修士論文

細粒度分割キャッシュのための
ロード命令毎の特徴を考慮した挿入位置調整手法

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 23 年度入学　23413523 番

小田 遼亮

平成 25 年 2 月 5 日

細粒度分割キャッシュのための
ロード命令毎の特徴を考慮した挿入位置調整手法

小田 遼亮

内容梗概

これまでゲート遅延に対し，配線遅延が相対的に増大してきた事により，配線遅延がプロセッサ性能に与える影響が大きくなってきた．このため，配線遅延を隠蔽するキャッシュメモリの重要性が高まってきた．

この一方で，集積回路の微細化に伴うリーク電流の増大による消費電力及び発熱量の増大といった問題から，プロセッサの動作周波数の向上は困難になってきている．このことから，現在では消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力を向上させるため，単一チップ上に複数のコアを搭載したマルチコアプロセッサが広く普及している．このようなマルチコアプロセッサでは，キャッシュを有効活用するため複数のコアによってキャッシュを共有する場合が多い．ここで，複数のコアによって共有されるキャッシュ上では，他のプロセスにエントリを追い出される干渉が発生し，キャッシュヒット率が著しく低下する場合がある．このため，干渉の発生を抑制する手法として，各コアに占有領域を与えるキャッシュ分割手法が提案されている．

このようなキャッシュ分割手法の 1 つである Vantage では，多くの分割領域を管理できる．しかしながら，Vantage では多くの分割領域を効率的に管理する手法について深く考察されている一方で，その追い出しアルゴリズムの選択については深く考察されていない．そこで，本研究では Vantage に適した追い出しアルゴリズムとして，エントリの挿入位置を動的に調整する手法を提案する．また，Vantage が多くの分割領域を管理可能な点に着目し，Vantage 上でロード命令単位での分割領域割り当てを可能とする手法を提案する．また，ロード命令単位で割り当てられた各分割領域において別々の挿入位置を選択する事により，ロード命令毎の特徴を考慮した挿入位置の調整を実現し更なるキャッシュ高性能化を図る．

提案した手法の有効性を検証するため，既存の Vantage に提案手法を実装し，SPEC CPU 2006 を用いてシミュレーションにより評価した．その結果，単一のプログラムを実行する構成においてセットアソシアティブキャッシュに対し，既存の Vantage では最大 24.9%，平均 3.17%であった IPC 向上率が，ロード命令毎にキャッシュを分割した上で挿入位置を調整するモデルでは最大 26.2%，平均 4.09%まで向上する事が確認できた．

細粒度分割キャッシュのための
ロード命令毎の特徴を考慮した挿入位置調整手法

目次

1 はじめに

プロセッサの高速化は，主に集積回路の微細化による高クロック化によって実現されてきた．しかしながら，集積回路の微細化に伴い配線遅延がゲート遅延に対して相対的に増大し，配線遅延がプロセッサ性能に与える影響が大きくなってきた事により，高クロック化のみではプロセッサを高速化させる事が難しくなった．このため，配線遅延を隠蔽するための手法としてキャッシュメモリが着目され，その高速化手法が広く研究されてきた．キャッシュメモリを高性能化させるための手法は多岐に渡り，その動作をフルアソシアティブキャッシュに近づける手法 [1, 2]，追い出しの優先度を変える手法 [3, 4, 5, 6] ，データをプリフェッチする手法 [7, 8] など様々な手法が存在する．

この一方で，集積回路の微細化に伴う消費電力及び発熱量の増大といった問題から，プロセッサのクロック周波数の向上自体も困難になってきている．このような中で，クロック周波数向上以外のプロセッサ高速化手法として，SIMD やスーパスカラのような命令レベル並列性（Instruction-Level Parallelism: ILP）に着目した手法が利用されるようになってきた．しかしながら，プログラム中の ILP には限界があり，ILP に依存した性能向上も頭打ちになりつつある．このため，現在ではプロセッサあたりの処理能力を向上させるため，単一チップ上に複数のコアを搭載したマルチコアプロセッサが広く普及している．このようなマルチコアプロセッサでは，各コアの動作周波数を低く抑え，消費電力及び発熱量の問題の解決を図っている．このため，配線遅延の相対的な増大がプロセッサ性能に与える影響は抑えられつつあるが，その影響は依然として大きくキャッシュ性能を向上させる重要性も依然として大きい．

さて，このようなマルチコアプロセッサでは，キャッシュを有効活用するために複数のコアがラストレベルキャッシュを共有する場合が多い．ここで，複数のコアに共有されるキャッシュ上では，他のプロセスにエントリを追い出される**干渉**が発生し，プロセッサの性能が著しく低下してしまう場合がある．そこで，このような干渉の発生を抑制する手法として，各コアが利用可能なキャッシュ領域を制限するキャッシュ分割手法が提案されてきた．

このようなキャッシュ分割手法の 1 つである Vantage[9] では，多くの分割領域を管理できる．このため，多くのプロセスが同時に実行されるような場合にも，全てのプロセスに対して分割領域を割り当てる事ができるため，その性能を大きく向上させる事ができる．しかしながら，Vantage では多くの分割領域を効率的に管理する手法について深く考察されている一方で，その追い出しアルゴリズムの選択については深く

考察されていない．そこで，本研究ではVantageに適した追い出しアルゴリズムとして，エントリの挿入位置を動的に調整する手法を提案する．また，Vantageが多くの分割領域を管理可能な点に着目し，Vantage上でロード命令単位での分割領域割り当てを可能にする手法を提案する．そして，ロード命令単位で割り当てられた各分割領域において別々の挿入位置を選択する事により，ロード命令毎の特徴を考慮した挿入位置の調整を実現し更なるキャッシュ高性能化を図る．

以下2章では，Vantageに適した追い出しアルゴリズムを考察するために，既存の追い出しアルゴリズム，本研究で着目する手法であるVantageのベースとなるZCache，そしてVantageの動作について説明する．3章及び4章では，本論文で提案する手法について説明する．5章でこれらのモデル，及びこれらを組み合わせたモデルの性能を評価し，最後の6章において結論を述べる．

2　研究背景

本章では，高性能な追い出しアルゴリズムについて考察するために，これまで提案されてきた追い出しアルゴリズムについて説明する．また，本研究で着目するVantageのベースとなる連想度向上手法，キャッシュ分割手法についても説明し，その後Vantageについて説明する．

2.1　追い出しアルゴリズムの改良

最適な追い出しアルゴリズムとして，再参照されるまでの期間が最も長いエントリを追い出す方式であるOPT[10]が知られている．しかしながら，OPTを実現するためにはアクセスされるアドレスの順序情報を事前に知る必要があるため，OPTに基づいたキャッシュを実装する事は現実的ではない．このため，一般的なキャッシュメモリで利用されている追い出しアルゴリズムは，再参照されるまでの期間が最も長いエントリを予測し，エントリの追い出し順を決定している．例えば，よく知られるLeast Recently Used（LRU）追い出し方式では，最後に利用された時刻の最も古いエントリが，再参照されるまでの期間が最も長いエントリであると予測している．しかし，LRU方式を利用した場合，一度しか参照されないアドレスへのアクセスが多発するストリーミング処理が実行された際には，今後再参照される事の無い多数のエントリによって，キャッシュ上から有用なエントリが追い出されてしまう．また，非常に大きなワーキングセットを持つプログラムが実行された場合，キャッシュ上に配置されたエントリが再参照される前に追い出されてしまうスラッシングが発生する．このよう

に，LRU 方式では特定のアクセスパターンが発生した場合にキャッシュヒット率が著しく低下してしまうという問題がある．このため，LRU 方式においてキャッシュヒット率を低下させるようなアクセスパターンが発生した場合の性能低下を抑える，様々な追い出しアルゴリズム改良手法が提案されている．このような追い出しアルゴリズムは，再参照されないエントリを予測する手法と，再参照される確率の高いエントリの優先度を上げる手法の 2 つに大別する事ができる．

まず，再参照されないエントリを予測する手法として Lai ら [11] は過去のアクセスパターンを利用した手法を提案している．この手法では，再参照されないと予測したエントリを **Dead** とし，そのようなエントリをキャッシュ上から早期に追い出す．また，Khan ら [12] は一度も再参照されず Dead になると予測したデータをキャッシュ上に配置しない事によって更に性能を向上させた．しかし，これらの手法では再参照されないエントリのみを予測するため，その性能向上幅は限定的である．

この一方で，Lee ら [3] は再参照される確率の高いエントリの優先度を上げる手法として，Least Frequently Used（LFU）追い出し方式を提案している．この手法では，当該エントリへのアクセス頻度が最も低いエントリを追い出す．これにより，一度しか参照されないアドレスへのアクセスを多発するストリーミング処理や，全てのエントリが再参照される前に追い出されてしまうスラッシングの発生による性能低下を抑制できる．また，キャッシュに配置されてから一度でも再参照された事がある**既参照エントリ**を，キャッシュに配置されてから一度も再参照された事がない**未参照エントリ**よりも優先する手法が多く提案されている．このような手法として，Jaleel ら [5] は Re-Reference Inteval Prediction （RRIP）を提案し，Khan ら [6] は Dynamic Segmentation を提案している．これらの手法はいずれも低いハードウェアコストで実現可能でありながら，非常に高い性能を引き出す事ができる．また，これらの手法は再参照されないエントリを予測する手法よりも細かい優先度の調整が可能である．そこで，本研究では RRIP と Dynamic Segmentation に着目し，より高性能なキャッシュ追い出し方式について考察する．このため，まずはこれらの手法と LRU 方式における，既参照エントリの優先度について**図 1** を用いて説明する．なお，図 1 では，右にあるエントリほど優先度が低く，追い出され易いとする．

まず，LRU 方式では図 1(a) に示すように，キャッシュミスが発生した場合，一番右の位置に存在している優先度が最も低いエントリを追い出す．そして，要求されたデータに対応するエントリを，優先度が最も高い一番左の位置に**挿入**する．また，キャッシュヒットしたエントリは，優先度が最も高い一番左の位置に移動させる．このように，

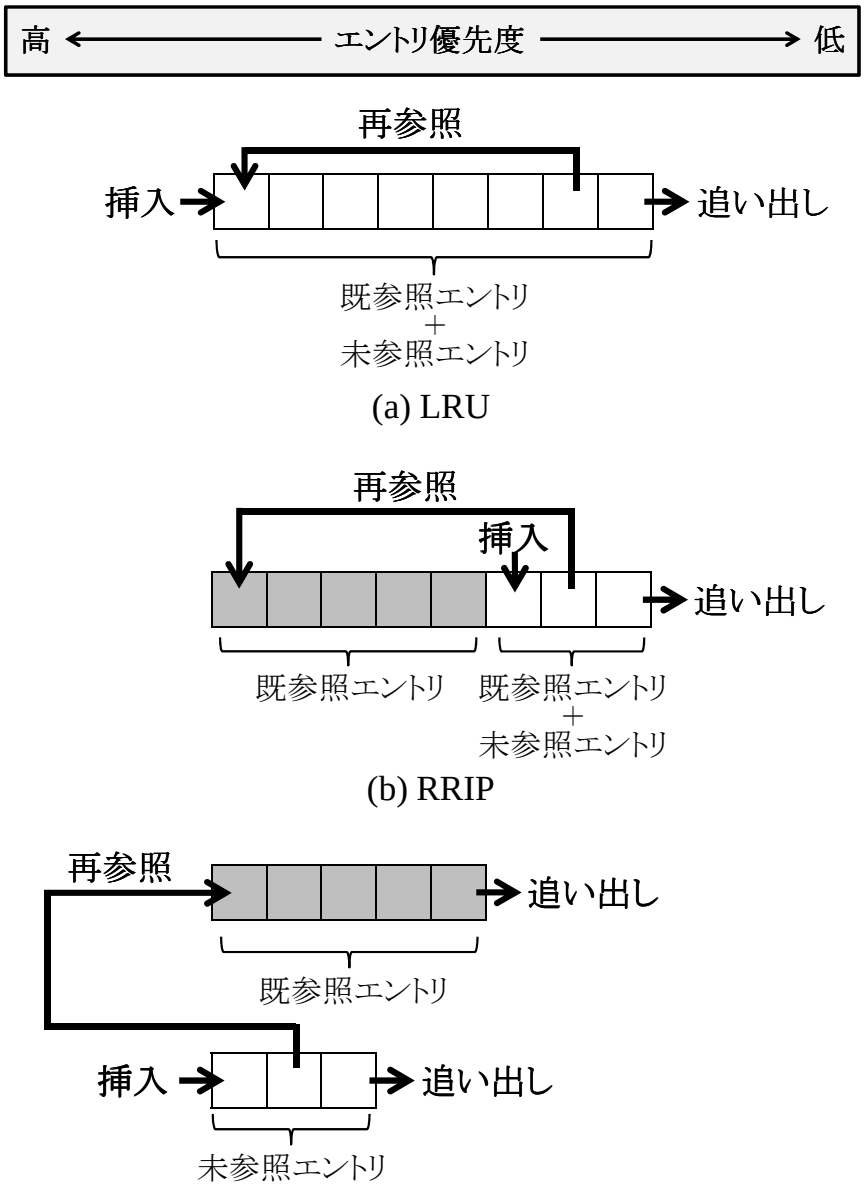


図 1: 各方式における既参照エントリの優先度

LRU 方式では未参照エントリと既参照エントリがキャッシュ上で区別されていない．

また，RRIP では図 1(b) に示すように，追い出しエントリの選択とキャッシュヒットしたエントリの移動は，LRU 方式と同様に動作する．この一方で，新しいエントリを挿入する際の動作は LRU 方式とは異なり，優先度がある程度低い位置へと挿入する．このように動作する事で，図 1(b) における濃い灰色の領域には，既参照エントリのみが存在する事になる．ここで，濃い灰色の領域に存在する既参照エントリは，未参照

エントリよりも高い優先度を持つ．このように，RRIP ではエントリの挿入位置を調整する事によって，既参照エントリの優先度を高くしている．

この一方で，Dynamic Segmentation では図 1(c) に示すように，キャッシュ内の理想的と思われる未参照エントリ数を指定する．そして，保持する未参照エントリ数を指定された数に近づけるため，保持している未参照エントリ数が指定した数よりも多い場合には未参照エントリを追い出し，そうでない場合には既参照エントリを追い出す．また，新しいエントリは未参照エントリの中で最も優先度の高い位置に挿入し，キャッシュヒットしたエントリは既参照エントリの中で最も優先度の高い位置に移動させる．これにより，現在保持している未参照エントリの数が，指定された数よりも多かった場合には，図 1(c) における濃い灰色の領域に存在する既参照エントリの優先度が未参照エントリよりも高くなる．しかし，このような方針のみに基づいてキャッシュ管理を行うと，キャッシュヒット率が著しく低下する場合がある．そこで，Dynamic Segmentaion では，未参照エントリ数を指定する方式と LRU 方式の性能を実行時に比較し，より高い性能を示す方式を動的に選択する事で更なる性能向上を図っている．

2.2 キャッシュ連想度の向上

キャッシュを利用してプロセッサを高速化させるためには，前節までに述べたような追い出しアルゴリズムの改良によってキャッシュヒット率を向上させるだけでなく，データを要求されてから当該データを返すまでのルックアップ速度を向上させる事も重要である．ここで，キャッシュのルックアップ速度を向上させるためにはキャッシュの構成方式が非常に重要となる．まず，単純なキャッシュ構成方式としては，キャッシュ上の全エントリを検索する**フルアソシアティブキャッシュ**が存在する．しかしながら，この方式では要求されたアドレスに対応するブロックを発見するために，キャッシュ全体を検索する必要があるため，ルックアップ速度が非常に遅い．このため，ルックアップ速度を向上させる手法として，検索するべきエントリの数を減らす手法が考えられてきた．この中でも，データのアドレスから格納場所が一意に決定する**ダイレクトマップキャッシュ**は，ルックアップ速度が非常に速い．

しかし，ダイレクトマップキャッシュでは，格納場所が一致する 2 つの異なるアドレスが交互にアクセスされるとスラッシングが発生してしまうため，キャッシュヒット率が著しく低下してしまう可能性が高い．よって，現在ではルックアップ速度を向上させつつスラッシング等によるキャッシュヒット率の低下を防ぐ，セットアソシアティブキャッシュが広く用いられている．このセットアソシアティブキャッシュでは，あるア

ドレスに対応するデータを格納可能な場所の数を表す**ウェイ数**を増加させる事によって，そのキャッシュがどれだけフルアソシアティブキャッシュに近い動作をするかを表す**連想度**を向上させる．しかしながら，ウェイ数を増加させた場合には，ルックアップ速度が低下してしまう．

そこで，ルックアップ速度の低下を抑えつつ連想度を向上させるための手法として，Agrawal ら [13] は 2 つのハッシュ関数を利用し，ダイレクトマップキャッシュ上で 2 段階のアクセスを可能にする Hash-rehash を提案した．この Hash-rehash では，ダイレクトマップキャッシュよりも速い平均ルックアップ速度を持ちながら，2 ウェイのセットアソシアティブキャッシュよりも低いキャッシュミス率を実現している．また，セットアソシアティブキャッシュにおいて一部のラインへアクセスが集中した場合，キャッシュの利用効率が低下してしまうため連想度が著しく低下する．このため，一部のラインに対するアクセスの集中を緩和し，連想度を向上させる手法が提案されている．このような手法として，Qureshi ら [2] はライン毎に利用可能なウェイ数を増減させる The V-way Cache を提案した．また，Seznec ら [1] はウェイ毎に別々のハッシュ関数を利用し，アクセスを分散させる Skewed-associative cache を提案した．ここで，Skewed-associative Cache は，格納可能な場所を増やさずに連想度を向上させる点が他の手法とは異なっている．

2.3 キャッシュ分割

複数のプロセスを同時に実行する場合には干渉が発生し，キャッシュヒット率が著しく低下してしまう場合がある．そこで，本節では干渉の発生原因と，干渉を回避するためのキャッシュ分割手法について説明する．

2.3.1 干渉の発生と回避

キャッシュミスが多発するようなプログラムを，他のプログラムと同時に実行した場合，他のプログラムの性能が大きく低下する場合がある．このような性能低下の発生と，これまで提案されてきた性能低下回避手法について**図 2** を用いて説明する．なお，図 2 はキャッシュサイズを増減させた場合の **Miss Per Kilo Instruction（MPKI）** を示している．

まず，401.bzip2 や 456.hmmer はキャッシュサイズを大きくした場合に性能が向上するプログラムであり，456.hmmer はキャッシュサイズをある程度まで大きくすると MPKI がほぼ 0 になるプログラムである．この一方で，410.bwaves は再参照されないデータを多く使用するプログラムであるため，キャッシュサイズを大きくした場合にも

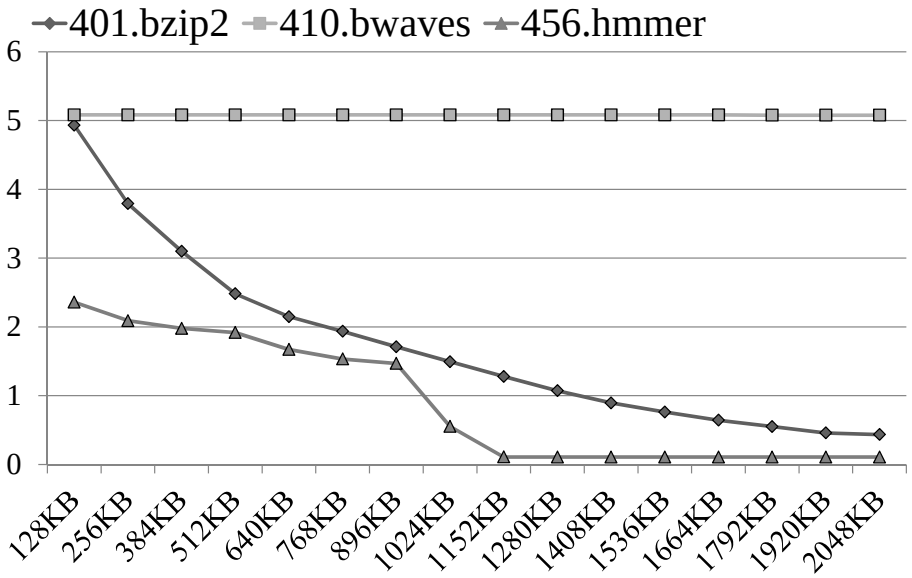


図 2: キャッシュサイズと MPKI

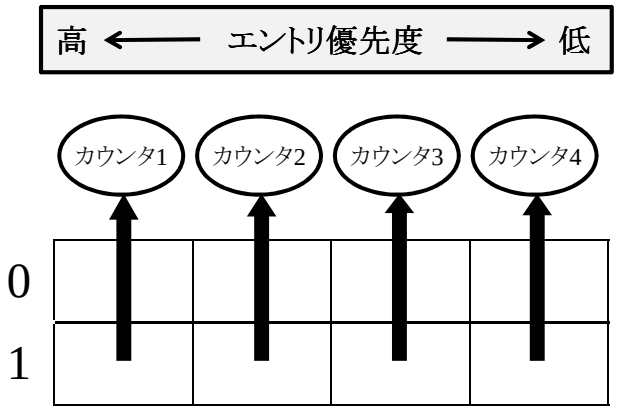
MPKI が非常に大きい．このように，再参照されないデータを多く使用するプログラムを，401.bzip2 や 456.hmmer のようにキャッシュサイズに依存して性能を向上させるプログラムと同時に実行した場合，再参照される可能性の高い有用なエントリがキャッシュから追い出されてしまい，大きく性能が低下してしまう可能性がある．

そこで，このような性能低下を回避するために，各コアやスレッドに対して占有領域を与えるキャッシュ分割手法が提案されている．このような手法として，Chiou ら [14] はウェイ毎にキャッシュを分割する Column Cache を提案した．また，Ranganathan ら [15] はウェイ数よりも細かい粒度でのキャッシュ分割手法を提案しているが，この手法ではライン毎にキャッシュを分割するため，占有可能な領域を変更する際にはキャッシュの内容をフラッシュをする必要があり，その性能が大きく低下してしまう可能性がある．

2.3.2 分割領域サイズの目標値決定

キャッシュ分割手法は，実際に分割領域に対応しているエントリ群を足しあわせたサイズである**分割領域サイズ**を制御する手法と，分割領域サイズの目標値である**占有サイズ**を決定する手法の 2 つの要素からなっている．前項では，前者の制御手法をいくつか紹介したが，本項では後者の占有サイズ決定手法ついて説明する．

これまで提案されてきた占有サイズ決定手法は，局所的なヒット情報を用いる手法と大域的なヒット情報を用いる手法の 2 つに大別する事ができる．これらの内，局所的



(a) サンプリングモニタ

ヒットカウンタ	ヒット数
カウンタ1	100
カウンタ2	0
カウンタ3	30
カウンタ4	0

(b) ヒット数

図 3: サンプリングモニタにおけるヒット数

な情報を用いる手法として Dybdahl ら [16] は CPAR を提案した．この CPAR では，各コアの占有サイズを 1 ウェイ分増加させた場合に増加するヒット数を予測するために，最後に追い出したエントリに対応するタグの情報を記憶しておく Shadow Tag Table を各コアに追加する．そして，キャッシュミスが発生した場合，要求されたアドレスに対応するタグで Shadow Tag Table を検索する．ここで，該当するタグが Shadow Tag Table に記憶されていた場合，当該 Shadow Tag Table に対応するコアのヒットカウンタをカウントアップする．このようにして，占有サイズを 1 ウェイ分大きくした時に増加するであろうヒット回数を計測しておく．また，各コアの占有サイズを 1 ウェイ分減少させた場合に減少してしまうヒット数を予測するために，最も古いエントリが存在するウェイにおけるキャッシュヒット回数を計測しておく．そして，このように計測した 2 つのヒット回数を比較し，よりヒット回数が多くなるように占有サイズを増減させる．CPAR は，このように占有サイズを指定する事で，少ないハードウェアコストでの占有サイズ決定を実現している．しかしながら，CPAR では占有サイズを 1 ウェイ分ずつしか変化させる事ができず，各コアが必要とするエントリの数が大きく変化する場合に，適した占有サイズへ変化させるまでの期間が長くなってしまう．このため，CPAR を利用する場合の性能向上率は抑制されてしまう可能性がある．

この一方で，Settle ら [17] は大域的な情報を用いた占有サイズ決定手法を提案している．この手法では，大域的な情報を収集するため**図 3**(a) に示すような，タグ情報を記憶する**サンプリングモニタ**を各コアに追加する．ここで，このサンプリングモニタ

はセットアソシアティブ方式をとっており，その追い出しアルゴリズムにはLRU方式を採用している．また，このサンプリングモニタはウェイ毎にヒットカウンタを持っており，各ウェイに対応するエントリがヒットした回数を計測している．なお，この例におけるサンプリングモニタのライン数は2であり，ウェイ数は4である．ここで，サンプリングモニタはLRU追い出し方式を採用しているため，カウンタ1と2の合計ヒット数はウェイ数が2であるモニタの合計ヒット数と一致し，カウンタ1から3の合計ヒット数はウェイ数が3であるモニタの合計ヒット数と一致する．このため，各ヒットカウンタの値が図3(b)に示すような値となっていた場合，ウェイ4におけるヒット数が0である事から，占有サイズを3ウェイ分より大きくしたとしてもヒット数が増加しないと予測できる．このようにして，様々なサイズにおけるヒット回数を知る事ができるため，各コアが必要とするエントリの数が大きく変化する場合にも，適した占有サイズへ変化させるまでの期間を短くする事ができる．

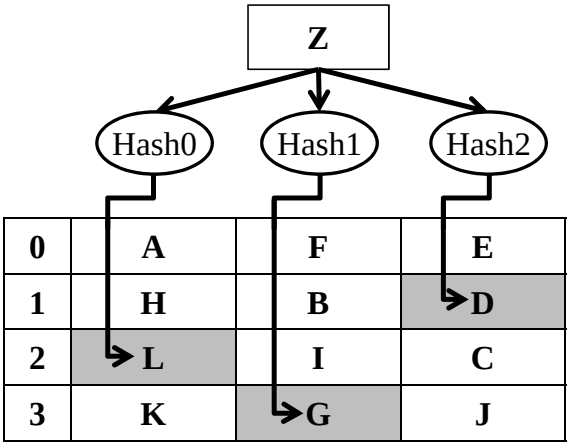
ここで，Settleらの手法では占有サイズ決定アルゴリズムとして，各コアの占有サイズを1ウェイ分増やした場合にヒット数が最も大きくなるコアへ，占有可能ウェイを1ウェイずつ順に割り当てていくGreedyアルゴリズムを利用している．このため，図3(b)に示すように，カウンタ2に対応するヒット数が0である場合，カウンタ3に対応するヒット数が多かった場合にも，当該コアに対応する占有サイズが大きく設定されない可能性がある．このため，図2で示した456.hmmerのような，ある程度大きいサイズの領域を与えなければMPKIが減少しないプログラムに対して，十分な大きさの占有サイズが割り当てられず，その性能が低下してしまう可能性がある．この一方で，Qureshi[18]らは占有サイズを一度に複数ウェイ増加させた場合のヒット数を考慮し，占有サイズを決定するLookaheadアルゴリズムを提案している．このアルゴリズムを利用した場合には，456.hmmerのようなプログラムに対しても十分な大きさの占有サイズを設定できる．

2.4 Vantage：細粒度なキャッシュ分割

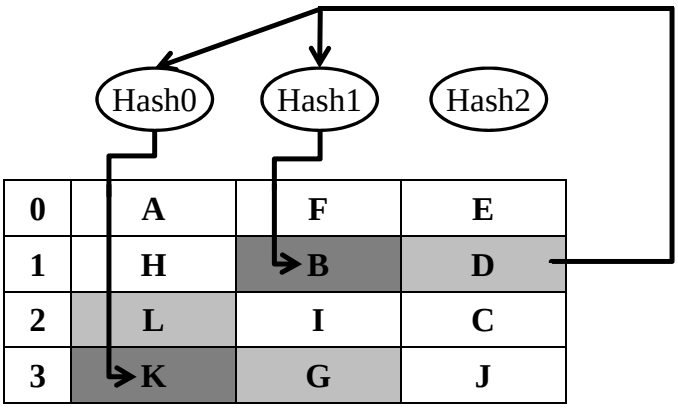
前節までに述べた連想度向上手法や分割手法には，いくつかの問題点が存在している．そこで，本節ではこれらの問題点と，これらの問題点を解決する手法であるVantageについて説明する．

2.4.1 ウェイ数と連想度の分離

2.2節で述べた連想度向上手法は，いずれも連想度がウェイ数に依存している．このため，これらの手法による連想度向上幅は限定的であり，性能向上幅が限られてし



(a) 一回目のルックアップ



(b) エントリの移動

図 4: 再帰的な追い出し

まうという問題がある．この一方で，Vantage はベースとなる連想度向上手法として，ウェイ数と連想度を独立させる ZCache[19] を利用し，この問題を解決している．

この ZCache では，Skewed-associative cache[1] と同様にウェイ毎に別々のハッシュ関数を利用した上で，キャッシュ上での再帰的な追い出しを適用する．それでは，このような再帰的な追い出しについて**図 4** を用いて説明する．なお，この図では説明を簡単にするためにウェイ数を 3，ライン数を 4 としており，キャッシュの左の番号は各ラインの番号を示している．また，キャッシュ上のアルファベットはキャッシュ上に格納されているデータに対応するアドレスを示している．

まず，キャッシュが図 4(a) のようになっている状態でアドレス Z が検索されると，ウェイ毎に別々のハッシュ関数が利用され，ウェイ毎に別々のラインが得られる．こ

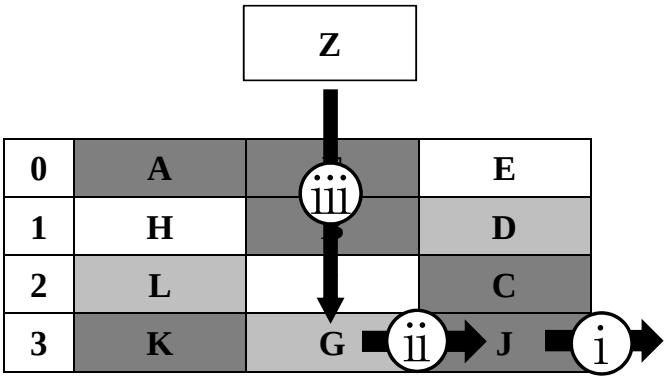


図 5: 追い出し動作

の時，得られたラインの中にアドレス Z のデータは格納されていないためキャッシュミスとなる．このように，キャッシュミスとなった場合 Skewed-associative cache では，図 4(a) で発見された配置可能ブロックに存在する 3 つのエントリから追い出しエントリを選択しなくてはならない．しかしながら，これら 3 つのエントリが持つデータに対応するアドレスは Z とは異なるため，これらのエントリを別のウェイへと移動させると，現在追い出し候補として選択されていないブロックに配置する事ができる．そこで，ZCache では既にキャッシュ上に存在しているエントリを別のウェイへと移動させ，これら 3 つのエントリをキャッシュ上から追い出さないまま，アドレス Z をキャッシュへと配置可能にする．ここで，キャッシュ上に存在しているエントリを別のウェイへと移動させる場合，移動先に存在しているエントリをキャッシュ上から追い出さなくてはならない．例えば図 4(b) に示すように，アドレス D を別のウェイへと移動させる場合，アドレス D を別のウェイに対応するハッシュ関数に通したラインに存在する，アドレス K とアドレス B に対応するデータの内のいずれかをキャッシュ上から追い出さなくてはならない．また，アドレス L や G を別のウェイへと移動させる場合にも同様に，移動先のブロックに存在するエントリをキャッシュ上から追い出す必要がある．なお，このようなキャッシュ上でのエントリ移動は再帰的に適用できるため，エントリの移動回数を増やす事によって，より多くの追い出し候補を見つける事ができる．

それでは，このような再帰的な追い出しを適用する場合，どのような手順でキャッシュ上のエントリを移動させる必要があるかを**図 5** を用いて説明する．なお，この例ではアドレス G を一番右のウェイに対応するハッシュにかけた場合にはライン 3 が得られるとする．このような状況において，アドレス J に対応するエントリが追い出し候補として選択された場合に，要求されたアドレス Z のデータをキャッシュへ配置す

るためには，G を移動させなくてはならず，G を移動させるためには J を追い出さなくてはならない．そこで，まず最初に J をキャッシュから追い出し (i)，その後 G を J があったブロックへと移動させ (ii)，要求された Z に対応するデータを G が存在していたブロックへと配置する (iii)．なお，これらの動作は主記憶からアドレス Z に対応するデータを転送してくるまでの間に完了していれば良い．このため，再帰的な追い出し動作はルックアップ速度の低下には繋がらない．

このようなウェイを跨いだ追い出しを実現するためには，ラインの中でのエントリ優先度だけではなく，キャッシュ全体の中でのエントリ優先度を知る必要がある．しかし，全てのエントリ間で優先順位を付けた場合には，各エントリが保持する優先順を表現するためのデータが非常に大きくなってしまう．そこで，ZCache はキャッシュ全体の中での優先度を表現しつつ，各エントリが保持するデータ量を削減する追い出しアルゴリズムとして**粗粒度 LRU 方式**を採用している．この粗粒度 LRU 方式では，大域的な時刻を管理し，各エントリが最後にアクセスされた時のタイムスタンプを記憶する．そして，追い出し時には各エントリが持つタイムスタンプを比較し，その中から最も古いタイムスタンプを持つエントリを追い出す．ここで，この大域的な時刻は 8 ビットで表現しており，キャッシュサイズの 5%分のデータへのアクセスが起こる度に更新する．このように，大域的な時刻を表現するためのビット数を減らす事で，各エントリが保持するタイムスタンプのビット数を削減している．

2.4.2 キャッシュサイズ管理の制限緩和

これまで，ウェイ毎にキャッシュを分割する手法 [14] が着目され，この手法を利用するキャッシュ分割方式が多く提案されてきた [17, 16, 20]．しかしながら，ウェイ数に制限されるキャッシュ分割手法では，多くの分割領域を管理する事ができない．例えば 4 ウェイのセットアソシアティブキャッシュでは，最大 4 個の分割領域しか管理できないため，5 個以上のプログラムを同時に実行した場合には，これら全てのプログラムに分割領域を与える事ができない．

これに加え，これまで提案されてきた手法では分割領域サイズを厳格に制御するため，挿入するラインと同じ分割領域に存在するエントリを追い出し候補として選択しなくてはならないという問題がある．このように，分割領域サイズを厳格に制御する追い出し手法では，多くの分割領域を扱う際にその連想度が大きく低下してしまう．例えば，あるコアの占有するウェイが 1 ウェイである場合には，追い出し候補として発見されるエントリは 1 つだけであり，その連想度はダイレクトマップキャッシュと同一になってしまう．このような分割領域サイズの厳格な制御による連想度低下は，小さ

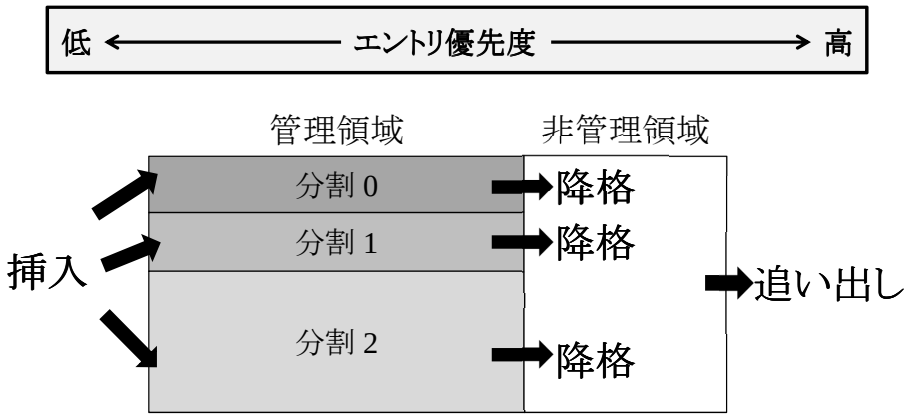


図 6: 管理領域と非管理領域

なサイズの分割領域で発生しやすい．このため，ウェイ数よりも細かい粒度でキャッシュを分割する Vantage では，分割領域サイズの厳格な制御による連想度低下が深刻な問題となってしまう．

そこで，Vantage では分割領域サイズ制御の厳格さを緩和し，別の分割領域に存在するエントリを追い出し候補として選択可能にする事で，連想度の低下を抑制する．しかしながら，このように分割領域サイズ制御の厳格さを緩和した場合，連想度を向上させる事ができる反面キャッシュ管理が複雑になってしまう．そこで，Vantage はキャッシュ管理の複雑さを解消するため**図 6** に示すように，まずキャッシュを管理領域と非管理領域に分け管理領域のみを分割する．なお，この図では右にあるエントリほど優先度が低いとする．そして，キャッシュミスが起こった場合には再帰的な検索によって多くの追い出し候補を見つける．この時，発見した追い出し候補の中から，優先度が一定値以上低い候補を非管理領域に**降格（Demote）**していく．また，キャッシュ上から追い出すエントリには，優先度が低いエントリのみが存在する非管理領域のエントリを選択する．そして，キャッシュに配置するエントリは管理領域に挿入する．ここで，各分割領域が規定した占有サイズを超える領域を必要とする場合には，非管理領域のブロックを一時的に利用する．Vantage は，このような動作により分割領域間での干渉を抑制しつつ，分割領域サイズの厳格な制御の緩和を実現する．

2.4.3 Setpoint-based Demotion

前項で述べた，分割領域サイズを柔軟に制御するキャッシュ上で，各分割領域のサイズを占有サイズに近づけるためには，分割領域毎の降格するべきエントリ量を適切に決定する必要がある．Vantage では，このような決定を Setpoint-based Demotion と

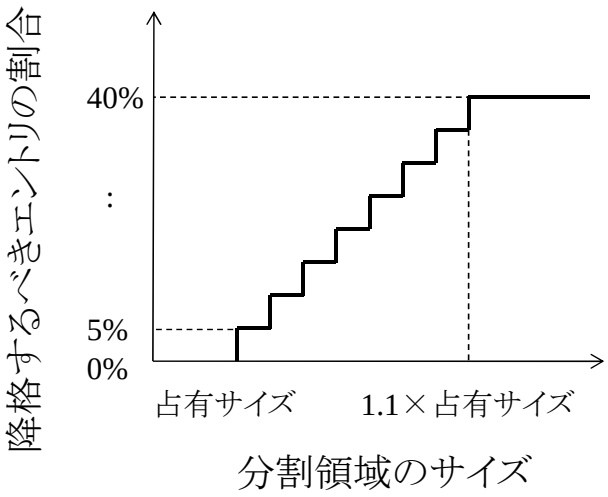


図 7: 分割領域サイズと降格するべきエントリの割合

呼ばれる手法を利用して実現する．この手法では，**図 7** に示すように降格するべきエントリの割合を各分割領域のサイズに比例させて決定する．ここで，分割領域サイズが占有サイズよりも小さい場合にはエントリを降格する必要が無いと判断し，追い出し候補として見つけたエントリの中から降格するべきエントリの割合を 0%にする．そして，分割領域サイズが占有サイズの 1.1 倍以上になった場合には，見つけたエントリの中から降格するべきエントリの割合を最大にする．ここで，Vantage では過剰な割合の降格を抑制するため，降格するべきエントリ割合の最大値を 40%としている．なお，この分割領域サイズの閾値は，占有サイズが変更されるタイミングで計算しておく．そして，現在の分割領域サイズが変わった場合には，分割領域サイズがあらかじめ計算された閾値を超えているかどうかによって降格すべきエントリの割合を決定する．

ここで，Vantage は基本的な追い出しアルゴリズムとして 2.4.1 節で説明した粗粒度 LRU 方式を用いている．このため，一定以上に優先度の低いエントリを降格するためには，タイムスタンプを利用して古さを指定しなくてはならない．そこで，Vantage は **Setpoint** と呼ばれるタイムスタンプを利用し，この Setpoint よりも古い追い出し候補を降格する．それでは，このような降格動作について**図 8** を用いて説明する．ここで，この図では，各エントリを**個体**，各エントリが保持するタイムスタンプを**データ**，タイムスタンプを**階級**とした場合の**相対度数分布**に対応する**ヒストグラム**を表している．まず，Setpoint よりも古いタイムスタンプを持つエントリを降格対象にすると，Setpoint までの**相対累積度数**と一致する割合のエントリが降格対象となる．この

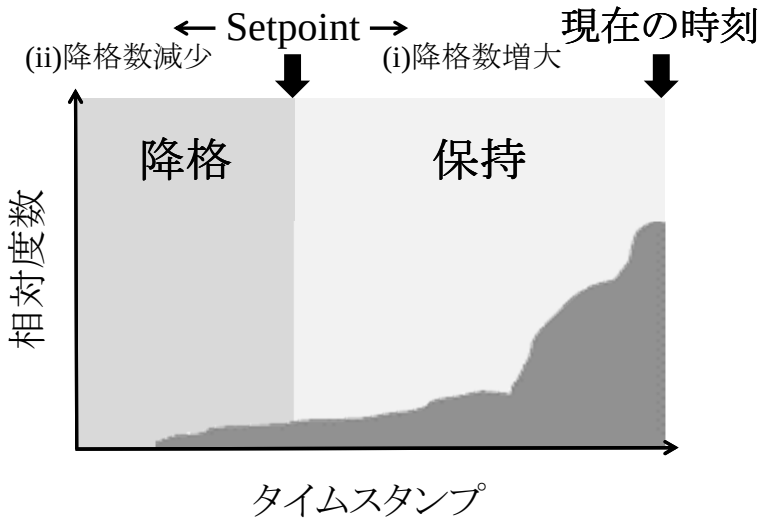


図 8: Setpoint とタイムスタンプ毎の相対度数分布

ように Setpoint を利用し，決定された割合でのエントリ降格を実現するためには，各階級における**相対度数**の情報が必要になる．しかしながら，このような情報を管理するためのコストは非常に大きい．そこで，Vantage では Setpoint の値を 256 個の追い出し候補を見つける度に調整する事により，タイムスタンプ毎の相対度数情報を管理せず，決定した割合での降格を実現する．この調整を実現するためには，256 個の追い出し候補を見つける間に降格したエントリ数と，降格するべきエントリ数とを比較する．そして，降格したエントリ数の方が少なかった場合には，Setpoint をインクリメントする事でその値を現在の時刻に近づける (i)．このように，Setpoint を右側にずらす事で，Setpoint よりも左側の領域に存在している降格するエントリの量が増えるため，より多くのエントリが降格されるようになる．この一方で，降格したエントリの方が多かった場合には，Setpoint をデクリメントする事で現在の時刻から遠ざける (ii)．このように Setpoint を左側にずらす事で，Setpoint よりも左側の領域に存在している降格するエントリの量が減るため，より少ないエントリが降格されるようになる．以上のように Setpoint を調整する事で，Setpoint よりも古いタイムスタンプを持つエントリの割合を，決定した割合に近づける事ができる．

3　細粒度分割キャッシュにおける挿入位置調整

Vantage では，キャッシュを細かく分割する手法について深く考察されている一方で，追い出しアルゴリズムの選択については深く考察されていない．そこで，本章で

は Vantage に適した追い出しアルゴリズムについて考察し，そのようなアルゴリズムとして，エントリの挿入位置を動的に調整するアルゴリズムを提案する．

3.1 挿入位置調整による既参照エントリの優先

本節では既存の追い出しアルゴリズム改良手法を，Vantage に適用する場合の問題について考察する．また，それぞれの問題を解決するための手法を提案する．

3.1.1 連想度の向上

2.1 節で説明した，挿入位置の指定によって既参照エントリの優先度を高くする RRIP では，各エントリが RRPV ビットと呼ばれるビット列を保持する事で当該エントリの優先度を表現している．ここで，RRIP ではハードウェアコストを削減するために RRPV ビットを 3 ビットとしている．このため，RRPV ビットを用いて表現可能である優先度は 8 段階でしかなく，この段階数は LRU 方式の 16 ウェイセットアソシアティブキャッシュと比較して半分になってしまっている．しかしながら，RRIP の性能をセットアソシアティブキャッシュキャッシュの性能と比較した場合，エントリ優先度の段階数が減った事による性能低下よりも，既参照エントリの優先度を高くする事による性能向上が勝る場合が多いため，全体としては性能が向上する場合が多い．

この一方で，多くの追い出し候補を見つける事ができる ZCache と，エントリ優先度が 8 段階である RRIP とを併用した場合には，見つけた追い出し候補同士が同一の優先度を持つ可能性が高くなる．ここで，同一の優先度を持つ追い出し候補間では優先順位を付ける事ができず，いずれかの候補をランダムに追い出す必要がある．このため，ZCache と RRIP を併用する場合にはランダムな追い出しが増加してしまい，その連想度を向上させる事が困難となる．そこで，本研究では粗粒度 LRU 方式をベースとしながら既参照エントリを優先する事により，連想度の向上による性能向上と既参照エントリの優先による性能向上を両立させる手法を提案する．

3.1.2 既参照エントリの優先度

2.1 節でも述べたように，既参照エントリの優先度を高くする手法は，キャッシュへ挿入するエントリの挿入位置を指定する手法と，キャッシュが保持する未参照エントリ数を指定する手法の 2 つに大別できる．これらの手法は，それぞれ既参照エントリの優先度を変化させた場合の性能について深く考察した上で提案されている．また，Khan ら [6] は Dynamic Segmentation の性能を RRIP と比較しており，その性能が RRIP よりも向上する事を示している．しかしながら，これらの手法間ではエントリ優先度の段階数が異なっており，その連想度は異なっている．このため，キャッシュが保持する

未参照エントリ数を指定する方式と，エントリの挿入位置を指定する方式の，どちらがより高い性能を示す方式であるかについては不明なままである．

そこで，Vantage に適した高性能な追い出しアルゴリズムを考えるために，まずはこれらの方式の性能を比較し，どちらの方式を採用するべきかについて考える．ここで，両方式の公平な性能比較を実現するためには，これらの方式を適用したモデルにおける連想度を一致させる必要がある．そこで，各方式を LRU 追い出しをベースとする 16 ウェイのセットアソシアティブキャッシュに適用し，そのモデルにおいて既参照エントリの優先度を変化させた場合の性能を比較する．

まず，未参照エントリ数を指定する手法の性能を調べるために，各ラインが保持する未参照エントリ数の目標値を 1 から 16 まで変化させた場合の性能について評価する．この手法は Khan らが提案している **Static Cache Segmentation（SCS）**[12] と同一の手法である．また，挿入位置を指定する手法の性能を調べるため，エントリの挿入位置をウェイ 1 からウェイ 16 まで変化させた場合の性能についても評価する．なお，ウェイ 1 には最も古いエントリが，ウェイ 16 には最も新しいエントリが存在しているとする．この手法は，以降 **Static Insert Adjust（SIA）**と呼ぶ．また，この評価ではキャッシュサイズを 512KB とし，SPEC CPU 2006 のプログラムの内いくつかを 500M 命令のスキップ後 500M 命令実行した場合の IPC を評価した．

図 9 にこれらの手法を評価した結果を示す．この図において，縦軸は IPC を表しており，各プログラム毎に性能変化があった範囲の IPC を示している．また，横軸は SCS 方式における保持する未参照エントリ数の目標値及び SIA 方式における挿入位置を示している．なお，この評価結果では，評価したプログラムを殆ど性能が変化しない Insensitive，LRU 方式における性能の方が高い LRU friendly, 既参照エントリの優先度を変える事によって大きく性能が変化する Sensitive に分類している．

まず，Insensitive に分類した 433.milc や 434.zeusmp などはキャッシュサイズを非常に大きくした場合にも，そのキャッシュミス率が殆ど変化しないプログラムである．このようなプログラムでは，既参照エントリの優先度を変化させた場合にも性能は殆ど変化しない．次に，LRU Friendly に分類した 401.bzip2 や 403.gcc では，SCS 方式と LRU 方式の IPC を比較すると常に SCS 方式の IPC が低くなっている．特にキャッシュが保持する未参照エントリ数の目標値を多く指定した場合には，SCS 方式の性能が激しく低下する事が多い．この一方で，SIA 方式の挿入位置を MRU 位置に近づけた場合の動作は LRU 方式の動作に近づくため，LRU 方式と比較して殆ど性能が低下しない．

そして，Sensitive に分類したプログラムの中でも 456.hmmer では，SCS 方式での

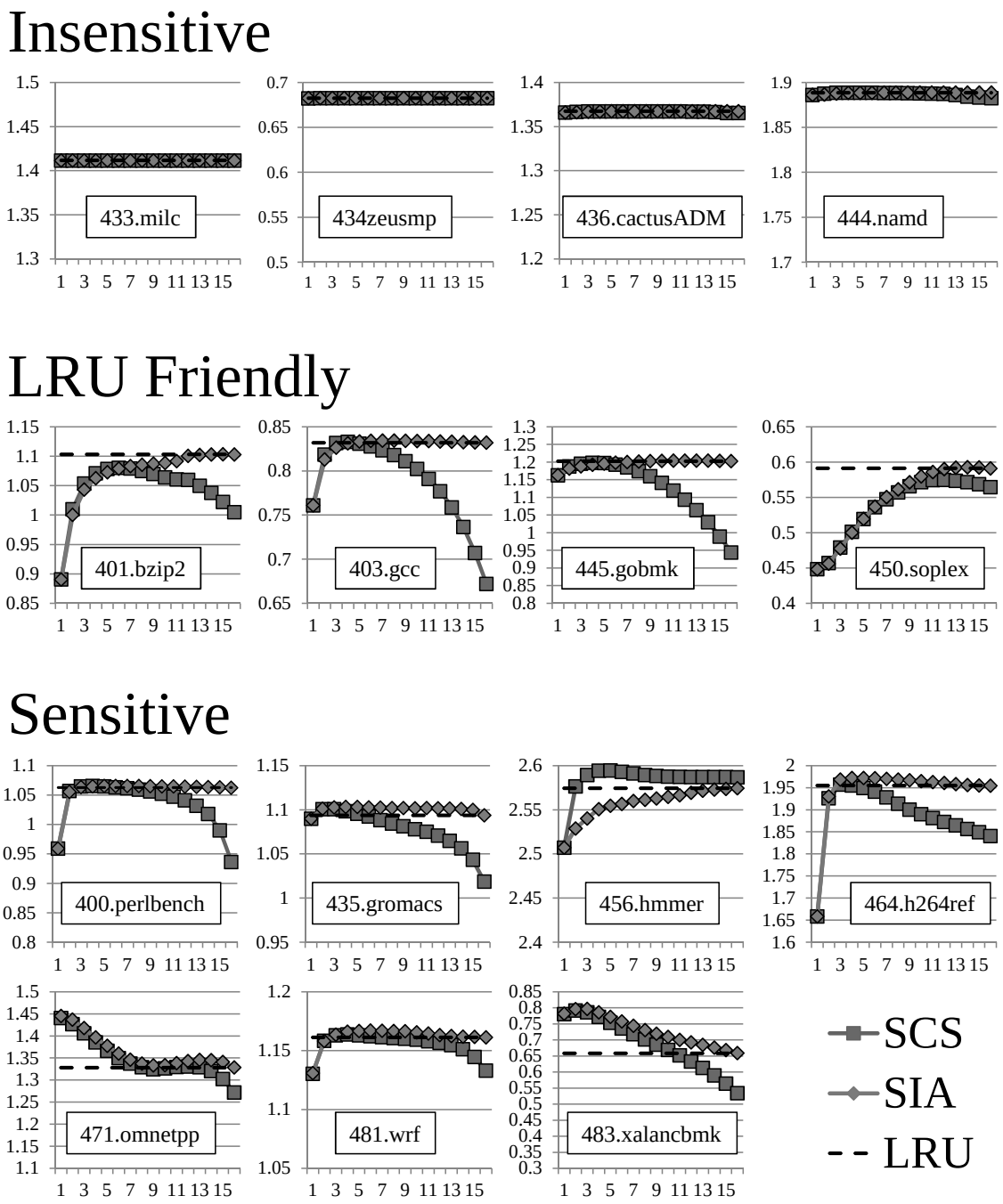


図 9: SCS 方式，SIA 方式，LRU 方式の IPC

み性能が向上している．これは 456.hmmer を実行した場合には，非常に長い間隔で最参照される未参照エントリが存在し，その未参照エントリを保持し続ける事によって，当該エントリをキャッシュヒットさせる事ができたからであると考えられる．この一

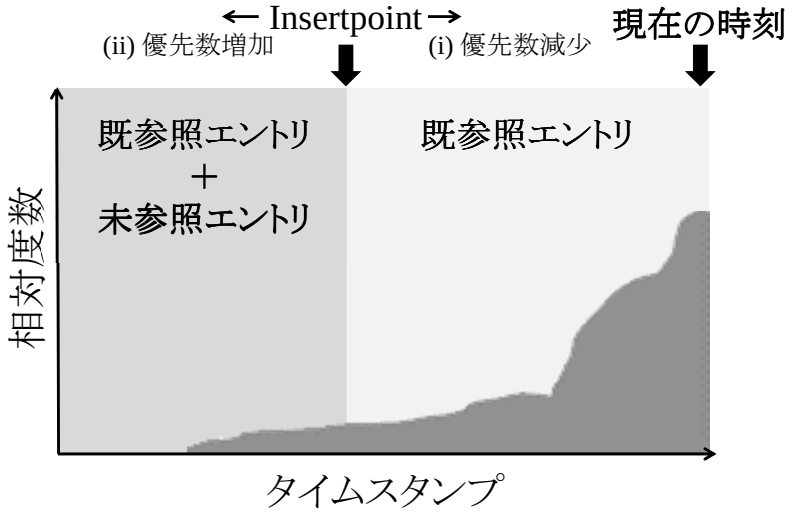


図 10: Insertpoint と既参照エントリ

方で，SIA 方式では未参照エントリを長期間保持し続ける事ができなかったため，性能を向上させる事ができなかったと考えられる．しかしながら，456.hmmer を除く全てのプログラムでは，SIA 方式の内のいずれかの挿入位置を指定するモデルが最も高い性能を示している．このように挿入位置を指定する方式は，未参照エントリ数の目標値を指定する方式よりも高い性能を示す傾向にある事が確認できた．また，SIA 方式では多くの挿入位置で性能を高く保つ事ができている一方で，SCS 方式ではキャッシュが保持する未参照エントリ数の目標値を多く設定すると激しく性能低下してしまう場合が多い．

以上のように，挿入位置を指定する方式は，様々なプログラムに対して安定した性能向上を見せる事がわかった．また，キャッシュが保持する未参照エントリ数の目標値を指定する方式を Vantage に適用する場合，未参照エントリ用と既参照エントリ用の分割領域を作る必要があり，これらの分割領域では別々の時刻を利用する事になる．このため，Dynamic Segmentation のようにキャッシュが保持する未参照エントリ数の目標値を指定する方式と LRU 方式とを切り替える手法を，Vantage 上で実現する事は困難である．そこで，本研究では安定した性能向上を見せ，かつ LRU 方式への切り替えが容易である挿入位置指定方式を利用した追い出しアルゴリズムを提案する．

3.2 Insertpoint-based Insertion

本提案手法では，粗粒度 LRU 方式をベースとした追い出しアルゴリズム上での挿入位置指定を実現するために，**Insertpoint** と呼ぶタイムスタンプを用意する．そして，

キャッシュへとエントリを挿入する際には，Insertpoint の値を当該エントリの持つタイムスタンプとして設定する．この一方で，キャッシュヒットしたエントリのタイムスタンプは，既存手法と同様に現在の時刻で更新する．このように，エントリ挿入時のタイムスタンプの設定動作を変更する事で**図 10** に示すように，Insertpoint よりも新しいタイムスタンプを持つエントリが既参照エントリのみとなり，それらのエントリが未参照エントリよりも高い優先度を持つ事になる．

ここで，Insertpoint へエントリを挿入するモデルと，現在の時刻へエントリを挿入するモデルから，より高い性能を示すモデルを動的に選択するためには，両モデルの性能を実行時に知る必要がある．これらの内，現在の時刻へエントリを挿入するモデルの性能は，既存の Vantage と同様に 2.4.3 節で述べたサンプリングモニタを利用して知る事ができる．この一方で，挿入位置を指定するモデルの性能を知るためには，挿入位置を指定可能なモニタが必要になる．なお，エントリを挿入する位置は占有サイズに伴って変化するため，占有サイズが変化した場合の性能を予測するためには，モニタにおける挿入位置も変化させる必要がある．このため，各占有サイズにおける挿入位置を指定したモデルの性能を知るには，設定可能な占有サイズの種類と同数のモニタが必要になる．しかしながら，このように多数のモニタを用意すると，その実現に必要なハードウェアが膨大な量になってしまう．ここで，このような分割領域数に比例するハードウェア量の増加は，細粒度に占有サイズを指定する事が可能である Vantage において特に大きな問題となる．例えば，キャッシュサイズの 1/256 の占有サイズの分割領域を管理する事が可能な構成では，1 つの分割領域を管理するために 257 個のモニタが必要になる．これに加えて，占有サイズを決定するためのアルゴリズムも非常に複雑になり，その決定に要する計算時間が許容できない程に長くなると考えられる．

そこで，本提案手法では占有サイズの決定には，既存の Vantage と同様に LRU 方式のサンプリングモニタを利用する．そして，決定された占有サイズの分割領域において，LRU 方式用モニタでのヒット数と挿入位置指定モデル用モニタでのヒット数とを比較し，よりヒット数の多くなる手法を動的に選択する．このため，挿入位置指定モデル用モニタでは規定された占有サイズにおけるヒット数のみを知る事ができれば良く，ヒット数をウェイ毎に観測する必要がない．そこで，挿入位置指定モデル用モニタには**図 11** に示すようにヒットカウンタを 1 つだけ持たせる．なお，この図では占有サイズがキャッシュサイズの 2/3 と規定されたコアが持つ，最大ウェイ数が 6 であるモニタを示している．このため，占有サイズよりも大きいウェイに対応する斜め罫線の引かれた 2 つのウェイは利用していない．このように，占有サイズに対応するウェイ

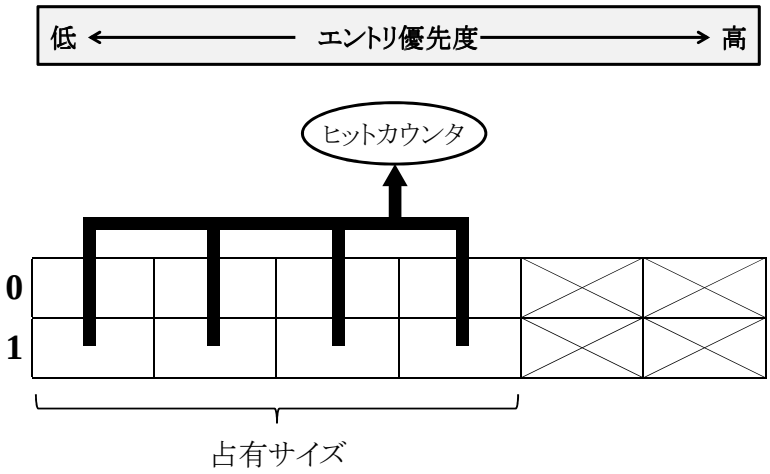


図 11: 挿入位置指定モデル用モニタ

のみを利用する事により，占有サイズにおけるヒット数を観測する事ができる．

なお，占有サイズを決定した状態で，LRU 方式のモデルと挿入位置指定モデルから，より高い性能を示すモデルを選択するためには，占有サイズを決定するよりも高い頻度で両モデルにおけるヒット数を比較しなくてはならない．このような高い頻度での性能比較を実現するためには，これらのモデルにおける短い期間毎のヒット数が必要になる．ここで，挿入位置指定モデル用のモニタにおいて短い期間毎のヒット数を観測するためには，そのヒット数を短い期間毎にリセットするだけよい．この一方で，LRU 方式用モニタでは通常の期間毎のヒット数を記憶しておく必要があるため，そのヒット数を短い期間毎にリセットしてはならない．そこで，LRU 方式用モニタにおける短い期間毎のヒット数を知るために，直近のサンプリング期間までに観測されていたヒット数を記憶するレジスタを追加する．そして，このレジスタで記憶されているヒット数を，現在のヒット数から減じる事によってその間に発生したヒット数を知る事ができる．なお，占有サイズを決定するための計算コストが非常に高い一方で，両モデルのヒット数を比較するコストは非常に低いため，高い頻度での比較が性能に与える影響は非常に小さいと考えられる．

ここで，以上のように動作する挿入位置指定モデル用のモニタでは，エントリの挿入位置をウェイ数で指定するため，優先度を上げる既参照エントリの個数がわかっている．この一方で，Insertpoint を利用するキャッシュ上において，図 10 に示すような既参照エントリのみが存在している領域に存在するエントリの数を知るためには，タイムスタンプ毎のエントリ数の情報が必要となる．しかしながら，2.4.3 項で述べたよ

うに，このような情報を管理するコストは非常に大きいため，Vantage ではこの情報を管理していない．そこで，本提案手法では挿入位置よりも新しい位置にあるべきエントリの割合を設定する事で Insertpoint を調整する．このような調整は，2.4.3 項で述べた Setpoint を調整する方法と同様に，256 個の追い出し候補を見つける度に行う．そして，挿入位置よりも新しい位置にあるべきエントリ数と Insertpoint よりも新しいエントリの数を比較し，前者の方が多かった場合には Insertpoint を右に移動させる事で，Insertpoint よりも新しいタイムスタンプを持つエントリを減少させる（図 10(i)）．この一方で，後者の方が多かった場合には Insertpoint を右に移動させる（図 10(ii)）．このように動作させる事によって，Vantage の動作とモニタの動作を近づける事ができ，挿入位置指定用モニタを利用して Vantage における性能を正確に予測する事ができるようになる．

3.3 複数位置への挿入

前節では，MRU 位置以外の一箇所だけにエントリを挿入可能であるモデルの実現方法について考察してきた．しかしながら，3.1.2 節で述べたように，挿入位置を変化させる事によって性能が変化するため，複数箇所からエントリの挿入位置を選択する事ができれば，更なる性能向上が可能であると考えられる．そこで，本節では粗粒度 LRU 上で，複数の挿入位置から最も性能を向上させる箇所を動的に選択可能にする手法の実現方法について説明する．

複数箇所からの挿入位置選択を実現するためには，**図 12** に示すように複数の Insertpoint を用意する．なお，本提案手法では挿入可能な位置を 4 箇所としており，それぞれの挿入位置に対応する Insertpoint を Insertpoint1〜4 と呼ぶ．ここで，Insertpoint1〜4 で優先するべき既参照エントリの割合はそれぞれ 15/16，3/4，1/2，1/4 とする．これにより，現在の時刻と 4 つの Insertpoint から挿入位置を選択する事ができるようになるため，既参照エントリの優先度を 5 種類から選択できるようになる．更に，Insertpoint1〜4 を利用した際の性能を予測するため**図 13** に示すように複数の挿入位置指定モデル用モニタを用意する．なお，この図では占有サイズがキャッシュサイズの 3/4 と規定されたコアが持つ，最大ウェイ数が 16 でライン数が 1 のモニタを示している．このため，占有サイズよりも大きいウェイに対応する斜め罫線の引かれた 4 つのウェイは利用していない．また，このモニタにおいて優先される既参照エントリを灰色で示している．ここで，Insertpoint1〜4 に対応するモニタである Monitor1〜4 では，優先する既参照エントリの割合を Insertpoint1〜4 と同様に 1/4，1/2，3/4，15/16

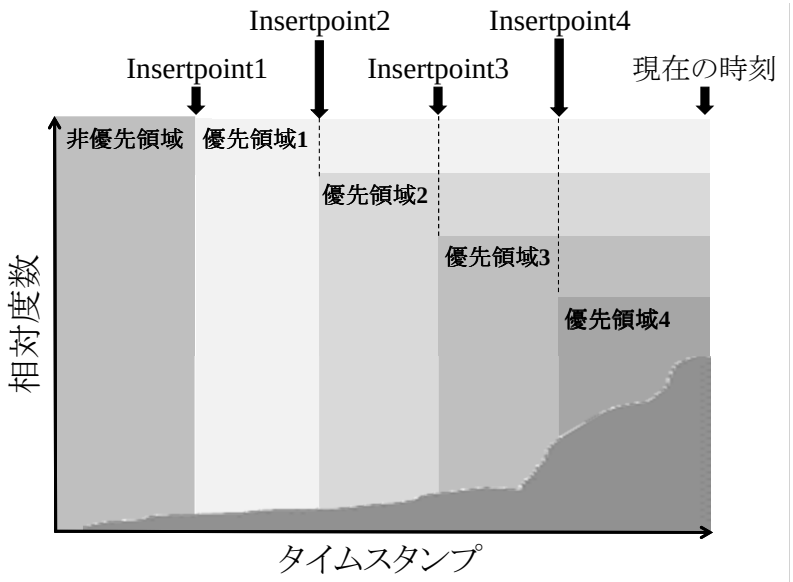


図 12: 複数の Insertpoint と複数の優先領域

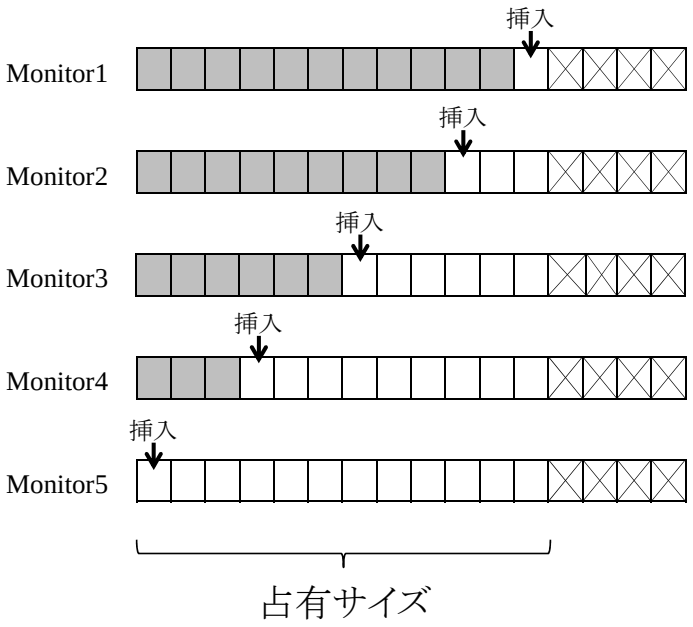


図 13: 複数の挿入位置指定モデル用モニタ

としている．また，現在の時刻に対応するモニタである Monitor5 では，優先する既参照エントリの割合を 0 としている．このため，各モニタにおいて優先している既参照エントリの数は上から順に，占有サイズに対応するウェイ数である 12 に各モニタで優先するべき既参照エントリの割合を掛けた 11 個，9 個，6 個，3 個，0 個となっており，

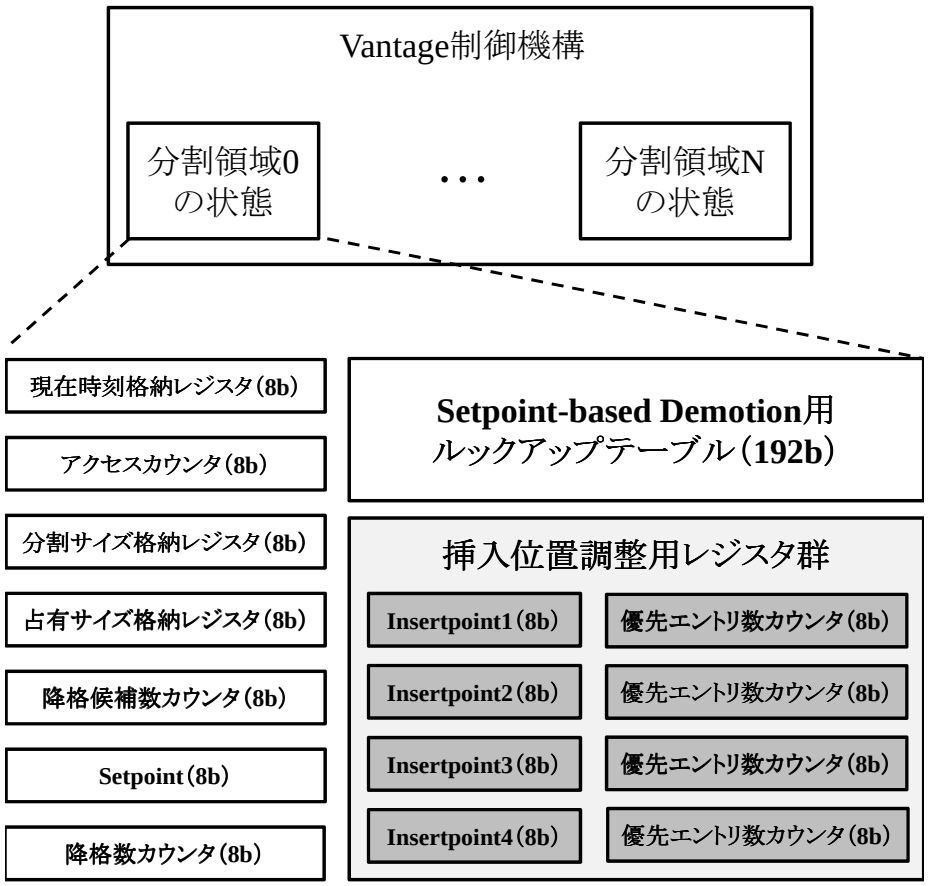


図 14: Insertpoint 用追加ハードウェア

エントリの挿入位置はウェイ 12，10，7，4，1 となっている．

3.4 追加ハードウェア

以上までに述べた挿入位置調整手法を実現するため，Vantage 制御機構に追加するハードウェアについて，**図 14** を用いて既存の Vantage に必要なハードウェアと共に説明する．なお，この図における丸括弧内の数字は，それぞれのハードウェアに必要となるビット数を示している．

まず，既存の Vantage では粗粒度 LRU を実現するために，**現在時刻格納レジスタ**を利用して現在の時刻を管理している．また，キャッシュサイズの 5%分へのアクセスが起こる度に現在の時刻を更新するために，アクセス回数をカウントしておく**アクセスカウンタ**を利用する．そして，2.4.3 項で説明したような降格するべきエントリ量の決定を実現するため，**分割領域サイズ格納レジスタ**で記憶する分割領域のサイズと**占有サイズ格納レジスタ**で記憶する占有サイズを利用して，8 段階の **Setpoint-based**

Demotion 用ルックアップテーブルを検索する．そして，256 個の追い出し候補を見つける度に Setpoint の値を調整するため，降格可能な候補として発見したエントリの数をカウントしておく**降格候補数カウンタ**を利用している．ここで，Vantage では分割領域サイズが占有数を下回る場合に Setpoint を調整していない．しかしながら，Setpoint を調整しない状況でエントリの挿入位置を古い位置に指定すると，挿入したエントリが非常に短い期間で追い出される場合がある．このような動作は，挿入位置調整手法の性能を大きく低下させる可能性がある．そこで，本提案手法では Setpoint の値を常に調整するように Vantage の動作を変更する．

また，挿入位置調整手法を実現するためには灰色で示す領域に存在する挿入位置調整用のレジスタ群を追加する．まず，**Insertpoint1〜4** は 4 つの Insertpoint を記憶するために利用する．そして，256 個の追い出し候補の中で，この Insertpoint よりも新しかったエントリの数をカウントしておくために**優先エントリ数カウンタ 1〜4** を利用する．ここで，挿入位置は各分割領域毎に決定するため，これらのレジスタ群は分割領域毎に追加する必要があり，挿入位置の調整に必要となるレジスタ群の実装に必要なビット数は合計で分割領域数 ×64bit となる．

4　ロード命令毎の特徴を考慮した挿入位置調整

本章では，Vantage においてロード命令単位で分割領域を割り当てる手法を提案する．また，この手法を 3 章で提案した追い出し手法と組み合わせ，ロード命令毎の特徴を考慮した挿入位置調整を実現する．

4.1　ロード命令単位での分割領域割り当て

ロード命令毎に，アクセスするアドレスの数や，エントリを再参照する間隔には特徴がある．このため，ロード命令毎のアクセスパターンに着目し，追い出し順を変化させれば，更にキャッシュ性能を向上させる事ができると考えられる．

ここで，ロード命令毎のアクセスパターンを考慮する手法としては，浅見ら [21] が提案した，ロード命令単位での分割領域割り当てを可能にする手法が存在している．この手法では，プロセス間だけでなく単一プロセス内でも干渉が発生する事に着目し，ロード命令単位で分割領域を割り当てる．そして，各ロード命令が必要とするエントリをキャッシュ上で保護する事により，性能向上を図っている．このように，浅見らの手法は単一プロセス内で発生する干渉の抑制を目的としている．このため，追い出しアルゴリズムの選択については考察しておらず，その追い出しアルゴリズムとしては

PC	PID	キャッシュミス数
0x120	1	980
0x324	1	200
0x220	1	1233
0x400	2	1020
-	-	-

図 15: キャッシュミス頻発命令検出表

LRU 方式を利用している．しかしながら，ロード命令毎のアクセスパターンを考慮して追い出しアルゴリズムを変化させれば，更に性能を向上させる事ができると考えられる．そこで，本提案手法では 3 章で述べた追い出しアルゴリズムをロード命令単位での分割領域割り当て手法と組み合わせ，分割領域毎に別々の挿入位置を選択する手法を提案する．ここで，浅見らの提案した手法はフルアソシアティブキャッシュに適用する手法であり，そのままでは Vantage に適用できない．また，この手法の実現に必要なハードウェアコストについては考察されておらず，その実現可能性は不明である．そこで，本論文では Vantage 上でロード命令単位での分割領域割り当てを実現する方法について説明し，その実現に必要なハードウェアについても説明する．

以下，Vantage 上でロード命令単位での分割領域割り当てを実現する方法を，ロード命令と分割領域の関連付け，分割領域の統合，分割領域 ID の解放，分割領域サイズの制御の順に説明する．

4.2 ロード命令と分割領域の関連付け

ロード命令単位での分割領域割り当てを実現するためには，ロード命令と分割領域を関連付けなくてはならない．本節では，このような関連付けを実現する方法について説明する．

4.2.1 キャッシュミス頻発命令の検出

Vantage において管理可能な分割領域の数は，ハードウェコストの観点から有限である．このため，全てのロード命令を区別しロード命令と同数の分割領域を管理する事は，現実的には困難である．

そこで，本提案手法ではキャッシュミスを頻発させるロード命令を識別し，そのようなロード命令のみに分割領域を関連付ける事で，必要となる分割領域数の低減を図

PC	PID	分割領域ID
0x800	1	3
0x804	1	3
0x200	2	4
0x204	2	4
-	-	-

図 16: 分割領域 ID 管理表

る．ここで，このようなキャッシュミス頻発命令の検出は**図 15** に示すような，ロード命令毎のキャッシュミス回数を記憶しておく検出表を利用して実現する．そして，このキャッシュミス回数が一定の閾値よりも大きくなったロード命令をキャッシュミス頻発命令として検出する．なお，この検出表ではロード命令の PC とキャッシュミス回数に加え，プロセス ID も記憶している．これは，異なるプロセスの持つプログラムカウンタの値が一致してしまう場合に，このような違いを区別しなければ全く関係の無いロード命令が同じものとして扱われてしまうためである．

ここで，本提案手法では理想的な性能を知るため，登録可能なロード命令数の制限をなくし，無制限とした状態で性能を評価している．一方で，キャッシュミスを頻発する命令は，単一のプログラム内では 10 個程度かそれよりも少ない数である事が知られている [7]．また，堀部らが提案したキャッシュミス頻発命令検出器 [22] では，そのハードウェアコストを削減した場合にも検出性能が大きく低下する事はない．このように，キャッシュミス頻発命令検出器のハードウェアコスト削減は比較的容易であると考えられる．このため，本提案手法の性能を維持したまま，キャッシュミス頻発命令検出表のハードウェアコストを削減する事も可能であると考えられる．

4.2.2 ロード命令と分割領域 ID の関連付け

ロード命令単位での分割領域割り当てを実現するためには，前項で述べた検出表によって検出されたロード命令と，分割領域 ID を関連付ける必要がある．そこで，これらの対応関係を**図 16** に示すような，分割領域 ID 管理表を利用して管理する．そして，キャッシュへのアクセス時にはこの管理表を検索する．この時，ロード命令に対応する PC/PID の組が管理表に登録されていれば，その ID を分割領域 ID として利用する．一方で，ロード命令に対応する PC/PID の組が登録されていなかった場合には，そのアクセスに利用する分割領域 ID として既存の Vantage と同様にプロセスに関連付け

られている ID を利用する．

また，新たに検出されたキャッシュミス頻発命令を管理表へ登録する際には，他のロード命令に関連付けられていない分割領域 ID を選択する必要がある．このような選択を実現するためには，各分割領域 ID が既にロード命令に対して割り当てられているかどうかを判断する必要がある．そこで，Vantage 制御機構に対して各分割領域 ID が既にロード命令に関連付けられているかどうか示すフラグを追加する．そして，新たに管理表へ登録するロード命令には，このフラグがセットされていない分割領域 ID を割り当てる．また，キャッシュ上の各エントリは，当該エントリに関連付けられている分割領域 ID を識別するために**分割タグ**を保持している．ここで，この分割タグをエントリ挿入時のみにしか設定しない場合には，キャッシュヒットを繰り返すエントリが古い分割タグを持ち続けてしまう．そこで，キャッシュ上に存在しているエントリがアクセスされた場合，当該タグの持つ分割タグをロード命令に関連付けられた分割領域 ID で書き換える．そして，各分割領域のエントリは既存の Vantage と同様に管理する事によってロード命令単位での分割領域割り当てが実現できる．

4.2.3 分割領域のサイズ決定

4.2.1 項で述べた方法で検出されたロード命令を，当該命令が検出されたタイミングで分割領域 ID 管理表へ登録した場合には，その時点では割り当てた分割領域 ID の占有サイズが 0 になっている．この時，占有サイズが 0 である分割領域に，再参照される可能性の高いエントリが配置されると，当該エントリが再参照される前に追い出され，性能が著しく低下してしまう可能性がある．

そこで，本提案手法では 4.2.2 項で述べたようなロード命令の関連付けを，占有サイズを変更するタイミングで行う．このような動作を実現するためには，ロード命令に対応する PC/PID 組を保持可能であるキューを利用する．このキューにおいて，一回のサンプリング期間に多くのロード命令が検出された場合にも，検出された全ての命令に対応する PC/PID 組を記憶可能にすると，キューを実現するためのハードウェアコストが非常に大きくなってしまう．しかしながら，このキューのサイズは 4.2.1 項で述べた検出表の動作を変更する事によって削減できる．具体的には，検出表におけるキャッシュミス回数を計測するカウンタを飽和カウンタとして実装し，このカウンタが飽和している状態でキャッシュミスが発生した際に，キューが空いていれば当該ロード命令に対応する PC/PID 組を登録する．これにより，キューが使い切られてしまうような場合，そのサンプリング期間にはロード命令を登録できなくなるが，次のサンプリング期間にはロード命令を登録できるようになる．なお，本提案手法ではキュー

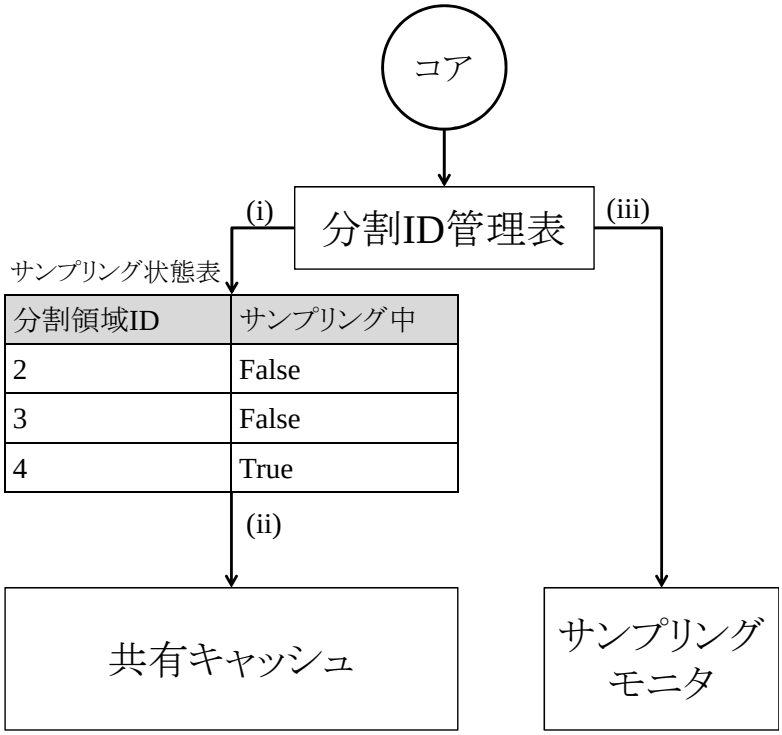


図 17: サンプリング状態表を利用した分割領域 ID 選択

の最大サイズを管理可能である分割領域の最大数と同数にする．これは，作成可能な分割領域の最大数以上のロード命令をキューへ登録できたとしても，キューへ登録されたロード命令の全てを分割領域と関連付ける事はできないためである．

ここで，新たに管理表へ登録したロード命令に関連付けられた領域でのヒット数は，そのタイミングでは計測されていない．このため，ロード命令を登録したタイミングで，必要な分割領域のサイズを予測して，分割領域を作成する事は困難である．そこで，ロード命令を管理表へ登録した後に一回分の観測期間ヒット数を観測し，その後に分割領域を作成するように動作させる．このような動作を実現するためには，キャッシュ上で利用する ID をプロセス番号に対応する ID としながら，サンプリングモニタ上で利用する ID を分割領域管理表で管理される ID とする必要がある．そこで，本提案手法では**図 17** に示すように，当該分割領域 ID がサンプリングの最中であるかどうか記憶するサンプリング状態表を管理する．

それでは，サンプリング状態表を利用した場合の動作について説明する．まず，この状態表ではサンプリング中である事を示すため，ロード命令を分割 ID 管理表へ登録する際に，当該命令を関連付ける分割領域 ID に対応する，サンプリング中かどうかを示すフラグをセットする．そして，サンプリング期間が終わったタイミングで，サン

プリング中を示すフラグを全てクリアする．また，この状態表を利用してキャッシュへのアクセスを制御するために，キャッシュへのアクセスが発生した際には，4.2.2 項で述べた分割領域 ID 管理表を検索する．そして，得られた分割領域 ID に対応するフラグを参照し (i)，このフラグがセットされている場合には，ロード命令に対応する分割領域 ID を利用せずプロセス番号に対応する分割領域 ID を利用させキャッシュにアクセスさせる (ii)．この一方でサンプリングモニタでは，サンプリング状態である場合にも分割 ID 管理表から得られた ID を利用させる (iii)．

4.3 分割領域の統合

ロード命令単位で分割領域を割り当てる際には，複数の分割領域が同一のエントリを利用した場合にその性能が低下してしまう可能性がある．本節では，このような性能低下が発生する原因と，このような性能低下の回避方法について説明する．

4.3.1 複数のロード命令による同一エントリへのアクセス

あるエントリが，別々の分割領域 ID に関連付けられたロード命令によって参照されると，複数のサンプリングモニタにおいて同一のアドレスに対応するエントリでのヒットが観測されるため，過剰なサイズの分割領域が割り当てられる可能性がある．このような動作について**図 18** に示す擬似コードを用いて説明する．

図 18 に示すプログラムは，行列の和を求める関数（1〜9 行目），行列の積を求める関数（11〜20 行目）及び main 関数（22〜29 行目）からなる．また，24 行目の関数呼び出しを実行するまでには行列積演算関数と行列和演算関数が何度か実行されており，これらの関数内の計算に利用される各ロード命令（5,6,16,17 行目）が分割領域と既に関連付けられているとする．それでは，このようにロード命令が関連付けられた状況で，24 行目以降が実行される場合のサンプリングモニタの動作について考える．

まず，24 行目において関数 MatrixAdd が呼び出されると，5,6 行目のロード命令によるキャッシュアクセスが発生する．このため 5,6 行目のロード命令に関連付けられたサンプリングモニタでは，引数として与えられた配列 A と配列 B のアドレスに対応するタグを記憶する．これに続いて 26 行目から MatrixAdd が呼び出されると，この関数には先程と同じ配列が引数として与えられているため，同一の配列がアクセスされる．これにより，サンプリングモニタ上では配列 A と B のアドレスに対応するエントリがヒットする．このため，各ロード命令に対応する占有サイズは関数に与えられた配列と同じサイズ必要であると判断され，各ロード命令に対応する占有サイズが関数に与えられた配列と同一のサイズになる．このように，ロード命令単位で分割領域を

```
1  MatrixAdd(A[],B[],n) {
2      for(i=0;i<n;i++){
3          for(j=0;j<n;j++){
4                  :
5              C[i][k] = A[i][k]    // load A[i][k]
6                      + B[i][k];   // load B[i][k]
7                  :
8          }}
9  }
10
11 MatrixMul(A[],B[],n) {
12     for(i=0;i<n;i++) {
13         for(j=0;j<n;j++) {
14             for(k=0;k<n;k++) {
15                     :
16                 C[i][j] = a[i][k]  // load A[i][k]
17                         * b[k][j]; // load B[k][j]
18                     :
19             }}}
20 }
21
22 int main() {
23         :
24     MatrixAdd(A,B,n);
25         :
26     MatrixAdd(A,B,n);
27     MatrixMul(A,B,n);
28         :
29 }
```

図 18: 行列演算

割り当てる事で，必要以上に多くのエントリを保持しないようにする事ができるため，他の有用なエントリを保護する事ができる．

この一方で，これに続いて 27 行目において関数 MatrixMul が呼び出された場合には，16〜17 行目のロード命令がそれぞれ配列 A と B にアクセスする．ここで，これらのロード命令は，一度の関数呼び出しで同一アドレスへのアクセスを何度も引き起

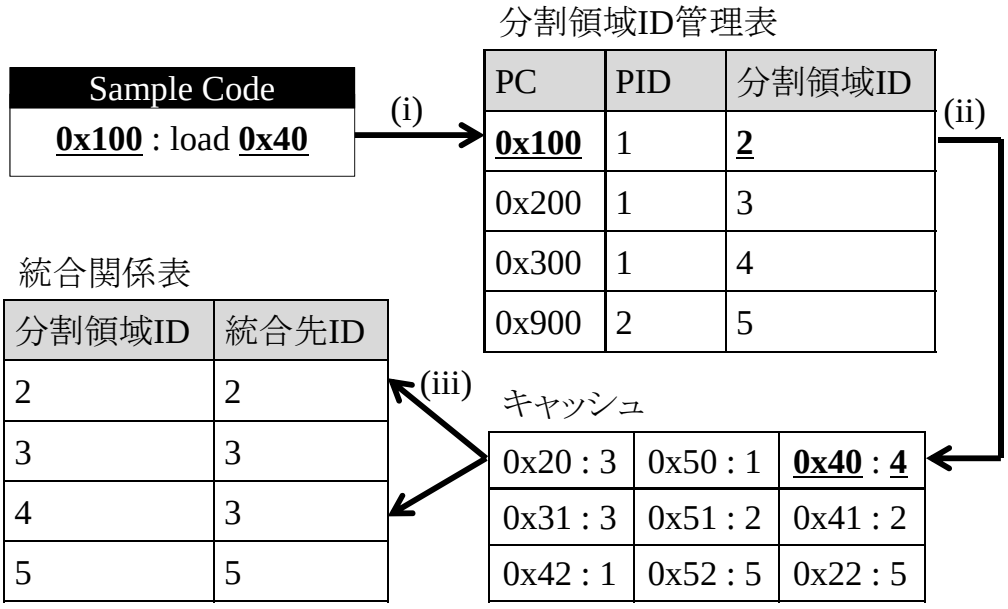
こすため，各ロード命令に対応するサンプリングモニタにおいて配列 A と B のアドレスに対応するエントリがヒットする．このため，MatrixAdd を呼び出した時と同様に，これらのロード命令に対応する分割領域も配列と同程度のサイズに設定される．このように動作した場合，各ロード命令に対応する占有サイズの合計は配列 A と B を合わせたサイズの 2 倍程度になる．しかしながら，図 18 に示すプログラムのように，プログラム内で利用される配列が A と B のみであるような場合，このプログラムが必要とするキャッシュサイズは配列 A と B を合わせたサイズでしかない．このように，同一のエントリにアクセスする複数のロード命令が，それぞれ分割領域と関連付けられている場合には，占有サイズが過剰に大きく設定される可能性がある．このような場合，他の分割領域が圧迫されるため，キャッシュ分割による性能向上が妨げられてしまう可能性がある．

4.3.2 分割領域の統合

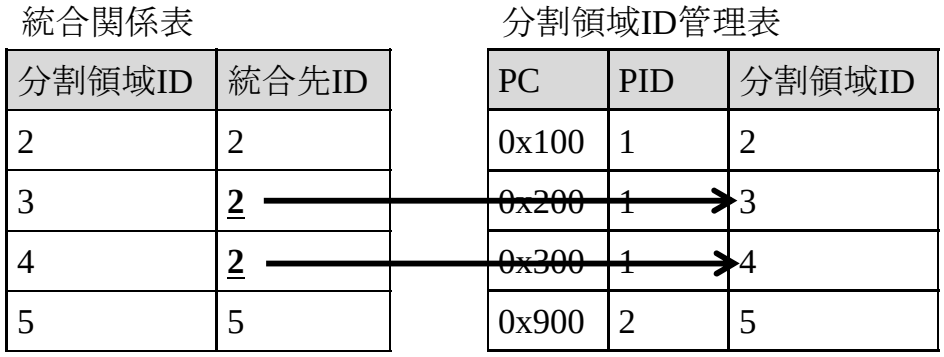
前項で述べたように，別々の分割領域に関連付けられたロード命令が同一のエントリを利用する場合には，キャッシュ分割による性能向上が妨げられてしまう可能性がある．この原因は，複数のサンプリングモニタにおいて同一のアドレスに対応するエントリが保持され，それら全てのエントリにおいてヒットが観測される事にある．

このような性能低下を回避するためには，複数のロード命令によって利用されるアドレスに対応するエントリが，同一のサンプリングモニタのみに存在している必要がある．このような動作は，あるエントリが別々の分割領域に関連付けられたロード命令によって利用された場合に，それらのロード命令を同一の分割領域に関連付ける事で実現できる．ここで，あるエントリが別々の分割領域に関連付けられたロード命令によって利用されるかどうかを検出するためには，アクセスされるエントリの持つ分割タグと，アクセスを発生させたロード命令に関連付けられた分割領域 ID を比較する必要がある．この比較の結果，両者の値が異なっており，これらの値が共にロード命令に関連付けられた分割領域 ID である場合には，当該エントリが別々の分割領域に関連付けられたロード命令に利用された事がわかる．このような場合，これらの分割領域 ID に関連付けられたロード命令を，全て同一の分割領域 ID に関連付ける事によって分割領域を統合する．

このように分割領域を統合する場合，占有サイズを設定し直さなければならない．ここで，占有サイズを設定し直す際には 2.4.3 項で述べたルックアップテーブルの値も再設定する必要があるため，その切り替えコストが非常に大きくなる．このため，複数の分割領域を統合するべきであると判断された場合，統合するべきであるという事



(a) 統合関係表の更新



(b) 分割領域ID管理表の更新

図 19: 統合関係表の更新

を次の占有サイズ変更のタイミングまで記憶しておき，占有サイズを変更するタイミングで分割領域を統合する．このように，統合先の分割領域 ID を記憶しておくためには，**図 19**(a) に示すような，統合先の分割領域 ID を管理する統合関係表を利用する．ここで，統合関係表において統合先 ID と分割領域 ID が一致している場合，その分割領域を別の分割領域に統合する必要は無い．この一方で，図 19(a) に示す分割領域 ID4 に対応する分割領域 4 のように，統合先 ID が自身に対応する ID とは異なるような場

合，当該分割領域は統合先 ID に対応する分割領域に統合するべきである．なお，この図におけるキャッシュ内の数字は，コロンの左側がエントリが保持するデータのアドレスを示しており，右側が当該エントリの持つ分割領域 ID を示している．また，図中のサンプルコード内のロード命令を実行するプロセスの PID と，そのプロセスに対応する分割領域 ID は共に 1 であるとする．

それでは，各管理表とキャッシュが図 19(a) に示すような状況で，図中のサンプルコードが実行された場合の動作について説明する．このようなロード命令が実行されると，まず当該ロード命令に関連付けられた分割領域 ID が分割領域 ID 管理表から検索される (i)．今回は，PC が 0x100 であるエントリが管理表に登録されており，このアクセスを発生させたロード命令に対応する分割領域 ID は 2 である事がわかる．これに続いて，キャッシュからアドレス 0x40 に対応するエントリが検索されると，キャッシュ上に存在する当該アドレスに対応するエントリが発見される (ii)．この時，発見されたエントリが保持する分割タグの値と，ロード命令に対応する分割領域 ID の値が異なっているため，これらの分割領域は統合するべきであると判断される．このような場合，分割領域 2 及び 4 に対応する統合先 ID を参照する (iii)．ここで，分割領域 4 は既に分割領域 3 に統合するべきであると判断されている．このように，既に統合するべき分割領域がある場合，それらの分割領域も含めた全ての分割領域を統合する．なお，本提案手法では，統合すべき 2 つの分割領域をどちらの分割領域に統合するかという選択を単純にするため，統合先 ID が大きい分割領域を小さい分割領域に統合する．このため，例では分割領域 4 及び 3 が分割領域 2 に統合されるべきと判断され，分割領域 2,3,4 は全て分割領域 2 として統合される．また，この他にも分割領域 3 に統合されるべきと判断された領域が存在する場合，それらの分割領域も全て分割領域 2 に統合される．これは，統合関係表における統合先 ID が 3 であるエントリの値を全て 2 に書き換える事で実現できる．そして，このように更新した統合関係表の情報を利用して占有サイズの変更時に分割領域を統合する．ここで，分割領域を統合するためには図 19(b) に示すように，統合関係表から分割領域 ID と統合先 ID が異なる分割を探し，分割領域 ID 管理表における当該分割領域 ID に一致する値を全て統合先 ID で書き換える．なお，統合した後の占有サイズは統合される前の占有サイズを足しあわせたサイズとする．

この一方で，アクセスされたエントリの持つ分割領域 ID がロード命令に関連付けられた ID であり，当該エントリにアクセスしたロード命令が分割領域に関連付けられていない場合にも，複数のサンプリングモニタにおいて同一のアドレスに対応するエン

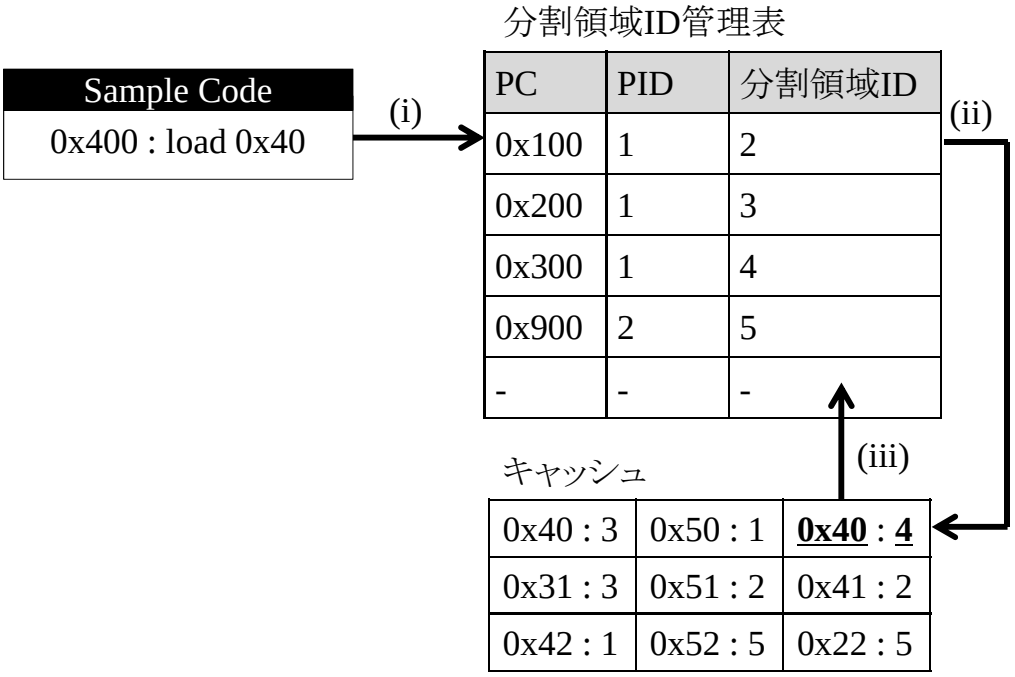


図 20: ロード命令の登録

トリが保持されてしまう．ここで，このような場合にも分割領域を統合すると，分割を作成する前の状態に戻ってしまい，ロード命令単位での分割領域割り当て自体が適用できなくなってしまう．そこで，このようなアクセスが発生した場合，アクセスを発生させたロード命令とエントリが持つ分割領域 ID と関連付ける事で，同一のアドレスに対応するエントリが 1 つのサンプリングモニタのみに存在するよう動作させる．

それでは，このようなロード命令の関連付けについて**図 20** に示す例を用いて説明する．なお，キャッシュ内の値が示す内容は先程と同様であり，サンプルコードを実行するプロセスに対応する分割領域 ID は 1 であるとする．分割領域 ID 管理表とキャッシュが図 20 に示すような状況でサンプルコードのロード命令が実行されると，まず先程と同様に分割領域 ID 管理表が検索される (i)．ここで，先程の例とは異なり当該ロード命令に対応する PC/PID 組が管理表には登録されていない．これに続いて，キャッシュからアドレス 0x40 に対応するエントリが検索されると，このアドレスに対応するエントリが発見される (ii)．この時，このエントリが持つ分割タグの値は 4 となっており，この値はプロセスに対応する分割領域 ID である 1 とは異なっている．そこで，このようなアクセスを発生させたロード命令は分割領域 ID 管理表に登録する (iii)．

また，エントリの持つ分割領域 ID がプロセスに割り当てられた値と一致する場合に

も，これらの分割領域を統合するべきではない．これは，新しく分割領域ID管理表に登録したロード命令によってアクセスされるエントリが，過去に同一のロード命令によってアクセスされていた可能性があるためである．そこで，プロセスに割り当てられているIDを持つエントリがアクセスされた場合には4.2.2項で述べたように，その分割タグを書き換えるのみとする．

4.4 分割領域IDの解放

Vantageにおいて管理可能な分割領域の数は，ハードウェコストの観点から有限である．このため，当該領域に存在するエントリへのアクセスが発生しないような分割領域を維持し続けると，別のロード命令が利用するエントリに分割領域IDを割り当てられなくなる．そこで，このような分割領域に関連付けられているIDを解放し，別のロード命令へ割り当て可能にする．

4.4.1 アクセスの発生しない分割領域の検出

ある分割領域に存在するエントリが殆ど利用されない場合に，その分割領域に関連付けられた分割領域IDを解放するためには，そのような分割領域を検出しなくてはならない．これを実現するため，各分割領域に対して当該分割領域に存在するエントリが利用されているかどうかを検出する2ビットのConfidence Counterを追加する．また，このConfidence Counterを動作させるために，当該領域へのアクセスが発生したかどうかを表す1bitのフラグも追加する．

このConfidence Counterは，初期値を0として動作を開始する．そして，占有サイズを変更するタイミングで，アクセスが発生していたかどうかを表すフラグを参照し，このフラグがセットされていなかった場合，その値をカウントアップする．この一方で，フラグがセットされていた場合には，その値をリセットする．このように動作させると，アクセスが起こっていない期間が連続した場合にはカウンタ値が増大していく．そして，カウンタの値が3になった場合，その分割領域がアクセスの発生しない領域であると判断する．

4.4.2 解放動作

前項で述べた方法で検出されたアクセスの発生しない分割領域や，統合された分割領域IDを解放する場合には，当該IDに対応する分割タグを保持するエントリを全て降格する必要がある．これは，キャッシュ上に当該IDに対応する分割タグを保持するエントリが存在している状況で，分割領域IDを別のロード命令に関連付けた場合，統合する必要の無い分割領域を統合するべきであると判断してしまう可能性があるため

である．そこで，本提案手法では解放する分割領域 ID に対応する分割タグを保持するエントリを，全て降格してから分割領域 ID を利用可能にする．

このような動作を実現するためには，各分割領域が解放動作中である事を示すフラグを追加する．そして，このフラグがセットされている間は当該フラグに対応する分割領域 ID にロード命令を関連付けない．また，統合された分割領域の ID やアクセスが発生しない分割領域に対応するフラグをセットした上で，当該分割領域のサイズを 0 と設定する事で，元来 Vantage が備える動作によりエントリが降格される．そして，解放中を示すフラグがセットされている分割領域 ID に対応する，分割タグを持つエントリ数が 0 になった時に，当該フラグをクリアする．これにより，全エントリの降格が完了した分割領域の分割領域 ID へと，新たにロード命令を関連付け可能にする．このように動作させる事によって，統合する必要の無い分割領域を誤って統合するべきであると判断する事を無くし，安全に分割領域 ID を解放する．

4.5 分割領域サイズの制御

ロード命令単位で分割領域を割り当てた場合，既存の Vantage よりも多くの分割領域を管理する事になる．ここで，このように多くの分割領域を管理する際には，それぞれの分割領域の占有サイズ設定が性能に与える影響が大きくなる．そこで，本節ではこれまで考慮されていなかった分割領域毎の特徴を考慮した上で，占有サイズを決定する手法を提案する．

4.5.1 過剰ブロックの割り当て

既存の Vantage では，各分割領域の占有サイズを決定するために 2.3.2 項で説明した Lookahead アルゴリズムを利用している．この Lookahead アルゴリズムでは，どの分割領域の占有サイズを大きくしてもヒット数を増加させられないと判断した場合，どの分割領域の占有サイズを大きくするかを規定していない．このため，分割領域 ID が最も小さい分割領域の占有サイズのみが大きくなるようブロックが割り当てられる．

しかしながら，ロード命令単位で分割領域を割り当てるような場合，1 つの分割領域に対応する占有サイズのみを大きくするようなブロック割り当ては，性能を著しく低下させる可能性がある．これは，ロード命令がアクセスするアドレスの種類は，プログラム実行に伴って大きく変化する事が多いためである．そこで，本提案手法ではどの分割領域の占有サイズを大きくしてもヒット数が増加しないと判断された場合，各分割領域に対して均等にブロックを割り当てる．ただし，必要とする占有サイズが常に一定であるような分割領域や，キャッシュヒットが殆ど発生しない分割領域に追加

のブロックを与えても性能が向上しない場合が多いため，このような領域には追加のブロックを割り当てないようにする．本提案手法では，以上で述べたキャッシュミスが殆ど発生しない分割領域を検出するために，2 ビットの Confidence Counter を追加する．この Confidence Counter では，直近のサンプリング期間に発生していたキャッシュミス回数が 8 回よりも少なかった場合に，その値をカウントアップする．ここで，8 回以上のキャッシュミスが発生した事を検出するためには，キャッシュミス回数をカウントする 3bit の飽和カウンタを利用する．この一方で，キャッシュヒットが殆ど発生しない分割領域の検出には Confidence Counter を利用せず，直近のサンプリング期間にヒットが発生していたかどうかによってのみ決定する．これは，実行される事が少なくなったロード命令に対応する分割領域のサイズを早期に小さくするためである．

4.5.2 占有サイズの小さい分割領域

ロード命令単位で分割領域を割り当てる場合，各分割領域が必要とするサイズが，既存の Vantage よりも小さくなる可能性が高い．ここで，あるロード命令に対応する分割領域の必要とするサイズが，設定可能な最低サイズよりも小さい場合，そのロード命令が実際に必要としているサイズ以上の領域が与えられる．このため，小さいサイズの分割領域が多数存在する場合，実際には全く必要とされていない不要な領域が，多く確保されてしまう．このように，不要な領域が確保された場合他の分割領域が圧迫され，キャッシュ分割による性能向上が妨げられてしまう可能性がある．また，占有サイズの小さい分割領域が作成されすぎた場合には，別のロード命令への分割領域 ID の関連付けができなくなってしまう．そこで，占有サイズの小さい分割領域同士を統合する事により，これらの問題を回避する．

このような占有サイズの小さい分割領域を検出するために，Confidence Counter を追加する．この Confidence Counter は，サンプリングモニタにおいて小さいウェイでのみヒットが観測される場合に，その値をカウントアップする．これは，小さいウェイでのみヒット数が観測される場合，分割領域のサイズを大きくした場合にもヒット数が増加しないためである．なお，この Confidence Counter のビット数を，4.4.1 項で示した Confidence Counter と同様に 2 ビットとした場合，将来的には大きい占有サイズを必要とするようになる分割領域であっても，誤って占有サイズの小さい分割領域と判断してしまう可能性が高い．そこで，占有サイズの小さい分割領域を検出するための Confidence Counter は 3 ビットとし，検出のための閾値を大きくすることにより，誤った検出を抑制する．また，プロセスに対応する分割領域は，将来的に大きな占有サイズを必要とする可能性が高いため，占有サイズの小さい分割領域として検出しな

い．これに加えて，別々のプロセスのロード命令に対応する分割領域を統合した場合，それらのロード命令が干渉を発生させる可能性が高いため，占有サイズの小さい分割領域の統合はプロセス毎に行う．これを実現するため，プロセスに対応する分割領域は，占有サイズの小さい分割領域を統合している領域の ID を保持する．この一方で，ロード命令に対応する分割領域 ID は，そのロード命令を持つプロセスに対応する分割領域の ID を記憶する．そして，占有サイズの小さい分割領域が検出された場合，当該領域に関連付けられたロード命令を持っているプロセスに対応する分割領域から，占有サイズの小さい分割領域を統合している領域の ID を取得する．そして，取得した分割領域 ID に対応する分割領域へ検出された分割領域を統合する．

4.5.3　必要な占有サイズの予測が困難な分割領域

前項で述べたように，あるロード命令がアクセスするアドレスはプログラムの実行に伴って大きく変化する場合が多い．このため，直近のサンプリング期間に観測したヒット数が，次のサンプリング期間には大きく異なる値になる可能性が高い．これにより，当該ロード命令に適した占有サイズを設定する事ができず，ロード命令単位で分割領域を割り当てる事による性能向上を妨げてしまう可能性がある．

そこで，観測するヒット数が大きく変化するロード命令に関連付けられた分割領域をプロセスに対応する分割領域へ統合する手法を提案する．このような手法を実現するためには，ヒット数が大きく変化するロード命令に関連付けられた分割領域を検出しなくてはならない．しかしながら，観測するヒット数が大きく変化する分割領域を検出するためには，非常に複雑な検出機構が必要になる．そこで，観測するヒット数の大きな変化自体ではなく，観測するヒット数を大きく変化させる可能性が高い分割領域を検出する事により，検出機構を単純にする．このような検出機構を実現するために，様々なプログラムを実行して必要領域のサイズを調査した結果，必要領域サイズが大きく変化する可能性の高い領域には

（I）．一定サイズの領域を与えた場合にキャッシュヒット数が大きく増加する

（II）．キャッシュミス回数が非常に多い

という共通点が存在する事が分かった．そこで，本提案手法では（I）（II）に示す特徴を持つ分割領域を，観測するヒット数が大きく変化する可能性が高い分割領域として検出する．この検出を実現するためには，2bit の Confidence Counter を追加する．この Confidence Counter では，Lookahead アルゴリズムによって一度にキャッシュサイズの 8/256 以上のブロックが与えられ，直近のサンプリング期間におけるキャッシュミス回数が 1024 回を超えていた場合，その値をカウントアップする．ここで，キャッ

表 1: シミュレータ諸元

Processor	
ISA	Alpha
number of cores	1 or 4 cores
issue order	out-of-order
issue width	int:2, fp:2, mem:2
instruction window	int:32, fp:16, mem:16
branch pred	8KB g-share
BTB	2K entries, 4 ways
RAS	8 entries
L1 I&D cache	32 KBytes
ways	4 ways
latency	2 cycle
line size	64 Bytes
L2 cache	0.5 or 2 MBytes
ways	4 ways
latency	8 cycles
line size	64 Bytes
Memory	infinity
latency	200 cycles

シュミス回数が 1024 回を上回っているかどうかを判断するためには，4.5.1 項で述べた 8 回までのキャッシュミスをカウント可能な 3bit の飽和カウンタを 10 ビットに拡張する．

5 評価

3 章及び 4 章で述べた 2 つの提案手法を実装し，シミュレーションによる評価を行った．本章では，これら 2 つの手法及びこれらを組み合わせた手法の性能について考察する．

表 2: SPEC CPU 2006 のベンチマークの分類

Insensitive	400.perlbench, 410.bwaves, 416.gamess, 435.gromacs, 444.namd, 445.gobmk, 447.dealII, 453.povray, 454.calculix, 456.hmmer, 458.sjeng, 464.h264ref, 465.tonto, 481.wrf
Cache-friendly	401.bzip2, 403.gcc, 434.zeusmp, 436.cactusADM, 437.leslie3d, 473.astar
Cache-fitting	450.soplex, 470.lbm, 471.omnetpp, 482.sphinx3, 483.xalancbmk
Thrashing/Streaming	429.mcf, 433.milc, 459.GemsFDTD, 462.libquantum

5.1 評価環境

シミュレータには鬼斬弐 [23] を利用した．この評価に用いたシミュレータ構成を**表 1** に示す．本研究では，2 つの提案手法が各プログラムに対してどのような性能を示すか確認するため，まずは単一コア構成での性能を評価した．ここで，単一コア構成では L2 キャッシュのサイズを 512KB とした．また，複数のプログラムが同時に実行される複雑な状況で，提案した両手法がどのような性能を示すか確認するためマルチコア構成でも評価した．このマルチコア構成ではコア数を 4 とし，L2 キャッシュのサイズを 2MB とした．なお，両提案手法における作成可能分割領域の最大数は 64 個とし，両提案手法は L2 キャッシュにのみ適用した．また，占有サイズの決定頻度は 5M サイクル毎とし，挿入位置の調整頻度は 500K サイクル毎とした．そして，4.2.3 項で述べた検出表において，キャッシュミス頻発命令を検出するための閾値は 2048 回とした．

評価対象のプログラムとしては，SPEC CPU 2006 の全 29 プログラムを選択し，その入力データセットとしては最もサイズの大きい referenced を利用した．単一コア構成では，これらのプログラムから 1 つのプログラムのみを選択し，500M 命令のスキップ後 500M 命令実行した場合の性能を評価した．

この一方で，マルチコア構成では 4 つのプログラムを組み合わせたワークロードを，20G 命令のスキップ後 2G 命令実行した場合の性能を評価した．ここで，ワークロードにおける 4 つのプログラムの組み合わせは，[9] と同様の方法で選択した．この方法では，まず**表 2** に示すように SPEC CPU 2006 の 29 プログラムを，キャッシュサイズによって殆ど性能が変化しない Insensitive，キャッシュサイズを大きくするほど性能が向上するプログラムである Cache-freindly，キャッシュサイズを大きくすると，ある一

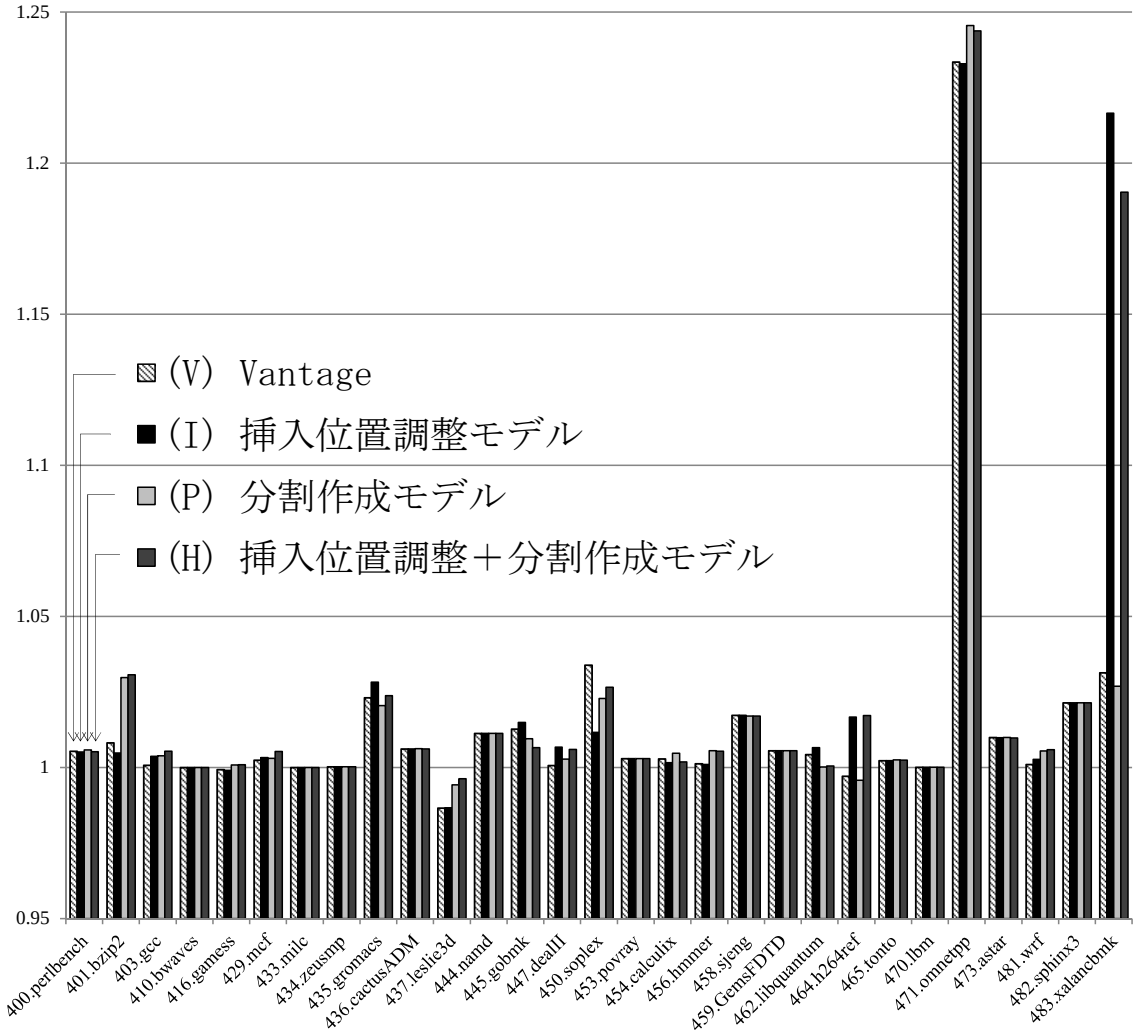


図 21: 単一コア構成における評価結果

定のサイズでミスがほぼ無くなる Cache-fitting，キャッシュの容量を大きくしてもミス数が非常に多い Thrashing/Streaming の 4 種類に分類している．そして，各分類から繰り返しを許して 4 つの分類を選択する組み合わせ 35 種類をクラスとし，各クラスにおいてランダムにプログラムを選択した組み合わせを 10 個ずつ作る事により計 350 個のワークロードを作っている．よって，本論文でもこの方法に順じて評価を行った．

5.2 単一コア構成

評価は Vantage，挿入位置を調整するモデル，ロード命令毎にキャッシュを分割するモデル，挿入位置調整とロード命令毎のキャッシュ分割を組み合わせたモデルについて行った．**図 21** 中で各ベンチマークプログラムの結果を 3 本のグラフで示しており，

それぞれのグラフは左から順に

(V) 既存の Vantage

(I) 挿入位置を調整するモデル

(P) ロード命令毎にキャッシュを分割するモデル

(H) (I) と (P) を組み合わせたモデル

を利用して，各プログラムを実行した際の IPC を示している．また，各 IPC は 16 ウェイのセットアソシアティブキャッシュにおける IPC を 1 として正規化しており，その値の 0.95 以上の部分を示している．なお，16 ウェイのセットアソシアティブキャッシュは，本来 4 ウェイのセットアソシアティブキャッシュよりもルックアップ速度が遅くなるはずであるが，本評価では同一のルックアップ速度として評価した．

まず 471.omnetpp の性能を見ると，全てのモデルにおいてセットアソシアティブキャッシュよりも IPC が大きく向上している．これは，全ての提案モデルが粗粒度 LRU を利用しており，再帰的な追い出し候補の収集により，連想度を向上させる事ができたためである．

続いて，483.xalancbmk の性能を見ると，挿入位置調整モデル (I) が大きく IPC を向上させている．これは，3.1.2 項で示したように 483.xalancbmk が既参照エントリを優先した場合，大きく性能を向上するプログラムであるためである．また，同様な理由から挿入位置調整モデル (I) を利用した場合，403.gcc, 435.gromacs, 445.gobmk, 447.dealII, 464.h264ref, 481.wrf を実行した際の IPC が向上した．ここで，挿入位置調整モデル (I) を利用した場合の挿入位置の割合を**図 22** に示す．この図では，横軸が各プログラムを示しており，縦軸が各プログラムを実行した際に選択された挿入位置の割合を示している．また，凡例は挿入位置が MRU 位置からどの程度の位置であるかを示している．なお，MRU 位置へ挿入をする場合の動作は通常の LRU 方式のキャッシュと同じ動作となる．この図を見ると，挿入位置調整モデル (I) を利用した場合に IPC を向上させた 403.gcc や 483.xalancbmk は非常に高い割合で，MRU 以外の位置に挿入されている．この一方で，410.bwaves, 433.milc, 434.zeusmp, 453.povray, 459.GemsFDTD, 470.lbm では，殆ど挿入位置が調整されていない．このため，これらのプログラムの IPC は図 21 に示すように，殆ど変わらなかったと考えられる．この一方で，401.bzip2, 450.soplex では，多くの割合で挿入位置が調整されているにも関わらず IPC が向上していない．この原因について調査したところ，これらのプログラムでは挿入位置が頻繁に変化していた．このため，最も性能を向上させる挿入位置の予測が外れてしまう事が多く，その性能が低下してしまったと考えられる．

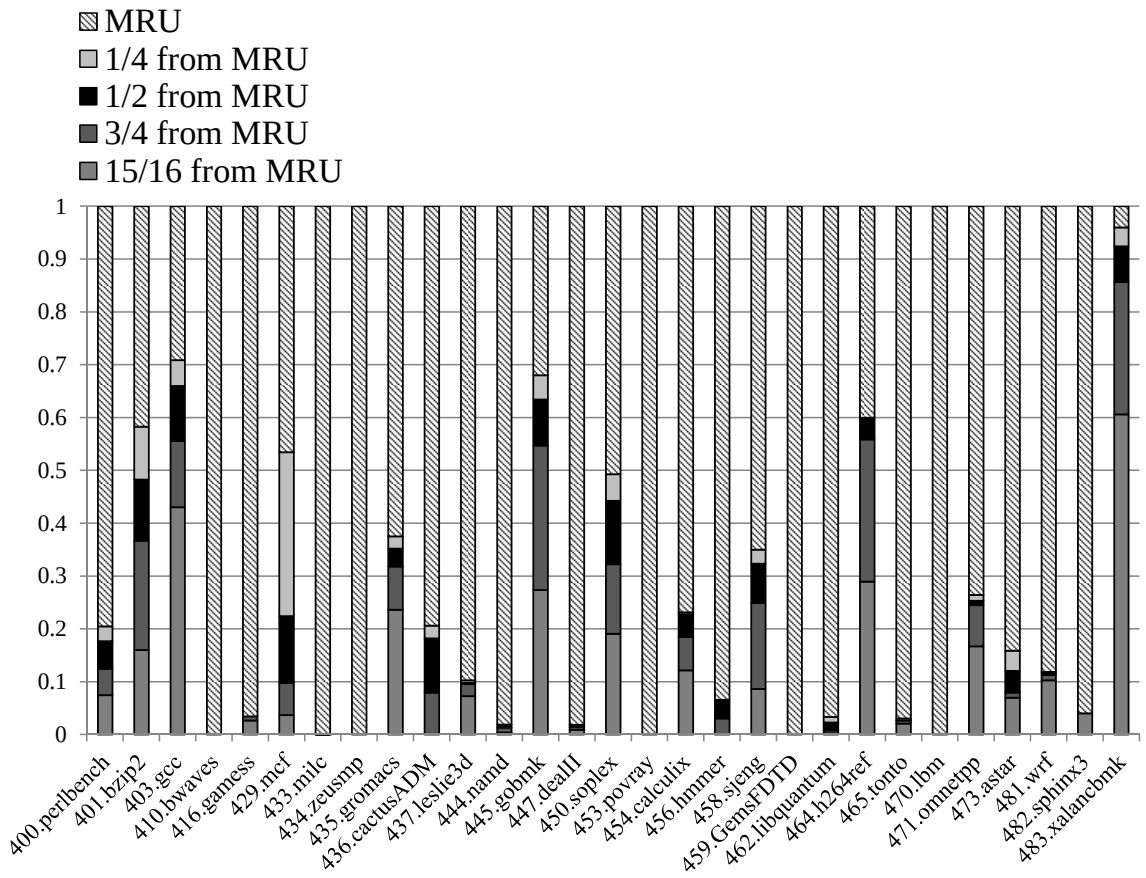


図 22: 挿入位置の割合

次に，分割モデル (P) を利用した場合の性能を見ると，401.bzip2, 429.mcf, 437.leslie3d, 456.hmmer, 481.wrf, 483.xalancbmk の性能が向上している．これは，繰り返し再参照されるエントリを配置する分割領域を作り，そのようなエントリを保護したためである．また，キャッシュミスを頻発するような命令に対応する分割領域を作った事により，そのような命令が要求したエントリによって，有用なエントリが追い出されてしまう状況を抑制する事もできている．

そして，これらの手法を組み合わせたモデル (H) を利用した場合の平均及び最大高速化率は，**表 3** に示すように，2 つの手法 (I)(P) のどちらを利用した場合よりも高くなった．これは，2 つの手法 (I)(P) を組み合わせた事によって，挿入位置調整の適用単位を細かくしたためだと考えられる．

ここで，このような細かい単位での挿入位置調整が性能に与える影響について考察するために，2つの手法を組み合わせた際に性能が向上したプログラムである 483.xalancbmk

表 3: 単一コア構成における IPC 向上率

	Mean	Max
(V) Vantage	3.17%	24.9%
(I) 挿入位置調整モデル	3.80%	25.8%
(P) 分割モデル	3.33%	24.9%
(H) 分割+挿入位置調整モデル	**4.09%**	**26.2%**

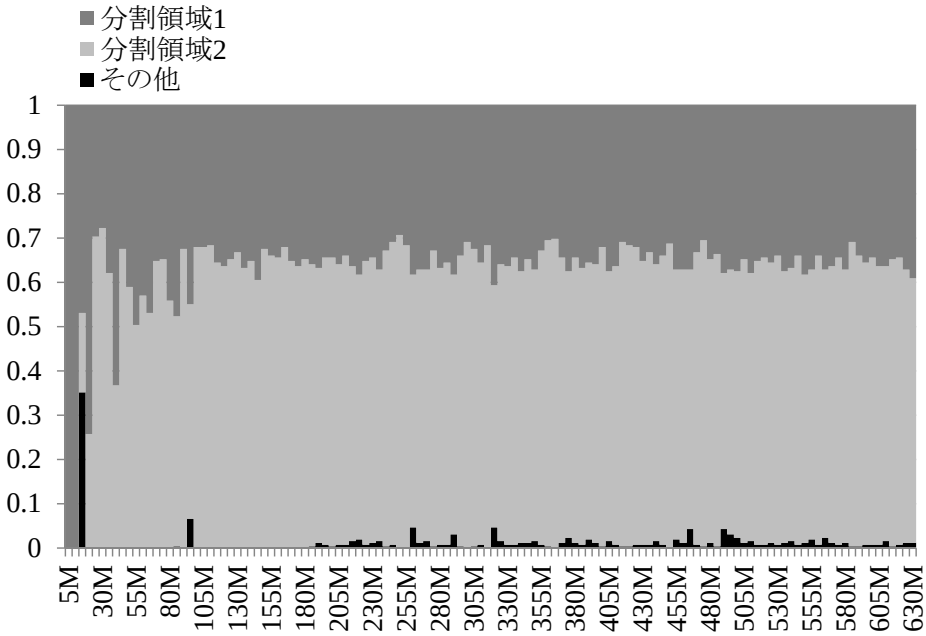


図 23: 483.xalancbmk における占有数

に着目し，その分割領域毎の挿入位置割合を調査した．まず，483.xalancbmk が作る分割領域のサイズについて調べた結果を**図 23** に示す．この図において，横軸は実行サイクルを示しており，その値は 5M サイクル毎に区切られている．なお，このサイクルは実行開始から 630M サイクル目までを示している．また，縦軸は各分割領域サイズのキャッシュサイズに対する割合を示している．そして，凡例は上から順にプロセスに対応する分割領域 ID が 1 である分割領域，ロード命令に関連付けられている分割領域 ID が 2 である分割領域，それ以外の分割領域 ID に関連付けられている分割領域を示している．結果を見てみると，最初の 5M サイクルではキャッシュミスを頻発するロード命令を検出できないため，プロセスに対応する分割領域 ID が 1 である領域のみが存在している．また，次の 5M サイクルではいくつかのロード命令が分割領域 ID

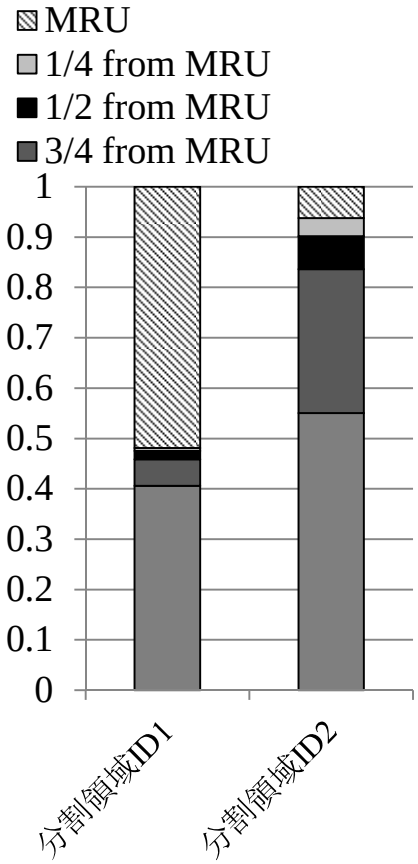


図 24: 483.xalancbmk における挿入位置の割合

に関連付けられ，それらの分割領域の占有サイズが 15M サイクル目から設定されている．そして，15M サイクル以降はキャッシュ領域の殆どが分割領域 ID1 と 2 の分割領域に占められている．このため 483.xalancbmk では，これらの分割領域におけるエントリの追い出し動作が性能に大きな影響を与えていると考えられる．

そこで，483.xalancbmk の性能向上について考察するために，分割領域 ID が 1 と 2 に対応する分割領域における挿入位置の割合を調べた．この結果を**図 24** に示す．この図から，分割領域 ID2 に関連付けられた領域では，先程と同様に挿入位置が調整される割合が非常に多い事がわかる．この一方で，分割領域 ID1 に関連付けられた分割領域では，挿入位置として MRU 位置が選択されている割合が多い．このように，分割領域 ID1 に関連付けられた分割領域では LRU 方式と同じ動作をさせる事で，その性能を最も向上させる事ができる場合が多かった事がわかる．この一方で，挿入位置を調整するモデル (I) では，このように LRU-friendly な挙動をするエントリと，ロード命令に関連付けられたエントリとが区別されずに扱われていたため，有用なエントリ

が追い出されてしまう場合があったと考えられる．以上のような理由から，挿入位置をより細かい単位に適用するモデル (H) を利用する事で，性能を向上させる事ができたと考えられる．

5.3 マルチコア構成

複雑な状況における各モデルの性能を考察するために，マルチコア構成においても 5.2 節と同様に 4 つのモデルの性能を評価した．この評価結果を**図 25** に示す．この図において，各ラインは各ワークロードを実行した際の IPC を示しており，その値は先程と同様に 16 ウェイのセットアソシアティブキャッシュの IPC を 1 として正規化している．また，この図における X 軸は各ワークロードを示しており，これらのワークロードは性能向上率によってソートされている．

まず挿入位置調整モデル (I) では，その性能が平均的に大きく向上している．これは，5.2 節で述べたように，既参照エントリを優先する事によって大きく性能向上させるプログラムが性能向上したためである．また分割モデル (P) では，Vantage(V) がセットアソシアティブキャッシュに対して大きく性能向上させるような場合，その性能向上率が低くなっている．これは，Vantage(V) が干渉を回避する事で性能を向上させており，この干渉による影響は，キャッシュミス発生回数が多いプログラムを実行した際に大きくなる可能性が高く，このような場合には各ロード命令が必要とする分割領域のサイズが不安定になるためである．このようにして，各分割領域が必要とする占有サイズの予測が困難となり，分割モデル (P) の性能が低下してしまったと考えられる．しかしながら，分割モデル (P) では Vantage(V) が LRU に対して大きく性能を向上させる事ができていない場合には，その性能を向上させる事ができている．これは，必要な分割領域サイズを予測困難にするロード命令に分割領域を与える事で，再参照される確率の高いエントリが必要とする分割領域のサイズを安定させる事ができたためである．この結果，分割モデル (P) を利用した場合，前半 231 個のワークロードにおける性能向上率が Vantage(V) よりも向上した．このように，分割モデル (P) では性能向上率の低いワークロードの性能向上率を大きくする事ができた．

最後に 2 つの手法を組み合わせたモデル (H) では，Vantage による性能低下を緩和した上で，その性能を平均的に向上させる事ができている．この結果，各モデルの平均高速化率と最大高速化率は**表 4** に示すような値となり，挿入位置を調整するモデル (I) が最も高い平均高速化率を示し，2 つの手法を組み合わせたモデル (H) が最も高い最大高速化率を示した．

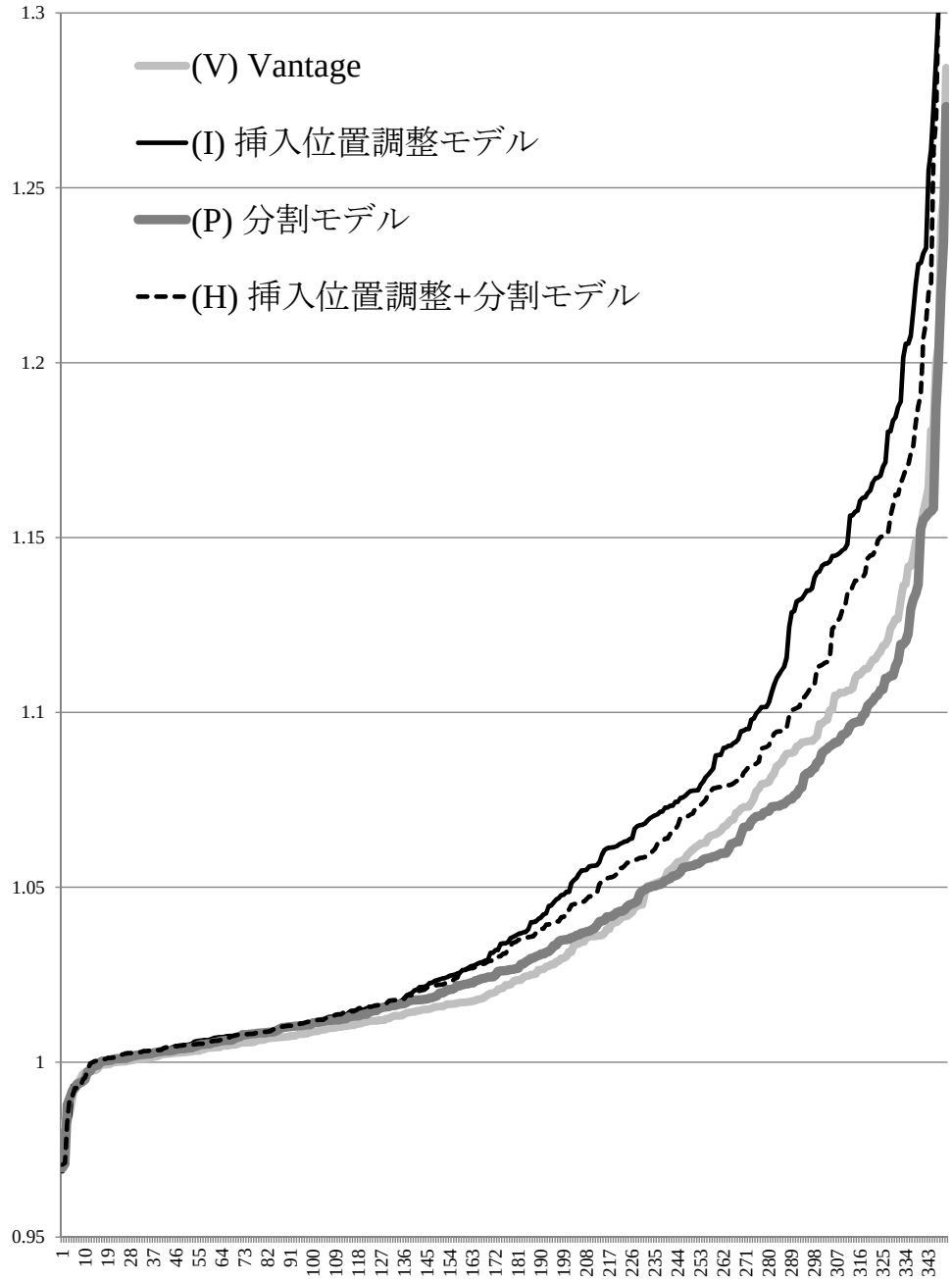


図 25: マルチコア構成における評価結果

5.4 実装コスト

これまで提案してきた手法を実装するために必要なハードウェアコストについて考察する．また，これらの手法は非常に多くのハードウェアを必要とするため，これらを削減する方法についても考察する．

表 4: マルチコア構成における IPC 向上率

	Mean	Max
(V) Vantage	4.12%	28.4%
(I) 挿入位置調整モデル	**5.86%**	**41.5%**
(P) 分割モデル	4.00%	27.3%
(H) 分割+挿入位置調整モデル	5.20%	40.1%

まず，挿入位置調整を実現するためには，3.4 節で述べた Vantage 制御用ハードウェアに加えて，サンプリングモニタを実装するためのハードウェアが必要になる．また，ロード命令毎のキャッシュ分割を実現するためには，サンプリングモニタ以外には主にキャッシュミス頻発命令検出器，分割領域 ID 管理表，統合関係表が必要となる．ここで，今回の評価ではキャッシュミス頻発命令検出器と分割領域 ID 管理表のサイズは無制限としており，理想的な状態での性能を評価した．しかしながら，3.4 節で述べたように，キャッシュミス頻発命令検出表の容量は，性能を大きく落とさず削減する事が可能であると考えられる．また，単一コア構成において，実際に分割領域 ID 管理表へと登録されたロード命令の最大数は 437.leslie3d を実行した場合の 114 個であった．このため，分割領域 ID 管理表で管理すべきロード命令数は多くないと考えられ，性能を落とさずに分割領域 ID 管理表の容量を削減する事は可能であると考えられる．ここで，64bit 環境ではロード命令と PID を記憶するために必要なビット数はそれぞれ 64bit と 22bit 必要である．このため，これらの表に登録可能なロード命令の数を削減し，例えば 128 エントリずつ保持可能とした場合には，それぞれに必要なビット数は約 1.3KB となる．これに加えて，統合関係表を実現するためには作成可能な分割数 ×64bit 必要になる．更に 4.4 節以降で述べた手法を実現するためには，全てを合わせて作成可能な分割数 ×22bit 必要になる．このため，提案手法を実現するために必要な記憶ビット数は，サンプリングモニタを除いた場合，約 2.6KB+ 作成可能な分割数 ×86bit となる．本評価では作成可能な分割数を 64 としたため，この必要ビット数は合計で約 3.3KB となる．

この一方で，サンプリングモニタを Settle らと同様の精度で実装した場合には，モニタのサイズが非常に大きくなり，膨大なハードウェアが必要になってしまう．しかしながら，ヒット数観測の精度低下を許容して，モニタのウェイ数及びライン数を削減した場合には，各モニタのサイズは大きく削減する事ができる．例えば，サンプリ

表 5: サンプリングモニタ用の HW コスト

各タグエントリのサイズ	47 bits
モニタのウェイ数	64
ライン毎の合計サイズ (47bits×64)	376 B
各モニタに必要となる合計サイズ (376B×4)	1504 B
挿入位置調整用モニタのサイズ	1504 B
分割領域用モニタの数	64
合計サイズ (1504B×(64+1))	95 KB

ングモニタのウェイ数を 64 としライン数を 4 とした場合には，**表 5** に示すように，モニタの実装に必要となる合計ビット数は 1504B となる．

また，挿入位置調整用のモニタでは，割り当てられた分割領域のサイズにおけるヒット数のみを観測している．この事から，全ての分割領域に対応する挿入位置調整用モニタを足しあわせたサイズは，LRU 方式用のサンプリングモニタ 1 つと同一である事がわかる．このため，分割領域がどれだけ作られた場合にも，挿入位置調整用モニタの合計サイズはサンプリングモニタ 1 つ分と同一のサイズで実装可能であると考えられる．このようにサンプリングモニタを実装した場合には，ロード命令毎に挿入位置を調整するモデル (H) の実現に必要なモニタの合計サイズは (作成可能な分割数 +4)×1504B となる．このため，作成可能な分割領域数が 64 個である場合には，モニタに必要なビット数は約 95KB となる．以上のような理由から，ロード命令毎に挿入位置を調整するモデル (H) の実現に必要となる合計ビット数は約 98.3KB となるため，その実現は十分に可能であると考えられる．

6 おわりに

本論文では，細粒度なキャッシュ分割に適した追い出しアルゴリズムとして，エントリの挿入位置を動的に調整する手法を提案した．また，この追い出しアルゴリズムをより細かい単位で適用するための手法として，Vantage 上でロード命令単位での分割領域を割り当てを実現し，ロード命令単位で割り当てられた分割領域毎に異なる挿入位置を選択可能にする手法も提案した．

これらの提案手法の有効性を確認するため，SPEC CPU 2006 ベンチマークを用いて評価を行った．この結果，単一コア構成において従来モデルでは最大 24.9%，平均

3.17%であったIPC向上率を，挿入位置調整モデルを利用する事で，最大25.8%，平均3.80%とする事が確認できた．また，ロード命令単位で分割領域を割り当てるモデルではIPC向上率を最大24.9%，平均3.33%まで向上し，これらを組み合わせたモデルでは最大26.2%，平均4.09%まで向上することを確認した．これにより，提案手法の有効性を確認でき，細かい粒度で追い出しアルゴリズムを変更する手法の有用性を示す事ができた．また，これまで理想的な状況で考えられていた，ロード命令単位での分割領域割り当て手法を実現するためのハードウェアコストについて考察し，その実現可能性を示す事もできた．

本研究の今後の課題としては，短期的にはハードウェアコストの削減が挙げられる．まず，キャッシュミス頻発命令検出器と分割領域ID管理表では，全てのPCを登録する事ができると想定しているため，これらの管理表のライン数に制限を設けた場合の性能を評価する必要がある．また，ロード命令毎のキャッシュ分割を適用した場合，必要となるサンプリングモニタのハードウェアコストが非常に大きくなる．そこで，このサンプリングモニタのハードウェアコストを削減した場合の性能についても評価するべきである．

また，長期的な目標としてはプログラム中のQoSを保証したい処理を指定する事で，分割領域のサイズを自動的に調整する事が可能なモデルを作る事が挙げられる．このような調整は，ロード命令単位で分割領域を割り当てる手法を利用する事によって実現可能であると考えている．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室および松井研究室の方々に感謝致します．

著者発表論文

論文

1. Ryosuke ODA, Tatsuhiro YAMADA, Tomoki IKEGAYA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “Input Entry Integration for an Auto-Memoization Processor”, Proc. 3rd Workshop on Ultra Performance and Dependable Acceleration Systems (UPDAS), pp.179–185 (2011).

2. Kazutaka KAMIMURA, Ryosuke ODA, Tatsuhiro YAMADA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “A Speed-Up Technique for an Auto-Memoization Processor by Reusing Partial Results of Instruction Region”, Proc. 3rd Int’l. Conf. on Networking and Computing (ICNC), pp.49–57 (2012).
3. Tomoki IKEGAYA, Ryosuke ODA, Tatsuhiro YAMADA, Tomoaki TSUMURA, Hiroshi MATSUO, Yasuhiko NAKASHIMA: “A Hybrid Model of Speculative Execution and Scout Threading for Auto-Memoization Processor”, Proc. of Int’l. Symp. on System-on-Chip (SoC), pp.22–28 (2011).

報文

1. 小田 遼亮, 山田 龍寛, 池谷 友基, 津邑 公暁, 松尾 啓志, 中島 康彦: “自動メモ化プロセッサの入力値エントリ統合による高速化”, 情処研報 (SWoPP2011), Vol.2011-ARC-196, No.2, pp.1–10 (2011).
2. 山田 龍寛, 小田 遼亮, 池谷 友基, 津邑 公暁, 松尾 啓志, 中島 康彦: “命令区間の特徴を用いた自動メモ化プロセッサの再利用率向上手法”, 情処研報 (SWoPP2011), Vol.2011-ARC-196, No.1, pp.1–7 (2011).
3. 神村 和敬, 山田 龍寛, 小田 遼亮, 津邑 公暁, 松尾 啓志, 中島 康彦: “再利用対象区間の細分化による自動メモ化プロセッサの高速化”, 情処研報 (SWoPP2012), Vol.2012-ARC-201, No.16, pp.1–8 (2012).

参考文献

[1] Seznec, A.: A case for two-way skewed-associative caches, *Proc. 20th Annual Annual Int’l Symp. on Computer Architecture (ISCA)*, ACM, pp. 169–178 (1993).

[2] Qureshi, M. K., Thompshon, D. and Patt, Y. N.: The V-way Cache: Demand Based Associativity via Global Replacement, *Proc. 32nd Annual Int’l Symp. on Computer Architecture (ISCA)*, ACM, pp. 544–555 (2005).

[3] D.Lee, J.Choi, J.H.Kim, S.H.Now, S.L.Min, Y.Cho and C.S.Kim: LRFU:A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies, *IEEE Trans. on Computers*, Vol. 50, No. 12, pp. 1352–1361 (2001).

[4] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C. and Emer, J.: Adaptive insertion policies for hight performance caching, *Proc. 34th Annual Int’l Symp. on Computer Architecture (ISCA)*, ACM, pp. 381–391 (2007).

[5] Jaleel, A., Theobald, K. B., Steely, S. C. and Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP), *Proc. 37th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 60–71 (2010).

[6] Khan, S. M., Wang, Z. and Jimenez, D. A.: Decoupled dynamic cache segmentation, *Proc. 18th IEEE Symp. on High Performance Computer Architecture (HPCA)*, IEEE Computer Society, pp. 1–12 (2012).

[7] Collins, J. D., Z, H. W., Tullsen, D. M., Hughes, C., fong Lee, Y., Lavery, D. and Shen, J. P.: Speculative Precomputation:Long-range Prefetching of Delinquent Loads, *Proc. 28th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 14–25 (2001).

[8] Lai, A.-C., Fide, C. and Falsafi, B.: Dead-block prediction and dead-block correlating prefetchers, *Proc. 28th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 144–154 (2001).

[9] Sanchez, D. and Kozyrakis, C.: Vantage: Scalable and Efficient Fine-Grain Cache Partitioning, *Proc. 38th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 57–68 (2011).

[10] Belady, L. A.: A study of replacement algorithms for a virtual-storage computer, *IBM Systems Journal*, Vol. 5, No. 2, pp. 78–101 (1966).

[11] Lai, A.-C. and Falsafi, B.: Selective, Accurate and Timely Self-Invalidation Using Last-Touch Prediction, *Proc. 27th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 139–148 (2000).

[12] Khan, S. M., Tian, Y. and Jimenez, D. A.: Sampling Dead Block Prediction for Last-Level Caches, *Proc. 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, IEEE Computer Society, pp. 175–186 (2010).

[13] Agrawal, A. and Pudar, S. D.: Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches, *Proc. 20th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 179–193 (1993).

[14] Chiou, D., Jain, P., Rudolph, L. and Devadas, S.: Application-specific memory management for embedded systems using software-controlled caches, *Proc. 37th Annual Design Automation Conf. (DAC)*, ACM, pp. 416–419 (2000).

[15] Ranganathan, P., Adve, S. and Jouppi, N. P.: Reconfigurable caches and their application to media processing, *Proc. 27th Annual Int'l Symp. on Computer*

Architecture (ISCA), ACM, pp. 214–224 (2000).

[16] Dybdahl, H., Stenstrom, P. and Natvig, L.: A cache-partitioning aware replacement for chip multiprocessors, *Proc. 13th Int'l Conf. on High Performance Computing (HiPC)*, Springer-Verlag, pp. 22–34 (2006).

[17] Settle, A., Connors, D., Gibert, E. and González, A.: A dynamically reconfigurable cache for multithreaded processors, *Journal of Embedded Computing - Issues in embedded single-chip multicore architectures*, Vol. 2, No. 2, pp. 221–233 (2006).

[18] Qureshi, M. K. and Patt, Y. N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *Proc. 39th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, IEEE Computer Society, pp. 423–432 (2006).

[19] Sanchez, D. and Kozyrakis, C.: The ZCache: Decoupling Ways and Associativity, *Proc. 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, ACM, pp. 187–198 (2010).

[20] 小川周吾, 入江英嗣, 平木敬: 部分的試行に基づく動的キャッシュ分割方式, 情報処理学会論文誌：コンピューティングシステム, Vol. 2, No. 3, pp. 1–11 (2009).

[21] 浅見公輔, 倉田成己, 塩谷亮太, 三輪忍, 五島正裕, 坂井修一: 命令グループごとのキャッシュ・パーティショニングの予備評価, 情報処理学会研究報告 2012-ARC-201, No. 5, pp. 1–8 (2012).

[22] 堀部悠平, 三輪忍, 塩谷亮太, 五島正裕, 中條拓伯: 選択的キャッシュ・アロケーション:マルチスレッド環境におけるキャッシュ利用効率の向上手法, 情報処理学会研究報告 2010-ARC-190, No. 1, pp. 1–8 (2010).

[23] 塩谷亮太, 五島正裕, 坂井修一: プロセッサ・シミュレータ「鬼斬弐」の設計と実装, 先進的計算基盤システムシンポジウム (SACSIS), pp. 120–121 (2009).