組込みプロセッサのたの命令キャッシュインデックス再配置手法

請 園 智 玲† 田 中 清 史†

一般的なキャッシュメモリはブロックアドレスの下位フィールドによって生成されるインデックスによってセット毎にアクセスを分散させている．しかしながら，インデックスによるアクセス分散は規則性が乏しく，しばしば競合性ミスを誘発し，十分なキャッシュ容量を備える場合にもメモリ資源を有効利用できない可能性が存在する．また，組込みプロセッサは資源制約により，十分なキャッシュ容量を提供することができずに，競合性ミスを発生しやすくなる．本研究ではキャッシュのセットのアクセス負荷分散に着目し，インデックス変換する機能を備えたアクセスの負荷分散ハードウェアをキャッシュミスハンドリング機構に追加することにより，セットのアクセスを平準化し，キャッシュ全面の記憶領域を有効利用する手法を提案する．

Reduction of Misses in Instruction Caches by Reindexing

TOMOAKI UKEZONO† and KIYOFUMI TANAKA†

In conventional cache memories, accesses are distributed among sets by indexes that correspond to a lower field of a block address. However, the access distribution by indexes exhibits poor regularity, which often leads to conflict misses. Therefore, even when large cache memories are used, it is difficult to achieve efficient usage of the whole memory resources. Especially, embedded processors cannot povide enough cache capacity of caches and would generate more conflict misses caused by resource restriction. This paper focuses on the access distribution among sets in caches, and proposes a technique that efficiently uses the whole cache memory resources by leveling accesses to sets, where the cache-miss handling mechanism involves access distribution hardware with re-indexing.

1. はじめに

最もキャッシュ容量を有効に利用できるキャッシュ構成の 1 つがフルアソシアティブ方式である．フルアソシアティブ方式は全てのキャッシュブロックを連想探索可能とし，更に LRU 置換方式の特性から最も有用なキャッシュブロックをキャッシュに残すことが可能である．この実装に関して，その回路規模と遅延量の大きさから一定サイズ以上のキャッシュを構成する場合には現実的ではない．この為，現在一般的なキャッシュメモリの構成はセットアソシアティブ方式が主流である．セットアソシアティブ方式では，ブロックアドレスの下位フィールドによって生成されるキャッシュインデックスによってセットを区別し，区別したセット内で連想探索を行う．一部の組込みプロセッサでは，高連想度キャッシュを採用することで，小さいキャッシュの面積を有効利用を実現している[1)2)]．しかしながら，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．[3)]

現実的な連想度のセットアソシアティブ及びダイレクトマップのキャッシュ全体のメモリ区画の有効利用がなされているかを考える場合，キャッシュインデックスによる各セットへの参照の分散が有効に働いているかを知る必要がある．明らかに参照の負荷が偏る場合は，有効に参照されていないセットが存在していることとなる．この状態ではプログラム実行中にキャッシュメモリの全区画を有効に利用しているとは言えない．

本稿では，セットで参照負荷が偏る場合に，特定のセットに集中した参照負荷を他のセットに分散する手法を提案する．提案手法は組込みシステム開発時のソフトウェアチューニング手法に位置づけられる．提案手法は特殊な参照インデックス分散ハードウェアと事前実行による参照アドレス解析により実現する．参照アドレス解析は命令キャッシュに限定した．命令アクセスとデータアクセスでは参照傾向が異なり，命令ア

† 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology

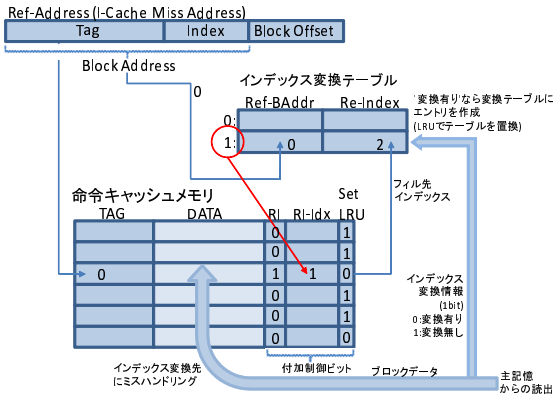

図 1 インデックス変換ハードウェアのミスハンドリング時の振る舞い.

クセスはデータアクセスに比べ参照傾向にランダム性が少なく解析が容易であることと，命令アクセスはアウトオブオーダ実行によるメモリレベル並列性を導入できず，キャッシュミス率が直接性能低下となる点にある．当然，提案するキャッシュ機構はデータキャッシュにも同様に適用できるが，以上の理由から，本稿では命令キャッシュへの適用のみを優先して評価した．

2 節では，解析結果を用いインデックス変換を行う付加ハードウェアの詳細を説明する．3 節では，2 節で紹介するハードウェアを利用するために事前実行で行う解析手法を示す．4 節では，提案ハードウェアが各セットの参照負荷を平準化した様子を示す．また，平準化の効果がキャッシュの性能にどの程度の影響を与えるかを示し，その性能と他の構成のキャッシュ性能と比べ評価する．5 節では他の命令キャッシュミス解消のための関連研究を示す．最後に 6 節で本稿の提案を結論づける．

2. インデックス変換手法

インデックス変換ハードウェアのミス時の動作を図 1 に示す．インデックス変換を行うために従来のキャッシュ機構に 2 つの機能を追加する．

1 つ目はインデックス変換テーブルである．インデックス変換テーブルは特定ブロックアドレス (Ref-BAddr) と変換後インデックス (Re-Index) の対を記憶しておくテーブルである．参照時は Ref-BAddr で連想探索し，ヒットした Re-Index を使用して命令キャッシュをアクセスする．インデックス変換テーブルは LRU で管理され，テーブルエントリを全て使用して更に新しいインデックス変換情報を加えるときは，最も最近までテーブルヒットされていないエントリと置換される．インデックス変換テーブルが置換により追い出される場合，キャッシュメモリの該当キャッシュブロックは無効化される．また，該当キャッシュブロックが他のキャッシュブロックによって追い出される場合も同様にインデックス変換テーブルを無効化する．このことから，インデックス変換テーブルエントリが指し示す先のキャッシュブロックが命令キャッシュ内に存在しない状態や，どのインデックス変換テーブルエントリも指示していないブロックが命令キャッシュに残り続ける状態は存在しない．

2 つ目はキャッシュへの制御フィールドの追加である．インデックス変換制御には各セット毎に 3 つの制御情報が必要である．インデックス変換でフィルしたことを示す RI ビット (1 ビット)，紐付けられたインデックス変換テーブルのインデックスを示す RI-Idx フィールド (図のインデックス変換テーブルエントリ数では 1 ビット)，直近でアクセスされていないセットを示す Set-LRU ビット☆．これら 3 つの制御ビットが追加される．提案するキャッシュメモリはインデックス変換テーブルの Re-Index のみではなく，通常のインデックスを用いたアクセスも行う．通常のインデックスでアクセスする際はインデックス変換でフィルされたブロックがヒットしてはいけないため，Re-Index アクセスでヒットすべきブロックと通常アクセスでヒットすべきブロックを識別する必要がある．RI ビットはその識別に使用する．キャッシュメモリがセットアソシアティブ構造である場合に，複数のインデックス変換テーブルエントリが同一インデックスを指す場合がある．この時，単一セットで tag が複数ヒットする可能性が発生するため，RI-Idx はそれを識別するために使用される．Set-LRU はインデックス変換テーブルエントリ生成時に Re-Index を決定するために使用する．本稿の提案は特定セットに集中するアクセスをアクセスの少ないセットに分散させることを目的とする．このため，インデックス変換テーブルエントリを作成するときにアクセスが少ないセットを変換先 (Re-Index) に選ぶ必要がある．本稿評価では，Set-LRU から Re-Index の選択を行う際には，Set-LRU が 0 である最も若いセット番号☆☆を選択するアルゴリズムで評価した．本稿ではこの指標として命令キャッシュへのアクセスの順序情報を採用し，擬似 LRU により最

☆ 図 1 では 1 ビットで特定時間内の参照有無のみを示す疑似 LRU 実装 (1 でアクセス済，全セットの擬似 LRU ビットが 1 に揃った場合に反転で全 LRU ビットが 0 になり，初期化される) で示されている．この疑似 LRU 実装はセット数が多くなるキャッシュ構成でハードウェア資源の節約になる

☆☆ 図 1 では上から 3 番目のセットで，インデックスが 2 のセット

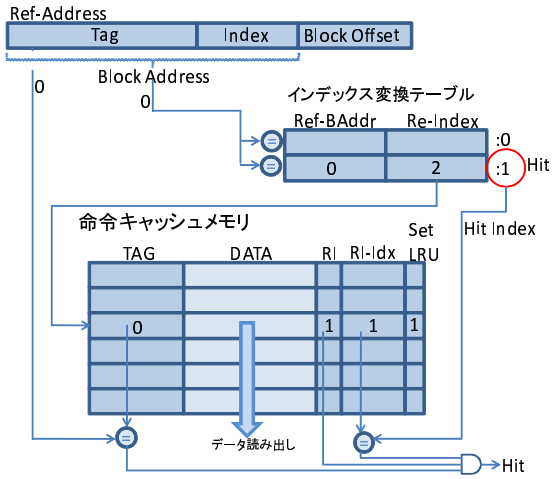

図 2 インデックス変換ハードウェアの参照時の振る舞い．

近までアクセスされていないセットを変換先として選択することにした．

本提案では，インデックス変換テーブルエントリを作成するミスハンドリングか否かは，ミスハンドリング時に主記憶から到着したキャッシュブロックに付加される 1 ビットのインデックス変換要否情報で決められる．この 1 ビットの変換要否情報の導入は，どのキャッシュブロックが競合性ミスを引き起こすかを予めシステム設計者が知っていることを前提としている．予め競合性ミスの原因となるキャッシュブロックが分かるとするならば，本方式のキャッシュを導入することで，通常のキャッシュアクセス速度をそのままに，その時に使用していないセットを有効に利用してキャッシュミス率を低下させることができる．この 1 ビットの変換要否情報の生成方法は次節で詳しく説明する．変換要否情報をどのように主記憶に配置し，メモリインターフェースを用いてどのように転送するかはこれまでの我々の先行研究で議論してきた[4)5)]．全てのメモリブロックに 1 ビット付加する実装は，16 バイトブロックのキャッシュメモリで主記憶の 0.8%程度の記憶領域オーバヘッドで実現可能であり，更にブロックサイズを大きくすることで，このオーバヘッドが縮小していくことが分かっている．更に当論文では，この 1 ビットを転送するためのメモリインターフェースの修正にも言及している．

Re-Index 機能参照を持つキャッシュはアクセスを逐次的に 2 段階で行う．インデックス変換テーブル参照で通常の参照の遅延を増加させないために，段階分けされている．1 段階目のアクセスではインデックス変換テーブルを参照しないため，通常の遅延で参照できる．通常の参照でキャッシュミスした場合にのみインデックス変換テーブル経由の参照が発生する．通常のキャッシュ参照中に 2 段階目の参照の準備であるインデックス変換テーブルの参照は完了しているため，この場合，実質インデックス変換テーブルの参照は隠蔽される．このことから，提案するキャッシュは 2 段階目のキャッシュ参照が要求される場合，直ちに次のクロックサイクルでキャッシュ参照を行える準備を整えることができる．本稿評価では，最終的にインデックス変換テーブル経由の参照でヒットした場合は，2 回分のキャッシュ参照コストのみが加算されるシミュレーションを行っている．

また，この 2 回分のキャッシュ参照のオーバヘッドを隠蔽するためにはキャッシュメモリのマルチポート化が有効である．通常経由とインデックス変換テーブル経由の参照を同時に発行できるキャッシュメモリを想定することで，オーバヘッドを削減することができる．

インデックス変換ハードウェアの参照時の動作を図 2 に示す．まず，命令キャッシュへのアクセスが発生した時に，変換されていない通常のインデックスをもって命令キャッシュにアクセスする．この時の検索対象は RI が 0 であるブロックのみである．その通常の命令キャッシュアクセスと同時にインデックス変換テーブルへのアクセスも行われる☆．この段階でキャッシュヒットすれば，それは通常のキャッシュヒット，両方ミスしたのであれば，通常のキャッシュミスである．もし，通常のキャッシュアクセスでミスし，インデックス変換テーブルがヒットした場合，Re-Index を用いて再度命令キャッシュにアクセスを行う．図 2 はこのときの状態を示している．Re-Index を用いた命令キャッシュ参照は通常時のアクセスと異なり，RI が 1 のブロックのみを対象として TAG 一致検索を行う．加えて，連想度が 2 以上の場合は，2 つ以上の RI が 1 のブロックが存在できるため，RI-Idx とインデックス変換テーブルのヒットしたエントリのインデックスを比較する (図 2 はダイレクトマップの例なので必要ないが，比較動作を行っている)．TAG，RI，RI-Idx が全て条件を満たした時に，インデックス変換後のキャッシュブロックがヒットすることとなる．

3. 事前実行によるインデックス変換要否情報の生成

一般的な組込みシステムでは組込まれたソフトウェ

☆ この時点では変換テーブルの参照のみが行われ，キャッシュメモリへの参照は行われないことに注意．

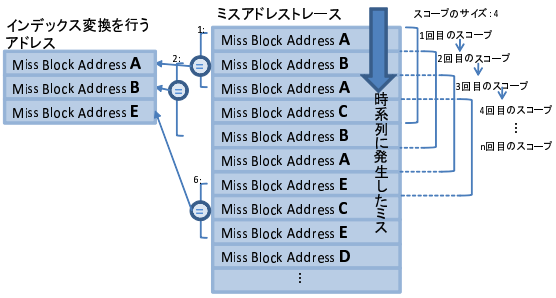

図 3 ミスアドレストレース解析の概略図.

アは更新されず，組込み対象のハードウェアが役割を終えるまで使用され続ける．このため，組込みソフトウェア開発は与えられたハードウェアで少しでも性能が発揮できるように慎重にチューニングされる．また，そのための人的コストは製品価値を高める上で支払う価値が有るとされる．本提案は組込みソフトウェア開発のチューニング段階の提案であると位置づけられる．2 節で示すハードウェアが提供される CPU を採用した際に，インデックス変換要否情報を組込みソフトウェアのバイナリと同じ扱いで用意しておくことで，性能向上を狙うことが出来る．

インデックス変換要否情報は完成したソフトウェアを事前実行し，解析することで生成する．生成には 2 ステップを要する．1 ステップ目は事前実行を行い，キャッシュミスが発生した時系列のブロックアドレスの列 (ミスアドレストレース) を取得することである．2 ステップ目はそのトレースを解析し，競合性ミスを発生させるブロックアドレスを抽出することである．

1 ステップ目のミスアドレストレースの取得は実機上でハードウェアデバッカを用いても，CPU シミュレータ等を用いてもよい．それらの何れかの手段で，対象とするキャッシュメモリで起きたミスの原因となった参照アドレスの列を取得しファイル化する．取得するアドレス列で必要な情報はブロックアドレスであり，この列は対象としているキャッシュ構成によって変化する．

2 ステップ目の解析は 1 ステップ目で取得したミスアドレストレースを用いて行う．ミスアドレストレースを入力として，解析し，最終的にはインデックス変換を行う (インデックス変換要否情報が 1 の) アドレスの列を出力する解析プログラムが処理を行う．ミスアドレストレースの解析の概要を示した図を図 3 に示す．

図のミスアドレストレースと示されているアドレス列は 1 ステップ目で取得した対象キャッシュでミスしたブロックアドレスの列である．図に登場するアドレスは A〜E の 5 つで，これらはミスが発生した順で時系列に並んでいる．このアドレスを時間的に古い順で，1 アドレスつずつ解析する．解析はそのアドレスから連続するスコープサイズ分のアドレスの一致を調べることで行われる．図の例ではスコープサイズは 4 である．1 回目の解析ではスコープ内に A，B，A，C のアドレスがある．この中で一致するのは A である．解析プログラムはこの A を抽出し，インデックス変換を行うアドレスのリストに追加する．2 回目の解析では，次のアドレス B から始まる 4 つのアドレス列に解析対象を移す．このとき 4 つのアドレス列は B，A，C，B であり，B が一致する．これも同様にインデックス変換を行うアドレスのリストに追加される．3 回目の解析では対象が A，C，B，A となり A が一致するが，A は既にインデックス変換を行うアドレスのリストに存在するため，追加が行われない．この手順の解析を続けていくと，図の例では 6 回目の解析で E がインデックス変換を行うアドレスのリストに追加されることとなり，最終的に例示の範囲のアドレス列では A，B，E がインデックス変換対象のアドレス列となる．

解析で使用したスコープサイズは大まかに時間の範囲を意図しており，ある限られた短い時間で同じブロックアドレスがミスするならば，そのブロックが格納されるべきセットで競合性ミスが発生しており，そのブロックの格納先を使用頻度の低いセットに変更すれば，ミス数が減少すると言う予測に基づいている．当然，このスコープサイズは大きくするほど，抽出できるインデックス変換を行うアドレス数を増やすことができるが，同時に性能に大きく影響を及ぼさない競合性ミスを検出する可能性があり，そのようなブロックは変更したセットの先で新たな競合性ミスを生む可能性が有る．このため，スコープサイズはプログラム毎に適切に設定すべきパラメータである．このことから，組込みソフトウェア開発において，本提案方式を採用する開発者はこのパラメータの最適値を探す作業を行わなくてはならない．

4. 評　　価

本節では，2 節で示したハードウェアをシミュレーション上で評価し，セットの参照負荷分散が実現できているかを示すと共に，提案ハードウェアがどの程度キャッシュミス率を低減したかを示す．シミュレーションは SimpleSclar/ARM[6] と MiBench[7] を用い行った．SimpleSclar/ARM のキャッシュシミュレー

表 1 SimpleSclar/ARM キャッシュシミュレーションパラメータ.

ifqsize/decode/commit width	4
ruu size	16
lsq size	8
mplat	3
speed	1
bimod	2048
il1 size	2KB
il1 way	4
il1 block size	16B
il1 hit latency	1 cycle
il1 miss pnalty	60 cycles
dl1 size	512B
dl1 way	2WAY
dl1 block size	16B
dl1 hit latency	1 cycle
dl1 miss pnalty	60 cycles
mem width	8B

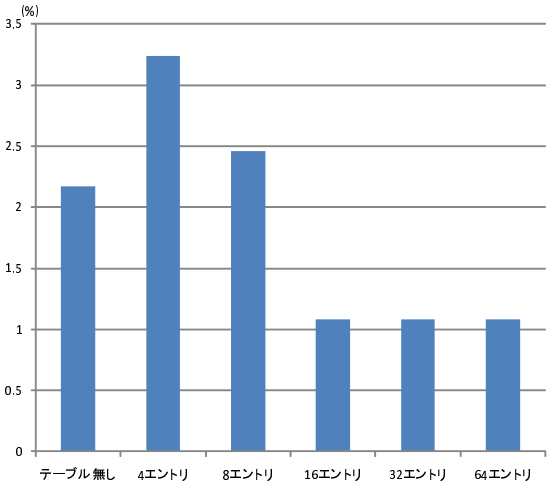

図 4 提案手法のインデックス変換テーブルのエントリ数を変化させたときのミス率の変化.

ション部には 2 節で示したインデックス変換ハードウェアをシミュレーションモデルに加えた．シミュレーションは sim-outorder を使用した．MiBench の中から 14 アプリケーションを選択し☆，入力は large を選択した．表 1 に SimpleSclar/ARM のシミュレーションパラメータを示す，

4.1 投入ハードウェア量による性能変化の比較

図 4 にスコープサイズを 16 にし，インデックス変換テーブルを 4〜64 エントリに変化させた場合の命令キャッシュミス率の変化を示す．後のキャッシュ性能比較で示すが，16 のスコープサイズは極めて妥当なスコープサイズである．縦軸がミス率 (%) で横軸が変化させたインデックス変換テーブルのエントリ数となっている．キャッシュ構成は 2K バイト 4 ウェイセットアソシアティブ構成でワークロードは basicmath である．basicmath は MiBench ベンチマークセットの中の 1 つのアプリケーションである．後のキャッシュ性能比較で示すが，本評価の対象ワークロードで最も提案手法が効いたアプリケーションである．4 エントリ〜16 エントリの間は，ほぼ均等にキャッシュミス率が減少し，エントリ資源投入分の効果がリニアに出ている．しかしながら，64 エントリと 32 エントリと 16 エントリの間の差は無い．以上の結果から 16 エントリで提案手法は最大の効果が得られており，それ以上のエントリサイズは意味をなさないことがわかる．またテーブルを持たない通常実行と比べ，4 エントリと 8 エントリは性能が劣っていることがわかる．これはインデックス変換テーブル内でスラッシングが発生することに起因している．ここで言うスラッシングとはインデックス変換を伴うミスハンドリング後，テーブルエントリの容量不足で直ぐにエントリが置換される現象のことである．テーブルエントリの置換が発生すると，置換対象の変換テーブルに紐付けられたブロックが無効化されるため，これが短時間で多発するスラッシング状態では，本来影響の無いはずの他の広範囲のセット内のブロックを無効化しすることになる．この状態が発生すると，著しい性能低下が発生し，4 エントリと 8 エントリの結果が示す性能低下につながる．これらのことから，後のキャッシュ性能比較では本評価において十分なエントリ数である 16 エントリのインデックス変換テーブルで評価している．

本評価で示したインデックス変換テーブルのエントリ数 (16 エントリ) は全てのキャッシュ構成で有効である訳ではなく，キャッシュ構成によって変化する．本稿の評価はブロックサイズ 16 バイトの 2K バイト 4 ウェイセットアソシアティブ構成行ったため，32 セット 128 キャッシュブロックが存在する．このことから変換テーブルエントリを 16 とした場合，最大で 12.5%のキャッシュブロックがインデックス変換状態で格納されることになる．他のキャッシュ構成でインデックス変換テーブルのエントリを用意する場合も同様に，この程度の割合のエントリ数を用意した場合に良いバランスのセット間負荷分散が実現できる可能性が高い．

4.2 キャッシュ性能比較

提案手法を適用した場合のミス数の減少率を図 5 に

☆ 2KB-4WAY 構成のキャッシュでミス率が 0.5%以下のアプリケーションとシミュレータの都合で実行ができなかったアプリケーションを除いている．

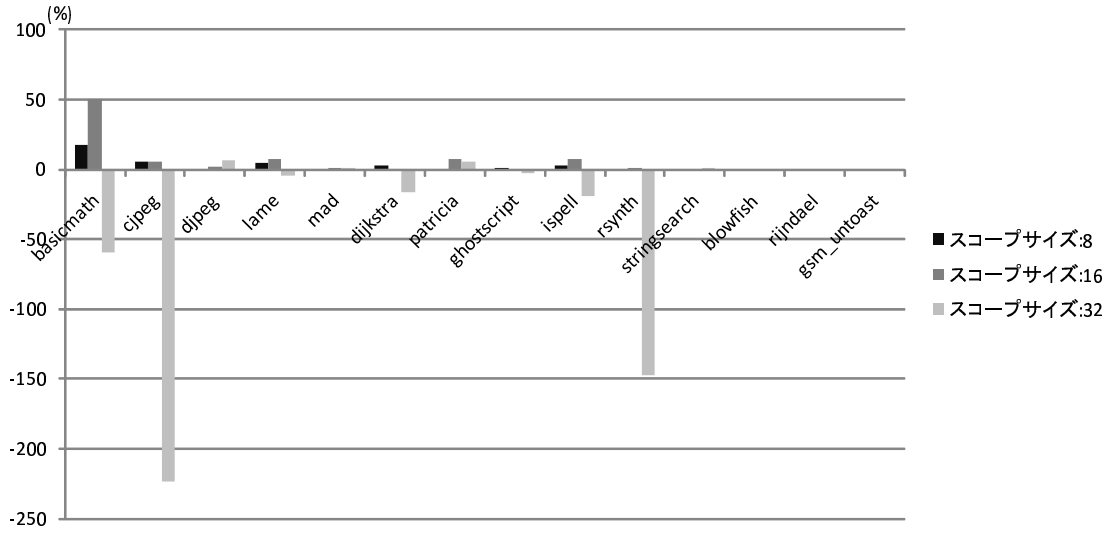

図 5 提案手法適用時のミス数減少率.

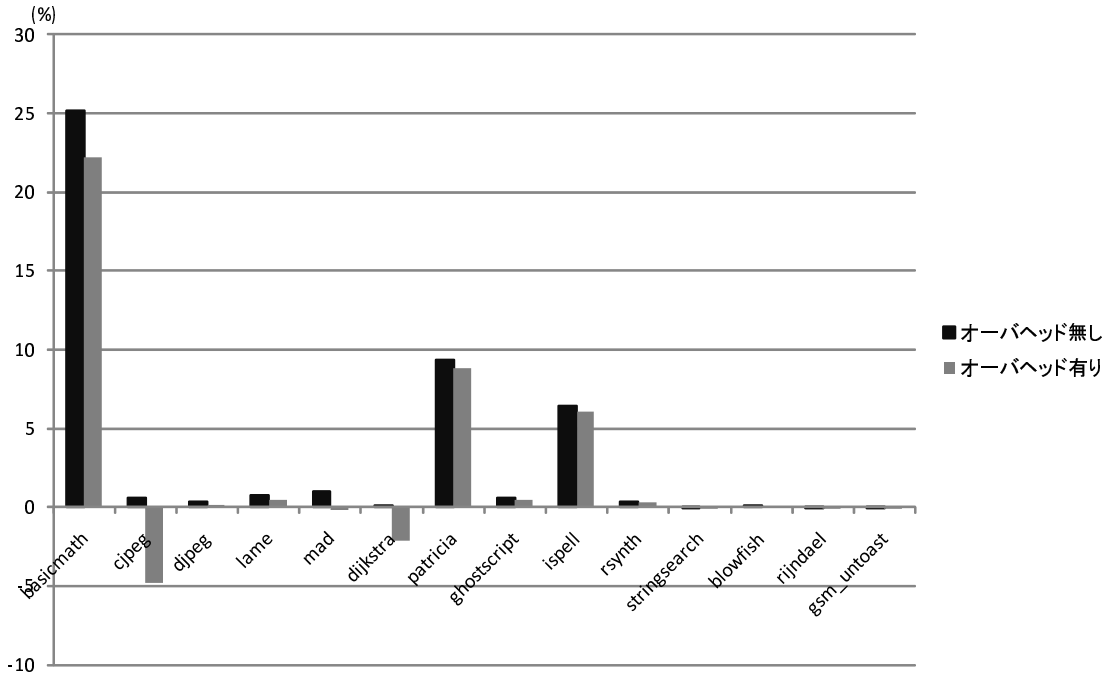

図 6 提案手法適用時のクロックサイクル性能向上率.

示す．縦軸は減少率（%）を表しており，例えば，プラス方向に 50%であれば，ミス数が半分に，逆にマイナス方向に 50(%) であれば，ミス数が倍になったことを示す．横軸は計測したアプリケーションである．各アプリケーション内には黒，灰色，薄い灰色の 3 本の棒がある．各色の棒は 3 節で示したスコープサイズの違いである．黒が 8，灰色が 16，薄い灰色が 32 である．まず着目するのが，薄い灰色の棒である．スコープサイズが 32 である場合，目立ってミス数が増加している．特に basicmath，cjpeg，dijkstra，ispell，stringsearch の 5 つのアプリケーションで性能低下が激しい．これはセット分散させたブロックが原因で新たな競合性ミスが発生し，何もしない状態よりミス数が増えているために起きていると考えられる．一方，

黒色の棒 (スコープサイズ 8) と灰色 (スコープサイズ 16) の場合は目立つ性能低下は無く☆，basicmath，cjpeg，lame，dijkstra，patricia，ispell で目に見える性能向上を得ている．特に basicmath における性能向上は劇的であり，黒色の棒が 17.8%，灰色が 50.2%のミス数削減を実現している．適切なスコープサイズはアプリケーションにより異なるが，今回評価したアプリケーションでは 32 は大きすぎ，8 では小さすぎる結果となった．本評価は一律に 2 のべき乗のサイズで試して性能を評価したが，実際の組込みソフトウェア開発においては実装したアプリケーション毎に適切なスコープサイズを探す必要がある．本研究では命令キャッシュ参照を対象にした．このことから，最適なスコープサイズはプログラムの制御構造に深く関係していると言える．命令キャッシュ上で競合性ミスを多発させる典型例は，ループ内で逐次に呼ばれる関数間や関数呼び出し間で発生するインデックス競合である．スコープサイズを小さくした場合，ループ内でも時間的に近くに発現する競合にのみ着目することになる．逆にスコープサイズ極端に大きくすれば，そのループ全体で起こる競合に着目するかもしれない．スコープサイズを大きくして性能が低下する例は，与えられたキャッシュサイズがフルに活用されている状況で再配置が行われ，結果的に競合を起こすセット数が増加する場合である．図 5 の評価では dijkstra がその例である．dijkstra は与えられたキャッシュ面積のほとんどを有効に利用する制御構造を持っており，8 に比べ 16 のスコープサイズで行うインデックス再配置が悪影響を与えている．逆に basicmath はキャッシュサイズが有効に利用できない制御構造を持ち，特定セットへのインデックス競合のみが性能に大きく影響している．basicmath の様な特性を持つアプリケーションを見つけることは容易である．シンプルなループ構造でかつ，ループボディの処理が単純であるにも関わらず，キャッシュ性能が向上しない場合は basicmath と同じ特性を持つことが考えられる．この場合は提案手法が有効であり，かつ，一定閾値まではスコープサイズを大きくするに従い，性能向上を大きくすることができる．

提案手法を適用した場合のクロックサイクル性能の向上率を図 6 に示す．図は図 5 で示したスコープサイズの中で最も性能が良かったスコープサイズ 16 の場合のクロックサイクル性能向上率である．縦軸は性能向上率 (%) を表しており，例えば，プラス方向に 10(%) であれば，実行に必要なクロックサイクル数が 9/10 になったことを示しており，逆にマイナス方向に 10%であれば，実行に必要なクロックサイクル数が 11/10 になったことを示している．横軸は図 5 と同様にアプリケーションである．各アプリケーション内には黒色と灰色の棒が 2 つあり，黒色はインデックス変換テーブル参照のためのオーバヘッドサイクルが無い状態の性能向上率であり，灰色はオーバヘッドサイクルが有る状態の性能向上率である．黒色の棒は理想的にインデックス変換テーブルの参照がクリティカルパスにならずに，マルチポートのキャッシュメモリに同時にインデックス参照ができるハードウェアを設計できた場合に実現する．灰色の棒は通常アクセスのクロックサイクルの後，次のクロックサイクルで変換後のインデックスで再参照する実装で実現できる．basicmath，patricia，ispell でオーバヘッド無し，有り，共に大きな性能向上を観測できた．☆☆これら大きな性能改善を示したアプリケーションの内，特に basicmath では 20% 以上の性能向上を達成しており，このようなアプリケーションが存在することは本提案を導入する大きな動機づけとなる．一方，djpeg，dijkstra ではオーバヘッドが加算されることで，性能が低下することが確認された．このようにミス率を下げた場合でも，実実行時間が低下する現象が確認できたため，本提案を採用する際には注意を持ってチューニングを行う必要がある．

5. 関 連 研 究

本節では，本提案手法に関係のあるの他の手法を紹介する．

まず，最初に挙げられるのが Victim Cache[9] である．Victim Cache は小さい容量のキャッシュで大きな効果を得ることができることから，今日のプロセッサでは標準的に搭載される機構である．Victim Cache は本提案同様に競合性ミス削減能力を持つ．本提案は既存キャッシュ内の参照頻度の低いブロックを利用することで競合性ミスを回避するが，Victim Cache は競合性ミスを回避するための専用のキャッシュ機構を持つ．このことから既存資源の有効利用を目的とした本研究とは根本的にアプローチが異なる．

高連想度キャッシュを提供することによりキャッシュ

☆ グラフで潰れているがいくつかのアプリケーションで極微少のミス数増加を観測した．

☆☆ 図 5 で示した patricia，ispell のミス数改善率に比べ，図 6 における評価の場合にクロックサイクル性能改善率が大きく見えるのは，これらアプリケーションは元々キャッシュミス率が高く，命令キャッシュミスによる性能低下が大きなオーバヘッドとなっていたためである．

ミス削減を狙うプロセッサが存在する.(StrongARM[1] (32-way), XScale[2] (32-way) など.) 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.[3]

キャッシュへのブロック挿入を制御するという，本研究の特徴に類似する研究として，我々が以前に提案したキャッシュのスラッシング緩和手法がある[4][5]．当手法は本手法と異なり，競合しているブロックを他のセットへ再配置するのではなく，極めて小さいバッファに格納する．バッファは極短時間の空間的局所性のみを提供する目的で配置され，フィル先をバッファに指定された複数のブロックがそのバッファを取り合う形となる．このようなブロック配置を行うことで，最低限，既存キャッシュに残されるブロックのヒットを保証し，極度の競合性ミス (スラッシング) が発生してキャッシュが機能しなくなることを回避する．本提案との共通点はフィル先の指定によりキャッシュ性能改善を試みる点である．このことから本提案と組み合わせることが容易であり，バッファを導入し，本提案のフィル先の選択肢を増やすことで，更なる性能改善が見込める．

6. 終わりに

本稿は，セットアソシアティブキャッシュメモリのセットの参照負荷を分散するためのインデックス変換手法を提案した．特定セットに参照が偏ることがミス数増大を招き，キャッシュ性能を低下させることを議論し，この問題を解決するために，インデックス変換を行い，セットの参照負荷を分散させるハードウェアとミスアドレス解析手法を提案し，シミュレーションベースでキャッシュ性能評価を行った．提案手法はキャッシュ容量を増加させることなく，最大の性能を発揮したアプリケーションで 25.1%の性能向上を観測した．

本稿の評価は命令アクセスのみに履歴情報を用いた動的なインデックス変換を適用して評価した．命令キャッシュのインデックス衝突を回避する手段は他にも考えられる．例えば，主記憶に配置する命令自身の配置アドレスをコンパイラやアセンブラの最適化により，削減する静的方法である．但し，この方法はコンパイラやアセンブラが実行パスを限定できるプログラムに限られるため，全てのプログラムに適用することは難しい．加えて，ハードウェアで実現することでコンパイラが知ることのできないマルチプロセス／マルチスレッド実行等の動的な参照傾向変化の要因にも追従して制御することができる．

今後の課題として，データキャッシュに本インデックス再配置手法の適用し，その評価を行うことを予定している．また，本提案は組込みシステム用の提案であるため，提案機構のハードウェア量と消費電力量が問題となることから，これらの指標を含めた評価を予定している．

参 考 文 献

1) J.Montanaro, et al., “A 160-MHz, 32-b, 0.5-W, CMOS RISC Microprocessor”, IEEE Journal of Solid-State Circuits, volume 31, number 11, November 1996.
2) Intel Corporation, “Intel XScale Microarchitecture, Technical Summary”, 2000.
3) A.Veidenbaum, D.Nicolaescu, “Low Energy, Highly-Associative Cache Design for Embedded Processors”, Proc. of Intl. Conf. on Computer Design (ICCD), pp.332–335, 2004.
4) 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.
5) 請園智玲，田中清史，“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”，第 183 回計算機アーキテクチャ研究会，情報処理学会研究報告，(オンライン)，2010.
6) http://www.simplescalar.com/v4test.html
7) M.Guthaus, J.Ringenberg, D.Ernst, T.Austin, T.Mudge, R.Brown, “MiBench: A free, commercially representative embedded benchmark suite”, Proc. of IEEE Intl. Workshop on Workload Characterization, 2001 (WWC-4).
8) http://www.eecs.umich.edu/mibench/
9) 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.