修士論文

プリフェッチの性能を引き出すキャッシュ置換アルゴリズム

The Cache Replacement Policy for Prefetch Performance

平成30年01月31日提出

指導教員

坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-166419　甲地 弘幸

概要

プロセッサの発展は，現代の情報化社会を支える上で必要不可欠である．プロセッサの構成要素の 1 つとして，キャッシュ・メモリがある．キャッシュ・メモリは，メイン・メモリへのアクセス・レイテンシを隠蔽するものであるが，置換アルゴリズムによって性能が変わってくる．キャッシュ・メモリの置換アルゴリズムとしては，LRU を用いるのが一般的であるが，近年の大容量の Last Level Cache(LLC) などでは，ハードウェア的に実現が困難であったり，そもそも性能を引き出すことができなくっている．また，キャッシュ・メモリの性能を引き出す技術としてプリフェッチがあるが，置換アルゴリズムが，プリフェッチを考慮しない場合，性能が低下してしまうことがある．そのため，LLC 向けの，プリフェッチを考慮した置換アルゴリズムを考える必要がある．

我々は，プリフェッチを考慮した置換アルゴリズムとして，プリフェッチ・ラインへのデマンド・アクセスとプリフェッチ・アクセスでの汎用的な動作を提案する．今回の提案を，THE 2nd CACHE REPLACEMENT CHAMPIONSHIP にて提案された最新手法である SHiP++ と Hawkeye に適用させて性能を評価する．また，プリフェッチを有効活用する手法である PrePromotion を適用させることで更なる性能向上を図った．

提案により，プリフェッチありの LRU に対して，Instructions Per Cycle(IPC) が，SHiP++ に適用したもので 3.37%，Hawkeye に適用したもので 3.02% の性能向上を達成した．

目 次

図 目 次

表 目 次

第1章　はじめに

半導体技術の進歩により，現在のプロセッサの動作速度は非常に高速なものとなっている．プロセッサの速度向上を支える技術のひとつとしてキャッシュ・メモリが存在する．一般的に，プロセッサの動作速度に比べて，DRAM であるメイン・メモリへのアクセス・レイテンシは数十倍から数百倍ほどである．プロセッサとメイン・メモリの間に階層的にキャッシュを配置することで，この速度のギャップを隠蔽することが可能となる．

キャッシュの容量は，メイン・メモリよりも小さいため，プログラムで用いるデータを全て格納できないことがある．そのため，必要に応じて，キャッシュからキャッシュ・ラインを追い出す場合がある．このとき，キャッシュから追い出すラインを決定するアルゴリズムのことを置換アルゴリズムという．置換アルゴリズムは，最も古いラインを追い出す Least Reently Used(LRU) [2] が最も一般的である．メイン・メモリが大きくなるに従い，キャッシュのサイズも大きくなり，また，現在ではキャッシュを，L1 Cache，L2 Cache，Last Level Cache(LLC) の三層構造にすることもある．そのため，LLC は，上位のキャッシュに時間的局所性や空間的局所性が吸収されてしまい，LRU では性能を大きく引き出すことが出来ない [3]．また，LRU は連想度が大きいキャッシュでは実現が困難である．よって，LLC の性能を引き出すためには，LLC 向けの置換アルゴリズムを開発する必要がある [4].

Jaleel らは，LLC 向けの置換アルゴリズムとして Re-Reference Interval Prediction(RRIP) [5] を提案した．RRIP は，連想度に依存しない任意の bit をキャッシュ・ラインに付加することでラインごとの優先順位を決定することが可能であり，また，複数の置換アルゴリズムを切り替える Set Dueling Monitor [6] を用いることで，複数のアクセス・パターンを含むプログラムでも性能を引き出すことに成功した．Wu らは，特徴量ごとに使用されるラインかどうかを学習し，使用される可能性が高いラインをキャッシュに残すことで，SDM を用いた RRIP に勝る置換アルゴリズムである Signature-based Hit Predictor(SHiP) [7] を提案した．

キャッシュの性能を引き出す技術として，置換アルゴリズムでは救うことが不可能な初期参照ミスを減らすことが可能なプリフェッチがある．置換アルゴリズムは，

今までアクセスがあったラインの情報を用いて，過去の繰り返しが起こることを学習するが，プリフェッチは，過去の情報から未来のアクセスを予測し，あらかじめキャッシュに挿入する技術である．そのため，プリフェッチは，今までアクセスのないラインをキャッシュに挿入することが可能となる．プリフェッチは，簡単な予測でもキャッシュの性能を大きく引き上げることが可能であり，現在のアクセスから次のラインをプリフェッチするシーケンシャル・プリフェッチャ[8]，連続する領域をプリフェッチするストリーム・プリフェッチャ[9] や同じアドレスの飛び幅だけずらしたラインをプリフェッチするストライド・プリフェッチャ[10] などから，複雑な学習機構を持つものまで存在する [11, 12].

置換アルゴリズムも，プリフェッチもどちらもキャッシュ・メモリを支える重要な技術であるが，プリフェッチは，アクセスの来ないラインをキャッシュに挿入することがある．RRIP や SHiP は，デマンド・アクセスと プリフェッチ・アクセスを区別しないため，プリフェッチ存在下では，性能が下がってしまうことがある．そのため，置換アルゴリズム側は，キャッシュの性能を引き出すために，プロセッサからのロード/ストア・アクセス（デマンド・アクセス）とプリフェッチ・アクセスを区別することで，プリフェッチ由来のラインの価値を正しく見積もり，プリフェッチによってキャッシュが不要なラインで満たされることを防ぐ必要がある．プリフェッチを考慮した置換アルゴリズムは，様々提案されている [13–15] が，Wu らは，プリフェッチを意識した置換アルゴリズムとして Parefetch-Aware Cache Management(PACMan) [16] を提案した．PACMan は，プリフェッチ・ラインの特性を調査し，RRIP をベースとして，デマンド・アクセスとプリフェッチ・アクセスにおける動作を変更した手法である．PAMan は，プリフェッチ・ラインはなるべくすぐ追い出そうとする手法であり，デマンド・アクセスに対してプリフェッチ・アクセスの価値を低く見積もっているが，同様な考えの手法も提案されている [17, 18]．一方，プリフェッチの情報を有効利用しようとする手法として，PrePromotion [19–21] がある．PrePromotion では，プリフェッチャを参照予測器と考え，プリフェッチ・アクセスが来たラインがキャッシュに存在していた場合には，追い出されないように保護する．これにより，参照が近づいているラインが追い出されなくなり，性能を向上させることに成功した．

我々は，プリフェッチにより挿入されたラインの扱いや，プリフェッチ・アクセスの傾向に着目し，プリフェッチを考慮した置換アルゴリズムを提案する．今回の提案は主に 2 つあり，プリフェッチで挿入されたラインへの最初のデマンド・アクセスは，デマンドによる挿入と考え，そこで挿入の処理を行うものと，何度もプリフェッチ・アクセスが来るラインは，デマンド・アクセスが来る直前にもプリフェッチ・アク

セスが来ると考え追い出されやすい位置に挿入するものである．前者は，任意のプリフェッチを考慮しない置換アルゴリズムに適用可能であり，後者は，Stubborn [22] や，Hawkeye [1,23] のようなキャッシュの一部を追い出されにくい領域にする手法に適用可能である．そのため，最新手法である SHiP++ [24] に前者の手法を，Hawkeye に後者の手法を適用させ，性能の計測を行った．

提案手法により，プリフェッチ存在下の LRU に対し，Instruction Per Cycle(IPC) が SHiP++ に適用させた場合で，3.37% 上昇し，Hawkeye に適用させた場合で，3.02% 向上し，SHiP++(2.83%) や，Hawkeye(2.66%) にも勝る性能を達成した．

以降，第 2 章，第 3 章，第 4 章では，関連する置換アルゴリズムといくつかのプリフェッチャについて述べ，第 5 章では，提案手法について述べる．第 6 章では，提案と既存手法との性能を評価し，第 7 章でが，この論文をまとめと今後の課題について述べる．

第2章　置換アルゴリズムとプリフェッチ

2.1　置換アルゴリズム

2.1.1　ラインの置換

キャッシュは，メイン・メモリに比べると高速である代わりに小容量となっている．そのため，キャッシュからラインを追い出す必要が生じる場合がある．ラインの追い出しが発生するのは，キャッシュ・セットが既にラインで満たされている場合 (競合性ミス) と，プログラムでアクセスしたいメモリ範囲がキャッシュの容量を越す場合 (容量性ミス) である．これらのミスが発生した場合は，メイン・メモリから必要なラインを挿入し，不必要なラインをキャッシュから追い出すことになる．この動作をラインの置換と呼び，置換アルゴリズムによって追い出すラインが決定される．キャッシュ・ミスを減らすには，理想的には次に参照される間隔 (再参照間隔) が最も長いラインを追い出せばよい OPT [25] が知られている．しかし，これを実現するためには未来の参照に関する情報が必要となり，一度実行したプログラムのアクセス履歴を用いない限り不可能である．そのため，OPT の近似となる置換アルゴリズムを用いられている．この章では，基本的な置換アルゴリズムである Least Recently Used(LRU) [2] と，現在多くの置換アルゴリズムが採用する Re-Reference Interval Prediction [5] について説明する．

2.1.2　LRU

LRU は，現在最もよく使われている置換アルゴリズムの 1 つで，直近に参照されたラインは再参照されやすいという時間的局所性を考慮したものである．LRU は，最終アクセスが最も古いラインを優先的にキャッシュから追い出すアルゴリズムであり，挿入するラインや参照のあったラインを，再参照される可能性が高いラインとして，キャッシュから追い出されにくくする．また，このような最近アクセスの

あったラインを Most Recently Used(MRU) ラインと呼ぶ．例えば，2 ウェイ・セット・アソシアティブ型キャッシュの場合，各ライン毎に 1 ビットのカウンタを持ち，アクセスのあったほうのビットを 0 にし，なかったほうのビットを 1 にすることで，LRU を実現できる．ただし，厳密に LRU を実現するには，連想度分の履歴を保持しなければならないので，連想度が上がるにつれハードウェア的に実装が難しくなる．そのため，連想度が大きい場合には，ランダム置換や，LRU の近似的アルゴリズムを使うことがある [3]．また，キャッシュの容量が大きくなるにつれ，ランダムに置換しても LRU とのキャッシュ・ミス率があまり変わらなくなってくる [3] ので，大容量キャッシュでは，ハードウェア的に実装の容易なランダム置換を用いることもある [26].

しかし，LRU は，ストリーミング・アクセスや，スラッシング・アクセスのような，一度参照がきた後に次の参照が来ない，または，来ても長期間後になるような時間的局所性に乏しいアクセスが来ると，再参照間隔が長いラインを再参照間隔が短いラインとしてキャッシュに保持してしまい，キャッシュを汚染してしまうという問題がある [5].

2.1.3 RRIP

RRIP [5] は，LRU の問題点を改善する目的で考案されたものである．RRIP では，過去のアクセスから再参照間隔を予測し再参照間隔の長いものを追い出すことで，再参照間隔の長いラインによってキャッシュが汚染されるのを防ぐことができる．RRIP には，ストリーミング・アクセスに耐性を持つ Static RRIP(SRRIP) と，SRRIP を発展させた，ストリーミング・アクセスとスラッシング・アクセスのどちらにも耐性の持つ Dynamic RRIP(DRRIP) がある．

SRRIP は，ストリーミング・アクセスなどの一度参照したラインを再参照しないようなアクセスに耐性を持つために考案された．Re-Reference Prediction Values(RRPV) という，M ビットのカウンタを各キャッシュラインに付加し，$\mathrm{RRPV} = 0$ であるラインは再参照間隔がセット内で最も短いラインとし，$\mathrm{RRPV} = 2^{\mathrm{M}} - 1$ であるラインは再参照間隔がセット内で最も長いラインとしている．動作としては，セット・アソシアティブ型キャッシュにおいてキャッシュ・ミスが発生した時に，セットの中から $\mathrm{RRPV} = 2^{\mathrm{M}} - 1$ のものをあるラインから探し，見つけたら $\mathrm{RRPV} = 2^{\mathrm{M}} - 2$ として置き換える．見つからなかった場合，セット内のライン全ての RRPV をインクリメントし，再度同じ作業を $\mathrm{RRPV} = 2^{\mathrm{M}} - 1$ のラインが見つかるまで繰り返す．

キャッシュに存在するラインに参照が来た (ヒットした) 場合，RRPV = 0 とする．

挿入時の再参照間隔を RRPV = $2^M - 2$ と，ひとまず長めに見積もり，キャッシュから追い出されやすくしておく．そのラインがキャッシュから追い出される前に再参照が来た場合は再参照間隔が短いと予測し，RRPV = 0 とするので，再参照間隔が無限大であるストリーミング・アクセスに耐性を持つことができる．例えば，ストリーミング・アクセスから抜けた時に，ストリーミング・アクセス以前からキャッシュ存在していたラインへのアクセスがあった場合，ストリーミング・アクセス中は，再参照間隔が無限大であるストリーミング・アクセスで用いたラインが追い出されるので，キャッシュ・ミスが起こらなくなる．ただし，ストリーミング・アクセスがくる直前に RRPV = $2^M - 2, 2^M - 1$ であるラインは保持されない場合がある．LRU と比べると，挿入時の再参照間隔を LRU のように常に最短と予測せず，一度再参照間隔予測するための学習期間があるため，ストリーミング・アクセスでキャッシュが汚染されにくくなっている．また，ラインあたりに必要なコストが，LRU の場合は，セット内のラインの参照履歴が全て必要なため，ウェイ数の増加に伴って指数的に必要な情報量が増えていくが，RRIP の場合は，任意のビットを各ラインに付加するだけなので，ウェイ数の大きいキャッシュでは，LRU に比べて RRIP のコストは小さくなる．

M = 2 の時の動作を図 2.1 に示す．図では，左から右に向かって各ラインの RRPV の値を調べている．一番上の図は，ライン E を挿入する時に，キャッシュの中で左から順に RRPV を調べて，RRPV = 3 である B と置き換えている．真ん中の図は，ライン E を挿入する時に，キャッシュの中で左から順に RRPV を調べて，RRPV = 3 であるラインが存在しないので，全てのラインの RRPV をインクリメントし，再度左から順に調べて，RRPV = 3 である A と置き換えている．一番下の図は，ライン C を挿入する時に，既にキャッシュに C が存在するので，RRPV = 0 としている．

しかし，SRRIP はストリーミング・アクセスに耐性を持つが，ループを何度もするようなスラッシング・アクセスをするプログラムに対して，ラインの置換が何度も行われ (スラッシング) ミスが頻発してしまうという欠点がある．

そこで，スラッシング・アクセスに耐性を持つために提案されたアルゴリズムが DRRIP である．DRRIP では，ストリーミング・アクセスに耐性を持つ SRRIP と，スラッシングに耐性を持つ Bimodal RRIP(BRRIP) [5] を Set Dueling Monitor(SDM) [6] を用いて，動的に選択することで実現している．ここで，BRRIP とは，SRRIP を基本にして，新しいラインを挿入する時に大抵は再参照間隔を最も長いラインとして，RRPV = $2^M - 1$ で挿入するが，たまに再参照間隔が中程度なラインとして，

$\mathrm{RRPV} = 2^{\mathrm{M}} - 2$ で挿入している．こうすることで，ループ中に $\mathrm{RRPV} = 2^{\mathrm{M}} - 2$ として持ってきたラインをキャッシュに蓄えることができ，次のイテレーションでの再参照時にキャッシュ内に保持できるようになるため，スラッシングに耐性を持つことができる．また，SDM とは，特定の置換アルゴリズムでのみ動くセットを複数用意し，キャッシュ・ミス率を評価し，結果の良い置換アルゴリズムを選択するという方法である．

2.1.4 Stubborn キャッシュ

LRU や，RRIP は，比較的短い間隔で参照が来るラインをキャッシュに保持し，ヒット率を増加させる手法であるが，Stubborn キャッシュは，参照間隔が長い，長期再参照でのミスを減らす手法である [22]．この手法は，従来の置換アルゴリズムの考えと大きく異なり，キャッシュ内に追い出しを発生させない領域を持つ．LLC は，LRU などで動作させると 90% 近くがもう利用されないライン，すなわち，デッド・ブロックである [4]．そのため，ラインを追い出さない領域を持っても，従来手法では救うことのできない長期債参照ミスを減らすことで，キャッシュの性能を向上させることが可能となっている．

2.2 プリフェッチ

2.2.1 プリフェッチの概念

キャッシュ・ミスを減らし，キャッシュの性能を上げる方法として，プリフェッチがある．プリフェッチは，キャッシュの参照履歴から未来の参照を予測し，実際に参照が来る前にキャッシュに挿入するという技術である．プリフェッチは，ストリーミング・アクセスや，スラッシング・アクセスのような再参照されない，または，再参照間隔が非常に長いアクセス・パターンに対して有効であり，置換アルゴリズムのみでは減らせないキャッシュ・ミスを減らすことができる．例えば，メモリ・アクセスの空間的局所性に着目し，あるラインへのアクセスに付随して，その付近のアドレスを持つラインも同時にキャッシュに挿入することでストリーミング・アクセスに耐性を持つことができる [8]．プリフェッチ・アクセスは，ロード命令や，ストア命令などのメモリ・アクセス (デマンド・アクセス) によるアクセスと同様に，目的とするラインがアクセスしているキャッシュに存在しなかった場合に，さらに下層

のメモリにアクセスし，見つかったところから上のメモリ全てにラインの挿入を行う [8]．ただし，デマンド・アクセスの場合は L1 キャッシュにまでラインを挿入するが，プリフェッチの場合，ラインの挿入はプリフェッチャを接続しているキャッシュまでとなる．また，異なるプリフェッチ同士は，どのラインをプリフェッチするか，いつプリフェッチをかけるか，そして，どこに挿入するかによって区別される [10]．この章では，主要なプリフェッチである，A.J.Smith の提案したストリーミング・プリフェッチ [8] と，J.W.C.Fu らが提案したストライド・プリフェッチ [10] について説明する．

2.2.2 ストリーミング・プリフェッチ

ストリーミング・プリフェッチとは，あるラインへのアクセスに付随して，そのラインのアドレスから連続する範囲にあるラインをキャッシュへ挿入するプリフェッチである．これにより，ストリーミング・アクセス，スラッシング・アクセスのどちらに対してもキャッシュ・ミスを減らすことができる．また，プリフェッチをかけてキャッシュに挿入するまでの時間を考慮して，連続したアドレスのうち，現在アクセスしているアドレスに近いものから順にいくつか飛ばして，そこからいくつかを実際のプリフェッチに用いる．このとき，飛ばすライン数を Distance，プリフェッチするライン数を Degree と呼ぶ．Distance と Degree に関する図を，図 2.2 に示す．本論文にて扱うストリーミング・プリフェッチャは，キャッシュ・ミスが発生したときにプリフェッチをかけ，各置換アルゴリズムがデマンドによって挿入する位置にプリフェッチされたラインを挿入するものとする．

2.2.3 ストライド・プリフェッチ

ストライド・プリフェッチとは，あるラインへのアクセスに付随して，Stride Prediction Table(SPT) [10] を用いて，現在アクセスしているアドレスと，直近にアクセスしたアドレスとの差 (ストライド) を計算し，その差が 0 でない時，その分だけ進んだアドレスをプリフェッチするというものである．これにより，2 次元配列へのアクセスが続くプログラムでのキャッシュ・ミスを減らすことができる．また，プリフェッチをより正確にするために，過去のストライドの値を管理する新たなフィールドを SPT に追加することで，現在のストライドの値と，過去のストライドの値を比較し，異なる場合にはプリフェッチをしないようにしている．これにより，異な

るループに入ったときや，ランダムなストライドになるときにプリフェッチしなくなるので，プリフェッチがより正確なものになる．

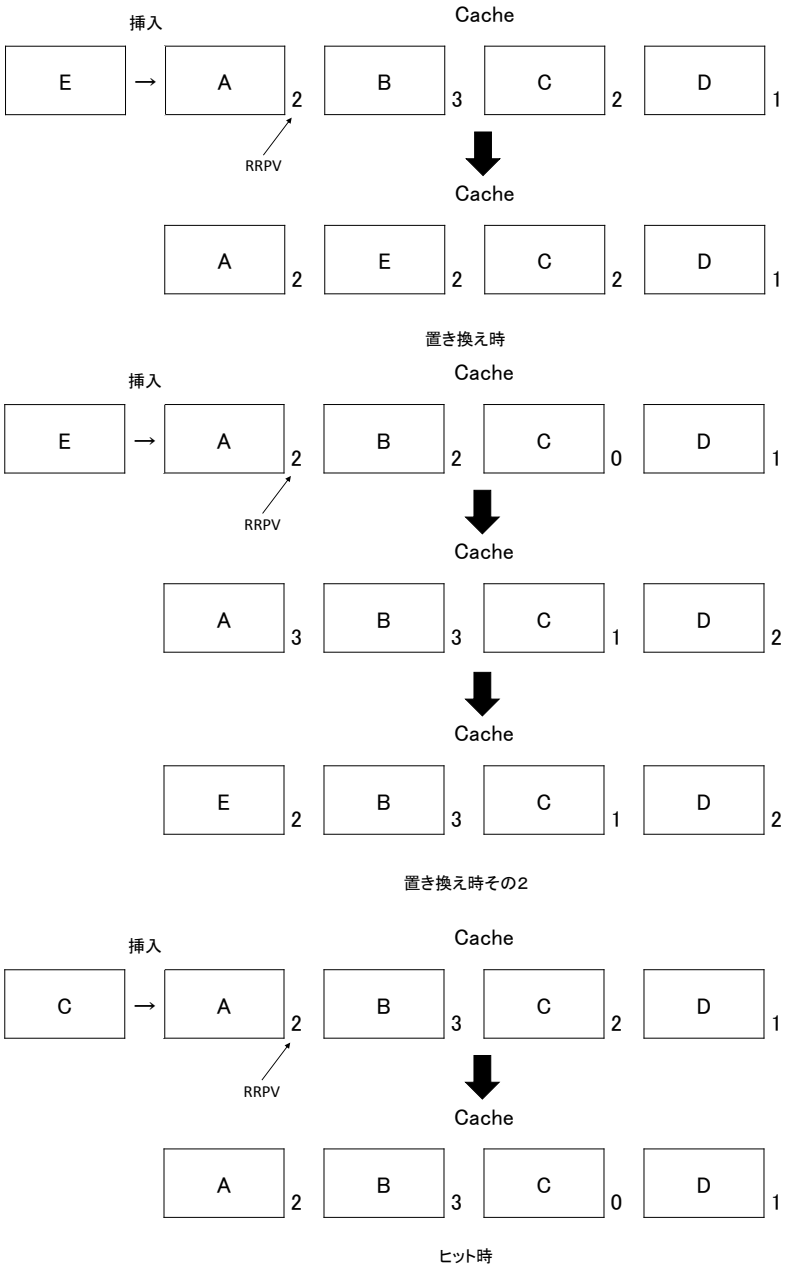

図 2.1: SRRIP の動作

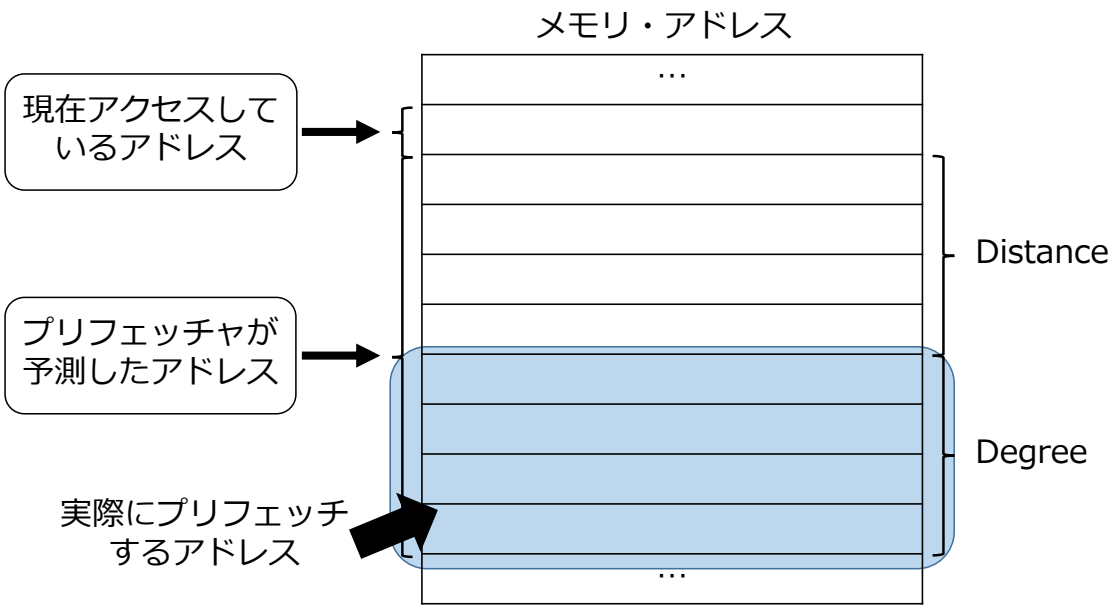

図 2.2: Distance と Degree

第3章　プリフェッチを考慮した置換アルゴリズム

3.1　プリフェッチと置換アルゴリズムの組み合わせ

2.1 や，2.2 で述べた 2 つの技術は，互いが互いの弱点を補うものとして同時にキャッシュに適応することが考えられた．石井らは，プリフェッチされたラインを再参照のこないライン (デッド・ブロック) と仮定して，それらのラインをキャッシュから積極的に追い出すようなキャッシュ・マネージメントとして Cache Replacement based on Access map Pattern matching(CRAP) [17] や，プリフェッチによってキャッシュが汚染されるのを防ぐために Prefetch-Aware Cache Line Promotion(PACP) [18] などを提案した．また，C.J.Wu らは，既存の置換アルゴリズムでは，プリフェッチ由来のラインと，デマンド由来のラインとを等価に扱ってしまうことで，置換アルゴリズムが正常に機能しないと考え，Prefetch-Aware Cache Management(PACMan) [16] を提案した．また，置換アルゴリズムの性能を競うワークショップである THE 2nd CACHE REPLACEMENT CHAMPIONSHIP にて Signature-Based Hit Predictor++(SHiP++) [24] と Hawkeye [23] の 2 つの手法が優秀な成績を収めた．この章では，PACMan と最新の手法について説明をする．

3.2　PACMan

置換アルゴリズムとプリフェッチをキャッシュに同時に適応させた場合，プリフェッチ由来のラインによって置換アルゴリズムが意図した動作をしない場合がある [16]. 例えば，プリフェッチ由来のラインと，デマンド由来のラインでは，デマンド由来のラインはキャッシュに挿入する時点で参照回数が 1 回なのに対し，プリフェッチ由来のラインはキャッシュに挿入する時点では，参照回数が 0 回にも関わらず，どちらのラインも，LRU の場合は MRU 側に，SRRIP の場合は $\mathrm{RRPV} = 2^{\mathrm{M}} - 2$ としてキャッシュに挿入してしまう．これによって正しい参照間隔を推定できず，プログラ

ムによっては性能を落としてしまう場合すらある．また，C.J.Wu らは，LLC において，プリフェッチされたラインの再参照可能性が低く，プリフェッチによってキャッシュが汚染されることがあることを発見した [16]．これらの事実を基にして考案されたのが PACMan である．PACMan は，基本を DRRIP として，**プリフェッチによるアクセス**と，**デマンドによるアクセス**とを区別することで，RRPV 値の更新時における動作を変え，プリフェッチによってキャッシュが汚染されないようプリフェッチされたラインを積極的に追い出すようにした．PACMan には以下の 4 つの動作がある．

1. PACMan on Misses (PACMan-M)

   RRIP において，プリフェッチ・アクセスがキャッシュ・ミスを起こしたときのみ動作を変え，プリフェッチによる挿入は $\mathrm{RRPV} = 2^{\mathrm{M}} - 1$ として挿入する．こうすることで，プリフェッチによってキャッシュが汚染され，キャッシュ性能の低下を回避できる．

2. PACMan on Hits (PACMan-H)

   RRIP において，プリフェッチによるキャッシュ・アクセスがヒットした時のみ動作を変え，プリフェッチによってヒットしたラインの RRPV は更新しない．こうすることで，プリフェッチによって予測しにくいラインをキャッシュに保持しやすくなる．

3. PACMan on Hits and Misses (PACMan-HM)

   1 と 2 を同時に利用したもの．それぞれの利点である，キャッシュ・ポリューションの回避と，プリフェッチで予測されにくいラインをキャッシュに保持できるようになる．

4. Dynamic PACMan (PACMan-DYN)

   上述した 3 つのアルゴリズムでは，プリフェッチが非常に有効であるプログラムでは，逆に悪影響を与えてしまうので，SDM を用い，最もミスが少ないものを決定し，動的に使い分ける．この時，SDM で比較するアルゴリズムは，PACMan を用いないか，PACMan を用いる場合は，PACMan-M であるか，PACMan-H であるか，PACMan-HM で あるかの 4 通りと，RRIP のうち，SRRIP に適用するか，BRRIP に適用するかの 2 通りの，合計 8 通りが存在する．しかし，BRRIP の動作と PACMan-M の動作が近いことから，BRRIP + PACMan-H と，

BRRIP + PACMan-HM は本質的に同じであり，PACMan-H があらゆる場合において有効であるので，PACMan-H を常に使うと考えて，SRRIP + PACMan-H，SRRIP + PACMan-HM，BRRIP + PACMan-HM の 3 つを動的に選択する．

3.3 SHiP++

3.3.1 概要

ここでは，以降の PC ベースの学習器をつけた置換アルゴリズムのベースとなった Signature-based Hit Predictor(SHiP) とそのプリフェッチ拡張 SHiP++ について述べる [7,24]．SHiP は，手法名のとおり，キャッシュに挿入するラインに対してアクセスが来るかどうかを予測する．SHiP 自体は，予測器であるため，任意の置換アルゴリズムをベースにして適用することが可能であるが，以降は，性能がよく省面積である 2bit の RRIP に適用した場合について述べる．

DRRIP では，SDM を用いて，その瞬間ごとに最も性能を発揮する置換アルゴリズムを切り替えているが，SRRIP と BRRIP との大きな違いは挿入時の RRPV，すなわち，挿入する位置である．また，キャッシュの一部を最適でない置換アルゴリズムで動かす必要があるため，キャッシュ全体を同じ置換アルゴリズムで動かす場合に比べて，性能が下がる可能性がある．そこで，SHiP では，キャッシュ・ラインの挿入が発生するときに，そのアクセスが持つ特徴量を用いて学習テーブルを参照し，ヒットする可能性が高いものは RRPV = 2 で挿入し，そうでないものは RRPV = 3 として挿入する．ここで，特徴量としては，PC の一部を用いると，最もアクセスの再現性が大きく，性能が上昇するため，以降，SHiP で用いる特徴量は PC とする．

CRC2 [27] にて，SHiP のプリフェッチ拡張である SHiP++ [24] が提案されている．通常の SHiP では，プリフェッチ・アクセスとデマンド・アクセスを区別しないため，アクセスの来ないプリフェッチ・ラインの挿入位置を良くしてしまい，性能が下がってしまう場合がある．そこで，SHiP++ では，以下の 5 つの改良を SHiP に対して加えている．

1. 学習テーブルの確信度に依存して挿入位置を決定する．
2. 同じ特徴量を持つラインでの学習は 1 度のみ．
3. WriteBack 由来のラインは RRPV = 3 にする．
4. デマンド・アクセスとプリフェッチ・アクセスで学習器を分割する．

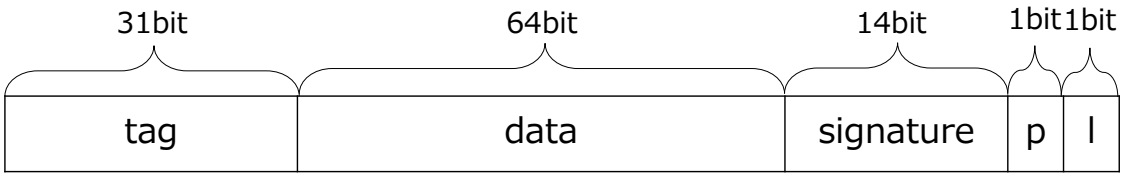

図 3.1: キャッシュ・ラインのビット・フィールド

5. プリフェッチ・ラインとデマンド・ラインのヒット時の動作を異なるものにする．

具体的な処理や考えに関しては，以降説明を行う．

3.3.2 学習方法

ここでは，SHiP++ の具体的な学習方法について説明する．SHiP++ では，PC にハッシュをかけたものをインデックスとする学習テーブル Signature History Counter Table(SHCT) を用いる．SHCT は，各エントリごとに飽和カウンタ (SHCTR) を持ち，アクセスが来るものほど値が大きくなるように学習する．SHiP++ では，エントリ数を 2^{14}，SHCTR の bit 数を 3bit としている．また，SHCTR の初期値を 1 とし，学習のタイミングとその方法は以下のようにしている．

挿入時

挿入時には，SHCT を更新しないが，キャッシュ・ラインに対し，PC にハッシュをかけて 14bit にした特徴量と，プリフェッチで挿入されたかどうかを判断する 1bit と，学習済みかどうかを判断する 1bit を追加する．ここで，上述したとおり，デマンド・アクセスと プリフェッチ・アクセスで学習テーブルを分離する必要があるため，特徴量の 14bit の内 1bit は，プリフェッチによる挿入かどうかを判断するのに用いる．キャッシュ・ラインの持つフィールドは図 3.1 のようになっている．ここで，フィールドの p は，プリフェッチされたかどうか，l は学習済みかどうかを表す．

ヒット時

1. デマンド・アクセスの場合

   同じラインへのアクセスで SHCT の学習をしていない場合のみ，ラインの持つ特徴量に対応する SHCT のエントリの SHCTR をインクリメントする．その後学習済み bit をたて，プリフェッチ bit を下げる．

2. プリフェッチ・アクセスの場合

   プリフェッチによって挿入されたラインにヒットした場合，学習済みでない場合のみ，低確率でラインの持つ特徴量に対応する SHCT のエントリの SHCTR をインクリメントし，学習済み bit をたてる．プリフェッチで挿入されたラインでない場合，1 の場合と同様に処理を行う．

すなわち，プリフェッチで挿入されたラインへの プリフェッチ・アクセスでの SHCT の更新は低確率であること意外は デマンド・アクセスと プリフェッチ・アクセスで同様な処理を行う．

追い出し時

追い出すラインの学習済み bit を確認し，学習済みである場合はそのラインの持つ特徴量に対応する SHCT のエントリの SHCTR をインクリメント，そうでない場合はデクリメントする．

全てのラインに対して図 3.1 のようにフィールドを割り当てると必要な記憶領域が非常に大きくなってしまう．そこで，SDM の考えを流用し，一部のセットへのアクセスに対してのみ上述した学習を行う．

3.3.3 挿入位置の決定

新しいラインを挿入するとき，デマンド・アクセスの場合はアクセスを生成している PC，プリフェッチ・アクセスの場合はプリフェッチをトリガした PC をそれぞれ特徴量とし，SHCT の対応するエントリの SHCTR を読み出す．SHCTR の値に依存して以下のように RRPV を決定する．

SHCTR $==$ 0 の場合

RRPV = 3 として挿入．

$1 \leqq$ SHCTR $\leqq 6$ の場合

RRPV = 2 として挿入．

SHCTR $==$ 7 の場合

デマンド・アクセスの場合，RRPV = 0，プリフェッチ・アクセスの場合，RRPV = 1 として挿入．

以上の処理とは他に，WriteBack による挿入では，SHCTR に関係なく RRPV = 3 として挿入する．これは，WriteBack が ロード/ストア ではないため，続くアクセスが発生しにくいからである．

3.3.4 その他の動作

SHiP++ では，それぞれのアクセスの傾向を踏まえた処理を加えている．

1. プリフェッチで挿入されたラインへのデマンド・ヒット時には，RRPV = 3 にする．

   これは，プリフェッチで挿入されたラインは 1 度しか用いられないという思想に基づいている．

2. WriteBack がヒットした場合は，RRPV = 3 にする．

   これは，上述した WriteBack の性質を踏まえている．

3.4 Hawkeye

3.4.1 着想

Hawkeye [1, 23] は，Akanksha らによって提案された手法であり，SHiP に代表されるような既存の置換アルゴリズムの学習方法とは大きく異なる．今まで述べてきた既存手法では，過去のアクセスによる学習を，一度アクセスが来たキャッシュ・ラインは近いうちにアクセスが来る，使われずに追い出されたキャッシュ・ラインは次も使われない，などといったヒューリスティックに基づいて，あらかじめある程度の学習の方向性を与えることで行っている．しかし，ヒューリスティックに従わないアクセスには対処することができなくってしまう．そこで，Hawkeye では，学習を，最も性能の良い置換アルゴリズムである OPT [25] の振る舞いから行うことを提案している．OPT は，ヒューリスティック由来の置換アルゴリズムではないため，同じ挙動を示せば，アクセス・パターンに依存せず性能が良くなると考えた．Hawkeye の機構について述べる前に，OPT の動作を紹介する．

3.4.2 OPT

OPT は，Belady によって最も性能が良い置換アルゴリズムであることを証明された手法である．キャッシュ・メモリが発生して，ラインを追い出すとき，同じセット内のキャッシュ・ラインのうち，最も遠い未来にアクセスが来るものを追い出す．ここで,「最も遠い未来にアクセスが来るキャッシュ・ライン」を知るためには，事前に同じプログラムを実行し，メモリ・アクセスを全て記録する必要があるため，実際のプロセッサに搭載することは不可能である．

3.4.3 学習方法

上述したとおり，OPT の動作を実際のプロセッサで動作させることはできない．Akanksha らは，過去のアクセスに関しては OPT と同じ振る舞いをすることができることに着目した．現時点から未来の情報を参照するのではなく，現時点が過去のある時点にとっての未来であるため，少し昔の時点での OPT であれば，プロセッサ内で実現可能である．Hawkeye は，この事実を学習に用いるために，以下の 2 つについて調査を行っている．

1. OPT の動作を再現するのに必要な記憶容量はどの程度か．
2. OPT が過去に残した/捨てたキャッシュ・ラインは今後も残す/捨てるのか．また，その振る舞いを特徴付ける量は何なのか．

まず，1 について，図 3.2 では，横軸をアクセス履歴を確保する用量，縦軸に実際の OPT の振る舞いとの差異を表している．横軸の 1X や， 2X は，キャッシュ容量の 1 倍，2 倍の記憶領域という意味である．このグラフのを見ると，LRU では，40% 程しか OPT と同じ振る舞いをしていないのに対し，容量に制限のある Belady では，キャッシュの容量の 8 倍 (8X) を超えたあたりから，OPT との差異が 10% を下回る，すなわち，90% 以上の確率で，OPT と同じ振る舞いをすることがわかる．

次に，2 について，図 3.3 に示す．図 3.3 は，横軸に SPEC CPU 2006 のベンチマーク，縦軸に，同じ PC での Load 命令に対する OPT の振る舞いの一致率を表している．これを見ると，平均して，90% 以上の同じ PC を持つ Load 命令が，OPT によって同じ処理を受けることがわかる．そのため，PC 毎に，OPT が挿入するラインを次のアクセスまでに追い出したかどうかを学習することで，OPT の挙動に近づくことができる．以下，Hawkeye の具体的な構成について述べる．

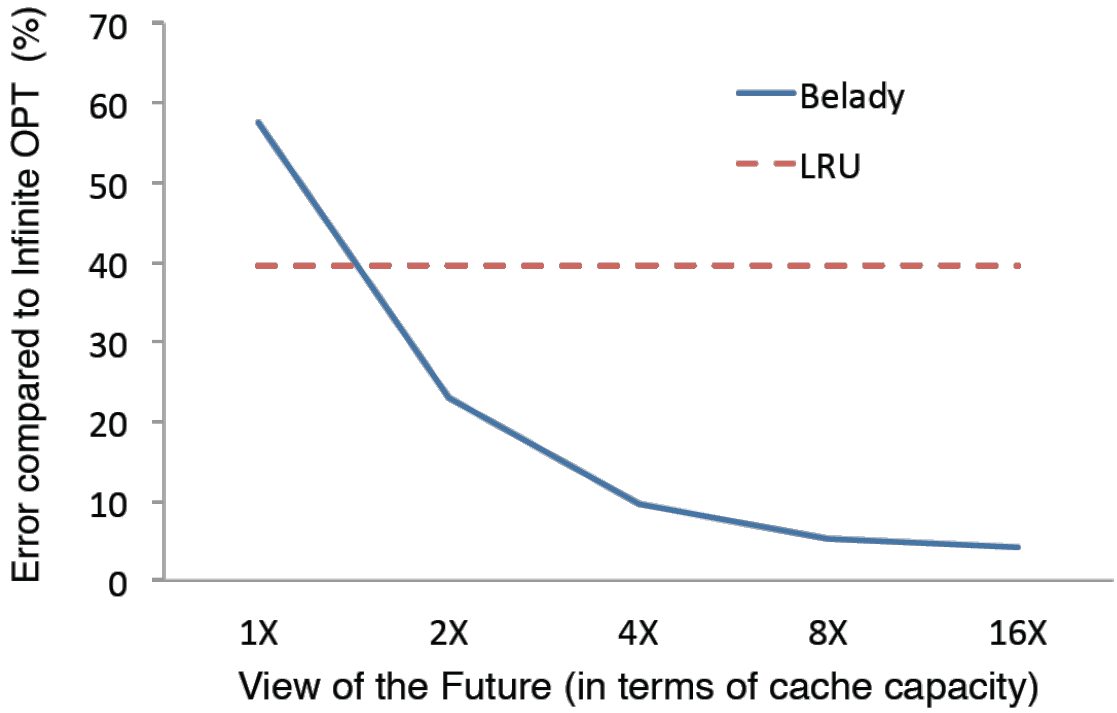

図 3.2: 記憶容量に対する OPT の振る舞いとの相違 [1]

3.4.4 構成要素

Hawkeye は図 3.4 に示すように OPTgen と Hawkeye Predictor という 2 つの主要な構成要素によってなっている．ここでは，それぞれについて述べる．

3.4.5 OPTgen

OPTgen は，OPT だとしたらどのラインをキャッシュに残すかを決定するモジュールである．よって，あるアクセスが来たときに OPT 下でそれがヒットになるかミスになるかを判断することが可能である．そのために，OPTgen では，*usage interval* と，*liveness interval* を定義している．*usage interval* とは，あるライン X へのアクセスがあったときに，そのラインへの次のアクセス (X' とする) があるまでの期間を表し，*liveness interval* は，あるラインがキャッシュ内に存在していた期間を表す．このとき，X の *usage interval* の間で，*liveness interval* が被っているラインの数がキャッシュの容量と一致する場合にはミス，それ以外の場合にはヒットと判断する．

例を図 3.5 に示す．図 3.5 では，X' がヒットかミスかを予測しようとしている．このとき，キャッシュの容量は 2 だとし，A，B，C に関する *liveness interval* は 1 つも

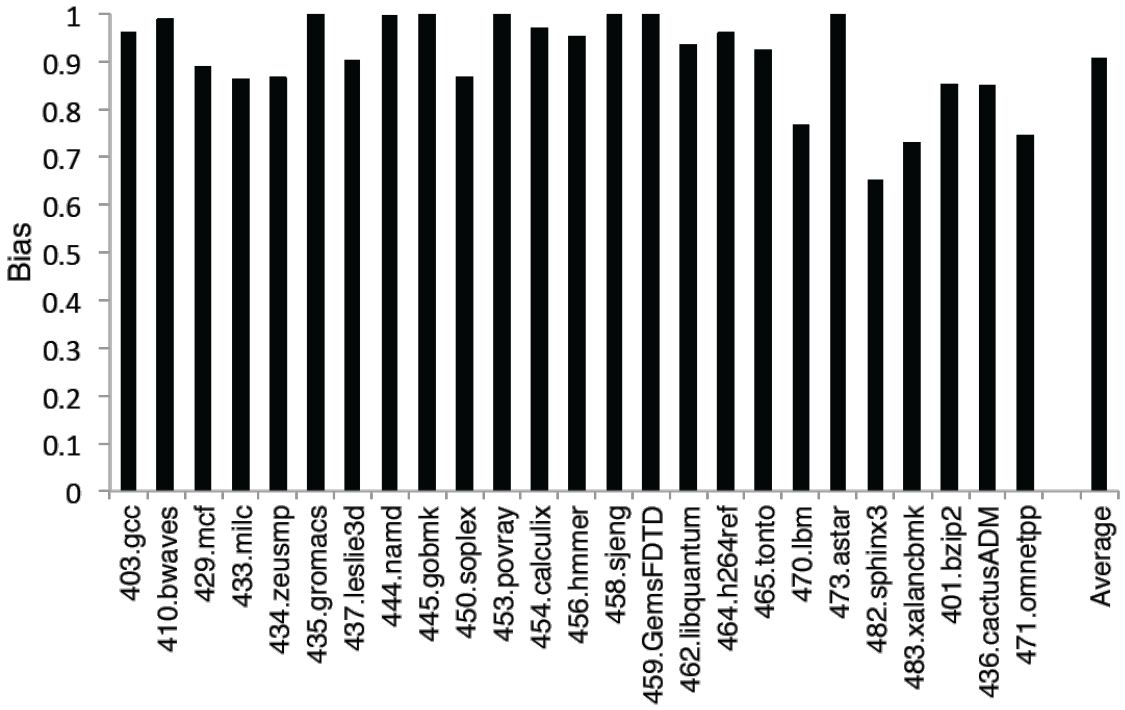

図 3.3: 同じ PC での Load 命令における，OPT の振る舞い [1]

被っていないとする．すると，X’ のアクセス時点では，*liveness interval* が被るものが 1 つもないので，X’ はヒットと判断することができる．

OPTgen は上述した操作をするために *occupancy vector* を用いる．*occupancy vector* の各エントリは，ある時点において *liveness interval* が被っているラインの数を保持する．*occupancy vector* の動作は以下のように定義される．

- ロードの度に新しいエントリを割り当て，エントリの中身は 0 にする
- X に対するロードが初めてだった場合は特別な処理をしない

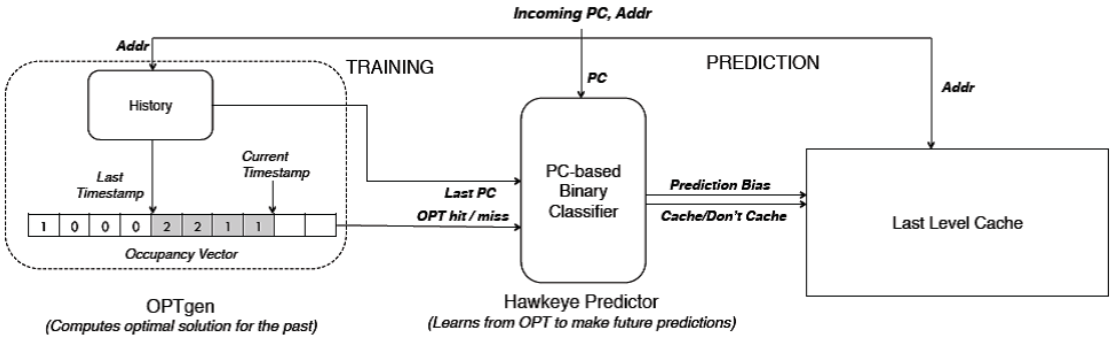

図 3.4: Block diagram of the Hawkeye replacement algorithm [1]

- X に対するロードが初めてでない場合は，X の *usage interval* 間に存在するエントリの値を調べ，全てキャッシュの容量未満ならエントリの値をインクリメントし，ヒットしたと判断．そうでない場合は，ミスだと判断し，エントリの値はいじらない．

occupancy vector の具体的な処理を図 3.6 に示す．図 3.6(a) は実際のアクセス，(b) は (a) のアクセスに対する OPT の動作，(c) は *occupancy vector* を用いた OPTgen の動作であり，キャッシュの容量は 2 としている．(c) の T = 2，6，8，9，10，11 ときに過去のアクセス情報からヒットまたはミスを予測している．例えば，T = 2，6，8 における B，A，D のアクセスに対して，それぞれ 1 つ前の同じアクセスまでの間 (*usage interval*) のエントリで値がキャッシュの容量 (= 2) となるものを探して見つからないため，全て 1 だけインクリメントしてヒットと予測している．これは，T = 10 でも同様である．一方，T = 9，11 では同様な処理を行ったとき，エントリの内部の数値がキャッシュ容量と一致するものがあるため，ミスと判断し，エントリの値には変更を加えていない．ただし，OPTgen の動作は bypass を想定しており，ミスしたときにミスしたラインを必ずしもキャッシュに保持するわけではない．そのため，bypass を考慮しないのであれば，新しいエントリを割り当てる時に値を 0 の代わりに 1 にする．

ここで注目してもらいたいのが，OPTgen の振る舞いが OPT と完全に一致しており，T = 2 や T = 6 のような再参照の間隔が大きくことなるアクセスでも問題なく予測ができていることと，OPT ライクな振る舞いを *occupancy vector* を用いることでハードウェアで実装可能なことである．また，セット・アソシアチブ型のキャッシュに対しては各セットに対して *occupancy vector* を用いる．

しかし，OPTgen が OPT のように振舞うためには，上述したとおり，キャッシュの容量の 8 倍の履歴を保持する必要がある．そこで，Hawkeye では，Sampler Cache を用いている [4]．Sampler Cache は，学習用のキャッシュであり，計算に要するデータなどを持たない代わりに，学習に必要な情報を持たせる．Hawkeye では，Sampler Cache に，ハッシュをかけた tag，PC，タイムスタンプ，プリフェッチによる挿入かどうかを表すビット，Sampler Cache を LRU で動作させるためのメタデータを保持させる．Sampler Cache は，LLC にアクセスがあった場合，同時にアクセスされ，Sampler Cache でのヒット/ミスに対して OPTgen を動作させる．ここで，LLC へのアクセス全てで学習を行うためには，Sampler Cache のサイズを大きくする必要があるが，SDM [6] の考えから，キャッシュ全体の一部に対するアクセスに対してのみ，Sampler Cache が作動するようにすることで，容量を抑えている．

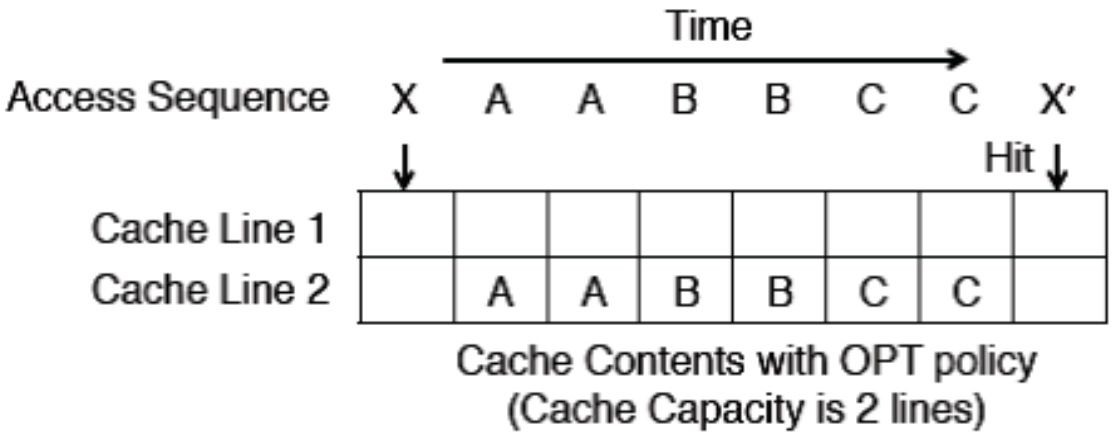

図 3.5: Intuition behinde OPTgen [1]

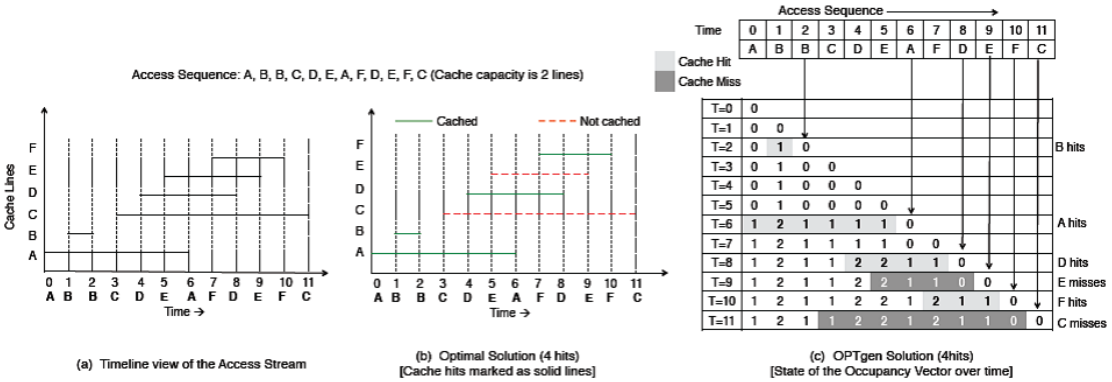

図 3.6: Example to illustrate OPTgen [1]

3.4.6 Hawkeye Predictor

次に Hawkeye Predictor について述べる．この予測器は，ある PC によるロード命令でロードされるラインが cache-friendry か cache-averse かを判断する予測器である．上述した通り，OPT 下では，同じ PC によるロード命令におけるキャッシュの振る舞いが 90% 以上の割合で一致するため，OPTgen によってあるライン X がヒットしたと予測された場合予測器は X へのアクセスをポジティブに学習し，ミスだと判断された場合にはネガティブに学習する．

Hwkeye Predictor は 2K エントリを持ち，学習用に 5bit のカウンタを持ち，11 ビットのハッシュ化された PC をインデックスとして用いる．この 5bit のカウンタの最上位ビットが 1 だった場合は cache-friendly，0 だった場合は cache-averse と予測する．

3.4.7 動作

Hawkeye が実際にどのように動作するのかを述べる．Hawkeye は policy を実行するために 3 ビットの RRPV [5] カウンタを用いる．このカウンタは，値が小さいほど追い出されにくく，大きいほど追い出されやすいことを表す．

毎回のキャッシュ・アクセスに対してヒット，ミスに関わらず，Hawkeye Predictor はそのアクセスが cache-friendly か cache-averse かを予測する．RRPV カウンタの値の定め方は以下に従う．

- ヒットしていて，cache-averse と予測された場合．RRPV = 7 とする．
- ヒットしていて，cache-friendly と予測された場合．RRPV = 0 とする．
- ミスしていて，cache-averse と予測された場合．RRPV = 7 とする．
- ミスしていて，cache-friendly と予測された場合．RRPV = 0 とし，その他のラインの RRPV が 6 未満なら 1 だけインクリメントする．

追い出しが発生する場合には，RRPV = 7 であるラインを探し追い出す．もし，RRPV = 7 であるラインがない場合，RRPV が最も大きいラインを追い出す．ここで，RRPV = 7 となるラインがないときはフェーズが切り替わったと考えることができ，そのときの動作は実質的には LRU と同じになっている．こうすることで，フェーズの切り替わったときに誤った予測をし続けることがなくなる．

3.4.8 プリフェッチ拡張

上述した Hawkeye では，プリフェッチの存在を考慮していない．そのため，プリフェッチが存在する状況下では，性能が下がってしまう．そこで，Hawkeye のプリフェッチ存在下も提案されている [23]．プリフェッチの扱いを考える上で，Akanksha らは，キャッシュへのアクセスを以下の 4 つに分類している．

1. デマンド・アクセスのあとにデマンド・アクセスがくるもの: D-D
2. プリフェッチ・アクセスのあとにデマンド・アクセスがくるもの: P-D
3. デマンド・アクセスのあとにプリフェッチ・アクセスがくるもの: D-P
4. プリフェッチ・アクセスのあとにプリフェッチ・アクセスがくるもの: P-P

D-D や，P-D では，デマンド・アクセスで終了しているため，実際に CPU で使用されるデータへのアクセスであるのに対し，D-P と P-P ではプリフェッチ・アクセスで終了しているため，そのアクセスによるデータが，実際に CPU で使用されるかはわからない．そのため，Prefetch-Aware な Hawkeye では，プリフェッチ・アクセスによって終了するアクセス（以下 ”*-P” と表記） アクセスによって Haekeye Predictor が学習しないようにしている．しかし，Akanksha らは，キャッシュの利用効率と，メモリ・アクセスによるトラフィックがトレードオフの関係にあるため，上述したように，ただ ”*-P” アクセスを無視するだけでは，キャッシュの利用効率が上昇したとしても，プリフェッチ・アクセスが連続することによって，トラフィックが増加し，性能が減少する可能性があることに着目した．これは，プリフェッチ・アクセスは，挿入しようとするラインが既にキャッシュ上に存在するかどうかを先に判断して，存在しない場合のみ プリフェッチ・アクセスを発行するからである．例えば，P-P が短いインターバルで連続するようなアクセスが来た場合，最初の プリフェッチ・アクセスをキャッシュに挿入することで，以降の プリフェッチ・アクセスによって，トラフィックが発生しないが，P-P によるラインを全てキャッシュに挿入しなかった場合，以降の プリフェッチ・アクセス全てで，トラフィックが発生ししまう．そこで，”*-P” アクセスを全ては無視せず，インターバルが短いもののみキャッシュに挿入する，すなわち，Hawkeye Predictor でポジティブに学習するようにしている．

3.4.9 Perceptron Learning

Perceptron Learning [28](以下，Perceptron) は，SHiP のような特徴量ごとに学習する手法である．Perceptron の学習は，Sampling Dead Block Prediction(SDBP) [4] の影響を大きく受けており，PC，アドレス，命令列など様々なものを特徴量として，それぞれの特徴量に対して SHCT を持つ．再参照の予測は，それぞれの特徴量で SHCT にアクセスし，SHCT の持つ値の和が，ある閾値を越えるかどうかで判断を行う．Perceptron では，複数の特徴量に対して学習を行うことで，アクセス・パターン毎に影響の出やすい特徴量が異なっても精度の高い予測をすることが可能となっている．また，Perceptoron のプリフェッチ拡張として，Multiperspective Reuse Prediction [13] といった手法も提案されている．

第4章　PrePromotion

4.1　PrePromotionのモチベーション

4.1.1　プリフェッチによるアクセス予測

LRU や RRIP などの置き換えアルゴリズムはどれくらい最近用いられたか，どれくらいの頻度で用いられたかなどの過去の履歴を用いてラインの置き換えを決定している．しかし，これらの置き換えアルゴリズムでは過去の履歴のみに頼っているために，再参照が迫っているラインですらキャッシュから追い出してしまう可能性がある．そこで，我々は再参照がせまっていることを予測する予測器をキャッシュに適応することを考えた．予測器としてプリフェッチャを用いることを考え，キャッシュ中における予測的中率の調査を行った．そのデータを図 4.1，図 4.2 に示す．

図では，キャッシュから追い出された全ラインのうち，そのラインが予測されたかどうか，そしてそのラインが再参照されたかどうかを示している．ただし，図 4.2 では，416.games と 453.povray において，キャッシュのサイズが大きく，プログラムに使うデータが全てキャッシュ上に乗ってしまい，追い出しが発生していないため，値を 0 にしている．図 4.1 を見ると，re-referenced after predicted，re-referenced before predicted，NOT re-referenced but predicted が再参照を予測されたラインであり，re-referenced but NOT predicted と NOT re-referenced and NOT predicted が再参照を予測されていないラインであるが，再参照を予測されたラインの内，実際に再参照された割合が，再参照を予測されてないラインの内，再参照されたラインの割合より大きいことが見て取れる．全ベンチマークの合計での割合で見ると，再参照が予測されたラインの内，実際に再参照されたラインは約 38% 再参照が予測されていないラインの内，再参照されたラインの割合が約 12 %であった．全ラインの内，再参照されたラインの割合が約 28% であることを考えると，プリフェッチャによって再参照の予測ができることがわかる．一方，図 4.2 を見ると，あまり予測が成功しているように見えないが，これは，プリフェッチャを L2 キャッシュにかけており，予測が成功した場合のアクセスがほとんど L2 に吸収されてしまうからである．

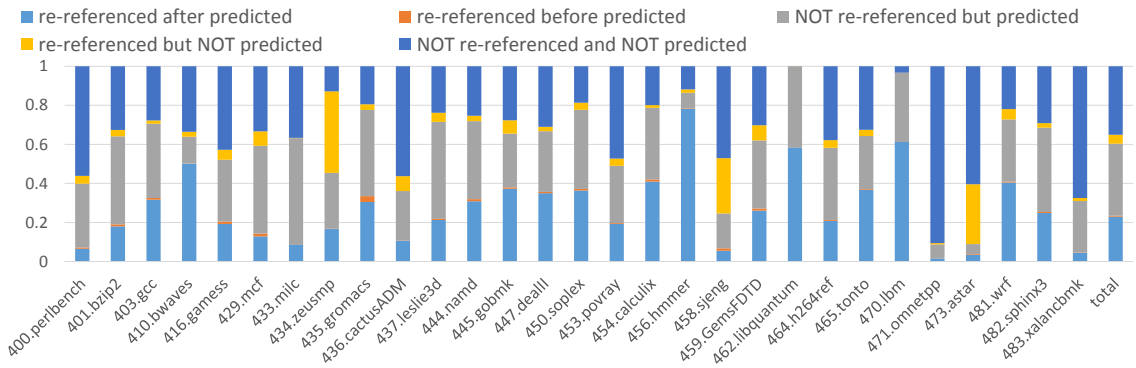

図 4.1: 置き換えアルゴリズムが DRRIP の場合の L2 キャッシュにおけるプリフェッチャが参照予測したラインの振る舞い

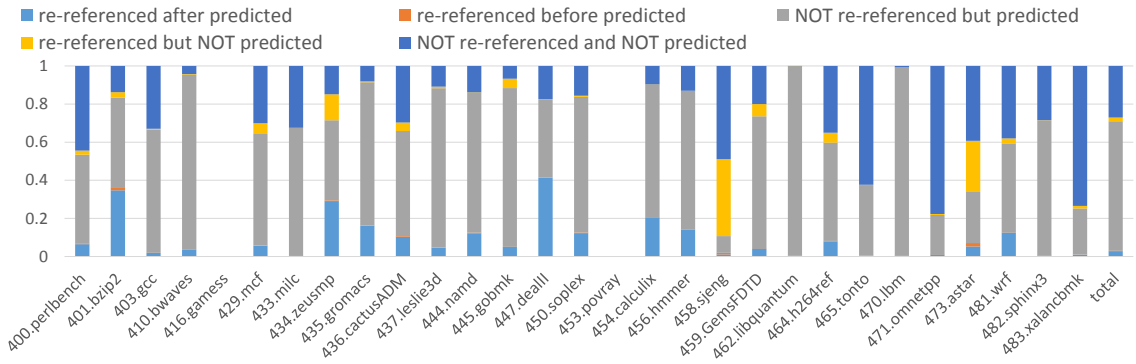

図 4.2: 置き換えアルゴリズムが DRRIP の場合の L3 キャッシュにおけるプリフェッチャが参照予測したラインの振る舞い

4.1.2　プリフェッチされたラインの振る舞い

PACMan に代表されるようなプリフェッチを意識した置き換えアルゴリズムは大半がプリフェッチされたラインが再度用いられることが少ないと考え，キャッシュに乗せたあとすぐに追い出すようなアルゴリズムになっている．我々は，置き換えアルゴリズムとプリフェッチを同時に用いるにあたってキャッシュにおける参照の傾向を調査することにした．そのデータを図 4.3，図 4.4 に示す．

図では，キャッシュから追い出された全ラインのうち，プリフェッチ由来のラインか，デマンド由来のラインか，何度参照されたか，を表している．図 4.4 において，416.games と 453.povray が 0 になっているのは上述したとおりである．デマンドによるラインの参照回数が 2 回以上，1 回なのに対してプリフェッチによるラインのみ，参照回数が 2 回以上，1 回，0 回となっているのは，デマンドはそれ自体が参照であり，必ず 1 回以上参照されるのに対し，プリフェッチはキャッシュに乗せても

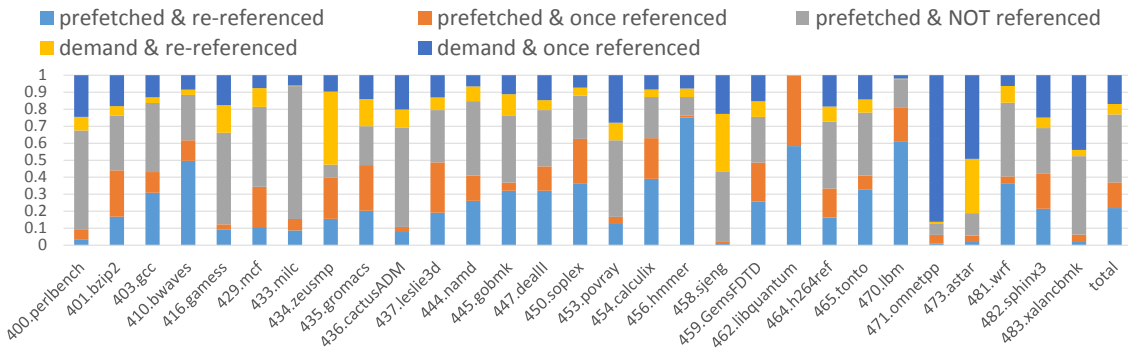

図 4.3: 置き換えアルゴリズムが DRRIP の場合の L2 キャッシュにおけるプリフェッチによるラインとデマンドによるラインの振る舞い

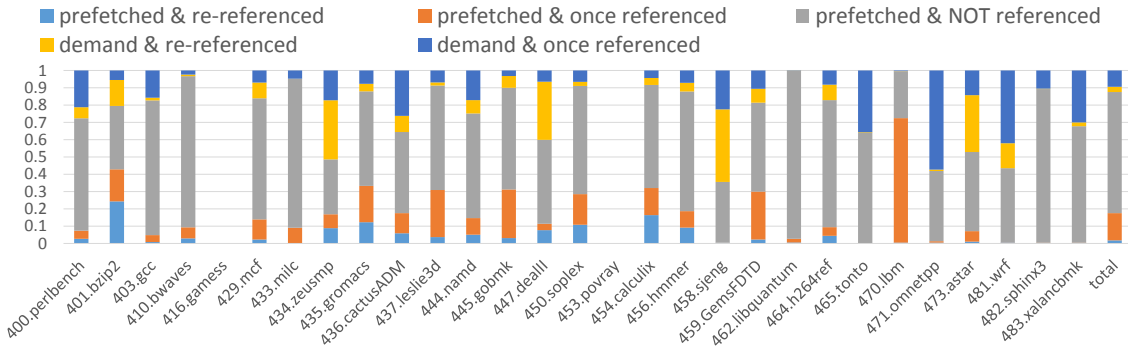

図 4.4: 置き換えアルゴリズムが DRRIP の場合の L3 キャッシュにおけるプリフェッチによるラインとデマンドによるラインの振る舞い

参照されないことがあるからである．図 4.3 を見ると，prefetched & re-referenced，prefetched & once referenced，prefetched & NOT referenced がプリフェッチ由来のラインであり，demand & re-refereced，demand & once referenced がデマンド由来のラインであるが，プリフェッチ由来のラインの内，prefetched & re-referenced が占める割合と，デマンド由来のラインの内，demand & re-referenced が占める割合が大体似ていることがわかる．全ベンチマークの合計で割合を見ると，全ラインの内，再参照されるラインが約 28% であり，プリフェッチされて再参照されるのが約 28 %なのに対し，デマンドされて再参照されるのが約 26% となっている．このことから，キャッシュに乗せたラインがデマンド由来だろうが，プリフェッチ由来だろうが再参照される可能性は大差がないことがわかる．一方，図 4.4 では，全ベンチマーク合計で見たとき，プリフェッチ由来のラインがほとんど再参照されないとなっているが，これはプリフェッチャを L2 キャッシュにかけており，プリフェッチされたラインが L3 の参照傾向を持たないことにある．

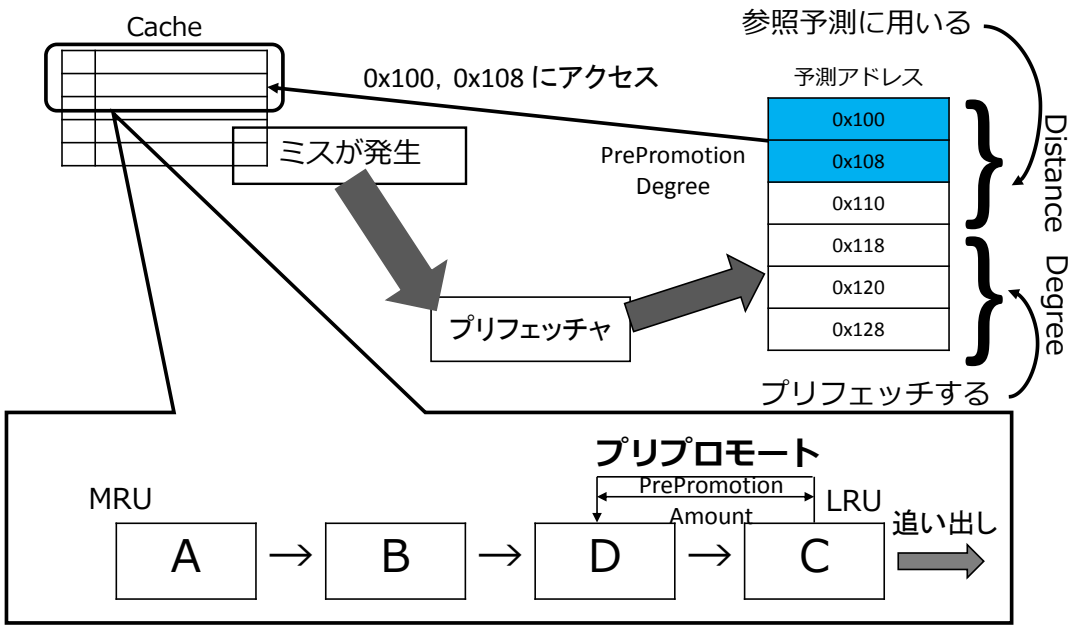

図 4.5: PrePromotion の動作

4.2 PrePromotion の動作

上述した 2 つの調査を基に考案されたのが PrePromotion [19] である．キャッシュの容量の増加に伴い，以前なら参照間隔が大きく，再参照される時にはキャッシュから追い出されていたラインがキャッシュに残っている可能性が出てきている．そこで，それらのラインに参照が迫っているときにキャッシュから追い出されにくくするために，プリフェッチャが予測したアドレスを用いて参照を予測し，予測したアドレスを持つラインが既にキャッシュに存在していた場合に置き換えアルゴリズムの状態を更新している．例えば，置き換えアルゴリズムが LRU の場合，ヒットしたラインを MRU 側に移動させ，RRIP の場合，ヒットしたラインの RRPV を 0 にしている．

実際の動作を図 4.5 に示す．まず，キャッシュでのミスをトリガにプリフェッチャが作動し，いくつか予測アドレスを出す．それらの予測アドレスの内，現在ミスしたアドレスに近いものからいくつかを飛ばして，そこから更にいくつかのアドレスを実際のプリフェッチに用いる．また，プリフェッチャの予測したアドレスの内，ミスしたアドレスに近いものからいくつかのアドレスを参照予測に用いる (参照予測に用いるアドレス数を PrePromotionDegree とする)．これらのアドレスを用いてキャッシュにアクセスをし，図のようにアドレス C を持つラインを MRU 側に移動させる．

第5章　提案手法

5.1　概要

今回，我々が提案するプリフェッチを考慮した置換アルゴリズムはある程度の汎用性がある．そのため，最新の手法から SHiP++ と Hawkeye に対して，今回の手法を適用させ，手法の効果を確認する．なお，Hawkeye に適用させる本提案は，Stubborn キャッシュや，Hawkeye のように，キャッシュに追い出されにくい領域を作る必要があるため，SHiP++ には適用させていない．以下で，それぞれの手法に対して適用させた手法について述べる．

5.2　提案手法の適用

SHiP++ への適用

SHiP++ では，プリフェッチで挿入されたラインに対する最初のデマンド・アクセスで $\mathrm{RRPV} = 3$ にしていた．これは，3.3 でも述べた通り，プリフェッチで挿入したラインは一度しか使われないという考えに基づいている．しかし，プリフェッチで挿入されなかったとしても，同じラインへのデマンド・アクセスは発生するため，プリフェッチによって挿入にかかる遅延が短縮されただけと考えられる．そうすると，プリフェッチで挿入されたラインへの最初のデマンド・アクセスでは，挿入時と同じ処理をするべきと考えられる．

図 5.1，図 5.2 に LLC における実際のアクセスの傾向を示す．図は，SPEC CPU2006 [29] から 20 個のベンチマークにおけるアクセスの傾向を示している．また，LLC の置換アルゴリズムは，DRRIP とした．それぞれ，横軸が同じラインへの 2 回目以降のデマンド・アクセスの回数，縦軸がデマンド・アクセスが 1 回以上くるラインの総数に対する比率を表す．ただし，横軸の 20 は，20 回以上のものを全てまとめている．凡例の Demand Line，Prefetch Line は，それぞれデマンドで挿入されたライン，プリフェッチで挿入されたラインを意味する．ここで，Demand Line は，挿

入時点でデマンド・アクセスが来ていると考え，以降のデマンド・アクセスの回数をカウントするのに対し，Prefetch Line は，挿入時点では，デマンド・アクセスが来ていないので，最初にデマンド・アクセスが来たときを 0 回と考え，以降のデマンド・アクセスの回数をカウントしている．すなわち，Demand Line は，挿入後のデマンド・アクセスの回数，Prefetch Line は，最初のデマンド・アクセス後のデマンド・アクセスの回数を表す．図を見ると，GemsFDTD と sphinx3 が Demand Line と Prefetch Line で大きく傾向が異なる以外は，程度の大小はあるが，ほぼ同様のアクセス傾向を示している．

よって，我々は，プリフェッチで挿入したラインへの最初のアクセスで，SHCT を参照し，RRPV を決定し直す手法を提案する．このとき，SHiP++ のときと同様にプリフェッチで挿入されたかどうかを判断するプリフェッチ bit を下げる．

Hawkeye への適用

Hawkeye は，cache-friendly であるラインと cache-averse であるラインが同じセット内に同時に存在する場合，cache-friendly であるラインは絶対に追い出されないという性質がある．これは，野村らによる Stubborn 戦略 [22] と類似の戦略であると考えることができる．Stubborn 戦略では，キャッシュから追い出さない領域を確保するが，これにより再参照間隔が広いアクセスによるミスを減らすことが出来る．また，Hawkeye は，OPTgen を用いることで擬似的に未来を見ることが可能になるため，例えば，プリフェッチ・アクセスが連続したあと，デマンド・アクセスが来る場合，最後のプリフェッチ・アクセスのみキャッシュに残せばよいことが分かる．このプリフェッチ・アクセス自体は，利用されるデータを挿入しているため，cache-friendly と考えるのが良いと思われるが，最後の P-D のインターバルのみに着目すると，ここでミスを起こさないためには，途中の P-P は何度追い出されてもよく，最後の P-D におけるプリフェッチ・アクセスのみキャッシュで保持されればよいことがわかる．そのため，このような P-P-⋯-P-D となるようなプリフェッチ・アクセスは cache-averse としてよいと考えられる．

そこで，我々は，P-P となるプリフェッチ・アクセスの回数をカウントし，予測器によって，cache-friendly と判断されたもので，回数が閾値を越えたものはネガティブに，そうでないものはポジティブに学習する手法を提案する．学習のタイミングは，デマンド・アクセスで終端する場合のみとする．ここで，閾値を越えないものでポジティブに学習するのは，そのプリフェッチ・アクセスを参照までの間隔が短い

アクセスと認識するためである．また，予測器が対象とするプリフェッチ・アクセスを cache-averse と判断した場合は，回数が閾値を越えたものをポジティブに，そうでないものはネガティブに学習する．これは，何度もプリフェッチ・アクセスが来ているにも関わらず，OPT がそのラインを追い出してしまったが，その後にアクセスが来るものを cache-friendly と認識してもらうためである．

今回，閾値としては最も性能が向上した 9 を用いることとする．

5.3 PrePromotion の適用

PrePromotion は，適用させることで，任意の置換アルゴリズムの性能を引き出すことが知られている [20]．そこで，我々は PrePromotion を SHiP++ や Hawkeye に適用することで更なる性能向上を図る．SHiP++ と Hawkeye では，RRPV の持つ意味合いが異なるため，PrePromotion アクセスが来たときの処理を二つの手法で異なるものとする．

SHiP++ への適用

SHiP++ では，普通の RRIP と同様に RRPV を用いているため，PrePromotion の適用は [20] と同様の処理を行う．すなわち，PrePromotion がヒットした場合は，$\mathrm{RRPV} = 0$ とする．そうでない場合は，何も処理をしない．

Hawkeye への適用

Hawkeye では，$0 \leqq \mathrm{RRPV} \leqq 6$ であるラインは，cache-friendly であり，$\mathrm{RRPV} == 7$ のラインは，cache-averse である．そのため，PrePromotion によってヒットしたラインを全てプロモートしてしまうと，cache-averse なものから cache-friendly なものへの転移が発生する可能性がある．そこで，PrePromotion は，プリフェッチャによってアクセスが発生することから，プリフェッチ用の Hawkeye Predictor に問い合わせ，cache-friendly と判断された場合は $\mathrm{RRPV} = 0$ に，cache-averse と判断された場合は，$\mathrm{RRPV} = 7$ とする．これにより，PrePromotion アクセスが発生したタイミングで，従来の再参照距離の更新を行うだけでなく，cache-friendly であるか cache-averse であるかの予測の更新も行うことができる．

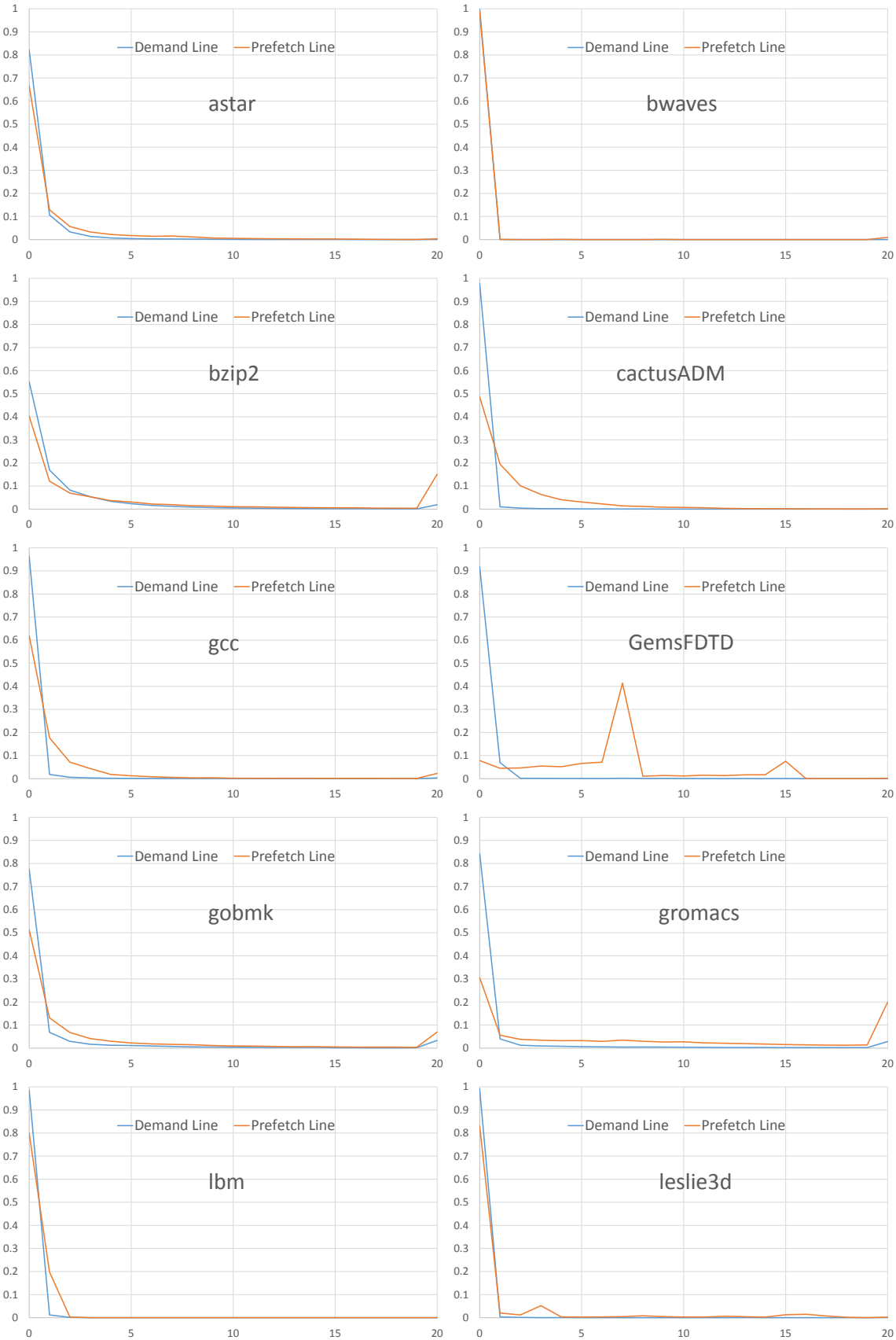

図 5.1: デマンド・ラインとプリフェッチ・ラインのアクセス傾向 1

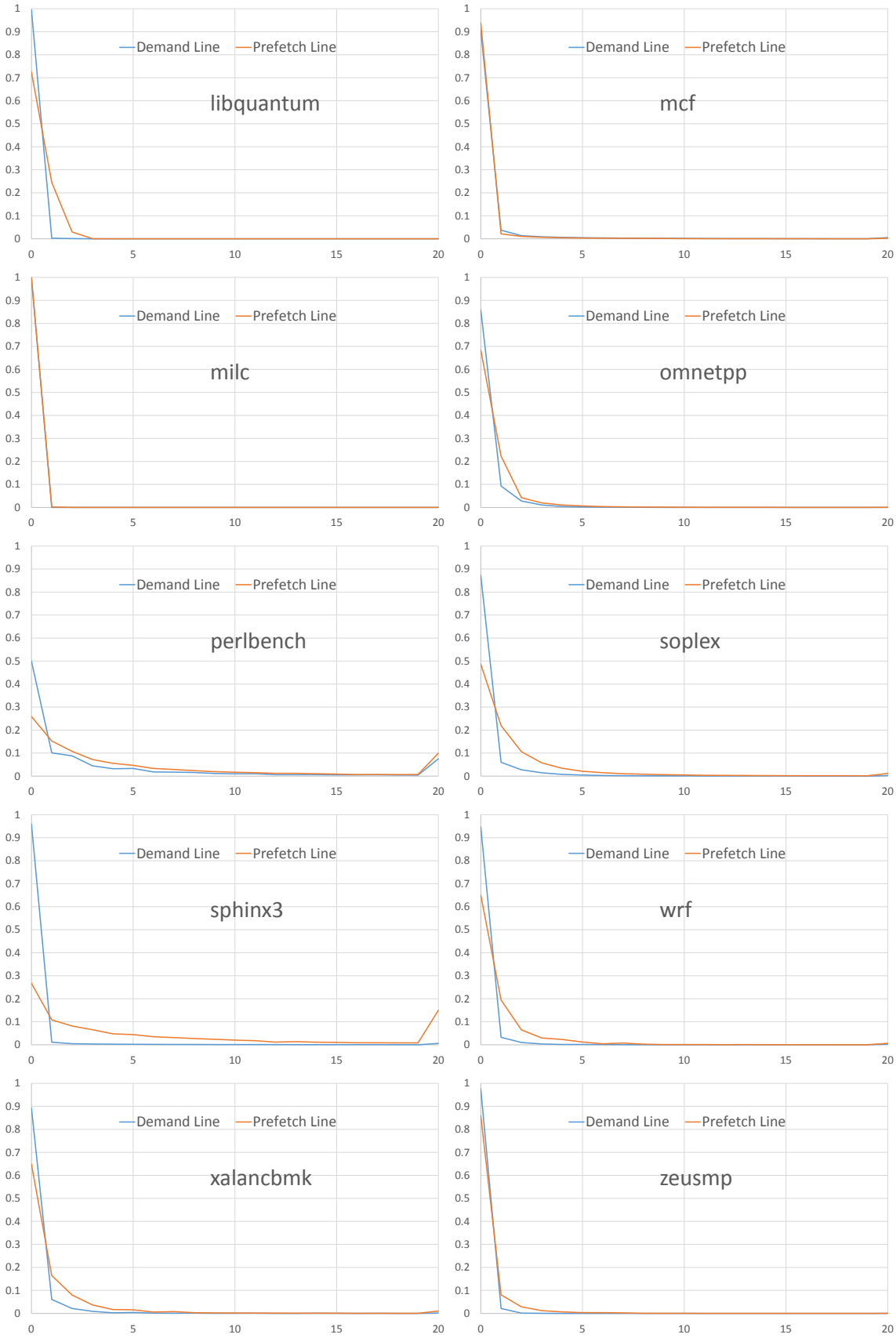

図 5.2: デマンド・ラインとプリフェッチ・ラインのアクセス傾向 2

第6章　評価

6.1　評価対象

本論文では，CRC2 [27] で提供されているシミュレータ上に SHiP++，Hawkeye と今回の提案手法 2 種と PrePromotion を実装し，評価を行う．評価を行う置換アルゴリズムは，LRU，SHiP++，Hawkeye，Proposal_based_SHiP++(以下，Proposal_S)，Proposal_based_Hawkeye(以下，Proposal_H)，SHiP++ + PrePromotion(以下，SHiP++_PP)，Hawkeye + PrePromotion(以下，Hawkeye_PP)，Proposal_S + PrePromotion(以下，Proposal_SPP)，Proposal_H + PrePromotion(以下，Proposal_HPP) とする．

6.2　評価モデル

アーキテクチャの構成は表 6.1 とし，L3 キャッシュの置き換えアルゴリズムのみを切り替えて評価を行う．各手法の用いる RRPV のビット数は，SHiP++ とこれをベースとする提案手法では 2bit，Hawkeye とこれをベースとする提案手法では 3bit とする．シミュレータは上述したとおり，CRC2 が github で公開しているシミュレータ [30] を用いる．ベンチマークは SPEC CPU2006 [29] から，CRC2 の github のレポジトリにある /sim_list/crc_list.txt に記載されているベンチマークを利用した．このベンチマークは，SPEC CPU2006 の中で，MPKI が 1 を超えるものを選択している．また，ベンチマークは CRC2 が提供している trace ファイル [31] を用いる．

6.3　評価

6.3.1　性能評価

評価結果を図 6.1，図 6.2 に示す．図 6.1 は，SHiP++ をベースとした提案手法の性能を，図 6.2 は，Hawkeye をベースとした提案手法の性能をそれぞれ表している．2 つのグラフは，横軸が，ベンチマークを表しており，縦軸は，LRU をベースとし

表 6.1: アーキテクチャの構成

プロセッサ	
issue width	6
ROB	256 entry
branch pred	8KB, g-share
BTB	2KB entry
LSQ	load:72 entry, store:56 entry
キャッシュ	
L1 I キャッシュ	32KB, 8 way, 64B line, 1cycle latency, LRU
L1 D キャッシュ	32KB, 8 way, 64B line, 4cycle latency, LRU
L2 キャッシュ	256KB, 8 way, 64B line, 8cycle latency, LRU
L3 キャッシュ	2MB, 16 way, 64B line, 20cycle latency
Main Memory	tRP: 11cycle latency, tRCD: 11cycle latency, tCAS: 11cycle latency
Swap Latency	100,000cycle latency
L1 プリフェッチャ	Next Line Prefetcher, Distance:0, Degree:1
L2 プリフェッチャ	Stream Prefetcher, Distance:4, Degree:4, PrePromotionDegree:4

た場合の Instructions Per Cycle(IPC) の上昇率を表している．ここで，IPC とは，プロセッサが 1 サイクルの間に何命令実行するかを表す指標であり，この値が大きければ大きいほど，高速であると言える．また，geomean は，IPC 向上率の幾何平均が LRU を 1 とした場合に，何 % 上昇したかを表す.

SHiP++

まず，図 6.1 を見ると，総じて Proposal_S の性能が他と比べて良くなっている．bwaves，lbm，milc，omnetpp などの一部のベンチマークでは，SHiP++ に比べて Propsoal_S の性能が少し低下しているが，図 5.1 と図 5.2 を見ると，これらのベンチマークでは，アクセスの傾向が Demand Line と Prefetch Line でほとんど一致している．しかし，bwaves，lbm，milc の Demand Line に着目すると，キャッシュに挿入後，ほとんどデマンド・アクセスが来ないことがわかる．元の，SHiP++ では，プリフェッチにデマンド・アクセスが来た場合には，RRPV = 3 としていたため，このアクセス傾向に適しているが，本提案では，デマンド用の SHCT を参照して RRPV を決定するため，RRPV ¡ 3 となる可能性があり，このような結果が得られたと考えられる．一方，似たようなアクセス傾向を持つ mcf では，性能を向上させているが，これは，割合で見ると，デマンドで挿入したラインにほとんどデマンド・アクセス

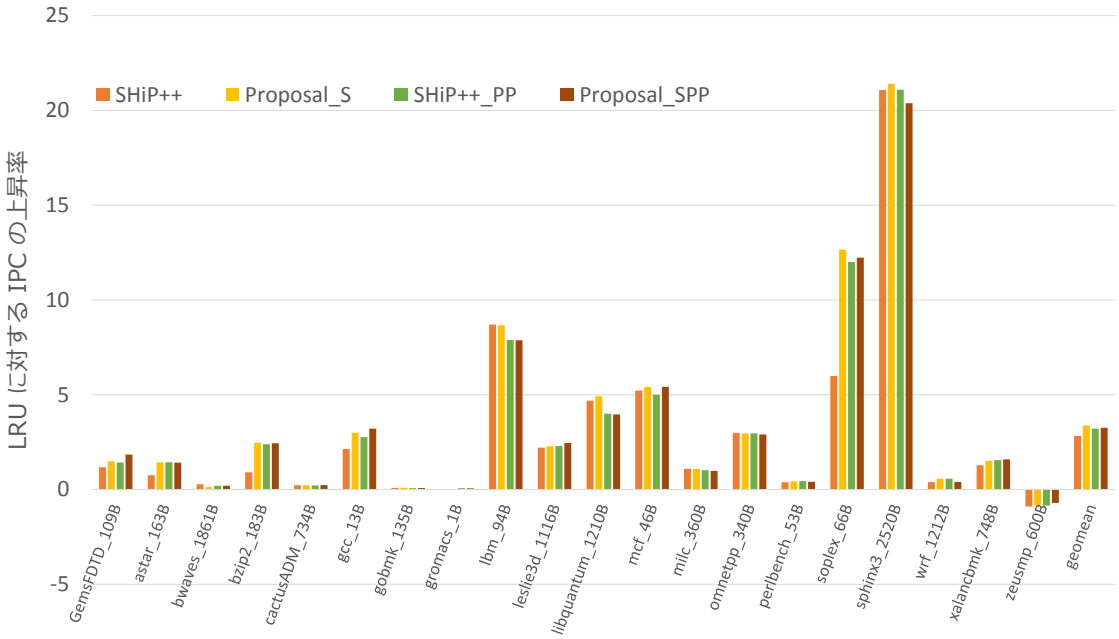

図 6.1: SHiP++ をベースとした提案手法の性能

が来ていないように見えるが，絶対値で見た場合，何度もデマンド・アクセスが来るラインが存在するためであると考えられる．

次に，SHiP++_PP を見ると，こちらも，ベースの SHiP++ に比べて性能が伸びていることがわかる．いくつか性能が低下しているベンチマークが存在するが，こちらも，総じて見ると，PrePromotion により，性能が向上することがわかり，SHiP++ においても PrePromotion が有用なことがわかる．

最後に，Proposal_SPP を見ると，これも，SHiP++ よりは性能を上げているが，sphinx3 では，Proposal_S にも，SHiP++_PP にも性能が劣っている．そのため，PrePromotion と提案を単純に組み合わせても性能が 2 つの手法分向上するわけではないことがわかる．

geomean は，SHiP++ が，2.83%，Proposal_S が，3.37%，SHiP++_PP が，3.21%，Proposal_SPP が，3.26% LRU に対して，IPC が向上しており，Proposal_S が，現状最も性能を向上させることがわかる．

Hawkeye

図 6.2 を見ると，Proposal_H が最も性能が良いことがわかる．特に，libquantum で性能を伸ばしているが，libquantum は，プリフェッチ・アクセスが非常に多いベンチマークであるため，今回の手法によって，上手くプリフェッチの価値を見積もる

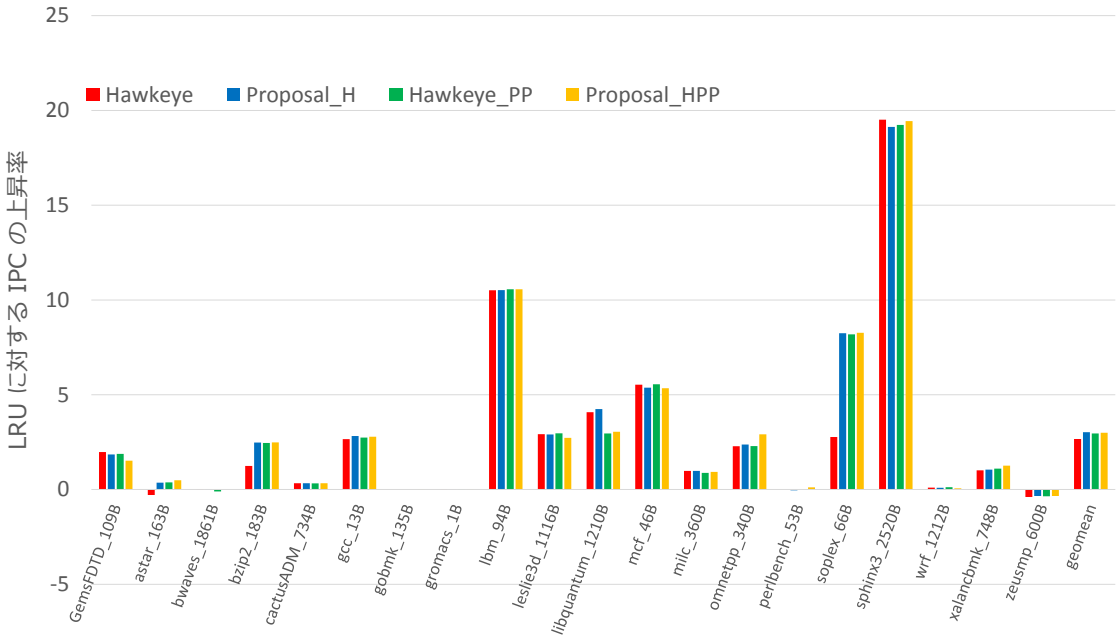

図 6.2: Hawkeye をベースとした提案手法の性能

ことができたと考えることが出来る．一方，sphinx3 では，性能を落としているが，このベンチマークでは，プリフェッチで挿入したラインへのデマンド・アクセスもプリフェッチ・アクセスも多く，大体が閾値を越える前にデマンド・アクセスが来ることでポジティブに学習していた．そのため，デマンド・アクセスが来るかなり前に，cache-friendly なラインとしてキャッシュに挿入してしまい，IPC が低下したものと考えられる．

Hawkeye_PP と，Proposal_HPP は，SHiP++ の場合と同様に，PrePromotion 単体でも性能を向上させるにも関わらず，提案と組み合わせることで，提案単体に比べて性能を落としてしまうことがわかる．

geomean は，Hawkeye が，2.66%，Proposal_H が，3.02%，Hawkeye_PP が，2.96%，Proposal_HPP が，3.00% LRU に対して， IPC が向上しており，こちらも SHiP++ の場合と同様に，Proposal_H が，最も性能が良い．

6.3.2 面積評価

ここでは，それぞれのベースに対して，どの程度追加の記憶領域が必要になるかを議論する．記憶領域以外に関しては，組み合わせ回路で構成できるため，規模がそこまで大きくならないとして今回は，無視する．

SHiP++

Proposal_S では，SHiP++ に対して，追加で記憶領域を用意する必要がない．

また，PrePromotion を用いる場合，プリフェッチャとキャッシュに多少の変更を要するが，こちらも記憶領域を増やす必要はない．

Hawkeye

Proposal_H では，P-P アクセスをカウントするカウンタを OPTgen ごとに設ける必要がある．元の Hawkeye では，OPTgen が 64 個あり，OPTgen のサイズが 1024 であるため，カウンタのために要する記憶領域は，カウンタのビット数 $\times 64 \times 1024 \div 8[\mathrm{bytes}]$ となる．今回の提案では，閾値を 9 としているため，4bit のカウンタが必要となるため，カウンタのために 32KB の記憶領域が必要となる．これは L1 キャッシュと同程度のサイズが必要になるため，多少性能が低下しても，閾値を 6 として，3bit のカウンタとするか，OPTgen の全てのエントリにカウンタを持たせず，カウンタだけ分離して構築するなどして記憶領域を削減する必要がある．

PrePromotion に関しては，SHiP++ の場合と同様に，追加の記憶領域を設ける必要はない．

第7章　おわりに

7.1　まとめ

キャッシュの置換アルゴリズムは，LRU のようなものから，RRIP などの，大容量な LLC 向けのものが求められるようになった．また，プリフェッチのことを考慮した置換アルゴリズムを考えることで，プリフェッチによる悪影響を抑え，メリットを最大限引き出す必要がある．

我々は，プリフェッチを考慮した置換アルゴリズムとして，汎用的な手法を提案した．その手法を適用させる対象として，最新の手法である SHiP++ と Hawkeye をベースとして選択した．SHiP++ をベースとした手法では，プリフェッチで挿入したラインへのデマンド・アクセスで，価値判断をデマンド用のテーブルに移行させた．Hawkeye をベースとした手法では，何度もプリフェッチ・アクセスが来るラインは，後でまたプリフェッチ・アクセスが来ると考えて追い出してもよいと判断するように変更を加えた．提案手法により，それぞれベースから性能を向上させることに成功した．また，プリフェッチを有効活用する手法である PrePromotion を同時に適用させてみた．PrePromotion は，適用させる置換アルゴリズムの性能を向上させるが，今回我々が提案した手法とは上手くかみ合わなかった．

7.2　今後の課題

Proposal_S，Proposal_H のどちらも，PrePromotion をどのように適用させればよいかを考える必要がある．また，Proposal_H に関しては，カウンタのための記憶領域を削減する方法を考える必要がある．更に，今回の 2 つの提案を 1 つの置換アルゴリズムに適用させた場合の性能について議論する必要がある．

今回の性能測定は，THE 2nd CACHE REPLACEMENT CHAMPIONSHIP の configure 2 によって行ったが，プリフェッチ存在下のマルチコア用の設定である，configure 4 でも性能を調査する必要がある．

関連図書

[1] A. Jain and C. Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 78–89, June 2016.

[2] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[3] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2011.

[4] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pp. 175–186, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, Vol. 38, No. 3, pp. 60–71, June 2010.

[6] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pp. 381–391, New York, NY, USA, 2007. ACM.

[7] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 430–441, New York, NY, USA, 2011. ACM.

[8] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, Vol. 11, No. 12, pp. 7–21, December 1978.

[9] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 63–74, Feb 2007.

[10] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. *SIGMICRO Newsl.*, Vol. 23, No. 1-2, pp. 102–110, December 1992.

[11] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Software, IEE Proceedings-*, pp. 96–96, Feb 2004.

[12] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. Ac/dc: an adaptive data cache prefetcher. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pp. 135–145, Sept 2004.

[13] Daniel A. Jiménez and Elvira Teran. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pp. 436–448, New York, NY, USA, 2017. ACM.

[14] 弘幸甲地, 英嗣入江, 修一坂井. プリフェッチラインの再参照間隔を予測するキャッシュマネジメント. Technical Report 12, 東京大学大学院情報理工学系研究科, 東京大学大学院情報理工学系研究科, 東京大学大学院情報理工学系研究科, jul 2017.

[15] K. Zhang, Z. Wang, Y. Chen, H. Zhu, and X. H. Sun. Pac-plru: A cache replacement policy to salvage discarded predictions from hardware prefetchers. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 265–274, May 2011.

[16] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Pacman: Prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 442–453, New York, NY, USA, 2011. ACM.

[17] 石井康雄, 稲葉真理, 平木敬. マップ型履歴を用いたプリフェッチ方式とキャッシュ置換方式の協調動作. 研究報告計算機アーキテクチャ（ARC）, Vol. 2010, No. 13, pp. 1–8, jul 2010.

[18] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Unified memory optimizing architecture: Memory subsystem control with a unified predictor. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pp. 267–278, New York, NY, USA, 2012. ACM.

[19] 力翠湖, 眞島一貴, 藤原大輔, 吉見真聡, 吉永努, 入江英嗣. プリフェッチ情報から再参照予測を行うキャッシュライン置き換えアルゴリズム. 情報処理学会研究報告. 計算機アーキテクチャ研究会報告, Vol. 2013, No. 20, pp. 1–7, jul 2013.

[20] 甲地弘幸, 入江英嗣, 坂井修一. D-6-14 置き換えアルゴリズムとプリフェッチが協調動作するキャッシュマネージメント・プリプロモーションの評価 (d-6. コンピュータシステム, 一般セッション). 電子情報通信学会総合大会講演論文集, Vol. 2016, No. 1, p. 68, mar 2016.

[21] 甲地弘幸, 入江英嗣, 坂井修一. Pre-promotion の動的切り替え手法の検討 (コンピュータシステム). 電子情報通信学会技術研究報告 = IEICE technical report : 信学技報, Vol. 116, No. 177, pp. 103–107, aug 2016.

[22] H. Nomura, H. Katchi, H. Irie, and S. Sakai. Stubborn strategy to mitigate remaining cache misses. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 388–391, Oct 2016.

[23] Calvin Lin Akanksha Jain. Hawkeye: Leveraging belady's algorithm for improved cache replacement. `http://crc2.ece.tamu.edu/submaterial/Hawkeye.pdf`.

[24] Aamer Jaleel Moinuddin Qureshi Vinson Young, Chia-Chen Chou. Ship++: Enhancing signature-based hit predictor for improved cache performance. `http://crc2.ece.tamu.edu/submaterial/Ship++.pdf`.

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

[26] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pp. 267–272, New York, NY, USA, 2004. ACM.

[27] THE 2ND CACHE REPLACEMENT CHAMPIONSHIP. `http://crc2.ece.tamu.edu`.

[28] E. Teran, Z. Wang, and D. A. Jimnez. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[29] The Standard Performance Evaluation Corporation. SPEC CPU2006 suite. `http://www.spec.org/cpu2006/`.

[30] Champsim simulator. `https://github.com/ChampSim/ChampSim`.

[31] Crc2_treace. `https://www.dropbox.com/sh/hh09tt8myuz0jbp/AACAS5zMWHL7coVuS-RbpUksa?dl=0`.

研究業績

口頭発表（査読なし）

1. 甲地 弘幸，入江 英嗣，坂井 修一: Pre-Promotion の動的切り替え手法の検討，情報処理学会研究報告 2016-ARC-221

2. 甲地 弘幸，入江 英嗣，坂井 修一: プリフェッチラインの再参照間隔を予測するキャッシュマネジメント，情報処理学会研究報告 2017-ARC-227

謝辞

本研究を進めるにあたり，多くの方にお世話になりました．

坂井修一教授，入江英嗣准教授とは，相談会やグループミーティングを通じて，研究の方向性や提案についてアドバイスをいただいたり，論文の添削や，学会等の発表練習を見ていただいたりと，研究全般に関してお世話になりました．名古屋大の塩谷亮太准教授には，直接の指導教官でないにも関わらず，研究やシミュレータに関する質問に答えていただきました．

また，八木原晴水さんと赤羽彩子さんによって，研究室の環境が改善され，非常に研究しやすい環境が提供されました．

研究室の同期や，先輩後輩とは，時には議論したり，談笑したりすることで，非常に有意義に研究室生活を送ることが出来ました．

本論文は，自分ひとりでは完成しえないものであると痛感すると同時に，坂井・入江研究室のみなさまに感謝したいと思います．

本論文の研究は一部，文部科学省研究費補助金 No. 25730028 による．