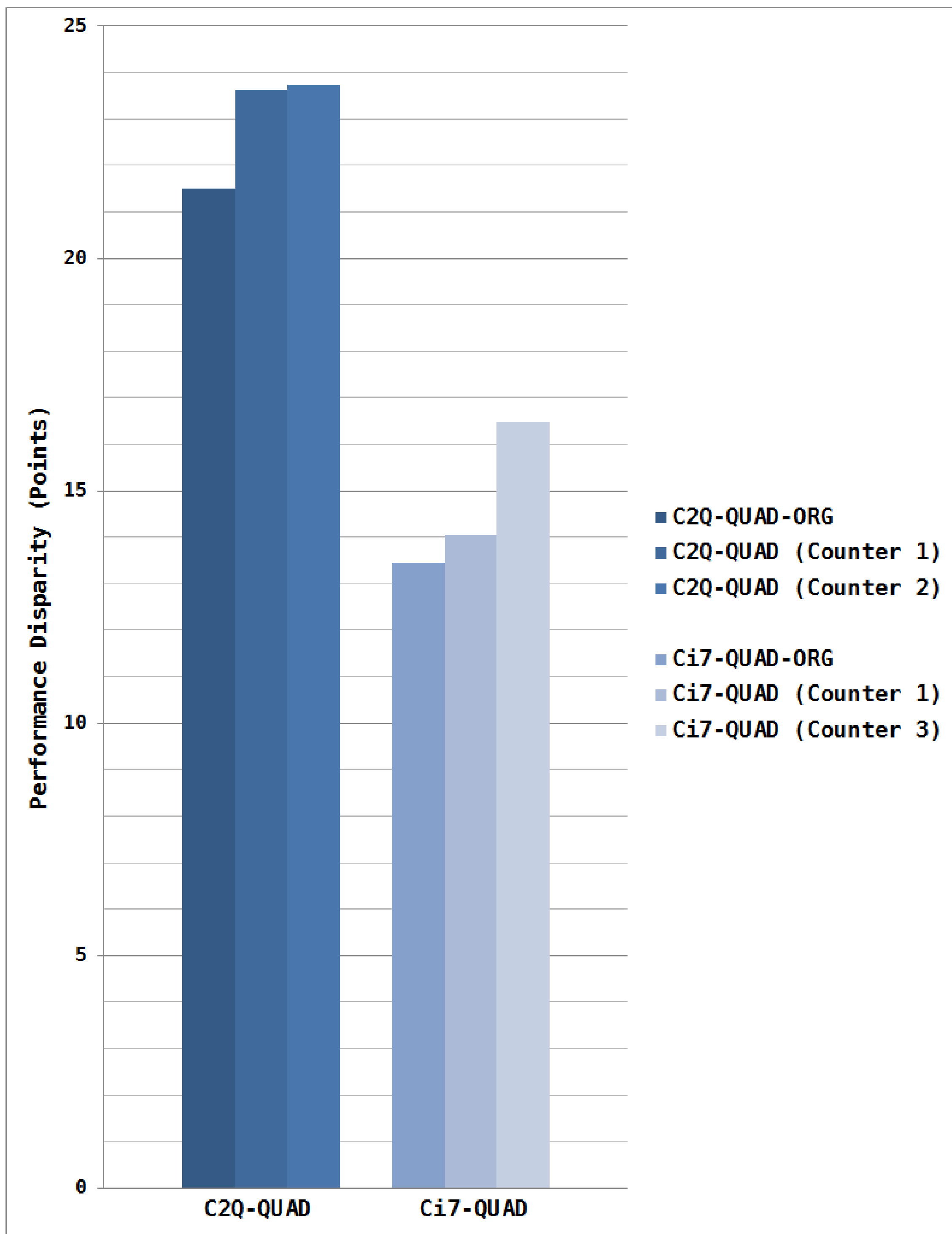

図 31: 4 コア，カウンタごとの Fairness 評価

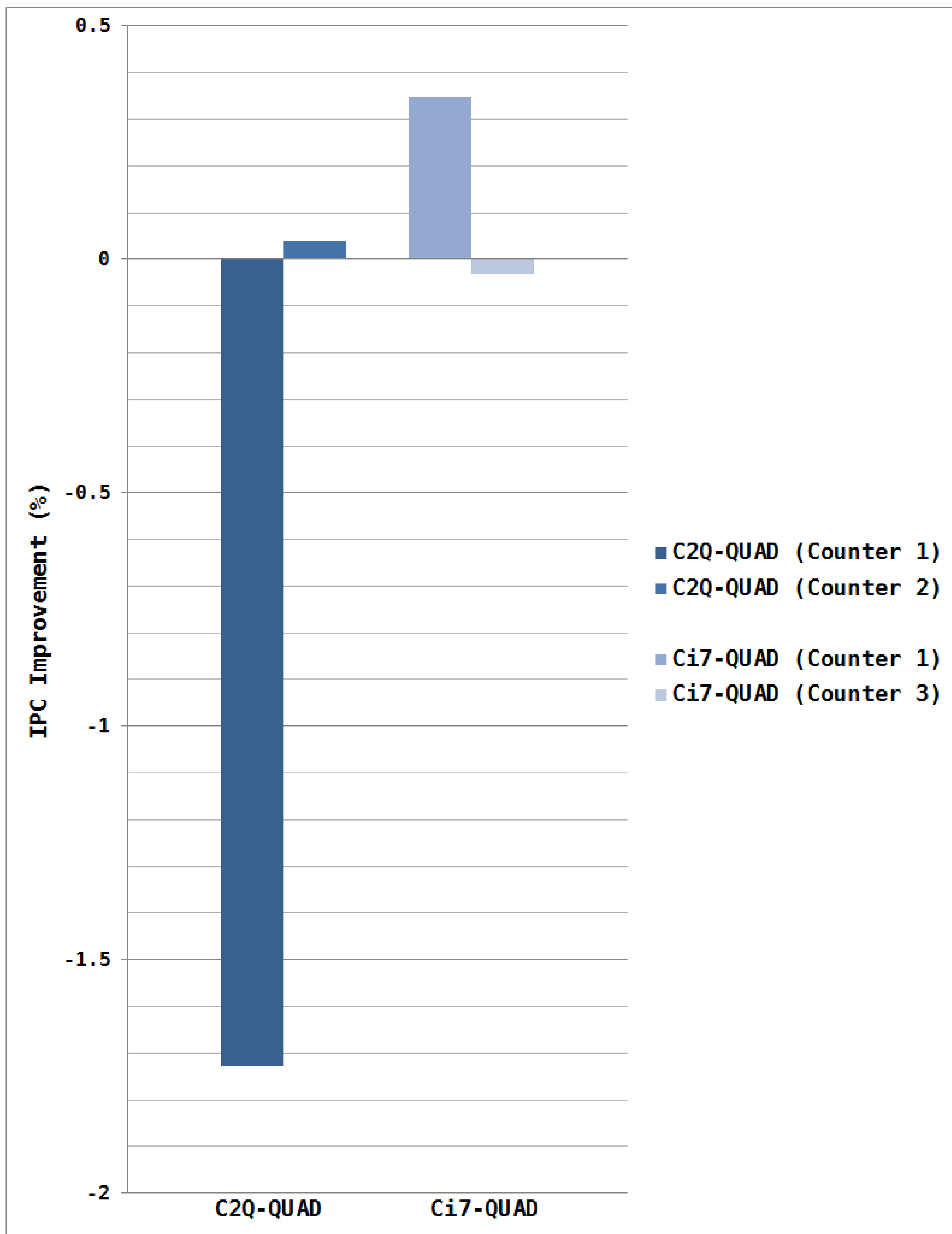

図 32: 4 コア，カウンタごとのトータルスループット評価

表 7: 2 つのコアを用いる場合の結果のまとめ

		C2Q-SHR	Ci7-DUAL
Fairness （ポイント）	ORG	4.94	3.18
	提案手法	**3.75**	3.69
ORG に対する トータルスループット向上率 (%)	ORG	-	-
	提案手法	**+0.48**	**+1.48**

表 8: 4 つのコアを用いる場合の結果のまとめ

		C2Q-QUAD	Ci7-QUAD
Fairness （ポイント）	ORG	21.50	13.45
	提案手法	23.63	14.05
ORG に対する トータルスループット向上率 (%)	ORG	-	-
	提案手法	-1.73	**+0.35**

5 考察

4 コアを使う場合における性能低下に関する考察

本評価の結果より，2 コアを使う場合においては Fairness，トータルスループットが ORG よりも向上していたが，4 コアを使う場合においてはそれがみられる例が極端に少なくなった．以下，これに関して考察する．

利用できるコアをすべて使う方針の問題

本提案手法は，競合が生じないようにスケジューリングすることを目標としていたが，利用できるコアは全て使ってプロセスを動かす方針であったため，同時に実行させるプロセスが多くなることで全体の共有資源の要求率が高まり，競合が生じてしまう可能性が高い．

また本提案手法では，予測値を順位付けし相対的に要求率が大きいプロセスと小さいプロセスを選択するようにしていたものの，絶対的な要求率でみた場合には共有資源のバンド幅を超えていた可能性がある．このような状況下では競合が生じる．

以上より，どのようにプロセスを選択しても要求率が共有資源のバンド幅を超えてしまう場合，競合が生じないようにするためには，コアにプロセスを割り当てないようにすることが考えられる．もしくは文献 [10, 17] の手法のようにコアをプロ

セス実行のために使用せずプリフェッチのためなどに使用する必要もあると考えられる．

不適切なアフィニティ設定

不適切なアフィニティ設定も原因の一つとして考えられる．本提案手法では，マイグレーションによる性能低下を防ぐためにアフィニティ設定を行っていた．そのような条件下であるコアで相対的に要求率が低いプロセスと，他のコアで相対的に要求率が高いプロセスを選択したとしても競合が発生する可能性がある．

例えば図 33 のような状況がある．これは，2 コアを使って 4 つのプロセスをスケジューリングする場合を示している．それぞれのプロセスの共有資源の要求率は図の通りであるとすると，本提案手法のスケジューリングアルゴリズムではコア 0 で相対的に要求率が低いプロセス B，コア 1 で相対的に要求率が高いプロセス C を選択される．しかしこのとき要求率は $30 + 90 = 120\%$ となり共有資源のバンド幅を超え，競合が生じる．プロセスごとの要求率がこの値でしばらく変わらず，1 実行フェーズの時間よりも長く続く場合にはこのアフィニティ設定は不適切であるといえる．このような状況では，マイグレーションを起こしてプロセス B とプロセス C を入れ替えることによって要求率を 100%以内に保つことができる．よって適切にマイグレーションさせることによって性能を向上させることができると考えられる．

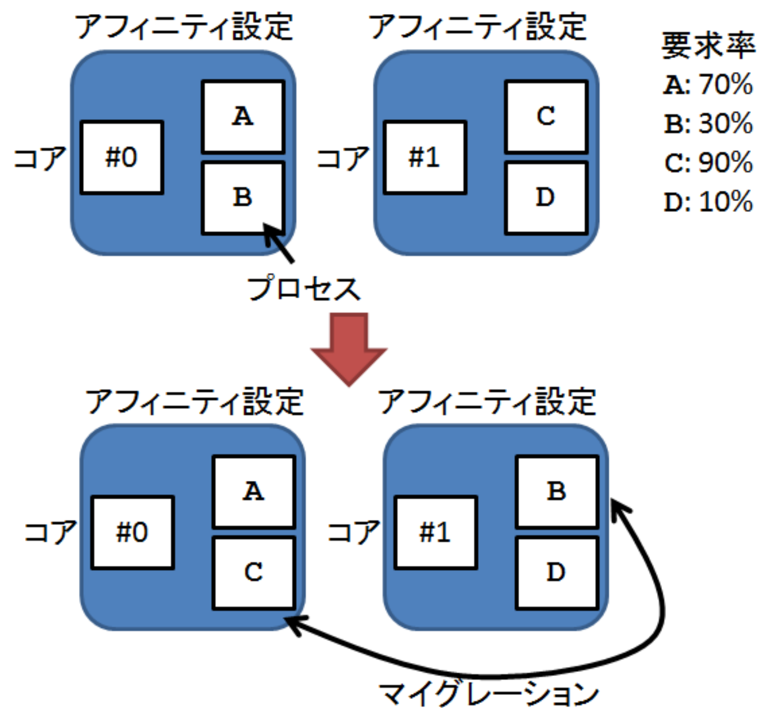

図 33: 不適切なアフィニティ設定をマイグレーションにより解決する例

6 関連研究

共有資源の競合は，CMP の性能を低下させる大きな要因の一つであるため，CMP における共有資源の競合に着目した研究は広く行われている．

プロセスの実行スピード制御

文献 [18, 19] では，プロセスの実行スピードを調節することで共有資源の競合を緩和させ，Fairness やトータルスループットの向上させた「トラクションコントロール実行」が提案されている．本提案手法は，プロセスの実行スピードを変化させることはせず，各実行フェーズごとに実行するプロセスを選択してスケジューリングを行うという点でこの研究と異なる．

スレッド数の動的な最適化

文献 [15] では，スレッド数増加によるメモリバス競合を防ぐためにスレッド数を動的に制御し最適化する手法である “Feedback-Driven Threading”が提案されている．本提案手法は，スレッド数を制御するのではなくプロセスの共有資源の要求度

に応じて各実行フェーズごとにスケジューリングを行うことにより競合を緩和させるという点でこの研究と異なる．

SMT を共有資源の対象にした命令フェッチポリシー

文献 [11] では，対象とする共有資源を Simultaneous Multi-Threading (SMT) に対応したプロセッサとして，スループットや Fairness を改善する手法が提案されている．本提案手法で対象とする共有資源は，SMT プロセッサではなく CMP のキャッシュやメモリバスである点がこの研究と異なる．

キャッシュミスを減らすためのプリフェッチを行う手法

文献 [10, 17] では，主記憶より共有キャッシュにプリフェッチすることのみを行うスレッドを別に用意し，メモリウォール問題による性能低下を緩和するヘルパースレッド実行方式が提案されている．本研究は，プロセス実行以外の目的のためにはスレッドを作らずに全てのスレッドをプロセス実行に用いるという点でこの研究と異なる．

競合に着目した性能予測モデル

文献 [21, 22] では，共有資源の競合に着目し，CMP を搭載した多くのマシンで性能を正確にモデリングする手法が提案されている．本提案手法は，共有資源の競合に着目し競合を減らして性能向上を目指すスケジューリング手法である点でこの研究と異なる．

7 おわりに

7.1 まとめ

CMP の共有資源の競合による性能低下により，CMP が本来持っている演算能力を発揮しきれないことを改善するため，本研究では共有資源の要求率を予測して競合を生じさせないようにする実行フェーズ単位でのプロセススケジューリング手法を提案した．本提案手法は，OS 上のプロセススケジューラに改良を加えて，RUN キュー上のプロセスを実行フェーズ単位で共有資源の要求率を予測し，競合を生じさせないようにスケジューリングする手法である．

本提案手法を実現するためのスケジューリングアルゴリズムをスケジューラに導入した．

本提案手法を組み込んだスケジューラの評価を，クアッドコアの CMP を搭載した 2 種類の実機を用いて，4 種類の評価環境で行った．また本提案手法におけるパラメータを様々に変えて評価を行った．この評価では，OS を拡張することはせずに，本提案手法を組み込んだスケジューラの役割を果たすプログラムである Manipulator を設計して，これを用いた．

評価を行った結果、2 つのコアを用いる場合には効果が出ていたが，4 つのコアを用いる場合にはスケジューラのオーバヘッドや，不適切なアフィニティ設定の問題などの要因により効果が上がらなかった．

7.2 今後の課題

今後の課題として以下のものがあげられる．

- マイグレーションの導入
  評価結果を考察することにより推測された不適切なアフィニティ設定より，この問題を解決するためにはスケジューリングアルゴリズム中にマイグレーションを導入することが課題となる．

  あるコアに共有資源の要求率の高いプロセスが多くアフィニティ設定されている場合は，他のコアの要求率が低いプロセスとをマイグレーションさせて要求率の高いプロセスを分散させるようにするとこの問題を解決できると考えられる．

- パラメータチューニング
  本論文の評価においては，まだ一部のパラメータ設定やカウンタしか用いてい

ないため，評価に含まれるパラメータ設定が最適なものであるという確証は得られていない．よって，様々な設定を用いて多くの実験を行い，パラメータを最適化するべきである．

- 性能予測モデルの組み込み
  文献 [21, 22] では，共有資源の競合に着目し，性能をモデリングする手法が提案されている．これによれば，競合が性能に与える影響を正確にモデリングできる．よって，文献 [21, 22] の統計的学習手法によるモデルを本提案手法に組み込むことが今後の課題としてあげられる．

- 多コア CMP での評価
  CMP に搭載されるコア数は年々増加している．各 CMP ベンダにより，8 つのコアを搭載するオクタコア CMP などの開発も行われているが [14]，本提案手法の評価は 4 コアの場合までしか行われていない．したがって，より多くのコアが搭載された CMP でも本提案手法が有効であることを示すことができるように本提案手法を多コア CMP 上で評価することが必要である．

- OS への実装
  また，本提案手法はまだ OS のスケジューラとして実装できていないため，OS に実装することも今後の課題である．

  OS のプロセススケジューリングに基づく手法を利用する場合には直接的に実行プロセスを制御できるためオーバヘッドが少なくなるため，更なる性能向上が期待できる．

  その際には，本提案手法では実行するプロセスの優先度は全て一定であると仮定しているため，実際のプロセスの優先度との兼ね合いを考慮しなければならない．

謝辞

本論文を執筆するにあたって，ご多忙中にもかかわらず終始熱心なご指導をいただきました高性能コンピューティング学講座の本多弘樹教授，近藤正章准教授，平澤将一助教に深く御礼申し上げます.

また，本論文執筆中に有益な助言を下さった本講座の OB である渡邊啓正博士（現在，HPC システムズ株式会社技術開発本部），大島聡史博士（現在，東京大学情報基盤センター）に深く感謝いたします.

研究室での生活についてアドバイスを下さいました本講座博士後期課程の田邊浩志さん，片山典彦さん，本講座の先輩である板倉佑輔さん（現在，三洋電機株式会社），下田和明さん（現在，東芝ソリューション株式会社），増永明剛さん（現在，日本電気通信システム株式会社），佐藤知明さん（現在，ルネサステクノロジ株式会社），史晨悦さんに感謝の意を表します.

本研究を進めるにあたり，激励の言葉を下さいました本講座の同期である鈴木翔平くん，松本優人くん，本講座の 1 年生の佐々木信くん，富田翔くん，中尾信高くん，西川優くん，穂園智哉くん，山下良くん，山藤友紀くん，久保彰利くん，鈴木淳也くんに心より御礼申し上げます.

そして，大学院での生活において大きな心の支えとなっていただき，お世話になりました本専攻情報システム基礎学講座の星原隼人さん，門屋彰くん，青木玄明くん，石津圭太くん，小林昌平くんに深く感謝いたします.

参考文献

[1] Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S. and Sancho, J.C.: A Performance Evaluation of the Nehalem Quad-Core Processor for Scientific Computing, *Parallel Processing Letters*, Vol. 18, No. 4, pp. 453–469 (2008).

[2] Barrodo, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R. and Verghese, B.: Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing, *Proceedings of the 27 th Annual International Symposium on Computer Architecture*, pp. 282–293 (2000).

[3] Bentley, B.: Validating the Intel® Pentium® 4 Micropocessor, *Proceedings of the 38 th Annual Design Automation Conference*, pp. 244–248 (2001).

[4] Chandra, D., Guo, F., Kim, S. and Solihin, Y.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture, *Proceedings of the 11 th International Symposium on High-Performance Computer Architecture*, pp. 340–351 (2005).

[5] Dorsey, J., Searles, S., Ciraula, M., Johnson, S., Bujanos, N., Wu, D., Braganza, M., Meyers, S., Fang, E. and Kumar, R.: An Integrated Quad-Core Opteron™ Processor, *Proceedings of the IEEE International Solid-State Circuits Conference 2007*, pp. 102–103 (2007).

[6] George, V., Jahagirdar, S., Tong, C., Smits, K., Damaraju, S., Siers, S., Naydenov, V., Khondker, T., Sarkar, S. and Singh, P.: Penryn: 45-nm Next Generation Intel® Core™ 2 Processor, *Proceedings of the IEEE Asian Solid-State Circuits Conference 2007*, pp. 14–17 (2007).

[7] Kim, S., Chandra, D. and Solihin, Y.: Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture, *Proceedings of the 13 th International Conference on Parallel Architectures and Compilation Techniques*, pp. 111–122 (2004).

[8] Kondo, M., Sasaki, H. and Nakamura, H.: Improving Fairness, Throughput and Energy-Efficiency on a Chip Multiprocessor through DVFS, *ACM SIGARCH Computer Architecture News*, Vol. 35, No. 1, pp. 31–38 (2007).

[9] Koufaty, D. and Marr, D.T.: Hyperthreading Technology in the Netburst Microarchitecture, *Proceedings of the 36 th Annual International Symposium on Microarchitecture*, pp. 56–65 (2003).

[10] Lu, J., Das, A., Hsu, W.-C., Nguyen, K. and Abraham, S.G.: Dynamic Helper Threaded Prefetching on the Sun UltraSPARC® CMP Processor, *Proceedings of the 38 th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 93–104 (2005).

[11] Luo, K., Gummaraju, J. and Franklin, M.: Balancing Throughput and Fairness in SMT Processors, *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 164–171 (2001).

[12] Marsan, M.A., Balbo, G., Conte, G. and Gregoretti, F.: Modeling Bus Contention and Memory Interference in a Multiprocessor System, *IEEE Transactions on Computers*, Vol. C-32, No. 1, pp. 60–72 (1983).

[13] Pollack, F.J.: New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies, *Proceedings of the 32 nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 2–3 (1999).

[14] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T. and Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing, *Proceedings of the 42 nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 10–21 (2009).

[15] Suleman, M.A., Qureshi, M.K. and Patt, Y.N.: Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-threaded Workloads on CMPs, *Proceedings of the 13 th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 277–286 (2008).

[16] Wu, Q. and Mencer, O.: Evaluating Sampling Based Hotspot Detection, *Proceedings of the 5 th International Workshop on Applied Reconfigurable Computing*, pp. 28–39 (2009).

[17] 今里賢一, 福本尚人, 井上弘士, 村上和彰: 演算/メモリ性能バランスを考慮したCMP 向けヘルパースレッド実行方式の提案と評価, 電子情報通信学会技術研究報告, Vol. 108, No. 28, pp. 75–80 (2008).

[18] 近藤正章, 佐々木広, 中村宏: 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, 情報処理学会研究報告, ARC-174, pp. 79–84 (2007).

[19] 近藤正章, 佐々木広, 中村宏: 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, 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, pp. 111–123 (2008).

[20] 近藤正章, 中村宏: CMP 向け動的電源電圧・周波数制御手法, 情報処理学会論文誌, コンピューティングシステム, Vol. 48, No. SIG13 (ACS19), pp. 260–269 (2007).

[21] 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).

[22] 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).