Vol.2012-ARC-200 No.13
Vol.2012-OS-121 No.13
2012/5/8

命令グループのワーキング・セットに着目したキャッシュ・マネジメント

浅見 公輔[†1]　倉田 成己[†1]　塩谷 亮太[†2]　三輪 忍[†1]　五島 正裕[†1]　坂井 修一[†1]

概要：
プロセッサの性能向上のために，キャッシュ・ミスを減らす努力が必要不可欠である．しかし，従来のキャッシュの容量効率は依然として低く，容量効率を上げるためのキャッシュ・マネジメントが必要である．我々は，著しいキャッシュの汚染が起こることによってキャッシュの容量効率が下がることに着目し，キャッシュで汚染が起こっている様子の可視化を行った．

本稿では，キャッシュの容量効率に対処する新しいキャッシュ・マネジメントを提案する．1 つ以上のメモリ・アクセス命令をグループ化した命令グループ という新しい考え方を導入し，命令グループごとに，そのワーキング・セット・サイズの大きさでキャッシュ・パーティショニングを行う．命令グループが利用できるキャッシュ・サイズを，そのワーキング・セット・サイズに限定することで，従来のキャッシュ・パーティショニングよりも効率的にキャッシュを利用することができる．

提案手法の予備評価として，命令グループごとの Utility-based Cache Partitioning の実装と評価を行った．共有キャッシュとなっている L2 キャッシュを命令グループごとに Utility-based Cache Partitioning することによって，最大で 52.1%，平均で 3.9%性能が改善した．

1. はじめに

プロセッサの周波数は向上を続けているが，その一方で主記憶のレイテンシは相対的に増大している [1]．主記憶のレイテンシがプロセッサの性能に与える影響は大きく，主記憶のレイテンシを緩和するためにコアと主記憶の間にキャッシュが搭載される．近年では，キャッシュを多階層化することで，広がり続けるプロセッサと主記憶の性能ギャップを埋める傾向にある．

それでもなお，キャッシュ・ミス・ペナルティは未だにプログラムの実行時間を増大させる主要因である．プロセッサの性能向上のためにキャッシュ・ミスを減らす努力が一層求められている．

しかし，従来のキャッシュは，その限られた容量を有効に利用することができていない．これは，キャッシュ・ラインのリプレースメントが，そのラインの再利用性の高さに関わらず行われるためである．結果としてキャッシュ上に再利用性の低いラインが増えるため，従来のキャッシュの容量効率は低い [2]．

我々は著しいキャッシュの汚染が起こることによって，キャッシュの容量効率が下がることに着目する．そして，キャッシュで汚染が起こっている様子の可視化を行う．

キャッシュの容量効率を改善させてプロセッサの性能を上げるためのキャッシュ・マネジメントが多数研究されている．本稿では，我々はキャッシュの容量効率に対処する新しいキャッシュ・マネジメント を提案する．1 つ以上のメモリ・アクセス命令をグループ化した命令グループという考え方を導入し，命令グループごとにワーキング・セット・サイズの大きさのキャッシュ領域を割り当てるキャッシュ・パーティショニングを提案する．提案手法を用いることで，従来のキャッシュ・パーティショニングよりも効率的にキャッシュ を利用することができる．

本稿は以下のような構成になっている．2 章でキャッシュが著しい汚染を受けてその容量効率を低下させている様子を可視化した結果を示す．3 章で，本稿で提案する命令グループごとのキャッシュ・パーティショニングの説明を行う．そして，4 章で提案手法の予備評価結果と考察を述べ，5 章で本稿のまとめと今後の課題を述べる．

2. キャッシュの汚染

キャッシュを著しく汚染する典型的なプログラムの例として，ストリーミング・アクセス を行うプログラムがある．

従来のキャッシュでは，LRU を用いてリプレースするキャッシュ・ラインを選ぶことが多い．しかし，LRU では

†1　東京大学大学院情報理工学系研究科
†2　名古屋大学大学院工学研究科

表 1 鬼斬弐の主なパラメータ

ベンチマーク	SPEC CPU2006 401.bzip2+429.mcf
構成	1 コア，2 スレッド，SMT
L1D	4-way 64B line, 32KB
L1I	4-way 64B line, 32KB
L2	8-way 64B line, 4MB

ストリーミング・アクセスのような著しいキャッシュの汚染から既存のキャッシュ・ラインを救い出すことができない．そのため，共有キャッシュ上でストリーミング・アクセスを行うスレッドが実行されると，他スレッドのキャッシュ・ミス数が増加し，プロセッサ全体の性能に悪影響を及ぼす．

実際にストリーミング・アクセスによってキャッシュが汚染され，キャッシュ・ラインが次々と追いだされていることを，キャッシュの様子の可視化を行うことで示す．キャッシュの様子の可視化を行うために，プロセッサ・シミュレータ鬼斬弐 [3] で SPEC CPU2006 の 401.bzip2 と 429.mcf を 1 コア，2 スレッドの SMT の構成で実行させた．主なパラメータは**表 1** のようになっている．

図 1 は，ベンチマークを双方 1G サイクルスキップした後，149M サイクル実行した時の様子から，1M サイクルごとのキャッシュの様子の可視化を行ったものである．

それぞれの図において，左上の大きい長方形 2 つ分が L1 データ・キャッシュ，右上の大きい長方形 2 つ分が L1 命令キャッシュ，下の横長の長方形 8 つ分が L2 キャッシュの様子を表している．そして，それぞれのキャッシュ内の小さい長方形 1 つ分がキャッシュ・ライン 1 つ分を表している．赤系の色のキャッシュ・ラインが bzip2 がアロケートしたキャッシュ・ラインであり，青系の色のキャッシュ・ラインが mcf がアロケートしたキャッシュ・ラインである．アクセスがないキャッシュ・ラインの彩度はサイクルが進むごとに徐々に下がるようになっている．つまり，彩度の高いキャッシュ・ラインは最近アクセスがあったキャッシュ・ラインである．

150M サイクル実行するまでは，L1 データ・キャッシュも L2 キャッシュも，mcf のキャッシュ・ライン（青色）よりも bzip2 のキャッシュ・ライン（赤色）のほうが多い．しかし，151M サイクル実行すると L1 データ・キャッシュは mcf のキャッシュ・ラインで埋め尽くされてしまっており，152M サイクル実行後は L2 キャッシュも mcf のキャッシュ・ラインがほとんどを占めている．ここから，150M サイクル実行後，mcf のプログラムのフェーズが変わった瞬間，mcf はストリーミング・アクセスを始め，わずか 2M サイクルほどで mcf のキャッシュ・ラインが bzip2 のキャッシュ・ラインの大部分をリプレースする．

mcf によってアロケートされるキャッシュ・ラインが非常に多いため，LRU を用いても bzip2 を mcf によるキャッシュの汚染から救い出すことはできず，mcf によるキャッシュの汚染が bzip2 の性能に悪影響を与える．

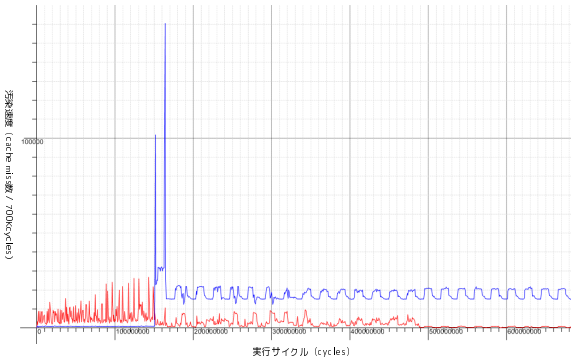


図 2 bzip2 と mcf の時間あたりのキャッシュ・ミス数の増減

図 2 は，bzip2 と mcf の時間あたりのキャッシュ・ミス数の増減を表したグラフである．

横軸は命令スキップ直後からの実行サイクル数，縦軸は 700K サイクルあたりのキャッシュ・ミス数を示している．赤色のグラフが bzip2 のグラフで，青色のグラフが mcf のグラフである．図 2 からも，実行サイクル数が 150M サイクルに達した時にプログラムのフェーズが変わり，mcf の時間あたりのキャッシュ・ミス数が激増したことがわかる．mcf のフェーズが変わる前は，mcf より bzip2 のほうが，時間あたりのキャッシュ・ミス数が多いが，フェーズが変わった後は bzip2 より mcf のほうが多くなる．

以上のように，従来のキャッシュではプログラムによってキャッシュが著しく汚染され，その容量効率を低下させることが現象として起こっていることがわかる．

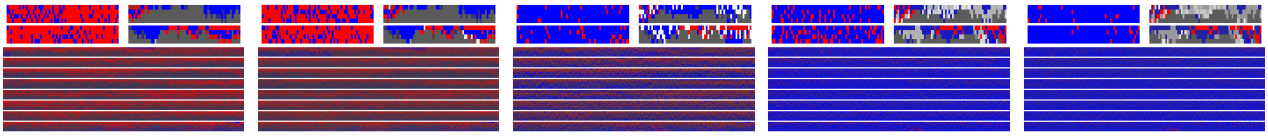
3. 提案手法

2 章で述べたように，LRU による制御がうまく働かないのは，キャッシュの汚染速度が，その他の有用なラインへの参照頻度を上回っている場合である．このような場合，従来のキャッシュ・パーティショニングでは，高速に汚染を行うスレッドを個別のパーティションに隔離することにより，有用なラインが追い出されてしまうことを防いでいる．これに対し，本稿では命令グループ毎のキャッシュ・パーティショニングを提案する．提案手法では，一定の命令グループ毎に，それがアクセスするワーキング・セットの大きさに応じてパーティショニングを行う．

以下の 3.1 節では，まず単一の命令についてワーキング・セットの概念を説明し，それに基づいたパーティショニングについて述べる．次の 3.2 節では，これを命令グループ毎に行う方法について説明する．

3.1 単一命令のワーキング・セットとパーティショニング

まず，単一の命令のワーキング・セットについて，**図 3** と**図 4** を用いて説明する．図 3 は，2 次元画像データに対するフィルタ処理を想定した擬似コードである．2 次元画



149M サイクル　150M サイクル　151M サイクル　152M サイクル　153M サイクル

図 1　149M〜153M サイクル実行した際のキャッシュの様子

```
for(i<SIZEY) {
  for(j<SIZEX) {
    for(-1<y<1) {
      for(-1<x<1) {
        load array[i+y][j+x];
      }
    }
  }
}
```

図 3　アクセスを行う命令

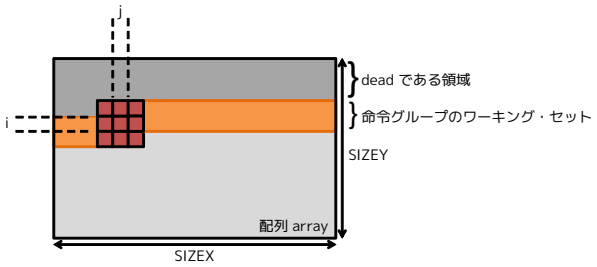


図 4　アクセスの様子

像データは，2 次元配列 array 上に展開されており，座標 (x, y) のピクセル・データは array[y][x] によってアクセスすることができる．同図のフィルタは，配列上の近傍 3×3 ピクセルを参照するものであり，近傍 3×3 の処理を 2 次元配列上の位置 i について行っている．このフィルタ処理によるメモリ・アクセスの様子を表したのが図 4 である．図上の矩形は，2 次元配列 array を 2 次元上に表したものであり，アドレスは左から右に，上から下に増加する．フィルタ処理は図上において左上から始まり，左から右に進む．また，右端まで処理が達すると，1 ピクセル下の左端に戻る．

ある命令におけるキャッシュ上のワーキング・セットとは，キャッシュを追い出されるまでに将来にわたってその命令によってアクセスを受けるラインの集合である．これを図 4 を例として説明する．図 4 上の赤い部分はループの最内周 2 つによってアクセスされる領域である．また，オレンジ色の領域は，この処理を行う命令 load におけるワーキング・セット である．1 ピクセルについての処理である赤い四角は左から右に，上から下に移動していく．このため，図上のオレンジ色の領域は，この処理によって近い将来にアクセスを受ける領域，すなわちワーキング・セットとなっている．このオレンジ色の領域より上部の濃い灰色の部分については，この後アクセスを受けることがないため，deadblock となる．

キャッシュ上ではこのワーキング・セットのみが性能向上に寄与するラインであり，それを外れた deadblock については，将来アクセスを受けることがないため，性能向上に寄与しない．そこで，この命令ごとに対し，キャッシュ上に確保できる容量のパーティショニングを行う．パーティショニングによる制限は，ワーキング・セットのサイズに基づいて行う．たとえば上記の場合，パーティションの容量をオレンジ色のワーキング・セットに設定することにより，そこから溢れた deadblock は優先的に置き換え対象となるため，キャッシュを汚染せずに速やかに切り離される．

従来のスレッドごとのパーティショニングでは，スレッド内の命令同士を区別してないため，高速に汚染を行う命令により，同一スレッド内の有効なワーキング・セットが追い出されてしまっていた．これに対し，提案手法ではより細粒度にパーティショニングを行っているため，汚染を最低限に防ぎつつ，スレッド内の他のワーキング・セットを有効に使うことができる．

3.2　命令グループ毎のワーキング・セット

実際のプログラムでは，各ワーキング・セットは複数の命令群によってアクセスされることが多い．たとえば，図 3 ではループによって近傍ピクセルの処理を行っていたが，実際には高速化のために**図 5** のようにアンローリングされる事が普通である．このような場合，複数の命令が同一の配列に対してアクセスするため，ワーキング・セットもこれらの命令群において共有される．

このため，提案手法では同一のワーキング・セットにア

```
for(i<SIZEY) {
  for(j<SIZEX) {
    load array[i-1][j-1];
    load array[i-1][j];
    load array[i-1][j+1];
    load array[i][j-1];
    load array[i][j];
    load array[i][j+1];
    load array[i+1][j-1];
    load array[i+1][j];
    load array[i+1][j+1];
  }
}
```

図 5　単一のワーキング・セットにアクセスを行う命令グループの例

クセスする命令群をまとめて扱う．具体的には，同一のラインにアクセスする命令を検出し，それらの命令群ごとにまとめてパーティショニングを行う．

3.3 キャッシュ階層ごとのワーキング・セット

ワーキング・セットはキャッシュの階層ごとに異なるため，階層ごとに異なる制御を行う必要がある．以下ではこのことについて，例を用いて説明する．

今，上記のフィルタ処理の例における画像のサイズが幅 1024× 高さ 768 であり，1 ピクセルが 1 バイトであったとする．すると，ワーキング・セットのサイズは高さ方向に数ピクセル分の領域となるため，たかだか数 KB 内におさまる．このため，L1 データ・キャッシュ上にワーキング・セットを保持することにより，うまく働くことができる．

L2 キャッシュは，通常 L1 キャッシュよりも数倍以上大きな容量を持つため，これらのラインを同様にして保持することができる．しかし，これらのラインへのアクセスは L1 キャッシュ上で完結しているため，L2 キャッシュ上では追い出されるまでにアクセスされることはない．このため，L2 キャッシュからみたワーキング・セットは 0KB となる．

3.4 提案手法の工程

提案手法のキャッシュ・マネジメントは以下のような工程で行う．

(1) 命令グループを作成する
(2) 命令グループのワーキング・セット・サイズを観測する
(3) 命令グループの割り当てキャッシュ・サイズを調節してキャッシュ・パーティショニングを行う

このうち，(1) について 3.4.1 項で，(3) について 3.4.2 項で詳しく説明する．

3.4.1 命令のグループ分け

命令グループは次のように作成することを提案する．

- 同じキャッシュ・ラインに触った命令群を命令グループとする．
- 1 つの命令は 1 つの命令グループに所属する．

図 6 は，命令がアクセスしたラインの情報からどのように命令グループを作るかを表した図である．右矢印はサイクルの流れを表し，“命令 0 がライン A にアクセスする → 命令 1 がライン B にアクセスする → 命令 2 がライン A にアクセスする → ···” のようにアクセスが続くことを表す．

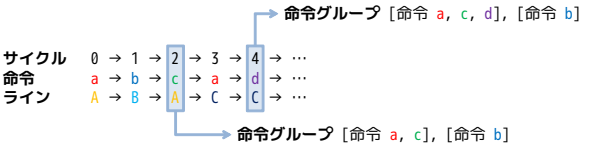


図 6 命令グループの作り方

このようなアクセスがあった場合，命令グループは次のように作る．まずサイクル 2 の時，命令 a と命令 c は同じライン A にアクセスしたため，同じ命令グループに入れられる．命令 b は A とは異なるライン B にアクセスしたため，命令 a と命令 c とは異なる命令グループに属する．サイクル 3 の時，命令 a はライン C にアクセスする．そのため，サイクル 4 でライン C にアクセスする命令 d は，命令 a と命令 c と同じ命令グループに所属させる．

このように，命令と，その命令がアクセスしたラインの情報から命令をグループ分けする．

3.4.2 命令グループごとのキャッシュ・パーティショニング

従来のキャッシュ・パーティショニングでは，セット・アソシアティブ・キャッシュをウェイ方向に分割していた [4], [5]．これは，従来のキャッシュ・パーティショニングではキャッシュをスレッドごとに分割していたからであり，共有キャッシュ上で同時に動作させるスレッドの数が，セット・アソシアティブ・キャッシュの連想度よりも小さいためである．

一方で，提案手法では，スレッドごとではなく，命令グループごとのキャッシュ・パーティショニングを行う．同時に管理する命令グループの数がキャッシュの連想度を超えた場合，命令グループ同士で限られたウェイを取り合うため，結果として命令グループ同士で競合を起こしてしまう問題が発生する．

命令グループ同士の競合を防止するために，提案手法ではキャッシュのセット方向への分割を行う．セット方向に分割することで，ウェイ方向に分割するときよりも割り当てキャッシュ・サイズを細かく調節することができるため，キャッシュの割り当てサイズをワーキング・セット・サイズにより近づけることが可能となる．

セット方向に分割するにあたり，従来よく採用されてきたセット・アソシアティブ・キャッシュではなく，キャッシュに V-way Cache[6] や，ZCache[7] のような，セットごとにウェイ数を動的に変更することができるキャッシュを採用することで，セット方向へのパーティショニングを実現することができると考えられる．

4. 予備評価

4.1 予備評価モデルの実装

提案手法の予備評価として，命令グループごとの割り当てキャッシュ・サイズ を Utility-based Cache Partitioning(UCP)[5] のアルゴリズムを使って決め，キャッシュ・パーティショニングを行うモデルの評価を行った．

UCP では，元々スレッドごとの割り当てキャッシュ・サイズを動的に求めるアルゴリズムが提案されていた．スレッドごとの UCP では，次のようにスレッドごとの割り当てキャッシュ・サイズを動的に求める．

共有キャッシュ上で動作させるスレッドの数のタグ・ア

表 2 ポジションごとのヒット・カウンタの値 (上) とその累積 (下)

← more recent

	MRU			LRU
スレッド A	100	50	25	0
スレッド B	200	150	50	25

	U_1	U_2	U_3	U_4
スレッド A	100	150	175	175
スレッド B	200	350	400	425

レイを用意する．タグ・アレイはそれぞれ，共有キャッシュと同じ数のエントリを持っており，共有キャッシュと独立してタグ・ラインのアロケーションとリプレースメントを行う．さらに，各スレッドごとに，キャッシュのウェイの数のカウンタを用意する．このカウンタを使って，タグ・アレイへのアクセスがヒットした際，そのヒットが LRU から MRU のうちどのポジションのウェイへのヒットであったかを観測して，ポジションごとに何回ヒットしたかを計測する．

共有キャッシュへのアクセスの際，アクセスを行ったスレッドのタグ・アレイにもアクセスを行う．タグ・アレイへのアクセスがヒットしたならばカウンタを更新して，ミスしたならばタグ・ラインのアロケーションとリプレースメントを行う．

ポジションごとのヒット・カウンタの値は一定サイクル毎に読み出す．そして，MRU から LRU まで，最後にアクセスがあった時間順にウェイを並べ，MRU から数えたウェイ数 (i) とそのポジションまでのヒット・カウンタの累計値 (U_i) を計算する．仮にウェイ数が 4 のキャッシュであった場合，ポジションごとのヒット・カウンタの値と，その U_i の例を**表 2** に示す．このヒット・カウンタの累計値 U_i は，スレッドが使用できるキャッシュ・サイズが i ウェイ分である場合の，タグ・アレイへのヒット数である．UCP は，タグ・アレイへのヒット数が最大となるウェイ数の組み合わせで，スレッドにキャッシュ・サイズを割り当てる．表 2 の例の場合，4 ウェイのうち，スレッド A に 1 ウェイ割り当て (U_1)，スレッド B に 3 ウェイ割り当て (U_3) た時に，タグ・アレイへのヒット数が最大となるため，スレッド A に 1 ウェイ，スレッド B に 3 ウェイキャッシュ領域を割り当てる．

このようにして，一定サイクル毎にパーティション・サイズを変更する．

このアルゴリズムを命令グループに同じように適用し，キャッシュ・パーティショニングを行うモデルをプロセッサ・シミュレータ鬼斬弐上に実装した．タグ・アレイのポジション・ヒット・カウンタを用いた割り当てキャッシュ・サイズの決定は，厳密に最適解を求めるのではなく，UCP の論文で提案されている Lookahead Algorithm を使って求める．

表 3 予備評価に用いたプロセッサのパラメータ

ベンチマーク	SPEC CPU2006 全ベンチマーク +470.lbm
命令セット	Alpha
構成	2 コア，キャッシュは L2 のみ共有
フェッチ幅	4
発行幅	int:2, fp:2, mem:2
実行ユニット	int:2, fp:2, mem:2
命令ウィンドウ	int:32, fp:16, mem:16
BTB	4-way 2K エントリ
gshare	32K エントリ PHT 10bit グローバル分岐履歴
RAS	8 エントリ
L1D	4-way 64B-line 16KB, 2 サイクル
L1I	4-way 64B-line 16KB, 2 サイクル
L2(shared)	16-way 64B-line 1MB, 10 サイクル
主記憶	200 サイクル

4.2 予備評価結果

4.2.1 パラメータと評価モデル

予備評価は**表 3** のパラメータを使い，次の 5 通りのモデルを，共有キャッシュである L2 キャッシュに実装して評価をとった．

Base セット・アソシアティブ・キャッシュの共有キャッシュに何もキャッシュ・マネジメントを行わないモデル

UCP セット・アソシアティブ・キャッシュの共有キャッシュにスレッドごとの UCP を実装したモデル

V-way 共有キャッシュに V-way Cache を用いたモデル

ThreadP 共有キャッシュである V-way Cache にスレッドごとの UCP を実装したモデル

GroupP 共有キャッシュである V-way Cache に命令グループごとの UCP を実装したモデル

評価するベンチマークは，SPEC CPU2006 の全ベンチマーク 29 本と 470.lbm の組み合わせ 29 通りを実行した．470.lbm はストリーミング・アクセスを繰り返してキャッシュを激しく汚染する典型的なプログラムであり，lbm による共有キャッシュの汚染を防止することで性能を大きく向上させることができると予想される．そのため，lbm と全ベンチマークの組み合わせ 2 スレッドを実行して評価を行った．ベンチマークはどちらも 1G 命令スキップさせた後，2 スレッド合計 250M 命令実行させて評価している．

4.2.2 最適再パーティショニング間隔の検討

まず，UCP，ThreadP，GroupP の 3 通りのモデルにおいて，再パーティショニングを行う間隔の設定を変え，Base を 1 とおいた時の全ベンチマークの相対 IPC の平均を測定した．**図 7** に UCP モデルの相対 IPC 平均，**図 8** に ThreadP モデルの相対 IPC 平均，**図 9** に GroupP モデルの相対 IPC 平均を示す．

グラフより，UCP モデルは再パーティショニング間隔 1M サイクルの時，ThreadP モデルは再パーティショニ

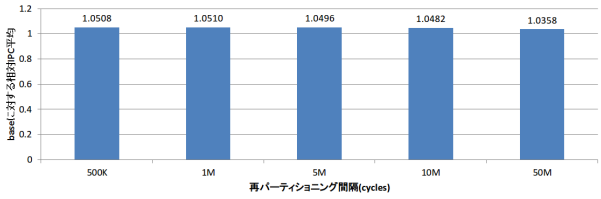


図 7 UCP モデルの Base に対する相対 IPC 平均

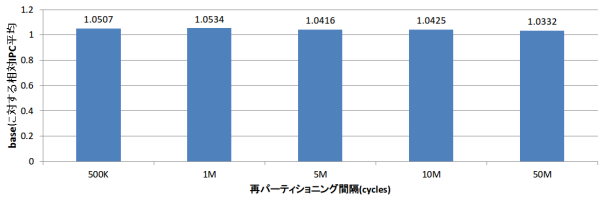


図 8 ThreadP モデルの Base に対する相対 IPC 平均

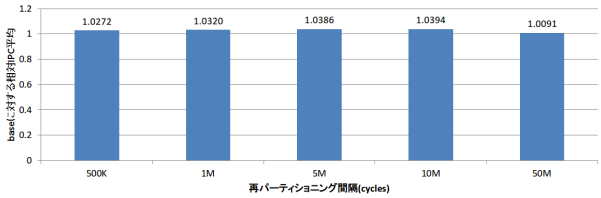


図 9 GroupP モデルの Base に対する相対 IPC 平均

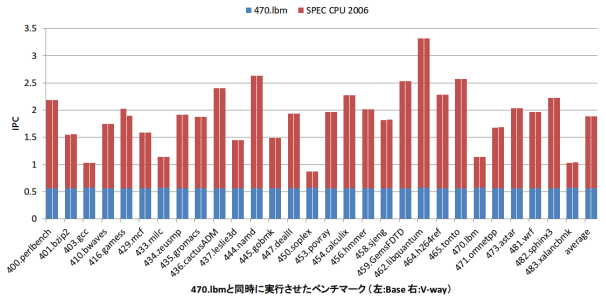


図 10 Base モデルと V-way モデルの IPC 比較

ング間隔 1M サイクルの時，GroupP モデルは再パーティショニング間隔 10M サイクルの時，最も平均での性能向上率が高いことがわかる．そのため，以下ではこの再パーティショニング間隔での結果を用いることで考察を行う．

4.2.3 V-way モデルの IPC

ThreadP モデルと，GroupP モデルは，どちらも V-way Cache に対してキャッシュ・パーティショニングを行うものである．そのため，まず共有キャッシュを，セット・アソシアティブ・キャッシュから V-way Cache に変えることでどれほど IPC に影響を与えるのかを評価した結果が**図 10** である．

結果から，V-way Cache にすることで，最大では，483.xalancbmk と lbm の組み合わせで，Base に比べて 1.4%合計 IPC が向上した．逆に 416.gamess と lbm との組み合わせで，Base に比べて 6.4%IPC が低下しており，全ベンチマークの平均では 0.2%合計 IPC が低下していた．

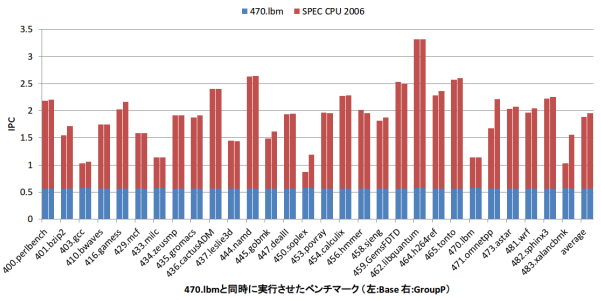


図 11 Base モデルと GroupP モデルの IPC 比較

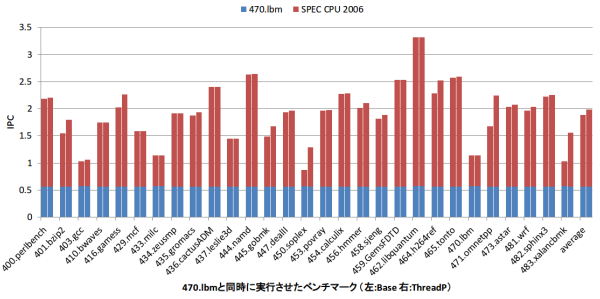


図 12 Base モデルと ThreadP モデルの IPC 比較

ここから，共有キャッシュを 16-way のセット・アソシアティブ・キャッシュから V-way Cache にしても大きく合計 IPC が変化することはないことがわかる．

4.2.4 GroupP モデルの IPC

続いて，GroupP モデルの IPC の評価結果を**図 11** に示す．

結果から，483.xalancbmk と lbm の組み合わせの時，Base モデルと比較したときの合計 IPC 向上率が最大となっており，52.1%合計 IPC が向上した．一部合計 IPC が低下した組み合わせがあり，最も低下した組み合わせは 456.hmmer と lbm の組み合わせで，3.3%低下していた．全ベンチマークの Base と比べた合計 IPC 向上率の平均は 3.9%であった．

動作させたスレッドごとの IPC をみてみると，lbm の IPC は全ベンチマーク平均で，Base と比べて 0.1%低下しているにすぎず，各ベンチマークごとにみても，ほとんど IPC は低下していない．結局，命令グループごとに UCP を行って命令グループが使用できるキャッシュ・サイズを制限しても，lbm の性能は著しく悪化しない．

4.2.5 ThreadP モデルの IPC

次に，ThreadP モデルの合計 IPC の結果を**図 12** に示す．

結果から，483.xalancbmk と lbm の組み合わせの時，Base モデルと比較したときの合計 IPC 向上率が最大となっており，GroupP モデルと同じく 52.1%合計 IPC が向上した．全ベンチマークの，Base と比べた合計 IPC 向上率の平均は 5.3%であった．

4.2.6 UCP モデルの IPC

GroupP モデル，ThreadP モデルは V-way Cache に UCP

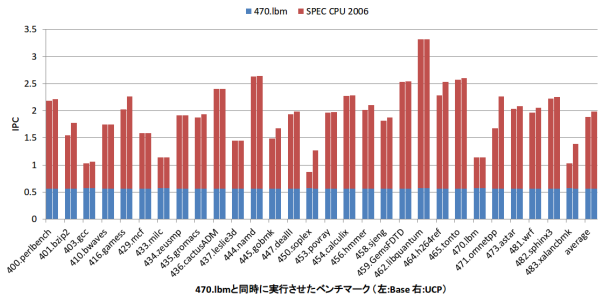


図 13 Base モデルと UCP モデルの IPC 比較

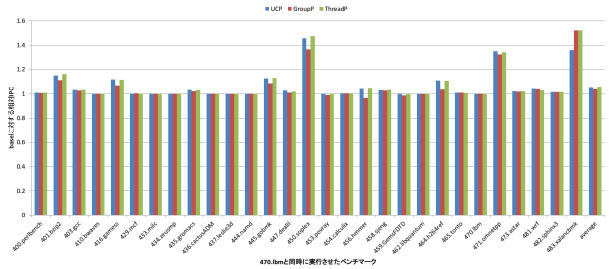


図 14 UCP，GroupP，ThreadP の Base に対する相対 IPC

を実装したモデルだが，UCP モデルはセット・アソシアティブ・キャッシュに UCP を実装したモデルである．UCP モデルの IPC の結果を**図 13** に示す．

この結果から，450.soplex と lbm の組み合わせの時，Base モデルと比較したときの合計 IPC 向上率が最大となっており，45.8%合計 IPC が向上した．全ベンチマークの，Base と比べた合計 IPC 向上率の平均は 5.1%であった．

4.2.7 UCP，GroupP，ThreadP の Base に対する相対 IPC

4.2.4 項，4.2.5 項，4.2.6 項のまとめとして，UCP モデル，GroupP モデル，ThreadP モデルの，Base モデルを 1 とした時の，各ベンチマークの相対 IPC の評価結果を**図 14** に示す．

結果から，ほとんどのベンチマークで GroupP モデルよりも ThreadP モデルのほうが Base モデルに対して合計 IPC の向上率が高く，全ベンチマーク平均では 1.4%高かった．

UCP モデルと ThreadP モデルの，Base モデルに対する合計 IPC 向上率を比べると，483.xalancbmk と lbm の組み合わせを動作させた時を除くとほとんど合計 IPC 向上率は変わらなかった．全ベンチマークで合計 IPC 向上率の平均をとると，ThreadP モデルのほうが UCP モデルよりも 0.24%高い結果となった．483.xalancbmk と lbm の組み合わせの時は，ThreadP モデルのほうが 16.3%合計 IPC 向上率が高かった．

5. おわりに

我々は，著しいキャッシュの汚染が起こることでキャッシュの容量効率が下がることに着目し，従来のキャッシュで起こっている汚染現象の可視化を行い，その結果を本稿で示した．

続いて，命令グループのワーキング・セットに着目したキャッシュ・パーティショニング を提案した．提案手法を用いることで，従来のキャッシュ・パーティショニングよりも，キャッシュを効率的に利用することができる．

提案手法の予備評価として，命令グループごとの Utility-based Cache Partitioning の実装と評価を行った．共有キャッシュとなっている L2 キャッシュを命令グループごとに Utility-based Cache Partitioning することによって，最大で 52.1%，平均で 3.9%性能が改善した．

今後は，命令グループのワーキング・セットを観測する手法を提案することが課題となる．その後，観測したワーキング・セット・サイズでキャッシュ・パーティショニングするモデルのシミュレータへの実装と，性能の評価を行う予定である．

参考文献

[1] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture: A Quantitative Approach 4th Edition*, Morgan Kaufmann Publishers (2006).
[2] Liu, H., Ferdman, M., Huh, J. and Burger, D.: Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency, *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pp. 222–233 (2008).
[3] 塩谷亮太，五島正裕，坂井修一：プロセッサ・シミュレータ「鬼斬弐」の設計と実装，先進的計算基盤システムシンポジウム SACSIS2009，pp. 120–121 (2009).
[4] Dybdahl, H., Stenström, P. and Natvig, L.: A cache-partitioning aware replacement policy for chip multiprocessors, *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pp. 22–34 (2006).
[5] Qureshi, M. K. and Patt, Y. N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pp. 423–432 (2006).
[6] Qureshi, M. K., Thompson, D. and Patt, Y. N.: The V-Way Cache: Demand Based Associativity via Global Replacement, *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pp. 544–555 (2005).
[7] Sanchez, D. and Kozyrakis, C.: The ZCache: Decoupling Ways and Associativity, *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pp. 187–198 (2010).