修士論文

CMPにおけるインデックスごとの有効利用ウェイ数を考慮した共有キャッシュ分割に関する研究

電気通信大学　大学院情報システム学研究科
情報システム基盤学専攻

0853013　鈴木 翔平

指導教員　本多弘樹
近藤正章
渡辺俊典

2010年 1月 28日 提出

目 次

1 はじめに 1
1.1 研究背景 1
1.2 研究の目的 2
1.3 論文の構成 3

2 従来の共有キャッシュ分割方式 4
2.1 一般的なキャッシュメモリ 4
2.2 分割の必要性 6
2.3 従来の共有キャッシュ分割方式 7
2.3.1 UCP における分割割合の決定方法 8
2.3.2 UCP の詳細 10
2.3.3 UCP における L2 キャッシュ内のブロックの管理 15

3 セットごとの有効利用ウェイ数を考慮した共有キャッシュ分割 17
3.1 従来手法の問題点 17
3.2 セットごとの分割割合決定 19
3.3 分割機構のデータ削減 25
3.4 分割割合の精度向上 27
3.4.1 カウンタ値の一時保存 27
3.4.2 分割割合決定サイクルの短縮 28

4 評価 29
4.1 評価環境 29
4.2 評価指標 31
4.3 セットごとに分割割合を決定することの有効性の確認 32
4.4 サンプルタグ領域のセット数を変化させた場合の評価 33
4.5 カウンタ値を一時保存する領域のデータサイズを変化させた場合の評価 34
4.6 分割割合の更新間隔を変化させた場合の評価 35
4.7 ハードウェアオーバーヘッドの評価 36

5 関連研究 38

6 終わりに 40
6.1 まとめ 40

6.2 今後の課題 40

参考文献 42

図 目 次

1 共有キャッシュ内のブロックの状態 6
2 再利用率の高いデータが多く残るように分割割合を決定する 7
3 対象とする CMP モデル 8
4 キャッシュ内における各コアのブロック占有状況 9
5 利用可能ウェイとキャッシュヒット回数の関係 10
6 UCP のシステムの概観 11
7 キャッシュヒット回数を L2 キャッシュで計測することの問題点 12
8 L2 キャッシュへブロックを挿入する場合 13
9 L2 キャッシュへアクセスする場合 13
10 スタックディスタンスプロファイルによって得られた利用可能ウェイ数とキャッシュヒット回数の関係 14
11 UCP におけるブロックの管理 15
12 各セットにおける利用可能ウェイ数とキャッシュヒット回数の関係 18
13 提案手法の概略図 19
14 従来手法におけるカウンタとサンプルタグ領域の構成 20
15 提案手法におけるカウンタとサンプルタグ領域の構成 20
16 全てのセットにおいて分割割合を個々に決定する手法の流れ 21
17 分割割合の決定 22
18 提案手法におけるブロックの管理 23
19 占有可能ブロック数 > 実際に存在するブロック数の場合 23
20 占有可能ブロック数 <= 実際に存在するブロック数の場合 24
21 追い出せるブロックが存在しない場合 24
22 セット数を制限する手法における分割割合決定の流れ 26
23 カウンタ値の一時保存 27
24 各セットにおける分割割合の更新間隔が長くなる 28
25 評価ベンチマークの有効利用ウェイ数のばらつき 31
26 全てのセットの分割割合を一度に決定する方法の性能評価 33
27 全てのセットの分割割合を一度に決定する方法の性能評価 34
28 全てのセットの分割割合を一度に決定する方法の性能評価 35
29 分割割合の更新間隔を変化させた場合の性能評価 36

表 目 次

1 ヒットカウンタ 14
2 評価対象の主なの構成 30
3 分割機構に用いるデータの内訳 37

1 はじめに

1.1 研究背景

近年，CPU の性能・機能は著しい向上を遂げてきた．これは半導体製造技術の進歩によるチップの微細化，高速化が急速に進んだことによりチップ上に多くのハードウェア資源（演算ユニット，レジスタファイル，キャッシュメモリなど）を集積することが可能になったためであり，現在の１チップ上のトランジスタ数は 10 億個レベルに達しようとしている [1].

このような中で，CPU は動作周波数と IPC（Instruction Per Cycle: 1 サイクルで実行できる命令数）の向上によるシングルコアプロセッサの性能向上が図られてきた．動作周波数の向上はパイプラインの細分化を進めることで，IPC の向上は命令レベル並列性（Instruction-Level Parallelisum:ILP）を高める out-of-order 実行とそれに付随する様々な高速化技術を投入することでそれぞれ達成されてきたが，これら２つによる性能向上は様々な弊害を伴っていた．例えば動作周波数の向上はトランジスタのスイッチング数増加による消費電力をと発熱量の増大を招き，ILP の向上はスケジューリング制御を複雑化させたことによる回路の配線遅延の増大を招いた．また近年は回路の微細化によるリーク電流の増加も大きな問題となっている．

トランジスタ数の増加・動作周波数の向上・ILP の向上が進めば進ほどこれらの弊害は大きな問題となるため，ここ数年はシングルコアプロセッサの性能向上は頭打ちになっており，より効率のよい仕組みが求められるようになった．

このような現状を打開するためのアプローチの１つとして登場し，現在注目を集めているのが，一つのチップ上にプロセッサコアを複数搭載する Chip-Multi Processor(もしくはマルチコアプロセッサ．以下 CMP) である [2]．CMP は複数のプロセスやスレッドを各コアで独立して実行することでシステム全体の処理能力を向上させるものである．従来のシングルコアプロセッサが ILP を高めることで処理能力を向上させるのに対し，CMP ではスレッドレベル並列性（TLP:Thread-Level Parallelism）を高めることで処理能力を向上させる。

CMP では，CPU 全体でコアが一つの場合より多くのメモリアクセスが発生するため，コア一つあたりのメモリバンド幅が相対的に縮小する．そのため CMP では，コアが一つの場合と比べ，性能低下を避けるためにオンチップ・キャッシュメモリのヒット率がさらに重要となる．

一般に CMP ではメモリに近い低次かつ大容量のセットアソシアティブキャッシュを複数コアが共有する。これはコア間でキャッシュを共有することで，一つのコア

が大容量のキャッシュを利用できることや，コア間で共通のデータを使用する際にデータのコピーを作らずに済むため無駄無くキャッシュを利用できるといった利点があるためである．一方で，扱うデータがコア間で異なりまたそれぞれのデータを全て格納できるほど共有キャッシュのサイズが大きくない場合，コア間でアクセス競合が発生する．これによって参照頻度の高いデータが参照頻度の低いデータに置き換えられたり，多くのデータを必要とするコアのデータでキャッシュが占有されその他のコアのデータがキャッシュに残らないといった状況が発生してしまい、キャッシュミスが増加してしまう．さらにチップ上のコア数の増加に比例してミスも増加する．

共有キャッシュにおけるコア間の競合ミスを避けるため，共有キャッシュの領域を分割し各コアが利用できる領域に制限を加える手法が提案されている [4][5][6]．これらの手法では共有キャッシュ内の各ブロックに対して占有させるコアを決め，他のコアによるブロックの追い出しをさせないことで競合ミスを防止する．

しかしこれらの手法では分割割合を「キャッシュ全体をどのような比率で分割するか」ということのみに着目して決めており，またキャッシュ全体における各コアのブロック数が決められた分割割合と等しくなるようにブロックの管理を行うため，キャッシュ内の各セットにおいて各コアが占有するブロックの割合がどのようになっているかは把握していない．このため，キャッシュ全体では決められた分割割合になったとしても各セットにおいてアクセス競合や効率の悪い追い出しブロックの選択等が行われてしまう可能性がある．

以上のことから，既存研究における共有キャッシュ分割の手法では各セットにおいて最適な分割割合が存在し，かつ実際にセットに存在する各ブロックの割合が最適な分割割合と異なっていてもシステムはそれを自発的に改善することができないため，結果として性能低下を招く恐れがある．

1.2 研究の目的

本研究では，これまでの共有キャッシュ分割方式が考慮していなかったセットごとの最適な分割割合に着目し、セットごとに異なる分割割合を設定する方式を提案する．各コアにおける共有キャッシュのキャッシュヒット回数をセットごとに調べ，それぞれに異なる分割割合を定めることで従来手法からの更なる性能向上を目指す．

また分割割合を決定するための情報はアプリケーションの実行とは無関係であり，データサイズはできる限り小さいことが望ましい．既存手法 [5] ではキャッシュヒット回数を計測するインデックス数を制限することでデータサイズの縮小を図ってい

る．提案手法においてもセット数の制限によって少ないデータサイズで実装可能なシステムを構築する．

1.3　論文の構成

本稿の構成は以下の通りである．2 章ではキャッシュの基本とこれまで提案されてきた共有キャッシュ分割の方式について述べる．3 章ではセットごとの有効利用ウェイ数を考慮した共有キャッシュ分割について述べる．4 章では提案手法についてシミュレーションを行い，その評価を示す．5 章では 2 章で述べた以外の共有キャッシュ分割方式について関連研究を述べる．6 章ではまとめと今後の課題について述べる．

2　従来の共有キャッシュ分割方式

本章では，キャッシュメモリの基本的事項と共有キャッシュ分割方式の従来手法について説明する．

2.1　一般的なキャッシュメモリ

キャッシュメモリとはCPUとメインメモリとの性能差を隠蔽するためにCPU内に搭載されるオンチップメモリのことであり，メインメモリと同様にアドレスによってデータを引く．一般的にキャッシュメモリはメインメモリより容量が小さいため，効率良く利用できるようデータ格納構造・データ置換アルゴリズムの工夫やキャッシュメモリの階層化がなされている．以下にデータ格納構造，データ置換アルゴリズム，キャッシュメモリの階層化について説明を行う．

- データ格納構造：
  キャッシュメモリはデータをあるまとまった単位で管理しており，この単位データをブロック，その大きさをブロックサイズと呼ぶ．データのアクセス要求があった時にはそのデータがキャッシュに存在するか，するならばどのブロックかを瞬時に検索する必要がある．そのためメモリアドレスの一部を用いてある程度の格納場所を限定することで検索速度を高めている．この格納場所をセット，各セットのアドレスをインデックスと呼ぶ．また１つのセットに格納するブロック数によって３つの方式に分けられる．
  - ダイレクトマップ方式：
    １つのセットに１つのブロックのみを格納する方式．メモリアドレスによって格納位置が一意に定まるため構成が簡単であるが，同一セットに異なるアドレスが転送されると必ずブロックの入れ替えが発生するため，キャッシュヒット率は高くない．
  - セットアソシアティブ方式：
    １つのセットに複数のブロックを格納する方式．参照アドレスの上位ビットを，タグと呼ばれるキャッシュメモリの各ブロックに格納されたメインメモリの上位ビットと比較することでキャッシュヒット/ミスの判定を行う．１つのインデックスに格納できるブロック数をウェイ数と呼び，ウェイ数がNの場合，Nウェイセットアソシアティブと呼ぶ．

- フルアソシアティブ方式：
  セットが1つのみで、キャッシュ内すべてのブロックが検索対象となる方式．ブロックの入れ替えが発生しにくく利用効率の点からは理想的だが，実装コストや複雑さの面から通常は用いられない．

- データ置換アルゴリズム：
  キャッシュに新規データを格納できる空のブロックが無い場合，格納するためには既存データと新規データを置き換える必要がある．ダイレクトマップ方式ではデータの格納場所はメモリアドレスによって一意に定まるため置き換える既存データも一意に定まるが，セットアソシアティブ方式とフルアソシアティブ方式ではセットに複数のブロックが存在するため，置き換え候補のデータも複数存在する．どのデータを残すかはキャッシュヒット率に大きな影響を与えるため，いくつかのデータ置換アルゴリズムが考えられている．

  - 千里眼置換アルゴリズム：
    今後長期に渡って必要とされないデータを常に置換対象にするアルゴリズム．最も効率的だが，あるデータを将来に渡ってどれだけ必要としないかを予測することは不可能であり，一般的に実装することは不可能である．

  - First In First Out アルゴリズム（FIFO）：
    最古に格納されたデータを置換対象とするアルゴリズム．各データのアクセス頻度を考慮しないため，キャッシュヒット率は高くない．

  - Least Recently Used アルゴリズム（LRU）：
    最古にアクセスされたデータを置換対象にするアルゴリズム．セット内の各ブロックについて，どの時点でアクセスされたかという情報を保持し，この情報によって置換対象データを選択する．置換対象のデータが格納されたブロックをLRUブロックと呼び，逆に一番最近にアクセスされたデータが格納されたブロックをMRUブロックと呼ぶ．一般的なキャッシュメモリではこの方式もしくはこの方式を一部変更したものを使用している．

- キャッシュメモリの階層化：
  CPUの処理速度が向上するにつれてメインメモリとCPUの速度差が大きくなったため，キャッシュメモリを多段階層構造にすることでこの速度差を埋める試みが行われてきた．階層化されたキャッシュメモリはCPUに近い側から

L1 キャッシュメモリ、L2 キャッシュメモリと呼び，CPU に近い側を高次キャッシュ，メインメインメモリに近い側を低次キャッシュとも呼ぶ．

2.2　分割の必要性

本節では本研究が対象とする共有キャッシュを分割利用することの必要性について説明する．一般的にキャッシュメモリのデータ置換アルゴリズムに LRU を用いることが多いが，共有キャッシュにおいて LRU を用いると各コアで実行するアプリケーションのアクセス頻度によって占有領域が決まってしまうという問題がある．例として異なる変数に次々アクセスしキャッシュアクセス頻度の高いアプリケーションを実行するコア A と，A で実行されるアプリケーションよりもアクセス頻度は低いもののデータの再利用率が高いアプリケーションを実行するコア B が，ブロック数 8 のフルアソシアティブキャッシュを共有する場合を考える（図 1）．

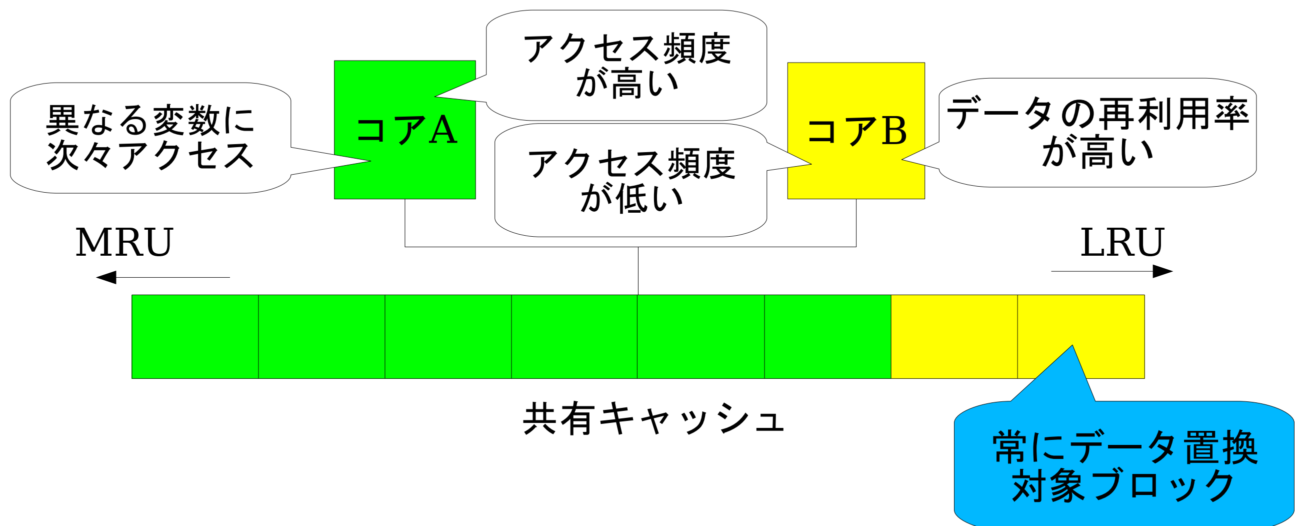

図 1: 共有キャッシュ内のブロックの状態

LRU ではキャッシュ内の各ブロックがどちらのコアのデータを格納しているかといった情報を持たない．したがってどちらのコアからのアクセスであるかを区別せず LRU 情報の更新を行う．したがって，このような場合ではアクセス頻度の高いコア A のブロックがデータ置換対象ブロックになりにくい MRU 側に集中し，逆にコア B のブロックは LRU 側に集中してしまいデータ置換対象になりやすくなってしまう．結果としてキャッシュ内には再利用率の低いコア A のブロックが多く残ることになり，キャッシュを有効利用することができない．

この問題を解決するためのアプローチの一つが，共有キャッシュを分割し，各コアが占有できる領域を制限する方法である．これによりキャッシュ内に再利用率の高いデータを多く残すことで，キャッシュを利用効率を高めることができる（図 2）．次節では従来の共有キャッシュ分割方式について説明を行う．

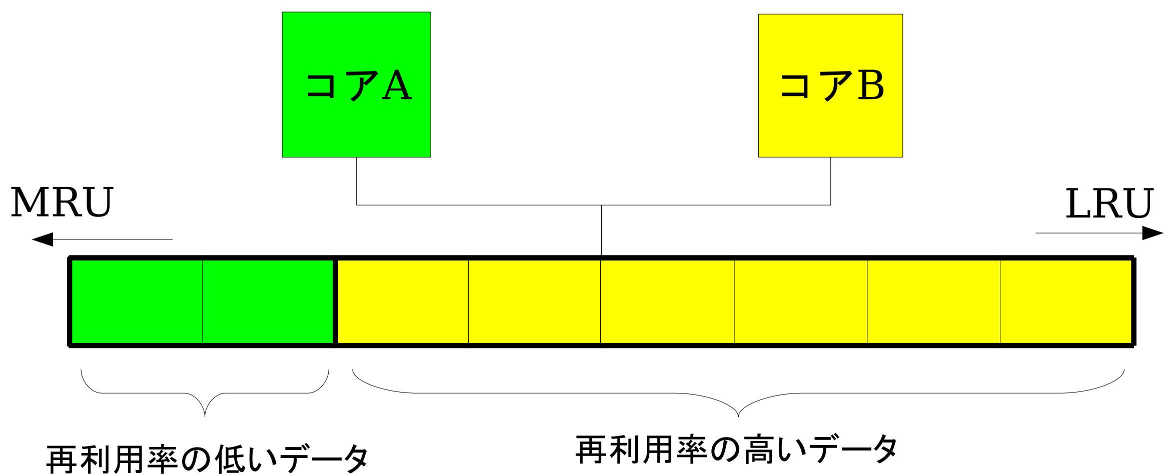

図 2: 再利用率の高いデータが多く残るように分割割合を決定する

2.3　従来の共有キャッシュ分割方式

後述する提案手法と密接に関わるため，従来手法として Utility-Based Cache Partitioning(以下 UCP)[5] について詳細に説明する．

なお本論文で対象とする CMP モデルを図 3 に示す．本 CMP は 2 つのプロセッサコアを搭載し，それぞれのコアはセットアソシアティブ方式の非共有型 L1 命令キャッシュと L1 データキャッシュを有する．またチップ上にセットアソシアティブ方式の共有 L2 キャッシュを有し，キャッシュ分割は L2 キャッシュにて行われる．各コアとの接続は共有バスを想定する．またそれぞれのコアでは独立したプログラムを同時に実行し，コア間で共有するデータは存在しないものとする．

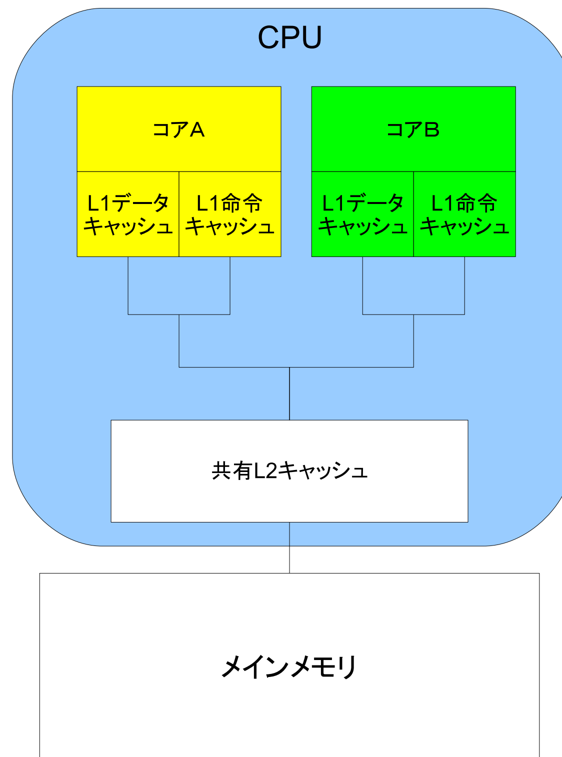

図 3: 対象とする CMP モデル

2.3.1　UCP における分割割合の決定方法

UCP では，共有 L2 キャッシュにおけるキャッシュヒット回数が最も多くなるように分割割合を決定する．キャッシュヒット回数が多くなればキャッシュの利用効率が高まり，プロセッサ全体を考えたときに最も性能向上が期待できるためである．

各コアは占有できる領域をブロック単位で割り当てられるが，UCP では L2 キャッシュの 1 ウェイ分に相当するブロック数を一単位として割り当てられる．したがって N ウェイセットアソシアティブの L2 キャッシュにおける各コアが占有可能なブロック数の割合は以下のように決められる．

$$コア\ A\ の占有ブロック数 : コア\ B\ の占有ブロック数 = X : N - X (1 \leq X < N) \quad (1)$$

各コアに対して 1 ウェイ分に相当するブロック数を一単位として割り当てるのは，L2 キャッシュがセットアソシアティブ方式であり，後述する各コアにおける L2 キャッシュでのキャッシュヒット回数の測定がウェイ単位で行われるためである．また L2 キャッシュ内では決められた分割割合によって各コアのブロック数が調整されるが，キャッシュ全体における各コアが占有するブロック数の比が決められた分割割合に

なればよい．したがってL2キャッシュ内においては各セットにおける各コアが占有するブロックの数は様々である．例としてL2キャッシュがウェイ数8，セット数4で分割割合がコアA：コアB=3：5だった場合のL2キャッシュ内における各コアのブロック占有状況の一例を示す．

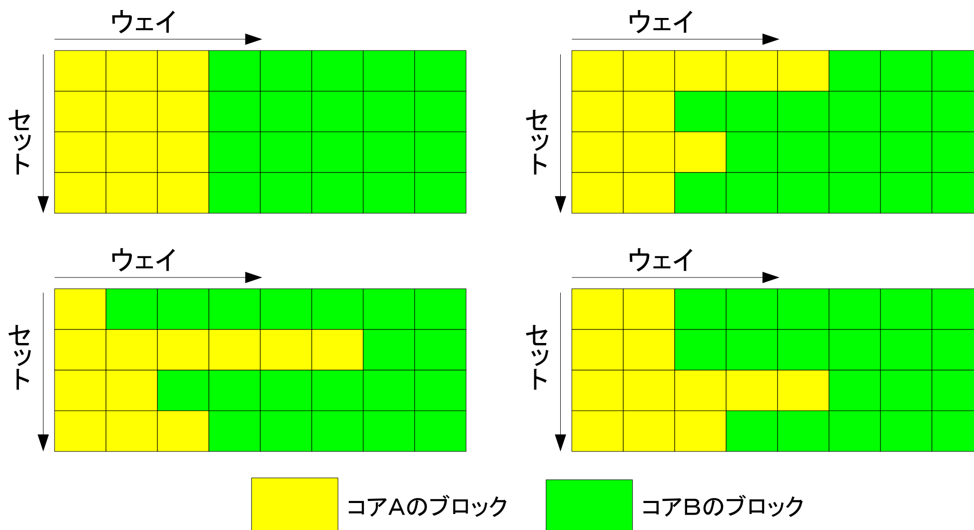

図 4: キャッシュ内における各コアのブロック占有状況

分割割合は各コアにおいてL2キャッシュの利用可能領域を制限した場合に，キャッシュヒット回数がどの程度減少するかを調べることによって決定される．各コアで実行されるアプリケーションをそのアプリケーション単独で実行する場合を想定し，L2キャッシュの利用可能領域をウェイ単位で制限していき，利用可能領域の大きさとキャッシュヒット回数の関係を調べる．そして分割割合として定義したときに最もキャッシュヒット回数が多くなる割合が選択される．

UCPにおける分割割合決定方法の例として，vprとequakeという2つのベンチマークをサイズが1MB，ウェイ数が16の共有L2キャッシュを持つCMPで同時実行することを考える．予備実験として各ベンチマークを単独で実行し，またL2キャッシュの全セットにおいて利用可能なウェイ数を1ウェイから16ウェイまで変化させ，それぞれの場合についてキャッシュヒット回数を計測した．シミュレーションによって得られた結果を図5に示す（その他のパラメータについては4章で説明する）．

図5より，vprは利用可能ウェイ数が多ければ多いほどキャッシュヒット回数が増加するのに対し，equakeは利用可能ウェイ数が2ウェイ以上になってもキャッシュヒット回数が増加しない，つまりL2キャッシュが3ウェイ分あれば十分であるとい

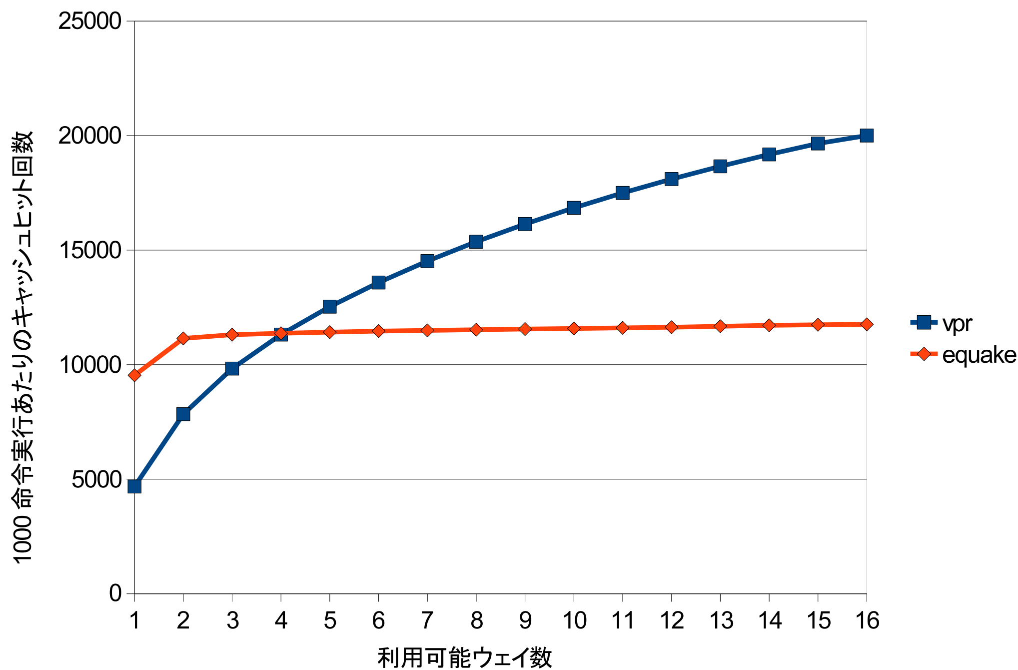

図 5: 利用可能ウェイとキャッシュヒット回数の関係

うことがわかる．したがってこの 2 つのベンチマークを同時実行する場合，equake の利用可能ウェイ数を 2 ウェイに制限し残りを vpr が使えるようにすればシステム全体として最もキャッシュヒット回数を多くすることができる. この例においては各ベンチマークを単独で実行して得られた情報を基に分割割合を考えたが，実際のシステムではアプリケーションの実行中に利用可能ウェイ数とキャッシュヒット回数の関係を調べる．次節では UCP のシステムについて詳しく説明する．

2.3.2　UCP の詳細

UCP は，2.3.1 で示した「各アプリケーションにおいて有効利用しているウェイ数」をアプリケーション実行中に調べ，最もキャッシュヒット回数が多くなるように共有キャッシュの領域分割を行う．そのために，各コアにおいて L2 キャッシュにおけるキャッシュヒット回数の計測および利用可能ウェイ数とキャッシュヒット回数の関係を調べるための仕組みが必要となる．そこで UCP では，キャッシュヒット回数の計測を各コアの外部に設置するサンプルタグ領域で行い，利用可能ウェイ数とキャッシュヒット回数の関係をスタックディスタンスプロファイル [7] と呼ばれる手法で調べる．キャッシュヒット回数の計測はウェイ数と同じ数のカウンタを用いて行われ，キャッシュヒットした場合はヒットしたブロックの LRU 情報と対応するカ

ウンタの値がインクリメントされる．図 6 に UCP の概観を示す．

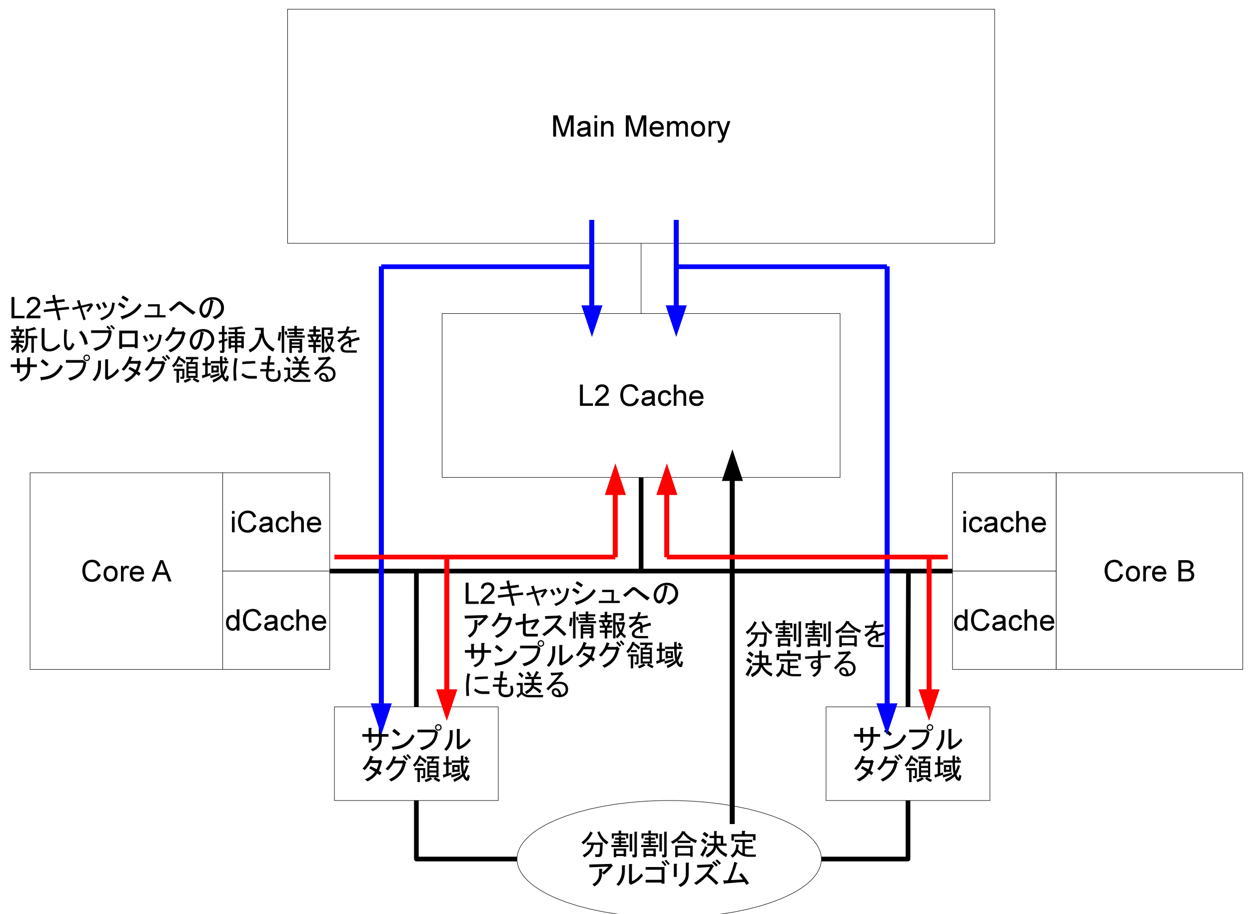

図 6: UCP のシステムの概観

サンプルタグ領域は，アプリケーションを同時に実行しながらも各アプリケーションが L2 キャッシュを単独で使用している状況を擬似的に表現するためのものである．L2 キャッシュにおいてキャッシュヒット回数を計測する方法も考えられるが，図 7 のように L2 キャッシュ内には各コアのブロックが存在しているため，LRU 情報は他のコアによるブロックの挿入・置換によって容易に変動してしまい正確なキャッシュヒット回数の計測をすることができない．

各サンプルタグ領域は監視するコアが決められており，監視対象コアによる L2 キャッシュの利用情報を用いる．構造としては L2 キャッシュとほぼ同様であるが，キャッシュヒット回数の計測のみを行うため，サンプルタグ領域の各ブロックにはデータ部分が存在せず、タグ情報のみを取り扱う．

メインメモリから L2 キャッシュへ監視対象コアの新しいブロックが挿入された場

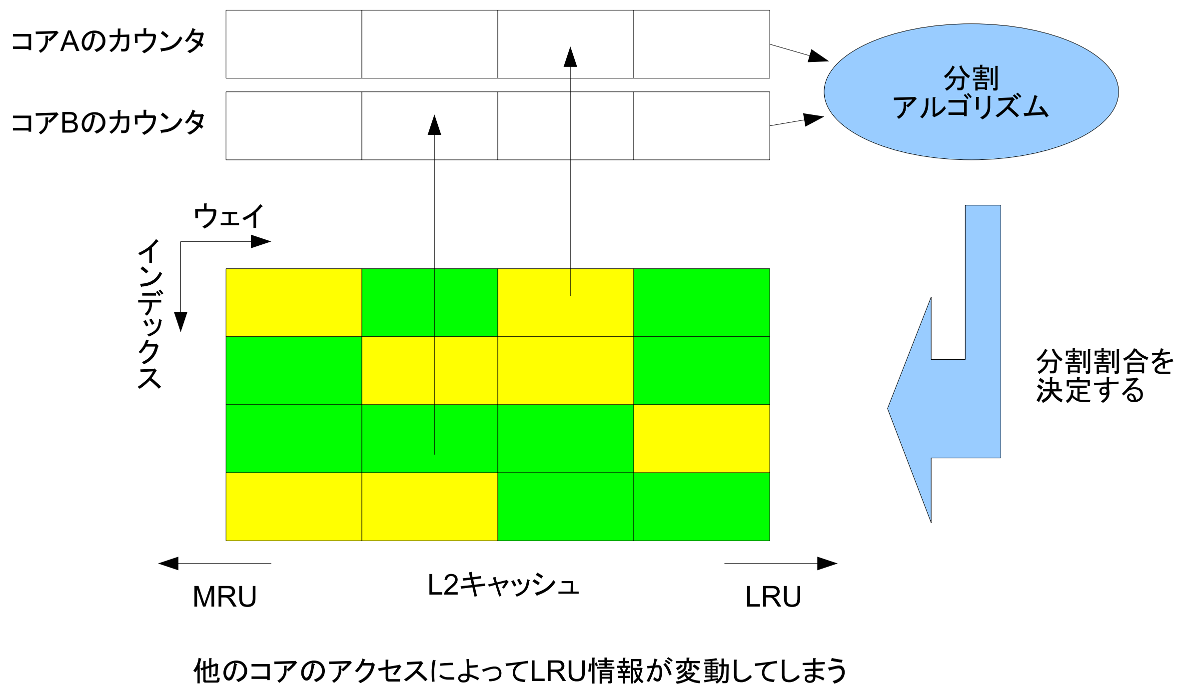

図 7: キャッシュヒット回数を L2 キャッシュで計測することの問題点

合，そのブロックのタグ情報のみを取り出し，サンプルタグ領域の該当するインデックスにおいてタグ情報の挿入および LRU による追い出しを行う (図 8)．また監視するコアから L2 キャッシュへのアクセスがあった場合は，キャッシュから直接情報を得る場合と同様の操作をサンプルタグ領域内で行う (図 9)．これによって，各コアが L2 キャッシュを単独で使用した状況を作って情報取得を行うことが可能になる．

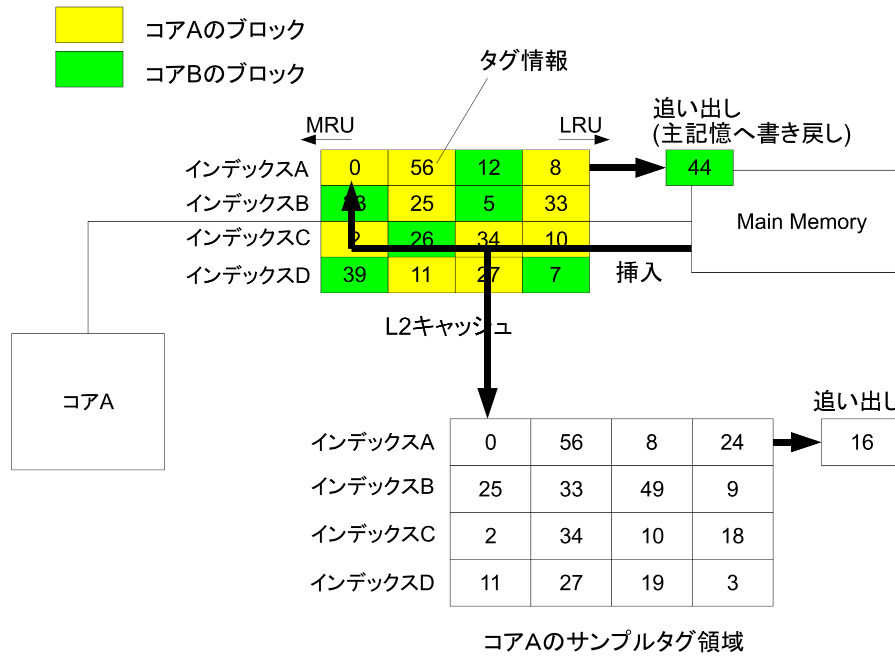

図 8: L2 キャッシュへブロックを挿入する場合

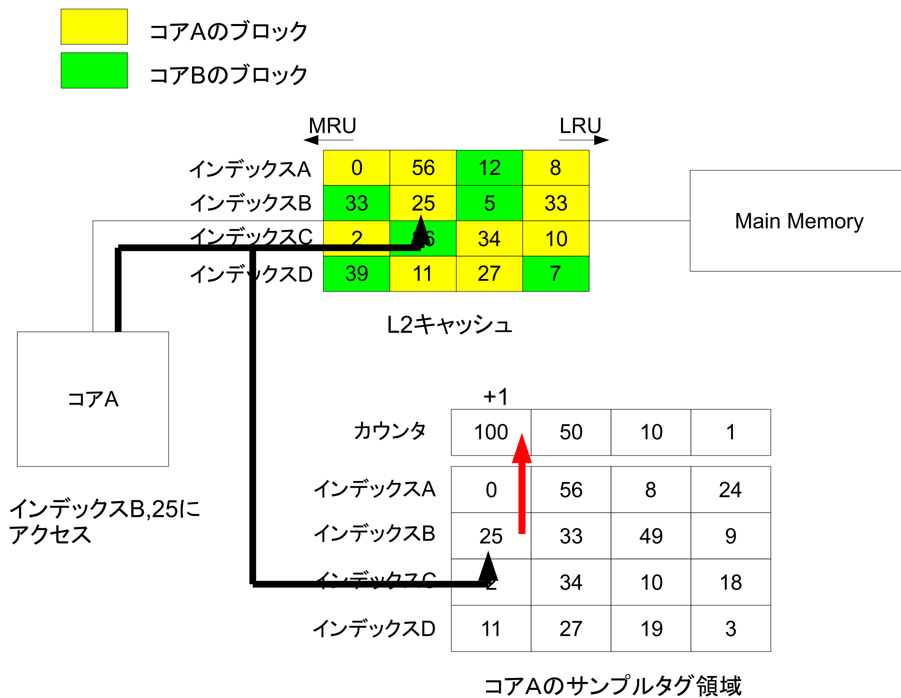

図 9: L2 キャッシュへアクセスする場合

サンプルタグ領域においてキャッシュヒット回数が計測されたのち，スタックディスタンスプロファイルによる利用可能ウェイ数とキャッシュヒット回数の関係を計算によって求める．例えば 4 ウェイの L2 キャッシュへのアクセスが 100 回行われ，その結果各カウンタの値が表 1 のようになったとする．

表 1: ヒットカウンタ

カウンタ番号	カウンタ値
1	30
2	20
3	15
4	10
キャッシュミス=25	

もし使えるウェイ数が 4 ウェイから 3 ウェイに減ったとすると，LRU 情報 4 でヒットしていたものが全てミスとなるため，減少するキャッシュヒット回数はカウンタ番号 4 の値，つまり 10 回となる．同様に 3 ウェイから 2 ウェイに減ると 15 回減り，2 ウェイから 1 ウェイに減ると 20 回減ることになる．このように考えることで，1 ウェイから 4 ウェイまで全ての場合について図 10 のように利用可能ウェイ数とキャッシュヒット回数を求めることができる．

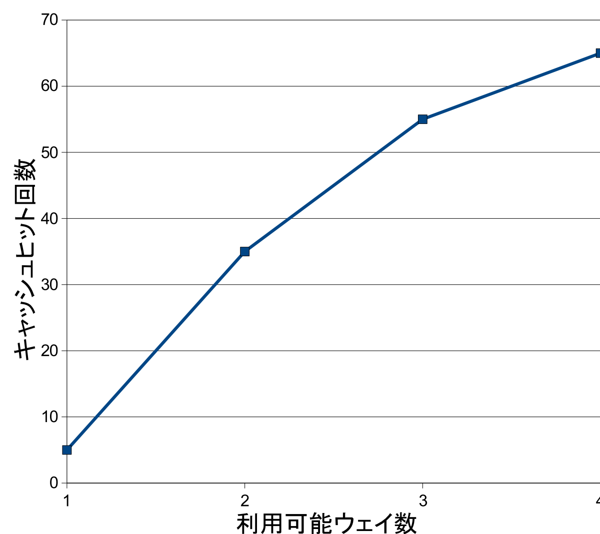

図 10: スタックディスタンスプロファイルによって得られた利用可能ウェイ数とキャッシュヒット回数の関係

2.3.3　UCP における L2 キャッシュ内のブロックの管理

L2 キャッシュの分割は，各コアのブロック総数の比が決められた分割割合と等しくなるように行われる．したがって，L2 キャッシュ内のブロックを管理するための仕組みが必要となる．

まず，L2 キャッシュ内に各コアのブロックがいくつあるかを管理するためのカウンタが設置される．そして各ブロックに対してどちらのコアのデータかを識別する識別ビットが付加される．

図 11 に UCP におけるブロック数の管理方法を示す．新しいブロックが L2 キャッシュに挿入された場合，どちらのコアのデータかを識別ビットに記録し，挿入された方のコアのブロック数を管理するカウンタがインクリメントされる．また L2 キャッシュからブロックが追い出された場合，追い出されたブロックの識別ビットからどちらのコアのブロックが追い出されたかを判別し，追い出された方のコアのブロック数を管理するカウンタがデクリメントされる．

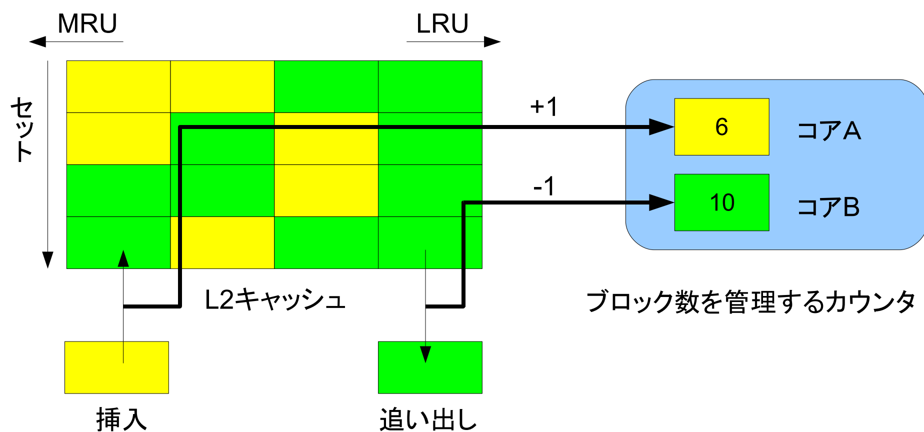

図 11: UCP におけるブロックの管理

L2 キャッシュ内の全てのブロックにデータが入っている場合，新しいブロックが挿入されたら既に L2 キャッシュ内に存在するブロックを追い出す必要がある．したがってどのブロックを追い出すかを決定する必要がある．追い出すブロックの決定は「割り当てられたウェイ数 ×L2 キャッシュの総インデックス数（割り当てられた量）」とブロック数を管理するカウンタから得られる「実際に L2 キャッシュ内に存在するブロック数（実際の量）」との比較によって決定される．

- 割り当てられた量　＞　実際の量　の場合：
  単純な LRU によって追い出すブロックを決定する
- 割り当てられた量　＜　実際の量　の場合：
  新しく挿入されたブロックと同じ識別ビットを持つブロックの中で最も LRU ブロックに近いものを追い出す

もし追い出せるブロックが存在しない場合は単純な LRU によって追い出すブロックを決定する.

3　セットごとの有効利用ウェイ数を考慮した共有キャッシュ分割

本章では本研究の提案手法であるセットごとの有効利用ウェイ数を考慮した共有キャッシュ分割の実現方法について述べる. 各セットにおいて別々にキャッシュヒット回数と利用可能ウェイ数を調べ，セット１つにつき１つの分割割合を決定する.

3.1　従来手法の問題点

従来手法の問題点は，各セットにおいて最適な分割割合と異なる割合で各コアにブロックが割り当てられる状況が発生し，キャッシュミスが増加することである．2.3.1 節においてアプリケーションの種類によって有効利用しているウェイ数に違いがあることを示したが，キャッシュの各セットにおいて有効利用しているウェイ数が異なっていた場合，最適な分割割合はセットごとに異なるはずである．従来手法は各セットにおいて実際にどのような割合で各コアのブロックが存在しているかは把握していないため，各セットにおいて最適な分割割合と異なる割合で各コアにブロックが割り当てられたとしても，それを検知したり最適な分割割合と同じ割合になるように各コアのブロック数を調整したりすることができない.

上述の問題点は「アプリケーションを実行する際，キャッシュの各セットにおいて有効利用しているウェイ数が異なっている」ことが仮定となっているため，この仮定が妥当なものかどうかを確認する必要がある．そこでまず実験によって各セットにおいて有効利用しているウェイ数が異なっているか否かを確認した．この実験は利用可能ウェイ数とキャッシュヒット回数の関係をセットごとに調べ，各セットにおける利用可能ウェイ数とキャッシュヒット回数の関係をシミュレーションによって調べることにより，有効利用しているウェイ数がセットごとに異なっていることを確認するものである．なおシミュレータのパラメータは 2.3.1 節の実験と同様とした．結果を図 12 に示す．なお図 12 では代表として対象ベンチマークを vpr としセットのインデックスが 1, 100, 819, 968 のものを掲載する.

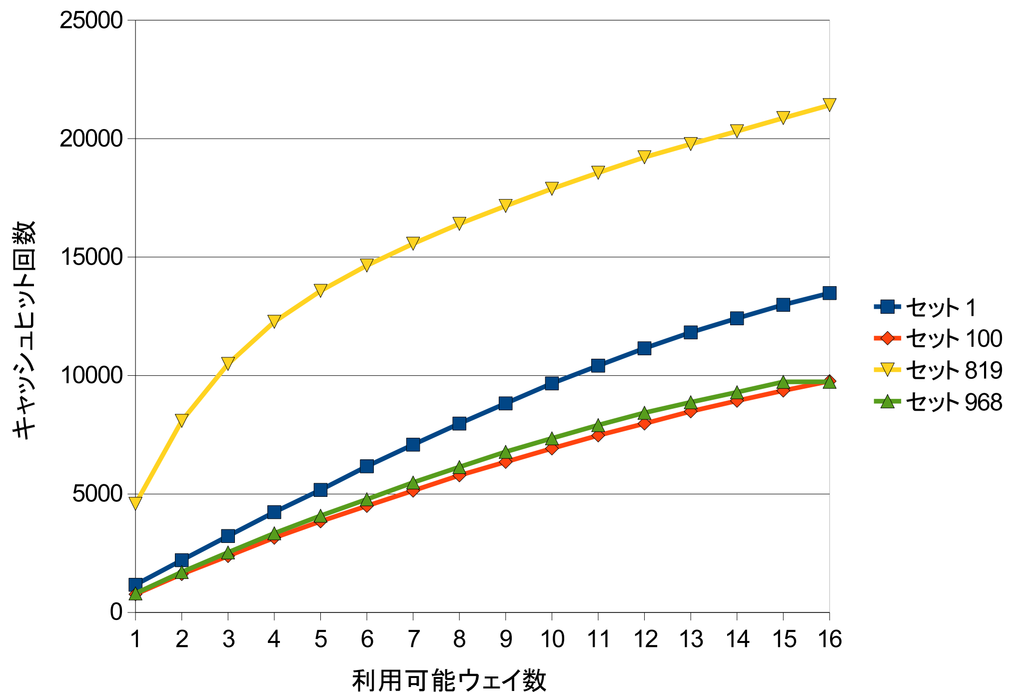

図 12: 各セットにおける利用可能ウェイ数とキャッシュヒット回数の関係

図 12 において，各セットではいずれも利用可能ウェイ数が増えれば増えるほどキャッシュヒット回数が増加しているが，その増え方はセットごとには様々である．またその他のベンチマークについても利用可能ウェイ数とキャッシュヒット回数の関係はセットごとに異なっていた．このことからアプリケーションを実行する際，キャッシュの各セットにおける有効利用しているウェイ数は異なっており，従来手法では各セットの有効利用ウェイ数の違いを考慮しない分割を行うためキャッシュミス回数が増加してしまう可能性があることが確認できた．

このような従来手法の問題点を踏まえ，本研究ではキャッシュヒット回数がセットごとに異なることを考慮にいれた共有キャッシュ分割方式を提案する．すなわち各セットにおいて利用可能ウェイ数とキャッシュヒット回数の関係を別々に調べ，**セットごとに異なる分割割合を設定する**ことで前述した従来手法の問題点を解決し，共有キャッシュのさらなる利用効率向上を図る (図 13).

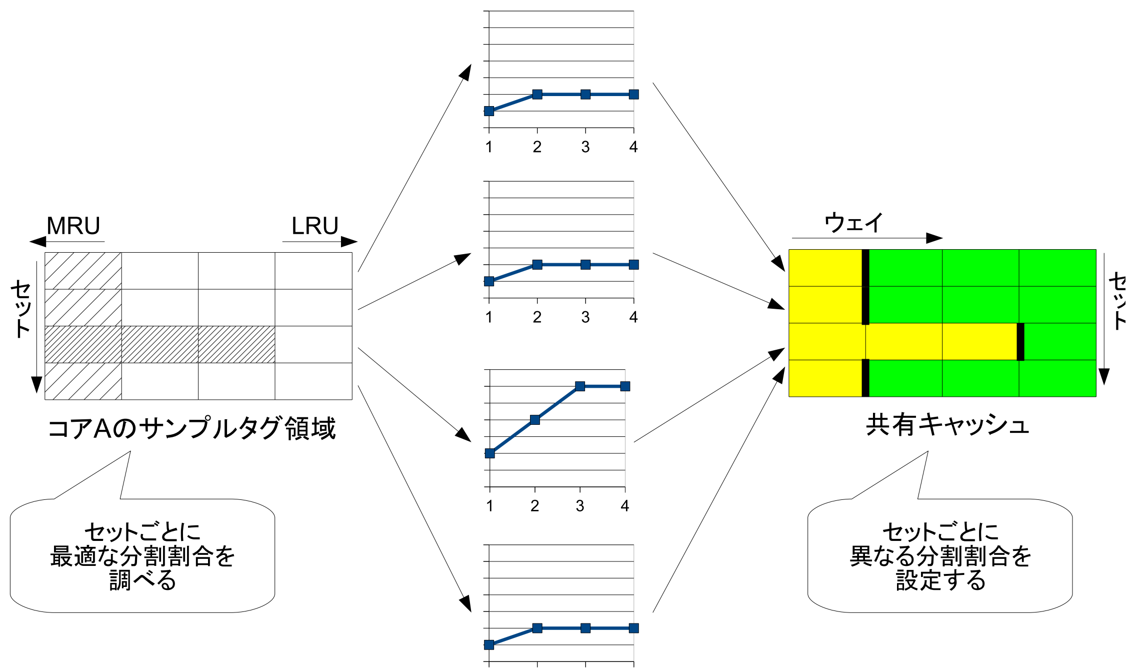

図 13: 提案手法の概略図

3.2 セットごとの分割割合決定

本研究の提案手法は UCP の拡張によって実現する．UCP においてはェイごとに全セットに対するキャッシュヒット回数の合計を計測していた．これに対し提案手法ではウェイごとのキャッシュヒット回数をそれぞれのセットごとに計測する．そしてセットごとに最適な分割割合を求め，決定する．

提案手法の単純な実現方法として，全てのセットにおいて各ウェイに対応するカウンタを設置し，各セットにおいて個別にキャッシュヒット回数の計測および利用可能ウェイ数とキャッシュヒット回数の関係を調べる手法が考えられる．この方法を採用した場合の従来手法からの変更点としては，従来手法では図 14 のように LRU 情報と対応するカウンタの値をインクリメントするが，提案手法では図 15 のようにセットのインデックスと LRU 情報が対応するカウンタの値をインクリメントすることになる．また分割割合を決定する際は，図 16 のように各セットにおけるキャッシュヒット回数を計測したカウンタの値を基に各セットの利用可能ウェイ数とキャッシュヒット回数の関係を調べ，1 つのセットにつき 1 つの分割割合を決定する．

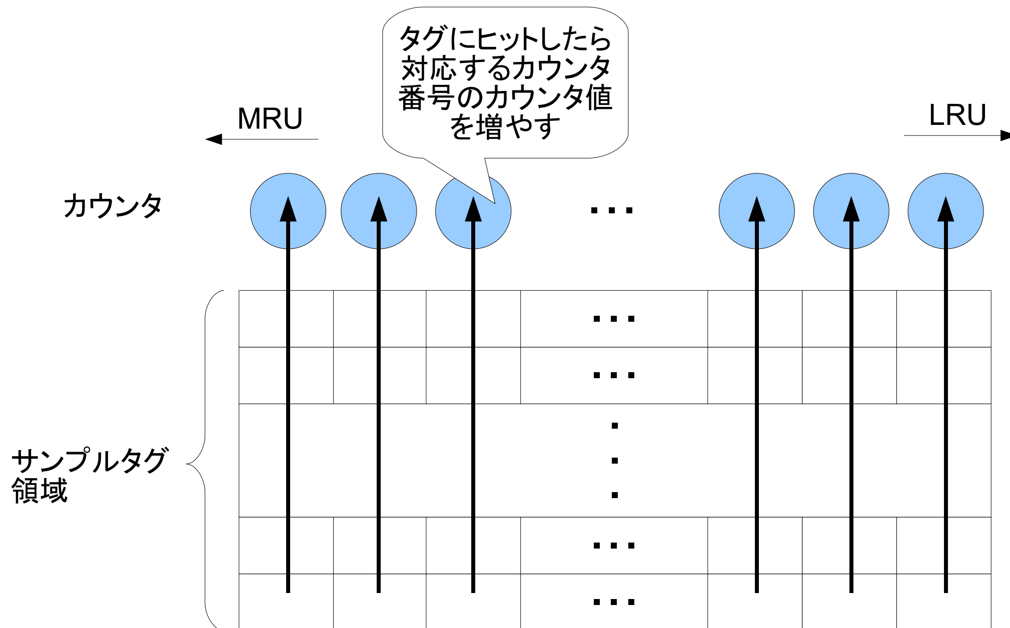

図 14: 従来手法におけるカウンタとサンプルタグ領域の構成

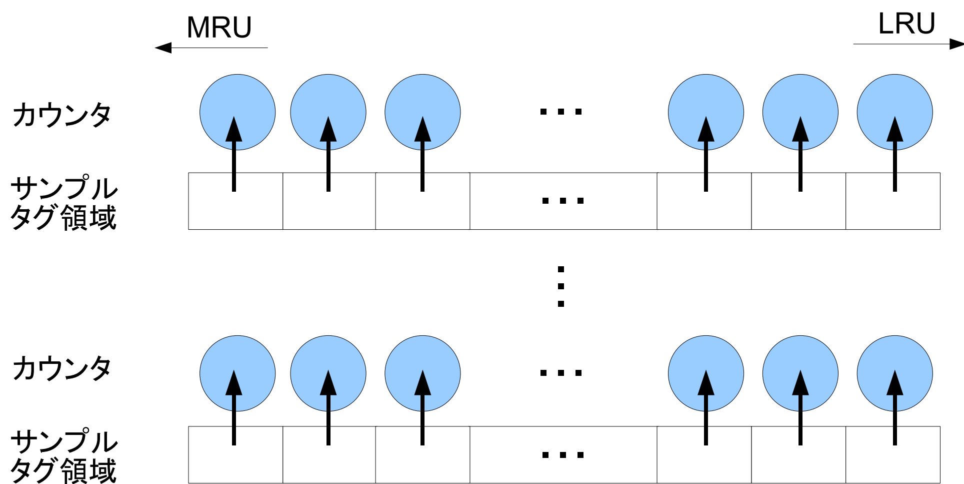

図 15: 提案手法におけるカウンタとサンプルタグ領域の構成

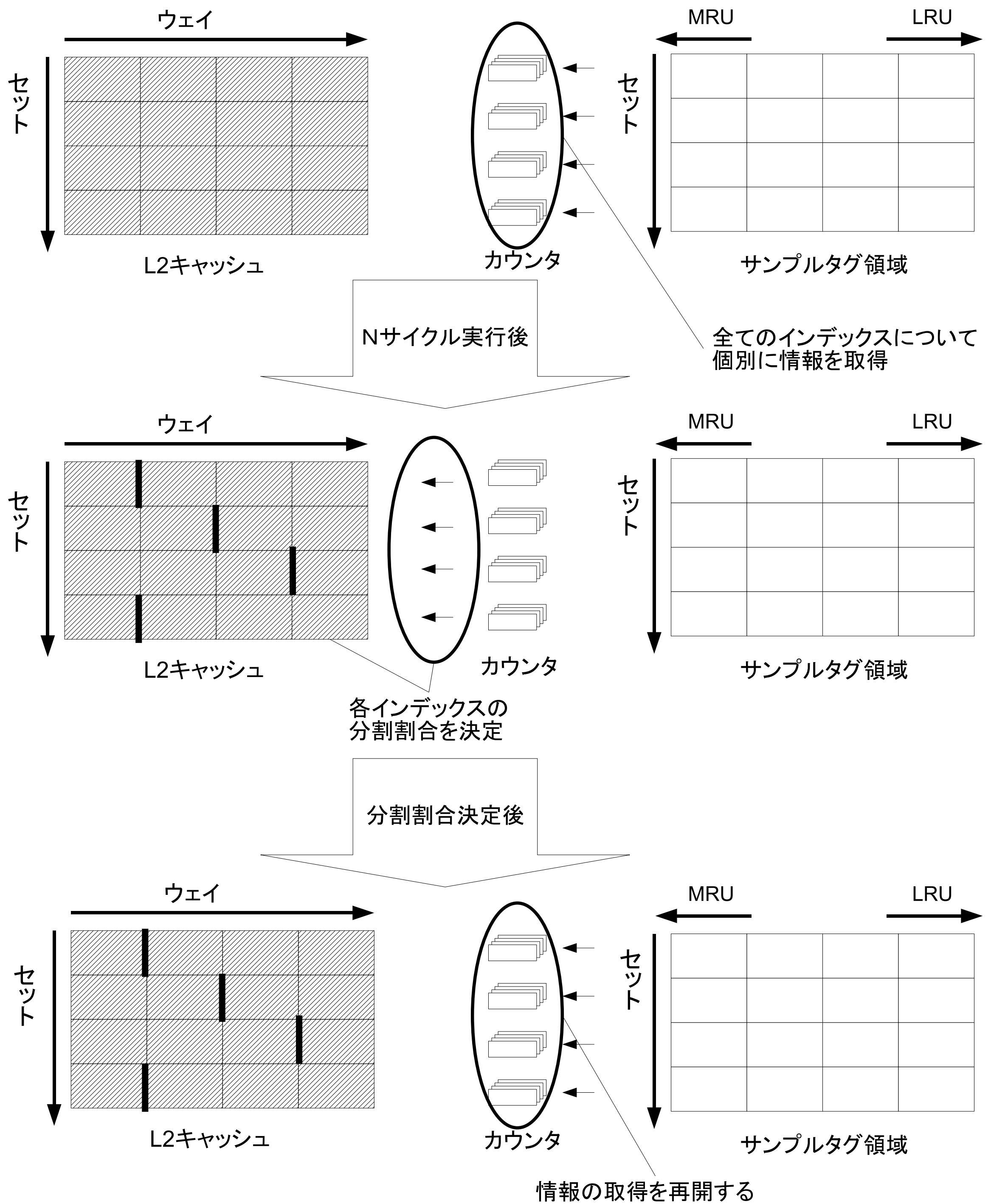

図 16: 全てのセットにおいて分割割合を個々に決定する手法の流れ

提案手法における分割割合の決定方法は UCP と同様である．しかし UCP ではキャッシュ全体の分割割合を決めるため最適な分割割合を調べる操作は一回で済むが，提案手法ではセットごとに異なる分割割合を決めるため最適な分割割合を調べる操作は共有キャッシュのセット数と同じ回数が必要である．ここで改めて分割割合の決定方法について詳述する．

キャッシュヒット回数の計測はヒットしたブロックの LRU 情報に 対応する番号のカウンタをインクリメントすることで行われる．したがって各カウンタには「MRU ブロックでヒットした回数」「MRU ブロックの次に MRU 側にあるブロックでヒットした回数」「その次に MRU 側にあるブロックでヒットした回数」といった情報が蓄積されることになる．そして分割割合を決定する際は全体で最もキャッシュヒット回数が多くなる割合を分割割合とする．例えば 4 ウェイのキャッシュを 2 コアで共有し，それぞれのコアにおけるキャッシュヒット回数が図 17 のようになったとする．この場合 4 ウェイなので 4 つのカウンタ値が最も大きくなる組合せが選択されるため，コア A:コア B ＝ 2:2 で分割することとなる．

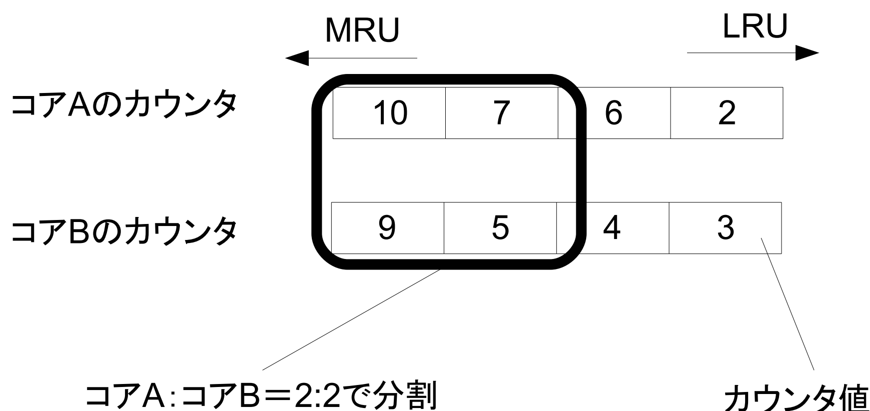

図 17: 分割割合の決定

また各セットにおける分割割合を決定することで，各セットにおいて各コアが占有可能なブロック数も決定する．このため，共有キャッシュ内で各コアが占有しているブロック数を管理する方法に変更が必要となる．2.3.3 節で従来手法におけるブロックの管理方法を述べたが，これは共有キャッシュ全体において各コアが占有しているブロック数を管理するものだった．提案手法では，図 18 のようにセットごとに各コアの占有ブロック数を管理する．

新しいブロックが挿入され既に存在するブロックを追い出す必要がある場合は，各

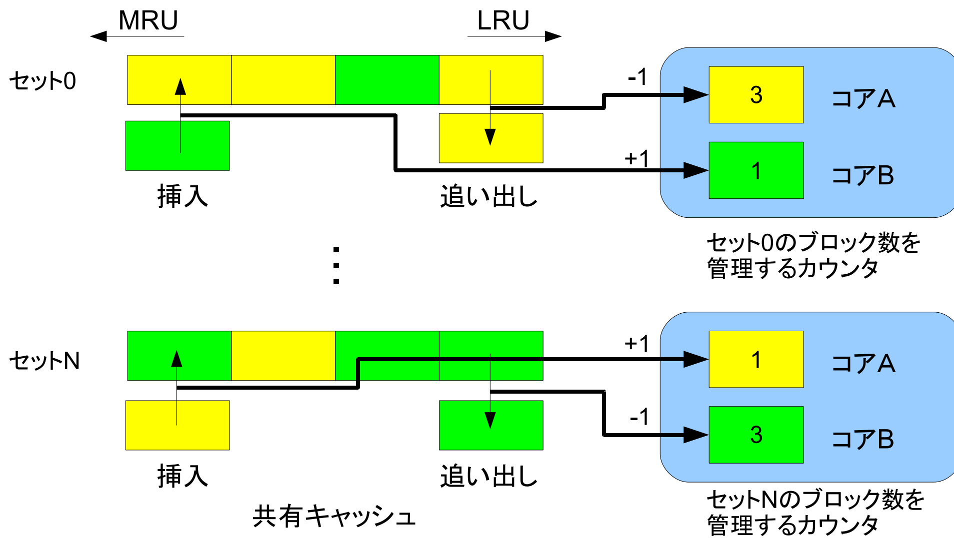

図 18: 提案手法におけるブロックの管理

セットにおける各コアが占有可能なブロック数と実際に存在するブロック数とを比較しどのブロックを追い出すかを決定する．

- **占有可能ブロック数** > **実際に存在するブロック数**の場合
  他のコアのブロックの中で最も LRU 側にあるブロックを選択する．図 18 は共有キャッシュのセット A において，コア A が占有しているブロック数が占有可能ブロック数より少ない状態でコア A のブロックを新しく挿入する場合を示している．このような場合はセット A に存在するコア B のブロックで最も LRU 側にあるブロックが追い出すブロックとなる．

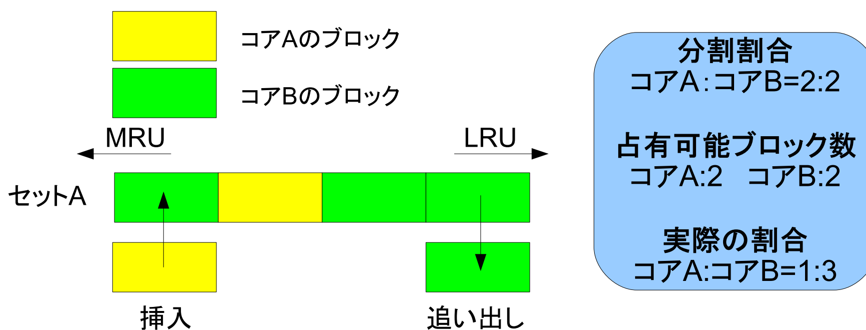

図 19: 占有可能ブロック数 > 実際に存在するブロック数の場合

- **占有可能ブロック数** <= **実際に存在するブロック数**の場合

挿入するブロックとコアが同じブロックの中で最も LRU 側にあるブロックを選択する．図 20 は共有キャッシュのセット A において，コア A が占有しているブロック数が，コア A が占有可能なブロック数と等しい状態で，ア A のブロックを新しく挿入する場合を示している．このような場合はセット A に存在するコア A のブロックで最も LRU 側にあるブロックが追い出すブロックとなる．

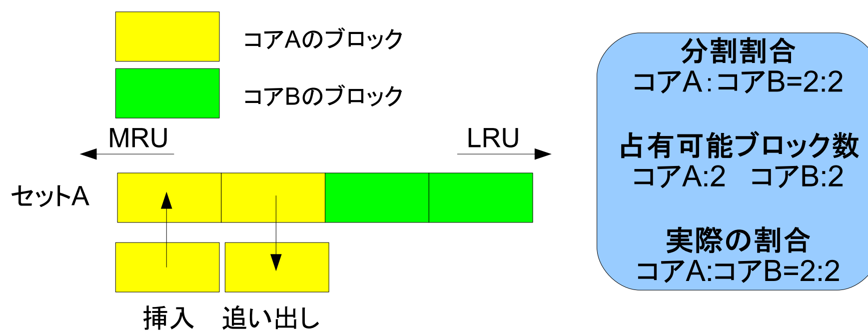

図 20: 占有可能ブロック数 <= 実際に存在するブロック数の場合

- **追い出せるブロックが存在しない**場合
  追い出そうとしたブロックが，データが上位キャッシュに存在したりデータの読み書き中などで追い出せない場合がある．その場合はそのセットの中で最も LRU 側にあるブロックを選択する．図 21 は図 20 とほとんど同じ状況であるが，コア A のブロックが何らかの理由で追い出すことができない場合を示している．このような場合はセット A に存在するブロックで最も LRU 側にあるブロック追い出すブロックとなる．

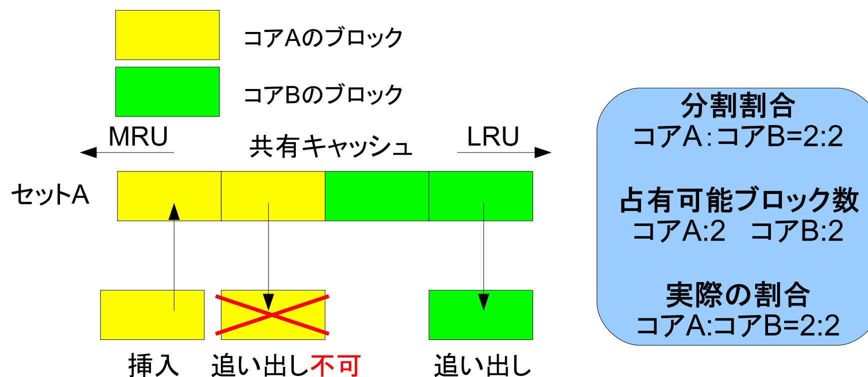

図 21: 追い出せるブロックが存在しない場合

3.3 分割機構のデータ削減

3.2 節で述べた手法では，共有キャッシュ内の全てのセットについて異なる分割割合を同時に決定する必要があるため，サンプルタグ領域とキャッシュヒット回数を計測に多くのデータを必要とするため，ハードウェアオーバーヘッドが大きくなってしまう．すなわちサンプルタグ領域のセット数は共有キャッシュと同じ数必要であり，キャッシュヒット回数を記録するカウンタは共有キャッシュの総ブロック数 (インデックス数 × ウェイ数) が必要となる．仮にこの手法をシステムとして実装した場合，共有キャッシュのサイズが 1MB，インデックス数が 1024，ウェイ数が 16 とすると，サンプルタグ領域とカウンタの合計データ量が約 128KB になってしまい，L2 キャッシュサイズの約 13%に達してしまう．サンプルタグ領域とカウンタはアプリケーションの実行とは関係無い部分，すなわち分割機構のオーバーヘッドであるため，一部のデータを削減しデータサイズを小さくする必要がある．

分割機構のデータ量を減らすため，サンプルタグ領域のセット数を制限する方法を考える．3.2 節で述べた手法では全てのセットにおいてキャッシュヒット回数の計測および分割割合の決定を一度に行ったが，本手法ではこれらの操作を行う対象のセットを一部のセットに限定し，また対象のセットにおいて分割割合が決定されたら別のセットを対象に同様の操作を行う．以下に本手法による情報取得方法および分割割合決定の手順を示す．

1. 共有キャッシュにおいて分割割合を定めるインデックスの範囲を決める．
2. 決められたインデックスにおいてキャッシュヒット回数の計測を始める．
3. 一定のサイクル (N サイクルとする) の間アプリケーションを実行し，キャッシュヒット回数を計測する．
4. N サイクル経過した後，決められたインデックスにおいて分割割合を定める．
5. サンプルタグ領域およびカウンタ値を初期状態 (すべてゼロ) に戻す．
6. 分割割合を定めるインデックスの範囲を前回と異なる場所にする．
7. 2 に戻る．

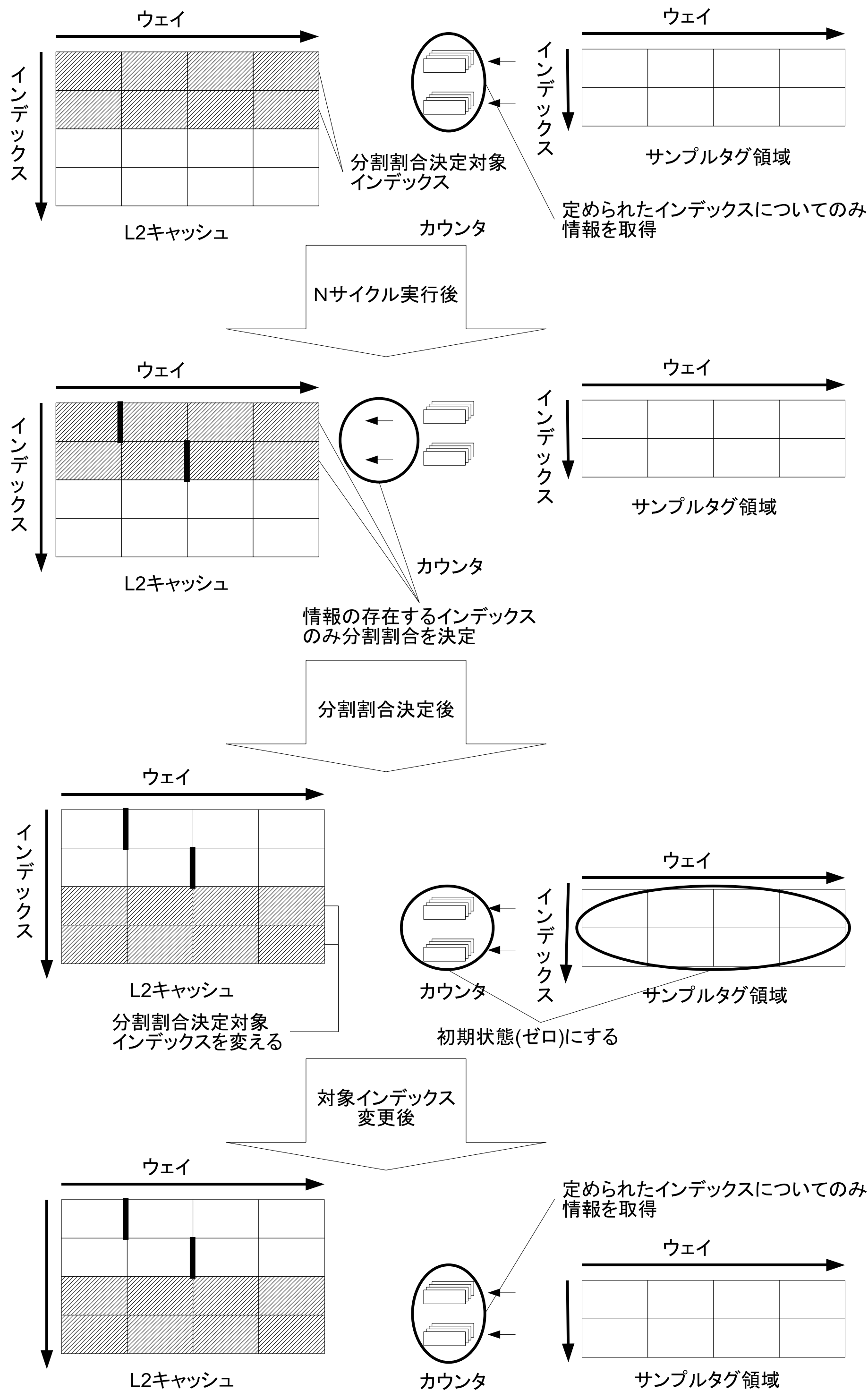

図 22: セット数を制限する手法における分割割合決定の流れ

3.4 分割割合の精度向上

3.3 節の手法では，一度分割割合が決定されるとカウンタ値はゼロに戻すため，次回分割割合の決定が行われる際は前回の情報が全て失われてしまうことになる．したがって分割割合を決める際に十分な情報が得られず，各セットにおける最適な分割割合が得られない可能性がある．また分割割合の決定は決められたセット数ずつ行われるため，各セットにおいては分割割合が更新される期間が長くなってしまう．このためアプリケーションのデータアクセスパターンの変化に対応できない可能性がある．

そこでこれらの問題に対し，前者にはキャッシュヒット回数を一時保存するデータ領域を付け加えることで，後者には分割割合を決めるためのサイクルを短くすることで問題の解消を図る．

3.4.1 カウンタ値の一時保存

本研究の提案手法は，過去に得られたキャッシュヒット回数をもとに将来最もキャッシュヒット回数が多くなると予測される割合で分割割合を決定するため，過去に得られたキャッシュヒット回数が重要な情報となる．しかし 3.3 節の手法では各セットにおいて一度分割割合が決定されると，他のセットにおけるキャッシュヒット回数の計測のためにカウンタ値をゼロに戻さなければならないため，次に分割割合を決定する際は過去の情報が全て失われた状態で行わなければならない．このような状況になることを防ぐため，前回分割割合の決定を行うセットに指定された時に得られたキャッシュヒット回数を別のデータ領域に一時的に保存し，次回のキャッシュヒット回数の計測を保存された値から始めるようにする．これによって過去に得られたキャッシュヒット回数を失わずに分割割合を決定することが可能となる．

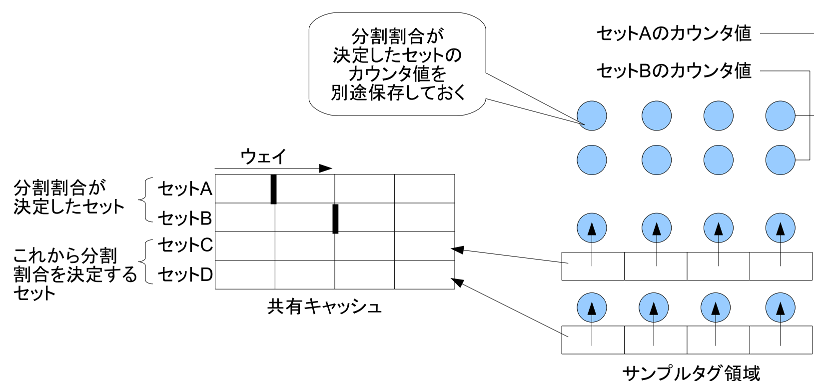

図 23: カウンタ値の一時保存

3.4.2 分割割合決定サイクルの短縮

一度に分割割合を決定するセット数を制限することで，各セットにおける分割割合の更新時間が相対的に長くなる．例えばキャッシュの総セット数の 1/2 をサンプルタグ領域のセット数にすると，サンプルタグ領域のセット数を制限しない場合と比べて各セットの分割割合の更新期間は 2 倍になる．

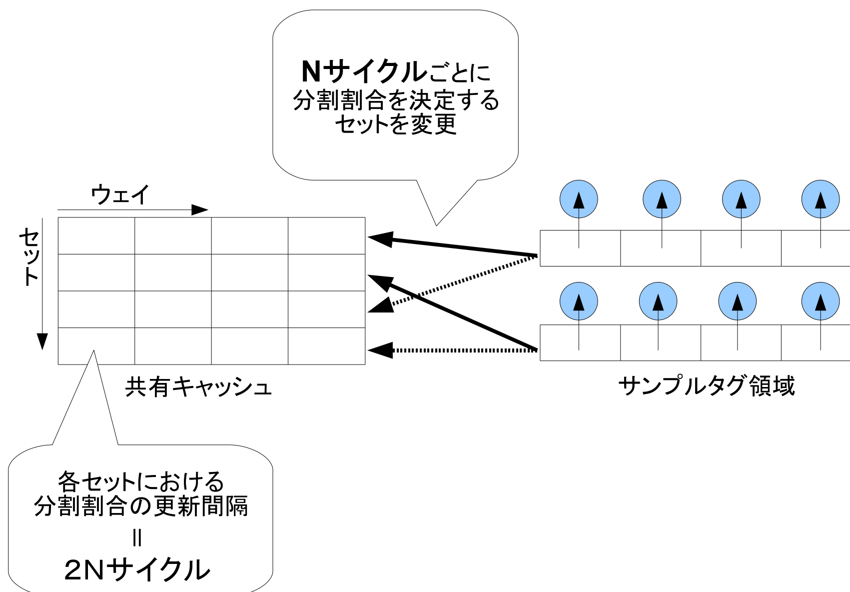

図 24: 各セットにおける分割割合の更新間隔が長くなる

こうなるとアプリケーションのデータアクセスパターンが変化し最適な分割割合が変化したとしても各セットにおける分割割合を変えることができないため，望ましくない割合での分割をしなくてはならない．このような状況を防ぐため，キャッシュヒット回数を計測し分割割合を決定する一連のサイクルを短くする．これによってアプリケーションのデータアクセスパターンの変化に柔軟に対応することが可能となる．

4　評価

本章ではシミュレーションにより提案手法，UCP，および単純な LRU の比較を行い，提案手法の有効性を示す．

4.1　評価環境

シミュレーションにはミシガン大学で開発されたプロセッサシミュレータ M5 2.0[8] を使用する．M5 は共有キャッシュ型 CMP の評価が可能なもので，初期状態のキャッシュの利用方法として単純な LRU が実装されている．これに UCP および提案手法の仕様に従って変更を加えたものを使用することで、単純な LRU，UCP，提案手法の比較を行う．

UCP の実装は 2.3 節で説明した内容に従って行った．なお文献 [5] を参考に，サンプルタグ領域のセット数を 32 個に限定し，キャッシュヒット回数を計測するセットは任意に選ばれたものとする．また分割割合の更新は 500 万サイクルごとに行われることとした．それ以外の LRU から変更する点は以下の通りである．

- サンプルタグ領域およびキャッシュヒット回数を計測するカウンタの設置
- L2 キャッシュ内の各ブロックに対して識別ビットの付加
- L2 キャッシュ内の各コアのブロック総数を管理するカウンタの設置
- 追い出しブロックの選択を 2.3.3 節で説明したように変更

また提案手法においては，まず全てのセットにおいて一度にキャッシュヒット回数の計測と分割割合の決定を行う手法を実装し，セットごとに分割割合を決めることの有効性を確認する．次に 3.2 節と 3.3 節で述べたサンプルタグ領域のセット数を制限する方法および分割機構のデータ量を削減する方法を実装し，性能がどのように変化するかを調査する．

対象とする CMP は 2.3 節で示した図 3 であり，その他のパラメータを表 2 に示す．L2 キャッシュのサイズが 1MB と CMP に搭載されるキャッシュのサイズとしては小さいが，これは各手法の影響が顕著に出るようにするためである．また L1 キャッシュのサイズも小さいが，これは L1 キャッシュのサイズが大きすぎると L1 キャッシュで多くキャッシュヒットしてしまい L2 キャッシュをあまり使わないという状況になってしまうためである．

表 2: 評価対象の主なの構成

項目	設定値
コア数	2
命令発行方式	アウトオブオーダ
命令発行幅	8/clock cycle
命令デコード幅	8/clock cycle
キャッシュ構造	サイズ，ウェイ数，ブロックサイズ，レイテンシ
L1 命令キャッシュ	16kB，2，64B，1clock cycle
L1 データキャッシュ	16kB，2，64B，1clock cycle
L2 共有キャッシュ	1MB，16，64B，15clock cycle
L2 共有キャッシュセット数	1024
メインメモリレイテンシ	400clock cycle
L1-L2 間バス幅	32B
L2-メインメモリ間バス幅	8B

使用するベンチマークプログラムは SPEC 2000 の中からセットごとの有効利用ウェイ数にばらつきがあるもの 3 つ，ないもの 3 つの合計 6 つを選び，全ての組み合わせについて評価を行う．

またセットごとの有効利用ウェイ数にばらつきがあるかどうかを，以下の方法によって判定する．まず利用可能ウェイ数を 1 ウェイから最大ウェイまで変化させ，それぞれの場合において各セットのヒット率を求め，ヒット率とヒット回数の積を各ウェイ数における「有効利用ポイント」とする．次にキャッシュの全セットにおける利用可能ウェイ数を半分にした時の「全体のヒット率 × 全体のヒット回数/総セット数」をしきい値とし，各セットの有効利用ポイントがウェイ数をいくつ減らした時にそのしきい値を下回るかを調べる．そして下回るウェイ数+1 を「そのセットを有効に利用しているウェイ数」とする．

有効利用ウェイ数のばらつきの有無は以下の方法によって判定する．まず全てのセットにおいて有効利用ウェイ数を調べ，有効利用ウェイ数ごとにセット数を調べる．そしてセット数が最も多くなる有効利用ウェイ数において，セット数がキャッシュの総セット数の 50%以上になる場合をばらつきが無い，50%未満の場合をばらつきがあるとした．各ベンチマークの有効利用ウェイ数のばらつきを図 25 に示す．

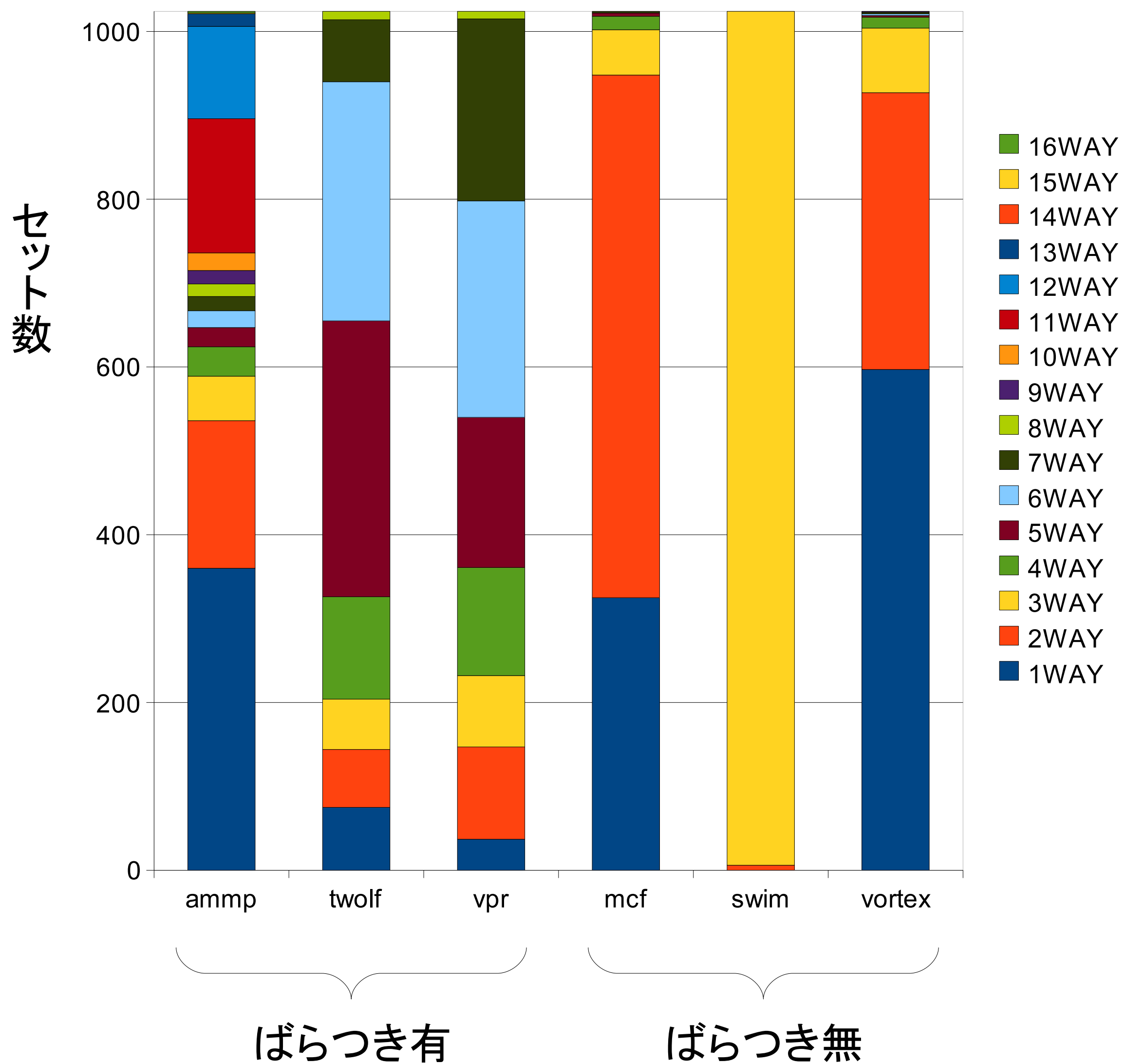

図 25: 評価ベンチマークの有効利用ウェイ数のばらつき

4.2　評価指標

評価指標は Weighted Speedup を用いる．Weighted Speedup は，CMP や SMT のような複数スレッド，複数プロセスを同時実行するプロセッサにおいて，システム全体の性能を評価するために Snavely らにより導入された評価指標である [9]．Weighted Speedup では同時実行するアプリケーションの IPC を，それぞれのアプリケーションを単独で実行したときの IPC によって重み付けをして評価する．こうすることで

単独で実行したときにIPCが高いアプリケーションと低いアプリケーションを同時実行した時の性能評価において，各アプリケーションのIPCの違いを考慮した評価が可能となる．

Weighted Speedupの式を以下に示す．

$$Weighted\ Speedup = \sum_{i=1}^{N}(IPC_i/Single\ IPC_i) \quad (2)$$

式(1)において，Nは同時実行するアプリケーションの数(=コア数)であり，$Single\ IPC_i$は各アプリケーションをシングルコアで単独実行した時のIPC，IPC_iは各アプリケーションを他のアプリケーションと同時に実行したときのIPCである．

4.3　セットごとに分割割合を決定することの有効性の確認

初めに，本研究の提案手法であるセットごとに分割割合を決定する手法によって性能向上が認められるかを評価する．提案手法におけるサンプルタグ領域のセット数をL2キャッシュと同じ数にして，全てのセットの分割割合を一度に決定する方法の評価を行う．なお提案手法における分割割合更新の間隔はUCPと同じ500万サイクルとした．図26に結果を示す．グラフの横軸はベンチマークの組み合わせを表し，縦軸はWeighted Speedupである．

図26より，提案手法では有効利用ウェイ数にばらつきがあるもの同士を同時実行した場合（ammp+twolf〜twolf+mcf）とばらつきがあるものとないものを同時実行した場合（twolf+swim〜vpr+vortex）において，ほとんどの組み合わせででUCPよりも高い性能となっており，提案手法によってUCPから平均3%，最高9%の性能向上を得られることがわかる．これは有効利用ウェイ数にばらつきがあるものを同時実行させることでキャッシュの各セットにおける最適な分割割合にもばらつきが出るため，セットごとに分割割合を決定することの利点が活かされた結果と考えられる．一方で，ばらつきがないもの同士を同時実行した場合(mcf+swim〜swim+vortex)では，UCPと同じもしくは低い性能になってしまっている．これは有効利用ウェイ数のばらつきが無いもの同士の組み合わせではセットごとの最適な分割割合にあまり違いが出ないため，セットごとに分割割合を決めることの効果が薄くなってしまうためと考えられる．

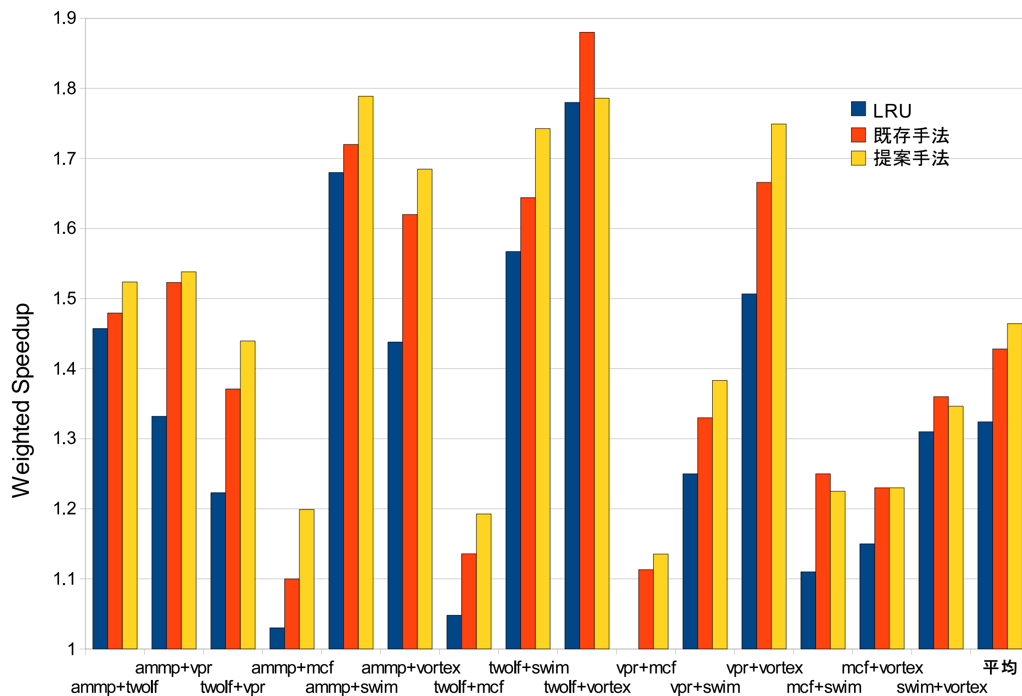

図 26: 全てのセットの分割割合を一度に決定する方法の性能評価

4.4 サンプルタグ領域のセット数を変化させた場合の評価

3.3 節で述べたサンプルタグ領域のセット数を制限する手法において，サンプルタグ領域のセット数を変化させた場合の性能評価を行った．サンプルタグ領域のセット数を 32，64，128，256，512 と変化させ，それぞれの場合において Weighted Speedup を測定する．

なお 3.4.1 節で説明したカウンタ値を一時保存するデータ領域として 4Byte 各セットの各ウェイに設置した状況で評価を行った．また 4.3 節で性能向上が見られなかった，有効利用ウェイ数にばらつきが無いもの同士の組み合わせは評価対象外とした．結果を図 27 に示す．

図 27 より，全ての組み合わせにおいてサンプルタグ領域のセット数が少なくなればなるほど性能が低下していることがわかる．これは分割割合を決定するセットが一部分に制限されることで他のセットにおいて分割割合の更新間隔が延びてしまうことと，各セットにおいてキャッシュヒット回数を計測できない時間が存在するため分割割合を決めるための十分な情報量が得られないことが原因であると考えられる．

またサンプルタグ領域のセット数ごとに見ると，32 の場合ではどの組み合わせにおいても著しく性能が低下してしまうが，64 以上からは 1024 の場合，すなわち L2

キャッシュのセット数とサンプルタグ領域のセット数が同じ場合とあまり性能が変わらなくなっており，サンプルタグ領域のセット数が 64 の場合では UCP から約 2%の性能向上を得ることができている．

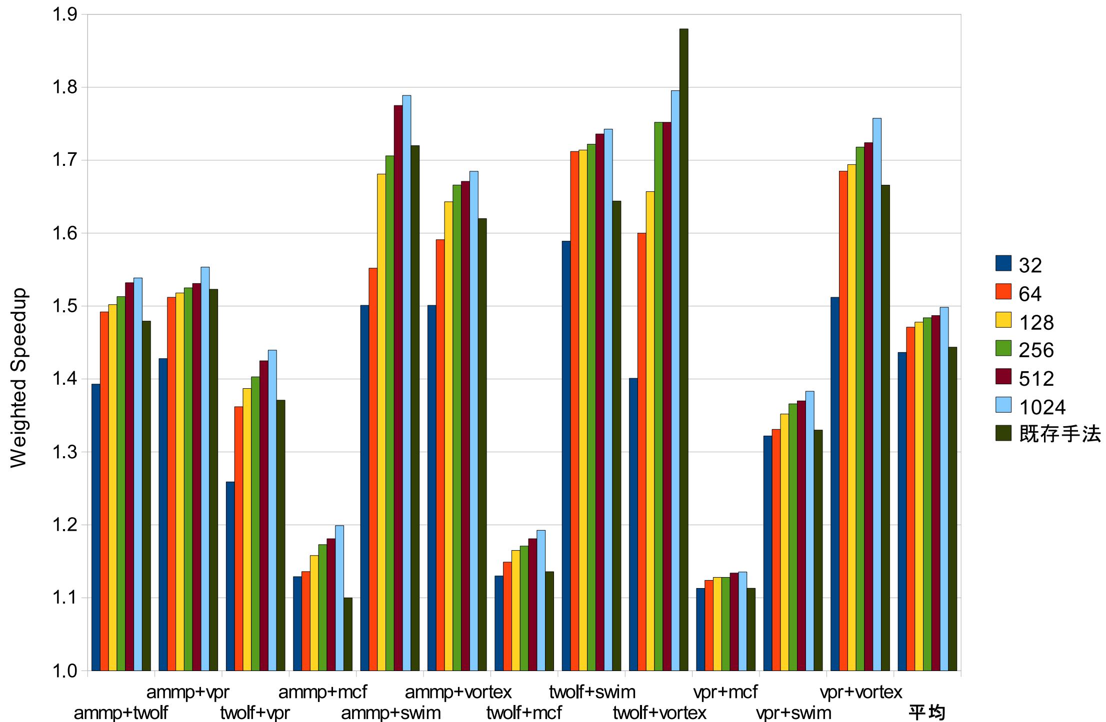

図 27: 全てのセットの分割割合を一度に決定する方法の性能評価

4.5　カウンタ値を一時保存する領域のデータサイズを変化させた場合の評価

4.4 節の評価では，カウンタ値を一時保存する領域のデータサイズを 4B としたが，この領域はサンプルタグ領域同様分割機構のオーバーヘッドであるため，なるべく小さいサイズにする必要がある．そこでカウンタ値を一時保存する領域のデータサイズを 1B，2B としたときに性能がどのように変化するかを調べた．データサイズを小さくすることでカウンタ値がオーバーフローする可能性があるが，そのような状況が発生したセットにおいては全てのカウンタ値を 1/2 にすることで対処した．またサンプルタグ領域のセット数は 64 とした．結果を図 28 に示す．

図 28 より，一時保存する領域のデータサイズが 1B の場合では性能が著しく低下するが，2B の場合では 4B の場合とほぼ変わらない性能を維持している．1B の場合

で性能が低下する要因としては，1B のデータ量では最高で 255 までしか計測できないため，カウンタ値のオーバーフローが頻繁に発生するためと考えられる．また 2B の場合であまり性能が低下しない要因としては，多くの場合でカウンタ値は 10000 前後までしか増えないため，サイズが 2B であっても余裕を持って計測することが可能であるためと考えられる．

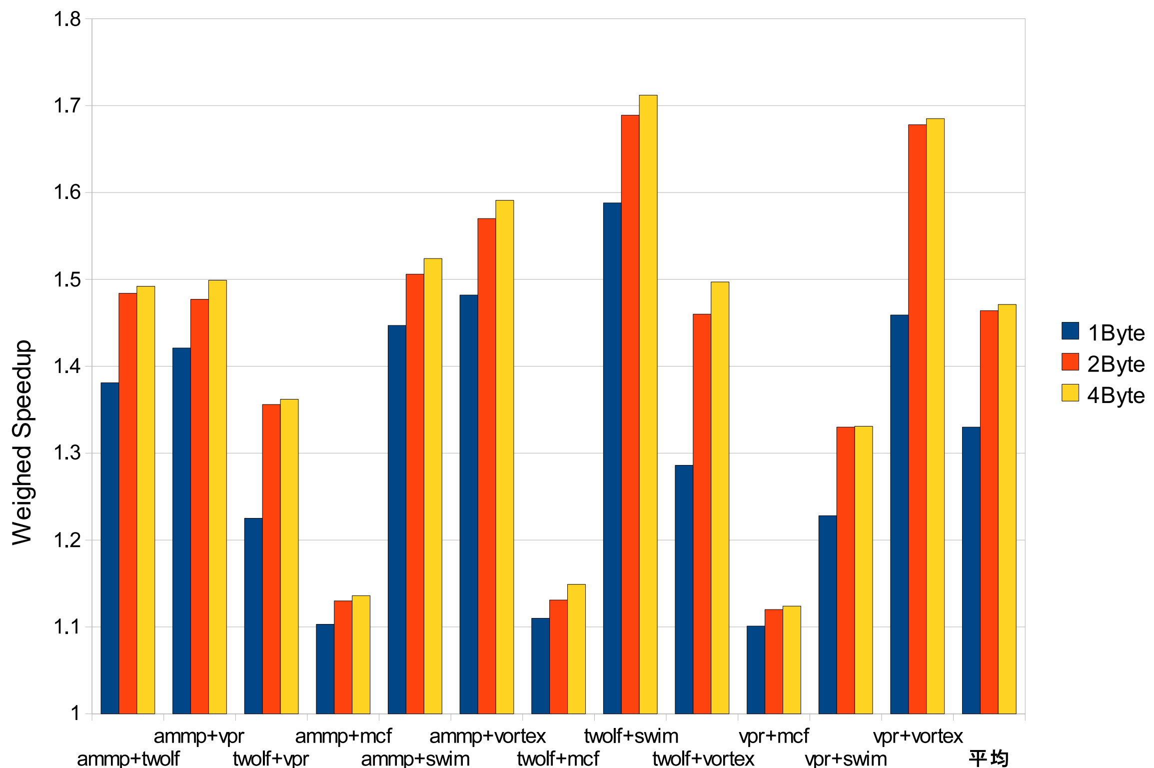

図 28: カウンタのデータサイズを変化させた場合の性能評価

4.6 分割割合の更新間隔を変化させた場合の評価

3.4.2 節で示した分割割合の更新間隔を短くする手法が性能向上につながるかを調べるため，分割割合の更新間隔を変化させて評価を行った．4.4 節，4.5 節における評価では分割割合の更新間隔を 500 万サイクルとしたが，これを 100 万サイクル，200 万サイクル，300 万サイクル，400 万サイクルと変化させた場合について Weighted Speedup を測定する．なおサンプルタグ領域のセット数は 64，カウンタ値を一時保存するデータ領域のデータサイズを 2B とした．結果を図 29 に示す．

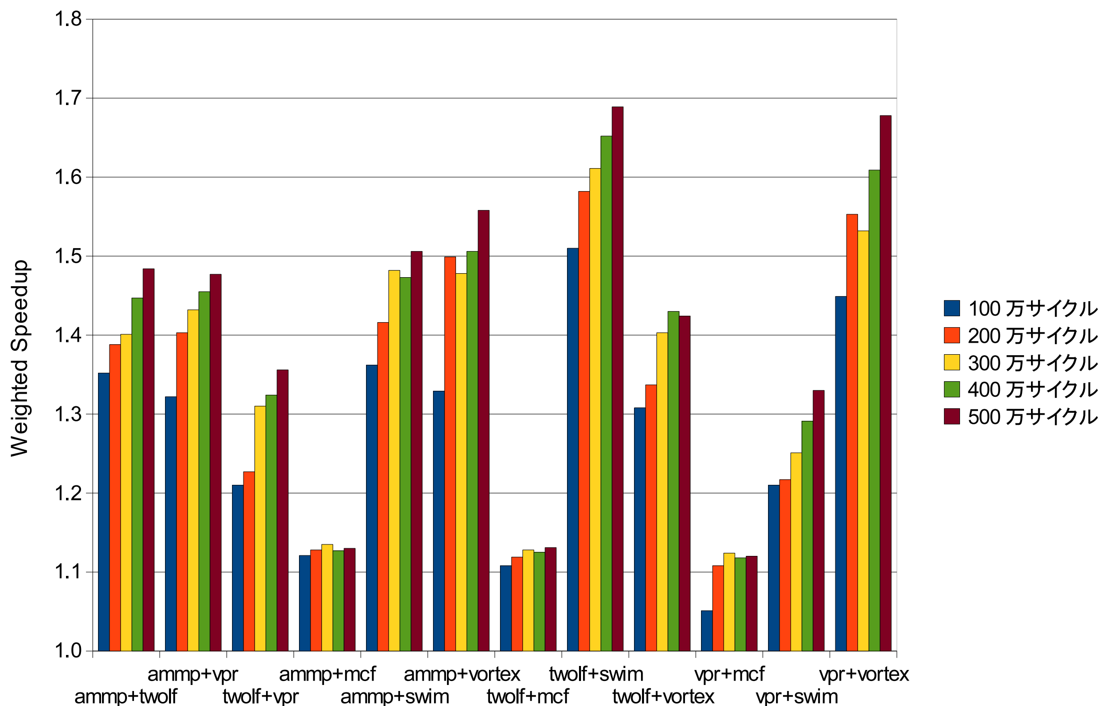

図 29: 分割割合の更新間隔を変化させた場合の性能評価

図 29 より，ほぼ全ての場合において分割割合の更新間隔を短くすればするほど性能が低下する結果となった．また一部の組み合わせ (ammp+mcf，twolf-vortex，vpr+mcf) においては更新間隔が 500 万サイクルよりも短い場合で性能が向上しているが，500 万サイクルとの違いはわずかである．性能が低下した原因として考えられるのは，分割割合の更新が終わった後にサンプルタグ領域を初期状態に戻すことでキャッシュヒット回数をうまく計測できなくなるためと考えられる．あるセットが分割割合を決定するセットに指定されたとき，サンプルタグ領域にはタグ情報が何も入っていないため，同じデータに 2 回アクセスすることで初めてキャッシュヒットが発生する．したがって更新間隔が短くなればなるほどキャッシュヒットが起こりづらい状況を作り出してしまったと考えられる．

4.7 ハードウェアオーバーヘッドの評価

本節では提案手法を実装することでかかるハードウェアのオーバーヘッドを評価する．なおサンプルタグ領域のセット数を 64，カウンタ値を一時保存するデータ領域のデータサイズを 2B とする．表 3 に分割機構に用いるデータの内訳を示す．

表 3: 分割機構に用いるデータの内訳

項目	含まれるデータ	データ量
サンプルタグ領域の各ブロック	タグ情報 24bit + LRU 情報 4bit	28bit
サンプルタグ領域の各セット	ブロック ×L2 キャッシュウェイ数 16	56B
サンプルタグ領域全体	セット × セット数 64	3712B
ヒットカウンタ	L2 キャッシュセット数 ×L2 キャッシュウェイ数 16×2B	2048B
システム全体で分割機構に必要なデータ	（タグ領域全体+ヒットカウンタ）× コア数 2	11520B
各セットにおけるブロック数管理カウンタ	4bit×L2 キャッシュセット数 1024	512B

表 3 より，分割機構で使用するデータ量が 11520B，L2 キャッシュ内の各セットにおいて各コアのブロック数を管理するためのカウンタが 512B となり，提案手法を実装するためにはシステム全体で 11KB 必要なことがわかる．これは L2 キャッシュサイズの約 1%にあたり，十分に小さいハードウェアオーバーヘッドで提案手法を対象 CMP に組み込むことが可能である．

5 関連研究

関連研究として，2.3 節で説明した UCP 以外の共有キャッシュ分割方式について説明する．UCP，提案手法，および以下に説明する手法は，いずれも共有キャッシュにおけるキャッシュヒット回数が最も多くなるような割合で共有キャッシュの分割を行う．

異なるプロセス間でのキャッシュ分割

Stone らは複数のプロセスが利用するキャッシュについて，ミスを最小化するウェイ単位の分割方式を提案した [3]．Stone らの方式では各プロセスが使用するキャッシュのウェイ数とミス数の関係を事前に調べる．この結果からキャッシュ全体のミスが最小となるウェイの分割割合を求める greedy アルゴリズムを提案した．しかし Stone らの方式では事前実行により共有キャッシュの割り当てウェイ数を変化させた場合のミス回数の記録が必要であり，アプリケーションの実行中に分割に必要な情報を取得しない点で本研究と異なる．

外部に情報取得機構を持たないキャッシュ分割

Suh らは CMP 上の共有キャッシュのミスを最小化するために，共有キャッシュを動的にウェイ単位で分割する方式を提案した [4]．この方式では各プロセスを使用するキャッシュについてウェイあたりのミス減少数である marginal gain を定義して，gain が最大値となるウェイ単位で分割割合を探索することでミスを低減する．しかし Suh らの提案方式は，CPU 上の各コアに対してウェイ毎の marginal gain を調べるために多くのハードウェア資源を必要とする．また本研究と異なる点として外部に情報取得機構を持たないことが挙げられる．Suh らの方式では分割に必要な情報の取得を共有キャッシュから直接行うため，情報の取得が正確に行うことができない．

全ての分割割合を網羅しないキャッシュ分割

Dybdahl らは，各コアが占有可能な共有キャッシュのウェイ数を一つ増減した場合のミス数を予測して，キャッシュミスを低減するように分割割合を変更する方式を提案した [6]．この方式はキャッシュの各セットにコア毎に，直前にキャッシュから置換されたデータへのアクセスを調べる Shadow tag を追加する．同時に LRU ブロックのヒット回計測して、各コアの占有ウェイ数を一つ増減した場合のミス数を

算出する．またキャッシュヒット回数を取得する部分を制限してハードウェアオーバーヘッドを削減するという点が本研究と似ている．しかしこの方式ではウェイ数を制限したのに対し，本研究ではセット数を制限することでハードウェアオーバーヘッドを削減する．

ブロックの LRU 情報を変えることによるキャッシュ分割

Jaleel らは，本研究のように共有キャッシュの分割を各コアのブロック数を管理する明示的な方法ではなく，新しくキャッシュに入ってきたブロックの LRU 情報を操作することで擬似的な分割を行う方式を提案した [10]．この方式では，キャッシュを多く割り当てられたコアのブロックが新しくキャッシュに入ってきたときは最も新しいブロックとして扱い，キャッシュを少なく割り当てられたコアのブロックが新しくキャッシュに入ってきたときは最も古いブロックとして扱う．こうすることで，多く割り当てられた方のブロックが残りやすく，少なく割り当てられた方のブロックが追い出されやすくなり，明示的な分割割合を設定することなくキャッシュの分割を行っている．

6 終わりに

6.1 まとめ

本研究では共有キャッシュを搭載した CMP において，各コアにおいて異なるアプリケーションを実行した際に共有キャッシュで発生する競合ミスを削減し，システム全体の性能を向上させるための共有キャッシュ分割機構を提案した．共有キャッシュ分割の従来手法において問題となっていた各セットにおいて最適な分割割合が保たれないことについて，各セットにおいて別々にキャッシュヒット回数を計測し個別の分割割合を決定することで問題の解決を図った．

また分割機構のハードウェアオーバーヘッドを小さくするため，キャッシュヒット回数の計測および分割割合の決定を行うセットを制限する手法を用いることで，共有キャッシュサイズの 1%にあたるデータ量で提案手法を実装できることを示した．

評価においては，提案手法を実装したシミュレータを用いることにより，従来手法より約 2%の性能向上を得られたことを確認した．

6.2 今後の課題

今後の課題として，以下の 2 つが考えられる．

1. **2 コアより多いコアを搭載する CMP への提案手法の適用**

   本研究では 2 コアの CMP を対象に手法を提案し評価を行った．しかし昨今では 4 コアや 8 コアといった CMP が登場しており，今後はさらにコア数が多くなることが予想される．コア数が増えれば増えるほどキャッシュ内のブロックの管理や分割割合の決定方法が複雑になるため，提案手法を 2 コアより多いコアを搭載する CMP へ適用するためにはこれらをより簡便に行える仕組みを作る必要がある．

2. **分割機構のデータ量の更なる削減**

   本研究で提案した手法は過去に得られた情報の保存場所としてカウンタ値の一時保存領域を用いており，またそれに多くのデータ量を必要とする．また課題の一つ目とも関係するが，コア数が増えればより多くのデータが必要となるため，分割機構のデータ量をさらに削減する必要がある．

謝辞

本研究を進めるにあたり、多大なるご指導ならびにご助言を下さいました高性能コンピューティング学講座本多弘樹教授，近藤正章准教授，平澤将一助教に深く感謝いたします．また，研究を進める上でご支援やご助言をいただきました高性能コンピューティング学講座の皆様に深く感謝いたします．

参考文献

[1] S.Rusu,H.Mulijono,J.Stinson,D.Ayers,J.Chang,R.Varada,M.Ratta,S.Kottapalli ,S.Vora:A 45 nm 8-core Enterprise Xeon®Processor,IEEE Journal of Solid-State Circuits,Vol.45,Issue3,pp.7-14,2010

[2] L.A.Barroso,K.Gharachorloo,R.McNamara,A.Nowatzyk,S.Qaderr,B.Sano,S.Smith, R.Stets,B.Verghese:Piranha: a scalable architecture based on single-chip multiprocessing,Proceedings of the 27th annual international symposium on Computer architecture,pp.282-293,2000

[3] H.S.Stone,J.Turek,J.L.Wolf:Optimal partitioning of cache memory,IEEE Transactions on Computers,pp.1054-1068,1992

[4] G.E.Suh,L.Rudolph,S.Devadas:Dynamic Partitioning of Shared Cache Memory,Journal of Supercomputing,28,pp.7-26,2004

[5] M.K.Qureshi,Y.N.Patt:Utility-Based Cache Partitioning:A Low-Overhead,High Performance,Runtime Mechanism to Partition Shared Caches,In Proceedings of the 39th Annual International Symposium on Microarchitecture,pp.423-432,2006

[6] H.Dybdahl,P.Stenstrom,L.Natvig:A cache-partitioning aware replacement policy for chip multiprocessors,In Proceedings of 2006 ACM Conference on High Performance Computing,pp.22-34,2006

[7] R.L.Mattoson,J.Gecsei,D.Slutz,I.Traiger:Evaluation Techniques for Storage Hierarchies,IBM Systems Journal,9(2),1970

[8] N.L.Binkert,E.G.Hallnor,S.K.Reinhardt:network-oriented Full-System Simulation using M5,Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads,February 2003

[9] A.Snavely,D.M.Tullsen:Symbiotic jobscheduling for a simultaneous multi-threading processor,Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operationg Systems,pp.234-244,November 2000

[10] A.Jaleel,W.Hasenplaugh,M.Qureshi,J.Sebot,S.Steely,J.Emer:Adaptive insertion policies for managing shared caches,Proceedings of the 17th international conference on Parallel architectures and compilation techniques,pp208-219,2008