動的プログラミング言語向け共通 JIT コンパイラ基盤の実装と評価

東間 日向　福田 浩章

動的プログラミング言語の高速化手法として JIT コンパイルがある．しかし成熟した処理系は規模が大きく，既存のインタプリタへの後付けは容易でない．インタプリタの記述から JIT コンパイラを自動生成するメタトレーシング JIT はこの負担を軽減するが，専用の言語でインタプリタを一から実装する必要があり，既存の処理系には適用できない．そこで本研究では，既存のインタプリタを再実装することなく JIT コンパイラを備えた処理系を得られるシステムを提案する．提案システムは既存のインタプリタを WebAssembly へコンパイルし，実行された WebAssembly 命令列をトレースして JIT コンパイルする．利用にあたってはインタプリタのソースコード上でループを構成する後方ジャンプ箇所に注釈を付与するだけでよく，WebAssembly へコンパイル可能な言語で実装された処理系であれば適用できる．提案システムを Lua，JavaScript，Ruby，Python の処理系である PUC-Rio Lua，QuickJS，CRuby，CPython に適用し，9 種類のプログラムを 4 言語で記述した 36 ケースのベンチマークのうち 19 ケースで，同一のソースコードから機械語へコンパイルした処理系を上回る性能を確認した．

1　はじめに

動的プログラミング言語は，変数や関数の引数に型を宣言せず，型の決定や名前の解決を実行時に行う言語である．Python，Ruby，JavaScript などがこれに該当し，Web アプリケーション開発，データ分析，アプリケーションへの組み込みなど，幅広い分野で用いられている．これらの言語では型宣言が不要であるため記述量が少なく，また型の整合性を保つための修正も生じにくい．このため，仕様が確定していない段階から試行錯誤を伴う開発を進めやすい．一方で，型が実行時まで確定しないことは実行速度の低下を招く．静的プログラミング言語では，コンパイル時に型が確定するため，静的解析の結果に基づいて効率的な機械語を生成できる．これに対し動的プログラミング言語では，演算のたびにオペランドの型を検査し，呼び出すべきメソッドを探索する必要があるため，同様の手法をそのまま適用できない．

この問題に対する代表的な解決策が JIT（Just-In-Time）コンパイルである．JIT コンパイルは，プログラムの実行中に観測した型などの情報を用いて機械語を生成する手法であり，静的解析だけでは確定しない情報を実行時の観測で補うことで，インタプリタによる実行を上回る速度を得る．しかし，JIT コンパイラの実装には多大な労力を要し，その負担は言語ごとに繰り返し生じる．既存の言語処理系が JIT コンパイルを利用できるようにする方針は，大きく 2 つに分けられる．

1 つは，既存の処理系に JIT コンパイラを追加する方針である．CRuby の YJIT [9]，PHP の OPcache JIT [23]，CPython の実験的な JIT コンパイラ [7] がこれにあたる．これが利用できるのは対象とする言語の処理系に JIT コンパイラが実装されている場合に限られる．例えば，Lua の主要な実装である PUC-Rio Lua は現在も JIT コンパイラを備えていない．もう

---

* Implementation and Evaluation of a Language-Independent JIT Compiler Framework for Dynamic Languages

This is an unrefereed paper. Copyrights belong to the Author(s).

Hinata Toma, 芝浦工業大学大学院理工学研究科電気電子情報工学専攻, Dept. of Electrical Engineering and Computer Science, Shibaura Institute of Technology.

Hiroaki Fukuda, 芝浦工業大学工学部情報工学科, Dept. of Information Engineering, Faculty of Engineering, Shibaura Institute of Technology.

1 つは，同じ言語を対象とする処理系を新たに実装する方針である．Lua の LuaJIT [16]，Python の PyPy [20] がこれにあたる．これらはリファレンス実装を上回る実行速度を示す一方で，言語仕様の更新に追随し続ける必要がある．この負担は大きく，実際に，対応する言語仕様がリファレンス実装より遅れている例が見られる．LuaJIT が対応するのは 2006 年に公開された Lua 5.1 であり，Lua の言語仕様はその後 5.5 まで更新されている [14]．PyPy が対応するのは Python 3.11 であり [17]，CPython が実装する 3.14 [19] とは 3 世代の差がある．JIT コンパイラを実装する負担を軽減する試みとして，複数の言語に適用できる処理系構築基盤がある．PyPy の基盤である RPython [4] [20] や，GraalVM の Truffle [28] は，インタプリタの記述から JIT コンパイラを導出することで，言語ごとに JIT コンパイラを設計する作業を不要にする．しかし，これらを用いる場合も言語処理系そのものは基盤に合わせて記述し直す必要があり，仕様追随の負担は残る．

ここまでに述べた負担は，いずれも処理系を新たに実装することに起因する．そこで本研究では，既存の処理系の実装をそのまま利用して JIT コンパイルを可能にすることで，その導入コストを大幅に削減することを目標とする．ただし，処理系が実現している言語仕様は変更しないことを条件とする．

本研究で提案するシステムは，既存のインタプリタのソースコードに変更を加えることで，そのインタプリタを JIT コンパイラを備えた処理系として動作させる．処理系の実装者が行う変更は，ループを構成する後方ジャンプの箇所に，プログラムカウンタを引数として渡す関数呼び出しを挿入することのみである．

提案システムは，このインタプリタを WebAssembly へコンパイルしたうえで実行し，実行された WebAssembly の命令列をトレースして機械語へコンパイルする．JIT コンパイラが対象とするのは WebAssembly の命令のみであるため，言語ごとに実装する必要がなく，複数の言語の処理系で同一の実装を利用できる．本手法を適用できるのは，バイトコード VM 方式であり，かつ WebAssembly へコンパイル可能な言語で実装された処理系である．

本研究では，Lua，JavaScript，Ruby，Python の処理系である PUC-Rio Lua，QuickJS，CRuby，CPython に本手法を適用した．必要となった変更はそれぞれ 13 行，27 行，71 行，9 行であった．評価の結果，PUC-Rio Lua と QuickJS において特に高い性能が得られ，同じ処理系を機械語へコンパイルして実行したものと比較して最大で約 6 倍の実行速度を示した．一方，CRuby と CPython では性能が向上する場合が限られた．この差が生じた要因については 6 章で述べる．

以降，2 章でトレーシング JIT について，3 章で WebAssembly について，本研究の前提となる知識を述べる．4 章では本手法の基本的な考え方とシステムの構成を示し，5 章ではその実装のうち特に留意した点について述べる．6 章では 4 つの言語処理系への適用とその評価結果を示す．7 章で関連研究との関係を述べ，8 章でまとめと今後の課題を述べる．

2 トレーシング JIT

本論文で対象とするのは，バイトコード VM 方式の処理系である．この方式では，ソースコードを内部形式の命令列であるバイトコードへ変換したうえで，仮想機械がこれを解釈して実行する．仮想機械は次に実行する命令の位置をプログラムカウンタとして保持し，その位置の命令を取り出し，命令に応じた処理へ分岐したのち，プログラムカウンタをインクリメントすることで次の命令へ進めることを繰り返す．以降このサイクルをインタプリタループと呼ぶ．本研究で適用対象とした PUC-Rio Lua，QuickJS，CRuby，CPython はいずれもこの方式を採る．抽象構文木を直接辿って実行する方式の処理系については，以降では扱わない．

インタプリタによる実行が，同じ処理を機械語として直接実行する場合と比べて遅くなる要因は 2 つある．1 つは，動的に決まる要素を実行のたびに解決することである．値の型が実行時に定まるため，加算のような演算であっても，両辺の型を判別したうえで処理を選ぶ必要がある．メソッドや属性の参照も，名前から対象を実行時に探索する．もう 1 つは，インタプリタ固有の処理の存在である．例えば 2 つの値を

```
1 L: load  i
2    load  n
3    jump_if_ge  E
4    load  s
5    load  i
6    add
7    store s
8    load  i
9    load  1
10   add
11   store i
12   jump  L
13 E: load  s
14   return
```

図 1　ループを含むバイトコード列

```
1 load  i
2 load  n
3 jump_if_ge E
4 load  s
5 load  i
6 add
7 store s
8 load  i
9 load  1
10 add
11 store i
12 jump  L
```

図 2　図 1 から記録されたトレース

加算する場合，CPU での直接実行ではオペランドをレジスタに置いて加算命令を 1 つ実行すれば完了する．これに対しインタプリタでは，加算命令の読み出しと分岐，オペランドの読み出し，両辺の型の判別，型に応じた処理の選択と実行，結果の書き戻し，プログラムカウンタの更新が必要となる．

上述した処理は，JIT コンパイルを行うことで削減できる．まず，ある演算に与えられた値の型や，ある呼び出しで実際に選ばれたメソッドを，実行しながら観測して記録する．次に，繰り返し同じ結果が観測された場合，以降も同じであると仮定し，その仮定のもとで不要となる命令を取り除く．型が定まっていれば演算ごとの型の判別が，呼び出し先が定まっていれば名前からの探索が，それぞれ不要になる．こうして得られた命令列を機械語へコンパイルする．この時点で実行する命令の並びは確定しているため，インタプリタループによる命令の読み出しと分岐も不要となる．仮定はプログラムの意味として保証されるものではないため，機械語には仮定が成立するかを実行時に確かめる検査を挿入する．この検査をガードと呼ぶ．ガードが成立しない場合は，機械語の実行を中止してインタプリタでの実行へ戻す．これを脱最適化と呼ぶ．

JIT コンパイルには，機械語を生成する範囲の取り方にいくつかの方式があり，その 1 つがトレーシング JIT である．トレーシング JIT は，一定回数以上実行されたループ（以下，ホットループ）を対象として，その実行中に通過した命令列を記録し，これを単位として機械語を生成する．この記録をトレースと呼ぶ．条件分岐に達した場合は，実行された側の命令だけが記録され，実行されなかった側は記録されない．したがってトレースは分岐を持たない直線的な命令列になる．トレーシング JIT は，条件分岐が常に同じ方向に分岐するなどの仮定を利用し，この仮定が成り立たない場合には脱最適化を行う．

図 1 は，変数 i が n に達するまで s に i を加算するループのバイトコード列である．以下では説明のため単純化した命令名を用いる．3 行目の `jump_if_ge` は，i が n 以上である場合に E へ分岐する．この条件が成立せずループが繰り返された場合，図 2 のトレースが記録される．記録された経路は E へ分岐しなかった側であるため，i が n より小さいことを確かめるガードが `jump_if_ge` の部分に挿入され，トレースが機械語へコンパイルされる．以降コンパイルされた機械語が実行され，ガードが成立しない場合はインタプリタに戻り E へ分岐する．

トレースが直線的であることにより，値の伝播や定数の畳み込みといった最適化を，分岐の合流を考慮せずに適用できる．トレースの記録から機械語の実行に至る具体的な手順については，4 章で提案システムの構成とあわせて述べる．

トレースを単位とする動的最適化は Dynamo [1] で示され，これをバイトコード VM 方式に適用したものとして HotpathVM [12]，TraceMonkey [11] がある．動的プログラミング言語の処理系では LuaJIT [16]，PyPy [4] がこの方式を採用している．

```
1 (module
2   (import "env" "print" (func $print (param i32)))
3   (func $main
4     i32.const 42
5     call $print))
```

図 3　テキスト形式で記述したモジュールの例

3　WebAssembly

複数の言語に対して同じ最適化を適用するには，対象の処理系をいったん共通の中間言語へコンパイルし，その中間言語に対する JIT コンパイラで最適化するという構成を取ることができる [2] [22] [29]．この構成では，対象の言語が増えても JIT コンパイラ側の実装は変わらず，中間言語へコンパイルできる言語で記述された処理系であれば適用できる．

本研究ではこの中間言語として WebAssembly [13] [25] を用いる．WebAssembly は，サンドボックス内のコードを機械語に近い速度で実行することを目的として策定された仮想命令セットであり，バイナリ形式とテキスト形式の 2 つの表現を持つ．特定の言語に依存しない命令形式として設計されており，C 言語をはじめとする複数の言語のコンパイル先として用いられている．本研究が対象とする処理系はいずれも C 言語で記述されているため，再実装を伴わずに WebAssembly へ変換できる．

WebAssembly のプログラムはモジュールを単位とする．モジュールは関数，グローバル変数，線形メモリなどの宣言からなり，命令列は関数の本体として現れる．テキスト形式で記述したモジュールの例を図 3 に示す．

WebAssembly はスタックマシンであり，命令のオペランドはスタックから取り出され，結果はスタックへ積まれる．例の `i32.const` は値をスタックへ積み，`call` は積まれた値を取り出して呼び出す関数の引数とする．

値を保持する領域は 3 種類ある．ローカル変数は関数の呼び出しごとに確保され，引数もこれに含まれる．`local.get` と `local.set` で読み書きする．グローバル変数はモジュールごとに実体を持ち，関数の呼び出しをまたいで値を保持する．`global.get` と `global.set` で読み書きする．線形メモリはモジュールが持つ連続したバイト列であり，`i32.load` および `i32.store` が，スタックから与えられた整数をアドレスとして読み書きする．C 言語をコンパイルした場合，ポインタが指す領域はこの線形メモリ上に置かれる．

モジュールは，外部との接点としてインポート関数を宣言できる．インポート関数は，モジュール内に定義を持たない関数であり，その実体はモジュールのインスタンス化時に WebAssembly の実行環境が与える．実行環境はこれを任意の処理として実装できるため，モジュール側では通常の関数呼び出しとして記述しながら，その呼び出しを実行環境側で定義できる．例の `$print` がインポート関数であり，モジュール名 env と関数名 print の組で指定される．

4　提案手法

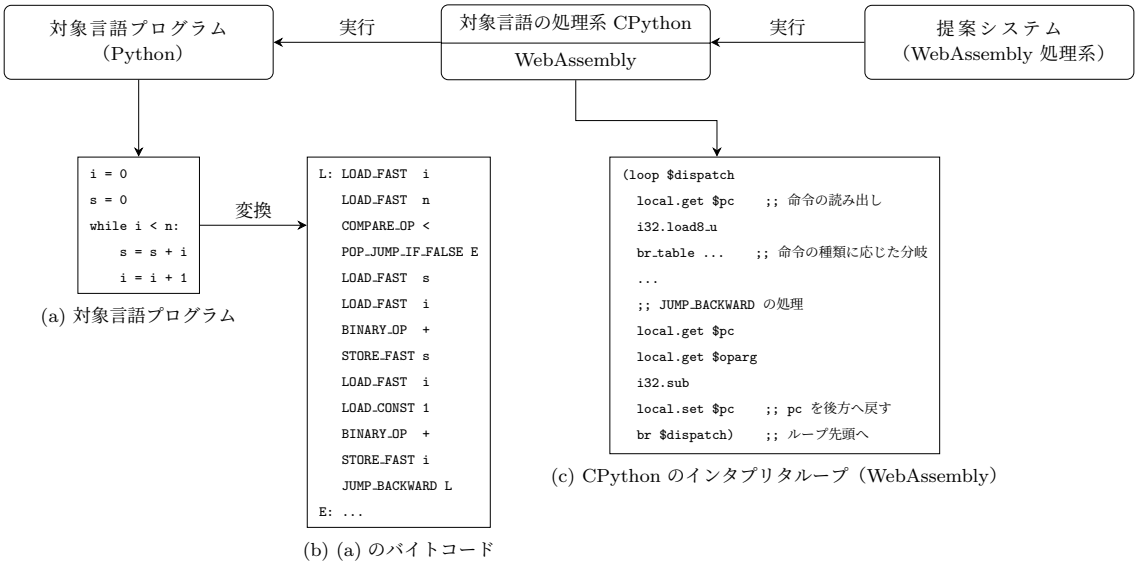
4.1　用語と前提

本章以降では，インタプリタが実行する言語を対象言語，対象言語で記述されたプログラムを対象言語プログラム，インタプリタのソースコードが記述されている言語を実装言語と呼ぶ．例えば CPython に本手法を適用した場合，対象言語は Python，対象言語プログラムは利用者が記述した Python のソースコード，実装言語は CPython のインタプリタの記述に用いられている C 言語である．

本手法では，インタプリタのソースコードを WebAssembly へコンパイルする．実行時には，WebAssembly の実行環境がこのモジュールを実行し，モジュールとして動作するインタプリタが対象言語プログラムを実行する．

CPython に本手法を適用した場合の構造を図 4 に示す．C 言語で記述された CPython のソースコードを WebAssembly へコンパイルし，提案システムがこれを実行する．CPython は対象言語プログラムである Python のソースコード (a) をバイトコード (b) へ変換し，(c) に示すインタプリタループでこのバイトコードを実行する．

このように実行が 2 段の構造をとることに対応し



(a) 対象言語プログラム

(b) (a) のバイトコード

(c) CPython のインタプリタループ（WebAssembly）

図 4　CPython に本手法を適用した場合の構造

て，ループも 2 種類に区別される．

インタプリタループ　2 章で述べたインタプリタループ．CPython では，バイトコード命令の種類ごとの処理を含む C 言語のループがこれにあたり，WebAssembly へコンパイルすると図 4(c) のようなループになる．

対象言語プログラムのループ　対象言語プログラムに記述されたループ．図 4(a) のような Python の while 文や for 文がこれにあたる．

4.2　中間言語（WebAssembly）に対するトレーシング JIT の課題

本研究が最適化の対象とするのは対象言語プログラムのループである．WebAssembly のインタプリタループに対してトレーシング JIT をそのまま適用しても，対象言語プログラムのループは最適化されない．本節では，CPython を例にその理由を述べる．

WebAssembly の命令列の上でループとして現れるのは，図 4(c) のインタプリタループだけである．トレーシング JIT は命令列上の後方ジャンプによってループを検出するため，検出されるのは (c) 末尾の，`loop` の先頭へ制御を戻す `br $dispatch` であり，記録されるトレースはバイトコード命令 1 つ分の処理に留まる．

トレーシング JIT は，記録時に通過した経路を以降の反復でも通ることを仮定する．(c) で記録される経路には `br_table` による命令の種類に応じた分岐が含まれるため，この仮定は，以降も同じ種類のバイトコード命令が読み出されることを要求する．図 4(b) で `LOAD_FAST i` を読み出した時点でトレースを記録した場合，次に読み出される `LOAD_FAST n` は同じ種類であるが，その次の `COMPARE_OP` は種類が異なるためガードが不成立となる．対象言語プログラムは異なる種類の命令を並べたものであるから，この仮定は数反復のうちに成立しなくなる．すなわち，命令の種類が変わるたびに脱最適化が起こり，生成した機械語はほとんど実行されない．

この脱最適化を回避するには，(c) のインタプリタループではなく，(a) の Python のループをトレースの単位として識別する必要がある．この単位で記録すれば，ループ本体に対応する (b) のバイトコード命令列全体の処理が 1 本のトレースに収まる．この命令列は反復の間変化しないため，命令の種類に関するガードは以降の反復でも成立し，同じトレースを繰り返し利用できる．

この識別には，対象言語プログラムの後方ジャンプ

の位置を知る必要がある．ループの反復ではプログラムカウンタが同じ位置に戻り，これを生じさせるのは後方ジャンプだけだからである．しかし，対象言語プログラムのプログラムカウンタは (c) の`$pc` のようにインタプリタが管理する変数にすぎず，その更新のうちどれが後方ジャンプにあたるかを，WebAssembly の命令列から判別することはできない．一方，処理系の実装者にとっては，どの箇所がプログラムカウンタを後方へ戻すかは既知である．CPython であれば `JUMP_BACKWARD` 命令の実装がこれにあたる．そこで提案システムでは，処理系の実装者がインタプリタのソースコード上でこの箇所に注釈を付与することで，対象言語プログラムの後方ジャンプを観測できるようにする．

4.3 システムの概要

提案システムは対象言語の処理系をコンパイルしたモジュールを入力とし，これを実行しながら，対象言語プログラムのループに対応する区間を機械語へコンパイルする．図 5 に，提案システムの利用手順と構成要素を示す．以下，この手順に沿って述べる．

はじめに，対象とする処理系のソースコードへ注釈を加える (1)．注釈とは提案システムが提供する関数の呼び出しであり，プログラムカウンタを後方へ戻す箇所に挿入する．PUC-Rio Lua に加えた変更のうち，宣言と，注釈を挿入した箇所の 1 つを図 6 に示す．

前半は宣言である．注釈の関数 `backedge` は処理系の側に定義を持たないため，あらかじめインポート関数として宣言しておく．

後半の `dojump` は，分岐命令の飛び先へ制御を移す既存のマクロであり，変位 `offset_`をプログラムカウンタに加算する．ここに加えた注釈は，`if (offset_ < 0) backedge(...)` の 1 行である．変位が負であるとき，すなわちプログラムカウンタが後方へ移動したときに限り，移動後のプログラムカウンタの値を引数として `backedge` を呼び出す．

`backedge` は，プログラムカウンタを後方へ戻す箇所のすべてに挿入する必要がある．挿入されていない箇所は，そこを後方ジャンプとするループが提案システムから観測されず，機械語へコンパイルされない．PUC-Rio Lua にはここに示した以外にも後方ジャンプがあり，それらにも同じ呼び出しを挿入した．

次に，注釈を加えた処理系を WebAssembly へコンパイルする (2)．`backedge` は WebAssembly のインポート関数としてコンパイルされる．

最後に，得られたモジュールと，対象言語プログラムを提案システムに与える (3)．以降，処理系が対象言語プログラムの後方ジャンプを実行するたびに，提案システムの `backedge` が呼び出される．このとき渡されるプログラムカウンタの値は，対象言語プログラムにおけるループの先頭位置を一意に示す．したがって提案システムは，この値ごとに呼び出し回数を数えることで，対象言語プログラムのどのループが何回反復したかを，処理系の内部構造に依存せずに知ることができる．回数が閾値を超えたループが，トレースの記録対象となる．

4.4 システムの構成

提案システムは，WebAssembly の命令列を解釈実行するインタプリタを中心に構成される．以下，実行の流れに沿って各要素の役割を述べる．実行は WebAssembly インタプリタ (4) から始まる．前節で述べたとおり，注釈による呼び出しの回数が閾値を超えたループはホットループとして検出され，その実行がトレースとして記録される (5)．

トレースには実行された WebAssembly の命令が記録される．記録を終えると，得られたトレースを最適化して機械語へコンパイルする．以降そのホットループの反復では，この機械語を実行する (6)．

脱最適化の際には，インタプリタがその地点から実行を継続できるよう，次の 2 つを行う必要がある．1 つは，機械語の実行中に更新された値を，CPU レジスタやメモリからインタプリタが管理する領域へ書き戻すことである．もう 1 つは，インタプリタのプログラムカウンタを，成立しなかったガードに対応する位置へ更新することである．これを状態の復元と呼び，その方法は 5 章で述べる．

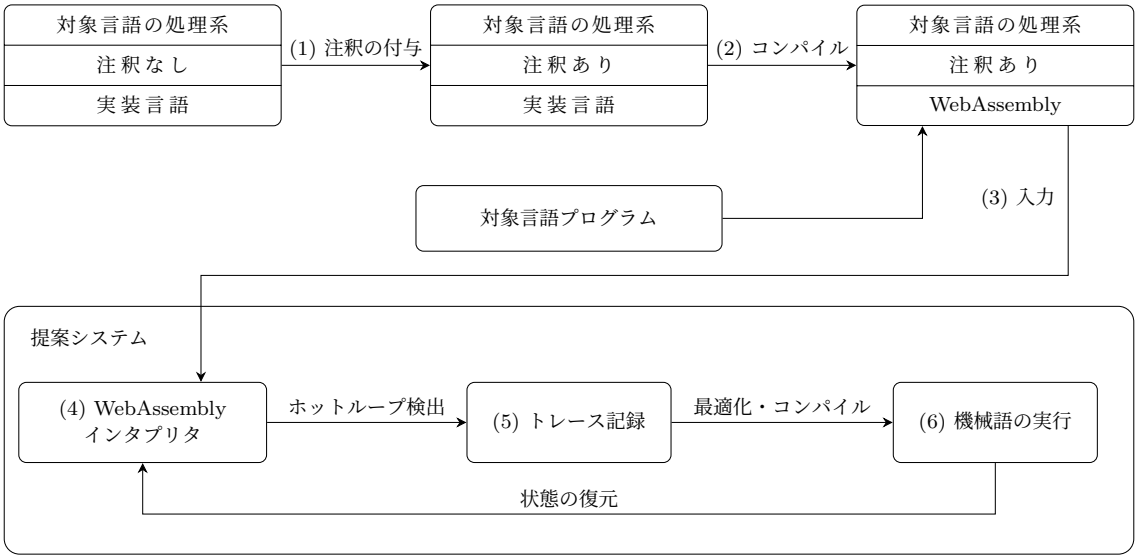


図 5　提案システムの利用手順と構成

```
1 __attribute__((import_module("env"), import_name("backedge")))
2 extern void backedge(unsigned int pc);
3
4 #define dojump(ci,i,e) { \
5   int offset_ = GETARG_sJ(i) + e; \
6   pc += offset_; \
7   if (offset_ < 0) backedge((unsigned int)(uintptr_t)pc); \
8   updatetrap(ci); \
9 }
```

図 6　PUC-Rio Lua に加えた変更（インポート関数の宣言と，注釈を挿入した箇所の 1 つ）

4.5　インタプリタループの畳み込み

トレースには，対象言語プログラムのループ 1 反復分の命令列が収まる．その内容は，インタプリタループを対象言語の命令の個数だけ並べた形になる．本節では，この命令列のうちインタプリタループを定数の畳み込みによって取り除く方法を述べる．以降この最適化をインタプリタループの畳み込みと呼ぶ．

トレースを最適化前に独自の中間表現へ変換する．WebAssembly はスタックマシンであり，ある命令が消費する値がどの命令の結果であるかが命令列の形からは直接読み取れないためである．提案システムはトレースを，単一代入形式の中間表現へ変換する．この形式では値の定義と使用の関係が表現の上に現れるため，以下に述べる最適化を適用できる．以降，最適化はこの中間表現に対して行うものとして述べる．

図 7 の (a) は，説明のため簡略化したインタプリタの実装である．(b) はこれを実行した場合に記録されたトレースの一部を中間表現で示したものである．r1，r2 などが中間表現における値であり，それぞれ一度だけ定義される．

インタプリタは，pc の指す位置から命令語 i を読み出し，そのオペコードに応じた処理へ分岐し演算を行う．トレースにはこの手順がそのまま現れる．対象言語プログラムが求めているのは r10 の乗算 1 つであり，それ以外はすべてインタプリタが命令を実行するための処理である．

pc の指すバイトコード列は線形メモリ上に格納されており，実行中に書き換えられない．したがって r2

(a) インタプリタループ

```
 1 for (;;) {
 2   i = *pc;              // 命令の読み出し
 3   switch (opcode(i)) {  // 命令に応じた分岐
 4     ...
 5     case MUL:
 6       dst = base + arg_a(i);
 7       lhs = base + arg_b(i);
 8       rhs = base + arg_c(i);
 9       *dst = *lhs * *rhs;
10       pc++;
11       break;
```

(b) 記録されたトレースの中間表現

```
 1 r1 = local.get 2                          ; pc
 2 r2 = i32.load [r1]                        ; i = *pc
 3 r3 = i32.and  r2 127                      ; opcode(i)
 4 r4 = i32.load [(i32.shl r3 2) + 11552]    ; switch のジャンプ表
 5 guard r4 == 23                            ; case MUL への分岐
 6 r5 = (local.get 5) + ((r2 >>  4) & 4080)  ; dst
 7 r6 = (local.get 5) + ((r2 >> 12) & 4080)  ; lhs
 8 r7 = (local.get 5) + ((r2 >> 20) & 4080)  ; rhs
 9 r8  = i64.load [r6]
10 r9  = i64.load [r7]
11 r10 = i64.mul  r8 r9
12 i64.store [r5] = r10                      ; *dst = *lhs * *rhs
```

(c) 定数の畳み込みを適用した後の中間表現

```
1 guard local 2       == 125936             ; pc
2 guard local 5       == 102960             ; base
3 guard mem[125936]   == 67306776           ; *pc
4 r8  = i64.load [103008]
5 r9  = i64.load [103024]
6 r10 = i64.mul  r8 r9
7 i64.store [103040] = r10
```

図 7　インタプリタループ (a)，記録されたトレースの中間表現 (b)，および定数の畳み込みを適用した後の中間表現 (c)

を定数へ畳み込めれば，r3 も r4 も定数となる．r4 が定数であれば，ガードはコンパイル時に除去できる．r5 から r7 も r2 からの演算であるから同様に定数となる．このような連鎖によってインタプリタループを畳み込むことができる．

畳み込みの妨げとなるのは，読み出す値がコンパイル時に定まらないことである．そこで本手法は，この値を記録時に観測された数値で置き換える．例えば `r1 = local.get 2` がトレースとして記録されたとき `local.get 2` の結果 125936 も同時に記録し，この値を定数として畳み込む．

しかし，読み出しを一律に定数へ置き換えると，反復ごとに変化する値まで固定され，計算結果が変わる．例えば線形メモリには，バイトコード列だけでなく，対象言語プログラムが操作するオブジェクトも置かれている．そこで，定数として扱った値には，記録時と同じ値であることを確かめるガードを置く．一方，機械語には記録時の値が畳み込まれており，値が変化した後に再利用すると計算結果が変わる．このため，ガードが成立しない場合は，その機械語を破棄してインタプリタでの実行へ戻り，次に同じループがホットループとなった時点で記録からやり直す．またガードを置くだけでは，破棄と再コンパイルが繰り返されて性能が低下する．そこで同じループについて記録を一定の回数繰り返し，すべての記録で一致した位置のみを定数へ置き換える．

5　実装

提案システムは Rust で実装した．機械語へのコンパイルには Cranelift [8] を用いた．

WebAssembly の値には 32 ビットと 64 ビットの整数および浮動小数点数のほか，ベクトル型と参照型がある．実装ではインタプリタはこれらをすべて実行できるが，ベクトル型または参照型の値を含むトレースは機械語へコンパイルする前に除外し，そのループはインタプリタによる実行のままとする．線形メモリを伸長する命令も同様に機械語へコンパイルする対象から外している．WebAssembly の実行環境が提供するシステムインタフェースの仕様である WASI [26] については，対象言語の処理系が，スクリ

プトファイルを読み込み，結果を出力するために必要な範囲で実装した．具体的にはファイル記述子に対する入出力，コマンドライン引数の取得，時刻と乱数の取得などである．

トレースの記録と管理の構成は TraceMonkey [11] を参考にした．ホットループについて最初に記録した 1 本のトレースを，ルートトレースと呼ぶ．同じガードでの脱最適化が繰り返される場合は，その地点を起点として別のトレースを記録し，既存のトレースに接続する．このように既存のトレースから派生したトレースを，分岐トレースと呼ぶ．分岐トレースの記録は，ルートトレースと同じループの後方ジャンプに到達した時点で完了する．1 本のルートトレースと，そこから派生した分岐トレースの集合をトレースツリー [12] と呼び，機械語へのコンパイルはこの単位で行う．この構成により，ホットループ内で分岐先が安定しない場合でも機械語を実行できる割合が増える．また，記録を完了できないループ，すなわち記録の途中で現在の関数から抜けるループや，記録した命令数が上限を超えるループは，以後の記録の対象から外し，インタプリタによる実行に委ねる．

4 章で述べたとおり，ガードが成立しなかった場合は，その地点でインタプリタが持つべきローカル変数とスタックの内容を書き戻したうえで，インタプリタが実行を引き継ぐ．提案システムは，これらの各要素へトレース中のどの値を書き戻すかと，インタプリタが実行を再開する位置とを記述した表を，ガードごとに 1 つ生成する．これをスナップショットと呼ぶ．トレース中のガードの個数とスナップショットの個数は一致する．書き戻しの対象となるのは，そのガードの地点までにトレースが更新したローカル変数である．図 7 の (a) における pc のように，インタプリタが反復のたびに更新する変数がこれにあたる．トレースが読み出しただけのローカル変数は，インタプリタのフレームが既に正しい値を保持しているため対象としない．グローバル変数と線形メモリは復元の対象に含めない．インタプリタと機械語が同一のメモリを読み書きしているため，脱最適化が起きた時点の内容がそのまま正しいからである．ガードが成立する限りスナップショットは参照されず，状態の復元はガードが成立しなかったときにのみ行われる．

表 1　適用した処理系と注釈による変更量

対象言語	処理系	版	変更行数
Lua	PUC-Rio Lua	5.5.0	13
JavaScript	QuickJS	2026-06-04	27
Ruby	CRuby	4.0.3	71
Python	CPython	3.14.6	9

6　適用と評価

本手法を，Lua，JavaScript，Ruby，Python の 4 言語の処理系，すなわち PUC-Rio Lua 5.5.0 [15]，QuickJS（2026-06-04 版）[3]，CRuby 4.0.3 [21]，CPython 3.14.6 [18] に適用した．

各処理系に加えた変更行数を表 1 に示す．変更行数は 4 章で述べた注釈の挿入によるもののみであり，WebAssembly へのビルドのための設定ファイルの変更などは含まない．

行数の差は，後方ジャンプが処理系内でどのように実装されているかによって決まる．QuickJS が 27 行となるのは同一の後方ジャンプがオペランドの幅ごとに別の命令として実装されているためであり，CPython が 9 行にとどまるのは命令の定義が専用の記述から生成され，後方ジャンプを行う 3 つの命令が 1 つの記述を共有しているためである．

Ruby はこれらと異なる．Ruby で繰り返しを記述する標準的な方法は `arr.each { ... }` や `n.times { ... }` であり，これらは C 言語で実装されたメソッドがブロックを繰り返し呼び出す構造をとる．バイトコード上の後方ジャンプ箇所のみに注釈を挿入した場合，検出できるループは `while` によるものに限られる．そこで，`each` や `times` に対応する C 言語の実装にも注釈を挿入した．その結果，他の 3 処理系より変更行数が多い．

同一のスクリプトをネイティブインタプリタと提案システムとで実行し，プロセスの生成から終了までの時間を比較した．

提案システム　計測の対象である．表 1 の処理系を WebAssembly へコンパイルしたモジュールを，インタプリタとトレーシング JIT で実行する．

ネイティブインタプリタ　同じソースコードから機

表 2　計測環境

項目	内容
CPU	AMD Ryzen 7 7735U
主記憶	19.1 GiB（WSL2 への割り当て）
OS	Arch Linux（WSL2）
カーネル	6.6.87.2
提案システム	rustc 1.93.0 による release ビルド
対象言語の処理系	wasi-sdk 27 [27] でビルド

械語へコンパイルした処理系である．WebAssembly を経由せず機械語として直接実行されるため，WebAssembly インタプリタによる実行より原理的に高速である．

ベンチマークには 9 種類のプログラムを用い，4 言語それぞれに同じ処理を記述して計 36 ケースとした．内訳は，整数の累積和（sum），累積和に剰余演算を適用して値を固定長整数の範囲に収めたもの（sum_modulo），フィボナッチ数列の反復計算（fib），ライプニッツ級数による円周率の近似（pi），三重ループによる加算（nested_loop），配列の交換整列（bubblesort），エラトステネスの篩（sieve），べき乗法による行列のスペクトルノルムの計算（spectral_norm），行列積（matmul）である．これらは本研究で作成したものであり，いずれもループが実行時間の大部分を占め，外部ライブラリを用いない．

計測では，36 ケースを 1 回ずつ実行することを 10 回繰り返した．1 つのケースの中では，提案システムとネイティブインタプリタを続けて実行する．速度比は，同一の計測内で続けて実行した両者の実行時間の比である．各ケースの代表値には，10 回の計測で得た速度比の中央値を用いた．

計測の対象はループの実行時間であるが，実際に計測できるのはプロセス全体の時間であり，処理系の起動に要する時間も含まれる．そのため各ケースの引数は，提案システムのループ実行時間がその起動時間と 10:1 の比になるように定めた．計測環境を表 2 に示す．

処理系の版は表 1 に示したとおりであり，ネイティブインタプリタも同一のソースコードから生成した．CRuby と CPython は JIT コンパイラを無効にしてビルドした．表 3 に，ネイティブインタプリタの実行時間を提案システムの実行時間で割った値を示す．

表 3　ネイティブインタプリタに対する速度比（太字は提案システムが速いケース）

ベンチマーク	Lua	JavaScript	Ruby	Python
sum_modulo	**3.50**	**6.00**	**1.51**	**1.61**
sum	**5.93**	**1.63**	0.03	**1.18**
fib	**3.01**	**5.55**	0.01	0.01
bubblesort	**3.03**	**4.97**	0.07	0.62
spectral_norm	**2.95**	**2.17**	0.02	0.27
nested_loop	**1.53**	**3.66**	0.75	0.78
pi	**1.08**	**1.64**	0.02	**1.11**
sieve	0.15	0.11	**1.36**	0.45
matmul	0.04	0.59	0.01	0.86

1 を超える場合，提案システムのほうが性能が高いことを表す．

36 ケースのうち 19 ケースで提案システムがネイティブインタプリタを上回った．言語ごとの内訳は，Lua が 9 ケース中 7 ケース，JavaScript が 7 ケース，Ruby が 2 ケース，Python が 3 ケースである．4 言語すべてで下回るベンチマークは matmul のみであり，sieve は Ruby でのみ 1 を超える．

36 ケース中 33 ケースでは，10 回の計測で得た速度比の最大値と最小値の差が中央値の 25%以内に収まった．これを超えるのは Python の fib，Ruby の nested_loop，Ruby の pi の 3 ケースであり，いずれも 10 回のうち 1 回だけネイティブインタプリタの実行時間が中央値の 1.7〜2.3 倍になった．しかし速度比の代表値には中央値を用いているため，表 3 の値はこの 1 回の影響を受けない．提案システム側の実行時間のばらつきは，線形メモリ上に確保するヒープの配置が実行ごとに変わり，記録されるトレースが変わることによる．対象言語の文字列ハッシュのシードを固定する条件で計測すると，これら 3 ケースにおける提案システム側の実行時間の最大値と最小値の比は，1.20〜1.35 倍であったものが 1.05〜1.19 倍に下がる．

ネイティブインタプリタを上回った 19 ケースでは，4 章で述べたインタプリタループの畳み込みが適用され，かつ生成された機械語がループの反復のほぼすべてを実行している．表 4 に，4 言語すべてで上回ったベンチマークである sum_modulo について，最も多く実行された経路の最適化前後の命令数，およびループの反復のうち機械語が実行した割合を示す．

削減率は 72〜88%であり，ガードの個数は Lua で

表 4　sum_modulo における命令数の削減と，ループの反復のうち機械語が実行した割合

対象言語	最適化前の命令数	最適化後の命令数	機械語が実行した反復の割合
Lua	416	52	99.99997%
JavaScript	739	156	99.99961%
Ruby	1864	442	99.99963%
Python	2392	667	99.99629%

62 個から 4 個，JavaScript で 116 個から 35 個，Ruby で 303 個から 70 個，Python で 314 個から 115 個へ減少する．この削減率から，これらのケースではインタプリタループの畳み込みが有効に機能したと考えられる．最適化後の 1 反復の命令数は対象言語によって幅があり，Lua の 52 に対し Python は 667 である．これは 1 つのバイトコード命令に対応する処理量の差を反映している．

一方，ネイティブインタプリタに及ばないケースがある．原因は 2 つに分けられる．以下では，`perf` によりユーザ空間で実行された CPU 命令数を計測した．内訳については，原因となる処理を取り除いたスクリプトを実行し，その命令数との差から求めた．

1 つは，実行された命令の大半が本手法の適用範囲外にあることである．本手法が最適化の対象とするのは，注釈の置かれた箇所，すなわち対象言語プログラムのループである．一方，処理系の内部には C 言語で記述されたループがある．ガベージコレクション，記憶領域の確保，組み込み関数の実装に含まれるループがこれにあたる．これらは注釈を持たないため，ホットループとして検出されず，インタプリタによる実行のままとなる．

遅いケースでは，対象言語プログラムのループの外にある命令が，実行された命令の大部分を占める．以下に示す割合は，いずれも各ケースで実行された命令の総数を 100%としたときの値である．ループの外にある命令の割合は，Lua の sieve で 94.4%，JavaScript の sieve で 79.3%，Ruby の pi で 97.8%である．Lua の sieve では，ガベージコレクションが 45.8%，テーブルの再ハッシュと要素の挿入が 48.6%を占め，両者の合計はループの外にある 94.4%と一致する．JavaScript の sieve では，配列の初期化を行う組み込み関数 `fill` が 79.3%の大半を占める．Ruby の pi では，ガベージコレクションが 74.5%，浮動小数点数のヒープ確保とその演算が 23.3%を占める．

JavaScript の sieve では，配列の初期化を組み込み関数から対象言語プログラムのループへ書き換えると，計算結果を変えないまま，実行された命令数が 21.96×10^9 から 9.83×10^9 へ減少する．組み込み関数の本体は処理系の C 言語のコードであり，注釈を持たない．一方，同じ処理を対象言語プログラムのループとして記述すると，注釈の置かれた箇所を通過するため，ホットループとして検出される．

Ruby の pi でガベージコレクションの割合が大きいのは，比較する 2 つの実行ファイルで値表現が異なるためである．CRuby は，値を表す 1 ワード内に浮動小数点数を直接埋め込む表現（flonum）を持つ．この表現は，1 ワードの幅が倍精度浮動小数点数の幅以上の場合にのみ用いられる．32 ビットの WebAssembly では 1 ワードが 4 バイトであるため flonum は用いられず，演算のたびに浮動小数点数がヒープへ確保される．pi のループは 1 反復あたり 2 個の浮動小数点数を確保する．比較対象である 64 ビットのネイティブインタプリタでは flonum が用いられ，この確保は起きない．64 ビットの WebAssembly を用いた計測は行っていない．

もう 1 つは，機械語へのコンパイルに要する時間であり，実行時間の最大 48.6%を占める．コンパイル時間については，CPU 命令数ではなく実時間で計測した．コンパイル処理の所要時間をプロセスの実行中を通じて積算し，プロセス全体の経過時間と比較したものであり，各ケースを 3 回実行して中央値を用いた．36 ケース中 7 ケースで，コンパイル時間が全体の実行時間の 10%を超えた．残る 29 ケースはいずれも 10%未満であり，うち 25 ケースは 5%未満である．表 5 に，この 7 ケースについて，コンパイル時間が占

める割合，記録されたトレースの種類ごとのコンパイル回数，および 1 回あたりのコンパイル時間を示す．

コンパイル回数を増やしているのは分岐トレースであり，10%を超えた 7 ケースのうち 6 ケースで，コンパイル回数の過半を分岐トレースが占める．分岐トレースは，ガードが繰り返し不成立となった箇所ごとに新たに記録してコンパイルされるため，これらのケースには脱最適化が繰り返される経路が存在することが分かる．トレース 1 回あたりのコンパイル時間は対象言語によって異なり，36 ケース全体では Lua で 2.1〜9.1 ミリ秒，JavaScript で 4.2〜17.3 ミリ秒，Ruby で 15.1〜65.7 ミリ秒，Python で 9.5〜30.3 ミリ秒である．Ruby の matmul はコンパイル回数と合計のコンパイル時間がともに最大であり，120 回のコンパイルに 5.12 秒を要している．

7 関連研究

注釈によって対象言語の実行位置を最適化系へ伝える手法は，Sullivan ら [24] が示している．これは機械語を対象とする動的最適化系 DynamoRIO [6] をインタプリタの実行に適用したものであり，注釈は 2 種類の情報からなる．1 つは対象言語プログラム上の実行位置であり，本手法の注釈と同じ役割を果たす．もう 1 つはバイトコード列を格納した記憶領域の宣言である．トレース中のバイトコードの読み出しを定数へ畳み込むには，その領域が実行の間書き換えられないことを処理系が知る必要がある．本手法がこれを実行時の観測によって判定し，その値にガードを置いて実行時に確認するのに対し，この研究では処理系の実装者が宣言する．評価は C 言語で記述された小規模な VM に対して行われている．

3 章で挙げた Yermolovich ら [29] は，C 言語で記述された Lua のインタプリタを ActionScript のバイトコードへコンパイルし，インタプリタループを示すヒントを与えて，トレーシング JIT を備えた Tamarin-Tracing 上で実行する．既存の処理系を再実装せずに中間言語へコンパイルする点は本研究と一致するが，用いるのは ActionScript の実行のために設計された既存の VM であり，適用対象も Lua の 1 言語である．本研究は，複数の処理系を同一の実装で最適化するために，WebAssembly に対するトレーシング JIT を実装している．WebAssembly を対象としたメタトレーシングの試みとしては Brown ら [5] の発表があり，実装は進行中のものとして報告されている．

インタプリタの記述から JIT コンパイラを導出する手法として，メタトレーシングと部分評価がある．メタトレーシングを採る PyPy [4] [20] は，RPython で記述したインタプリタから，その実行をトレースする JIT コンパイラを自動的に生成する．インタプリタに与える注釈は 2 種類で，うち 1 つはプログラムカウンタを後方へ戻しうる箇所を示すものであり，本手法の後方ジャンプの注釈と同じ情報を伝える．部分評価を採る Truffle [28] は，Java で記述したインタプリタと解釈対象のプログラムとを入力とする部分評価によって最適化コードを導出する．いずれも言語の意味をインタプリタとして一度記述すればよく，JIT コンパイラの実装工数を削減するという点で本研究と目標を共有するが，対象となるのはそれぞれ RPython と Java で記述されたインタプリタであり，既存の処理系には適用できない．本研究は，C 言語で記述された既存の処理系を WebAssembly へコンパイルすることでこの制約を取り除いている．一方で PyPy には，ある値を定数として扱わせる注釈のように，性能を得るための追加の注釈が用意されている．実装者はこれらの注釈を通じて，バイトコード列が実行中に書き換えられないといった知識を JIT コンパイラへ伝えられる．本手法にはこのような注釈がないため，同等の情報を実行時の観測によって取得している．

weval [10] も部分評価を採るが，対象とするのは C 言語などで実装された既存のインタプリタを WebAssembly へコンパイルしたモジュールである．既存の処理系を再実装せずに用いる点，WebAssembly を対象とする点，プログラムカウンタを渡す呼び出しをインタプリタへ注釈として挿入する点で，本研究と類似している．異なるのは最適化を行う時点である．weval は対象言語プログラムのバイトコードを入力として，実行に先立ってそれに特化した関数を生成する．インタプリタループはこの時点で展開され，バイトコード命令ごとの制御フローが確定するため，

表 5　実行時間に占めるコンパイル時間の割合（10%を超えた 7 ケース）

ケース	コンパイル時間が占める割合	コンパイル回数 ルート	コンパイル回数 分岐	1 回あたりのコンパイル時間
Lua matmul	48.6%	10	23	7.6 ミリ秒
JavaScript matmul	45.6%	11	42	8.8 ミリ秒
Python fib	39.9%	7	31	24.4 ミリ秒
Ruby matmul	25.5%	35	85	42.6 ミリ秒
Python spectral_norm	24.8%	10	61	24.8 ミリ秒
Lua spectral_norm	13.6%	14	0	9.1 ミリ秒
Python matmul	12.1%	20	24	30.3 ミリ秒

生成に要する時間は実行時間に含まれない．一方，実行時にしか得られない情報は利用できない．本研究はこれに対し，実行中に記録したトレース上の値を観測された定数へ置き換え，その仮定が成立することをガードによって実行時に確認する．両者が用いる情報は排他的でないため，組み合わせる余地がある．

8　まとめと今後の課題

本研究では，既存のインタプリタを再実装することなく JIT コンパイラを備えた処理系を得るシステムを提案し，実装した．処理系の実装者が行う変更は，ループを構成する後方ジャンプの箇所に注釈を加えることのみである．PUC-Rio Lua，QuickJS，CRuby，CPython の 4 処理系に適用し，必要となった変更はそれぞれ 13 行，27 行，71 行，9 行であった．9 種類のプログラムを 4 言語で記述した 36 ケースのベンチマークでは，ネイティブインタプリタを 19 ケースで上回った．

以下に，本研究で扱えなかった課題を 2 点述べる．

1 つは，事前の最適化との組み合わせである．提案システムは，トレースの記録と機械語の生成をすべて実行時に行い，機械語へコンパイルされない範囲は WebAssembly インタプリタで実行する．6 章で示したとおり，ネイティブインタプリタに及ばないケースでは，実行された命令の大部分が対象言語プログラムのループの外，すなわちガベージコレクション，記憶領域の確保，組み込み関数の実装にあり，これらは注釈を持たないため提案システムの最適化対象とならない．このような本手法の最適化対象とならないコードを事前コンパイルしておくことで，機械語が実行される割合を高め，性能を向上させられると考えられる．さらに進んだ事前最適化として，7 章で述べた weval [10] がある．weval が生成するモジュールでは，バイトコードの読み出しと命令の種類による分岐が既に解決されている一方，値の型に依存する処理は実行時に残る．これを提案システムの入力とすれば，事前に解決された範囲をそのまま利用したうえで，残る部分に実行時の観測に基づく最適化を適用できる可能性がある．

もう 1 つは，外部ライブラリの呼び出しへの対応である．本論文の評価は，外部ライブラリを用いないプログラムに限られている．実際に利用されるプログラムの多くは，C 言語で実装された外部ライブラリを呼び出す．外部ライブラリを WebAssembly へコンパイルする場合，処理系と外部ライブラリの双方が同一の命令列として実行されるため，提案システムによる両者の境界をまたぐ最適化が原理的に可能である．ネイティブインタプリタでは外部ライブラリは既に機械語であり JIT コンパイラの対象とならないことと対照的であり，本手法の利点となりうる．また外部ライブラリの呼び出しに対応することで，実際のアプリケーションを対象とした評価が可能になる．

謝辞

本研究の遂行にあたり，議論を通じて有益な助言をいただいた分散システムソフトウェア研究室の皆様に感謝します．

参 考 文 献

[1] Bala, V., Duesterwald, E., and Banerjia, S.: Dynamo: a transparent dynamic optimization system, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, ACM, 2000, pp. 1–12. DOI: 10.1145/349299.349303.

[2] Bebenita, M., Brandner, F., Fähndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H.: SPUR: a trace-based JIT compiler for CIL, *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, ACM, 2010, pp. 708–725. DOI: 10.1145/1869459.1869517.

[3] Bellard, F. and Gordon, C.: QuickJS Javascript Engine, `https://bellard.org/quickjs/`. 参照 2026-08-06.

[4] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A.: Tracing the meta-level: PyPy's tracing JIT compiler, *Proc. 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, ACM, 2009, pp. 18–25. DOI: 10.1145/1565824.1565827.

[5] Brown, A. and Tolmach, A.: Meta-tracing Interpreters in WebAssembly, WebAssembly Workshop (WAW 2025), POPL 2025 併設, Denver, `https://popl25.sigplan.org/details/waw-2025-papers/2/Meta-tracing-Interpreters-in-WebAssembly`, 2025. work in progress, 正式な会議録・DOI なし. 参照 2026-08-06.

[6] Bruening, D., Garnett, T., and Amarasinghe, S.: An infrastructure for adaptive dynamic optimization, *Proc. International Symposium on Code Generation and Optimization (CGO 2003)*, IEEE, 2003, pp. 265–275. DOI: 10.1109/CGO.2003.1191551.

[7] Bucher, B. and Ostrowski, S.: PEP 744 – JIT Compilation, Python Enhancement Proposals, `https://peps.python.org/pep-0744/`, 2024. 参照 2026-08-06.

[8] Bytecode Alliance: Cranelift Code Generator, `https://cranelift.dev/`. ソースは Wasmtime のリポジトリに含まれる. 参照 2026-08-06.

[9] Chevalier-Boisvert, M., Gibbs, N., Boussier, J., Wu, S. X., Patterson, A., Newton, K., and Hawthorn, J.: YJIT: A Basic Block Versioning JIT Compiler for CRuby, *Proc. 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2021)*, ACM, 2021, pp. 25–32. DOI: 10.1145/3486606.3486781.

[10] Fallin, C. and Bernstein, M.: Partial Evaluation, Whole-Program Compilation, *Proc. ACM Program. Lang.*, Vol. 9, No. PLDI(2025), pp. 324–347. DOI: 10.1145/3729259.

[11] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M.: Trace-based just-in-time type specialization for dynamic languages, *Proc. 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, ACM, 2009, pp. 465–478. DOI: 10.1145/1542476.1542528.

[12] Gal, A., Probst, C. W., and Franz, M.: HotpathVM: an effective JIT compiler for resource-constrained devices, *Proc. 2nd International Conference on Virtual Execution Environments (VEE '06)*, ACM, 2006, pp. 144–153. DOI: 10.1145/1134760.1134780.

[13] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J.: Bringing the web up to speed with WebAssembly, *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, ACM, 2017, pp. 185–200. DOI: 10.1145/3062341.3062363.

[14] Lua.org, PUC-Rio: Lua versions, `https://www.lua.org/versions.html`. 参照 2026-08-06.

[15] Lua.org, PUC-Rio: The Programming Language Lua, `https://www.lua.org/`. 参照 2026-08-06.

[16] Pall, M.: The LuaJIT Project, `https://luajit.org/`. 参照 2026-08-06.

[17] PyPy: Download and Install, `https://pypy.org/download.html`. 参照 2026-08-06.

[18] Python Software Foundation: Python, `https://www.python.org/`. 参照 2026-08-06.

[19] Python Software Foundation: Status of Python versions, Python Developer's Guide, `https://devguide.python.org/versions/`. 参照 2026-08-06.

[20] Rigo, A. and Pedroni, S.: PyPy's approach to virtual machine construction, *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06), Dynamic Languages Symposium (DLS 2006)*, ACM, 2006, pp. 944–953. DOI: 10.1145/1176617.1176753.

[21] Ruby Community: Ruby Programming Language, `https://www.ruby-lang.org/`. 参照 2026-08-06.

[22] Savrun-Yeniçeri, G., Zhang, W., Zhang, H., Seckler, E., Li, C., Brunthaler, S., Larsen, P., and Franz, M.: Efficient hosted interpreters on the JVM, *ACM Transactions on Architecture and Code Optimization*, Vol. 11, No. 1(2014), pp. 9:1–9:24. Article 9. DOI: 10.1145/2532642.

[23] Stogov, D. and Suraski, Z.: PHP RFC: JIT, php.net wiki, `https://wiki.php.net/rfc/jit`, 2019. PHP 8.0 で実装. 参照 2026-08-06.

[24] Sullivan, G. T., Bruening, D. L., Baron, I., Garnett, T., and Amarasinghe, S.: Dynamic native optimization of interpreters, *Proc. 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME '03)*, ACM, 2003, pp. 50–57. DOI: 10.1145/858570.858576.

[25] W3C WebAssembly Working Group: We-

bAssembly Core Specification, Version 2.0, W3C Candidate Recommendation Draft, https://www.w3.org/TR/wasm-core-2/, 2025. 参照 2026-08-06.

[26] WASI Subgroup, W3C WebAssembly Community Group: WASI: The WebAssembly System Interface, https://wasi.dev. 最新のマイルストーンリリースは WASI 0.3. 参照 2026-08-06.

[27] WebAssembly Community Group: wasi-sdk, https://github.com/WebAssembly/wasi-sdk. 参照 2026-08-06.

[28] Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D., and Grimmer, M.: Practical partial evaluation for high-performance dynamic language runtimes, *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, ACM, 2017, pp. 662–676. DOI: 10.1145/3062341.3062381.

[29] Yermolovich, A., Wimmer, C., and Franz, M.: Optimization of dynamic languages using hierarchical layering of virtual machines, *Proc. 5th Symposium on Dynamic Languages (DLS '09)*, ACM, 2009, pp. 79–88. DOI: 10.1145/1640134.1640147.