Vol.2012-ARC-201 No.14
2012/8/1

命令グループごとのキャッシュ・パーティショニングの予備評価

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：
共有キャッシュで同時に動作するスレッドの数は近年増加傾向にあり，共有キャッシュに対するマネジメントの必要性が高くなってきている．キャッシュ・ラインのリプレースメント・ポリシとして LRU が採用されることが多いが，共有キャッシュでは LRU による制御がうまく働かないことがあり，スレッド間競合を招く．本稿では命令ごとに必要とするキャッシュ・サイズが異なっていることに着目し，命令グループのワーキング・セットの大きさにパーティション・サイズを合わせるキャッシュ・パーティショニングを提案する．提案手法により，従来手法のスレッドごとのキャッシュ・パーティショニングよりも効果的にキャッシュを利用できる．提案手法の予備評価として，命令グループごとに Utility-based Cache Partitioning を行うモデルと，命令グループに対してそれが必要としているキャッシュ・サイズを最大で利用可能にするモデルの実装と評価を行った．結果，マルチスレッド環境で，前者のモデルでは最大 53.9%，平均 3.6%，後者のモデルでは最大 54.6%，平均 3.5%性能が向上した．

1. はじめに

近年では，スレッド・レベル並列性を利用したマルチスレッド・プロセッサが普及している．1 つのチップに複数のコアを搭載した CMP(Chip Multi Processor) は，ハイエンド・プロセッサから組み込みプロセッサに至るまで，広く採用されている．また，1 つのコア上で複数のハードウェア・スレッドを同時に実行する，SMT(Simultaneous Multi Threading) をサポートするプロセッサも増えてきている．

マルチスレッド・プロセッサでは，普通スレッド間でキャッシュが共有される．共有キャッシュでは，複数のスレッドからアクセスが集中し，しばしばスレッド間でキャッシュ・ラインの競合が発生する．このようなスレッド間の競合によって，プロセッサ全体の性能低下が引き起こされる [1].

共有キャッシュで同時に動作するスレッドの数は近年増加傾向にある．そのため，共有キャッシュに対するマネジメントの必要性がますます高くなってきている．

従来のキャッシュでは，キャッシュ・ラインのリプレースメント・ポリシとして LRU が用いられることが多い．しかし，共有キャッシュでは，LRU ではスレッド間の競合に対処することはできない．これは，memory intensive なスレッドが実行されると，キャッシュが memory intensive なスレッドの，再利用性の低いキャッシュ・ラインによって著しく汚染されるためである．再利用性の高いキャッシュ・ラインが，再利用性の低いキャッシュ・ラインによってリプレースメントされてしまい，キャッシュ・ミスを招く．結果として，プロセッサ全体の性能が低下する．

Memory intensive なスレッドによる共有キャッシュの汚染に対処するためのキャッシュ・マネジメントとして，スレッドごとのキャッシュ・パーティショニング [2], [3], [4] が研究されてきた．スレッドごとのキャッシュ・パーティショニングとは，共有キャッシュ上で動作するスレッドごとに，利用できるキャッシュ領域を制限する手法である．これによって，著しくキャッシュを汚染するスレッドを隔離することができるため，汚染を防止することができる．

本稿では，1 つ以上のメモリ・アクセス命令群をまとめ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，利用できるキャッシュ領域を制限するキャッシュ・パーティショニングを提案する．

図 1 は，SPEC CPU2006 のベンチマーク lbm において，命令グループごとに利用できる L2 キャッシュの大きさを変化させて 1M サイクル実行した時の，命令グループごとの L2 キャッシュ・ヒット数を示したグラフである．L2 キャッシュはフル・アソシアティブ・キャッシュとし，64B

[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

© 2012 Information Processing Society of Japan

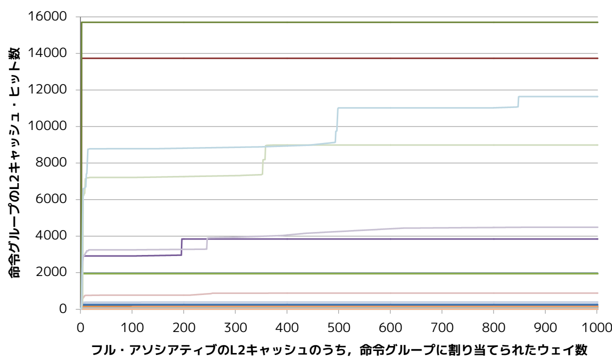


図 1 lbm の命令グループごとに，利用できる L2 キャッシュ・サイズを変化させた時の L2 キャッシュ・ヒット数

単位のウェイを割り当てることで，利用できるキャッシュ・サイズを変化させている．図上方の 2 つの命令グループは，割り当てウェイ数に対する L2 キャッシュ・ヒット数が，割り当てウェイ数 1 で飽和している．そのため，2 つの命令グループについては，利用できるウェイ数が 1 でも，それ以上でも，L2 キャッシュ・ヒット数は変わらない．一方で，図中程の 4 つの命令グループは，割り当てウェイ数 200～900 でキャッシュ・ヒット数が飽和する．ここで，前者の 2 つの命令グループに対しては割り当てるキャッシュ・サイズ は少なくて良いが，後者の 4 つの命令グループに対しては他命令グループよりも多めにキャッシュ・サイズを割り当てなければ，キャッシュ・ミスを多く発生させてしまう．

このように，命令グループごとの必要なキャッシュ・サイズは大きく異なっている．そのため，命令グループごとにキャッシュ・パーティショニングを行えば，従来より効果的にキャッシュ・パーティショニングが行える．

本稿は以下のような構成になっている．続く 2 章で，共有キャッシュ上で LRU による制御がうまく働かないことがあることをキャッシュを可視化することで示す．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モデルを説明し，5 章で，予備評価結果と考察を述べる．最後に，6 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

従来のキャッシュの多くは，LRU を用いでラインのリプレースメントを行っている．しかし，複数のスレッドによってキャッシュが共有される場合，LRU による制御がうまく働かず，スレッド間競合を多く引き起こす．Memory intensive なスレッドを共有キャッシュ上で動作させると，memory intensive なスレッドの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り，LRU による制御がうまく働かない．

本章では，実際に共有キャッシュ上で LRU による制御がうまく働かないことがあることを，キャッシュの様子の可視化を行うことで示す．例として，SPEC CPU2006[5] のベンチマークの中の bzip2 と mcf を同時に実行させ，memory intensive なプログラムである mcf がキャッシュを著しく汚染し，LRU による制御がうまく働かないことを示す．

キャッシュの可視化は，プロセッサ・シミュレータ鬼斬弐 [6] を使用したシミュレーションの結果を用いて行った．プロセッサの構成は 1 コア，2 スレッドの SMT とし，L1 キャッシュはデータ・命令ともに 4-way のセット・アソシアティブでサイズは 32KB，L2 キャッシュは 8-way のセット・アソシアティブでサイズは 4MB である．

図 2 はベンチマークを双方 1G 命令スキップした後，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を可視化したものである．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 つ分がライン 1 つ分を表している．赤系の色のラインが，bzip2 がアロケートしたラインであり，青系の色のラインが，mcf がアロケートしたラインである．アクセスがない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，ラインの彩度が高いほど，そのラインに最近アクセスがあったことを表している．

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 のライン（青色）よりも bzip2 のライン（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・キャッシュが mcf のラインで埋め尽くされ，152M サイクル実行すると，L2 キャッシュも mcf のラインがほとんどを占めるようになる．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 がメモリに対してストリーミング・アクセスを始め，その後わずか 2M サイクルほどで mcf のラインによって bzip2 のラインの大部分がリプレースメントされる．mcf は次から次へと新しいラインをアロケートするため，そのラインの再利用性は低く，キャッシュは激しく汚染される．

このように，memory intensive なスレッドである mcf がキャッシュを汚染する速度が，bzip2 がラインを参照する頻度を上回るために，LRU による制御がうまく働かず，キャッシュが mcf のラインによって著しく汚染されることがわかる．mcf によるキャッシュの汚染は bzip2 のキャッシュ・ミスを招き，プロセッサの性能に悪影響を与える．

図 3 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 1G 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，青色のグラフが mcf のグラフである．図 3 からも，150M サイクルほど実行してプログラムのフェーズが変わった後

© 2012 Information Processing Society of Japan

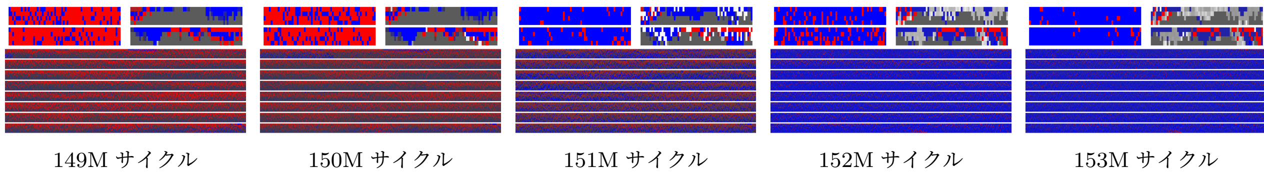


図 2 149M〜153M サイクル実行した際のキャッシュの様子

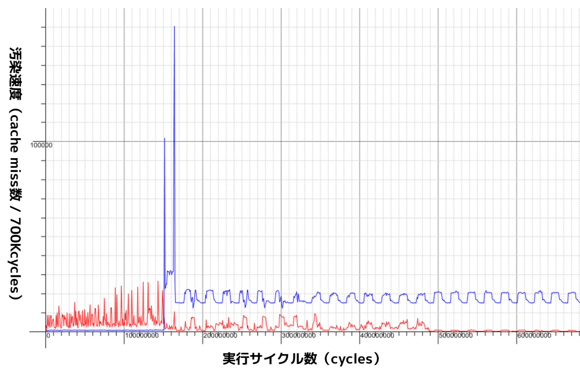


図 3 bzip2 と mcf の時間あたりのキャッシュ・ミス数の増減

は，mcf は bzip2 よりも時間あたりのキャッシュ・ミス数，つまりラインのリプレースメント頻度が高く，mcf の汚染速度が bzip2 の参照頻度を上回る状態が続くことがわかる．結果として，この共有キャッシュの LRU による制御はうまく働いていない．

3. 命令グループごとのキャッシュ・パーティショニング

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，従来のスレッドごとのキャッシュ・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ごとのキャッシュ・パーティショニングを提案する．提案手法では，命令グループごとに，それがある時間中に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．

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 4** と**図 5** を用いて説明する．

図 4 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 (x, y) 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 5 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．図 5 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．

従来のキャッシュでは，キャッシュ上にないデータにアクセスするたびにラインのリプレースメントとアロケートが行われてきた．赤い部分が配列 array 内の図の位置にある時，図上オレンジ色の領域と灰色の領域を合わせた部分がキャッシュにすでにアロケート済みである．図上灰色の領域は，この後アクセスを受けることがない deadblock である．一方，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である．このオレンジ色の領域は，ループの最内周 3 つを 1 回実行する時間での，load 命令のワーキング・セットである．

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 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 load 命令に対して，キャッシュ上に確保できる容量を制限するパーティショニングを行う．パー

```
for(y<SIZEY) {
  for(x<SIZEX) {
    for(-1<=yy<=1) {
      for(-1<=xx<=1) {
        load array[y+yy][x+xx];
      }
    }
  }
}
```

図 4 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

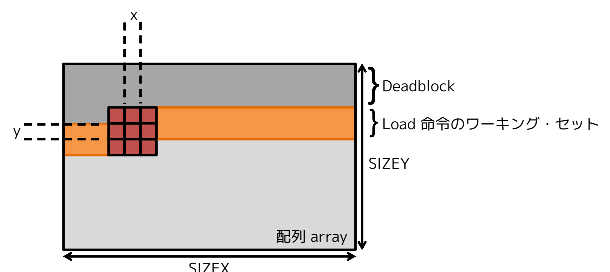


図 5 フィルタ処理アクセスの様子

© 2012 Information Processing Society of Japan

ティショニングによる容量の制限は，ワーキング・セットのサイズに基づいて行う．例えば上記の場合，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 は優先的にリプレースメント対象となるため，キャッシュを汚染せずに速やかに切り離される．

このように，命令のパーティション・サイズをある時間中にアクセスするワーキング・セットに合わせることで，キャッシュを有効に利用することができる．

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．例えば，図4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 6** 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．このプログラムで命令一つ一つに対して，そのワーキング・セットの大きさに基づいてキャッシュを割り当てると，命令群に対してはワーキング・セット・サイズ × 命令の数の大きさのキャッシュが割り当てられることになる．

それよりも，命令群をまとめて一つのグループとして取り扱い，グループに対して，ワーキング・セットの大きさのキャッシュを割り当てたほうがキャッシュを効率的に利用できる．

このため，提案手法では同一のワーキング・セットにア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．命令群の分け方については，3.5 節で詳しく述べる．

3.3 キャッシュ階層毎のワーキング・セット

命令のワーキング・セットは，観測する時間が等しくとも，キャッシュの階層ごとに異なっている．そのため，パーティショニングの際には，キャッシュの階層毎に異なる制御を行う必要性がある．本節ではこのことについて，例を用いて説明する．

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

図 6 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

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・セットの大きさは高さ方向に数ピクセル分の領域となるため，高々数 KB 内に収まる．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で，フィルタ処理は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，フィルタ処理の例において，キャッシュへのアクセスは 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，リプレース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

提案手法は，以下のような 3 つの工程に分けることができる．

(1) 命令グループの作成
(2) 命令グループに割り当てるキャッシュ・サイズの決定
(3) セット単位でのキャッシュ・パーティショニング

このうち，(1) について 3.5 節で，(3) について 3.6 節で詳しく説明する．

3.5 命令グループの作成

命令グループは，命令がどのラインにアクセスしたか，という情報を使って以下のような方針で作成する．

- 同一のラインにアクセスする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属する

3.5.1 命令グループ作成の方法

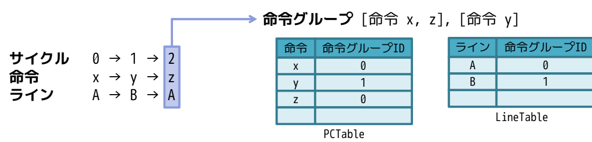
具体例を使って命令グループの作成法を説明する．

図 7(a) では，サイクルが進むごとに “命令 x がライン A にアクセスする → 命令 y がライン B にアクセスする → 命令 z がライン A にアクセスする” と，アクセスが続く例を示している．この時，命令 x と命令 z が同じライン A に対してアクセスしており，命令 y はライン B に対してアクセスしている．そのため，命令 x と命令 z は同じ命令グループに所属させ，命令 y はそれとは異なる命令グループに所属させる．

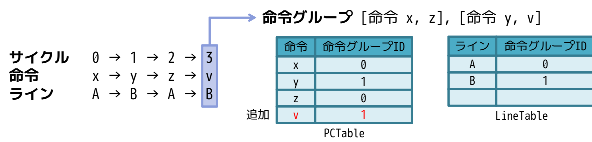
メモリ・アクセス命令が実行されるたびに，

(1) 命令 (プログラム・カウンタ) ごとに，どの命令グループに属しているか記録するテーブル (PCTable)
(2) ラインごとに，どの命令グループがアクセスしているか記録するテーブル (LineTable)

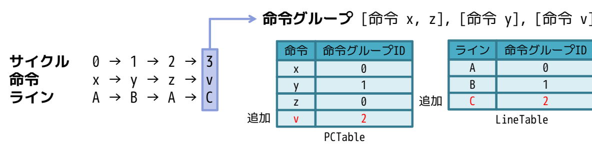
の 2 つのテーブルが参照され，メモリ・アクセス命令のグ



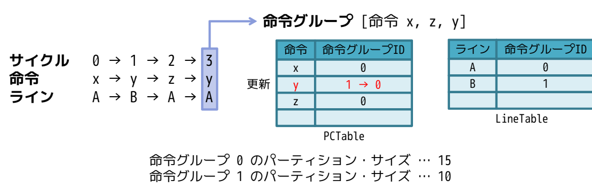
(a) アクセス例とその時作成される命令グループ



(b) (a) の後命令 v がライン B にアクセスした時



(c) (a) の後命令 v がライン C にアクセスした時



(d) (a) の後命令 y がライン A にアクセスした時

図 7 命令グループ作成の例

ループ分けが行われる．そして，グループ分けの結果をもとに，テーブルを更新する．図 7(a) の左のようにアクセスが続くと，PCTable と LineTable には同図右のように記録される．命令グループは，個々に振り当てられた ID によって管理される．

メモリ・アクセス命令が実行されたら，命令がアクセスしたラインの情報をもとに，2 つのテーブルを以下のように参照する．

- PCTable を参照し，命令（プログラム・カウンタ）に対応する命令グループが記録されているか調べる
- LineTable を参照し，ラインに対応する命令グループが記録されているか調べる

両テーブルに対する参照のヒット/ミスの結果を使って，命令グループを作成する．

1 つのテーブルに対して参照がヒットし，もう一方のテーブルに対して参照がミスした場合，ヒットしたテーブルから得られた命令グループ ID が，その命令のグループ ID となる．命令のグループ ID が決まったら，その ID によってミスしたテーブルに項目を追加する．例えば，**図 7(b)** では，図 7(a) の例の後，命令 v がライン B に対してアクセスがあった例を表している．この例では，LineTable に対する参照はヒットし，命令 v の命令グループ ID は 1，という情報が得られるが，PCTable に対する参照はミスする．そのため，PCTable に命令 v の項目を追加し，命令グループの ID1 を記録する．

両テーブルに対して参照がミスした時，その命令が所属する新しい命令グループを作成する．**図 7(c)** の例では，PCTable に対する参照も，LineTable に対する参照もミスするため，命令 v から成る命令グループが作成され，ID 2 を振り当てられる．

両テーブルに対して参照がヒットした場合，得られた ID が両テーブルで同じ時，命令グループ ID は読み出された値を使い，テーブルの更新は行わない．しかし，両テーブルから読み出された ID が異なっている時，それぞれのグループ ID のキャッシュ・パーティション・サイズを読み出し，パーティション・サイズが大きい方をその命令のグループ ID とする．そして，その ID でテーブルを更新する．**図 7(d)** では，PCTable から読み出されたグループ ID が 1 で，LineTable から読み出されたグループ ID が 0 だった例である．ここで，命令グループ 0 のパーティション・サイズが 15 で，命令グループ ID1 のパーティション・サイズが 10 であるとする．この時，命令 y は，パーティション・サイズの大きい ID0 の命令グループに属すると決める．そして，PCTable に記録されている命令 y に対応する命令グループ ID を，1 から 0 に更新する．

3.5.2 メモリの動的割り当てへの対処

LineTable に対して，ラインの登録は物理アドレスで行う．ここで，スレッドが malloc や free を使ってメモリの動的割り当てと開放を繰り返すと，関連性のない命令同士が同じ物理アドレスに対してアクセスを行う可能性がある．この関連性のない命令同士で命令グループを作成してしまうことを防ぐため，LineTable は定期的にクリアする必要性がある．

3.6 セット単位でのキャッシュ・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 [2], [3]．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[7]や，ZCache[8]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，セット単位でのキャッシュ・パーティショニングを実現することができる．

3.7 命令データへの対処

キャッシュにアロケートされるラインは，メモリ・アクセス命令の実行によってアロケートされるデータのラインだけでなく，命令それ自体のデータのラインも含まれている．提案手法では，この命令データもパーティショニング対象にする．

フェッチするスレッドごとに，命令データのライン・グループを考える．例えば共有キャッシュ上でスレッドAとスレッドBが動作する場合，スレッドAがフェッチする命令データ用のライン・グループと，スレッドBがフェッチする命令データ用のライン・グループを考慮する．そして，命令グループの場合と同様に，この命令データ用のライン・グループに対しても，割り当てるキャッシュ・サイズの決定とパーティショニングを行う．

4. 予備評価

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 を決めるアルゴリズムが異なる2つのモデル, UCPモデル（**UCP-1**）とMaxモデル（**Max-1**）をプロセッサ・シミュレータ鬼斬弐上に実装して評価を行った．UCPモデルについては4.1節で，Maxモデルについては4.2節で説明を行う．この両モデルについて，実際には，求めた割り当てキャッシュ・サイズを一律1.5倍した値を割り当てたモデル（**UCP-1.5**, **Max-1.5**）の性能も評価した．後者の時，命令グループごとの割り当てウェイ数の合計がキャッシュ・サイズ を超えることがある．そのため，4.3節で述べるようにリプレースメント対象の選択を行う．

予備評価対象とするのは，シングルスレッド実行時とマルチスレッド実行時の各モデルのプロセッサの性能である．シングルスレッド実行時の評価では，L2キャッシュをキャッシュ・パーティショニングした時と，L1データ・キャッシュとL2キャッシュを共にキャッシュ・パーティショニングした時の性能を計測した．マルチスレッド実行時では，2コアのCMPの構成でL2キャッシュのみを共有キャッシュとしてキャッシュ・パーティショニングした時と，1コア2スレッドのSMTの構成でL1データ・キャッシュとL2キャッシュを共にキャッシュ・パーティショニングした時の性能を計測した．

表 1　予備評価に用いたプロセッサのパラメータ

命令セット	Alpha 21264
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

表 2　予備評価に用いたキャッシュのパラメータ．L2をパーティショニングする時（上），L1とL2をパーティショニングする時（下）

L1D	4-way 64B-line 16KB, 2 サイクル
L1I	4-way 64B-line 16KB, 2 サイクル
L2	FullAssoc, 64B-line 1MB, 10 サイクル
主記憶	200 サイクル

L1D	FullAssoc, 64B-line 16KB, 2 サイクル
L1I	4-way 64B-line 16KB, 2 サイクル
L2	FullAssoc, 64B-line 1MB, 10 サイクル
主記憶	200 サイクル

予備評価に用いたプロセッサのパラメータは**表 1**の通り，キャッシュのパラメータは，**表 2**の通りである．上の表はL2キャッシュをパーティショニングする時のパラメータで，下の表はL1データ・キャッシュとL2キャッシュをパーティショニングする時のパラメータである．今回は，V-way CacheやZCacheを用いたセット単位でのパーティショニングの予備評価として，フル・アソシアティブ・キャッシュを用いてウェイ単位でのパーティショニングを行った．

L1データ・キャッシュの再パーティショニング間隔は100Kサイクル，L2キャッシュの再パーティショニング間隔は1Mサイクルとした．

比較対象とするモデルとして，表2のパラメータのキャッシュを使い，キャッシュ・パーティショニングを行わないでLRUによる制御を行うBaseモデル（**Base**）と，スレッドごとにUCPを行う従来手法のモデル（**Thread**，マルチ・スレッド実行時のみ）も実装し，評価を行った．

4.1 UCPモデル

UCPモデルは，Utility-based Cache Partitioning(UCP)[3]のアルゴリズムを使って命令グループごとの

© 2012 Information Processing Society of Japan

割り当てキャッシュ・サイズを決めるモデルである．UCP は，セット・アソシアティブ・キャッシュを，スレッド毎に，ウェイ単位で，動的にパーティショニングする手法である．UCP では，共有キャッシュ上で動作する各スレッドごとに，キャッシュを占有できたと仮定するシミュレーションを動的に行う．シミュレーションから，スレッドごとのパーティション・サイズとキャッシュ・ヒット数の関係を得る．この関係を使って，合計キャッシュ・ヒット数が最大になるように，一定サイクルおきに実際のパーティション・サイズを決定する．

UCP のアルゴリズムを命令グループごとのキャッシュ・パーティショニング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．つまり，本来の UCP ではスレッドごとにキャッシュを占有できたと仮定するシミュレーションを行なうが，今回は命令グループごとにキャッシュを占有できたと仮定するシミュレーションを行い，合計キャッシュ・ヒット数が最大となるようにパーティション・サイズを決定する．

4.1.1 UCP のアルゴリズム

UCP のアルゴリズムを説明する．

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，共有キャッシュへのアクセスの際同時にアクセスされる．そして，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．そして，タグ・アレイへのアクセスがヒットした際，そのヒットがタグ・アレイ上の 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，ポジションごとのヒット回数を計測する．

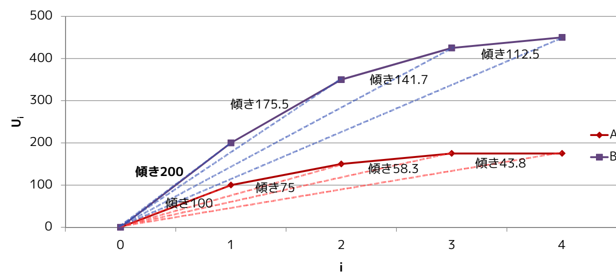
ポジションごとのヒット・カウンタの値は一定サイクルおきに読み出す．そして，MRU から LRU まで，最後にアクセスがあった時間順にウェイを並べ，MRU から数えたウェイ数 (i) とそのポジションまでのヒット・カウンタの累計値 (U_i) を計算する．仮にウェイ数が 4 のキャッシュであった場合，ポジションごとのヒット・カウンタの値と，その U_i の例を**表 3** に示す．このヒット・カウンタの累計値 U_i は，スレッドが使用できるキャッシュ・サイズが i ウェイ分である場合の，タグ・アレイへのヒット数である．UCP は，タグ・アレイへの合計ヒット数が最大となるウェイ数の組み合わせで，スレッドにキャッシュ・サイズを割り当てる．表 3 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・アレイへの合計ヒット数が最大となるため，スレッド A に 1 ウェイ分，スレッド B に 3 ウェイ分キャッシュを割り当てる．この計算を，一定サイクルおきに行い，パーティション・サイズを決定する．なお，ヒット・カウンタの値は，パーティション・サイズを計算した直後に半分にし，値の半分を引き継ぐ．

表 3 ポジションごとのヒット・カウンタの値 (上) とその累積 (下)

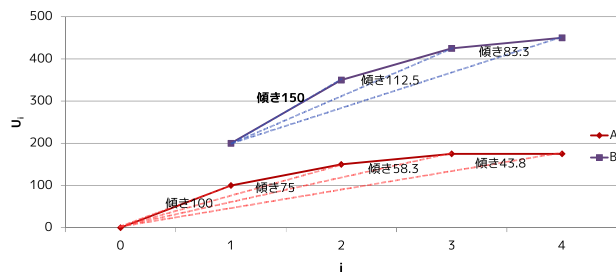
← more recent

	MRU			LRU
スレッド A	100	50	25	0
スレッド B	200	150	75	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	425	450



(a) A，B ともに割り当てられたウェイ数が 0 の時



(b) (a) の後，B にウェイが 1 割り当てられた時

図 8 表 3 の U_i のグラフ

4.1.2 Lookahead Algorithm

上記に述べたように，合計ヒット数の最大値を求める問題の最適解を求めるのは困難である．そのため，UCP の著者らによって提案されている Lookahead Algorithm を用いて近似解を求める．

Lookahead Algorithm を，再度表 3 の例を使って説明する．**図 8** は，横軸に i，縦軸に U_i をとって表 3 をグラフにしたものである．Lookahead Algorithm は，割り当てウェイ数あたりのヒット数が大きいスレッドから，ウェイを優先的に割り当てるアルゴリズムである．割り当てウェイ数あたりのヒット数とは，U の式では

$$U_{ij} = \frac{U_i - U_j}{i - j}$$

と表す値である．ここで，U_{ij} は，スレッドに割り当てるウェイ数が j から i に増えた時の，割り当てウェイ数あたりのヒット数を表している．

スレッド A，スレッド B，それぞれ割り当てウェイ数 0 の状態から計算を始める．すでに割り当てられたウェイ数

© 2012 Information Processing Society of Japan

がA，Bともに0のため，それぞれのスレッドについて，U_{10}・U_{20}・U_{30}・U_{40} を計算する．これは，**図8(a)** の点線の傾きとして，グラフ上に表されている．点線の中で，一番傾きが大きいのはスレッドBにウェイを1割り当てる時で，その傾きは200であることがわかる．そのため，スレッドBに対してウェイを1割り当てる．

続いて，スレッドAの割り当てウェイ数0，スレッドの割り当てウェイ数1の状態で，U_{ij} を再計算する．**図8(b)** の各点線の中で，一番傾きが大きいのは，スレッドBにウェイをさらに1割り当てるときで，その傾きは150である．そのため，スレッドBに対して，ウェイをさらに1割り当てる．

このように割り当てウェイ数あたりのヒット数が一番大きいスレッドと，割り当てウェイ数の組み合わせの計算と割り当てを続け，キャッシュの全領域が割り当てられるか，すべてのスレッドに対して割り当てウェイあたりのヒット数が0になるまで計算を続ける．

4.2 Maxモデル

Maxモデルのアルゴリズムでは，UCPと同様にタグ・アレイを用いたシミュレーション を行い，命令グループごとの割り当てウェイ数とキャッシュ・ヒット数の関係から割り当てキャッシュ・サイズを決める．

UCPでは，Lookahead Algorithmを用いて，全命令グループの割り当てキャッシュ・サイズがキャッシュ全体の大きさを超えないように，命令グループ が利用できるキャッシュ・サイズが融通されていた．その一方でMaxモデルのアルゴリズムでは，命令グループが利用できるキャッシュ・サイズの融通を行わず，キャッシュ・ヒット数が割り当てウェイ数に対して飽和するウェイ数，つまり命令グループが必要とするキャッシュ・サイズを，命令グループが利用できるように設定する．

図8の例では，スレッドAは，割り当てウェイ数3の時にキャッシュ・ヒット数が飽和しており，スレッドBは，割り当てウェイ数4の時にキャッシュ・ヒット数が飽和している．そのため，Maxモデルのアルゴリズムでは，スレッドAが利用できるウェイ数を3，スレッドBが利用できるウェイ数を4に設定する．

4.3 リプレースメント対象の決定

Maxモデルや，求めた割り当てキャッシュ・サイズを一律1.5倍して割り当てるモデルでは，全命令グループに割り当てられたキャッシュ・サイズの合計が，キャッシュ・サイズ全体の大きさを超えることがある．

ここで，この予備評価において，リプレースメント対象の選択は次のようにして行う．例えば，4-wayのキャッシュでスレッドAが利用できるウェイ数が3，スレッドBが利用できるウェイ数が4の時，スレッドAが利用しているウェイ数が3，スレッドBが利用しているウェイ数が1であったとする．ここで，動作中のどのスレッドに対しても，利用しているウェイ数が利用できるウェイ数以下となっている．この場合，キャッシュ全体からリプレースメント対象を選択する．もし，利用しているウェイ数が，利用できるウェイ数よりも多いスレッドがある場合，そのスレッドのパーティションがリプレースメント対象となる．

つまり，命令グループ個々に割り当てられたキャッシュ・サイズを，命令グループが利用できるキャッシュ・サイズの最大値とみなし，キャッシュ領域の厳密な隔離を行わない．他のキャッシュの利用状況によっては，利用中のキャッシュ・サイズ が割り当てキャッシュ・サイズ以下であっても，ラインがリプレースメントされることがある．

5. 予備評価結果

シングルスレッド実行時，マルチスレッド実行時の各モデルの予備評価結果を以下に示す．

5.1 シングル・スレッド実行の予備評価結果

シングルスレッド実行時，命令グループごとにキャッシュ・パーティショニング を行った時の予備評価結果を示す．測定には，SPEC CPU2006の計29本のベンチマークを使用し，500M命令スキップ後，250M命令実行して評価を行った．

L2キャッシュに対してパーティショニングを行った時のIPCの評価結果を**図9**に，L1データ・キャッシュとL2キャッシュに対してパーティショニングを行った時のIPCの評価結果を**図10**に示す．図9と図10において，各棒グラフは左から順にBase，UCP-1，UCP-1.5，Max-1，Max-1.5の各モデルのIPCを表している．

図9, 図10で，Baseと比べてIPCが比較的大きく異なったhmmerとgamessについて，BaseのIPCを1とした時の各モデルの相対IPCをまとめた表が**表 4**である．表4には，各モデルの全29ベンチマークの平均IPCについて，Baseの平均IPCを1とした時の相対平均IPCも載せている．

hmmerではとりわけBaseに対してIPCが向上し，L2

表 4 シングルスレッド実行時の, 各モデルのhmmerとgamessの, Baseに対する相対IPCと相対平均IPC

	モデル	hmmer	gamess	average
L2$を パーティショニング	UCP-1	1.072	0.7565	0.9887
	UCP-1.5	1.087	0.9943	1.001
	Max-1	1.064	0.7565	0.9884
	Max-1.5	1.020	0.9943	0.9985
L1D$ & L2$を パーティショニング	UCP-1	1.067	0.5688	0.9789
	UCP-1.5	1.069	0.9799	0.9995
	Max-1	1.036	0.6209	0.9814
	Max-1.5	1.017	0.9715	0.9972

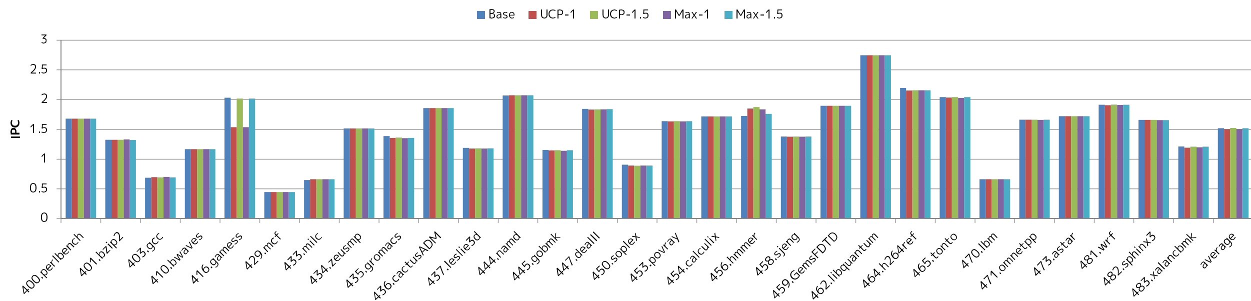


図 9 シングルスレッド実行時，L2 キャッシュに対してパーティショニングした時の IPC

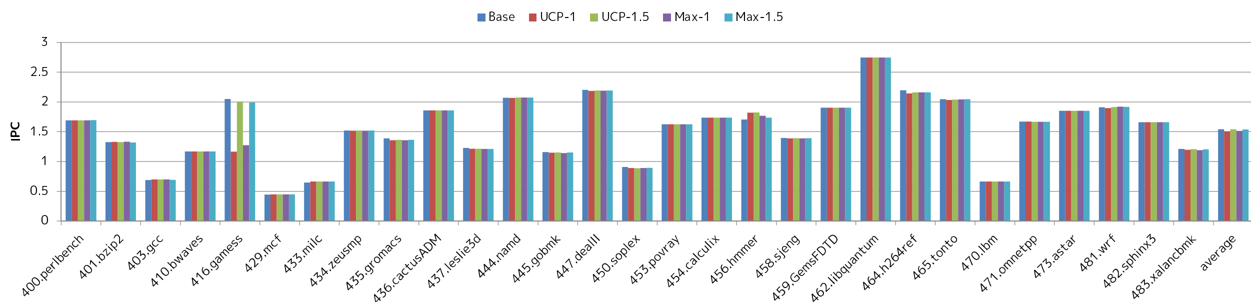


図 10 シングルスレッド実行時，L1 データ・キャッシュと L2 キャッシュに対してパーティショニングした時の IPC

```
for(i<L) {
  for(j<=i) {
    load array[j];
  }
}
```

図 11 gamess で多数実行されるロード命令のアクセスの様子の擬似コード

キャッシュのみパーティショニングした際には，UCP-1 では Base に比べて 7.2%IPC が向上し，UCP-1.5 では 8.7%IPC が向上した．

一方，gamess では UCP-1 も Max-1 も，Base に対して IPC は大きく低下した．L1 データ・キャッシュと L2 キャッシュをパーティショニングした際には，UCP-1 で Base に比べて 43.1% IPC が低下し，Max-1 では 47.9%IPC が低下した．だが，UCP-1.5 や Max-1.5 では，UCP-1 や Max-1 ほど大きな IPC の低下はみられなかった．これは，gamess では**図 11** の擬似コードに示すように配列に対してアクセスするようなロード命令が多数実行されているからである．このロード命令は，最外周のループが実行されるたびに，アクセスする配列が大きくなっていく．そのため，一定サイクルおきに割り当てウェイ数とキャッシュ・ヒット数の関係をシミュレーションしても，図 11 のようなロード命令に対しては常に必要な大きさよりも小さめにキャッシュ・サイズが割り当てられてしまい，キャッシュ・ミスを多数招いていた．アルゴリズムで求めた割り当てキャッシュ・サイズを一律 1.5 倍した場合，この小さめにキャッシュ・サイズを割り当ててしまう問題が解消されるため，大きな性能低下は起こらなかった．

なお，それ以外の多くのベンチマークではどのモデルも Base に比べて大きな IPC の違いはみられなかったため，表 4 のように，平均では大きく Base から離れた IPC になることはなかった．

5.2 マルチスレッド実行の予備評価結果

マルチスレッド実行時，命令グループごとにキャッシュ・パーティショニングを行った時の予備評価結果を示す．

測定では，SPEC CPU2006 のベンチマークのうち，gcc 以外の計 28 本のベンチマークと，SPEC CPU2006 内の lbm の組み合わせ計 28 通りの，2 スレッドの組み合わせで実行を行った．gcc と lbm の組み合わせでは，gcc の命令から成る命令グループの数が他ベンチマークと比べても多くなり，Lookahead Algorithm の計算量が大きくなったため，シミュレーションが長い時間終了しなかった．そのため，今回予備評価対象から外した．ベンチマークは双方 500M 命令スキップ後，2 スレッド合わせて 250M 命令実行することで評価を行なっている．

CMP の構成でパーティショニングを行った時の IPC の評価結果を**図 12** に，SMT の構成でパーティショニングを行った時の IPC の評価結果を**図 13** に示す．

図 12 と図 13 は，lbm ともう一方のベンチマークの，それぞれのスレッドの IPC の積み上げ棒グラフとなっている．各棒グラフは，左から順に Base，Thread，UCP-1，UCP-1.5，Max-1，Max-1.5 モデルの IPC を表している．命令グループごとのキャッシュ・パーティショニングの予備評価結果は，各実行スレッドの組み合わせの 6 本の棒グ

© 2012 Information Processing Society of Japan

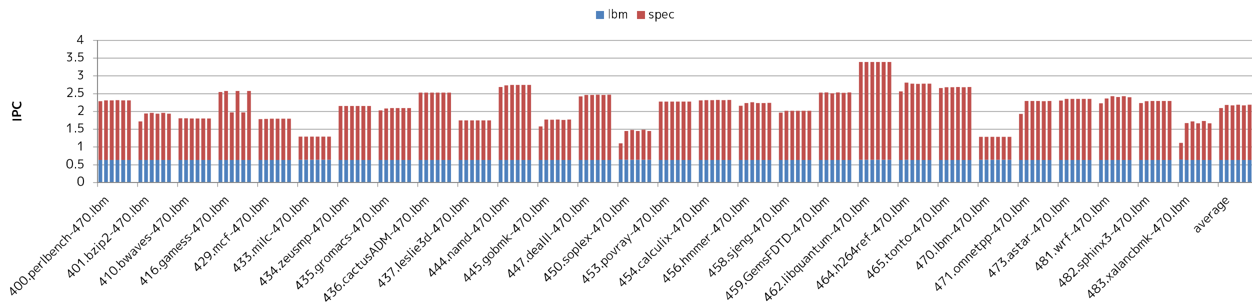


図 12 CMP 構成でマルチスレッド実行時の IPC．モデルは左から順に Base，Thread，UCP-1，UCP-1.5，Max-1，Max-1.5．

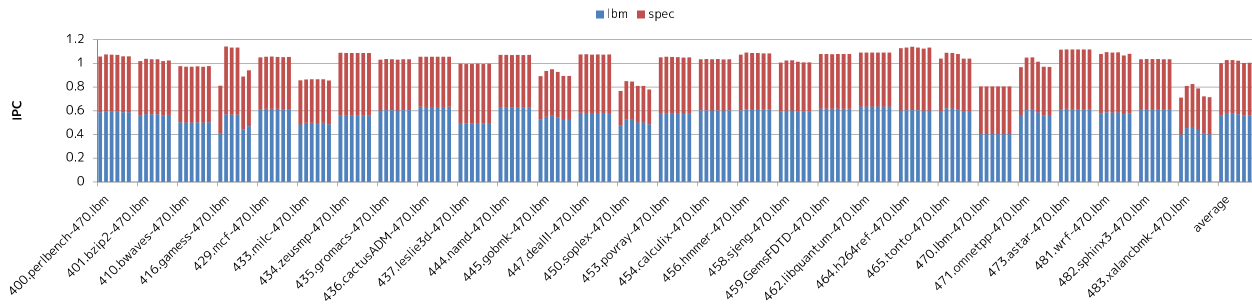


図 13 SMT 構成でマルチスレッド実行時の IPC．モデルの並び順は CMP 構成のグラフと同様．

表 5 マルチスレッド実行時の，各モデルの gamess-lbm と xalancbmk-lbm の組み合わせの，Base に対する相対合計 IPC と相対平均合計 IPC

	モデル	gamess-lbm	xalancbmk-lbm	average
CMP	Thread	1.010	1.493	1.043
	UCP-1	0.7752	1.539	1.035
	UCP-1.5	1.011	1.488	1.044
	Max-1	0.7752	1.546	1.035
	Max-1.5	1.011	1.489	1.044
SMT	Thread	1.407	1.137	1.028
	UCP-1	1.396	1.159	1.029
	UCP-1.5	1.395	1.108	1.023
	Max-1	1.094	1.014	1.005
	Max-1.5	1.159	1.003	1.006

ラフのうち，右 4 本に該当する．

図12，図13 で，Base と比べて IPC が大きく異なった gamess-lbm と xalancbmk-lbm の組み合わせについて，Base の合計 IPC を 1 とした時の各モデルの相対合計 IPC をまとめた表が**表 5** である．

CMP の構成で，UCP-1 も Max-1 も，Base に対する IPC 向上率が一番大きいベンチマークの組み合わせは，xalancbmk-lbm の組み合わせであり，UCP-1 では 53.9%，Max-1 では 54.6%，Base に対して IPC が向上した．この組み合わせでは，Thread は Base に対して 49.3%IPC が向上しているため，従来手法よりも性能が向上していると言える．

CMP の構成で，gamess-lbm の組み合わせを実行した時には，UCP-1 では 22.5%，Max-1 でも 22.5%，Base に比べて IPC が低下した．このように IPC が低下した原因は，gamess がメモリに対して 5.1 節で述べたようなアクセスをするため，常に必要な大きさよりもキャッシュ・サイズが小さめに割り当てられたからである．UCP-1.5，Max-1.5 ではこの問題が解消されるため，どちらも Base に比べて 1.1%IPC が向上した．

その一方で，SMT の構成で，gamess-lbm の組み合わせを実行すると，UCP-1 では 39.6%，Max-1 では 9.4%，Base に比べて IPC が向上している．この組み合わせの時，図 13 から，キャッシュ・パーティショニングすると，gamess の IPC だけでなく，lbm の IPC も Base に比べて大きく向上していた．

SMT の構成で xalancbmk-lbm の組み合わせを実行した時には，UCP-1 は 15.9%，Max-1 は 1.4%Base に比べて IPC が向上した．この組み合わせでは，Thread は Base に対して 13.7%IPC が向上しているため，従来手法よりも若干の性能向上がみられた．

相対合計 IPC では，表 5 のように，CMP 構成で UCP-1 モデルでは Base に比べて 3.5%，Max-1 モデルでも Base に比べて 3.5%向上した．UCP-1.5 モデルと Max-1.5 モデルでは gamess-lbm の組み合わせで大きな性能低下が起こらないため，UCP-1.5 モデル，Max-1.5 モデル共に Base に比べて 4.3%向上した．SMT 構成では，UCP-1 モデルでは Base に比べて 2.9%，Max-1 モデルでは 0.5%の向上がみられた．UCP-1.5 モデルでは Base に比べて 2.3%，Max-1.5 モデルでは Base に比べて 0.6%の向上と，CMP 構成のよ

うに，割り当てキャッシュ・サイズを 1.5 倍したことによる大きな性能の変化は起こらなかった．また，Max モデルは UCP モデルに比べて，Base からの相対合計 IPC が小さく，Base と性能はほぼ変わらなかった．

6. おわりに

本稿では，まず共有キャッシュ上では LRU による制御がうまく働かないことがあることを，キャッシュを可視化することで示した．

そして，命令ごとに必要とするキャッシュ・サイズが異なっていることに着目し，命令グループのワーキング・セットの大きさにパーティション・サイズを合わせるキャッシュ・パーティショニングを提案した．

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 を行うモデルと，命令グループが必要とするキャッシュ・サイズを測定してその大きさだけ最大で利用できるようにするモデルの実装と評価を行った．また，両モデルで求めた割り当てキャッシュ・サイズを 1.5 倍した大きさだけ，命令グループが最大で利用できるようにしたモデルも実装し，評価した．

結果，2 スレッドの CMP 構成で，命令グループごとに Utility-based Cache Partitioning を行うモデルでは，最大で 53.9%，平均で 3.6%IPC が向上した．命令グループが必要なキャッシュ・サイズを最大で利用可能なモデルでは，最大で 54.6%，平均で 3.5%IPC が向上した．

2 スレッドの SMT 構成では，命令グループごとに Utility-based Cache Partitioning を行うモデルでは，最大で 39.6%，平均で 2.9%IPC が向上した．命令グループが必要なキャッシュ・サイズを最大で利用可能なモデルでは，最大で 39.5%，平均で 2.3%IPC が向上した．

求めた割り当てキャッシュ・サイズをそのまま利用可能なキャッシュ・サイズにすると，徐々にアクセスするメモリ領域が増えるようなロード命令においてキャッシュ・ミスが多発する問題があることがわかった．これは，求めた割り当てキャッシュ・サイズを 1.5 倍した大きさを最大で利用可能にしたモデルでは問題が解消された．

今後は，命令グループごとのワーキング・セット・サイズの観測法の提案や，提案手法での使用ハードウェア量を減少させる工夫の考案が課題となる．

参考文献

[1] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely, Jr., S. and Emer, J.: Adaptive insertion policies for managing shared caches, *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 208–219 (2008).

[2] 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).

[3] 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).

[4] Suh, G. E., Devadas, S. and Rudolph, L.: A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning, *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pp. 117–128 (2002).

[5] The Standard Performance Evaluation Corporation: *SPEC CPU2006 suite http://www.spec.org/cpu2006/*.

[6] 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).

[7] 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).

[8] 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).

© 2012 Information Processing Society of Japan