情報処理学会研究報告
IPSJ SIG Technical Report

マルチコア・プロセッサ向けのヘルパースレッドによるキャッシュ制御支援手法の検討

橋本崇浩[†1]　井上功一[†1]　近藤正章[†1]
平澤将一[†1]　本多弘樹[†1]

近年１チップ上に複数のコアを搭載するマルチコア・プロセッサ構成を用いることが主流となっている．今後もコア数は増加すると予想されるが，現在では多くのコアを活用できるような並列プログラムは限られており，増加するコアを有効利用することは重要な課題である．また，それらマルチコア・プロセッサでは，キャッシュメモリの有効利用という観点から共有キャッシュメモリを実装することが多い．しかし，他のスレッドとのアクセスパターンやアクセス間隔などの違いから，再利用性の高いデータがキャッシュから追い出さてしまうキャッシュ競合が問題となることがある．

そこで本研究では，共有キャッシュの置換制御の補助を行う専用スレッドをヘルパースレッドとして遊休コア上で動作させ，キャッシュの競合を緩和させることで性能向上を図る手法を検討する．ヘルパースレッドは，他コアで動作するスレッドのキャッシュミスの情報を取得してデータの再利用性を予測しつつ，再利用性の低いデータを次に当該セットでキャッシュミスが生じた際にキャッシュから追い出され易くなるよう制御することで競合の緩和を狙う．本手法の評価を行った結果，共有キャッシュにおける競合頻度が高い場合，提案手法によって性能を向上させることが可能であることを確認した．一方で，現状ではソフトウェアによる処理がキャッシュミスイベントの発生頻度に追いつかず，性能向上率は高くないことがわかった．

1. はじめに

半導体プロセスの微細化に伴いプロセッサの消費電力，および発熱量は年々増大しており，従来のクロック周波数向上によるシングルコア・プロセッサの高性能化は困難となっている．そのため，近年では１つのチップ上に複数のコアを搭載するマルチコア・プロセッサが主流となっている．今後も，マルチコア・プロセッサのコア数は増加すると予想され，将来的には 100 コア以上ものコア数を持つ汎用プロセッサが登場すると考えられる．しかし，現在ではまだ多くのコアを活用できるような並列プログラムは限られており，複数のアプリケーションを同時並行的に実行する場合でも，十個以上ものコアを用いる状況は多くない．そのため，増加するマルチコア・プロセッサのコアを有効利用することが重要な課題である．

マルチコア・プロセッサでは，キャッシュメモリの有効利用の観点から，複数のコアでキャッシュ(一般的にはラストレベルキャッシュ) を共有する場合が多い．しかし，この共有キャッシュにおいて複数のコアでキャッシュ領域の奪い合いが生じ，キャッシュ競合が発生すると，性能低下を引き起こすという問題がある．特に，スレッド間のデータのアクセスパターンやアクセス間隔などの違いから，本来では再利用性の高いデータがキャッシュから追い出されてしまうと，シングルスレッドで動作した場合と比較して性能低下が深刻となる．

複数あるコアを有効活用し，キャッシュヒット率を向上させることでメモリウォールの緩和に取り組む手法として，ヘルパースレッド[2),5),7)] による手法が知られている．それらの手法の多くは，まず対象となるメインスレッドからメモリアクセスに関連する命令を抜き出してヘルパースレッドを生成する．そして，そのヘルパースレッドにより，主記憶アクセスのために時間のかかるロード命令を，メインスレッドの代わりに前もって共有キャッシュへプリフェッチする．これにより，本スレッドが要するはずだったロード命令による主記憶アクセスレイテンシを隠蔽することができ，大幅な性能向上が期待できる．しかしマルチコア・プロセッサ環境において，複数のスレッドを同時に実行する場合などでは，プリフェッチによるキャッシュヒット率が向上するスレッドがある一方で，プリフェッチによってデータが追い出されてキャッシュヒット率が低下してしまうこともある．

本稿では，今後増えると予想される遊休状態のコアの有効活用と，共有キャッシュの競合による性能低下の防止を狙い，共有キャッシュの置換対象制御を行う専用スレッドをヘルパースレッドとして動作させ，共有キャッシュの競合を緩和させることでメモリ性能の向上を図る手法を検討する．ヘルパースレッドは，他のコアで動作するスレッドのキャッシュミスの情報を取得しつつ，そのミスデータの再利用性を判定する．そして，再利用性の低いデータに対しては，ハードウェアの置換制御アルゴリズムに関わらず，次に当該セットでキャッシュミスが生じた際にキャッシュから追い出され易くするよう制御する．これにより，再利用性の高いデータがキャッシュに残る確率が高くなり，共有キャッシュ上での競合が緩和されると期待できる．

これまでにも，キャッシュ競合への対処を目的として，キャッシュ分割などハードウェアによる対処手法が多く研究されてきた[6)]．しかし，キャッシュ競合の状況は，プログラムのメモリアクセスパターンや，同時に実行するスレッドの組み合わせに強く依存する．よって，

†1 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

ある特定の手法があらゆるキャッシュ競合時において効果的に動作するとは限らない．ヘルパースレッドを用い，ソフトウェアによりデータの置換制御を支援することで，実行しているプログラムに合わせて柔軟に制御が行えるため，従来手法のハードウェアベースの手法に比べて有用な手法になり得ると考えられる．

本稿の構成は以下の通りである．2 章では関連研究として，従来のヘルパースレッドによるキャッシュヒット率向上手法について述べる．次に 3 章で提案するキャッシュ置換制御手法について述べ，4 章で性能評価を行う．最後に 5 章まとめと今後の課題について述べる．

2. 関 連 研 究

マルチコア・プロセッサ環境におけるヘルパースレッドを用いたキャッシュヒット率向上に関する手法はこれまでにも多く研究されている[1),4),5),7)]．

Kamruzzaman らの手法[5)] では，コンパイラによってループイタレーションを chunk と呼ばれるサイズに分割し，ロード命令とそれに関係する命令のみで構成された chunk をメインスレッドとは別のコアでヘルパースレッドとして実行させる．そして，ヘルパースレッドによりプリフェッチされたデータをメインスレッドからアクセスしようとした際に，メインスレッドをヘルパースレッドが実行されたコアへとマイグレーションし，すでにキャッシュにプリフェッチされたデータを利用することで，メモリレイテンシの隠蔽を狙う．

Ganusov ら[4)] や今里ら[1)] の手法では，ヘルパースレッドによるソフトウェアベースのプリフェッチ手法を提案している．メインスレッドが動作するコアは共有ラストレベルキャッシュでキャッシュミスが発生した時，そのミス情報を提案手法用に拡張した専用バッファへ格納する．ヘルパースレッドは専用バッファにミス情報があることを検知したら，そこからミス情報を取り出し，取得した情報を元にアドレスを予測しつつ，プリフェッチ命令を発行する．

これらの手法では，共有キャッシュで問題となるキャッシュ競合に対処したものではなく，ヘルパースレッドによるプリフェッチ量やタイミングが適切でない場合，プリフェッチによってキャッシュにある有用なデータが追いだされ，競合の問題がより深刻になる可能性もある．これに対し，本稿で提案する手法はキャッシュ競合が頻発するような状況において，ヘルパースレッドにより再利用性を判断しつつ，キャッシュ置換制御を行うものであり，従来のヘルパースレッドの利用法とは異なるものである．

3. キャッシュ置換制御手法の検討

本章では，共有ラストレベルキャッシュを持つコア数 k のプロセッサ環境を例として，ヘルパースレッドによるキャッシュ置換制御に必要なプロセッサハードウェアの拡張と，ヘルパースレッドによるキャッシュ置換制御アルゴリズムについて説明する．

3.1 ハードウェアの拡張

ソフトウェアから共有ラストレベルキャッシュの置換制御の支援を行い，キャッシュ競合を緩和して性能を向上させるためには，ソフトウェアからのキャッシュミス情報の取得や，置換情報の通知が行えるようにハードウェアを拡張する必要がある．本稿で提案するキャッシュ置換制御手法を実装したプロセッサの構成の全体を**図 1** に示す．通常のプロセッサ構成に，ミス情報を格納する *MSHR: Miss State Holding Register* をソフトウェアから読み込むための専用経路，キャッシュ置換制御を行うキャッシュコントローラ (Cache Controller) へ情報を設定する専用経路を追加し，また通常の LRU 以外の置換制御を行えるようにするためのキャッシュコントローラ自体にも拡張が必要となる．ここではコア 0〜コア k 上で動作する通常のスレッドを**コンピューティングスレッド (computing)**，コア k+1 上で動作するキャッシュ置換制御を行うスレッドを**ヘルパースレッド (helper)** と定義し，説明する．

ヘルパースレッドによるキャッシュ置換制御支援の一連の流れと，その際に必要となるハードウェアの拡張は次の通りである．

(1) キャッシュミスイベントの検知と取得：
通常，共有ラストラベルキャッシュでコンピューティングスレッドのロード・ストア命令によるキャッシュミスが生じた場合，キャッシュミスの情報が MSHR へ書き込まれる．ここで，キャッシュミス情報はロード・ストア命令によってアクセスされた参照アドレスのみでなく，各コアがそれぞれ持つ識別用の番号であるコア ID と，ロード・ストア命令の仮想アドレスを保持するように拡張する．

(2) ヘルパースレッドによるキャッシュミスイベントの読み込み：
ヘルパースレッドは，定期的に MSHR にキャッシュミス情報があるかをチェックし，ある場合は MSHR からそれを読み出す．この際，ヘルパースレッドはメモリマップド I/O 方式により，通常のロード命令により MSHR の情報を読み出せるようにする．取得したキャッシュミス情報をを基に，当該ロード・ストア命令によってキャッシュに読み込まれるデータに再利用性があるかどうかを予測する．ヘルパースレッドの再利用性予測アルゴリズムについては 3.3 節にて述べる．

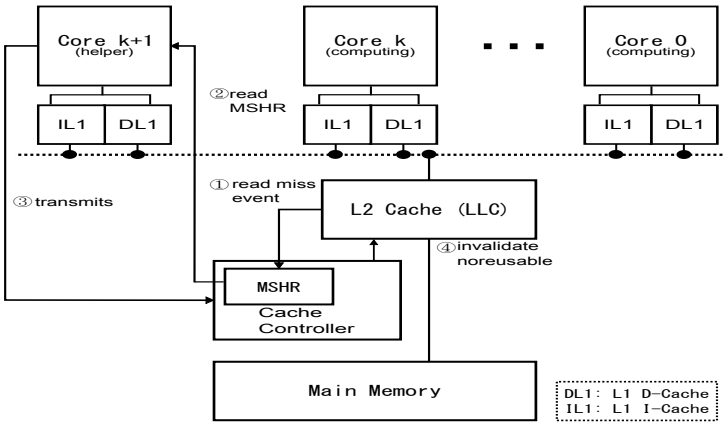

図 1 ヘルパースレッドによるキャッシュ置換制御手法の全体図

(3) キャッシュコントローラへの制御情報通知：
ヘルパースレッドが再利用性が低いと判断したキャッシュミスイベントに関しては，そのデータを追い出し易くするために，MSHR 上の置換制御フラグをセットする．この際にも，メモリマップド I/O 方式により通常のストア命令によりフラグの書き込みを行う．

(4) キャッシュ置換制御：
ヘルパースレッドから MSHR に書き込まれた置換制御フラグを基に，共有キャッシュに対してキャッシュ置換制御を実行する．置換制御に関しては，次節で詳述する．

3.2 キャッシュ置換制御支援

主記憶にアクセスしたデータがプロセッサチップに読み込まれると，キャッシュコントローラは LRU などの置き換えアルゴリズムに従って置換対象のラインを選択し，新しいラインをその置換対象ラインと置き換える．この際に，例えば LRU 置き換えアルゴリズムであれば，当該ラインを MRU ラインと設定するなど，置換制御用の情報を更新する．本提案手法では，再利用性がない判断されたラインについて，次に当該セットでキャッシュミスが発生した際に置換対象ラインとして選択され易くなるように，この置換制御用情報部分の更新を拡張する．なお本稿では，ベースとなるライン置き換えアルゴリズムとして LRU アルゴリズムを用いることとし，以降ではこの LRU アルゴリズムを前提として説明する．

主記憶からデータが読み込まれた際に，キャッシュコントローラは MSHR の当該エントリの置換制御フラグをチェックし，次のどちらかの処理を行う．

(1) キャッシュ置換制御フラグがセットされていない：
再利用性があるラインとして，LRU 方式に基づき MRU ラインとして当該ラインの LRU 情報を更新する．

(2) キャッシュ置換制御フラグがセットされている：
再利用性がないラインと見なし，キャッシュから追い出されやすくなるように LRU ラインとして当該ラインの LRU 情報を更新する．

なお，再利用性がないラインとして判定されたラインにおいて，ミスによりキャッシュにデータを読み込んでから，直後に再び数回のアクセスがあり，その後キャッシュから追い出されるまでアクセスされないような場合がある．この際，直後のアクセスで当該ラインを MRU ラインとしてしまうと，本手法により追い出されやすくしたラインがすぐには追い出されなくなってしまう．そこで，キャッシュ置換制御フラグがセットされていた場合は，そのラインへアクセスがあったとしても，LRU 情報の更新は行わないように制御する．このために，各キャッシュラインへ LRU 制御抑制ビットを設け，再利用性がないと判断されたラインに対しは，置き換え時に本ビットをセットすることで対応する．

3.3 ヘルパースレッドによるキャッシュ置換制御アルゴリズム

ヘルパースレッドは，ハードウェアからキャッシュミス情報を取得し，それを基に当該キャッシュミスデータの再利用性の予測する．再利用性がないと判断した場合に限り，置換制御フラグのセットを行う．本稿ではヘルパースレッドによる再利用性の判断として，ストリームベースと，プロファイルベースの 2 つの手法を検討する．

ストリームベース手法

ストリームベース手法は，あるロード・ストア命令によるデータアクセスが，配列を一定間隔で連続的に参照するような場合，すなわちストリームアクセスを検出した場合，その命令は再利用性が少ないと判断する手法である．ストリームベースのキャッシュ置換制御アルゴリズムの擬似コードを**図 2** に示す．

ヘルパースレッドは，キャッシュミスが発生した場合に，アクセスされたデータの再利用性を判断するために，アクセス履歴テーブルを作成する (図 2 の 1 行目)．このアクセス履歴テーブルは，ロード・ストア命令のアドレスをインデックスとしてアクセスするハッシュテーブルの構造を持ち，ロード・ストア命令毎にアクセスしたアドレスの情報を管理する．具体的には，どのコアからのキャッシュミスイベントか判断するコア ID，各ロード・ストア命令がアクセスしたアドレス，および前回のアクセス時のアドレスとのアドレス差（以降，

```
1.  RPT ← Create_Reference_Prediction_Table(TABLE_SIZE)
2.  while Computing-threads run do
3.    curMSHR = Read_current_MSHR()
4.    if !curMSHR /* there is no miss event */ then
5.      continue
6.    end if
7.    if (Entry ← Get_RPT_entry(curMSHR_inst_addr)) != NULL then
8.      /* a corresponding Reference Prediction Table entry exists */
9.      Curret_Addr ← Entry_prev_addr + Entry_stride
10.     Entry_stride ← curMSHR_addr - Entry_prev_addr
11.     Entry_prev_addr ← curMSHR_addr
12.     if curMSHR_addr == Curret_Addr && Entry_state == STEADY then
13.       Set_Replacement_Control_flag()
14.     end if
15.     Entry_state ← Update_State(Entry_state, Current_addr, curMSHR_addr)
16.   else
17.     /* Initialize an entry for coresponding load/store instruction address */
18.     Set_RPT_entry(curMSHR_inst_addr, curMSHR_addr)
19.   end if
20. end while
```

図 2 ストリームベースのキャッシュ置換制御を行うヘルパースレッドアルゴリズム

ストライドと呼ぶ）をアクセス履歴テーブルに記録する．このテーブル表は，キャッシュプリフェッチのために，Chen らが提案した Reference Pred iction Table (RPT)[8] をベースに，本提案手法向けに拡張したものである．前回参照されたアドレス ($Entry_{prev\_addr}$) とストライド ($Entry_{stride}$) との和を予測アドレス ($Current\_Addr$) とし，現在のミスアドレスと一致する場合，それをキャッシュ置換制御の対象とする．

コンピューティングスレッドが実行終了するまでの間，ヘルパースレッドは動作し続け (図 2 の 2 行目)，キャッシュミスが発生し，MSHR にキャッシュミス情報が登録されるのを待ち続ける (図 2 の 3〜6 行目)．MSHR からキャッシュミスイベントを取得すると，取得した情報のロード・ストア命令のアドレスを取り出し，それに一致するエントリがアクセス履歴テーブルにあるかチェックする (図 2 の 7 行目)．一致するエントリがあった場合，予測アドレスを計算し，参照されたアドレスとそのストライド値を更新する (図 2 の 9〜11 行目)．

予測アドレスが現在のキャッシュミスしたアドレスと一致し，かつ当該エントリの状態が予測可能状態 (STEADY)，つまり前回までのデータアクセスにおいてストライド値が一定であったならば，そのロード・ストア命令はストリームアクセスを行うものである．すなわち，本命令によりアクセスされたデータはある時間間隔においては再利用性のないデータであるととみなす．そこで，キャッシュコントローラへキャッシュ置換制御の対象命令であることを通知するために，MSHR のキャッシュ置換制御フラグをセットする (図 2 の 12〜14 行目)．

また，当該エントリの状態をこれまでの状態，および今回アクセスされたアドレスの情報をもとに更新する (図 2 の 15 行目)．一方，アクセス履歴テーブルに一致するエントリがない場合，新たにそのキャッシュミスイベントを記録するエントリを追加する (図 2 の 18 行目)．

プロファイルベース手法

プロファイルベース手法は，あらかじめロード・ストア命令についてプロファイリングを行い，再利用性の低いデータアクセスを行う命令を既知としてファイルに保存しておき，その情報に基づいて再利用性を判断する手法である．プロファイルベースのキャッシュ置換制御アルゴリズムの擬似コードを**図 3** に示す．図 2 のアルゴリズムとの違いは最初に RPT を作成をせず，プロファイリングしたデータを読む点 (図 2 の 1 行目) である．この手法では，プロファイルデータ中の再利用性の低いデータアクセスを行う命令に関する情報に基づいて，その命令をキャッシュ置換制御の対象とするか判断する．MSHR にキャッシュミス情報が登録されるのを待ち続ける (図 3 の 3〜6 行目) 点はストリームベース手法と同様であり，異なるのはデータの再利用性の予測を行う際に，ストライド値を計算するのではなく，プロファイリングデータ中にあるミスした命令アドレスが存在するかをチェックする点である (図 3 の 7〜9 行目)．図 2 のアルゴリズムに比べテーブル操作やストライドの計算が必要なく，高速に再利用性の予測ができると期待される．

4. 評　　価

4.1 評価環境

評価には，並列プログラムを含む X86 バイナリをシミュレート可能な Multi2Sim[10] をベースに，3.1 節で述べた提案手法のハードウェア拡張を実装した評価環境を用いた．

ベンチマーク・プログラムには SPEC2006 の中から共有キャッシュでの競合が起こりやすいプログラムをいくつか選択し，使用した．各プログラムは，最初の 10 億命令を Fast Forward し，そこからの 1 億命令を評価対象とする．なお，各スレッドで 1 億命令の実行が終了するタイミングは異なるため，スレッド毎に 1 億命令を実行した時点での統計情報

```
1.  profData ← Read_Profiling_Data()
2.  while Computing-threads run do
3.    curMSHR ← Read_current_MSHR()
4.    if !curMSHR then
5.      continue
6.    end if
7.    if isExist(curMSHR, profData) != 0 then
8.      Set_RPT_entry(curMSHR_inst_addr, curMSHR_addr)
9.    end if
10. end while
```

図 3 プロファイリング結果を用いたキャッシュ置換制御のアルゴリズム

を評価結果とするが，全コンピューティングスレッドが少なくとも 1 億命令を実行し終えるまで各スレッドの実行は継続するものとした．また，1 コアで 1 スレッドを実行するものとし，SMT (Simultaneously Multh-Threading) は使用しない．

評価に用いるシミュレータのパラメータを**表 1** に示す．評価の対象となるプロセッサは，各コアで専用の命令／データキャッシュを持ち，また共有ラストレベルキャッシュとして L2 キャッシュを持つ．L1 キャッシュのコヒーレンスは，MOESI プロトコルにより制御される．

評価では以下の手法の性能を比較する．

- Base: 通常の LRU 置き換えアルゴリズムを用い，コンピューティングスレッドのみを実行した場合．
- $\mathsf{Stride}_{\mathsf{helper}}$: ストリームベースの再利用性判定手法を用い，ヘルパースレッドによりキャッシュ置換制御の支援を行った場合．
- $\mathsf{Profile}_{\mathsf{helper}}$: プロファイルベースの再利用性判定手法を用い，ヘルパースレッドによりキャッシュ置換制御の支援を行った場合．
- $\mathsf{Stride}_{\mathsf{hard}}$: ストリームベースの再利用性判定手法をハードウェア上に実装しキャッシュ置換制御の支援を行った場合．
- $\mathsf{Profile}_{\mathsf{hard}}$: プロファイルベースの再利用性判定手法をハードウェア上に実装しキャッシュ置換制御の支援を行った場合．

ヘルパースレッドによるもの ($\mathsf{Stride}_{\mathsf{helper}}$，$\mathsf{Profile}_{\mathsf{helper}}$) に加えて，ハードウェア上で再利用性予測アルゴリズムを用い，同様にキャッシュ置換制御支援を行うもの ($\mathsf{Stride}_{\mathsf{hard}}$，$\mathsf{Profile}_{\mathsf{hard}}$) についても評価する．これにより，同様の置換制御支援をソフトウェアとハードウェアで行う場合を比較でき，ソフトウェアによって置換制御支援を行う上でのキャッシュミスイベントに対する追従性などを評価可能である．

表 1 評価に用いるプロセッサの仮定

Fetch/Issue/Commit	4 instruction / cycle
Branch prediction	Two level adaptive (1K choice table) Bimode (1K-entry) PHT (1K-entry)
BTB	256 sets, 4way
IQ size	40
LSQ size	20
Functional units	Int: 4 ALU, 1 Mult/Div, 4 Addr-gen. FP: 2 Add, 1 Mult/Div Load/Store: 2 ports
L1 I-Cache	32KB, 8-way, 64B block 2 clock cycles hit latency
L1 D-Cache	32KB, 8-way, 64B block 2 clock cycles hit latency
L2 Cache	1MB, 8-way, 64B block 9 cycle hit latency
Cache replacement policy	LRU
Coherence protocol	Write Invalidiate MOESI
Main memory	200 cycle latency

評価に用いるスレッド数は，コンピューティングスレッド数が 2 で，ヘルパースレッド数が 1 の場合，およびコンピューティングスレッド数が 4 で，ヘルパースレッド数が 1 の場合の二通りの評価を行った．

なお，ヘルパースレッドの命令コードやアクセス履歴テーブルを含むデータセットは L1 キャッシュ容量に収まるものとし，ヘルパースレッドは L1 キャッシュに 100%ヒットするとして評価を行った．これはキャッシュラインのロックや，スクラッチパッドメモリの技術を用いることで容易に実装可能であると考えられる．

4.2 キャッシュ置換制御の評価

本稿では評価指標に Weighted Speedup[9] を使用する．1 つのコンピューティングスレッドのみ実行した時の IPC を基準とし，それに対して，複数のコンピューティングスレッドを同時に実行した時の IPC との比率を求めその合計値を指標とする．

2 スレッドの場合の評価結果

まず最初にコンピューティングスレッド数が 2，ヘルパースレッド数が 1 の場合について評価行う．コンピューティングスレッド数が 2 つの時の Weighted Speedup を**図 4** に，ま

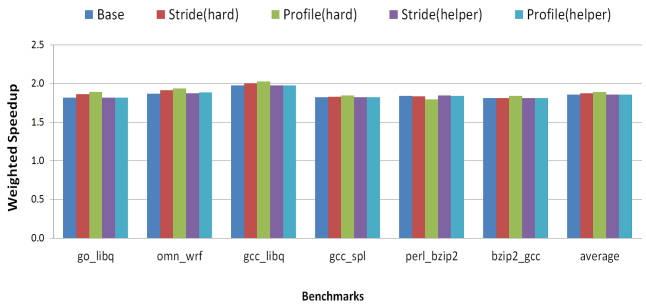

図 4 Weighted Speedup（コンピューティングスレッド数: 2）

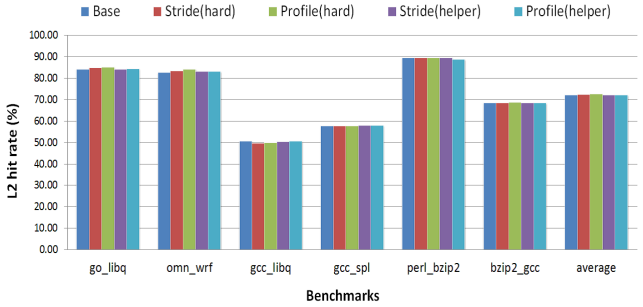

図 5 L2 キャッシュヒット率（コンピューティングスレッド数: 2）

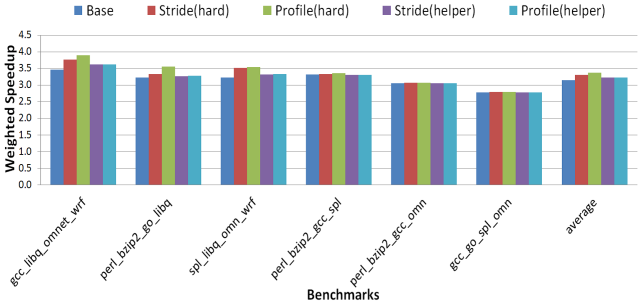

図 6 Weighted Speedup（コンピューティングスレッド数: 4）

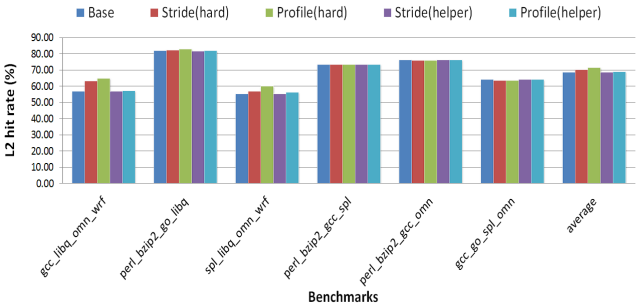

図 7 L2 キャッシュヒット率（コンピューティングスレッド数: 4）

た，この際の共有ラストレベルキャッシュのヒット率を**図 5** に示す．

図 4，図 5 の横軸は，各ベンチマークの組み合わせ中で，左の 3 つがキャッシュ競合が頻繁に発生する組み合わせ，次の 3 つがキャッシュの競合がほとんど発生しなかった組み合わせ，一番右が平均を示している．図 4 から，キャッシュの競合が頻繁にあった組み合わせの場合でも，ハードウェアベースの手法，ヘルパースレッドによる手法ともに Weighted Speedup はほとんど改善していないことがわかる．最大でも，$\mathsf{Stride}_{\mathsf{hardware}}$ において，gobmk+libquantum の組み合わせの場合に Weighted Speedup が約 3.8%の改善にとどまった．また，ヘルパースレッドを用いた $\mathsf{Stride}_{\mathsf{helper}}$ と $\mathsf{Profile}_{\mathsf{helper}}$ のどちらの場合でも，Weighted Speedup が約 0.8%程度しか改善されなかった．図 5 を見ると，共有ラストレベルキャッシュのヒット率もほとんど改善していないことがわかる．これは，コンピューティングスレッドが 2 スレッドの場合では，共有ラストレベルキャッシュにおける競合の頻度がそもそも低く，提案するキャッシュ置換制御手法が適用される頻度が低いためと考えられる．そのため，キャッシュの競合があまりなかった組み合わせでは，ハードウェアやヘルパースレッドを用いた手法によるキャッシュ置換制御の支援は Weighted Speedup 改善に貢献しなかったという結果となった．

4 スレッドの場合の評価結果

次にコンピューティングスレッド数が 4，ヘルパースレッド数が 1 の場合について評価結果を示す．コンピューティングスレッド数が 4 つの時の Weighted Speedup を**図 6** に，共有ラストレベルキャッシュのヒット率を**図 7** に示す．

コンピューティングスレッド数が 2 と 4 を比較すると，4 スレッドの方がより Weighted Speedup の改善が見られ，$\mathsf{Stride}_{\mathsf{hardware}}$ 手法を用いた場合に Weighted Speedup が最大で約 12.4%，$\mathsf{Stride}_{\mathsf{helper}}$ 手法を用いた場合に約 5.6%改善した．これはコアの数を増やすこと

によって共有ラストレベルキャッシュにおけるキャッシュ競合の頻度が上がり，キャッシュ置換制御手法を適用する機会が増えたためと考えられる．そのため，同時に実行されるスレッド数が多く共有キャッシュ競合が起こりやすい場合には，本提案手法により効果的に共有キャッシュにおけるキャッシュ競合の発生頻度を抑制できる可能性があると言える．

図 5 から，コアの数が増加することによって Weighted Speedup が改善したのと同様に，共有ラストレベルキャッシュのヒット率の改善も見られた．gcc+libquantum+omnetpp+wrf の組み合わせの時，共有ラストレベルキャッシュのヒット率が 6.4 ポイント改善し，キャッシュ置換制御手法による再利用性の高いデータを保持することがキャッシュヒット率の向上に繋がることがわかった．

カバレッジ

図 8 にハードウェアによるキャッシュ置換制御支援と比較した場合に，ヘルパースレッドでどの程度キャッシュ置換制御支援が行えたかの割合，すなわちハードウェア制御に対するヘルパースレッド制御のカバレッジの値を示す．図からもわかるように，ヘルパースレッド制御によるカバレッジは，ストリームベース／プロファイルベース共に低く，約 2%〜26%程度しかないことがわかった．スレッド数が増加するとキャッシュミスの頻度が高くなり，キャッシュ競合の頻度も高くなるが，今回のようにヘルパースレッドを 1 コアのみで動作させる場合カバレッジがに低下してしまう．コンピューティングスレッド数が 4 スレッドの場合の評価において，ハードウェアで制御した場合に比べ，ヘルパースレッドを用いた場合の性能向上が低かったのは，このカバレッジの低さが原因であると考えられる．

本評価の結果，スレッド数に関わらずヘルパースレッドによるキャッシュ置換制御手法は大きな性能向上を得ることができなかった．ヘルパースレッドはデータ再利用性の予測に数十から百数十命令分の実行時間を要するので，キャッシュミスイベントに対して十分に制御が追いついていないためであると考えられる．この点は，ヘルパースレッドのコードをさらに最適化すること，あるいはヘルパースレッドを複数コアで実行するなどで改善できると考えられ，本研究の今後の課題である．

5. おわりに

本稿ではマルチコア・プロセッサにおける共有キャッシュの競合に着目し，ヘルパースレッドによるキャッシュ置換制御手法の検討とその評価を行った．共有ラストレベルキャッシュで起きたキャッシュミスイベントをヘルパースレッドが取得できるように，ミス情報を格納する MSHR をソフトウェアから読み込むための専用経路，キャッシュ置換制御を行うキャッシュコントローラへ情報を設定する専用経路の追加などを実装し，評価を行った．

図 8 ヘルパースレッドで置換制御を実行できた割合

評価の結果，共有ラストレベルキャッシュにおけるキャッシュの競合頻度が高い場合，キャッシュ置換制御手法を用いることによって，競合を抑制し，性能を向上できる可能性があることを確認した．ただし，現状ではヘルパースレッドの制御速度がキャッシュミスイベントの発生頻度に十分追いつけず，性能向上率が高くないことがわかった．

今後の課題としては，まずヘルパースレッドのコード最適化やヘルパースレッドを複数コアで実行するなどでカバレッジを改善することがあげられる．また，再利用性が低いと判断されるデータブロックをより高い精度かつ高速に予測できるアルゴリズムを研究することも重要である．そして，ソフトウェアによる制御という特徴を生かして，実行しているプログラムの特徴に合わせて適応的にキャッシュ置換の補助制御を行うとこも今後の課題である．

謝辞 本研究の一部は，独立行政法人新エネルギー・産業技術総合開発機構（NEDO）のプロジェクト「極低電力回路・システム技術開発（グリーン IT プロジェクト）」の支援により行われたものである．

参 考 文 献

1) 今里賢一, 福本尚人, 井上弘士, 村上和彰： 適応的ヘルパースレッド実行に基づくマルチコア向け演算/メモリ性能バランシング. 情報処理学会研究報告.2009-ARC-183(16), 2009.
2) Hong Wang, Perry H.Wang, Ross Dave Weldon, Scott M.Ettinger, Hideki Saito, Milind Girkar,Steve Shih-wei Liao,John P. Shen：スペキュレーティブ・プリコンピュ

テーション：マルチスレッディング・リソースの活用によるレイテンシの削減, *Intel Technology Journal Q1*, 6, 1, 2002.
3) Doug Joseph and Grunwald, Dirk, Prefetching using Markov predictors, *Proceedings of the 24th annual international symposium on Computer architecture*, pp.252–263, 1997.
4) Ilya Ganusov, Martin Burtscher, Efficient emulation of hard-ware prefetchers via event-driven helper threading, *Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT2006)*, pp. 144-153, 2006.
5) Md Kamruzzaman and Steven Swanson and Dean M. Tullsen, Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads, *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS2011)*, vol.46, pp.393–404, 2011.
6) Marios Kleanthous and Yiannakis Sazeides, CATCH: a mechanism for dynamically detecting Cache-Content-Duplication and its application to instruction caches, *Proceeding of the conference on Design, automation and test in Europe (DATE 2008)*, pp.1426-1431, 2008.
7) Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen, Speculative precomputation on chip multiprocessors, *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.
8) Tien-Fu Chen, Jean-Loup Baer, Effective Hardware-Based Data Prefetching for High-Performance Processors, *IEEE Transactions on Computers*, 44, pp.609-623, 1995.
9) Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, Keith I. Farkas, Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance, *Proceedings of the 31st International Symposium on Computer Architecture, June, 2004*, 32, 2, 2004.
10) http://www.multi2sim.org/