公開

研 究 終 了 報 告 書

「汎用性と高性能を両立するハイブリッド型実行時コンパイラ」

研究期間： 2020 年 11 月～2023 年 3 月
研 究 者： 伊澤 侑祐

1． 研究のねらい

本研究は実行時コンパイラという高速化技術に着目している。実行時コンパイラは、Java や JavaScript 言語などの仮想機械で採用されており、高性能を達成するために不可欠なものとなっている。また、本研究はメタ実行時コンパイラフレームワークという、特定の言語に依存しない実行時コンパイル技術にも着目している。インタプリタ定義から実行時コンパイラを生成することができるため、あらゆる言語に適用できる実行時コンパイラを実現できる。また、確立された技術を言語ごとにコンパイラを 1 から実装し直す必要がないという開発上の利点もある。

本研究は、様々なコンパイルが可能なハイブリッド型コンパイラを、メタ実行時コンパイラフレームワークとして実現する。あらゆる言語でプログラムのさらなる高速化が可能な技術的基盤の構築がねらいである。

実行時コンパイラのコンパイル方式には様々な種類があり、その中でもメソッド単位でコンパイルを行うメソッド方式と、実行履歴単位でコンパイルを行うトレーシング方式の 2 つが多くの仮想機械で採用されてきた。この 2 つの方式はそれぞれ適したプログラムが異なっているため、プログラムによっては 1 つの方式だけでは高速化に限度がある。両者を同時に使うことのできるハイブリッドコンパイル技術を実現できれば、プログラムのさらなる高速化を実現できる。さらに、ハイブリッドコンパイル技術をメタ実行時コンパイラフレームワークとして実現し、様々な言語でインタプリタを記述するだけでハイブリッドコンパイル技術を適用することを目指す。具体的には、高速な Python/Ruby 処理系である PyPy や Topaz を生成するために用いられている RPython フレームワークにハイブリッドコンパイル技術を実現する。特に、Python 言語は機械学習のみならず一般的なソフトウェア開発に広く用いられている言語であるため、RPython フレームワークを基盤とすることは本研究のアウトリーチに最適だと考えている。

2． 研究成果

(1)概要

メタ実行時コンパイラフレームワーク RPython に基づくメタハイブリッコンパイル技術の基盤構築

実行時コンパイル技術には様々な種類がある。実行履歴型やメソッド型など、高速な機械語を生成するための技術だけでなく、スレッデッドコード生成や基本ブロック型コンパイルといった、仮想機械の起動時における性能を向上するための技術も存在する。これらのコンパイル技術はそれぞれに利点があり、Java や JavaScript などの産業向け仮想機械では複数のコンパイル技術を組み合わせることによって高性能を達成している。一方、現状のメタ実行時コンパイラフレームワークはそのような機能はサポートされていない。その理由の一つに、コンパイル範囲や最適化レベルの異なる複数のコンパイラを生成する機能が備わっていなかったこ

ACT-X

とが挙げられる。

本研究は、コンパイル範囲・最適化レベルの異なるコンパイラを複数生成するための基盤技術をメタ実行時コンパイラフレームワーク RPytnon に実現した。RPython フレームワークは、ユーザが定義したインタプリタから実行履歴型の実行時コンパイラを生成する。本研究では、メタコンパイラをコンパイル範囲・最適化レベルに応じて 1 つずつ実装するのではなく、メタ実行時コンパイラの振舞をインタプリタによって制御することで実現しており、実現コストが小さいことも特徴である。生成された実行時コンパイラがインタプリタを実行しながらコンパイルを行うことに着目し、インタプリタに実行時コンパイラを制御するための特殊な制御コードを挿入してコンパイラの振舞を制御し、生成された中間コードをつなぎ合わせることによって本技術を達成している。制御コードを解釈するため、そして中間コードをつなぎ合わせるためにコンパイラを改変する必要はあるものの、コンパイラ全体を拡張するより低コストで実現できることも確認した。

評価・実験をするに当たり、高速な機械語を生成する目的の実行履歴型コンパイラに加えて、プログラムの実行初期における性能を向上する目的のメソッド型コンパイラを生成する仕組みを RPython フレームワークに実装し、その仕組みを Smalltalk 処理系に対して適用した。実験の結果、実行初期性能を改善するためのメソッド型コンパイラを組み合わせて実行することによって、実行履歴型コンパイラのみの場合よりも実行初期性能を改善することが確かめられた。

（2）詳細

研究テーマ A「メタ実行時コンパイラフレームワーク RPython を用いたスレッデッドコード生成」（Journal of Object Technology に出版済）

本研究は、仮想機械の起動時における性能を改善するためのスレッデッドコード生成手法をメタ実行時コンパイラフレームワークにおいて実現するアイディアを提案した。スレッデッドコードとは、各ハンドラへの呼出が並んでいるコード形式のことである。スレッデッドコード形式のプログラムは、コードサイズが小さく、インタプリタ実行よりも速い実行速度が特長である。このコード形式にコンパイルし実行することによって、仮想機械の起動時における性能の改善が期待できる。実現に当たり、研究成果 A によって示されたメソッド単位でのコンパイルを行う手法を、実用的なメタ実行履歴コンパイラフレームワークである RPython に適用する技術を提案した。結果、メソッド単位でのスレッデッドコード生成を実行履歴型コンパイルと組み合わせることにより、実行履歴型コンパイルのみの場合と比べて、仮想機械の起動時における性能を向上することが確認できた。

仮想機械型の処理系は、頻繁に実行されるコードは実行時コンパイルし、それ以外はインタプリタで実行する実行モデルを採用していることが一般的である。一方、初期化に非自明な処理のかかるアプリケーションの場合、実行時コンパイラが生成した機械語をほとんど実行せずアプリケーションの主処理実行へ移ってしまう。初期化処理に費やしたコンパイル時間とメモリが無駄になってしまうため、インタプリタと実行時コンパイラ以外に、最適化をあまり行わない

代わりに素早くコードを生成することのできる基本コンパイラを併用する仮想機械も存在している。

特定の言語に依存した仮想機械であれば、中間表現、インタプリタ、最適化器や実行時コンパイラなどの部品を自由にプログラム可能なため、基本コンパイラを一から作成することによって実現できる。一方、メタコンパイラフレームワークの場合、インタプリタはユーザが自由に記述できる一方、それ以外の最適化器や実行時コンパイラはフレームワークによって生成されたものを使用しなければならないため、これまでの手法で実現することは容易ではなかった。本研究では、ユーザが自由に記述できるインタプリタを用いて、生成された実行時コンパイラの振舞を制御することによって基本コンパイルを実現した。特に、メタ実行履歴コンパイラフレームワーク RPython を用いることによって、中間表現も自由に定義することが可能なため、スレッデッドコードという形式のコード生成も可能であることを示した。

この技術は、制御コードが挿入されたインタプリタ定義（method-traversal interpreter）と最小限のコンパイラ拡張（trace-stitching）によって実現されている。図 1 は、method-traversal interpreter と trace-stitching を用いてコンパイルする流れを示している。図 1 の左に表示されているプログラムを本研究の技術を用いてコンパイルするとき、method-traversal interpreter を用いてプログラムの実行履歴を収集する。次に、得られた中間表現を trace-stitching を利用してつなぎ合わせ、目的のコンパイルコードを得る（図 1 右）。

通常、メタ実行履歴型コンパイラは実際に実行した命令を収集する（トレースする）ため、プログラム中の分岐は片方しか収集しない。一方、method-traversal interpreter は実行履歴型コンパイラがプログラム中のすべての分岐を辿ることができるようになる。さらに、コンパイルの開始点・終了点をプログラム中のメソッドに限定することによってメソッド型コンパイルを達成できるという仕組みである。図 2 に Method-traversal interpreter を用いた実行履歴の収集の概観を示す。Method-traversal interpreter は、メタ実行履歴型コンパイラの挙動を制御コードを用いて変更する。具体的には、次のステップを踏む。

1. 分岐が発生するたびに実行しなかった分岐への行き先を保存する。
2. メソッドの終端に到達した場合、保存した未到達分岐への行き先があるかどうかをチェックする。
3. 未到達の分岐へ命令収集器を移動（バックトラック）させる。同時に、終端においてどのような命令があるべきか（図 2 の場合は JUMP 命令）を示した疑似コードを残す。
4. メソッド呼出があった場合、コードサイズの肥大化を避ける目的で呼出先の展開は行わない。

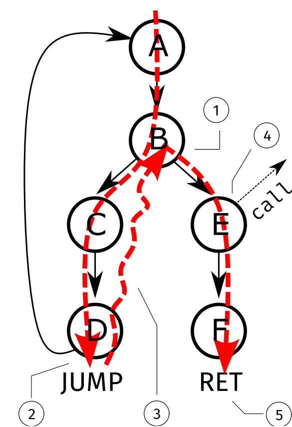

図 1. Method-traversal interpreter を用いた実行履歴の収集

ACT-X

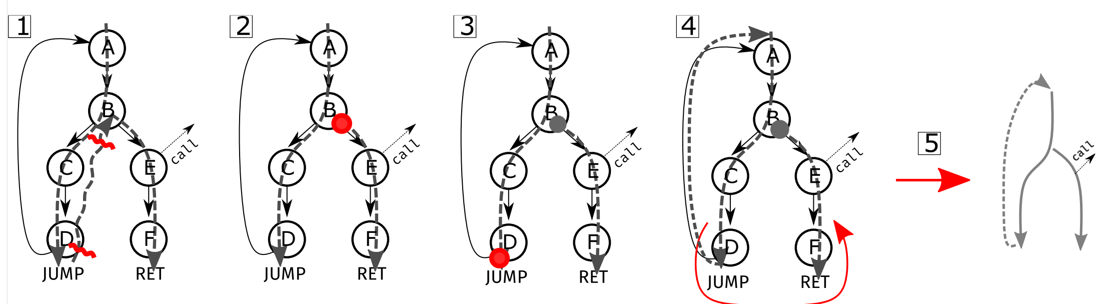

図 2. Trace-stitching の概観

⑤. メソッドの終端に到達し、保存した未到達の分岐がない場合、この時点で実行履歴の収集を終了する。

Method-traversal interpreter を用いて収集した実行履歴は一直線になっており、元のプログラムの制御構造を失なっている。そこで、trace-stitching を用いて得られた中間表現（トレース）から元の制御構造を復元する。図 3 に trace-stitching がどのように元の制御構造を復元するのかを示す。 具体的には、次のステップを踏む。

1. バックトラックが発生した場所（method-traversal interpreter におけるステップ③）でトレースを切り離す。
2. 切り離した先のトレースを元のトレースと接続する。
3. バックトラックが発生した場所（method-traversal interpreter におけるステップ③）に置かれた「JUMP 命令を復元するための疑似命令」を元に JUMP 命令を復元する。
4. 切り離された先のトレースの中で使用しなければならない命令が元のトレースに残っていた場合、その命令を元のトレースから切り離されたトレースへ複製する。
5. 最後に、トレースを機械語へ変換する。

以上のアイディアを Implementation, Compilation, Optimization of OO Languages, Programs and Systems (ICOOOLPS) 2021 国際ワークショップにて発表し、Journal of Object Technology に論文を出版した。

研究成果 B「RPython での様々なコンパイルの実現を目的とした”浅い”トレース手法」(ICOOOLPS 2022 国際ワークショップにて発表済み、MoreVMs 2023 国際ワークショップにて発表済み)

メタ実行履歴コンパイラフーレムワークは、インタプリタから実行履歴型のコンパイラを生成する処理系である。メタ実行履歴コンパイラは、実行履歴を収集する際、インタプリタ定義を実行する性質がある。Method-traversal interpreter は、メソッド型コンパイルを行うためにメタ実行履歴コンパイラにプログラムの全ての分岐を辿らせるため、プログラムにファイルへの書き込みやヒープへの読み書きといった副作用があった場合、method-traversal interpreter がプログラムの状態を壊す恐れがある。メタ実行履歴コンパイラでメソッド型コンパイルを行うときに正しい実行結果を得るためには、（1）実行履歴の収集中に分岐が発生するたびに状態を巻き戻す仕組みか、あるいは（2）状態を改変せずに実行履歴を収集する仕組みが必要である。

ACT-X

本研究は、(2) を実現するため、新たに「トレースの浅さ」を定義した。さらに、設定された浅さより深いハンドラは実行しないトレース手法 shallow tracing も併せて提案した。

トレースの浅さとは、インタプリタに定義されたハンドラがトレースに残存している度合いである。本研究では、図 3 のバイトコードをトレースしたとき、インタプリタのハンドラ呼出が全て残っているトレースを深さ 1 (図 3) 、スタック操作が畳み込まれているトレースを深さ 2 (図 3) 、そしてハンドラ呼出が全て展開されたトレースを深さ 3 (図 3) とした。特に、深さを 1 に設定すると、メソッド型スレッデッドコード生成が可能になるだけでなく、インタプリタ内部の状態やヒープを改変せずにトレースが取得できるため、(2) 状態を改変せずにトレースする手法が達成できる。

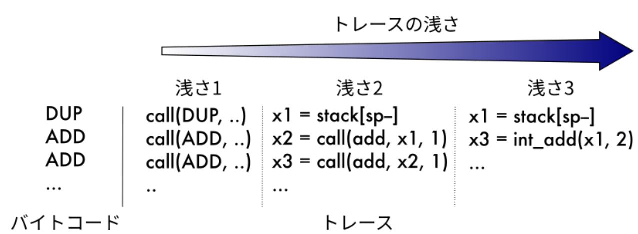

図 3. トレースの浅さ

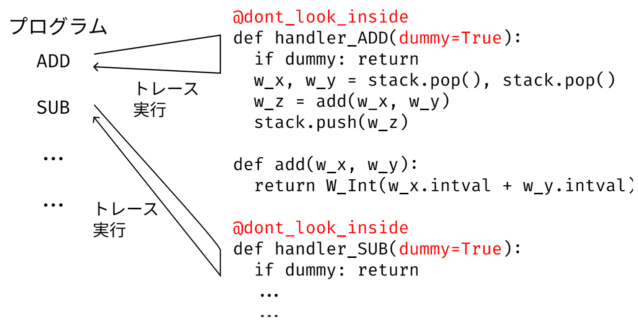

図 4. Shallow tracing の概観

Shallow tracing は、インタプリタ定義におけるイディオムと、コンパイラの小さな改変によって実現できる。Shallow tracing についての概観を図 4 に示す。図 4 では、各ハンドラ (handler_XXX という名前が付いた関数) のそれぞれに dont_look_inside というデコレータと dummy フラグがある。dont_look_inside デコーレタは、RPython が提供している命令で、dummy フラグの設定はインタプリタ実装者が行う。Shallow tracing を行う時、dummy フラグが真に設定される。すると、handler から即座に戻るための文がハンドラ本体の先頭に挿入されている。したがって、メソッド型コンパイル時にトレース実行時にハンドラの中身までを実行することなく、収集したい命令のみをトレースすることができる。

メソッド型スレッデッドコード生成をメタ実行履歴コンパイラで可能になったことにより、最適化レベルの異なる 2 つのコンパイルを組み合わせることが可能になった。実行履歴型コンパイラはプログラムのピークタイム性能の向上に寄与するのに対し、スレッデッドコード生成はプログラムの実行初期の性能を向上することが期待できる。

研究成果 A と本研究が提案した技術を RPython フレームワーク、そして Smalltalk 処理系である PySOM に実装し、実際にスレッデドコード生成を実行履歴型コンパイラと組み合わせることによってプログラムの初動性能が改善するのか性能評価を実施した。実行履歴型コンパイラのみを用いたプログラム実行と、スレッデッドコード生成と実行履歴型コンパイラを組み合わせたプログラム実行を比較した結果、後者は最大で 32%ほど初動性能が向上することが確かめられた。また、ピークターム性能については、最終的に両実行は同じ実行履歴型コンパイラを用いたプログラム生成・実行を行うので、ほぼ同じピークタイム性能が得られた。

研究成果 C「RPython コミッターである Carl Friedrich Bolz-Tereick 氏との共同研究」

研究成果 A・B は、RPython/PyPy コミッターである Carl Friedrich Bolz-Tereick 氏との共同研究によって実施された。ACT-X への採択を契機とした国際会議への積極的な参加がきっかけとなり、Carl Friedrich 氏とコンタクトが可能になった。コロナ禍ではあったものの、オンラインでのコミュニケーションを積極的に図った結果、研究を共同で推進することができるレベルへと信頼関係を構築することができ、論文発表へ至ることもできた。さらに、コロナ禍での行動制限が解除されつつあった 2022 年 6 月に、ACT-X 研究での研究成果で海外発表を行った。研究開始から約 1 年半経過していたが、初めて Carl Friedrich 氏と対面での交流を持つことができた。この機会によって、非同期・オンラインが主体の国際共同研究であっても何ら支障なく研究を遂行できている。

3．　今後の展開

「あらゆる言語の性能を向上させるための適応的な実行時コンパイルフレームワークの実現」

今後、2025 年度を目安に、ハイブリッド型コンパイルを発展させ、あらゆるプログラミング言語に対して適応可能な適応的コンパイル方式の実現に取り組む。適応的コンパイルとは、プログラムの実行状況に応じて最適なコンパイルを適用するための最適化手法である。従来、適応的コンパイル手法では、実装されている最適化手法を段階化し、状況に応じて使い分けるための戦略を決めていた。今後、本研究が目指す適応的コンパイルとは、最適化レベルの段階化と適用戦略を決めるだけでなく、コンパイル範囲も決定できるようにし、あらゆる言語で効果的に機能するための戦略も策定する。

現段階では、RPython フレームワーク上に 2 つの異なる最適化レベル・コンパイル範囲を持ったメタコンパイラが実現されている。適応的コンパイルの実現に向けて、まず 2023 年度では最適化レベルの高いメソッド型コンパイルを実現する。Shallow tracing におけるトレースの浅さを最も深いレベルにすることによって達成できるが、インタプリタやヒープの状態の巻き戻しが必要になる。この課題に対し、読み書きが行われたメモリ領域のみを巻き戻す仕組みをメタ実行履歴コンパイラにおいて実現することで解決できると考えている。具体的には、トレース時のメモリへの読み書きを逐一記録しておき、トレースが終了したら読み書きしたメモリ領域の中で競合のあるものを巻き戻す仕組みである。愚直に全てのメモリ領域をコピーする場合、読み書きが行われない領域までコピーすることになるため、空間的に無駄が大きい処理となってしまうが、読み書きが行われた部分のみを巻き戻すことができれば、効率の良い巻き戻し手法を実現できると考えている。

さらに、2024 年度においては、実用的なプログラミング言語処理系への適用を目標に本研究を発展させていく。社会実装を見据え、Python や Ruby などの言語処理系における実現が不可欠だと考えている。実世界で動くプログラムにおいても本研究が提案する技術がプログラミング言語処理系の性能向上に寄与するのか検証するためにも、実用的な処理系への適用を行っていきたい。

現状、ハイブリッド型コンパイル方式を RPython フレームワークで実装し、参照として Smalltalk

言語の基本機能を実装した処理系で実現している。RPython フレームワークによって実現されている処理系として、Python 処理系の PyPy や Ruby 処理系の Topaz などがある。これらに対して本研究が実現する技術を適用し、実世界のプログラムを動かすことができるようになれば、メソッド型・実行履歴型を融合したコンパイルを行うことの有用性を検証することのみならず、実用的な適応的コンパイル戦略の策定にも着手することができる。

4. 自己評価

研究目的の達成状況

当初掲げた予定をおおむね達成できたと考えている。当研究の目標は、汎言語で高性能なハイブリッド型実行時コンパイラの実現である。汎言語の達成のために、研究提案時は Truffle/Graal という Oracle Lab.によって開発されているメタコンパイラフレームワークの利用を想定していた。しかし、新型コロナウイルスの流行によって予定していた研究協力者との交流が困難になってしまったため、当研究室にノウハウがある RPython フレームワークを用いることになった。汎言語については、RPythonフレームワーク上に当研究成果を実現できたことによって、ある程度達成できたと考えている。実際にいくつかの言語で実現するケーススタディを行えれば実証できると考えている。

高性能の達成については、プログラムの実行初期の性能向上は達成できたと考えている。高度な最適化を行う実行履歴型コンパイルの他に、スレッデッドコードという形式のコードを生成する軽量メソッド型コンパイル手法を実現し、両者と組み合わせるコンパイル方式を実現したことによって達成できたと考えている。今後、さらなる高性能の達成のために、軽量メソッド型コンパイル手法を発展させ、高度な最適化を行うメソッド型コンパイルの実現を行っていく。

研究の進め方（研究実施体制及び研究費執行状況）

研究実施体制は問題なく組むことができた。研究の遂行に当たって助言を得ることができる研究者との共同研究を開始することができたのが大きかったと考えている。研究費執行状況については、新型コロナウイルスの流行によって海外出張が制限されてしまったことにより、当初予定していた出張がキャンセルされてしまった影響で、予定していた執行額を下回ってしまった。

研究成果の科学技術及び社会・経済への波及効果

ACT-X プログラムは、当研究者が提案する研究の遂行において非常に有益であった。その理由は 3 つある。1 つ目は、領域の中で多様な研究者との接点を持つことができた点である。従来の研究活動の場合は、研究室や学会といった場で出会う研究者との交流は可能であるものの、自らが関わっている研究分野の周辺にいる研究者との交流を持つことは難しかった。ACT-X に参加したことによって、第一線で活躍している幅広い研究者と議論する機会を年に 2・3 回以上いただくことができたのは、自分の研究を見つめなおすだけでなく、今後の分野横断的な共同研究に向けた礎を築くことができたと考えている。

２ つ目に、研究アドバイザーからの助言を定期的に頂くことができた点である。研究室や外部の共同研究者とのミーティングを重ねるだけでは得られないような、様々な視点からのコメントをもらえたことによって客観視できたことも有益であった。このような機会をキャリアの序盤で得られたことは何事にも替えがたいと感じている。

３つ目は、研究資金だけでなくRA 支援金も頂けたことである。生活するための十分な資金を頂けることによって、精神的に余裕をもって研究に専念できたと実感している。

５．　主な研究成果リスト

（1）代表的な論文（原著論文）発表

研究期間累積件数：2 件

1. Yusuke Izawa Hidehiko Masuhara, Carl Friedrich Bolz-Tereick. "Threaded Code Generation with a Meta-Tracing JIT Compiler." Journal of Object Technology. 2022. Volume 2. Page 1 – 11.

言語実現フレームワークとは、インタプリタ定義から実行時コンパイラやごみ集めなどが備わった仮想機械を生成する処理系である。一方、実行時間の短いプログラムに有効な軽量なコンパイルを行う仕組みが備わっていない。これは、言語実現フレームワークの仕組み上、インタプリタ以外のコンポーネントは自由な拡張が難しいからである。本研究は、インタプリタ定義を用いて実行履歴型言語実現フレームワークに軽量なコンパイルを行うスレッデッドコードコンパイラを新しく追加する。PyPy 処理系でスレッデッドコード生成の予備実験を行った結果、高度な最適化を行うコンパイラと比べて 60%ほどコンパイル時間を削減し、インタプリタ実行より 7%実行速度を向上することが確認できた。

2. Yusuke Izawa. "Supporting multi-scope and multi-level compilation in a meta-tracing just-in-time compiler framework." Ph.D. thesis. Tokyo Institute of Technology. 2023.

コンパイラは、人間にとって書きやすく読みやすい水準のソースコードから CPU などの機械にとって実行可能な水準まで変換を行う重要なプログラミング言語技術の一つである。本研究は、言語実現フレームワークというプログラミング言語処理系を生成可能な技術を用い、インタプリタ定義を用いてコンパイラの大幅な改変なしに様々なコンパイル手法を導く技術を探求する。具体的には、言語実現フレームワークが生成するコンパイラの振舞いを制御するための命令をインタプリタ定義に挿入することによって実現している。RPython というフレームワークをベースに実現を行い、実行履歴単位でのコンパイラにメソッド単位のコンパイル方式を付加するだけでなく、違った最適化レベルの共存まで行えることを実証した。これまでの研究では、様々な振舞いを実現するためには振舞いごとにコンパイラを一つずつ容易していたが、本研究によって言語実現フレームワークのエコシステム内であればインタプリタ定義を用いるだけで実現できるという点が独創的である。

（2）特許出願

なし。

（3）その他の成果（主要な学会発表、受賞、著作物、プレスリリース等）

1. Izawa, Yusuke, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, and Youyou Cong (July 13, 2021). Threaded Code Generation with a Meta-tracing JIT Compiler. The 16th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2021). Virtual. arXiv: 2106.12496v4.
2. Izawa, Yusuke, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick (Jan. 17, 2022b). Two-level Just-in-Time Compilation with One Interpreter and One Engine. In: The ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. PEPM 2022. Virtual. arXiv: 2201.09268.
3. Izawa, Yusuke and Hidehiko Masuhara (June 7, 2022). Taming an Interpreter for Threaded Code Generation with a Tracing JIT Compiler. The 17th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2022). Berlin, Germany.
4. Izawa, Yusuke, Hidehiko Masuhara, and Carl Friedrich Bolz-Tereick (Mar. 13, 2023). Interpreter Taming to Realize Multiple Compilations in a Meta-Tracing JIT Compiler Framework. The 7th MoreVMs workshop (MoreVMs’23). Tokyo, Japan.