ロード/ストアの命令アドレスによる選択的キャッシュ・ライン・アロケーション

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

マルチスレッド実行環境では，キャッシュは複数スレッドによって共有されており，各スレッドからのアクセスが集中する．そのため，共有キャッシュでは多くの競合が発生し，プロセッサ全体の性能を低下させる要因となっている．共有キャッシュにおける競合を緩和する研究は従来から行われてきたが，それらはいずれも，スレッドの挙動に着目したものであった．すなわち，キャッシュを有効活用できないスレッドを検出し，それが利用するキャッシュの容量を制限することで，プロセッサ全体の性能を改善する．一方，我々は，個々のロード/ストア命令の挙動に着目する．すなわち，キャッシュを有効活用できない命令にキャッシュを利用させないようにすることで，プロセッサ全体の性能を改善する．具体的には，ほとんどヒットしないラインをアクセスする命令を検出し，そのような命令がキャッシュ・ミスした際に，ミスしたラインをキャッシュにアロケートしないようにする．マルチスレッド・プロセッサに本手法を適用した結果，性能が最大で 19.5% 向上することがわかった．

Selective Cache Line Allocation with Load/Store Instruction Address.

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

In multithreaded processors, caches are exploited by plural threads. Therefore, several cache lines conflict on the shared chaches, then, it causes that the performance degrade in the processors. To solve this problem, a lot of researches have been done, however, all of them focus on aspects of the threads. That is, they detect a thread which can not exploit shared caches efficiently, and they limit the thread to exploit the chaches. On the other hand, we focus on aspects of the instructions. That is, we detect an instruction which accesses an unuseful cache line, and we avoid the line being allocated to the caches when the instruction misses the caches. Above technique achieves that the performance improves 19.5% in maximum when it is applied to multithreaded processors.

1. はじめに

近年では，スレッド・レベル並列性を利用する事で高いスループットを達成する，**マルチスレッド・プロセッサ**が広く普及してきている．一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・プロセッサのみならず，一部の組み込みプロセッサにも採用されている．また，面積効率に優れるマルチスレッド実行形態として，一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)** をサポートするものも増えてきている．これらを組み合わせた構成もあり，一つのチップ上で 4 ～ 12 のスレッドが動作する事もある．

このようなプロセッサは，普通，スレッド間でキャッシュを共有する．SMT では，L1 キャッシュを始めとする，すべてのキャッシュが共有される．一方，CMP では，容量の大きな下位のキャッシュのみが共有される．このように，近年のプロセッサでは，下位のキャッシュ程，多くのスレッドからのアクセスが集中する．

共有キャッシュではしばしば**競合**が発生する．特にキャッシュにおけるスレッド間の競合は，プロセッサ全体の性能に与える影響が非常に大きい．スレッドの組み合わせによっては，それ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)]．

† 東京農工大学大学院 工学府
Graduate School of Engineering, Tokyo University of Agriculture and Technology
†† 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
††† 日本学術振興会 特別研究員
Research Fellow of the Japan Society for the Promotion of Science

近年では，チップに搭載されるキャッシュの容量は年々増加しているが，一つのチップ上で動作するスレッドの数も増えてきている．そのため，共有キャッシュに対するマネジメントの重要性がさらに高まっている．

共有キャッシュに対しマネジメントを行う代表的な手法として，**キャッシュ・パーティショニング**[2)~5)] が挙げられる．スレッドには，より多くの容量のキャッシュを利用することで，性能が大幅に向上するものがある．その一方で，単に容量を浪費するだけのスレッドもある．これらのスレッドが同時に実行された場合，共有キャッシュにおいては，後者のスレッドよりも前者のスレッドにより多くの容量を利用させた方がよい．キャッシュ・パーティショニングは，後者のスレッドが利用できるキャッシュ容量を制限することで，プロセッサ全体のスループットを改善する．

このように，共有キャッシュに対するマネジメントは，これまでは，スレッド間の利害関係に着目し，研究が行われてきた．すなわち，どちらのスレッドにより多くの共有リソースを利用させた方が効果的か，という観点で研究が進められてきた．

一方，我々は命令間の利害関係に着目する．限られたリソースを誰に利用させるのが最も効果的かという問題は，スレッド間だけでなく命令間でも起こる．あるスレッド 1 つに着目した場合でも，スレッド内の命令の中には，繰り返しヒットするラインばかりをアクセスしている命令や，ほとんどヒットしない無駄なラインばかりをアクセスする命令が存在する．後者の命令がアクセスするラインよりも，前者の命令がアクセスするラインをキャッシュに置いた方がよい．

本稿では，個々のロード/ストア命令の挙動に着目し，キャッシュ・ラインのアロケーションを制御する手法を提案する．ほとんどヒットしないラインをアクセスする命令を検出し，そのような命令がキャッシュ・ミスした際は，ミスしたラインをアロケートしないようにする．このように，命令ごとにキャッシュの利用を制限することで，競合の影響をより効果的に緩和することを目指す．

本稿は以下の構成となっている．まず次章では，共有キャッシュに対する従来のマネジメント手法の代表例として，キャッシュ・パーティショニングを詳しく取り上げる．また，シングル・スレッド・プロセッサの L1 キャッシュにおいては，個々のロード/ストア命令の性質に着目してマネジメントを行った例がある．それについても詳しく述べる．3 章では提案手法を詳しく述べる．手法の評価は 4 章で行い，5 章でまとめる．

2. 関 連 研 究

2.1 マルチスレッド・プロセッサ向けのキャッシュ・マネジメント

マルチスレッド・プロセッサを対象とした，キャッシュ・マネジメントの代表的な手法にキャッシュ・パーティショニング[2)~6)] が挙げられる．プログラムによって，そのメモリ・アクセスの特性は様々であり，より多くの容量を割り当てる事で，大きな性能改善を期待できるプログラムもあれば，ほとんどヒットしないラインばかりをアクセスし，容量を浪費するプログラムも存在する．キャッシュ・パーティショニングは，動的にキャッシュを分割し，後者のようにキャッシュの容量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する一方で，前者のような特性を持つスレッドが，より多くの容量を利用できるようにする．このようにすると，よりキャッシュの容量を必要とするスレッドがキャッシュを利用しやすくなり，スループットが向上する．

キャッシュ・パーティショニングの代表的な手法に，Utility Based Cache Partitioning(UCP)[2)] が挙げられる．UCP では，複数のタグ・アレイ，および多数のカウンタを用いて，あるスレッドのパフォーマンスが，キャッシュ容量にどれくらい依存するのかを計測する．そして，より，キャッシュの容量がパフォーマンスにクリティカルなスレッドに対し，多くの容量を割り当てる一方，パフォーマンスがキャッシュの容量にほとんど依存しないスレッドが利用できる容量を制限する．この手法は大きくスループットを改善できる半面，ハードウェア・オーバヘッドが大きいという問題点がある．

Dybdahl らは，比較的単純なハードウェアで効果的なキャッシュ・パーティショニングを実現する手法[3)] を提案している．この手法では，タグ・アレイの代わりにシャドウ・タグを利用し，あるスレッドが利用できる way を，現状より 1 増やした場合に増えるヒット数，および 1 減らした場合に増えるミス数だけを計測する．このようにする事で，比較的簡単なハードウェアで，効果的なパーティショニングを実現できる．

また小川らは，カウンタを使わずに容量の配分を動的に変更する手法[5)] を提案している．この手法は，キャッシュの容量すべてを各スレッドに配分してしまうのではなく，どのスレッドからも利用できる共有領域を一部残しておく．そして，この共有領域上のラインがヒットした場合に，ヒットしたスレッドへの容量の配分を 1way 増やす．このようにする事で，カウンタを使わずに，動的に容量の配分を変更する．

2.2 シングルスレッド・プロセッサ向けのキャッシュ・マネジメント

個々のロード/ストア命令の性質に着目し，キャッシュの最適化を行う関連研究として，Dual Data Cache が挙げられる．Dual Data Cache は，時間的局所性をターゲットとした時間的局所性用のキャッシュと，空間的局所性をターゲットとした空間的局所性用のキャッシュとの二つを用い，キャッシュの利用効率を向上する手法である．具体的には，命令毎のメモリ・アクセ

スをモニタリングし，その局所性に適したキャッシュにのみラインをアロケートする．

Rivers らは，ある命令のアクセスするラインが，主に時間的局所性によってヒットしやすいラインなのか，空間的局所性によってヒットしやすいラインなのかをモニタリングする事で，ラインを適切なキャッシュにアロケートする[7]．具体的には，個々のデータに対し，それがヒットしたかどうかを表す 1bit のフラグを追加する．もし，まったく同一のデータが複数回参照された場合，そのデータを含むラインは時間的局所性によってヒットしやすいラインであると判断される．また，同一ライン中の複数のデータのヒットのフラグがセットされた場合，そのラインは空間的局所性によってヒットしやすいラインであると判断される．

Tyson らは，個々のロード/ストア命令のヒット率に着目し，無駄なラインのアロケートを省略する手法[8]を提案している．一般に，プログラム中のキャッシュ・ミスの大半が，ごく少数の静的な命令によって引き起こされている事が知られている[9]．このように頻繁にキャッシュ・ミスをする命令は，ほとんど時間的局所性のないメモリ・アクセスをしていると考えられ，多くの容量を浪費する主な要因となっている事が考えられる．そこで，頻繁にキャッシュ・ミスを引き起こす命令に対して，ラインをキャッシュにアロケートさせないようにする事で，容量の浪費を防ぎ，キャッシュを有効活用する．

このように，多くの Dual Data Cache の手法は個々のロード/ストア命令の**参照の局所性**に着目し，L1 キャッシュを対象として制御を行う．一方我々は，ある命令がアクセスするラインの**再利用性**に着目する．すなわち，ほとんどヒットしないラインばかりをフェッチしてくる命令を検出し，そのような命令にはラインをキャッシュにアロケートしないように制御を行う．

3. 選択的キャッシュ・ライン・アロケーション

本章では，個々のロード/ストア命令の性質に着目し，ラインを適切な階層にアロケートする事で容量を有効活用する手法を提案する．

3.1 提案の概要

一つのプログラム中には，様々なメモリ・アクセスをするロード/ストア命令が存在している．例えば，ループ変数の参照のように，一つのラインを繰り返し参照する命令もある．その一方で，配列に対するストライド・アクセスのように，ほとんどヒットしない無駄なラインばかりをアクセスする命令も存在する．

無駄なラインを多くアクセスしている命令が頻繁に実行されると，無駄なラインによってキャッシュが埋めつくされてしまう．それによって，まだヒットするはずのラインが，無駄なラインの代わりにリプレースされてしまう．これは，キャッシュの容量を有効に活用できているとは言えない．

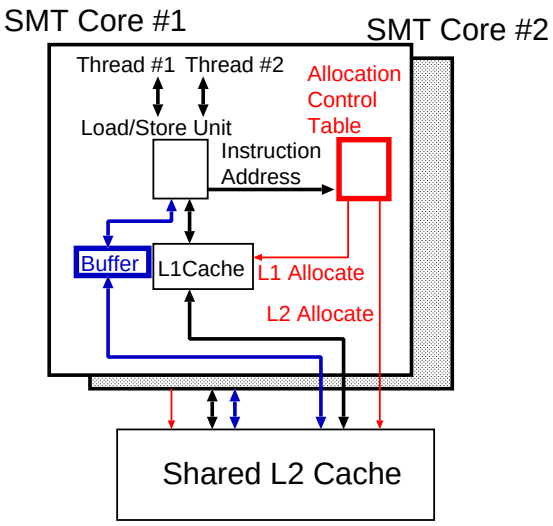

図 1 システムの構成

そこで，キャッシュの各階層において，このような命令を検出する．そして，このような命令がアクセスするラインをその階層にアロケートしないようにすることで，容量を有効活用する．

次節から，提案手法の詳細について述べる．

3.2 ハードウェア構成

提案手法では，次のようなラインを無駄にアロケートされたラインであるとみなす．

- ほとんどヒットしないライン．
- ごく短期間のうちにしか参照されないライン．

提案手法では，ある階層に対して，無駄なラインを一定数以上アロケートした命令を，その階層のキャッシュを有効活用できない命令であると判断する．そして，以後その命令がアクセスするラインをその階層にアロケートしない．これを実現するためのシステム構成は，**図 1** のようになる．なおこの図は 2 コアの CMP で，各コアが 2 スレッド動作の SMT プロセッサである場合を想定している．

提案手法における主な追加ハードウェアには，以下のものがある．

Allocation Control Table(ACT)
: キャッシュを有効活用できない命令を記録するためのテーブル．ロード/ストアの命令アドレスをタグとし，数 bit 程度のカウンタをエントリとする．このカウンタにより，その命令がアロケートした無駄なラインの本数をカウントする．なお，ACT は各階層毎に用意する．

バッファ
: L1 と並列に参照される小容量のバッファ．短期間のうちにしか参照されないラインを，L1 キャッシュの代わりに保持する．

また，無駄なラインを検出するため，キャッシュ・ラインに以下のフィールドを追加する．

ヒット数カウンタ
そのラインが有用かそうでないかを区別するため，そのラインにヒットした数を数えるカウンタを追加する．このカウンタは 3 bit 程度の飽和カウンタとして実装される．ラインがキャッシュにアロケートされた際，0 に初期化される．カウンタの値があるしきい値を超えた場合，そのラインは有用と見なす．

ラインをフェッチした命令のアドレス
ラインがリプレースされる際，それが無駄なものだった場合は，このアドレスを ACT に登録する．

なお，提案手法をマルチコア SMT に適用する場合は，ACT はコア毎に用意する．各コア内で動作するスレッド間では ACT を共有する．

3.3 動　　作

提案手法の動作を，学習と読み出しについてそれぞれ説明する．

3.3.1 ACT の学習

ACT の学習は，基本的には各階層からラインがリプレースされる度に行われる．まず，リプレースされたラインのヒット数のカウンタを参照する．もし，カウンタが飽和していなければ，そのラインはほとんどアクセスされない無駄なラインであるとみなされる．この時，そのラインをキャッシュにアロケートした命令のアドレスが，新しく ACT に登録される．このようにして，無駄なラインをアクセスしやすい命令が，順次 ACT に登録されていく．なお，ACT のエントリのリプレースは LRU などにしたがって行う．

ACT には，有用なラインをアクセスする命令が誤って登録され，学習が誤って進んでしまうこともある．すると，たとえ有用なラインであっても，ACT は，このような命令がキャッシュ・ミスした際にそれをアロケートしない，と判断してしまう．これを防ぐため，ACT のカウンタを，以下の 2 つのタイミングでリセットする．

(1)　有用なラインがリプレースされた時
(2)　有用なラインにヒットした時

これらのタイミングで，ラインに保持しているフェッチした命令のアドレスを用いて ACT を参照し，ACT にヒットした場合はカウンタをリセットする．さらに，プログラムの実行フェーズによって，メモリ・アクセスの特性が変わる事を考慮し，ACT の内容は定期的にフラッシュする．

3.3.2 ACT の読み出し動作

ACT の読み出しは，基本的にはキャッシュ・ミスが発生した場合に行われる．

ある階層でキャッシュ・ミスが発生した場合，ミスしたロード/ストアの命令アドレスで，その階層に対応する ACT を参照する．もし，該当するエントリが存在し，かつカウンタが飽和していた場合，新しいラインをその階層にアロケートしない．それ意外の場合，すなわち ACT にミスした場合と，該当するカウンタがまだ飽和していない場合，ラインは通常と同様にその階層にアロケートされる．

```
LOOP:
0x100:  load  $1, 500($2)  # a[i]
0x104:  add   $3, $3, $1   # sum += a[i]
0x108:  addi  $2, $2, 1    # i++
0x10C:  cmp   $2, ARRAY_SIZE
0x110:  blt   LOOP
```

図 2　配列アクセスの擬似アセンブラコード

ACT の読み出しにはある程度のレイテンシを要するが，これは次のようにして隠蔽することができる．ロード/ストアが実行された際，キャッシュのヒット/ミスに関わらず，ACT を参照する．そうすれば，それぞれの階層でキャッシュ・ミスが判明するまでのサイクル数を，ACT の読み出しに費すことができる．次章で述べる評価では，このような工夫を行ったものとして，ACT のレイテンシを評価項目から除いてある．

以上が提案手法の基本的な動作の流れである．

3.3.3 動　作　例

シングルスレッドが実行される場合を例に，提案手法の動作を説明する．

図 2 に示すプログラムが実行されるものとする．このプログラムは，整数型の配列 a の，全要素を足し合わせるプログラムである．なお，配列のサイズは L2 キャッシュのサイズよりも大きいものとする． このコードが実行されていく様子を説明する．なお，ラインのヒット数を数えるカウンタは 1bit とし，ACT のカウンタも，1bit であるものとして説明する．

まず，0x100 番地のロード命令が実行され，a[0] へのアクセスがすべての階層にミスする．この時，ACT の読み出しが行われる．ACT は初期状態でまだ何も登録されていないため，ラインはすべての階層にアロケートされる．この時，ミスしたラインをアクセスした命令のアドレス 0x100 と，ヒット数の初期値 0 がラインに書き込まれる．

続いて a[1] がアクセスされると，ラインはバッファと L1 両方にヒットする．バッファにヒットした場合，そのラインはごく短期間のうちに繰り返し参照されたものと考えられる．ごく短期間のうちにしか参照されないラインは，キャッシュにアロケートすべきでないラインと考えるので，この場合は，L1 キャッシュの該当するラインのヒット数をインクリメントしない．

実行が進み，ラインのリプレースが発生すると，ACT の更新が行われる．**図 3** に，a[4] がアクセスされ，a[0] がリプレースされた時の状態を示す． ACT が更新される手順は，次のようになる．(1)a[4] へのアクセスがすべての階層にミスする．(2)ACT にはまだ何も登録されていないため，すべての階層にラインが

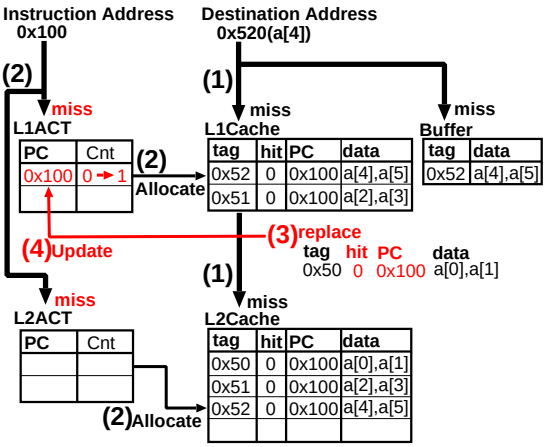

図 3 L1 からラインがリプレースされた時の動作

アロケートされる．(3) この時，L1 キャッシュで a[0] と a[1] を含むラインのリプレースが発生する．ラインのヒット数は 0 であるため，これは無駄にアロケートされたラインとみなされる．(4) この時，ラインをフェッチした命令のアドレス，0x100 を ACT に登録し，L1 のカウンタをインクリメントする．このようにして，0x100 の命令が ACT に登録され，エントリが有効になる．

ACT に命令が登録された状態で，キャッシュ・ミスが発生した時の状態を**図 4** に示す．この時の動作は次のようになる．(1)a[6] へのアクセスはすべての階層にミスする．(2) 実行されたロード命令のアドレス 0x100 で ACT を参照するとヒットし，L1 のカウンタが飽和しているため，ラインは L1 にアロケートされない．

またこの時，L2 で a[0]，a[1] を含むラインのリプレースが発生し，ACT の更新が行われる．先程と同様，このラインのヒット数は 0 であるため，このラインをキャッシュにアロケートした命令のアドレス，0x100 が，L2 に対応する ACT に登録され，カウンタが 1 にインクリメントされる (3)．以後，0x100 の命令がキャッシュ・ミスした場合，ラインは L1 にも L2 にもアロケートされなくなる．

以上のようにして，提案手法ではラインのアロケートを制御する．

3.4 誤登録への対処

プログラムが扱うデータ構造によっては，同じ命令がアクセスするデータでも，参照回数に大きく偏りが出る場合がある．

例えばスタックへのアクセスは，頂点に集中しやすい．そのため，次々に新しいデータがアクセスされていくと，底に近いデータは，ほとんどヒットせずにリプレースされてしまう事もある．しかし，スタックは時間的局所性を活用しやすいデータ構造であるため，キャッシュにラインをアロケートすれば繰り返し参照

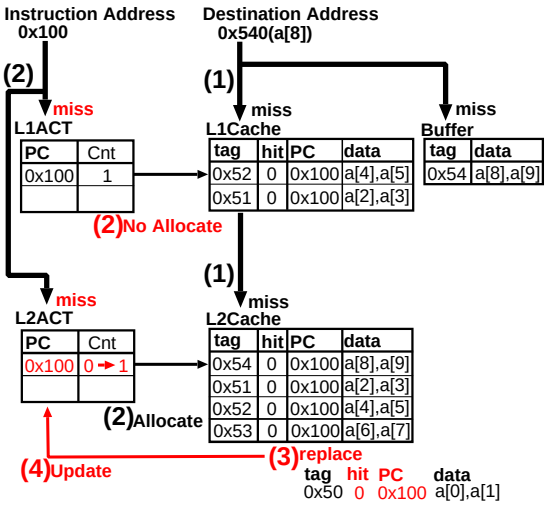

図 4 L2 からラインがリプレースされた時の動作

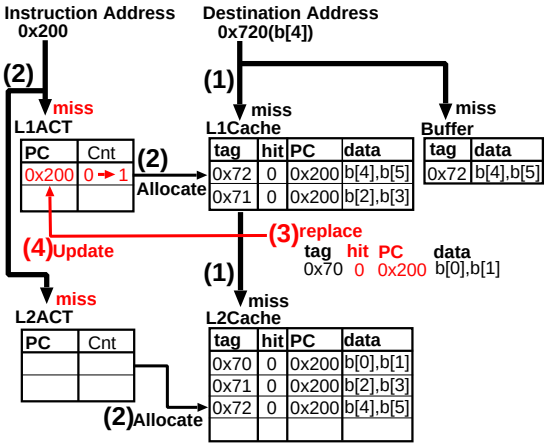

図 5 b[0] が L1 からリプレースされた時の動作

される．もし，このような命令が ACT に登録されてしまった場合に，そのエントリを無効化する動作について説明する．

今，整数型の配列 b がスタックとして利用されているものとする．b[0] から順にデータがスタックに積まれていき，今，b[4] のデータがプッシュされたとする．この時の状態を**図 5** に示す．まず，b[4] へのライトアクセスはすべての階層にミスする (1)．この時はまだ ACT に何も登録されていないため，b[4]，b[5] のラインはキャッシュにアロケートされ，b[0]，b[1] のラインがリプレースされる (2)．リプレースされたラインは一度もヒットしないままリプレースされたため，このラインをアクセスしたストア命令のアドレス，0x200 が ACT に書き込まれ，L1 に対応するカウンタが 1 にインクリメントされる (3)．

この時，b[4] がスタックからポップされ，続けて b[3] のデータがポップされた時の状態を**図 6** に示す．b[3]

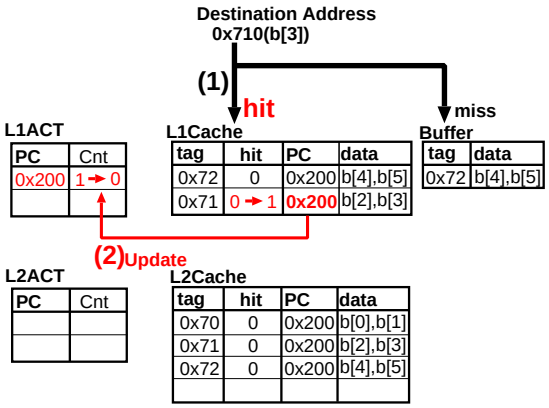

図 6 b[3] がキャッシュ・ヒットした時の動作

へのアクセスはバッファにミスし，L1 キャッシュ上でヒットする (1)．この時 L1 キャッシュ上の該当するラインのヒット数がインクリメントされ，しきい値以上になるため，有用なラインであるとみなされる．そして，このラインをアロケートした命令のアドレス 0x200 で L1 に対応する ACT がアクセスされ，該当するエントリのカウンタが 0 にリセットされる．

以上のようにして誤登録された命令のエントリが無効化される．また，プログラムのフェーズによってラインのアクセスのされ方が変わる可能性があるため，メモリ・アクセスがある一定数に達する毎に，ACT の内容をすべて消去する．

以上が提案手法の動作である．次章から評価について詳しく述べる．

4. 評　　価

提案手法をプロセッサ・シミュレータ，鬼斬弐[10] 上に実装し，以下のモデルについて評価を行った．命令セットには，Alpha 命令セットを用い，測定には SPEC2006 の INT，FP から合わせて 28 本のベンチマークを使用した．

- **BASE** 何も制御を行わない，通常のキャッシュを用いたプロセッサ．
- **SC** 提案手法を適用し，ラインのアロケートの制御を行うキャッシュを用いたプロセッサ．

SC モデルのプロセッサでは，次の各パラメータを階層毎に設定する．

- ラインの有用/無用を判断するための，ヒット数のしきい値
- ACT のカウンタのビット数
- ACT をフラッシュするインターバル

なお，テーブルのフラッシュは，全メモリ・アクセスのカウントが，ある一定数に達する毎に行う．

本評価では，BASE モデルに対する SC モデルの性能改善率を評価した．評価に用いたプロセッサのパラメータを**表 1** に示す．なお，CMP におけるプロセッサの構成では，L1 キャッシュ，および ACT はコア毎に用意され，L2 キャッシュはコア間で共有される．SMT においては，L1 キャッシュ，L2 キャッシュ，ACT はすべてスレッド間で共有される．

表 1 評価に用いたプロセッサのパラメータ

parameter	remarks
simulator	鬼斬弐
fetch width	4
issue width	4
instruction queue	int 32, fp 16, mem 16
execution unit	int ALU 2, fpALU 1, LD/ST 2
ACT	64 Entry, 8-way
Buffer	1KB, 64B/line, 16-entry, 3cycle
L1 I-cache	32KB, 64B/line, 4-way, 1 cycle
L1 D-cache	32KB, 64B/line, 4-way, 3 cycle
L2-cache	4MB, 64B/line,8-way, 15cycle
memory	200cycle

表 2 評価に使用するベンチマーク

競合を起こしやすい	429.mcf, 470.lbm 433.milc, 459.GemsFDTD 410.bwaves, 437.leslie3d
競合に影響されやすい	483.xalancbmk, 447.dealII 401.bzip2, 456.hmmer 471.omnetpp

表 3 CMP に適用する場合の SC モデルの各パラメータ

parameter	remarks
ヒットのしきい値	L1:1 L2:1, 2, 4
ACT のカウンタのビット数	L1:5bit L2:1bit
ACT のフラッシュ	L1:なし L2:1k〜500k アクセス毎

以下，それぞれのプロセッサ構成における評価結果について述べる．

4.1 CMP における評価結果

CMP の構成において，BASE モデルに対する性能改善率の評価を行った．評価に使用する 2 スレッドは，キャッシュ・パーティショニングの論文[5] で用いられた，競合を起こしやすいスレッド 6 種類と，競合によって性能低下しやすいスレッド 5 種類との総当たりの組み合わせを実行した．**表 2** に，実行したベンチマークを示す．

SC モデルは，**表 4** に示す各パラメータについて評価を行った．なお，L2 キャッシュは 4MB とした．まず，SC モデルにおける適切なパラメータを求めるため，各パラメータを設定した場合の平均の IPC 改善率を測定した．評価の結果を**図 7** に示す．

グラフの横軸は，ACT をフラッシュするインターバルに対応しており，横に並んだ棒グラフは，左から順に，ヒットのしきい値を 1 にした場合，2 にした場合，4 にした場合にそれぞれ対応している．

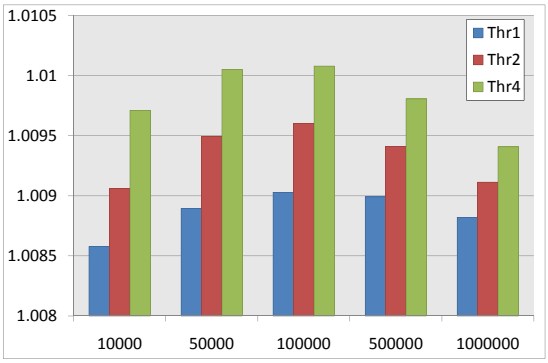

図 7 CMP における平均 IPC 改善率

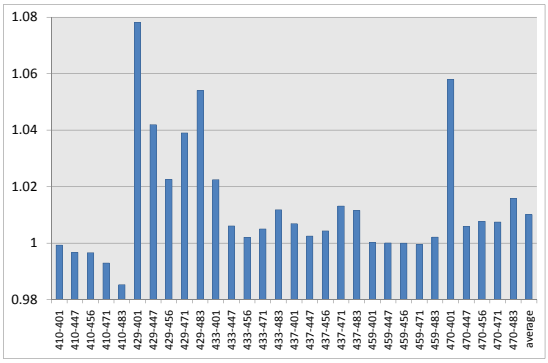

図 8 各ベンチマークの組み合わせにおける性能改善率

評価の結果から，ヒットのしきい値は 4 程度，フラッシュのインターバルは 10k アクセス程度が適切である事がわかる．

続いて，各 2 スレッドの組み合わせにおける，性能改善を評価した結果を**図??**に示す．

評価の結果から，特に 429.mcf と他のベンチマークの組み合わせにおいて提案手法の効果が高い．最大では，429.mcf と 401.bzip2 との組み合わせにおいて 7.8 % IPC が改善した．429.mcf はメモリ・インテンシブで，巨大なデータ構造をアクセスするプログラムである．そのため，無駄なラインを多くアクセスしている事が考えられ，これらをアロケートしないようにする事で大きく性能が改善したと考えられる．なお，平均では 0.95 % IPC が改善した．

一方，410.bwaves との組み合わせは，わずかながら性能低下が見られる．これは，カウンタの bit 数を 1 としたため，一度でも無駄なラインをアクセスしてしまった命令が誤って登録され，ヒットによるエントリの無効化が行われるよりも前に，いくつかの有用なラインがアロケートされなくなった結果，性能低下を引き起こしたものと考えられる．

4.2 SMT における評価結果

SMT プロセッサに提案手法を適用した場合の評価

表 4 SMT に適用する場合の SC モデルの各パラメータ

parameter	remarks
ヒットのしきい値	L1:1 L2:1, 2, 4
ACT のカウンタのビット数	L1:2〜7bit L2:1bit
ACT のフラッシュ	L1:100K〜10M アクセス毎 L2:10k〜1M アクセス毎

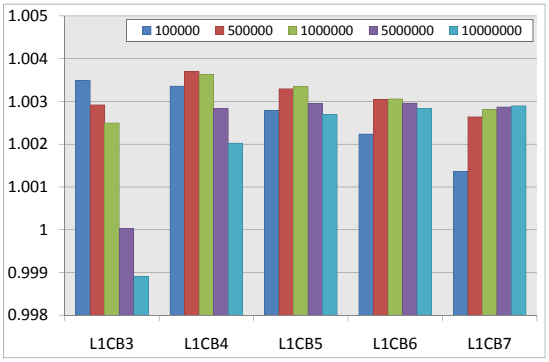

図 9 L1 のみに提案手法を適用した場合の平均 IPC 改善率

結果を述べる．実行するベンチマークの組み合わせは，CMP に適用した場合と同じ組み合わせを使用した．SC モデルにおいては，以下の各パラメータについて評価を行った．

まず最適なパラメータを求めるため，L1 キャッシュにのみ提案手法を適用した場合の，各パラメータにおける平均 IPC 改善率を評価した．評価の結果を**図 9** に示す．

グラフの横軸は ACT のカウンタの bit 数に対応しており，五つ並んだ棒グラフは，左から順に，ACT をフラッシュするインターバルをそれぞれ 100k アクセス毎，500k アクセス毎，1M アクセス毎，5M アクセス毎，10M アクセス毎とした場合に対応している．縦軸はそれぞれのパラメータにおける平均 IPC 改善率を表している．評価の結果から，ACT のカウンタのビット数は 4，ACT をフラッシュするインターバルは 500k アクセス毎が適切である事がわかる．

続いて，すべての階層に提案手法を適用した場合の，各パラメータにおける平均 IPC 改善率について評価を行った．L1 キャッシュに適用するパラメータは，ACT のカウンタを 4bit，ACT をフラッシュするインターバルは 500k アクセス毎とした．評価の結果を**図 10** に示す．

グラフの縦軸は，各パラメータにおける IPC 改善率を表している．横軸は，ACT のフラッシュのインターバルに対応しており，三つ並んだ棒グラフは，左から順に L2 におけるヒットのしきい値を 1 とした場合，2 とした場合，4 とした場合にそれぞれ対応する．評価の結果から，SMT の構成で，L2 に対し提案手法

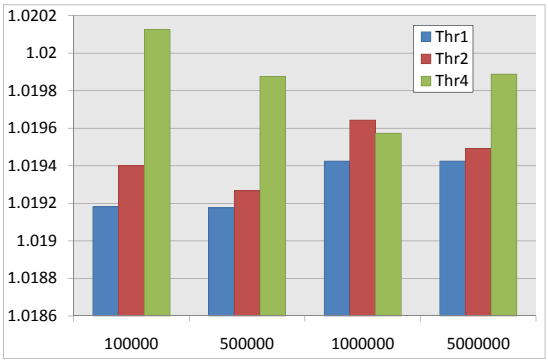

図 10 SMT における平均 IPC 改善率

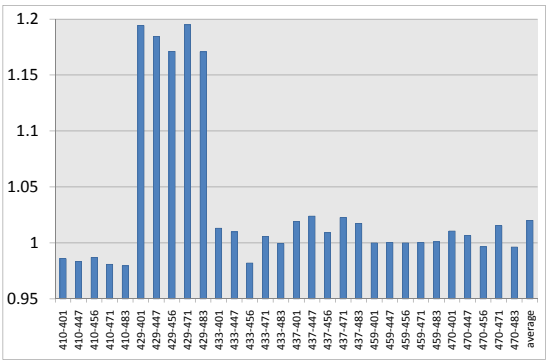

図 11 各プログラムの組み合わせにおける IPC 改善率

を適用する場合，ヒットのしきい値は 4 程度，ACT をフラッシュするインターバルは 100k アクセス程度が適当である事がわかる．

最後に，各プログラムの組み合わせにおける IPC 改善率を**図 11** に示す． 評価の結果から，特に 429.mcf と他のプログラムを組み合わせた場合に，著しい性能改善が見られ，最大では 19.5 %性能が改善した．また，CMP の場合と同様，410.bwaves との組み合わせにおいてはわずかに性能低下が見られた．平均では 2 %性能が改善し，提案手法の有効性が示された．

5. ま と め

本稿では，マルチスレッド環境における共有キャッシュのマネジメント手法として，個々のロード/ストア命令の挙動に基づき，ラインのアロケートを制御する手法を提案した．個々のロード/ストア命令の挙動に基づいてラインのアロケートを細かく制御する事により，効果的に競合の影響を回避する事で性能を改善する．

提案手法をシミュレータに実装し，評価を行った結果，SMT プロセッサにおいて最大で 19.5 %，平均で 2 %性能が改善し，マルチスレッド環境における提案手法の有効性が示せた．

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) 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).
3) 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).
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) 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).
6) 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).
7) 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).
8) Tyson, G., Farrens, M., Matthews, J. and Pleszkun, A. R.: A modified approach to data cache management, *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 93–103 (1995).
9) Collins, J., Wang, H., Tullsen, D., Hughes, C., Lee, Y.-F., Lavery, D. and Shen, J.: Speculative precomputation: long-range prefetching of delinquent loads, pp. 14 –25 (2001).
10) 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).