Vol.2010-ARC-190 No.1
2010/8/3

選択的キャッシュ・アロケーション：マルチスレッド環境におけるキャッシュ利用効率の向上手法

堀部悠平[†1]　三輪忍[†1]　塩谷亮太[†2,†3]
五島正裕[†2]　中條拓伯[†1]

マルチスレッド実行環境における共有キャッシュでは，スレッド間でメモリ・アクセスが競合し，パフォーマンスが大きく低下してしまう恐れがある．そのため，従来より競合を緩和する研究が数多くなされてきた．従来の研究の多くは，スレッド毎のメモリ・アクセスの性質に着目し，競合を起こしやすいスレッドに対し，リソースへのアクセスに何らかの制限を加える．一方我々は，ロード命令単位の挙動に着目し，より細粒度にリソースへのアクセスを制御する手法を提案する．具体的には，キャッシュに置いてもほとんどヒットしないラインをロードする命令に対し，キャッシュ・ラインをアロケートしないように制御する．提案手法を 2 スレッド間で共有される L2 キャッシュに適用し，評価を行った結果，IPC が最大で 24.2%，平均で 2.9%改善した．

Selective Cache Allocation: Efficient Cache Management in a Multi-threaded Environment

Yuhei Horibe,[†1] Shinobu Miwa,[†1] Ryota Shioya,[†2]
Masahiro Goshima[†2] and Hironori Nakajo[†1]

Recently, Multithreaded processors are commonly used. Sometimes, performance degrades when a conflict occured on shared resources. Especially, conflict on a cache memory is critical to its performance. Effective management of cache resources have been studied extensively over the years. Many of them, focussing on behavior of each thread. When a thread interferes with the other, its shared resource accesses are limited, allowing the other thread to use more resources. On the other hand, we focus on behavior of each instruction. When a load instruction fetches useless lines so many times, we never allocate all of the cache line fetched by this instruction into the cache. We evaluated our method applied to the L2 cache shared among 2 cores. The result shows that our method can improve performance up to 24.2%, and 2.9% on average.

1. はじめに

近年，スレッドレベル並列性を利用することでプロセッサ全体の処理能力を向上させる，**マルチスレッド・プロセッサ**が主流となっている．一つのチップ上に複数のコアを搭載した**CMP**（Chip Multi Processor）は，ハイエンド・プロセッサではもちろん，一部の組み込みプロセッサにおいても採用されている．また，ハイエンド・プロセッサの各コアは，最近では，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）プロセッサであることが多い．

マルチスレッド・プロセッサにおいては，複数のスレッドによって，プロセッサ内のリソースの一部が共有される．CMP では，2 ～ 8 個程度のコアが，最下位のキャッシュ・メモリやメモリ・バスを共有する．SMT プロセッサでは，PC やマップ表などフロントエンドに位置する一部のリソースを除いて，ほとんどのものが共有される．もちろん，L1 データ・キャッシュも共有されるリソースの一つである．

共有リソースにおいてはしばしば競合が発生するが，この競合がプロセッサ性能全体に与える影響は非常に大きい．あるスレッドが共有リソースを占有することで，同時に実行中の他のスレッドの実行を大幅に遅滞させてしまうことも珍しくない．スレッドの組み合わせによっては，スレッドを単独で実行した場合の半分程度の性能しかでないこともある[1)]．マルチスレッド環境において，リソース競合は深刻な問題となっている．

この問題を緩和するため，これまで数多くの研究が行われてきた．例えば，トラクション・コントロール実行[1)] は，CMP においてリソース競合によって性能が低下している場合，キャッシュ・ミス率が相対的に高く，メモリ・アクセス頻度の高いスレッドが稼働するコアの電源電圧と動作周波数を低下させる．競合を引き起こすスレッドの共有リソースへのアクセス頻度を減らすことで，競合の影響を緩和する．また，SMT プロセッサにおいても，共有リソースの使用状況を見て，特定のスレッドの共有リソースへのアクセスを制限する手法[2)]，あるいは，同時に実行するスレッド数そのものを制限する手法[3)] などが提案されている．

共有リソースの中でも特に，キャッシュにおける競合は性能低下の要因となりがちである．

†1 東京農工大学大学院 工学府
Graduate School of Engineering, Tokyo University of Agriculture and Technology
†2 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
†3 日本学術振興会 特別研究員
Research Fellow of the Japan Society for the Promotion of Science

そのため，共有キャッシュの問題に関しては，これまで集中的に研究が行われてきた．その代表例がキャッシュ・パーティショニング[4)–7)] である．キャッシュ・パーティショニングでは，スレッド数に応じて，CMP の共有キャッシュを論理的に分割する．分割は，通常，ウェイ方向に行われる．各スレッドの共有キャッシュへの書き込みは，それぞれに割り当てられたウェイに限定されるため，競合の影響をある程度緩和できる．スレッド毎に使用できるウェイの配分は，スレッドのふるまいを見て適宜決定する．

上述の手法はいずれも，スレッドのふるまい，あるいは，共有リソースの使用状況といった粗い粒度の情報を利用し，共有リソースへのアクセスをコントロールする．また，コントロールする対象は，スレッドという粗い粒度を単位とする．そのような意味において，これらの手法は**共有リソースの粗粒度な管理手法**と位置付けることができる．

ところで，共有キャッシュにとっては，そのヒット率を最大化する上で，それへのアクセスとスレッドとの関係を気にする必要はない．あるメモリ・アクセスが，競合を頻繁に引き起こすスレッドからのものだろうがそうでなかろうが，競合の影響を軽減する上でほとんど関係ない．スレッドに関係なく，単に，頻繁に参照されるラインをなるべくキャッシュに残し，そうでないラインをキャッシュに置かないようにしてもまったく問題ない．**すべてのアクセスをシングル・スレッドからのものと見なし，共有キャッシュを管理してよい**．

従来手法がスレッドを単位として共有リソースを管理していたのに対し，上述のアプローチは，個々の参照という細粒度な情報に着目し，ラインという細粒度な単位をコントロールする．そのような意味において，**共有リソースの細粒度な管理手法**と言える．

我々は，共有キャッシュの新たな管理手法として，**選択的キャッシュ・アロケーション**を提案する．提案手法では，キャッシュ・パーティショニングなどとは異なり，キャッシュへのアクセス制限をスレッド単位で行うことはしない．すべてのアクセスをシングル・スレッドからのものと見なし，個々のアクセス単位でキャッシュの利用を制限する．より具体的には，ほとんど参照されないラインをロードする命令については，キャッシュ・ミスした場合でも，そのラインをキャッシュにアロケートしないようにする．このように，個々のアクセスや参照されるラインなど，従来よりも細かい単位で共有キャッシュの管理を行う．

個々のメモリ・アクセス，あるいは，ライン単位でキャッシュの管理を行った例は過去にあるが，それらはすべて，シングルスレッド・プロセッサの L1 キャッシュを対象としたものである[8)–10)]．我々のように共有キャッシュを対象としたものはない．

本稿の構成は以下のとおりである．まず次章において，マルチスレッド環境におけるキャッシュの利用効率向上を狙った関連研究について述べる．続く 3 章では，我々の提案手法である，選択的キャッシュ・アロケーションについて詳しく述べる．評価は 4 章で行い，5 章でまとめる．

2. 関 連 研 究

マルチスレッド環境における，キャッシュ・メモリの利用効率向上を目指した代表的な手法として，**キャッシュ・パーティショニング[4)–7)]** が挙げられる．キャッシュ・パーティショニングは，キャッシュの容量を動的に分割し，それぞれのスレッドに対し適切な容量を配分する事で，競合の影響を緩和し，キャッシュ・ヒット率を改善する手法である．

一般に，LRU で動作するキャッシュにおいては，より頻繁にメモリ・アクセスをするアプリケーションに，より多くのキャッシュ・ラインがアロケートされやすい[5)]．このため，メモリ・インテンシブなスレッドと，そうでない他のスレッドが同時に動作した場合，メモリ・インテンシブなスレッドの方により多くのキャッシュ・ラインが割り当てられる．しかし，メモリ・インテンシブなスレッドが，局所性のほとんどないメモリ・アクセスをするスレッドであった場合，キャッシュ・ヒットしない無駄なラインに多くの容量を使われる事になり，キャッシュの利用効率が著しく低下してしまう．特にストリーミング・アプリケーションや，ワーキングセットがキャッシュに対して大きいアプリケーションでは，このような性質が現れやすい．

キャッシュ・パーティショニングは，このようなキャッシュを有効に活用できないスレッドが利用できる容量を制限し，よりキャッシュを有効に活用できるスレッドに多く配分する．すなわち，キャッシュ・ヒット率が，キャッシュ容量によりセンシティブなスレッドに，より多くの容量が割り当てられる．キャッシュ・パーティショニングによる容量の分割は，基本的には way 単位で行われる．

Suh らの手法[4)] では，あるスレッドに，それぞれの way 数を割り当てた場合のヒット数を，カウンタを用いて計測する．そして，way の分割の組み合わせの中で，キャッシュ・ヒットを最大化する組み合わせを求め，分割を決定する．

具体的には，スレッド毎に way 数分のカウンタを用意する．このカウンタのインデックスは，LRU における way 番号に対応している．そして，メモリ・アクセスがヒットする度に，ヒットした way 番号に対応するカウンタがインクリメントされる．このようにする事で，あるスレッドに，ある way 数を割り当てた場合のキャッシュ・ヒット数を計算する事ができる．そして，分割の組み合わせ毎にキャッシュ・ヒット数を見積り，ヒット数が最大となる組み合わせを決定する．

Dybdahl ら[7] の手法は，シャドウタグを用いる事で，各スレッドに割り当てる way 数を１増やした場合と，１減らした場合のキャッシュ・ヒット数を見積もり，最適な分割を決定する．シャドウタグとは，あるセットの中で，もっとも最近リプレースされたラインのタグだけを保持するバッファである．このシャドウタグをセット毎にスレッド数分用意する．すなわち，あるスレッドがアクセスしたラインの中で，一番最近リプレースされたもののタグをシャドウタグに保持しておく．そして，メモリ・アクセスが，あるスレッドのシャドウタグにヒットした回数をカウントする事で，割り当てる way 数を１増やした場合のヒット数の増加を見積もる．同様に，スレッド毎に，次のリプレース候補のラインがヒットした回数をカウントする事で，割り当てる way 数を１減らした場合に減少するヒット数を見積もる．以上のようにして，way の分割を変更した場合のヒット数の増減を見積り，ヒットを最大化するように分割を決定する．

一方，キャッシュ・パーティショニング以外のアプローチとして，小笠原らの L1/L2 キャッシュ・アクセス動的切り替え方式[2] が挙げられる．一般に，同時に動作しているスレッドの中で，キャッシュ・ミス率が相対的に高く，頻繁にメモリ・アクセスをしているスレッドが，他のスレッドのメモリ・アクセスを阻害しやすい[1] という性質がある．この手法では，同時に動作しているスレッドの中で相対的にキャッシュ・ミス率が高いスレッドに対し，そのスレッドがアクセスしたキャッシュ・ラインを最上位のキャッシュにアロケートしないよう制御を行う．

これらの手法はいずれも，**スレッド毎**のメモリ・アクセスの性質に着目し，キャッシュを有効に活用できないスレッドに対し，キャッシュ・リソースへのアクセスを制限する事で利用効率を向上させる．

3. 選択的キャッシュ・アロケーション

我々は，ロード命令単位のメモリ・アクセスの性質に着目し，細粒度にキャッシュ・ラインのアロケートを制御する事で，キャッシュの利用効率を向上する手法を提案する．

まず，提案手法のシステム全体の構成について説明する．

3.1 システム全体の構成

図 1 に，2 コアの CMP で，各コアが 2 スレッドを同時に実行する SMT であった場合の提案手法のシステムの構成を示す．なおこのモデルでは，コア毎に専有の L1 キャッシュと，コア間で共有される L2 キャッシュを持っている．従って，L1 キャッシュは 2 スレッド間で，L2 キャッシュは合計 4 スレッド間で共有される．

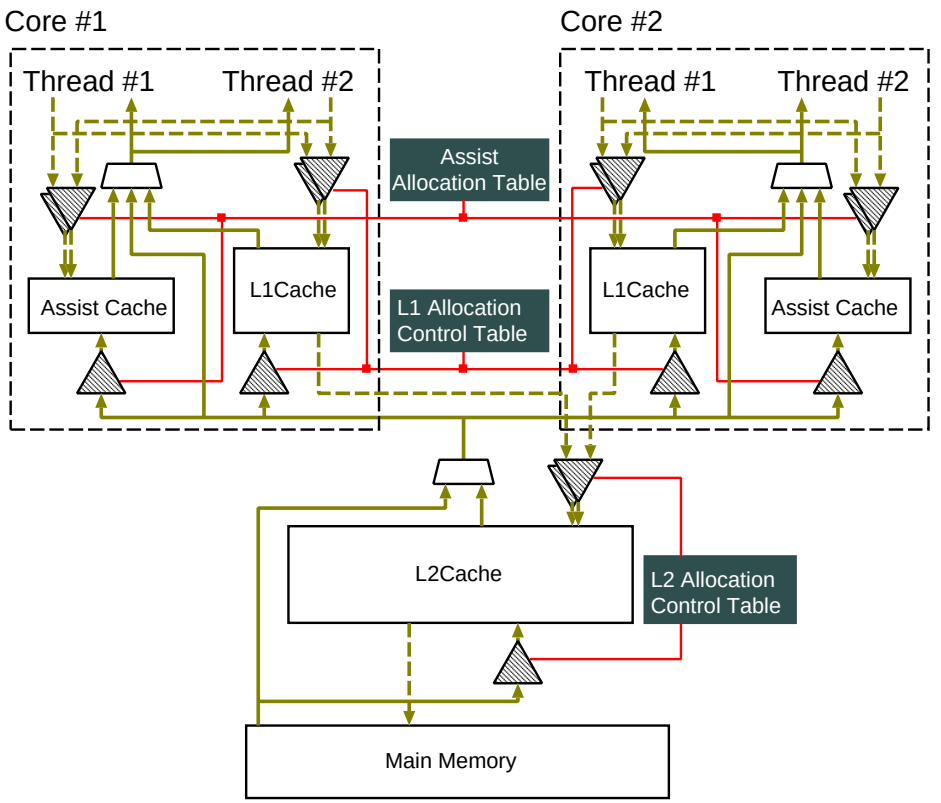

図 1 システムの構成

我々の手法では，階層毎にメモリ・アクセスを監視する．そして，それぞれの階層において，キャッシュ・ミスが発生する度に，ミスしたキャッシュ・ラインをそこに置くかどうか判断する．ミスしたキャッシュ・ラインをアロケートするかどうかの判断に用いる情報は，**Allocation Control Table(ACT)** に保持しておく．ACT は，数エントリ程度の CAM である．従来手法では，スレッド毎のメモリ・アクセスの性質を考慮するため，スレッド毎にパフォーマンスカウンタなどが必要であった．しかし，我々の手法はスレッドを区別せず，すべてシングルスレッドからのメモリ・アクセスとみなして動作をする．そのため，ACT はスレッド数やコア数に関係なく，各階層に一つあれば良い．

さらに各キャッシュ毎に，置かないと判断されたラインを書き込まないようにするための仕組みとして，下位のキャッシュからのライト・ポートにスリー・ステート・バッファを追加する．置かないと判断されたキャッシュ・ラインは，この階層をバイパスされ，直接上位の階層へ送られる．

また，**ごく短時間に同一ラインへのアクセスが集中し，それ以降二度と使われなくなるキャッシュ・ライン**も存在する．このようなラインは，一通りアクセスされた後二度と使われないままキャッシュの容量を浪費する．本来このようなラインはキャッシュに置くべきではない．ストリーミング・アクセスや，キャッシュ・サイズに対して大きなワーキングセットへのアクセスは，このようなキャッシュ・ラインを大量にロードする可能性がある．

そこで，L1 キャッシュと並行して **Assist Cache** を追加する事で，これに対処する．このようなメモリ・アクセスによってロードされたキャッシュ・ラインを Assist Cache に置くようにすると，このラインを L1 キャッシュに置かなくてよくなる．

また，Assist Cache に置くべきキャッシュ・ラインの情報を保持しておくためのテーブルとして，**Assist Allocation Table(AAT)** を用意する．

以上が提案手法の全体のハードウェア構成である．

3.2 キャッシュ・ラインのアロケート制御

キャッシュ・ラインをその階層にアロケートするかどうかの判断は，ロード命令毎のメモリ・アクセスの性質に着目し，行われる．具体的には，あるロード命令がアクセスするキャッシュ・ラインに再利用性があるかどうかを判断し，ないと判断された場合，以後そのロード命令がアクセスするキャッシュ・ラインをその階層に置かないようにする．

次節では，提案手法のより具体的なハードウェア構成について述べる．

3.3 ハードウェア構成

再利用性のないキャッシュ・ラインを判別するため，キャッシュの各エントリに以下の情報を追加する．

- そのラインがキャッシュに置かれてからヒットした回数をカウントするカウンタ
- そのラインをキャッシュにロードした命令の PC

そして，キャッシュ・ラインのリプレースが発生する度にラインのヒットカウンタの値を取り出し，あるしきい値と比較する．そして，カウンタの値がしきい値以下だった場合，そのラインをロードした命令のアドレスを取り出し，**Allocation Control Table(ACT)** に登録する．

ACT は，ロード命令の命令アドレスをインデックスとし，各エントリに 2bit のカウンタを持っている．このカウンタは，偶然ほとんど再利用されないラインをロードした命令をテーブルに登録する事を防ぐために利用される．

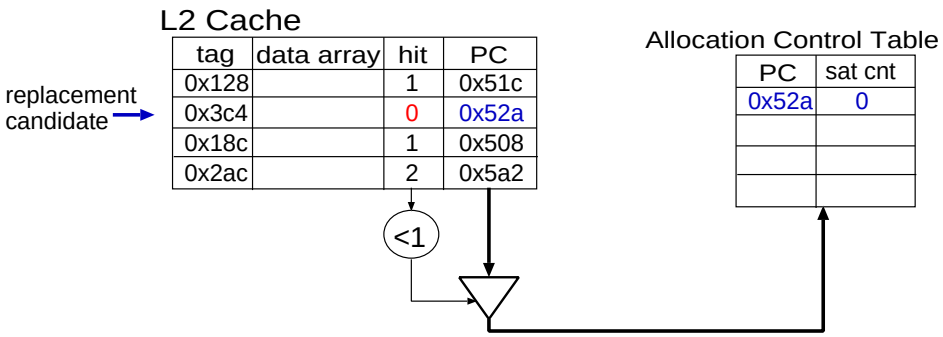

図 2 ACT の更新動作

提案手法の基本的な動作を，ACT の更新と読み出しに分けて，具体的な例を用いて説明する．

ACT の更新は，キャッシュ・ミスによってリプレースされたラインの情報を用いて行われる．リプレースされたキャッシュ・ラインのヒット・カウンタの値を参照し，もしカウンタの値が設定したしきい値以下だった場合，そのキャッシュ・ラインがほとんど再利用性のないキャッシュ・ラインだとみなされる．このしきい値を**登録のしきい値**と呼ぶ事にする．このしきい値を大きく設定するほど，再利用性がないと判断されるキャッシュ・ラインの数が増え，キャッシュに置かれなくなるキャッシュ・ラインの数が増える．

キャッシュ・ラインに再利用性がない，と判断された場合，それをロードした命令の命令アドレスを ACT に登録する．登録動作は，取り出したロード命令のアドレスで ACT を参照し，

ヒットした場合　そのエントリの確信度カウンタの値を 1 インクリメントする．

ミスした場合　その命令アドレスをテーブルに書き込み，確信度カウンタを 0 にセットする．

提案手法では，確信度カウンタに 2bit の飽和カウンタを用いる．そして，当該エントリのカウンタが飽和した場合，すなわち，ほとんど再利用されないラインを三つ以上，同じ命令がロードした場合，その命令がアクセスするキャッシュ・ラインをキャッシュにアロケートしないようにする．登録のしきい値を 1 とした場合の具体的な動作の例を，**図 2** に示す．

今，キャッシュ・ミスが発生し，0x3c4 のラインがリプレースされたとする．この時，そ

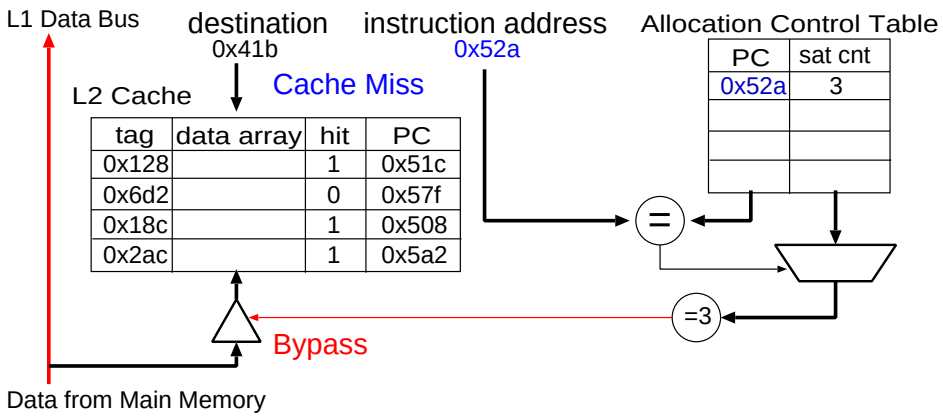

図 3 ACT の読み出し動作

のエントリからヒットカウンタの値を取り出し，しきい値と比較する．この場合，ヒットカウンタは 0 であり，しきい値 1 を下回る．そこで，このラインをロードした命令のアドレス，0x52a 番地を取り出し，ACT を参照する．この時，該当するエントリが ACT にないので，新しく 0x52a を登録し，そのエントリのカウンタを 0 にリセットする．もし該当するエントリがテーブルに存在した場合は，このカウンタの値を 1 インクリメントする．

このようにして，ロード命令の ACT への登録動作が行われる．

一方，読み出し動作は，キャッシュ・ミスしたロード命令の命令アドレスを用いて行われる．ミスしたロード命令の命令アドレスで ACT を参照し，

ヒットした場合 該当するエントリのカウンタの値を参照し，カウンタが飽和していた場合，キャッシュ・ミスしたラインをそのキャッシュにアロケートしない

ミスした場合 何もしない

ACT の参照動作の具体的例を**図 3** に示す．

今，0x41b へのアクセスがキャッシュ・ミスしたとする．この時，このメモリ・アクセスを生成したロード命令のアドレス，0x52a で ACT を参照する．この時，0x52a のエントリが ACT 上に存在し，カウンタの値が飽和しているため，この 0x41b というラインは L2 に置かれず，L1 キャッシュのデータ・バスへとバイパスされる．

以上のようにして，キャッシュ・ラインのアロケートを制御する．

なお，提案手法を最下位のキャッシュに適用する場合には，ACT の参照とアロケートの判断は，メイン・メモリへのアクセスが完了するまでに行われれば良い．従って，複数サイクルをかけて参照をする事ができるため，ACT をフルアソシアティブのキャッシュとして構成する事も可能である．

また，ACT のエントリのリプレースは，LRU によって行う．LRU のタイムスタンプの更新は，ACT への参照がヒットする毎に行う．このようにする事で，時間的局所性のないロード命令の中で，特にメモリ・インテンシブなものがテーブルに残りやすくなる．

しかし一方，偶然再利用性のないキャッシュ・ラインをロードしてしまった命令が間違って登録されてしまった場合，複数回参照される可能性のあるキャッシュ・ラインまでキャッシュに置かれなくなってしまい，パフォーマンスが低下してしまう恐れがある．それだけでなく，該当するロードが参照したラインがキャッシュに置かれなくなってしまうため，それを置かなくしたという判断が正しかったのかどうかを知る事も困難である．

一つの解決策は，命令がテーブルに登録されてから，キャッシュ・ラインを置かれなくなってしまうまでの間に，判断を間違って登録したエントリを，テーブルから削除する方法である．この一つの方法として，キャッシュ・ヒット時にエントリの削除を行う方法を説明する．キャッシュ・ヒット時に，アクセスされたラインからヒットカウンタの値を取り出し，これをあるしきい値と比較する．もし，カウンタの値が設定したしきい値を越えていた場合，そのラインを再利用性のあるキャッシュ・ラインであったとみなす．そして，そのラインをアクセスしたロードの命令アドレスで ACT を参照し，もし該当するエントリがあれば，これを削除する．この時に用いるしきい値を，**削除のしきい値**と呼ぶ事にする．この方法では，しきい値を小さく設定するほど，誤って登録されるロード命令の数は減る．しかし，あるロードがアクセスする大半のラインが再利用性のないラインで，一部が再利用されるラインであったとしても，そのロードのエントリがテーブルから削除されてしまう．このしきい値の設定については，評価の項で述べる．

もう一つの方法は，ある一定の時間間隔毎にテーブルの内容をすべて削除するという方式である．この場合も，テーブルをフラッシュする時間間隔を短くするほど，誤って登録されるロードは減る一方，キャッシュ・ラインを置かなくした事の効果が現れにくくなる．このフラッシュの時間間隔についても，評価の項で詳しく述べる．

4. 評　　価

提案手法を 2 コア間で共有される L2 キャッシュに適用し，キャッシュ・ヒット率，および IPC 改善率について評価を行った．評価では 2 コアの CMP を仮定し，コア毎に専有の

表 1 プロセッサの構成

parameter	remarks
way	8
L1-D Cache	16KB 4-way, 64B/line, 3cycle
L2 Cache	4MB 8-way, 64B/line, 15cycle
Main Memory	200cycle
ACT	32-128 Entry, Fully Associative

L1 キャッシュと，コア間で共有される L2 キャッシュを持つものとした．

4.1 評価環境

提案手法をアーキテクチャ・シミュレータ鬼斬弐[11] に実装し，評価を行った．評価に用いたプロセッサのパラメータを表 1 に示す．

評価には，SPEC CPU2006 のベンチマーク二本の組み合わせのうち，キャッシュ・ヒット率 70%を下回る組み合わせの中から，特にメモリ・インテンシブな組み合わせ 24 組と，メモリ・アクセスの少ない組み合わせ 8 組を選んだ．なお，先頭の 1G 命令をスキップし，続く 100M 命令を実行した時のキャッシュ・ヒット率，および IPC を評価した．

4.2 評価結果

まず予備評価として，ACT のサイズを無限大とし，次の各パラメータを変化させた場合のキャッシュ・ヒット率について評価を行った．

登録のしきい値 リプレースされたキャッシュ・ラインのヒットカウントの値が，この値未満だった場合，それをロードした命令が ACT に登録される

削除のしきい値 キャッシュ・ヒット時に，キャッシュ・ラインのヒットカウントの値がこの値以上だった場合，それをロードした命令のエントリが ACT から削除される

評価では，何も制御を行わない，通常の LRU で動作するキャッシュを BASE モデルとし，削除のしきい値を 1, 2, 4, 8 と変化させた場合の提案手法のモデルを DTx と表している．

図 4 に，登録のしきい値を 1 として，削除のしきい値を変化させた場合のキャッシュ・ヒット率を示す．

グラフの横軸はベンチマークの組み合わせを表しており，縦軸は L2 キャッシュのヒット率を表している．なお，一番右端に平均を示している．ベンチマークは，左から 24 本が特にメモリ・インテンシブな組み合わせで，右から 8 本がメモリ・インテンシブでないプログラムの組み合わせになっている．グラフの結果から，平均のヒット率を比較すると，提案手法のモデルのキャッシュ・ヒット率はほとんどのプログラムで，BASE モデルのヒット率を

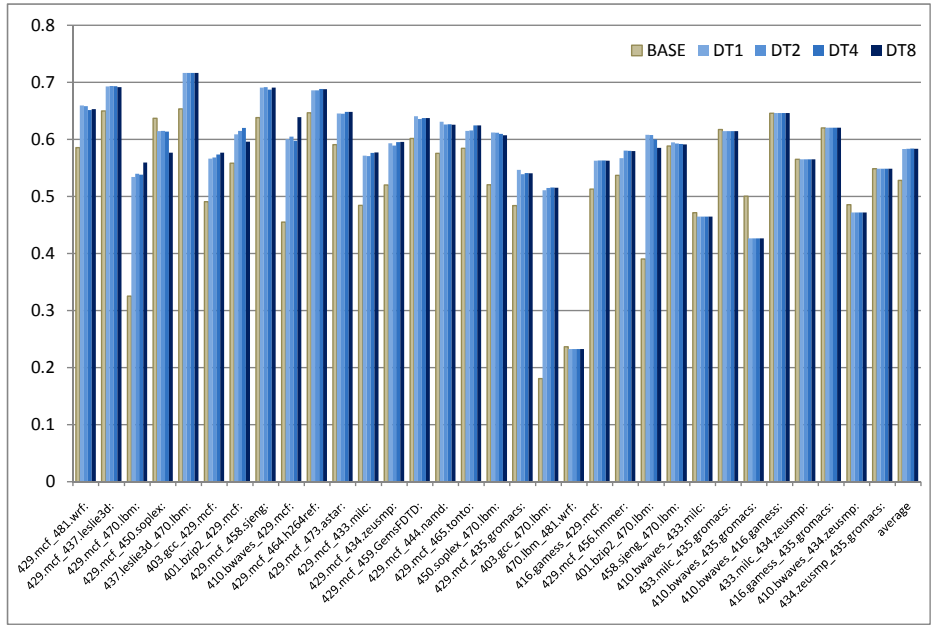

図 4 削除のしきい値を変化させた場合のキャッシュ・ヒット率

大きく上回っている．特に，メモリ・インテンシブなプログラムの組み合わせで，大きな改善が見られる．また，しきい値を大きくする事でキャッシュ・ヒット率が改善するプログラムと，かえって悪化するプログラムとが存在しており，平均ではしきい値の違いによるヒット率の変化はほとんど見られない．

続いて，削除のしきい値を 1 に固定し，登録のしきい値を変化させた場合のキャッシュ・ヒット率の評価結果を**図 5** に示す．先ほどと同様，何もしない通常の LRU で動作するキャッシュを BASE モデルとし，提案手法の登録のしきい値をそれぞれ 1, 2, 3, 4 と変化させたモデルを ETx として表している．

グラフの平均を見ると，1 回もヒットしなかったラインをロードした命令を ACT に登録する方式のキャッシュ・ヒット率がもっとも良い．以降，評価では登録のしきい値，削除のしきい値を 1 としたものを用いる事とする．

図 6 に，ACT のエントリ数をパラメータとした場合のキャッシュ・ヒット率の評価結果を

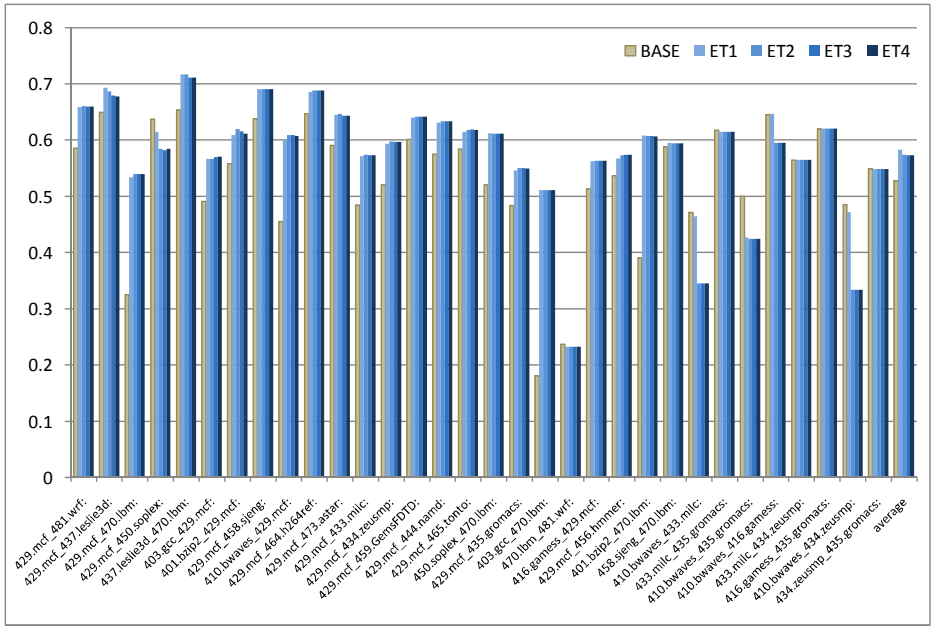

図 5 登録のしきい値を変化させた場合のキャッシュ・ヒット率

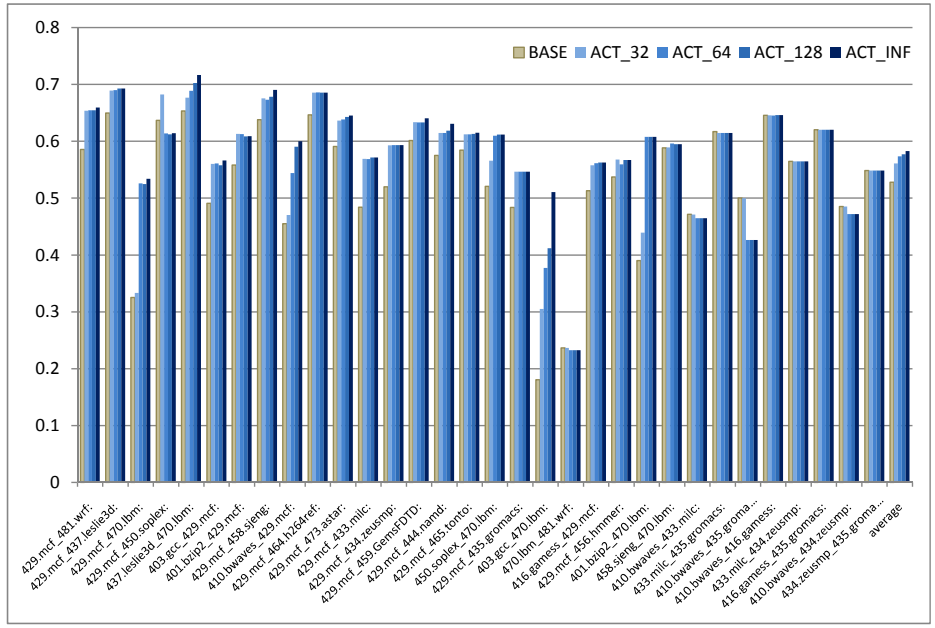

図 6 ACT のエントリ数を変化させた場合のキャッシュ・ヒット率

示す．なお，ACT のリプレースには LRU を用いる．グラフでは，何もしない通常のキャッシュを BASE モデルとし，提案手法の各エントリ数に対応するモデルを ACTx と表記している． 評価の結果から，ほとんどのプログラムでキャッシュ・ヒット率に大きな改善が見られる．エントリ数無限においては，gcc と lbm の組み合わせで最大 33%キャッシュ・ヒット率が改善した．さらに，平均では 5,5%キャッシュ・ヒット率が改善している．エントリ数有限のものでは，64 エントリ程度あれば，キャッシュ・ヒット率が十分に改善する事がわかる．なおエントリ数 64 において，bzip2 と lbm の組み合わせにおいて，最大 21.7%と大きくキャッシュ・ヒット率が改善している．平均では 4.6%キャッシュ・ヒット率が改善した．

図 7 に，エントリ数を変化させた時の IPC 改善率を示す．結果から，エントリ数無限の場合において，gcc と lbm の組み合わせにおいて，最大 47.8%，平均で 3.7%IPC が改善した．エントリ数 64 の場合でも，最大で 24.2%，平均で 2.9%IPC が改善した．

5. ま と め

本稿では，マルチスレッド環境におけるキャッシュの利用効率を向上する手法として，ロード命令単位のメモリ・アクセスの性質に着目し，局所性のないメモリ・アクセスをするロード命令に対し，キャッシュ・ラインをアロケートしないように制御する手法を提案した．

2 コアで共有される L2 キャッシュにおいて評価を行った結果，64 エントリのテーブルを用いるモデルで最大 21.7%，平均で 4.6%と大きくキャッシュ・ヒット率を改善した．なお，IPC の改善率は，最大 24.2%，平均 2.9%であった．

以上の結果から，複数のスレッドで共有される L2 キャッシュにおいて，ロード命令単位で細粒度にキャッシュ・ラインのアロケートの制御をする手法の有効性が示された．

今後の課題として，各コアが 2 スレッドの SMT で動作する 2 コアの CMP において，キャッシュ・ヒット率の評価を行う予定である．また，関連研究であるキャッシュ・パーティ

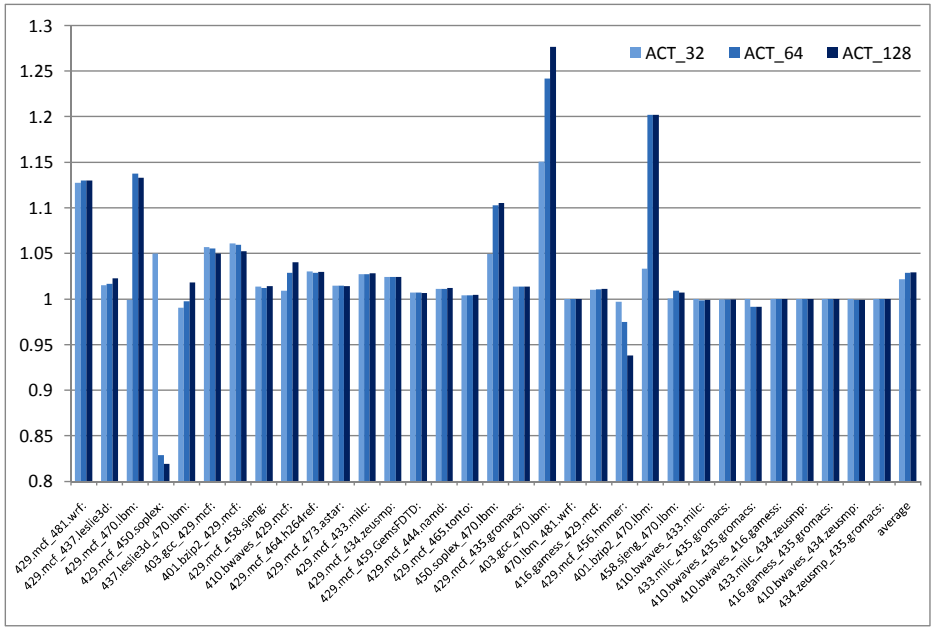

図 7 IPC 改善率

ショニングを実装し，比較を行っていく．

謝辞 本研究の一部は，文部科学省共生情報工学推進経費による．

参 考 文 献

1) 近藤正章，佐々木広， 中村宏：トラクションコントロール実行：CMP 向けプロセス実行制御方式の提案，情報処理学会論文誌コンピューティングシステム（ACS）， Vol.1, No.2, pp.111–123 (2008).
2) 小笠原嘉泰， 三輪忍，中條拓伯：SMT プロセッサにおける L1/L2 キャッシュアクセス動的切替え方式，情報処理学会論文誌コンピューティングシステム（ACS）， Vol.2, No.3, pp.12–25 (2009).
3) 内倉要，笹田耕一，佐藤未来子，加藤義人，大和仁典，中條拓伯，並木美太郎：SMT プロセッサにおけるスレッドスケジューラの開発 (スレッド・プロセス)，情報処理学会研究報告. [システムソフトウェアとオペレーティング・システム]， Vol.2004, No.63, pp.141–148 (2004-06-17).
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*, pp.117–128 (2002).
5) Qureshi, M.K. and Patt, Y.N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.423–432 (2006).
6) 小川周吾，入江英嗣， 平木敬：部分的試行に基づく動的共有キャッシュ分割方式，情報処理学会論文誌コンピューティングシステム（ACS）， Vol.2, No.3, pp.1–11 (2009).
7) Dybdahl, H., Stenstrom, P. and Natvig, L.: A Cache-Partition Aware Replacement Policy for Chip Multiprocessors, *In Proceedings of 13th International Conference of High Performance Computing (HiPC* (2006).
8) Gonzlez, A., Aliagas, C., Valero, M. and Nord, C.: A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality, pp.338–347 (1995).
9) Farrens, M., Tyson, G., Matthews, J. and Pleszkun, A.R.: A Modified Approach to Data Cache Management, *In Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp.93–103 (1995).
10) Rivers, J.A., Tam, E.S., Tyson, G.S. and Davidson, E.S.: Utilizing reuse information in data cache management, *Proceedings of the 1998 ICS*, ACM Press, pp. 449–456 (1998).
11) 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， Vol.2009, No.4, pp.120–121 (2009).