情報処理学会論文誌 Vol. 52 No. 12 3160–3171 (Dec. 2011)

組込みプロセッサ向けキャッシュフィル制御方式

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，組込みシステムが扱うタスク数およびデータセットは大規模化している．これら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キャッシュメモリのフィル制御方式を提案し議論する．提案するフィル制御方式は小規模キャッシュメモリの参照局所性に合わせてフィル先（通常のキャッシュ/バッファ）の選択を行う．これにより，小規模キャッシュメモリで大規模なプログラムバイナリや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．

Cache Fill Control Method for Embedded Processors

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

Recently, the number of tasks and data sets that are dealt with in embedded systems are increasing as embedded applications are enlarged. It is difficult to implement large cache memories in accordance with these applications, since embedded systems are often required to be inexpensive. In this paper, we propose and discuss control of filling for cache memories with a small-sized buffer. The proposed method selects fill targets (conventional cache memories or small buffers) by knowing access locality in the caches. This can alleviate performance degradation when small cache memories deal with large program binaries and data sets.

1. はじめに

比較的高速な組込みプロセッサの登場により，組込みアプリケーションが大規模化している．近年の組込みシステム開発では組込み/リアルタイム OS を利用することが一般的になりつつあり，このことからもバイナリサイズは増大傾向にある．また，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傾向がある．このような状況では，上昇した動作周波数を活かすためにメモリアクセスレイテンシを隠蔽する大規模なキャッシュメモリが必要とされる．しかしながら，肥大化するバイナリやデータセットに追従してキャッシュメモリの容量を増加させることは，歩留まり低下を招き，最終的には製品コスト上昇につながるため，組込みプロセッサの設計方針としては採用し難い．

限られたキャッシュ容量でヒット率を向上させる方策としてコードサイズを削減することがあげられる．その 1 つに，異なる命令長（16 ビット，32 ビット）が混在する RISC 命令セットアーキテクチャの使用がある（この種のプロセッサとして，SH-2A [1] や ARM [2] が存在する）．これは 32 ビット命令による実行の高効率と 16 ビット命令によるコード密度効率を両立させることが狙いであるが，16 ビット命令を使用した部分の実行効率が低くなることは避けられないことが確認されている[3]．

もう 1 つのコードサイズ削減方針として，コード圧縮技術の利用がある．組込み向けコード圧縮方式として様々な研究が存在する[4),5)]．高い圧縮率のアルゴリズムの使用によりコードサイズ削減が得られる反面，実行時に復元処理のオーバヘッドが存在するため，参照頻度の高いキャッシュ内の命令やデータには適用困難である．

キャッシュミスを低減させるために，ARM ファミリの中には通常のキャッシュ動作ではなく，プログラムから制御可能な高速なローカル密結合メモリ（TCM: tightly coupled memories）を持つものがある．同様に Cell BE の SPE が持つローカルストア（LS）[6] はプログラム制御可能な高速メモリである．このようなスクラッチパッドメモリはプログラマによる明示的なプリロードを前提とするため，通常のキャッシュと比較し，キャッシュミスという予測困難なレイテンシ増大は防ぐことができるものの，プログラマの負担が存在する．

また，高連想度キャッシュを提供することによりキャッシュミス削減を狙うプロセッサが存在する（StrongARM [7]（32-way），XScale [8]（32-way）等）．高連想度キャッシュでは高速タグ検索を可能とするために CAM を使用することになり，ハードウェアサイズおよび電力消費の観点からは低コストプロセッサでは採用困難である[9]．

本研究はキャッシュミス数を減少させ，キャッシュミスペナルティを増やさないようにするための素案である．キャッシュミスペナルティを削減することでプログラムが完了するまでのクロックサイクル数を削減することができる．本研究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する場合に，それを

†1 北陸先端科学技術大学院大学情報科学研究科
School of Information Science, Japan Advanced Institute of Science and Technology

緩和する手法の提案を行う．提案手法は 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すべ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．提案するキャッシュメモリはこれまでに紹介した小規模キャッシュの有効利用の手法にくらべ，ハードウェアの規模およびオーバヘッド，プログラマへの負荷等の観点から実際の組込みシステム開発への導入が容易である．

本稿は以下の章で構成される．2 章では本研究のアプローチを明確化し，メモリアクセストレースの解析手法と解析後の情報の利用方法について議論する．3 章では，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．4 章で提案手法の効果を示すための評価を行う．6 章で提案手法の関連研究を述べる．最後に，7 章で結論を述べる．

2. 事前実行によるトレース解析とフィル先選択情報の生成

本研究はスラッシングの原因となるブロックの一部を “キャッシュに入れない” ことで，最低限キャッシュに格納されているブロックのヒットを保証することにより，性能低下の緩和を狙う．“キャッシュに入れない” ということは，キャッシュに格納する前（フィル時）にキャッシュに格納すべきか否かを判断するということになる．これは当該ブロックがキャッシュにフィルされた場合の，いわば未来の振舞いを事前に知っているからこそ可能となる．

キャッシュブロックの振舞いを事前に知るために，本研究では事前実行によって得るメモリアクセストレースを利用するアプローチを採用する．多くの組込みシステムでは，システムのハードウェア構成および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．また，組込みシステムは汎用システムに比べソフトウェア規模が小さい傾向にあり，開発者は製品出荷前に十分なソフトウェアチューニングを行う傾向にあることから，本アプローチは十分な妥当性を持つ．

本章では，メモリアクセストレースの収集法およびその解析について議論する．メモリアクセストレースを解析することにより “キャッシュに格納すべきでないブロック” のブロックアドレスを知ることができる．また，後の 3 章でそのブロックアドレスを用いてキャッシュフィル制御を行うハードウェアを示す．本章で示す解析と 3 章で示すその解析情報を利用するキャッシュハードウェアにより，本稿で提案するフィル制御が実現する．

2.1 メモリアクセストレース解析

まず，事前実行でメモリアクセスのトレースを取得する必要がある．しかしながら，比較的規模の小さい組込みプログラムであっても，命令/データすべてのメモリアクセスのトレースは膨大な量となり，それを記録として残すことはきわめて難しい．そこで，本研究はメモリアクセストレースをキャッシュミス発生時のトレースにのみ限定し，着目した．本研究ではこのトレースは SimpleScalar 4.0 コードベースの CPU シミュレータ SimpleScalar/ARM [11] を用いてキャッシュメモリシミュレーション行い取得した．さらにこの中からスラッシングに関係するトレースに候補を絞るために，ブロックアドレスごとにミス回数を集計し解析する．

ブロックごとのミス数の一例を**図 1**，**図 2** に示す．使用したベンチマークアプリケーションは MiBench Version 1.0 [12] の basicmath である．図 1 は 512 B-4Way で命令キャッシュを構成したときの各ブロックアドレスに対するミス数を示している．縦軸はミス数で，横軸は各ブロックアドレスを示している．収集できるミスブロックアドレスに連続性は保証されないので，横軸はアドレスで連続していない．図ではブロックアドレスを昇順でソートして

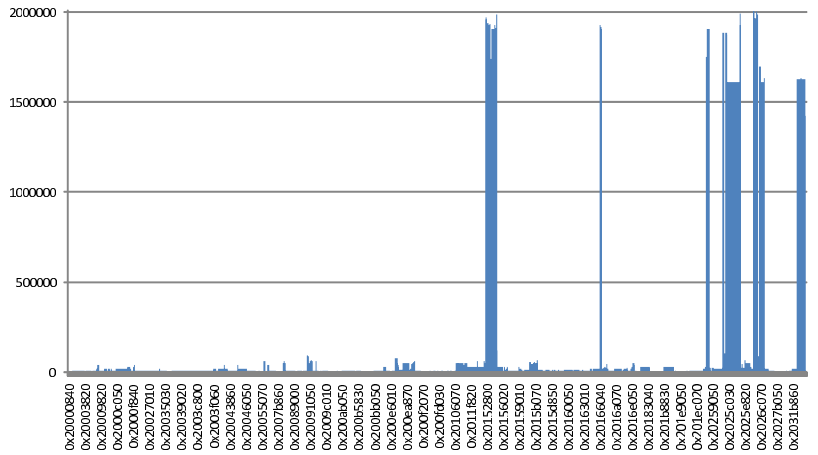

図 1　512 B-4WAY 構成命令キャッシュの basicmath におけるブロックアドレスごとのミス数
Fig. 1　Miss rates of each block address in 512 B-4WAY cache for basicmath.

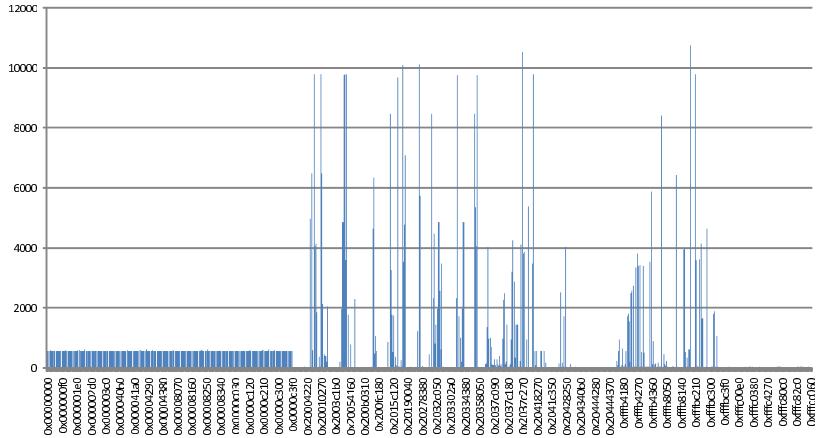

図 2　4 KB-4WAY 構成データキャッシュの basicmath におけるブロックアドレスごとのミス数
Fig. 2　Miss rates of each block address in 4 KB-4WAY cache for on basicmath.

表示している．図 1 の大きな傾向として，ほとんどのブロックは 100〜2,000 以内のミス数に収まっているのに対し，一部のブロックのみが 1,500,000 回以上のミスを発生させていることが分かる．図 2 は 4 KB-4Way でデータキャッシュを構成したときの各ブロックアドレスに対するミス数を示している．たいていのミスブロックアドレスは 4,000 回以内のミスに収まっているにもかかわらず，ごく稀に 10,000 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分かる．図 1，図 2 を通して，ミス数の高いブロックは，リプレースされやすいにもかかわらず，リロードされる頻度が高いため，発生している．

本研究では，こ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．“キャッシュに入れないブロックアドレス” はこのスパイクを示すブロックアドレスから選ばれる．これらスパイクを持つブロックアドレスを各セットごとにインデックス番号で集合化する（集合はセット数分存在する）．このようにしてできたブロックアドレス集合はそれぞれがスラッシング関係にあるブロックアドレス集合となる．この集合内からウェイ数分のブロックアドレスを除外した残りのブロックアドレスを “キャッシュに入れないブロックアドレス” とする．この考え方で最も大事な要素は “集合内からウェイ数分のブロックアドレスを除外” することである．ここで除外したメモリブロックはキャッシュ内でスラッシングから守られる，すなわちヒットを保証することとなる．

後の 3 章で詳しく紹介するが，キャッシュに格納しないメモリブロックは完全にノンキャッシュリードされるわけではく，ごく小規模なバッファに格納し，そこから読み出される．これ以降，本稿ではこのキャッシュに入れないメモリブロックを指すブロックアドレスをバッファに格納するブロックアドレスと呼称する．

2.2　フィル先選択情報の生成

本稿の評価では，バッファに格納するブロックアドレスを解析する際，トレースを入力とした解析プログラムで自動計算した．その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ごとのミス数を集計（図 1，図 2 で示したデータを作成）．
(2)　事前実行においてトレース収集したときのキャッシュ構成の情報（ブロックサイズ，セット数）をアルゴリズムのパラメータとして用いて，セットごとのブロックアドレスの集合を生成．
(3)　生成した集合ごと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，そのリストを生成（スラッシング関係にあるブロックアドレス集合の生成）．
(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（スラッシングが原因で最も多いミスを発生させたブロックアドレスのヒットを保証）．

以上の手順を経て完成したリストは，バッファへ格納するブロックアドレスのリストである．本稿で提案するハードウェアはこのブロックアドレスリストによりフィル時にキャッシュへ格納するかバッファへ格納するかを判断することになる．

3.　提案キャッシュメモリシステムとフィル制御方式

本章では，バッファへ格納するブロックアドレスのリストを扱うキャッシュメモリシステムのハードウェアとその動作を述べる．

3.1　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 3** 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命令フェッチもしくはロード/ストアを実行すると，最初に命令/データキャッシュを参照し，キャッシュメモリ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，キャッシュメモリと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，キャッシュメモリ/バッファから命令実行ユ

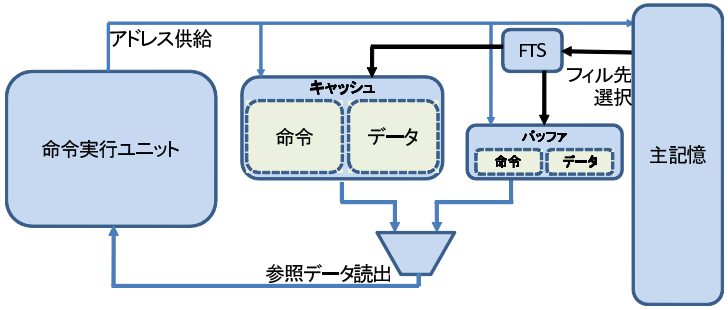

図 3　提案するキャッシュメモリシステムの概要図
Fig. 3　An overview of proposed cache memory system.

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，主記憶からメモリブロックをロードするが，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キャッシュメモリ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．FTS が選択先を選ぶ根拠として使用するのはバッファへ格納するブロックアドレスリストである．

3.2　FTS の実装

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である．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を見てフィル先を決定する機能を持つ選択回路となる．これ以外の FTS の実装に関する考察は我々の先行研究ですでに発表した[10]．

提案するキャッシュメモリシステムのフィル対象はキャッシュメモリとバッファの 2 種類であるため，各ブロックにつき 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，たとえば 16 バイトキャッシュブロックで 0.8%程度，さら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，1 回のバースト転送で 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る．このとき，データだけではなく，他にパリティビット等を付加して転送する方式が多い．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実現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，フィル先情報は実行バイナリと同位置に用意され，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．

3.3　FTS によるスラッシング緩和の例

これまで，2.2 節の手法で得た情報を基に，FTS がフィル先を選択することを述べた．提案手法のスラッシング緩和の例を**図 4** 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 回ずつ繰り返し参照される状況を示している．キャッシュメモリのウェイ数は 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，キャッシュメモリは 1 度もヒットしない．この状況を一般的にスラッシングという．

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ても，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，たとえば，キャッシュの空間的局所性に代表されるような，ご

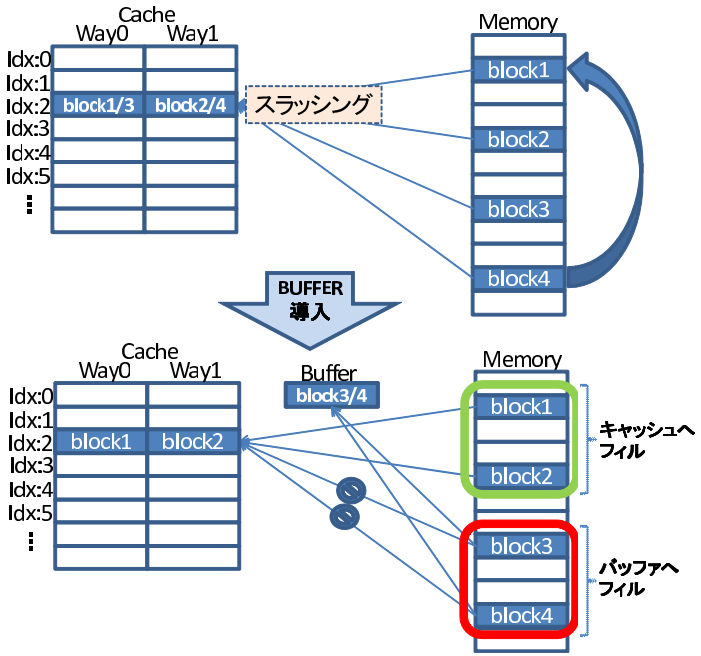

図 4　小容量バッファを用いたスラッシング緩和の概要図
Fig. 4　An overview of alleviation of slashing using a small buffer.

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．

4. 評　　価

本章では，提案手法がどの程度のキャッシュミス率削減効果を持つかを評価する．

4.1 評 価 環 境

評価は SimpleScalar/ARM [11] を用いて命令・データキャッシュシミュレーションを行い，キャッシュミス率を算出することで行った．事前実行によるメモリアクセストレースの取得も同様に SimpleScalar/ARM を使用している．

16 バイトのブロックサイズを持つ命令キャッシュ/データキャッシュに対して，キャッシュサイズを変えてキャッシュミス率を評価した．ここで用いる “キャッシュミス率” という言葉はキャッシュ単体でのミス率を指すものでなく，バッファでのミスも含めたキャッシュシステム全体のミス率を示している．また，後の 4.2 節で示す比較対象手法のミス率算出に関しても同様で，victim cache で発生するミスはキャッシュシステム全体のミスとして換算してミス率を計算している．個々のキャッシュ構成にはバッファと 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 [12] である．実行したバイナリは MiBench の HP [13] 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ケーションを除外した．命令キャッシュの評価対象となったバイナリは，automotive から 3，consumer から 4，network から 2，office から 4，security から 2，telecom から 1 の計 16 アプリケーションである．データキャッシュの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．すべてのアプリケーションの入力データは large を選択した．

評価したキャッシュ構成はすべての評価で 4 ウェイセットアソシアティブ構成．命令キャッシュの評価では 512 バイト，1 K バイト，2 K バイトの 3 つの容量で評価した．データキャッシュの評価では，512 バイト，1 K バイト，2 K バイト，4 K バイトの 4 つの容量で評価した．すべてのキャッシュ構成でバッファサイズは 1 エントリ（1 キャッシュブロック分）で評価した．これは後に示す評価結果から，少なくとも命令キャッシュにおいては 1 エントリで十分な効果を得ているためである．データキャッシュに関しては，さらに 1 KB 構成に対するバッファエントリ増加の効果の評価を付け加える．バッファエントリ増加の意義は評価を参照しながら考察する．

4.2 比較対象の手法

本稿では提案手法の性能の比較対象として victim cache [14] をあげる．victim cache の手法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）に退避させる．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ウェイ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うえで有効である．victim cache の詳細と本研究との定性的な違いは

後の 6 章で述べるが，本節では victim cache がなぜ本手法の比較対象としてふさわしいかを議論する．

3.3 節で示したとおり，本手法はスラッシング緩和の手法である．スラッシングは，1 つのセットをウェイ数を超える複数のメモリブロックが取り合う現象である．このとき，キャッシュとして投入した記憶領域はメモリブロックの時間的局所性にうまく対応できずに，本来，最低限提供しなければならない性能を下回り性能低下をもたらす．本手法はスラッシング要因となるメモリブロックを見つけ出し，“キャッシュに入れない” ことでスラッシングからキャッシュを守り，最低限の性能を保証することを目的としている．そのため，本手法の本質的な性能向上要因は “キャッシュに入れない” ことのみである．当然，“キャッシュに入れないと判断したメモリブロック” に対する参照はすべてミス扱いとなる．しかしながら，本研究の狙いはこのミス扱いをすべて被ったとしても，スラッシングから保護したメモリブロックのヒットを保証した方が，最終的な性能で上回るという予測に基づいている．

一方，最終的に提示した本手法はバッファを備える．本手法のバッファは時間的局所性に対応するためのバッファではない．バッファの存在意義は “キャッシュに入れない” と決めたメモリブロックの持つ空間的局所性にのみ対応し，救済することである．つまり，単純に時間的極性を有効に使いたいためにキャッシュ容量を増やす目的では配置されていない．一方，victim cache 方式は 1 度リプレース対象となったブロックを victim cache 内に格納し，victim ヒット時にキャッシュに書き戻す．つまり，victim cache を配置する目的は元来のキャッシュの持つ時間的局所性に対応する能力を補強することである．このように，記憶領域の用途・目的の観点から本手法のバッファと victim cache は異なる．本稿で victim cache を比較対象に用いた 1 つの目的は，想定する状況で，評価の公平性のためにキャッシュシステムすべての記憶領域の容量を同一にすることがである．キャッシュに必要な記憶素子容量を同一にすることで，キャッシュシステム全体での記憶素子の利用の効率性を比較することが可能である．また，2 つ目の目的はキャッシュ参照時の遅延量を同等にすることである．victim cache 方式は提案方式と同様に，キャッシュと並列に参照されるため図 3 で示したキャッシュシステムから命令実行ユニットへデータを供給するために追加するマルチプレクサも同様に必要となることから，キャッシュ参照時の遅延増加も同一となる．この 2 つの目的により，同じ記憶素子容量でかつ同じ参照遅延増加の比較対象であり，本手法の比較対象としてきわめてふさわしいといえる．

また，本節最初で述べたように，提案手法の大半の性能向上は “キャッシュに入れない” ことで得られるため，バッファをいかにヒットさせ有効に利用したかで性能差を測ることはできない．このことから，比較する双方の付加記憶領域（バッファおよび victim cache）のみのヒット率を比較する評価は行わない．このため，本章の評価は提案手法と比較対象手法のそれぞれの付加記憶領域のヒット判定を含めたキャッシュシステム全体のミス率で評価を行っていることに注意する必要がある．

4.3　評 価 結 果

4.3.1　命令キャッシュへ適用した場合の効果

図 5 に命令キャッシュに提案手法を適用した場合の効果と比較対象である victim キャッシュとの性能比較を示す．それぞれアプリケーションには 9 つの棒があり，9 つの棒は 3 つずつひとかたまりとなっている．それぞれのかたまりはキャッシュサイズで異なり，かたまりの中の 3 つの棒は左から通常キャッシュのミス率，victim 付きキャッシュのミス率，提案手法を備えたキャッシュのミス率となっている．かたまりの中の 3 つの棒は右に行くほど薄くなるように表示している．

まず，本稿の命令キャッシュの評価では全体を通して victim cache は大きくキャッシュミス数を減らすことはできていなかった．最も大きくミス数を減らしたもので，2 K バイト容量時での basicmath で約 6.6%の削減である．一方，提案手法は basicmath，susan，djpeg，lame，ghostscript，ispell，rsynth，blowfish，rijindael 等のアプリケーションの 512 バイト〜1 K バイト構成で劇的なミス率削減効果を観測できる．この中で最も大きなミス率削減効果を示したのは 1 K バイト構成時の rijindael であり，約 65%のキャッシュミスを削減している．

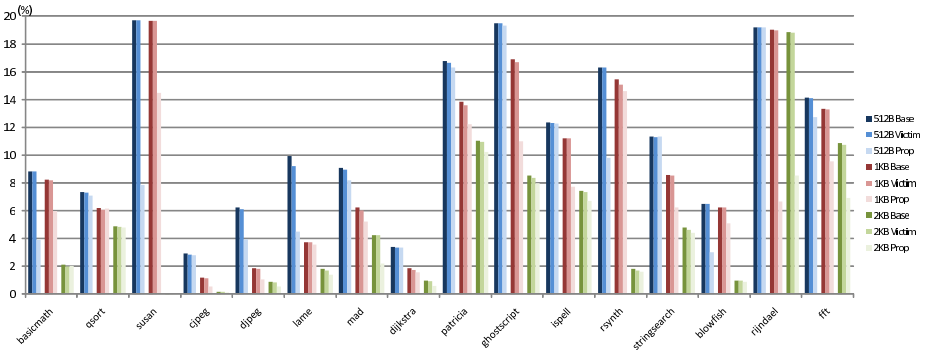

図 5　命令キャッシュミス率
Fig. 5　Miss rates of instruction cache.

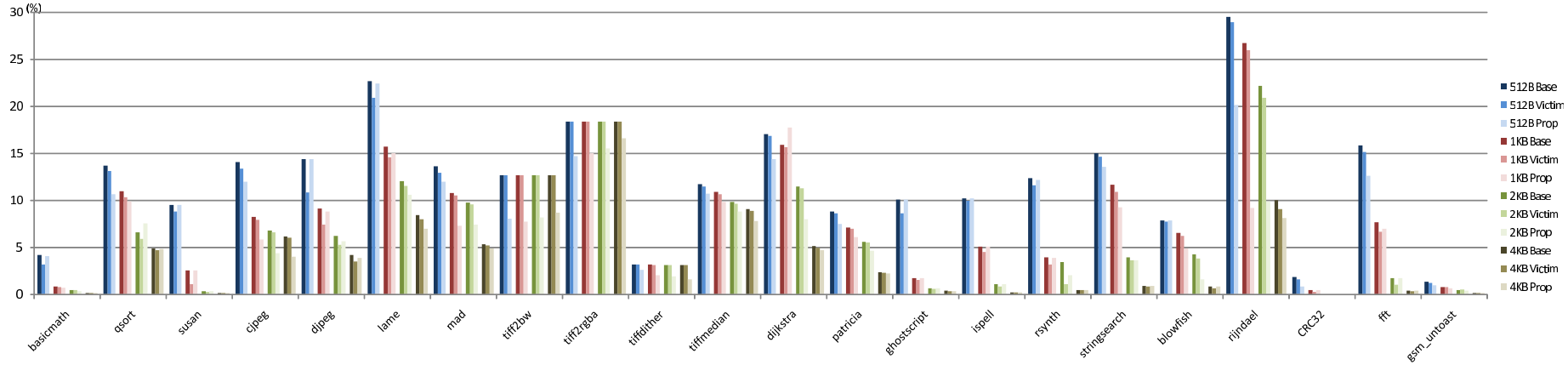

図 6　データキャッシュミス率
Fig. 6　Miss rates of data cache.

平均で，victim cache は 512 バイトのときに約 1.0%のミス率削減，1 K バイトのときに約 1.5%のミス率削減，2 K バイトのときに約 2.8%のミス率削減，全平均で約 1.83%のミス率削減効果となった．提案手法の平均は 512 バイトのときに約 20.8%のミス率削減，1 K バイトのときに約 25.8%のミス率削減，2 K バイトのときに約 19.2%のミス率削減，全平均で約 21.99%のミス率削減効果となった．

この結果から，提案手法が victim cache より同様のハードウェア資源を効果的に利用していることは明らかである．この効果は事前実行によるメモリアクセストレースの解析が大きく貢献している．この評価から，組込みシステムに小規模命令キャッシュの使用が強いられる場合，スラッシングを緩和するためのメモリアクセストレース解析は労力を支払う価値のあるチューニング作業であることが実証された．

また，本評価のキャッシュ構成はデータの見やすさのため，頻繁に採用される連想度である 4 ウェイセットアソシアティブのみで評価したが，これより連想度の低い構成での評価は先行研究ですでに行われ，ダイレクトマップと 2 ウェイセットアソシアティブ構成での評価では 4 ウェイセットアソシアティブと同様の傾向を持つという結果が得られている[10]．

4.3.2　データキャッシュへ適用した場合の効果

図 6 にデータキャッシュに提案手法を適用した場合の効果と比較対象である victim キャッシュとの性能比較を示す．図の見方は図 5 と同様であるが，4 K バイト構成時の結果があるため，棒のかたまりは 4 つあり，全部で 12 の棒がアプリケーションごとに存在する．

データの場合は命令の場合とは異なり，いくつか提案手法が victim cache よりキャッシュミス率削減効果で劣るアプリケーションが存在した．目立つのは，susan，djpeg，ghostscript，ispell，rsynth 等である．しかしながら，提案手法の効果が存在した場合の削減効果は命令キャッシュの場合と同様に大きく，特に rijndael 等は 65%以上のキャッシュミスを削減する効果を観測している．平均で，victim cache は 512 バイトのときに約 6.1%のミス率削減，1 K バイトのときに約 9.6%のミス率削減，2 K バイトのときに約 10.2%のミス率削減，4 K バイトのときに約 5.9%のミス率削減，全平均で約 8.00%のミス率削減効果となった．この削減効果は命令キャッシュに比べて大きい．提案手法の平均は 512 バイトのときに約 14.2%のミス率削減，1 K バイトのときに約 15.5%のミス率削減，2 K バイトのときに約 18.9%のミス率削減，4 K バイトのときに約 10.0%のミス率削減，全平均で約 14.70%のミス率削減効果となった．提案手法は個々で性能が victim cache に劣るアプリケーションやキャッシュ構成があるものの，効果が効いた場合の削減効果が大きいため，平均した場合にはやはり性能改善率が高い．

データキャッシュに関して提案手法の効果を結論付けるために，1 つ見落とすことのできない結果が dijkstra の 1 KB 構成時に出ている．この結果では通常のキャッシュのミス率より提案手法を備えたキャッシュのミス率が悪くなっている．提案手法が通常キャッシュより悪

くなる可能性は 2 つある．1 つ目はメモリアクセストレース解析アルゴリズムの問題で，明らかにキャッシュに入れておく方がよいメモリブロックをバッファに入れてしまった場合である．2 つ目はバッファ容量の問題で，バッファが該当メモリブロックの持つ空間的局所性に対応できないくらい早く追い出されてしまった場合である．2 つ目の状況はスラッシング状態であったときの方がこの空間的局所性に対応できるだけの存在時間を有しており，フィル制御によってそれが失われた場合に起こる．本評価のメモリアクセストレース解析アルゴリズムは全評価通して同一であり，性能低下が dijkstra 以外で観測されなかったことから，前者が原因であるとは考え難い．もし，後者のみが原因であれば，バッファのエントリ数を増加させることで，バッファ内のメモリブロックの生存時間を増加させ，空間的局所性に対応させることで性能改善が期待できる．そこで，本評価では 1 K バイト構成時でバッファエントリ数を 2，4，8 エントリと増加させて性能を再度計測した．

バッファエントリ数を増加させた場合のキャッシュ性能を**図 7** に示す．各アプリケーションに 5 つの棒があり，左からフィル制御なしのミス率，1 エントリバッファの場合のミス率，2 エントリバッファの場合のミス率，4 エントリバッファの場合のミス率，8 エントリバッファの場合のミス率を示している．最も注目すべきアプリケーションは dijkstra である．1 エントリバッファのときは通常キャッシュに性能が劣っているが，2 ブロックバッファ以降は性能が逆転している．これは事前の予想どおり，バッファの空間的局所性に対応する性能

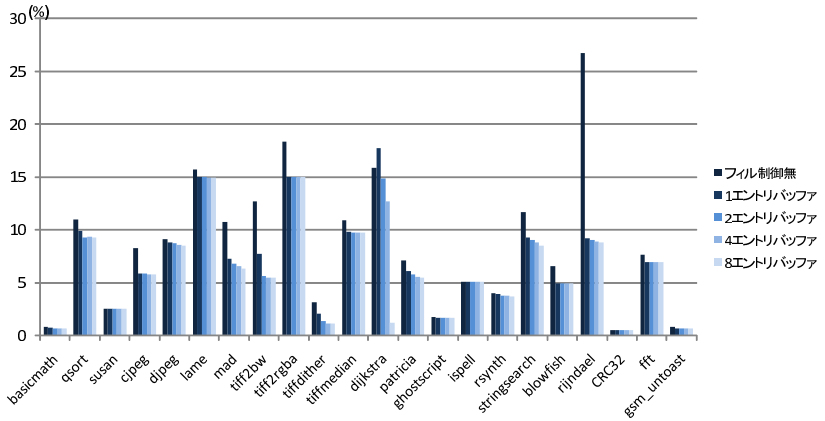

図 7　1 KB-4WAY 構成時のバッファエントリ数増加によるミス率の変化
Fig. 7　Variation in miss rates of 1 KB-4WAY cache by increasing buffer entries.

が性能低下要因となっていたことにほかならない．また，図から他のアプリケーションに対しても微量ながら性能向上に貢献していることが分かる．データ参照は命令参照と異なり，演算のオペランドとして要求される配列等が主記憶上に散在した場合に，同時に複数系列の連続したアドレスの参照が発生する場合がある．このような参照列が対象とするメモリブロックがスラッシング関係にあり，かつ，その関係を持つメモリブロックのフィル先がバッファに割り当てられた場合に，dijkstra のような結果をもたらすと考えられる．この解決のためには，本稿で示したバッファエントリ数を増加させるハードウェア的アプローチのほかに，解析アルゴリズムを洗練させるアプローチでも回避が可能である．

5. 回路面積および性能オーバヘッドの議論

提案方式は事前実行により得たフィル先情報を主記憶に格納し，キャッシュミス時にメモリブロックとフィル先情報を同時に読み出すことで，追加ハードウェアである FTS がフィル先（キャッシュor バッファ）を選択する．3.2 節では，提案手法を実現するための主記憶オーバヘッドは 1%以下であることを述べた．主記憶容量の増加のほかにも，CPU 内に FTS とバッファを追加し，さらに CPU と主記憶の間でフィル先情報を受け渡すハードウェア修正を加えなければならない．本章では，このハードウェア修正によって生ずる回路面積と性能のオーバヘッドを見積もり，議論する．

5.1 FTS 追加によるオーバヘッド

図 4 で示される FTS の内部を詳細に示す図を**図 8** に示す．FTS 追加のための回路修正はきわめて単純である．キャッシュメモリの書き込みデータ入力ポートへの配線を分岐してバッファの書き込みデータ入力ポートまで接続し，主記憶から送られる 1 ビットのフィル先情報を基に，キャッシュメモリとバッファへの書き込み許可信号を AND 論理でマスクするのみである．この修正に要する回路増加規模は配線レイアウトによるが，たとえば本研究の比較対象である victim cache に比べ，キャッシュとバッファの間の配線がなく[⋆1]，使用する配線数とトランジスタ数は少ない．実際に市販されている CPU で victim cache が十分に小さいハードウェアとして実装されている[16] ことを考慮した場合，FTS の実装面積的な実現可能性はきわめて高い．

FTS 実装時の性能オーバヘッドに関して，victim cache と比較した場合，FTS を追加し

⋆1 victim cache を採用した場合，本手法と比べ，キャッシュから victimcache 本体（本研究におけるバッファ相当）へのブロックデータの書き出し，victim cache からのキャッシュへのブロックデータの復帰の回路が必要となる．

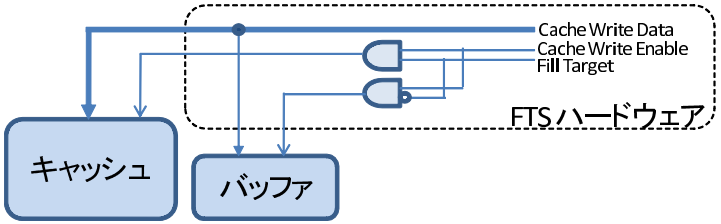

図 8　FTS 詳細回路図
Fig. 8　Detailed circuit schematic of FTS.

たことによる回路遅延増加はキャッシュミスハンドリング時の遅延にのみに影響する．バッファ/victim に関する遅延増加は同一サイズであることから，ほぼ同一である．FTS 追加により最大遅延が変動する場合，動作周波数の低下やミスハンドリング時のクロックサイクル数の増加の可能性が考えられる．この評価は実際に LSI で実装した場合にのみ得られるが，本研究ではアーキテクチャの性能をシミュレーションベースで評価したため，考察のみを行う．図 8 で示したとおり FTS はきわめて単純なロジックであり，配線のレイアウトの複雑さも victim cache に比べ複雑とはいえない．このことから，LSI 設計者にとって FTS の導入時の性能オーバヘッドの発生のリスクは低いと考えられる．

5.2　フィル先情報転送メモリインタフェース修正のオーバヘッド

次に，FTS と主記憶の間のハードウェア実装に関して議論する．3.2 節では，FTS と主記憶の間の転送でフィル先情報はパリティビット等と同様にキャッシュブロックのデータに付随する情報として扱われることを述べた．本節では，フィル先情報の転送を実現する具体的ハードウェアを例にあげ，提案手法の回路オーバヘッドを議論する．

従来の組込みシステムにおいて，主記憶は CPU パッケージ外に配置され，128 ビット以内のバス（フロントサイドバス等）で結合され，パラレル通信によりデータ転送が実現されていた．現在，一般的な汎用コンピュータシステムで採用されているメモリインタフェースは HyperTransport [17] や QPI [18] である．このようなメモリインタフェースは主記憶や他のプロセッサに対し PtoP（Point to Point）で接続し，かつ接続する配線数を極少数にし，動作周波数を上げることで，双方向シリアル通信に近づけた形で広帯域のデータ転送を実現している．これらの PtoP メモリインタフェースは 2001 年以降にハイパフォーマンスプロセッサ市場で登場し，10 年の歳月を経て広く普及した．それを受け，近年，ARM 社は組込みシステム向けの PtoP 接続インタフェースの AMBA4 [19),20)] を発表している．AMBA4 は Coretex A-15 プロセッサ[21)] 等で採用されている．PtoP 接続のメモリインタフェースの転送はフロントサイドバス接続の転送に比べて，動作周波数で優位なため，接続幅が狭くてもデータ転送能力に見劣りがなく，かつ，マザーボード上での CPU と主記憶間の配線数を削減することが可能となるため，組込みシステムのコスト削減につながることから，今後の組込みシステム設計で主流のメモリインタフェースとなることは間違いない．

これらの PtoP 接続のメモリインタフェースは，物理的な接続幅に関係なく，アドレス・コマンド等の制御情報，データ本体をフォーマット化し，パケットとし分割して扱い，データ転送を行う．たとえば，HyperTransport では，すべてのパケットは 32 ビットの固定長パケットで扱われる．1 回の転送は複数のコントロールパケットとデータパケットの集合を送信することで成立する．転送に必要な制御情報はコントロールパケットと呼ばれる集合の先頭のパケットに必ず格納される．コントロールパケットは最後の 8 ビットを用いて次に続く複数のパケットの種類と数を指定することが可能である．このため，HyperTransport では，制御情報と転送データのどちらも，実質，可変長の情報を転送することができる．

上記のことから，たとえば，HyperTransport のメモリインタフェースによってフィル先情報転送が実現される場合，フィル先情報はコントロールパケットのフォーマットに間借りする形となる．このため，フィル先情報を主記憶から転送する実装を施した場合の配線数の増加等はない．また，この特徴により，既存のハードウェアに対して行う修正が送信側と受信側のモジュールのみに限定される特性を得ることができる．本提案において，この 2 つの修正対象モジュールは CPU 内部に組み込まれるメモリコントローラと DRAM モジュールである．CPU の修正は提案のコア部分である FTS とバッファを組込むため，必然的に必要となる．DRAM モジュールに関しては，提案を実現するために新しく設計する必要がある．まず，メモリコントローラと DRAM 間のプロトコルを一致させる必要がある．さらに，フィル先情報を読み出し要求時に取り出し，転送時にそのフィル先情報を制御情報に付加する機能を有しなければならない．また，本提案は事前実行情報で得たフィル先情報をプログラムバイナリファイルに付加し，プログラムローダがプログラム開始以前に主記憶にセットしておくことを前提としている．これを実現するためにはソフトウェアから DRAM モジュールへフィル先情報を書き込む機能を提供しなければならい．このためには，新しく制御情報にコマンドを追加定義し，DRAM への特殊書き込みをプロトコルとして用意する[⋆1]

⋆1 “該当するメモリブロックアドレスの先頭に特殊書き込み扱いで（アドレス空間識別子を用いる等）フィル先情報の値をストアする命令を発行する” 等の実装が考えられる．

ことで実現することができる．

本提案実装時の転送オーバヘッドに関して，上述のとおり，1 つのコントロールパケットで転送できる有効な制御情報は 24 ビットである．そのため，制御情報は 24 ビットにパディングされる．もし，フィル先情報をコントロールパケットに組み込むことでパディング時に 24 ビットに収まらない場合，コントロールパケット数が 1 つ増加することになる．この場合に提案手法は転送オーバヘッドが生ずる．メモリアクセス時の制御情報のビット数はそのシステムで採用するアドレスビット数により大きく変動するため，このオーバヘッドが定常的に発生するか否かは実装依存となるが，本研究で提案するフィル先情報が 1 ビットのみであることを考えた場合，この可能性が低いことが分かる．

本節のこれまででは，PtoP 接続時の提案手法の実装に関して議論した．これ以降は，従来型のフロントサイドバスによる接続時の実装に関して議論する．この接続方式では，PtoP 接続とは異なり，コマンド制御，パリティビット，アドレス，データの接続が物理的に独立しており，それぞれの接続が独立した意味を持つ．このため，フロントサイドバス方式で本提案を実装する場合は，フィル先情報を独立した信号線として増設し，接続する必要がある．市販されているメモリコントローラでは，パリティ等付加情報は独立した信号線として存在するため[22]，フィル先情報もこれと同列に扱うことで実現できる．しかしながら，PtoP 接続と異なり，CPU，フロントサイドバス，メモリコントローラ，DRAM モジュール等，メモリ参照に関連するすべてのハードウェアの修正を要するため，既存部品の流用が困難になる点でシステムのコスト性を損なう可能性がある．このことから，本提案は今後主流となる PtoP 接続方式のプロトコルの拡張として実現した場合に，組込みシステムにとって有益となると結論付けることができる．

6. 関 連 研 究

本章では，提案手法と同じ目的を持つ他の手法を 2 つ紹介し，提案手法との違いを述べる．

1 つ目は victim cache である．victim cache の概要は 4.2 節ですでに述べた．本章では本提案との本質的な違いに着目して考察する．提案手法が 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，事前実行により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” 制御を行う．これはいわば，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の重要性は低い．バッファを用いた競合性ミス回避という点で，victim cache は一見，本手法と同じアプローチの手法に見えるが，本質的な部分で目的が異なる．

2 つ目は，Time Based Load Filter（TBLF）[15] 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すべてを一時格納する．その後，ミスにより LB からブロックが追い出されるときに，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あ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 から追い出されるとき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ない．この点が本稿の提案と共通している．しかしながら，本稿の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ごと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．このような大規模かつ複雑なハードウェアを要するキャッシュメモリシステムは組込みシステムには向いていないといえる．

7. お わ り に

本稿は，キャッシュメモリ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，特定メモリブロックのヒットを保証し，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．本稿の評価で，提案手法は命令キャッシュの場合で平均約 21.99%，データキャッシュの場合で平均約 14.70%のキャッシュミス率削減効果を持つことを示した．

提案手法の本質は 2 つある．1 つ目はキャッシュに入るべきではないブロックを事前に知ることで，キャッシュ内で頻繁に参照されるブロックが追い出されることを防ぎ，それらブロックのヒットを保証することである．2 つ目は空間的局所性にのみ対応する目的のバッファを持つことで，小さいメモリ領域を効率的に利用することである．提案手法はこの 2 つの特性をうまく組み合わせ，少ないハードウェア量で大きな効果を得ている．特定ブロックをキャッシュに入れないことで，頻繁に参照されるブロックのヒットの保証を行い性能向上を得る，その一方で，小規模バッファを用いることでキャッシュに入れないとして諦めたブロックのキャッシュミスの中で，最低限空間的局所性に関する参照に対してヒットを提供し，さらに性能向上を得ている．つまり，1 つ目の本質的な欠点を 2 つ目で補う形となっている．

本稿の評価では，提案手法をデータキャッシュに適用した場合の効果で，バッファサイズが 1 エントリの場合に，ブロックの空間的局所性が十分に活かされずにキャッシュミスが増加したアプリケーションが 1 つ存在した．そのアプリケーションの場合でも，バッファサイズを増加させればキャッシュミスを減少させることができることが確認されている．この例から，本手法をデータキャッシュに適用する場合にはバッファサイズの調整が重要な指標となることを議論した．

本手法の着眼点はキャッシュ資源としてハードウェアを投資した分だけの性能を得るというコンセプトに基づく．特にキャッシュメモリの場合はシステム上で動作するプログラムにより，必要十分な容量を提供することが理想であるが，容量が足りていない場合，その性能低下はリニアに低下せず，急激な性能低下を示すことがある．この特性はコストセンシティブな組込みシステム設計者にとって，ハードウェア投資量を決める際につねに悩まされる問題となる．本研究の最も重要な貢献はこの問題に対処する現実的な手法を提案したことである．本稿の評価で示したように，トレースの解析およびフィル制御は自動化が可能であり，プログラマやソフトウェア開発者の負担が小さい．また，既存のキャッシュハードウェアに対する追加および修正は非常に小さく，現実的なハードウェア規模および複雑性での実現が可能である．

今後の課題として，メモリアクセストレース解析アルゴリズムの洗練があげられる．本稿で示したアルゴリズムは最適な解を生成するとはいい難い．本稿のデータキャッシュにおける評価で示したように，提案手法は適用時にパフォーマンス低下をもたらす可能性を持つためである．このことから，今後はパフォーマンス低下を防ぐ保証をするアルゴリズムの考案が必要となる．特にデータ参照に関して，バッファのエントリ数や参照系列に着目し，バッファ等のハードウェア資源量やアクセスの系列に着目してさらに精細な解析を行うことが必要である．

本手法の比較として，一部のブロックをロックしリプレースを制御する手法があげられる．ロック手法は本手法と同様にスラッシング時のヒット保証を可能にすることができるため，比較対象として有力である．

また，システム開発者のために，多量のトレースをとらなくても一部のトレースをとることのみで同様の効果を得ることができることを示せた場合，それは本提案手法が積極的に採用されるための重要な指標となる．このため，今後，小さいワークロードでトレースをとり，そのトレースの解析結果を基に，実際のフルワークロードで実行した性能を示す必要がある．データ参照に関する解析は，ワークロードが大きくなる場合に，主記憶中に存在するメモリブロック数が増加するため効果はないが，命令参照に関しては，小さいワークロードとフルワークロードでバイナリ中に存在する命令は不変のため，小さいワークロードでのトレース収集で十分である可能性がある．

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」の一環として行われた．

参 考 文 献

1) ルネサスエレクトロニクス：SH-2A，SH2A-FPU ソフトウェアマニュアル Rev.3.00 (2005).
2) Seal, D., ARM Limited: *ARM Architecture Reference Manual, 2nd ed.*, Addison-Wesley (2000).
3) 笹山高志，田中清史：タスクの優先度を考慮したバイナリ最適化，組込みシステムシンポジウム 2009 論文集（ESS 2009），pp.127–132 (2009).
4) Wolfe, A. and Chanin, A.: Executing Compressed Programs on an Embedded RISC Architecture, *Proc. Intl. Symp. on Microarchitecture*, pp.81–91 (1992).
5) Lefurgy, C., Bird, P., Chen, I.-C. and Mudge, T.: Improving Code Density Using Compression Techniques, *Proc. Intl. Symp. on Microarchitecture*, pp.194–203 (1997).
6) Pham, D. et al.: The Design and Implementation of a First-Generation CELL Processor, *IEEE Intl. Solid-State Circuits Conference*, pp.184–185 (2005).
7) Montanaro, J. et al.: A 160-MHz, 32-b, 0.5-W, CMOS RISC Microprocessor.
8) Intel Corporation: Intel XScale Microarchitecture, Technical Summary (2000).
9) Veidenbaum, A. and Nicolaescu, D.: Low Energy, Highly-Associative Cache Design for Embedded Processors, *Proc. Intl. Conf. on Computer Design* (*ICCD*), pp.332–335 (2004).

10) 請園智玲，田中清史：組込みプロセッサ向け命令キャッシュ制御方式の検討，組込みシステムシンポジウム 2010 論文集（ESS 2010），pp.81–86 (2010).
11) available from ⟨http://www.simplescalar.com/v4test.html⟩.
12) Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. and Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite, *Proc. IEEE International Workshop on In Workload Characterization 2001* (*WWC-4*) (2001).
13) available from ⟨http://www.eecs.umich.edu/mibench/⟩.
14) Jouppi, N.P.: Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, *Proc. Intl. Symp. on Computer Architecture*, pp.364–373 (1990).
15) 檜田敏克ほか：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).
16) AMD Athlon™ Processor and AMD Duron™ Processor with Full-Speed On-Die L2 Cache, A White Papaer by AMD (2000), available from ⟨http://www.datasheetarchive.com/pdf-datasheets/Datasheets-1/DSA-4978.html⟩.
17) HyperTransport™ Technology I/O Link, A White Paper by AMD (2001), available from ⟨http://www.hypertransport.org/docs/wp/25012A_HTWhite_Paper_v1.1.pdf⟩.
18) Intel® QuickPath Architecture: A new system architecture for unleashing the performace of future generations of Intel® multi-core microprocessors, A White Paper by Intel (2008), available from ⟨http://www.intel.com/technology/quickpath/whitepaper.pdf⟩.
19) AMBA AXI and ACE Protocol Specification (2011), available from ⟨https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR500-DA-10033-r0p0-00bet0/IHI0022D2c_amba_axi_protocol_spec_beta.pdf⟩.
20) AMBA® 4 AXI4™, AXI4-Lite™, and AXI4-Stream™ Protocol Assertions, A User Guide by ARM® (2010), available from ⟨https://silver.arm.com/download/download.tm?pv=1092727⟩.
21) Brian Carlson, Going “beyond a faster horse” to transform mobile devices, A White Paper by Texas Instruments (2011), available from ⟨http://focus.ti.com/pdfs/wtbu/SWPT048.pdf⟩.
22) Intel® E7520 Memory Controller Hub (MCH) Datasheet, A Detasheet by Intel Corporation (2005), available from ⟨http://www.intel.org/Assets/PDF/datasheet/303006.pdf⟩.

(平成 23 年 3 月 1 日受付)
(平成 23 年 9 月 12 日採録)

請園　智玲（正会員）

昭和 53 年生．平成 21 年北陸先端科学技術大学院大学情報科学研究科博士後期課程情報システム学専攻修了．博士（情報科学）．平成 23 年より北陸先端科学技術大学院大学助教．プロセッサアーキテクチャ，組込みシステムの研究に従事．電子情報通信学会会員．

田中　清史（正会員）

昭和 46 年生．平成 12 年東京大学大学院理学系研究科情報科学専攻修了．博士（理学）．平成 13 年より北陸先端科学技術大学院大学准教授．並列計算機アーキテクチャ，プロセッサアーキテクチャ，メモリシステム，組込みシステムの研究に従事．IEEE-CS，ACM，電子情報通信学会各会員．