Vol.2019-ARC-236 No.24
2019/6/12

キャッシュ・パーティショニングによる性能向上のための MLP を意識した実行サイクル数の推定

今泉 勇斗[1,a)]　塩谷 亮太[2]　安藤 秀樹[1]

概要：一般にマルチコア・プロセッサでは，最終レベル・キャッシュ (LLC: last-level cache) は複数のコアで共有される．この LLC のヒット率は各アプリケーションの実行性能に大きな影響を与えるため，各コアで稼働しているアプリケーションに LLC を適切に分割して与えるキャッシュ・パーティショニングが研究されている．キャッシュ・パーティショニングには種々の手法が提案されているが，その多くではミス数に基づいた制御を行う．しかし，ミス数が性能に与える影響は一定ではなく，メモリレベル並列性 (MLP: memory-level parallelism) の影響により同じミス数であっても性能に与える影響は大きく変化する．このため，ミス数に基づいたパーティショニングでは適切ではない容量の割り当てを行ってしまう場合がある．これに対し，本研究の目的は，ミス数ではなく，MLP を考慮して推定した性能に基づきキャッシュ・パーティショニングを行うことである．これを実現するため，本論文では各コアに割り当てるキャッシュ容量を変化させた場合の性能の推定方法を提案する．

1. はじめに

一般にマルチコア・プロセッサでは，最終レベル・キャッシュ（LLC：last-level cache）は複数のコアで共有されている．そのような構成においては，各コアで実行されているアプリケーションへ LLC の容量をどのように割り当てるかが性能へ大きく影響を与える．

最も簡単なキャッシュの割り当て方は，各コアの要求する容量に応じて割り当てを行うことである．この方法は，キャッシュの一般的な置換方法である LRU（least-recently used）で実現できる．この方法ではアクセス数が多いコアほど，より多くのラインがキャッシュに保持されるようになる．しかし，割り当てたキャッシュ容量に対して得られる性能はアプリケーションの性質によって大きく異なる．極端な場合，キャッシュ容量を多く割り当てても全く性能が向上しないアプリケーションも存在する．そのため，単に要求に応じてキャッシュの容量を割り当てる方法は適切ではない．

これに対し，**キャッシュ・パーティショニング**と呼ばれる手法が研究されている [1–10]．これは，複数のコアで共有されるキャッシュにおいて，各コアごとに適切なキャッシュ容量を割り当てる手法である．

多くのキャッシュ・パーティショニングの手法は，ミス数を基準としてパーティショニングを行う．以下，それらの手法を**ミス・ベースの手法**と呼ぶ．ミス・ベースの手法は，アプリケーションのキャッシュ・ミスによるストール時間が，基本的にはキャッシュ・ミス数に強く相関することを利用する．これに基づき，ミス・ベースの手法では一般に，同時実行されるアプリケーションにおけるミス数の合計が最小になるようにキャッシュ容量を割り当てる．

しかし，ミス・ベースの手法は全てのミスが性能に同程度影響を与える前提をとっているため，不適切な割り当てとなってしまう場合がある．これは，メモリ・レベル並列性（MLP：memory-level parallelism）によって隠蔽されるミス・レイテンシについて考慮が払われていないことによる．一般に，キャッシュ・ミスの処理を行っている間に並列して他のキャッシュ・ミスが発生した場合，新たなミスによるレイテンシの大部分は隠蔽される．一方で，ミスが直列に発生した場合はそれぞれのミスでストールが発生する．このように，ミスが並列して発生するか直列に発生するかによって，ミスが性能に与える影響は大きく異なる．この点がミス・ベースの手法では考慮されないため，キャッシュ容量を不適切に割り当ててしまう場合がある．

これに対し，本研究の最終的な目標は，ミス数ではなく，MLP を考慮して推定した性能に基づきキャッシュ・パーティショニングを行うことである．従来のように単純にミ

[1] 名古屋大学大学院 工学研究科
Graduate School of Engineering, Nagoya University
[2] 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, Tokyo University
[a)] imaizumi@ando.nuee.nagoya-u.ac.jp

ス数のみに基づくのではなく，性能の推定に基づいてパーティショニングを行うことで，プロセッサの全体の性能を向上させる．

この目標のために，本論文では MLP を意識した実行サイクル数の推定方法について提案する．本手法では LLC で発生したミスを，孤立して発生したミスと MLP によって隠蔽されるミスに分類する．そして，孤立して発生したミスについてのみプロセッサはストールするとし，全実行サイクルを推定する．これにより，高い精度の性能推定が実現する．

本論文では，キャッシュのパーティショニングはウェイ・ベースに行うと仮定し，あるウェイ数のキャッシュを用いて実際に実行した際に，別のウェイ数で実行した場合の実行サイクル数を推定し，その誤差を評価した．その結果，例えば 1 ウェイのキャッシュによる実行時に 16 ウェイのキャッシュによる実行サイクル数を推定した場合，ミス数のみに基づいて実行サイクル数を推定した場合の誤差率が平均 80%であったのに対し，提案手法では 6% と大幅に精度が改善したことを確かめた．また，逆に 16 ウェイのキャッシュによる実行時に 1 ウェイ実行時の実行サイクル数を推定した時の誤差率は，112% から 9% まで改善した．

2. 関連研究

本節では，既存のパーティショニング手法に関してまとめる．Qureshi らは，ミス・ベースの手法を提案した [1]．この手法では，ウェイ数を変化させることで減少するミス数を “有用度” と呼び，有用度が大きいコアに対してより多くのウェイ数を割り当てる．しかし，この手法は全てのミスを等価に扱っているため，最も性能が向上するとは限らない．

Moreto らは，MLP を考慮したパーティショニング手法を提案している [11]．この手法では，上記の有用度の計算において，MLP を置き換えのコストとして取り入れている．この手法では MLP を元にミスの重み付けを行っているが，ミス数を元に割り当て方を決定していることには変わらず，性能を推定して割り当てを決定しているわけではない．

その他にも，キャッシュ・パーティショニングには様々な手法が提案されている [2–10]．しかし，MLP を考慮し性能を推定してパーティショニングを行う手法は存在しない．

3. 提案手法

本節では，MLP を考慮した実行サイクル数の推定方法について提案する．キャッシュ・パーティショニングはウェイ・ベースに行うと仮定し，提案手法では，あるウェイ数のキャッシュを用いてプログラムを実行している際に，別のウェイ数の場合の実行サイクル数を推定する．なお以下では，特に断りがない場合はキャッシュとは LLC を指すものとする．

3.1 推定の概要

ウェイ数を変化させた場合の実行サイクル数の推定は，以下のように行われる．

(1) **ヒット/ミス推定**：
実際のキャッシュ・アクセスごとに，推定対象のウェイ数の構成（以下では「推定ウェイ構成」と呼ぶ）で実行した場合のキャッシュのヒット/ミスを推定する．以下では，この時の推定結果がミスであるものを**推定ミス**と呼ぶ．

(2) **露出推定ミス/隠蔽推定ミスの判定**：
次に，各推定ミスについて，他のミスの処理中ではない時刻に孤立して発生する**露出推定ミス**か，その他のミスの処理中に並列して発生する**隠蔽推定ミス**かを判別する．前者の孤立して発生するミスはレイテンシが露出し実行サイクル数が増加するが，後者の並列処理されるミスはレイテンシが隠蔽されるため実行時間は増加しない．

(3) **実行サイクル数の推定**：
実行サイクル数の推定は，**推定計算サイクル数**と**推定ストール・サイクル数**の合計によって求める．推定計算サイクル数はキャッシュ・ミスの処理が行われていない区間の実行サイクル数であり，実際の実行の際にキャッシュ・ミスでストールしていなかった時間である．一方，推定ストール・サイクル数はキャッシュ・ミスによりストールすると推定されるサイクル数であり，前述した露出推定ミスの発生回数にメモリ・アクセス・レイテンシを乗じたものとする．

以降では，これらの動作について詳しく説明する．

3.2 ヒット/ミス推定

ウェイ数を変えた場合のアクセスのヒット/ミスを推定するために，最大サイズの LLC のタグ・アレイを設ける．あるコアに w ウェイを割り当てた場合のヒット/ミスの推定は，LRU スタックの状態をみることで行う．各アクセスごとに，そのアクセスが LRU スタックの上位 w ラインにヒットすれば推定ヒット，そうでなければ推定ミスである．

3.3 ウィンドウによる判定の概要

露出推定ミス/隠蔽推定ミスを判定するために，**図 1** に示す時間軸上の，次の 3 つのウィンドウを定義する．

- **ストール・ウィンドウ**：
これは，推定ミスが生じてから，そのデータが得られると推定されるまでの期間である．図 1 に表すように，推定したい M ウェイで動作させた時のキャッシュ・ミスが発生してからデータを得られると推定される期

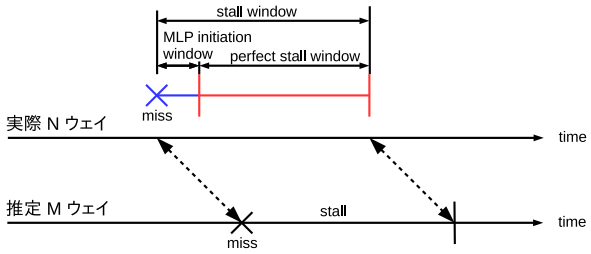

図 1　ストール・ウィンドウ（stall window）と MLP 開始ウィンドウ（MLP initiation window），完全ストール・ウィンドウ（perfect stall window）

間を，現在動作している N ウェイのキャッシュでの実行時間軸上にマッピングして管理する．

- MLP 開始ウィンドウ：
  これは，推定ミスが発生してからプロセッサが完全にストールするまでの期間である．一般に，キャッシュ．ミスが発生した場合でも，プロセッサはただちにはストールしない．ミスを生じた命令がリオーダ・バッファ（ROB : Reorder Buffer）の先頭に至り，ROB からのリタイアを詰まらせ，ROB が一杯になって初めてプロセッサは完全にストールする．MLP 開始ウィンドウ内ではプロセッサは完全にはストールしていないため，新たにキャッシュ・ミスが発生する場合がある．この期間に生じたミスは ROB を詰まらせたミスの処理とオーバーラップして処理されるため， MLP によりメモリ・レイテンシはほぼ完全に隠蔽される．
- 完全ストール・ウィンドウ：
  完全にプロセッサがストールしている期間を表す．実際のプロセッサではこの期間にミスが発生することはないが，そのキャッシュよりウェイ数が少ない場合の推定を行う場合は，推定ミスが発生する場合がある．これに関しては，3.4.3 節 にて詳しく述べる．

これらのウィンドウを推定対象となる全てのウェイ数ごとに用意し，対応するウェイ数の推定ミスの発生ごとに更新する．後で詳しく述べるように，基本的には推定ミスが MLP 開始ウィンドウ内にあった場合は隠蔽推定ミスと判定し，そうでない場合は露出推定ミスと判定する．

以下の節ではこれらのウィンドウの更新方法と，隠蔽/露出推定ミス判定方法の詳細について説明する．ウィンドウの更新方法は，推定ミスがロード・ミスによるものか命令アクセスによるミスによるものかで若干異なる．以下ではまずロードによる推定ミスの場合について述べ，その後両者の違いについて述べる．なお，ストア・ミスについては多くの場合は実行時間に影響を与えないため，提案手法では単純に無視する．

3.4　ウィンドウの更新

本節では，ロード・ミスによるウィンドウの更新について述べる．図 1 に示すように，推定ミスが発生するタイミングは，1) ストール・ウィンドウ外，2) MLP 開始ウィンドウ内，3) 完全ストール・ウィンドウ内の 3 つに分類することができる．以下ではそれぞれの場合の動作を説明する．

3.4.1　ストール・ウィンドウ外で推定ミスが発生した場合

図 2 に，ストール・ウィンドウの外で推定ミスが発生した場合のタイミング図を示す．この推定ミスは他の推定ミスの処理によって隠蔽されないため，露出推定ミスとなる．露出推定ミスが検出されると，ストール・ウィンドウおよび MLP 開始ウィンドウを設定する．

ストール・ウィンドウは，その開始点は露出推定ミスが発生したサイクルであり，終了点はそこからメイン・メモリのアクセス・レイテンシが経過したサイクルである．

一方，MLP 開始ウィンドウはサイクル数ではなく，命令シーケンス番号で表現される．その開始点は，露出推定ミスを発生させた命令の命令シーケンス番号であり，終了点はそれに （ROB サイズ −1） を加えたものである．

3.4.2　MLP 開始ウィンドウ内で推定ミスが発生した場合

図 3 に，MLP 開始ウィンドウ内で推定ミスが発生した場合のタイミング図を示す．この場合，他のミスと並列に処理することができる．そのため，推定ミスは隠蔽推定ミスとなる．ストール・ウィンドウ，MLP 開始ウィンドウは変化しないため，更新は行わない．

なお，命令間の関係に応じて例外的な処理を行う必要がある場合がある．これについては 3.5 節で述べる．

3.4.3　完全ストール・ウィンドウ中で推定ミスが発生した場合

完全ストール・ウィンドウ中で推定ミスが発生した場合，それは露出推定ミスとなる．このことを**図 4** を用いて説明する．同図において，上の時間軸は実際の実行（N ウェイ）の時間軸を表しており，M（$< N$）ウェイの実行に対応する完全ストール・ウィンドウ中で推定ミス miss2 が生じている（マーク（1））．一方，下の時間軸は推定の実行（M ウェイ）の実行軸を表している．miss2 の発生が推定されたサイクルでは，実際の実行時間軸上では，プロセッサは実際には完全にストールしているわけではないので miss2 の発生は推定されうる．一方，推定実行時間軸上では，miss2 の発生サイクル（マーク（2））は miss1 の処理中のでプロセッサは完全ストールしている．したがって，miss2 の処理の開始は miss1 の処理が終了した後となる（マーク（3））．このため露出推定ミスとなる．この miss2 の処理の開始は miss1 の処理が終了した後のため，現在の実行（N ウェイ）の時間軸上において，miss1 の処理 が完了した後にストール・ウィンドウ，MLP 開始ウィンドウを設定する（マーク（4））．

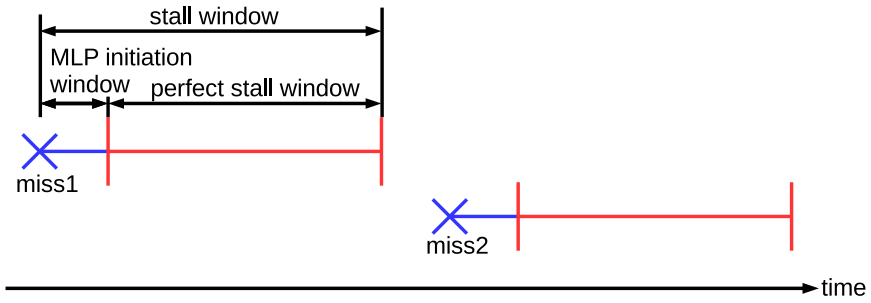

図 2　ストール・ウィンドウの外でミスが発生した場合

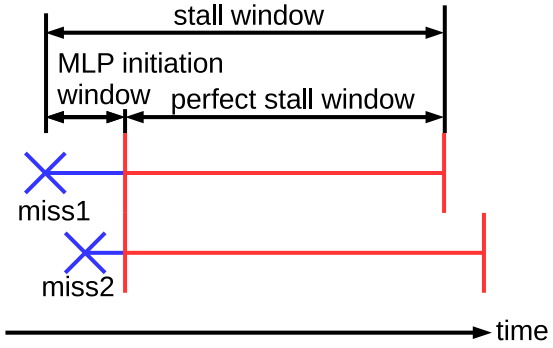

図 3　MLP 開始ウィンドウ中にミスが発生した場合

3.5　MLP 開始ウィンドウ内で推定ミスが発生した場合の例外的措置

3.5.1　現在の MLP 開始ウィンドウの開始点より古い命令が推定ミスを起こした場合

命令がアウト・オブ・オーダ実行されているため，現在の MLP 開始ウィンドウの開始点よりも古い命令が推定ミスを起こすことがある．この場合，現在のストール・ウィンドウと MLP 開始ウィンドウをその時起きた推定ミスを基準として再設定する．

3.5.2　推定ミスの依存元に MLP 開始ウィンドウを開始させたミスがある場合

推定ミスの依存元に MLP 開始ウィンドウを開始させた推定ミスが存在した場合，隠蔽推定ミスではなく露出推定ミスとして処理する．なぜなら，この推定ミスは，先行する推定ミスと直列に実行されるためである．よって メモリ・レイテンシは隠蔽されず，ストール・サイクル数はメモリ・レイテンシだけ増加する．またこの場合，既に開始されているストール・ウィンドウの終了点から新しいストール・ウィンドウを開始する．また，既に開始されている MLP 開始ウィンドウの後ろに同様に新しい MLP 開始ウィンドウを設定する．

この依存関係の検出には，1 ビットのエントリを論理レジスタ数分だけ持つテーブルを用いる．MLP 開始ウィンドウを開始した命令がリタイアすると，そのディスティネーション・レジスタに対応するテーブルのエントリに 1 を書き込む．各命令のリタイア時にはソース・レジスタに対応するエントリを読み出し，それらの論理和をディスティネーション・レジスタに対応するエントリに書き込むことで，依存関係を伝播させていく．推定ミスが解決された際には，全エントリのビットをクリアする．

3.6　命令アクセス・ミスによるウィンドウの更新

これまでロード・ミスについて，ウィンドウの更新方法について述べたが，本節では命令アクセスによるミスが生じた場合について説明する．命令アクセスによる推定ミスが発生した場合，その命令で命令フェッチが止まってしまうため，これより以前にフェッチされた命令のみ並列に実行することができる．そのため，MLP 開始ウィンドウもその推定ミスより古い命令で設定する必要がある．

まず，命令アクセスに起因する推定ミスが孤立して発生した場合について述べる．**図 5** に命令アクセスによるミスが発生したタイミングを示す．命令キャッシュに起因するミス miss1 が孤立して発生したとする．この時，命令フェッチが停止するため，実行される可能性がある命令は miss1 より命令シーケンス番号が古い命令のみとなる．そのため，MLP 開始ウィンドウとして設定する範囲は，miss1 の命令シーケンス番号から（ROB のサイズ - 1）だけ引いたものから miss1 の命令シーケンス番号の範囲となる，

次に，命令アクセスによる推定ミスが MLP 開始ウィンドウの中で発生した場合について述べる．**図 6** に命令アクセスによるミスが発生したタイミングを示す．まず，通常の孤立推定ミス miss1 が発生し，ストール・ウィンドウと MLP 開始ウィンドウを設定していたものとする．そこへ，命令アクセスによる推定ミス miss2 が現れたものとする．この時，miss2 は miss1 と並列に処理されるため，隠蔽推定ミスとなる．しかし， miss2 より後の命令は並列に実行できない．そのため， miss1 によって設定された MLP 開始ウィンドウの範囲を， miss1 の命令シーケンス番号に ROB の大きさだけ加えた値までとされていたものを，

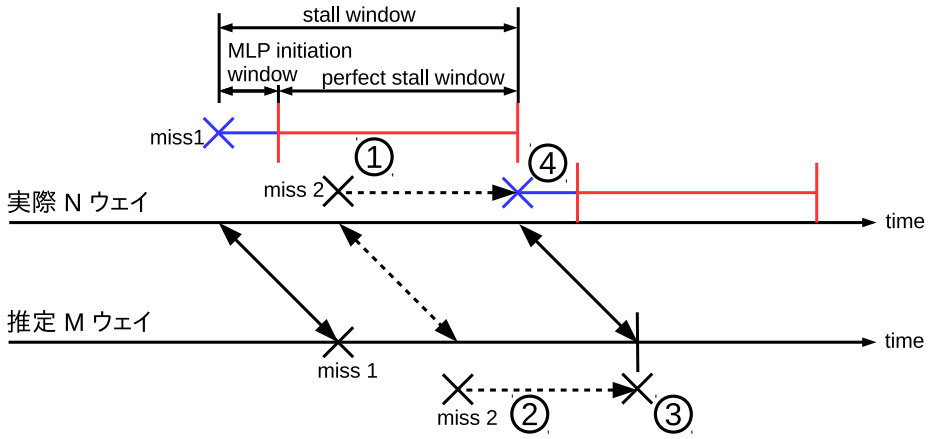

図 4　完全ストール中に発生したミスの開始点を移動

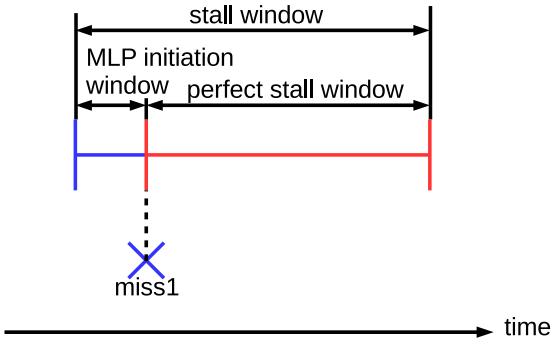

図 5　命令アクセスによるミスが発生した場合

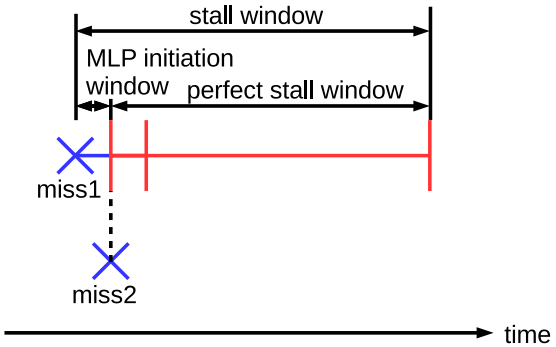

図 6　MLP 開始ウィンドウ中に命令アクセスによるミスが発生した場合

miss2 の値までと変更する．

3.7　ストール・サイクル数の推定

実行サイクル数の推定は，推定計算サイクル数と推定ストール・サイクル数の合計によって求める．

このうち，推定計算サイクル数はキャッシュ・ミスの処理が行われていない区間の実行サイクル数である．通常，この推定計算サイクル数はキャッシュ・ミスの有無によってはほとんど変化しないと予想されるため，実際にプログラムを実行している際にキャッシュ処理を行っていなかったサイクル数とする．

一方推定ストール・サイクル数は，前述の露出推定ミスの発生回数にメモリ・アクセス・レイテンシを乗じたものとする．なお，隠蔽推定ミスについては他のミスと並列実行されてレイテンシが隠蔽されるため，ストール・サイクル数には影響を与えないとする．

ストール・サイクル数の推定は命令がリタイアされた際に行う．あるキャッシュ・ミスの処理が露出するか/隠蔽されるかの判定はミスの発生時に行っている．しかし，ミスを生じたロードは分岐予測ミスによってフラッシュされ，露出/隠蔽の判断も無効となる．このため，ストール・サイクル数の推定はコミット時に行う．

4.　評価

本節では，提案手法による推定の精度の評価について述べる．

4.1　評価環境

gem5 stable_2015_09_03 [12] をベースに提案手法を実装したシミュレータを用いて評価した．使用した命令セットは ARMv8 であり，ベンチマークとして，SPEC CPU 2006 [13] の 29 本のアプリケーションを使用した．プログラムのコンパイルには gcc ver.4.9.3 を使用した．入力には ref データ・セットを用い，先頭の 16G 命令をスキップし，その後の 100M 命令を実行した．**表 1** にベースとなるプロセッサの主な構成を示す．このプロセッサの LLC は L2 キャッシュである．

以下では，提案手法による推定の誤差率について評価する．実際の実行サイクル数に対する推定実行サイクル数の誤差率 $error\_rate_{exec}$ は以下の式により表される．

表 1　プロセッサの構成

frequency	4GHz
Pipeline width	8-instruction wide for each of fetch, decode, issue, and commit
ROB	200 entries
Issue queue	100 entries
LQ	100 entries
SQ	100 entries
Branch prediction	64K-entry PHT Tournament BP 16-bit history 64K-entry PHT GAg 64K-entry BHT 64K-entry PHT PAg 4K-set BTB, 10-cycle mispred penalty
Function unit	4 iALU, 2 iMULT/DIV, 2 Ld/St, 4 fpALU, 2 fpMULT/DIV/SQRT
L1 I-cache	32KB, 8-way, 64B line
L1 D-cache	32KB, 8-way, 64B line, 2 ports, 2-cycle hit latency
L2 cache	2MB, 16-way, 64B line, 12-cycle hit latency
Main memory	300 cycle latency, 32 B/cycle bandwidth

$$error\_rate_{exec} = \left(\frac{exec\_cycles_{estimate}}{exec\_cycles_{actual}} - 1 \right) \qquad (1)$$

このうち，$exec\_cycles_{estimate}$，$exec\_cycles_{actual}$ はそれぞれ 推定実行サイクル数 ，およびウェイ数で実際に実行した際の実行サイクル数である．以下では，MLP を考慮せず全てのミスが直列に発生すると仮定した場合（以下ではミス数をもとに推定する場合と呼ぶ）の実行サイクル数の誤差率と比較を行いながら評価を行う．

4.2　評価結果

4.2.1　MPKI

まず図 **7** に，背景として各ウェイ数のキャッシュを用いて実際に実行した際の MPKI（miss per kilo instruction）を示す．ベンチマークは MPKI が降順になるよう並べてある．LLC のウェイ数を変化させることで MPKI が大きく変動していることがわかる．以下ではこの MPKI を参照しながら，各モデルの誤差率について議論する．

4.2.2　実 1 ウェイ，8 ウェイ，16 ウェイ構成時の推定結果の平均

本節では，1，8，16 ウェイで実行している際，に N ウェイ（N = 1,4,8,12,16）を推定した場合の推定精度について述べる．図 **8** に実際の実行サイクル数に対する推定実行サイクル数の誤差率の平均を示す．

図 8 を見ると，ミス数を基に推定を行った場合は平均で 80%〜110% 程度の誤差が出ているのに対して，提案手法はいずれの場合も誤差が 10% よりも低く，高い精度で推定を行えていることが分かるる．

4.2.3　実 1 ウェイ構成時の推定結果

本節では，実際に 1 ウェイで実行している際に，N ウェイ（N = 1,4,8,12,16）で実行した場合の実行サイクル数の

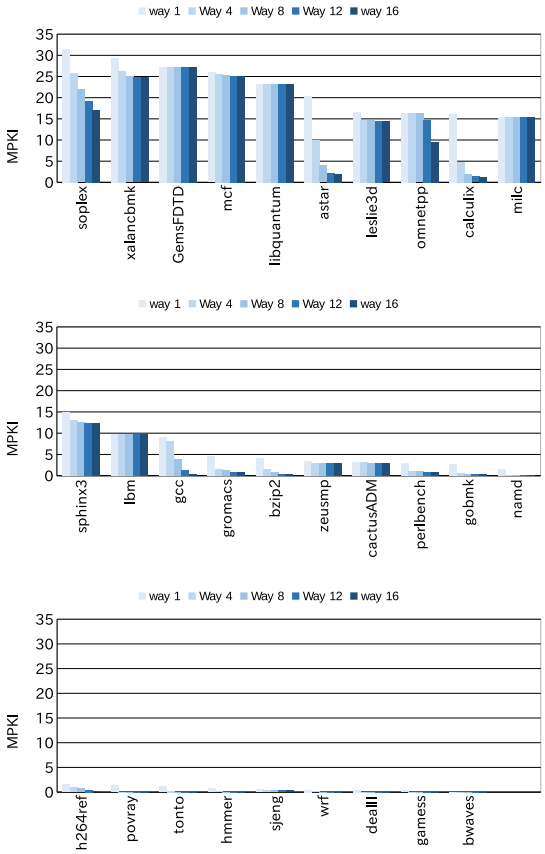

図 7　各ウェイ数で実際に実行した場合の MPKI

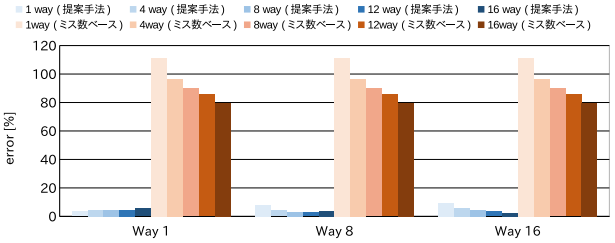

図 8　1 ウェイ，8 ウェイ，16 ウェイで実行時に他のウェイ数での実行サイクル数を推定した場合

推定結果について述べる．図 **9** に実際の実行サイクル数に対する推定実行サイクル数の誤差率を示す．

実際に 1 ウェイで実行した際に 16 ウェイの実行サイクル数を推定する場合の誤差率は，ミス数を基に推定する場合は 80% であるのに対して提案手法では 6% であり，大幅に精度を改善している．

また，29 本のベンチマークの内 21 本のベンチマークでいずれのウェイ数での推定の場合も誤差率は 10 % 以下におさまった．最も大きく誤差が生じていたのは gcc で 16 ウェイを推定している時に 28% の誤差が生じていた．

4.2.4　実 8 ウェイ構成時の推定結果

同様に，8 ウェイで実行している際に N ウェイ（N = 1,4,8,12,16）の実行サイクル数を推定し評価する．図 **10** に実際の実行サイクル数に対する推定実行サイクル数の誤差率を示す．

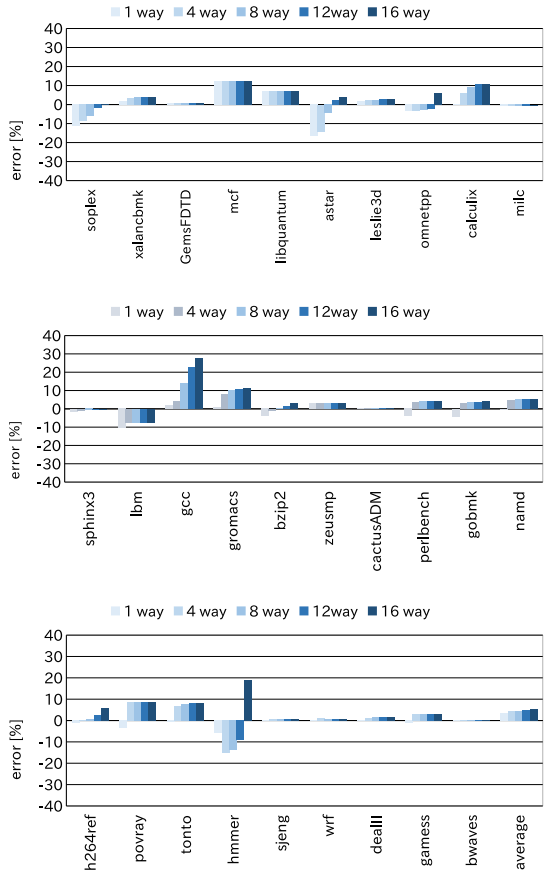

図 9 1 ウェイで実行した時に他のウェイ数での実行サイクル数を推定した場合

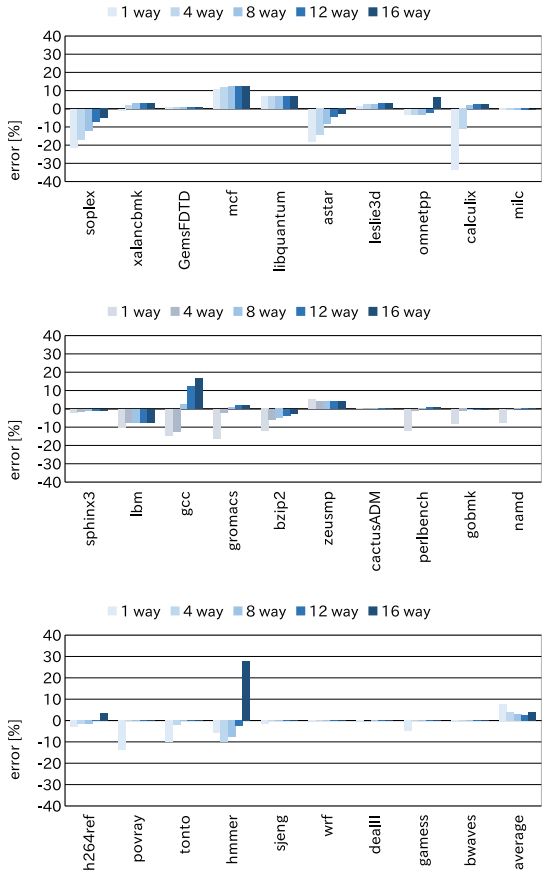

図 10 8 ウェイで実行した時に他のウェイ数での実行サイクル数を推定した場合

1 ウェイの実行サイクル数を推定する場合のエラー率の大きさの平均は，ミス数を基に推定する場合は 112% であるのに対し，提案手法では 8% とより正しく実行サイクル数を推定することができている．16 ウェイの実行サイクル数を推定する場合のエラー率の大きさの平均についても，ミス数を基に推定する場合は 80% であるのに対して提案手法では 4% となっており，キャッシュ容量を変えた時の実行サイクル数についても正しく推定することができている．

この場合は，29 本のベンチマークの内 18 本のベンチマークでいずれのウェイ数を推定する場合も誤差率が 10 % 以下という結果となった．また，26 本のベンチマークでいずれのウェイ数での推定の場合も誤差率が 20% 以下という結果となった．最も大きく誤差が生じていたのは calculix で 1 ウェイを推定している時に -33% の誤差が生じていた．

4.2.5 実 16 ウェイ構成時の推定結果

同様に，16 ウェイで実行している際に N ウェイ（N = 1,4,8,12,16）の実行サイクル数を推定し評価する．**図 11** に実際の実行サイクル数に対する推定実行サイクル数の誤差率を示す．

16 ウェイの実行サイクル数を推定する場合のエラー率の大きさの平均は，提案手法では 6% とミス数を基に求めた場合よりも正しく実行サイクル数を推定することができている．1 ウェイの実行サイクル数を推定する場合のエラー率の大きさの平均についても，提案手法では 11% となっており，キャッシュ容量を変えた時の実行サイクル数についても正しく推定することができている．

この場合は，29 本のベンチマークの内 17 本のベンチマークでいずれのウェイ数での推定の場合も誤差率が 10 % 以下という結果となった．また，25 本のベンチマークでいずれのウェイ数を推定する場合も誤差率が 20% 以下という結果となった．最も大きく誤差が生じていたのは calculix で 1 ウェイを推定している時に -35% の誤差が生じていた．

5. まとめ

多くのキャッシュ・パーティショニングの手法では，ミス数に基づいて割り当てを行うため，不適切な割り当てを行ってしまう場合がある．これに対し，本研究の目的は，ミス数ではなく，MLP を考慮して推定した性能に基づきキャッシュ・パーティショニングを行うことである．これを実現するため，本論文では各コアに割り当てるキャッシュ容量を変化させた場合の性能の推定方法を提案した．

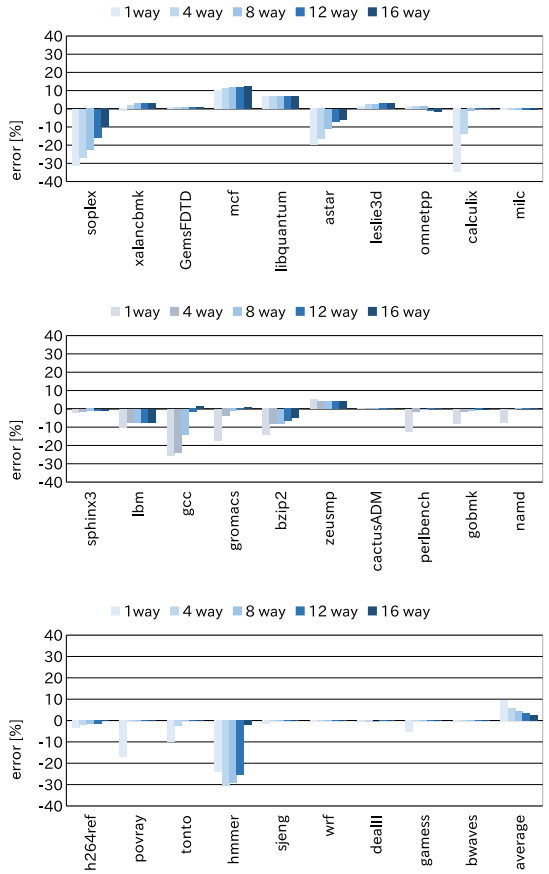

図 11 16 ウェイで実行した時に他のウェイ数での実行サイクル数を推定した場合

評価の結果，ミス数のみに基づいて実行サイクル数を推定する手法と比べて実行サイクル数の推定精度を大幅に改善していることを確かめた．

謝辞

本研究の一部は，日本学術振興会 科学研究費補助金基盤研究 (C)（課題番号 16K00070）及び，科学研究費補助金若手研究 (A) (課題番号 16H05855) による補助のもとで行われた．

参考文献

[1] Qureshi, M. K. and Patt, Y. N.: Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches, *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pp. 423–432 (2006).

[2] Xie, Y. and Loh, G. H.: PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches, *Proceedings of the 36th Annual International Symposium on Computer architecture*, pp. 174–183 (2009).

[3] Sanchez, D. and Kozyrakis, C.: Vantage: scalable and efficient fine-grain cache partitioning, *Proceedings of the 38th Annual International Symposium on Computer architecture*, pp. 57–68 (2011).

[4] Manikantan, R., Rajan, K. and Govindarajan, R.: Probabilistic shared cache management (PriSM), *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 428–439 (2012).

[5] Wang, R. and Chen, L.: Futility scaling: High-associativity cache partitioning, *Proceedings of the 47th Annual International Symposium on Microarchitecture*, pp. 13–17 (2014).

[6] H. Cook, M. M., S. Bird, K. D., Patterson, D. A. and Asanovic, K.: A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness, *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 308–319 (2013).

[7] Subramanian, L., Sechadri, V., Ghosh, A., Kahn, S. and Multu, O.: The application slowdown model: quantifying and controlling the impact of inter-application interfence at shared caches and main memory, *Proceeding of the 48th Annual International Symposium on Microarchitecture*, pp. 62–75 (2015).

[8] 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).

[9] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C. and Emer, J.: Adaptive insertion policies for high performance caching, *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 381–391 (2007).

[10] Jaleel, A., Theobald, K. B., Steely, S. C. and Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP), *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 60–71 (2010).

[11] Moreto, M., F. J. Cazorla, A. R. and Valero, M.: MLP-aware dynamic cache partitioning, *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, pp. 337–352 (2008).

[12] : http://www.gem5.org.

[13] The Standard Performance Evaluation Corporation: *SPEC CPU 2006 Suite*.