社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－SLDM－133 (27)
2008 / 1 / 17

アプリケーションプロセッサのL1データキャッシュ最適化手法

東條 信明† 戸川 望† 柳澤 政生† 大附 辰夫†

† 早稲田大学大学院基幹理工学研究科情報理工学専攻
〒169-8555 東京都新宿区大久保3-4-1
E-mail: †toujyo@ohtsuki.comm.waseda.ac.jp

あらまし 本稿では複数のキャッシュ構成の中から特定のアプリケーションに適した構成を選択するためのキャッシュ最適化手法を提案する．本手法は，アプリケーションソースコードと制約条件を入力することで，メインメモリとL1キャッシュからなるメモリ階層全体の平均アクセス時間を評価指標として，対象となるアプリケーションに特化したキャッシュ構成を求める．キャッシュ容量，ブロックサイズ，連想度を最適化し，アプリケーションに最適なL1データキャッシュ構成を目指す．計算機実験を行った結果を既存手法と比較し，有効性を確認した．
キーワード データキャッシュ，キャッシュ最適化，組み込みシステム，プロセッサコア

An L1 Data Cache Optimization Algorithm for Application Processor Cores

Nobuaki TOJO†, Nozomu TOGAWA†, Masao YANAGISAWA†, and Tatsuo OHTSUKI†

† Dept. of Computer Science and Engineering, Waseda University
3-4-1 Okubo, Shinjuku, Tokyo, 169-8555 Japan
E-mail: †toujyo@ohtsuki.comm.waseda.ac.jp

Abstract One major factor in improving the performance of embedded processors is the use of data and instruction caches. In this paper, we propose an L1 data cache optimization algorithm which selects a suitable cache configuration for a given embedded application. Our algorithm can have the area constraint by introducing CRMF (Configuration Reduction approach by the Miss Factor) and CRCB(Configuration Reduction approach by the Cache Behavior). Our algorithm finally selects best cache size, block size and associativity under the area constraint for a targeted application. We demonstrate the effectiveness of our algorithm by applying it to Mediabench.
Key words data cache, cache optimization, embedded system, processor core

1. まえがき

近年，CPU等の処理装置が高速化してきているが，メモリ等の記憶装置の速度は処理装置の性能向上に追いつくことができていない．このため，記憶装置と処理装置の性能差が拡大し，問題となっている．これを解決するために用いられる手法の1つがキャッシュあるいはキャッシュメモリである．しかし，キャッシュの構成や動作するアプリケーションの内容によっては，キャッシュ内のデータを頻繁に入れ替える必要が生じ，結果的にメインメモリのみの場合とキャッシュを組み込んだ場合がほとんど変わらないアクセス速度になる可能性もある．また，搭載できるキャッシュの面積には制限があり，無制限にキャッシュを大きくすることはできない．よって，こうした問題を考慮したキャッシュ構成を選択することが求められている．

組み込みシステムの場合，実際に組み込みシステム上で動かすアプリケーションは少数であり，汎用プロセッサのように様々なアプリケーションを考慮する必要はない．つまり，動作させるアプリケーションさえ高速に実行できれば，他のアプリケーションの実行が遅くても問題ない．そのため，キャッシュ構成をアプリケーションに合わせて最適化することが可能となる．

キャッシュ構成の最適化手法には，大きく分けて2つの種類がある．1つ目は，文献[6]のように解析的手法を用いることで，誤差は20%程度と大きいが，高速に最適化を行う手法である．2つ目は，文献[1],[5]のようにシミュレーションベースの手法を用いることで，実行時間は遅くなるものの，正確に各構成の性能を得ることができる手法である．特に，文献[1]はキャッシュの性質を利用することで，文献[5]の平均45倍高速にキャッシュ構成を探索することができる．

本稿では，シミュレーションベースの手法に焦点を当てたCRCB (Configuration Reduction approach by the Cache Be-

havior)1 手法，CRCB2 手法，CRMF (Configuration Reduction approach by the Miss Factor)1 手法，CRMF2 手法を提案する．キャッシュ構成 s は，キャッシュ容量 m，ブロックサイズ b，連想度 a によって与えられる．キャッシュ容量の最大値を m_m，ブロックサイズの最大値を b_m，連想度の最大値を a_m，アプリケーションのメモリアクセス数を n 回とすると，全構成の中で何らかの指標が最適な構成を得るにはキャッシュのヒット/ミス判定を $O(n \times \lg m_m \cdot \lg b_m \times a_m^2)$ 回行う必要がある．これに対し，文献 [1] で提案された複数連想度同時探索手法，および本稿で提案する CRCB1 手法，CRCB2 手法を導入すると，単純なキャッシュシミュレーションと同様の正確な結果を出力しつつ，ヒット/ミス判定を $O(n \times \lg m' \cdot \lg b' \times a_m)$ $(m' \leqq m_m)$ $(b' \leqq b_m)$ で実行することができる．さらに，本稿で提案する CRMF1 手法，CRMF2 手法を導入すると，必ずしも最適なキャッシュ構成が選択できるわけではないが，ヒット/ミス判定を $O(n \times \lg m'' \times a' + n \times \lg b' \times a')$ $(m'' \leqq m')$ $(a' \leqq a_m)$ で実行し，メモリアクセス時間の最小化が期待できる．

さらに，計算機実験の結果を示し，提案手法の優位性や CRMF を利用した場合でも本稿で実験を行ったアプリケーションに対しては最適なキャッシュ構成が選択されていることを確認した．

2. キャッシュ

本稿では，オンチップ上の L1 データキャッシュを対象とする．データの置き換え手法は LRU 法(注1)を用いる．

2.1 キャッシュ構成

キャッシュ構成は，キャッシュ容量 m，ブロックサイズ b，連想度 a によって与えられる．キャッシュ容量 m は，キャッシュに入れることのできるデータの合計ビット数を示す．

キャッシュには，空間的局所性を利用したブロックという手法が用いられている．これは，メインメモリからキャッシュにデータを入れる場合，必要なデータ以外に決められたサイズのデータをキャッシュに一緒に格納するという手法である．これにより，参照されたデータの近くのデータを一緒にキャッシュに置くことが可能となり，キャッシュヒット率を向上することが可能となる．このキャッシュにデータを格納する際のデータサイズのことをブロックサイズと呼ぶ．

キャッシュには，データの配置制約が 3 種類存在している [4]．データの配置先が一意に決定されているダイレクトマップ方式，その逆にデータの配置先が任意であるフルアソシアティブ方式，両者の中間の方式であるセットアソシアティブ方式である．セットアソシアティブ方式は，各ブロックをある決められた数にまとめたセットという単位で管理する方法である．セット数 $n_{set} = m/(b \times a)$ となり，各データの配置先セットは一意に決められており，セット内のどのブロックに配置するかは任意となっている．セットアソシアティブ方式において，各セットにいくつのブロックが割り当てられているのかを示す値が連想度である．また，ダイレクトマップの場合は $n_{set} = m/b$ となるため，$a = 1$，フルアソシアティブ方式の場合は $n_{set} = 1$ となるため，$a = m/b$ と表現できる．

(注1)：LRU 法とは，Least Recently Used 法の略であり，使用頻度の一番低いデータから置き換えていく手法のことである．

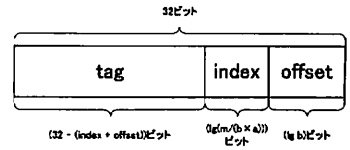

図 1 メモリアドレス 32 ビット，キャッシュ容量 m，ブロックサイズ b，連想度 a の場合の offset，index，tag の各ビット数

2.2 キャッシュミス要因

全てのキャッシュミスは，容量ミス，競合ミス，初期参照ミスという 3 つの要因に分類することができる [4]．本節では，各ミス要因とキャッシュ容量 m，ブロックサイズ b，連想度 a との関連性についてまとめる．

2.2.1 容量ミス

m がアプリケーションの実行に必要なデータサイズより小さい場合，キャッシュに入りきらずに追い出されるデータが必ず存在する．こうして追い出されたデータに再度アクセスした際に発生するキャッシュミスのことを容量ミスと呼ぶ．

容量ミスを抑えるためには，m を大きくすればよい．これにより，追い出されるデータ量が小さくなり，容量ミスを減らすことができる．ただし，m を大きくすると，面積やキャッシュアクセス時間が悪化することになる．

2.2.2 競合ミス

同一のセットに割り当てられるメモリアドレス数が a を超えている場合，セット内に入りきらずに追い出されるデータが必ず存在する．こうして追い出されたデータに再度アクセスした際に発生するキャッシュミスのことを競合ミスと呼ぶ．

競合ミスを抑えるためには，a を大きくすればよい．これにより，セット内のブロック数が増加し，競合が発生する確率が少なくなるため，競合ミスが減る．ただし，a を大きくすると，面積やキャッシュアクセス時間が増加することになる．

2.2.3 初期参照ミス

あるデータに対する最初のアクセスの際，キャッシュ上にはそのデータが存在していないため，必ずキャッシュミスとなる．このキャッシュミスのことを初期参照ミスと呼ぶ．

初期参照ミスを抑えるためには，b を大きくすればよい．これにより，アプリケーションを構成するブロック数が減少し，初期参照自体が少なくなるため，初期参照ミスを減らすことができる．ただし，キャッシュ上のブロック数も減少するため，競合ミスを増加させる可能性もある．さらに，メインメモリからキャッシュへの転送時間が長くなるため，キャッシュミスペナルティが増大する．

2.3 キャッシュのアドレス処理

キャッシュでは，どのセットにデータを格納するのかを決定するため，メモリアドレスを offset，index，tag の 3 つのフィールドに分割処理している [4](図 1)．

キャッシュでは，index の値を用いてデータの格納先セットを決定する．キャッシュに目的のデータがあるかどうか確認する際には，格納先セットの各ブロックの tag を確認し，tag が一致した場合はヒット，一致しなかった場合はミスとなる．

3. 複数のキャッシュ構成の探索手法

キャッシュ構成 s は，キャッシュ容量 m，ブロックサイズ

b，連想度 a を使って，$s=(m,b,a)$ と記述することができる．今，キャッシュ容量の最大値を m_m，ブロックサイズの最大値を b_m，連想度の最大値を a_m として，キャッシュ容量が $64,128,256,\cdots,m_m$，ブロックサイズが $8,16,32,\cdots,b_m$，連想度が $1,2,3,\cdots,a_m$ と変化させた際のキャッシュ構成を考える．この場合，$(\lg m_m \cdot \lg b_m \times a_m)$ 個のキャッシュ構成がある．

アプリケーションの 1 つのメモリアクセスを考える．1 つのキャッシュ構成 $s=(m,b,a)$ が与えられたとき，このアクセスがキャッシュにヒットしているかミスしているかを調べるには，

（1） メモリアドレス m_a から，index 部 i_a，tag 部 t_a を算出する．index 部 i_a に対応するセットには，a 個のブロックが配置されている．

（2） a 個のブロックについて，1 番目から順に，t_a が存在するかどうか調べる．

（3） (2) において，t_a が j 番目に存在すればヒットとし，index 部 i_a に対応するセットの j 番目のブロックを 1 番目のブロックとする．

（4） (2) において，t_a が存在しなければミスとなりミスすれば，index 部 i_a に対応するセットの a 番目のブロックを t_a に置換し，このブロックを 1 番目のブロックとする．

上記の操作のうち，(1) と (3) は一般に $O(1)$ の時間で処理される．一方，(2) は，1 つずつ順にブロックを調べる必要があるため，$O(a)$ の時間を要する．

あるアプリケーションプログラムが与えられたとき，そのメモリアクセスの総数を n 回とする．上記の議論より，1 つのキャッシュ構成 $s=(m,b,a)$ について，キャッシュのヒット/ミス判定は，$O(na)$ の時間を要することが分かる．$(\lg m_m \cdot \lg b_m \times a_m)$ 個のキャッシュ構成すべてについて，ヒット/ミス判定を行うには，$O(n \times \lg m_m \cdot \lg b_m \times a_m^2)$ の時間を要することが分かる．これは，全数探索によって，$(\lg m_m \cdot \lg b_m \times a_m)$ 個のキャッシュ構成の中で何らかの指標が最適なキャッシュ構成を得るためには，$O(n \times \lg m_m \cdot \lg b_m \times a_m^2)$ の時間が必要であることを表している．

これに対し本章では，まず文献 [1] で提案された複数連想度同時探索手法を導入することで，ヒット/ミス判定を $O(n \times \lg m_m \cdot \lg b_m \times a_m)$ に削減する．続いて，新たに CRCB1 手法を提案し，ヒット/ミス判定を $O(n \times \lg m' \cdot \lg b_m \times a_m)$ $(m' \leqq m_m)$ に削減する．さらに，CRCB2 手法を提案し，ヒット/ミス判定を $O(n \times \lg m' \cdot \lg b' \times a_m)$ $(b' \leqq b_m)$ に削減する．これらの 3 手法は，キャッシュの性質を用いることで，$O(n \times \lg m_m \cdot \lg b_m \times a_m^2)$ の時間を要する完全なキャッシュシミュレーションのための時間を削減することができることを示している．

本章ではさらに，メモリアクセス時間を最小化することを目的とした場合，ブロックサイズ b と，キャッシュ容量 m/連動度 a は，独立に最適化することができるという性質を示し，この性質をもとに CRMF1 手法を提案する．CRMF1 手法を用いることで，$O(n \times \lg m' \times a_m + n \times \lg b' \times a_m)$ 回のヒット/ミス判定で，メモリアクセス時間を最小化することができる．さらに加えて，初期参照ミスの回数が分かったとき，これを利用することで，メモリアクセス時間を最小化を目的に，ヒット/ミス判定回数を削減する CRMF2 手法を提案する．CRMF2 手法を用いることで，ヒット/ミス判定は，最終的に $O(n \times \lg m'' \times a' + n \times \lg b' \times a')$ $(m'' \leqq m')$ $(a' \leqq a_m)$ に削減することができる．CRMF1 手法と CRMF2 手法を適用すると，必ずしも $(\lg m_m \cdot \lg b_m \times a_m)$ 個のキャッシュ構成の中から，メモリアクセス時間が最小なキャッシュ構成を選択できるわけではないが，実験的にはメモリアクセス時間が最小なキャッシュ構成を選択できる．表 1 にキャッシュうヒット/ミス判定回数をまとめる．表 1 の Exact Simulation の項目は，全探索と同様の正確な結果を必ず出力できるかどうかを示している．

3.1 複数連想度同時探索手法 [1]

今，構成が (m,b,a) となる 1 つのキャッシュ構成を考える．このようなキャッシュに対して，以下の性質がある [1]:

性質 1. あるメモリアクセスについて，構成が (m,b,a) となるキャッシュでキャッシュミスが起こった場合，構成が (m,b,a') $(1 \leqq a' \leqq a)$ となるすべての構成でキャッシュミスが起こる．同様に，あるメモリアクセスについて，構成が (m,b,a) となるキャッシュでヒットした場合，構成が (m,b,a') $(a \leqq a' \leqq a_m)$ となるすべての構成でヒットする．

性質 1 を用いれば，構成が (m,b,a_m) となるキャッシュと，ある 1 つのメモリアクセスについて:

複数連想度同時探索手法

（1） メモリアドレス m_a から，index 部 i_a，tag 部 t_a を算出する．index 部 i_a に対応するセットには，a_m 個のブロックが配置されている．

（2） a_m 個のブロックについて，1 番目から順に，t_a が存在するかどうか調べる．

（3） (2) において，j 番目のブロックで t_a がヒットした場合，構成 (m,b,a) $(1 \leqq a \leqq j)$ の全キャッシュでヒットしており，構成 (m,b,a) $(j < a \leqq a_m)$ の全キャッシュでミスすることになる．このヒットしたブロックを 1 番目のブロックとする．

（4） (2) において，t_a がミスした場合，構成 (m,b,a) $(1 \leqq a \leqq a_m)$ の全キャッシュでミスしていることになる．a_m 番目のブロックと t_a を置換し，1 番目のブロックとする．

上記の操作は，連想度が 1 から a_m までの a_m 個のキャッシュ構成について，1 回のシミュレーションで同時にシミュレーションできることを表している．

例 1. $b=8$，$(a=1,2,\cdots,16)$，セット数 $(n_{set}=8,16,\cdots,512K)$ の各キャッシュ構成について 24 ビットのメモリアドレス "100101100011111101110010" のヒット/ミス判定を行う場合を図 2 を用いて考える．まず，offset は $\lg 8=3$ ビット必要なので，下位 3 ビットが offset となる．最小のセット数を 8 とすると，このとき必要な index のビット数は $\lg 8=3$ ビットとなる．そこで，残りの 21 ビットのうち，下位 3 ビット "110" を見てみる (図 2 中の L1)．すると，メモリアドレス "100101100011111101**110**010" のデータは，index"110" に対応するセットに格納されるということになる．シミュレーション上は，図 2 のように各セットのブロックはリスト構造を用いて表現されているため，index"110" と対応しているリスト L1 を調べることになる．この場合の tag は "100101100011111101" となる．次に，index"110 のリストを図 2 の一番下の○が示している連想度 1 と対応するブロックから矢印の示す順に tag が一致するかを調べていく．i 番目で tag が一致した場合は，キャッシュヒットと判断し，index"110" のリストの i 番目のブロックを 1 番目のブロックとする．16 番目のブロックまで調べても tag が一致しなかった場合は，16 番目のブロックに tag"100101100011111101" を格納し，このブロックを index"110" のリストの 1 番目のブロックとする．これで L1 の各構成に対するヒット/ミス判定が終了する．次に，L2 の各構成に対する判定を行う場合，セット数が 2 倍になり，index が 1 ビット増え，tag が 1 ビット減る．この例の場合，図 2 に示した通り，index が "1110" となる．このため，index"1110" に対応したリスト L2 を調べればよいことになる．これを全探索範囲について繰り返していくことで，ヒット/ミス判定を行う．

表 1　キャッシュヒット/ミス判定回数

手法	ヒット/ミス判定回数	Exact Simulation
全探索	$O(n \times \lg m_m \cdot \lg b_m \times a_m^2)$	YES
文献[1]	$O(n \times \lg m_m \cdot \lg b_m \times a_m)$	YES
文献[1]+CRCB1	$O(n \times \lg m' \cdot \lg b_m \times a_m)$ $(m' \leq m_m)$	YES
文献[1]+CRCB1+CRCB2	$O(n \times \lg m' \cdot \lg b' \times a_m)$ $(b' \leq b_m)$	YES
文献[1]+CRCB1+CRCB2+CRMF1	$O(n \times \lg m' \times a_m + n \times \lg b' \times a_m)$	NO
文献[1]+CRCB1+CRCB2+CRMF1+CRMF2	$O(n \times \lg m'' \times a' + n \times \lg b' \times a')$ $(m'' \leq m')$ $(a' \leq a_m)$	NO

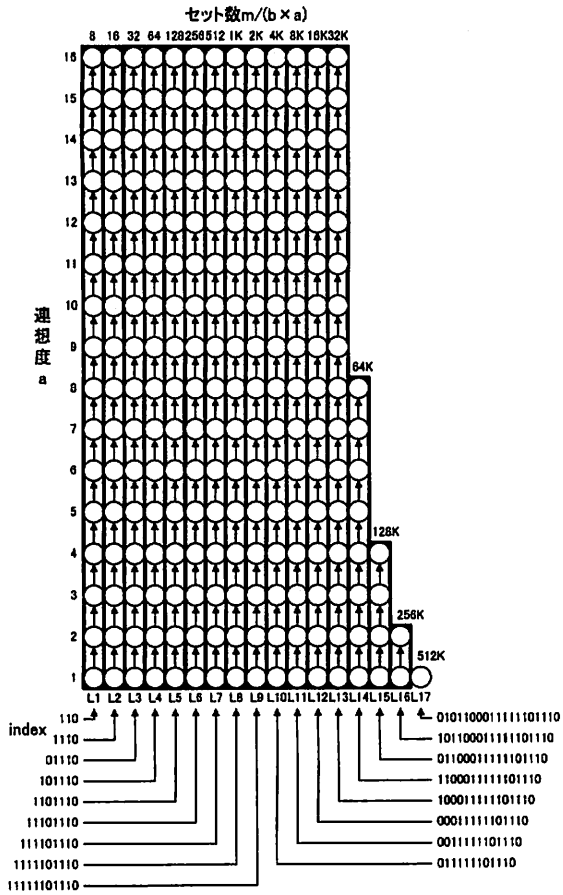

図 2　複数連想度同時探索手法

文献[1]では，1つのメモリアクセスについて，連想度が a_m，メモリ容量が $(512, 1024, \cdots, m_m)$，ブロックサイズが $(8, 16, \cdots, b_m)$ の各構成のシミュレーションを実行する．これにより，等価的に連想度が $(1, 2, \cdots, a_m)$，メモリ容量が $(512, 1024, \cdots, m_m)$，ブロックサイズが $(8, 16, \cdots, b_m)$ の各構成のシミュレーションを実行したことになる．アプリケーションプログラムのメモリアクセス数を n 回とすれば，キャッシュのヒット/ミス判定回数は $O(n \times \lg m_m \cdot \lg b_m \times a_m)$ となる．

3.2　CRCB1 手法

文献[5]によると，2つのキャッシュ s_1，s_2 について，s_1 のキャッシュ要素がすべて s_2 のキャッシュ要素に含まれていれば $s_1 \subset s_2$ と書き，s_2 でキャッシュミスが起これば s_1 でもキャッシュミスが起き，s_1 でキャッシュヒットが起これば s_2 でもキャッシュヒットすることが示されている．これをキャッシュの連想度と容量に適用すれば，次の性質を得ることができる:

性質 2. あるメモリアクセスについて，構成が $(m, b, 1)$ となるキャッシュでキャッシュヒットが起こった場合，構成が (m', b, a'), $(1 \leq a' \leq a_m)$, $(m < m' \leq m_m)$ となるすべての構成でキャッシュヒットが起こる．

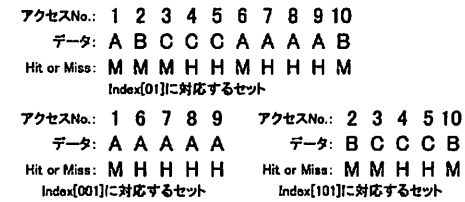

図 3　メモリアクセスの流れ

例 2. あるキャッシュ構成 $(m, b, 1)$ の中の index"01"に対応するセットに対して，アプリケーションのメモリアクセスが図 3 のように与えられたとする．図 3 中のアクセス No. はアクセス順序，データはメモリアドレス (ここでは簡略化のため，アルファベットで表記している)，Hit or Miss はキャッシュヒットした場合を H，ミスした場合を M として表記している．一方，キャッシュ容量を 2 倍にしたキャッシュ構成 $(2m, b, 1)$ を考える．すると，index が 1 ビット増えるため，キャッシュ構成 $(m, b, 1)$ の index"01"に対応するセットに割り当てられていたデータは，キャッシュ構成 $(2m, b, 1)$ では index"001"に対応するセットあるいは index"101"に対応するセットに割り当てられることになる．今，図 3 のようにデータ A が index"001"に対応するセット，データ B，データ C が index"101"に対応するセットに割り当てられたとする．この場合，キャッシュ構成 $(m, b, 1)$ でヒットしていたアクセス No.4, 5, 7, 8, 9 について，キャッシュ構成 $(2m, b, 2)$ では必ずヒットすることがわかる．

性質 2 を用いれば，構成が (m, b, a_m) となるキャッシュと，ある 1 つのメモリアクセスについて，複数連想度同時探索手法に加えて:

複数連想度同時探索手法 + **CRCB1** 手法

(5)　もし，a_m 個のブロックについて，1 番目のブロックで tag 部 t_a がヒットしたとすれば，キャッシュ構成 (m', b, a_m) $(m \leq m' \leq m_m)$ でも，すべて a_m 個のブロックについて，1 番目のブロックで，t_a がヒットするため，キャッシュ構成 (m', b, a_m) $(m \leq m' \leq m_m)$ のヒット/ミス判定を省略する．

上記の操作は，1 つのメモリアクセスについて，キャッシュ容量を $64, 128, 256, \cdots$ と順に変化させてヒット/ミスを判定すれば，あるキャッシュ容量 m 以降は，ヒット/ミス判定を省略できることを示している．つまり，アプリケーションプログラムのメモリアクセス数を n 回とすれば，ヒット/ミス判定回数は $O(n \times \lg m' \cdot \lg b_m \times a_m)$ $(m' \leq m_m)$ に削減される．

例 2 の場合，キャッシュ構成 $(m, b, 1)$ でヒットしたアクセス No.4, 5, 7, 8, 9 については，キャッシュ構成 $(2m, b, 1)$ においてヒット/ミス判定を省略しても問題ないことがわかる．

3.3　CRCB2 手法

性質 2 と同じように，キャッシュのブロックサイズに関する以下のような性質がある．

性質 3. 今，あるアプリケーションプログラムにおいて，連続した 2 つのメモリアクセス，つまり，$i-1$ 番目のメモリアクセ

スと i 番目のメモリアクセスの index および tag が等しい場合を考える．任意の容量 $(m = 64, 128, \cdots, m_m)$ ならびにブロックサイズ $(b = 8, 16, \cdots, b_m)$ を持つキャッシュ構成 (m, b, a_m) について，i 番目のメモリアクセスは必ずキャッシュヒットする．

例 3. 8 ビットのメモリアドレス "01110010"と "01110111"がこの順に連続でキャッシュにアクセスする場合を考える．ブロックサイズ $b = 8$ とすると，offset は $\lg 8 = 3$ ビット必要となる．この場合，メモリアドレス "01110010"と "01110111"の index および tag の値は "01110"となり，等しくなる．よって，この 2 つのメモリアドレスは同じ 1 つのブロックに対応していることになる．つまり，性質 2 が成立することになり，構成が $(m', 8, a')$ $(1 \leq a' \leq a_m)$ $(m < m' \leq m_m)$ となるすべての構成でキャッシュヒットが起こる．また，b を 2 倍にした場合，offset が 1 ビット増加し，index が 1 ビット減少する．メモリアドレス "01110010"と "01110111"の場合，index がおよび tag の値は "0111"となり，ここでも性質 2 が成立することになる．つまり，いかなる b を選択しても，性質 2 が成立することになり，性質 3 のようにまとめることができる．

複数連想度同時探索手法 + CRCB1 手法 + CRCB2 手法

（6） もし，$b = k$ $(8 \leq k \leq b_m)$ で，i 回目のアクセスと $i-1$ 回目のアクセスの offset および tag が等しい場合，i 回目のアクセスはキャッシュ構成 (m', b', a') $(1 \leq a' \leq a_m)$ $(8 < b' \leq b_m)$, $(64 < m' \leq m_m)$ で必ずヒットするため，キャッシュ構成 (m', b', a') でのヒット/ミス判定を省略する．

これによって，最終的に $O(n \times \lg m' \cdot \lg b' \times a_m)$ $(b' \leq b_m)$ にヒット/ミス判定回数を削減することができる．なお，複数連想度同時探索手法ならびに CRCB1 手法および CRCB2 手法は，全探索シミュレーションと完全に等価なシミュレーションをしていることになる．

3.4 CRMF1 手法

2 章で述べた通り，ブロックサイズは初期参照ミスに大きな影響がある．アプリケーションプログラムが同一の場合，ブロックサイズが同一であれば，どのキャッシュ容量，どの連想度においても初期参照ミスは同数である．これは，特にキャッシュミスはメモリアクセス時間に寄与するため，この最小化を目的とした場合には，ブロックサイズとキャッシュ容量/連想度を独立して，最適化できることを示唆している．つまり，ブロックサイズを一定のもと，キャッシュ容量/連想度に対してメモリアクセス時間を最小化し，その後，最適化されたキャッシュ容量/連想度のもと，ブロックサイズを最適化することができる．以下に，この考えをもとにした CRMF1 手法を提案する:

CRMF1 手法

（1） ブロックサイズ $b = 8$ として，アプリケーションプログラムの n 個のメモリアクセスそれぞれについて，複数連想度同時探索手法 + CRCB1 手法を用いて，構成 $(m, 8, a)$ $(m = 64, 128, 256, \cdots, m_m, a = 1, 2, \cdots, a_m)$ のキャッシュのヒット/ミスを判定する．

（2） (1) の結果から，それぞれのキャッシュ構成についてメモリアクセス時間を算出し，最小のメモリアクセス時間を与えるキャッシュ容量/連想度を m_{opt}，a_{opt} とする．

（3） アプリケーションプログラムの n 個のメモリアクセスそれぞれについて，CRCB2 手法を用いて，キャッシュ構成 (m_{opt}, b, a_{opt}) $(b = 16, 32, \cdots, b_m)$ のヒット/ミスを判定する．

（4） (3) の結果から，それぞれのキャッシュ構成についてメモリアクセス時間を算出し，最小のメモリアクセス時間を与えるブロックサイズを b_{opt} とし，キャッシュ構成 $s_{opt} = (m_{opt}, b_{opt}, a_{opt})$ を出力する．

ここでメモリアクセス時間 T は，キャッシュアクセス時間を T_c，メインメモリのアクセス時間を T_m，キャッシュ，メインメモリ間の転送時間を T_{c-m}，ミス数を n_{miss}，総アクセス数を n_{access} とし，式 (1) で算出する．ただし本手法では，文献 [1] を参考に，T_m を 19.5[ns]，T_{c-m} を 20[ns/B] としている．

$$T = T_c + (T_m + b \times T_{c-m}) \times n_{miss}/n_{access} \qquad (1)$$

CRMF1 手法を用いることで，$O(n \times \lg m' \times a_m + n \times \lg b' \times a_m)$ 回のヒット/ミスの判定で，メモリアクセス時間を最小化することができる．

3.5 CRMF2 手法

今，アプリケーションプログラムの n 個のメモリアクセスそれぞれについて，複数連想度同時探索手法と CRCB1 手法，CRCB2 手法を用いて，構成 $(64, 8, a)$ $(a = 1, 2, \cdots, a_m)$ のキャッシュのヒット/ミスが判定されたとする．このとき同時に，初期参照ミスの回数を計測することができる．

続けて，さらにアプリケーションプログラムの n 個のメモリアクセスそれぞれについて，キャッシュ容量が大きいキャッシュ構成を対象に，ヒット/ミスを判定すると考える．ところが，ブロックサイズが同一の場合，いくらキャッシュ容量 m を大きくしても，初期参照ミスを低減することはできない．これは，初期参照ミスがブロックサイズにのみ依存するためである．さらに，同様の性質が連想度に対しても存在している．

この事実を利用すると，以下のような探索構成の削減が可能となる．今，構成 $(64, 8, a)$ $(a = 1, 2, \cdots, a_m)$ のヒット/ミスが判定され，初期参照ミスの回数を計測できたとする．ここで，構成 $(128, 8, a)$ では，容量が 2 倍になるので，キャッシュのセット数も 2 倍になり，ミス数は半分になると予想する．実際には，セットアソシアティブ方式の場合，メモリアドレスによって格納先セットが一意に決定されているため，これよりもミス数は増加する．つまり，実際よりもミス数が少なく予想される．これを踏まえると，式 (1) を利用して探索構成の削減が可能となる．以下に，この考えをもとにした CRMF2 手法を提案する:

CRMF2 手法

（1） $m = 64$ とする．

（2） CRMF1 手法 (1) で得られる構成 $(m, 8, a_m)$ のミス数を $n_{miss}(m)$ とし，このうち初期参照ミス数を n_{com} とする．

（3） (2) において，$n_{miss}(m) = n_{com}$ の場合，$a_m \leftarrow a_{m-1}$ とする．

（4） $n_{miss}(m)/2 \leq n_{com}$ を満たす限り，(a)，(b) を繰り返す．

（a） $n_{miss}(m) = n_{miss}/2$ とする．

（b） $m = 2m$ とする．

（5） $m = 2m$ として，(2) を行う．

この手法を用いることで，ヒット/ミス判定は $O(n \times \lg m'' \times a' + n \times \lg b' \times a')$ $(m'' \leq m')$ $(a \leq a_m)$ で実行できる．

4. L1 データキャッシュ最適化手法

3 章で提案した手法をベースにキャッシュ構成最適化システムを提案する．本手法は入力されたアプリケーションと面積制約に合ったキャッシュ構成を選択することを目的としている．キャッシュ構成最適化システムの概要を図 4 に示す．キャッシュアクセス時間と面積は，CACTI4.2 [3] を用いて算出する．対象アプリケーションのメモリアクセスのトレースファイルは，

表 2 実験結果

アプリケーション	トレースサイズ [B]	面積制約 [mm^2]	キャッシュ構成 (m[KB],b[B],a)	提案手法 [sec] (百万回)	文献[1]+CRCB1+CRCB2 [sec] (百万回)	文献[1] [sec] (百万回)	文献[7] [sec]	全探索 [sec] (百万回)
ADPCM	5222285	なし	(16,16,1)	2.18(2.00)	3.34(11.88)	14.57(57.83)	9.40(-)	56.05(201.75)
ADPCM	5222285	0.5	(16,32,1)	2.31(2.51)	3.35(11.88)	13.92(57.83)		56.44(201.75)
ADPCM	5222285	1.0	(16,16,1)	2.27(2.00)	3.37(11.88)	14.45(57.83)		56.40(201.75)
JPEG	55041814	なし	(256,32,1)	23.92(3.96)	43.01(16.90)	194.86(627.71)	99.08(-)	745.06(2137.82)
JPEG	55041814	1.0	(64,64,2)	22.21(3.31)	42.27(16.90)	194.70(627.71)		740.55(2137.82)
JPEG	55041814	2.0	(128,32,2)	22.50(2.92)	42.03(16.90)	192.90(627.71)		747.08(2137.82)

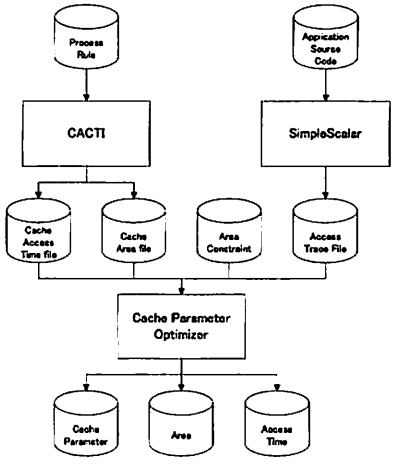

図 4 提案システム

Simple Scalar/PISA 3.0d [8] を用いる.

キャッシュ容量 m, ブロックサイズ b, 連想度 a の探索範囲は式 (2)〜(4) のようになる. ただし, CACTI の仕様上, $m/(b \times a) < 8$ の場合は面積, アクセス時間の見積もりができないため, 対象となる構成数は 460 個となる.

$$m = 2^i : 6 \leqq i \leqq 22 \tag{2}$$

$$b = 2^i : 3 \leqq i \leqq 10 \tag{3}$$

$$a = 2^i : 0 \leqq i \leqq 4 \tag{4}$$

キャッシュ構成 $s = (m, b, a)$ のアクセス時間を $T(s)$, 面積を $A(s)$, 面積制約を A_c とする. 本手法では, 式 (5) を満たす構成 (s_{opt}) を探索することを目的とする.

$$S = \{s | A(s) \leqq A_c\}$$
$$\forall s \in S \text{ について}, T(s_{opt}) \leqq T(s) \tag{5}$$

制約条件がない場合, 3 章で提案した複数連想度同時探索手法, CRCB1 手法, CRCB2 手法, CRMF1 手法, CRMF2 手法を用いて, $A_c = \infty$ とした式 (5) を満たす構成を探索する.

面積制約を導入する場合, 3 章のアルゴリズムに加え, 制約を利用した探索範囲の削減を行っている. まず, $b = 8$ の探索範囲において, キャッシュ構成 $s = (m, b, a)$ を考える. m, a を固定し, どんな b を選択しても, 面積制約を満たせない場合は探索対象から除外し, ヒット/ミス判定を行わない. これによって, 探索範囲を削減しつつ CRMF1 手法を適用することができる. さらに, b を操作する際は面積制約を満たさない構成のヒット/ミス判定を行わない. これにより, さらに高速にヒット/ミス判定を行うことができる.

5. 計算機実験

キャッシュ構成最適化システムを C++言語によって計算機上に実装した. 実験環境は, OS が Debian GNU/Linux, CPU は Intel Pentium III 933MHz, メモリ容量 384MB, コンパイラが GNU/GCC version3.4.6, 対象アプリケーションは MediaBench [2] の adpcm, jpeg のエンコーダである. 計算機実験結果を表 2 に示した. トレースサイズの項目はトレースファイルのサイズ, 実行時間の括弧内に表記されている数値はヒット/ミス判定回数を示している. また, 文献 [7] を除き, 全手法で同じ解が選択された.

表 2 から, 本手法は文献 [1] との比較では, adpcm エンコーダでは平均 6.36 倍, jpeg エンコーダでは平均 8.50 倍, 文献 [7] との比較では, adpcm エンコーダで 4.31 倍, jpeg エンコーダで 4.14 倍実行時間が改善されている. さらに, CRMF1 手法および CRMF2 手法を実装せず, 全探索と完全に等価なシミュレーションを行った場合でも, 文献 [1] との比較では, adpcm エンコーダでは平均 4.89 倍, jpeg エンコーダでは平均 4.58 倍, 文献 [7] との比較では, adpcm エンコーダで 2.81 倍, jpeg エンコーダで 2.30 倍実行時間が改善されている.

6. むすび

本稿では, キャッシュ構成最適化アルゴリズムを提案し, その評価実験を行った. 既存手法と比較し, 最大で 8.50 倍高速にキャッシュ構成を最適化することを示した. 今後は, 命令キャッシュや L2 キャッシュに適用できるようにする必要がある.

文献

[1] A. Japasatya, A. Ignjatovic, and S. Parameswaran, "Finding optimal L1 cache configuration for embedded systems," in *Proc. of ASP-DAC 2006*, pp.796–801, 2006.

[2] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications system," in *Proc. of 30th Annual International Symposium on Microarchitecture*, 1997.

[3] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "CACTI 4.0," *HP Lab Technical Reports*, 2006.

[4] J. L. Hennessy and D. A. Patterson, コンピュータ・アーキテクチャ, 日経 BP 社, 1999.

[5] J. Edler and M. D. Hill, "Dinero IV trace-driven uniprocessor cache simulator," http://www.cs.wise.edu/ markhill/DineroIV/.

[6] J. J. Pieper, A. Mellan, J. M. Paul, D. E. Thomas and F. Karim, "High level cache simulation for heterogeneous multiprocessors," in *Proc. of the 41st Design Automation Conference*, pp. 287–292, 2004.

[7] 堀内一央, 小原俊逸, 戸川望, 柳澤政夫, 大附辰夫, "アプリケーションプロセッサ向けデータキャッシュ最適化システムとその評価," 信学技報, VLD2006-122, ICD2006-213, 2007.

[8] T. Austin, E. Larson and D. Ernst, "SimpleScalar:an infrastructure for computer system modeling," *Computer Vol.35*, pp. 59–67, 2002.