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提案

橋本崇浩[†] 近藤正章[†]
和田康孝[†] 本多弘樹[†]

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，増加するコアの有効利用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れ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提案する．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は次に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，また再利用性の高いデータは追い出されにくくなるように制御することで競合の緩和を狙う．ソフトウェアにより制御することで，アプリケーションに合わせてキャッシュ置換制御を最適化することができるため，有効な手法になり得ると考えられる．複数の再利用性予測アルゴリズムを用いて，本手法の評価を行った結果，提案手法によって最大で 11%程度，性能を向上させることが可能であることがわかった．

Assisting Cache Replacement by Helper-Threading for Chip-Multiprocessor

Takahiro Hashimoto,[†] Masaaki Kondo,[†] Yasutaka Wada[†]
and Hiroki Honda[†]

As the number of cores on multicore processors increases, idling cores also increase because most of today's applications cannot fully utilize multiple cores. Effectively utilizing idle cores is an important research topic. The shared cache contention between multiple threads in multicore processors is still a big issue to be addressed. In this study, we propose a helper-threading approach to mitigate the cache contention by making good use of data locality. The helper thread assists a cache replacement decision by estimating data locality so that the cache replacement can be flexibly optimized by software depending on the characteristics of executing applications. We evaluate the proposed technique and the results reveal that, at maximum, about 11% of higher performance is obtained by our helper-threading.

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は重要な課題である．

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，共有キャッシュにおいて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

† 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

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問題の緩和に取り組む手法として，ヘルパースレッド[1)~4)]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，メイン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，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図る手法を提案する．共有キャッシュ上で生じたキャッシュミス情報を基にどのデータをキャッシュに残すべきか予測を行う専用のスレッドをヘルパースレッドとして動作させる．再利用性の高いデータはデータ置換の対象になりにくいように Most Recently Used (MRU) の位置に，一方，再利用性の低いデータは再利用性のあるデータを追い出さないよう Least Recently Used (LRU) の位置にデータを挿入することによって，再利用性の高いデータがキャッシュに残りやすくなるよう制御する．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の緩和による性能の向上を目指す．

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[5)~7)]．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依存する．よって，ある特定の手法があらゆるキャッシュ競合下で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．

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[2)~4),8)]．

Kamruzzaman らの手法[4)]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でアクセスしようとした際には，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，キャッシュ上のデータを利用することで，メモリアクセスレイテンシの隠蔽を狙う．

今里ら[2)]や Ganusov ら[8)]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際に，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，取得した情報を基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への対処を目的としたものではない．また，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上の有用なデータが追い出され，競合がより深刻化する可能性もある．

また，Kandemir らはマルチスレッドプログラム向けに，ヘルパースレッドによって適切なキャッシュ分割割合を求めつつ，それをハードウェアに通知してキャッシュ分割を行うことで，平均メモリアクセス時間を最小化する手法を提案している[12)]．ただし，データの再利用性についてはあまり考慮されていない．

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利用法とは異なる．

3. ヘルパースレッドによるキャッシュ置換制御手法

本章では，共有ラストレベルキャッシュ(LLC) を持つコア数 k+1 のプロセッサ環境を例として，ヘルパースレッドがデータの再利用性予測を行い，その予測結果を用いてキャッシュ置換制御を行うまでの一連の流れについて述べる．また，ヘルパースレッドによるキャッシュ置換制御に必要なハードウェアの拡張と，ヘルパースレッドによる再利用性予測アルゴリズムについて述べる．なお，キャッシュミス情報を格納する *Miss State Holding Register (MSHR)* を持つプロセッサを前提とする．

将来的にコア数が増加すると，全コアで LLC を共有するのは現実的ではなく，数コアから十数コア程度の単位で LLC を共有することが多いと考えられる．

その際には LLC を共有するグループ毎にヘルパースレッドを実行する必要があるが，本章で述べる提案手法はそのグループ毎に独立に適用できる．そのため，本稿では 4〜8 コア程度で LLC を共有することを想定して，説明および評価をを行う．

3.1 ヘルパースレッドの動作

図 1 にヘルパースレッドによるキャッシュ置換制御支援手法の全体図を示す．図 1 では，コア 0 からコア k 上でコンピューティングスレッドを，コア k+1 上でヘルパースレッドが動作している様子を示している．

まずヘルパースレッドは遊休状態であるコアで動作しており，共有の LLC でロードミスが起きたと仮定する．通常，LLC 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，キャッシュミス情報は MSHR へ書き込まれ，主記憶へデータ・アクセスが行われる．以下ヘルパースレッドの動作の流れを説明する．

(1) キャッシュミス情報の読み出し

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，キャッシュミス情報がある場合 MSHR からそれを読み出す (図 1 の (1))．なお，この情報読み出しはメモリマップド I/O 方式により行われ，通常のロード命令により読み出しが可能である．

(2) 再利用性の予測

取得したキャッシュミス情報を基に，そのミスしたデータのアドレス，あるいはそのアドレスが含まれるページ全体の再利用性を予測する (図 1 の (2))．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．

(3) 置換制御情報の通知

データの再利用性予測を行った結果に従い，キャッシュ置換制御のための情報をキャッシュコントローラへ通知する (図 1 の (3))．ヘルパースレッドにより通知される置換情報は，新たに追加する置換制御テーブルへ格納する．置換制御テーブルについては 3.2 節にて述べる．キャッシュコントローラは，予測した再利用性に基づき，再利用性が高いと判断されたデータは追い出されにくく，再利用性が低いと判断されたデータは追い出されやすくなるように，キャッシュ置換制御を行う．

以上のように，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るようにキャッシュ置換制御の補助を行う．これにより，共有キャッシュ上での競合が緩和されると期待できる．また，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と比較して，有用な手法になり得ると考えられる．

3.2 ハードウェアの拡張

本研究の提案手法を用いるには，ソフトウェアから

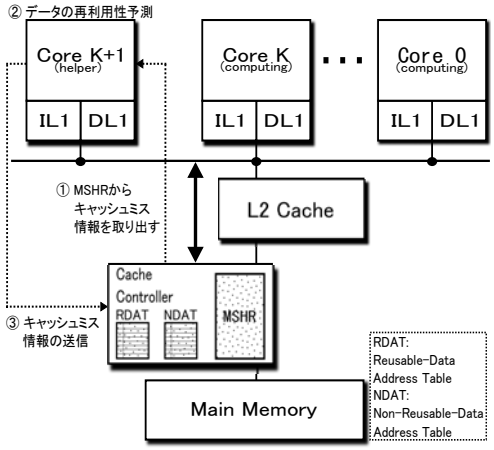

図 1 ヘルパースレッドによるキャッシュ置換制御支援手法の全体図

のキャッシュミス情報の取得やキャッシュデータの再利用性結果の通知，またそれに基づいた置換制御が行えるように，ハードウェアを拡張する必要がある．

MSHR に保存する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物理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．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．そのため，MSHR をソフトウェアから読み込むための専用経路が必要になる．同じく，キャッシュコントローラ (CacheController) へ情報を設定する専用経路を新たにハードウェアに追加する．

通常の LRU 以外の置換制御を行えるようにするために，キャッシュコントローラ自体にも拡張が必要となる．キャッシュコントローラにはヘルパースレッドにより通知されたデータの再利用性予測情報を記録できるように，次の 2 つの置換制御テーブルを新たに追加する．

- RDAT: Reusable-Data Address Table
  再利用性があると判断したアドレスを保持するテーブル．キャッシュミスが生じた際に，RDAT 中に当該アドレスが属するページがある場合，データは MRU 位置へ挿入される．
- NDAT: Non-Reusable-Data Address Table
  再利用性がないと判断したアドレスを保持するテーブル．キャッシュミスが生じた際に，NDAT 中に当該アドレスが属するページがある場合，データは LRU 位置へ挿入される．

共有キャッシュでキャッシュミスが生じた際，キャッシュコントローラはこれらの置換制御テーブルへアクセスを行い，キャッシュミスアドレスと比較する．本提案手法では，基本的に LRU 方式に基づいてキャッシュ置換制御を行われるが，キャッシュミスしたデー

```
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_NDAT_entry(curMSHR_addr)
9.   end if
10.  end while
```

図 2 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

タを下位階層のメモリからリードし，キャッシュに書き込む際の位置が異なる．通常は MRU 位置にデータを挿入するが，提案手法では RDAT 中に当該アドレスが属するページがある場合，データは MRU 位置へ挿入され，NDAT 中に当該アドレスが属するページがある場合，データは LRU 位置へ挿入される．両者のテーブルともに当該アドレスが属するページが存在しない場合には，セット上の MRU と LRU の中間位置に挿入する．これらのテーブルを利用するかどうかは，オプションとしてヘルパースレッドから制御可能とする．

3.3 ヘルパースレッドのデータの再利用性予測アルゴリズム

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，それを基にキャッシュデータの再利用性予測を行い，その結果に応じて RDAT と NDAT へのアドレス情報の書き込みを行う．本稿では，ヘルパースレッドの再利用性予測アルゴリズムとして，プロファイルベース手法，ストリームベース手法，仮想拡張セット手法の 3 つを提案する．

3.3.1 プロファイルベース手法

プロファイルベース手法は，あらかじめ各プログラムをプロファイル実行し，共有キャッシュ上で生じるキャッシュミス情報のログを収集する．このキャッシュミス情報から再利用性を予測する．再利用性の予測には，取得したキャッシュミスログをオフラインで解析し，あるプログラム実行期間中に 1 度しかメモリアクセスされないデータブロックを再利用性の低いデータと定義する．この，再利用性の低いデータブロックのアドレスリストを保存しておき，ヘルパースレッドはその情報に基づいて再利用性を判断する．

実行時にはまず，コンピューティングスレッドとして実行されているプログラムに対応するアドレスリストを自身のメモリ (スクラッチパッドメモリ) にロードする．MSHR からキャッシュミスしたアドレスを取得し，アドレスリストに一致するアドレスがある場合には，NDAT にそのページのアドレスを書き込む．なお，本手法では再利用性ありに関するアドレス情報は用いていないため，RDAT は使用しない．

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 2 に示す．まず最初に，プロファイリングにより得られたアドレスリストのロード

```
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))
8.      != NULL then
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
13.       && Entry_state == STEADY then
14.      Set_NDAT_entry(Current_Addr)
15.    end if
16.    Entry_state ← Update_State(Entry_state,
17.         Current_addr, curMSHR_addr)
18.  else
19.    /* Initialize an entry for coresponding
20.      load/store instruction address */
21.    Set_RPT_entry(curMSHR_inst_addr, curMSHR_addr)
22.  end if
23.  end while
```

図 3 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

を行う (図 2 の 1 行目)．そして，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情報を取得した後，アドレスリスト中にミスしたメモリアドレスが存在するかチェックを行い，存在した場合にはそのページ情報を NDAT に設定する (図 2 の 7〜9 行目)．後述する手法とは異なり，データの再利用性予測のための必要な処理が単純であり，高速にデータの再利用性予測結果をキャッシュコントローラへ通知できると期待される．

3.3.2 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，その命令がアクセスするメモリ領域は再利用性がないと判断する手法である．この手法はストライド・プリフェッチの際に用いられるアルゴリズムをベースに考案したものである．なお，本手法でも再利用性ありに関する予測は行わず，RDAT は使用しない．

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 3 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 3 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たハッシュ構造を持ち，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，ストライドと呼ぶ）をアクセス履歴テーブルに記録する．この履歴テーブルは，キャッシュプリフェッチのために，Chen らが提案した Reference Prediction Table (RPT)[9] 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 3 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 3 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をチェックする (図 3 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 3 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．つまり，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みなし，NDAT へそのデータが含まれるページを登録する (図 3 の 12〜15 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 3 の 17 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情報をエントリへ追加する (図 3 の 21 行目)．

3.3.3 拡張仮想セット手法

拡張仮想セット手法は，再利用性が高いにも関わらず競合により共有キャッシュから追い出され，すぐにまたアクセスされるデータも多いことに着目し，追い出されたアドレス情報に基づいてデータの再利用性予測を行う手法である．本手法は，共有キャッシュから追い出されたデータのアドレスをヘルパースレッドが記録しておき，それが再びアクセスされるかをチェックする．そして，アクセスされたタイミングによってデータの再利用性の有無を決定する．

共有キャッシュから追い出されたデータをチェックする際，ヘルパースレッドで共有キャッシュ内の全セットをチェックするのは現実的に困難である．そこで共有キャッシュの一部セットをサンプリング対象とする．プロファイル手法やストリームベース手法とは異なり，この手法が収集する情報は共有キャッシュにおけるキャッシュミス情報ではなく，サンプリング対象となっているセットでキャッシュミスが発生した際に追い出されたデータのアドレスである．その情報を MSHR より収集することで，データの再利用性予測を行う．

図 4 に本手法による再利用性予測の概要を示す．共有キャッシュから追い出されたラインのアドレス管理のために，ヘルパースレッドのスクラッチパッドメモリ内に，サンプルセットから追い出されたラインのタグやコア番号を管理するテーブルを作成する．このテーブルは，そのサンプルセットが実際の共有キャッシュよりも連想度の高い仮想的なウェイがあるかのように情報を管理する．ヘルパースレッドが管理するセットを仮想セットと呼び，実際の共有キャッシュのセットは実セットと呼ぶことにする．仮想セットは実セットで追い出された情報を保持し，LRU で置換が行われるかのように振る舞う．実セットでキャッシュミスが生じても，仮想セット上でヒットすることがある．

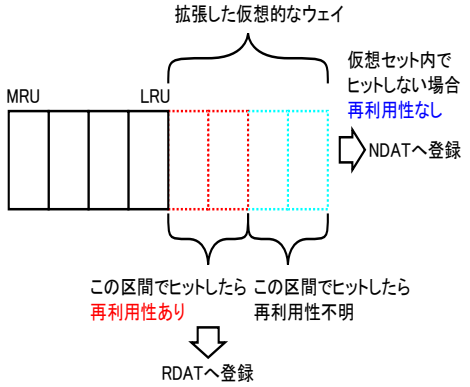

図 4 仮想拡張セット手法

```
1.  extensionArray ← Extension_Array(WAY)
2.  while Computing-threads run do
3.    curMSHR ← Read_current_MSHR()
4.    if !curMSHR then
5.      continue
6.    end if
7.    wayNo ← isHit(curMSHR, extensionArray)
8.    if isNotExist(wayNo, extensionArray) then
9.      Set_NDAT_entry()
10.   else
11.     if isReusableWayNo(wayNo, extensionArray) then
12.       Set_RDAT_entry()
13.     else
14.       Set_UNKNOWN_entry()
15.     end if
16.   end if
17. end while
```

図 5 仮想拡張セット手法のキャッシュ置換制御のアルゴリズム

本提案手法では，仮想セット上でヒットした場合に，どの位置でヒットしたかにより再利用性を判断する．例えば，図 4 にあるように，4 ウェイ分の情報を仮想セットで管理する場合を考える．この場合，仮想セット内の MRU 側の 2 ウェイ部分でヒットした場合は，再利用性が高いと見なし，そのアドレスを含むページを RDAT へ登録する．一方で，仮想セットからも追い出された場合には，再利用性がないラインであると見なし，そのアドレスを含むページを NDAT へ登録する．仮想セット内の LRU 側の 2 ウェイでヒットした場合は，再利用性が中程度あると考え，どちらのテーブルにも登録しない．従って，そのデータは再びキャッシュミスが生じた際には，前節で述べたように実セット上の中間位置に挿入されることになる．

仮想拡張セット手法のキャッシュ置換制御アルゴリ

表 1 ベンチマークの組み合わせ

MA-high	MA-low	MA-mix
401.bzip	444.namd	400.perlbench
410.bwaves	459.GemsFDTD	471.omnetpp
433.milc	465.tonto	459.GemsFDTD
462.libquantum	473.astar	444.namd
483.xalancbmk	481.wrf	483.xalancbmk

ズムの擬似コードを図 5 に示す．まず共有キャッシュの一部セットからサンプリングした情報を保持するために，共有キャッシュより連想度の高い仮想的なウェイ数の配列を生成する (図 5 の 1 行目)．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 5 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 5 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やコア識別用 ID などを取り出す．次に取得キャッシュミス情報に該当するデータが仮想セット内に格納されているかチェックを行う (図 5 の 7 行目)．この時，該当するウェイがない場合，それは再利用の低いデータとしてキャッシュコントローラへ通知する (図 5 の 9 行目)．一方，該当するウェイがあり，そのウェイが再利用性の高い位置と一致する場合，そのデータは再利用性が高いとみなし (図 5 の 12 行目)，そうでない場合はデータの再利用性は不明としてキャッシュコントローラへ通知する (図 5 の 14 行目)．

4. 評　　価

4.1 評価方法

評価には，PTLsim をベースに構築されたマルチコア・シミュレータ MARSSx86[11] を用いる．本シミュレータに対し，3.2 節で述べた提案手法のハードウェア拡張を実装した上で評価実験を行う．

ベンチマーク・プログラムには SPEC-CPU2006 の中からいくつかを選択し，表 1 に示すベンチマークの組み合わせ毎に評価を行った．*MA-high* はメモリアクセス回数の多いベンチマークの組み合わせ，*MA-low* はメモリアクセス回数の少ないベンチマークの組み合わせ，*MA-mix* はそれらをミックスした組み合わせを表している．ベンチマークプログラムは静的リンクによるコンパイルを行い，最適化オプションは-O3 とした．各プログラムは，最初の 10 億命令を FastForward 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，各スレッドで 1 億命令を実行し終えた時点での性能や各種統計情報を当該スレッドの評価結果として記録する．その上で，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し，最終的な評価結果を取得する．

表 2 評価実験環境

number of Cores	5
Fetch/Issue/Commit	4 instruction / cycle
Branch prediction	2-level adaptive (1K choice table) Bimode (1K-entry) PHT (1K-entry)
BTB	256 sets, 4way
IQ size	40
LSQ size	48
ITLB/DTLB size	32
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	4GB, 200 cycle latency

表 3 置換制御回数 (CT 数:4 HT 数:1 MA-low)

	組み合わせ	ミス回数	LRU 挿入	置換
prof	MA-low1	203	2 (0.99%)	1
	MA-low2	1,420	8 (0.59%)	6
	MA-low3	1,504	13 (0.84%)	11
	MA-low4	1,376	6 (0.46%)	6
	MA-low5	1,387	8 (0.61%)	10
strm	MA-low1	204	20 (9.98%)	16
	MA-low2	1,329	1,151 (86.58%)	290
	MA-low3	1,368	1,141 (83.37%)	295
	MA-low4	1,100	887 (80.56%)	254
	MA-low5	1,220	771 (63.14%)	256
vset	MA-low1	200	28 (14.04%)	22
	MA-low2	1,217	261 (21.42%)	207
	MA-low3	1,274	241 (18.91%)	193
	MA-low4	1,163	592 (50.91%)	325
	MA-low5	1,204	601 (49.85%)	365

なお，1 コアで 1 スレッドを実行するものとした．

評価に用いるシミュレータのパラメータを表 2 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．

評価には，プロファイルベース手法，ストリームベース手法，仮想拡張セット手法をそれぞれ用い，ヘルパースレッドを実行しない通常のキャッシュ置換制御をした場合との比較を行う．評価指標には Snavely らが考案した *Weighted SpeedUp*[10] を用いた．

RDAT，NDAT のエントリはヘルパースレッドにより適切に管理される必要がある．ただし，本評価では，ヘルパースレッドからのキャッシュ置換制御による性能向上の可能性を評価することに注力するために，RDAT，NDAT は十分多数のエントリ数を持つ，すなわち，エントリの置き換え制御は必要ないものとして評価を行った．ヘルパースレッドによるエントリの置き換え制御を含めた評価は今後の課題である．

4.2 評価結果

図 6 に評価結果を示す．横軸は，各ベンチマークの組み合わせで，縦軸は提案手法による通常のマルチコアに対する性能向上比を示している．また，*prof*,*strm*, *vset* はそれぞれプロファイルベース手法，ストリームベース手法，仮想拡張セット手法の結果を示している．また，図 7 は，各手法を用いた場合の L2 キャッシュのヒット率を，表 3 はそれぞれの手法での全ミス回

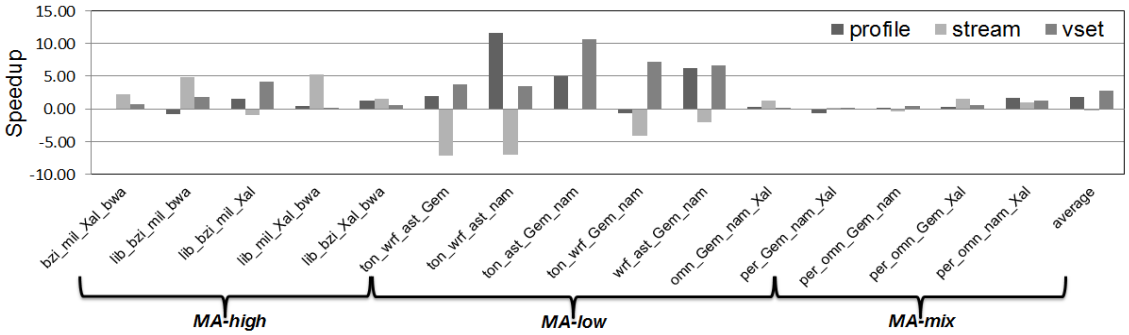

図 6 性能向上比

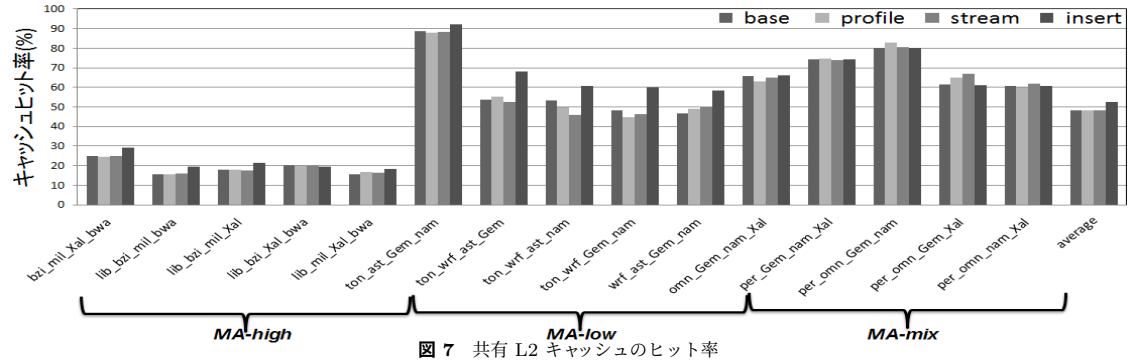

図 7 共有 L2 キャッシュのヒット率

数，NDAT テーブルのアドレスと一致したことにより LRU 位置に挿入された回数とその割合，また LRU 位置に挿入され，実際にすぐに置換されたラインの数を示している．これは MA-low のプログラムの組み合わせのみを表しており，図 6 の MA-low のアプリケーションの左から MA-low1 として表している．

図 6 より，通常の実行に比べて，ヘルパースレッドによりキャッシュ置換の補助を行うことで多くの場合で性能が向上していることがわかる．キャッシュミス情報をもとに，再利用性を予測して，キャッシュのデータ置換の際に，どこにラインを挿入するかを最適化することで，キャッシュミスが減少するためである．図 7 からも提案手法を用いることで実際に L2 キャッシュヒット率が向上していることがわかる．

特に，プロファイルベース手法と仮想拡張セット手法を用いた場合に性能向上が大きい．プロファイルベース手法では，あるプログラム実行期間に 1 度しか L2 キャッシュ上でアクセスされないデータのアドレスをプロファイリングによりあらかじめ抽出し，そのデータをキャッシュから追い出されやすく制御している．その結果，再利用性の高いデータがキャッシュ中に残りやすくなることから，キャッシュ競合が減少し性能が向上したと考えられる．このプロファイルベース手法により，最大で 11.5%の Weighted SpeedUp の向上が得られることから，あらかじめプロファインリング実行ができるような状況では，効果的な手法であると考えられる．

仮想拡張セット手法では，再利用性のないと考えられるデータを追い出しやすくすると同時に，キャッシュ追い出し情報をもとに，追い出された後にすぐに再びアクセスされるデータ領域を検出し，それらが追い出されにくくなるよう制御するため，より効果的に再利用性を活用できると考えられる．そのため，場合によってはプロファイルベース手法よりも良い性能を達成できることもある．また，平均で見た場合も，3.9%の Weighted SpeedUp の向上を達成しており，プロファイルベース手法の平均 1.8%よりも良い結果が得られている．これは表 3 に示されているように，LRU 位置に挿入されたラインが多い事にも起因している．

一方で，ストリームベース手法では，性能が逆に低下している場合も多い．これは，再利用性能の予測が間違っていることが原因であると考えられる．ストリームベース手法では，ストリーム型アクセスは大きな配列を順次アクセスしていくような場合が多く，データの再利用性がないことが多いという論拠をもとに，ストリーム型アクセスを検出したらそのデータ領域を追い出しやすくするものである．しかし，ストリーム型アクセスでも，配列サイズが小さく，多重ループで何回も同じ配列がアクセスされる場合には，本来再利用性の高いデータとなる．そのような配列を，キャッシュから追い出しやすくしてしまうと性能が大きく低下する．これが，今回ストリームベース手法がうまく働か

ない原因であると考えられる．しかし，MA-high の組み合わせでは，ストリームベース手法が一番良い結果となっている場合もある．メモリインテンシブなプログラムでは配列サイズが大きく，ストリーム型アクセスとなるデータ領域の再利用性がないため，本手法でも再利用性の予測が正しく行えたためと考えられる．

上記のように，ある 1 つの手法が種々のパターンで効果的に動作するとは限らない．特にプロファイルベース手法は，あらかじめプロファイリング実行が行える環境であれば有効であるが，プロファイル実行をする時間的余裕がない場合，または入力データセットが違うとメモリアクセスの振る舞いが大きく異なるような場合には利用することはできない．一方で，仮想拡張セット手法やストリームベース手法は，動的に再利用性を予測するため，どのような場合であっても適用することが可能である．さらに，その際の性能向上はプログラムの特性によっても異なる．提案手法は，各コアで実行するアプリケーションの組み合わせや，状況に応じてヘルパースレッドのプログラムを変更することで，様々な場合にも効果的に動作するように最適化することができるため有用である．これは，ハードウェアベースの手法に比べても，本提案手法の大きな利点であると考えられる．

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，またコア数の増加にともない増えてしまう遊休状態のコアの存在に着目し，ヘルパースレッドによるキャッシュ置換制御支援手法の提案とその評価を行った．提案手法は，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して再利用性を予測しつつ，キャッシュ置換制御のための情報をキャッシュコントローラに通知し，キャッシュ上でのデータ競合を抑えて性能向上を狙う手法である．

提案手法を評価した結果，共有ラストレベルキャッシュに対してヘルパースレッドによるキャッシュ置換制御を支援することで，最大で 11.5%の性能向上が達成できることを確認した．また，ある 1 つの手法が全てのパターンで効果的に動作するとは限らず，各コアで実行するアプリケーションの組み合わせや，状況に応じてヘルパースレッドのプログラムを変更することで，様々な場合にも効果的に動作するように最適化することができるため，有用な手法であると考えられる．

今後の課題としては，ヘルパースレッドのコードをより最適化する，あるいは別のアルゴリズムを考案するなどして，再利用性予測の精度を向上させることが挙げられる．また，実行中のプログラムに応じて，自動的にヘルパースレッドコードを選択することも必要になる．さらに，ハードウェアベースの手法と比較しつつ，種々のアプリケーションで評価することも今後の課題である．

謝辞 本研究の一部は，JSPS 科研費 24680004，ならびに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) H. Wang et al. “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*, 2002.
2) 今里賢一ら, 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 年 4 月.
3) J.A. Brown et al. “Speculative Precomputation on Chip Multiprocessors” *Proc. the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, Nov. 2001.
4) M. Kamruzzaman et al., “Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads”, *Proc.the 16th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS2011)*, pp.393–404, March 2011.
5) M. Kleanthous and Y. Sazeide, “CATCH: a Mechanism for Dynamically Detecting Cache-Content-Duplication and its Application to Instruction Caches”, *Proc. 2008 Design, Automation and Test in Europe (DATE2008)*, pp.1426–1431, March 2008.
6) A. Jaleel et al., “Adaptive Insertion Policies for Managing Shared Caches”, *Proc. the 17th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT2008)*, pp.208–219, Oct. 2008.
7) M.K. Qureshi et al., “Adaptive Insertion Policies for High Performance Caching” *Proc. the 34th Int’l Sym. on Computer Architecture,(ISCA 2007)*, pp.381–391, June 2007.
8) I. Ganusov and Martin Burtscher, “Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading”, *Proc. the 15th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT2006)*, pp.144–153, Sep. 2006.
9) T. Chen and J. Baer, “Effective Hardware-Based Data Prefetching for High-Performance Processors”, *IEEE Transactions on Computers*, Vol.44, No.5, pp.609–623, May 1995.
10) A. Snavely and D. Tullsen, “Symbiotic Jobscheduling for A Simultaneous Multithreading Architecture”, *Proc. the 9th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS2000)*, pp.234–244, Nov. 2000.
11) A. Patel et al. “MARSS: A Full System Simulator for Multicore x86 CPUs”, *Proc. the 48th Design Automation Conference (DAC2011)*, pp.1050–1055, June 2011.
12) M. Kandemir et al., “A Helper Thread Based Dynamic Cache Partitioning Scheme for Multithreaded Applications”, *Proc. the 48th Design Automation Conference (DAC2011)*, pp.954–959, June 2011.