インタプリタにおけるコード配置効果による性能変動のマイクロアーキテクチャ要因分析

高 磊坪　阿部 奏太　鵜川 始陽　塩谷 亮太

バイトコードインタプリタは幅広い言語処理系で用いられており，その実行性能の向上は重要な課題である．その高速化手法の一つに，関数や基本ブロックのバイナリ上の配置（コード配置）を最適化する方法がある．先行研究では，インタプリタの命令ハンドラ順序を遺伝的アルゴリズムで最適化し，平均 7.7%の性能向上を達成した．この結果は，処理内容が同じでも，命令ハンドラのコード配置によってインタプリタの実行性能が変化することを示している．しかし，この最適化では，実行時間だけを評価指標とするため，性能向上の要因を説明できない．また，候補ごとにインタプリタのビルドとベンチマークの実行を繰り返すため，探索コストが大きい．そこで本研究では，Ruby 3.3.1 のバイトコードインタプリタを対象に，実機プロファイリングとトレース駆動シミュレーションを用いて，命令ハンドラのコード配置による性能変動の要因を分析した．実機プロファイリングでは，多くの CPU とベンチマークの組合せで，フロントエンドからの命令供給の遅延と投機失敗後のパイプライン再開を示す指標が，実行時間と正の相関を示した．トレース駆動シミュレーションでは，コード配置に伴う命令アドレスの変化が，小容量の命令キャッシュと分岐先バッファにおけるセット割り当てと競合状態を変化させることを確認した．さらに，マイクロベンチマークにより，これらの構造におけるアドレス依存のセット競合が実機性能を低下させることを確認した．以上の結果は，命令ハンドラのコード配置を最適化する際に，配置後の命令アドレスとセット競合を考慮することの重要性を示唆する．

1　はじめに

バイトコードインタプリタは，多くのプログラミング言語処理系で広く利用されている．例えば，Ruby と Python の主要な処理系である CRuby [18] と CPython [15] は，バイトコードインタプリタによってプログラムを実行する．バイトコードインタプリタは，ソースプログラムから生成されたバイトコードを読み出し，対応する命令ハンドラを実行する．この処理はバイトコードごとに繰り返されるため，バイトコードの読み出し，命令ハンドラへの制御移動，および命令ハンドラの実行に伴うコストは，プログラム全体の実行性能に大きな影響を与える．

バイトコードインタプリタの実行性能に影響を与える要因の一つに，プロセッサのフロントエンドにおける命令供給がある．現代の高性能プロセッサでは，この命令供給を低遅延で継続するため，命令キャッシュや分岐先バッファ（Branch Target Buffer：BTB）などの構造をフロントエンドに備えている．これらの構造は情報を高速に参照できる一方で，保持できる情報の量は限られている．必要な命令や分岐先情報が保持されていない場合，フロントエンドからの命令供給が滞り，実行性能が低下する．

我々は，コード配置が上記のようなフロントエンド構造における競合の発生状況を変化させ，それによってバイトコードインタプリタの実行性能に大きな影響を与える可能性に着目する．一般に，コード配置を変更すると，各命令のアドレスも変化するため，命令キャッシュや BTB における競合の発生状況も変化する．特に，バイトコードインタプリタでは，一部の命令ハンドラとその周辺コードが短い間隔で繰り返し実行される．そのため，頻繁に使用される命令や分岐先情報が競合すると，命令供給の停滞が繰り返される場合がある．

したがって，コード配置によってこれらの競合を抑

Microarchitectural Analysis of Performance Variation Due to Code Layout Effects in Interpreters

Leiping Gao, Kanata Abe, Tomoharu Ugawa, Ryota Shioya, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

えることで，バイトコードインタプリタの実行性能を改善できる可能性がある．このようなコード配置と実行性能の関係を利用した手法として，Huang ら [6] は，組込み向けの JavaScript 仮想マシンである Embedded JavaScript Virtual Machine（eJSVM）[8] の命令ハンドラの配置順序を遺伝的アルゴリズム（Genetic Algorithm：GA）によって最適化する手法を提案した．この手法により，CPU とベンチマークの組合せごとに 0.8〜23.0%（平均 7.7%）の実行速度向上を達成した．

しかし，Huang らの GA による最適化には，性能改善の要因を説明できないことと，探索に大きな計算コストを要することという二つの課題がある．第一に，ベンチマークの実行時間だけを適応度として CPU をブラックボックスとして扱うため，高性能な配置を発見できても，性能が向上した要因を説明できない．第二に，候補となる命令ハンドラの配置順序ごとにインタプリタのビルドとベンチマークの実行を繰り返すため，探索には膨大な計算コストを要する．性能変動の要因を明らかにし，その知見を探索空間の絞り込みや配置設計に利用できれば，探索の効率化とさらなる性能向上につながる可能性がある．

そこで本研究では，命令ハンドラのコード配置による性能変動の要因をマイクロアーキテクチャの観点から分析する．具体的には，Ruby 3.3.1 [19] のバイトコードインタプリタを対象として，実機プロファイリング，トレース駆動シミュレーション，およびマイクロベンチマークを用いる．実機プロファイリングでは，複数の CPU 上で命令ハンドラの配置順序が異なるインタプリタを実行し，トップダウン・マイクロアーキテクチャ解析手法（Top-down Microarchitecture Analysis Method：TMA）[21] を用いて，実行時間とハードウェア性能指標の関係を分析する．トレース駆動シミュレーションでは，分岐予測器や命令キャッシュの状態など，実機では直接観測しにくい CPU 内部の挙動を解析する．マイクロベンチマークでは，命令キャッシュと BTB におけるアドレス依存のセット競合が実機性能に与える影響を検証する．

実機プロファイリングでは，TMA を用いた定量的な分析により，CPU による違いはあるものの，コード配置による性能変動がフロントエンドの命令供給および投機実行に関する問題と関係することを確認した．多くの CPU とベンチマークの組合せにおいて，TMA の分類項目であるフロントエンド依存（Frontend Bound）と投機の問題（Bad Speculation）の割合が，実行時間と正の相関を示した．トレース駆動シミュレーションでは，コード配置に伴う命令アドレスの変化が，命令キャッシュと BTB におけるセットの割り当てと競合状態を変化させることを確認した．さらに，マイクロベンチマークにより，これらの構造におけるアドレス依存のセット競合が実機性能を低下させることを確認した．以上の結果は，命令ハンドラのコード配置を最適化する際に，配置後の命令アドレスとセット競合を考慮することの重要性を示唆している．

2 バイトコードインタプリタとコード配置最適化

2.1 バイトコードインタプリタ

バイトコードインタプリタは，プログラムの命令をバイトコードと呼ぶ中間表現に変換して，仮想マシン上で実行する．バイトコードは，処理の種類を示すオペコードと，処理に用いるオペランドから構成される．バイトコードインタプリタは，バイトコードを一つずつ読み出して実行する．

各オペコードに対応する処理を実装したコードのことを命令ハンドラと呼ぶ．各命令ハンドラは，演算や変数アクセスなど，それぞれのバイトコードの処理を担う．プログラムの実行時には，バイトコードインタプリタがバイトコードを読み出し，オペコードに対応する命令ハンドラを実行する処理を繰り返す．図 1 に，プログラムのソースコードをコンパイルしてバイトコードを生成し，バイトコードインタプリタがそれを一命令ずつ読み出して実行する流れを示す．

この反復処理において，ハンドラ間の制御の移動には間接分岐命令が用いられることが多い．そのため，バイトコードインタプリタでは，ハンドラ間の間接分岐命令が高頻度に実行される場合が多い．

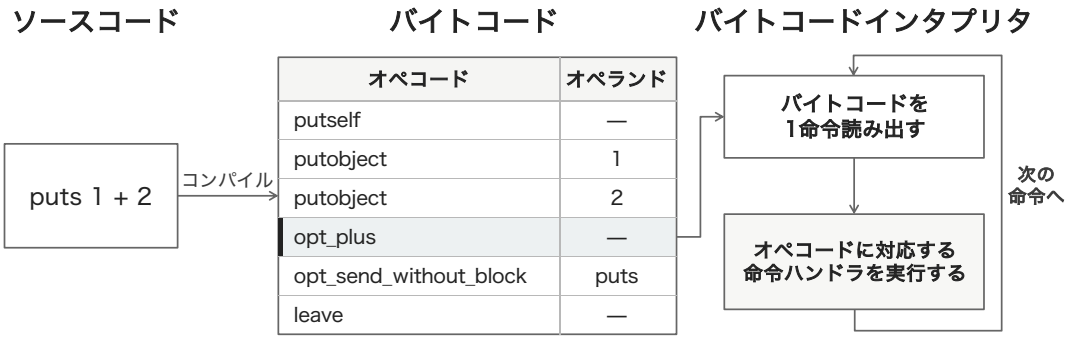

図 1　バイトコードインタプリタによるプログラム実行の流れ

2.2　実行ファイル上のコード配置と性能

プログラムの処理内容が同じであっても，コード配置によって実行性能に差が生じることがある．ここで，コード配置とは，コンパイル後の実行ファイルにおいて，関数や基本ブロックなどのコードがメモリ上に配置される位置や順序を指す．たとえば BOLT は，実行プロファイルを参考にコード配置を最適化しており，命令キャッシュや Instruction Translation Lookaside Buffer（iTLB）のミス，分岐予測ミス数を削減することで高速化を実現している [13].

本研究で対象とする命令ハンドラの順序の変更は，コード配置を変更する方法の一つである．命令ハンドラの順序を変更すると，各命令ハンドラのメモリ上の位置が変化する．その結果，プログラムの処理内容が同じであっても，命令キャッシュや iTLB，分岐予測器などの挙動が変わり，実行性能に差が生じ得る．これらの動作については，第 3 節で説明する．

2.3　GA による命令ハンドラの順序の最適化

Huang ら [6] は，JavaScript インタプリタ eJSVM を対象に，遺伝的アルゴリズム（GA）を用いて命令ハンドラの順序を最適化した．GA による最適化では，ハンドラを並べ替えた eJSVM をビルドしてベンチマークを実行し，実行時間を計測する．実行時間をもとに選択（淘汰），交叉，突然変異を繰り返すことで，CPU とベンチマークの組合せごとに性能の高い順序を探索する．

ARM および x86 プロセッサを用いた評価では，GA によって得られたハンドラ順序によって最適化対象のベンチマークの実行速度が 0.8〜23.0%（平均 7.7%）向上したと報告されている [6]．一方で，この手法は CPU 内部の挙動を考慮せず，ベンチマークの実行時間のみを基準として探索するため，特定の配置によって性能が向上した原因は明らかにされていない．

3　マイクロアーキテクチャと性能分析

本節では，背景としてプロセッサのマイクロアーキテクチャと，性能の分析方法について説明する．

3.1　アウト・オブ・オーダー実行プロセッサの構成

現代の代表的な高性能プロセッサは，パイプライン処理やスーパースカラ方式，アウト・オブ・オーダー実行などによってプログラムを高速に実行している．パイプライン処理は命令処理を複数の段階（ステージ）に分割して実行する方式である．スーパースカラ方式では，複数のパイプラインを用いて命令を並列に処理する．アウト・オブ・オーダー実行では，プログラム順とは無関係に実行可能な命令から順に実行する．これらの方式を活用することで，現代の高性能プロセッサは単位時間当たりに処理できる命令数を増やしている．

図 2 に，アウト・オブ・オーダー実行プロセッサにおける命令処理の流れを示す．命令のパイプライン処理は，大きくフロントエンド，バックエンド，リタイ

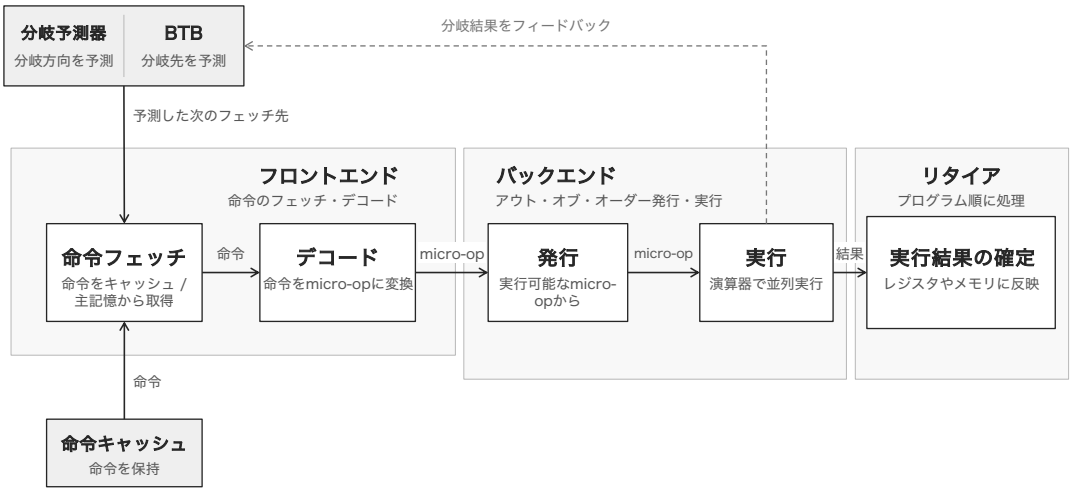

図 2　アウト・オブ・オーダー実行プロセッサにおける命令処理の流れ

アの各段階に分けられる．フロントエンドは実行する命令をフェッチしてデコードし，プロセッサ内の処理の単位であるマイクロ命令（micro-operation：μop）に分割した上でバックエンドに供給する．バックエンドは，命令間の依存関係を考慮しながら，実行可能な μop を演算器に発行する．実行が完了した命令はプログラム順にリタイアされ，実行結果がレジスタやメモリなどに反映される．

プロセッサが高い性能を発揮するためには，フロントエンドからバックエンドへ十分な数の μop を継続的に供給する必要がある．命令供給を継続するために，フロントエンドでは命令を保持する命令キャッシュや，分岐命令の分岐先アドレスを保持する BTB などのキャッシュが用いられる．命令キャッシュミスや BTB ミスが発生すると，下位のキャッシュや主記憶からの命令取得や分岐先の解決を待つ必要があり，命令供給が停滞してパイプラインの処理効率が低下する．

これらのキャッシュでは命令アドレスに基づいてデータの格納先が決められるため，コード配置によって格納位置の競合の発生が変化する．以下では，命令キャッシュ，BTB，キャッシュで発生する競合，アウト・オブ・オーダー実行プロセッサの性能分析手法について説明する．

3.2　命令キャッシュ

命令キャッシュは，実行に必要な命令データをプロセッサの近くに保持し，命令フェッチにかかる遅延を減らす記憶機構である．

一般に，プロセッサは階層化された複数のキャッシュを持つ．Level 1（L1）命令キャッシュは小容量で低遅延であり，その下位には L1 命令キャッシュよりも大容量だがデータの取得に時間がかかる Level 2（L2）および Level 3（L3）キャッシュが配置される．フェッチする命令が L1 命令キャッシュに存在しない場合は，下位のキャッシュを確認する．下位のキャッシュにも命令が存在しない場合は主記憶から取得する．L1 命令キャッシュに命令が存在しない場合は命令のフェッチに時間がかかり，バックエンドへの μop の供給が停滞する可能性がある．

命令キャッシュは，一般にセットアソシアティブ（set-associative）方式が採用されており，命令アドレスに基づいて格納先が決定される．セットアソシアティブ方式で発生する競合については，第 3.4 節で説明する．

3.3　分岐予測と分岐先バッファ（BTB）

プログラムには実行結果に応じて次に実行する命令アドレスを変化させる分岐命令が含まれている．分岐先アドレスが確定するのはバックエンドだが，分岐先

が確定するまで命令フェッチを止めていると μop の供給が止まってしまう．そのため，分岐が成立するかを予測する方向予測と，分岐先アドレスを予測する分岐先予測によって，投機的に分岐先の命令をフェッチしている．予測が外れると，投機的に実行した誤った経路上の命令を破棄する必要がある．命令の破棄と正しい分岐先からの処理の再開には時間がかかるため，分岐方向や分岐先アドレスの予測ミスが発生するとパイプラインの利用効率が低下する．

BTB は，分岐命令のアドレスと分岐先アドレスの対応を過去の分岐先の結果から学習し，命令フェッチ段階で分岐先を予測する記憶機構である．BTB ミスが発生すると分岐先の取得が遅れ，フロントエンドからの μop 供給が停滞する．BTB はセットアソシアティブ方式で構成され，命令キャッシュと同じように複数階層化される場合がある．分岐先情報の格納先は分岐命令のアドレスに基づいて決定される．

3.4 セット競合

セットアソシアティブ方式のキャッシュは，複数のセットから構成され，各セットには way 数分のキャッシュラインが存在する．データを格納するときは，まずアドレスに基づいて一つのセットを選択し，そのセット内のいずれかのキャッシュラインにデータを格納する．

図 3 に，512 ライン，256 セットで構成される 2-way セットアソシアティブキャッシュの例を示す．この例では，主記憶のブロック番号を 256 で除した余りによって，格納先のセットが決定される．したがって，ブロック 0，256，512，768 はいずれもセット 0 に割り当てられる．一方，セット 0 を構成するキャッシュラインは 2 本であるため，これらのブロックのうち同時に保持できるのは 2 個までである．

同時に使用されるデータ数が一つのセットの way 数を超えると，いずれかのデータがキャッシュラインから追い出される．追い出されたデータを再び参照すると，キャッシュミスが発生する．同じセットに割り当てられるデータが way 数を超え，それらが繰り返し参照される場合，同一セット内でデータの追い出し合いが発生する．このような複数のラインが同一のセットを使用する状況をセット競合と呼ぶ．

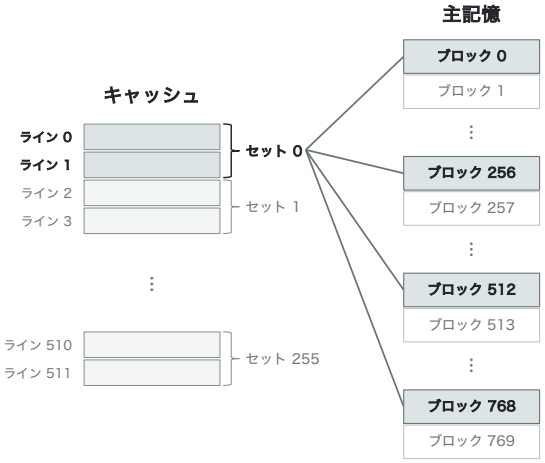

図 3　セットアソシアティブキャッシュの構成

コード配置を変更すると，分岐命令を含む命令のアドレスが変化する．したがって，実行する処理が同じであっても，命令キャッシュや BTB のセットへの割り当てが変わり，セット競合の発生状況も変化する．後のマイクロベンチマークを用いた評価では，アドレスに依存するセット競合が実際に性能へ影響し得ることを確認する．

3.5 トップダウン・マイクロアーキテクチャ解析手法（TMA）

アウト・オブ・オーダー実行プロセッサでは，命令キャッシュミス，分岐予測ミス，ロード命令のデータ待ち，実行資源の競合など，複数の性能低下要因が時間的に重なって発生する．したがって，個別のハードウェア性能イベントの発生回数だけでは，各イベントが性能低下へどの程度寄与したかを直接評価できず，重なり合う要因を重複なく分類することも難しい．

トップダウン・マイクロアーキテクチャ解析手法（Top-down Microarchitecture Analysis Method：TMA）は，プロセッサが 1 サイクルに処理できる μop 数をスロットという単位で数え，その使用状況を排他的かつ階層的に分類することで，この問題に対処する [21]．TMA では，総実行サイクル数で実行できたはずの最大 μop 数を総スロット数として，各ス

ロットを最上位で Retiring，Frontend Bound，Bad Speculation，Backend Bound のいずれかに分類する．

- Retiring は，μop が正常に処理され，最終的にリタイアされたスロットを指す．
- Bad Speculation は，分岐予測ミスなどによって，リタイアされない μop の処理やパイプラインの回復に費やされたスロットを指す．
- Frontend Bound は，フロントエンドからバックエンドへ十分な μop を供給できなかったスロットを指す．
- Backend Bound は，データ待ちや実行資源の不足などにより，バックエンドで処理が停滞したスロットを指す．

TMA では，リタイア情報や投機実行の誤りからの回復情報などを用いて Retiring と Bad Speculation を算出し，残りのスロットをフロントエンドからの命令供給状況に基づいて Frontend Bound と Backend Bound に分類する．各スロットを最上位の四分類のいずれかに割り当てることで，同じ処理能力を複数の要因へ二重に計上することなく，性能低下の要因がフロントエンド，投機の問題，またはバックエンドのどこに集中しているかを把握できる．さらに，各分類を階層的に細分化することで，プロセッサ全体の概観から，より具体的な性能のボトルネックへと段階的に調査対象を絞り込める．Linux の perf[10] では，ハードウェア性能カウンタを用いて TMA に対応する指標を取得できる [9]．

本研究では，命令ハンドラの順序と実行時間の関係を TMA によって分析する．TMA により，コード配置による性能変動がフロントエンド，投機実行，またはバックエンドのいずれと関係するかを絞り込み，後続のトレース駆動シミュレーションで詳細に解析すべき機構を選定する．

4 研究目的と分析の方針

第 2 節で述べたように，GA による命令ハンドラ順序の最適化は高性能なコード配置を発見できる一方，性能改善の要因を説明できず，探索コストも大きい．第 3 節で説明したように，コード配置に伴う命令アドレスの変化は，命令キャッシュや BTB のセットの割り当てを変化させるため，実行性能に影響する．本節では，コード配置，マイクロアーキテクチャ上の挙動，および実行性能の関係を明らかにするため，本研究の研究課題，仮説，および分析方針を示す．

4.1 目的

本研究では，命令ハンドラの順序の変更に伴う命令アドレスの変化がマイクロアーキテクチャの挙動とどのように関係し，配置間の実行時間差として現れるのかを分析する．具体的には，命令ハンドラの配置によって命令キャッシュおよび BTB のセット割り当てと競合状態がどのように変化するかを分析する．さらに，競合に関与する命令アドレスとコード領域を特定し，コード配置，セット競合，および実行性能の関係を検討する．これらの関係を整理することで，マイクロアーキテクチャ上の挙動を考慮した，新しいコード配置最適化手法を設計するための基礎を得ることを目指す．

この目的に対し，本研究では，命令ハンドラの順序を変更した際のマイクロアーキテクチャの挙動について次の仮説を立て，分析を行う．

1. 命令キャッシュや BTB におけるセット競合の程度が変化する．
2. 競合の多い配置では，競合ミスによる命令供給の停滞によって実行時間が増加する．

命令ハンドラの順序を変更すると，各ハンドラと，その内部に含まれる命令（分岐命令を含む）のアドレスが変化する．これにより，命令キャッシュや BTB におけるセットへの割り当ても変わり，頻繁に使用されるエントリが特定のセットへ集中し得る．この集中によって同一セットで使用されるエントリ数が way 数を超えると，追い出しが増加する．その結果，命令キャッシュミスや分岐先予測の失敗によってフロントエンドからの μop 供給が停滞し，実行時間が増加すると推測する．

4.2 分析の方針

本研究では，コード配置，CPU 内部の挙動，および実行性能の関係を，次の四つの分析によって調べる．

1. 命令ハンドラ配置を変えた実行ファイルを実機

で動かし，配置の変更によって生じる実行時間の差の大きさを確認する（第 5.2 節，第 6.1 節）．
2. 実行時間の異なる配置について，TMA によって性能変動と関係するパイプライン上の領域を絞り込む（第 5.3 節，第 6.2 節）．
3. 命令の実行履歴（トレース）と命令キャッシュおよび BTB モデルを用いて，セットごとの状態を調べ，配置によるアドレス変化がセット割り当てと競合状態を変えるという仮説を検討する（第 5.4 節，第 6.3 節）．
4. マイクロベンチマークを用いて，命令キャッシュおよび分岐先バッファにおけるコードのアドレスによって発生する競合が，実機の性能を低下させることを検証する（第 5.5 節，第 6.4 節）．

5 実験方法

5.1 実験対象と実験環境

本研究では，Ruby 3.3.1 [19] のバイトコードインタプリタである Yet Another Ruby VM（YARV）[20] を分析のために使用する．YARV は computed goto を用いた direct-threaded 方式で命令ハンドラ間を遷移する．インタプリタ自体の挙動を評価するため，実行時コンパイルを行う Yet Another Ruby JIT（YJIT）[4] は無効化した．

実験では，102 個の命令ハンドラをランダムに並べ替え，それぞれの順序ごとに Ruby をビルドした．各ハンドラの処理内容は変更しないため，生成した Ruby 実行ファイルは，主として命令ハンドラのバイナリ上の配置が異なる．また，元の命令ハンドラ順序を持つ Ruby 実行ファイルをベースラインとする．

ベンチマークには，Are We Fast Yet（AWFY）ベンチマーク集 [11] の Bounce，CD，List，Mandelbrot，NBody，Permute，Queens，Richards，Sieve，および Towers を用いた．

実機プロファイリングには，表 1 に示す四つの CPU のうち，三つの x86-64 プロセッサを用いた．測定対象プロセスは特定の CPU および NUMA ノードに固定し，Address Space Layout Randomization（ASLR）を無効化した．また，実行順による偏りを抑えるため，ベンチマークと配置の順序は測定回ごとにランダム化した．

5.2 命令ハンドラ順序による性能変動の確認

命令ハンドラの順序だけで実行性能がどの程度変化するかを調べるため，Intel Xeon w3-2435 と Rockchip RK3588 に搭載された Cortex-A76 で，異なるハンドラ順序を持つ 3,000 個の Ruby の性能を測定した．各 Ruby 実行ファイルを 5 回実行し，その中央値を代表実行時間とした．CPU とベンチマークの組合せごとに，代表実行時間が最小の配置を最速配置，最大の配置を最遅配置とする．配置間の性能差を次式で定義する．

$$\Delta_{\text{layout}} = \frac{T_{\text{slow}} - T_{\text{fast}}}{T_{\text{fast}}} \times 100 \tag{1}$$

ここで，T_{fast} と T_{slow} は，それぞれ最速配置と最遅配置の代表実行時間である．

5.3 プロファイリングによる要因の絞り込み

命令ハンドラ順序による性能変動の要因を絞り込むため，性能の差と関係するパイプライン上の要因をハードウェア性能カウンタによって分析する．まず，トップダウン解析の Level 1 に相当する Retiring，Bad Speculation，Frontend Bound，および Backend Bound の四分類を用いて，配置間の性能差がフロントエンド，投機の問題，またはバックエンドのいずれと関係するかを調べる．次に，実行時間との関係が確認された分類を Level 2 の指標によって細分化する．

5.3.1 測定対象と測定手順

CPU とベンチマークの組合せごとに，代表として 50 個の違うハンドラ順序を持つ Ruby（以下では「配置」）を測定対象として選定した．最速から最遅まで，違う実行性能を持つ代表的な配置を選出するため，Intel では 3,000 配置，AMD では 1,000 配置について事前に実行時間を測定し，最小時間から最大時間の範囲を 50 等分し，それぞれ実行時間が最も近い配置を選んだ．測定前にすべての組合せを一度実行してウォームアップし，その後，各組合せを 5 回ずつ測定した．各配置では，実行時間が 5 回の中央値となる結果（実行時間と各性能指標）を分析に用いた．また，時間経過や実行順序による偏りを抑えるため，

表 1　実機評価環境

CPU	マイクロアーキテクチャ	命令セット	Linux カーネル	perf
Intel Xeon w3-2435	Sapphire Rapids (Golden Cove)	x86-64	7.0.0-28-generic	7.0.12
AMD Ryzen Threadripper PRO 7945WX	Zen 4	x86-64	6.8.0-136-generic	6.8.12
AMD EPYC 9654	Zen 4	x86-64	7.0.0-28-generic	7.0.12
Rockchip RK3588	Cortex-A76 + Cortex-A55	AArch64	6.1.84-8-rk2410	6.1.176

各回でベンチマークと配置の実行順序をランダム化した．

5.3.2　相関分析

相関分析は CPU とベンチマークの組合せごとに行う．実行時間と各性能指標の関係は Pearson の積率相関係数 r で評価し，$|r| \geq 0.5$ かつ r の 95%信頼区間が 0 を含まない場合を相関ありと定義した．相関係数を計算するだけでなく散布図も確認し，少数の外れ値や明らかな非線形関係によって相関が生じていないかを調べた．

5.4　シミュレーションによる競合分析

実機プロファイリングでは，性能差と関係するパイプライン領域を調べられる一方，命令キャッシュや BTB の内部状態と，競合に関与するアドレスを観測することは難しい．そこで，コード配置によるアドレス変化がセット割り当てと競合状態に与える影響を調べるため，構成を変更できるシミュレータを用いる．

5.4.1　シミュレータと実行トレース

シミュレーションには，6th Championship Branch Prediction（CBP2025）で公開された CBP2025 Simulator Framework [3] を用いる．このシミュレータに，命令キャッシュのセット，アドレスごとのアクセス，ミス，追い出しを記録する機能と，容量や way 数を変更できる BTB 機構を追加した．

実行トレースは，QEMU 9.0.1 [16] の aarch64-linux-user モードを用いて取得した．10 種類のベンチマークについて 9 つの配置で実行し，合計 90 トレースを取得した．各トレースはおおむね 10 億命令であり，Ruby の起動，ベンチマークハーネス，ベンチマーク本体，および終了処理を含む．10 億命令のうち，Ruby 自身の起動と終了処理はおおむね 2.7 億命令である．

5.4.2　命令キャッシュと BTB における競合分析

命令キャッシュにおけるセット競合の分析には，8 KiB（128 エントリ，32 セット，4-way，64-byte ライン），Least Recently Used（LRU）置換方式のセットアソシアティブ構成のキャッシュを用いた．

BTB におけるセット競合の分析には，128 エントリ，32 セット，4-way の構成を用いた．CBP2025 シミュレータには BTB 機構が搭載されていないため，LRU 置換方式の単一階層のセットアソシアティブ構成の BTB を実装した．AArch64 命令の 4-byte アラインメントを考慮し，$(pc \gg 2) \bmod N$（N：セット数）によってセットを選ぶようにした．

5.4.3　競合に関与するコード領域の特定

競合に関与するコード領域を調べるため，トレース上の実行時アドレスを Executable and Linkable Format（ELF）内の相対アドレスへ変換し，シンボル情報，DWARF デバッグ情報，および命令ハンドラのアドレス情報と対応付ける．この方法によって，特定した命令アドレスが，Ruby のどの部分（命令ハンドラ，またはそれ以外の部分）に所属するかを分析する．

5.5　セット競合による性能低下の検証

命令キャッシュおよび分岐先バッファ（BTB）におけるセット競合が実機性能に与える影響を検証するため，Rockchip RK3588 に搭載された Cortex-A76 上でマイクロベンチマークを実行する．ベンチマークは，AArch64 の無条件直接分岐命令 `b` のみからなるリング状の制御フローを実行する．比較する配置間では，分岐命令の本数，種類，および論理的な実行順序を同一に保ち，テスト対象となる分岐命令のアドレス

配置を変更することで，L1 命令キャッシュにおけるセットへの集中度を変化させる．

Cortex-A76 の L1 命令キャッシュは，容量 64 KiB，4-way，64-byte ラインの 256 セットで構成される [17] [1]．命令アドレス（PC）に対応するセットは $(PC \gg 6) \bmod 256$ によって決まる．したがって，16 KiB 離れたキャッシュラインは同一セットへ割り当てられる．

図 4 に，5 本の分岐命令を用いたマイクロベンチマークの例を示す．いずれの配置でも，分岐命令は `B0`→`B1`→ `B2`→`B3`→ `B4`→`B0` の順に繰り返し実行される．図 4(a) の集中配置では，分岐命令を 16 KiB 間隔で配置する．このとき，5 本の分岐命令を含むキャッシュラインはすべてセット 0 へ割り当てられる．また，セット当たりのライン数が Cortex-A76 の 4-way を超える．一方，図 4(b) の分散配置では，隣接する分岐命令を 16 KiB + 64 byte 間隔で配置する．これにより，5 本の分岐命令を含むキャッシュラインはセット 0 からセット 4 へ一つずつ分散して割り当てられる．

この二つの配置を比較することで，分岐命令の本数と論理的な実行順序を揃えたまま，L1 命令キャッシュのセット競合が実行性能に与える影響を評価する．

次に，BTB におけるセット競合の検証方法について述べる．Cortex-A76 の BTB 構造については，Technical Reference Manual（TRM）[1] に記載された L1 BTB data location encoding を参考に，L1 BTB が 1024 セット，4-way で構成されると仮定した．PC に対応するセットは $(PC \gg 5) \bmod 1024$ で決まると推測する．推測ではセットの選択に命令アドレスの下位 5 ビットを使用していないため，同じ 32-byte 領域内にある分岐命令は同一のセットに割り当てられると推測した．

AArch64 の命令は 4-byte であるため，一つの 32-byte 領域には最大 8 個の分岐命令を配置できる．これらは 1 つの 64-byte キャッシュライン内に収まるため，命令キャッシュの使用ライン数を増やさずに，同一の BTB セットに対応する分岐命令数を増やせる．実験では，32-byte 領域を四つ用意し，四領域を一つの BTB セットへ集中させる配置と，二つの BTB セットへ二領域ずつ分散した配置を比較する．この二つの配置では，四つの領域を同じ L1 命令キャッシュセットへ割り当てる．使用するライン数は 4 本であり，4-way の範囲内に収まるため，命令キャッシュの条件を変えずに BTB セットの集中度合だけを変更できる．これにより，BTB のセット競合による性能低下を検証できる．

各条件を 9 回測定し，その中央値を代表値とする．性能指標として，IPC，L1 命令キャッシュミス，iTLB ミス，および分岐予測ミスを perf で取得する．

表 2　3,000 配置における最速配置と最遅配置の性能差

CPU	性能差の範囲	中央値
Intel Xeon w3-2435	4.64–16.35%	11.41%
Rockchip RK3588 (Cortex-A76)	3.63–11.05%	5.40%

6　実験結果

6.1　命令ハンドラ順序による性能変動

表 2 に，3,000 配置の最速配置と最遅配置の性能差を示す．Intel Xeon w3-2435 では，最遅配置の実行時間は，最速配置の実行時間と比べて 4.64–16.35%長く，10 種類のベンチマークにおける中央値は 11.41%であった．Cortex-A76 では，最遅配置の実行時間は最速配置と比べて 3.63–11.05%長く，その中央値は 5.40%であった．

Intel Xeon w3-2435 では，最速配置の実行時間はベースラインより 0.52–12.14%短く，実行時間短縮率の中央値は 1.66%であった．Cortex-A76 では，最速配置の実行時間はベースラインより 1.18–8.91%短く，実行時間短縮率の中央値は 2.22%であった．この実験から，命令ハンドラの処理内容を変えなくても，その配置順序によって Ruby の実行性能が変化することを確認した．

6.2　プロファイリングによる要因の絞り込み

CPU とベンチマークの組合せごとに，50 配置について実行時間とトップダウン解析指標の相関を分析した．表 3 に，TMA の Level 1 の四分類について，それぞれの割合と実行時間が正または負の相関を示し

図 4　L1 命令キャッシュのセット競合を検証するマイクロベンチマークの例

たベンチマーク数を示す．

Retiring の割合は，すべての CPU とベンチマークで実行時間と負の相関を示した．この傾向は，命令ハンドラの配置が変わってもリタイアする命令数は変わらないため，配置による実行時間の変動に伴った Retiring の割合の変動によるものである．

Frontend Bound との正の相関は，Intel Xeon w3-2435 では 10 個中 4 個，AMD Ryzen Threadripper PRO 7945WX では 9 個，AMD EPYC 9654 では 10 個のベンチマークで確認された．Bad Speculation との正の相関は，それぞれ 2 個，8 個，9 個のベンチマークで確認された．一方，Backend Bound では，Intel Xeon w3-2435 で正と負がそれぞれ 2 個，AMD Ryzen Threadripper PRO 7945WX で 1 個と 2 個，AMD EPYC 9654 で 1 個と 8 個であり，CPU およびベンチマークによって相関の向きが異なった．

さらに，三つの CPU で共通して取得した Level 2 指標についても，実行時間との相関を分析した．Frontend Bound はレイテンシ（Frontend Latency）と帯域幅（Frontend Bandwidth），Bad Speculation は分岐予測ミス（Branch Mispredicts）とその他のパイプラインの再開（Pipeline Restarts），Backend Bound はメモリ（Memory Bound）とコア（Core Bound）に分類した．表 4 に，各指標と実行時間が正または負の相関を示したベンチマーク数を示す．各欄は「正の相関数／負の相関数」を表す．

Frontend Latency との正の相関は，Intel Xeon w3-2435 では 10 個中 3 個，二つの AMD プロセッサでは 10 個すべてで確認された．Branch Mispredicts との負の相関は，AMD Ryzen Threadripper PRO 7945WX では 7 個，AMD EPYC 9654 では 8 個で確認された．Pipeline Restarts との正の相関は，Intel Xeon w3-2435 では 3 個，二つの AMD プロセッサではそれぞれ 9 個で確認された．一方，Frontend Bandwidth，Memory Bound，および Core Bound では，CPU やベンチマークによって相関の向きが異なった．

したがって，実行時間が長い配置では，主にフロントエンドからの命令供給の遅延や，投機の問題後のパイプライン再開が増える傾向を確認できる．

6.3　シミュレーションによる競合分析

6.3.1　命令キャッシュ競合

シミュレータ上で，同じ命令キャッシュ容量とベンチマークでも，配置によってミス率に差が生じることを確認した．8 KiB（128 エントリ，32 セット，4-way，64-byte ライン）の命令キャッシュ構成では，9 配置の最大ミス率と最小ミス率の比はベンチマークによって 1.05–2.39 倍であった．ミス率の比は Sieve ベンチマークで最も大きかった．

Sieve の 9 配置を比較した結果を図 5 に示す．図では，配置を命令キャッシュミス率の昇順に並べている．ミス率は 0.962–2.289%，上位 4 セットが全体のミス数に占める割合は 19.7–61.4%であった．

Ruby 1403 と Ruby 2551 では，命令キャッシュへのアクセス数と使用したキャッシュライン数がほぼ同

表 3　実行時間と相関を示したベンチマーク数（正／負）

CPU	Retiring	Frontend Bound	Bad Speculation	Backend Bound
Intel Xeon w3-2435	0/10	4/0	2/0	2/2
AMD Ryzen Threadripper PRO 7945WX	0/10	9/0	8/0	1/2
AMD EPYC 9654	0/10	10/0	9/0	1/8

表 4　実行時間と相関を示した Level 2 指標のベンチマーク数（正／負）

	Frontend Bound		Bad Speculation		Backend Bound	
CPU	Frontend Latency	Frontend Bandwidth	Branch Mispredicts	Pipeline Restarts	Memory Bound	Core Bound
Intel Xeon w3-2435	3/0	2/0	0/0	3/0	0/2	3/1
AMD Ryzen Threadripper PRO 7945WX	10/0	3/1	0/7	9/0	1/3	2/1
AMD EPYC 9654	10/0	7/1	0/8	9/0	1/8	3/2

じである．それにもかかわらず，Ruby 1403 のミス率は Ruby 2551 の約 2.38 倍であり，上位 4 セットが占めるミスの割合も 20.3%から 61.4%へ増加した．したがって，よくアクセスされる（ホットな）キャッシュラインのセット割り当てが変わり，競合ミスに差が生じることを確認した．

6.3.2　BTB 競合

Sieve の 9 配置の結果を図 6 に示す．図では，配置を BTB ミス率の昇順に並べている．BTB ミス率は 13.898–19.873%，上位 4 セットが占めるミスの割合は 17.60–41.33%であった．

Ruby 2551 は BTB ミス率とミスの集中度がともに最も低く，Ruby 14 はともに最も高かった．Ruby 14 の BTB ミス率は Ruby 2551 の約 1.43 倍である．また，Ruby 1498，Ruby 1169 および Ruby 14 では，BTB ミスの約 41%が上位 4 セットに集中し，ほかの配置よりも BTB ミス率が高かった．したがって，分岐命令アドレスの変化によって BTB のセットの割り当てが変わり，特定のセットへのミスの集中度と BTB ミス率に差が生じることを確認した．

6.3.3　競合に関与するコード領域

命令キャッシュでは，8 KiB（128 エントリ，32 セット，4-way，64-byte ライン）の構成において，Ruby 1403 のセット 0 に，5 本のホットなコードが集中していた．そのうち 4 本は，`getglobal`，`branchif`，`opt_aset` などを含む `vm_exec_core` 領域内のラインであり，残る 1 本は配列操作を行う `rb_ary_store` のラインであった．4-way のセットに 5 本のラインが割り当てられたため，これらは循環的に追い出されていた．この 5 本は全命令キャッシュミスの 23.03%，上位 5 個の追い出し関係は全追い出しの 22.82%を占めた．一方，Ruby 2551 では，上位 5 個の追い出し関係が占める割合は 2.22%であった．

BTB モデルでも同様の競合が確認された．128 エントリ，32 セット，4-way 構成において，Ruby 14 では一つのセットに少なくとも 5 個のホットな分岐命令が集中していた．その内訳は，`vm_exec_core` 領域内の `expandarray` に関係する分岐が 1 個，VM 呼出し処理などに含まれる分岐が 4 個であった．このセットにおけるミスは全 BTB ミスの 30.18%を占め，上位 5 個の追い出し関係は全追い出しの 27.25%を占めた．

Sieve の 9 配置では，命令キャッシュと BTB のいずれにおいても，`vm_exec_core` 領域内のコードだけでなく，それ以外のコードとの競合が確認された．

6.4　セット競合による性能低下の検証

命令キャッシュの実験では，同一セットで使用するライン数が 4 本以下の場合，集中配置と分散配置の性能は同等であった．一方，4-way を超える 5 本を同一セットへ配置すると，IPC は 1.000 から 0.201 へ 79.9%低下した．このとき L1 命令キャッシュの MPKI も 0.039 から 987.455 へ増加したことから，性能低下

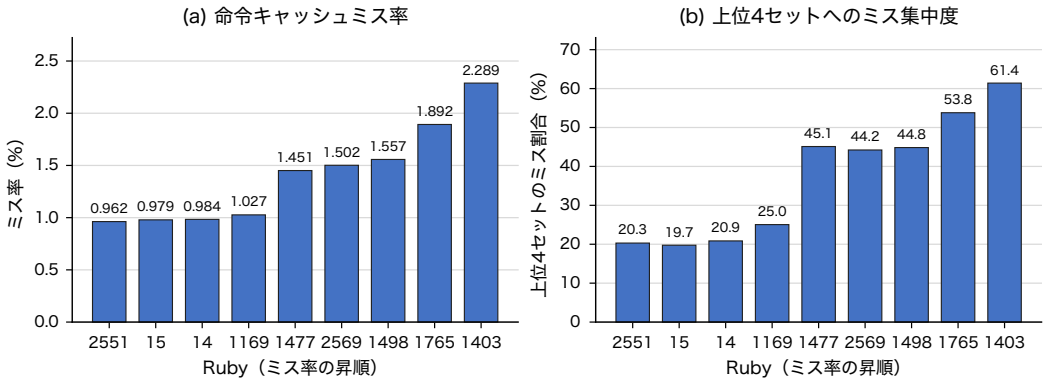

図 5　Sieve の 9 配置における命令キャッシュ競合

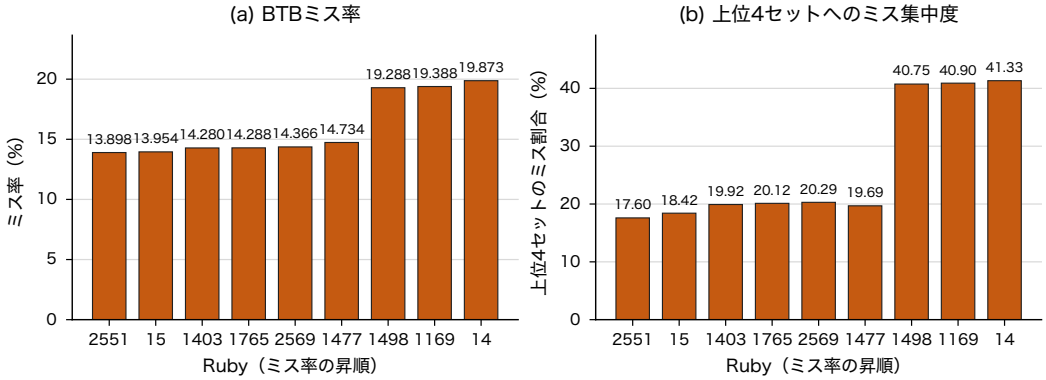

図 6　Sieve の 9 配置における BTB 競合

は L1 命令キャッシュのセット競合といえる．

BTB の実験では，一つのセットで使用されるテスト対象分岐数が 8 個となる分散配置と，16 個となる集中配置を比較したところ，分岐数が 8 個の場合の IPC は 0.494 だったが，16 個になると 0.285 へ 42.4%低下した．一方，L1 命令キャッシュの MPKI はそれぞれ 0.079 と 0.143 であり，iTLB ミスもほぼ同程度であったため，この性能低下は，分岐命令アドレスに依存する分岐先予測器の構造の競合の可能性が高い．

7　性能変動要因の考察

7.1　コード配置による性能変動の解釈

第 6.1 節で示したように，命令ハンドラの処理内容が同じでも，配置によって実行時間が変化した．また，第 6.2 節および表 3 に示したように，実機では，実行時間が Frontend Bound および Bad Speculation と関係する CPU とベンチマークが多かった．この結果は，配置による性能変動に，フロントエンドからの命令供給と投機実行が関与している可能性を示唆する．

一方，シミュレーションでは，命令キャッシュについて，配置によってホットな命令のセット割り当てとミスの集中度が変化した（第 6.3.1 節，図 5）．BTB

についても，分岐命令アドレスの変化によって，ミスのセット集中度とミス率が変化した（第 6.3.2 節，図 6）．さらに，競合には命令ハンドラを含む領域だけでなく，VM 補助処理などの周辺コードも関与した（第 6.3.3 節）．また，マイクロベンチマークでは，命令キャッシュのセット競合，BTB のセットの競合に伴う性能低下を観測した（第 6.4 節）．

これらの結果を合わせると，コード配置から性能変動へ至る経路の候補を，次のように整理できる．

ハンドラ順序の変更
↓
命令・分岐アドレスの変更
↓
セット割り当てと競合状態の変更
↓
ミスまたは命令供給遅延の増加
↓
実行時間の増加

7.2 コード配置最適化への示唆

本研究の結果は，命令ハンドラの実行頻度や遷移頻度だけでなく，配置後の命令アドレスを検討する必要性を示唆する．頻繁に実行されるコードを近接させても，それらが少数のセットへ集中すれば，小容量の命令キャッシュや BTB で競合が増える可能性がある．また，第 6.3.3 節に示したように，競合には VM 補助処理などのハンドラ外コードも関与する可能性がある．したがって，命令ハンドラだけを独立に考慮するのではなく，同時に実行される周辺コードを含めてセットへの割り当てを評価することが重要である．

この考え方は，候補配置を事前に絞り込む指標へ応用できる可能性がある．例えば，動的なアクセス頻度と配置後のアドレスから各セットへ割り当てられるホットエントリ数を推定し，way 数を超える集中が生じる配置の優先度を下げる方法が考えられる．

7.3 本研究の制約と今後の課題

シミュレーションによるセット単位の詳細分析は，主に Sieve の 9 配置と，競合を観測しやすい小容量の命令キャッシュおよび BTB モデルを対象とした（第 5.4 節，第 6.3 節）．したがって，観測した競合が他のベンチマークや実機でも同じ形で生じるとは限らない．特に，実機 BTB の容量，階層，way 数，およびインデックス関数（セットを決める方法）が，今回実装した BTB モデルと同一であるとは限らない．

第 5.5 節では，我々が作成したマイクロベンチマークで競合と性能低下を確認しており，これは Ruby の配置間性能差が同じ競合によって生じたことを直接示すものではない．実機プロファイリングで得た相関も因果関係を直接示さない．以上から，本研究は，コード配置によってセット競合の状態が変化し得ることを示したが，その競合が実機上の性能差の主要因であるとは結論しない．

今後は，競合に関与すると推定したコード間に No Operation 命令（NOP）などのパディング（Padding）を挿入し，対象コードのアドレスだけを段階的に移動させる実験を行う．また，予測したセット割り当て，ミス率，および実機上の実行時間が同時に変化するかを調べることで，コード配置から性能変動に至る因果関係を検証する．

8 関連研究

実行プロファイルに基づくコード配置最適化は古くから研究されている．Pettis と Hansen は，頻繁に制御が遷移する基本ブロックを近接して配置し，命令メモリ階層のオーバーヘッドを削減する手法を提案した [14]．Jiménez は，エッジプロファイルを用いて条件分岐命令のアドレス下位ビットをパディングによって調整し，Pattern History Table（PHT）における競合を抑制する手法を提案した [7]．近年では，BOLT は実行時プロファイルを用いてリンク後の基本ブロックおよび関数を再配置し，iTLB ミス，命令キャッシュミス，および分岐ミスの削減を実現している [13]．

インタプリタでは，Ertl と Gregg は，命令ハンドラの複製やスーパー命令化（superinstructions）によって，threaded-code インタプリタにおける BTB の間接分岐予測精度を改善する手法を提案している [5]．Brunthaler は，頻繁に連続して実行される命令ハンドラを近接配置し，命令キャッシュ局所性を改

善する手法を提案した[2]．McCandless と Gregg は，Python および Lua の命令ハンドラ順序をフィードバック駆動で探索し，配置による性能差が主として間接分岐予測と関係することを示した[12]．Huang らは，eJSVM の命令ハンドラ順序を GA によって最適化し，平均 7.7%の性能向上を達成した[6]．

これらに対し，本研究は Ruby インタプリタを対象として，命令のアドレス配置による命令キャッシュおよび BTB のセット割り当てと追い出し関係を分析し，命令ハンドラ内外のコード間で生じるセット競合を確認している点に特徴がある．

9 おわりに

本研究では，Ruby 3.3.1 のバイトコードインタプリタを対象として，命令ハンドラのコード配置による性能変動を，実機プロファイリングとトレース駆動シミュレーションによって分析した．

実機評価では，配置による実行時間の増加に伴い，Frontend Bound および Bad Speculation の割合が増加する CPU とベンチマークの組合せが多く，実行時間の増加はフロントエンドや投機の問題と関係する傾向が示された．シミュレーションでは，分岐命令を含む命令のアドレスが配置によって変化し，命令キャッシュと BTB のモデルにおけるセット割り当ておよび追い出し関係が変化した．また，競合には命令ハンドラだけでなく，VM の補助処理などの周辺コードも関与していた．さらに，マイクロベンチマークでは，命令キャッシュおよび BTB のセットの競合が引き起こす性能低下を観測した．以上の結果は，命令ハンドラの実行頻度や遷移頻度だけでなく，配置後の命令アドレスと，周辺コードを含むセットへの割り当てを考慮する必要性を示している．

ただし，本研究のシミュレーションは小容量のキャッシュモデルと一部の命令ハンドラ順序のみを対象としており，観測した競合が実機上の Ruby の配置間性能差の主要因であることを直接示すものではない．今後は，NOP などのパディングによって対象コードのアドレスを制御する実験を行い，予測したセット割り当て，ミス率および実行時間が同時に変化するかを調べることで，コード配置と性能変動の因果関係を検証する．

参 考 文 献

[1] Arm Limited: Arm Cortex-A76 Core Technical Reference Manual, `https://developer.arm.com/documentation/100798/0401/`, 2023. Accessed 2026-08-09.

[2] Brunthaler, S.: Interpreter instruction scheduling, *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, Berlin, Heidelberg, Springer-Verlag, 2011, pp. 164–178.

[3] CBP2025 Organizing Committee: CBP2025 Simulator Framework, Software and official documentation, 2025. Accessed 2026-08-09.

[4] Chevalier-Boisvert, M., Kokubun, T., Gibbs, N., Wu, S. X. A., Patterson, A., and Issroff, J.: Evaluating YJIT ' s Performance in a Production Context: A Pragmatic Approach, *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, New York, NY, USA, Association for Computing Machinery, 2023, pp. 20–33.

[5] Ertl, M. A. and Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, New York, NY, USA, Association for Computing Machinery, 2003, pp. 278–288.

[6] Huang, W., Marr, S., and Ugawa, T.: Optimizing the Order of Bytecode Handlers in Interpreters using a Genetic Algorithm, *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, New York, NY, USA, Association for Computing Machinery, 2023, pp. 1384–1393.

[7] Jiménez, D. A.: Code placement for improving dynamic branch prediction accuracy, *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, New York, NY, USA, Association for Computing Machinery, 2005, pp. 107–116.

[8] Kataoka, T., Ugawa, T., and Iwasaki, H.: A framework for constructing javascript virtual machines with customized datatype representations, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, New York, NY, USA, Association for Computing Machinery, 2018, pp. 1238–1247.

[9] Linux Kernel Developers: Using TopDown Metrics, `https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/topdown.txt`. Accessed 2026-08-09.

[10] Linux perf Developers: perf(1)—Performance Analysis Tools for Linux, `https://man7.org/linux/man-pages/man1/perf.1.html`, 2024. Accessed

2026-08-09.

[11] Marr, S., Daloze, B., and Mössenböck, H.: Cross-language compiler benchmarking: are we fast yet?, *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, New York, NY, USA, Association for Computing Machinery, 2016, pp. 120–131.

[12] McCandless, J. and Gregg, D.: Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures: or: how I learned to stop worrying and love hill climbing, *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, New York, NY, USA, Association for Computing Machinery, 2011, pp. 161–170.

[13] Panchenko, M., Auler, R., Nell, B., and Ottoni, G.: BOLT: A Practical Binary Optimizer for Data Centers and Beyond, *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 2–14.

[14] Pettis, K. and Hansen, R. C.: Profile guided code positioning, *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, New York, NY, USA, Association for Computing Machinery, 1990, pp. 16–27.

[15] Python Software Foundation: CPython Source Code, `https://github.com/python/cpython`. Accessed 2026-08-09.

[16] QEMU Project Developers: QEMU Version 9.0.1, `https://gitlab.com/qemu-project/qemu/-/tags/v9.0.1`, 2024. Accessed 2026-08-09.

[17] Rockchip Electronics Co., Ltd.: RK3588 Brief Datasheet, `https://www.rock-chips.com/uploads/pdf/2022.8.26/191/Rockchip%20RK3588%20Brief%20Datasheet%20V1.0_20250828.pdf`, 2022. Accessed 2026-08-09.

[18] Ruby Core Team: Ruby Source Code, `https://github.com/ruby/ruby`. Accessed 2026-08-09.

[19] Ruby Core Team: Ruby Release 3.3.1, `https://github.com/ruby/ruby/releases/tag/v3_3_1`, 2024. Accessed 2026-08-09.

[20] Sasada, K.: YARV: yet another RubyVM: innovating the ruby interpreter, *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, New York, NY, USA, Association for Computing Machinery, 2005, pp. 158–159.

[21] Yasin, A.: A Top-Down method for performance analysis and counters architecture, *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.