セルフホスト型ランタイムによる WebAssembly インストルメンテーションの実現可能性検討

中田 裕貴　松原 克弥

クラウドコンピューティング基盤に欠かせない機能のひとつである VM マイグレーションの実現には，VM や VM 内で稼働するアプリケーションの実行状態を任意のタイミングで抽出・復元する技術が必要不可欠である．本研究では，エッジ・クラウドコンピューティング向けシステム基盤としての活用が注目されている WebAssembly を対象として，VM マイグレーションに最適なバイナリインストルメンテーション機能の実現手法として，セルフホスト型ランタイムに着目した．多種多様なランタイムが現存し，ネイティブコンパイル等の最適化技術が適用されることも多い WebAssembly アプリケーションの実行状態抽出には，ランタイム中立性と実行時性能を両立できるバイナリインストルメンテーションが求められる．本稿では，WebAssembly アプリケーションの実行状態を取得するためのインストルメンテーション機能を備えたセルフホスト型ランタイムについて，既存インストルメンテーション手法との比較や予備実験による評価により，その実現可能性を検討する．

1 はじめに

バイナリインストルメンテーション (Binary Instrumentation) は，アプリケーションの実行状態を観察するための技術である．変数やメモリの値，リソース使用量，各命令の実行回数などの統計情報を取得するコードや機能をアプリケーションに加えることで，バグの追跡やフローの可視化，プロファイリング，リアルタイム監視などを実現する．バイナリインストルメンテーションは，アプリケーション実行前にバイナリへインストルメントコードを静的に挿入する手法と，アプリケーション実行中にランタイム内で動的に挿入する手法がある．

著者らは，インストルメントコードの動的挿入によるバイナリインストルメンテーションが，仮想命令セットアーキテクチャである WebAssembly（以下，Wasm）[1] を用いたエッジコンピューティング環境における VM マイグレーションの実現に有用であると考えている．VM マイグレーションは，あるマシンで実行中のアプリケーションを実行状態と共に別マシンに移し，処理を途中から再開させる技術である．アプリケーション実行ランタイムでのインストルメントコードの動的挿入を用いて，アプリケーションの内部状態を取得と移行先ランタイムでの内部状態の復元を行うことで，VM マイグレーションを実現できる．対して，静的挿入は，インストルメントコードの呼び出しによってアプリケーション性能も低下する [2] だけではなく，実行状態の復元や特定の内部状態に関するインストルメンテーションが難しく，VM マイグレーションには適さない．エッジコンピューティングは，ユーザ近傍に存在するコンピューティングリソースを活用し，デバイス環境とエッジ環境，クラウド環境など多様な CPU アーキテクチャのコンピューティング環境が連携する分散処理環境を実現する [3] [4]．このような環境では，計算性能が要求される処理のクラウドへのオフローディングや，アプリケーションユーザの移動 [5] に合わせた処理のハンドオフに VM マイグレーションを活用することで，アプリケーションの可

Feasibility Study of WebAssembly Binary Instrumentation with Self-hosted Runtime

Yuki Nakata, 公立はこだて未来大学/さくらインターネット株式会社, Future University Hakodate/SAKURA internet Inc..

Katsuya Matsubara, 公立はこだて未来大学, Future University Hakodate.

用性を向上できる[6][7][8][9]．エッジコンピューティングは，各環境で使用される CPU アーキテクチャが異なるため[10]，あらゆるプログラミング言語で記述されたプログラムを Wasm バイトコードにコンパイルし，多様な CPU アーキテクチャ上でバイトコードを実行できる VM を作成する Wasm が有用である．また，Wasm ランタイムは，デバイスやクラウド環境など，それぞれに特化したランタイム実装が数多く存在するため，エッジコンピューティングにおける，アプリケーションのオフローディングやハンドオフをシームレスに実行できる[11]．

しかしながら，Wasm におけるインストルメントコードの動的挿入は，多様な Wasm ランタイムの存在やランタイムごとのアプリケーション実行方式の差異によって実現が難しい．Wasm は，バイトコードや仮想マシンの標準仕様があるが，仕様を実現する実装はランタイムによって多様である．そのため，複数のランタイムを用いた環境では，各ランタイムにバイナリインストルメンテーション機能を実装する必要がある．また，Wasm ランタイムは，バイトコードを逐次解釈するインタプリタ方式や，バイトコードを実行中にネイティブバイナリへコンパイルする JIT（Just-In-Time）方式，実行前にコンパイルする AOT（Ahead-Of-Time）方式など多様な実行方式が存在する．実行方式によってアプリケーションが使用する Wasm 仮想マシン内リソースや実メモリ・CPU などのリソース使用特性が変化するため，バイナリインストルメンテーション機構も実行方式を意識して実装しなければならない．

本研究は，セルフホスト型 Wasm ランタイムによるランタイム・実行方式に非依存な動的なバイナリインストルメンテーション機構を提案する．バイナリインストルメンテーション機構が実装された Wasm ランタイムを Wasm にセルフホストコンパイルし，あらゆる Wasm ランタイム上でセルフホスト Wasm ランタイムを用いてアプリケーションを実行する．セルフホスト型 Wasm ランタイムによって各ランタイムへのインストルメンテーション機構の実装とネイティブ環境における実行方式を意識する必要がなくなる．

本稿では，セルフホスト型 Wasm ランタイムによるランタイム・実行方式非依存なバイナリインストルメンテーションの有用性と実現可能性を明らかにするために，Wasm における既存のインストルメンテーション手法を整理し，それらの課題を明らかにする．そして，セルフホスト化した既存の Wasm ランタイムと既存の静的なインストルメントコード挿入手法で実行時性能を評価し，動的なインストルメントコード挿入手法とセルフホスト化した Wasm ランタイムの利点と課題を明らかにする．明らかにした課題をもとに，セルフホスト化を前提とした Wasm ランタイムの設計方針について議論する．

2 既存手法とその課題

Wasm バイトコードへのインストルメンテーション手法は，静的なインストルメントコード挿入手法，ランタイム内での動的なインストルメントコード挿入手法と，ランタイム外でのインストルメンテーション手法がある．これらの手法は，アプリケーション性能やインストルメンテーション機構実装の複雑さ，正確な内部状態の取得においてトレードオフがある．

2.1 静的なインストルメントコード挿入手法

Wasabi[12] は，Wasm バイトコード実行前にインストルメントコードを挿入することで，Wasm アプリケーション内状態の取得と複雑な解析を実現している．インストルメントコードは，命令単位や，関数呼び出し前・呼び出し後などあらゆる場所に挿入されるため，アプリケーションの状態を詳細に把握できる．バイトコードを改変して状態収集を実現するため，AOT や JIT などの実行方式の違いによる影響は受けない．Wasabi は，複雑な解析を実現するために，バイトコードに挿入されるインストルメントコードは解析処理を含んでおらず，JavaScript で記述された解析用関数を呼び出す．高水準言語の機能を用いて解析関数を記述することで，単純な命令数のカウントのみならず，コールグラフの作成，メモリアクセスの追跡などを容易に実現できる．また，プログラムの元の挙動に対する悪影響を軽減するために，挿入されたインストルメントコードはデータフローや制御フローを改変せず，挿入前のバイトコードと同等の振る

舞いを維持する．

しかしながら，Wasabi はあらゆる命令の前後にインストルメントコードを挿入し，状態取得用の JavaScript 関数呼び出しが大量に発生するため，アプリケーション性能が大幅に低下する．また，アプリケーションの規模や，取得したい内部状態の粒度に応じてバイトコードのサイズも肥大化する．

Observe SDK [13] は，Wasabi と同様にバイトコードへのインストルメントコードの挿入と，Wasm へコンパイルする前のアプリケーションコードにインストルメントコードを追加する手段を提供する．また，インストルメントコードと既存のモニタリングツールを連携させることで，内部状態を可視化し，アプリケーションの状態を詳細に把握できる．しかしながら，バイトコードへのインストルメントコードの挿入は Wasabi と同様の課題がある．また，コンパイル前のアプリケーションコードにインストルメントコードを追加することは，アプリケーション実装に使用可能なプログラミング言語の制約を発生させる．

2.2 ランタイム内での動的なインストルメントコード挿入手法

Wizard [2] は，Wasm ランタイム内で実行中のバイトコードにインストルメントコードを挿入することで，インストルメンテーションによるオーバヘッドを最小限に抑えている．Wasabi と異なり，収集したい状態に関わる命令の前後にインストルメントコードを挿入するのではなく，対象の命令をインストルメントコードに置き換える．インストルメントコードの挿入と抜去は動的に動的に実行され，インストルメントコードが挿入されていない場合は通常の Wasm バイトコード実行であり，実行性能に与えるオーバヘッドを大幅に削減できる．また，ランタイムにインストルメンテーション機構が実装されているため，ランタイムが管理しているスタックやメモリに関する状態にもアクセス可能であり，より多くの状態を収集できる．

しかしながら，ランタイムを拡張して実装されているため，多様な Wasm ランタイムを組み合わせた環境における状態収集には課題がある．Wizard の Wasm エンジンは，インタプリタ方式と JIT 方式をサポートしているため，実行方式ごとに異なるインストルメンテーション機構を実装している．Wizard と同様のアプローチを用いた場合，ランタイムとその実行方式を考慮したインストルメンテーション機構を複数実装する必要がある．

2.3 ランタイム外でのインストルメンテーション手法

Dtrace [14] は，アプリケーションや OS カーネルなど，システム全体のあらゆる状態をリアルタイムに収集できる．アプリケーションに対して動的にインストルメンテーションできるため，アプリケーションやバイナリの改変は不要である．Dtrace は，OS カーネルの機能として動作するため，アプリケーション性能に与えるオーバヘッドを最小限に抑えることもできる．

Dtrace は Wasm ランタイム外からインストルメンテーションを行うため，取得可能なアプリケーション内部状態に制約がある．Dtrace で取得できる内部状態の粒度は，プロセスやスレッド単位であるため，Wasm にコンパイルされたアプリケーションに対して Dtrace を使用した場合，内部状態には Wasm ランタイムが使用したものも含まれている．ランタイム外部から取得した内部状態からランタイム状態とアプリケーション状態を分類するには，ランタイムの設計や実装を完全に把握する必要があり，アプリケーションの内部状態のみを正確に取得するのは難しい．

3 提案手法

本研究は，セルフホスト型 Wasm ランタイムによるランタイム・実行方式非依存な動的なバイナリインストルメンテーション機構を提案する．図 1 のように，バイナリインストルメンテーション機構実装した Wasm ランタイムを Wasm にセルフホストコンパイルし，あらゆる Wasm ランタイム上でセルフホスト Wasm ランタイムを用いてアプリケーションを実行する．セルフホスト化するランタイムにのみインストルメンテーション機構を実装し，既存のあらゆる Wasm ランタイム上で実行すればよいため，各ランタイムの実装に合わせたインストルメンテーション機構が不要にする．JIT・AOT コンパイルはセル

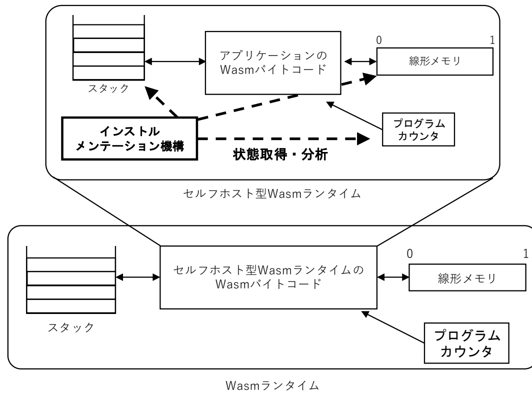

図 1　セルフホスト型 Wasm ランタイムによる 動的インストルメンテーション

フホスト型 Wasm ランタイムに対して適用されるため，セルフホスト型 Wasm ランタイム内のインストルメンテーション機構は，これらの実行方式の考慮が不要となる．また，既存の静的なインストルメントコードの挿入手法は，アプリケーションの大規模化に合わせてバイトコードのサイズも増大するが，セルフ Wasm ランタイムのバイトコードサイズは一定であるため，インストルメンテーションによるバイトコードサイズの増大を軽減できる．

4　セルフホスト型 Wasm ランタイムの予備評価

既存の Wasm ランタイムをセルフホスト化し，セルフホスト型 Wasm ランタイムがアプリケーション性能に与えるオーバヘッドを評価した．既存の Wasm ランタイム上でセルフホスト化したランタイムを用いてベンチマークプログラムを実行することで，セルフホスト型 Wasm ランタイムの性能を計測した．

セルフホスト化した Wasm ランタイムには Wasm3，セルフホスト化したランタイムを実行するランタイムには，Wasm3 と Wasmtime，WasmEdge を使用した．Wasm3 はインタプリタ方式のランタイムであり，セルフホストコンパイルに対応している．Wasmtime は，Wasm ランタイムにおける事実上のリファレンス実装であり，JIT コンパイルによって Wasm バイトコードを実行する．WasmEdge は，Wasm バイトコードをネイティブバイナリに AOT コンパイルできる．また，WasmEdge はインタプリタでも実行にも対応している．

表 1　実験に使用したマシンの仕様

OS	Ubuntu 22.04.4 LTS(Linux 5.15.0)
CPU	Intel(R) Xeon(R) Silver 4208 8Core 2.10GHz
メモリ	32 GB
ストレージ	SSD 480GBx2 (RAID1)

実験環境には，さくらの専用サーバ PHY RX2530 M5 8 コア 1CPU を使用した．使用したマシンの詳細な仕様は表 1 に示す．使用した Wasm ランタイムのバージョンは，Wasm3 は v0.5.0，Wastime は 22.0.0 (9f695788a 2024-06-04)，WasmEdge は 0.14.0-48-gd50c269c であった．

4.1　状態収集に関する性能評価

Wasm バイトコードへの静的なインストルメントコード挿入手法である Wasabi と，セルフホスト化した Wasm3 の双方でインストルメンテーションを実施しながらベンチマークプログラムを実行し，完了するまでの時間を計測した．内部状態の分析では，ベンチマークのメモリアクセス命令を解析し，使用したメモリのページサイズの合計を計算した．ベンチマークプログラムは，100 万項のライプニッツ級数を用いた円周率計算を使用した．Wasm3 は，このようなインストルメンテーション機構が実装されていないため，Wasm3 の内部状態を用いて同等の分析を行う機能を実装し，実行時間を計測した．

Wasabi は 2024 年 8 月 5 日時点での GitHub 上の最新版（コミットハッシュ：21a322b7faac）を使用した．Wasabi の実行には，JavaScript と Wasm の実行環境である Node.js 20.16.0 を用いた．

表 2 に示した実験結果から，セルフホスト化した Wasm ランタイムによるインストルメンテーションは，実行時性能に与えるオーバヘッドを既存の静的なインストルメントコード挿入手法よりも軽減できる可能性がある．Wasabi の実行に用いた Node.js の Wasm エンジンは，Wasm バイトコードをネイティブバイナリに JIT コンパイルして実行するが，セル

表 2　内部状態分析に関する性能

計測対象	実行にかかった時間 (Min.)
Wasabi	25.26
Wasm3 on Wasm3	23.09
Wasm3 on Wasmtime	7.22
Wasm3 on WasmEdge（AOT）	18.02
Wasm3 on WasmEdge（インタプリタ）	208.76

フホスト環境と比較して，ベンチマークプログラムの実行に多くの時間を必要とした．Wasabi の内部状態取得・分析関数は，JavaScript で記述され，それらの関数を呼び出すための命令がベンチマークプログラムの Wasm バイトコード内あらゆる箇所に挿入される．これらの関数呼び出しと Javascript で記述された処理の実行がベンチマークプログラムの実行性能に与えるオーバヘッドとなっていることが推測できる．対して，セルフホスト環境は，JIT コンパイルを用いた Wasmtime や AOT コンパイル時の WasmEdge だけではなく，インタプリタ方式の Wasm3 を用いた場合でも Wasabi より高速であった．セルフホスト化された Wasm3 内にインストルメンテーション機構を実装しているため，分析関数をバイトコードから複数回呼び出す処理を不要にできる．また，ランタイムがバイトコードを実行時の Wasm の仮想マシンの状態を内部状態として使用できるため，命令そのものの解析による内部状態取得も不要である．この結果から，セルフホスト型 Wasm ランタイムは，静的なインストルメントコード挿入手法と比較して軽量なインストルメンテーションが可能であるといえる．

セルフホスト化された Wasm ランタイムは，実際に実行する Wasm ランタイムの実行特性を維持したままインストルメンテーションが可能であることが判明した．JIT コンパイルする Wasmtime や AOT コンパイルを行う場合の WasmEdge は，セルフホスト Wasm3 がネイティブバイナリに変化するため，Wasm3 やインタプリタの WasmEdge でセルフホスト Wasm3 を実行する場合よりも性能が向上した．また，同じインタプリタ方式を採用した Wasm3 と WasmEdge で実行にかかる時間が大幅に変化したのは，Wasm3 のインタプリタはバイトコード命令のデコード処理に関する最適化がされた高速インタプリタ方式を採用しているためである．これらの結果から，セルフホスト型 Wasm ランタイムの性能は，実行するランタイムの実行特性に依存するといえる．

4.2　セルフホスト化による実行時オーバヘッド

セルフホスト化がアプリケーション性能に課すオーバヘッドを明らかにするために，Wasm3 と Wasmtime 上で直接ベンチマークプログラムを実行した場合と，これらのランタイム上でセルフホスト化した Wasm3 を用いてベンチマークプログラムを実行した場合で実行にかかる時間を評価した．ベンチマークプログラムには，プログラミング言語の性能を評価する The Computer Language24.06 Benchmarks Game に掲載されている，N=16000 でのマンデルブロ集合の計算とビットマップ形式での描画プログラム [15] を使用した．

表 3 に示した実行時間から，セルフホスト環境はアプリケーション実行にかかる時間が大幅に増加することが判明した．Wasm3 環境では実行にかかる時間が約 3641%，Wasmtime 環境では約 11160%増加した．Wasm ランタイムは，ベンチマークプログラムの Wasm 命令に加えて，セルフホスト化した Wasm3 の命令も解釈・実行する必要があり，実行しなければならない処理が増大するためであると推測している．また，Wasmtime で用いている JIT コンパイルが適用されるのは，ベンチマークプログラムではなく，セルフホスト化した Wasm3 の命令解釈・実行処理のみであるため，ネイティブコンパイルによる最適化がベンチマークプログラムに対して適用されなくなる．

Linux のパフォーマンス解析ツールである Perf [16] を用いて Wasm3 環境の総命令数と呼び出され

表 3　セルフホスト型 Wasm ランタイムによる実行時間の変化

計測対象	実行にかかった時間 (Sec.)
Wasm3	83.53
Wasm3 on Wasm3	3125.61
Wasmtime	14.02
Wasm3 on Wasmtime	1579.32

た関数を比較し，セルフホスト化によって増大したオーバヘッドを分析した．Wasm3 で直接ベンチマークプログラムを実行した際の CPU の総命令数は 400,849,582,525 回であり，Wasm3 上でセルフホスト Wasm3 を用いてベンチマークプログラムを実行すると 318,276,978,517,504 回であった．約 4459% 命令数が増加しており，ベンチマークプログラムの実行時間と同様な増加であった．

表 4 に示した，ベンチマークプログラム実行中のランタイム処理における上位 5 処理から，セルフホスト環境ではランタイムの処理内容が大きく変わることが判明した．Wasm3 で直接ベンチマークプログラムを実行した場合，単体で最も実行されたのはスタック処理（28.22%）であるが，2 位から 5 位までは算術演算処理であり，これらは全体の 57.31%を占めた．対して，Wasm3 上でセルフホスト化した Wasm3 を用いてベンチマークプログラムを実行した場合，スタック処理は 25.03%，線形メモリ処理が 15.51%，制御処理が 15.60%であった．直接ベンチマークプログラムを実行した場合と比べ，セルフホスト環境では算術演算処理よりも線形メモリと制御に関する処理が大幅に増加した．Wasm は，数値をスタックに格納し，Wasm が配列や構造体など，数値以外の値を線形メモリ領域に格納する．ランタイムでのスタック実装に配列を使用していた場合，セルフホスト化によってスタック操作も実際に実行されるランタイムでは線形メモリ操作に変更されてしまう．加えて，ランタイム内では解析したバイトコードや内部状態を構造体や配列に格納する．これらの操作によって線形メモリ処理が増加した．制御処理の増加は，セルフホスト化された Wasm3 のインタプリタ処理が要因であると推測している．

5　セルフホストに特化した Wasm ランタイムの検討

4 章の予備実験の結果から，セルフホスト化されることを前提とした Wasm ランタイムを設計する必要がある．既存の Wasm ランタイムのセルフホスト化によるインストルメンテーションは，既存の静的なインストルメントコード挿入手法と比較して軽量な内部状態の取得・解析が可能である一方，セルフホスト Wasm ランタイムを使用せずに直接 Wasm プログラムを実行した場合に比べ，大幅に性能が低下した．これは，既存の Wasm ランタイムがセルフホスト化されることを前提とされておらず，セルフホスト型 Wasm ランタイムと，それを実行するランタイムで二重のバイトコード解釈と実行が行われ，実行すべき命令が膨大になることが要因である．

セルフホストに特化した Wasm ランタイムの設計として，セルフホスト型 Wasm ランタイムとそれを実行する Wasm ランタイムで二重に実行される機能を削減する．インストルメンテーションに必要最低限なランタイム機能のみをセルフホスト型 Wasm ランタイムに実装することで，実行すべき命令数を削減し，セルフホスト化によるオーバヘッドの削減を目指す．

削減可能な機能として，Wasm ランタイムのサンドボックス機構がある．Wasm ランタイムは，バグや悪意のあるプログラムからユーザや実行環境を保護するために，サンドボックス内で Wasm プログラムを実行する．Wasm のサンドボックスは，実行前の検証と，Wasm プログラムが使用するスタックや線形メモリをソフトウェアによって分離する Software-based Fault Isolation(以下，SFI) によって実現される．実行前の検証は，実行前に未定義関数の呼び出しや型の一致，スタック操作などの命令の有効性確認などを確認する．Wasm における SFI は，線形メモリの境界値チェック，制御フローの整合性保証，スタックの保護などを実施する．これらの SFI 実装は，アプリケーションの Wasm 化した際におけるアプリケーション実行性能の大幅な低下の要因として，広く知られている [17] [18] [19] [20]．セルフホスト型 Wasm ランタイム

表 4　ベンチマーク実行中におけるランタイム処理の割合

順位	Wasm3	Wasm3 on Wasm3
1	op_SetSlot_f64 (28.22%)	op_CopySlot_32 (16.57%)
2	op_f64_Add_rs (22.15%)	op_i32_Load_i32_s (15.51%)
3	op_f64_Multiply_rs (21.53%)	op_CallIndirect(8.71%)
4	op_f64_Multiply_ss (7.22%)	op_SetSlot_i32 (8.36%)
5	op_f64_Subtract_rs (6.41%)	op_Entry (6.89%)

を用いた場合，セルフホスト環境とそれを実行するランタイムで SFI 実装が二重実行されるだけではなく，4.2 節で述べたように，メモリ操作に関する命令が増加しているため，線形メモリの境界値チェック処理が全体の処理時間の多くを占めることが推測できる．セルフホスト型 Wasm ランタイムにおける SFI 実装の削減は，セルフホスト環境における Wasm アプリケーションの実行性能向上に寄与できる．また，セルフホスト型 Wasm ランタイムを実行するランタイムは，境界値チェックなどの SFI をセルフホスト Wasm ランタイムに対して適用する．そのため，セルフホスト型 Wasm ランタイムとアプリケーションはサンドボックス内で実行されており，Wasm の安全性は維持できる．

6　おわりに

著者らは，ランタイム・実行方式に非依存なバイナリインストルメンテーションを実現するために，セルフホスト型 Wasm ランタイムを提案した．動的なインストルメントコードの挿入は，Wasm を用いたエッジコンピューティングにおける VM マイグレーションに有用であるが，多様な Wasm ランタイムの存在やランタイムごとのアプリケーション実行方式の差異によって実現が難しい．実験の結果，セルフホスト型 Wasm ランタイムによるバイナリインストルメンテーションは，既存の静的なインストルメントコード挿入手法より高速であるが，依然として通常の通常実行時より性能が大幅に低下することが判明した．そこで，二重で実行されるサンドボックス機構の削減によるセルフホストを前提とした Wasm ランタイムの方針を示し，議論した．

今後は，議論したセルフホスト特化型 Wasm ランタイムの実装を進め，有用性を評価する．また，Wasm バイトコードの命令解釈・実行処理においてもセルフホスト化を前提とした高速化手法の検討を進めたい．

参 考 文 献

[1] WebAssembly Community Group. Webassembly specification — webassembly 2.0 (draft 2024-08-11). `https://webassembly.github.io/spec/core/`, 2022. (Accessed on 08/12/2024).

[2] Ben L. Titzer, Elizabeth Gilbert, Bradley Wei Jie Teo, Yash Anand, Kazuyuki Takayama, and Heather Miller. Flexible non-intrusive dynamic instrumentation for webassembly. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, p. 398–415, New York, NY, USA, 2024. Association for Computing Machinery.

[3] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, Vol. 8, pp. 85714–85728, 2020.

[4] Hang Liu, Fahima Eldarrat, Hanen Alqahtani, Alex Reznik, Xavier de Foy, and Yanyong Zhang. Mobile edge cloud system: Architectures, challenges, and approaches. *IEEE Systems Journal*, Vol. 12, No. 3, pp. 2495–2508, 2018.

[5] Pengcheng Zhang, Yaling Zhang, Hai Dong, and Huiying Jin. Mobility and dependence-aware qos monitoring in mobile edge computing. *IEEE Transactions on Cloud Computing*, Vol. 9, No. 3, pp. 1143–1157, 2021.

[6] Keerthana Govindaraj and Alexander Artemenko. Container live migration for latency critical industrial applications on edge computing. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1, pp. 83–90, 2018.

[7] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing*, Vol. 18, No. 9, pp. 2020–2033, 2019.

[8] Wei Lu, Xianyu Meng, and Guanfei Guo. Fast service migration method based on virtual machine

technology for mec. *IEEE Internet of Things Journal*, Vol. 6, No. 3, pp. 4344–4354, 2019.

[9] Daigo Fujii, Katsuya Matsubara, and Yuki Nakata. Stateful vm migration among heterogeneous webassembly runtimes for efficient edge-cloud collaborations. In *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '24, p. 19–24, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. Computation offloading toward edge computing. *Proceedings of the IEEE*, Vol. 107, No. 8, pp. 1584–1607, 2019.

[11] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Webassembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, FRAME '22, p. 3–8, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, p. 1045–1058, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Dylibso, Inc. Observability in webassembly — dylibso developer resources. `https://dev.dylibso.com/docs/observe/overview`, 2024. (Accessed on 08/14/2024).

[14] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *2004 USENIX Annual Technical Conference (USENIX ATC 04)*, Boston, MA, June 2004. USENIX Association.

[15] Isaac Gouy. mandelbrot rust #4 program (benchmarks game). `https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/mandelbrot-rust-4.html`. (Accessed on 08/15/2024).

[16] The Linux Kernel Organization. Perf wiki. `https://perf.wiki.kernel.org/index.php/Main_Page`. (Accessed on 08/16/2024).

[17] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: near-zero-cost transitions for webassembly and sfi. *Proc. ACM Program. Lang.*, Vol. 6, No. POPL, jan 2022.

[18] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. Leaps and bounds: Analyzing webassembly ' s performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 256–268, 2022.

[19] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, p. 266–281, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 107–120, Renton, WA, July 2019. USENIX Association.