修 士 論 文

デッドブロック予測に基づく
動的キャッシュパーティショニング

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

齋藤 好宗

2015 年 3 月

修 士 論 文

デッドブロック予測に基づく
動的キャッシュパーティショニング

指導教員　田中清史 准教授

審査委員主査　田中清史 准教授
審査委員　井口寧 教授
審査委員　金子峰雄 教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

1010026 齋藤 好宗

提出年月: 2015 年 2 月

目 次

第1章　はじめに

現在，多くのマルチコアプロセッサは，チップ内の全てのコアで共用されるキャッシュメモリを持つ．このキャッシュメモリは，コア間の競合によってプロセッサの性能を低下させる問題がある．これは，プロセッサの設計の主流が，チップ内の 1 つのコアしか持たなかったものから，複数持つようになったにも関わらず，キャッシュメモリの設計が本質的に変化していないことに起因する．この問題を防ぐ方法として，動的キャッシュパーティショニングと呼ばれる方法が研究されてきた．本研究では，従来の動的キャッシュパーティショニングの方式とは異なる視点の方式を提案し，従来の方式との差異を明らかにする．

1.1　研究背景

1.1.1　マルチコアプロセッサ

2000 年以前は，単一のスレッドを実行するシングルコアプロセッサがプロセッサの設計の主流であった．シングルコアプロセッサの設計では，命令レベル並列性[1]を利用することでスレッドの処理性能の向上を図ることが着目された．しかし，焦点はスレッドレベル並列性[2]を利用してスループット[3]の向上を図ることに移った．現在，プロセッサの設計は，複数のコアが複数のスレッドを同時に処理するマルチコアプロセッサが主流である．この方向転換の主な要因には，空冷チップの最大消費電力が改善されないことや，命令レベル並列性を利用した性能向上が困難であること，主記憶装置のアクセスレイテンシ[4]がプロセッサの性能向上に比べてあまり変化していないことなどが挙げられる [2].

半導体の集積密度の微細化が進んだことで，単一の集積回路に複数のプロセッサが搭載されたマルチコアプロセッサが製造可能となった．半導体の集積度の微細化は年月とともに進んでおり，それに応じてマルチコアプロセッサのコア数は増加し続けている．

1.1.2　キャッシュメモリ

主記憶装置のアクセスレイテンシの問題はシングルコアプロセッサが主流の時分より大きな問題であった．これを軽減するため，一般的なプロセッサは集積回路上にキャッシュ

[1]命令間に存在する並列性.
[2]スレッド間に存在する並列性.
[3]一定時間あたりの処理能力.
[4]アクセスに必要な時間.

メモリまたはキャッシュと呼ばれる小型の記憶装置を持つ．キャッシュメモリには，主記憶装置にアクセスされたデータや命令が自動的に格納され，プロセッサはキャッシュメモリにアクセスしたい命令やデータがあった場合，主記憶装置の代わりにアクセスすることで，本来必要なアクセスレイテンシを隠蔽できる．

キャッシュメモリに必要な命令やデータ入っていることをヒット，その逆をミス，ミスの際によりプロセッサから遠い記憶装置から読み込む時間のことをミスペナルティと呼ばれる．キャッシュメモリはヒット率が高い程，主記憶装置のアクセスレイテンシの問題を軽減でき，ミス率が高いほど計算機はミスペナルティに費やす時間が多くなる．それゆえ，ヒット率を上げることと，ミスペナルティを下げることが，計算機の性能を改善する上で必要になる．典型的なヒット率とミスペナルティの改善についてはそれぞれ後述する．

置き換えアルゴリズム

一般的なキャッシュメモリは，メモリ参照の空間的局所性[5]と時間的局所性[6]を利用して，ヒット率を上げる．空間的局所性を利用するため，キャッシュメモリはメモリ空間をブロックと呼ばれるまとまり単位に分割し，その単位で格納する．そして，時間的局所性を利用するため，最近利用されたブロックを優先的に保持しようとする．

キャッシュメモリの最も単純な設計では，各ブロックを配置可能な場所は 1 つであり，いくつかのブロックがその場所を巡って置き換え合う．これによって時間的局所性を上手く利用できず発生するミスを競合ミスという．競合ミスを減らす単純な方法は，ブロックを配置可能な場所を複数用意することである．この場所の数を連想度，それらの場所をまとめてセットという．連想度が高い程，ヒット率は高い傾向にあるが，アクセスレイテンシは大きくなる．それゆえ，相応の連想度でそれを効率的に利用する置き換え方針が必要となる．

理想的な置き換えアルゴリズムは，次に使用されるまでの時間が最も長いものを優先的に置き換える方式である (デッドブロックと呼ばれる二度とアクセスされないブロックがあればそれが最優先)．しかしそれらは未来の情報であり，汎用計算機のキャッシュメモリで完全に実装することは不可能である．また，実装できたとしても複雑なアルゴリズムをハードウェアで実装することは，キャッシュメモリのアクセスレイテンシを上げる．そこで，キャッシュメモリでは，メモリ参照の時間的局所性を利用し，過去や現在の使用状況に基づいて将来的に必要な命令やデータを予測する方法が使用される．一般的なキャッシュメモリでは，LRU 方式やそれに類する方式が用いられている．

マルチレベルキャッシュ

半導体の集積度の微細化が進んだことで，複数のキャッシュメモリを階層的に用意し，プロセッサに近いキャッシュメモリでミスした際に，プロセッサから遠いキャッシュメモ

[5]あるアドレスが参照されると，それに近いアドレスが間もなく参照される傾向があること．
[6]あるアドレスが参照されると，同じアドレスが近い将来再び参照される傾向があること．

リにアクセスすることで，ミスペナルティを減らすアーキテクチャが実装可能になった．このような技法をマルチレベルキャッシュと呼ぶ．本論文では，よりプロセッサに近い方を上位，その逆を下位と呼ぶ．マルチレベルキャッシュはシングルコアプロセッサが主流であった時分より存在したが，マルチコアプロセッサが主流となった現在，アクセス数が多い上位キャッシュを各コアで専有し，下位キャッシュを集積回路内の全てのコアで共有する構成が多い．この共用されるキャッシュメモリを共有キャッシュという．下位キャッシュに共有キャッシュを使用することは，コア毎に専有キャッシュを設ける場合に比べて，コア間の共有データを 1 ブロックで保持でき，かつコア間の競合が低い場合はキャッシュ全体の利用効率が高いために，資源面での効率が良い．ただし共有キャッシュは，ブロックを格納するスレッドと置き換えられるスレッドが別の場合がある．局所性の法則はスレッドの性質であるため，この特性は共有キャッシュの競合ミスが増える原因となる．

この問題はスレッド数が増える程，発生しやすい．すなわち，今後コア数が増加するとともにより大きな問題となる．これを解決するため，実行状況に応じて，置き換え対象となるブロックをコアまたはスレッド毎に制限する動的キャッシュパーティショニングが研究されてきた (なお本研究では，1 つのコアは 1 つのスレッドを実行するため，”コア”と”スレッド”を区別しない)．キャッシュパーティショニングは，IPC や公平性，QoS 管理の向上を図るためにさまざまな方式がある．キャッシュパーティショニング機能を加えることで，セットはスレッド毎に独立した連想度を持つことができる．これにより，既存の置き換えアルゴリズムを活かしながら，スレッド間の競合による性能低下を回避したり，コア毎に安定した性能を発揮したりすることができる．動的キャッシュパーティショニングによって IPC を向上させるためには，適切な実行状況を判断する方法と，ハードウェアが複雑化することによってアクセスレイテンシが上がり過ぎないことが必要である．

1.2 研究動機

1.2.1 動的キャッシュパーティショニング

動的キャッシュパーティショニングで実行状況を判断するために，さまざまな方式が提案されてきた．

Stone ら [5] は，事前実行によって連想度とミス数の関係を調べ，この結果から一定周期毎にミスが最小となるように各パーティションの連想度を決定する greedy アルゴリズムを提案した．このアルゴリズムでは，一定周期で連想度を 1 つ増加した場合にどの程度ミスが減るか，すなわちどの程度性能が向上するかを”Marginal Gain”と定義し，”Gain”が最大となるパーティションサイズを検出することで，適切なパーティションを決定する．このアルゴリズムについては，2.1.2 で詳述する．Stone らの方式は事前実行が必要であり，汎用計算機で実装することが現実的でない．そこで，Suh ら [7] は前の周期の”Marginal Gain”の近似値を LRU とヒットのパターンから計測し，それを用いて性能が増加するパーティションサイズを予測する方式を提案した．Suh らの方式については，2.2 で更に詳しく

説明するが，この方式はスレッド数や連想度，セット数に応じて非常に多くのハードウェアカウンタが必要となる．これを減らすため，Suh らはいくつかのセットでハードウェアカウンタを共用し，chunk と呼ばれる粒度でパーティションサイズを決めることで，ハードウェアコストの削減を図った．また，Suh らの方式のハードウェアコストを更に減らすため，Qureshi ら [8] は，ヒット数の計測を一部のセットに限定する Unity-Based Cache Partitioning(UCP) を，Dybdahl[9] らはモニタリングをセット内のいくつかのブロックに限定する Cache-PartitioningAware Replacement Policy(CPARP) を提案した．

これらの共有キャッシュのミス数やヒット数に着目した方式とは別に，ブロックの性質に着目して動的キャッシュパーティショニングの方式のパーティションサイズを決める方式もある．小川ら [10] は，置き換えブロックの性質に注目し，Victim キャッシュへのアクセス数やヒット数に基づいてパーティションサイズを決定する Victim Guided Cache Partitioning(VGCP) を提案した．Victim キャッシュとは，キャッシュメモリから追い出されたブロックを格納するキャッシュメモリである．VGCP で追加されるような Victim キャッシュは，1 つのセットしかなく，置き換えアルゴリズムは LRU または擬似 LRU であるため，短い期間内に追い出されたブロックが格納されている．これにより，スラッシングによる競合ミスのペナルティを減らすことができる．VGCP は，コア毎に独立した Victim キャッシュを用意し，それらのアクセス数やヒット数を利用することで，少ないハードウェアコストで実装可能である．

また，小川ら [11] は，コア毎に独立したパーティション (Private Partition:PP) とは別に，PP から溢れたブロックを格納するコア全体で共用するパーティション (Shared Partition) を設けることで，連想度の増加が必要な状況を動的に検出する History-Free Shared Cache Allocation(HFCA) を提案した．HFCA では連想度単位でキャッシュパーティショニングを行う．HFCA の詳細は 2.3 節で説明する．

このように動的キャッシュパーティショニングには，さまざまな方式がある．しかし従来の方式は，連想度を上げた場合のヒット率の増加を判断することに焦点が当てられており，連想度を減らした場合のミスの増加を判断することには焦点が当たっていない．本研究では，連想度を減らした際の性能低下が低いスレッドを判断し，動的キャッシュパーティショニングのパーティションサイズを減らすことに焦点を置いた方式を提案する．

1.2.2 デッドブロック予測

デッドブロックの情報は，置き換えアルゴリズムによる性能向上や，電力や熱量の削減に利用することができる．そのため，デッドブロックをハードウェアによって予測する方法とその情報を利用するアーキテクチャが研究されてきた．

本研究では，デッドブロックの情報の新たな利用方法として動的キャッシュパーティショニングのための指標に使用する．なお本研究では，デッドブロック予測を利用した置き換えアルゴリズムと，キャッシュパーティショニングによる性能改善が混同し，評価が困難になるのを避けるため，デッドブロック情報はキャッシュパーティショニングのみに使用

する．

デッドブロックを予測するための方法には，ブロックが再参照されるまでの時間を調べる方法や，アクセスされるブロックアドレスのパターンを調べる方法などがある．

1.3 論文の構成

本論文は全 5 章で構成される．本章では，本研究が扱う問題とその背景，そしてそれを解決するためにどのような研究が行われてきたかを詳述した．第 2 章では，その方式の中でも代表的な Suh らの方式について言及する．そして第 3 章では，本研究の提案方式について説明し，第 4 章にて第 2 章で説明した方式と本方式の比較を行う．最後に，第 5 章にて，本論文のまとめと今後の課題について論じる．

第2章　関連研究

本章では，はじめにStoneらが提案した理想的なキャッシュパーティショニングについて説明し，次にSuhらが提案した動的キャッシュパーティショニングと，小川らが提案したHFCAについて詳述する．そして本章の最後に，Suhらが提案した方式と小川らが提案したHFCAについての問題点について言及する．

2.1　Optimal Cache Partitioning

Stoneら [5] は，事前実行した情報により，一定周期毎の性能が最大となるようにパーティションを変更する動的キャッシュパーティショニングの方式を提案した．本節では，そのアルゴリズムについて説明する．

2.1.1　Marginal Gain

Stoneらの方式は，連想度を1つ上げた場合の性能改善率を”Marginal Gain”と定義する．”Marginal Gain”が最大となるようにパーティションサイズを変更することで，性能改善を図っている．”Marginal Gain”は次のような式で表される．

$$g_i(c_i) = m_i(c_i) - m_i(c_i + 1) \tag{2.1}$$

式と変数はそれぞれ次のような意味を持つ．

$g(assoc)$: 連想度を $assoc$ から1つ上げた場合の性能改善量 (ミスの削減数).

$m(assoc)$: 連想度が $assoc$ である時のミス数.

i: パーティションの識別値．本論文ではスレッド毎にパーティショニングしているため，スレッドの識別値と同義.

c_i: スレッド i が利用可能な連想度．つまりパーティションサイズ.

Stoneらの方式では，事前実行によって周期毎に連想度とミスの関係を調べることによって，$m(assoc)$ を取得する．連想度とミスの関係の例を図2.1に示す．この図はSPEC CPU 2000の4つのアプリケーションのある周期をシングルコアプロセッサでシミュレーション実行し

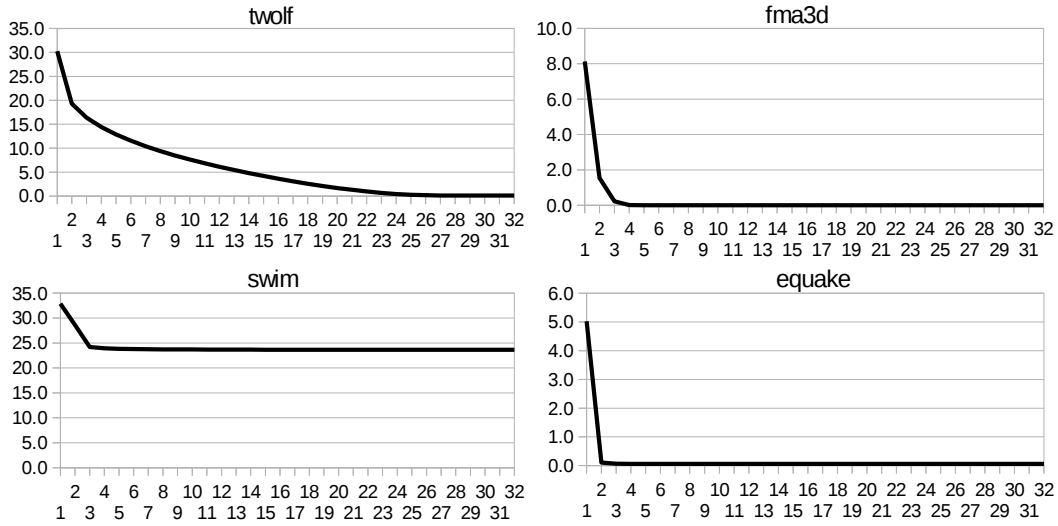

図 2.1: アプリケーションと連想度

た場合に，連想度の変化に伴って 2 次キャッシュの MPKI(Misses Per Kiro Instruction) がどれぐらい改善されるかを検証したものである．ここで X 軸は連想度，Y 軸は MPKI を意味する．彼らの方式では，このように取得した値を各スレッドの $m(assoc)$ とし，$g(assoc)$ を算出する．

2.1.2 パーティションサイズ決定アルゴリズム

Stone らの方式は，一定のアクセス単位周期でミス数が最小となるようにキャッシュパーティショニングを行う．彼らの方式では，パーティションサイズを決定するために greedy アルゴリズムが使用される．greedy アルゴリズムとは，ある問題を複数に分割して分割された部分問題毎に解を得る方法である．彼らの方式では，セット単位で”Marginal Gain” が最大となるパーティションの組み合わせを求める．以下にそのアルゴリズムの詳細を示す．

1. 全てのパーティションのサイズ (c_1 から c_N までの各値) を 0 で初期化する．
2. $g(c_k)$ が最大のスレッド k の連想度 c_k を 1 つ増やす．
3. 2 の処理を「セットの連想度 $= \sum_{i=1}^{N} c_i$」になるまで繰り返す．

2.2 Suh らの方式

Stone らは事前実行により，$g(assoc)$ の値を求めたが，汎用計算機でこの情報を利用することは現実的でない．そこで，Suh ら [7] は前の周期の”Margianl Gain”の近似値から次

	MRU lru=0	2nd lru=1	3rd lru=2	4th lru=3	5th lru=4	6th lru=5	7th lru=6	LRU lru=7
スレッド A	α	β						
スレッド B				γ				
スレッド C								

図 2.2: LRU を利用した Marginal Gain のカウンタ

の周期で性能が高くなるパーティションの組み合わせを予測する方法を提案した．

Suh らの仕組みでは，置き換えアルゴリズムに標準の LRU 方式を使用することを前提としている．説明を簡単にするため，まずはセットが 1 つのみのキャッシュ(フル・アソシアティブキャッシュ) を例に説明する．彼らの方式では，スレッド数と連想度の組み合わせと同数のハードウェアカウンタを用意する．対応するスレッドがヒットした際にその LRU 値に対応するカウンタがインクリメントされる．これらのカウンタを使用して各周期の”Marginal Gain”の近似値を求める．

例えば 3 つのスレッドで使用される連想度 8 の共有キャッシュの場合，図 2.2 で示すような 24 個のカウンタを用意する．スレッド A のアクセスがヒットした場合，アクセスされたブロックが MRU であれば α のカウンタを，lru=1 のブロックであれば β のカウンタを，アクセスさしたスレッドが B で lru=3 がであれば γ のカウンタをインクリメントする．これらのカウンタは周期の切り替わりでパーティションサイズの再計算を行う時に，次の周期のための”Marginal Gain”の値として使用される．例えばスレッド A のパーティションサイズが 0 から 1 増えた時の $g_A(0)$ は α の値を，同じようにスレッド B のパーティションサイズが 0 から 1 増えた時の $g_B(0)$ は β の値とする．パーティションサイズの計算には，Stone らの方式で使用されたアルゴリズムを改良したものが使用される．この詳細については 2.2.1 項で説明する．パーティションサイズの計算終了後，カウンタの値は重み $\delta(0 < \delta \leq 1)$ を掛けられる．これにより，2 つ以上前の周期の情報も考慮される．

Suh らは前の周期，特に 1 つ前の周期の”Marginal Gain”の近似値から次の周期で高い性能が出るようなパーティションの組み合わせを求めた．置き換えアルゴリズムが標準の LRU の場合，パーティションサイズとヒット数の関係は，LRU 値で判断できる．例えばスレッド A のパーティションサイズが 1 の時のヒット数は，lru=0 のヒット数を数えることで近似的に得ることができる．

パーティション内のブロックは，それを使用するスレッド以外のスレッドから置き換えられることはない．それゆえ，パーティション毎に独立して LRU を管理することで，そのパーティションサイズ以下で，前の周期でどの LRU 値でどれぐらいヒットしたかを正確に計測できる．しかしそうした場合，キャッシュ全体で見た時に，どのスレッドが時間的局所性の低いブロックへのアクセスが多いのか，または時間的局所性の高いブロックへのアクセスが多いのかの判断ができなくなる．Suh らの方式ではキャッシュ全体の性能を

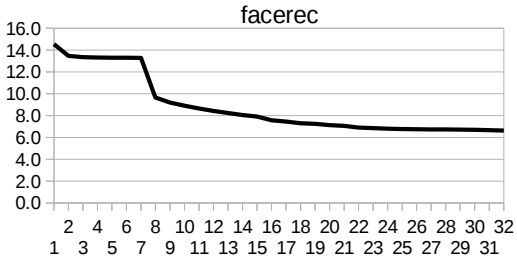

図 2.3: アプリケーションと連想度 (2)

向上することを目的としているため，LRU 情報はキャッシュ全体で管理される．

2.2.1 改良型パーティションサイズ決定アルゴリズム

図 2.1 で示したアプリケーションは，連想度を上げた際のミス数は凸型に変化する．しかし，いくつかのアプリケーションは，図 2.3 のようにある一定の範囲では連想度を上げても比較的に削減するミス量が少なく，それを過ぎた場合に多くなる．このような特性のアプリケーションに Stone らのアルゴリズムを適応した場合，パーティションを一定量割り当てて”Marginal Gain”の値が低くなった段階で，それよりも大きなパーティションでの”Marginal Gain”の情報が反映されない．それゆえ，より優れたパーティションサイズの組み合わせがあっても，それを検出できない．これによる性能低下を軽減するため，Suh らの方式では初期値に無作為な値を使用した改良型のパーティションサイズ決定のアルゴリズムが使用される．詳細は次の通りである．

C_{thread}: スレッドのパーティションサイズ．

$g_{thread}(C_{thread})$: $thread$ のパーティションを C_{thread} から C_{thread}+1 に増やした場合の”Marginal Gain”．

1. 全てのパーティションのサイズ (C_1 から C_N までの各値) を無作為な値で初期化する．
2. $g_{thread}(C_{thread})$ が最大のスレッド max と，$g_{thread}(C_{thread}-1)$ が最小のスレッド min を見つける．max と min は異なるスレッドとする．
3. $g_{max}(C_{max}) > g_{min}(C_{min}-1)$ の時，C_{min} を 1 減らし，C_{max} を 1 増やす
4. $g_{max}(C_{max}) \leq g_{min}(C_{min})$ になるまで 2 から 3 の処理を繰り返す．繰り返しの最大回数を連想度の数とする．
5. 新しいパーティションと古いパーティションの”Marginal Gain”の合計を比較し，高い方が選ばれる．

複数のセットがあるキャッシュ(セットアソシアティブキャッシュ) にこの方式を導入する場合，セット毎に連想度とスレッド数に応じたハードウェアカウンタが必要となる．このハードウェアコストを軽減するため，Suh らはいくつかのセットでカウンタを共用する方法を提案した．この方法では，例えばハードウェアカウンタを全てのセットで共用する場合，パーティションを 1 つ増加することは全てのセットの連想度を 1 つ増やすことを意味する．

2.2.2 置き換えアルゴリズムの改良

キャッシュパーティショニングは，再構成可能キャッシュ[12] によって実装可能となる．再構成可能キャッシュは，連想度やブロックサイズなどの構成を動的に変更可能なキャッシュメモリのことである．通常のキャッシュの置き換え方針は，セット内の全てのブロックから置き換え対象が選択されるが，再構成可能キャッシュを利用することで，サブセットを設定し，置き換え対象の範囲をそれに属するブロックに制限することができる．

Suh らの方式では，基本的に置き換え対象とするサブセット (パーティション) を選択し，その中の LRU ブロックが選ばれる．選択されるパーティションは，次のようなルールで決まる．

- ミスが発生したスレッドの使用しているブロック数が割り当てられたパーティションサイズよりも小さい場合:
  ”使用しているブロック数-パーティションサイズ”の値が最大のスレッドのパーティションの LRU ブロックを選択 (条件にあうスレッドが複数ある場合はその中から無作為に選ばれる).

- ミスが発生したスレッドの使用しているブロック数が割り当てられたパーティションサイズよりも大きいか等しい場合:
  自身が使用するブロック郡の中の LRU ブロックを選択．ただし使用しているブロック数が 0 の場合は，以下の処理を行う．

  使用しているブロック数が 0 の場合:
  セット全体の LRU ブロックを選択.

2.3 HFCA

小川ら [11] は，HFCA(History Free Cache Allocation) と呼ばれるヒット数やミス数以外の情報を利用した動的キャッシュパーティショニングの方式を提案した．HFCA は，パーティションの増加が必要なスレッドを検出し，その都度動的に連想度単位でのパーティショニングを行う．HFCA の最大の特長はハードウェアカウンタを必要としなことである．HFCA では，スレッド毎に専有するパーティション (PP) とは別に全てのスレッドで

共用する共有パーティション (SP) を設け，それを使ってパーティションの増加が必要なスレッドを検出する．

HFCA ではスレッド毎に LRU 情報を管理する．そして，PP にはより最近アクセスされたブロックを優先的に格納し，入りきらなかったブロックを SP に格納する．共有キャッシュにアクセスがあった場合，それがミスならば，SP を最も多く使用しているスレッドの LRU ブロックを追い出し，そこに格納する (SP を最多利用するスレッドが複数ある場合はそれらから無作為に選ばれる)．ヒットの場合，そのブロックを使用するスレッドとそれが格納されたパーティションが異なる場合，PP が小さいとみなし，パーティションサイズを 1 つ増やし，その分ヒットしたパーティションを減らす．なおヒットしたブロックの所有者とアクセスしたスレッドが異なる場合，アクセスしたスレッドが所有する LRU のブロックがヒットしたと仮定して処理を行う．図 2.4 は連想度 12 のセットを HFCA で 4 つのスレッドで分配する場合の例である．ここで，色付きのセルはその色に対応するスレッドの PP，色なしのセルは SP を，列はそのスレッドが所有するブロックであることを意味する．この図では，スレッド 1 でミスがあった場合と，その後にスレッド 0 でヒットがあった場合の例を示している．

HFCA ではパーティションサイズが増加する程，SP が小さくなり，最終的に SP は 0 になってパーティションの更なる増加が不可能になる．これを回避するため HFCA ではリセット機能によって，PP を減らし SP を増やす．リセット機能は，一定周期毎に PP のサイズを半分にし (切り上げ)，その分を SP に加える．

HFCA は SP が大きいほどパーティションの増加の必要性検出しやすい傾向があるが，その反面スレッド間の競合によるミスを防ぐことが難しくなる．

2.4 問題点

キャッシュパーティショニングでは，パーティションサイズを減らしたスレッドの性能低下の影響が小さく，パーティションサイズを増やしたスレッドの性能向上の影響が大きくなければならない．そうでなければ，パーティションサイズを減らしたスレッドの影響で，全体の性能が低下する．しかし従来のキャッシュパーティショニングは，パーティションサイズを増やした場合の性能が改善が着目されており，減らしたパーティションによる性能低下については，特に重視されていない．

HFCA では，PP を減らす方法は重視されておらず，リセット機能によって PP が減らされる．しかしリセット機能は周期によって自動で行われるため，それがもたらす性能低下の影響は予測できない．

Suh らの方式では，ヒット数が少なかったスレッドのパーティションサイズが減らされる傾向がある．しかし，そのスレッドのミス情報は考慮されておらず，ミスの多いスレッドが更に性能低下することで，結果としてプロセッサ全体の性能が低下することがある．また，あるスレッドに偏った最適化が行われてプロセッサ全体の性能が上がった場合，他のコアの性能を下げることで，公平性を低下させる．このような動的キャッシュパーティ

ショニングは，コア数の増加に伴って性能低下するスレッドの数が増える傾向があるため，スケーラビリティが低い．

本研究では，従来の方式とは逆の視点として，パーティションサイズを減らしても性能低下が小さいスレッドを検出し，その分を他のスレッドのパーティションに割り当てる方式を提案する．これにより，従来の方式よりもコア毎に安定した性能改善を図る．

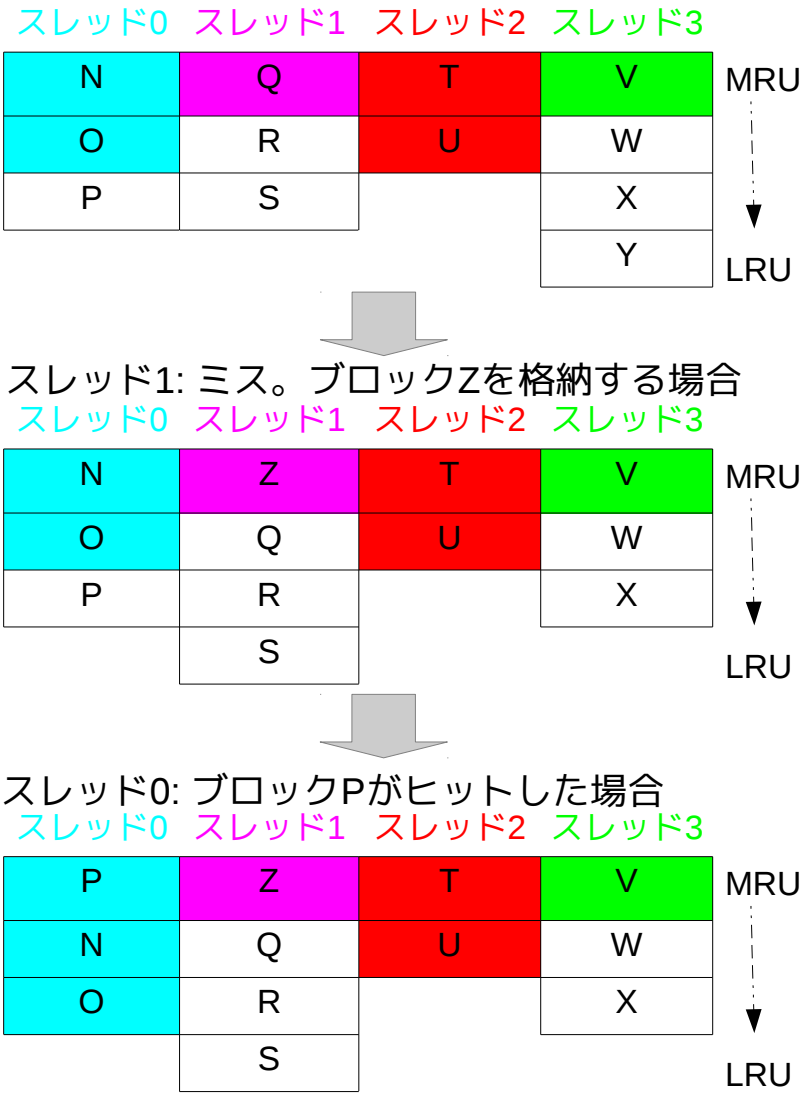
スレッド0 スレッド1 スレッド2 スレッド3
N Q T V
O R U W
P S X
Y
MRU
LRU
スレッド1: ミス。ブロックZを格納する場合
スレッド0 スレッド1 スレッド2 スレッド3
N Z T V
O Q U W
P R X
S
MRU
LRU
スレッド0: ブロックPがヒットした場合
スレッド0 スレッド1 スレッド2 スレッド3
P Z T V
N Q U W
O R X
S
MRU
LRU

図 2.4: HFCA

第3章　提案手法

本研究では，パーティションのサイズを減らしても性能への影響が少ない状況を検出し，その分他のパーティションを増やすことで性能改善を図る．これによって，従来の動的キャッシュパーティショニングよりもコア毎の公平性を改善する．本提案方式では，パーティションのサイズを減らすための情報として，デッドブロック予測の情報を使用する．

3.1　はじめに

デッドブロック情報を利用しようとする場合，デッドブロックと呼ばれる言葉は2種類の意味で使用される．1つはスレッドのメモリ空間全体の最後のアクセスが終わったブロックで，もう1つはあるメモリ階層で最後にアクセスされてから追い出されるまでの物理的なブロックである．後者の情報を知ることで，優先的にブロックを置き換えてキャッシュの性能を上げることや，そのブロックの電気をカットすることで電力の削減ができる．そのため，以前からデッドブロックの予測方法と，その情報の利用方法が研究されてきた．本研究では，その情報を動的キャッシュパーティショニングに使用し，パーティションサイズを減少した際の性能低下が少ないスレッドを予測する．

3.2　仕組み

本提案方式では，セット毎に連想度単位でのキャッシュパーティショニングを行い，デッドブロックを検出したパーティションを過分なサイズであると判断した時にパーティションサイズを減少する．ただし，あるパーティションに1つのデッドブロックを見つけただけで，そのスレッドのパーティションサイズを減らそうとするのは，判断理由としては弱い．パーティションサイズの減少は性能低下を引き起こすことに繋がるので，慎重な判断が必要である．それゆえ，本提案方式では，セット内でデッドブロックが複数ある場合にパーティションサイズを減少する．これにより，より利用状況が少ないことを判断でき，かつデッドブロックを検出するタイミングによって偶発的にデッドブロックが見つかるような状況を削減する．

この方法でシミュレーション検証を行った際，デッドブロックの検知とそれに伴うキャッシュパーティショニングをデッドブロックの発生に近いタイミングで行うものほど，デッドブロックを追い出しやすかった．この状況に近づけるため，本提案方式では，ミスが

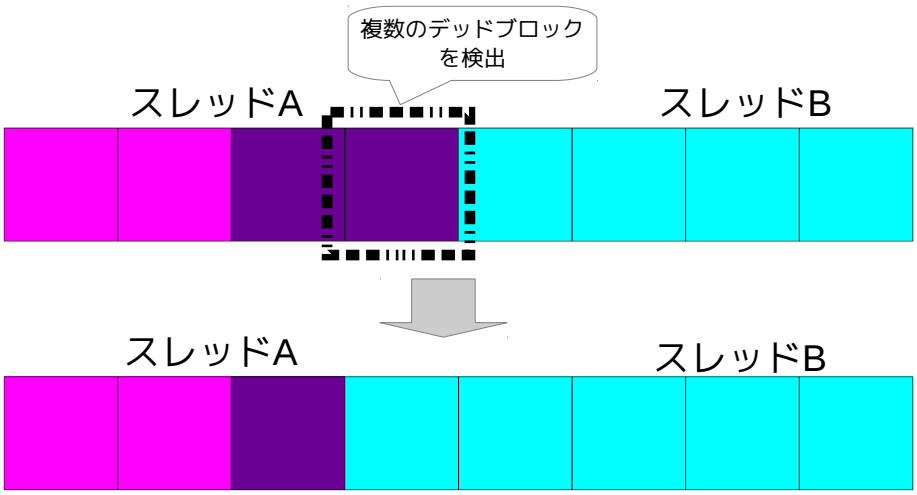

図 3.1: 本提案方式の振る舞い

発生した際，置き換えアルゴリズムによって置き換えブロックを決定する前にキャッシュパーティショニングを行う．詳細は後述するが，本提案方式ではデッドブロックを判断するための時間にキャッシュに対するアクセス回数を利用しているため，ヒットとミス両方でキャッシュパーティショニングを行う場合とミスの時だけ行う場合では，異なるパーティショニングになるが，ミス発生の時にのみキャッシュパーティショニングを行うことで，キャッシュパーティショニングによるレイテンシをミスペナルティで隠蔽する狙いがある．また，固定周期を使用しないことで，それを計測するためのハードウェアを排除できる．なお置き換えアルゴリズムにはSuhらの方式と同様の方式を使用する (ただし本方式では，後述する仕組み上最低パーティションサイズは1で，それ以下にはならない)．

本提案方式では，2章で紹介した従来の方式とは異なり，ヒットやミスに直接的に関係なくアクセスパターンでパーティションサイズが決まる．図3.1で示す通り，本提案方式では，スレッドのパーティションの利用頻度が低い場合や一部のブロックのみが頻繁に利用されている場合に，パーティションサイズが減少する．減少する数 s については本評価では事前のテストで平均的に性能が高かった”検出したデッドブロック数-1”を使用する．減らした連想度は，デッドブロックを1つも待たないパーティションに分配する．本研究では，これを割り当てたパーティションによる性能向上が評価に大きく影響しないように，パーティション増加の影響が大きいスレッドを検出するような仕組みは持たず，デッドブロックを1つも持たないパーティションで等分に配分する (均等に分配できない場合はパーティションサイズが小さいパーティションを優先する)．

3.3 デッドブロック予測

本研究ではキャッシュに格納されてから，追い出される前の最後のアクセスまでをライブタイムとし，それ以降をデッドとする．すなわち理想的なデッドブロック予測器の振る舞いは，ブロックが追い出される前の最後のアクセスが終了次第，デッドと判断することである．本提案方式のキャッシュパーティショニングは，デッドブロックの検出がライブタイムの終了に近い時期であるほどスレッドの実行状況の判断に適し，かつキャッシュ全体の利用効率の改善にも繋がるため，高い性能を発揮する．

本提案方式では，ブロックへのアクセス毎にブロックに時間情報を更新し，これを利用して，一定時間アクセスのなかったブロックをデッドと予測する．時間は共有キャッシュへのアクセス回数で表し，スレッド毎に独立してアクセス回数を計測する．これはデッドブロックの特性がアプリケーションの影響によるところが大きいためであり，他のスレッドの影響を受けることを防ぐためである．本研究では，”$(last\_access_{block} + d) < current\_time_{thread}$” の条件を満たしたものをデッドとする．各変数が示す意味は，以下の通りである．

d: デッドブロックと判断するための最後のアクセスからの時間 (スレッド毎の共有キャッシュへのアクセス回数).

$current\_time_{thread}$: スレッドごとに計測される時間．本研究ではスレッドごとのアクセス回数.

$last\_access_{block}$: ヒットしたブロックへの最後のアクセス時間.

デッドブロックを判断するためのしきい値 (d) は，低いほど検出数は増え，デッドになった瞬間に近づくが，ライブブロックをデッドブロックとみなす判断ミスが増える．本提案方式では，パーティションの一部の時間的局所性の高いブロックのみが使用されるスレッドならば，しきい値を下げた方が他のスレッドの性能向上する機会が増える．例えば

しかし再利用性が高いが時間的局所性の低いブロックが多いスレッドは，パーティションサイズの減少に伴って性能が低下する．それゆえ，スレッド毎の利用状況を判断し，適切なしきい値を使用することが望ましい．本研究では，デッドと判断するためのしきい値をヒット時に動的に更新する．新しいしきい値は，式 3.1 で示す通り，最後にアクセスされてから再アクセスされるまでの時間の長さに重み (δ) をかけ，それとしきい値を足したものを半分にした値を使用する．

$$d_{new} = (d_{old} + (current\_time_{thread} - last\_access_{block}) * \delta))/2 \tag{3.1}$$

3.4 リセット機構

本方式は，パーティションのサイズを減らしても性能への影響が小さいスレッドを検出する方式であり，減らした分を割り当てたパーティションのスレッドがどれぐらい性能改

善されるかは予測できない．また，頻繁にミスを発生するスレッドがある時，ミスの原因が競合ミスによるものなのか，それ以外なのかの判断ができず，パーティションサイズを減らすことができない．

従来の方式では，ヒット数に着目しているため，ミス数が多いパーティションは，パーティションサイズが減らされる可能性がある．そのミスの原因が競合ミスであれば，パーティションを増やすことで，それを利用するスレッドは性能が改善されるが，それ以外の原因の場合，パーティションを増やすことはキャッシュの利用効率を低下させ，プロセッサ全体の性能低下を招くことがある．

本提案方式では，そのようなパーティションも増加対象に選ばれるが，増加しても競合ミスが多い場合，そのままそのスレッドが資源を専有する．しかしそのようなスレッドがパーティションサイズを減らしても良いものなのかの判断がつかず，それによって性能が低下することがある．この問題を軽減するため，本方式では一定時間ミスが発生してもキャッシュパーティショニングが行われなかったセットのパーティションサイズを均一分割にリセットする．

キャッシュパーティショニングが行われなかった理由が無駄のない配分によるものだった場合，リセット機能は性能低下を引き起こす．しかし本提案方式によってパーティションサイズが増えたスレッドは，パーティションサイズの増加の影響が大きいものを検出している訳ではないため，そうなることは運次第である．本提案方式では，そうなることを期待するよりもあるパーティションが無駄に資源を利用することを防ぐことを優先する．

第4章　評価

本研究の評価では，SPEC CPU 2000 ベンチマーク集 [3] の各アプリケーションから，メモリアクセスをトレースし，そのデータを使用してキャッシュシミュレーションを行う．SPEC CPU 2000 の詳細については 4.1 節にて説明する．コンピュータの性能を比較する場合，ユーザがいくつかのコンピュータに対して同じ負荷 (ワークロード) を与えることで，それらを比べることができるが，全く同じ負荷を与えるのは困難である [1]．ベンチマークとは，コンピュータの性能を比較するためにユーザが与えるであろう負荷を再現するプログラムのことである．なお負荷 (ワークロード) とは，コンピュータ上で実行される一連のプログラムだけでなく，実際のプログラムの構成に似せて作成したもののことも指す．本研究の評価では，ワークロードとして，計算機シミュレータ Simple Scalar[4] によって SPEC CPU 2000 のいくつかのアプリケーションから生成したトレースファイルをコア数分だけ組み合わせたものを使用する．プログラムの組み合わせについては 4.3 節で詳述する．

本章では，4.1 節でワークロードとして使用する SPEC CPU 2000 の説明を行い，4.2 節で評価項目,，4.3 節で評価に使用するワークロード，4.2 節で評価環境については詳述する．そして，4.5 項で本提案方式と従来の方式との性能評価を行う．

4.1　SPEC CPU 2000

SPEC CPU 2000[3] とは，非営利団体 SPEC(Standard Performance Evaluation Corporation[1]) が提供するベンチマーク集である．SPEC は，現代のコンピュータシステム用の標準ベンチマーク集を開発することを目的に活動する組織であり，多数のコンピュータベンダからの出資によって維持される．SPEC CPU の各プログラムは，整数用 (CINT) と浮動小数点用 (CFP) の 2 種類に分類される．SPEC CPU 2000 の構成は，それぞれ整数用の CINT2000 が C や C++によって記述された 12 本のプログラム，浮動小数点陽の CFP2000 が C や Forctran-77，Fortran-90 によって記述された 14 本のプログラムである．各プログラムの概要を，表 4.1 に示す [1].

SPEC CPU 2000 では，各アプリケーションの入力データに”ref”と”test”，”train”の 3 種類が用意されており，本評価では，”ref”を使用する．

[1] `https://www.spec.org/`

4.2 評価指標

本研究では，プロセッサ全体の性能 (IPC)，コア数の増加に伴う性能改善の効果 (Weighted Speedup) を評価する．

1 章で述べた通り，共有キャッシュは上位キャッシュのヒット率にも影響を及ぼす．それにより，マルチコアプロセッサでは各スレッドのヒット状況によって進む速度に差が生まれ，それによって，共有キャッシュへアクセスされる順番や，一定時間あたりに処理する命令数も変化する．それゆえ，キャッシュパーティショニングによる性能改善の効果は，ヒット率で判断することは困難である．そこで，性能改善の指標として，マルチコアプロセッサ全体の IPC(Instruction Per Cycle) を評価する．また，マルチコアプロセッサでは，IPC は式 4.1 のように各コアの IPC の総和であるため，一部のコアに偏った最適化を行うことで，IPC が向上することがある．このような振る舞いは，公平性を下げ，コア数の増加に伴う性能改善の効果を低下させる傾向がある．そこで，本評価では，IPC に公平性を考慮した評価方法である Weighted Speedup を使用する．Weighted Speedup は，式 4.2 のようにして計算される．SIngle IPC は，コア数以外の構成をそのままに，アプリケーション単体を実行した場合の性能である．すなわちコア間の競合の影響を受けない場合の性能を意味する．

$$IPC_{sum} = \sum(IPC_i) \tag{4.1}$$

$$WeightedSpeedup = \sum(IPC_i/SingleIPC_i) \tag{4.2}$$

4.3 ワークロード

本研究では，SPEC CPU 2000 のアプリケーションをそれぞれのスレッドで 1 つ実行する．各アプリケーションは以下の 5 つのグループに分け，それらの組み合わせを 1 つのワークロードとして実行する．グループは各アプリケーションをシングルコアプロセッサで，連想度を変えてシミュレーション実行した場合の MPKI(Misses Per Kiro Instructions) によって判断する．

A. 連想度を増加する程 MPKI が低下するアプリケーション (図 4.1).

B. 連想度 1 の時の MPKI が 1 未満のアプリケーション (図 4.2).

C. 連想度 1 の時の MPKI が 1 以上で小量の連想度を追加することで，MPKI が 1 未満に収束するアプリケーション (図 4.3).

D. 小量の連想度で MPKI が 1 より大きな値で収束するアプリケーション (図 4.4).

E. 連想度に伴う MPKI の改善の収束箇所が複数あるアプリケーション (図 4.5).

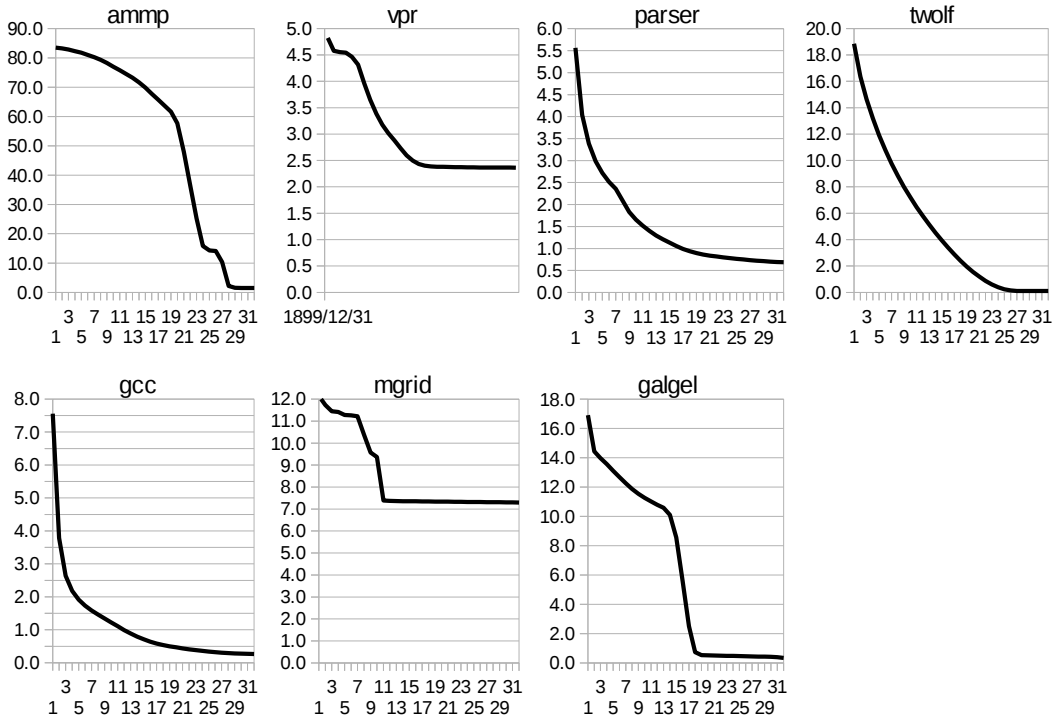

図 4.1: 連想度を増加する程 MPKI が低下するアプリケーション (グループ A)

各アルファベットは，以降でグループを指す場合の識別文字として使用する．各アプリケーションが属するグループと，連想度の変更に伴う MPKI は，各図に示す通りである．

実行するワークロードを表 4.2 に示す．

4.4 評価環境

本研究では，Simple Scalar を利用して取得したトレースファイルを使用し，オリジナルのキャッシュシミュレータによって評価を行う．Simple Scalar[4] とは，プログラムの性能解析や試作のマイクロアーキテクチャの設計，ハードウェア・ソフトウェア強調検証などを行うためのツールセットである．Simple Scalar のシミュレータは，Alpha や PISA，ARM，x86 の ISA のエミュレートを行うことができる．本評価では，SPEC CPU 2000 のアプリケーションを Alpha でコンパイルしたものから，トレースファイルを生成する．なお，トレースファイルを生成するシミュレータは最大同時命令発行数が 1 のインオーダ実行のプロセッサモデルを使用する．また，本評価では共有データはないものとする．

本研究で評価に使用するキャッシュシミュレータのパラメータは表 4.3 の通りである．本

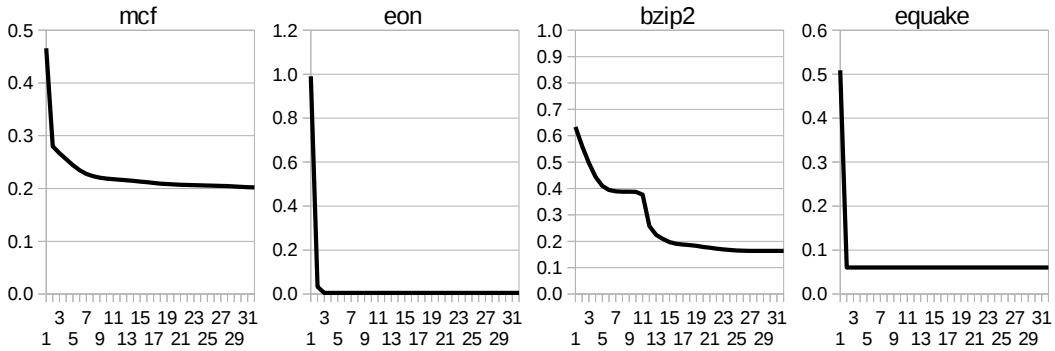

図 4.2: 連想度 1 の時の MPKI が 1 未満のアプリケーション (グループ B)

提案方式のしきい値の計算に使用する重みは 64 を使用する．また，あるセットでミスが発生してもキャッシュパーティショニングが行われないようなことが 8 回続いた場合にリセットを行う．各値はワークロードによってはより性能の高い値があるが，全体的に性能が高かったものを使用している．

本研究では，本提案方式と次の 3 つの方式を比較する．

- EQ(均一分割)
- HFCA
- Suh らの方式

動的キャッシュパーティショニングの各方式の詳細は 2 章で説明した通りである．それぞれの方式のパラメータは，事前実行により性能が平均的に高かったものを使用する．HFCA のリセット値は 500,000 回アクセスとする．Suh らの方式については，本提案方式と比較対象である HFCA に合わせて，連想度単位のキャッシュパーティショニング，すなわちセットごとにパーティションサイズを決定する．カウンタセットの周期は，セットごとに 200 回のアクセスとし，パーティションサイズ計算終了後にカウンタに掛けられる重さ (δ) は 0.5 とする．なお比較対象の 3 つの方式のプロセッサの構成については，本提案方式と同じである．

4.5 評価結果

本節では，はじめに本提案方式と従来の方式である Suh らの方式の IPC を比較し，特徴を検証する．そして 4.5.2 項では，更に均一分割と小川らの方式である HFCA を加えて IPC を比較する．最後に，4.5.3 項で Weighted Speedup の値を比較する．なお本評価では，デッドブロック予測の影響で上位キャッシュの性能が低下することで，評価が困難に

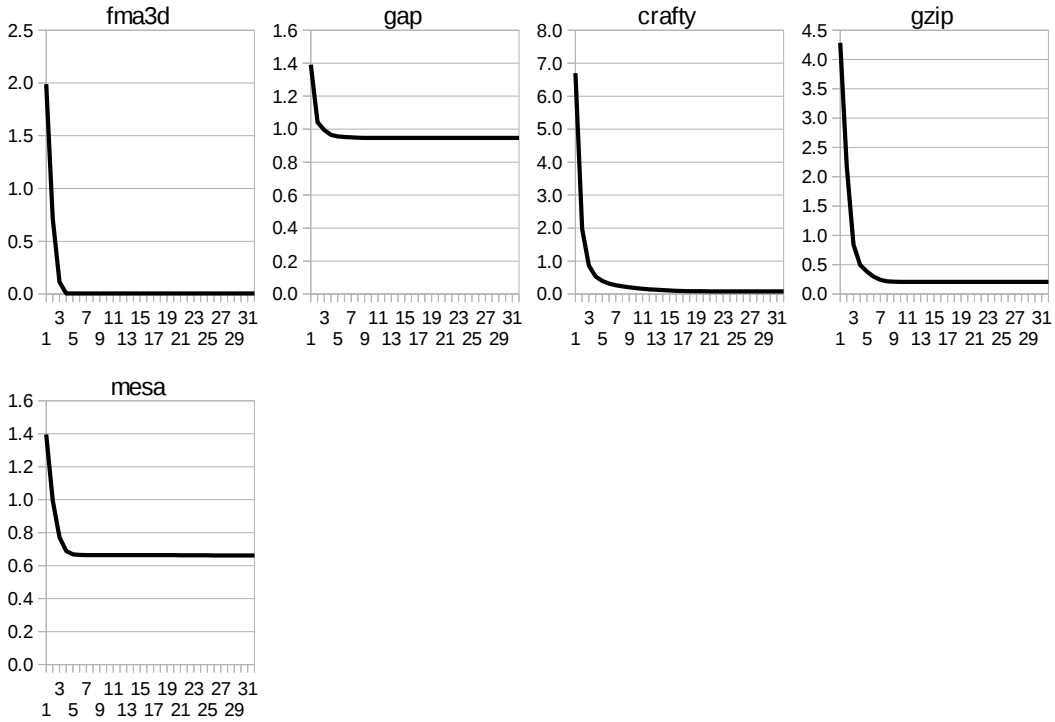

図 4.3: 連想度 1 の時の MPKI が 1 以上で小量の連想度を追加することで，MPKI が 1 未満になるアプリケーション (グループ C)

なることを防ぐため，Exclusion Property を採用する．補足として Inclusion Property を採用した場合の各方式の IPC を付録 A に示す.

4.5.1 特性

各方式の性能を比較した場合の結果は，図 4.6 の通りである．この図の X 軸は通常の LRU 方式の IPC を 1 とした場合の性能比である．評価結果は，Suh らの方式が全体的に提案方式よりも高い傾向を示し，平均で本方式より約 3.0%程性能が高かった．特にワークロード 05 やワークロード 07 のようにグループ A を多く含む組み合わせの時に性能が高い傾向があった．ただし，ワークロード 01 やワークロード 12 のようにグループ D やグループ E を含むような場合は，本提案方式は Suh らの方式よりも高い性能を示した.

Suh らの方式がパーティションサイズの増加が必要なスレッドを予測するのに対し，本提案方式ではパーティションサイズの減少が必要なスレッドを予測する．それゆえ，グループ D のようにミス数が多いスレッドやグループ E のようにパーティションサイズの増加の判断が困難なスレッドで高い性能を発揮したと推察される．また，本研究では，パーティションサイズを増加するスレッドに着目していないため，性能を大きく改善できるグループ A において，Suh らの方式に比べて非常に低い性能になった.

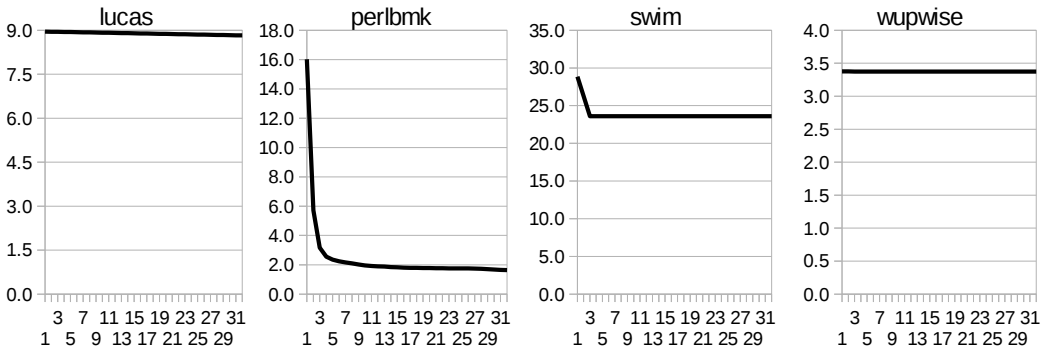

図 4.4: 小量の連想度で MPKI が 1 より大きな値で収束するアプリケーション (グループ D)

4.5.2 IPC の評価結果

本項では前項で評価した方式に加えて EQ や HFCA などの方式の評価結果を加えることでより汎用的なデータを示す．評価結果を図 4.7 に，そのデータの平均を表 4.4 に示す．図は前項で使用した図と同様 LRU の IPC を 1 とした時の比率である．使用するワークロードの詳細は 4.3 節で説明した通りである．

平均的に最も高い性能を発揮したのは，Suh らの方式であった．Suh らの方式は，表 4.4 で示す通り，平均的にも IPC が高く，最も改善された組み合わせでは LRU から 37.9%の性能が改善された．

本提案方式は，平均では HFCA や均一分割より性能が高かったが，ワークロード 06 では EQ を下回る結果となった．従来の方式とは異なり，パーティションサイズを性能低下しても影響が少ないパーティションを減らしているため，これは避けたい問題だった．

4.5.3 Weighted Speedup の評価結果

各方式の Weighted Speedup の評価結果は図 4.8 の通りである．本提案方式は，Suh らの方式の IPC と比較して約 1.5%，HFCA に比べて約 0.1%低い結果となった．それぞれの平均は表 4.5 の通りである．

4.5.4 評価結果の考察

全体の IPC を比較すると，本提案方式は HFCA に比べて安定した性能だったが，Suh らの方式に比べると性能が低く，一部の組み合わせの時だけ性能が高いような結果となった．また，均一分割より性能が低下するようなワークロードがあった．本提案方式はパー

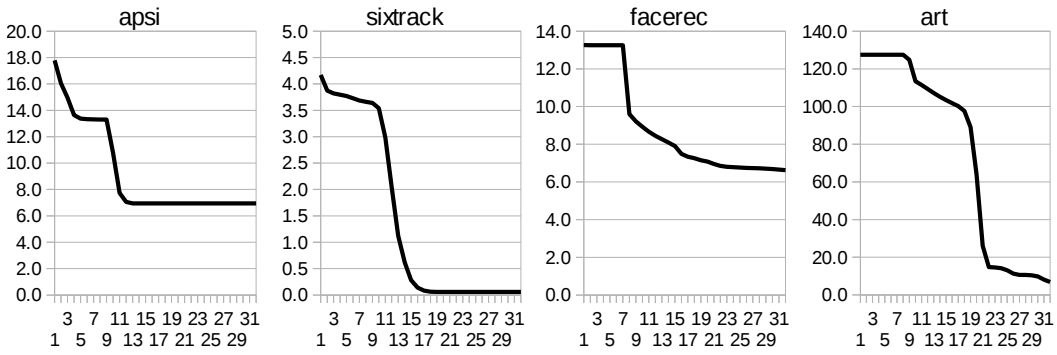

図 4.5: 連想度に伴う MPKI の改善の収束箇所が複数あるアプリケーション (グループ E)

ティションサイズを小さくしても性能に影響しないコアを予測しているため，このような結果は，本提案方式の意図とは外れる結果である．この問題の原因には，次のことが考えられる．

- デッドブロック予測の方法.
- パーティションサイズは十分な量と判断したが，時間的局所性の低いブロックが多かった.

本提案方式で使用される動的なデッドブロックのしきい値の算出は，固定値に比べて平均的に高い性能が発揮されたが，それでも改善の余地は見られた．図 4.9 や図 4.10，表 4.6 で示すように，2 次キャッシュのアクセス間隔は，キャッシュ全体で計測してもアプリケーションによってばらつきが大きい．その上，キャッシュパーティショニングを行う場合，パーティションサイズの変化に伴い，更に変化する．この傾向はグループ A や E で特に顕著に見られた．また，しきい値の計算に使用する重みについても，事前の評価で平均的に高い性能となるものを選択したが，本提案方式が均一分割よりも低い結果となった

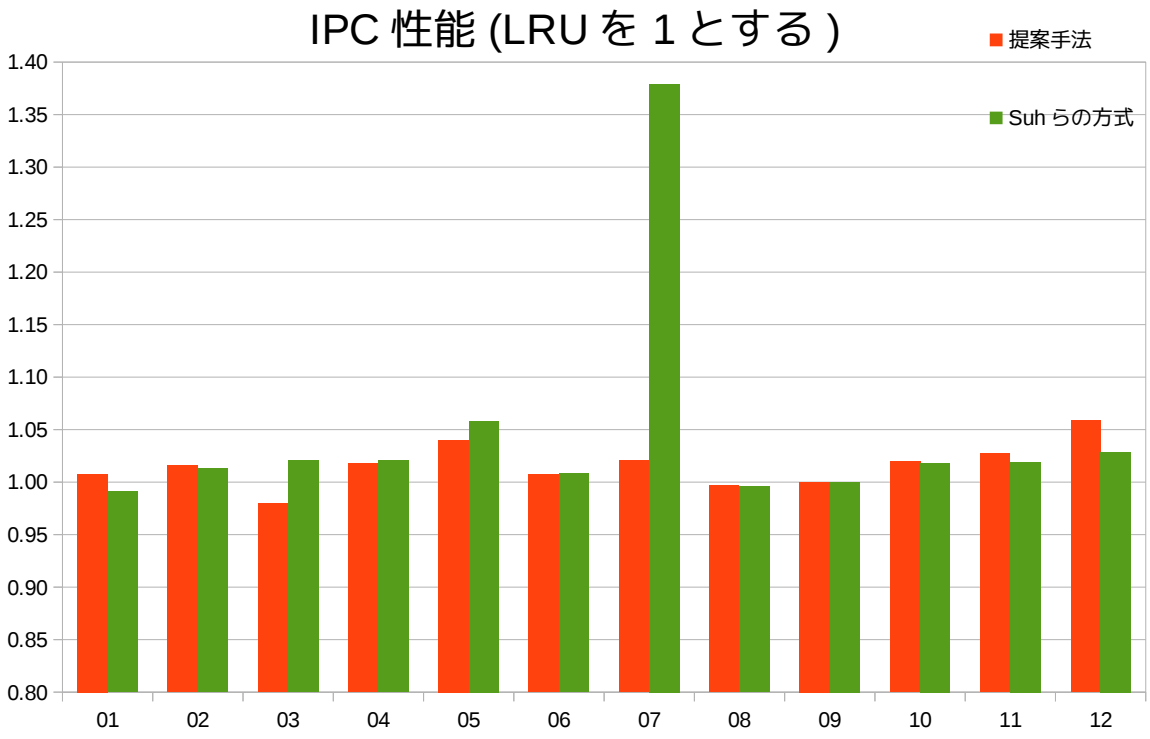

図 4.6: IPC 性能 (LRU を 1 とする)

ワークロードでも，値を変えると前述の結果を示した．それゆえ，デッドブロック予測によっては更なる性能改善が可能である．

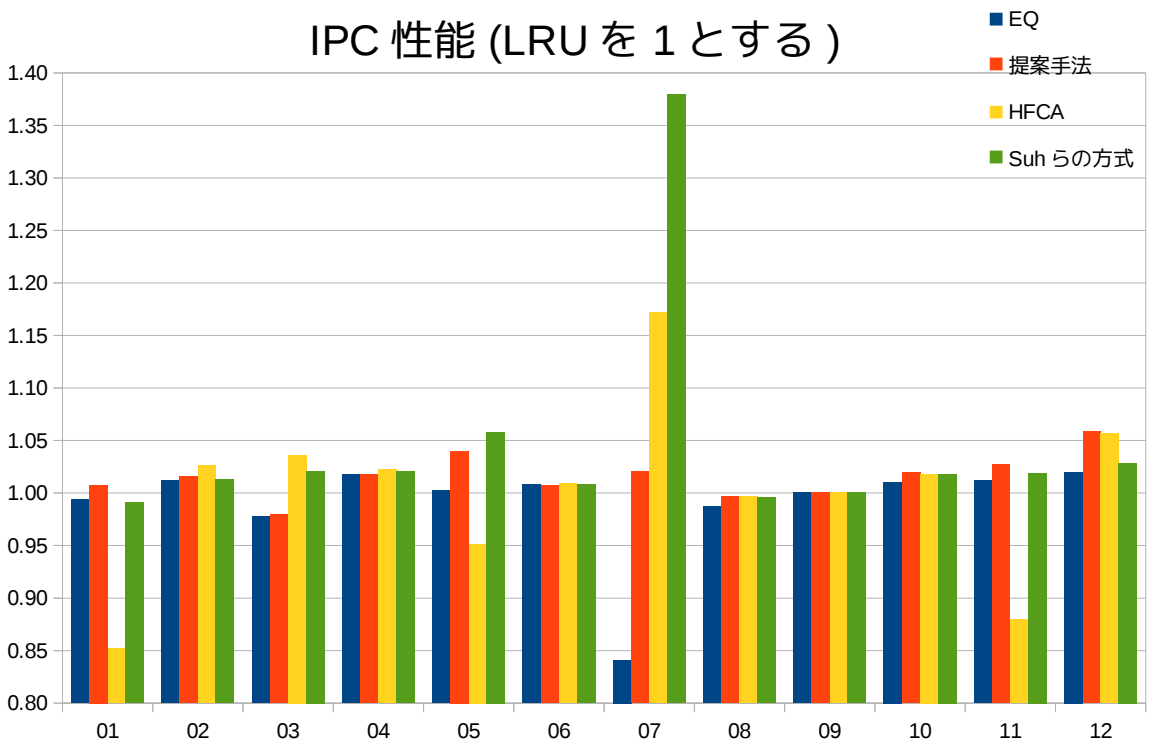

図 4.7: IPC 性能 (LRU を 1 とする)

	名前	説明
CINT2000	gzip	圧縮
	vpr	FPGA 回路の配置と配線
	gcc	GNU C コンパイラ
	mcf	組み合わせの最適化
	crafty	チェス・プログラム
	parser	言語処理
	eon	コンピュータによる視覚化
	perlbmk	Perl アプリケーション
	gap	群論, インタプリタ
	vortex	オブジェクト指向データベース
	bzip2	圧縮
	twolf	配置と配線のシミュレータ
CFP2000	wupwise	量子色力学
	swim	浅水モデル
	mgrid	3 次元のポテンシャルの場における
	applu	放物線/楕円の偏微分方程式
	mesa	3 次元グラフィック・ライブラリ
	galgel	流体力学計算
	art	ニューラル・ネットワークを用いた画像認識
	equake	地震波伝搬のシミュレーション
	facerec	顔の画像認識
	ammp	化学シミュレーション
	lucas	素数のテスト
	fma3d	有限要素法による衝突シミュレーション
	sixtrack	高エネルギー物理の粒子加速器設計
	apsi	気象学:汚染物質の拡散

表 4.1: SPEC CPU 2000

ワークロード名	アプリケーション名 (グループ) ...			
ワークロード 01	fma3d(C)	swim(D)	art(E)	apsi(E)
ワークロード 02	bzip2(B)	vpr(A)	swim(D)	fma3d(C)
ワークロード 03	wupwise(D)	eon(B)	art(E)	apsi(E)
ワークロード 04	vortex(C)	swim(D)	gzip	fma3d(C)
ワークロード 05	gcc(A)	twolf(A)	gzip(C)	art(E)
ワークロード 06	fma3d(C)	swim(D)	mcf(B)	equake(B)
ワークロード 07	ammp(A)	twolf(A)	swim(D)	lucas(D)
ワークロード 08	bzip2(B)	equake(B)	vortex(C)	crafty(C)
ワークロード 09	mcf(B)	equake(B)	swim(D)	lucas(D)
ワークロード 10	vortex(C)	crafty(C)	swim(D)	lucas(D)
ワークロード 11	vortex(C)	crafty(C)	art(E)	facerec(E)
ワークロード 12	swim(D)	lucas(D)	art(E)	facerec(E)

表 4.2: ワークロード (11-19)

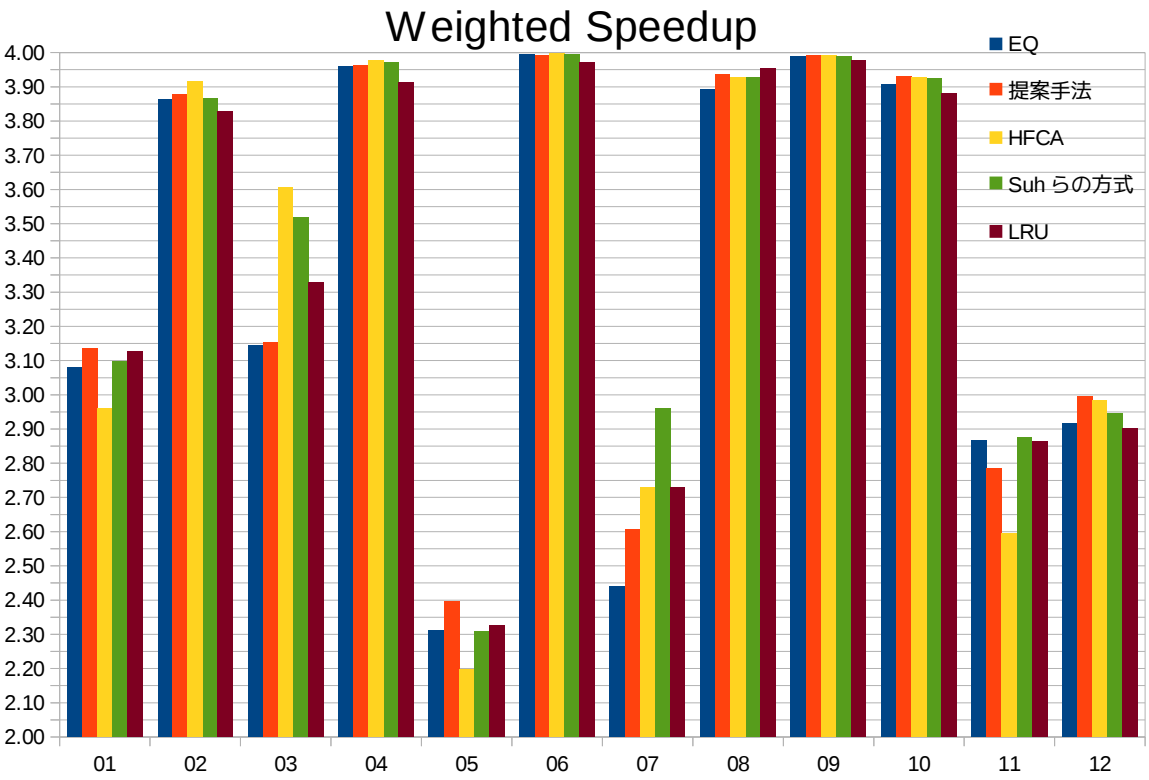

図 4.8: Weighted Speedup)

プロセッサ	コア数	4
	最大同時命令発行数	1
	インオーダ	
L1 命令キャッシュ	キャッシュサイズ	16KB
	連想度	2way
	ブロックサイズ	64Byte
	セット数	128
	ミスペナルティ	15 cycle
	置換方針	LRU
L1 データキャッシュ	キャッシュサイズ	16KB
	連想度	2way
	ブロックサイズ	64Byte
	セット数	128
	ミスペナルティ	15 cycle
	置換方針	LRU
L2 共有キャッシュ	キャッシュサイズ	2MB
	連想度	16way
	ブロックサイズ	64Byte
	セット数	1024
	ミスペナルティ	285 cycle

表 4.3: パラメータ

EQ	0.99015915
提案手法	1.01595779
HFCA	1.00166921
Suh らの方式	1.04604949

表 4.4: IPC 性能の平均値 (LRU を 1 とする)

EQ	3.36348477
提案手法	3.39633829
HFCA	3.40079941
Suh らの方式	3.44804144
LRU	3.39995194

表 4.5: Weighted Speedup の平均値

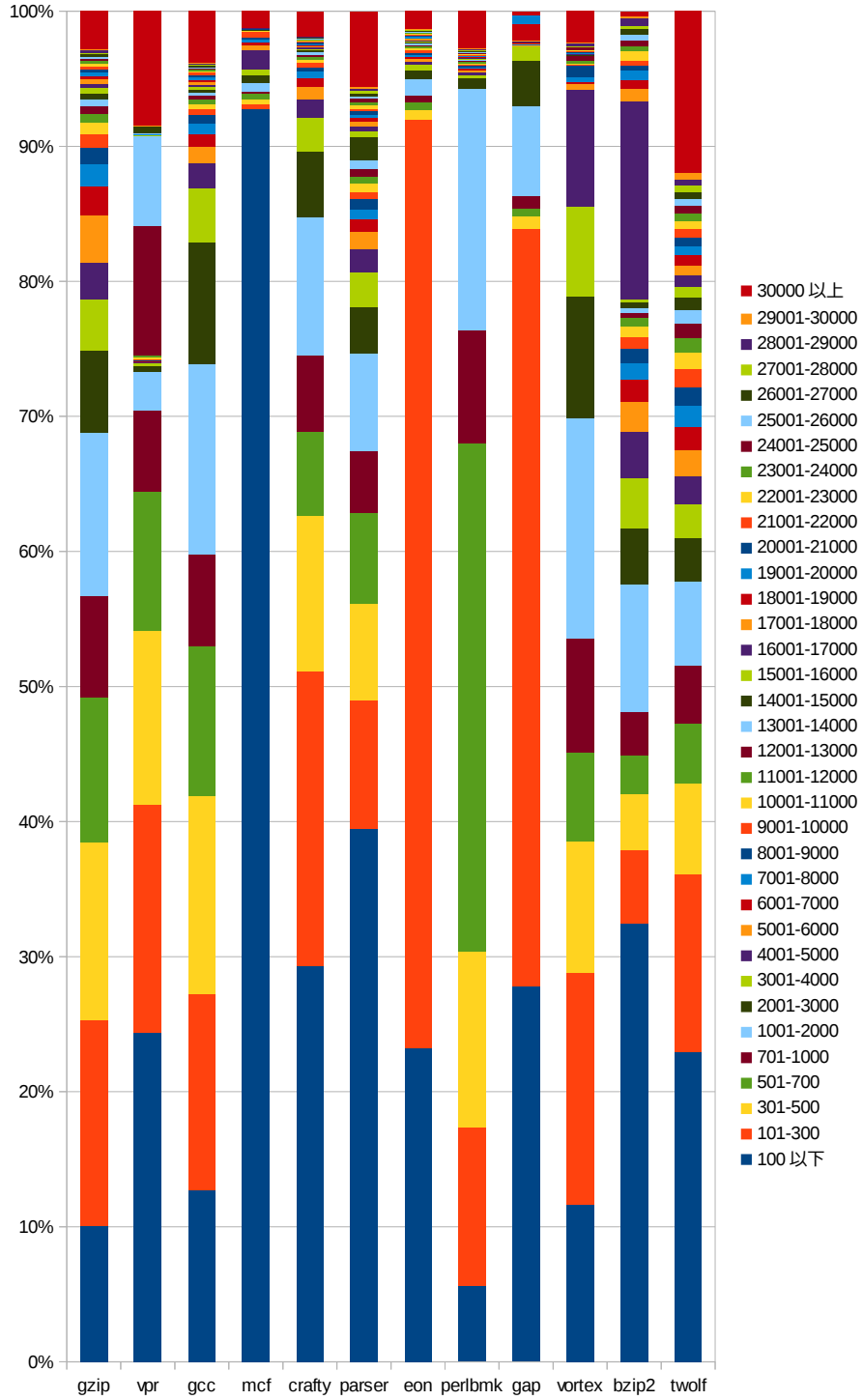

図 4.9: ブロックの最後のアクセスからヒットまでの時間とヒット数の関係 (CINT)

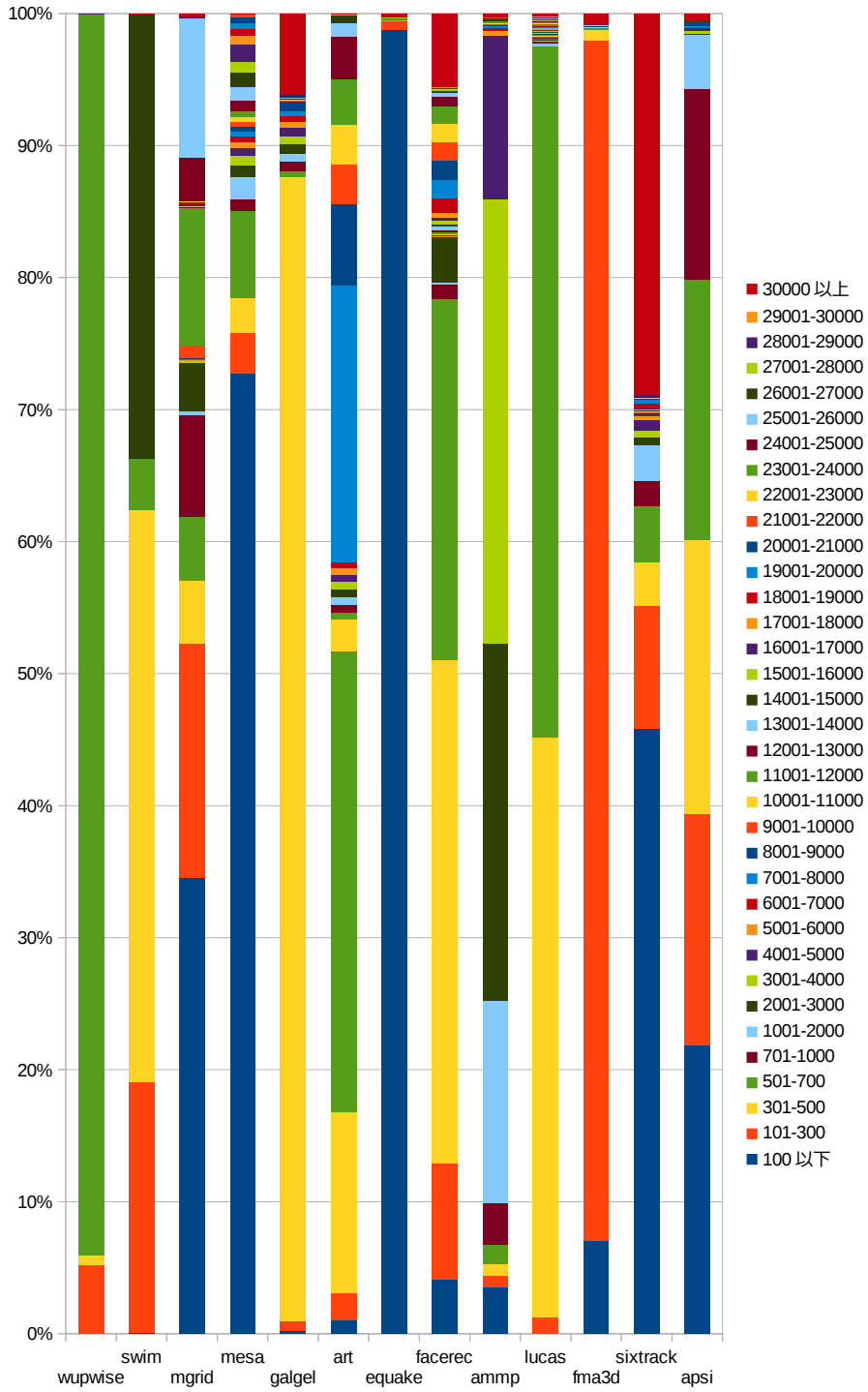

図 4.10: ブロックの最後のアクセスからヒットまでの時間とヒット数の関係 (CFP)

ワークロード	最小時間	最大時間	平均
ammp	1	20969394	6988.448602
apsi	1	178985	1016.403407
art	1	125145	9865.632506
bzip2	1	105820	25443.400574
crafty	1	10967391	9427.738174
eon	1	1258568	1139.638178
equake	1	2585663	284.480180
facerec	1	96569	5905.651539
fma3d	1	262811	31285.615569
galgel	1	1224706	8779.355022
gap	1	146598	869.250530
gcc	1	2109584	6681.136752
gzip	1	1095501	4834.458167
lucas	15	32517	850.992677
mcf	1	544626	1907.399164
mesa	1	546519	1697.987911
mgrid	1	95698	5273.794045
parser	1	1719773	9337.245501
perlbmk	1	4874709	3165.636849
sixtrack	1	1990599	13853.717490
swim	1	36518	1159.045724
twolf	1	1184335	10567.398397
vortex	1	3201082	4534.540296
vpr	1	153277	6325.383789
wupwise	46	16935	502.093149

表 4.6: 16way Single Core 実行時のアクセス間隔

第5章　まとめ

本研究では，新たにデッドブロック予測を用いた動的キャッシュパーティショニングを提案した．従来の方式がパーティションサイズを増やした際の性能向上が効果が大きいスレッドを予測し，パーティショニングを行うのに対し，本提案方式ではパーティションサイズの減少に伴う性能低下が小さいスレッドを検出し，パーティショニングを行う．

パーティションサイズの増加に着目した Suh らの方式と小川らの方式 (HFCA) と比較した時，IPC では HFCA より約 1.4%高い結果となったが，Suh らの方式より約 3.0%低い結果となった．また，Weighted Speedup では Suh らの方式に比べて 1.5%，HFCA に比べて約 0.1%低かった．また，本提案方式と Suh らの方式と比較した時の特性として，Suh らの方式がパーティションサイズの増加による効果が大きいアプリケーションを含むワークロードの時に高い効果を発揮し，本提案方式はミス数が多いアプリケーションやパーティションサイズの増加の判断が困難なアプリケーションを含むワークロードで性能改善が見られた．

5.1　今後の課題

本提案方式は，デッドブロック予測に使用するしきい値が性能に与える影響が大きいため，しきい値の動的算出方法に改善の余地がある．また，本研究ではデッドブロック予測器の実装に関しては言及しておらず，それらについては更に検討する必要がある．

また，その他に以下の検討課題がある．

- 置き換えアルゴリズムにデッドブロック予測の情報を利用．
- パーティションサイズの増加に着目した方式との組み合わせ．

本研究では，評価を複雑にしないため，デッドブロック情報をキャッシュパーティショニングの予測にのみ使用したが，本提案方式は Suh らの方式とは異なり，置き換えアルゴリズムに応用できる．それゆえ，置き換えアルゴリズムに使用することで更なる性能改善が可能である．

付 録A　Inclusion Property

Inclusion Property で比較する上で，各アプリケーションを次のような基準でグループ分けを行う．

A. 連想度を増加する程 MPKI が低下するアプリケーション (図 A.1).

B. 連想度 1 の時の MPKI が 10 未満で小量の連想度を追加することで，MPKI が 1 以下になるアプリケーション (図 A.2).

C. 連想度 1 の時の MPKI が 10 以上で小量の連想度を追加することで，MPKI が 1 以下になるアプリケーション (図 A.3).

D. 小量の連想度で MPKI が 1 より大きな値で収束するアプリケーション (図 A.4).

E. 連想度に伴う MPKI の改善の収束箇所が複数あるアプリケーション (図 A.5).

ワークロードは表 4.2 で示したものと同じものを使用するが，Inclusion Property と Exclusion Property ではいくつかのアプリケーションが異なるグループとなる．
IPC の比較結果は図 A.6 に，その平均値は A.1 に示す．

EQ	0.993826567
提案手法	1.020556666
HFCA	0.949688891
Suh らの方式	1.039719409

表 A.1: IPC 性能の平均値 (LRU を 1 とする)

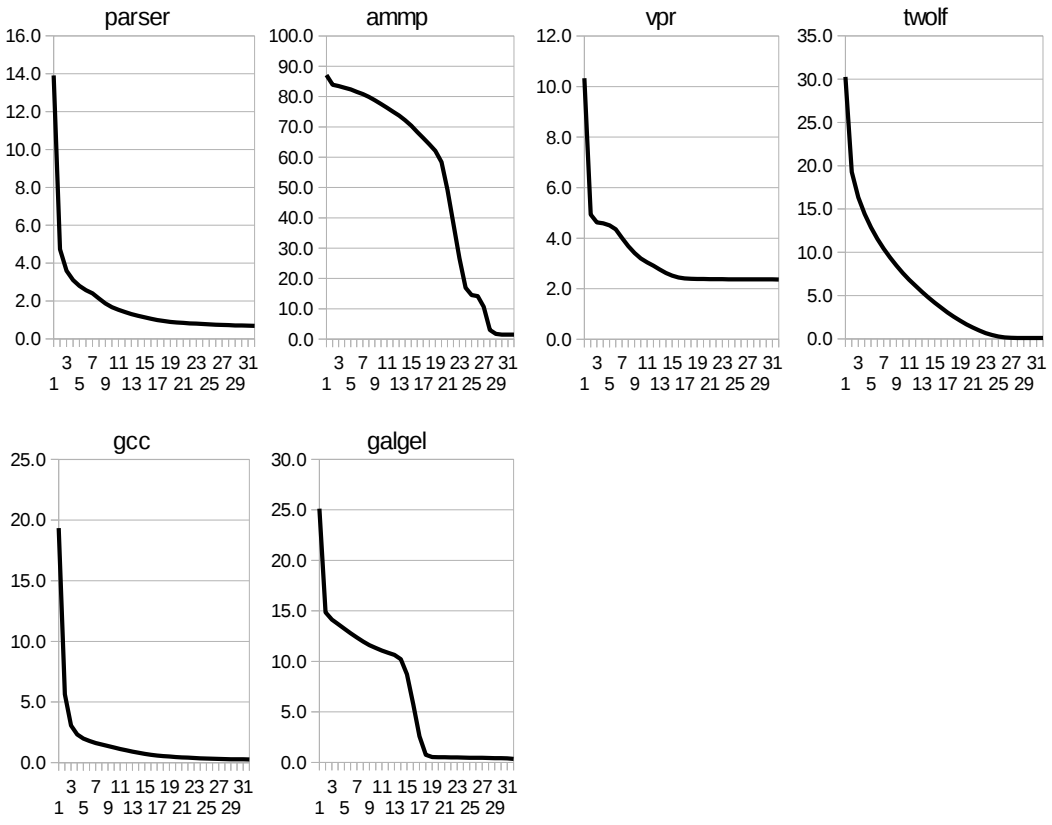

図 A.1: 連想度を増加する程 MPKI が低下するアプリケーション (グループ A)

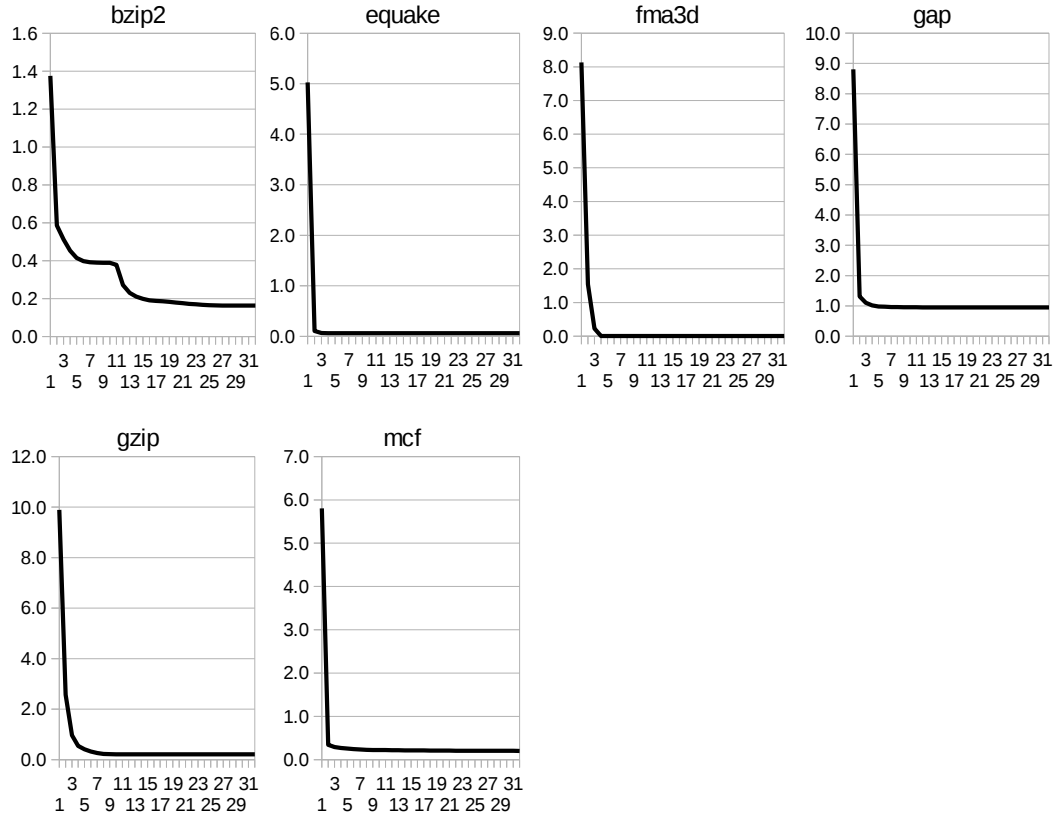

図 A.2: 連想度 1 の時の MPKI が 10 未満で小量の連想度を追加することで MPKI が 1 以下になるアプリケーション (グループ B)

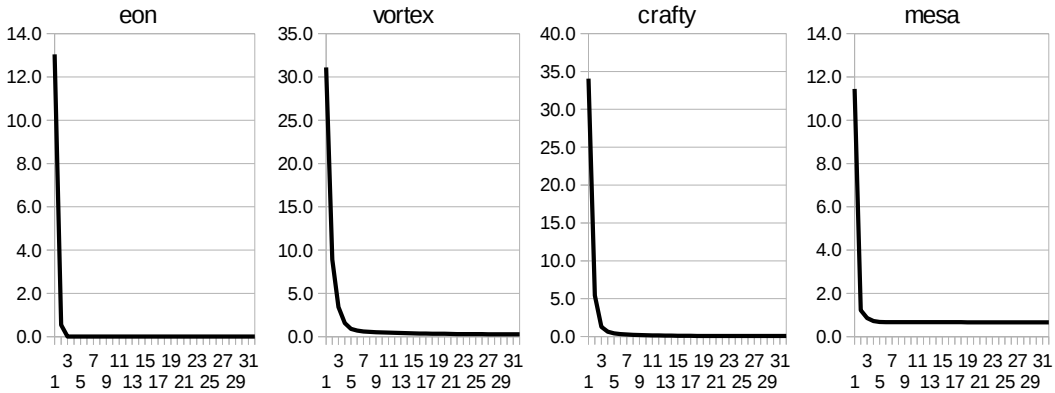

図 A.3: 連想度 1 の時の MPKI が 10 以上で小量の連想度を追加することで MPKI が 1 以下になるアプリケーション (グループ C)

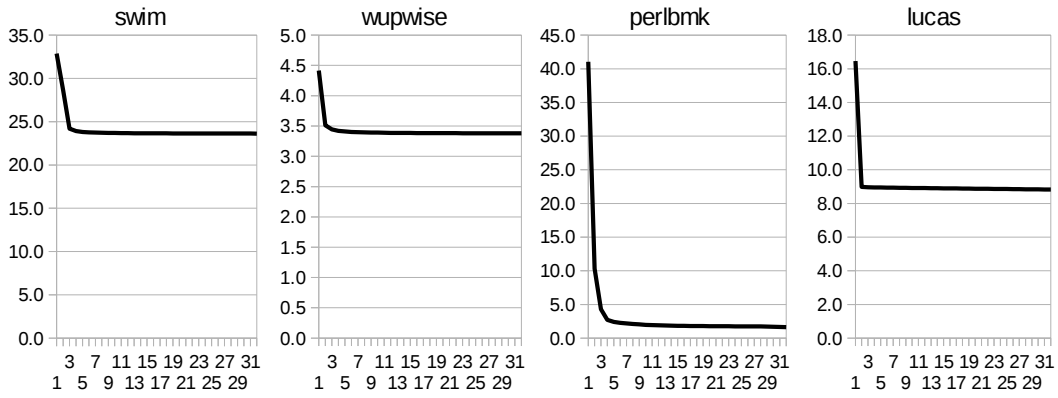

図 A.4: 小量の連想度で MPKI が 1 より大きな値で収束するアプリケーション (グループ D)

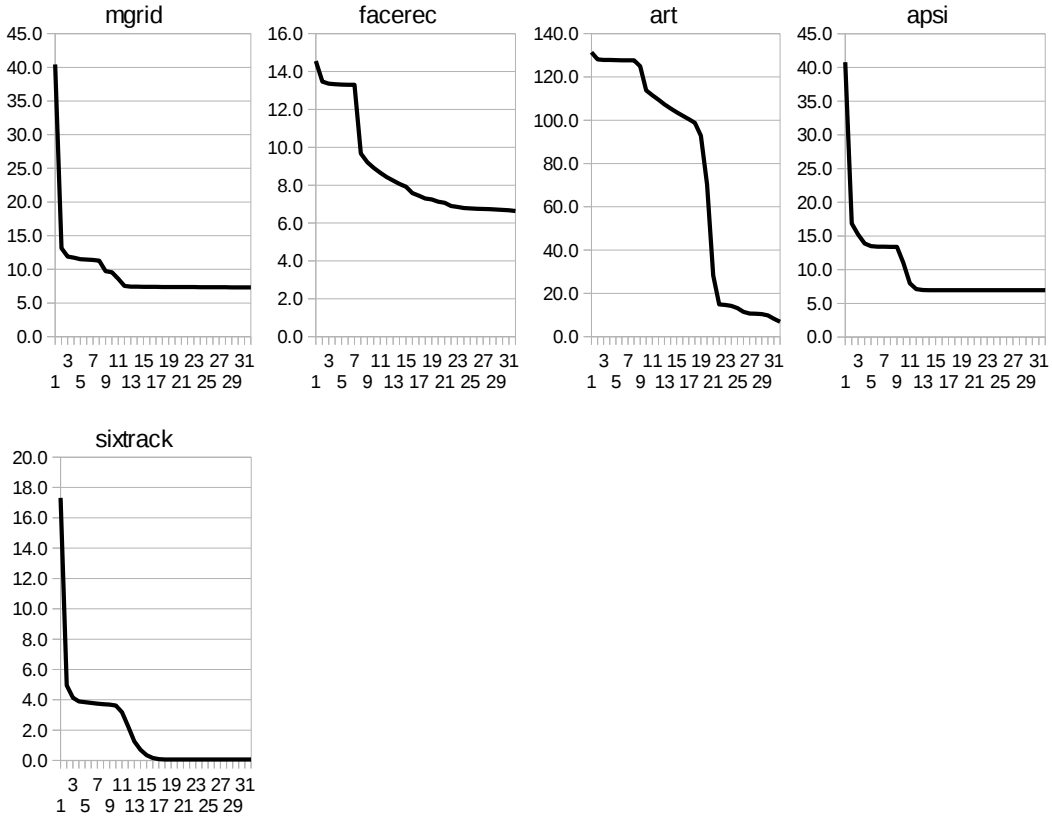

図 A.5: 連想度に伴う MPKI の改善の収束箇所が複数あるアプリケーション (グループ E)

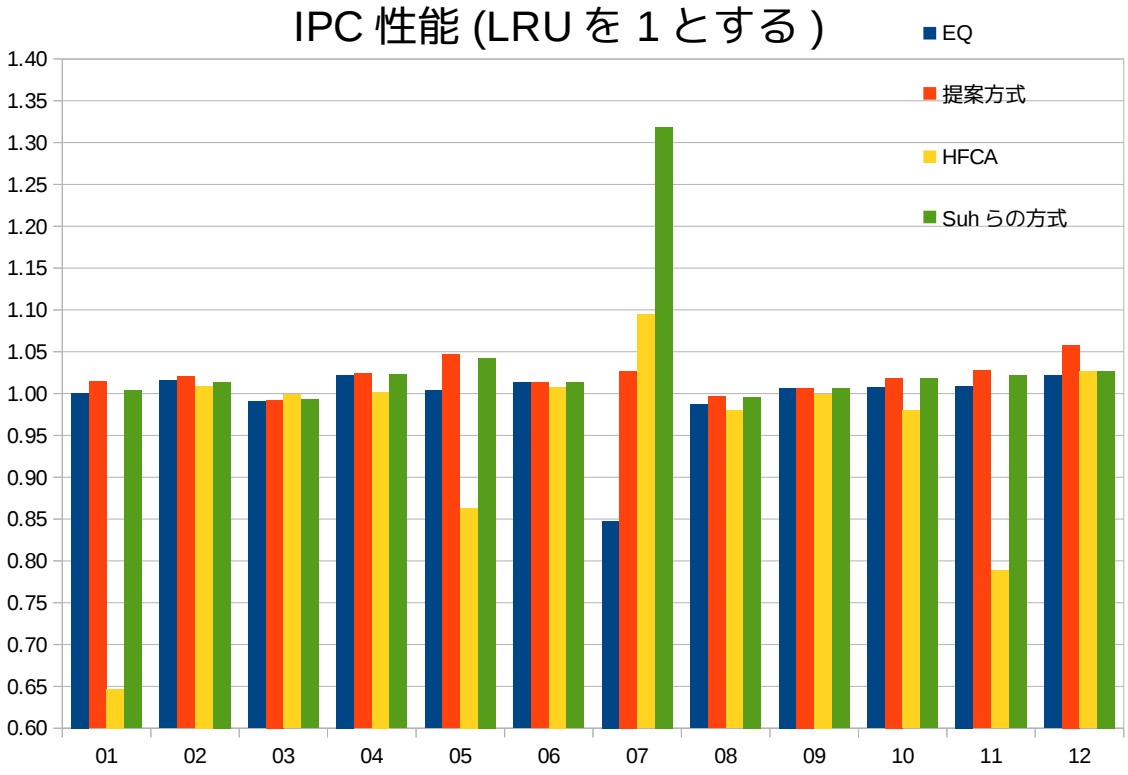

図 A.6: IPC 性能 (LRU を 1 とする)

参考文献

[1] David A.Patterson and John L.Hennessy 成田光彰訳「コンピュータの構成と設計 ハードウェアとソフトウェアのインタフェース 第3版」pp232–238

[2] John L.Hennessy and David A.Patterson "COMPUTER ARCHITECTURE A Quantitative Approach Fourth Edition" 情報処理, pp3-4

[3] Standard Performance Evaluation Corporation(Last Update2001/10/30) "SPEC CPU2000: Read Me First" ⟨https://www.spec.org/cpu2000/docs/readme1st.html⟩ (accessed 2015/01/20)

[4] Simple Scalar LLC "Simple Scalar Overview" ⟨http://www.simplescalar.com/overview.html⟩ (accessed 2015/01/20)

[5] Harold S. Stone, Fellow, IEEE, John Turek, Member IEEE, and Joel L. Wolf "Optimanl Partitioning of Cache Memory" IEEE TRANSACTION ON COMPUTER VOL.41, NO. 9, SEPTEMBER 1992

[6] G.Edward Suh, Larry Rudolph, and Srinivas Devadas "Dynamic Cache Partitioning for Simultaneous Multithreading Systems" PROCEEDINGS OF THE IASTED INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, 2001, August

[7] G.Edward Suh, Larry Rudolph, and Srinivas Devadas "Dynamic Partitioning For Shared Cache Memory" The Journal of Supercomputing, 2002, July

[8] Moinuddin K. Qureshi, Yale N. Patt "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches" The 39th Anuual IEEE/ACM International Symposium on Microarchitecture(MICRO '06), pp423-432(2006)

[9] Haakon Dybdahl, Per Stenstrom, and Lasse Natvig "A Cache-Partitioning Aware Replacement Policy for Chip Multiprocessors" Proc. 2006 ACM Conference on High Performance Computing, pp22-34(2006)

[10] 小川周吾, 入江英嗣, 平木敬「置換データの性質に着目した動的キャッシュパーティショニング」情報処理学会研究報告 IPSJ SIG Technical Report Vo.2009-ARC-184 No.20 2009/8/5

[11] 小川周吾, 入江英嗣, 平木敬「アクセス履歴の不要なマルチコア CPU 向け共有キャッシュ配分方式」先進的計算機版システムシンポジウム SACSIS 2010, 平成 22 年 5 月 pp 267-276

[12] 小笠原嘉泰, 他「マルチスレッドプロセッサにおける再構成可能キャッシュメモリ」情報処理学会研究報告. MPS, 数理モデル化と問題解決研究報告 2006(68), 67-70

[13] 広山貴之「キャッシュブロックの配置法の実用性に関する研究」2013, 修士論文