修 士 論 文

時間情報に基づいたキャッシュの置換方式に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

塩安 麻人

2006 年 3 月

修 士 論 文

時間情報に基づいたキャッシュの置換方式に関する研究

指導教官　田中清史 助教授

審査委員主査　田中清史 助教授
審査委員　日比野靖 教授
審査委員　井口寧 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

410058 塩安 麻人

提出年月: 2006 年 2 月

概 要

近年，CPU と主記憶の性能格差が問題となっており，遅延の大きい主記憶へのアクセス回数を減らすために，高いキャッシュヒット率を達成することが重要である．キャッシュの容量を増やすことや，連想度を高くすることはヒット率向上に有効であるが，キャッシュへのアクセスレイテンシが増大する問題がある．したがって，限られた連想度のアソシアティブ方式を使用しているが，置換方式の選択がヒット率に大きく影響する．

本研究ではキャッシュヒット率向上を目的とし，キャッシュブロックのアクセスに関する時間情報に基づき，今後使用されないと予測されるブロックを置換の対象として選択する方式を提案する．本方式をシミュレータに実装し, SPECint95 で評価した.

目 次

第1章　はじめに

1.1　研究背景と目的

近年プロセッサの動作速度が飛躍的に向上している．これに伴い, 主記憶の動作速度が相対的に遅くなり, プロセッサと主記憶の性能格差が問題となっている．この問題に対処するためにキャッシュメモリを用いているが, キャッシュミスによる主記憶へのアクセスが頻発した場合にやはり性能が大きく低下する．主記憶へのアクセス増加を防ぐために, 上位階層のキャッシュでは高いキャッシュヒット率を維持させる必要がある．ヒット率の向上を図るために, キャッシュの容量や連想度を高くすることは可能であるが, アクセス時間の観点から見ると有効ではない．したがって, 限られた連想度のアソシアティブ方式を使用する．アソシアティブ方式のキャッシュでミスが発生すると, 複数のブロックの中から置換対象のブロックを選択しなければいけない．近い将来にアクセスされるブロックを置換対象として選択した場合, ミスが起こる可能性が高い．近い将来にアクセスされないブロックを優先して置換対象として選択することが重要である．現在, キャッシュブロックを置換する方式として一般的にLRU 法が使用されている．本論文では 1 次データキャッシュおよび, 1 次命令キャッシュにおいてブロックの置換を行う際に, キャッシュブロックの持つ時間情報 [1] に基づいて, 今後アクセスされないブロックを予測し, 置換対象として選択する方式を提案する．これにより, キャッシュミス数を削減しキャッシュヒット率向上を目的とする．

1.2　本論文の構成

本論文は以下のように構成する．
第 2 章では従来のキャッシュメモリ, 及び時間情報について述べる．
第 3 章では提案手法である時間情報に基づいた置換方式について述べる．
第 4 章では提案手法の有効性をシミュレーションにより評価し, 考察を述べる．
第 5 章では関連研究を紹介する．
最後に第 6 章で本論文の結論を述べる．

第2章　キャッシュメモリ

本章では, 従来のキャッシュメモリ,LRU 法, 及びキャッシュブロックの時間情報について述べる.

2.1　従来のキャッシュメモリ

2.1.1　キャッシュの基本構成

キャッシュとはメモリ参照に関する以下の傾向を利用し, 低速な主記憶へのアクセスを減らすために設けられる高速で少量のメモリである.

- あるアドレスを参照した場合, 高い確率でその周辺のアドレスを参照する傾向
- あるアドレスを参照した場合, 高い確率で短時間のうちにそのアドレスを再び参照する傾向

前者を参照の空間的局所性と呼び, 後者は参照の時間的局所性と呼ぶ.

一般的なキャッシュは, 図 2.1 に示すように参照アドレスの上位部分を格納するタグ領域とデータを格納するデータ領域のテーブルで構成している. それぞれのエントリは参照アドレスの中位部分（インデックスフィールド）を基にアクセスを行う.

プロセッサからメモリ参照の要求があると, 参照アドレスのインデックスフィールドが指し示すエントリのタグとデータを同時に読み出す. プロセッサが要求するデータがキャッシュ内に存在するかどうかを判断するために, 参照アドレスの上位部分とキャッシュ内のタグを比較する. 比較した２つの値が一致した場合には, 参照アドレスの上位部分とキャッシュ内のタグが比較される. ２つの値が一致した場合には, キャッシュ内に格納されているデータをプロセッサへ供給する. プロセッサが要求する参照データがキャッシュ内に存在した場合をキャッシュヒットと呼び, 存在しない場合をキャッシュミスと呼ぶ. キャッシュミスの原因は以下の３つに分けることができる.

- 初期参照ミス
  - まだキャッシュに読み込まれていないブロックを最初にアクセスした際に発生するミス
- 容量性ミス

  - プログラムを実行する間に必要となるすべてのブロックをキャッシュに保持することができないことに起因するミス

- 競合性ミス
  - 複数のブロックが１つのキャッシュエントリを巡って競合するために発生するミス

プロセッサの初期状態では, キャッシュ内にはデータが存在しない. この時のタグの比較結果は有効ではない. したがって, キャッシュ内には有効なデータが存在するかどうかの有効ビットを備えている.

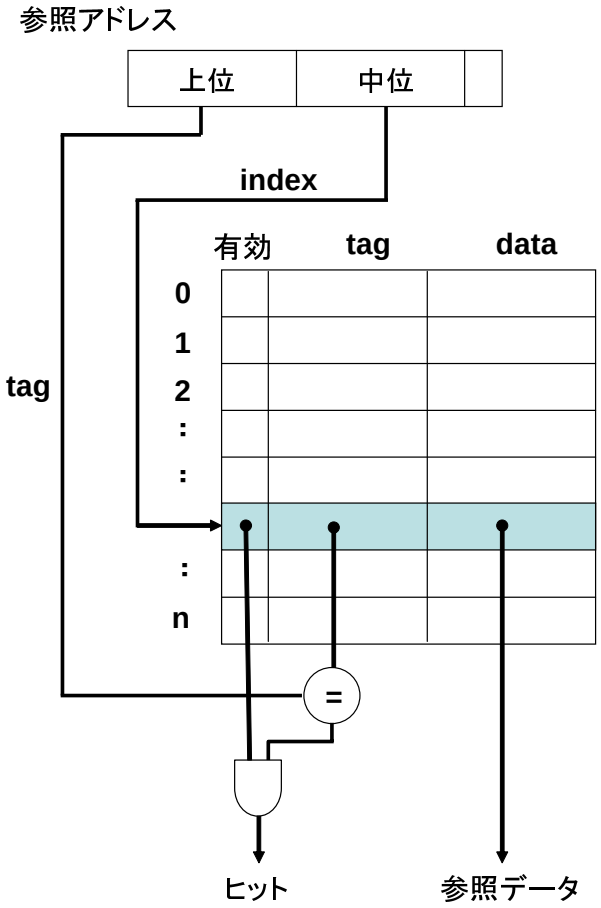


図 2.1: ダイレクトマップ方式のキャッシュ

2.1.2 連想度

図 2.1 に示すキャッシュの構造では, データを格納する場所は参照アドレスのインデックスフィールドが指し示す 1 箇所のみである. これはダイレクトマップ方式と呼ぶ. 参照アドレスのインデックスフィールドが等しく, 上位部分（タグフィールド）が異なるデータは, 同一のキャッシュエントリをめぐって互いにリプレースしあい, 競合性ミスが発生する.

競合性ミスを削減する方法として, 図 2.2 に示すような参照アドレスのデータを格納可能な場所を複数用意するセットアソシアティブ方式がある. データを格納する場所の数を連想度と呼び, データやタグを格納する場所をウェイと呼ぶ.
セットアソシアティブ方式のキャッシュの場合, 参照データを複数の場所へ配置できるため, 多くのキャッシュミスを削減することができる. プロセッサからメモリ参照が発生したとき, 2 ウェイセットアソシアティブキャッシュは, 以下のように動作する.

- 全てのウェイにおいて, 参照アドレスのインデックスフィールドが指し示すエントリのタグとデータを同時に読み出す. データを格納する場所が複数存在するため, 連想度の数だけタグとデータの読み出しを行う.
- 上記において読み出されたタグとプロセッサが出力した参照アドレスのタグフィールドの値を比較する.
- タグ比較結果に基づき, 比較結果が一致したウェイが存在する場合, そのウェイから読み出したブロックを選択する.

一般にキャッシュの大きさが同じ場合, 連想度の高いキャッシュの構成の方が競合性ミスを多く削減することができる. しかし, 2 ウェイセットアソシアティブ方式のキャッシュの場合, 図 2.2 に示すように比較器 2 個とセット内の 2 つの候補から 1 つのデータを選択する 2 対 1 のマルチプレクサを必要とする. したがって, アクセス時間が増大する. 連想度, アクセス時間, ミス率には表 2.1 のようなトレードオフの関係がある.

	アクセス時間	ミス率
連想度 高い	大きい	低い
連想度 低い	小さい	高い

表 2.1: 連想度の影響

ダイレクトマップ方式では, 連想度の欠如が原因で多くの競合性ミスが生じる. 一方, ヒットにおけるアクセス時間の観点では, ダイレクトマップ方式のキャッシュは, セットアソシアティブ方式のキャッシュより性能は良い.

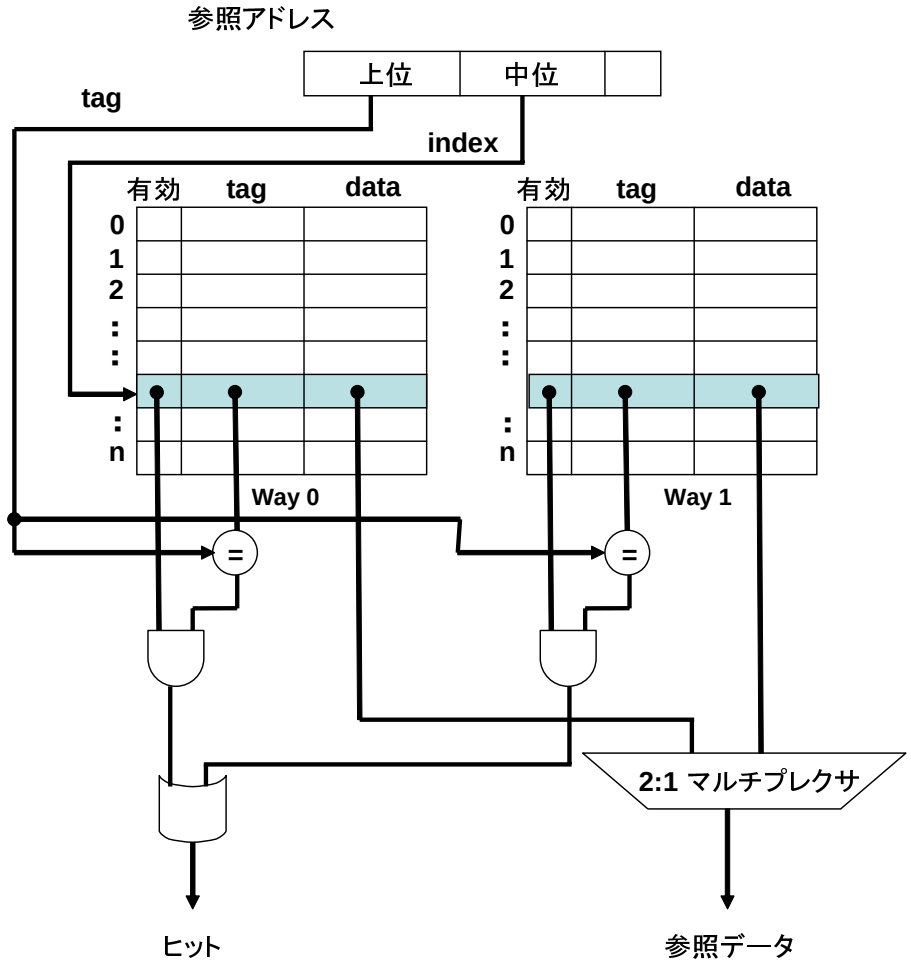


図 2.2: セットアソシアティブ方式のキャッシュ構造

セットアソシアティブ方式の連想度を最も高くすると, メモリブロックをキャッシュ内の任意のエントリに配置することが可能になる. この方式はフルアソシアティブ方式と呼ばれ, 図 2.3 に示すような構造になる. フルアソシアティブ方式のキャッシュで求めるデータを見つける場合, 求めるブロックが任意の場所に存在するためキャッシュ内の全てのエントリを検索する必要がある. そのため, キャッシュヒットにおけるアクセス時間が増大する. フルアソシアティブ方式は, ブロック数が少ないキャッシュにのみ適用している.

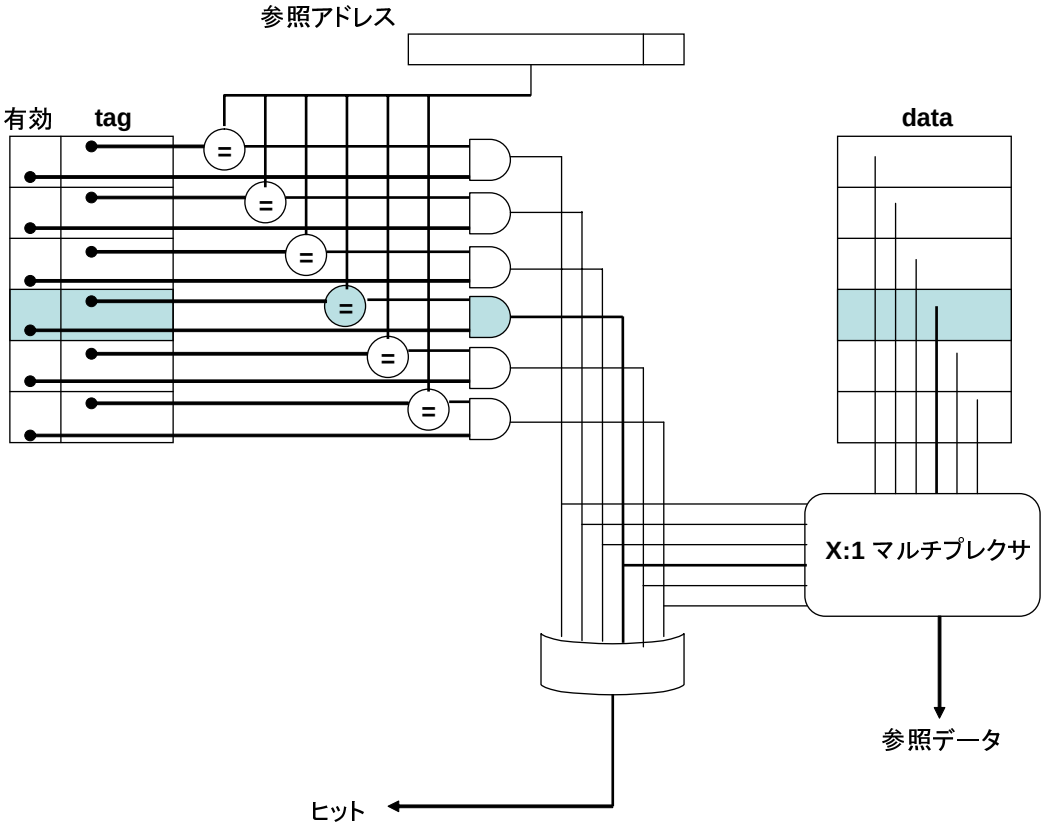


図 2.3: フルアソシアティブ方式のキャッシュ構造

2.2 LRU 法

ダイレクトマップ方式のキャッシュでミスが発生したときは, 要求されたブロックを収めている場所は 1 箇所しかないため, そこを占有しているブロックを置き換える. アソシアティブ方式のキャッシュでミスが発生したときは, 要求されたブロックを収める場所を選択できるため, どのブロックを置き換えるかを決めなければならない. フルアソシアティブ方式のキャッシュの場合は, すべてのブロックが置き換えの候補となる. セットアソシアティブ方式のキャッシュの場合は, 該当のセットの中からブロックを選択する.

最も一般的な方法は LRU（least recently used）法である.LRU 法では使用されずにいた時間が最も長いブロックを置き換える. それを実現するには, セット内の各要素が使われた時がほかの要素より早いか遅いかという情報を記憶する. 2 ウェイセットアソシアティブキャッシュの場合には, 各セットごとに 1 ビット保持すればよい. そして, 要素を参照するたびに, どちらが使用されたかをそのビットに記憶する. 連想度を上げるにつれて,LRU 法の実現は難しくなる.

2.3 時間情報

キャッシュブロックの参照される時間情報に関して, 図 2.4 のように live time, dead time などがあり, 次のように定義している.

- live time
  - ブロックがキャッシュにロードされてから, リプレースされる前の最後のヒットまでの間の時間.
- dead time
  - 最後のヒットから, リプレースされるまでの間の時間.

図 2.4 において縦線はキャッシュへのアクセスを表し, 横線は時間を表している. 1 の時間で,A というブロックがキャッシュにロードされ, アクセスを繰り返し, 2 の時間に A への最後のアクセスがある. さらに, 3 の時間に A はリプレースされ,B がキャッシュにロードされアクセスを繰り返すというアクセス履歴を表している. つまりこの図では 1 から 2 までの間の時間がブロック A の live time となり, 2 から 3 までの間の時間がブロック A の dead time になる. さらに 3 から 4 がブロック B の live time, 4 から 5 がブロック B の dead time, 5 から 6 がブロック C の live time になる. すなわち,live time 中のブロックは近い将来にアクセスが発生し,dead time 中のブロックは近い将来にアクセスが発生しないということになる.

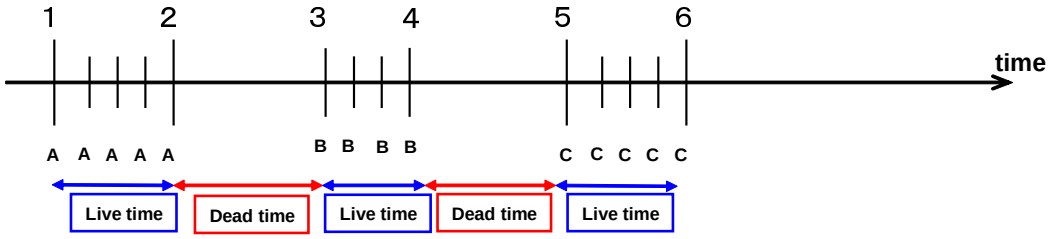


図 2.4: キャッシュブロックに関する時間情報

2.4 LRU 法の問題点

図 2.5 で表されるようなメモリアクセス傾向のあるとし, 4 ウェイセットアソシアティブキャッシュにおいてミスが発生した場合に LRU 法によって置換するという例を示す. 図 2.5 で縦線はキャッシュアクセスを表し, 横線は時間を表している. 四角で囲まれているところはキャッシュブロックが live time 中であることを表している. 図 2.5 でキャッシュア

クセスと表される時間に新たなブロックへのアクセスがあるとする. この場合そのアクセスから最も最近アクセスされていなかった４のウェイを置換対象として選択する.
しかし, ４のウェイで置換されたブロックは live time 中であるので置換された後すぐにアクセスされる. よってこのブロックに対するアクセスはミスとなり, またすぐキャッシュにロードされる. このミスにより LRU 法により３のウェイを置換し, ロードされる.
LRU 法を時間情報に当てはめて表すと, 最もアクセスされていなかったブロックを dead time 中になったと予測し, リプレース対象として選択することになる. 正確に dead time 中であるブロックを予測されているならば図 2.5 で表されるキャッシュアクセスの時間に dead time 中である３のウェイを選択し, リプレースする. このように選択されるなら LRU 法よりキャッシュミスが１回少なくなる.
つまりブロックが dead time 中であるか live time 中であるのかを正確に予測することでミス数を削減することが出来る.

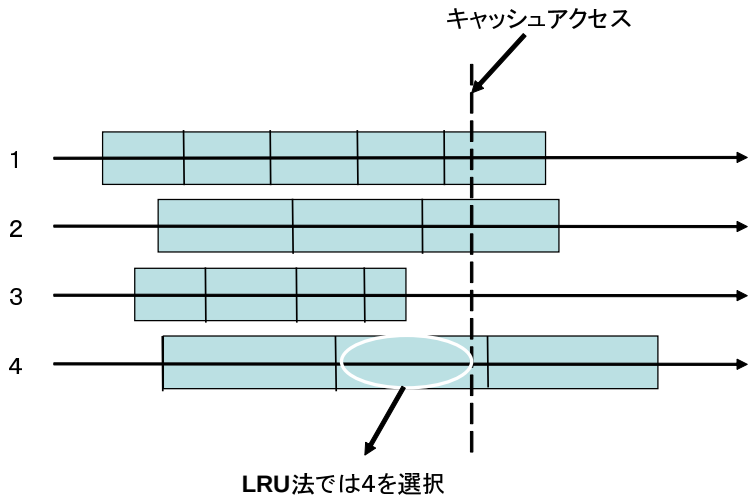


図 2.5: ４ウェイセットアソシアティブ方式のキャッシュにおけるキャッシュアクセスに対するリプレース対象の選択

第3章　時間情報に基づいた置換方式

アクセスレイテンシの大きい主記憶へのアクセスを少なくするためにキャッシュヒット率を向上させることが重要になっている. 容量が限られた上位階層のキャッシュでは置換方式が重要となる. ２章で述べたようにキャッシュブロックが live time 中であるか,dead time 中であるかを判断することはキャッシュミス数削減につながる.

本章ではセットアソシアティブ方式の１次キャッシュにおけるキャッシュヒット率向上のために, キャッシュブロックの時間情報に基づいて live time 中であるか,dead time 中であるかを予測し, リプレース対象を選択する置換方式を提案する.

3.1　提案手法

3.1.1　基本方針

本研究では, キャッシュブロックが dead time 中になったと予測する判断にキャッシュブロックの前回の live time を使用する. つまり１次キャッシュからキャッシュブロックが置換され, 次にそのキャッシュブロックが１次キャッシュにロードされた場合に前回の live time が今回も同じ位の live time であろうと予測するということである.
まず, キャッシュブロックが１次キャッシュにロードされると, キャッシュブロックごとに live time を計測する. そして, キャッシュブロックが置換される際に２次キャッシュの置換されるブロックに計測した live time を記憶させる. さらに, そのキャッシュブロックが１次キャッシュにリロードされる場合, ２次キャッシュからデータと一緒に記憶させておいた live time をロードする. ここでロードされたキャッシュブロックはロードされた live time の時間内は live time 中であると予測される.
これにより, 前回の live time の時間中は live time 中であると予測し, 優先的にリプレース対象とはしない. 前回の live time の時間を過ぎたキャッシュブロックを優先的にリプレース対象とする. さらにリプレースされたブロックがリロードされた場合にも同様に, 今回の live time を次にロードされた場合に使用する.

各キャッシュブロックに対して, 初めてロードされたブロックは前回の live time は存在しない. よって, 前回の live time は０, つまり dead time 中になっているとする. 置換ブロックの選択には以下の方針をとる.

- １つのウェイが dead time 中と予測され, その他のウェイが live time 中と予測された場合

  - dead time 中と予測されたウェイを選択

- ２つ以上のウェイが dead time 中と予測された場合
  - dead time 中と予測されるウェイの中から LRU 法によって選択
- すべてのウェイが live time 中であると予測された場合
  - LRU 法で選択

3.1.2 live time の更新と応用方針

3.1.1 で述べた方針は１次キャッシュで置換されたブロックの live time は２次キャッシュに記憶され, リロードされる場合, すべてのブロックが前回の live time を使用し, 置換対象を決定することになる. これでは競合性ミスが起こったときに置換されたブロックは本当の live time はもっと大きいはずが, ２次キャッシュに記憶される live time が極端に小さくなる. そして, 競合性ミスが解消されても, 前回の live time は極端に小さくなっているままである. これではそれ以降そのブロックが１次キャッシュで使用される際に, すぐ dead time 中であると予測され, そのブロックばかりリプレース対象となる可能性がある.
このように前回の live time が小さくなりすぎる場合に備えて, ある一定の閾値を決め, 今回の live time が閾値より小さくなった場合に今回の live time は更新しないという方針をとる.

さらに,3.1.1 で述べた基本方針を少し変え以下の方針を使用する.

- １つのウェイが dead time 中と予測され, その他のウェイが live time 中と予測された場合
  - dead time 中と予測されたウェイを選択
- ２つ以上のウェイが dead time 中と予測された場合
  - dead time 中と予測されるウェイの中から LRU 法によって選択
- すべてのウェイが live time 中であると予測された場合
  - 極端に残り live time の小さいキャッシュブロックを選択
  - なければ LRU 法によって選択

この方針は置換対象を決定する際にすべてのウェイが live time 中と予測される場合に予測される live time が極端に小さいキャッシュブロックを選択することで, 予測が外れて live time 中と予測される場合でも実際 dead time 中になっているかもしれないというキャッシュブロックを優先してリプレース対象とすることを狙う. これにより,LRU 法より正確な予測をすることを狙う.

3.2 提案手法の実現と動作

3.2.1 キャッシュの拡張

前節で述べた提案手法を実現するために, まず live time の取得方法を述べる. 時刻を表すためにグローバルクロックカウンタを用意する. これは１サイクルごとに１インクリメントされ現在の時刻を表す. さらに, 図 3.1 に示すようにキャッシュブロックごとに追加の時間除法フィールドを用意する. １次キャッシュのブロックには,prev live time, load time, last access time という時間情報フィールドを, ２次キャッシュのブロックには,prev live time という時間情報フィールドを用意する. prev live time とは前回の live time を記憶するフィールドである. load time とはブロックがキャッシュにロードされたときのグローバルクロックカウンタの値がセットされ, ロードされた時刻をを記憶するフィールドである. last access time とはキャッシュブロックにアクセスされたときのグローバルクロックカウンタの値がセットされ, アクセスがある度に更新されていく. これはキャッシュブロックへの最後のアクセス時間を記憶するフィールドである. このような追加フィールドとグローバルクロックカウンタを用意する.

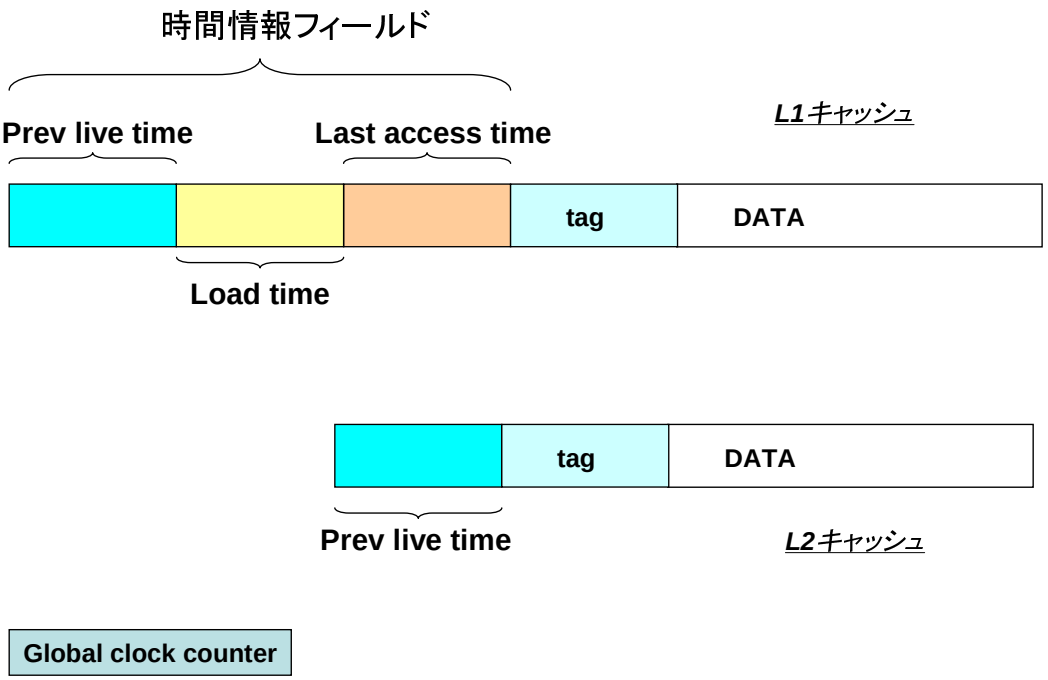


図 3.1: キャッシュの拡張

3.2.2 リプレースの際の動作

キャッシュブロックがリプレースされる際にリプレースされるブロックの追加フィールドを使用し, 以下の計算を行う.

$$prev\ live\ time = last\ access\ time - load\ time \tag{3.1}$$

これにより図 3.2 に示すようにキャッシュブロックがキャッシュにロードされてからリプレースされる最後のアクセスまでの間の時間である live time を取得する.

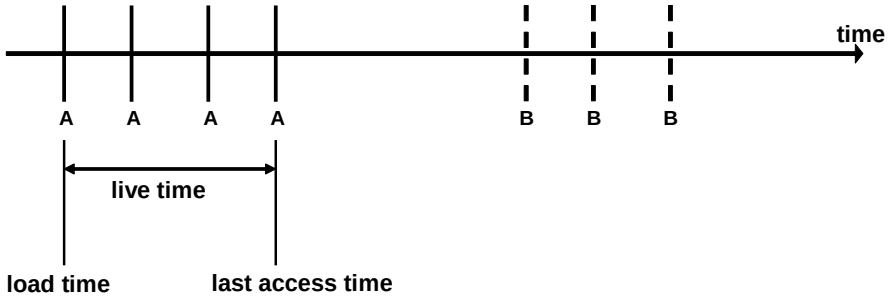
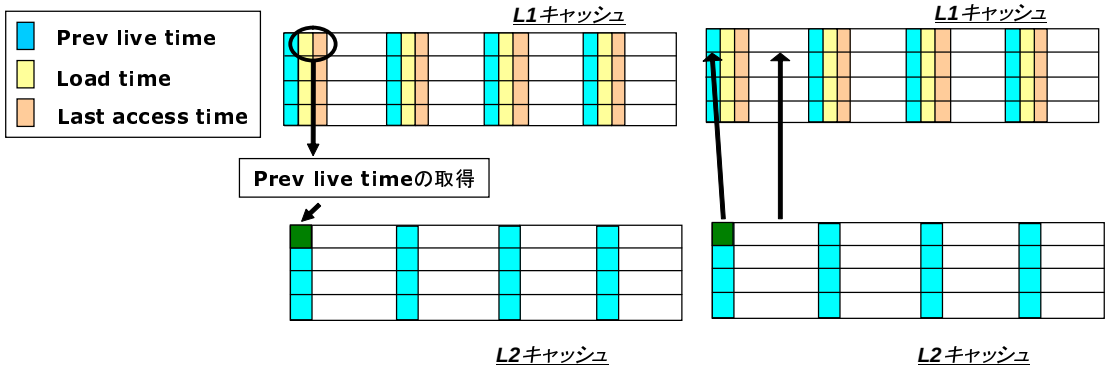


図 3.2: live time の取得

取得した prev live time は図 3.3（a）に示されるように, 2 次キャッシュ内のリプレースされるキャッシュブロックがある場所の prev live time に書き込まれる. そして, 時間が経過しリプレースされたブロックが 1 次キャッシュにリロードされる場合に, 先ほど 2 次キャッシュの prev live time に書き込まれた値は, 図 3.3（b）に示されるように, 2 次キャッシュからデータと一緒にロードされる.



(a) replace　　(b) reload

図 3.3: 前回の live time の推移

前回の live time を過ぎたか否かを判断するために, ２次キャッシュからデータと一緒にロードされた prev live time を使用する.prev live time が示す値はそのブロック前回１次キャッシュにロードされた際の live time を示している. よって, そのブロックが dead time 中になっていることを予測するためには prev live time を１サイクルごとに１デクリメントしていくことで値が０になれば dead time 中になっていると予測できる. しかし, すべてのキャッシュブロックに対して prev live を１サイクルごとにデクリメントしていては, 回路規模と消費電力の観点から問題となる. そこで, 現在の時刻を表す global clock counter, １次キャッシュにロードされた時刻を表す load time, 前回の１次キャッシュでの live time を表す prev live time を使用し以下のような計算を行い, リプレース対象を選択する.

- $(global\ clock\ counter - load\ time) - prev\ live\ time > 0$ の場合
  - dead time 中であると予測
- $(global\ clock\ counter - load\ time) - prev\ live\ time < 0$ の場合
  - live time 中であると予測

この計算は１次キャッシュでミスが起こり２次キャッシュおよび主記憶へデータ参照を行っている際に行うことができるため, 計算による計算時間を隠蔽することが出来る. さらに, 図 3.4 で示すように 1 次キャッシュでリプレース対象を決定する際に各ウェイごとに live time 中であるか, dead time 中であるかを並列に計算を行う. その結果によってリプレース対象を決定する. これにより２次キャッシュアクセスでのレイテンシの間にリプレース対象を決定することが可能である.

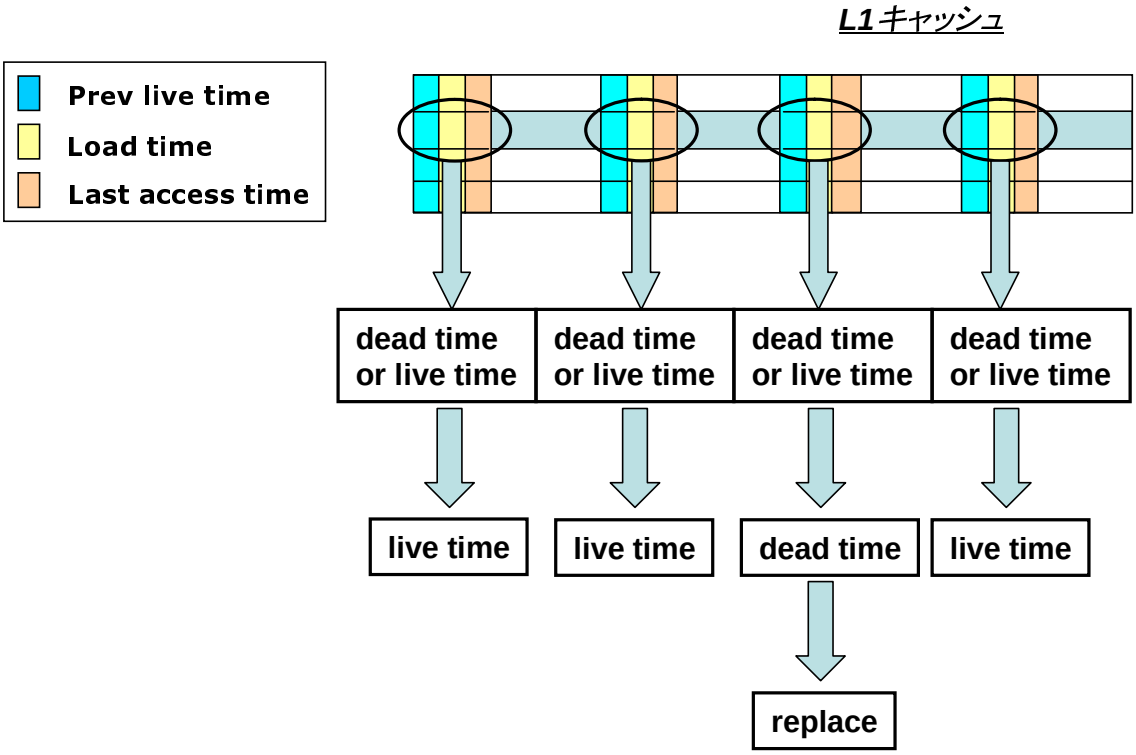


図 3.4: リプレースブロックの選択

3.2.3 応用方針の実装と動作

3.1.2 で述べた今回の live time が閾値より小さかった場合に今回の live time は更新せず前回の live time をそのまま使用するという方式では, 閾値を書き込むレジスタを用意する. それによりアプリケーションによってレジスタの値を変えることで閾値を変えれるようにする.

図 3.5 で示すようにレジスタに設定したい閾値をセットする. さらに,live time の計算を行い, 結果の live time と閾値との比較を行う. その結果によって今回の live time を 2 次キャッシュの prev live time に書き込むのか, 前回の live time を書き込むのかを決定する.

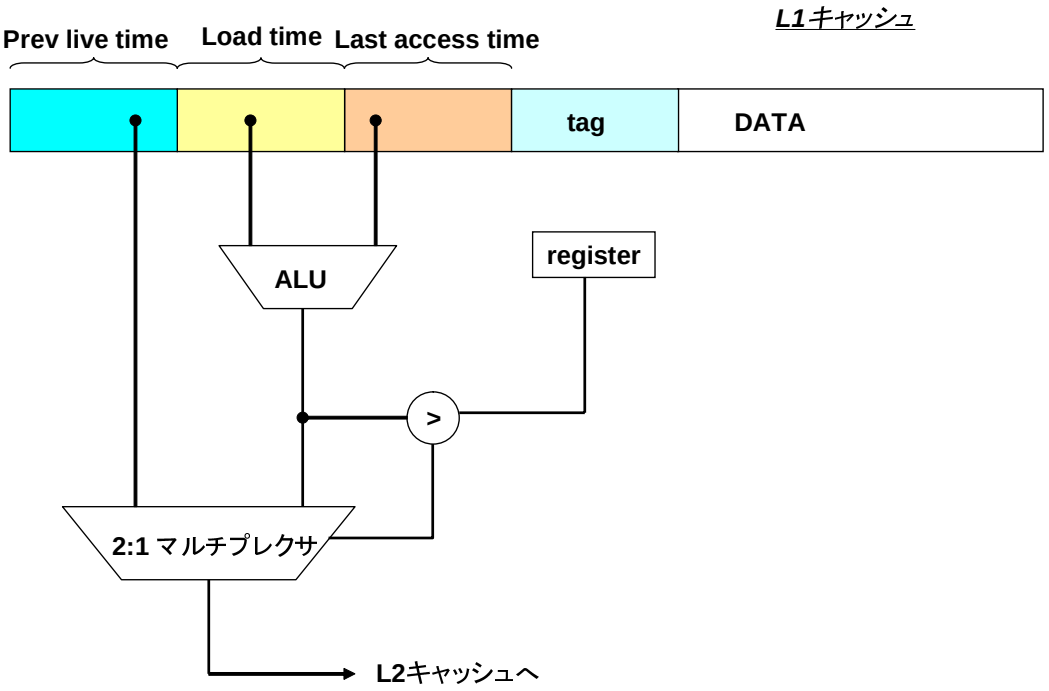


図 3.5: 閾値を設定した場合の live time の更新

3.2.4 時間情報フィールドの削減

図 3.1 で表されている時間情報フィールドは前節の説明では 1 クロック単位で正確に記憶させている. これではビット幅が大きくなりハードウェア量が大きくなる. この問題に対処するために図 3.6 で示すように制御タグフィールドの下位ビットを切り取り, 上位ビットのみを記憶させる. つまり,load time, last access time にはグローバルクロックカウンタの値の下位ビットを切り取った値が記憶され, prev live time には計算した live time の下位ビットを切り取った値が記憶される.

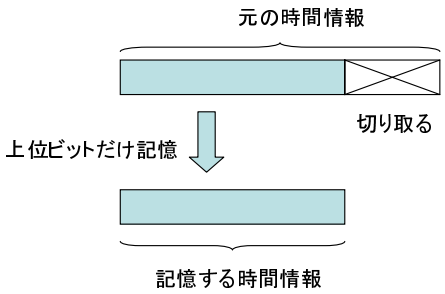


図 3.6: 時間情報の短縮

記憶していた時間情報をリプレースの際の計算で使用するときには, 図 3.7 で表されるように, 切り取ったビット幅と同じだけの 1 を埋める. これにより近似値で計算し, 予測することになるが, 各キャッシュブロックごとのタグ容量を削減することができる. 同様に, 上位ビットを切り取り, 下位ビットのみを記憶させることや, 上位ビット, 下位ビットの両方切り取り, 中位ビットのみを記憶させることで, さらにタグ容量を削減することができる.

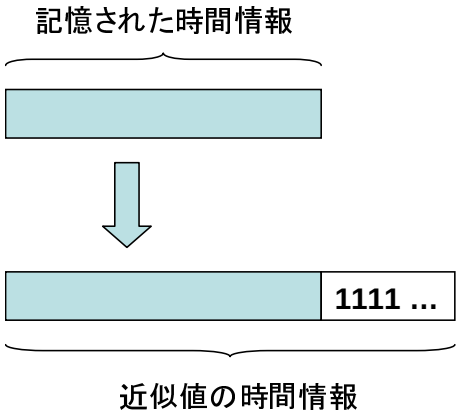


図 3.7: 時間情報の復元

第4章　評価

本章では, 本研究で提案する時間情報に基づいたキャッシュの置換方式の有効性をシミュレーションにより評価する.

4.1　ベンチマークプログラム

シミュレーションの対象プログラムに SPECint95[5] の 099.go, 124.m88ksim, 129.compress, 130.li のプログラムを用いる. ベンチマークプログラムの内容を表 4.1 にまとめる.

ベンチマーク	内容	入力データファイル
099.go	An internationally ranked go-playing program.	ref
124.m88ksim	A chip simulator for the Motorola 88100 microprocessor.	ref
129.compress	A in-memory version of the common UNIX utility.	train
130.li	Xlisp interpreter.	ref

表 4.1: SPECint95

4.2　評価方法

時間情報に基づいたキャッシュの置換方式を評価するために,C 言語で記述されたプログラムをコンパイルして生成したバイナリコード (MIPS64 命令セット [6]) を入力とするシミュレータを作成した. シミュレータはフェッチ, デコード, 実行, コミットを行うパイプラインシミュレータで, １クロックサイクルで４命令ごとにフェッチ, デコード, 実行, コミットを行うスーパースカラシミュレータである.
メモリ階層は１次データキャッシュ, １次命令キャッシュ, データ命令統合２次キャッシュと主記憶から構成される. １次データキャッシュおよび１次命令キャッシュはブロックサイズを 32 バイトの 4 ウェイセットアソシアティブで固定し, キャッシュサイズを 16KB,32KB,64KB の３種類で評価する.

評価対象として LRU 法を採用したキャッシュと比較評価を行う. 提案手法は前章で述べたように, 競合性ミスが発生している際に取得する live time が小さくなりすぎることを

防ぐために設ける閾値は, ミスが競合性ミスによるものか, 容量性ミスによるものかを判断するためにキャッシュのブロック数を基本とし, 適切な閾値を調べるためにブロック数×100 からブロック数÷100 までの間および, 閾値 0 つまり閾値を設けない場合で評価を行う. ブロック数は 16KB の場合は 512 となり,32KB の場合は 1024,64KB の場合は 2048 となる. また, さらに大きい閾値を設定した場合を評価するために, 各インデックスごとの最大の live time を測定し, 測定した値の平均値を閾値として利用する場合もまた評価する. これらをそれぞれシミュレータに実装し, シミュレータから出力されるキャッシュミス数で比較する.

さらに, 前章で述べた時間情報フィールドの削減方法を評価するために, 時間情報フィールドの prev live time の上位ビットを 4,8,12,16,20 ビット, 下位ビットを 4,8,12,16,20 ビット削減したものおよび, 上位ビットと下位ビットを 4,8,12 ビットずつ削減したものを評価する. prev live time の記憶ビットは 32 ビットからどれだけ削減できるかを評価した. よって,4 ビット削減したものは 28 ビットずつの情報タグフィールドがあるものとする.

4.3 結果と考察

4.3.1 099.go を入力とする結果と考察

099.go を入力とし, データキャッシュおよび命令キャッシュでの閾値を変化させた場合のキャッシュミス数をキャッシュサイズごとにグラフを図 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 に示す. インデックスごとの live time の最大値の平均は表 4.2 に示す.

キャッシュサイズ	データキャッシュの平均	命令キャッシュの平均
64KB	13399902	2126942
32KB	3805037	611177
16KB	1076385	388548

表 4.2: インデックスごとの最大値の平均

データキャッシュでは 64KB,32KB のキャッシュサイズの場合に閾値を小さく設定するにつれてミス数が減少している. この場合は閾値を設けないデータキャッシュが一番キャッシュミス数を削減している. これは閾値により live time を一定以上の値に保つことで, ミス数が増加していることから閾値による制限を加えず, 前回の live time をそのまま予測 live time として使うことで正確な予測ができ, ミス数削減につながった. 16KB のキャッシュサイズの場合は閾値を極端に大きくすることによってミス数を削減することができる. この場合は閾値をインデックスごとの live time の最大値の平均を閾値にした場合が最もミス数を削減している. これは閾値を高く設定することで, 予測 live time が大きくなり, すべて live time 中と予測する場合が多くなる. これにより,LRU 法と同様な選択をすることと, ある程度時間情報に基づいた置換をすることでミス数を削減につながった.

命令キャッシュではすべてのキャッシュサイズにおいて閾値を小さく設定した方がキャッシュミス数を削減することができる. すべてのキャッシュサイズで閾値を設定しない場合が最もミス数が削減される. 前回の live time を予測 live time として使うことで正確な予測ができていたと推測できる.

以下に最もキャッシュミス数の小さい閾値でのキャッシュヒット率および LRU 法でのキャッシュヒット率を示す.

データキャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	99.6416758	99.6939184	0.0522426
32KB	98.7546981	98.8214211	0.066923
16KB	97.2676973	97.2827831	0.0150858

命令キャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	99.2718567	99.4162404	0.1443837
32KB	98.7302117	98.8171292	0.0869175
16KB	95.2558469	95.8735702	0.6177233

表 4.3: キャッシュヒット率

データキャッシュでは 32KB の場合に最もミス率が向上する。キャッシュサイズが大きい場合に本提案手法の効果が大きい。命令キャッシュでは 16KB の場合が最もミス率が向上する。キャッシュサイズが小さい場合に本研究手法の効果が大きい。

図 4.7 と図 4.8 では時間情報フィールドの prev live time に対してビット幅を削減した場合のキャッシュミス数, 図 4.9 に live time の割合を示す. データキャッシュでは下位ビットを 12 ビットまで削減してもさほどミス数が増えない. これは 16 ビットまで削減した際に, 全体の半分以上の予測 live time が同じ値になり, 予測が困難になるためである. 上位ビットは 8 ビットまでさほどミス数が増えない. これは 8 ビット削減することによる影響を受けるのが全体の 1%であるためである. しかし,12 ビットまで削減すると全体に占める割合が増えミス数が増加する. これにより, 上位, 下位ビットを 8 ビットずつ計 16 ビット削減してもさほど影響を受けずに削減できる. 命令キャッシュでは下位 16 ビット削減してもミス数はさほど増えない. 上位ビットは 12 ビットまではさほどミス数は増えない. これは削減されることにより, 影響する割合が少ないからである. これにより, 上位, 下位ビットを 12 ビットずつ計 24 ビット削減してもさほど影響を受けずに削減できる.

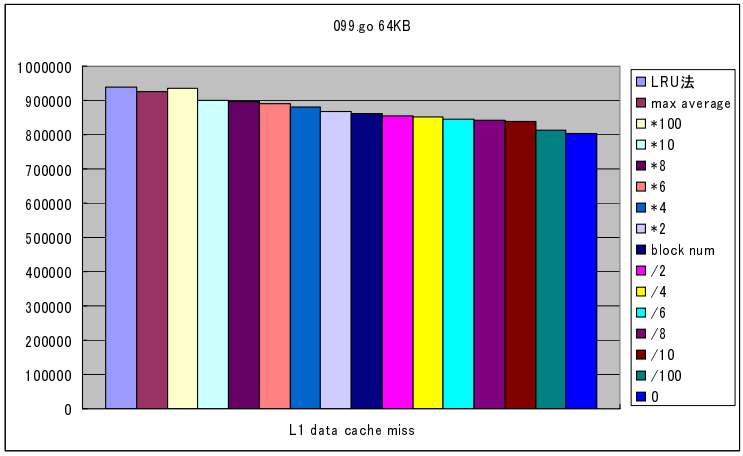


図 4.1: 閾値を変化させた場合の 64KB データキャッシュミス数

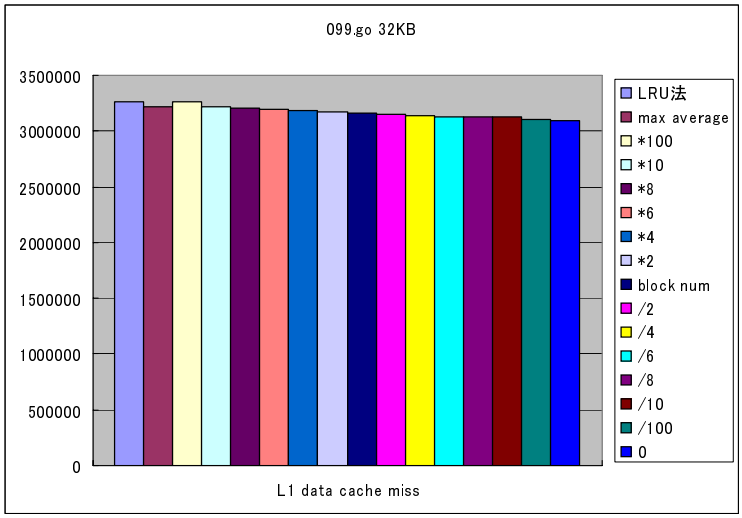


図 4.2: 閾値を変化させた場合の 32KB データキャッシュミス数

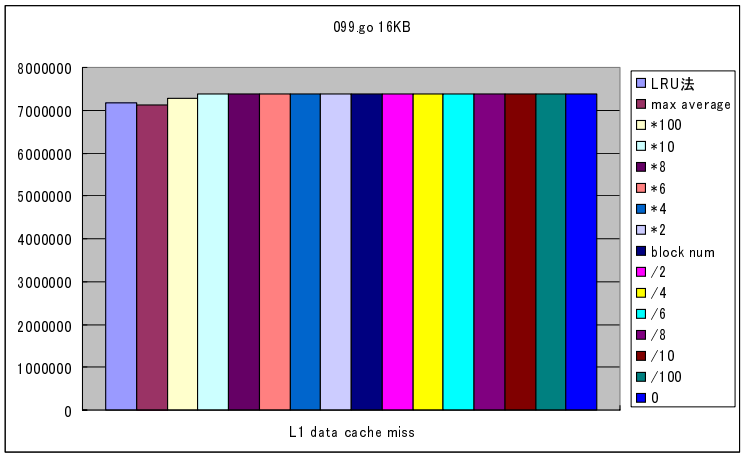


図 4.3: 閾値を変化させた場合の 16KB データキャッシュミス数

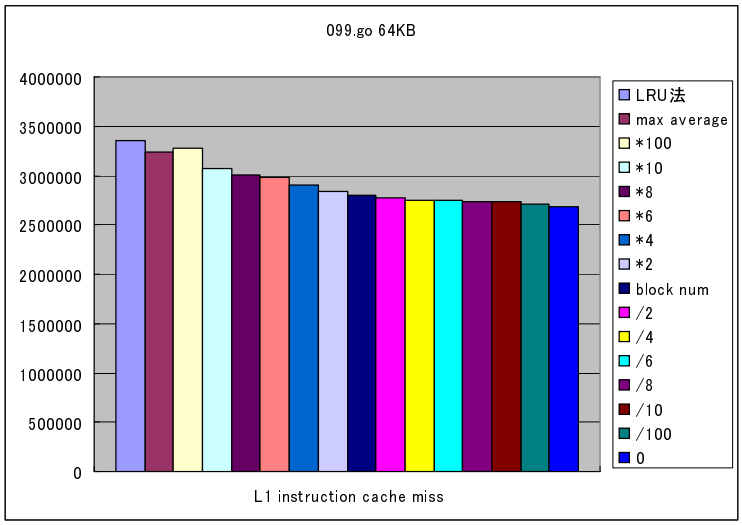


図 4.4: 閾値を変化させた場合の 64KB 命令キャッシュミス数

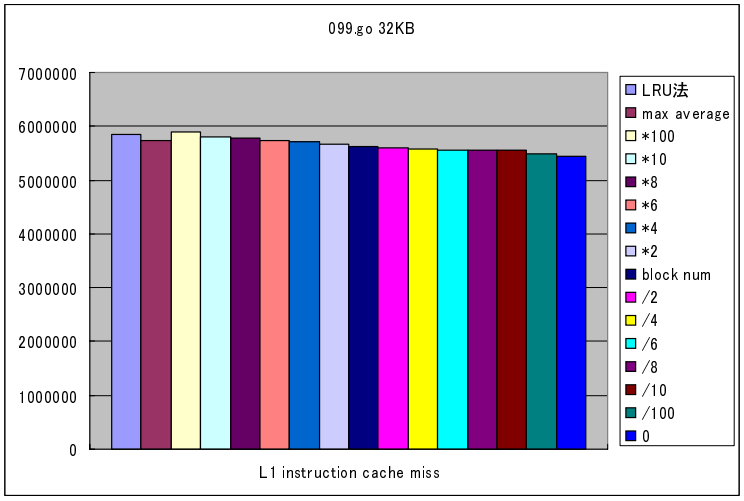


図 4.5: 閾値を変化させた場合の 32KB 命令キャッシュミス数

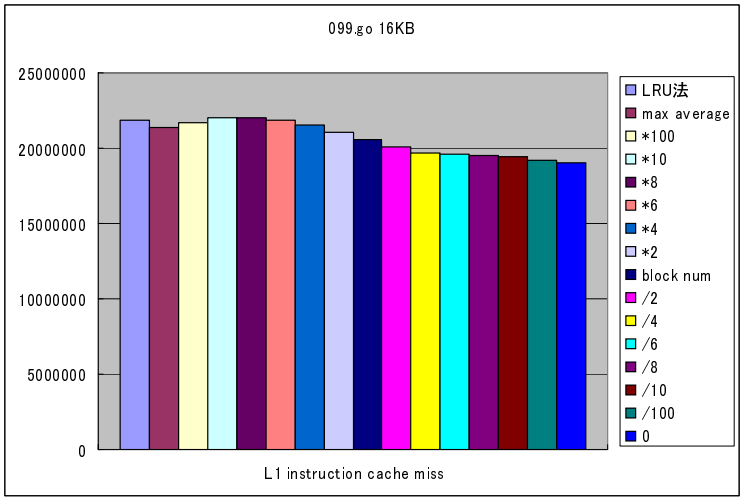


図 4.6: 閾値を変化させた場合の 16KB 命令キャッシュミス数

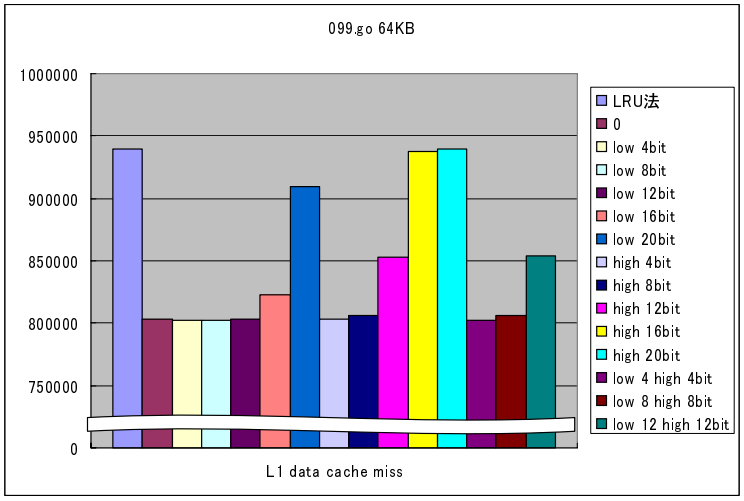


図 4.7: 時間情報フィールドを削減した場合の 64KB データキャッシュミス数

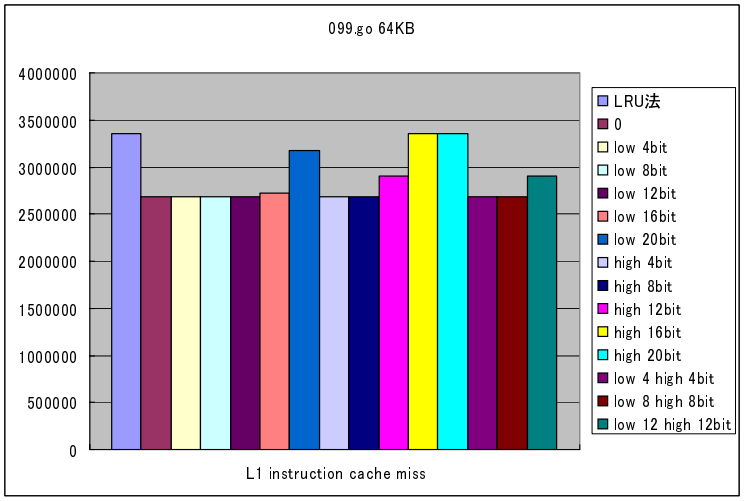
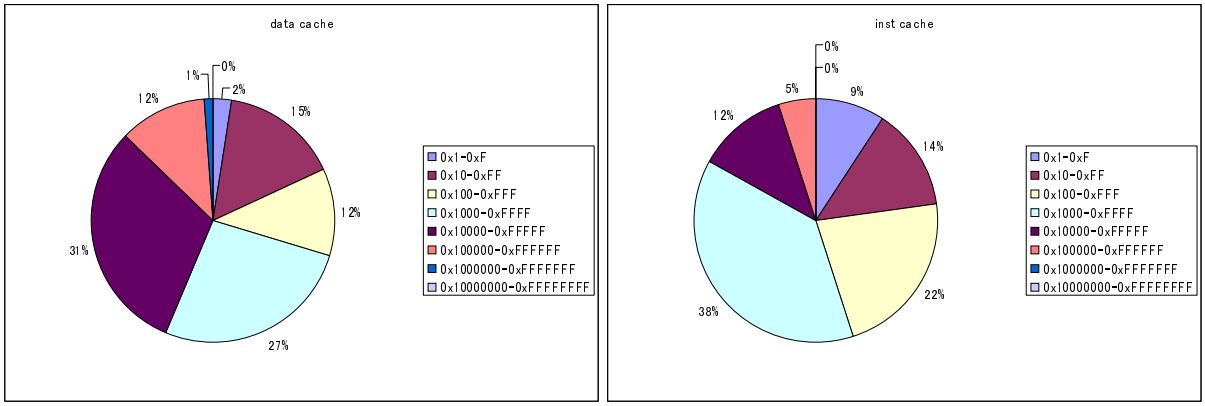


図 4.8: 時間情報フィールドを削減した場合の 64KB 命令キャッシュミス数



(a) data cache　　(b) instruction cache

図 4.9: 64KB キャッシュでの live time の割合

4.3.2　124.m88ksim を入力とする結果と考察

124.m88ksim を入力とし, データキャッシュおよび命令キャッシュでの閾値を変化させた場合のキャッシュミス数をキャッシュサイズごとにグラフを図 4.10, 4.11, 4.12, 4.13, 4.14, 4.15 に示す. インデックスごとの live time の最大値の平均は表 4.4 に示す.

キャッシュサイズ	データキャッシュの平均	命令キャッシュの平均
64KB	648317	0
32KB	506722	0
16KB	447998	69309

表 4.4: インデックスごとの最大値の平均

データキャッシュでは 32KB は閾値を設けないとミスが増加する. これはブロック数より小さい閾値を設定することで live time が小さすぎることを抑えることで的確な予測ができ, ミス数削減につながった. 16KB ではキャッシュブロック数 ×100 が最もミス数を削減している. これは閾値を高く設定することで, 予測 live time が大きくなり, すべて live time 中と予測する場合が多くなる. これにより,LRU 法と同様な選択をすることと, ある程度時間情報に基づいた置換をすることでミス数の削減につながった.

命令キャッシュでは閾値の設定を的確に行うことでキャッシュミス数の増加を抑えることができる. 閾値の設定を的確に行なわなければ予測 live time が適切に行われなくなり, ミス数が増加する.

以下に最もキャッシュミス数が最も小さくなる閾値でのキャッシュヒット率および LRU 法でのキャッシュヒット率を示す.

データキャッシュでは 16KB の場合に最もミス率が向上する。キャッシュサイズが小さ

データキャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	99.5250827	99.5250866	0.0000039
32KB	99.5242114	99.5242114	0
16KB	99.5237778	99.5238725	0.0000947

命令キャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	99.9998383	99.9998383	0
32KB	99.9998374	99.9998374	0
16KB	99.9997829	99.9997829	0

表 4.5: キャッシュヒット率

い場合に本提案手法の効果が大きい。

図 4.16 と図 4.17 では時間情報フィールドの prev live time に対してビット幅を削減した場合のキャッシュミス数, 図 4.18 に live time の割合を示す. データキャッシュではビットを削減してもミス数が増えない. これは小さい live time ばかりであるためビットを削減しても影響がないためである. 命令キャッシュについては初期ミスのみのため考察はしない.

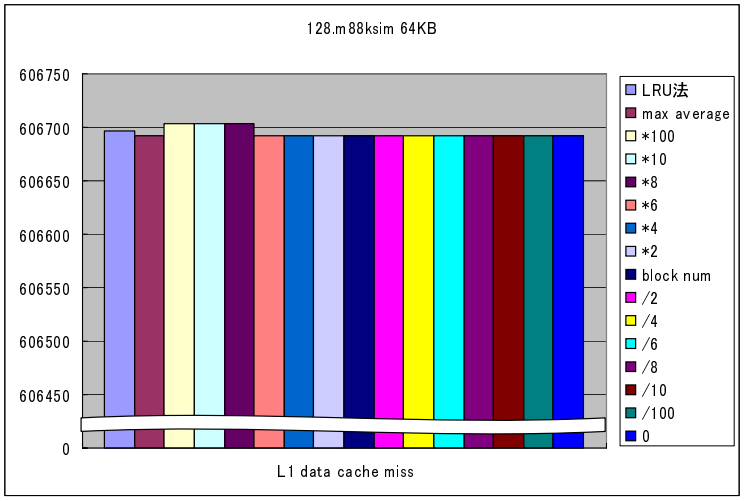


図 4.10: 閾値を変化させた場合の 64KB データキャッシュミス数

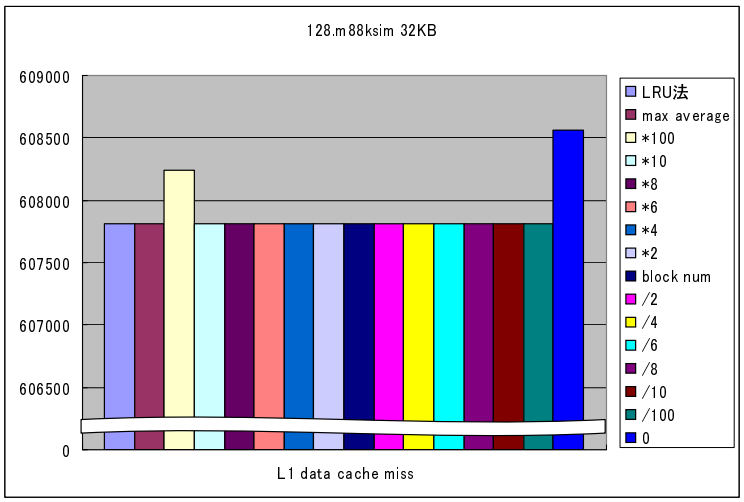


図 4.11: 閾値を変化させた場合の 32KB データキャッシュミス数

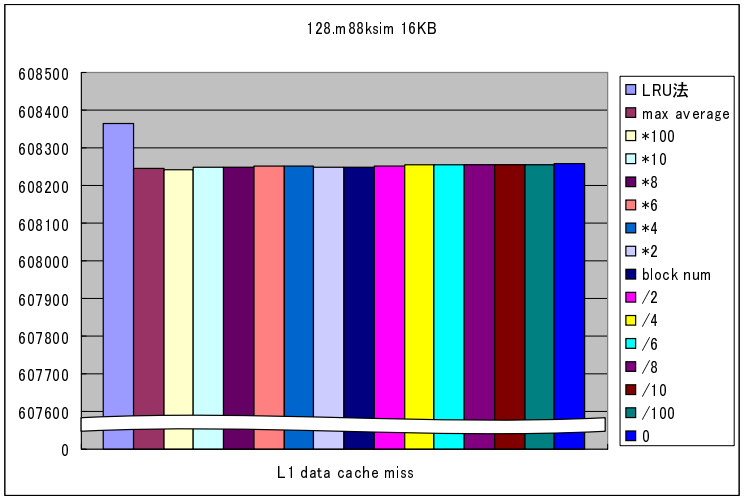


図 4.12: 閾値を変化させた場合の 16KB データキャッシュミス数

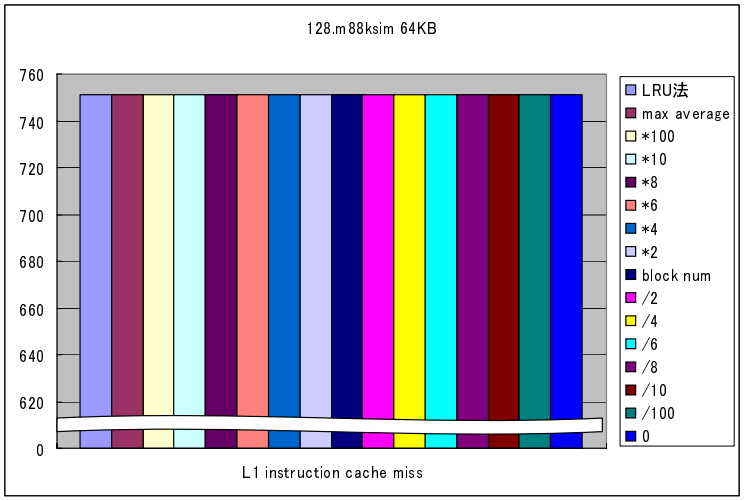


図 4.13: 閾値を変化させた場合の 64KB 命令キャッシュミス数

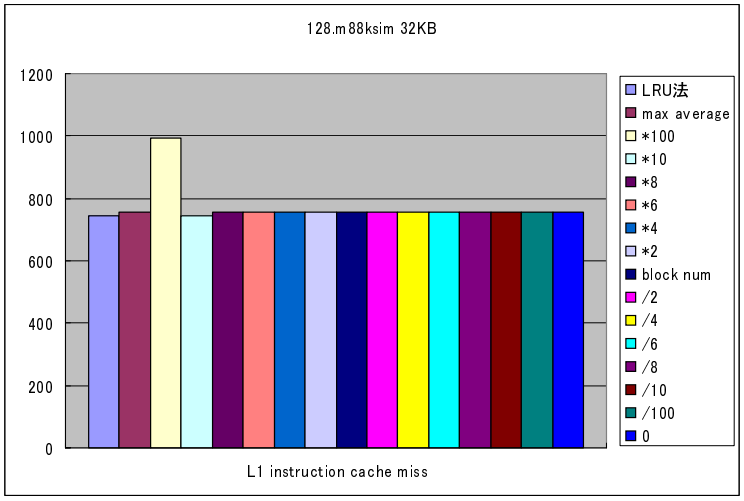


図 4.14: 閾値を変化させた場合の 32KB 命令キャッシュミス数

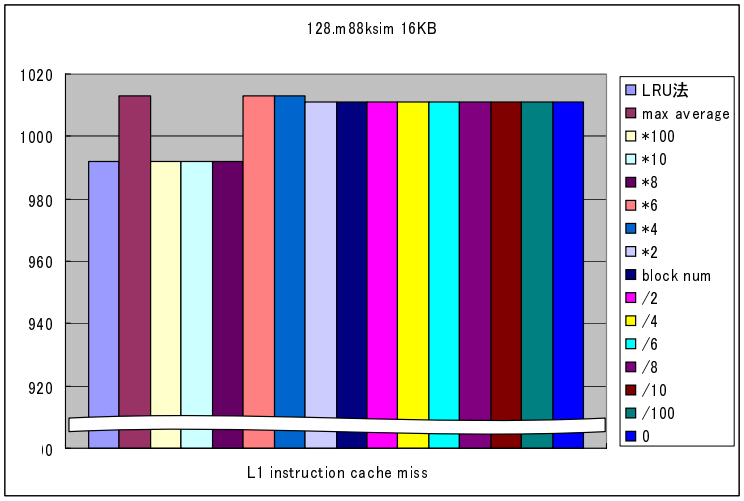


図 4.15: 閾値を変化させた場合の 16KB 命令キャッシュミス数

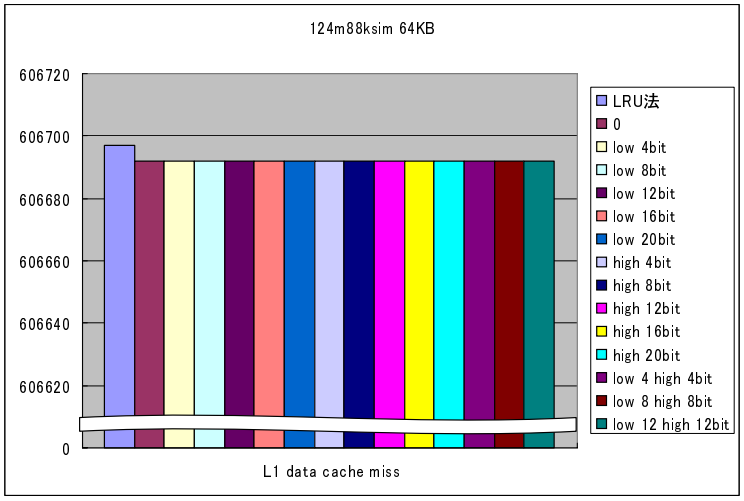


図 4.16: 時間情報フィールドを削減した場合の 64KB データキャッシュミス数

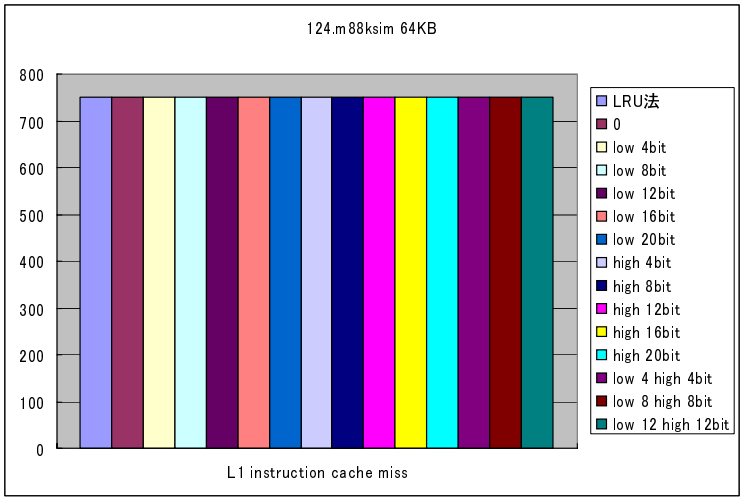


図 4.17: 時間情報フィールドを削減した場合の 64KB 命令キャッシュミス数

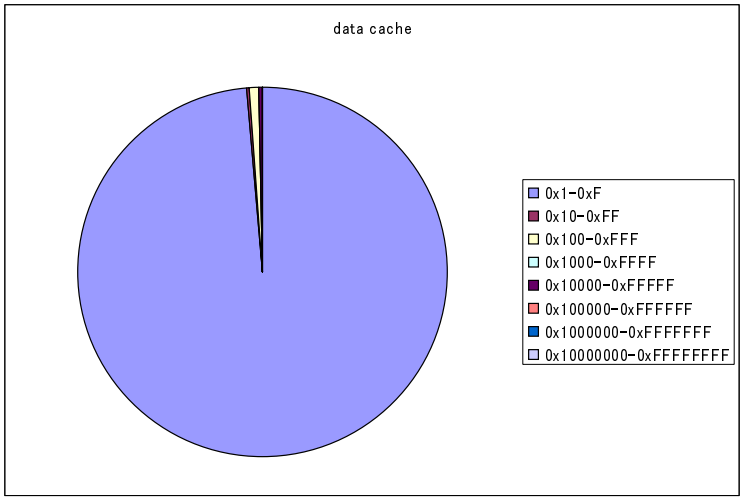


図 4.18: 64KB キャッシュでの live time の割合

4.3.3 129.compress を入力とする結果と考察

129.compress を入力とし, データキャッシュおよび命令キャッシュでの閾値を変化させた場合のキャッシュミス数をキャッシュサイズごとにグラフを図 4.19, 4.20, 4.21, 4.22, 4.23, 4.24 に示す. インデックスごとの live time の最大値の平均は表 4.6 に示す.

キャッシュサイズ	データキャッシュの平均	命令キャッシュの平均
64KB	214999	0
32KB	120424	0
16KB	27207	76721

表 4.6: インデックスごとの最大値の平均

データキャッシュでは閾値を大きく設定するにつれてミス数が減少している. 64KB,32KB ではともに閾値をインデックスごとの live time の最大値の平均を閾値にした場合が最もミス数を削減している. 16KB ではキャッシュブロック数 ×100 が最もミス数を削減している. この場合は閾値を大きく設定することでミス数を削減することができる. これは, 閾値を大きく設定することで予測 live time が大きくなり, 結果として再利用性の高いキャッシュブロックが長い間キャッシュにとどまる事でミス数が減少した.

命令キャッシュでは 64KB,32KB ともにキャッシュミス数が同じであるため評価できないが, 16KB ではキャッシュブロック数 ×100 およびインデックスごとの live time の最大値の平均を閾値にした場合が最もミス数を削減している. 命令キャッシュにおいても閾値を大きく設定することで再利用性の高いキャッシュブロックが長い間キャッシュにとどまることでミス数が削減した.

以下に最もキャッシュミス数の小さい閾値でのキャッシュヒット率および LRU 法でのキャッシュヒット率を示す.

データキャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	99.8468502	99.8552063	0.0083561
32KB	99.8118564	99.821541	0.0096846
16KB	99.7695653	99.7798336	0.0102683

命令キャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	99.9998686	99.9998686	0
32KB	99.9998686	99.9998686	0
16KB	99.9995792	99.9996211	0.0000419

表 4.7: キャッシュヒット率

データキャッシュでは 64KB の場合に最もミス率が向上する。キャッシュサイズが大きい場合に本提案手法の効果が大きい。

図 4.25 と図 4.26 では時間情報フィールドの prev live time に対してビット幅を削減した場合のキャッシュミス数, 図 4.27 に live time の割合を示す. データキャッシュでは下位ビットを 12 ビットまで削減してもさほどミス数が増えない. 上位ビットは 12 ビットまで削減してもミス数が増えない. これは live time が小さいため大きなビット幅が必要なかったためである. これにより上位, 下位ビットを 12 ビットずつ計 16 ビット削減してもさほど影響を受けずに削減できる. よって 16 ビット幅あればさほど影響がないことがわかる.

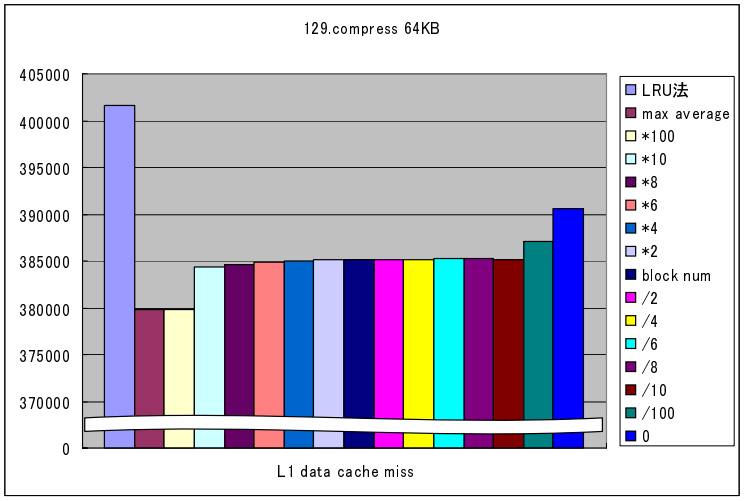


図 4.19: 閾値を変化させた場合の 64KB データキャッシュミス数

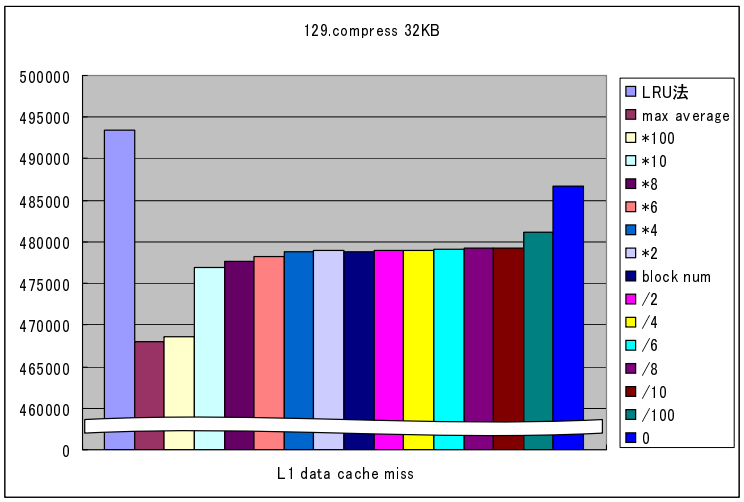


図 4.20: 閾値を変化させた場合の 32KB データキャッシュミス数

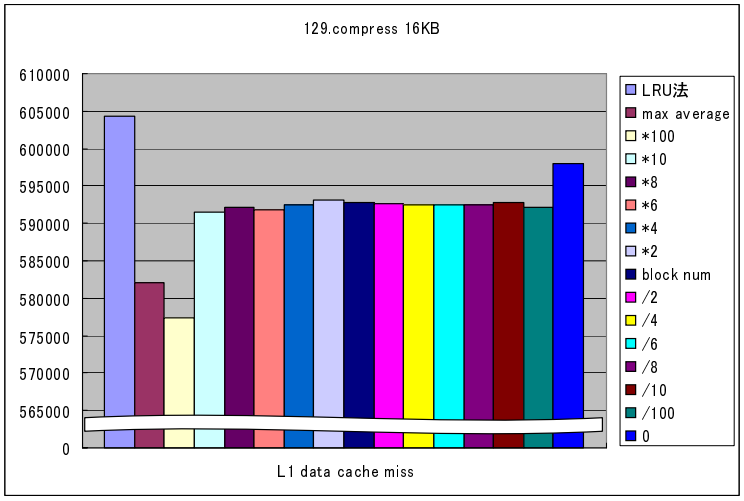


図 4.21: 閾値を変化させた場合の 16KB データキャッシュミス数

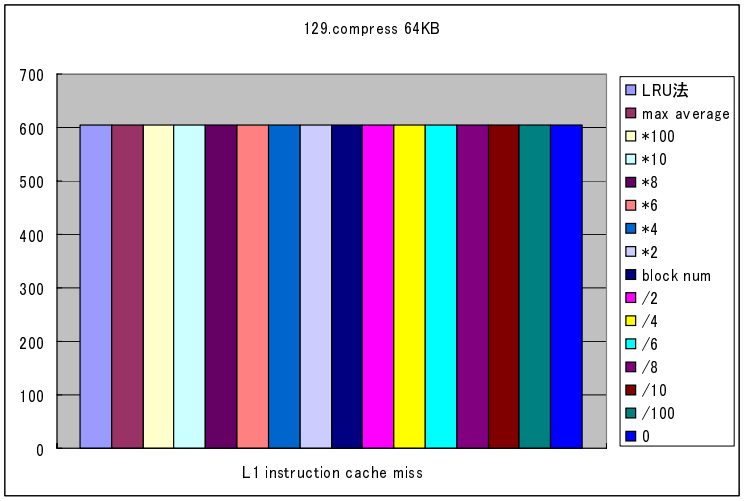


図 4.22: 閾値を変化させた場合の 64KB 命令キャッシュミス数

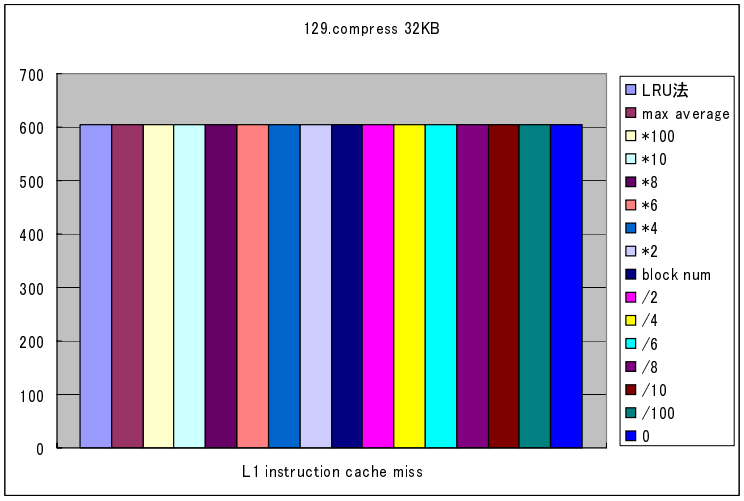


図 4.23: 閾値を変化させた場合の 32KB 命令キャッシュミス数

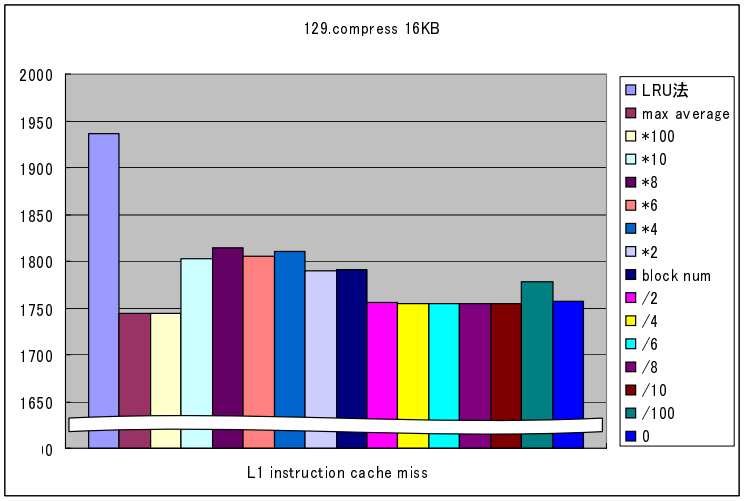


図 4.24: 閾値を変化させた場合の 16KB 命令キャッシュミス数

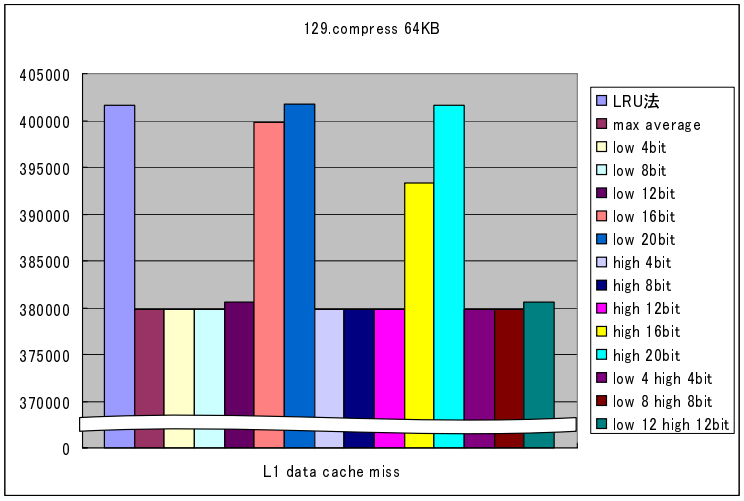


図 4.25: 時間情報フィールドを削減した場合の 64KB データキャッシュミス数

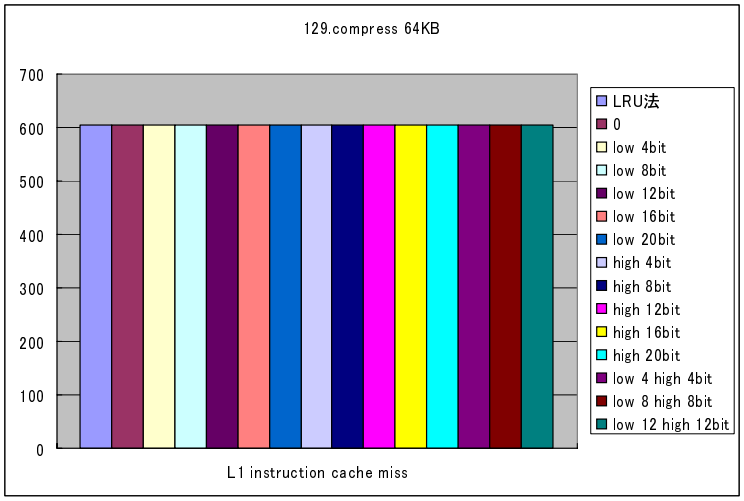


図 4.26: 時間情報フィールドを削減した場合の 64KB 命令キャッシュミス数

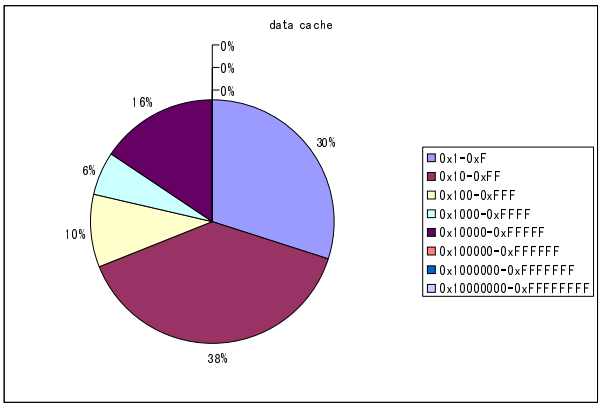


図 4.27: 64KB キャッシュでの live time の割合

4.3.4 130.li を入力とする結果と考察

130.li を入力とし, データキャッシュおよび命令キャッシュでの閾値を変化させた場合のキャッシュミス数をキャッシュサイズごとにグラフを図??, ??, ??, ??, ??, ??に示す. インデックスごとの live time の最大値の平均を表 4.8 に示す.

キャッシュサイズ	データキャッシュの平均	命令キャッシュの平均
64KB	2811231	0
32KB	326940	13790916
16KB	272697	44518477

表 4.8: インデックスごとの最大値の平均

データキャッシュではすべてのキャッシュサイズにおいて閾値を小さく設定する方がミス数が減少している. しかし, 小さくしすぎるとミスが増加している. 64KB のキャッシュの場合は閾値をブロック数 ÷100 に設定することで最もキャッシュミス数を削減している. 32KB のキャッシュの場合は閾値をブロック数 ÷16 に設定することで最もキャッシュミス数を削減している. 16KB のキャッシュの場合は閾値をブロック数 ÷6 に設定することで最もキャッシュミス数を削減している. この場合, ブロック数より小さい閾値を設定することで live time が小さくすぎることを抑えること的確な予測ができていたことによりミス数削減につながた.

命令キャッシュでは64KB のキャッシュではミス数がすべて同じであるため評価できないが, 32KB のキャッシュの場合は閾値をブロック数 ÷100 に設定することで最もキャッシュミス数を削減している. 16KB のキャッシュの場合は閾値をブロック数 ÷14 に設定することで最もキャッシュミス数を削減している. 命令キャッシュにおいてもブロック数より小さい閾値を設定することで live time が小さくすぎることを抑え, 的確な予測ができていた

ことによりミス数削減につながた.

以下に最もキャッシュミス数の小さい閾値でのキャッシュヒット率および LRU 法でのキャッシュヒット率を示す.

データキャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	97.9918867	98.132198	0.1403113
32KB	97.5139567	97.5621097	0.048153
16KB	97.1202655	97.1413707	0.0211052

命令キャッシュ

キャッシュサイズ	LRU 法	時間情報	差
64KB	99.9997074	99.999708	0.0000006
32KB	99.998947	99.9989772	0.0000302
16KB	99.9734264	99.9783004	0.004874

表 4.9: キャッシュヒット率

データキャッシュでは 64KB の場合に最もミス率が向上する。キャッシュサイズが大きい場合に本提案手法の効果が大きい。
データキャッシュでは 16KB の場合に最もミス率が向上する。キャッシュサイズが小さい場合に本提案手法の効果が大きい。

図??と図??では時間情報フィールドの prev live time に対してビット幅を削減した場合のキャッシュミス数, 図??に live time の割合を示す. データキャッシュでは下位ビットを 12 ビットまで削減してもさほどミス数が増えない. これは 16 ビットまで削減した際に, 全体の半分以上の予測 live time が同じ値になり, 予測が困難になるためである. 上位ビットは 12 ビットまでさほどミス数が増えない. これは８ビット削減することによる影響を受けるのが全体の 2%であるためである. しかし,16 ビットまで削減すると全体に占める割合が増えミス数が増加する. これにより, 上位, 下位ビットを 12 ビットずつ計 24 ビット削減してもさほど影響を受けずに削減できる. 命令キャッシュでは下位ビットを削減してもミス数に変化はない. これは 20 ビット削減することに影響する割合が小さかったためである. 上位ビットは８ビット削減するとミス数が増加する. これは live time が大きいものの割合いが大きいためである.

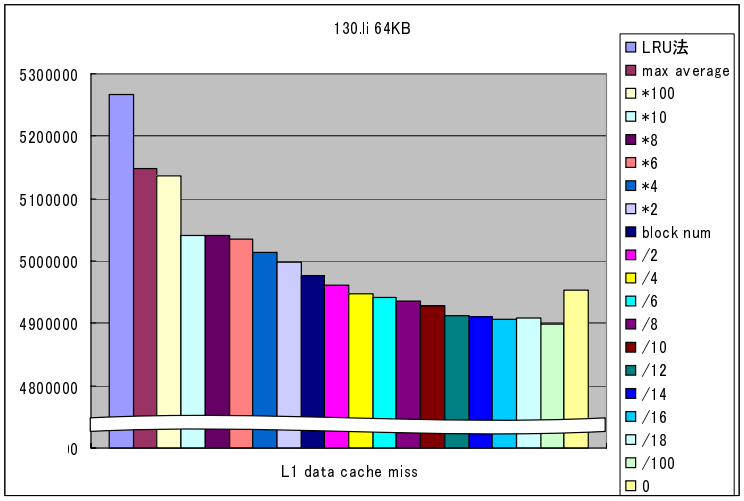


図 4.28: 閾値を変化させた場合の 64KB データキャッシュミス数

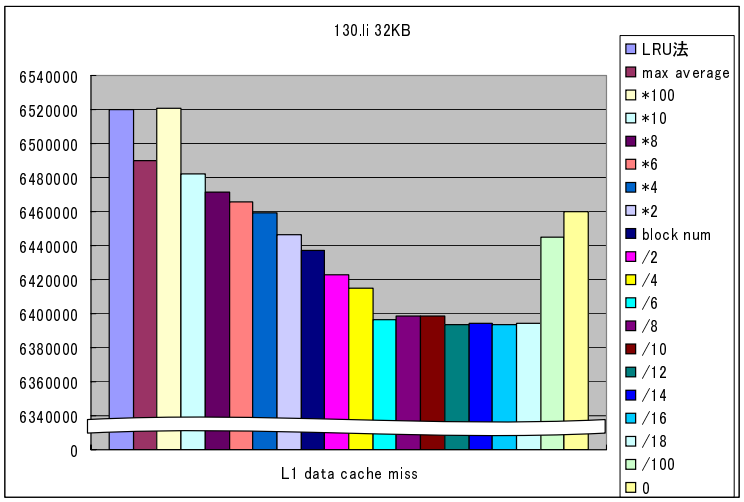


図 4.29: 閾値を変化させた場合の 32KB データキャッシュミス数

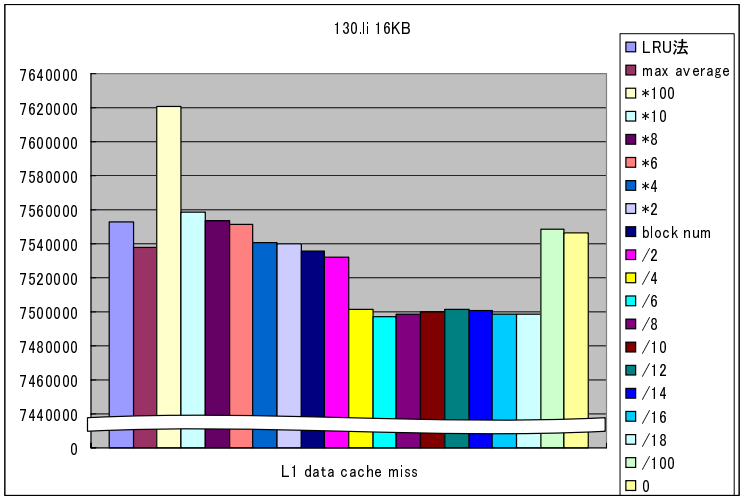


図 4.30: 閾値を変化させた場合の 16KB データキャッシュミス数

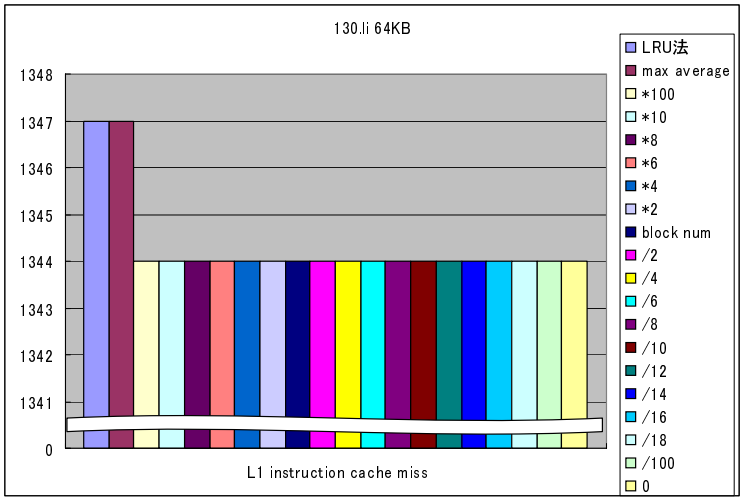


図 4.31: 閾値を変化させた場合の 64KB 命令キャッシュミス数

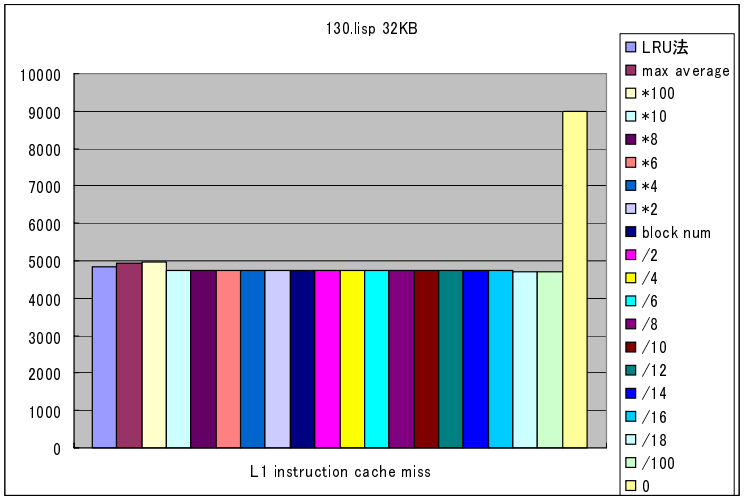


図 4.32: 閾値を変化させた場合の 32KB 命令キャッシュミス数

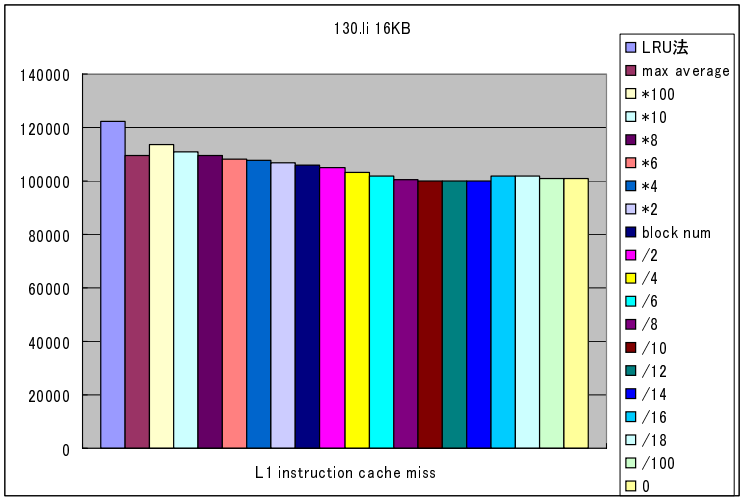


図 4.33: 閾値を変化させた場合の 16KB 命令キャッシュミス数

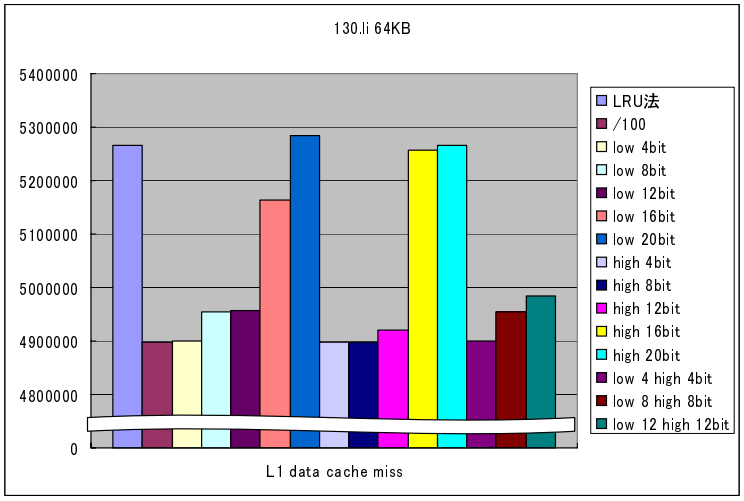


図 4.34: 時間情報フィールドを削減した場合の 64KB データキャッシュミス数

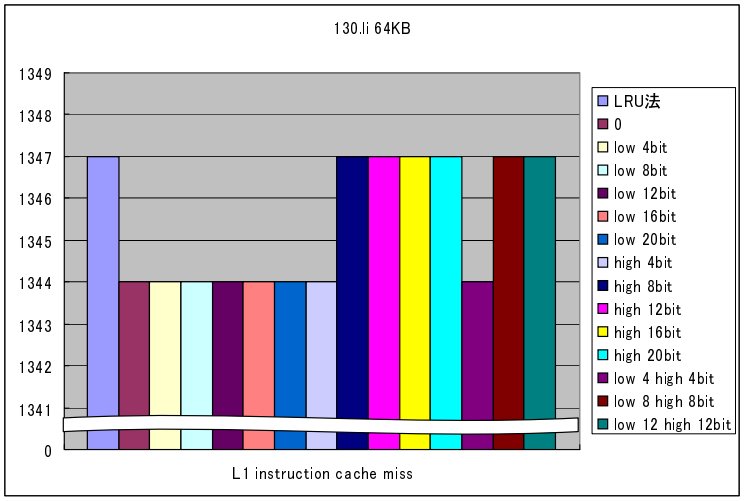
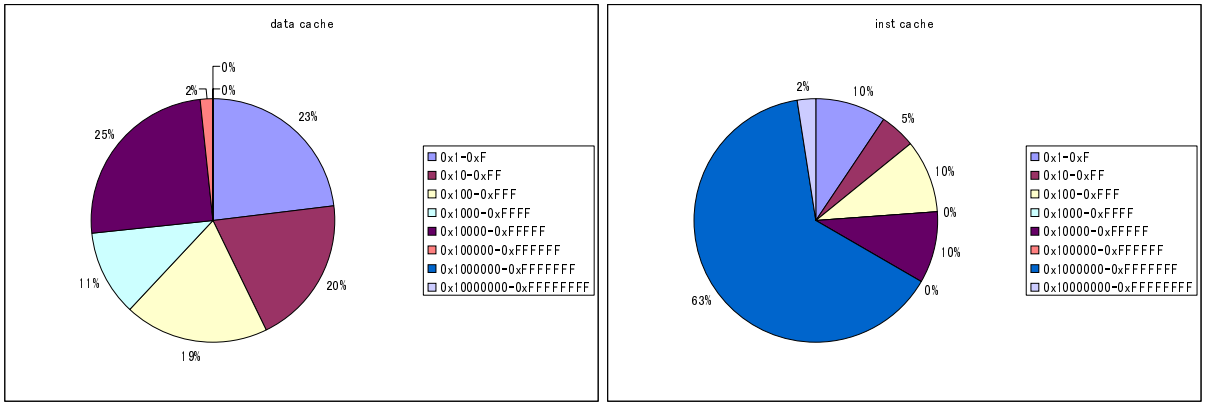


図 4.35: 時間情報フィールドを削減した場合の 64KB 命令キャッシュミス数



(a) data cache　　(b) instruction cache

図 4.36: 64KB キャッシュでの live time の割合

4.4　提案手法の考察

時間情報に基づいて置換ブロックを選択するにあたり, 最適な閾値を設定することでキャッシュミス数が削減する. 最適でない閾値を設定した場合,LRU 法に比べキャッシュヒット率が下がる場合もある. 最適な閾値は一定値ではなくアプリケーションやキャッシュサイズが違うことで変化する. 3.2.3 で述べたように, レジスタを用意しアプリケーションによって値を入れ替えることで, どんなアプリケーションに対しても閾値を変化させることができる. これにより, 最適な閾値を設定できキャッシュミス数の削減が可能である.

さらに時間情報フィールドである prev live time のハードウェア量の評価を行なった. 下位ビットを削減することで予測 live time が曖昧になる. これは予測 live time の大小に関わらず, すべてのブロックに対して影響が及ぶことであるが, 4,8,12 ビットを削減することはあまり, ミス数に影響は及ばない. 12 ビットで表される時間は live time で予測を行なうにあたり, 決め手となる精度ではない. 上位ビットを削減することは live time が小さいものばかりであるなら, 削減しても問題のないビットである. しかし, 上位ビットを削減することで大きな値が表されなくなる. 再利用性の高い live time の長いブロックでも, 一定以下の予測 live time となり, 長い間キャッシュに存在できなくなる. これらにより, あまり使用しない上位ビットと影響の少ない範囲の下位ビットを削減することで性能低下を抑えられる. prev live time の削減は性能とのトレードオフになるが, 結果から性能低下の少なかった上位, 下位を 8 ビットずつ計 16 ビット, もしくは 12 ビットずつ計 24 ビットを削減することが妥当である.

第5章　関連研究

時間情報を利用した研究として,victim cache に配置するか否かを判断する研究を２つ関連研究として示す. また, 時間情報を利用して, キャッシュの一部分への電源供給を遮断し, 消費電力削減を達成する関連研究を示す.

Timekeeping victim cache filter[3]

Timekeeping victim cache filter は, ブロックの競合により１次キャッシュからリプレースされたブロックの時間情報に基づき victim cache 内に配置するブロックを適切に選択して競合性ミスを削減する手法である.

Timekeeping victim cache filter では時間情報として dead time を使用する.dead time は, 競合性ミスの場合には短く, 容量性ミスの場合には長いという特徴がある. ブロックがキャッシュからリプレースされる際, そのブロックの dead time を基に容量性ミスか競合性ミスかを予測する. この時, 競合性ミスと判定されたブロックのみが victim cache に入るようにフィルタリングを行うことで, 再利用性の低いブロックが victim cache に配置することを防ぐ.

Reload interval victim cache filter[2]

reload interval filter は, 再利用性が高いか, 低いかを判断するために dead tiem と reload interval を使用する. reload interval とは同じブロックの使用によって生じる２つの世代の開始の間の時間である. つまり, ブロックがキャッシュにフィルされ, リプレースされた後同じブロックがキャッシュにフィルされるまでの間の時間のことである. 競合性ミスを起こす場合に dead time と reload interval は短いという傾向があり, 容量性ミスを起こす場合には dead time と reload interval は長いという傾向がある. 競合性ミスを起こすブロックは, 実際の dead time 中になっていなのにリプレースされている場合が多く, 再利用性の高いブロックであると判断することが可能である.

dead time,reload interval に基づき, 容量性ミスか, 競合性ミスであるかを判断し, 競合性ミスであると判断される場合に victim cache に配置する.

Cache decay[4]

cache decay は dead time 中になったブロックへの電源供給を遮断することで, リーク電流による電力消費を低減する. ほとんどのアプリケーションではキャッシュの中で dead time 中であるブロックが高い割合で存在する. よって,dead time 中であるブロックへの電源の遮断により低消費電力化を狙える.

cache decay ではブロックへ 10000 サイクルアクセスされなかった場合に電源供給を遮断する. つまり 10000 サイクルアクセスがない場合に dead time 中であると決めている.

第6章　おわりに

6.1　まとめ

本論文ではキャッシュブロックがdead time中になっているかの判断に前回キャッシュに存在した時のlive timeを次にロードされた場合に予測live timeとし,dead time中になったとする置換方式を提案した. さらに, 予測live timeが極端に小さくなることを防ぐために, 閾値を設定し, それより小さいlive timeだった場合にlive timeを更新せず, 前々回の予測live timeを使用する方式も提案した. 結果として時間情報に基づいて置換を行うにあたり, アプリケーションやキャッシュサイズにより適切な閾値を設定し, 置換を行うことでLRU法と比較しキャッシュミス数が減少し, キャッシュヒット率向上されることが示された. しかし, 閾値を適切な値でない場合ではかえってキャッシュミス数が増加する. さらに, アプリケーションやキャッシュサイズによって性能は変化するが, 時間情報タグを 上位ビットと下位ビットを数ビット切り取ることでハードウェア量を節約することができる. prev live timeは８ビット程度あればLRU法よりキャッシュミス数が削減できる.

6.2　今後の課題

1. 多数のアプリケーションによる評価
2. データ, 命令キャッシュ以外での置換方式
3. ハードウェア量の削減
4. マルチタスク環境での評価

１では, 今回評価に用いたベンチマークプログラムはSPECint95の４種類である. しかし, 本研究手法の有効性は, あらゆる種類のプログラムに対して検証する必要がある. 今後は,SPECint95以外の多数のベンチマークプログラムに対して効果を検証する.

２では, 時間情報に基づいて, １次データキャッシュ, １次命令キャッシュでの置換方式を検証した. 今後は２次キャッシュ, トレースキャッシュ, ビクティムキャッシュなど, さまざまなキャッシュにおいて時間情報に基づいた置換方式の検討を行いたい.

３では, 本研究では時間情報フィールドを数ビット削減する方法を論じたが, 未だコストが高い. さらなる削減をする手法を考案したい.

４では、本論文ではシングルタスクを実行した評価を行なった。マルチタスクを実行する上で本研究手法の有効性は評価していない。タスクを切り替えることで違うタスクのブロックは必要なくなるが、再利用性の高いブロックを残すことで元のタスクに切り替わったときにミス数が少なくなる可能性がある。今後は、マルチタスクを実行することによる本研究手法を検証する。

謝辞

本研究を行うにあたり御指導, 御鞭撻を頂いた北陸先端科学技術大学院大学情報科学研究科 田中清史 助教授に深く感謝するとともに, ここに御礼申し上げます.

貴重な御助言を頂ました本学の日比野 靖 教授, 井口 寧 助教授に深く感謝致します.

また, 研究を進める上で貴重な御意見を頂いた田中研究室の皆様をはじめとする多くの方々に深く感謝致します.

最後に日頃よりあたたかく見守ってくださった両親と祖父母に深く感謝致します.

参考文献

[1] David A. Wood, Mark D. Hill, R. E. Kessler. A Model for Estimaing Trace-Smple Miss Ratio. SCM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1991.

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

[3] 森田充. 時間情報を利用したキャッシュミスの削減に関する研究. 北陸先端科学技術大学院大学 情報科学研究科 修士論文, 2005

[4] Stefanos Kaxiras, Zhigang Hu, Margaret Martonosi. Cache Decay : Exploing Generational Behavior. The 29th Annual International Symposium on Computer Architecture, 2001.

[5] Standard Performance Evaluation Corporation http://www.spec.org/

[6] MIPS Technologies, Inc. MIPS64^{TM} Architecture For Programmers Volume Ⅱ: The MIPS64^{TM}Instruction Set. Document Number: MD00087 Revision 2.00 June 9, 2003

[7] Jhon L.Hennessy, David A.Patterson, 成田光彰 訳, コンピュータの構成と設計 第２版 [下] -ハードウェアのインターフェース-. 日経 BP 社. 1999.