Vol.2010-ARC-191 No.1
2010/10/18

組込みプロセッサ向けデータキャッシュ制御方式の検討

請園智玲[†1]　田中清史[†1]

近年，組込みアプリケーションの大規模化が要求され，組込みプログラムが扱うデータセットも同様に大規模化している．データセットの大規模化に合わせて大規模なキャッシュメモリを搭載することは，安価なシステムの設計が必要とされる組込みシステムにおいては難しい．本稿では，小容量バッファを組み合わせたデータキャッシュのフィル制御方式を議論する．また，小規模データキャッシュのデータの参照局所性に合わせたフィル先選択を行うことにより，小規模データキャッシュで大規模データセットを扱う場合のパフォーマンス低下の緩和が可能になる小容量バッファの利用法を提案する．

Considerations of Data Cache Control Method for Embedded Processors

Tomoaki Ukezono[†1] and Kiyofumi Tanaka[†1]

Recently, data sets that are dealt with in embedded programs are increasing as embedded applications is enlarged. It is difficult to implement large cache memories in accordance with increased data sets, since embedded systems are often required to be inexpensive. In this paper, we discuss control of filling for data caches with a small-sized buffer. Furthermore, we propose how to utilize the small-sized buffer to alleviate performance degradation by selecting fill targets when the small data caches deal with large data sets.

1. はじめに

近年，組込みシステム上で実行されるアプリケーションが大規模化されている．大規模化されたアプリケーションは大規模なデータセットを用いた計算を行うため，組込みプロセッサには，今後更に大規模なデータキャッシュメモリの搭載が要求される．一方で，組込みプロセッサは製品の一部として組み込まれるため，汎用システムに比べてコストセンシティブな設計が要求される．組込みシステムに大規模なキャッシュメモリを搭載することは，直接，歩留まりの低下による製品価格の上昇につながるため難しい問題となる．このような状況下で，十分なキャッシュメモリを提供できないためプロセッサの動作周波数を十分に活かすことのできない組込みシステムが開発される可能性がある．

本研究は十分なキャッシュメモリを提供できない場合に，データの参照傾向に応じて，適切な格納位置を知ることにより，少ないキャッシュメモリ資源を有効的に利用し，キャッシュミスによるパフォーマンス低下を緩和する手法を提案する．

キャッシュミスによるパフォーマンス低下要因は数多くあるが，最も最悪な要因がスラッシングである．小規模キャッシュメモリは，セット数またはウェイ数を少なく設定せざるを得ない．セット数を減らせば，1 セットに競合するメモリブロックを増やす可能性があり，ウェイ数を減らせば，1 セットが収容可能なメモリブロック数を減らすことになる．1 セットに対してウェイ数を超える参照メモリブロックの保持を要求された場合，当該ウェイは衝突するメモリブロック間でキャッシュブロックを取り合う状態となる．この状態を一般的にスラッシング状態と言う．このスラッシング状態では，当該セットで衝突するメモリブロック同士がミスとフィルを繰り返すが，最も最悪なケースでは，全参照が一度もヒットしない．このことから，小規模キャッシュメモリ構成は必然的にスラッシング状態発生の危険性を高めていると言える．また，スラッシング状態は特定セットを機能不全に陥れることから，スラッシングが発生した場合，キャッシュメモリは急激な性能の低下を余儀なくされる．これが，キャッシュサイズにキャッシュ性能が追従しない大きな原因の一つと言える．本稿では，小規模キャッシュメモリにおけるスラッシング状態に着目し，キャッシュメモリでスラッシングが発生した場合に，それを緩和する手法を提案する．本提案は 1〜8 ブロック以内の小規模のバッファを用いてスラッシング状態を緩和する．バッファはキャッシュと並列に置かれ，同時に参照される．本提案のコンセプトは非常に単純である．バッファ利用の主な目的はキャッシュで発生するスラッシングの原因となる参照を全てバッファに担わせて，本来キャッシュで発生するはずであったスラッシングをバッファ内で発生させることにより，スラッシングの影響範囲を限定することにある．この提案が理想的に実現できれば，最低限キャッシュ内に収まる範囲のメモリブロックに対する参照はスラッシングに関係なくヒットが保証され，結果的にキャッシュミス率低減につながる．

†1 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology (JAIST Hokuriku)

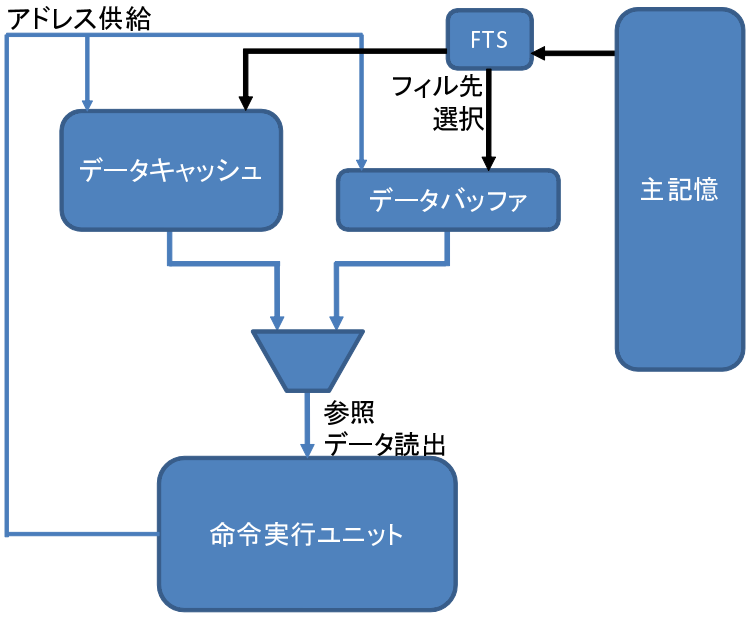

図 1 提案するキャッシュメモリシステムの概要図.

本稿は以下の節で構成される．1 節では本研究の背景として，組込みシステムにおける小規模キャッシュのスラッシング緩和の重要性に関する議論と提案手法の概要の説明を述べた．2 節では，実際にこの提案を行うために必要なキャッシュメモリシステムとそのシステムの持つ特殊なフィル制御に関して議論する．また，3 節では提案したキャッシュメモリシステムをスラッシング緩和に適用するための提案を行う．4 節で提案手法の効果を示すための予備評価を行う．5 節で提案手法の関連研究を述べる．最後に，6 節で結論を述べる．

2. 提案キャッシュメモリシステムとフィル制御方式

2.1 提案キャッシュメモリシステム

提案するキャッシュメモリシステムの概要図を図 1 に示す．通常のキャッシュシステムは命令実行ユニットがロード／ストアを実行すると，最初にデータキャッシュを参照し，データキャッシュに参照データが存在しなければ，キャッシュミスとなり，低階層のキャッシュあるいは主記憶から当該メモリブロックをロードする．提案するキャッシュメモリシステムは，データキャッシュとデータバッファにアドレスを与えて同時に参照を行う．どちらか一方でもヒットした場合は，データキャッシュ／バッファから命令実行ユニットにデータが供給される．逆にどちらにもヒットしない場合，つまりミスの場合，主記憶からデータをロードするが，このときに FTS(Fill Target Selector) を通してフィルが行われる．FTS は主記憶から到着したメモリブロックをデータキャッシュとデータバッファのどちらにフィルするかを選択するハードウェアである．FTS でフィル先を決定されたメモリブロックはどちらか一方に格納される．このキャッシュメモリシステムはデータキャッシュとデータバッファ間でキャッシュブロックを交換する様な仕組みはなく，それぞれ独立している．このため，リプレースはデータキャッシュ又はデータバッファの内部で完結している．このことから，データバッファと FTS は従来のデータキャッシュに付随する付加ハードウェアと位置付けることができ，FTS とバッファを付加することによるデータキャッシュの複雑性や参照遅延の増加を最低限に抑えられる．

2.2 フィル制御方式

FTS の実装にはいくつかの方法が考えられる．その中で最も FTS に要するハードウェア量を少なくする方法が，主記憶内の各メモリブロックのフィル先をプログラム実行前に静的に確定しておき，あらかじめ主記憶にセットしておく方法が考えられる．この場合，FTS は主記憶から送られてくるメモリブロックに付随するフィル先情報見て，フィル先を変更する機能を持つ単なるセレクタ回路となる．

提案するキャッシュメモリシステムのフィル対象はデータキャッシュとデータバッファの 2 種類なので，各ブロックに付き 1 ビット必要となる．この情報を付加することによるメモリオーバヘッドは，例えば 16 バイトキャッシュブロックで 0.8%程度，更にブロックサイズを大きくするに従いこのオーバヘッドは縮小していく．組込みシステムにおいてメモリ容量は製品コストを決める重要な要因となるが，1%以下の増加でスラッシング緩和が可能となるのであれば，提案キャッシュメモリシステムを採用する動機づけとなる．

この 1 ビットのフィル先情報の送付方法は採用するメモリインタフェースによって異なるが，現状で採用されているのメモリインタフェースの場合，バースト転送で 1 度に 1 キャッシュブロック分のデータを転送する方式が一般的であり，更にデータ転送時にパリティビット等を付加する方式が多い．このため，これら付加情報を拡張し，フィル先情報を添付することは現状のメモリインタフェースにおいても無理のない実装で送付することが可能である．フィル先情報の格納はメモリモジュールに対する変更が必要となる．本提案では，フィ

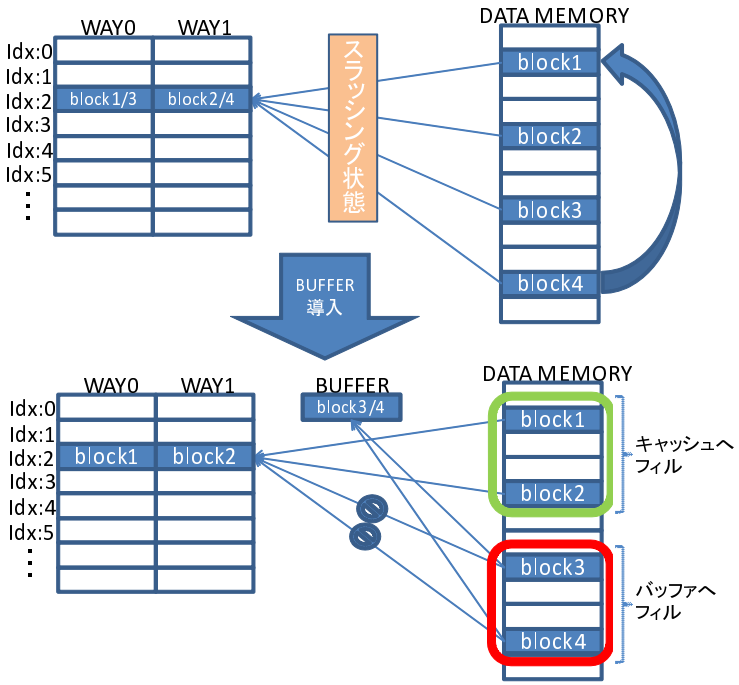

図 2 小容量バッファを用いたスラッシング緩和の概要図.

ル先情報は実行バイナリと一緒に用意され，プログラム実行前にメモリシステムにセットされることを想定している．多くの組込みシステムでは，バイナリは静的に用意され，システム起動時にロードされることから，フィル先情報も同様にシステム起動時に主記憶にセットされ，実行バイナリと並列に参照可能な形で格納されている必要がある．

3. FTS を用いたスラッシング緩和手法

3.1 FTS のスラッシング緩和適用の有効性

2.2 節で FTS がフィル先を選択することを述べた．本稿の目的は FTS とデータバッファを追加することにより，データキャッシュのスラッシングを緩和することである．小容量バッファを用いたスラッシング緩和の概要図を図 2 に示す．上部図は通常のキャッシュの場合のキャッシュフィルを示すもので，下部図はバッファと FTS によるフィル制御を加えた場合の図である．上部図はキャッシュのインデックス番号 2 に競合する block1〜block4 までのメモリブロックが 1 から 4 の順で各ブロックに対し 1 回づつ繰り返し参照される状況を示している．データキャッシュのウェイ数は 2 であるため，競合する 4 つのブロックを格納することができず，block4 までの初期参照が終わり (初期参照は当然ミス)，再度 block1 と block2 の参照を行ってもミスとなり，block1 と block2 のフィルで block3 と block4 がリプレースされ，その後の参照も，やはりミスとなる．このような状況では，何度参照を繰り返しても，データキャッシュは一度もヒットしない．この状況を一般的にスラッシングと言う．

下部図では上部図の参照でバッファを導入し，FTS で特定のメモリブロック (block3 と block4) をバッファに挿入した場合のフィルを示している．重要なのは block1 と block2，block3 と block4 のそれぞれのフィル先を分けたことである．バッファに block3 と block4 を格納することで block1 と block2 はリプレースされることなく，(初期参照を除いて) ヒットし続ける．図ではバッファは 1 ブロックのみのため，この 1 ブロックを block3 と block4 が取り合いやはりバッファ内でスラッシングを起こしているが，この例では，キャッシュミスを約半分に減らすことに成功している．

例示した参照列では，バッファに格納しなくとも，block3 と block4 内の該当データを直接命令実行ユニットに渡すバイパス回路が存在した場合，性能はそれと変わらない．しかしながら，実際の参照列では，例えば，キャッシュの空間的局所性に代表されるような，ごく短時間に 1 ブロックに参照が集中する参照列が多く存在する．バッファの存在はこの場合に有効となる．

3.2 フィル先情報の生成

本提案を実現するためには主記憶に格納するフィル先情報をいかに生成するかが問題となる．多くの組込みシステムでは，システムで実行されるプログラムが静的に決定していることが多い．本稿ではプログラムの事前実行によりメモリアクセスのトレースを取得し，そのトレースを解析することにより，スラッシングが発生するブロックアドレスを特定し，フィル先情報を生成する手法を提案する．

まず，スラッシング関係にあるブロックアドレスをメモリアクセスのトレースから取得するために，メモリアクセストレースをキャッシュミス発生時のアドレスに限定し，更にブロックアドレス毎にミス回数を集計した．数値は SimpleScalar 4.0 コードベースの CPU シミュレータ SimpleScalar/ARM[1] を用いてデータキャッシュシミュレーションを行い計測した．計測したベンチマークアプリケーションは MiBench Version 1.0[2] である．計測したデータキャッシュ構成はブロックサイズ 16 バイト，4KB サイズの 4WAY 構成である．図

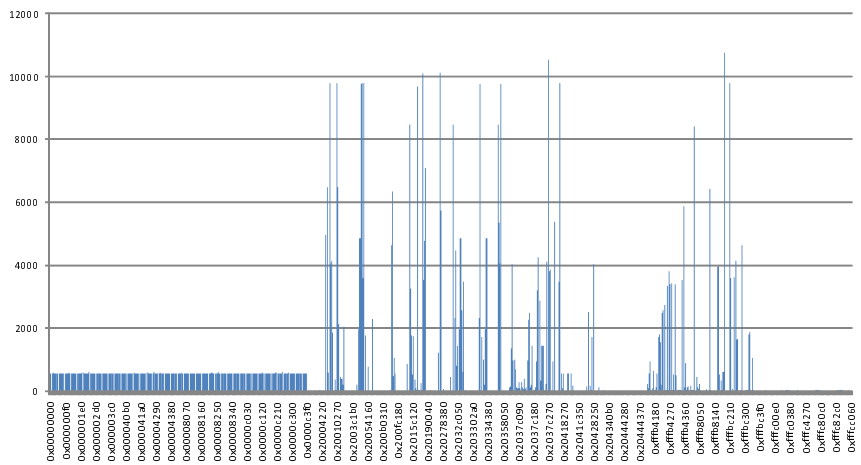

図 3 4KB-4WAY 構成時の basicmath におけるブロックアドレス毎のミス回数.

3 に MiBench のアプリケーションの一つ，basicmath のミスブロックアドレス毎のミス数を示す．図の横軸はミスブロックアドレスを昇順にソートしてあり，縦軸はミス数を示している．図は個々のアドレスに着目するのではなく，スパイクに着目する．たいていのミスブロックアドレスは 4000 回以内のミスに収まっているのにも関わらず，ごく稀に 10000 回近辺にミス数が到達するミスブロックアドレスが存在していることが判る．

高いミス数が観測されるということは，他のブロック参照と関係してミスが発生していることは明らかである．これは，リプレースされやすいにも関わらず，リロードされる頻度が高いことを示している．本稿では，これら 10000 回近辺の高いミス数を示すスパイクがスラッシングによるものであると推測した．

スラッシング状態を ‘一度もヒットしない状態’ と定義するならば，ミス数のみを調査しただけでは不十分であるが (リプレースされるまでに何回ヒットしたかという指標が無いため)，その解釈を ‘インデックス競合の問題で長期にわたる高頻度の参照をキャッシュミス無しで扱えない状態’ と拡大したときに，ミス数は十分な指標となる．本稿では，スラッシングと言う言葉をこの拡大した解釈で使用している．また，2 節で提案したキャッシュメモリシステムもヒットのない完全なスラッシングを対象とせず，短期的な局所性を満たすために，スラッシング関係のメモリブロックの一部をノンキャッシャブルに扱うわけではなく，バッファに挿入している．このことからも，本稿のスラッシング解析はミス数のみの解析で十分であると言える．

図 3 のようなミスブロックアドレス列を入手できた場合に，スパイク部を抽出し，バッファにフィルすべきブロックアドレスのリストを生成するアルゴリズムを以下に示す．

(1) トレースからブロックアドレス毎のミス数を集計.
(2) 計測時のブロックサイズ，セット数から，セット毎のブロックアドレス（(1) で集計したミス数の情報を含む）の集合を生成.
(3) 生成した集合毎に最大ミス数を見つける.
(4) 最大ミス数の 1/2 以上のミスを発生させるブロックアドレスをスラッシングによるミスが多発するブロックアドレスとしてマークし，そのリスト生成.
(5) 生成したリストをミス回数で降順ソートし，上位から計測時の Way 数分を除外.

以上の手順を経て完成したリストは，バッファへ挿入すべきメモリブロックのアドレスのリストである．本稿では，プログラム実行前にこのリストが指すメモリブロックのフィル先指定ビットが既に ‘バッファへフィル’ に設定されていることを前提にして，性能評価を行う．

4. 予備評価

本節では，これまでの節で述べた手法でどの程度のキャッシュミス率が削減可能かを予備評価する．

4.1 評価環境

評価は SimpleScalar/ARM[1] を用いてデータキャッシュシミュレーションを行いて計測した．事前実行によるミスブロックアドレス列の取得も同様に SimpleScalar/ARM を使用している．

評価したデータキャッシュ構成は 4KB-4WAY，2KB-4WAY，1KB-4WAY，512B2WAY，の 4 種類であり，全構成においてブロックサイズは 16 バイトとした．性能評価を行う個々のキャッシュ構成にはデータバッファと FTS をシミュレーションモデルとして実装した．SimpleScalar はメモリコントローラ以降の主記憶アクセスをシミュレーションしていない．このため性能評価実行シミュレーションでは，理想的にフィル先情報がミスハンドリング時に到着することを想定している．

計測したベンチマークアプリケーションは MiBench Version 1.0[2] である．実行したバイナリは MiBench の HP[3] より ARM 用プリコンパイルバイナリを取得した．この中から，元来のデータキャッシュミス率が低いため，最小構成のキャッシュで変化が現れないアプリ

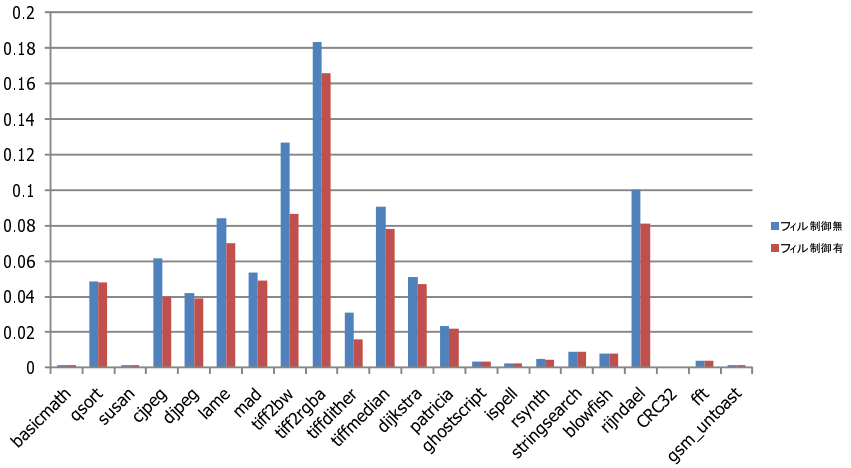

図 4 4KB-4WAY 構成時の提案手法によるミス率の変化.

ケーションを除外した．評価対象となったバイナリは，automotive から 3，consumer から 8，network から 2，office から 4，security から 2，telecom から 3 の計 22 アプリケーションである．全てのアプリケーションの入力データは large を選択した．

4.2 評価結果

通常実行時のデータキャッシュのミス率と，提案手法適用時のデータキャッシュのミス率を 4 キャッシュ構成に毎に図 4〜図 7 に示す．各図は横軸がアプリケーション，縦軸がミスの割合である．1 組の棒は左側がフィル制御無の場合のミス割合，右側が 1 ブロックのデータバッファを備えたフィル制御有の場合のミス割合を示している．

図 4 の 4KB-4WAY キャッシュ構成で平均 10.06%，図 5 の 2KB-4WAY キャッシュ構成で平均 18.93%，図 6 の 1KB-4WAY キャッシュ構成で平均 15.53%，図 7 の 512B-4WAY キャッシュ構成で平均 14.28%のキャッシュミス率削減効果が観測された．全 4 構成の平均削減率は 14.70%であった．最大の削減率を示したのは，1KB-4WAY 構成の rijndael で 65.55%のキャッシュミス率削減を達成している．唯一，1KB-4WAY 構成の dijkstra のみがフィル制御時にミス率を上昇させる結果となった．11.77%ミス率が上昇し，性能が低下している．この dijkstra のミス率上昇に対する議論は図 8 用いて後で詳しく行う．当初，キャッシュサイズの縮小化を進めた場合，スラッシング発生確率が上がり，削減割合が増えると予想していた．調査した各キャッシュ構成間である程度の傾向はみられるものの，キャッシュサイズの縮小に従い削減率が上がる結果とはならなかった．これはセット数を減少させた場合に，イ

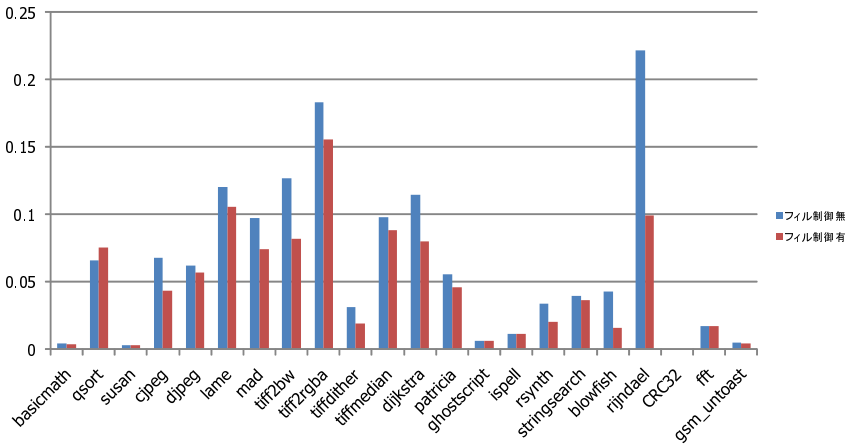

図 5 2KB-4WAY 構成時の提案手法によるミス率の変化.

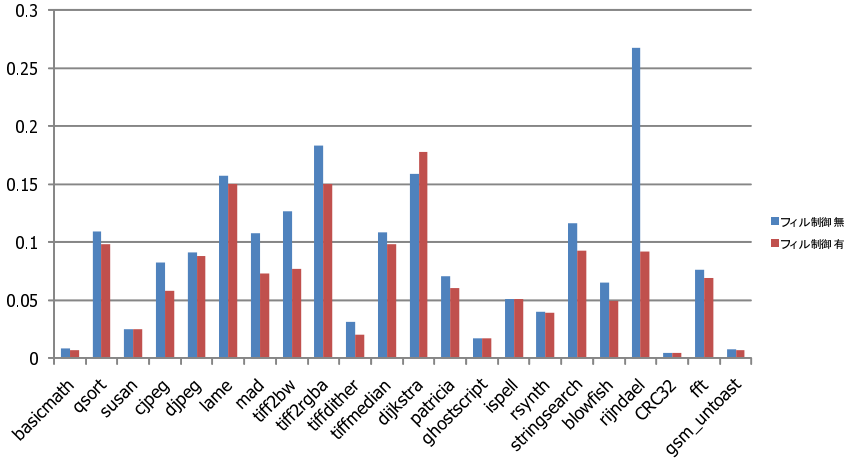

図 6 1KB-4WAY 構成時の提案手法によるミス率の変化.

Vol.2010-ARC-191 No.1
2010/10/18

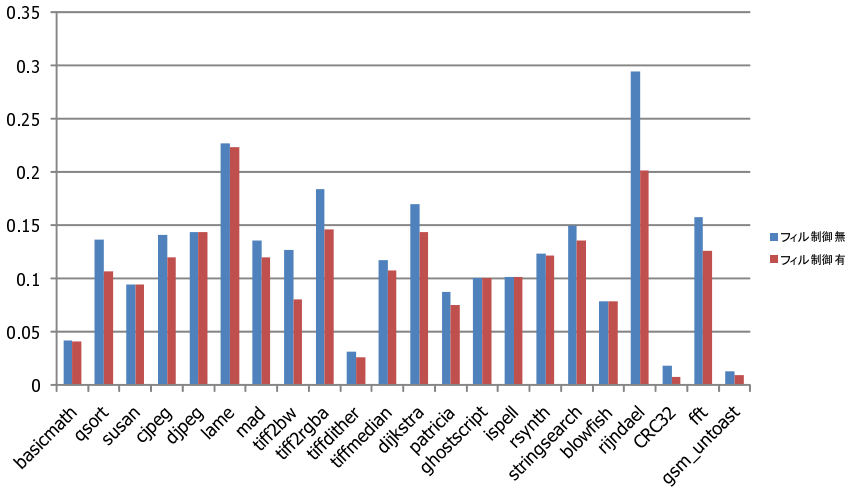

図 7 512KB-4WAY 構成時の提案手法によるミス率の変化.

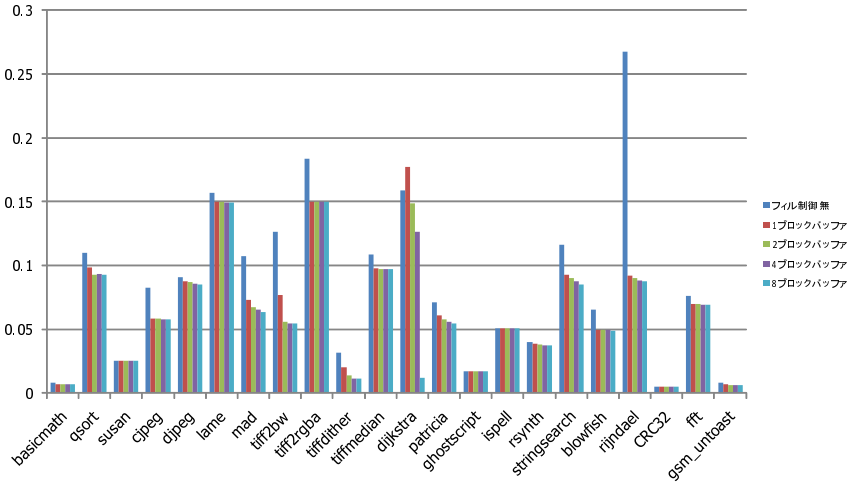

図 8 1KB-4WAY 構成時のバッファブロック数増加によるミス率の変化.

ンデックスの競合関係が変化するために，このような結果になったと考えられる．しかしながら，平均で約 15%のキャッシュミス率削減は小規模キャッシュにおいてフィル制御機構を導入する十分な動機づけとなる．

図 8 に 1KB-4WAY 構成のバッファのブロック数を変化させた場合のミス割合の変化を示す．棒は左からフィル制御機構を用いない場合，1 ブロックのデータバッファを持つ場合，2 ブロックのデータバッファを持つ場合，4 ブロックのデータバッファを持つ場合，8 ブロックのデータバッファを持つ場合のミス割合を示している．2 ブロック以上のデータバッファはフルアソシアティブで参照され，LRU で置換される．

dijkstra を除いて，通常実行時のミス率から一番大きな変化を見せるのは 1 ブロックのデータバッファを追加した場合である．2 ブロック以上のデータバッファはミス率削減効果は低いことがわかる．ハードウェアコストを鑑みた性能効率の面から予備評価を解析した場合，dijkstra の 8 ブロック時の劇的なミス率減少は注目に値するが，このような劇的な減少が観測されたのは dijkstra のみである．これは，全アプリケーション平均で考えた場合，ハードウェア効率としては低いことから，ハードウェア効率のみを考えた場合には，データバッファは 1 ブロックで十分であると結論付けることができる．

その一方で，キャッシュ性能低下の可能性を減らすという指標でバッファブロック数を考察することができる．この指標でデータバッファを 2 ブロック以上に増加させる意義を持つ良い例が dijkstra である．dijkstra における 1 ブロックデータバッファのキャッシュ性能はフィル制御を行わないキャッシュ性能に劣るが，2 ブロック以降の結果ではキャッシュ性能向上に貢献している．小容量のバッファ内でスラッシングが発生した場合，キャッシュ内でスラッシングを起こしていた時と比べ，スラッシング関係にあるブロックの生存時間を短くする可能性が有る．配列アクセスに纏わる空間的局所性など，ブロック内でごく短期間に頻発する参照を扱うことができないほど，バッファ内でのブロックリプレースが早い場合にミス回数を増加させる可能性が有る．このため，スラッシング解析でスラッシング関係にあるブロックアドレスを正確に解析できたと仮定しても，キャッシュ性能低下を招く可能性が有る．dijkstra の例では，1 ブロックデータバッファではバッファ容量が不十分であることを示しており，十分なバッファ容量を提供することで，フィル制御でキャッシュ性能低下につながるアプリケーションを少なくすることができることを示している．結論として，今回の予備評価の環境から，キャッシュ性能低下を完全に消すためにはには，データバッファは 2 ブロック以上であることが判った．

Vol.2010-ARC-191 No.1
2010/10/18

5. 関連研究

本稿では，キャッシュのフィル制御とデータバッファを組み合わせたデータキャッシュメモリステムの提案とフィル先を静的に決定するためのミスアドレス解析アルゴリズムを提案している．本研究の関連研究を 2 つ挙げる．

1 つ目は victim cache[4] である．victim cache はキャッシュからリプレースされたブロックを一時的にバッファ(victim cache) に退避する．その後，victim cache 内でヒットした場合ブロックはキャッシュに戻される．このキャッシュシステムは victim cache 容量分の記憶領域を仮想的に可変長キャッシュの WAY としてセット間で共有することに等しい．これは特定セットのスラッシング回避に小容量の記憶領域で対応する上で有効である．本提案が victim cache と本質的に異なる点は，スラッシングの回避を目的とするのではなく，緩和を目的としている点である．victim cache がスラッシングを回避する際には，victim cache の容量が重要となり，victim cache の容量を上回るようなスラッシング関係ブロックを扱う場合，スラッシングは回避されない．本提案はあらかじめ，スラッシング関係となるブロックを知っておくことで，'キャッシュに入れない' 制御をおこなう事により，スラッシングの回避を諦めており，その代わりに，最低限のヒットを保証することを目的としている．このため，本手法においてバッファの容量は重要ではない．バッファを用いた競合性ミス回避と言う点で，victim cache は一見本手法と同じ目的の手法に見えるが，本質的な部分で目的が異なるため，例えば，本手法と victim cache を組み合わせることも可能であると言える．

2 つ目は Time Based Load Filter(TBLF)[5] である．TBLF は Load Buffer(LB) と呼ばれる L1 と同時参照可能なバッファを用いる．LB は victim cache と異なり，リプレースされたブロックではなく，ミスによって主記憶からロードしたブロックを一時格納する．その後，ミスにより LB からブロックが追い出される時に，追い出される対象のブロックと当該ブロックが対応するキャッシュ内のセットに有るブロックとでリプレース対象を決定する．リプレース対象ブロックの決定は各ブロックが持つアクセス時刻を記録するタイムスタンプによって Access Interval が計算されることにより，行われる．victim cache と比べた場合の TBLF の最大の特徴はバッファを利用することにより，' キャッシュに入れない' という選択肢が増えたことである．LB から追い出される時にリプレース対象が LB から選ばれれば，当該ブロックがキャッシュに入ることは無い．この点が本稿の提案と共通している．しかしながら，本提案はスラッシングのみに着目することによって，事前実行によりスラッシング関係にあるブロックを解析することができ，この情報を本実行にフィードバックすることによって，従来のキャッシュに大幅な修正を加えずに，小規模のハードウェアを付加することによってフィルタを実現している．一方，TBLF ではフィルタを実行する際にタイムスタンプ等のブロック毎の記憶領域を必要とし，かつ，LB とキャッシュ間の比較回路等を必要とし，従来のキャッシュシステムに大きな修正を加えている．

6. おわりに

本稿は，データキャッシュと小容量のデータバッファを用いて，メモリブロックのフィル先をキャッシュまたはデータバッファのどちらかに限定することにより，スラッシングを緩和する手法を提案した．また，フィル先を静的に決定するために，スラッシング関係にあるブロックアドレスを事前実行により得たアドレストレースから生成するアルゴリズムを提案した．

本稿の予備評価では，提案手法を SimpleScalar/ARM シミュレータを用いて MiBench Version 1.0 の中から 22 アプリケーションを選出し評価を行った．1 ブロックのデータバッファで評価した結果，平均で 14.70%，最大で 65.55%のキャッシュミス率削減効果を観測した．

今後は，関連研究で紹介した他手法との性能比較を行う予定である．

謝辞 本研究の一部は科学研究費補助金若手研究（B）(20700045)「低消費電力高機能リコンフィギュラブルメモリシステムの研究」の一環として行なわれた．

参考文献

1) http://www.simplescalar.com/v4test.html
2) Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, Richard B. Brown, “MiBench: A free, commercially representative embedded benchmark suite”, Proc of IEEE International Workshop on In Workload Characterization, 2001 (WWC-4).
3) http://www.eecs.umich.edu/mibench/
4) N.P.Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”, Proc. of Intl. Symp. on Computer Architecture, pp.364–373, 1990.
5) 檜田 敏克，他，“キャッシュラインの時間情報を利用する Time Based Filter の提案”，情報処理学会 研究会報告（ARC-152），Vol.2003, No.27, pp.97–102, 2003.