修士論文

高次ストライドを利用した
適応型キャッシュメモリ管理手法

平成31年1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

48-176412 渋江 陽人

概要

ループ構造等により，プログラムには特定の処理を頻繁に実行する期間が存在する．そこで，その処理にプロセッサを適応させ，性能を向上させる手法が研究されている．そういった手法では，どのように適応させるかだけでなく，どの時点で適応させるかも問題である．

動作を切り替えるタイミングの検出において，我々はキャッシュメモリの最適化に利用可能な新規検出手法を提案している．本論文ではその検出手法をプリフェッチャに適用し，プロセッサに適応動作させる手法を提案する．

提案手法は，既存の学習機構にフェーズ検出器とテーブルを追加し，時間的な変化に応じて過去のパラメータを復元するものである．フェーズが切り替わるごとに学習結果を保存し，将来同じフェーズが見られたときに学習結果を復元させる．

プリフェッチャに対して提案手法を用いた結果，学習機構無しのプリフェッチャに対し性能向上が見られ，学習および復元というアプローチの有効性が示された．一方で，学習においてノイズとなる，ストリーム以外のアクセスによる性能低下も見られた．また，既存のフェーズ検出機構を利用したプリフェッチャでは大きく性能が低下するベンチマークが存在したのに対し，高次ストライドを用いたプリフェッチャは安定した性能で動作することが示された．SHiP に適用した結果，いくつかのベンチマークで性能が向上した一方で，大きく性能が低下するベンチマークが見られた．過去のパラメータを復元することが有効であることが分かった一方で，フェーズ検出における問題も明らかになった．

目 次

図 目 次

表 目 次

第1章　はじめに

プロセッサの動作効率を向上させるためには，動作に適した構成やアルゴリズムが有用である．例えば，計算性能が必要であれば計算性能の高い構成を利用し，データアクセスが処理を律速するのであれば電力効率の良い構成を利用することで，計算性能を犠牲にせずにプロセッサ全体の電力効率を高められる．しかし，プロセッサの実行する処理は実行しているプログラムによって異なるため，予め最適な構成やアルゴリズムが何であるかは分からないのが普通である．そこで，複数の構成を利用できるようにしておき，処理をトラッキング・解析してその時々で構成を切り替える手法が提案されている [1].

そういった，処理に適応するプロセッサに加え，学習のアプローチを組み込んだ手法も提案されてきている [2, 3, 4]．キャッシュメモリにおける置換アルゴリズムでも，プログラムごとにアクセスするパターンが異なるため，最適なアルゴリズムは一般に異なる．そこで，これらの手法ではアクセスしたデータをキャッシュに格納するべきかどうかを，処理しながら学習することにより最適化を行う．

さらにプログラムを細かく見ると，同じプログラム中でも処理の傾向が時々刻々と変化していることが分かる．例えば，初期化処理を行う期間，大量のデータの操作を行う期間など，異なる傾向を持つ期間が現れる．そういった，特定の傾向を持つ期間のことをフェーズと呼ぶ [5]．プログラム単位よりも細かいフェーズ単位でも，最適な構成は異なる．よって，より細粒度で切り替えを行う手法も提案されている [6].

本論文では，フェーズを用いて学習を行う手法を提案する．フェーズごとに学習を行い，フェーズが切り替わった際，それが以前に出現したフェーズであれば過去の学習結果を復元させる．この手法は既存の学習手法と組み合わせて使用できる．

可変ストリームプリフェッチャに提案手法を適用したものと，SHiP に提案手法を適用したものを用いて性能評価を行った．プリフェッチャに適用した結果，性能が下がることは無く，利用しないプリフェッチャと比べて性能が向上したベンチマークが見られる一方，ほとんど性能差が無いものも多く見られた．SHiP に適用した結果，いくつかのベンチマークで性能が向上することが確認されたが，特定のベンチマークで大きく性能が低下することが分かった．

本論文では，フェーズの定義とその検出手法について第 2 章で述べる．第 3 章で

は，処理に適応して動作する機構の例と，本論文で利用する既存手法について紹介する．第4章で提案手法を述べ，ベンチマークプログラムに対して提案手法を用いた結果を第5章に示す．提案手法に関する考察と検討事項を第6章で述べたのち，第7章で結論とする．

第2章　プログラムとフェーズ

2.1　プログラムに含まれるループ構造とその期間

一般にプログラムはプロセッサが取り扱い可能な命令の組み合わせで構成される．ある程度以上の規模のプログラムでは，配列やリストによって保持されているデータ群に繰り返し処理を行うなど，ループ構造を持つことが多い．ループ部分を実行しているプロセッサを時間軸で観察すると，その期間中はループに含まれる命令が繰り返し実行されている様子が見られる．つまり，ループ構造に起因して，特定の命令が多く実行される期間が生じる．

このループ中の処理にプロセッサを適応させて，その性能を向上させることができる．例えば，計算性能の不要な処理を繰り返すループを考えると，そのループが終了するまで，コアの計算性能は必要でない．そこで，計算性能の高いコアと低いコアを同じプロセッサ上に搭載し，そのループを処理している間は計算性能の低いコアを使用することで，電力消費を抑え，対電力性能が向上する [1, 6].

ここで問題となるのは，そのループをどう検出するかである．実際にはループは一例であり，その時々で行われるプロセッサの処理の全体的な傾向を捉えることが広い意味での目的である．傾向の例として，計算処理が多く行われる，メモリアクセスが多く行われる，などが挙げられる．また，その傾向が持続する期間のことをフェーズと呼ぶ.

2.2　フェーズの定義

フェーズとは，プロセッサがある特定の傾向を示す期間のことである [5]．本論文ではメモリアクセスに注目し，特定のアクセスパターンが多く見られる期間をフェーズと定義する．

支配的なアクセスパターンが持続する期間をひとつのフェーズとする．別のアクセスパターンが支配的になったとき，別のフェーズに切り替わったと見做す.

2.3 フェーズの検出

2.3.1 インターバルごとのカウント

本節では，メモリに限らない，一般のフェーズ検出手法について説明する．プロセッサに専用のカウンタを搭載し，一定命令数を実行するごとに集計する手法が多く提案されている．カウントの対象は手法により様々で，各命令アドレスごとの実行回数 [7]，キャッシュメモリのヒット数やミス数 [6] など，用途に応じて設定される．

この一定命令数のことをインターバルと呼ぶ．インターバルの適切な長さは，利用方法により異なる．例えば，インターバルを長くすると大きなループ構造，短くすると小さなループ構造の検出に有効である．捉えたい構造の大きさのほか，利用方法のオーバーヘッドに応じて，インターバルの長さが設定される．

インターバルごとの各カウントを比較し，カウントが大きく変化する，あるいは閾値を跨いだことをもってフェーズの切り替わりを検出する．ただし，インターバルが終了する時点までそのインターバルの特徴は分からないため，適した処理を選択する等のためにはそのインターバルの傾向を以前のインターバルの傾向から推測する必要がある．一般には，同様の傾向が持続するものと仮定される．

このアプローチのメリットとして，インターバルの長さによって様々なスケールでの検出が可能であること，カウントの対象として任意の特徴を設定できることが挙げられる．デメリットとして，インターバルの終了時点で検出するためインターバルより短い期間での変化が検出できないこと，検出の閾値の設定が難しいこと，カウンタや比較器などの追加ハードウェアが必要になることが挙げられる．

2.3.2 プログラム解析

プロファイリングによってプログラムの構造を把握する手法もフェーズ検出の上で有用である．プロファイリングとは，一度プログラムを実行し，どの関数が何度実行されたか，その関数の実行にどのくらいの時間がかかったかなどの情報を収集することである．プロファイリングによってプログラム中での大きなループを検出し，その先頭にマーカーとなる独自の専用命令を追加することで，プロセッサがループの先頭を検出できるようにする手法が提案されている [8, 9].

このアプローチのメリットとして，フェーズの切り替わりをほとんど正確に検出できることが挙げられる．デメリットとして，事前に行うプロファイリングの手間がかかることが挙げられる．

2.3.3 ハイブリッド

インターバルによる動的な検出と解析による静的な検出の両方を組み合わせた手法がある．マルチレベル・フェーズ [10] では，プロファイリングによって巨大なループや最外にある関数呼び出しを検出する．ここで検出した箇所に命令を埋め込むことで，ハードウェアにフェーズの切り替わりを通知する．また，BBV によって小規模なフェーズを検出する．

小さなフェーズがどのように出現しているかパターンを学習する．学習によって，インターバルの終了を待たずに現在のインターバルのフェーズをそれまでのフェーズの系列から推測することができる．学習を解析によって検出した大域的なフェーズごとに行うため，高精度な推測が可能である．

2.4 インターバル同士の比較

2.4.1 BBV

本節では，前節で述べたインターバルを用いた手法について詳しく紹介する．Basic Block Vector（BBV）[7] という方法では，実行している命令の格納されているメモリアドレス（命令アドレス）に着目する．命令アドレスごとにカウンタを設け，その命令が実行されるごとに対応するカウントを増やせば，どの命令が頻繁に実行されているのかを検出可能である．しかし，命令アドレスごとにカウンタを設けるには大容量の記憶域が必要であり，現実的ではない．そこで，いくつかの命令アドレスをまとめて取り扱うことを考える．

命令は分岐を除き，メモリに格納されている順番に実行される．そこで，分岐や合流を含まない連続した複数個の命令をまとめて，ひとつのカウンタでカウントする．この分岐やジャンプを含まない連続した複数個の命令のことを，基本ブロック（Basic Block）と呼ぶ．なお，基本ブロックの出現回数ではなく，基本ブロックに含まれる命令が実行された回数をカウントする．つまり，基本ブロックが 3 命令から成る場合，一連の命令が実行されるたびにカウンタは 3 増加する．

以上から，基本ブロックの数だけのエントリ数を持つカウンタが得られる．これを基本ブロックごとのカウンタのベクトルと見て，基本ブロックベクトル（＝ BBV）と呼ぶ．インターバルごとにカウントを行うことで，それぞれのインターバルの特徴を表す BBV を得ることができる．

あるインターバルの BBV と別のインターバルの BBV を比較することで，インターバル同士の類似度が分かる．この比較には，一般にマンハッタン距離が用いられる．マンハッタン距離とは，ベクトルの各成分で差を取り，その絶対値を足し合わせた

ものである．適当な閾値を定め，マンハッタン距離が閾値以下であれば，どちらのインターバルも同じ処理を行っていることになる．逆に，マンハッタン距離が閾値よりも大きければ，ふたつのインターバルは違う処理を行っていることになる．BBV の閾値は，インターバルの長さの 4 %，すなわち，ふたつの BBV の合計カウント数の 8 %が良いとされている [11]．過去のインターバルの BBV を保存しておくことで，過去のインターバルとの比較も可能である．

この手法のメリットはプロセッサの様々な部分に利用できることである．これはこの手法が命令アドレスに基づいているためである．デメリットは，基本ブロックごとにまとめても依然として数が多く，大量のカウンタを要求することである．

2.4.2 高次ストライド

メモリアクセスに着目した手法に，高次ストライドを用いた方法 [12] がある．3 個以上のメモリアクセスがあったとき，それらのデータアドレスの間隔が一定であるならば，そのアドレス間隔をストライドと呼ぶ．また，以下ではこのような複数のメモリアクセスのことをまとめて，ひとつのストリームと呼ぶ．この手法では，ストリームの出現パターンを元にフェーズを捉える．

ストリームが複数（3 個以上）存在するときを考える．まず，それぞれのストリームの先頭のアクセスの，データアドレスに着目する．このアドレス同士の間隔が一定であるとき，これはストリーム同士のストライドと見ることができる．この間隔は 2 次ストライドと呼ばれる．

いくつのストリームが見られたか，2 次ストライドごとに分けてカウントすることでインターバルごとのベクトルを得る．インターバル同士のベクトルの比較では，BBV と同様にマンハッタン距離を用いる．2 インターバルのカウントの合計数の 6 %を閾値とすると，検出の性能が良いとされている [12]．

また，キャッシュメモリが階層化されている場合，その影響も受ける．下位のキャッシュメモリへのアクセスは，上位のキャッシュメモリでヒットしたときなどに隠蔽される．本論文では簡単のため，全てのアクセスが分かるものとして扱う．

この手法ではメモリアクセスの順番が重要なため，実アクセスの順番を利用するとアウトオブオーダ実行の影響を受けてしまうことがある．しかし，アウトオブオーダ実行をする場合でも，実行の完了（リタイア）はインオーダに行われる．本論文では簡単のため，リタイア順にメモリアクセスを取り扱うこととし，アウトオブオーダ実行の影響は受けないこととする．

第3章　関連研究

3.1　適応型プロセッサ

3.1.1　コアを切り替えるプロセッサ

本節では，処理に応じて動作を切り替えるプロセッサ機構の例を紹介する．

コンポジット・コア [6] では，処理速度の速いコアと遅いコア（実際にはμエンジン）を単一のアーキテクチャ上に搭載し，実行中の処理の傾向によってどちらのコアで実行するかを切り替え，電力消費を低減する．前者のコアをビッグコア，後者をリトルコアと呼ぶ．計算処理が多いときには当然ビッグコアで実行することが望ましい．一方，メモリアクセスが集中するような場合，メモリアクセスがボトルネックとなり計算性能は比較的不要であるから，リトルコアで実行しても実行速度が大きく落ちることなく電力消費を抑えることができる．また，分岐予測ミスが頻発するような期間では，パイプラインの段数の違いにより，フラッシュされる命令が少ないリトルコアの方が高速に実行できる．

複数の要因を元に切り替えを行うため，専用のコントローラも含めて提案されている．インターバルごとにL2キャッシュのヒット数や分岐予測ミス数などをカウントし，それらに適当な重み付けをして足し合わせることで，非アクティブなコアでの性能スコアを算出する．アクティブなコアのスコアと比較し，非アクティブなコアでの性能の方が良ければ切り替えを行う．このコントローラでは，十分に短い期間で切り替えを行えることから，インターバルは非常に短いものと設定されている．インターバルの特徴が次のインターバルでも持続するものと仮定して切り替えを行っていることに注意したい．

コンポジット・コアと同様に，コアを切り替える手法にbig.LITTLE[1] がある．big.LITTLEでは百万から億単位の命令実行ごとに切り替えを行う．コンポジット・コアではより細かい，千個の命令ごとに切り替えが可能である．

3.1.2 学習する置換アルゴリズム

SHiP

プログラムを実行しながら学習を行うことは効率の良い動作をする上で有効である．その例として，プログラムの特徴を学習する置換アルゴリズムが近年提案されてきていることが挙げられる．そのひとつが，SHiP[2] である．この手法はメモリアクセス命令ごとにシグネチャを発行し，シグネチャごとにアクセスしたデータが再利用されるかどうかを学習する．

シグネチャの生成方法は 3 種類提案されている．アクセスしたデータのアドレスの上位ビット，データアクセス命令の命令アドレスのハッシュ，直前の一定数の命令の履歴である．3 つ目の，命令の履歴について詳しく述べる．履歴を管理するキューを用意する．このキューのエントリは 1 ビットであり，エントリはシグネチャの長さと同じ数だけある．データアクセス命令，すなわちロード命令とストア命令が実行されたならば 1，それ以外の命令が実行されたならば 0 をキューに格納する．シグネチャを発行する際には，その時点のキューをそのままシグネチャとして出力する．3 つの生成方法のうち，この命令の履歴による生成が最も性能が良いことが分かっている．

キャッシュメモリはラインごとにシグネチャ用の容量が付加されている．データ挿入時にシグネチャを生成し，挿入したラインの追加容量部に記録する．また，学習器はシグネチャの種類と同じ数の飽和カウンタから成るテーブルであり，SHCT と呼ばれる．飽和カウンタとは，カウンタの上限に達した後，インクリメントされても変化せず上限のままになるカウンタである．学習は SHCT のエントリのインクリメントによって行われる．ラインへの参照があったとき，そのラインのシグネチャに対応するエントリをインクリメントする．データがキャッシュメモリから追い出される際，再参照されていなければ，対応するエントリをデクリメントする．この学習によって，再参照されるかどうか，SHCT のカウンタを見ることによって推測できるようになる．

SHiP は様々な置換アルゴリズムと組み合わせて使用することができる．本論文では SHiP の提案論文と同様に RRIP[13] という手法のアルゴリズムによって置換を行う．RRIP は，ラインごとに RRPV と呼ばれる 3 ビットのカウンタを付与する．置換を行う際，セット中から RRPV が 3 であるラインを探して追い出す．該当するラインが存在しない場合，全ラインの RRPV をインクリメントし，再度探索する．SHiP では，学習結果を元にしてデータ挿入時の RRPV の値を定める．SHCT の飽和カウンタが 0 であれば RRPV を 3，そうでなければ 2 として挿入する．また，再参照されたとき，そのラインの RRPV を 0 にする．

Hawkeye

もうひとつ，学習する置換アルゴリズムを紹介する．

キャッシュの置換アルゴリズムの研究において，未来のメモリアクセスが全て分かっているものとしてBeladyのアルゴリズム[14]に則った置換を行うアルゴリズムをOPTと呼ぶ．Hawkeye[3]は，OPTgenと呼ばれる，OPTを再現する機構を利用している．読み込んだデータがOPTの元で再参照されるかどうか判断し，キャッシュに格納するかバイパスするかを決定する．

3.2 プリフェッチャ

3.2.1 ストリームプリフェッチャ

本論文では，フェーズ検出をプリフェッチャに適用した手法を提案する．そこで，本節ではプリフェッチャの一種であるストリームプリフェッチャについて述べる．

将来必要になるデータをハードウェアが予測し，下位のキャッシュやメインメモリから対象となるキャッシュに予め読み込んでおく技術がプリフェッチである．プログラマが明示的に指示するソフトウェアプリフェッチに対し，ハードウェアプリフェッチとも呼ばれる．以下では，ハードウェアプリフェッチのことを指してプリフェッチと呼ぶ．

配列などの，連続領域に配置されたデータを順番に調べるような場合に効果的な手法として，ストリームプリフェッチ[15]がある．

ストリームプリフェッチャは次のように動作する．動作する様子を図3.1に示す．まずキャッシュミスが発生したとき，そのアドレスをテーブルに保管する．テーブルに保管されているアドレスの前後の一定範囲のアドレスを監視する．監視している範囲でキャッシュミスが発生したとき，保管していたアドレスから見て新たにミスしたアドレスの延長線上にある一定範囲のアドレスを監視するように範囲を切り替える．切り替えた後の範囲でキャッシュミスが発生したとき，監視範囲の延長線上にある一定範囲をキャッシュに読み込み，読み込んだ範囲のさらに延長線上にある一定範囲を監視するように範囲を切り替える．以降，監視範囲にミスがあるごとにキャッシュへの読み込み（プリフェッチ）と監視範囲の更新を繰り返す．パラメータとして，監視する範囲，プリフェッチする量がある．

具体例を挙げる．監視する範囲は32バイト，プリフェッチ量は64バイトとする．アドレス1000にてキャッシュミスが発生したとする．このとき，テーブルに1000が登録され，前後32バイト，すなわち968から1032が監視される．次にアドレス1016にてキャッシュミスが発生したとする．このとき，監視範囲が1017から1048

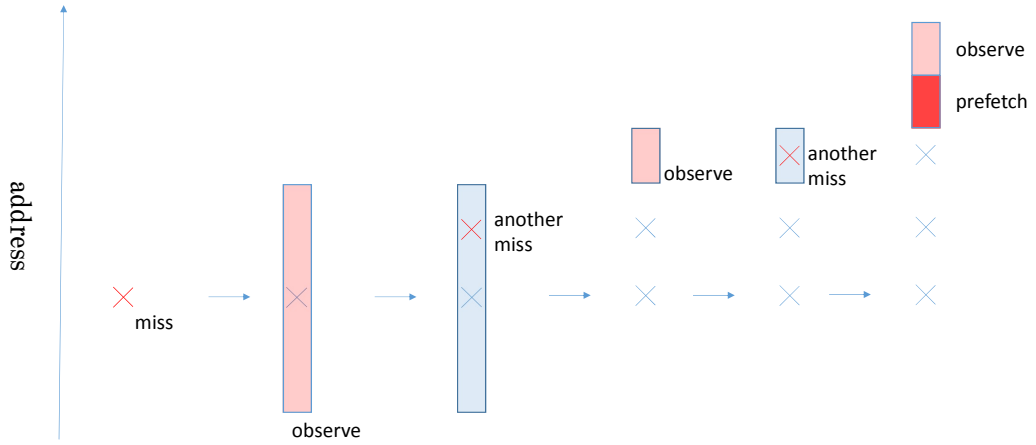

図 3.1: ストリームプリフェッチャの動作図

に切り替えられる．次にアドレス 1032 にてキャッシュミスが発生したとする．このときアドレス 1033 から 64 バイト，1096 までがプリフェッチされ，監視範囲が 1097 から 1128 に切り替えられる．

実際には，プリフェッチにかかる時間を考慮し，プリフェッチが間に合わないと考えられる，キャッシュミスの直後のアドレスをある程度スキップしてプリフェッチが行われる．このスキップする部分のことをディスタンスと呼ぶ．

以下では簡単のため，プリフェッチを行いテーブルの対応するエントリを更新（監視する範囲の更新やプリフェッチ量の更新）することを，単にストリームの更新と呼ぶこととする．

新しいストリームの追跡を開始する際，既にプリフェッチャのエントリが埋まっているとエントリを追い出す必要がある．追い出すとは，エントリのうち最も不要と思われるものを推測・選択して初期化し，再利用することである．この論文では，LRU を用いる．LRU とは，Least Recently Used の頭字語であり，利用されていない時間が最も長いものを選択して追い出すアルゴリズムである．

以下では単にストリームプリフェッチャと言ったとき，このストリームプリフェッチャのことを指す．

3.2.2 アダプティブなストリームプリフェッチャ

ストリームプリフェッチャの改良として，ストリームを更新するたびにプリフェッチ量を大きくする手法 [16, 17] が存在する．プリフェッチ量の増加の仕方によって大きく 2 種類に分けられ，1 ずつインクリメントする方法，指数的（倍々）に増加させる方法がある．以下ではこの種類のストリームプリフェッチャのことを可変ストリームプリフェッチャと呼ぶ．

指数的に増加させる方が高速に長いストリームに対応できる．一方で，短いストリームが多数現れるときや，キャッシュメモリの容量に厳しい制限があるときには無駄が大きいことが分かっている．

3.3 セット・デュエリング

本論文で提案する手法において，SDM（Set Dueling Monitor，セット・デュエリング・モニタ）[18] と呼ばれる機構を用いる．この機構はキャッシュメモリ上で利用するもので，複数の置換アルゴリズムを切り替えて使用することができる．

セット・デュエリングとは，複数の置換アルゴリズムをそれぞれ動作させ，性能を測定する手法である．セット・アソシアティブなキャッシュメモリには複数のセットがあるため，互いに異なる置換アルゴリズムで動作させることで性能を測定できる．複数あるセットのうちの一部でセット・デュエリングを行い，性能の良い方のアルゴリズムを残りのセットで使用する．ここで，セット・デュエリングを行うセット群のことを，性能をモニタリングすることから，セット・デュエリング・モニタと呼ぶ．

キャッシュメモリは非常に多くのセットからなるため，そのごく一部で性能を測定してもそれが汎用的に有効であるとは限らない．そこで，複数のブロックに分割し，そのそれぞれのブロック内でセット・デュエリングをそれぞれ行う．空間的局所性から，ブロックのサイズが十分小さければセット・デュエリングは有効である．図 3.2 は，1 ブロックにおいてふたつの SDM を使用する様子を示した図である．また，この手法自体を SDM を呼ぶこともある．

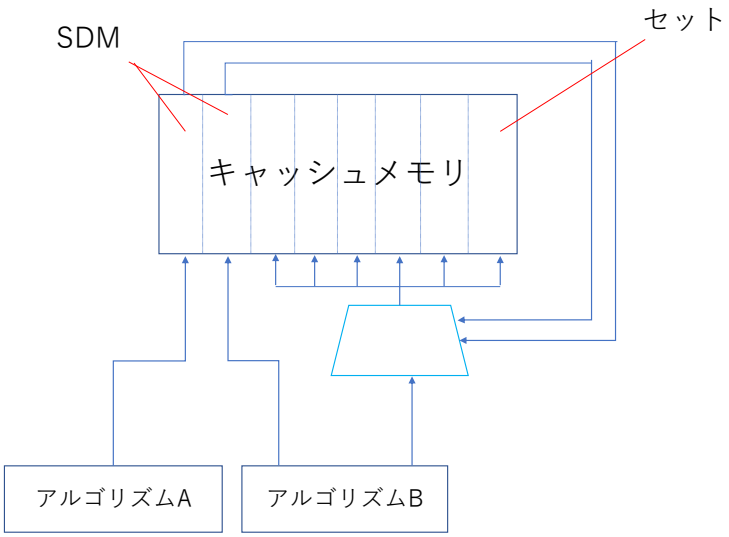

図 3.2: セット・デュエリングを行うときの入出力図

第4章　提案手法

4.1　概要

本論文では，フェーズを利用して過去のパラメータを復元する手法を提案する．既存の学習機構に適用することもできる．以下では適用する相手をターゲットと呼ぶ．

手法の概略図を図4.1に示す．フェーズ検出器と学習用のテーブルを用意する．ターゲットは何らかの可変なパラメータを持つものとする．ターゲットのパラメータを学習しつつテーブルに記録し，フェーズが再現したときにパラメータを復元する．

ターゲットとして，可変ストリームプリフェッチャを用いたもの，及びSHiP用いたものをそれぞれ評価実験で使用する．前者では，可変ストリームプリフェッチャにフェーズ検出器とテーブルを追加し，プリフェッチ量をフェーズごとに学習する．フェーズごとに学習させることで，それぞれのフェーズで最適なプリフェッチ量を探索し，効率の良いプリフェッチが可能となる．

プリフェッチャがエントリに追跡するストリームを登録するとき，次に行うプリフェッチ量の初期値が必要である．プリフェッチャのエントリの追い出しごとにプリフェッチ量をテーブルに記録し，記録した値を元に，適切と推測される値を初期値として設定する．

以下の説明では，あるインターバルの途中の時点を基準とし，その時間を現在と呼ぶこととする．

4.2　プリフェッチャへの適用

4.2.1　プリフェッチャの種類

本節では，対象として可変ストリームプリフェッチャを使用する場合について述べる．図4.2に概略を示す．

比較のため，表4.1に示すように2種類用意した．タイプ1の可変ストリームプリフェッチャは，プリフェッチを行いエントリを更新するごとに，次のプリフェッチ量を2倍にする．最小プリフェッチ量は8ライン，最大プリフェッチ量は256ラインとする．タイプ2の可変ストリームプリフェッチャは，プリフェッチを行いエントリ

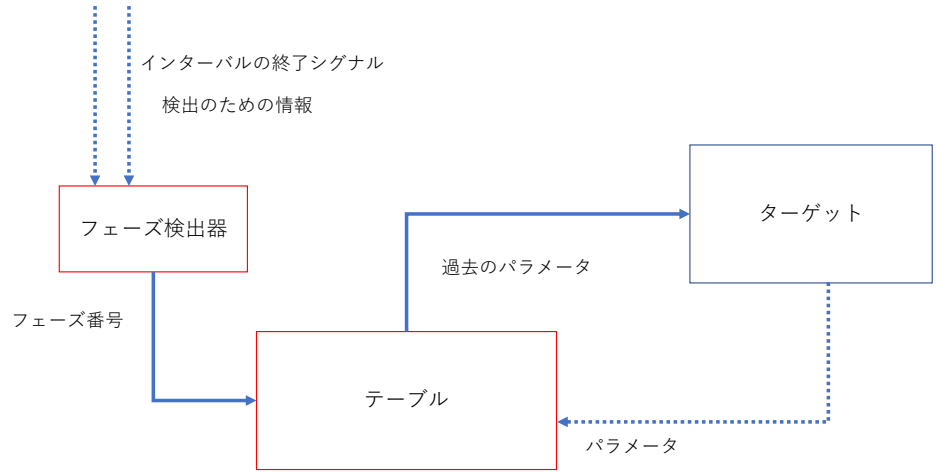

図 4.1: 提案手法概略図

表 4.1: 可変ストリームプリフェッチャの種類

	エントリ	増加の仕方
タイプ 1	8, 16, 32, 64, 128, 256	× 2
タイプ 2	8, 16, 24, 32, 40, 48, 56, 64	＋ 8

を更新するごとに，次のプリフェッチ量を 8 ライン増加させる．最小プリフェッチ量は 8 ライン，最大プリフェッチ量は 64 ラインとする．どちらのプリフェッチャも最大プリフェッチ量は十分であり，今回行った実験の範囲ではこの数値で性能に大きな悪影響を与えないことを確認してある．

フェーズ検出器として，高次ストライドによる検出を利用する．過去のインターバルのベクトルを保存し，類似したベクトル同士を同じ種類のフェーズとして扱う．新しい種類のフェーズが検出されたとき，順番に番号を付与する．

追加するテーブル及びプリフェッチャは現在のフェーズの番号を使用する．フェーズ検出器がそれらに対して現在のフェーズの番号を常に出力するのが理想である．しかし，インターバルが終了する時点まで本当の番号は分からないため，直前のインターバルと同じ特徴が持続していると仮定し，直前のインターバルのフェーズ番号で代用する．それぞれの入出力は図 4.3 のようになる．実線は常時送られるデータを，点線は特定のタイミングで送られるデータを表す．

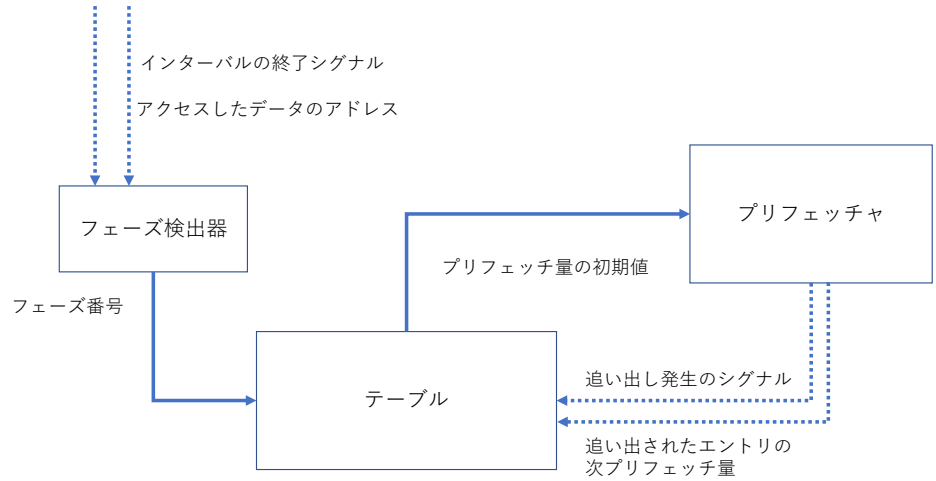

図 4.2: プリフェッチャに適用する場合の概略

4.2.2 カウンタによるプリフェッチ量の保存

フェーズ番号とプリフェッチ量をインデックスに取る，2 次元のテーブルを用いる．例としてタイプ 1 の可変ストリームプリフェッチャを利用する場合のテーブルの概略を図 4.4 に示す．

まず簡単のため，あるフェーズ番号に対応した 1 次元のテーブルについて説明する．

このテーブルはプリフェッチ量をインデックスに取る．また，プリフェッチ量の種類と同じ数のエントリを持つ．タイプ 1 の場合，6 エントリのテーブルとなる．プリフェッチャは追跡するストリームの数に上限があるため，一定数以上検出されると古いエントリを追い出す．追い出しは一般に LRU アルゴリズムによって行われる．プリフェッチャがエントリの追い出しを行うとき，追い出されるプリフェッチャのエントリが次にプリフェッチするはずだった量を，追い出しを行ったことと合わせてテーブルに通知する．テーブルは通知を受けると，そのプリフェッチ量に対応するエントリをインクリメントする．これによって，現れたストリームを長さごとにカウントすることができる．図 4.4 上の図は，何もカウントされていない状態から，次のプリフェッチ量が 32 だったプリフェッチャのエントリが追い出され，カウント処理を行った直後のテーブルの様子を示している．テーブルがカウントできる上限数は，十分大きな値であるとする．

この 1 次元のテーブルを，フェーズ番号ごとに用意する．なお，いくつのフェーズが検出されるかはプログラムを実行するまで分からない．そのため，この論文では十分な数のテーブルを用意できるものとする．プリフェッチャがエントリの追い

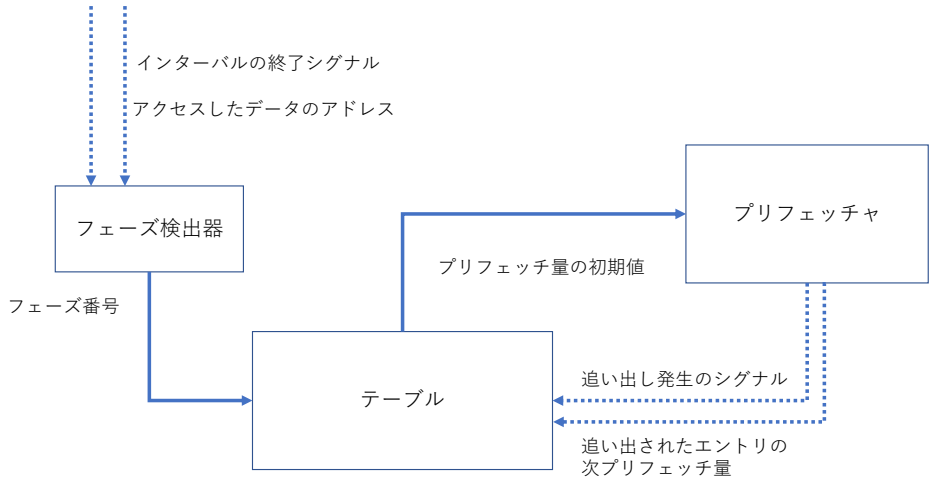

図 4.3: 提案手法における入出力

出しを行ったとき，1 次元のときの例と同様の通知を行う．テーブルはまずフェーズ検出器から受け取っているフェーズ番号を元に対応する 1 次元のテーブルを選択する．選択したテーブルに対し，1 次元のときの例と同様の処理を行う．図 4.4 下の図は，何もカウントされていない状態からカウント処理を行った直後の様子を示している．ただし，追い出されたプリフェッチャのエントリの次のプリフェッチ量は 64 であり，そのときのフェーズ番号は 2 だったとする．

実際のプロセッサでは十分な量を設けることは必ずしもできないため，カウントの上限およびテーブルのエントリ数については工夫を要する．利用可能な手法について 6 章で考察する．

4.2.3 フェーズ番号によるプリフェッチ量の復元

初期値設定のアルゴリズム

フェーズ番号に対応する 1 次元のテーブルに対し，プリフェッチ量の初期値を次のように定める．まず，1 次元のテーブルの全エントリの合計カウント数を求める．合計カウント数が一定数未満だった場合，初期値は最小のものとする．合計カウント数が一定数以上の場合，合計カウント数の一定割合を求める．ここで求めた値を超えるまで，大きなプリフェッチ量に対応するカウントから順に足し合わせる．最後に足し合わせたカウントに対応するプリフェッチ量より，ひとつ小さい値を初期値とする．

8	16	32	64	128	256
0	0	1	0	0	0

	8	16	32	64	128	256
1	0	0	0	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	0	0

図 4.4: フェーズ番号とプリフェッチ量を利用するテーブル

この論文中では,「合計カウント数の一定数」を 64,「一定割合」を 25 %とする．この値が適切かどうかについて，6 章で議論する．

タイプ 1 の可変ストリームプリフェッチャを利用した場合の具体例を示す．それぞれ対応するカウント数が 6，5，4，3，2，1 であったとき，合計カウント数は 21 であり，64 に満たない．よって，プリフェッチ量の初期値は 8 である．それぞれ対応するカウント数が 600，500，400，300，200，100 だったとき，合計カウント数は明らかに 64 を超えている．合計カウント数は 2100 であるから，後半の閾値は 525 となる．256 に対応するカウント数は 100 であり，これだけでは 525 に満たない．これに 128 に対応するカウント数を足し合わせると 300 であるが，まだ 525 に満たない．さらに 64 に対応するカウント数を足し合わせると 600 であり，525 を超えた．よって，64 よりひとつ小さい 32 を初期値として採用する．

アルゴリズムの背景

このアルゴリズムについて説明する．最も効率が良いと推測されるのは，対応するプリフェッチ量が大きいエントリから順にカウント数を足し合わせ，その合計が全カウントの一定割合を超えたとき，最後に足したエントリが対応する値よりひとつ小さい値である．

前提として，カウント数が不十分なとき，そのフェーズにおけるストリームの特徴をカウンタが捉えられていない．よって，この場合には基準となる最小の値を採用するべきである．以下では，カウント数が十分であるときについて説明する．こ

こでもプリフェッチ量が 8，16，32，64，128，256 ラインの 6 種類であったとして例示する．

まず，カウンタと実際のストリームの長さの関係について述べる．簡単のため，8 ラインのプリフェッチから始める状況を考える．32 に対応するカウントがあったとき，プリフェッチ量 8 と 16 のプリフェッチは実行され，32 のプリフェッチは実行されなかったということである．そのため，実際のストリームの長さは，8 より大きく 24 以下であると考えられる．理想的なプリフェッチはストリームの長さと等しいだけの量をプリフェッチすることであるから，多少の無駄を許して 32 という候補が考えられる．プリフェッチのディスタンスを考えても，32 というプリフェッチ量はおそらく十分大きな値である．また，16 というプリフェッチ量も，不十分である可能性を含むが候補として適切である．よって，最適なプリフェッチ量の候補として 16 と 32 のふたつがある．8 以外の初期値で始めたときも同様である．

次にテーブルと学習の関係について述べる．前述の例において，32 をプリフェッチ量の初期値として設定してしまうと学習に問題が生じる．今後のカウントの初期値が全て 32 になってしまうため，16 以下のカウントは今後現れなくなってしまう．つまり，プリフェッチ量を減らす方向の学習ができなくなってしまい，大きい値に収束してしまうのである．ひとつ小さい値である 16 を採用することによって，この問題を回避できる．以上が，テーブルに対応するプリフェッチ量よりも 1 段階少ない量を採用する理由である．

4.3 SHiP への適用

本節では，既存の学習機構に適用する場合について，SHiP を用いて説明する．概略を図 4.5 に示す．

フェーズの切り替わりごとに，その時点の SHCT をテーブルにフェーズ番号と共に記録する．過去の SHCT を利用した SHiP と，現在の SHCT を利用した SHiP を SDM にて利用する．ここで SDM を用いるのは，過去の SHCT で学習していない，もしくはしきれていないようなシグネチャが存在する可能性があるからである．そういったシグネチャに対して現在の SHCT を利用するため，完全な復元ではなく SDM を用いた一部復元としている．

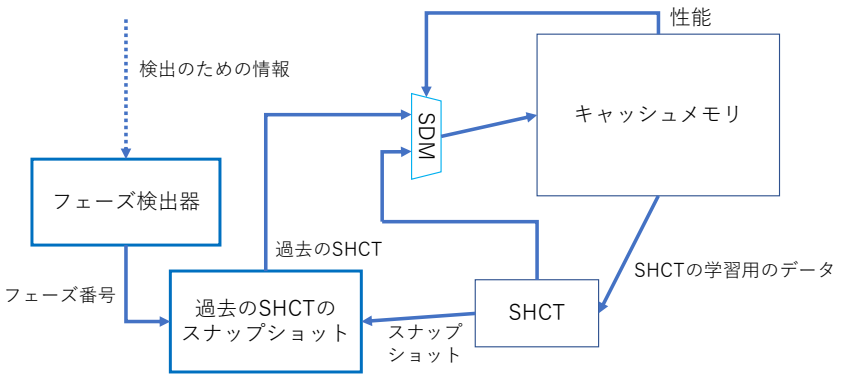

図 4.5: SHiP に適用する場合の概略

第5章　評価

5.1　使用ベンチマークと使用ツール

提案手法を用いた解析を行う対象として，SPEC CPU 2006[19] を採用した．なお，結果における GEOMEAN は SPEC CPU 2006 の全ベンチマークでの結果の幾何平均を取った値を表している．全てのベンチマークについて，先頭から 10G 命令をスキップし，その後の 1G 命令を実行した．シミュレータの鬼斬弐 [20] を用いた．ストリームプリフェッチャとして [21] の実装を用い，可変ストリームプリフェッチャはこれにプリフェッチ量を変化させる機構を加えた．ストリームプリフェッチャは L3 キャッシュメモリにのみ適用した．評価には IPC および L3 キャッシュメモリのヒット率を用いた．ただし，キャッシュメモリは L1 命令，L1 データ，L2，L3 の 4 つがあり，L3 が LLC である．

採用した共通パラメータは表 5.1 の通りである．

可変ストリームプリフェッチャのタイプは表 4.1 に従う．簡単のため，以下ではタイプ 1 の可変ストリームプリフェッチャに提案手法を適用したものをタイプ 1 の提案手法，タイプ 2 の可変ストリームプリフェッチャに提案手法を適用したものをタイプ 2 の提案手法と呼ぶ．

表 5.1: 共通パラメータ

L3 キャッシュ（LLC）	容量	2MB
	ラインの大きさ	64 バイト
	置換アルゴリズム	LRU
プリフェッチャ	追跡するストリームの本数	16
	ディスタンス	16 ライン
	初期プリフェッチ量	8 ライン
フェーズ検出器	インターバル	1M 命令

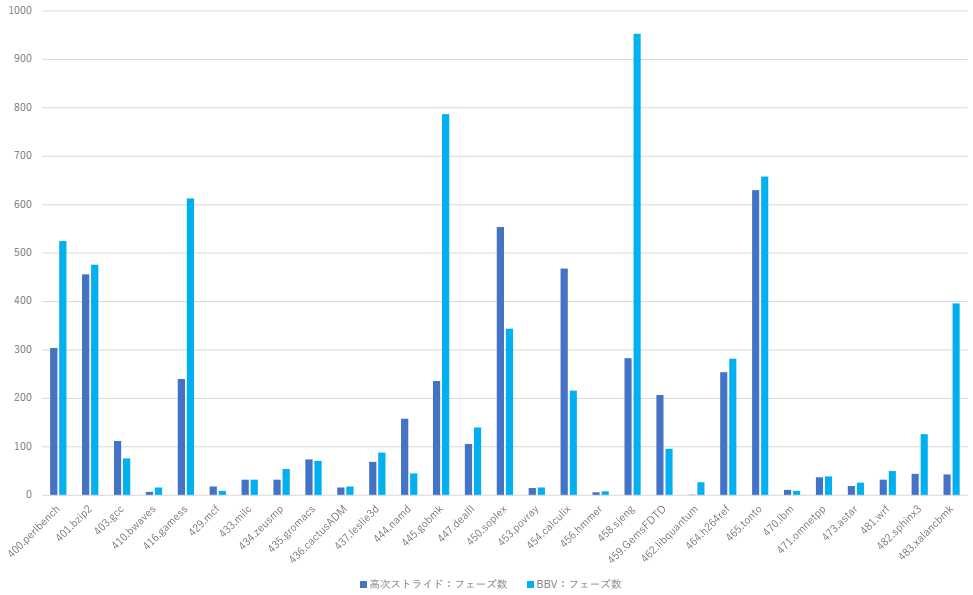

図 5.1: フェーズの出現種類数

5.2 フェーズの再現数

提案手法の評価に対する事前評価として，フェーズ検出について評価を行った．BBV および高次ストライドの両方について，評価を行った．フェーズの出現種類数を図 5.1，再現数を図 5.2，継続数を図 5.3 に示す．フェーズの出現種類数は，互いに異なる種類のフェーズがいくつ検出されたかを示す．再現数は，フェーズが切り替わった回数のうち，そのフェーズが過去に出現していた場合の数である．継続数は，インターバルを跨いだときにフェーズが変わらなかった回数である．

出現種類数から，確かにフェーズが検出されていることが分かる．殆ど検出されない，もしくは全てのインターバルで別フェーズと検出される，といった検出の問題が無いことが確認できた．再現数から，確かに同種のフェーズがある程度の時間経過後に再現していることが確認できた．継続数から，フェーズがある程度持続し，フェーズに合わせてパラメータを復元させても間に合う可能性が高いことが確認できた．

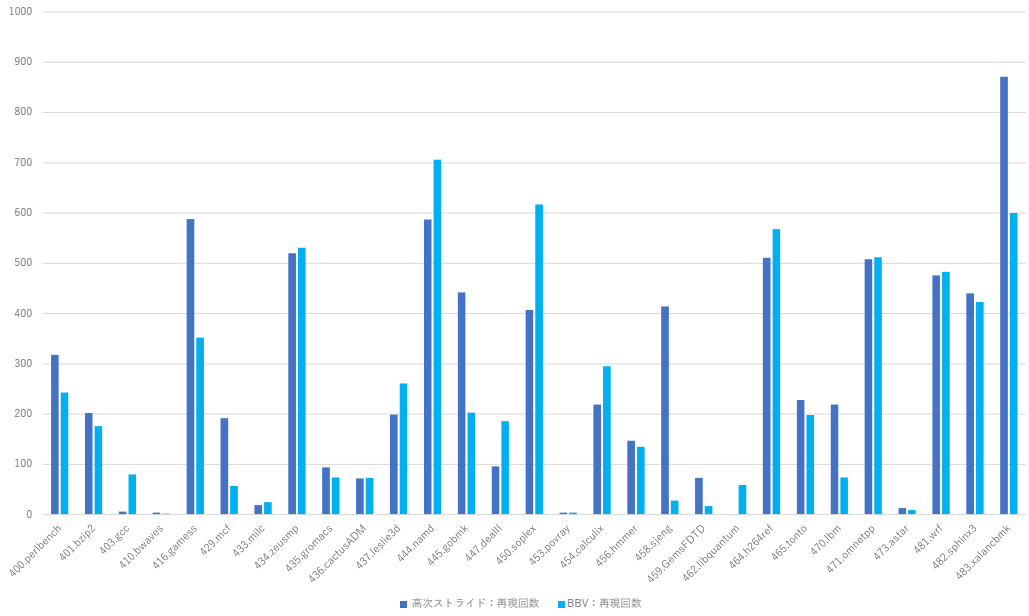

図 5.2: フェーズの再現数

5.3 可変ストリームプリフェッチャへの適用

5.3.1 可変ストリームプリフェッチャと提案手法の比較

タイプ 1

タイプ 1 の可変ストリームプリフェッチャを用いたときとタイプ 1 の提案手法を用いたときの性能を比較した結果を，図 5.4 および図 5.5 に示す．参考のため，プリフェッチ量が 8，64，256 のストリームプリフェッチャ3 種類と，プリフェッチャ無しの結果も示している．また，IPC については，可変ストリームプリフェッチャでの性能を 1 として正規化した結果を図 5.6 に示す．見やすさのため，プリフェッチャ無しの結果は省略し，0.9 から 1.1 の範囲を拡大している．

482.sphinx3 に代表される，プリフェッチ量が大きくなるにつれて L3 ヒット率が向上するようなベンチマークでは，提案手法が可変ストリームプリフェッチャを上回る L3 ヒット率を示す傾向があり，過去のパラメータの復元が効果的に作用していることが分かる．一方，433.milc や 437.leslie3d のようなプリフェッチ量 64 固定のストリームプリフェッチャを用いた際に他のプリフェッチャを用いた場合よりも高い L3 ヒット率を示すようなベンチマークでは，可変ストリームプリフェッチャと提案手法の間でほとんど性能に差が見られない．すなわち，適切なパラメータがあるにも関わらず学習できていないことが分かる．

450.soplex では，L3 ヒット率はプリフェッチ量 8 固定のストリームプリフェッチャと比べて提案手法および可変ストリームプリフェッチャの方が高い．しかし，IPC で

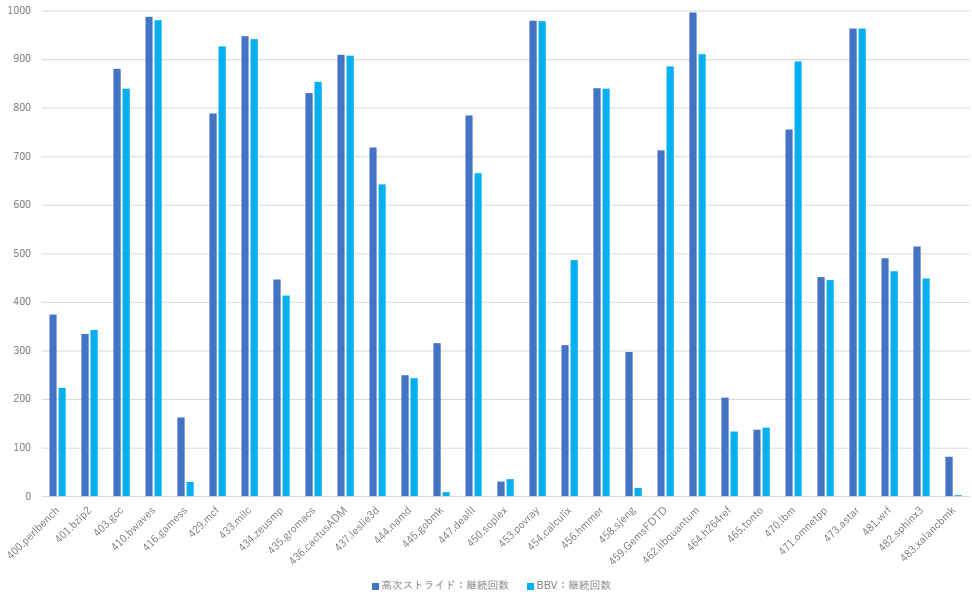

図 5.3: フェーズの継続数

は逆に低くなっている．詳しく調べたところ，プリフェッチ量が少ないときでもプリフェッチ中のデータへのアクセスが多数あることが分かった．プリフェッチの量を増やすことでヒット数自体は大きく増加しているもののミスも大きく増加しており，プリフェッチによる時間短縮の影響が小さいことと合わせて IPC の低下に繋がったと考えられる．

全体的な傾向として，提案手法は大きなプリフェッチ量で固定しているストリームプリフェッチャの性能に劣る．提案手法で行われる学習が最適なパラメータに収束しない理由について 6.3 節で考察する．

タイプ 2

タイプ 2 の可変ストリームプリフェッチャを用いたときとタイプ 2 の提案手法を用いたときの性能を比較した結果を，図 5.7 および図 5.8 に示す．参考のため，プリフェッチ量が 8，64 のストリームプリフェッチャ 2 種類と，プリフェッチャ無しの結果も示している．また，IPC については，可変ストリームプリフェッチャでの性能を 1 として正規化した結果を図 5.9 に示す．見やすさのため，プリフェッチャ無しの結果は省略してある．

プリフェッチ量が多いときに高い IPC や L3 ヒット率を示すベンチマークにおいて，提案手法がプリフェッチ量 64 固定のストリームプリフェッチャに対して劣る結果となっている．学習速度だけでなく，アルゴリズム自体の効率の問題が考えられる．前項と合わせて 6.3 節で考察する．

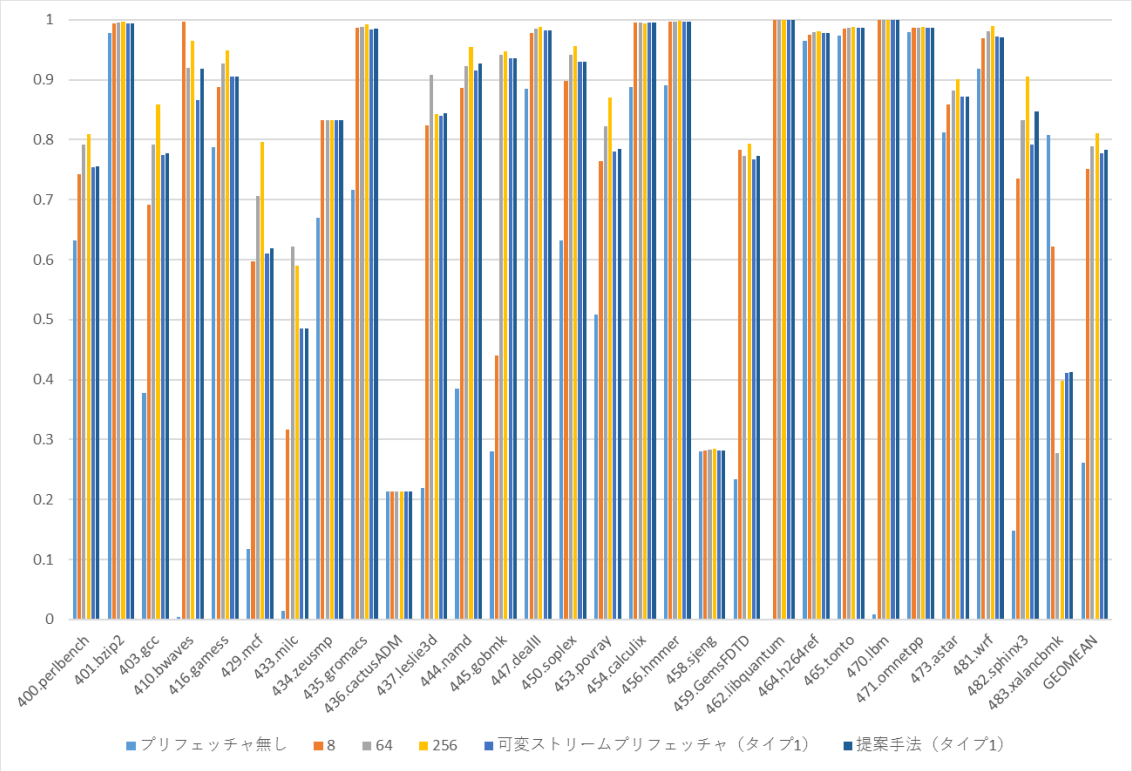

図 5.4: 可変ストリームプリフェッチャとの比較（タイプ 1-L3 ヒット率）

5.3.2　フェーズ検出手法の比較

タイプ 1

提案手法による性能と，フェーズ検出を BBV で行ったときの性能と比較した結果を図 5.10 および図 5.11 に示す．参考として，可変ストリームプリフェッチャの結果を併せて示している．

ほぼ全てのベンチマークにおいて，フェーズ検出手法に依らず可変ストリームプリフェッチャの性能を上回っていることが分かる．

しかし，483.xalancbmk において BBV は大きく性能が低下している．この影響で幾何平均における性能評価も低下している．ストリームの解析をする場合，BBV では正しいフェーズが捉えられない場合があることが示された．

タイプ 2

タイプ 1 と同様に比較した結果を図 5.12 および図 5.13 に示す．参考として，可変ストリームプリフェッチャの結果を併せて示している．

フェーズ検出手法に依らない性能を示すベンチマークが大半である．2 種類のフェーズ検出手法の両方で可変ストリームプリフェッチャの性能を下回った，433.milc およ

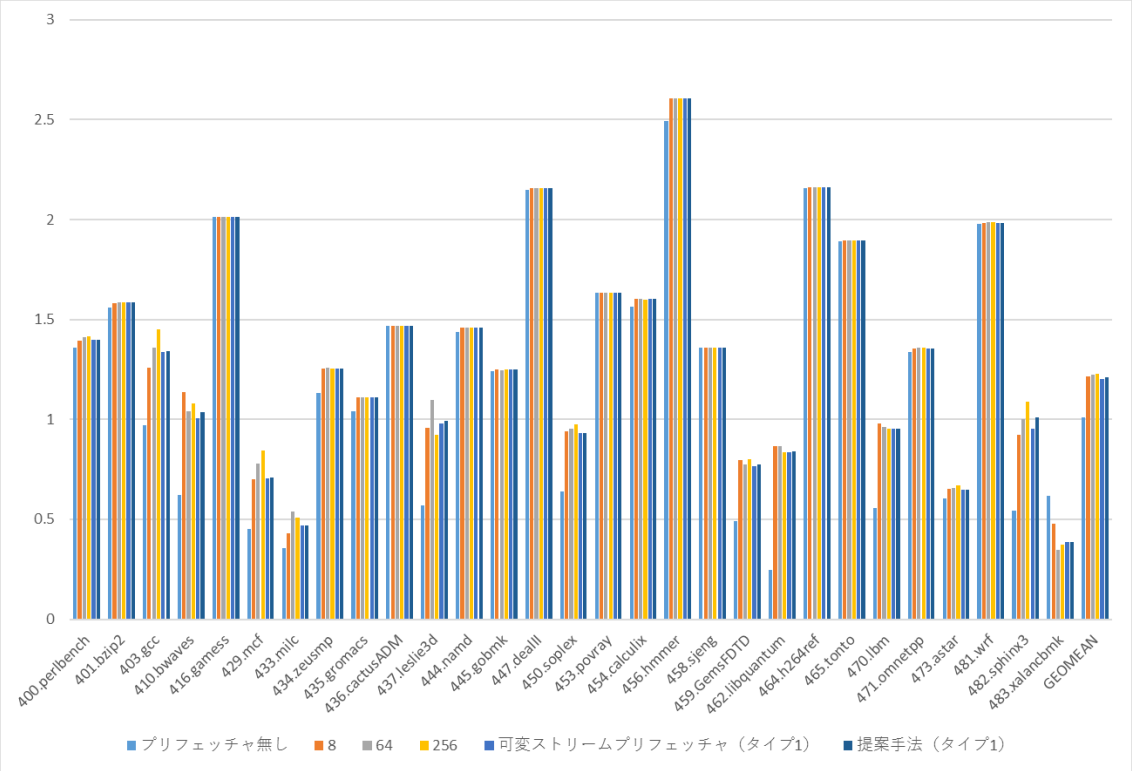

図 5.5: 可変ストリームプリフェッチャとの比較（タイプ 1-IPC）

び 437.leslie3d に注目する．5.3.1 節の実験結果を合わせると，最適なプリフェッチ量は 64 の前後にあるものと推測できる．タイプ 2 の最大プリフェッチ量は 64 であるから，正しく学習できれば性能は上がるはずである．しかしむしろ低下していることから，アルゴリズムの欠陥の存在が想定される．

5.3.3　インターバルの長さ

インターバルの長さを変えたときの性能の違いを図 5.14 および図 5.15 に示す．図の 1M や 100K はインターバルの命令数を表している．

ほとんどのベンチマークにおいて，性能が変わらない，もしくはインターバル長 1M 命令の方が性能が高いという結果が得られた．しかし，429.mcf や 435.gromacs ではインターバル長 100K の方が性能がわずかに高い．1M よりも短く，100K でのみ検出できるような長さのフェーズの存在が推測される．

5.4　SHiP への適用

SHiP に対して適用した結果を示す．ただし，SHiP の詳細な実装については，プリフェッチを考慮した SHiP である SHiP++[22] を用いた．SHiP，提案手法を SHiP に

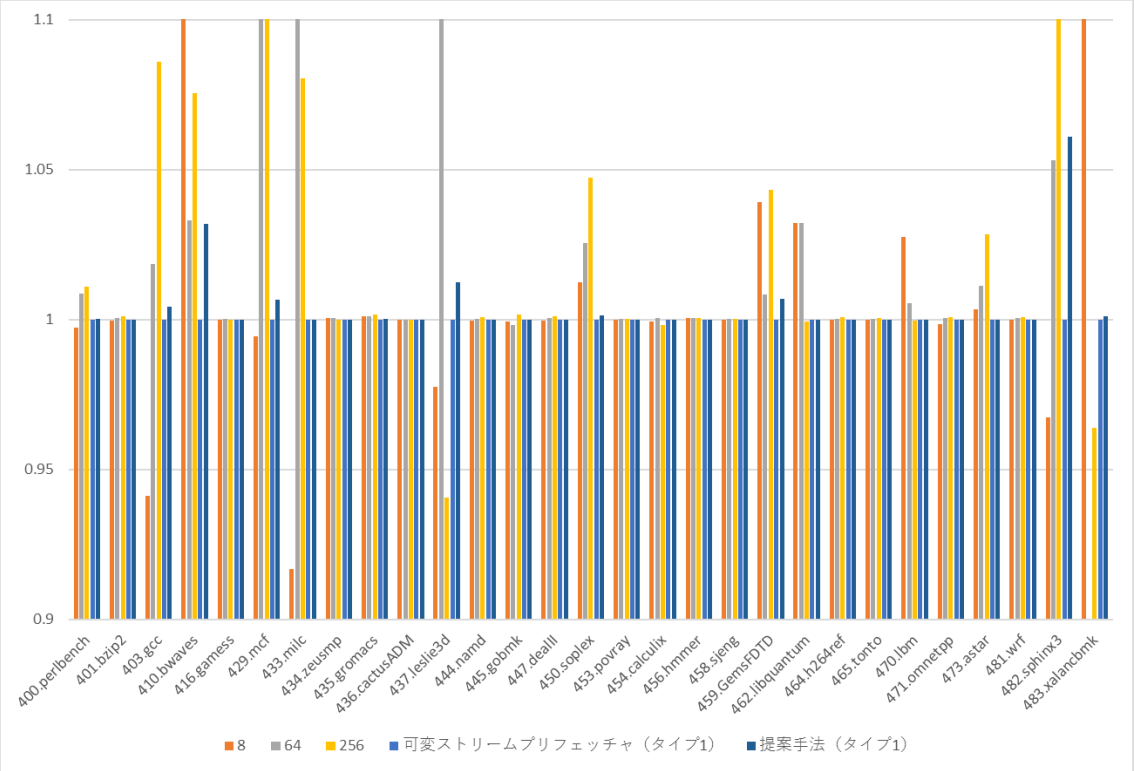

図 5.6: 可変ストリームプリフェッチャとの比較（タイプ 1-IPC，正規化）

適用したもの（SHiP ＋高次ストライド），提案手法のフェーズ検出を BBV に変えて SHiP に適用したもの（SHiP ＋ BBV）を比較した．

図 5.16 は L3 キャッシュのヒット率を示している．ほとんどのベンチマークで大きな差は見られないが，433.milc で SHiP ＋高次ストライドは大きく性能が低下している．その他，436.cactusADM や 444.namd，473.astar では性能向上が見られる．

図 5.17 は IPC を示している．差を分かりやすくするため，SHiP の結果で正規化したものを図 5.18 に示す．なお，433.milc における SHiP ＋高次ストライドの正規化した結果は 0.960 だが，分かりやすさのため省略している．L3 ヒット率と同様の傾向を示している．

5.5　評価のまとめ

事前評価の結果，ある程度のフェーズの再現と継続が見られ，パラメータの復元という提案手法が利用できることが分かった．

可変ストリームプリフェッチャに提案手法を適用した場合について 3 種類の比較を行った．ひとつ目に，可変ストリームプリフェッチャと提案手法の性能比較を行った．提案手法が可変ストリームプリフェッチャの性能を超えているベンチマークがある一方，ほとんど性能差が見られないベンチマークも多い．前者は学習するアプロー

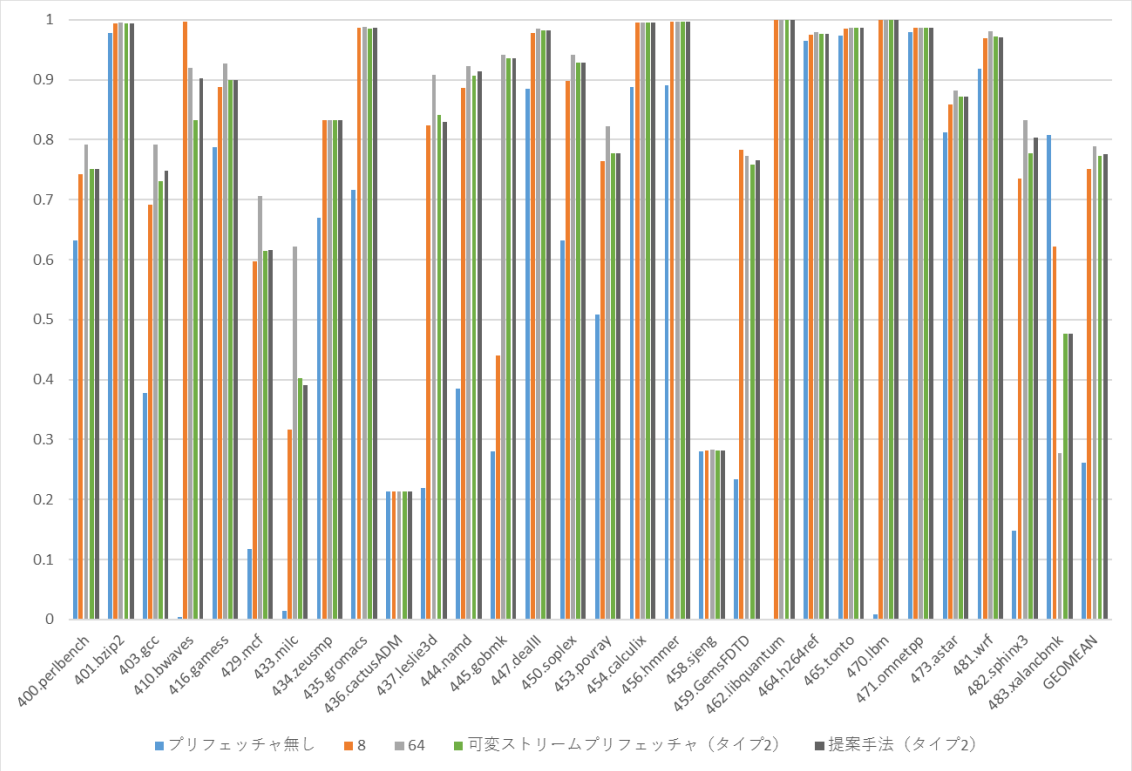

図 5.7: 可変ストリームプリフェッチャとの比較（タイプ 2-L3 ヒット率）

チが有効であることを示している一方で，後者は学習のアルゴリズムの欠陥の可能性が見られる．特に，固定値で動作するストリームプリフェッチャで提案手法よりも高い性能が出ていることは，適切なパラメータがあるにも関わらずそれを学習できていないことを示している．ふたつ目に BBV と高次ストライドの 2 種類のフェーズ検出手法同士の比較を行った．多くの場合で類似した性能を示すものの，多少の差が生まれることが確認された．特に BBV は 483.xalancbmk で大きく性能が低下するという欠陥が確認された．最後に，高次ストライドを利用する際のインターバルについて，1M および 100K 命令で提案手法を実行したときの性能比較を行った．多くの場合で 1M インターバルの方がわずかに優れた性能を示すが，一部のベンチマークでは 100K 命令の方が大きく性能が向上しており，100K インターバルでのみ検出できるフェーズの存在が推測される．

また，SHiP に提案手法を適用した場合について比較を行った．433.milc において高次ストライドは大きな性能低下を生じるが，それ以外では SHiP と同程度か SHiP を上回る性能を示すことが分かった．

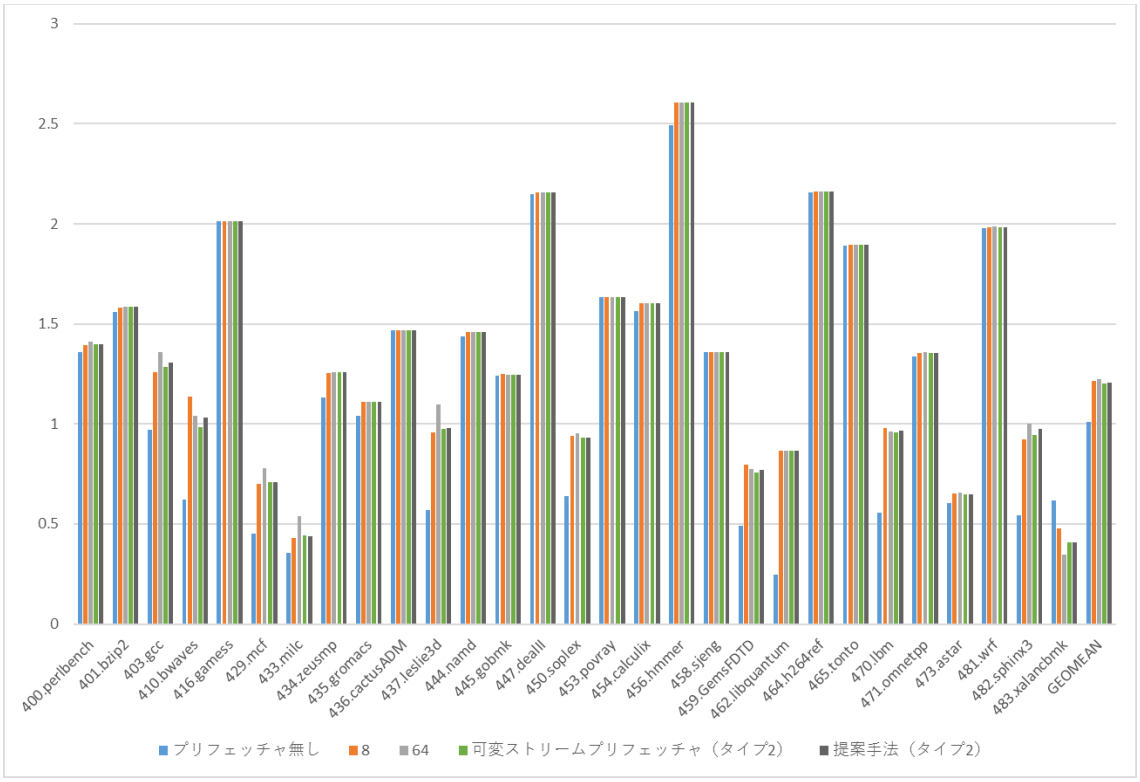

図 5.8: 可変ストリームプリフェッチャとの比較（タイプ 2-IPC）

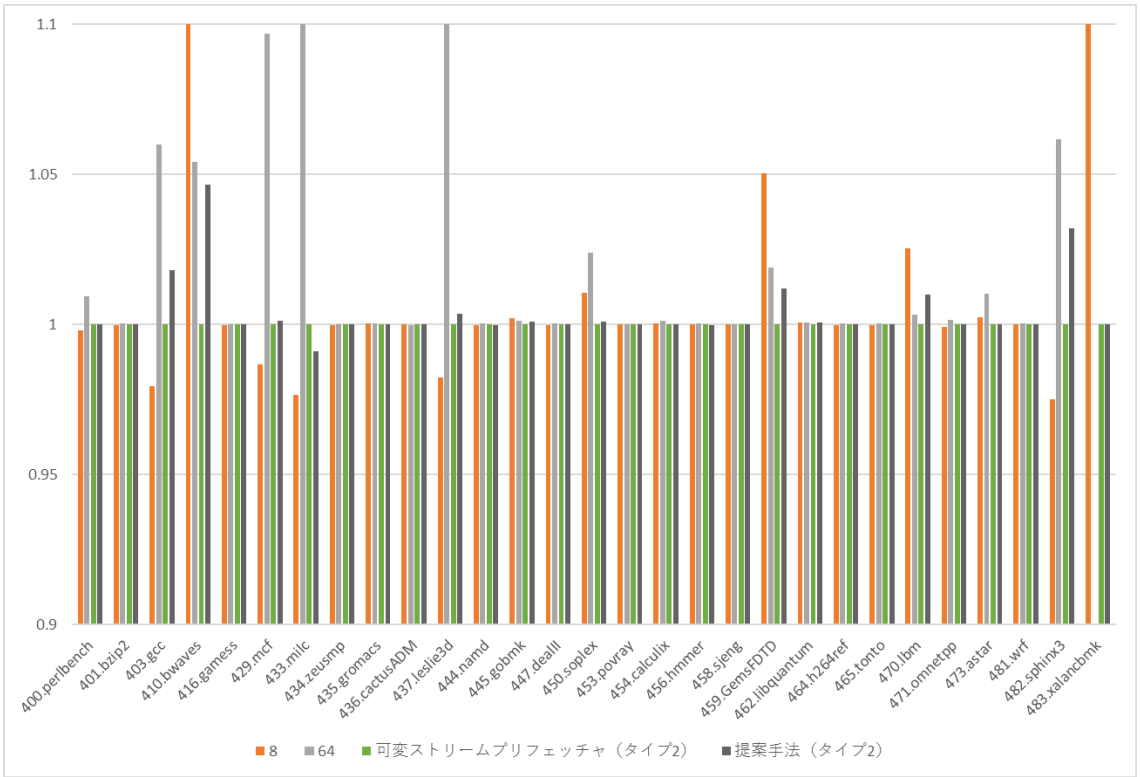

図 5.9: 可変ストリームプリフェッチャとの比較（タイプ 2-IPC，正規化）

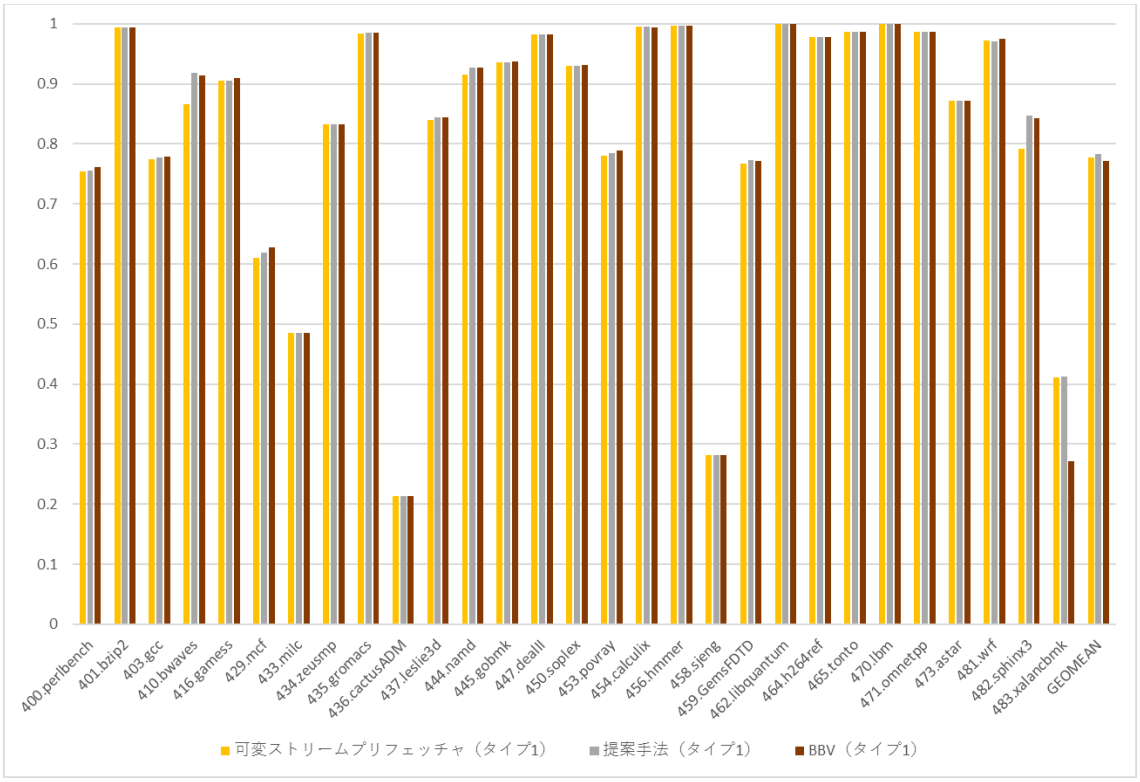

図 5.10: フェーズ検出手法の比較（タイプ 1-L3 ヒット率）

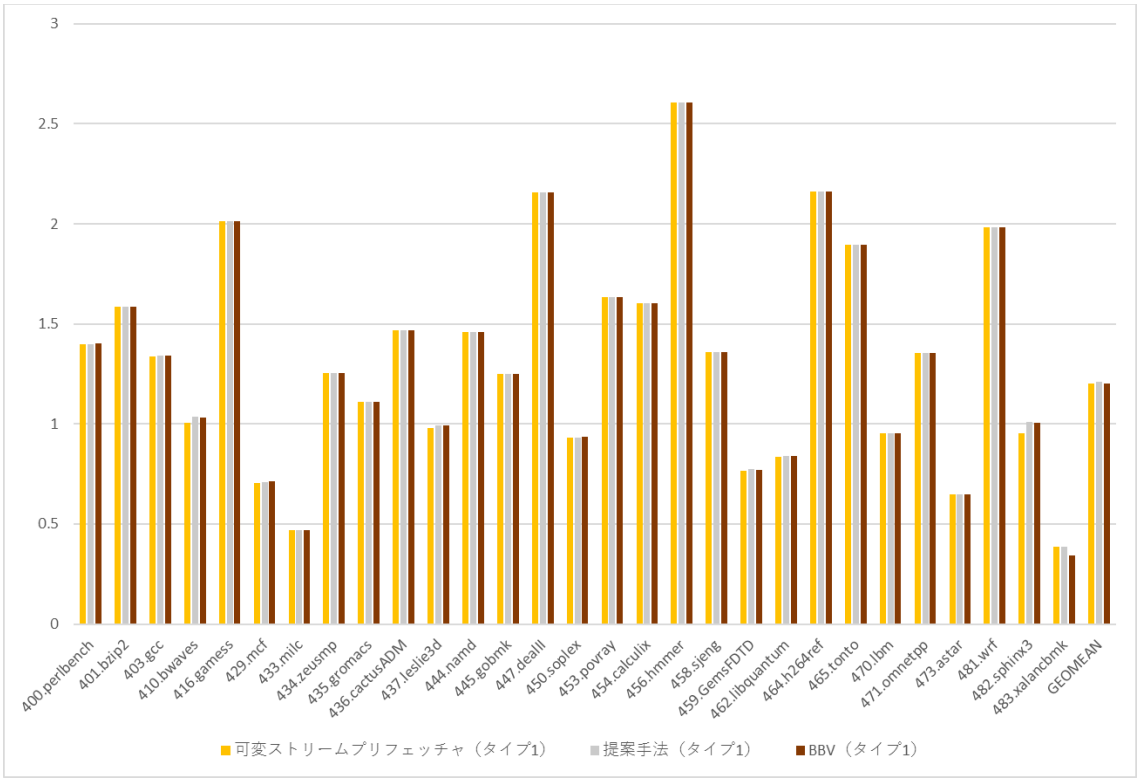

図 5.11: フェーズ検出手法の比較（タイプ 1-IPC）

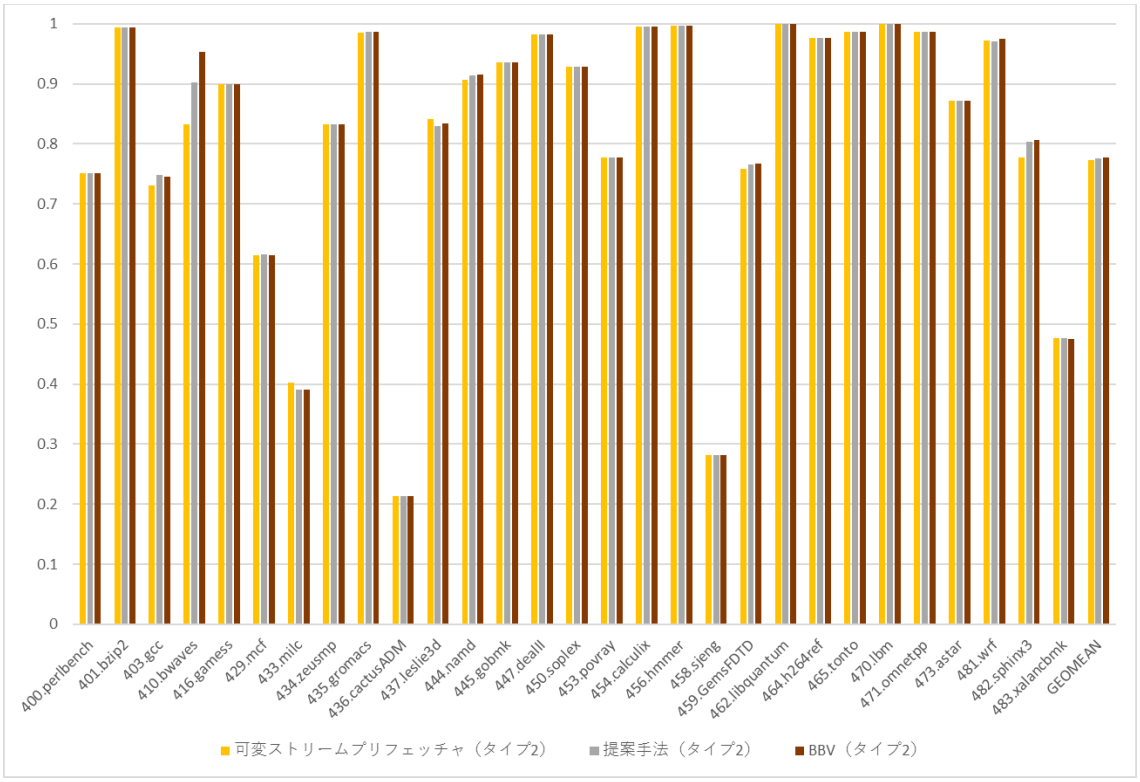

図 5.12: フェーズ検出手法の比較（タイプ 2-L3 ヒット率）

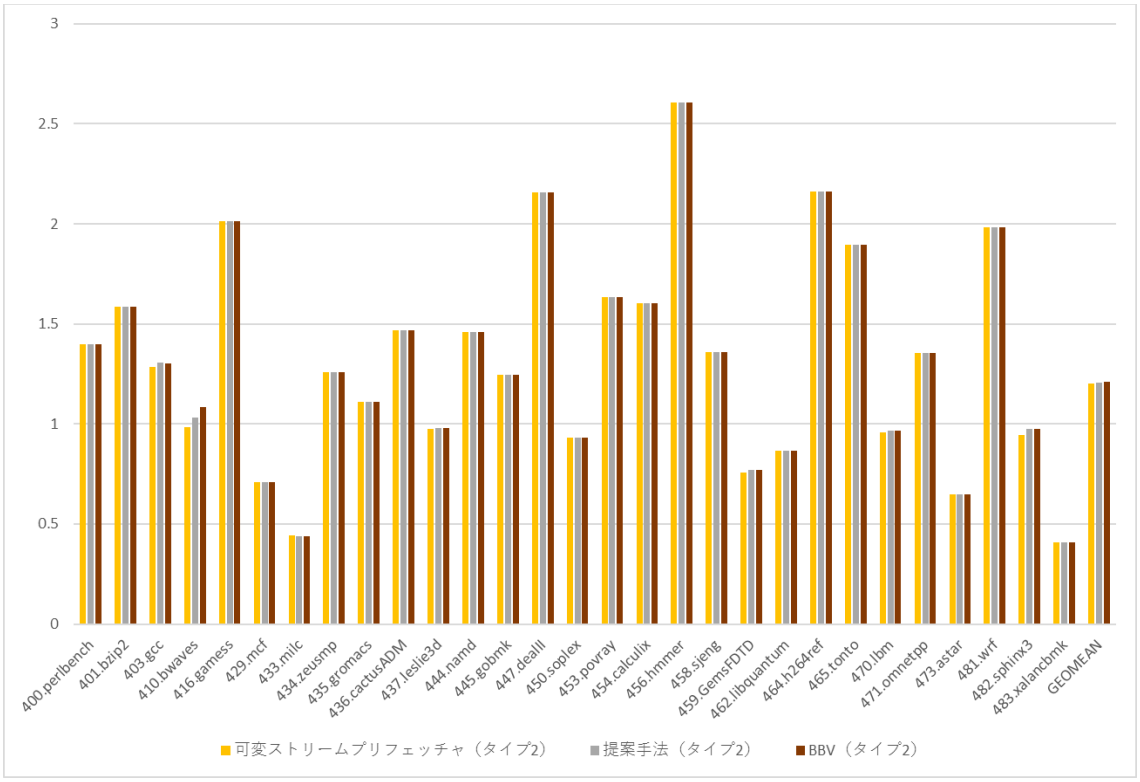

図 5.13: フェーズ検出手法の比較（タイプ 2-IPC）

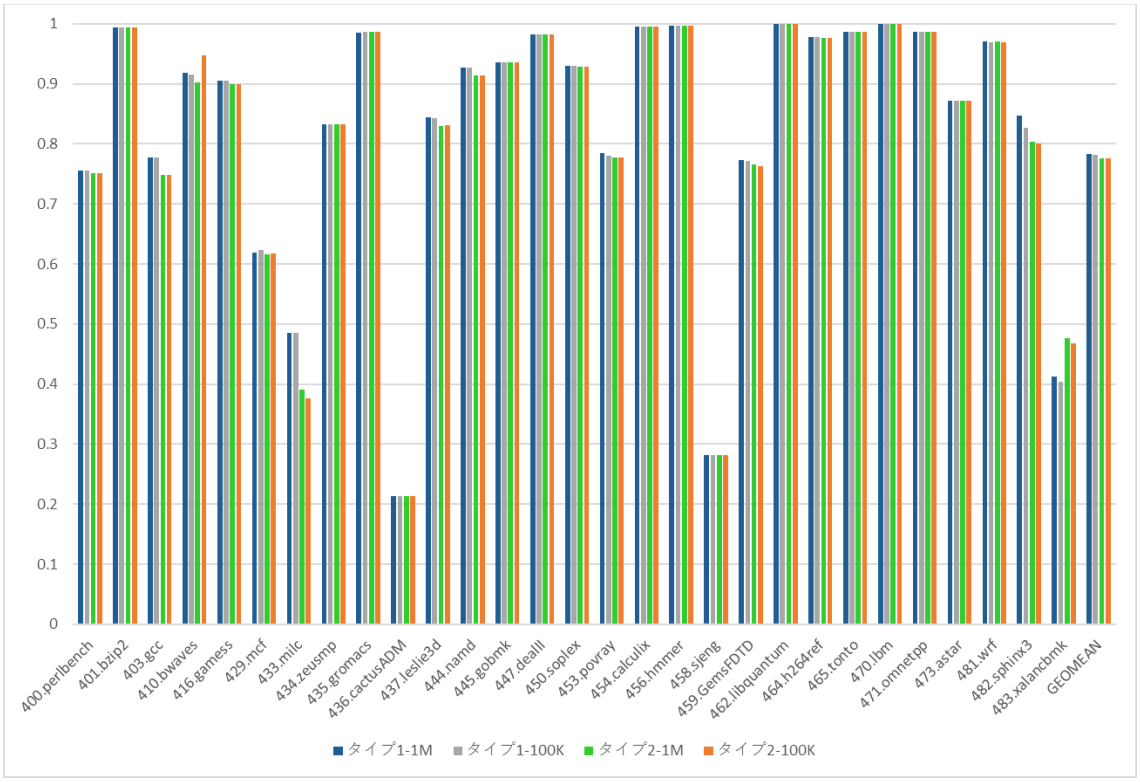

図 5.14: インターバル長による性能の違い（L3 ヒット率）

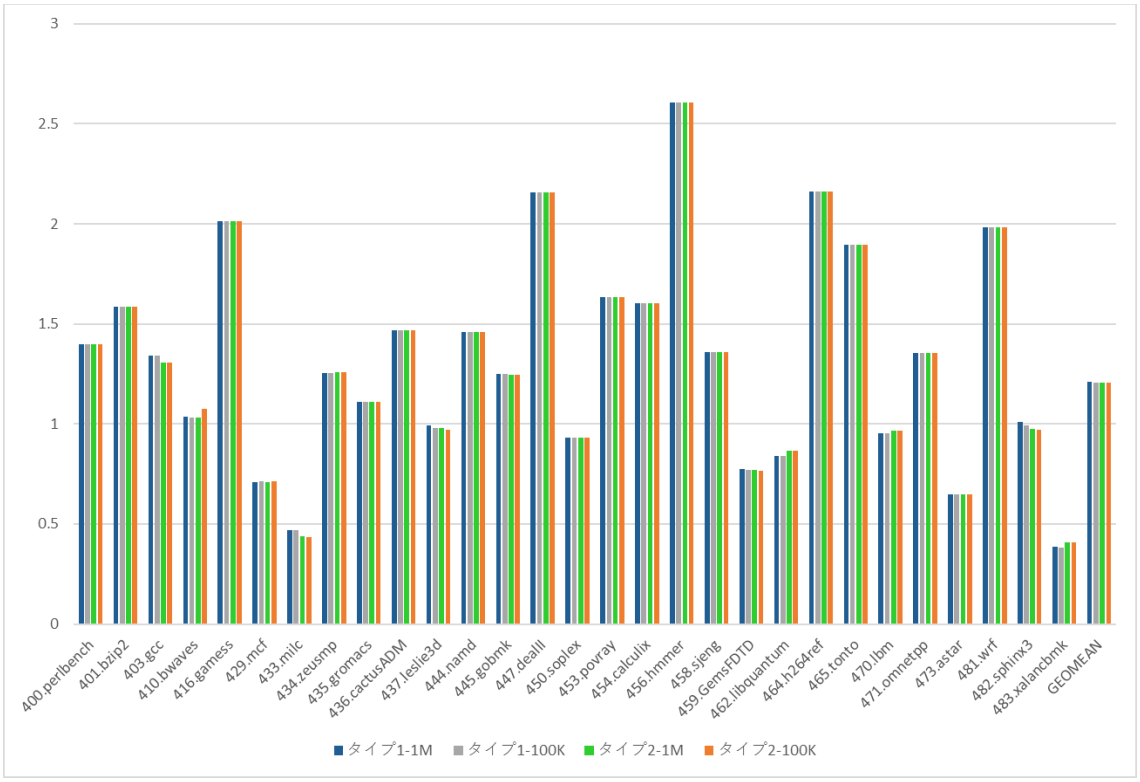

図 5.15: インターバル長による性能の違い（IPC）

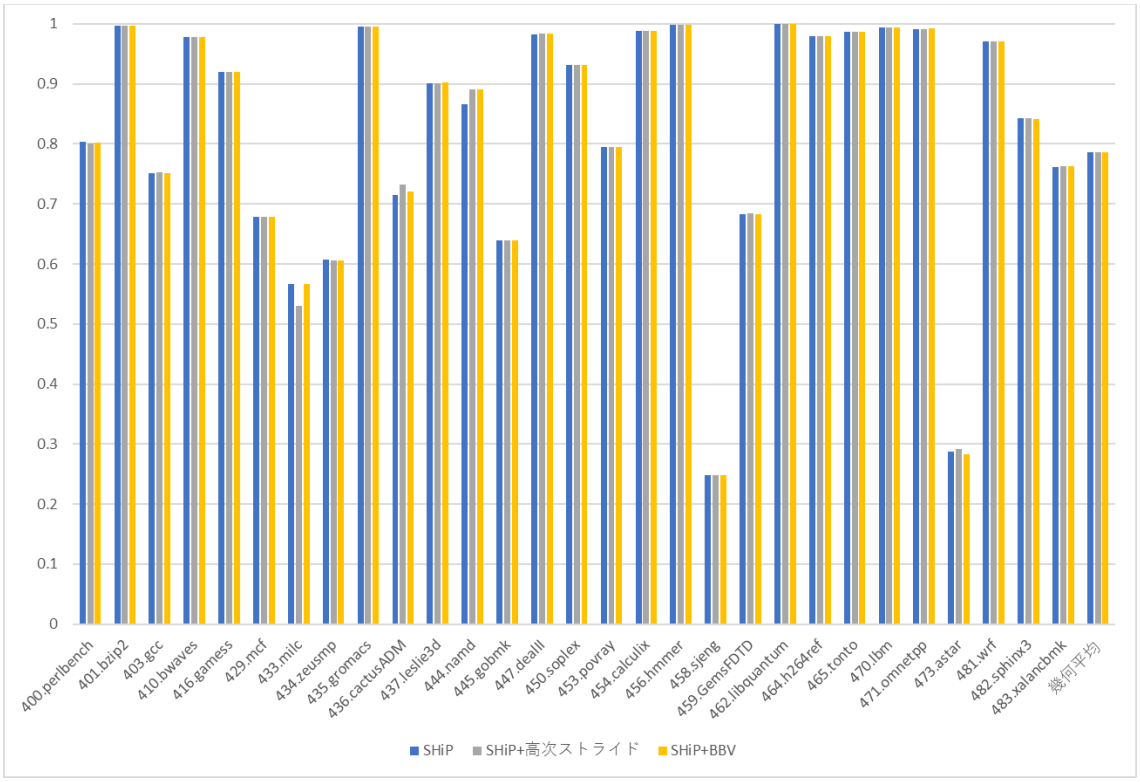

図 5.16: SHiP に適用したときの L3 ヒット率

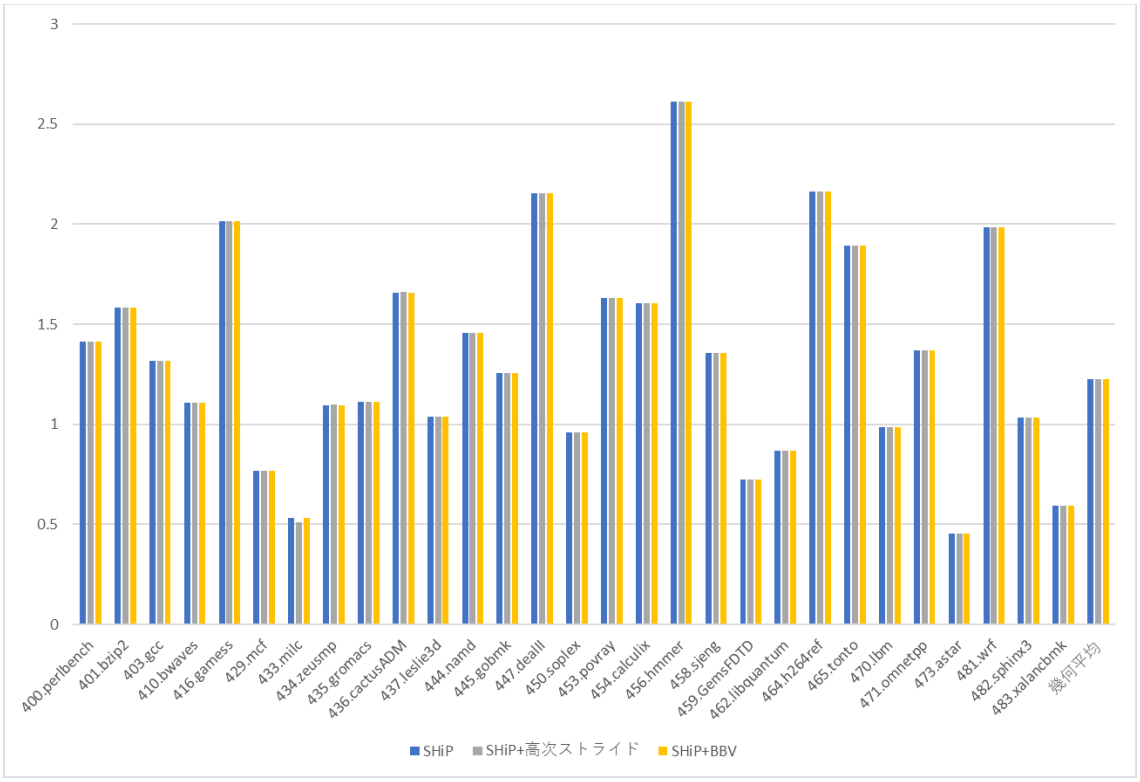

図 5.17: SHiP に適用したときの IPC

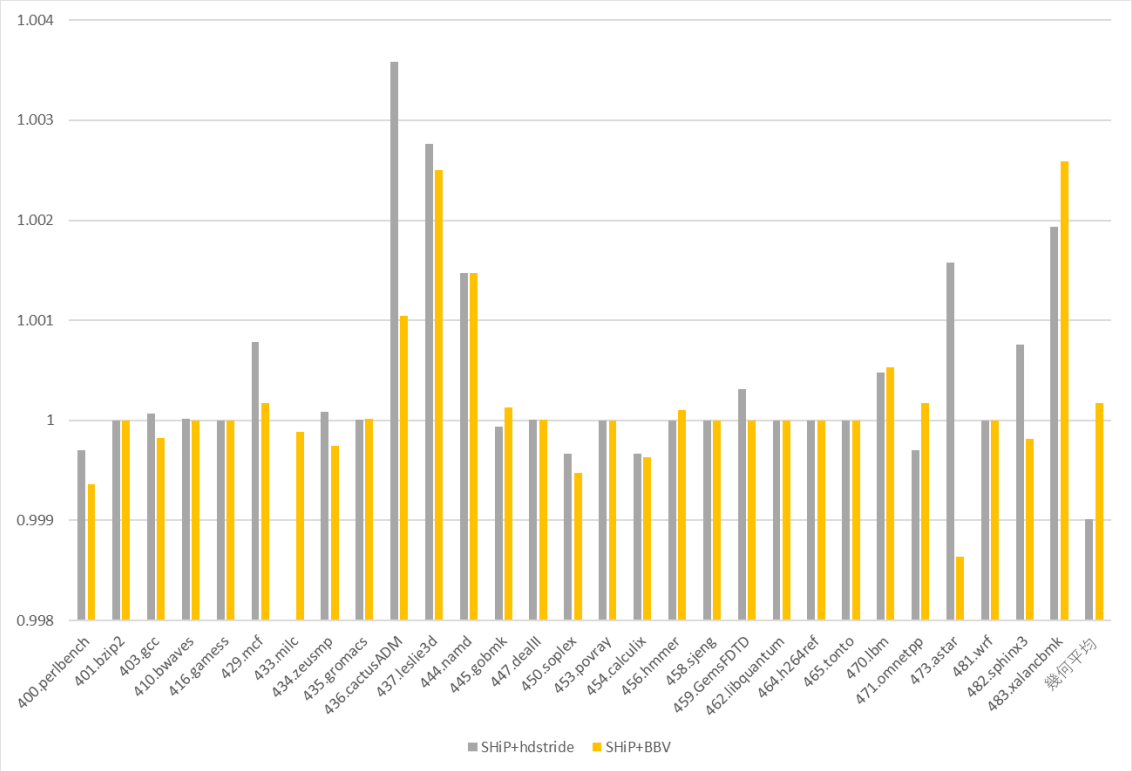

図 5.18: SHiP に適用したときの IPC（正規化）

第6章　考察

6.1　プリフェッチャの学習の理想化について

本論文の実験では理想的な環境を想定し，テーブルのカウント上限やエントリ数に制限を設けなかった．しかし，実プロセッサに搭載する上ではさらなる検討が必要である．

まず，カウントの上限について考える．フェーズごとに学習する都合上，学習にはある程度の時間がかかる．具体的な時間はプログラムごとに異なるため挙げられないが，何本のストリームがあったかできるだけ継続的にカウントするべきである．よって，定期的な初期化という方法を取ることは望ましくない．

ひとつの手段は，比較的小さな上限を設け，カウンタが飽和するたびにプリフェッチ量の初期値を更新する方法である．いずれかのエントリが上限に達したとき，全エントリを 0 に初期化しつつ，上限に達したエントリを元に初期値を設定する．この方法の場合，必要なリソースが，現在のインターバルに対するカウンタとフェーズごとの設定されている初期値を保存するだけの容量で済む．各フェーズについて過去のカウンタを保存しなくても良いため，容量効率が良い．一方で，局所的な外れ値に対して大きな影響を受ける．インターバル全体では短いストリームが多数現れるものの，途中の短期間に長いストリームが集中するような場合，カウンタが飽和してしまうとプリフェッチ量が更新されてしまう．インターバル長を十分短く取り，その期間を別のフェーズとすることで対処できるが，適切なインターバル長を予め定めることは難しい．

もうひとつの手段は，飽和するたびに全エントリを半分の値にする手法 [23] を利用することである．キャッシュメモリの置換アルゴリズムに関する論文中で提案されている手法である．この手法は過去の情報をある程度引き継ぐことができるため，この論文で提案している手法の特徴にも適している．

次に，エントリの数について考える．何種類のフェーズが検出されるのかは事前には分からない．そこで，キャッシュメモリの置換アルゴリズムと同様に，LRU などを用いて管理することが考えられる．キャッシュメモリとの違いとして，フェーズ検出器に変更を加えることでいくつかのフェーズをまとめて取り扱うことが可能である点が挙げられる．フェーズ検出器が区別しているフェーズのうち，テーブル

の傾向が類似しているものがあればそれらはまとめても性能が低下しない．まとめることによって容量効率を高めることが可能である．

6.2 提案手法の評価で利用したプリフェッチャのカウンタの閾値について

本論文では，カウンタを利用してプリフェッチ量の初期値を定める上でふたつの閾値を所与とした．4.2.3 で述べた，合計カウントの一定数および一定割合である．

ひとつめのパラメータである，提案手法を適用するために必要な，1 次元テーブルの合計カウント数について考える．この論文では 64 として評価を行った．各エントリがオーバーフローしない理想的なカウンタであることを想定したため，一度この値を超えてしまえば以降はずっと動作し続ける．つまりこの想定の元ではこのパラメータが与える影響は小さいため，1 桁などの極端に小さな値や，数千などの大きな値でなければ問題ない．理想的なカウンタでない場合について考察する．この値があまりに小さすぎると，はじめに現れたストリームが以降に大きな影響を及ぼしてしまう．逆にあまりに大きすぎると，プリフェッチ量の初期値に影響を与えることができず，フェーズ検出器の無い，通常のアダプティブなストリームプリフェッチャとして動作してしまう．長大なストリームが数本見られるときにそれを無視するためには，このパラメータともうひとつのパラメータの積が長大なストリームの本数を超えていればよい．無視する本数を定め，もうひとつのパラメータを決定した上で，必要な合計カウント数は定められる必要がある．

もうひとつのパラメータである，プリフェッチ量を定めるための，全ストリーム数に対する必要カウント数の割合について考える．この論文では 25 %として評価を行った．小さすぎると初期値が大きく設定され不必要に多くプリフェッチすることになり，大きすぎると逆にプリフェッチ量が不足することになる．全ストリームの半数を占めれば，その長さストリームが支配的であることは自明である．しかし，実際には短いストリームが多数検出されるため，半数では大きすぎることが分かっている．これはスタックの影響によるものである．スタックに複数の変数を確保する際，それがプログラム上配列でなかったとしても，メモリアクセスのみを見ると配列のように見え，それらへのアクセスがストリームとして観測されてしまう．スタックへのアクセスを無視した場合には半数で良いと考えられる．この論文ではスタックへのアクセスを無視しないため簡単に半数のさらに半分としたが，より適切な値が存在する可能性がある．

6.3 プリフェッチに適用した場合の提案手法における学習の特徴について

大きい値に収束しないよう，支配的なカウントに対応するプリフェッチ量のひとつ下の値をプリフェッチの初期値としていた．これが学習を阻害する可能性について考察する．

既にある程度学習が進み，初期値が 32 だったとする．このとき，ストリームにならないようなアクセスが生じ，プリフェッチャのエントリからそれが追い出されると，プリフェッチャは 32 をテーブルに対して出力する．結果，32 がテーブルにカウントされ，これが繰り返されるとプリフェッチ量を減らす方向に学習が進む．つまり，ストリームの本数に対してそれ以外のアクセスが多いと増やす方向の学習は進まないことになる．一方で提案手法は，一度ストリームが現れたときにそのストリームがどの程度の長さであるかを学習するものであるから，ストリームでないアクセスの影響は無視するべきである．実験の結果を考慮すると，多くのベンチマークにおいてストリーム以外の影響が大きく，学習に悪影響を与えていた可能性は十分考えられる．よって，どうストリーム以外のアクセスを無視するのか，無視したとしてどうプリフェッチ量を減らす学習を可能にするか，のふたつの課題がある．

そこで，プリフェッチャから追い出されるエントリが一度もプリフェッチを行っていない場合には無視する，という手法を考える．この場合，プリフェッチ量を減らす方向の学習をどう行えばよいか検討すればよい．最も簡単なのは，プリフェッチしたラインがミスしたかどうかをフィードバックすることである．プリフェッチした量の一定割合がミスしたらプリフェッチ量を減らす，などが考えられる．より理想的には，プリフェッチャが未追跡のストリームを検出して新しくプリフェッチをかけた際，プリフェッチされるラインがミスしていないか検出できるとよい．しかしこの手法の場合，キャッシュメモリにフラグを管理するためのビットを追加する必要があり，ミスしたかどうか分かるまで時間がかかる場合があるなどの問題がある．プリフェッチャの初期値設定のとき，ランダムに一部の初期値を変えることで学習させる手法も考えられる．基本的には提案手法と同様に処理し，一部は小さな初期値で始めることで，プリフェッチ量を減らす学習が可能である．ただし，プリフェッチャのプリフェッチ量のうち，最も小さい値まで初期値を減らすことはできない．テーブルのカウンタをインクリメントするためにはプリフェッチを行わなければならず，一度以上プリフェッチを行ったならば，小さい方からふたつ目以上のプリフェッチ量に対応するエントリしかインクリメントできないからである．

そこで，ランダムにプリフェッチ量の初期値を小さくしてプリフェッチを開始するのに加えて，提案手法で設定される初期値に対応するエントリはカウントを行わな

い，という改善手法が考えられる．例えばこの手法を用いると，提案手法によって初期値が 32 に設定されるとき，32 に対応するテーブルのカウントは増加せず，プリフェッチ量の初期値がときどき 16 や 8 に設定される．ストリーム以外のアクセスによってプリフェッチ量が 32 のままプリフェッチャから追い出しが起こったとしても，テーブルのカウントが増加しないため，初期値を減らす方向の学習は行われない．この方法ではどの時点からどの時点までカウントを行わないようにするかを考える必要がある．例として，ここで 6.1 節のカウンタを利用することを考えると，カウンタのうちひとつが飽和したときにそのカウンタのカウントを停止させ，別のカウンタも飽和したときに全カウンタの値を半分にする，という手法が考えられる．

6.4 SHiP に適用した場合の性能低下について

SHiP ＋高次ストライドでは，433.milc で大きな性能低下が生じた．SHiP ＋ BBV では性能が低下していないことから，フェーズ検出手法の問題であると推測できる．

433.milc について詳しく調査した結果を示す．図 6.1 は，433.milc における全メモリアクセスを，横軸に時間，縦軸にメモリアドレスを取り，キャッシュメモリのヒット率と合わせて示したものである．図 6.2 は，433.milc において高次ストライドによるフェーズ検出を用いたとき，検出されたフェーズに順番に数字を振り，各インターバルが何番のフェーズであるかをプロットしたものである．図 6.3 は，同様に BBV によるフェーズ検出を用いたときのフェーズ番号をプロットしたものである．

図 6.2 において，フェーズの再現が主に起きているのは 500 個目のインターバル前後の，0 番のフェーズが再現している箇所である．フェーズの再現が起こらない部分については，古い SHCT が参照されないため性能低下には繋がらない．よって，この 0 番フェーズが再現している箇所が原因であると推測される．図 6.3 でも同様に 0 番フェーズが再現している．一方で，初めに現れたときの長さが異なっている．このことから，高次ストライドを用いた際に 0 番フェーズと 1 番フェーズはほとんど同じ傾向なのにも関わらず区別されてしまい，0 番フェーズについて適切な量の学習ができなかったこと，この不十分な量の学習が偶然 SDM において良い性能を示し，広い範囲に適用された結果，適用された範囲では低い性能を示してしまったという可能性が考えられる．この推測が正しければ，この解決のためにはフェーズ検出の閾値を大きくすることが有効である．また，キャッシュメモリのセット分割をシグネチャと SDM を意識した大きさにすることで影響を減らすことができる可能性がある．

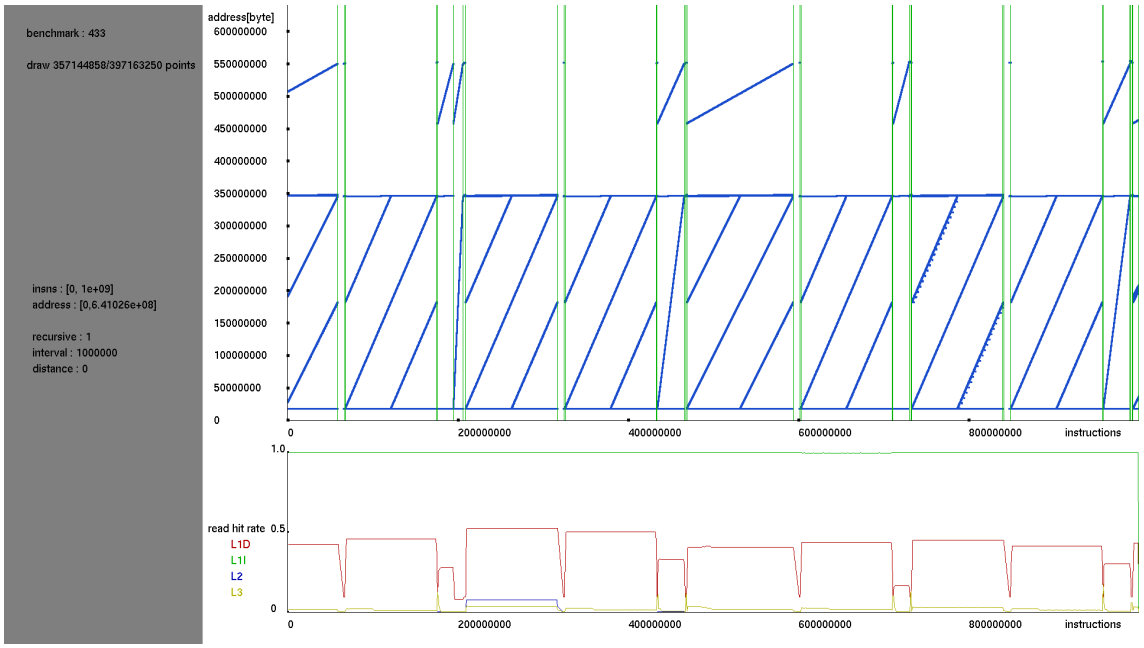

図 6.1: 433.milc におけるアクセスパターン

6.5 パラメータの復元の有効性について

SHiP に提案手法を適用した結果，僅かながらも性能が上がるケースが見られた．すなわち，過去の学習結果の方が有効であり，フェーズの再現までの間に不要な学習を行っていることが示された．

対策として，衝突の少ないよりよいシグネチャの検討が有効である．また，フェーズ再現時のパラメータ復元も有効である．しかしながら，SHCT は 16KB と大きく，フェーズごとに記憶するのは容量の無駄が大きく現実的ではない．そこで，保存する際に圧縮すること，飽和しているカウンタのみ記憶することといった工夫が必要である．

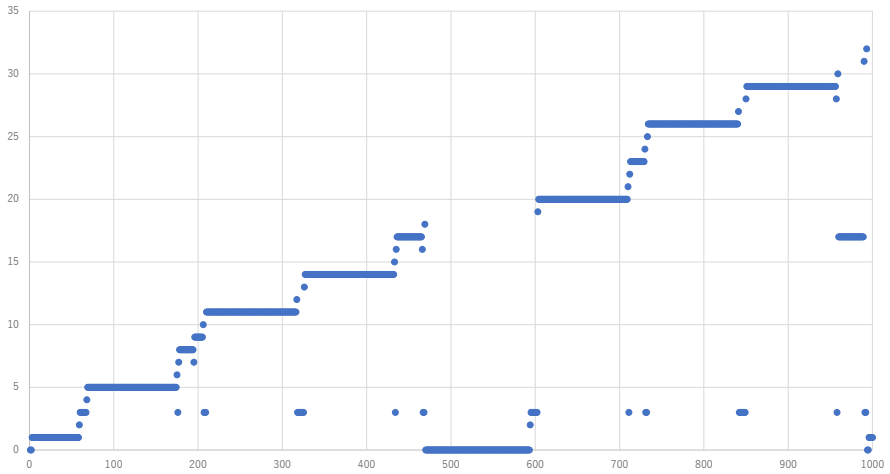

図 6.2: 433.milc において高次ストライドで検出されたフェーズ

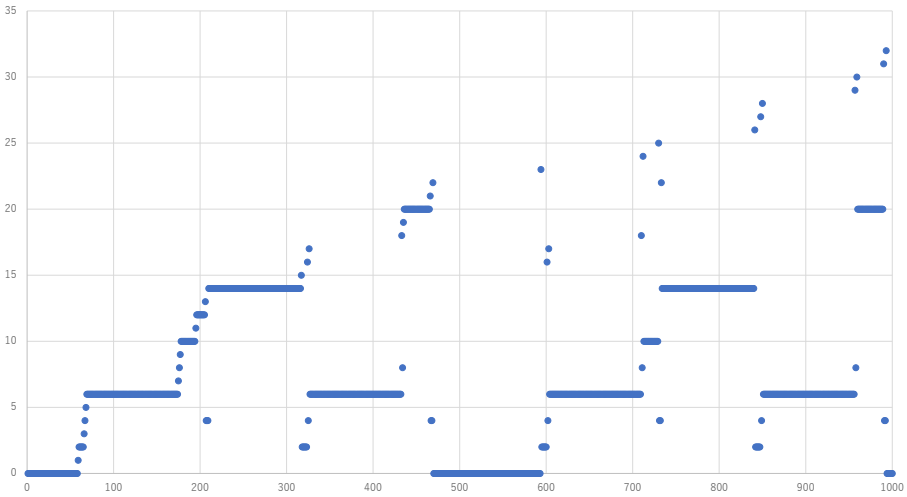

図 6.3: 433.milc において BBV で検出されたフェーズ

第7章　おわりに

本論文ではまずフェーズについて説明し，フェーズを学習に用いる手法を提案した．フェーズとパラメータをキーとしたテーブルを用いて，フェーズごとに最適なパラメータを学習し，復元する．プリフェッチ量可変のプリフェッチャと SHiP に利用する具体的な手法を示し，それぞれ性能評価を行った．

評価実験では学習の有効性が見られた．プリフェッチに適用した場合について，BBV が適さないベンチマークでも高次ストライドによる検出が機能することが確認された．一方，SHiP に適用した場合，高次ストライドを用いた場合のみ大きく性能が低下するベンチマークがあることが確認された．

高次ストライドにおける最適な閾値の設定には探索の余地があることが分かった．また，SHiP においてパラメータ再現による性能向上が僅かながら見られたことから，シグネチャの不適切な衝突が起きていることが分かった．

今後の課題として，高次ストライドを用いる際の最適な閾値についての探索が挙げられる．また，SHiP について，衝突の少ない，より良いシグネチャを生成するアルゴリズムの検討余地がある．また，提案手法が僅かながら有効であることから，学習による過去のパラメータの復元は有効な考えであることが示された．しかし，容量を考えるとコストが大きいため，容量を削減するための工夫が必要である．

謝辞

本論文の研究および執筆にあたり，研究室の皆様には大変お世話になりました．なんとか形になったことを喜ぶとともに，深く感謝するばかりです．

坂井修一教授には相談会やミーティングでの助言をいただいただけでなく，気付かないようなところでも支えていただきました．悩んでいるときにかけていただいた優しい労いの言葉が，とても嬉しかったことを覚えています．入江英嗣准教授には，研究や執筆に関する具体的な助言をいただきました．外部に投稿するか悩んでいたとき，力強い後押しが励みになりました．

秘書の八木原さん，赤羽さんには，居室の環境整備や備品の管理でお世話になりました．博士課程の野村さんには，研究や備品の管理上で細かい相談に乗っていただきました．

それ以外にも，同期をはじめとした研究室の皆様には大変お世話になりました．研究室の皆様，学生生活を援助してくれた両親，楽しい時間をくれた友人達・先輩方に，心より御礼申し上げます．

以上をもって，謝辞といたします．

参考文献

[1] P. Greenhalgh. Big.little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pp. 1–8, Sep. 2011.

[2] 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.

[3] 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.

[4] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 389–400, December 2012.

[5] M. Hind, V. Rajan, and P. F. Sweeney. Phase detection: A problem classification. Technical Report 22887, IBM Research, 2003.

[6] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. Composite Cores: Pushing Heterogeneity Into a Core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 317–328, Washington, DC, USA, 2012. IEEE Computer Society.

[7] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pp. 45–57, New York, NY, USA, 2002. ACM.

[8] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting Software Phase Markers with Code Structure Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pp. 135–146, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Transition Phase Classification and Prediction. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pp. 278–289, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Weihua Zhang, Jiaxin Li, Yi Li, and Haibo Chen. Multilevel Phase Analysis. *ACM Trans. Embed. Comput. Syst.*, Vol. 14, No. 2, pp. 31:1–31:29, March 2015.

[11] Ashutosh S. Dhodapkar and James E. Smith. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pp. 217–, Washington, DC, USA, 2003. IEEE Computer Society.

[12] 渋江陽人, 野村隼人, 入江英嗣, 坂井修一. ストライドアクセスの階層構造に着目したフェーズ検出. 電子情報通信学会技術研究報告 = IEICE technical report : 信学技報, Vol. 116, No. 510, pp. 9–14, 2017.

[13] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 60–71, New York, NY, USA, 2010. ACM.

[14] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, pp. 78–101, 1966.

[15] Alan Jay Smith. Cache Memories. *ACM Comput. Surv.*, Vol. 14, No. 3, pp. 473–530, September 1982.

[16] Myoung Kwon Tcheun, Hyunsoo Yoon, and Seung Ryoul Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *Proceedings of the 1997 International Conference on Parallel Processing (Cat. No.97TB100162)*, pp. 306–313, August 1997.

[17] Binny S. Gill and Luis Angel D. Bathen. Optimal Multistream Sequential Prefetching in a Shared Cache. *Trans. Storage*, Vol. 3, No. 3, October 2007.

[18] 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.

[19] Standard Performance Evaluation Corporation. Spec cpu® 2006. https://www.spec.org/cpu2006/.

[20] 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 SACSIS2009, Vol. 2009, No. 4, pp. 120–121, 2009.

[21] 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, February 2007.

[22] Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. SHiP + + : Enhancing Signature-Based Hit Predictor for Improved Cache Performance. 2017.

[23] Subhasis Das, Tor M. Aamodt, and William J. Dally. Reuse Distance-Based Probabilistic Cache Replacement. *ACM Trans. Archit. Code Optim.*, Vol. 12, No. 4, pp. 33:1–33:22, October 2015.

発表論文

[1] 渋江陽人, 野村隼人, 入江英嗣, 坂井修一. ストライドアクセスの階層構造に着目したフェーズ検出. システム・アーキテクチャ研究発表会（2016-ARC-217）. 一般社団法人情報処理学会, Mar., 2017.

[2] 渋江陽人, 野村隼人, 入江英嗣, 坂井修一. プログラムフェーズを利用した最適キャッシュ管理パラメータの再現手法. 研究報告システム・アーキテクチャ（2018-ARC-232）. 一般社団法人情報処理学会, Jul., 2018.