Vol.2022-HPC-184 No.4
2022/5/11

HPCアプリケーションにおける適応型ハードウェアプリフェッチャの評価

小田嶋 哲哉[1,a)]　渡辺 慎吾[1]　高木 紀子[1]　伊藤 真紀子[1]　吉川 隆英[1]　本藤 幹雄[1]

概要：キャッシュミスを削減し，プログラムの性能を向上させる方法としてデータ・プリフェッチがある．プログラムのアクセスパターンを推測し，自動的にプリフェッチを発行するハードウェアプリフェッチにはいくつかの種類がある．我々は，その中でも Best-Offset Prefetcher（BOP）に注目している．BOP は，SPEC CPU ベンチマークにおいて，固定長の距離でプリフェッチを発行する機構に対して，高い性能を発揮している．しかし，論文中では HPC アプリケーションにおける性能が示されていない．そこで本稿では，SPEC CPU ベンチマークだけなく，HPC アプリケーションにおいても BOP が有効に機能するかについて，キャッシュミスのアドレストレースを用いた評価を行った．その結果，1 次プリフェッチ機構としてストライドプリフェッチ，2 次プリフェッチ機構として BOP を搭載したと仮定する環境では，スドライドプリフェッチのカバレッジが高く，そこから漏れ出たアドレスには規則性がないため，BOP では有意な効果が得られないことが推測された．

1. はじめに

近年，プロセッサの演算性能は向上しているが，DRAM などのメインメモリのアクセスレイテンシはほとんど改善していない．そのアクセスレイテンシを隠蔽するためにはデータキャッシュが必要不可欠である．しかし，必要なタイミングにデータがキャッシュに存在していなければ，メモリへのアクセスが発生し，長いアクセスレイテンシによってアプリケーションの性能が低下してしまう問題がある．そのため，データ・プリフェッチを用いることで，プログラムのデータアクセスパターンを推測し，今後必要とされるデータがキャッシュに存在している確率を上げることで，アプリケーションの性能を向上させることができる．

プリフェッチには，アクセスパターンに応じて自動的にデータをメモリからキャッシュに挿入するハードウェアプリフェッチと，コンパイラやプログラマがコード中にプリフェッチ命令を明示的に追加するソフトウェアプリフェッチがある．本稿では，その中でもハードウェアプリフェッチ，特に，Best-Offset Prefetcher [1], [2]（以降，「BOP」と略す）に注目している．BOP は，過去のプリフェッチ距離の履歴を用いて，時間とともに変化するアプリケーションの動作に自動的かつ動的に追随するために，最適なプリフェッチ距離を学習する．文献 [1], [2] では，主に SPEC CPU ベンチマークの性能についてシミュレータを用いたペーパーマシン上で BOP の評価を行っているが，固定長の距離でプリフェッチを発行する機構に対して BOP は高い性能を発揮している．さらに，BOP を実装するために必要なハードウェアコストが小さいこともメリットの１つである．しかし，文献には HPC アプリケーションにおける性能に関する言及がない．そこで，本稿ではシミュレータによる詳細な評価を行う前段階として，キャッシュに 1 次プリフェッチ機構としてストライドプリフェッチ，2 次プリフェッチ機構として BOP を搭載したと仮定する環境において，アプリケーションのキャッシュミスアドレスのトレースを用いて，SPEC CPU ベンチマークだけでなく HPC アプリケーションに対しても BOP が有効に機能するかについて評価を行う．

2. データ・プリフェッチ

プリフェッチは，今後利用が予測されるデータを事前にメモリからキャッシュに読み込んでおく動作である．これによって，データが必要となる度にメモリへアクセスをするのではなく，より高速なキャッシュへのアクセスを増やすことでアプリケーションの性能向上が期待される．プリフェッチには，ハードウェアが自動的に今後必要とされるラインをメモリからキャッシュに挿入するハードウェアプリフェッチと，コンパイラやプログラマがコード上に明示的にプリフェッチを行う命令を追加するソフトウェアプ

[1] 富士通株式会社
[a)] odajima.tetsuya@fujitsu.com

リフェッチがある．特に，メモリアクセスのレイテンシが大きいハードウェアでは，そのアクセスレイテンシを隠蔽することがアプリケーションの性能向上には重要であり，ハードウェアプリフェッチの性能はプロセッサの使い勝手やチューニングの容易さに大きく影響する．本稿では，ハードウェアプリフェッチの技術についてのみ言及する．

2.1 ハードウェアプリフェッチ

ハードウェアプリフェッチには，プリフェッチするアドレスを予測する手法がいくつか存在する．ここでは，代表的なプリフェッチ手法を紹介する．

ネクストラインプリフェッチ

ネクストラインプリフェッチは，データアクセスが発生したラインが X であった場合，その次のライン $X+1$ をキャッシュに挿入する．この方式では，プリフェッチするラインを容易に決定することができるため，プリフェッチャを実装するコストが小さい．一方，プリフェッチを実行する距離が短いため，アクセスパターンが連続している状況ではデータが必要なタイミングまでにプリフェッチが完了しない可能性もある．その場合，プリフェッチが連続して失敗し，プログラムの性能低下の要因となる．

ストリームプリフェッチ

ストリームプリフェッチは，データアクセスが発生したラインに付随し，そのラインから連続するラインをキャッシュに挿入する．プリフェッチを開始するアドレスは，データアクセスが発生した次のラインだけでなく，いくつかのラインを飛ばした先から複数のラインをプリフェッチすることも可能である．このラインを飛ばす距離を調整することで，次のデータアクセスまでにキャッシュにデータが挿入されている確率を上げることができる．配列に対して，インクリメンタルにアクセスする場合には有効であるが，Array of Structure の要素へのアクセスや多次元配列の非連続方向のアクセスに対応することは困難である．

ストライドプリフェッチ

ストライドプリフェッチは，データアクセスが発生したラインに対して，直近にアクセスしたアドレスとのストライド（差）を計算し，その値を用いて飛び飛びのラインをキャッシュに挿入する．過去のアクセス履歴はテーブルなどに保存されており，現在のアクセスパターンと過去のアクセスパターンに乖離が発生した場合，プリフェッチを止めることも可能である．プログラムの実行箇所によって，データのアクセスパターンが変わることが多く，そのような場合に対して適宜ストライドを計算し，適切な距離に対するプリフェッチを発行することが可能である．

テンポラルプリフェッチ

これまで紹介してきたプリフェッチでは，ベースとなるアドレスに対して一定の距離を加算（または減算）してプリフェッチするアドレスを推測してきた．このような方式では，アクセスするアドレスの間隔が不規則であるが，同じアクセスパターンが何度も繰り返されるときに追随することは困難である．テンポラルプリフェッチ [3] は，過去のメモリアクセスパターンから相関関係を抽出することで繰り返される不規則なアクセスパターンに対してプリフェッチを発行することができる．

その他

これまで紹介してきたプリフェッチ方式以外にもプリフェッチ技術が開発されている．A64FX プロセッサでは，ストリームプリフェッチをベースとしたプリフェッチャを搭載しており，L1D キャッシュにおける 1 つのキャッシュアクセスに対して，各キャッシュ階層にそれぞれ適した距離のプリフェッチを発行する．詳しくは文献 [4] の 11 章を参照されたい．

2.2 HPC アプリケーションと SPEC CPU ベンチマーク

HPC アプリケーションと SPEC CPU ベンチマークでは，データアクセスパターンに大きな違いがある．HPC アプリケーションでは，メモリやキャッシュのバンド幅を最大限活用するためにデータアクセスが連続するようにプログラムが記述されていることが多い．このようなアクセスに対しては，アクセスがあったラインに対してメモリアクセスレイテンシを隠蔽できる距離を取ってプリフェッチを発行すれば，次のデータアクセス時にキャッシュにデータが挿入されている可能性が十分に高い．一方，SPEC CPU ベンチマークは様々な種類のプログラムが含まれる複合ベンチマークである．特に，SPECint ではコンパイラやインタープリタなど多くの部分でデータアクセスが連続しない可能性が高いプログラムが含まれている．このようなプログラムは非連続なデータ構造と複雑な実効パスを有するため，ストライドプリフェッチだけでは対応しきれない．

さらに，近年の HPC アプリケーションでは C++ライブラリや Python などのフレームワークで記述されているプログラムが増えてきている．このようなプログラムはデータ構造や実効パスが複雑化しており，従来の比較的単純なプリフェッチだけを採用してしまうと，性能が悪化する可能性がある．そのため，今後のプロセッサにはプログラムの特徴に応じて柔軟に対応し，かつ，高い精度を有するプリフェッチ機構が求められる．しかしながら，複雑なプリフェッチに対応するためには，必要なハードウェアコストが非常に大きくなってしまい，結果としてコア数が減少するなどのトレードオフとなることも考慮する必要がある．

3. Best-Offset Prefetcher

本稿では，適応型ハードウェアプリフェッチャである Best-Offset Prefetcher（BOP）の HPC アプリケーションにおける有効性を評価する．BOP については文献 [1], [2] に詳しいが，ここでは本稿を理解するための最小限の機能

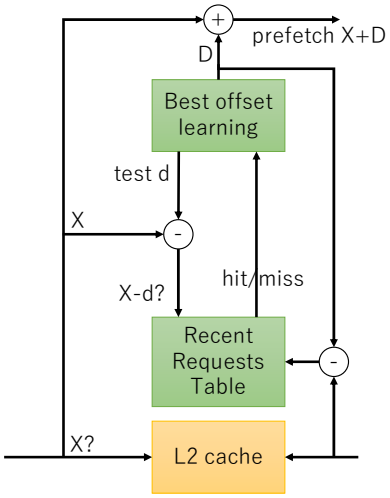

図 1 BOP の構成図．文献 [2] の Fig.1 から一部引用．

を述べる．

3.1 BOP の概要

BOP は，時間とともに変化するアプリケーションの動作に適用するように自動的かつ動的にプリフェッチ距離を設定する機構を有する．最良のプリフェッチ距離を推測するために，過去のプリフェッチを行った距離の履歴を用いて学習を行う．その後，学習が完了した時点で新たに最良であろうと思われる距離でプリフェッチを発行するようになる．BOP には，プリフェッチの適時性や精度を改善するための機構がいくつか導入されているが，本稿では主に BOP が学習可能なアクセスパターンと，そのためのオフセットリストに注目している．次節では，BOP におけるプリフェッチ距離の学習方法の概略について述べる．

3.2 プリフェッチ距離の学習

BOP の概略図を **図 1** に示す．図中の D は現在発行しているプリフェッチ距離を表し，ライン X のリード要求が L2 キャッシュでミスした場合，ライン $X+D$ のプリフェッチ要求が L3 キャッシュへ送られる．この D をアプリケーションの挙動に応じて動的に設定することで，最良のプリフェッチが期待される．

BOP における学習の基本概要は，ライン X がアクセスされたときに，直近にライン $X-d$ へのアクセスがあった場合，d は潜在的によいプリフェッチ距離であったという情報を繰り返し積み上げるということである．そして，ライン $X-d$ のアドレスをプリフェッチ要求のベースアドレスとして Recent Requests Table に記録する．そのテーブルにライン $X-d$ が存在している場合，ライン $X-d+D$ に対するプリフェッチ要求が最近に発行され，完了したということを意味する．したがって，D の代わりに距離 d でプリフェッチ要求が発行されていれば，ライン X に対するプリフェッチであったということが示される．

BOP では，図 1 の Best offset learning にオフセットリストとしていくつかの d がストックされており，ライン X への要求がある度に，先述した計算を複数の d に対して実行する．このとき，d がライン X のプリフェッチとしてヒットしていた場合は，その d を最良なプリフェッチ距離として重みを加算する．これを適当な回数行った後に，最も重みが大きい d を次のフェーズで D として設定し，プリフェッチを発行する．

4. キャッシュミスのアドレストレースによる BOP の有効性評価

本章では，SPEC CPU ベンチマークおよび，HPC アプリケーションにおける BOP の有効性を検証する．この検証のために，プロセッサシミュレータ鬼斬 [5], [6] を用いる．シミュレータでの実行結果を用いて，3 章の BOP の学習を模擬した計算を行い，その計算結果を解析してプリフェッチの有効性を評価する．文献 [1], [2] では，中間キャッシュである L2 キャッシュへ BOP を適用しているが，本稿では適用キャッシュ階層について言及しない．近年のプロセッサでは各キャッシュ階層にストリームプリフェッチやストライドプリフェッチを実装することが一般的であるため，それらと BOP を組み合わせることを前提としている．なお，本評価ではシミュレータの簡素化のために，よりプリフェッチのカバレッジが広いストライドプリフェッチを採用した．

4.1 評価方法

本評価では，鬼斬シミュレータを用いてキャッシュミスのアドレストレースを取得し，それを用いて BOP の学習を模擬する．

鬼斬は，東京大学 坂井・五島研究室で開発されている cycle-accurate プロセッサシミュレータである．命令セットアーキテクチャは RISC-V [7] をベースとしており，システムコールをエミュレーションするモデルを採用している．本評価では鬼斬の一部を修正し，L1D キャッシュにおけるデマンドアクセスミスが発生したアドレス値を出力させた．ストライドプリフェッチによるミスアクセスはシミュレータではプリフェッチミスとして分類され，デマンドミスには含まれない．これより，ストライドプリフェッチのカバレッジ外のキャッシュミスのみを取得している．

鬼斬によって得られたデマンドミスアドレス値を使用し，BOP の学習フェーズにおけるオフセットリストの重みを計算する．**図 2** を用いて，この計算方法の例を示す．鬼斬から得られたアドレス値に対して，オレンジ色の枠を設定する．本稿ではこの枠の範囲を「ウィンド」，ウィンドの最初のアドレス値を「baseAddr」と定義する．さらに，ウィンド内において baseAddr に対してラインの差を計算する対象を「targetAddr」とする．図 2 の一番上のウィンドを例にす

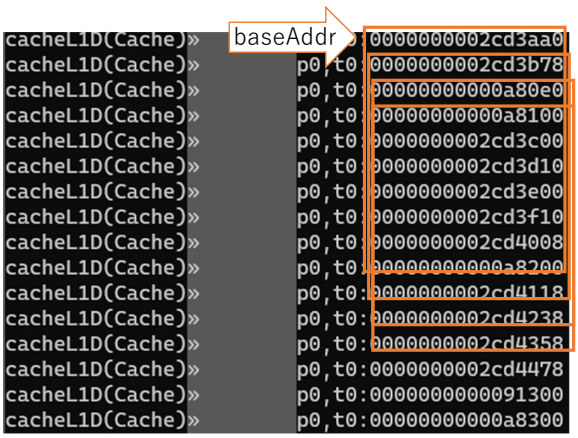

図 2　アクセス履歴による BOP の学習方法

表 1　鬼斬シミュレータのハードウェアパラメータ

項目		値
L1D$	容量	256KB
	Way	4
	プリフェッチャ	ストライド
L2$	容量	8MB
	Way	16
	プリフェッチャ	ストライド
データバスバンド幅		無制限
スレッド数		1

る．baseAddr が $0x2cd3aa0$ であり，1 つ目の targetAddr は $0x2cd3b78$ となる．これらのアドレス値はロード/ストア命令の実効アドレスであるため，それぞれの値をラインサイズ（本評価では 64 バイト）で割ることでラインサイズ単位に丸める．ここでは，baseAddr と targetAddr の差を「testLine」と定義すると次のような計算式となる：$testLine = targetAddr/64 - baseAddr/64$．実際に 1 つ目の例を計算してみると $testLine = 0x2cd3b78/0d64 - 0x2cd3aa0/0d64 = 0d3$ となり，オフセットリスト中のオフセット：+3 のスコアへ重み：1 を加算する．次に，targetAddr が 1 つ下に移動し，同様の計算を行ってオフセットリストのスコアを更新する．testLine の値が 0（つまり，同じラインでデマンドミスが発生した場合）またはオフセットリストに含まれない場合は，リストを更新せずに targetAddr が移動する．これをウィンド内のアドレス全てに対して計算を行う．本評価では，オフセットリストは 0 を除いた-1024~1024 として設定する．ウィンドサイズの設定は 256，つまり，1 つの baseAddr に対して 255 回計算を行う．その後，baseAddr が次のアドレス値に移動し，同様にウィンドも 1 つだけ移動する．この計算を鬼斬から得られたデマンドミスアドレスリストの全てに対して行う．

4.2　評価環境

評価に用いた鬼斬のパラメータは**表 1** に示すとおりである．キャッシュサイズが小さすぎる場合，頻繁にデータの入れ替わりが発生し，我々が意図していないデマンドミスが BOP 評価のノイズとして出現する可能性が高いため，本環境では L1D キャッシュサイズを大きく設定している．L1D キャッシュにはストライドプリフェッチャを搭載しており，前述の通り，ストライドプリフェッチャと BOP のハイブリッド構成を表現している．

コンパイラ環境として gcc version 8.1.0 を用い，RISC-V 向けのクロスコンパイラとしてビルドした．gcc8.1.0 では RISC-V の SIMD 拡張命令に対応していないため，本稿の評価では 64bit スカラで実行する．また，特定のシステムコールを用いることでプログラム中のシミュレーションを実行する範囲を指定することができる．測定したい部分まではエミュレーションモードで実行することで，全体のシミュレーション時間を削減することが可能である．一方，エミュレーションはキャッシュやレジスタを考慮した実行ではないため，シミュレーションが開始した時点ではデータの初期化に関しては不十分である可能性がある．しかし，それぞれのプログラムは実行時間が十分に長いためこの影響は最小限であると考えている．

4.3　評価ベンチマーク

本稿の評価には，SPEC CPU 2006 [8] の Integer プログラム（SPECint2006）と富岳重点課題アプリカーネル（fs2020-tapp-kernels） [9] を用いる．

SPECint2006 には C および C++で記述されたプログラムが 12 本用意されている．SPEC CPU ベンチマークは，実際のマシンを用いた実行に数時間から数十時間かかる規模であるため，すべてをシミュレーションで実行することは困難である．そのため，各プログラムの全実行命令数をエミュレーションモードで測定し，その値を用いて実行区間を均等に 20 分割する．各開始点までは鬼斬のエミュレーションモードを用いてスキップし，開始点から 20M 命令のシミュレーションを行う．これによって，プログラム実行の偏りによる影響を最小限にし，プログラム全体を平均的にサンプリングすることができる．

HPC ベンチマークとして，本評価では fs2020-tapp-kernels を用いた．これは，スーパーコンピュータ「富岳」におけるターゲットアプリケーションから 6 アプリの特徴的なカーネルをまとめたものである．カーネル自体は，A64FX に最適化されたコードとして提供されている．これらのカーネルは SPECint2006 とは異なり，シミュレータを用いた実行においても長くて 1 日で完了する規模のため，測定範囲は主カーネル部分全体とした．

4.4　SPECint2006 の評価

図 3 に，SPECint2006 のデマンドミス率を示す．縦軸のミス率は，L1D キャッシュにおけるデマンドミス数を全体

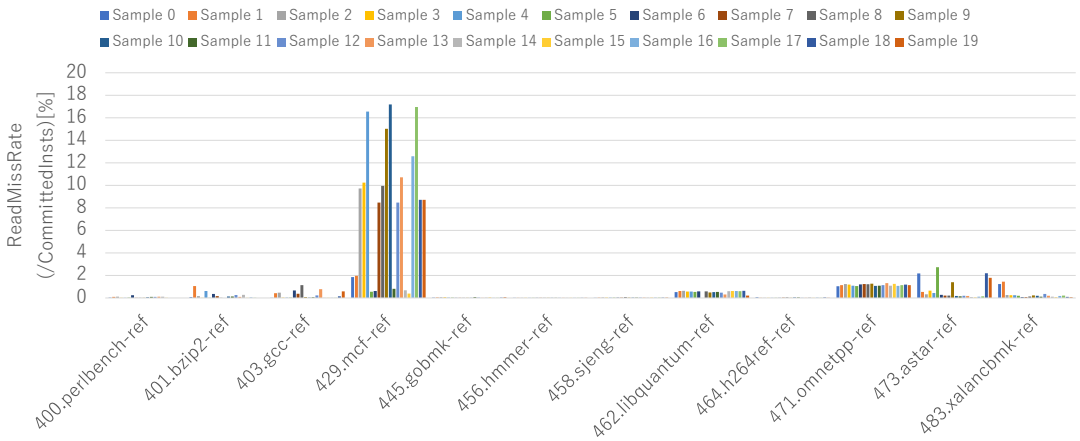

図 3 SPECint2006：Read Miss Rate (/CommittedInsts) [%]

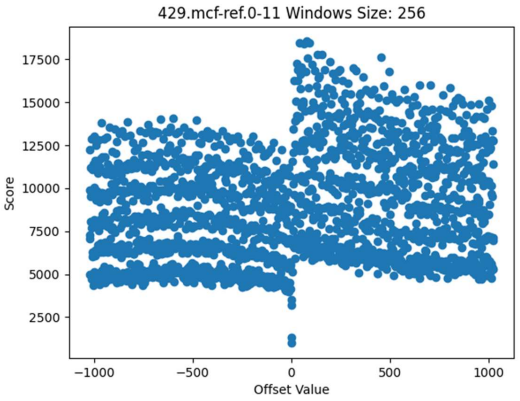

図 4 429.mcf Sample11 におけるオフセットリストスコア

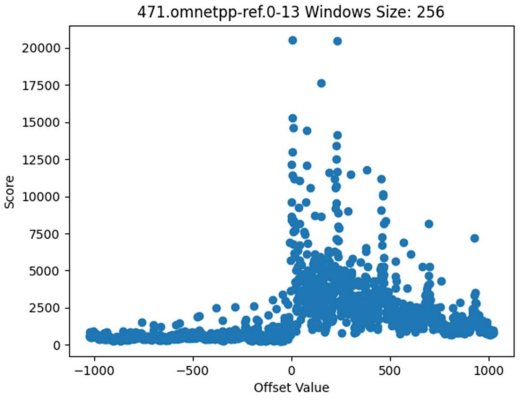

図 5 471.omnetpp Sample13 におけるオフセットリストスコア

のコミット命令数で割った値を割合として示している．各棒グラフは，実行したサンプル区間におけるミス率を示している．これより，ミス率は 429.mcf を除いて概ね 3%以内に収まっていることがわかる．つまり，比較的高機能なストライドプリフェッチャが搭載され，十分なキャッシュ容量を備える環境では全体のミス率がかなり低くなる．

このなかから特徴的なプログラムを 2 つ取り上げる．429.mcf に関してはミス率が 16%を超える区間もある一方で，1%未満の区間も存在しておりプログラム内におけるデータアクセスの挙動が大きく異なることがわかる．471.omnetpp に関しては，ミス率は 2%以内にとどまっているが，全測定区間で均一な値を示している．これらのプログラムについて，4.1 節で示したアクセス履歴による解析を行う．

表 2 SPECint2006 統計情報

429.mcf	Sample5	Sample11	Sample18
L1D$ NumPrefetch	1,332	425	394
NumReadAccesses	32,165,786	36,371,734	35,394,012
L1D$ ReadMissRate (/ComittedInst) [%]	16.55	17.18	16.97
471.omnetpp	Sample9	Sample13	Sample15
L1D$ NumPrefetch	9,150	8,835	8,651
NumReadAccesses	9,056,259	9,360,362	9,187,499
L1D$ ReadMissRate (/ComittedInst) [%]	1.26	1.32	1.26

表 2 に，2 つのプログラムにおけるミス率が高かった上位 3 つのサンプル区間の統計情報をそれぞれ示す．L1D$ NumPrefetch の値を見ると，429.mcf では L1D$のストライドプリフェッチャはプリフェッチの発行数がリードアクセス数に対して非常に小さいことから，データアクセスパターンはランダム性が非常に高いと言える．**図 4** に 429.mcf の Sample11 区間における BOP オフセットスコアのヒストグラムを示す．横軸はオフセットリストの値，縦軸は対応するオフセットのスコアを示す．これより，429.mcf ではオフセットリストに対してまんべんなくスコアが分布していることがわかる．BOP の特徴から，特定のオフセットのスコアが十分に大きく，その他のオフセットのスコアが十分に小さい時，最も高いプリフェッチ性能を達成できる．しかし，図 4 では最もスコアが大きいオフセットは 78 であるが，その近辺のオフセットのスコアも十分大きい．オフセット：78 におけるスコアを，オフセットリスト内の総スコアで割ることでリスト内の重みを計算することがで

表 3 fs2020-tapp-kernels 統計情報

	NICAM.vi_rhow_solver
L1D$ NumPrefetch	9,007,599
NumReadAccesses	46,827,544
L1D$ ReadMissRate (/ComittedInst) [%]	0.21
	QCD.jinv
L1D$ NumPrefetch	2,228,243
NumReadAccesses	127,805,519
L1D$ ReadMissRate (/ComittedInst) [%]	0.09

きるが，これはたかだか 0.1%に過ぎない．つまり，BOP が十分に機能するだけのアクセスパターンの特徴を持っていないことを示している．

同様に，表 2 より，471.omnetpp の L1D$ NumPrefetch は 429.mcf と比較すると値は大きく，それにともなってミス率も低くなっている．2 つのプログラムのシミュレーションを実行した命令数は同数であるため，プリフェッチ発行数がミス率に影響していることがわかる．しかし，リードアクセス数に対する NumPrefetch の割合は依然として小さい．**図 5** に 471.omnetpp の Sample13 区間における BOP オフセットスコアのヒストグラムを示す．これより，471.omnetpp は 429.mcf と比較してばらつき具合が小さい．図中の最も大きいスコアを持つオフセットは 3 であり，429.mcf と同様に総スコアに対する重みを計算すると 0.6%となる．429.mcf の重みよりは大きいが，同様に BOP が十分に機能するだけのアクセスパターンの特徴を持っていない．

4.5 fs2020-tapp-kernels の評価

図 6 に，fs2020-tapp-kernels のデマンドミス率を示す．縦軸のミス率は，L1D キャッシュにおけるデマンドミス数を全体のコミット命令数で割った値を割合として示している．各棒グラフは，実行した主カーネル部分全体のミス率を示している．これより，すべてのカーネルにおいてミス率が 0.25%以下であることがわかる．

表 3 に fs2020-tapp-kernels から，NICAM.vi_rhow_solver と QCD.jinv カーネルにおける統計情報を示す．これより，2 つのカーネルでともに NumPrefetch の値が大きいことがわかる．つまり，多くのアクセスストリームパターンでストライドプリフェッチャが有効に機能していると言える．NICAM だけでなく，他種のカーネルでも同様にプリフェッチ数が多い．この要因として，本評価に用いた HPC カーネルは Fujitsu A64FX プロセッサ向けに最適化されたコードであることが挙げられる．特に，A64FX ではキャッシュを有効に使用することが性能向上につながることが知られている．最適化は A64FX のキャッシュにフィットするようにデータアクセスを制限している．さらに，データアクセスパターンは SPECint2006 と比較してストライドプリフェッチャには予測しやすいことも影響し，ミス率が低下したと考えられる．

図 7 に NICAM.vi_rhow_solver における BOP オフセットスコアのヒストグラムを示す．これより，スコアの分布に規則性があるようにみえる．最も高いスコアを持つオフセットは 4 であり，そこから 4 の倍数であるオフセットが続いていく．オフセット 4 の総スコアに対する重みを計算すると 2.4%と SPECint2006 と比較して高い値を示している．しかしながら，ミス率は SPECint2006 の 1/10 ∼ 1/100 と極めて低い．

図 8 に QCD.jinv における BOP オフセットスコアのヒストグラムを示す．最も高いスコアを持つオフセットは 72 である．同時に，上位のスコアを持つオフセットを解析すると 8 の倍数であることがわかった．オフセット：72 の総スコアに対する重みを計算すると 0.9%である．

2 つのカーネルの評価より，BOP が十分に機能するだけのアクセスパターンの特徴を持っていないことが示された．

4.6 考察

SPECint2006 および fs2020-tapp-kernels を用いて BOP のオフセット推測の評価を行った．これらの評価より，共通する 2 点の結果が得られた．

- デマンドミス率の低さ
- デマンドミスアドレスのランダム性

文献 [1], [2] では，静的にプリフェッチ距離を設定するプリフェッチ機構と比較して最大で 30%の性能向上が得られている．しかし，ストライドプリフェッチのような比較的高機能なプリフェッチ機構がある環境では，規則性のあるデータアクセスパターンはそれらのプリフェッチ機構で検出されてしまう．そのため，本評価ではほとんどの測定においてデマンドミス率が非常に低かったと考えられる．ストライドプリフェッチによってフィルタリングされたデマンドミスのパターンはランダム性が高い．よって，ある評価空間で得られたプリフェッチ距離に対してプリフェッチを発行したとしても，そのプリフェッチがミス率の改善にほとんど寄与しない．

HPC アプリケーションのデータアクセスパターンは SPECint2006 よりも規則性が高いことが知られている．fs2020-tapp-kernels の結果より，すべてのカーネルでデマンドミス率が 0.25%以下であることから，ストライドプリフェッチャによるプリフェッチと比較的大容量のキャッシュによって大半のデマンドミスが削減されたと考えられる．1%以下のデマンドミスが発生するアプリケーションに対して BOP を使用したとしても，その中の数%ミスを改善することができるだけで，カーネル全体のミス率の向上にはほとんど影響しないことが想定される．

これまでの評価では，ある区間全体またはカーネル全体

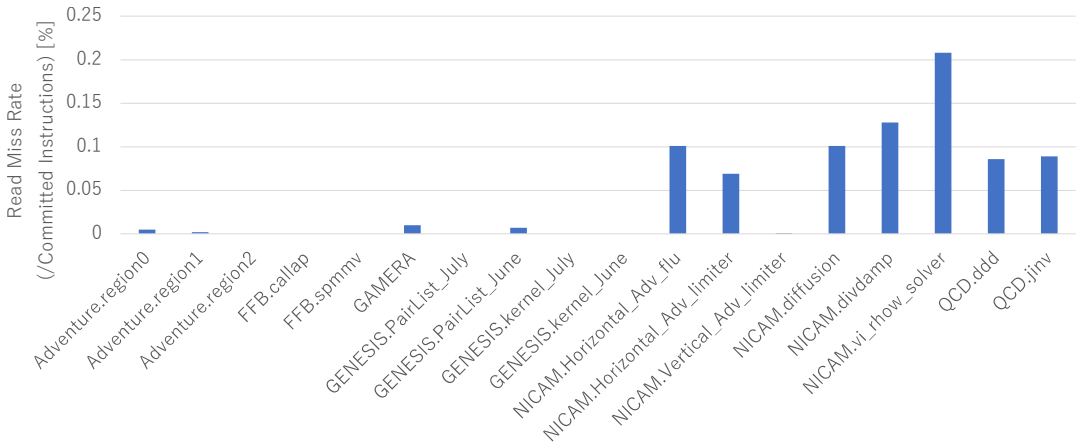

図 6 fs2020-tapp-kernels：Read Miss Rate (/CommittedInsts) [%]

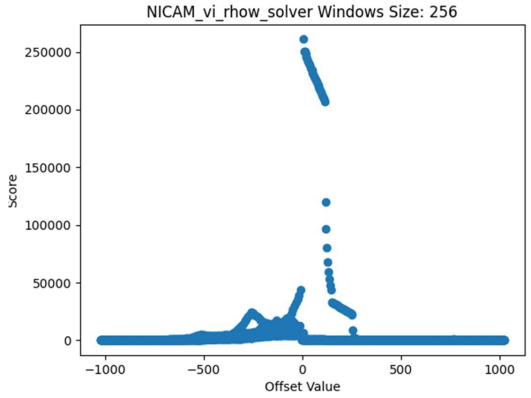

図 7 NICAM.vi_rhow_solver におけるオフセットリストスコア

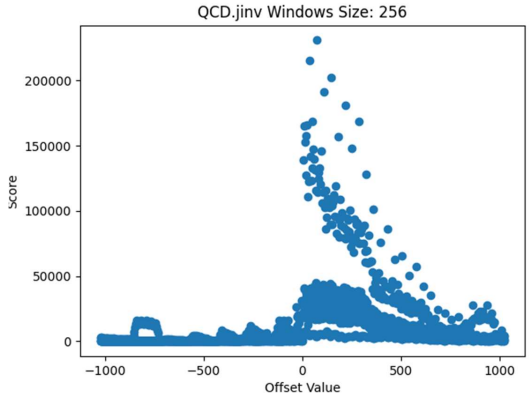

図 8 QCD.jinv におけるオフセットリストスコア

のオフセットスコアのヒストグラムによって BOP の有効性を検証してきた．しかし，より細かい区間ごとにオフセットを評価し，それを適用することで区間ごとのデータアクセスパターンに追随できることが BOP のメリットの 1 つである．この点についても，別途評価が必要であると考えている．**図 9** に 429.mcf の測定区間（20M 命令分）を 10K 命令に分割したオフセットスコアのヒストグラムを示す．時間軸は左上を先頭に，右方向へ，そして次の行へ続

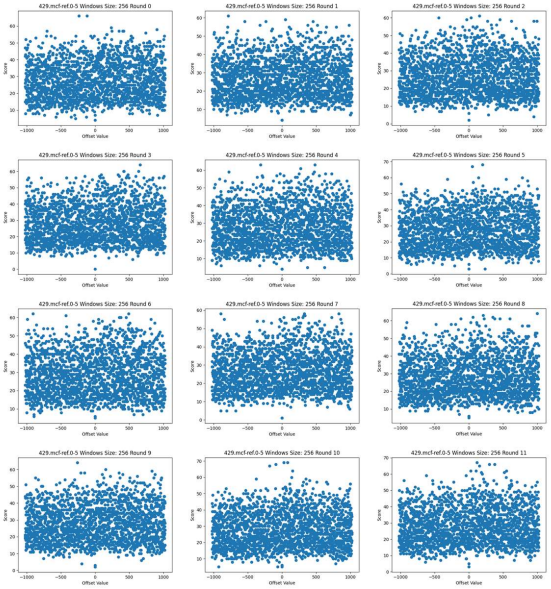

図 9 429.mcf の時系列順オフセットリストスコア

いている．これより，図 4 と比較しても評価区間を縮小してもアクセスパターンのランダム性が高いことには変わりがないことがわかる．確かに，区間ごとに最大のスコアを持つオフセットは異なっているが，そのオフセットにおける重みは非常に小さく，全区間の測定と同様に BOP が十分に機能するだけのアクセスパターンの特徴を持っていないことが示された．

これらの評価より，1 次プリフェッチ機構のストライドプリフェッチ，2 次プリフェッチ機構の BOP を組み合わせることを想定すると，BOP を搭載しても十分にプリフェッチが機能しないことがわかった．これは SPECint2006 だけでなく，fs2020-tapp-kernels のような HPC アプリケーションでも同様の結果であった．

5. おわりに

本稿では，キャッシュに 1 次プリフェッチ機構としてストライドプリフェッチ，2 次プリフェッチ機構として BOP

を組み合わせた環境における BOP の有効性について，SPECint2006 および fs2020-tapp-kernels を用いて評価を行った．評価には鬼斬シミュレータを用いて，L1D キャッシュにおけるデマンドミスが発生したアドレストレースを収集し，そのアドレス値をもとに BOP の学習パターンを模擬した計算を行い，オフセットリストとそのスコアによるヒストグラム解析を行った．その結果，SPECint2006 の 429.mcf 以外のプログラムでは L1D キャッシュデマンドミス率が 3.0%以下であり，特に fs2020-tapp-kernels による HPC アプリケーションカーネルの評価ではミス率が 0.25% 以下であった．SPECint2006 では，ストライドプリフェッチャによりフィルタリングされた L1D キャッシュデマンドミスのアドレスはランダム性が非常に高く，BOP によるプリフェッチ距離の追随性があったとしても，デマンドミス数の 0.6%程度しか改善できないという結果であった．fs2020-tapp-kerenels では，SPECint2006 よりもストライドプリフェッチによってフィルタリングされたアドレス値のランダム性は低かったが，そもそものデマンドミス率が低いため，SPECint2006 同様に BOP によるキャッシュミス率の改善によるアプリケーション全体の性能向上に与える影響は小さいことが想定される．これより，単純にストライドプリフェッチャを主として BOP を従とした組み合わせた環境では，SPECint2006 および HPC アプリケーションにおいて BOP を有効に活用することができないことがわかった．

今後は，データアクセスパターンが本質的に高いランダム性を持つアプリケーション，また，これまで評価してきたカーネルには存在しないデータアクセスパターンを持つアプリケーションについて，本稿で行ってきた評価を適用し，BOP に関する考察を深めたいと考えている．

参考文献

[1] Pierre Michaud. A Best-Offset Prefetcher. *The 2nd Data Prefetching Championship (DPC2)*, 2015.
[2] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, 2016.
[3] Wu, Hao and Nathella, Krishnendra and Pusdesris, Joseph and Sunwoo, Dam and Jain, Akanksha and Lin, Calvin. Temporal Prefetching Without the Off-Chip Metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, p. 996–1008, 2019.
[4] A64FX - Microarchitecture Manual. https://raw.githubusercontent.com/fujitsu/A64FX/master/doc/A64FX_Microarchitecture_Manual_jp_1.6.pdf.
[5] 鬼斬プロセッサシミュレータ Wiki. https://github.com/onikiri/onikiri2/wiki/JP-Home.
[6] 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 2009）, May 2009.
[7] RISC-V International. https://riscv.org/.
[8] SPEC CPU2006 Documentation. https://www.spec.org/cpu2006/Docs/.
[9] the kernel codes from Priority Issue Target Applications. https://github.com/RIKEN-RCCS/fs2020-tapp-kernels.