命令グループごとのキャッシュ・パーティショニング

浅 見 公 輔[†] 倉 田 成 己[†] 塩 谷 亮 太[††]
五 島 正 裕[†] 坂 井 修 一[†]

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，従来手法よりも効率的にキャッシュを利用でき，性能を向上させられる．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 をフル・アソシアティブ・キャッシュに対して行うモデルを評価した．結果，LRU よりも最大で 60.5%，平均で 9.16%と，従来手法よりも IPC が向上した．

Partitioning Cache by Instruction Groups

Kosuke Asami,[†] Naruki Kurata,[†] Ryota Shioya,[††]
Masahiro Goshima[†] and Shuichi Sakai [†]

The number of threads that work simultaneously on a shared cache has been increasing recently, and it becomes more important to manage a shared cache in some way. Many conventional caches use LRU as replacement policy, but LRU policy does not work well and can cause contention between threads on a shared cache. In this paper, we focus on a difference of the cache size that each instruction requires, and propose a cache partitioning method that fits the partition size to the working set size of an instruction group. The proposed technique can use a cache more efficiently and improves performance than conventional approaches. As the preliminary estimation, we evaluated Utility-based Cache Partitioning via-instruction group on full associative cache. Our evaluation shows that our proposal improves performance up to 60.5% and on average 9.16% over LRU, and this result is better than conventional approaches.

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

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University

ⓒ 2013 Information Processing Society of Japan

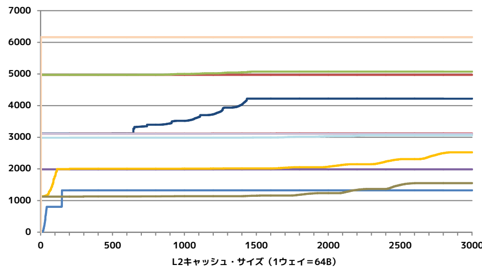

図 1　命令ごとに利用できる L2 キャッシュ・サイズを変化させた時のキャッシュ・ヒット数

グ[2),3)] 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．これにより，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，プロセッサ全体の性能を向上させることができる．

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[4)] の lbm において，命令ごとに利用できる 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，命令ごとの L2 キャッシュ・ヒット数を示したグラフである．この L2 キャッシュはフル・アソシアティブとし，命令ごとに利用できるウェイ数を変えることでキャッシュ・サイズを変えている．グラフの線は，それぞれ lbm 中の命令を表している．図上方の折れ線グラフは，利用できるウェイ数 1 でキャッシュ・ヒット数が 6000 近くで飽和している．この折れ線グラフで示された命令については，利用できるキャッシュ・サイズが 64B でもそれ以上でも，L2 キャッシュ・ヒット数は変わらない．つまり，この命令にとっては，L2 キャッシュ・サイズは 64B あれば十分であることがわかる．一方，図中程の折れ線グラフの中には，利用できるウェイ数 1500 近くで，キャッシュ・ヒット数が飽和している物もある．このような折れ線グラフで表された命令は，L2 キャッシュ・サイズが少なくとも 96KB 近く必要であることがわかる．このように，同一スレッド内の命令においても，命令ごとに必要なキャッシュ・サイズ は大きく異なっていることがわかる．そのため，従来のようにキャッシュをスレッドごとにパーティショニングするよりも，命令ごとにパーティショニングすることによって，より性能を向上することができる．また，命令ごとより，命令グループごとにパーティショニングすることで，キャッシュのより効率的な利用が可能となる．

2. 命令グループごとのキャッシュ・パーティショニング

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．

2.1 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 2** と**図 3** を用いて説明する．

図 2 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 3 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 3 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

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

図 2　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

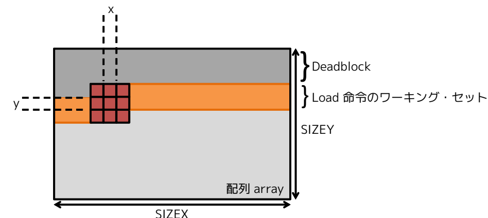

図 3　フィルタ処理アクセスの様子

ⓒ 2013 Information Processing Society of Japan

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．

2.2 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 2 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 4** のように，ループ内の処理は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．

```
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 4 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

2.3 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．

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．

2.4 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

(1) について，2.5 節で，(3) について，2.6 節で説明する．

2.5 命令グループの作成

2.2 節で述べたように，命令グループは同一のワーキング・セットにアクセスする命令群である．

提案手法では，以下のような方針で，実行時にハードウェアによって命令のグループ分けを行う．

- 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

具体的には，

(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
(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

の 2 つのテーブルを利用し，動的に同一ラインにアクセスする命令群を検出する．

2.6 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

ⓒ 2013 Information Processing Society of Japan

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[5)] 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．

2.7 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．

3. 予備評価

提案手法の有効性を示すため，従来手法と提案手法のモデルをプロセッサ・シミュレータ鬼斬弐[6)] 上に実装し，予備評価を行った．

2.6 節で述べたように，提案手法ではセット単位でのキャッシュ・パーティショニングをするために V-way cache 等が用いられるが，本稿の予備評価では，キャッシュにフル・アソシアティブ・キャッシュを利用し，フル・アソシアティブ・キャッシュ上に従来手法と提案手法を適用して，比較を行った．

フル・アソシアティブ・キャッシュを利用することによって，従来手法と提案手法の評価から競合性ミスの影響を無くすことができる．これによって，キャッシュの汚染によって引き起こされる，容量性ミスによる性能低下をどれだけ軽減できるかを測ることを目的としている．

3.1 評価対象モデル

提案手法の予備評価対象として，割り当てキャッシュ・サイズを Utility-based Cache Partitioning(UCP)[2)] で用いられる lookahead アルゴリズムを利用して求め，フル・アソシアティブ・キャッシュに対してパーティショニングを行うモデルを考える．このモデルを **Group** モデルと呼ぶ．UCP とは，従来手法であるスレッドごとのキャッシュ・パーティショニングの一手法である．

比較対象とするモデルは，キャッシュ・パーティショニングを行わず，LRU による制御を行う **Base** モデルと，スレッドごとに UCP を行う従来手法の **Thread** モデル，またデッド･ブロック予測の一手法である Sampling Dead Block Prediction[7)] を適用する **DBP** モデルである．デッド・ブロック予測とは，キャッシュ・マネジメントの既存手法であり，LRU なブロックよりも優先的に，deadblock であると予測したブロックをリプレースメント する手法である．

3.2 プロセッサのパラメータ

予備評価に用いるプロセッサは，1 コア・2 スレッドの SMT プロセッサ で，L1 データ・キャッシュと L2 キャッシュに対してマネジメントを行う．プロセッサのパラメータは**表 1** の通りである．

測定の際は，SPEC CPU2006[4)] の 29 本のベンチマークと 470.lbm の組み合わせ，計 29 通りの 2 スレッドの組み合わせで実行を行っている．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 命令実行して評価している．

Thread モデルと Group モデルにおいて，パーティション・サイズの変更は L1 データ・キャッシュでは 50k サイクルごと，L2 キャッシュでは 500k サイクルごとに行った．

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
L1D	FullAssoc, 64B-line 16KB, 3 サイクル
L1I	4-way 64B-line 16KB, 3 サイクル
L2	FullAssoc, 64B-line 1MB, 15 サイクル
主記憶	400 サイクル

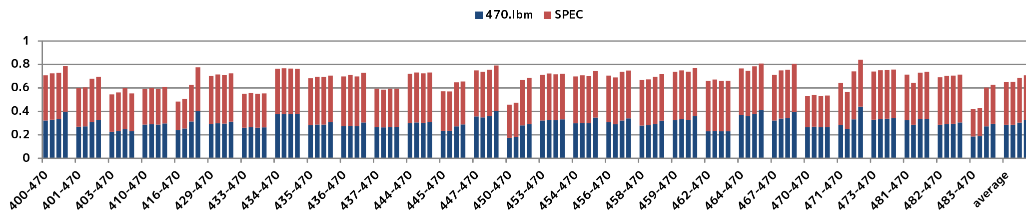

図 5 IPC の評価結果 左から順に Base,DBP,Thread,Group

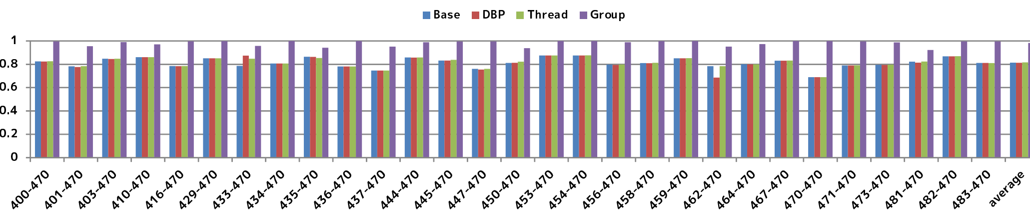

図 6 L2 キャッシュに対する write アクセスのヒット率の評価結果

3.3 予備評価結果

3.1 項で説明した各モデルにおける IPC の評価結果を**図 5** に示す．図 5 は，lbm ともう一方のベンチマークの，各スレッドの 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，DBP，Thread，Group モデルの IPC である．

Group モデルは，ほとんどの組み合わせにおいて，最も Base に対して IPC が向上したモデルであった．最大では，416.gamess と lbm の組み合わせの時，Base よりも 60.5%IPC が向上した．平均では 9.16%と，他モデルよりも IPC を向上することができた．

図 6 に，各モデルの L2 キャッシュに対する write アクセスのヒット率の評価結果を示す．Group モデルは平均で 98.2%と，他モデルよりも write アクセスのヒット率が大きく向上していた．

4. おわりに

本稿では，命令ごとに必要とされるキャッシュ・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，1 コア・2 スレッドの SMT プロセッサ上で，L1 データ・キャッシュと L2 キャッシュに対して命令グループごとに UCP を行うモデルの実装と評価を行い，既存手法と比較した．その結果，提案手法の予備評価モデルでは，最大で 60.5%，平均で 9.16%と，従来手法よりも IPC が向上した．

本稿の予備評価では，パーティション・サイズの決定アルゴリズムとして UCP で用いられているものを採用したが，追加ハードウェア量の問題も含め，どのようなアルゴリズムを用いるか，という検討が今後必要であると思われる．また，V-way Cache 等を用いたセットごとのキャッシュ・パーティショニングの具体的な方法の検討も，今後の課題である．

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) Qureshi, M. K. and Patt, Y. N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, MICRO '39 (2006).
3) 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).
4) The Standard Performance Evaluation Corporation: *SPEC CPU2006 suite* *http://www.spec.org/cpu2006/*.
5) Qureshi, M.K., Thompson, D. and Patt, Y.N.: The V-Way Cache: Demand Based Associativity via Global Replacement, ISCA '05 (2005).
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) Khan, S. M., Tian, Y. and Jimenez, D. A.: Sampling Dead Block Prediction for Last-Level Caches, MICRO '43 (2010).

ⓒ 2013 Information Processing Society of Japan