Vol.2012-ARC-200 No.14
Vol.2012-OS-121 No.14
2012/5/8

CMPにおけるキャッシュ・データを考慮したスレッド・スケジューリング手法の初期検討

三輪 忍[1,a)]　角崎 宏一[*,1]　佐々木 広[2]　中村 宏[1]

概要：多数のコアを有するCMPでは，チップ上に分散して配置されたRAMをラスト・レベル・キャッシュとして利用する．そのため，ラスト・レベル・キャッシュのアクセス・レイテンシは，コアとそれがアクセスするRAMとの物理的な位置関係によって異なる．これまで，データをキャッシュする位置を工夫することによって，キャッシュの容量効率を高めつつ平均アクセス・レイテンシの短縮を図る研究が行われてきた．しかし，そのようにデータの配置のみを考えていたのでは十分とは言えない．本稿では，データの配置だけでなくスレッドの配置も工夫することで，プロセッサの性能が向上する可能性があることを示す．また，データの配置を工夫する手法の1つであるCooperative Cacheを用いて予備評価を行った結果，ディレクトリ参照が性能上のボトルネックとなることがわかった．

キーワード：CMP，スレッド・スケジューリング，Cooperative Cache

1. はじめに

LSIの微細化にともない，1つのチップに搭載されるコア数は年々増加している．近年の商用プロセッサでは，2 ～ 4程度のコアからなるCMPが主流となっている．中には十数コアを1つのチップ上に搭載したものもあり，CMPのコア数は，今後，微細化が進むにつれてさらに増加すると期待されている．そのため，現在，数十コア，あるいは，百個を超えるコアを搭載したCMPに関する研究が盛んに行われている[6], [8], [16]．

多数のコアを有するCMPでは，ラスト・レベル・キャッシュの構成が通常とは異なる．キャッシュは，通常のCMPのように，チップ上のどこかに集中的に配置されるのではない．多コアのCMPでは，コアと少量のRAMを1つのノード（タイル）とし，それらを敷きつめることによってプロセッサ全体を構成する場合が多い（**図 1**）[6], [8], [16]．それぞれのノード間はNoCによって接続される．このような構成をとるのは，製造プロセスを簡略化し，コア数をスケールしやすくするためである．このようにして分散配置されたRAMをキャッシュとして利用する．

各RAMの利用方法は，大きく分けて以下の2つの選択肢がある．

shared 方式　各RAMを全てのコアで共有し，1つの巨大なラスト・レベル・キャッシュとして利用する方式．各RAMは共有キャッシュのバンクに相当する．バンクはアドレスによってインタリーブされる．各コアは全てのRAMを利用できるため，RAMの大きさを超えるデータを参照するスレッドが実行された場合でもデータをすべてキャッシュすることができる．したがって，後述するprivate方式と比べ，高いヒット率を実現できる．しかし，近傍のバンクに比べて遠方のバンクへのアクセスは時間がかかる（NUCAアーキテクチャ[9]）ため，キャッシュの平均アクセス・レイテンシは大きくなってしまう．

private 方式　各RAMをそれぞれのコアが専用のキャッシュとして利用する方式．各コアは1つのRAMしか利用できないため，RAMの大きさを超えるデータを参照するスレッドは，すべてのデータをキャッシュすることはできない．したがって，private方式のヒット率はshared方式のそれと比べて低くなる．しかし，ラスト・レベル・キャッシュへのアクセスは常にノード内のRAMに対するアクセスとなるため，レイテンシはshared方式と比べて短い．

1.1 CMPにおけるデータ配置手法

一般に，データはそれを参照するスレッドを実行しているノードの近傍にキャッシュするのがよい．各ノードは，

[1] 東京大学
The University of Tokyo
[2] 九州大学
Kyushu University
[a)] miwa@hal.ipc.i.u-tokyo.ac.jp
[*] 現在，九州電力株式会社所属

© 2012 Information Processing Society of Japan

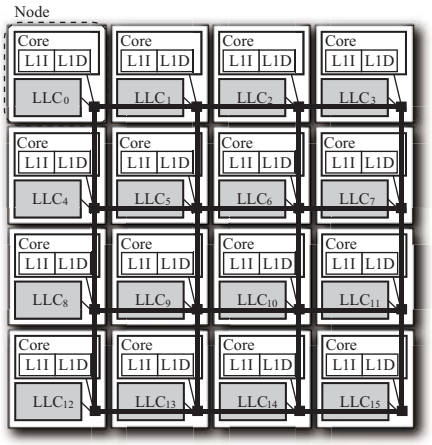

図 1 タイル型 CMP

そこで実行するスレッドが参照したデータをローカルな RAM にキャッシュする．ローカルな RAM の容量を超えた場合は，容量に余裕のある近傍の RAM にキャッシュする．そうすれば，shared 方式と同じく RAM の総容量を有効に利用しつつ，private 方式に匹敵するレイテンシを実現できる．

このような理想的なデータ配置を目指した研究がいくつか行われている [3], [4], [7], [9]．例えば，**Cooperative Cache**[2], [5], [13] は，他ノードの RAM を自身のノードの victim キャッシュとして利用する．各ノードが参照したデータは，基本的には，private 方式と同様，ローカルな RAM へとキャッシュする．そして，リプレースの際，一定確率で，ローカル RAM から追い出されたデータを隣接するノードへと転送し，そのノードの RAM にキャッシュする．再びそのデータが参照された場合は，メイン・メモリにアクセスしなくても，隣接ノードの RAM を参照すればデータを取得できる．

タイル型 CMP においては，オフチップに位置するメイン・メモリへのアクセスが性能上のボトルネックとなる．また，オンチップ・メモリのアクセス・レイテンシは，コアとそれがアクセスするメモリとの物理的な位置関係によって異なる．そのため，キャッシュのヒット率を高めつつ，平均アクセス・レイテンシを短縮することが重要である．

1.2 スレッドの配置問題

上述のように，従来の CMP のキャッシュに関する研究は，データをキャッシュする位置を工夫することで，キャッシュの容量効率を高めつつ平均アクセス・レイテンシの短縮を図っていた．しかし，そのようにデータの配置だけを考えていたのでは十分とは言えない．3 章で詳しく述べるように，どのノードでどのスレッドを実行するかによって，これらの手法の効果は変化し得るからである．スレッドの配置も含めて考えることでキャッシュをより有効に利用でき，プロセッサ全体の性能をさらに改善できると考えられる．

我々は，OS のスレッド・スケジューラを改良し，スレッドの配置を最適化することによって，CMP におけるキャッシュの利用効率を最大限高めることを考えている．そのための予備評価として，今回，スレッドの配置によって Cooperative Cache の効果がどの程度変化するかを測定した．その結果，CMP のキャッシュの利用効率を向上するには，ディレクトリ参照による性能オーバヘッドを削減しなければならないことがわかったので報告する．

以下まず次章では，Cooperative Cache を中心に，データ配置の最適化を目指した既存研究について詳しく述べる．続く 3 章では，我々が着目している，タイル型 CMP におけるスレッドの配置問題について説明する．4 章で評価を行い，5 章で関連研究について述べ，6 章でまとめと今後の課題について述べる．

2. キャッシュの利用効率向上のためのデータ配置手法

本章では，本稿で評価の対象とした Cooperative Cache についてまず説明する．次いで，その他のデータ配置手法についてまとめる．

2.1 Cooperatve Cache

Cooperative Cache は，private 方式において容量に余裕のある他ノードの RAM を victim キャッシュとして使用することにより，チップ全体のキャッシュの利用効率を高める手法である．詳しくは次章で述べるが，スレッドにはそれが使用するキャッシュ容量を増やすほどヒット率が向上し性能が向上するものとそうでないものとがある．そのため，後者のスレッドを実行するノードの RAM を前者のスレッドの victim キャッシュとして使用すれば，全体のヒット率は向上する．

Cooperative Cache の基本的な動作は以下の通りである．各コアは，private 方式と同様，それがアクセスしたラインをローカルな RAM にキャッシュする．ローカル・キャッシュからラインが追い出される際，必要に応じてラインを他のノードへと転送し，そのノードのキャッシュに格納する．再びそのラインが参照された際は，ラインを格納したリモート・キャッシュにアクセスすることでラインを取得する．このように，他ノードのキャッシュを victim キャッシュとして使用することにより，オフチップ・メモリへのアクセスを削減する．

図 2 に Cooperative Cache の構成を示す．Cooperative Cache の構成はいくつか存在する [2], [5], [13] が，本稿では分散ディレクトリ方式を採用した Distributed Cooperative Cache [5] を対象とする．これは，多コアの CMP では集中ディレクトリの参照は性能上のボトルネックとなりやすいためである．

© 2012 Information Processing Society of Japan

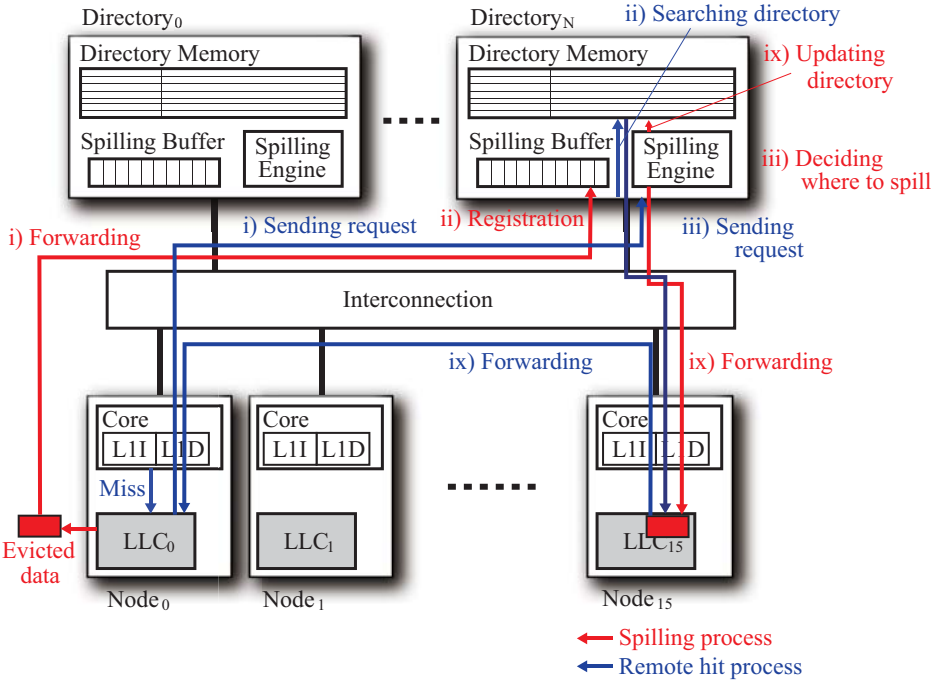

図 2 Distributed Cooperative Cache の構成と動作

Distributed Cooperative Cache では，各々のディレクトリを拡張し，Spilling Buffer と Spilling Engine を追加する．Spilling Buffer は，他ノードのキャッシュへ転送するラインを一時的に保持するバッファである．各ラインの転送先は Spilling Engine が決定する．ディレクトリはアドレスによってインタリーブされる．なお，ラインを他ノードのキャッシュへ格納する一連の処理を **spill** と呼ぶ．

spill は以下のようにして行われる．ローカル・キャッシュでリプレースメントが発生すると，まず，そのラインが spill 対象のラインか否か判断される．複製が他に存在しなければ，基本的には，そのラインは spill の対象となる．対象だった場合は以下のようにして spill を行う．

(1) まず，キャッシュから追い出されたラインを該当するディレクトリへ転送する．
(2) 当該ディレクトリ（図では N）では，受け取ったデータを Spilling Buffer に格納する．
(3) 各ディレクトリの Spilling Engine は，Spilling Buffer の先頭からラインを取り出し，spill 先を決定する．spill 先を決めるポリシーはいくつか存在し，例えば，ある一定の確率で隣接ノードのキャッシュに spill する [2]，spill 先のキャッシュのヒット率が向上する場合に spill する [13] などがある．
(4) spill 先が決定したら，該当するノードへラインを転送し，キャッシュに書き込む．また，そのラインのアドレスと転送先のノード ID を用いて Directory Memory を更新する．

spill したデータへのアクセスは，通常のコヒーレンス制御のプロセスと何ら変わりはない．すなわち，あるノード（図ではノード 0）でローカルのキャッシュにミスすると，

(1) 該当するディレクトリに問い合わせを行う．
(2) 問い合わせのあったディレクトリでは，Directory Memory を参照し，当該ラインが他ノードにキャッシュされているか調べる．図では，ノード 15 にキャッシュされていることがわかる．
(3) ラインをキャッシュしているノード 15 に対し，ラインを要求元のノード 0 へ転送するように要求する．
(4) リクエストを受け取ったノード 15 は，ラインを要求元のノード 0 へと転送する．

Cooperative Cache では，一度 spill されたラインが再び spill されることのないよう，spill される回数に制限を設けている（1-Fwd ポリシー）．これは，再利用されないラインが spill され続けることによって，キャッシュの容量を圧迫することを防ぐためである．このために，各キャッシュ・ラインに対し，そのラインが spill によってキャッシュされたラインか否かを判別するためのビットを追加する．このビットがセットされている場合は，ラインがキャッシュから追い出された場合でも spill しない．

各キャッシュ上では，そのノードで実行中のスレッドが割り当てたラインと，他ノードから spill されたラインとの間で競合が発生する．spill の発生頻度はスレッド毎のキャッシュ・ミス頻度に依存する．そのため，あるキャッシュでは，頻繁にミスするスレッドのデータを受け取ったことによって，そのノードで実行しているスレッドが参照する予定だったラインを追い出してしまうことがある．この競合の影響を緩和するため，キャッシュ・パーティショニングを利用することが提案されている [1]．

© 2012 Information Processing Society of Japan

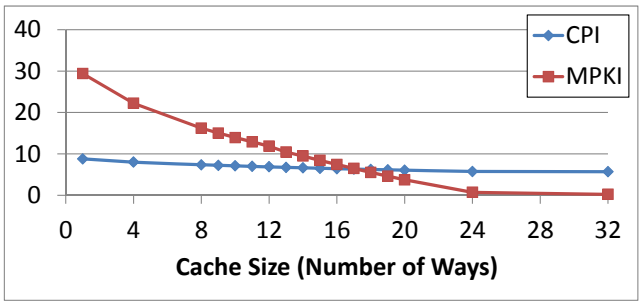

図 3 高 Utility スレッド（*twolf*）

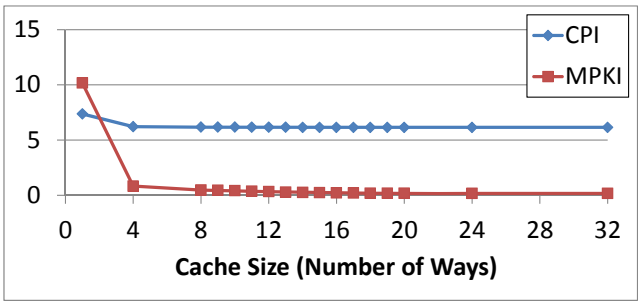

図 4 低 Utility スレッド（*crafty*）

2.2 その他の手法

CMP のオンチップ・メモリを有効利用する方法は，Cooperative Cache 以外にもいくつか提案されている．

D-NUCA[7], [9] は，shared 方式において，最近参照したラインに対するアクセスを高速化する．通常の shared 方式は，各 RAM がアドレスでインタリーブされているため，参照するデータのアドレスによっては，要求元のノードから遠く離れた RAM にアクセスしなければならない．それに対し，D-NUCA では，各 way を各バンクに対応させる．例えば，図 1 の例では，LLC_0 を way0，LLC_1 を way1，$\cdots$ のように対応づける．このようにすれば，最近参照したラインを参照元のノードのバンクに移動させることができる．

ただし，この方法はヒット/ミスの判定のために複数のバンクを検索する必要がある．そのため，検索に通常よりも多くのエネルギを消費するという問題を抱えている．また，ラインを格納する可能性のあるすべてのバンクにミスしなければキャッシュ・ミスか否かは判明しないため，ミスが判明するまでの時間が遅くなるという問題もある．

R-NUCA[4] は，各 RAM のインデクスを工夫することで，ラインが格納される RAM を，それを参照したコアから数ホップ以内の RAM に限定する．この方法を rotational interleaving と呼ぶ．ラインを配する領域の大きさは，ノードごとだけでなく，アドレス空間ごとにも変更する．OS のページ割り当てを利用して，private なデータが多いアドレス空間を 0 ホップの領域，すなわち，ローカルな RAM にマッピングし，shared データが多いアドレス空間はリモートの RAM にマッピングする．

3. CMP におけるスレッドの配置問題

前章で述べた手法はいずれも，キャッシュするデータとその位置を工夫することによって，キャッシュの容量効率を高めつつ平均アクセス・レイテンシの短縮を図っていた．しかし，さらなる性能向上を考えた場合，データの配置だけを考えていたのでは十分とは言いがたい．

例として，図 1 に示した 16 ノードからなるタイル型 CMP の各コアにおいて，異なる 16 個のスレッドを実行する場合を考える．各ノードは，図に示すように，メッシュ状のネットワークによって接続されている．以下，Cooperative Cache が適用されたとする．

3.1 Cache Utility と Cooperative Cache

一般に，キャッシュ容量を増減した場合の性能向上率はスレッドによって異なる．利用可能な容量が増えるにつれて性能が漸次向上するスレッドもあれば，そうではないスレッドもある．前者を Cache Utility[14] が高いスレッド（以下，**高 Utility スレッド**），後者を Utility が低いスレッド（以下，**低 Utility スレッド**）と呼ぶことにする．

それぞれの例を**図 3**，および，**図 4** に示す．グラフは，SPEC CPU 2000 ベンチマークの 2 つのプログラム（*twolf*, *crafty*）について，L2 キャッシュのセット数を固定した状態で way 数を増加させた時の，CPI，および，MPKI の変化を表している．横軸は way 数，縦軸は CPI，または，MPKI である．グラフより，*twolf* はキャッシュ・サイズの増加にともなって漸次 CPI が減少するのに対し，*crafty* は 4way 以上に増やしても CPI がほとんど変わらない．

このようにスレッドによって Utility は異なる．したがって，RAM 全体を効率良く利用するためには，高 Utility スレッドを実行中のノードから低 Utility スレッドを実行中のノードへデータを spill した方がよい．

3.2 スレッドの配置と Cooperative Cache の効果

図 1 の CMP において，ノード 0 で高 Utility スレッドが，ノード 1, 4, 5 で低 Utility スレッドが実行されたする．その様子を表したのが**図 5**(a) である．図において，各キャッシュは 4 つの領域に分割されており，分割された各領域は保持中のラインを表している．各ライン中の文字列は，そのラインがどのスレッドによって参照されたかを表す．各ラインは，それを参照したスレッドの Utility が高いほど濃色で，Utility が低いほど淡色で表現してある．ノード 0, 1, 4, 5 で，それぞれ，スレッド 0, 1, 4, 5 が実行されているものとする．各スレッドが**表 1** に示す way 数を利用できた場合に，全体のヒット率は最も高くなるものとする．

前述のように，データの spill は，高 Utility スレッドを実行中のノードから低 Utility スレッドを実行中のノードへ行った方がよい．そのため，図 5(a) のような配置でス

© 2012 Information Processing Society of Japan

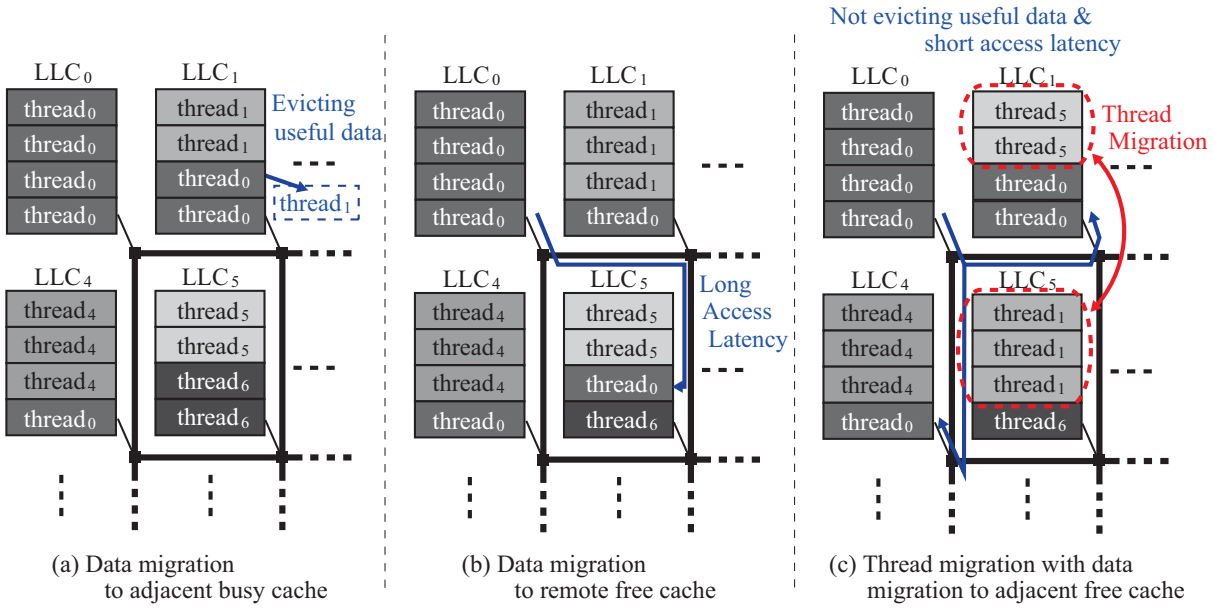

図 5 CMP におけるスレッドの配置問題

レッドが実行された場合，高 Utility スレッドを実行するノード 0 は，ラインを低 Utility スレッドを実行するノードに spill しようとする．その時の選択肢としてまず考えられるのは，低 Utility スレッドを実行中の隣接ノードへ spill することである．図では，スレッド 4 を実行しているノード 4 は 3 way あれば十分（表 1）なことから，ノード 0 からノード 4 に 1 ライン spill している．

このように spill 先を低 Utility スレッドを実行中の隣接ノードに限定してしまうと，十分なヒット率を達成できないことがある．例えば，図 5(a) では，スレッド 0 は 7 way を使用したい（表 1）がために，ノード 4 に 1 つのライン，ノード 1 に 2 つのラインを spill している．その結果，スレッド 1 は 3 way 分の容量を利用できた方がよいにも関わらず，ローカル・キャッシュを 2 way 分しか利用できない，という状況が生まれてしまう．

このような容量不足を避けるために，図 5(b) のように，離れた場所に位置する容量に余裕のあるノード（図ではノード 5）に spill するという選択肢も考えられる．しかし，離れたノードに spill した場合は，近くのノードに spill した場合と比べて，それにヒットした時のレイテンシが増加してしまう．また，パケットがより長い距離を伝搬することで，ネットワークが輻輳する可能性が増加する．

このような状況は，図 5(c) のようにスレッドを配置すれば回避できる．すなわち，スレッド 1 とスレッド 5 をスワップし，ノード 1 でスレッド 5 を，ノード 5 でスレッド 1 を実行する．そうすれば，ノード 0 がノード 1 にラインを 2 つ spill したとしても，スレッド 5 は 2 way 分の容量を利用できれば十分なため，容量不足に陥らない．

このように，どのような Utility のスレッドをどこで実行するかによって Cooperative Cache の効果は異なる．Cooperative Cache の効果を十分に引き出すためには，Utility を考慮してスレッドを実行するノードを変更した方がよいことがわかる．

表 1 各スレッドに割り当てるキャッシュ容量

スレッド	割り当てる容量（way）
0	7
1	3
4	3
5	2

4. 評価

我々は，Utility に応じてスレッドを実行するノードを変更するスレッド・スケジューラの開発を考えている．そのための予備評価として，今回，スレッドの配置によって Cooperative Cache の効果がどの程度変化するかを測定した．本章ではその結果を述べる．

4.1 評価環境および評価条件

プロセッサ・シミュレータ gem5[11] に Distributed Cooperative Cache を実装し，評価を行った．評価した CMP は 16 個のコアを持ち，それらが 4x4 のメッシュ状に並んでいる．各コアはインオーダで，ディレクトリは各ノードに分散されている．**表 2** にその他のパラメタをまとめる．

ベンチマーク・プログラムとして，SPEC CPU 2000 の中から**表 3** に示す 10 本のプログラムを選び，それらを組み合わせて使用する．10 本のプログラムを表 3 に示すように高 Utility スレッド，低 Utility スレッドに分類する．実行の際は，高 Utility スレッド/低 Utility スレッドそれぞれからプログラムを選択する比率（例えば 1 : 3, 1 : 1, 3 : 1）をまず決定し，次いで，その比率にもとづいてそれぞれの

© 2012 Information Processing Society of Japan

表 4 評価に用いたプログラム群

ID	Contents	#H : #L
com1	bzip2 twolf ammp twolf mesa crafty apsi mesa crafty crafty wupwise apsi wupwise applu applu mesa	1:3
com2	art110 art110 galgel ammp mesa mesa wupwise crafty applu mesa crafty crafty crafty apsi apsi mesa	
com3	galgel ammp art110 bzip2 wupwise wupwise mesa wupwise crafty mesa crafty mesa mesa crafty apsi applu	
com4	twolf ammp galgel art110 ammp ammp art110 art110 crafty crafty crafty apsi apsi applu crafty apsi	1:1
com5	art110 art110 twolf galgel twolf ammp ammp art110 wupwise applu crafty applu mesa crafty applu applu	
com6	bzip2 ammp twolf twolf bzip2 galgel galgel twolf mesa applu wupwise apsi mesa applu mesa apsi	
com7	ammp twolf galgel art110 art110 twolf bzip2 bzip2 ammp art110 bzip2 galgel mesa apsi wupwise applu	3:1
com8	galgel ammp art110 ammp twolf galgel twolf ammp art110 galgel bzip2 art110 mesa crafty wupwise mesa	
com9	bzip2 ammp ammp galgel bzip2 ammp ammp ammp twolf galgel galgel galgel apsi apsi apsi mesa	

グループから実行するプログラムをランダムに選択する．このようにして得られた，**表 4** に示す 9 種類のプログラム群を用いて評価した．

評価は以下の 4 つのモデルについて行う．

SHARED shared 方式

PRIVATE private 方式

CC Cooperative Cache

CCwCP Cooperative Cache とキャッシュ・パーティショニングを組み合わせた方式

Cooperative Cache を用いるモデルでは，高 Utility スレッドを実行するノードから低 Utility スレッドを実行するノードへとキャッシュ・データを spill する．spill の際，spill 先のノードの選び方，および，spill する量（リプレースメント何回に対し 1 回 spill するか）にはいくつかの選択肢がある．我々は，この問題を最小費用流問題に見立て，Kleinberg のアルゴリズム [10] にもとづき，総メモリ・アクセス時間が最小となる spill 先と spill 量を静的に決定した．この方法の詳細は紙面の都合により省略する．

スレッドの配置が Cooperative Cache に与える影響を調べるため，スレッドの配置パターンをランダムに決定して実行時間を測定する，という実験を複数回繰り返す．次節で示す結果は，24 回の試行を繰り返した結果である．なお，SHARED，および，PRIVATE については，上述の試行を行わず，あるスレッドの配置パターン 1 回についてのみ評価する．これは，これらのモデルではスレッドの配置による実行時間の変化がほとんど見られなかったためである[*1]．いずれの試行も，最初の 1G サイクルをスキップし，続く 200M サイクルを実行した．

表 2 評価した CMP の構成

Parameters	Remarks
# Core	16
Line Size	64B
L1 I/D Cache	16kB, 4-way, 1 cycle (per core)
L2 Cache	1MB, 16-way, 10 cycles (per core)
Directory Latency	12 cycles
Link Latency	4 cycle
Link Bit Width	128b
Router Pipeline	4 stages
Main Memory	3GB, 50ns
Coherence Protocol	MOESI

表 3 評価に用いたベンチマーク

Cache utility	Programs
High	art ammp galgel bzip2 twolf
Low	applu apsi crafty mesa wupwise

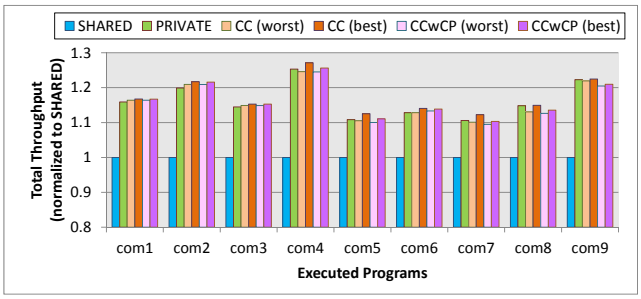

図 6 トータル・スループット

4.2 評価結果

図 6 に各モデルの性能を示す．グラフの横軸はプログラム群，縦軸はトータル・スループットである．プログラム群ごとの 6 つの棒グラフは，左から順に，SHARED，PRIVATE，CC（全 24 回の試行におけるワースト・ケース），CC（同ベスト・ケース），CCwCP（ワースト・ケース），CCwCP（ベスト・ケース）である．なお，各モデルの結果は SHARED のそれで正規化してある．

グラフより，まず，タイル型 CMP においてはラスト・レベル・キャッシュのアクセス・レイテンシが性能を左右する非常に重要な要素であることがわかる．各モデルの平均メモリ・アクセス・レイテンシを**図 7** に示す．ただし，グラフの結果は L1 キャッシュにヒットするアクセスを除いたものとなっている．グラフより，SHARED の平均アクセス・レイテンシは他の 5 つのそれと比べて極端に大き

[*1] 簡単のため，本実験では各ルータにメモリ・コントローラが付随していると仮定した．実際には，メモリ・コントローラの数は限られているため，メモリ・アクセスを行うスレッドが実行されているノードの位置とメモリ・コントローラの位置とよって，private 方式や shared 方式でも性能が変化すると予想される．この影響については今後評価する．

© 2012 Information Processing Society of Japan

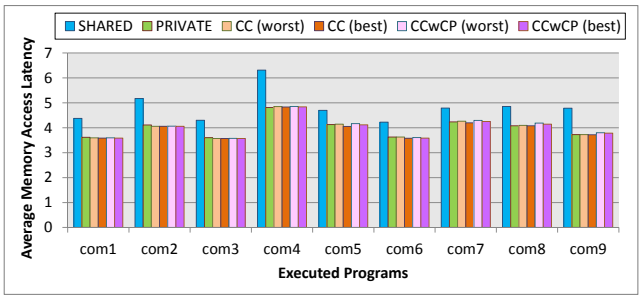

図 7 平均メモリ・アクセス・レイテンシ

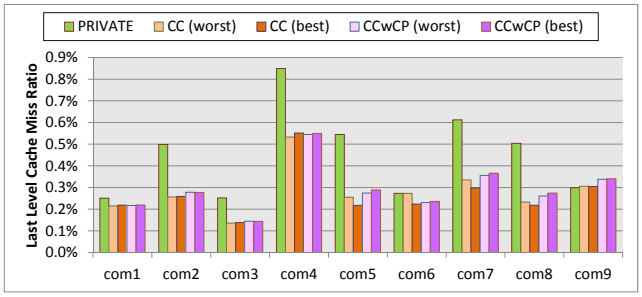

図 8 ラスト・レベル・キャッシュ全体のミス率

い．その結果，トータル・スループットでは，最も小さい場合（CC(worst) の com5）でも 10.1%，最も大きい場合（CC(best) の com4）では 27.1% の性能差を示す．

一方，残念ながら今回の実験では，Cooperative Cache を用いたモデルと PRIVATE との間に大きな性能差は見られなかった．この原因はいくつか考えられるが，1 つには後述するディレクトリ参照のオーバヘッドが挙げられる．このオーバヘッドが大きいために，リモート・キャッシュにヒットした場合でもあまりレイテンシを短縮できず，先行研究のような性能改善が達成できなかったと考えられる．また，今回の試行は実行サイクル数が 200M サイクルと短かったため，1MB のプライベート・キャッシュで容量は十分足りてしまい，ラスト・レベル・キャッシュ・ミスがあまり発生しなかったことも原因の 1 つである．

CC，および，CCwCP においてスレッドの配置を変更した場合も，トータル・スループットではほとんど差が見られなかった．ただし，後述するように，データをリモート・キャッシュからそれを要求したノードへ転送する時間は改善されている．リモート・キャッシュ・ヒット時に必要とされるその他の処理に要する時間が大きいため，この改善がトータル・スループットの改善に繋がっていない．

図 8 は L1 キャッシュにミスしたアクセスがメイン・メモリ・アクセスを行った割合，すなわち，ラスト・レベル・キャッシュ全体のミス率を表したグラフである．グラフの横軸はプログラム群，縦軸はメイン・メモリ・アクセスの割合を表す．gem5 の仕様の問題から SHARED のメイン・メモリ・アクセス数を取得できなかったため，グラフには SHARED 以外の 3 つのモデルの結果のみ示してある．

グラフより，Cooperative Cache を用いることによって

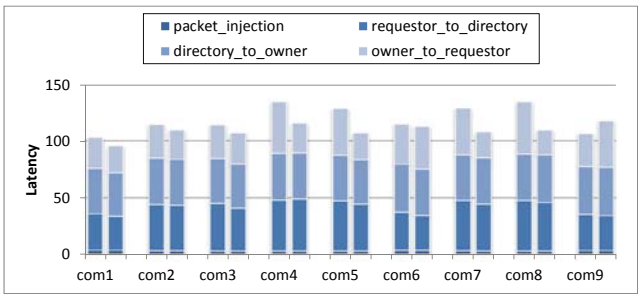

図 9 リモート・キャッシュの平均アクセス・レイテンシの内訳

RAM のトータルのミス率は改善する．改善率は実行するプログラムの組み合わせによって異なるが，最も効果が大きい com5 では，CC(best) の場合で 60.0%，CCwCP(worst) の場合で 49.7% の改善が見られた．ただし，ミス率の値そのものはあまり大きいとは言えず，その結果，ミス率の改善がトータル・スループットの改善に繋がっていない．

CCwCP におけるリモート・キャッシュの平均アクセス・レイテンシの内訳を**図 9** に示す．横軸はプログラム群，縦軸はレイテンシである．プログラム群ごとの 2 本の棒グラフは，左がワースト・ケース，右がベスト・ケースである．棒グラフは 4 色に塗り分けられており，それぞれの意味は以下の通りである．

packet_injection 命令が発行されてからパケットがネットワークに送出されるまでのレイテンシ

requestor_to_directory データを要求したノードからそのデータを管理するディレクトリにパケットが到達するまでのレイテンシ

directory_to_owner ディレクトリから要求データを保持するノードにパケットが到達するまでのレイテンシ

owner_to_requestor 要求データを保持するノードから要求元へデータを転送するレイテンシ

グラフより，スレッドの配置を変えることによってリモート・キャッシュの平均アクセス・レイテンシは短縮されることがわかる．その差は，最も大きい com8 で 25.1 サイクルであった．このように，スレッドの配置を変えることにより，Cooperative Cache におけるリモート・キャッシュの平均アクセス・レイテンシは短縮できる．

短縮はされたものの，依然として平均アクセス・レイテンシは大きい．ローカル・キャッシュのヒット・レイテンシは 10 サイクル（表 2）であるのに対し，リモート・キャッシュの平均アクセス・レイテンシは 95.8 ～ 134.9 サイクルと非常に大きい．これはメイン・メモリ・アクセスの平均レイテンシの半分ほどの大きさである．このように，リモート・キャッシュの平均アクセス・レイテンシが大きいため，CC や CCwCP のトータル・スループットの改善幅が小さいものと思われる．

平均アクセス・レイテンシの内訳を見てみると，ディレクトリ参照のオーバヘッドが非常に大きいことがわかる．

© 2012 Information Processing Society of Japan

要求元のノードからディレクトリ，および，ディレクトリから owner のノードへと要求を転送するのに要する時間は，平均で 68.2 ～ 85.9 サイクルにも達する．率にして，リモート・キャッシュ・アクセス・レイテンシの 61.2 ～ 77.0% がディレクトリ参照のために費されている．Cooperative Cache そのものの性能，および，スケジューラによってスレッドの配置を変えた場合の性能を向上させるためには，このオーバヘッドを削減することが重要である．

5. 関連研究

Michaud は，private 方式のキャッシュの容量を効果的に利用するため，スレッドをマイグレーションする手法を提案している [12]．CMP 上でシングル・スレッドを実行しているノードが 1 つしかない状況では，他ノードのキャッシュは何ら機能していない．そこで，Michaud の方法では，プログラムのワーキング・セットを等分し，プログラムの実行状態があるワーキング・セットから別のワーキング・セットに移った時にスレッドをマイグレーションすることで，シングル・スレッド実行時においても他ノードのキャッシュを有効利用する．このように Michaud の方法はシングル・スレッドが 1 つだけ実行されている状況を想定しており，我々が想定している複数のシングル・スレッドが実行される場合とは状況が異なる．

shared 方式におけるスレッド・マイグレーション手法もある．Shim らの方法では，リモート・キャッシュ・アクセスを減らすため，リモート・キャッシュ・アクセスが発生した際に，そのデータを要求したスレッドとそのデータをキャッシュしているノードで実行中のスレッドとをスワップする [15]．ただし，スレッドのスワップはハードウェア・レベルで行うことを想定しており，我々が考える，OS レベルでスレッドとノードのマッピングを変更するアプローチとは異なる．

6. まとめと今後の課題

本稿では，Cooperative Cache を適用したタイル型 CMP を対象に，スレッドの配置がプロセッサ性能に与える影響について調査した．スレッドの配置によって利用可能な近傍のキャッシュの量，および，リモート・キャッシュのアクセス・レイテンシが変わるため，スレッド配置を変えることによって Cooperative Cache を用いるプロセッサの性能は向上すると考えられた．ところが実際には性能はほとんど変わらず，その原因はディレクトリ参照のオーバヘッドにあることがわかった．

ディレクトリ参照のオーバヘッドを削減する方法として，例えば，spill したデータ用のプライベートなディレクトリをノード内に設けることが考えられる．各ノードは，ローカル・キャッシュにミスした場合はまずこのディレクトリを参照し，要求したデータが spill されていないか，および，spill されていた場合はその spill 先をチェックする．spill されていれば，このディレクトリから spill 先のノードへ，データ転送要求を直接送るようにする．このようにすれば，リモートのディレクトリを参照しなくても，spill したデータを取得できる．spill されるデータはすべてプライベートであるため，ディレクトリもプライベートでよい．

ディレクトリ参照のオーバヘッドの問題を解決した上で，スレッドの配置が Cooperative Cache に与える影響を改めて評価したいと考えている．スレッドの配置を変えることの効果を確認した上で，スレッド・スケジューラの開発を行う予定である．

謝辞 本研究の一部は半導体理工学センター（STARC）共同研究経費（課題名「NoC 型メニーコア SoC のアーキテクチャレベル設計最適化」）による．

参考文献

[1] Chang, J. et al.: Cooperative cache partitioning for chip multiprocessors, *ICS*, pp. 242–252 (2007).
[2] Chang, J. et al.: Cooperative Caching for Chip Multiprocessors, *ISCA*, pp. 264–276 (2006).
[3] Chaudhuri, M.: PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches, *HPCA*, pp. 227–238 (2009).
[4] Hardavellas, N. et al.: Reactive NUCA: near-optimal block placement and replication in distributed caches, *ISCA*, pp. 184–195 (2009).
[5] Herrero, E. et al.: Distributed cooperative caching, *PACT*, pp. 134–143 (2008).
[6] Howard, J. et al.: A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS, *ISSCC*, pp. 108–109 (2010).
[7] Huh, J. et al.: A NUCA substrate for flexible CMP cache sharing, *ICS*, pp. 31–40 (2005).
[8] Intel Corp.: From a Few Cores to Many: A Tera-scale Computing Research Review.
[9] Kim, C. et al.: An Adaptive, Non-Uniform Cache Access Architecture for Wire-Delay Dominated On-Chip Caches, *ASPLOS*, pp. 211–222 (2002).
[10] Kleinberg, J.: *Algorithm design*, Pearson Education India (2006).
[11] Martin, M. M. K. et al.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, *SIGARCH Comp. Arch. News*, Vol. 33, No. 4, pp. 92–99 (2005).
[12] Michaud, P.: Exploiting the Cache Capacity of a Single-Chip Multi-Core Processor with Execution Migration, *HPCA*, pp. 186–195 (2012).
[13] Qureshi, M. K.: Adaptive Spill-Receive for Robust High-Performance Caching in CMPs, *HPCA*, pp. 45–54 (2009).
[14] Qureshi, M. K. et al.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *MICRO*, pp. 423–432 (2006).
[15] Shim, K. S. et al.: Judicious Thread Migration When Accessing Distributed Shared Caches, *CAOS* (2012).
[16] Tilera Corp.: http://www.tilera.com/products/processors/TILE-Gx_Family.

© 2012 Information Processing Society of Japan