修士論文

題目

高効率な分割領域動的管理キャッシュの研究

指導教員

近藤 利夫 教授

2017年

三重大学大学院 工学研究科 情報工学専攻
コンピュータアーキテクチャ研究室

刀根 舞歌(415M511)

内容梗概

近年，スマートフォンやパソコンなどのプロセッサでは，高性能化のために様々な手法が提案されている．その一つとして複数のコアを用意し，並列に処理を行うマルチコア化が挙げられる．しかし，マルチコア環境では複数のメモリアクセスが同時に発生することで，シングルコア環境よりもデータの競合が発生しやすくなり，結果としてキャッシュミス率が高くなるという問題がある．メモリアクセスは性能のボトルネックの一つであるため，高性能化を達成するためには主記憶へのアクセス数の低減，すなわちキャッシュミス率の低減が重要である．キャッシュミス低減手法の一つに，キャッシュ・パーティショニングがある．キャッシュ・パーティショニングは，同じエントリに格納されるデータを複数保持するため同一エントリを複数用意したもの，すなわちウェイをコアごとに割り当て，他のコアのアクセスによるデータ競合を抑える．しかし，キャッシュ・パーティショニングはウェイ単位でキャッシュ領域割り当てを行うため分配粒度が粗く，キャッシュを効率的に使えない場合がある．また，複数のコア間でデータの共有ができないという問題がある．そこで筆者は，キャッシュを細かい領域に分割して管理し，さらにデータ共有ができる，セル・アロケーションキャッシュを提案している．セル・アロケーションキャッシュは，キャッシュをウェイより細かい単位である「セル」に分割して動的にコアに割り当てる手法である．セル・アロケーションキャッシュには，各コアのみが主記憶からキャッシュへ書き込める固有セルと，通常キャッシュと同様にアクセスできる共有セルの 2 種類のセルが存在する．固有セルと共有セルを，キャッシュミス率やデータ共有率でコアに割り当てることで，各コアに最適なキャッシュ領域を提供できる．しかし，セルのアクセス制限により，かえってキャッシュミスが増加してしまう問題がある．また調査の結果，連続したインデックスでキャッシュヒットやキャッシュミスが連続することが分かった．そこで本研究では，セルのアクセス制限を優先度に変更し，直近のアクセス履歴を用いて不要なデータの削除やセル割り当てを行うように，割り当てアルゴリズムを改良した．アクセス履歴は，直近のアクセスとインデックスを調べ，コア番号が全て同一の場合，他のコアの固有セルを共有セルに変更するときなどに使用する．また後述の Other Index Counter を用いて，疑似的にウェイを増やし未使用なエントリを減らす手法を提案する．Other Index Counter は本

来格納されるインデックスと異なるエントリに格納されているデータが存在するかどうかを表すカウンタである．このカウンタがセットされている場合，別のインデックスに求めているデータが存在するか探索する．カウンタがセットされていない場合，キャッシュにデータが存在しないので主記憶からデータを取得する．事前にカウンタを調べることで不要なキャッシュの探索を減らすことができる．このカウンタにより使用されていないエントリを有効活用できる．これらの手法を用いることにより，通常キャッシュと比べて，提案手法ではキャッシュミス率が平均 4.50%，最大 6.96%，実行サイクル数が平均 8.10%，最大 10.85%減少した．

Abstract

Multi-core processors are widely used to improve computational performance. However, the number of main memory accesses on multi-core processor is larger than that of single-core processor. Therefore, a cache system for multi-core processor has possibility to degrade system performance. To solve the problem, there are many studies to improve cache performance. Cache partitioning is one of the representatives. Cache partitioning allocates ways to each core on demand, and restricts access from other core. This limitation prevents cache conflicts. Cache partitioning allocates cache spaces to cores efficiently, but cannot allocate optimally for shared data. To solve the problems, this paper proposes Cell-allocation cache. However, Cell-allocation cache increases miss rates by the limitation to write data from main memory to cache on serial memory accesses. To solve this performance degradation, this paper proposes mechanism to detect serial memory accesses which occurs serial cache misses. This paper also replaces the access limitation by the access priority, and improves the allocation algorithm. The proposed method can reduce waste cache accesses, and increase pseudo ways. Compared with a normal cache, the proposed method reduces 4.50% of cache miss rate and 8.10% of execution cycles on average.

目 次

図 目 次

表 目 次

1　まえがき

近年，スマートフォンやパソコンなどのプロセッサでは，高性能化のために様々な手法が提案されている．高性能化の手法の一つとして，複数のコアを用意し，並列に処理を行うマルチコア化が挙げられる．しかし，マルチコア環境では複数のメモリアクセスが同時に発生することで，シングルコア環境よりデータの競合が発生しやすくなり，結果としてキャッシュミス率が高くなるという問題がある．メモリアクセスは性能のボトルネックの一つであるため，高性能化を達成するためには主記憶へのメモリアクセスの低減，すなわちキャッシュミス率の低減が重要である．そのため様々な手法が提案されている．

その一つに，キャッシュ・パーティショニング [1] がある．キャッシュ・パーティショニングはウェイをコアごとに割り当て，データ競合を抑える割り当てるウェイ数はコアの負荷状況で決まるため，各コアに必要な分だけキャッシュ領域を割り当てられる．しかし，ウェイ単位で割り当てを行うため分配粒度が粗く，キャッシュを効率的に使えない場合がある．また，実際のプログラムで使われている，複数のコアによるデータの共有ができない問題がある．そこで，本研究ではキャッシュを細かい領域に分割して管理し，データ共有ができるセル・アロケーションキャッシュ[2] を提案

している．セル・アロケーションキャッシュは，キャッシュをウェイより細かい単位である「セル」に分割して動的にコアに割り当てる手法である．セル・アロケーションキャッシュには，各コアのみがメモリからキャッシュへ書き込める固有セルと，通常キャッシュと同様にアクセスできる共有セルの 2 種類のセルが存在する．固有セルと共有セルを，キャッシュミス率やデータ共有率でコアに割り当てることで，各コアに最適なキャッシュ領域を提供できる．しかし，セルのアクセス制限により，かえってミスが増加してしまう問題がある．また調査の結果，連続したインデックスでキャッシュヒットやキャッシュミスが連続することが分かった．そこで本稿では，セルのアクセス制限を優先度に変更し，直近のアクセス履歴を用いて不要なデータの削除やセル割り当てを行うように，割り当てアルゴリズムを改良した．また，疑似的にウェイを増やし未使用なエントリを減らす手法を提案する．これらの手法を用いることにより，通常キャッシュと比べて，提案手法ではキャッシュミス率が平均 4.50%，最大 6.96%，実行サイクル数が平均 8.10%，最大 10.85%減少した．

2　キャッシュシステムの概括

キャッシュはコアと主記憶の間に設置される，主記憶より小容量で高速なメモリである．使用頻度が高いデータをキャッシュに格納し，キャッシュから読み出すことにより，主記憶へのアクセスを減らし，実行時間を抑えられる．最も単純なキャッシュは，アドレスから格納先が一意に決まるダイレクトマッピングキャッシュである．

2.1　ダイレクトマッピングキャッシュ

ダイレクトマッピングキャッシュの概念図を図 2.1 に示す．

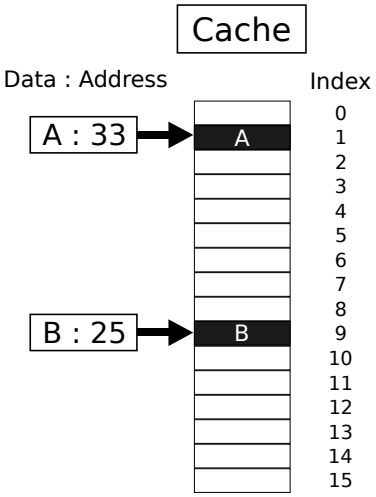

図 2.1: ダイレクトマッピングキャッシュの概念図

一般にダイレクトマッピングキャッシュはアドレスから剰余を用いて，インデックスと呼ばれる番地を計算する．具体的にはメモリアドレスを

address，キャッシュの管理単位であるライン長を line，キャッシュに格納できるデータ数を表す総エントリ数を entry とすると，インデックス index は下記の式 (1) で求まる (mod は剰余を表す).

$$\text{index} = \frac{\text{address}}{\text{line}} \bmod \frac{\text{entry}}{\text{line}} \tag{1}$$

そして，キャッシュの中のインデックスが一致するエントリに格納する．例えば図 2.1 のように，ライン長が 1，エントリ数が 16 のキャッシュの場合を考える．データ A のアドレスが 33 の場合，$33 \bmod 16 = 1$ となり，インデックスが 1 のエントリに格納する．次にデータ B を格納する．データ B のアドレスが 25 の場合，$25 \bmod 16 = 9$ となり，インデックスが 9 のエントリに格納する．このようにインデックスにばらつきを持たせて，なるべく異なる場所にデータを格納していく．しかし，同じインデックスのデータを格納する場合，現在格納しているデータを追い出してしまう．もし使用頻度の高いデータが追い出されると，キャッシュミスが増加し，主記憶へのアクセスが増えて実効性能が悪化する．そこで，多くのプロセッサではエントリを水平方向に増やした，セットアソシアティブキャッシュが用いられている．

2.2 セットアソシアティブキャッシュ

セットアソシアティブキャッシュの概念図を図 2.2 に示す.

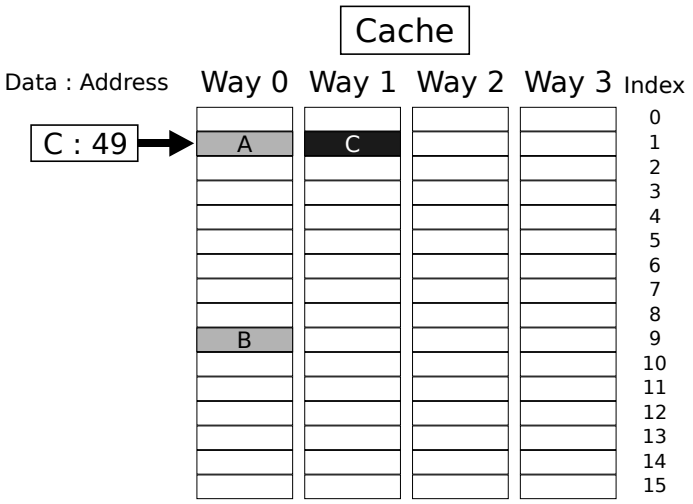

図 2.2: セットアソシアティブキャッシュの概念図

セットアソシアティブキャッシュは，1 つのエントリに複数のエントリを配置，すなわちウェイ数を増加し連想度を上げる．これにより，同じインデックスのデータを一度に複数保持できる．なお，同一インデックスの複数のエントリをまとめてセットと呼ぶ.

例えば図 2.2 のように，ウェイ 0 にデータ A とデータ B が格納されており，次にデータ C を格納する場合を考える．データ C のアドレスが 49 の場合，$49 \bmod 16 = 1$ となり，インデックスが 1 のエントリに格納しようとする．ここでダイレクトマッピングキャッシュでは，データ A とインデックスが同じであるため，データ A を追い出す．しかしセットアソ

シアティブキャッシュでは，図 2.1 のようにセット 1 のウェイ 1 が空いているため，ウェイ 1 に格納できる．次にデータ A を読み出す場合，ダイレクトマッピングキャッシュではキャッシュミスするが，セットアソシアティブキャッシュでは，キャッシュヒットとなり，ミスを低減できる．

このようにセットアソシアティブキャッシュでは，同じインデックスのデータを複数格納できるため，データの追い出しが減少し，主記憶へのアクセスが減り実効性能の低下を防げる．しかし，マルチコア環境ではコア同士で特定のインデックスを奪い合うと，データの追い出しが増加する恐れがある．

3　先行研究

3.1　キャッシュ・パーティショニング

従来より，シングルコア用キャッシュシステムにおいても，データの競合を避けるためセット・アソシアティブキャッシュが広く用いられている．セット・アソシアティブキャッシュは，ウェイ数を増やすことで，同じインデックスのデータを複数格納できるためデータの競合が減少し，メモリアクセスの削減に繋がる．しかし，マルチコア環境においては各コアが異なるアドレス空間にアクセスするため，メモリ競合が発生しやすい．この問題を解決する手法の一つとして，コアにウェイを動的に割当て，ミス減少数でウェイ数を調節し，各コアに最適な領域を確保することで競合を低減するキャッシュ・パーティショニングがある．プロセス同士が独立である場合，データを共有しないため，メモリ競合を引き起こしやすい．そこでキャッシュ・パーティショニングは，各コアの負荷に応じてウェイを割り当てることで，メモリ競合を防ぐ．キャッシュ・パーティショニングでは，各コアの負荷を 1 ウェイあたりのミス減少数で判断する．各コアのミス減少数が同程度の場合，ウェイはほぼ均等に割り当てられる．ミス減少数の低いコアが，よりメモリを必要とする場合は，ミス減少量の高いコアの保持するキャッシュを渡す．割り当てられたウェイは，そのコ

アからしかアクセスできない．ウェイにアクセスできるコアを制限すると，各コアが使うデータのみが割り当てられたウェイに格納される．そのため，別のコアのデータによる追い出しが減少する．このように動的にウェイをコアに割り当てることで，複数のコアによるメモリ競合を減らし，キャッシュを効率良く使うことができる．しかし，未使用のウェイでは無駄に電力を消費する．また，実際のプログラムで使われる，共有データについては考慮外である．

3.2 ウェイ・アロケーションキャッシュ

ウェイ・アロケーションキャッシュ[3] は前節のキャッシュ・パーティショニングを，省電力化したキャッシュである．一般にメモリアクセスには参照の局所性，すなわちアクセスするメモリアドレスが一部のメモリ領域に集中する傾向がある．そのため，キャッシュ・パーティショニングでは，割り当てられても使われないウェイが発生する可能性がある．このとき，待機時に漏れ出る電流に起因する無駄な電力，すなわちリーク電力が発生する．そこで，ウェイ・アロケーションキャッシュでは，使われていないウェイをシャットダウンし，電力供給を停止して不活性化する．これにより，性能を維持しつつ，無駄な電力を削減できる手法である．しかし，

ウェイ単位でしか割り当てやシャットダウンができないため，必要な部分だけを使用したいときでも，不要な領域が割り当てられ，不要な領域がシャットダウンできない．

3.3 Victim-Guided Cache Partitioning

キャッシュ･パーティショニングは，ミス減少数を求めるのに必要なハードウェアの規模や，性質上，置換方法がLRUに限定される問題がある．そこで，Victim-Guided Cache Partitioning[4] では各コアに独立したvictimキャッシュを用意し，そのヒット数でキャッシュをパーティショニングする．これにより，より単純な実装で，置換方式が限定されない構成にできる．ただし，ミスが増加する場合もあり，性能向上が不十分という問題がある．

3.4 パーティション・シェアリング

パーティション・シェアリング [5] は，キャッシュ領域を分割するパーティショニングと，キャッシュ領域を複数のコアが共有するシェアリングを組み合わせた手法である．キャッシュパーティショニングやウェイアロケーションキャッシュでは，分割された1つの領域には，1つのコアしか割り当てられない．これに対して，パーティション・シェアリングでは，分

割された1つの領域を，複数のコアが共有する．各コアのプログラムの動作に応じて，コアをグループ化し，キャッシュ領域を共有することで，メモリ競合を抑え，キャッシュを効率良く扱うことができる．しかし，パーティション・シェアリングのハードウェア構想・実装は行われていない．

4　可変レベルキャッシュ

筆者はこれまでにシングルコア用キャッシュの高性能化手法として，第3.2節のウェイ・アロケーションキャッシュを発展させた可変レベルキャッシュ[6]の研究を行ってきた．本研究で提案している手法は可変レベルキャッシュの発展型であるため，本章ではまず可変レベルキャッシュについて概括する．

可変レベルキャッシュは第3.2節のウェイ・アロケーションキャッシュを拡張し，キャッシュの$\frac{1}{4}$単位の領域の状態を，モードで切り換える手法である．可変レベルキャッシュの概要図を，図4.3に示す．

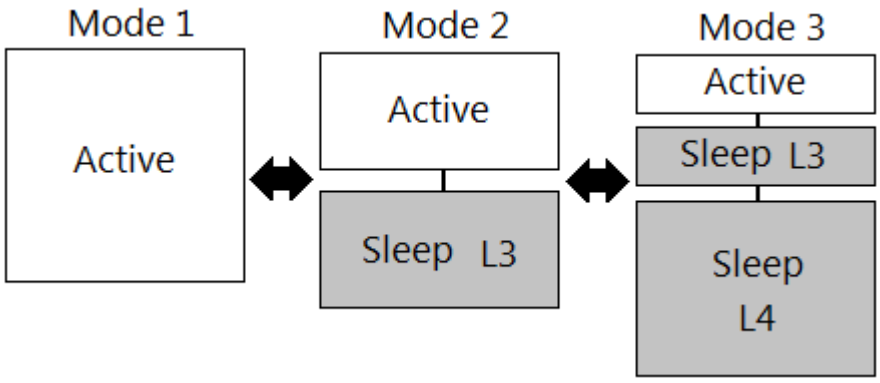

図 4.3: 可変レベルキャッシュの概念図

モードを切り換えると，モードに応じた領域をシャットダウン・スリープ状態にする．シャットダウン状態はデータを保持できない代わりに，電力を一切使わない状態である．スリープ状態はデータを保持できる最低

電力状態である．動的にモードを切り換えることで，ウェイ・アロケーションキャッシュより高性能・低消費電力で動作できる．モード 1 は通常キャッシュと同じ状態である．モード 2 はキャッシュの下半分の領域を，シャットダウンまたはスリープ状態にする．モード 3 はキャッシュの下 $\frac{3}{4}$ 領域を，シャットダウンまたはスリープ状態にする．モード 1 から実行し始め，アクセスやミス率を調べ，キャッシュ容量をあまり必要としていないと判断した場合，1 つ下位のモードへ移行する．モード 2 やモード 3 でミスが増加した場合，上位のモードへ戻る．このようにプログラムの挙動に応じて動的にキャッシュの状態を切り換え，不要な領域の電力を減らしつつ，性能を維持できる．

しかし，可変レベルキャッシュの研究対象はシングルコア・シングルスレッドであり，現在主流であるマルチコア・マルチスレッドについては考慮していない．また，キャッシュを $\frac{1}{4}$ 単位と大きく分けるため，必要な部分までシャットダウン・スリープ状態にする可能性もある．そこで，本研究ではマルチコア・マルチスレッドを扱え，より細かい領域でキャッシュを管理できるセル・アロケーションキャッシュを提案している．

5　セル・アロケーションキャッシュの提案

5.1　セル・アロケーションキャッシュの概要

先行研究で示した手法は，いずれもキャッシュ領域を分割して割り当てることで，キャッシュを効率良く扱っている．しかし，ウェイ単位での割り当ては分配粒度が粗いため，必ずしも最適なメモリ割り当てができるとはいえない．例えば，あるコアのアクセスが特定のインデックス付近にアクセスが集中すると，ウェイの一部分しか使っていないにもかかわらず，そのウェイを独占してしまう．そのため，実際には使われないキャッシュ領域が発生する．他のコアがそのキャッシュ領域を使おうとしてもアクセスできないため，キャッシュミス率が上昇し，メモリアクセスの増加を招く．また，前章で提案した可変レベルキャッシュは，不要な領域はシャットダウンやスリープ状態により電力削減できるが，キャッシュを $\frac{1}{4}$ 単位と大きく分けるため，必要な部分までシャットダウンやスリープ状態にする危険性もある．よって，分配粒度をより細かくする必要がある．

また，先行研究の多くは，複数のコアやスレッドが同じデータを用いる共有データを想定しておらず，自分に割り当てられたウェイ以外のウェイにアクセスできないため，データを共有できないという問題がある．実際のプログラムでは，マルチスレッドプログラムのように，複数のコア

間で同じデータを使う場合がある．また，可変レベルキャッシュはマルチスレッドが考慮外のため，必然的に共有データを想定していない．

これらの問題を解決するために，筆者はキャッシュ・パーティショニングを応用した，セル・アロケーションキャッシュを提案している．セル・アロケーションキャッシュの概念図を図 5.4 に示す．

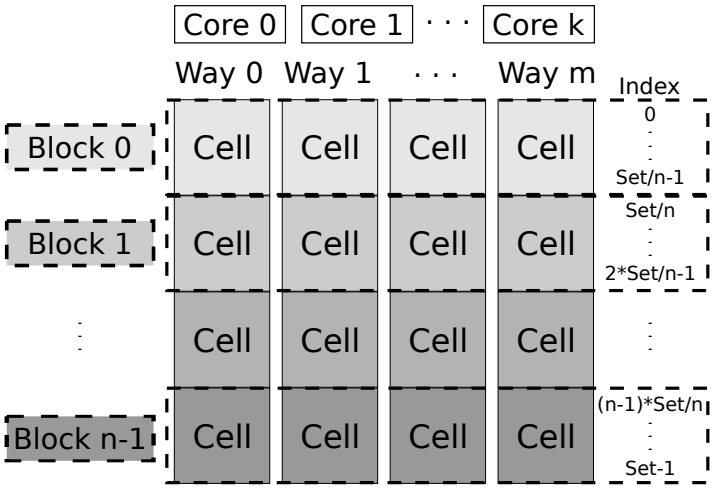

図 5.4: セル・アロケーションキャッシュの概念図

セル・アロケーションキャッシュは 1 個のウェイを，複数の細分化した領域である「セル」を単位としてコアに割り当てる．セルは複数のエントリをまとめたグループである．また，図 5.4 の破線のように，ウェイ方向に並んだセルのグループをブロックとする．

セル・アロケーションキャッシュは，一定サイクルごとに各ブロックでのミスを数えて，各コアがよくミスする，すなわち各コアがよく使用す

るブロックのセルをそのコアに割り当てる．各コアが必要としているセルだけを割り当てることで，キャッシュ領域をより活用できる．

また，データ共有を考慮し，各セルをデータ共有率とミス率に応じて，共有セルと固有セルに分ける．共有セルは，通常キャッシュと同様にアクセスできるセルである．固有セルは，キャッシュミス時の書き込み先を，要求したコアに割当てられたセルのみに限定するセルである．この 2 種類のセルを使い分けることで，データを共有しつつ，データの競合を減らし，キャッシュの有用性を高める．

このような特徴を持つセルを用いて，より効率的な割り当てを目指すキャッシュである．共有セルと固有セルについては第 5.2 節で，割り当てに用いる共有率については第 5.3 節で，セル・アロケーションキャッシュのアルゴリズムについては第 5.4 節で，セル・アロケーションキャッシュの動作については第 5.5 節で，セル・アロケーションキャッシュの評価については第 5.6 節で詳しく説明する．

5.2　共有セルと固有セル

第 3 章における先行研究の多くは，分割された領域に対して 1 つのコアのみがアクセスできた．しかし共有率の解析結果より，分割された領

域に複数のコアがアクセスできれば，性能が向上する可能性があることが分かった．そこで，複数のコアがアクセスできる共有領域，共有セルを追加する．共有セルは，通常キャッシュと同様にアクセスできるキャッシュ領域である．固有セルは，キャッシュミス時の書き込み先を，要求したコアに割当てられたセルのみに限定するセルである．多くの先行研究と異なり，コアからの読み書きは通常キャッシュと同様に固有セルでも行えるように拡張した．共有セルと固有セルの違いは，各コアのみがミス時に書き込めるセルと，全コアが通常キャッシュと同様にアクセスできる共有セルにデータを分けることにより，ミス時のデータの競合を低減してミス率の削減をはかる．

5.3 セルの共有率の定義

セル内の総エントリ数を entry_{all}，セル内の共有エントリの総数を entry_{shared} とするとセル割り当てに用いるセルの共有率 S_{cell} は式 (2) のように定義できる．

$$S_{cell} = \frac{\mathrm{entry}_{shared}}{\mathrm{entry}_{all}} \tag{2}$$

セル・アロケーションキャッシュでは，S_{cell} とキャッシュミス率を用いてセルをコアに割り当てる．初期状態では，キャッシュ上のセルは全て固

有セルであり，各コアにセルの数が均等になるように割り当てられる．セルの割り当て変更時に，最初に各 S_{cell} を確認し，S_{cell} が閾値以上であれば，共有セルに変更する．なお，今回は S_{cell} の値は実験的に求めた最適値を用いた．その後，各コアのキャッシュミス率でセル割り当てを行う．

5.4 セル割り当てアルゴリズム

セルをコアに割り当てるアルゴリズムを図 5.5 に示す．

セル割り当てアルゴリズムは，主に 3 つの関数から成り立っている．図 5.5(a) の基本処理を行う関数 Algorithm_for_cell_allocation，図 5.5(b) のどの領域でセル割り当てを行うか決める関数 Main_judgment，図 5.5(c) の実際にセルをコアに割り当てる関数 Cell_allocation である．なお，図 5.5 の PrivateCell[k] はコア k の固有セルを，Block[n] はブロック n を表す．

最初に，図 5.5(a) が実行される．まず現在のサイクル数を調べ，セルの割り当てを行うサイクルを超えているかどうか調べる．超えている場合，各セルの共有率を確認する．セルの共有率が上方の閾値を超えている場合，共有セルに変更する．反対に下方の閾値を下回る場合，最もセル数が多いコアの固有セルに変更する．共有率で共有セルに変更するかどうか処理した後，一定サイクル内にアクセスがあったか調べる．アク

(a) Algorithm_for_cell_allocation()

```
While(Current cycle > Interval)
 foreach Cell
    if(Share rate  > Upper Bound)
       Change to Shared Cell;
    else if (Share rate > Lower Bound)
       Change to PrivateCell[core having most Cells];
 if(Accesses > 0)
      CoreHigh = core number where miss rate is highest;
      CoreLow = core number where miss rate is lowest;
 if(Accessesof CoreHigh and CoreLow > 0)
      Main_judgement(CoreHigh, CoreLow);
 else if(Accesses of CoreLow = 0)
      Main_judgement(CoreHigh, CoreLow);
 set Label : RESET;
 Reset parameters;
```

(b) Main_judgement(CoreHigh, CoreLow)

```
Block1st = block number where misses the most;
Block2nd = block number where misses the 2nd most;
if(There are 1 or more PrivateCell[CoreLow] in Cache)
     Cell_allocation(CoreHigh, CoreLow, Block1st);
     Cell_allocation(CoreHigh, CoreLow, Block2nd);
     Cell_allocation(CoreHigh, CoreLow, Cache);
```

(c) Cell_allocation(CoreHigh, CoreLow, Here)

```
if(There are 2 or more PrivateCell[CoreLow] in Cache)
   PrivateCell[CoreHigh] = least recently used
                              PrivateCell[CoreLow] in Block[Here];
   Allocated = true;
else /* if(There is a PrivateCell[CoreLow] in Cache) */
   change PrivateCell[CoreLow] in Block[Here] to Shared Cell;
   Allocated = true;
if(Allocated)
   go to RESET;
```

図 5.5: 疑似関数によるセル割り当てアルゴリズム

セスがあった場合，保持するセル数が 1 個以上のコアの内，ミス率が最も高いコアと低いコアを求める．次に，両方のコアでアクセスがあったかどうか調べる．両方のコアでアクセスがあった場合，図 5.5(b) に移行する．

図 5.5(b) では，最初にミス率が最も高いコアのミス回数が，最も多い

ブロックと 2 番目に多いブロックを求める．次に，図 5.5(c) を実行し，最も多いブロックにある固有セルを渡すかどうか判断する．

図 5.5(c) では，ミス率が最も低いコアの固有セルが，キャッシュに 1 個以上ある場合，最も多いブロックにある固有セルの中から，ミス率が最も高いコアへ渡すセルを決める．このとき，セルを渡すコアの固有セル数が 2 個以上の場合，LRU を用いて最も最近使われていない固有セルを渡す．セルを渡すコアの固有セル数が 1 個の場合，その固有セルを共有セルに変更する．ここで共有セルに変更することにより，固有セルを持たないコアでもメモリからキャッシュへ書き込めるようにしている．この時点でセルの割り当てが行われている場合，セル割り当て判定を終了し，パラメータのリセットへ移行する．

ブロックでセル割り当てができなかったら，次に 2 番目のブロックで 1 番目のブロックと同様に，図 5.5(c) を行う．ただし，1 番目のブロックとは異なり，渡すセルは 2 番目に多いブロックにある固有セルの中から決定する．

1 番目と 2 番目の両方のブロックでセル割り当てが行われなかった場合，キャッシュ全体で同様に図 5.5(c) を行う．ただし，1 番目や 2 番目のブロックとは異なり，渡すセルはキャッシュ全体にある固有セルの中から

決定する．

この時点でセル割り当てが行われていない場合，ミス率が最も低いコアでアクセスがあったかどうか調べる．ミス率が最も低いコアでアクセスが無かった場合，図 5.5(b) に移行し，処理する．それでもセル割り当てが行われなかった場合，セルの割り当ては変更されない．

最後に，各パラメータをリセットする．一定サイクルごとに，この作業を繰り返す．

5.5 セル・アロケーションキャッシュの動作

セル・アロケーションキャッシュと通常キャッシュとの違いは，キャッシュミス時のデータの格納先を決める場合に，同じコアに属する領域へ優先的に書き込む点である．

共有セルと固有セルのキャッシュミス時の動作を説明する．あるコアでキャッシュミスが発生した場合，メモリからデータを取得する．その際の書き込み先は，共有セルかそのコアの固有セルが優先される．共有セルとそのコアの固有セルの両方が存在する場合，その中から LRU 法でどのセルに書き込むか決定する．共有セルとそのコアの固有セルに書き込めない場合にのみ，他のコアの固有セルを共有セルにして書き込む．

共有セルがあり，そのコアの固有セルが存在しない場合，共有セルの中からLRU法でどの共有セルに書き込むか決定する．共有セルに書き込めない場合にのみ，他のコアの固有セルを共有セルにして書き込む．

共有セルが無く，そのコアの固有セルが存在する場合，固有セルの中からLRU法でどの固有セルに書き込むか決定する．そのコアの固有セルに書き込めない場合にのみ，他のコアの固有セルを共有セルにして書き込む．

設計上，共有セルもそのコアの固有セルも存在しない状況は発生しない．この手法では，他のコアのデータを汚染しない分，各コアが格納できるデータ量は減少する．

5.6 セル・アロケーションキャッシュの性能評価

セル・アロケーションキャッシュは通常キャッシュと比較して，ミス率を最大0.43%，実行サイクル数を最大0.92%低減できたが，平均ではミス率が0.58%，実行サイクル数が2.90%増加した．原因として，あるコアのミス時に主記憶からキャッシュへ書き込む際に，他のコアの固有セルへ書き込めないという制限が考えられる．この制限により，そのコアの必要としているデータを追い出して上書きするため，かえってキャッシュミス

が増加してしまう問題がある．また，特定のブロックにアクセスが集中している場合，他のブロックのセルに空きが存在していても，インデックスが異なるため格納できず，アクセスが集中しているブロックにある必要なデータを追い出す可能性がある．

6　セル・アロケーションキャッシュの改良

6.1　セル・アロケーションキャッシュの問題点

セル・アロケーションキャッシュによって，コア間で共有データを使え，かつより細かくキャッシュ領域を割り当てることを可能にした．しかし，固有セルへの書き込み制限により必要なデータを上書きし，キャッシュミスが増加してしまうという問題が残っている．また，セル・アロケーションキャッシュでは特定のブロックにアクセスが集中している場合，他のブロックのセルに空きがあるにもかかわらず使われない可能性がある．さらに，プログラムのキャッシュへのアクセス特性を調査した結果，あるコアが連続するインデックスにアクセスする場合，ヒットまたはミスも連続することが判明した．そこで，セルのアクセス制限を優先度に変更し，格納先の変換により疑似的にウェイを増やして未使用なエントリを減らし，直近のアクセス履歴を用いて不要なデータの削除やセル割り当てを行うように改良した．

改良後のセル・アロケーションキャッシュは，メモリからキャッシュにデータを書き込むときに，本来のセット，固有セル，共有セルの順に格納先を探す．もし適切な格納先が存在しない場合，セル割り当てを無視して本来のインデックスに強制格納する．本来のセットが他のコアの固有

セルにあっても書き込めるため，他のコアの固有セルにあるエントリがあまり使われていないときに，従来では使えなかった場所を活用できる．

また，第6.2節で詳しく説明するように，ブロック番号を用いた格納先の変換により，擬似的にウェイを増加できる．ブロック番号を追加したセルの詳細図を図6.6に示す．

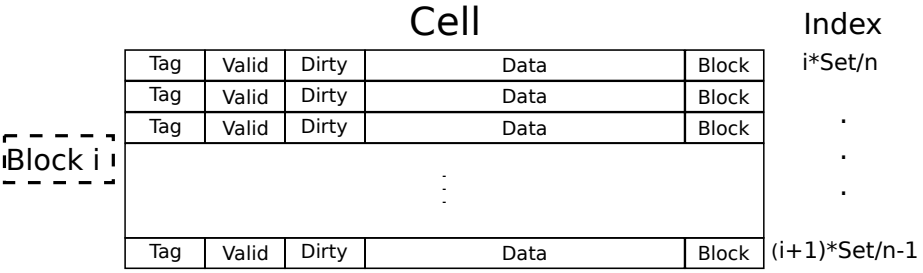

図 6.6: セルの詳細

ブロック番号は現在格納されているデータのインデックスが，実際にはどのブロックに所属するのかを示す番号である．格納されているデータのインデックスと，ブロック番号を組み合わせると，そのデータの本来のインデックスが求まる．本来のインデックスを $\mathrm{index}_{original}$，セット数を set，ブロック数を block，現在調べているインデックスを index，ブロック番号を num_{block} とすると，以下の式 (3) で求まる．

$$\mathrm{index}_{original} = \mathrm{index} \bmod \frac{\mathrm{set}}{\mathrm{block}} + \mathrm{num}_{block} \times \frac{\mathrm{set}}{\mathrm{block}} \tag{3}$$

例えば，セット数が4096，ブロック数が4，インデックスが1573，格納

されているブロック番号が 0 の場合，$1573 \bmod \frac{4096}{4} + 0 \times \frac{4096}{4} = 549$ という式で求められ，このデータの本来のインデックスは 549 となる．このように，ブロック番号を利用すれば，本来のインデックスのエントリに空きが無くても，別のインデックスのエントリに格納ができる．この動作は，一時的なウェイ数の増加である．ブロック番号は後述の Other Index Counter によって管理される．

また改良後のセル･アロケーションキャッシュは，直近のアクセス履歴を記録し，ミスが連続する領域のデータを無効化し，あるコアだけがキャッシュを使っている場合は他のコアの固有セルを全て共有セルに変更する．これにより，無駄なアクセスを防ぎ，より多くの領域を活用できる．

ブロック番号による格納先の変換については第 6.2 節で，アクセス履歴については第 6.3 節で，改良後のセル・アロケーションキャッシュのアルゴリズムについては第 6.4 節で，改良後のセル･アロケーションキャッシュの動作については第 6.5 節で詳しく説明する．

6.2 格納先の変換による擬似的ウェイ増加

従来のセル・アロケーションキャッシュでは，特定のブロックのみを多く使う場合，他のブロックのセルに未使用のエントリがあっても，イン

デックスが異なるために未使用のエントリを使えず，データの追い出しが発生することがあった．そこで，セルのブロック番号を管理するテーブル，Other Index Counter(OIC) を作成し，本来のセットから格納先を変更して，未使用のエントリを活用する手法を提案する．この手法では，OIC に本来のセットと異なるエントリに格納されたデータがあるかどうかを記憶し，読み込み時に参照することで，擬似的にウェイを増加できる．これにより，使われないエントリを減らすとともに，多く使われているインデックスのデータをより多く保持できる．

OIC は，各セットに対してブロック分のビット数を持つテーブルである．各ビットはブロックに対応しており，初期値は全て 0 である．あるインデックスに格納されるはずだったデータが他のセットに格納される際に，格納先のブロックに対応するビットを 1 にする．次にデータを読み込むときに，OIC を参照し，ブロック番号に対応するビットを確認することで，データがキャッシュに存在するか，どのブロックに存在するか判断できる．1 ビットでブロック番号を表すことで，ウェイや容量が増加しても，小さなハードウェアでブロック番号を管理できる．

セット数を set，ブロック数を block，インデックスを index とすると，

ブロック番号 num_{block} は式 (4) を満たす値となる．

$$\mathrm{num}_{block} \times \frac{\mathrm{set}}{\mathrm{block}} < \mathrm{index} < (\mathrm{num}_{block} + 1) \times \frac{\mathrm{set}}{\mathrm{block}} \tag{4}$$

例として，セット数が 16，ウェイ数が 4 で，格納したいデータのインデックスが 1 の場合を考える．インデックスが 1 の場合，本来のブロック番号は 0 である．

メモリからキャッシュに書き込むときに，まず通常通りインデックスでエントリを探す．インデックスが 1 の場合は，セット 1 のエントリに格納できるか確認する．格納できる場合は，図 6.7 のように OIC のセット番号が 1 のカウンタの，最上位ビットを 1 にし，格納エントリのブロック番号を 0 で上書きする．

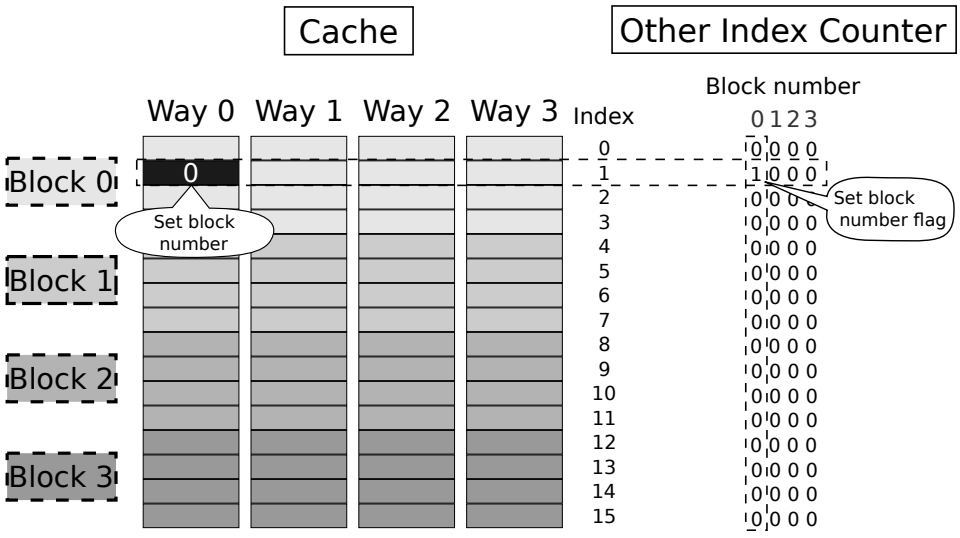

図 6.7: 本来のセットに格納する場合

本来のインデックスで格納できない場合は，インデックスをブロック数で割った剰余が同じになるセットに格納できるか確認する．インデックスが 1 の場合は，インデックスが 5,9,13 であるセットを確認する．このいずれかに格納できる場合は，そのエントリのブロック番号に，本来のブロック番号を格納する．例えば，セット 9 のエントリに格納できるならば，セット 9 の本来のブロック番号は 2 であるため，図 6.8 のように OIC のセット番号が 1 のカウンタの，上から 3 番目のビットを 1 にし，格納エントリのブロック番号を 0 で上書きする．

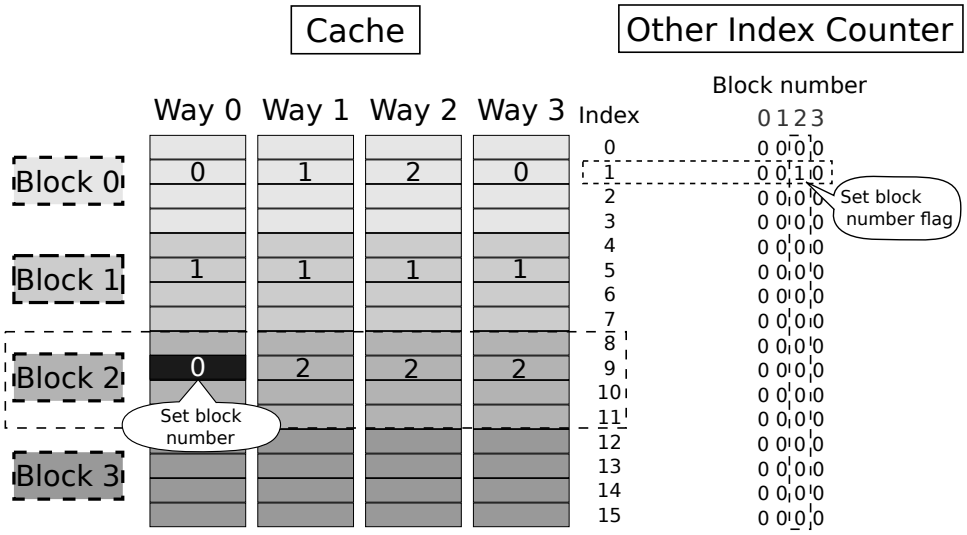

図 6.8: セット 9 に格納する場合

それでも格納できない場合は，本来のインデックスのエントリに格納する．このとき，追い出されるデータについて OIC を調べ，必要に応じて書き換える．図 6.9 のように追い出されるデータのブロック番号が 0 で

ある場合は，OIC を書き換える必要が無い．

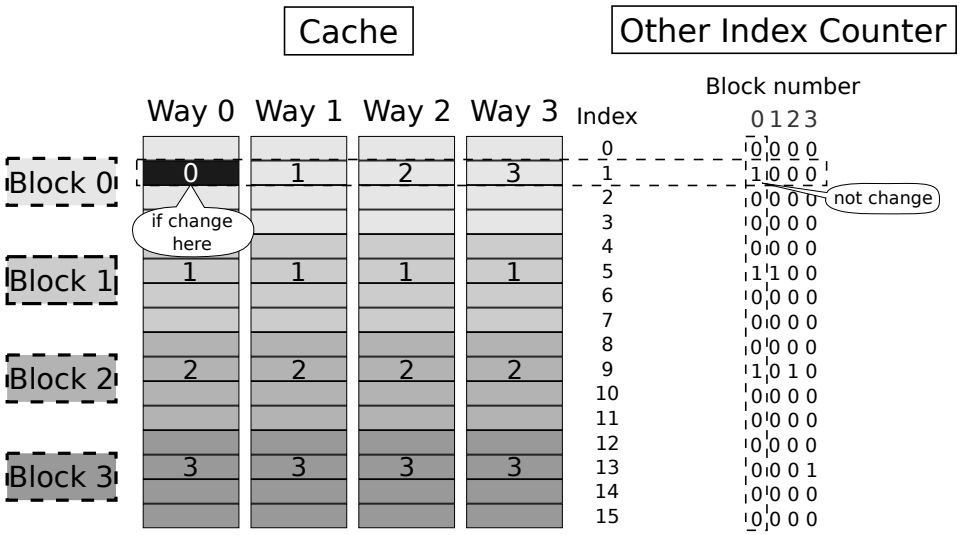

図 6.9: ブロック番号が 0 のデータを追い出す場合

図 6.10 のように，追い出されるデータのブロック番号が 0 以外である場合は，セット 1 に格納されている他のデータを調べ，追い出されるデータのブロック番号と同じブロック番号のデータがブロック 0 に存在するか確認する．存在する場合は OIC を書き換える必要は無い．

存在しない場合，ブロック 0 から追い出されるデータのブロック番号を持つデータが無くなるため，図 6.11 のように，OIC の追い出されるデータのセット番号のカウンタの，最上位ビットを 0 にする．最後に，格納エントリのブロック番号を 0 で上書きする．

反対に，キャッシュから読み込むときは，ブロック番号を確認する．そしてブロック番号を用いて，格納されているデータの本来のインデック

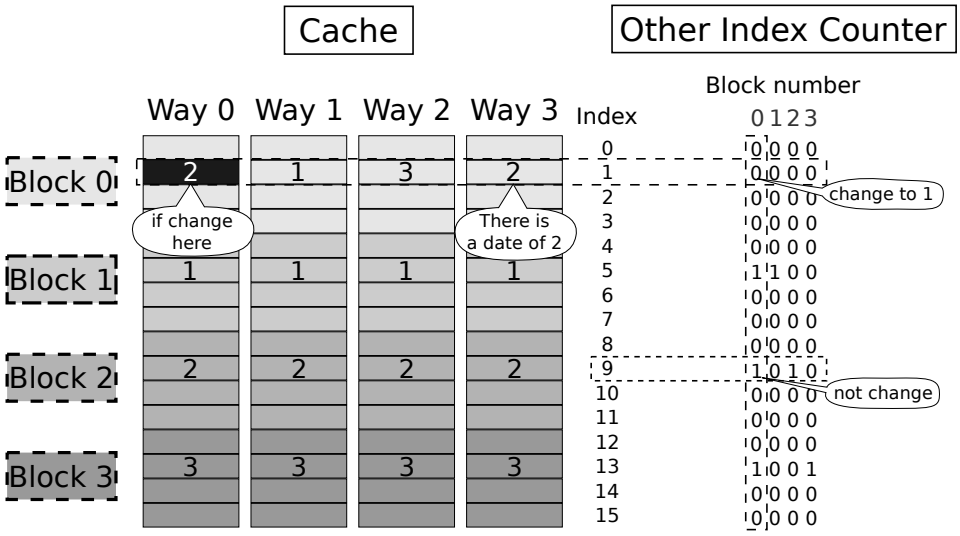

図 6.10: ブロック番号が 0 以外のデータを追い出す場合

スを計算する．セット数を set，ブロック数を block，現在調べているインデックスを index，ブロック番号を num_{block} とすると，本来のインデックス $\mathrm{index}_{original}$ は，式 (5) で求められる．

$$\mathrm{index}_{original} = \mathrm{index} \bmod \frac{\mathrm{set}}{\mathrm{block}} + \mathrm{num}_{block} \times \frac{\mathrm{set}}{\mathrm{block}} \tag{5}$$

例えば，インデックスが 1 の場合，まず図 6.12 のように，OIC のセットが 1 のカウンタを確認する．もし最上位ビットが 1 の場合，ブロック 0 にインデックスが 1 のデータが存在するので，セット 1 を調べる．そしてブロック番号が 0 のエントリがある場合，$1 \bmod \frac{16}{4} + 0 \times \frac{16}{4} = 1$ なので，本来のインデックスは 1 となり，インデックスが一致するデータと分かる．

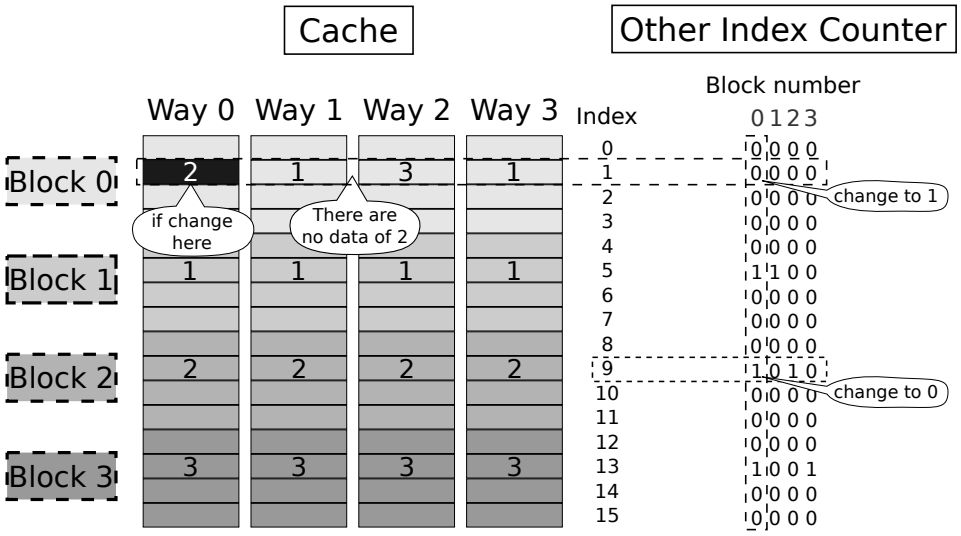

図 6.11: ブロック番号が 0 以外のデータを追い出し書き換える場合

もしブロック番号が 0 以外であれば，それは別のブロックに格納するはずだったデータと分かる．例えば，ブロック番号が 2 の場合，$1 \bmod \frac{16}{4} + 2 \times \frac{16}{4} = 9$ なので，本来のインデックスは 9 となり，インデックスが一致しない別のデータと分かる．

本来のインデックスでデータが見つからなかった場合，まず図 6.13 のように，OIC のセットが 1 のカウンタを確認する．最上位ビット以外で 1 がある場合，対応するブロックにインデックスが 1 のデータが存在するので，そのブロックの，セット数をブロック数で割った剰余が同じになるセットを探索する．そしてブロック番号が 0 のエントリがある場合，$9 \bmod \frac{16}{4} + 0 \times \frac{16}{4} = 1$ なので，本来のインデックスは 1 となり，このデータがインデックスが 1 のデータであると分かる．

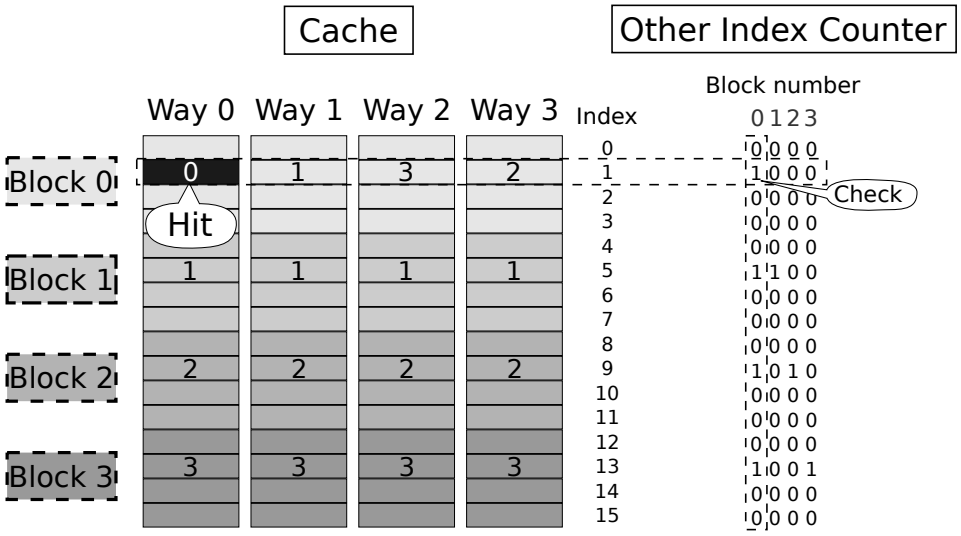

図 6.12: セット 1 の探索

6.3 アクセス履歴

プログラムのキャッシュへのアクセス特性を調査した結果，あるコアが連続するインデックスにアクセスする場合，ヒットまたはミスも連続することが判明した．そこで，直近のアクセス履歴を記録し，それを使うようにセル・アロケーションキャッシュのアルゴリズムを改良した．新しいセル割り当てアルゴリズムでは，キャッシュにアクセスがあったときに，格納先のセット，タグ，ヒット・ミスといった情報を記録する．直近の 8 回分のアクセスを記録しておき，キャッシュミス時とセルの再割り当て時にその履歴を用いる．

キャッシュミス時には，最近ミスしたコア番号とインデックスを調べ

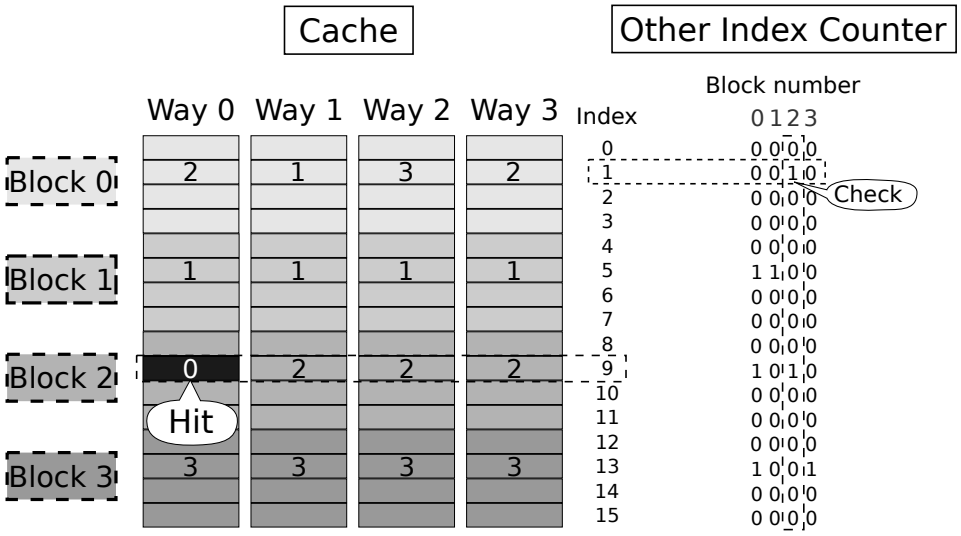

図 6.13: セット 9 の探索

る．もしコア番号とタグが全て同じで，インデックスが連続している場合，今後も同じデータをキャッシュしようとしてミスするとみなし，ミスしたセットの直後 16 セットにあるデータを無効化する．これによって無駄なアクセスを防ぎ，実行時間を短縮する．

セルの再割り当て時には，最近アクセスしたコア番号を調べる．全て同一だった場合，他のコアはしばらくキャッシュを使用しないとみなし，そのコア以外の固有セルを共有セルにする．こうすることでより多くの領域を活用できる．

6.4　改良後のセル割り当てアルゴリズム

改良後のセル割り当てアルゴリズムは，第5.4節で説明したアルゴリズムを拡張したものである，図5.5の(a)において，共有率を確認する前にアクセス履歴を確認する．直近8回のアクセスを調べ，コア番号が全て同一だった場合，他のコアはしばらくキャッシュを使用しないとみなし，そのコア以外の固有セルを共有セルにする．アクセス履歴による割り当てを行った場合，以降の処理は行わずパラメータのリセットまで移行する．コア番号が全て同一でない場合は，第5.4節で説明したアルゴリズムどおりにセルの再割り当てを行う．

6.5　改良後のセル・アロケーションキャッシュの動作

改良後のセル・アロケーションキャッシュは，あるコアでキャッシュミスが発生し，メモリからキャッシュにデータを書き込むときに，まず本来のインデックスに空きがあるかどうか確認する．空きが無い場合，アクセス履歴を調べ，直近8回のアクセスのコア番号を調べる．コア番号が現在ミスしているコアと同じで，全て同一だった場合，第6.2節で説明したように，そのコアの固有セル内で同じインデックスになりうるインデックスで空きが無いかどうか調べる．同様に共有セルでも調べる．もし適

切な格納先が存在しない場合，セル割り当てを無視して本来のインデックスに強制格納する．本来のインデックスが他のコアの固有セルにあっても書き込めるため，他のコアの固有セルにあるエントリがあまり使われていないときに，従来では使えなかった場所を活用できる．

7　性能評価

7.1　評価方法

改良したセル・アロケーションキャッシュをトレースドリブン型シミュレータの L2 共有キャッシュに実装し，評価した．比較対象は通常キャッシュとする．第 3 章で述べた先行研究は，データの共有が出来ないため，比較対象から除外する．評価項目は高性能を目標とするため，キャッシュミス率と実行サイクル数とする．評価環境は表 7.1 のとおりで，L2 共有キャッシュの全容量は 2MB である．

表 7.1: 評価環境のパラメータ一覧

コア数	4 cores	アクセス・レイテンシ	20 cycels
ウェイ数	8 ways	メモリ・レイテンシ	250 cycles
セット数	4096 sets	共有率の閾値 (上方)	30 %
ライン長	64 Bytes	共有率の閾値 (下方)	10 %

また，ベンチマークとして姫野ベンチマークと，Splash2 ベンチマークの中からワーキングセットの異なる 2 種類を選んで組み合わせ，評価した．

7.2　評価結果

通常キャッシュと提案手法のキャッシュミス率のグラフを，図 7.14 に記載する．

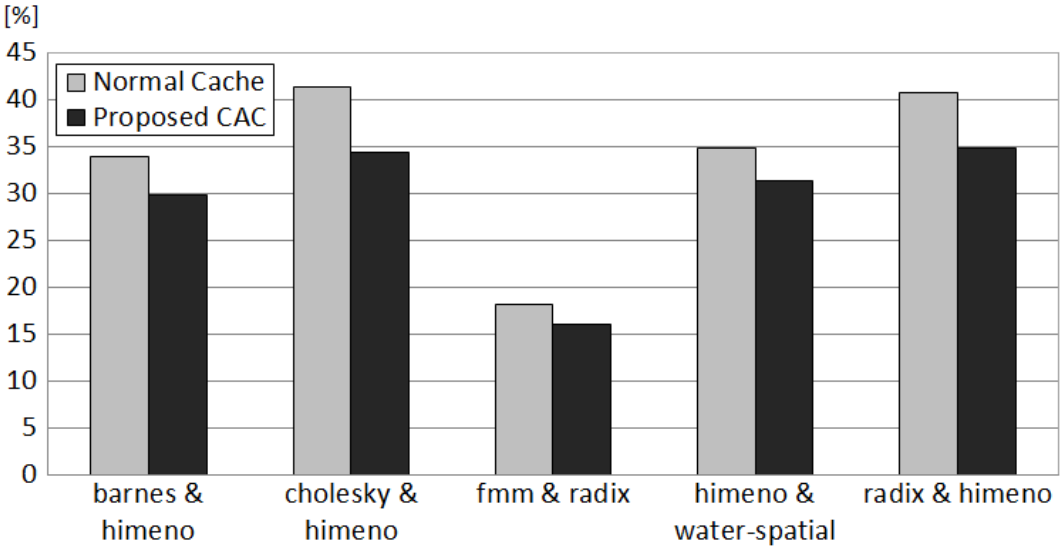

図 7.14: キャッシュミス率

また通常キャッシュと提案手法の実行サイクル数を，通常キャッシュで正規化したグラフを，図 7.15 に記載する．

通常キャッシュと比べて，提案手法ではキャッシュミス率が平均 4.50%，最大 6.96%，実行サイクル数が平均 8.10%，最大 10.85%減少した．

7.3 考察

通常キャッシュと比べて，提案手法ではキャッシュミス率を低減することで実行サイクル数を平均 8.10%，最大 10.85%減少した．今回は時間の都合上，比較的メモリ使用量の大きなベンチマークの組み合わせでしか評価ができなかったため，キャッシュの余剰領域が少なく大幅なミス率低減は得られなかったが，大規模なプログラムと小規模なプログラムを同

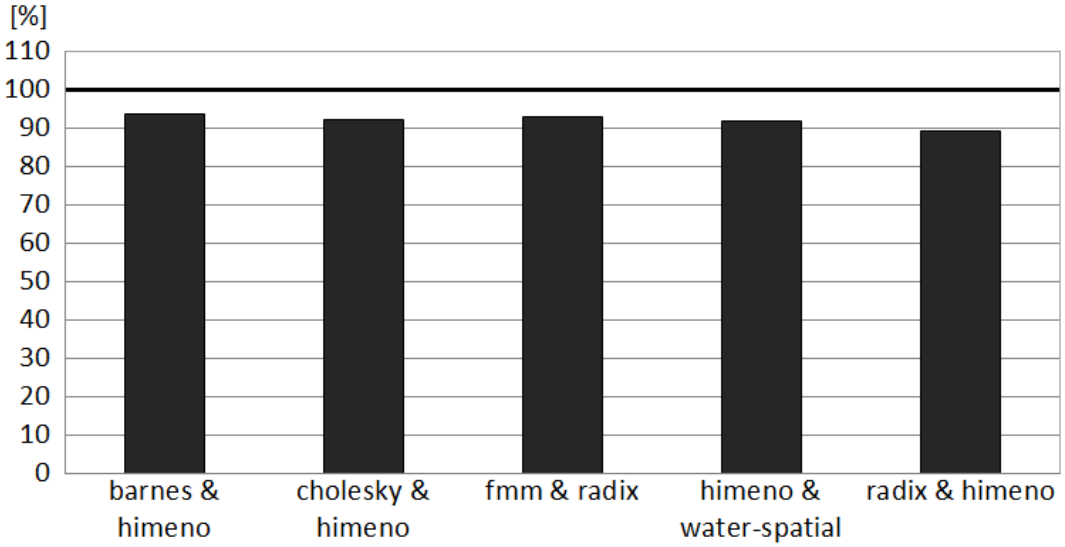

図 7.15: 実行サイクル比

時実行した場合は更なる性能向上が期待できる．

また，現在の手法では本来格納されるデータと異なるデータが格納されることで，データが存在しないのにキャッシュにアクセスし，ミスするという無駄な再アクセスの増加が考えられる．そこで，本来の格納先と異なるエントリに格納する際に，本来のデータの邪魔になりそうであれば書き込まないようにアルゴリズムを改良することで更なる性能向上が得られると考えられる．

8　あとがき

スマートフォンやパソコンなどのプロセッサでは，高性能化のために様々な手法が提案されており，その一つとして複数のコアを用意し，並列に処理を行うマルチコア化がある．しかし，マルチコア環境では複数のメモリアクセスが同時に発生することで，結果としてミス率が高くなるという問題がある．メモリアクセスは性能のボトルネックの一つであるため，高性能化を達成するためには主記憶へのアクセス数の低減，すなわちキャッシュミス率の低減が重要である．そこで筆者は，キャッシュをウェイより細かい領域である「セル」に分割して管理するセル・アロケーションキャッシュを提案している．しかし，セル・アロケーションキャッシュには，セルのアクセス制限によるミスの増加や，未使用セルの発生という問題がある．また調査の結果，連続したインデックスでヒット・ミスが連続することが分かった．そこで本研究では，セルのアクセス制限を優先度への変更，疑似的にウェイを増やし未使用なエントリを減らす手法，直近のアクセス履歴を用いた不要なデータの削除やセル割り当てを提案する．これらの手法を用いることにより，通常キャッシュと比較して，ミス率が平均 4.50%，実行サイクル数は平均 8.10%減少した．今後の展望として，データ書き込み先の入れ換えの制限や予測による無駄な再

アクセスの低減，セル割り当てアルゴリズムの改良が挙げられる．

謝辞

本研究の作成にあたり，ご指導を頂いた近藤利夫教授，佐々木泰敬助教授，深澤祐樹研究員にお礼申し上げます.

参考文献

[1] G. E. Sue, L. Rudolph, and S. Devadas, “Dynamic Partitioning of Shared Cache Memory,” Journal of Supercomputing, vol.28, No.1, pp.7-26, January 2004.

[2] 刀根舞歌，佐々木敬泰，深澤祐樹，近藤利夫，“キャッシュの分割領域の動的管理による高速化”，電子情報通信学会技術研究報告，Vol.116，No.177，pp.119-124，August，2016.

[3] 小寺 功消，“消費電力を考慮したウェイアロケーション型共有キャッシュ機構，” 情報科学技術レターズ，6 巻，p.55-58，August 2007.

[4] 小川周悟，入江英嗣，平木敬，“置換データの性質に着目した動的キャッシュパーティショニング”， 研究報告計算機アーキテクチャ (ARC)，2009-ARC-184(20)，pp.1-8，July 2009.

[5] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo, “Optimal Cache Partition-Sharing,” Parallel Processing (ICPP), 2015 44th International Conference on, pp.749-758, September 2015.

[6] 刀根舞歌，佐々木敬泰，深澤祐樹，近藤利夫，“可変レベルキャッシュのモード切り換えアルゴリズムの改良”，研究報告システム・アーキテクチャ(ARC)，Vol.2015-ARC-216，No.39，pp.1-8，August，2015.

[7] 姫野龍太郎，“姫野ベンチマーク”，<http://accc.riken.jp/supercom/himenobmt>，(2017 年 2 月 20 日アクセス).

[8] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. “The SPLASH-2 Programs: Characterization and Methodological Considerations,” In Proceedings of ISCA 22, pp.24-36, June 1995.