Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

グループ化したストリームからのフィードバックを用いたストリーム毎に最適化するストリーム・プリフェチャの高効率化

劉 兆良[1,a)]　塩谷 亮太[2]　安藤 秀樹[1]

概要：キャッシュ・ミスはコンピュータの性能を低下させる．特に，最終レベル・キャッシュ (LLC: last-level cache) のミスはペナルティが大きく，性能低下は著しい．プリフェッチは効果的な方法である．商用プロセッサで良く使われているプリフェッチャはストリームプリフェッチャである．予測が単純なため，無駄なラインを多くプリフェッチするという欠点がある．この問題を解決するため，アプリケーションや実行フェーズに応じてプリフェッチの積極度を変えることによって，効率的なプリフェッチを実現する手法として，Feedback Directed Prefetching(FDP) が提案された．FDP により，プリフェッチの効率は高められる．しかし，FDP 方式はすべてのストリームについてまとめて評価し，制御するために，各ストリームについて最適な制御とならないという欠点がある．これに対し，本研究では，この問題を解決するため，基本的には，ストリーム毎に評価し，その情報をフィードバックさせてプリフェッチの積極度を制御する．これにより，ストリーム毎に，プリフェッチの有効性を維持しつつ，無駄なメモリ・バンド幅消費を抑える．しかし，ストリームが短すぎるために，収集する情報が少ないストリームは学習できないという問題がある．これに対して，ストリームを生成する命令が同一と推定されるストリームをグループ化し評価し，その情報によってフィードバックする．SPECCPU2006 および SPECCPU2017 のベンチマークを用いて評価を行った結果，提案手法を導入することにより，FDP 方式に対し，7%ポイント/13%ポイント (SPECCPU2006/SPECCPU2017) の精度向上を達成した．また，18%ポイント/6%ポイント (SPECCPU2006/SPECCPU2017) の性能向上が得られることを確認した．

1. はじめに

キャッシュ・ミスはコンピュータの性能を低下させる．特に，最終レベル・キャッシュ (LLC: last-level cache) のミスはペナルティが大きく，性能低下は著しい．

これを抑制する手法として，データ・プリフェッチがある．プリフェッチとは，要求される前に予めキャッシュにデータを持ってくることである．これにより，ミスを未然に防ぐことができ，ペナルティを被ることを回避でき，性能を向上させることができる．

プリフェッチャに関して，多くの研究が行されている [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] ．しかし，商用プロセッサは，比較的単純なプリフェッチャがよく使われる．その一つはストリーム・プリフェッチャである．例えば，Intel Pentium 4[13]，Intel Sandy Bridge[14]，IBM POWER4, 5, 6, 7, 8[15], [16], [17], [18]，AMD Opteron[19] はストリーム・プリフェッチャを使用している．

ストリーム・プリフェッチャは，基本的にミスしたラインの前方 (または後方) を単純にプリフェッチするものである．ストリーム・プリフェッチャは，非常に有効であるが，予測が単純なため，無駄なラインを多くプリフェッチするという欠点がある．結果として，無駄なメモリ・バンド幅を消費し，電力を浪費する．

一般にミス率低下とメモリ・バンド幅消費はトレードオフの関係にある．つまり，積極的にプリフェッチを行うとミス率は低下するが，メモリ・バンド幅を多く消費する．逆に，消極的に行えば，メモリ・バンド幅の消費は抑えられるが，ミス率低下は限定的なものとなる．ミス率が増加すると，性能は低下する．

このトレードオフにおいて，アプリケーションや実行フェーズに応じてプリフェッチの積極度を与えることによって，効率的なプリフェッチを実現する手法として，Feedback Directed Prefetching(FDP)[5] が提案された．FDP は，プリフェッチに関するいくつかの評価値を使用して，積極

[1] 名古屋大学大学院工学研究科
[2] 東京大学大学院情報理工学系研究科
[a)] liu@ando.nuee.nagoya-u.ac.jp

度を制御する方式である．評価値としては，プリフェッチの精度，遅さ，およびプリフェッチによって引き起こされたキャッシュ・ポリューションの度合いを使用している．FDP により，プリフェッチの効率は高められるもの依然として，以下のような問題が残されている．すべてのストリームが合わせて評価され，平均評価値に基づいて全体の積極度を変更させている．このため，ストリーム毎に見ると，必ずしも最適ではない．

そこで，本論文では，上述の問題を解決するために，以下の手法を提案する．ストリーム毎に評価し，各ストリームについて最適な積極度に制御する．これにより，ストリーム毎に，プリフェッチの有効性を維持しつつ，無駄なメモリ・バンド幅消費を抑える．

しかし，この単純なストリーム・ベースの制御方法の導入だけでは，十分な効果は得られない．なぜなら，FDP の場合，全ストリームについて評価値が得られるので，短時間にフィードバック制御ができるが，ストリーム・ベースでは，ストリーム毎に評価値が必要なため，フィードバック制御に十分な量の情報が得られない．また，一般にストリームは十分な量の評価値が得られるほど，長くは続かず，評価値が得られないことも多い．そこで，ストリームを生成する命令が同じのストリームをグループ化し，評価する PC ベースの方法をあわせて提案する．

ストリーム・ベースで十分な評価値が得られるまでは，PC ベースで評価値を収集する．ストリームが続き，十分な評価値が得られたら，それを使ってフィードバック制御する．ストリームがフィードバック制御の前に終っても，再び現われたときは，以前の PC ベースの評価値を継承し，評価値を収集する．これにより，ストリーム・ベースの評価値の不足による制御不能状態を回避することができる．

本論文の構成は次の通りである．まず，第 2 章でストリーム・プリフェッチャの動作を説明し，そして，FDP の詳細を説明し，その欠点について述べる，続く第 3 章で本研究提案したストリーム・ベースの動作とその欠点について述べる．第 4 章でストリーム・ベースに基づいて，PC ベースを加えて，PC ベースの構造と動作について述べ，第 5 章で性能評価を行う．第 6 章でまとめる．

2. 関連研究

Jouppi はプリフェッチャによるキャッシュ汚染を防ぐストリーム・バッファを提案した [1]．これは，メモリ階層間に設置されるキューである．上位のキャッシュがミスを起こした場合に，そのアドレスに後続するラインを順にプリフェッチし，バッファに格納する．上位キャッシュのミス時にストリーム・バッファをアクセスし，ヒットすれば，そのラインをメモリ階層の上位へ移動するとともに残りのエントリをシフトし，空いたエントリに続くラインをプリフェッチする．

Feedback Directed Prefetching(FDP)[5] は，ストリーム・プリフェッチャにおいて，プリフェッチャに関する種々の性能を評価し，それをフィードバックすることによって，プリフェッチの積極度を自動的に制御する．具体的には，プリフェッチャの精度，遅さ，およびキャッシュポリューションを測定し，それらの評価値を使用してプリフェッチ距離およびプリフェッチ・ディグリーを制御する．

Variable Length Delta Prefetcher(VLDP)[8] は，長さの異なるデルタ履歴に対応したプリフェッチャである．キャッシュ・ミス間のデルタ履歴を収集し，異なるページのキャッシュ・ミスの順序を予測する．VLDP では，各デルタ履歴長に対応した複数の予測テーブルを用意する．長い履歴では一般的により正確な予測が得られるため，VLDP はミスデータのアドレスデルタに一致するエントリを持つ最長の履歴テーブルに基づいて予測する．

Indirect Memory Prefetcher(IMP)[9] は，A[B[i]] という形式の間接的なパターンをとらえるプリフェッチャである．IMP は，A[B[i]] の A の各要素のサイズとベース・アドレス (A [0] のアドレス) を識別することができる．そして，ソフトウェアが B[i] にアクセスすると，IMP は B[i+Δ] の値をプリフェッチする．そして，A の各要素のサイズとベース・アドレスを使用して，A[B[i+Δ]] のメモリ・アドレスをプリフェッチする．IMP は，新しいアプリケーション (機械学習，グラフィカル分析など) に特に役立つ．

Global History Buffer（GHB）[12] は，様々なプリフェッチ・アルゴリズムの実装に使用できるフレームワークを提供する．ここでは，プログラム・カウンタを使用して L2 キャッシュミスのストリームをローカライズする GHB PC/DC アルゴリズムを説明する．GHB 技術は，2 つのハードウェア構造，すなわちインデックステーブルとグローバル履歴バッファ（GHB）を用意する．GHB は循環バッファであり，各キャッシュミスは新しいエントリをバッファに挿入する．インデック・ステーブルは，PC をインデクスとする表であり，GHB へのポインタを保持している．インデックステーブルは，グローバル履歴内で最後に出現した PC を指す．GHB は，各バッファエントリは，最後のキャッシュ・ミスからのデルタと，現在の PC の次のインスタンスへのポインタとで構成されている．これにより，GHB 内のリンクをたどることにより，以前に見られたデルタを見つけることができる．

Ekivolos[11] は，十分な制御フロー命令を含むプリフェッチスライスを自動的に構築する事前計算プリフェッチャである．Ekivolos は，アプリケーションのバイナリだけを使用し，高精度の P スライスを自動的に構築する．この研究では，予測が困難なアクセス・パターンがある長いアクセス待ち時間のロードをターゲットにする．Ekivolos は，最適化された短いスライスを作成する従来の概念から逸脱している．これとは対照的に，より長いスライスであっても，

Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

メインスレッドより先に実行し，十分正確である限り，有用なプリフェッチを実行できる．

3. ストリーム・プリフェッチャ

本章では，本研究でベースとしたストリームプリフェッチャと本研究の先行研究を述べる．最初に，ストリームプリフェッチャの動作と欠点を説明し，その後，先行研究である FDP 方式の動作および欠点について具体的に説明する．

3.1 モデル化したストリームプリフェッチャ

本研究でベースとしたストリームプリフェッチャは，IBM POWER4 プロセッサのストリームプリフェッチャ [20] に基づいており，その実装についての詳細は文献 [1], [2], [20] に記載されている．このプリフェッチャは，ストリーム表 (ST:stream table) と呼ばれる表を用いて，ストリーム，すなわち，連続アクセスをモニターし，プリフェッチ要求を出し，キャッシュ・ブロックをメイン・メモリから，ラスト・レベル・キャッシュ（LLC）に持ち込む．このストリームプリフェッチャは，ストリーム表のエントリの数のアクセスストリームを追跡することができる．

各 ST エントリは，4 つのいずれかの状態を持つ．

(1) 無効：エントリには，追跡するためのストリームが割り当てられていない．最初は，すべてのエントリがこの状態にある．
(2) 割り当て：キャッシュ・ミスが生じたら，ST を参照し，エントリが割り当てられているかをチェックする．割り当てられていなければ，割り当てる．そして，アクセスしたライン番号 (FML: first miss line) をエントリに書き込む．
(3) トレーニング：FML より一定のライン数の範囲をトレイン・ウィンドウ（図 1 に示す）と呼び，この中にアクセスが行われたら，FML よりどちらの方向 (前方あるいは後方) のアクセスかを割り当てられた ST のエントリに記録する．トレイン・ウィンドウに 2 度連続して同じ方向のアクセスが行われたら，そのエントリのアクセスの予測方向をその方向と決定する．

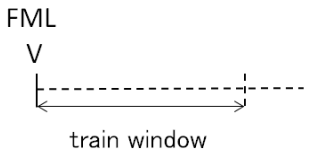


図 1 トレイン・ウィンドウ

(4) モニターとプリフェッチ要求の送出：FML より予測方向に予め定めたライン数離れたラインから一定の範囲をモニター領域と呼ぶ（図 2 参照）．モニター領域のライン数をプリフェッチ距離と呼び，この領域にアクセスがあったら，モニター領域の直後から予め決められたライン数だけプリフェッチする．このライン数をプリフェッチ・ディグリーと呼び，プリフェッチされたラインの領域を，プリフェッチ領域と呼ぶ．プリフェッチした後，モニター領域を予測方向にシフトする (モニター領域とプリフェッチ領域をプリフェッチ・ディグリー個のキャッシュブロックだけ移動させる)．

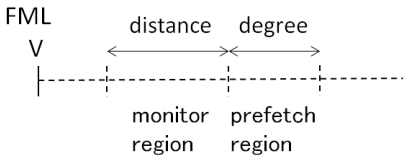


図 2 モニター領域とプリフェッチ領域

プリフェッチ距離とプリフェッチ・ディグリーは，プリフェッチャの積極性を決定する．従来のプリフェッチャ構成では，これらの値は，プロセッサの設計時に固定されている．これらが固定されたプリフェッチは，以下に示す問題がある：

(1) 無駄なラインを多くプリフェッチする．使用可能なメモリバンド幅の競合が増加する可能性がある．
(2) プリフェッチされたデータが，後にプログラムのロード命令によって必要とされるキャッシュブロックを置き換えるポリューションを引き起こす可能性がある．

3.2 FDP 方式

ストリームプリフェッチャは単純で有効的な方法であるが，無駄なラインを多くプリフェッチするという欠点がある．この問題を解決するため，Srinath らは FDP 方法を提案した．本節は FDP 方式の動作と欠点について詳細に説明する．

3.2.1 FDP 方式の概要

FDP は，ストリーム・プリフェッチャに，フィードバックメカニズムを追加して，プリフェッチの積極度を自動的に制御する方式である．FDP 方式は，フィードバック入力として 3 つの評価値すなわち，精度，遅さ，ポリューション率を使用する．定義と計算式を以下に示す．

(1) 精度：プリフェッチした全ブロックの中で実際にアクセスされたブロックの割合である．以下の式で定義される:
精度=プリフェッチされたブロックの中でアクセスされた数/プリフェッチ総数
(2) 遅さ：これは，プリフェッチャーによって生成されたプリフェッチ要求が，プリフェッチされたデータを必要とするデマンドアクセスに対してどれくらい適時に対応してないかの尺度である．ロード命令がプリフェッチデータを要求するまでに，プリフェッチデータがまだメインメモリから持ってこられていない場合，プリフェッチは遅いと言う．プリフェッチ要求が正確で

あっても，プリフェッチ要求が遅い場合，プリフェッチャはパフォーマンスを十分に向上させることができない．遅さは以下の式で定義される:
遅さ=遅延したプリフェッチ数/有効なプリフェッチ数
有効なプリフェッチ数とは，プリフェッチされたブロックのうちデマンド要求によって使用された数である．

(3) ポリューション率：以下の式で定義される:
ポリューション率=プリフェッチが原因によるデマンドミスの数/デマンドミスの数
プリフェッチにより追い出されたブロックが，ロード命令によって必要とされた場合，ミスが発生する．これをポリューションと定義する．ポリューション率が高い場合，プリフェッチされたデータによってキャッシュ内の有用なデータが追い出される可能性が高いため，プロセッサの性能が低下する可能性がある．さらに，高いポリューション率は，置換されたデータのメモリからの再フェッチを要求することとなり，より高いメモリ帯域幅消費をもたらす．

3.2.2 FDP 方式の動作

FDP は，前節で定義した精度，遅さ，およびポリューション率に基づいて，プリフェッチャ全体の積極度を動的に調整する．本節ではこれらの値の測定動作について説明する．

(1) 精度：プリフェッチ要求の精度を追跡するために，L2 キャッシュ内の各タグストアエントリにビット（$pref\_bit$）を追加する．プリフェッチされたブロックがキャッシュに挿入されると，そのブロックに関連付けられた $pref\_bit$ をセットする．プリフェッチャの精度は，2 つのハードウェア・カウンタを使用して追跡する．最初のカウンタ $pref\_total$ は，メモリに送信されたプリフェッチの数をカウントする．第 2 のカウンタ，$used\_total$ は，プリフェッチされたブロックのうち使用された数を追跡する．プリフェッチ要求がメモリに送信されると，$pref\_total$ をインクリメントする．$pref\_bit$ がセットされた L2 キャッシュブロックがデマンド要求によってアクセスされると，$pref\_bit$ をリセットし，$used\_total$ をインクリメントする．プリフェッチャの精度は，以下の式で与えられる：
精度=used_total/pref_total

(2) 遅さ：これを評価するため，MSHR[21] を用いる．MSHR は，すべてのインフライトメモリ要求を追跡するハードウェアである．メモリ要求に MSHR エントリを割り当てる前に，MSHR をチェックして，要求されたキャッシュブロックが以前のメモリ要求によって処理されているかどうかを調べる．MSHR 内の各エントリに，メモリ要求がプリフェッチャによって生成されたことを示すビット（$pref\_bit$）を用意する．プリフェッチ要求がメインメモリからのデータを待っている間に，プリフェッチしているアドレスにデマンド要求が生成された場合，プリフェッチ要求は遅いという．このような遅いプリフェッチを数えるために，ハードウェアカウンタ $late\_total$ を使用する．デマンド要求が $pref\_bit$ がセットされた MSHR エントリにヒットした場合，$late\_total$ カウンタをインクリメントし，その MSHR エントリ $pref\_bit$ をリセットする．遅さは，以下の式を与えられる：
遅さ=late_total/used_total

(3) ポリューション率：プリフェッチャによって引き起こされるデマンドミスの数を追跡するために，ブルームフィルタ [22], [23] を使用する．ブルームフィルタは，プリフェッチャによって引き起こされるデマンドミスの数を近似できる簡単でコスト効率の高いハードウェ機構である．図 3 に，この構成を示す．プリフェッチが原因で追い出されたブロックは，ブロックのアドレスに対応するフィルタのビットをセットする．そして，デマンドアクセスがキャッシュミスを発生した場合，デマンド要求のキャッシュブロックのアドレスを使用してフィルタにアクセスする．フィルタ内の対応するビットが設定されている場合は，プリフェッチャによってデマンドミスが発生したことを示す．この場合，プリフェッチャによって引き起こされるデマンドミスの総数を追跡するハードウェアカウンタ $pollution\_total$ をインクリメントし，フィルタのビットをリセットする．一方，別のカウンタ $demand\_miss\_total$ を用意し，デマンドミスの総数を数える．ポリューション率は，以下の式を与えられる：
ポリューション率=
pollution_total/demand_miss_total

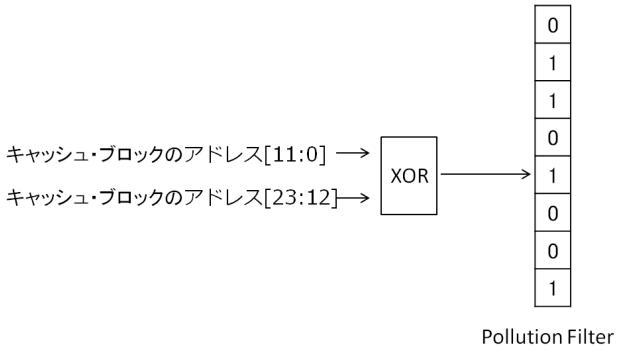


図 3 ポリューション・フィルタ

Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

3.2.3 カウンタの使用方式

本節ではフィードバック（学習と呼ぶ）において，3.2.2 節で説明したカウンタをどのように使用するかについて説明する．

学習するタイミングは，L2 キャッシュから追い出されたブロックの数で与える．このために，ハードウェアカウンタ，$eviction\_count$ を用意する．カウンタの値が静的に設定された閾値 $T_{interval}$ を超えると，学習を開始する．精度，遅れ，およびポリューション率の 3 つの評価値を計算する．これらの評価値は，プリフェッチャの積極度を調整するために使用される．

プログラムのフェーズに適応するために，各カウンタの値は次のように計算する．

今回学習のカウンタの値=(前回学習のカウンタ値+今回記録したカウンタ値)/2

今回記録したカウンタ値は，学習が終わればリセットされる．上式で計算した今回学習のカウンタ値は，次の学習に使われる．カウンタを更新するために使用される上記の式は，以前のすべての学習での動作を考慮しながら，最新の学習でのプログラムの動作により大きな重みを与える．

3.2.4 FDP 機構

本節では，FDP 機構について説明する．

$eviction\_count$ が $T_{interval}$（文献 [5] では 8192）に達すると学習を開始し，計算された精度，遅さ，およびポリューション率の値で，プリフェッチャの積極性を調整する．

表 1 に示すように，非常に消極的から非常に積極的まで，プリフェッチ距離とディグリーにより 5 つの異なる構成を用意する．この積極度は，動的構成カウンタによって与える．このカウンタは，3 ビットの飽和カウンタで，値 1 と 5 で飽和する．動的構成カウンタの初期値は 3 であり，積極度は中間的である．

表 1 ストリームプリフェッチャの設定

動的構成カウンタ	積極度	distance	degree
1	非常に消極的	4	1
2	消極的	8	1
3	中間的	16	2
4	積極的	32	4
5	非常に積極的	64	4

計算された精度，遅さ，およびポリューション率の値に基づいて動的構成カウンタの値を更新する．精度については，2 つの閾値（$A_{high} = 0.75$ と $A_{low} = 0.4$）と比較し，高，中，低に分類する．同様に，遅さについては閾値（$T_{lateness} = 0.01$）と比較し，遅延ありまたは遅延なしに分類する．最後に，ポリューション率については閾値（$T_{pollution} = 0.005$）と比較し，高，低に分類する．

表 2 に，3 つの評価値を使用してプリフェッチャの動的構成カウンタを調整する方法を詳しく示す．

表 2 FDP 方式積極度の変更表

精度	遅さ	ポリューション	動的構成カウンタの更新
高	あり	低	増加
高	あり	高	増加
高	なし	低	変更しない
高	なし	高	減少
中	あり	低	増加
中	あり	高	減少
中	なし	低	変更しない
中	なし	高	減少
低	あり	低	減少
低	あり	高	減少
低	なし	低	変更しない
低	なし	高	減少

3.2.5 FDP 方式の問題

FDP 方式はすべてのストリームについてまとめて評価し，制御するために，各ストリームについて最適な制御とならないという欠点がある．これを示すために，プリフェッチ精度の分布のばらつきを示す．具体的には，学習ポイント間において生成されたストリームについて，その精度の分布の重み付き四分位数で示す．重みは，プリフェッチされたブロックの個数である．図 4 に示す例を使って説明する．図は，ある学習ポイント間のストリームの状態を示している．「赤 1:30 個 PF 精度 50%」の意味は，このストリームでは，30 個のブロックをプリフェッチし，精度が 50%であったことを示す．同様に，「黒 1:30 個 PF 精度 75%」の意味は，このストリームでは，30 個のブロックをプリフェッチし，精度が 75%であったことを示す．矢印が長いほど，より多くのブロックがプリフェッチされたという意味である．

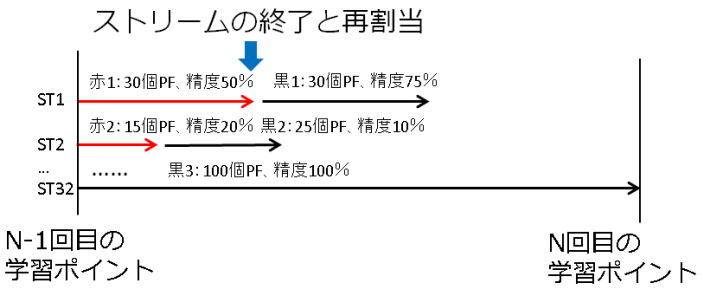


図 4 ストリームの状態

重み付き四分位数をどのようにして得たかを示す．精度データを重み付けして降順に並べる．ここで，重みは，前述したようにプリフェッチされたブロック数である．図 5 は，これをグラフにしたものである．黒 3 は，すべてのストリームの中で最も精度が高く，精度の降順にストリーム

Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

を並べた時の先頭のストリームである．その重みのプリフェッチブロック数は 100 なので，0〜100 まで黒 3 の点がプロットされる．次に精度が高いストリームは黒 1 である．プリフェッチブロック数は 30 なので，101〜130 までプロットされる．全プリフェッチブロック数は 200 なので，第 1 四分点と第 3 四分点はそれぞれ 50，150 である．これらの点における精度が第 1 四分位数と第 3 四分位数である．この例では，これらの値は 100%と 50%である．なお，黒矢印は学習ポイントでまだ終了しないストリームであり，赤矢印は終了したストリームであり，上述の計算においては，区別せず計算する．

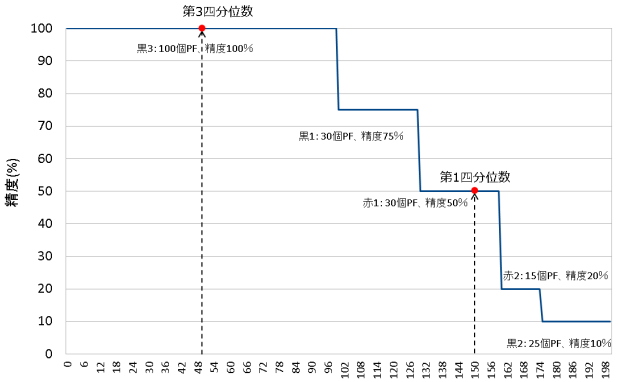


図 5 図 4 の例の重み付け精度折れ線グラフ

SPECCPU2006 の各ベンチマークプログラムについて，第 1 四分位数と第 3 四分位数の推移を求めた．その結果，次のベンチマークでばらつきが大きいことがわかった．例えば：SPECCPU2016 の *mcf*(図 6)．

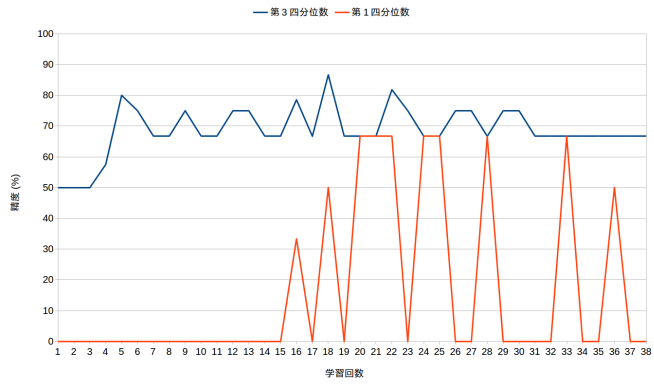


図 6 SPECCPU2016 の gobmk の第 3 四分位数と第 1 四分位数

以上の測定より，次のことが言える，FDP 方式は，高精度と低精度のストリームがあっても，全てのストリームをまとめて学習する．その結果，低精度のストリームの存在により，高精度のストリームの性能が制限される．あるいは逆に，高精度のストリームの存在により，低精度のストリームの積極度を下げることができず，メモリバイド幅を消費する．どちらも最適な状態ではない．

そして，低精度のストリームについては，多くのストリームの精度が低い場合を除き，この状況を改善することは難しい．その理由は，高精度と低精度ストリームを混在させた評価は，高精度のストリームにより多くの重みが占められるからである．通常，高精度のストリームは，有効であり，長く，多くのブロックをプリフェッチする．低精度のストリームは通常短い．したがって，低精度のストリームについて，それに最適な積極度に変更することは困難であり，無駄なメモリバイド幅を消費してしまう．

4. ストリーム・ベース手法

本章は，3.2.5 節で述べた問題を解決するストリーム・ベースの手法について説明する．しかし，この手法にも欠点があり，問題を部分的にしか解決できない．ストリーム・ベース手法について説明した後，この手法の問題点を述べる．

4.1 概要

FDP 手法は，すべてのストリームをまとめて学習し，それらの積極度をまとめて変更するので，一つ一つのストリームにとって，最適な積極度ではない可能性がある．この問題を解決するため，本研究では，ストリーム・ベースの制御法を提案する．

ストリーム・ベース手法では，ストリーム毎に，各ストリームに関する個別の情報をストリームテーブルに記録する．あるストリームがプリフェッチしたブロックの数が閾値に達すると，そのストリームのみについて学習し，積極度を変更する．このようにすれば，各ストリームを最適な積極度に変更することができ，他のストリームの影響を受けない．

本研究では，精度，カバー率，ポリューション率という三つの評価値を使用する．

(1) 精度:プリフェッチ精度は，プリフェッチした全ブロックの中で参照されたブロックの割合である．次のように定義する:
精度＝プリフェッチされたブロックのうち使用された数/プリフェッチされたブロックの数
プリフェッチの精度が高いほど，使用されないブロックをプリフェッチすることはなく，メモリバイド幅消費が少なくなる．

(2) カバー率:プリフェッチによって，どれだけのミスが回避できたかを示す．プリフェッチしなかった場合に発生するミスに対する，プリフェッチによって回避したミスの割合である．次のように定義する:
カバー率＝プリフェッチされたブロックのうち使用さ

Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

れた数/プリフェッチされたブロックのうち使用されたの数+ミス数
カバー率が高いほど，ミス率が低くなる．よって性能も高くなる．

(3) ポリューション率:プリフェッチにより追い出されたブロックが，後に必要になる割合である．次のように定義する:
ポリューション率=プリフェッチにより追い出されたブロックが必要とされた数/プリフェッチされたブロックの数
ポリューション率が高いほど，より多くの有効なブロックが追い出され，性能に悪影響を与える．追い出されたブロックは再びプリフェッチされる可能性があるが，この場合もポリューションと定義する．

4.2 構造

4.2.1 ストリーム・テーブルの構成

ストリーム・テーブルの 1 エントリの構成を図 7 に示す．エントリには，ストリーム・ベース手法のために，以下のフィールドが加えられている．

(1) *PF_LVL*(3 bits):このストリームの積極度．初期値は 3（中間的）である．
(2) *PF_Block*(8 bits):このストリームによってプリフェッチされたブロックの数．
(3) *PF_Used*(8 bits):このストリームによってプリフェッチされたブロックが使用された数．
(4) *PF_Miss*(16 bits):このストリームが存在している時，発生したミス数．
(5) *PF_Poll*(8 bits):このストリームが原因で，ポリューションが発生した回数．
(6) *ST_Num*(5 bits):このストリームの番号．

従来のフィールド			
PF_Block	PF_Used	PF_Miss	PF_Poll
PF_lvl	ST_Num		

図 7 ストリーム・テーブルのエントリ

4.2.2 キャッシュ・ブロックの構成

キャッシュ・ブロックの構成を図 8 に示す．以下のフィールドが加えられている．

(1) *V_PF*(1 bit):このブロックがプリフェッチされたことを示す．この時，以下の *ST_Num* のデータは有効である．
(2) *ST_Num*(5 bits):このブロックをプリフェッチしたストリームの番号．

4.2.3 ポリューション・テーブルの構成

ポリューション・テーブルの 1 エントリの構成を図 9 に示す．

従来のフィールド	
V_PF	ST_Num

図 8 キャッシュ・ブロック

(1) *V_PF*(1 bit):対応するブロックがプリフェッチにより追い出されたことを示す．この時，以下の *ST_Num* のデータは有効である．
(2) *ST_Num*(5 bits):対応するブロックを追い出したストリームの番号．

V_PF	ST_Num

図 9 ポリューション・テーブルのエントリ

4.3 動作

ストリーム・ベース手法の動作を，図 10 に示す全体構成を用いて説明する．

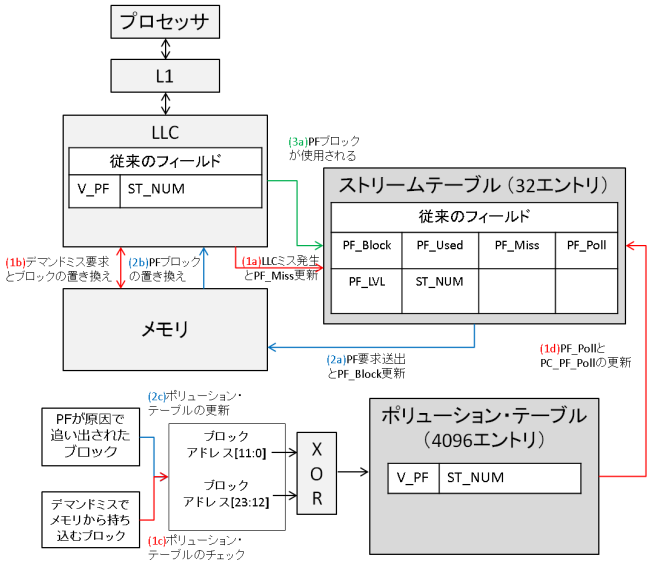


図 10 ストリーム・ベース手法の全体構造図

以下の三つの動作がある．

(1) LLC ミスが生じた場合 : (1a) ミスアドレスをストリームテーブルに送信する．ストリームテーブルでは，各エントリの *PF_Miss* をインクリメントする．ただし，*PF_Block* が 0 のエントリ (プリフェッチはまだ開始していない) の *PF_Miss* はインクリメントしない．
(1b)LLC ミスにより，メモリからキャッシュに持ち込むブロックのアドレスを用いて，(1c) 以下の計算で得られるビット列をインデクスとして，ポリューション・テーブルにアクセスする．
$ADDR[11:0]$ xor $ADDR[23:12]$
(1d) ポリューション・テーブルの *V_PF* が 1 の場合，ストリームテーブルの *ST_NUM* 番目のエントリの

PF_Poll をインクリメントする．その後，ポリューション・テーブルの V_PF，ST_NUM をリセットする．

(2) プリフェッチが生じた場合：ミスアドレスがストリームのモニター領域にアクセスした場合，(2a) プリフェッチ要求を出力する．そして，ストリームテーブルのこのストリームに対応するエントリ PF_Block をインクリメントする．

(2b) プリフェッチされたブロックはキャッシュに入れられる．キャッシュ・ブロックの V_PF を 1 にセットし，ストリームの番号をブロックの ST_NUM に書き込む．

(2c) プリフェッチが原因で追い出されたブロックは，ブロックのアドレスを用いて，以下の計算で得られるビット列をインデクスとして，ポリューション・テーブルにアクセスする．

$ADDR[11:0]$ xor $ADDR[23:12]$

アクセスするエントリの V_PF を 1 にセットし，このブロックを追い出したストリームの番号を ST_NUM に書き込む．

(3) LLC にヒットした場合：ブロックが使用されると，(3a) キャッシュ・ブロックの V_PF が 1 の場合，ストリームテーブルの ST_NUM 番目のエントリの PF_Used をインクリメントする．その後，キャッシュ・ブロックの V_PF，ST_NUM をリセットする．

4.4 学習

各ストリームについて，PF_Block が $N = 256$ に達すると，精度，カバー率，ポリューション率を計算し，積極度変更表に従って，積極度，すなわち PF_LVL を変更する．その後，PF_Block，PF_Used，PF_Miss，PF_Poll をリセットする．

計算された精度は，閾値（$A\_High = 0.75$，$A\_Very\_High = 0.95$）と比較され，低，高または超高に分類する．同様に，計算されたカバー率は閾値（$C\_High = 0.6$，$C\_Very\_High = 0.95$）と比較され，低，高または超高に分類する．最後に，計算されたポリューション率は閾値（$T\_Pollution = 0.05$）と比較され，高または低に分類する．

表 3 に，3 つの評価値を使用した積極度の変更方法を示す．プリフェッチの精度が低い場合，無駄なブロックを多くプリフェッチし，メモリバイド幅を消費するので，積極度を減少させる．精度が高く，カバー率が低い原因は二つある．一つは，プリフェッチが消極的すぎることが原因である．この場合，より積極的にしてカバー率を上げる．もう一つは，ポリューションが原因で，有効なブロックが追い出されている場合である，この場合，積極度を減少する．精度とカバー率の両方が高い場合，良好な状態であり，積極度は変更しない．しかし，精度，カバー率がともに 95%以上で，ポリューションも低い場合，さらに積極度を増加させる価値があるため，積極度を増加させる．

表 3 積極度変更表

精度	カバー率	ポリューション率	積極度の更新
超高	超高	低	増加
高	高	低	変更しない
高	高	高	変更しない
高	低	低	増加
高	低	高	減少
低	高	低	減少
低	高	高	減少
低	低	低	減少
低	低	高	減少

4.5 欠点

これまで説明したストリーム・ベース手法を評価したところ，その結果は期待したほど良くはなかった．具体的には，多くのベンチマークで精度が非常に低かった．例えばベンチマーク SPECCPU2006 の $milc$ では，精度はわずか 30% であり，$astar$ ではわずか 15% であった．この原因は，多くのストリームではプリフェッチしたブロック数が少なく，学習の閾値（$N = 256$）に達することができないことである．これにより，コントロールされないままストリームは中間的積極度でプリフェッチを続けてしまう．

図 11 に，各ベンチマークについて，学習の閾値に達したかそうでないかのストリームの内訳けを示す．青い棒は学習できなかったストリームの割合であり，赤い棒は学習できたストリームの割合である．平均として，わずか 5%のストリームしか学習できていない．

図 12 に，学習できなかったストリームについて，プリフェッチしたブロック数の分布を示す．

同図よりわかるように，平均で 85%のストリームは 31 ブロックもプリフェッチしていない．

これまでの評価では 32 エントリのストリーム・テーブルを使用した．エントリの数が少なにことにより，ストリームが非常に短いということも考えられるため，4096 エントリのストリーム・テーブルを使用して，評価した．結果を図 13 に示す．

同図よりわかるように，ベンチマーク $cactusADM$，$GemsFDTD$，$leslie3d$ で学習できたストリームの割合は向上した．しかし，平均としては，わずか 8%のストリームしか学習できていない．学習できなかったストリームのうち，90%のストリームは 31 ブロックもプリフェッチしていない．つまりストリームは非常に短いということである．したがって，学習させるために単純に閾値 N を減少し

Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

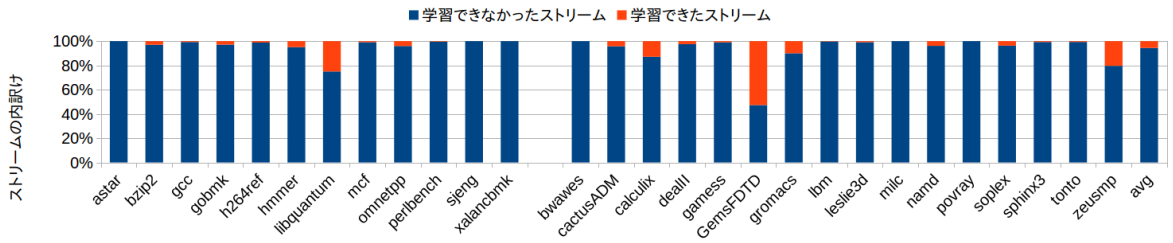


図 11 ストリームの学習率 (ストリームテーブル 32 エントリ)

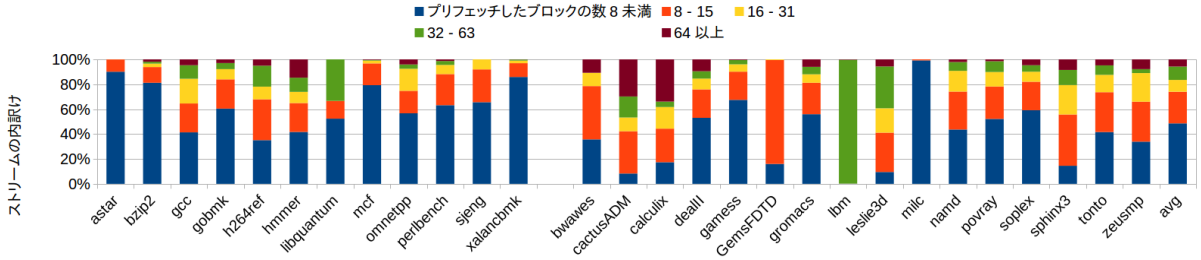


図 12 学習できなかったストリーム数の分布 (ストリームテーブル 32 エントリ)

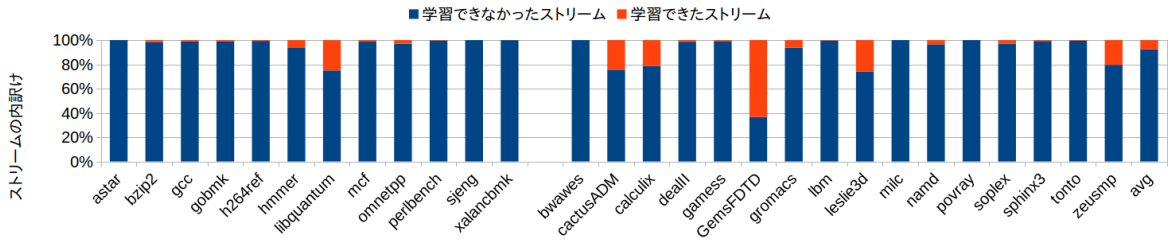


図 13 ストリームの学習率 (ストリームテーブル 4096 エントリ)

ても（例えば N は 128 あるいは 64 に減少），あるいはストリーム・テーブルのエントリを増加させても（例えばストリーム・テーブルのエントリは 64 あるいは 128 に増加），問題は解決できない．このため，これらのストリームはコントロールされず，多くの無駄なブロックをプリフェッチし，メモリバイド幅を消費してしまう．

5. PC ベース方式

5.1 概要

前述したように，多くのストリームは，プリフェッチ回数が少なく，学習点に到達できない．ストリーム・ベース手法はこのようなストリームに対処することができない．そこで，ストリームのプリフェッチが開始する前に，このストリームの適切な積極度を推定することを考えた．図 14 に示すように，ストリームには，プリフェッチが開始される前に利用可能な情報が 3 つある．それは $FML(first\ miss\ line)$ とプリフェッチの方向を決定する 2 つのミスである．異なるストリームでは，3 つのミスのアドレスは異なるが，3 つのミスの PC がそれぞれ同じストリームは，プリフェッチの性能（精度，カバー率など）について同様の傾向があると考えられる．本章では PC ベース方式を提案する．

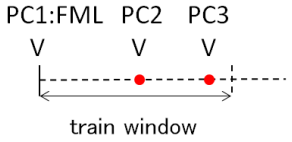


図 14 FML とプリフェッチ方向を決める二つのミス

PC ベース方式では，同じ三つの PC(PC1 = PC2 = PC3 ではなく，ストリーム A の三つの PC とストリーム B の三つの PC は同じという意味である) を持つストリームの情報をまとめて PC テーブルに収集して，評価する．また，同じ三つの PC を持つ新しいストリームに対しては，PC テーブル内の情報を用いて，プリフェッチが開始する前に，積極度を設定する．ここで，PC テーブルに収集された情報は，まだ学習点に到達していないストリームに対して提供される．学習点に到達したストリームについてはストリーム・ベース手法のみを使用する．

PC ベース方式では，複数のストリームは一つのストリームのように見える．一つ一つのストリームが短い場合でも，

Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

複数のストリームをまとめて評価すれば，学習点に達することができる．これにより，ストリーム・ベース方式の評価値の不足による制御不能状態を回避することができる．

利用可能なデータは三つあるが，本研究は，*FML* の PC とプリフェッチの方向を決定する最初のミスの PC のみ使用する．

5.2 PC ベースとストリーム・ベース方式を組み合わせた方式の構造

5.2.1 ストリーム・テーブルの構成

ストリーム・テーブルの 1 エントリの構成を図 15 に示す．エントリには，PC ベース手法のために，以下のフィールドが加えられている．

(1) *PC_table_index*(8 bits):当該ストリームが対応する PC テーブルのインデクス．

(2) *Not_Trained*(1 bit):当該ストリームが学習点に到達したかどうか．

PC_table_index を以下の計算より生成し，書き込む．

PC1[9:2] xor (PC2[9:2] を 4 ビット右循環シフト)

ここで PC1 は *FML* の PC であり，PC2 はプリフェッチの方向を決定する最初のミスの PC である．

従来のフィールド			
PF_Block	PF_Used	PF_Miss	PF_Poll
PF_lvl	ST_Num	PC_table_index	Not_Trained

図 15 ストリーム・テーブルのエントリ

5.2.2 キャッシュ・ブロックの構成

キャッシュ・ブロックの構成を図 16 に示す．以下のフィールドが加えられている．

(1) *V_PC*(1 bit):当該ブロックをプリフェッチしたストリームはまだ学習点に到達していないことを示す．この時，以下の *PC_table_index* のデータは有効である．

(2) *PC_table_index*(8 bits):当該ブロックをプリフェッチしたストリームが対応する PC テーブルのインデクス．

従来のフィールド	
V_PF	ST_Num
V_PC	PC_table_index

図 16 キャッシュ・ブロック

5.2.3 ポリューション・テーブルの構成

ポリューション・テーブルの 1 エントリの構成を図 17 に示す．以下のフィールドが加えられている．

(1) *V_PC*(1 bit):当該ブロックを追い出したストリームはまだ学習点に到達していないことを示す．*PC_table_index* のデータは有効であることを示す．

(2) *PC_table_index*(8 bits):当該ブロックを追い出したストリームが対応する PC テーブルのインデクス．

V_PF	ST_Num
V_PC	PC_table_index

図 17 ポリューション・テーブルのエントリ

5.2.4 PC テーブルの構成

PC テーブルの構造を図 18 に示す．

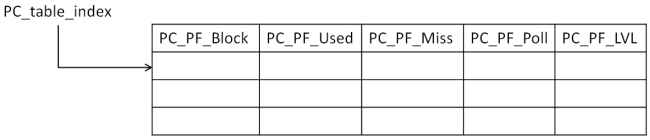


図 18 PC テーブル

(1) *PC_PF_LVL*(3 bits):当該ストリーム群の積極度．初期値は 3（中間的）である．

(2) *PC_PF_Block*(7 bits):当該ストリーム群によってプリフェッチされたブロックの数．

(3) *PC_PF_Used*(7 bits):当該ストリーム群によってプリフェッチされたブロックが使用された数．

(4) *PC_PF_Miss*(16 bits):当該ストリーム群が存在している時，発生したミス数．

(5) *PC_PF_Poll*(7 bits):当該ストリーム群が原因で，ポリューションが発生した回数．

5.3 PC ベースとストリーム・ベース方式を組み合わせた手法の動作

PC ベースとストリーム・ベース方式を組み合わせた手法の動作を図 19 を用いて説明する．

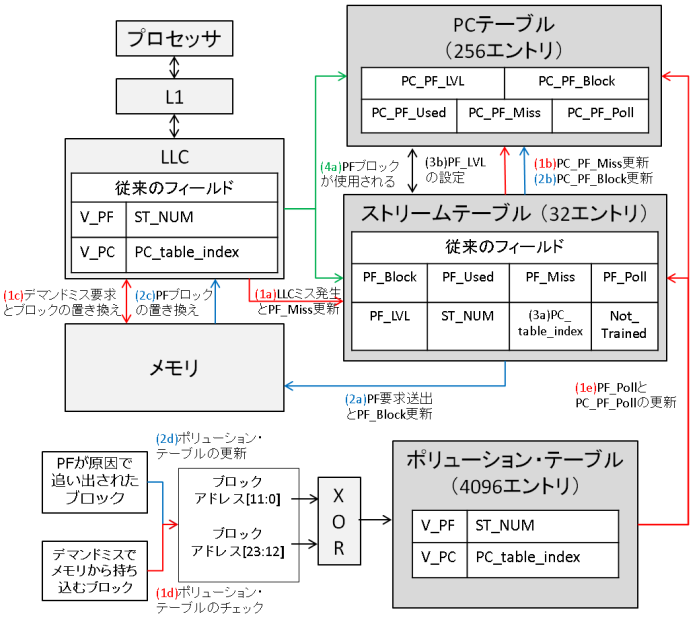


図 19 PC・ベースとストリーム・ベースを組み合わせた手法の全体構成図

以下の三つの動作がある．

(1) LLC ミスが生じた場合:(1a) ミスアドレスをストリームテーブルに送信し，ストリームテーブルの各エントリの PF_Miss をインクリメントする．ただし，PF_Block が 0 のエントリ (プリフェッチはまだ開始されていない) の PF_Miss はインクリメントしない．(1b) 同時に，まだ学習点に到達していない ($Not\_Trained = 1$) 場合，PC テーブルの PC_table_index 番目のエントリの PC_PF_Miss をインクリメントする．ただし，同じ PC_table_index を持つストリームが複数ある場合は，1 回だけカウントする（例えば，図 20 のストリーム 2 と 32）．

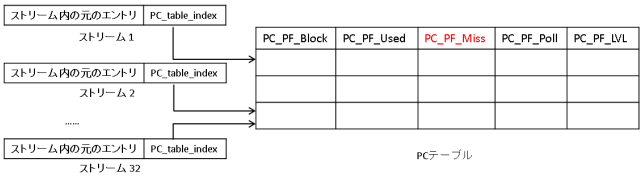


図 20 PC_PF_Miss の更新

(1c)LLC ミスにより，メモリからキャッシュに持ち込むブロックのアドレスを用いて，(1d) 以下の計算で得られるビット列をインデクスとして，ポリューション・テーブルにアクセスする．

$ADDR[11:0]$ xor $ADDR[23:12]$

(1e) ポリューション・テーブルの V_PF が 1 の場合，ストリームテーブルの ST_NUM 番目のエントリの PF_Poll をインクリメントする．V_PC が 1 の場合，PC テーブルの PC_table_index 番目のエントリの PC_PF_Poll をインクリメントする．その後，ポリューション・テーブルの V_PF，ST_NUM，V_PC と PC_table_index をリセットする．

(2) プリフェッチが生じた場合:ミスアドレスがストリームのモニター領域にアクセスした場合，(2a) プリフェッチ要求を出力する．そして，ストリームテーブルのこのストリームに対応するエントリ PF_Block をインクリメントする．(2b) 同時に，このストリームがまだ学習点に到達していない ($Not\_Trained = 1$) 場合，PC テーブルの PC_table_index 番目のエントリの PC_PF_Block をインクリメントする．

(2c) プリフェッチされたブロックはキャッシュに入れられる．キャッシュ・ブロックの V_PF を 1 にセットし，ストリームの番号をブロックの ST_NUM に書き込む．もし，ストリームがまだ学習点に到達していない ($Not\_Trained = 1$) 場合，キャッシュ・ブロックの V_PC を 1 にセットし，ストリームの PC_table_index をブロックの PC_table_index に書き込む．

(2d) プリフェッチが原因で追い出されたブロックのアドレスを用いて，以下の計算で得られるビット列をインデクスとして，ポリューション・テーブルにアクセスする．

$ADDR[11:0]$ xor $ADDR[23:12]$

アクセスするエントリの V_PF を 1 にセットし，このブロックを追い出したストリームの番号を ST_NUM に書き込む．もし，ストリームがまだ学習点に到達していない ($Not\_Trained = 1$) 場合，V_PC を 1 にセットし，ストリームの PC_table_index をポリューション・テーブルの PC_table_index に書き込む．

(3) プリフェッチが生じなかった場合 : LLC ミスアドレスがトレイン・ウィンドウにアクセスした場合で，トレイン・ウィンドウに 2 度連続して同じ方向のアクセスが行われたら，そのエントリのアクセスの予測方向をその方向と決定する．(3a)PC_table_index を生成し，書き込む．(3b)PC テーブルの PC_table_index 番目のエントリの PC_PF_LVL をアクセスする．PC テーブルの PC_PF_LVL を当該ストリームの積極度として設定する．

(4) LLC にヒットした場合 : ブロックが使用された場合で，(4a) キャッシュ・ブロックの V_PF が 1 の場合，ストリームテーブルの ST_NUM 番目のエントリの PF_Used をインクリメントする．キャッシュ・ブロックの V_PC が 1 の場合，PC テーブルの PC_table_index 番目のエントリの PC_PF_Used をインクリメントする．その後，キャッシュ・ブロックの V_PF，ST_NUM，V_PC と PC_table_index をリセットする．

5.4 学習

PC ベース方式の学習とストリーム・ベース方式の学習について，以下に説明する．

(1) ストリーム・ベースの学習 : ストリームテーブルの各エントリによって，PF_Block が $N = 256$ に達すると，精度，カバー率，ポリューション率を計算し，積極度変更表によって，PF_LVL を変更する．その後，PF_Block，PF_Used，PF_Miss，PF_Poll をリセットし，$Not\_Trained$ を 0 にセットする．

(2) PC ベースの学習 : PC テーブルの各エントリにおいて，PC_PF_Block が $M = 128$ に達すると，精度，カバー率，ポリューション率を計算し，積極度変更表によって，PC_PF_LVL を変更する．その後，PC_PF_Block，PC_PF_Used，PC_PF_Miss，PC_PF_Poll をリセットする．

閾値と積極度変更は 4.4 節の表 3 に示したとおりである．

Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17

6. 評価

本章では，前章で説明した方式についてシミュレーションによって，IPC，BPKI，精度およびカバー率を求めて，評価する．そして，ハードウェアコストを評価する．

6.1 評価環境

評価には，SimpleScalar Tool Set Version 3.0a[24] をベースに提案手法を実装したシミュレータを用いた．命令セットには Alpha ISA を用いた．ベンチマーク・プログラムとして，SPECCPU2006 と SPECCPU2017 を使用した．MPKI が 1 以上のベンチマークを表 4 に示す．プログラムは gcc ver.4.5.3 でコンパイルし，コンパイル･オプションには -O3 を用いた．プログラムの入力セットには SPEC2006 と SPEC2017 それぞれ ref データ・セット，refspeed データ・セットを使用し，プログラムの先頭 16G 命令をスキップした後の 100M 命令について測定した．性能の評価に用いたプロセッサの基本構成を表 5 に示す．各評価のパラメーターの値は表 6 に示す．

表 4　MPKI が 1 以上ベンチマーク

SPECCPU2006	SPECCPU2017
astar, mcf, omnetpp xalancbmk, cactusADM milc, soplex, sphinx3 perlbench, lbm, leslie3d	mcf, omnetpp xalancbmk, lbm

表 5　ベースプロセッサの構成

Pipeline width	8-instructions wide for each of fetch, decode, issue, and commit
Reorder buffer	256 entries
Issue queue	128 entries
Load/store queue	128 entries
Physical registers	256 for int and fp
Branch prediction	12-bit history gshare, 2048-set 4-way BTB, 10-cycle misprediction penalty
Function units	4 iALU, 2 iMULT/DIV, 2 fpALU, 2 fpMULT/DIV/SQRT
L1 I-cache	32KB, 8-way, 64B line
L1 D-cache	32KB, 8-way, 64B line, 2-cycle hit latency, non-blocking
L2 cache	2MB, 16-way, 64B line, 12-cycle hit latency
Main memory	300-cycle min. latency, 16B/cycle bandwidth
Data prefetchers	stream-based: 32-stream tracked, prefetch to L2 cache

表 6　評価のパラメーター

ストリーム・ベースの学習点 N	256
PC ベースの学習点 M	128
ストリームテーブルのサイズ	32 エントリ
PC テーブルのサイズ	256 エントリ
ポリューション・テーブルのサイズ	4096 エントリ
精度の閾値	0.75, 0.95
カバー率の閾値	0.6, 0.95
ポリューションの閾値	0.05

6.2 評価モデル

評価したプロセッサ・モデルを以下に示す．

- **FDP-Base**: 文献 [5] で使用している評価値を用いた FDP 方式．
- **FDP-new_adjustment**: 本研究で提案した三つの評価値を使用する FDP 方式．
- **PC-Base**: 提案手法のストリーム・ベース方式を使用せず，PC ベース方式のみ使用するモデル (プリフェッチの積極度は PC テーブルにより，変更する)
- **ST-Base+PC-Base**: 提案手法を適用するモデル．

6.3 評価結果

6.3.1〜6.3.4 節で，それぞれ精度，カバー率，IPC，BPKI を評価する．最後に 6.3.5 節で，4 つのモデルをまとめる．

6.3.1 精度

図 21 に，各モデルの SPECCPU2006 と 2017 におけるプリフェッチ精度を示す．

平均で，提案方式である ST-Base+PC-Base では，FDP-Base に対して，精度が 5%ポイント (SPECCPU2006)/13%ポイント (SPECCPU2017) 向上した．*astar*-2006 の精度は 17%から 52% に向上した．*fotonik*3*d*-2017 の精度は 4%から 82% に向上した．

FDP-Base の精度と FDP-new_adjustment の精度はほとんど変わらなかった．つまり，評価値を変更しても，精度に大きな影響はないことを示す．

全部のストリームを一緒に評価することが問題であることを示す．例えば，ベンチマーク *gobmk*-2006，*h*264*ref*-2006，*mcf*-2006，*astar*-2006，*gcc*-2006，*namd*-2006，*tonto*-2006，*omnetpp*-2017，*perlbench*-2017．本研究の方法を使用後，精度は改善された．

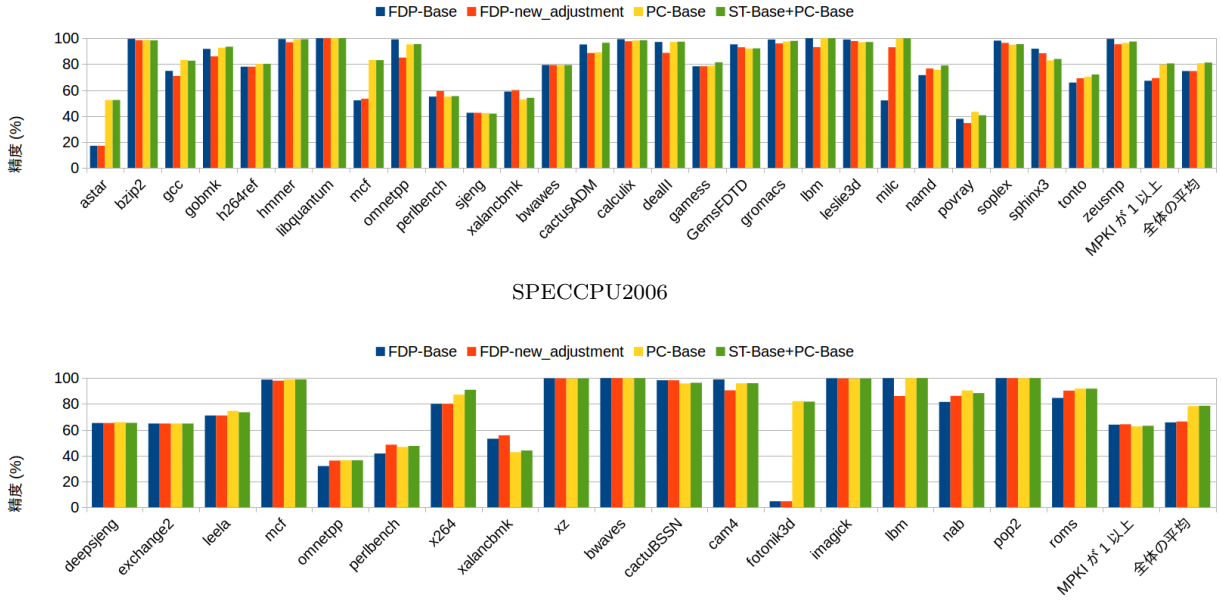
ST-Base+PC-Base 方式では，PC-Base 方式に対して，ほぼ同じ精度が得られた．差は 1%ポイント以内である．

MPKI の高いベンチマークについて，ST-Base+PC-Base 方式では，FDP-Base に対して，SPECCPU2006 の精度は平均 13%ポイント向上したが，SPECCPU2017 ではほぼ同じ精度となった．

6.3.2 カバー率

図 22 に，各モデルの SPECCPU2006 と 2017 におけるカバー率を示す．

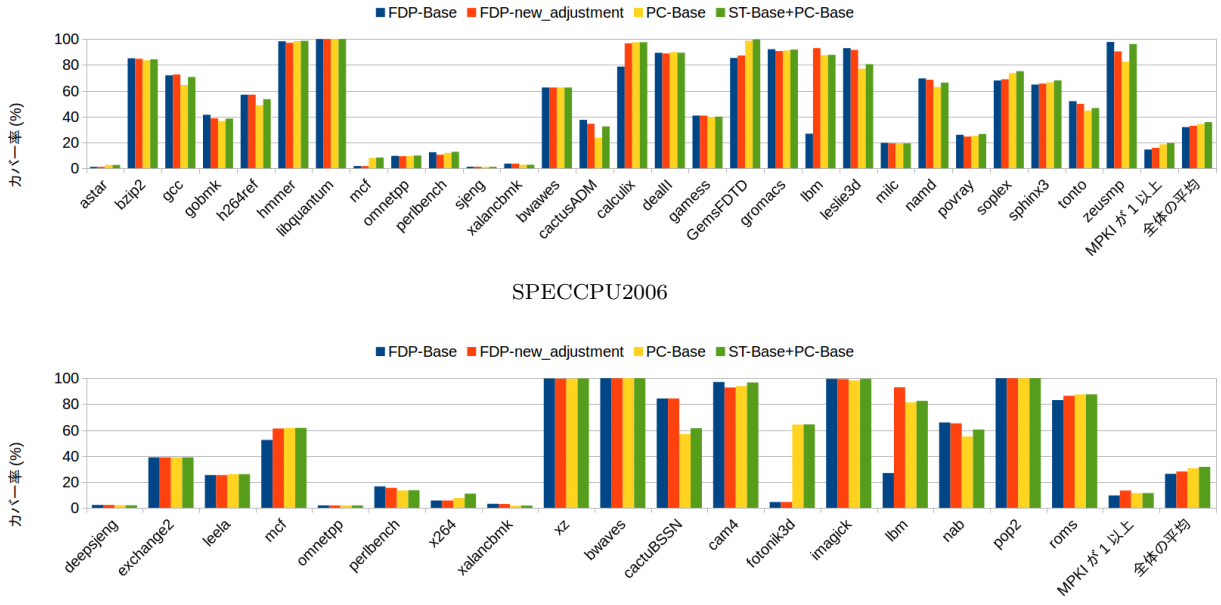
Vol.2019-ARC-235 No.21
Vol.2019-SLDM-187 No.21
Vol.2019-EMB-50 No.21
2019/3/17



SPECCPU2006

SPECCPU2017

図 21 精度



SPECCPU2006

SPECCPU2017

図 22 カバー率

平均で，提案方式である ST-Base+PC-Base では，FDP-Base に対して，カバー率が 4%ポイント (SPECCPU2006)/5%ポイント (SPECCPU2017) 向上した．*lbm*-2006 のカバー率は 27%から 87% に向上した．*fotonik3d*-2017 の精度は 4%から 64% に向上した．

MPKI の高いベンチマークについては，ST-Base+PC-Base 方式では，FDP-Base に対して，カバー率が 5%ポイント (SPECCPU2006)/2%ポイント (SPECCPU2017) 向上した．ベンチマークによっては，FDP-Base のカバー率より FDP-new_adjustment のカバー率のほうが高い (平均では，1%ポイント (SPECCPU2006)/2%ポイント (SPECCPU2017))．*calculix*-2006，*lbm*-2006，*mcf*-2017，*lbm*-2017 のカバー率は大幅向上する．つまり，文献 [5] の方法ではカバー率を評価値に使用しないので，高カバー率が保証できない．カバー率が保証できなければ，性能が低下してしまう．

ST-Base+PC-Base 方式と PC-Base 方式のカバー率はほとんど変わらない (2%ポイント (SPECCPU2006)/1%ポイ

ント (SPECCPU2017) 向上).

6.3.3 IPC

図 23 に各モデルの IPC について示す．また，図**??**に，プリフェッチを行われない場合からの増加率 (平均値のみ) を示す.

平均で，提案方式である ST-Base+PC-Base では，FDP-Base に対して，IPC が 18%ポイント (SPECCPU2006)/6%ポイント (SPECCPU2017) 向上した.

FDP-Base はカバー率を評価しないので，FDP-new_adjustment の IPC は FDP-Base より，9%ポイント (SPECCPU2006)/5%ポイント (SPECCPU2017) 向上した．カバー率が 1% ポイントだけ向上しても，パフォーマンスに大きな影響を与える (例えば，*soplex*-2006 の IPC は 4%ポイント向上し，*sphinx*3-2006 の IPC は 16%ポイント向上する).

ST-Base+PC-Base 方式では，PC-Base 方式に対して，SPECCPU2006 の IPC が平均 3%ポイント向上したが，SPECCPU2017 ほぼ同じ IPC となった.

MPKI の高いベンチマークについて，ST-Base+PC-Base 方式では，FDP-Base に対して，IPC が 18%ポイント (SPECCPU2006)/22%ポイント (SPECCPU2017) 向上した.

6.3.4 BPKI

本節では，各モデルの BPKI について評価する．$BPKI$[5] は以下の式で得られる：

$$Memory\ Bus\ Accesses\ per\ thousand\ retired\ Instructions = (prefetches + L2\ misses\ caused\ due\ to\ demand\ accesses) / (100M\ Instructions / 1000)$$

$BPKI$(プリフェッチなしの場合と比べた増加率) を図 25 に示す.

平均で，提案方式である ST-Base+PC-Base では，FDP-Base に対して，BPKI が 1%ポイント (SPECCPU2006)/4%ポイント (SPECCPU2017) 減少した.

一方，FDP-new_adjustment の BPKI が増加した．この原因は，FDP-new_adjustment 方法は，より高いカバー率を達成するため，より多くのブロックがプリフェッチされる一方，精度は低いで，このためより多くの無駄なラインをプリフェッチし，BPKI が増加した.

ST-Base+PC-Base 方式では，PC-Base 方式に対して，BPKI が 0.12%ポイント (SPECCPU2006 と SPECCPU2017) 減少した.

MPKI の高いベンチマークについて，ST-Base+PC-Base 方式と，FDP-Base では，SPECCPU2006 と SPECCPU2017 はほとんど変わらない結果となった.

6.3.5 要約

- FDP-Base と FDP-new_adjustment：FDP-new_adjustment 方式は，より高いカバー率を達成するため，より積極的に積極度を変更し，より多くのブロックがプリフェッチされる．結果として，カバー率がより高く，IPC がより高いが，BPKI が増加してしまう．二つの方法は全部のストリームをまとめて評価するので，精度はほぼ同じである.
- FDP-Base と ST-Base+PC-Base：提案手法である ST-Base+PC-Base はストリーム毎に評価し，最適な積極度に変更するため，カバー率がより高く，IPC がより高い．精度も保証するため，BPKI がより低い.
- ST-Base+PC-Base と PC-Base：これら 2 つの方法の結果は平均では同じように見えるが，実際にはベンチマークによっては差異が大きい．例えば，PC ベースでは *GemsFDTD*-2006，*leslie*3*d*-2006，*soplex*-2006，*sphinx*3-2006，*zeusmp*-2006 でカバー率と IPC が大幅に低下してしまう．これは，PC・ベース方式がストリーム・ベースを使用しない場合，学習できるストリームと共に評価するためである．学習できないストリームを，プリフェッチ効果が良く学習できるストリームとまとめて評価すると，結果として，無駄なブロックを多くプリフェッチし，ポリューションを引き起こし，性能が低下してしまう.

6.4 ハードウェアコスト

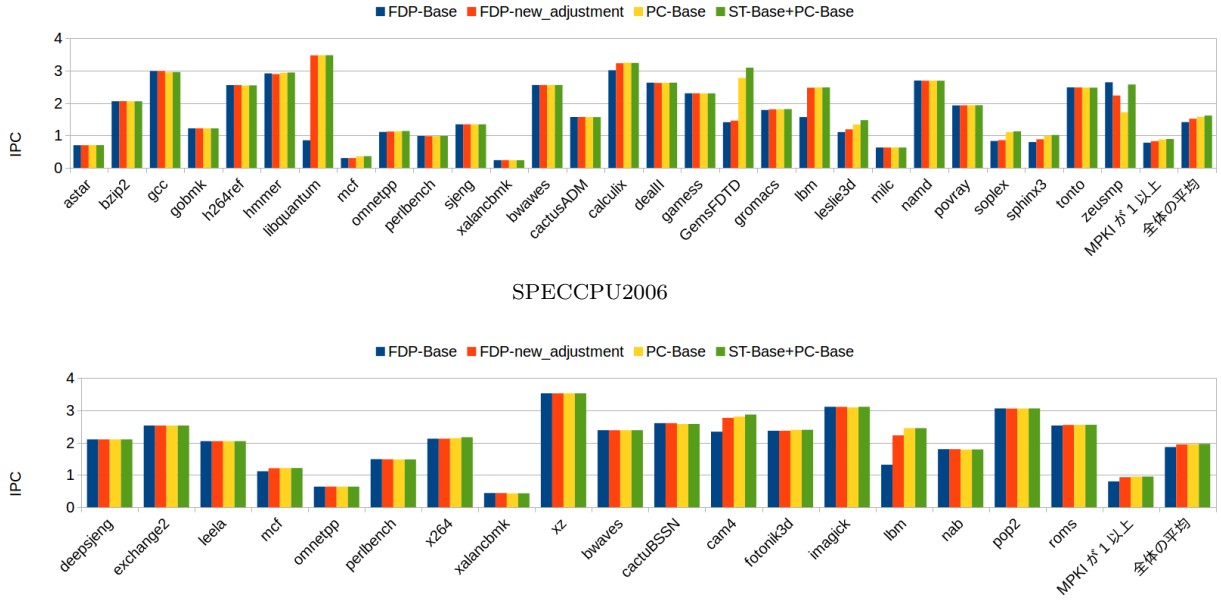
ハードウェア・コストについて示す，FDP 方式ではコストは 4.54KB であるのに対して，本手法では 68.97KB と大きい．しかし，これは 2MB L2 キャッシュのデータストアサイズ（タグは含まれていない）の 3.36%であり，十分小さいといえる．詳細を表 7 に示す.

7. まとめ

FDP 方式はすべてのストリームについてまとめて評価し，制御するために，各ストリームについて最適な制御とならないという欠点がある．本研究は，この問題を解決するため，ストリーム・プリフェッチャの効率を向上させる手法を提案した．基本的には，ストリーム毎に評価し，その情報をフィードバックさせてプリフェッチの積極度を制御する．これにより，ストリーム毎に，プリフェッチの有効性を維持しつつ，無駄なメモリ・バンド幅消費を抑える．そして，ストリームが短すぎるために収集する情報が少ないストリームについては，ストリームを生成する命令が同一と推定されるストリームをグループ化し評価し，その情報によってフィードバックする.

SPECCPU2006 および SPECCPU2017 のベンチマークを用いて評価を行った結果，提案手法を導入することにより，FDP 方式に対し，7%ポイント (SPECCPU2006)/13%ポイント (SPECCPU2017) の精度向上を達成した．また，平均して 18%ポイント (SPECCPU2006)/6%ポイント (SPECCPU2017) の性能向上が得られることを確認した.

謝辞 本研究の一部は，日本学術振興会 科学研究費補助金基盤研究 (C)（課題番号 16K00070）による補助のもと



SPECCPU2006

SPECCPU2017

図 23 IPC

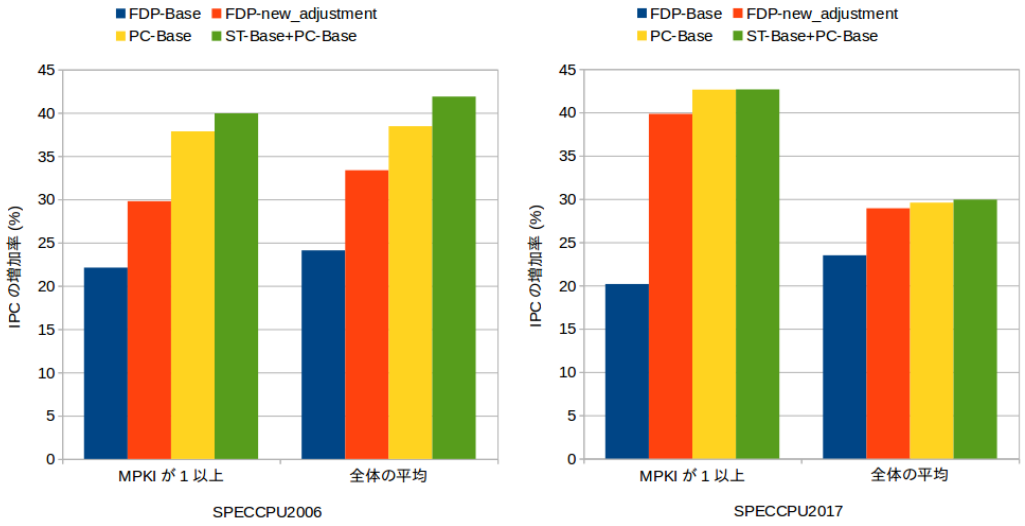


図 24 IPC の増加率 (PF なしをベースとして)

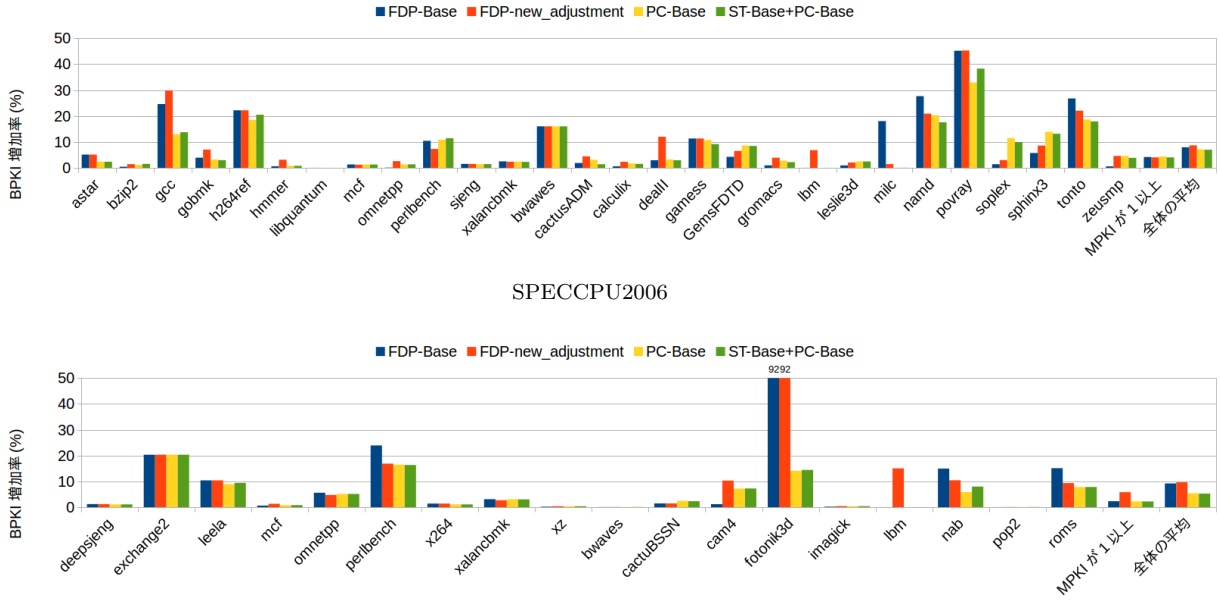
表 7 ハードウェア・コスト

(a) 本手法のハードウェア・コスト

ストリームテーブル	2 entries * 57bits/entry = 1824 bits
PC テーブル	256 entries * 40 bits/entry = 10240 bits
L2 キャッシュの各ブロックに加えられる 4 つのフィールド	32768 blocks * 15 bits/block = 491520 bits
ポリューション・テーブル	4096 entries * 15 bits/entry = 61440 bits
ハードウェアの総コスト	565024 bits = 68.97 KB

(b)FDP 方式のハードウェア・コスト

L2 キャッシュの各ブロックの pref-bit	32768 blocks * 1 bits/block = 32768 bits
ポリューション・フィルター	4096entries*1bits/entry=4096bits
フィードバック情報の推定に使用される 16 ビットカウンタ	11 counters * 16 bits/counter = 176 bits
各 MSHR エントリの pref-bit	128 entries * 1 bit/entry = 128 bits
ハードウェアの総コスト	37168 bits = 4.54 KB



SPECCPU2006

SPECCPU2017

図 25 BPKI

で行われた．

参考文献

[1] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 364–373.

[2] S. Palacharla and R. E. Kessler, “Evaluating stream buffers as a secondary cache replacement,” in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994, pp. 24–33.

[3] I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, “Memory-system design considerations for dynamically-scheduled processors,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 133–143.

[4] T. Sherwood, S. Sair, and B. Calder, “Predictor-directed stream buffers,” in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000, pp. 42–53.

[5] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “Feedback directed prefetching: Improving the performance and bandwidth-effciency of hardware prefetchers,” in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, February 2007, pp. 63–74.

[6] T. F. Chen and J. L. Baer, “Reducing memory latancy via non-blocking and prefetching caches,” in *Proceedings of the 5th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 51–61.

[7] D. Joseph and D. Grunwald, “Prefetching using markov predictors,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 252–263.

[8] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, “Efficiently prefetching complex address patterns,” in *Proceedings of the 48th Annual International Symposium on Microarchitecture*, December 2015, pp. 141–152.

[9] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “IMP: Indirect memory prefetcher,” in *Proceedings of the 48th International Symposium on Microarchitecture*, December 2015, pp. 178–190.

[10] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, “Dynamic speculative precomputation,” in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001, pp. 306–317.

[11] I. Atta, X. Tong, V. Srinivasan, I. Baldini, and A. Moshovos, “Self-contained, accurate precomputation prefetching,” in *Proceedings of the 48th International Symposium on Microarchitecture*, December 2015, pp. 153–165.

[12] K. J. Nesbit and J. E. Smith, “Data cache prefetching using a global history buffer,” in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, February 2004, pp. 96–105.

[13] G. Hinton, D. Sager, M. Upton, and D. Boggs, “The microarchitecture of the pentium® 4 processor,” *Intel Technology Journal*, 2001.

[14] Intel, “Intel 64 and ia-32 architectures optimization reference manual,” 2009.

[15] *http://chipdesignmag.com/display.php?articleId=4032*.

[16] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O’Connell, “Making data prefetch smarter: Adaptive prefetching on POWER7,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, September 2012, pp. 137–146.

[17] H. W. Cain and P. Nagpurkar, “Runahead execution vs. conventional data prefetching in the ibm power6 microprocessor,” in *Proceedings of 2010 IEEE International Symposium on Performance Analysis of Systems & Software*, March 2010, pp. 203–212.

[18] B. Sinharoy, J. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. E. Moreira *et al.*, "IBM power8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2:1–2:21, 2015.

[19] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[20] J. M. Tendler, J. S. Dodson, J. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[21] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, 1981, pp. 81–87.

[22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[23] J. K. Peir, S. C. Lai, S. L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *Proceedings of the 25th Anniversary Volum ACM International Conference on Supercomputing*. ACM, 2014, pp. 347–356.

[24] *http://www.simplescalar.com/*.