計算機アーキテクチャ 152－17
（２００３．３．１０）

キャッシュラインの時間情報を利用する Time Based Load Filter の提案

檜田敏克[†]　吉瀬謙二[†,††]
本多弘樹[†]　弓場敏嗣[†]

キャッシュアクセスのレイテンシが増加しており、小容量で高いヒット率を実現するキャッシュシステムが求められている。我々は小容量のキャッシュを有効に活用する機構として、Time Based Load Filter(TBLF)を提案する。Time Based Load Filter では L1 キャッシュと同じ階層に Load Buffer と呼ぶ小容量バッファを設置する。キャッシュミスしたラインを Load Buffer に配置し、L1 キャッシュ内にロードするラインのフィルタリングを行う。フィルタリングの際にはキャッシュラインの時間情報を利用する。本方式を機能レベルのシミュレータに実装し、SPECint2000 の 12 個のプログラムで評価したところ、ダイレクトマップのキャッシュに対して平均 5.04%のミスを改善できた。

Time Based Load Filter using Time Information on Cache Lines

TOSHIKATSU HIDA,[†] KENJI KISE,[†,††] HIROKI HONDA[†]
and TOSHITSUGU YUBA[†]

The latency of the cache access is on the increase, and the cache system which realizes a high hit rate in the small capacity is necessary. We propose Time Based Load Filter (TBLF) as a mechanism to use the small capacity cache effectively. The small capacity buffer called Load Buffer is located at the same level of L1 data cache. The cache line where a mistake was made is stored in Load Buffer, then only the selected line is moved to L1 cache. The time information of the cache line is used in the selection. We implemented TBLF on the functional level simulator and evaluated it with 12 benchmark programs of SPECint2000. We found that TBLF removes 5.04% of cache misses over a conventional direct mapped cache.

1. はじめに

半導体の微細化によりプロセッサの動作速度は飛躍的に向上している。これに伴い、メモリシステムの動作速度が相対的に遅くなり、メモリアクセスがプロセッサの性能に与える影響が深刻な問題となっている。メモリアクセスのレイテンシを隠蔽するために、多くのプロセッサでは大容量の階層化されたキャッシュを用いて性能向上を図ってきた。

しかし、近年では半導体の微細化により、ゲート遅延と比べて配線遅延が大きくなってきている。このため、キャッシュ容量を増加させるとアクセス時間が増大し、レイテンシの増加につながってしまう。現在では L1 キャッシュでさえ 1 サイクルでアクセスすることが困難な状況になってきている。今後、半導体の微細化がさらに進み、配線遅延が支配的になると言われている[1]。そのためにキャッシュアクセスのレイテンシはさらに増加すると考えられ、キャッシュのレイテンシとキャッシュ容量間のトレードオフが重要な問題となる。

配線遅延が支配的となる状況下では、L1 キャッシュなどの上位階層のキャッシュにおいて、キャッシュアクセスの低レイテンシを維持しながらヒット率を向上させる技術が必要となる。レイテンシを抑えるためには、上位階層のキャッシュの容量と連想度を高くできない。このような場合には、1 つのキャッシュエントリに複数のラインが割り当てられる。そのためにライン同士が競合してしまい、キャッシュに格納されたラインが再利用される前に置き換えが発生することで性能が低下する。

複数のラインが競合している場合には、競合しているラインの中から L1 キャッシュなどの上位階層のキャッシュへ配置するものを正しく選択し、限られた容量の上位階層のキャッシュを有効活用する必要がある。しかし、従来のキャッシュにおけるライン配置方式では、参照の局所性により上位のキャッシュでミスしたラインは自動的に上位の対応するキャッシュエントリに格納される。このために、頻繁にアクセスするキャッシュラインや再

† 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems The University of Electro-Communications
†† 科学技術振興事業団　さきがけ研究 21
“Information infrastructure and applications”, PROESTO, Japan Science and Technology Corporation(JST)

利用する前のキャッシュラインを不用意に置き換えてしまい、ミスの増加に繋がってしまうと考えられる。

そこで本稿では、ラインの競合による不用意な置き換えを軽減させる事を目的としたライン配置方式として Time Based Load Filter (TBLF)を提案し、その性能向上の可能性について検討する。この方式では従来のように L1 キャッシュなどの上位階層のキャッシュでミスしたラインをそのまま対応するキャッシュのエントリに配置するのでなく、小容量のバッファを設けてその中にラインを配置し、参照される間隔などの時間情報を観察する。そして、参照される間隔が長いために、再び参照される前に追い出されてしまうラインをキャッシュ内に入れないようにフィルタリングし、容量の限られた上位階層のキャッシュの有効活用を図る。

本稿の構成を以下に示す。2 章で関連研究について述べ、3 章で Time Based Load Filter を提案する。また 4 章で実装方法について述べ、5 章で評価および検討を行う。また 6 章で今後の課題について述べる。

2. 関連研究 Time-based Techniques

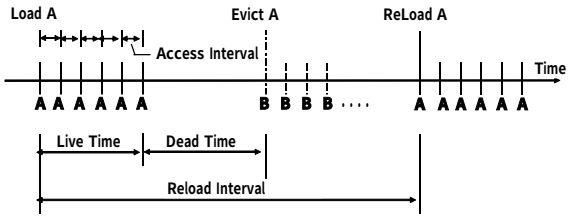

図 1　Time-based Techniques

図 1 において、ラインが参照される時間について観察を行う。あるライン A はプロセッサによって積極的に参照される時間と、キャッシュの中から追い出されるのを待っている時間に分けられる。この積極的に参照される時間は Live time、追い出されるのを待っている時間は Dead time と定義される[2]。Live time は新しくラインがキャッシュにロードされてから始まり、プロセッサによる最後の参照により終わる。また Dead time はそのラインが最後に参照されてからキャッシュの外に追い出されるまでの時間である。この他に、Live time 中でラインが参照される間隔を Access Interval, ラインがキャッシュへロードされる間隔を Reload Interval と定義されている[2]。ここでキャッシュの上位階層での Reload Interval は 1 つ下の記憶階層の Access Interval である。

文献[2]における詳細な調査により Live time、Dead time、Access Interval、Reload Interval の 4 つの時間情報にはそれぞれ特徴的な性質があるということが明らかになった。例えば、競合性のミスの場合には Dead time が短く、容量性のミスの場合には Dead time が長いという特徴がある。この観察から、ラインの競合によりキャッシュ中から追い出されたラインを小容量のバッファに一時的に保持することで競合を軽減させる victim cache[3]に対して、文献[2]では時間情報に基づき victim cache の中に配置するラインを効果的に選択する Timekeeping Victim Cache Filter が提案された。Timekeeping Victim Cache Filter では、ラインがキャッシュから追い出されたときにそのラインの Dead time を基に容量性のミスか、競合性のミスかを予測する。

この時、競合性のミスと判定されたものだけが victim cache に入るようにフィルタリングを行うことで、再利用する可能性の低いラインが victim cache の中に入ることを防いでいる。また、Live time を利用したプリフェッチも提案され、ライン参照の時間情報を利用することの有効性が明らかになっている。

3. Time Based Load Filter の提案

この章では 3.1 節において既存のキャッシュにおける問題点と原因について述べる。3.1 節の問題に対して、キャッシュラインに Time-based Techniques の考え方を導入し 3.2 節で考察し、3.3 節で Access Interval に着目したミスの緩和方法を述べる。3.4 節でキャッシュラインの時間情報を利用することにより、キャッシュへの不用意なロードを防ぐ Time Based Load Filter を提案する。

3.1. 既存のキャッシュの問題点と原因

1 つのキャッシュエントリにおいて、2 つのライン A と B が競合している図 2 のような場合を考える。

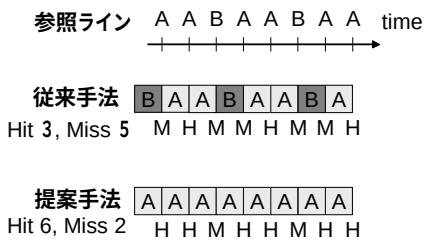

図 2　競合が発生しているキャッシュエントリ

従来のライン配置方法では、上位階層のキャッシュでミスしたラインは自動的に下位の記憶階層から対応するキャッシュエントリにロードされる。これはプログラムには参照の時間的局所性があり、最近参照したラインをキャッシュにロードするために発生する。

図 2 の従来手法おいて、ライン B が参照ミスによりキャッシュにロードされた後にライン A が参照される。このため、ライン B は再び参照される前に追い出されてしまう。またライン B をキャッシュ中にロードすることにより、後続のライン A の参照でミスが発生してしまう。

このような状況が発生する原因として、従来のライン配置方法では置き換え対象のラインが必要であるかどうかを考慮せずに、キャッシュでミスしたラインを上位階層のキャッシュに配置するからであると考えられる。

3.2. Time-based Techniques の導入

1 章で述べたように、容量が限られた上位階層のキャッシュを有効活用する場合には、必要のないラインを上位階層のキャッシュに入れない、あるいは残さないことが重要となる。

ここで、あるキャッシュラインを置き換える場合において、新しくキャッシュにロードされるラインと置き換えられるラインの状態に Time-based Techniques の考え方を導入し、次の(1)〜(4)の 4 つの状態に分類することを考える。

(1) 双方のラインが Live time
(2) 双方のラインが Dead time
(3) 新しくキャッシュに配置されるラインが Dead time、置き換えられるラインが Live time
(4) 新しくキャッシュに配置されるラインが Live time、置き換えられるラインが Dead time

プログラム中で 1 度しか使用しないラインは、キャッシュにラインが格納された時点で Live time が終わる。つまりこの時の Live time は０であり、キャッシュに格納された時点から Dead time が始める。この場合が新しくキャッシュにロードされるラインが Dead time にある状態である。

状態(1)は、図２で示した参照パターンのような場合である。この場合における競合する 2 つのラインは Live time である。そのため、キャッシュエントリから追い出されてしまっても再び参照されるので、同じキャッシュエントリをめぐって２つのラインがお互いを追い出しあう。このときラインの参照間隔である Access Interval が長いライン（図２ではライン B）は、Access Interval が短いラインによって再参照される前に置き換えられる。つまり状態(1)では Access Interval がより長いラインをキャッシュにロードする意味はなく、不用意な置き換えを招いてしまう。この状態(1)は容量の小さいキャッシュの場合に多く発生すると考えられる。

状態(2)は双方のラインがもう使われることがない場合である。この時、置き換えによるヒット率の低下は発生しない。しかし、キャッシュ中のラインを再参照されないもので置き換えているため無意味な置き換えだと考えられる。この置き換えを行うことによる問題点は下位の記憶階層とのトラフィックの増加である。

状態(3)は先に説明した一度しか使用しないラインが上位のキャッシュ階層に格納され、Live time であるラインを置き換えてしまう状態である。置き換えられるラインが Live time であるためにすぐにキャッシュから追い出されてしまう。

状態(4)は適切な置き換えといえる。なぜならば置き換えられるラインがもう使用されないからである。

従来のライン配置方法ではこれら状態(1)〜(4)の違いを考慮していない。このために状態(1)〜(3)の場合にヒット率の低下、あるいは記憶階層間のトラフィックの増加を招いてしまう。

文献[2]では、Dead time にあるラインを過去の Live time 履歴情報を利用して予測する方法が示されている。ここではキャッシュ中のラインが最後にアクセスされてから過去の Live time の 2 倍以上経過しているラインを Dead time にあると予測する。この手法のように何らかの方法でラインの Live time と Dead time を予測し、見分けることができる場合、状態(2)、(3)では、置き換えを行わないことでミスを防ぐことができる。

しかし状態(1)の場合には Live time にあるラインのうち、どちらのラインを配置する、あるいは残せばいいかという問題が生じるため、ラインが Live time にあるか Dead time にあるかを見分けることだけではミスを軽減させることは難しい。

3.3. Access Interval によるミスの緩和

前節の状態(1)に示した競合する 2 つのラインが Live time である時には、2 つのラインが交互に参照される場合とそうでない場合が考えられる。

2 つの競合するラインの Access Interval がほぼ同じでラインが交互に参照される場合には、交互のラインが順番にキャッシュから追い出されるので、どちらも再参照されることはなく下位の記憶階層とのトラフィックが増加するだけである。

またラインが交互に参照されない場合とは図 2 の参照パターンのように一方のラインが参照される間に、他方が複数回参照される場合である。一方が参照される間に複数回参照されるもの、すなわち Access Interval が短いものがある場合、他方の Access Interval が長いラインをキャッシュにロードしても再参照される前に必ずキャッシュから追い出されてしまう。例えば、図２の従来手法においてライン B はキャッシュへ格納しても再参照されることはない。

このようにどちらの場合においてもすべてのラインをキャッシュに配置するのではなく、競合する 2 つのラインが交互に参照される場合ではどちらか一方を配置せず、競合する 2 つのラインが交互に参照されない場合には Access Interval の長いラインをキャッシュにロードしないことによりミスを軽減できると考えられる。例えば図 2 の提案手法のように、ライン B をキャッシュに配置しな

いことでミスを 3 回防ぐことができる。このように Live time にある 2 つのラインが競合する場合、Access Interval に着目し、比較することでキャッシュにロードする必要のないラインを見つけ出すことができると考えられる。

3.4. Time Based Load Filter の提案

これまでの議論から、本稿では L1 キャッシュなどの容量の限られた上位階層のキャッシュエントリにおいて、競合するラインが Live time にあるかなどの時間情報を考慮することにより、キャッシュへロードすべきでないラインのフィルタリングを行い、不用意なラインの置き換えを防ぐ Time Based Load Filter (TBLF)を提案する。

TBLF では競合するラインの時間情報である Live time、Access Interval に着目し、3 章の 2 節で場合分けした状態の違いにより以下のフィルタリングを施す。

状態(1)の場合は競合するラインの Access Interval を比較し、Access Interval が長いラインを、また状態(2), (3)では全てのラインをキャッシュへ格納しないようにフィルタリングを行う。また状態(4)の場合には無条件でラインをキャッシュに格納する。

提案手法と従来のライン置き換え方法との違いは、キャッシュでミスが起きた場合に全てのラインをキャッシュに配置するのではなく、ラインをキャッシュに配置する前に、ラインの時間情報からキャッシュへ配置するラインを選択する点である。

提案手法のようにラインがキャッシュに配置される前にフィルタリングを行う手法の他に、文献[2]の Timekeeping Victim Cache Filter のようにラインがキャッシュから追い出された後に、どのラインをそのキャッシュ階層に残すかフィルタリングを行う方法がある。しかし本稿では、この 2 つの手法の比較は行っていない。

4. Time Based Load Filter の実装

この章では 4.1 節において TBLF の実装方法とその動作を示し、4.2 節で競合するラインの比較について述べる。4.3 節では前節の比較方法で使用する時間情報の取得方法について述べる。

4.1. Time Based Load Filter の実装方法と動作

TBLF は複数のラインが競合している場合に、競合しているライン同士の時間情報を考慮することで不要な置き換えが生じるラインをキャッシュに配置しないようにフィルタリングする手法である。

容量が少ない上位階層のキャッシュでは、この不用意な置き換えが生じる可能性が高いため、本稿では TBLF を小容量の L1 データキャッシュに適応する。

TBLF の構成を図３に示す。L1 データキャッシュと同じキャッシュ階層に小容量のバッファである Load Buffer（LB）を設け、この階層でミスしたラインを LB 内に格納する。LB はフルアソシアティブの FIFO 構造である。

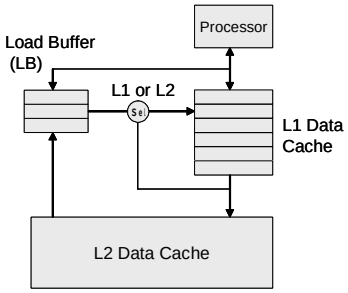

図３　Time Based Load Filter のブロック図

TBLF では、プロセッサからのライン参照はセットアソシアティブのキャッシュや victim cache の場合と同様に、LB と L1 データキャッシュを同時に参照する。このとき参照するラインが LB、あるいは L1 データキャッシュのどちらかでヒットした場合、従来と同じようにプロセッサにラインを供給する。

しかし、このキャッシュ階層でミスが起こった場合にはミスしたラインが LB へ格納され、替わりに LB から１つのラインが追い出される。

LB から 1 つのラインが追い出されたとき、追い出されたラインと L1 データキャシュ中の対応するラインで、時間情報によりどちらのラインを L1 データキャッシュに残すか、あるいは格納するかを判断する。ここで比較される時間情報の詳細は次節で述べる。この比較の際に、LB から追い出されたラインが L1 データキャッシュにロードする必要がないと判断された場合には L2 データキャッシュに格納され、必要と判断された場合にのみ L1 データキャッシュへ格納される。

ここでラインが L2 データキャッシュから直接 L1 データキャッシュに移動することはなく、必ず LB を経由して必要なものだけが L1 データキャッシュに供給される。またラインが L1 データキャッシュから LB へ移動することはなく、L1 データキャッシュから追い出されたものは必ず L2 データキャッシュへ書き戻される。

4.2. ラインの比較手法

TBLF では、ラインが LB から追い出されたときに、追い出されたラインと対応する L1 データキャッシュのラインの時間情報を比較する。

しかし、使用する Live time, Access Interval などの時間情報は正確に知ることはできない。なぜならば、あるラインがキャッシュ中に配置されている場合、自分自身が Live time なのか Dead Time なのかはキャッシュから追い出されるまで知ることができないからである。これは Access Interval に関しても同じであり、アクセスされる前に Access Interval を知ることができない。よって Live

time, Access Interval を前もって得るためには何らかの方法によりこれらを予測する必要がある。つまりライン間の時間情報を比較する際には、比較対象の 2 つのラインが Live time であるかどうかの予測と Access Interval の間隔、あるいはどちらの Access Interval が長いかという予測の方法と精度が重要な問題になる。

今回の予備評価では、競合性のミスが起こったときは 3 章 2 節の状態(1)の 2 つのラインが Live time であると判定し、フィルタリングを行う。それ以外の場合には 3 章 2 節の状態(4)であると判定をし、そのまま L1 データキャッシュにラインを格納する。このときの競合性ミスと容量性ミスの判定方法は、文献[2]の方法を採用する。具体的にはキャッシュラインが最後にアクセスされてからのロードストアの回数を計測し、L1 データキャッシュ中のエントリ全てにアクセスするために必要となる回数よりも多い場合には容量性のミスと判定し、少ない場合には競合性のミスと判定する。

また、Access Interval が短いラインを予測する方法として、ラインが最後に参照されてからの経過時間とそのラインにおける前回までの Access Interval との和を比較する方法を用いる。この理由として、キャッシュラインがアクセスされてからの経過時間が短いラインを Access Interval の短いラインと単純に判定した場合では、図 4(a)のように比較を行うタイミングで Access Interval の長いラインＢの経過時間が短い場合がある。そこで図 4(b)のように前回までの Access Interval と最後にアクセスされてからの経過時間を合計することで、比較を行う時点から 2 回前の参照までの時間を計測することができるので、図 4(a)のような場合にも対処できる。

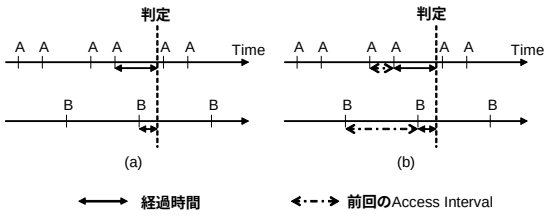

図４　Access Interval と経過時間

4.3. 利用する時間情報の取得

4.2 節で述べた Live time の判定方法と Access Interval の予測方法はともに最後のアクセスからの経過時間と前回アクセスした時までの Access Interval を予測のために使用する。本節ではこれらの時間情報の取得方法について述べる。

まず擬似的に時刻を表現する 32 ビット程度の飽和型カウンタを用意し、ロードあるいはストアが起こった時に値をインクリメントする。これは時間を表すグローバルタイムカウンタと呼ぶことにする。また LB と L1 データキャッシュの各ラインがこれらの時間情報をもつことができるように、図５に示すラインの拡張を行う。図５中の Last Access は 1 回前に参照された時刻を格納し、Old Time Stamp は 2 回前に参照された時刻を格納する。つまりラインを比較する時点において、そのときのグローバルカウンタと Last Access の差が最後にアクセスされてからの経過時間となり、Old Time Stamp から Last Access を引いたものが Access Interval となる。ラインの比較の際には Old Time Stamp からグローバルカウンタの値を引いた値を用いればよい。

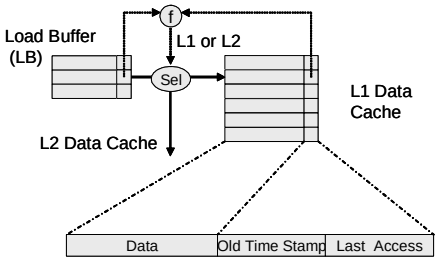

図５　キャッシュラインの拡張

キャッシュミスにより、ラインが LB に格納された際、あるいは、LB か L1 データキャッシュでラインがヒットした際に、そのときのグローバルタイムカウンタの値でラインに付随する Last Access の値を更新する。Old Time Stamp はそのときの Last Access の値で更新する。

5. 評価および検討

本章では TBLF を評価する。5.1 で評価環境と比較対照を示し、5.2 で評価結果を示す。

5.1. 評価環境と比較対照

評価に際して機能レベルシミュレータである SimAlpha1.0[4]へ L1 データキャッシュと TBLF を追加した。8K バイトのダイレクトマップの L1 データキャッシュにエントリ数を 1 から 64 の間で変化させた LB を追加する構成で評価を行った。ラインサイズは 32 バイトである。

提案手法の性能を、8～16Kbyte のダイレクトマップ、２ウエイセットアソシアティブのキャッシュと比較する。また本提案手法と同じく L1 データキャッシュに小容量のバッファ（vicitm cache）を付け加えた構成である L1 データキャッシュ＋vicitm cache という構成と比較する。ここで想定している victim cache は競合ミスにより L１データキャッシュから追い出されたラインをすべて順番に格納し、一時的に保持する FIFO 構造のバッファである。SPECint2000 の 12 個のプログラムを用いてミス率の測定をおこなった。

5.2. 結果および検討

キャッシュ容量が 8Kbyte のときのダイレクトマップ、2 ウエイセットアソシアティブ、ダイレクトマップ＋victim cache と提案手法 TBLF の評価結果を図 6 に示す。この時、victim cache と LB の容量は 256byte(8 エントリ)とした。

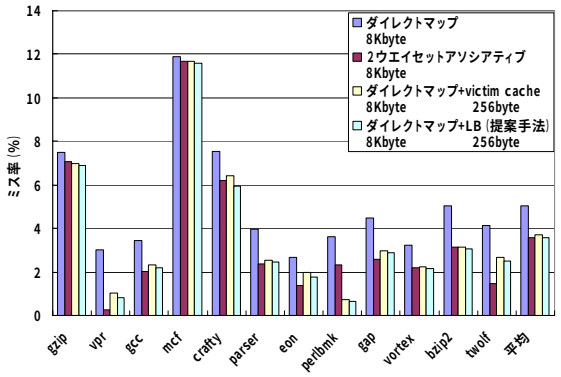

図6　プログラムごとのヒット率

それぞれのキャッシュにおける平均ミス率はダイレクトマップが 5.04%、2 ウエイセットアソシアティブが 3.56%、ダイレクトマップ＋victim cache が 3.72%となり、提案手法である TBLF では 3.57%という結果になった。

それぞれのプログラムの結果を比較すると、ダイレクトマップ、ダイレクトマップ＋victim cache という構成に対して、TBLF が全てのプログラムでミス率を減少できることが明らかになった。ここで、もっともミス率を減少させることができたのは、ダイレクトマップに対しては perlbmk の 2.95%、ダイレクトマップ＋victim cache に対しては crafty の 0.455%である。2 ウエイセットアソシアティブと比較した場合、6 個のプログラムにおいて TBLF のミス率のほうが低く、perlbmk においてはミス率を 1.64%低下させることができた。しかし perlbmk においてはダイレクトマップ＋victim cache の構成でもミス率を 1.58 %削減できており、L1 データキャッシュに付属する小容量のバッファがミス率の改善に役立っていると考えられる。ここで crafty、vortex、bzip2 の 3 つのプログラムは L1 データキャッシュ＋vicitm cache の構成では 2 ウエイセットアソシアティブのキャッシュよりもミス率が高いが、TBLF の場合には 2 ウエイセットアソシアティブのキャッシュよりミス率が改善されている。この場合はフィルタリングによりミスが改善されていると考えられる。

比較を行った各キャッシュ構成のミス率の平均を図 7 に示す。X 軸はキャッシュ全体の容量であり、図右にある TBLF は L1 データキャッシュの容量と、LB の容量を分けて表示している。この時のラインサイズは 32byte なので TBLF のグラフにおいて LB の左端が 1 エントリ、右端が 64 エントリとなる。エントリ数を増加させるにつれて一定の間隔でミス率が減少し、８ K の L1 データキャッシュに 16 エントリ 512byte の LB を追加した場合、8Kbyte の 2 ウエイセットアソシアティブや 16Kbyte のダイレクトマップのキャッシュより低いミス率を達成できた。

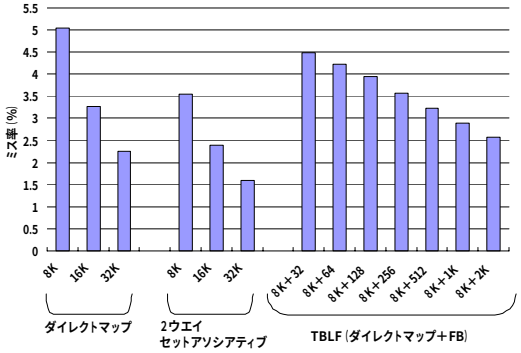

図7　平均ヒット率の比較

これらの結果より、提案した TBLF を用いて、L1 データキャッシュにラインがロードされる前に必要でないラインをフィルタリングすることが多くのプログラムで有効であるということが明らかになった。

6. 今後の課題

今回の評価に使用した SPECint2000 の全てのプログラムにおいて TBLF が一定の性能向上を達成することが明らかになった。しかしながら、今回行った評価では機能レベルシミュレータを用いており、性能を示す IPC にどのように影響を及ぼすかは評価できていない。そこでアウトオブオーダ実行をおこなうクロックレベルのシミュレータ上に提案手法を実装し、詳細な評価をおこなう必要がある。またラインの比較を行う際に、時間情報を判定する方法の改良も必要である。

参　考　文　献

[1] Changkyu Kim, Doug Burger, Stephen W. Keckler : An adaptive, Non-Uniform Cache Structure for Wire Delay Dominated On-chip Caches, Proceedings of the 10th annual international confirence on Architectural Support for Programming Languages and Operating Systems , (2002).

[2] Zhigang Hu, Stefanos Kaxiras, Margaret Martonosi :Timekeeping in the Memory System:Predicting and Optimizing Memory Behavior. The 29 th Annual International Symposium on Computer Architecture, (2002).

[3] John L. Hennessy and David A. Patterson: Computer Architecture A Quantitative Approach Third Edition, Morgan Kaufmann Pub., pp421-422(2002).

[4] 吉瀬謙二,本多弘樹,弓場敏嗣:ShimAlpha:C++で記述したもうひとつの Alpha プロセッサシミュレータ,情報処理学会研究報告, Vol.2002, No.81 (ARC:SWoPP),pp.163-168 (2002).