卒業論文

論文タイトルは４６文字で２行に収まります１２３４５６７８９０１２３４５６７８９０１２３４５６

公立はこだて未来大学
システム情報科学部　情報アーキテクチャ学科
情報システムコース　1099999

姓 名

指導教員　姓 名

提出日　20XX 年 1 月 XX 日

BA Thesis

Title in English within two lines: Lorem Ipsum Dolor Sit Amet, Consectetur Adipiscing Elit, Sed Do

by

Firstname Lastname

Information Systems Course, Department of Media Architecture
School of Systems Information Science, Future University Hakodate

Supervisor:　Firstname Lastname

Submitted on January XXth, 20XX

Abstract–

(Abstract should be about 150–200 words. Following is a sample text.) Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede. Mauris et orci. Aenean nec lorem. In porttitor. Donec laoreet nonummy augue. Suspendisse dui purus, scelerisque at, vulputate vitae, pretium mattis, nunc. Mauris eget neque at sem venenatis eleifend. Ut nonummy. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede. Mauris et orci. Aenean nec lorem. In porttitor. Donec laoreet nonummy augue. Suspendisse dui purus, scelerisque at, vulputate vitae, pretium mattis, nunc. Mauris eget neque at sem venenatis eleifend. Ut nonummy.

Keywords: Keyword1, Keyword2, Keyword3

概 要：

（概要は約 400 字とすること．以下はダミーテキスト）いろはにほへとちりぬるをわかよたれそつねならむういのおくやまけふこえてあさきゆめみしえひもせす．いろはにほへとちりぬるをわかよたれそつねならむういのおくやまけふこえてあさきゆめみしえひもせす．いろはにほへとちりぬるをわかよたれそつねならむういのおくやまけふこえてあさきゆめみしえひもせす．いろはにほへとちりぬるをわかよたれそつねならむういのおくやまけふこえてあさきゆめみしえひもせす．いろはにほへとちりぬるをわかよたれそつねならむういのおくやまけふこえてあさきゆめみしえひもせす．いろはにほへとちりぬるをわかよたれそつねならむういのおくやまけふこえてあさきゆめみしえひもせす．いろはにほへとちりぬるをわかよたれそつねならむういのおくやまけふこえてあさきゆめみしえひもせす．

キーワード： キーワード１, キーワード 2, キーワード 3

目次

第 1 章

はじめに

さまざまな産業向けやエンターテイメント関連のロボットシステム，自動車の自動運転技術や IoT システムのソフトウェア開発をサポートするフレームワークとして Robot Operating System（以下，ROS）の普及が増加している [1]．ROS のプログラミングモデルは，システムの各機能を独立したプログラムモジュール（ノード）として設計することにより，汎用性と再利用性を向上させて，各機能モジュール間のデータ交換を規定することで，効率的かつ柔軟なシステム構築を可能にしている．たとえば，カメラを操作して周囲の環境を撮影するノード，画像からオブジェクトを識別するノード，オブジェクトのデータをもとに動作制御を実行するノードを連携させることで，自動運転車の基本機能の一部を容易に実装できる．

ROS のプログラミングモデルは，ロボット/IoT とクラウドが協力する分散型システムにおいても有効である．ロボットシステムのソフトウェア処理は，外界の情報を取得する「センサー」，取得した情報を処理する「知能・制御系」，実際に動作するモーターなどの「動力系」の 3 要素に分類できる [2]．クラウドロボティクス [3] において，主に知能・制御系のノードを高い計算能力を持つクラウドに優先して配置することで，高度な知能・制御処理の実現を促進できる．さらに，ロボットが取得した情報や状態などをクラウドに集約・保存することで，複数のロボット間での情報共有と利用を容易にする．一方で，現行の ROS 実装では，各ノードの配置をシステム起動時に静的に設定する必要があり，クラウドとロボット間の最適なノード配置を事前に設計する必要がある．しかし，実際の環境で動作するロボットは，ネットワークの状況やバッテリー残量の変動など，システム運用前に予測することが困難な状況変化に対応する必要があり，設定したクラウドとロボット間のノード配置が最適でなくなる可能性がある．このような状況変化への対応として，ノードを動的に再配置するライブマイグレーション技術があるが，多くの場合でクラウドとロボット間の CPU アーキテクチャが異なり，命令セットがそれぞれ違うため，実行中のノードをシステム運用中にマイグレーションすることは技術的に困難である．

菅らは，WebAssembly（以下，Wasm）を用いることで，クラウドとロボット間での実行状態を含む稼働中ノードの動的なマイグレーションする手法を実現した [4]．Wasm とは

Web 上で高速にプログラムを実行するために設計された仮想命令セットアーキテクチャのことで，1 つのバイナリが複数のアーキテクチャで動作するため，異種デバイス間でのマイグレーションに適しているといえる．課題として，ROS 2 を Wasm 化したことでライブマイグレーション後のファイルサイズのオーバーヘッドが増大し，ノードの実行時間が大幅に増えてしまう問題が残った．柿本ら [5] は，組込みデバイス向けの軽量な ROS 2 ランタイム実装である mROS 2-POSIX を採用し，ROS ランタイムの Wasm 化にともなうオーバヘッド増加に対処した．しかし，採用された mROS 2-POSIX 上で指定されたメッセージを往復させるシンプルなアプリケーション上でしか評価実験はされていない [6]．そのため，ライブマイグレーション後のオーバーヘッド増加を解決するロボットソフトウェア基盤として，アプリケーションが複雑化した場合の動作が明らかでない．

本研究では，クラウドとロボット間での実行状態を含む稼働中ノードの動的マイグレーションの実現に向けた mROS 2-POSIX の性能評価を行い，mROS 2 が ROS 2 と比べて動的配置機構ロボットソフトウェア基盤としてどの点で優位性があるのか明らかにすることを目指す．

第 2 章

mROS 2-POSIX

第 3 章

提案手法

異種ランタイム間ライブマイグレーションを実現するためには，ランタイム内部構造の実装差異の吸収と，実行環境に依存しない内部状態表現が必要である．**??**章で述べたように，Wasm は仕様によってインスタンスやスタック，プログラムカウンタなどを定義しているが，すべてのランタイムが同一の手法でこれらを実装するとは限らない．例えば，WasmEdge は制御スタックが持つジャンプ先のラベルの情報などを，検証フェーズで前計算しておくことで，制御スタックを省略している．WAMR は検証フェーズを省略している．また，プログラムカウンタなどのアドレスによって表現する状態は，絶対アドレスの場合，他のマシンやプロセスにライブマイグレーションすることで，同じ内容を参照できなくなる．

本研究では，実行環境非依存な状態表現への変換と状態表現形式のランタイム間変換を行う．図 1 に本手法のライブマイグレーション機構における，保存・復元フローを示す．本提案手法は，実行状態の保存・復元機能からなり，実行状態の保存時に状態を変換する．

実行状態保存機能は，メモリインスタンス，グローバルインスタンス，スタック，プログラムカウンタの状態を変換し，保存する．インスタンスのうち，メモリとグローバル以外の要素は実行中に内部状態が変化しないため，保存対象外となる．絶対アドレスなどの実行環境に依存した実行状態は，オフセットによる相対アドレスなど，実行環境に非依存な状態に変換する．実行環境非依存な状態への変換後，マイグレーション先のランタイムに合わせた状態表現ファイルを生成し，保存する．

実行状態復元機能は，実行状態の初期化後に実行される．実行状態保存機能によって作成された実行状態ファイルをロードし，ファイルに記述された情報をもとに実行状態を復元する．実行フェーズで復元することで，実行中に状態が変化するデータのみを保存することを可能にする．

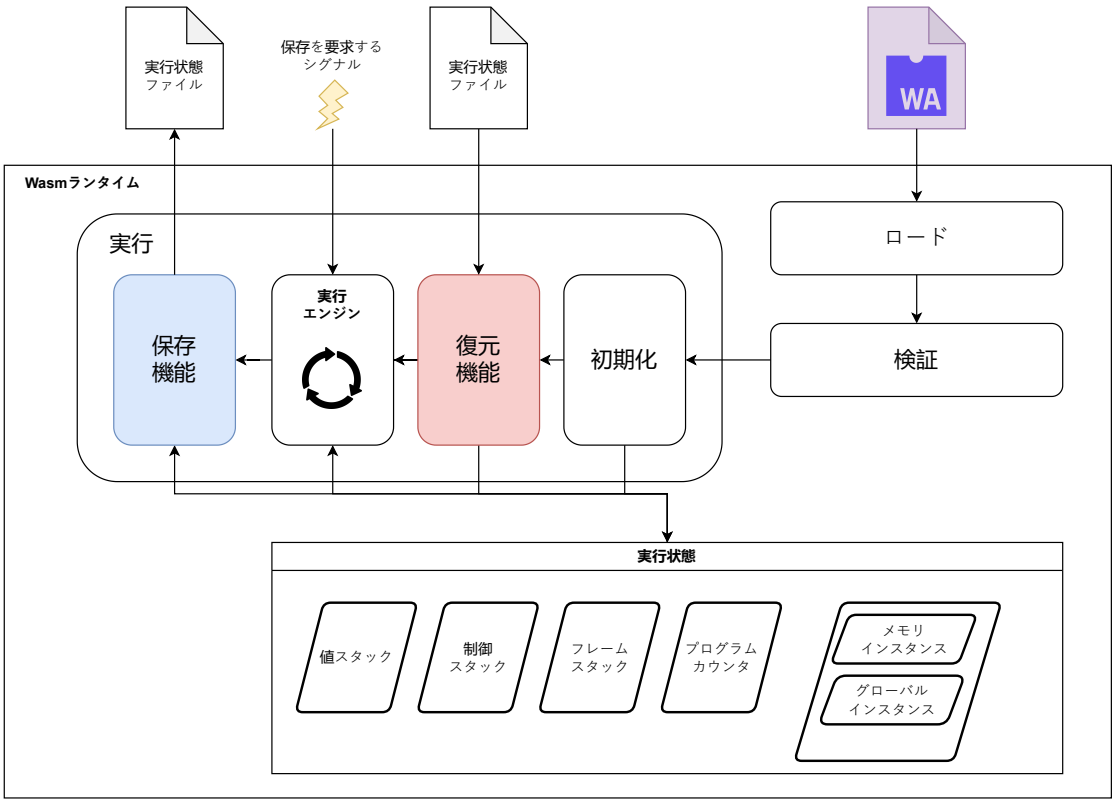

図 3.1　内部状態の保存・復元フロー

第 4 章

実装

マイグレーション対象の状態であるメモリ・グローバルインスタンスとプログラムカウンタ，スタックに対する，実行環境非依存な状態表現への変換と状態表現形式のランタイム間変換機能を WasmEdge と WAMR に実装した．

4.1 メモリ・グローバルインスタンス

WasmEdge と WAMR では，これらのインスタンスの表現方法が同一であるため，ランタイム間で変換せずにマイグレーションできる．メモリインスタンスは，現在のページ数と保存されたメモリのデータであるバイト列から構成される．グローバルインスタンスは，Wasm の値，つまり I32，F32，I64，F64，V128 のいずれかの型の値を持つ．これらの値を保存復元表現として使用した．

4.2 プログラムカウンタ

WasmEdge と WAMR では，プログラムカウンタの表現形式が異なる．WasmEdge は，Wasm 命令を Instruction クラスで表現し，インスタンス化した Instruction のメモリ連続な列によって命令群を表現する．Instruction クラスは，命令の引数やジャンプ先アドレス，ジャンプ後のスタックの位置など，命令に紐付いた様々な情報を持つ．プログラムカウンタは，Instruction 列における，次に実行する Instruction のポインタであり，プログラムカウンタを進めると，次の Instruction を参照する．WAMR は，プログラムカウンタを 1 つ進めると次の命令もしくは引数になる値のどちらかを参照する．

WasmEdge と WAMR は，どちらもプログラムカウンタに絶対アドレスを使用するため，相対アドレスを用いた実行環境非依存な状態表現への変換機能を実装した．WasmEdge に対する変換機能の実装では，関数インデックスとその関数の先頭の Instruction アドレスからのオフセットを使用した．WasmEdge に対する変換機能の実装では，プログラムカウンタ

を関数インデックスと関数先頭からのオフセットを使用した．

プログラムカウンタの構造の違いから，WasmEdge とは異なるオフセットを取るため，マイグレーション先のランタイムに合わせた相互変換機能を実装した．WasmEdge から WAMR へのオフセットの変換では，WasmEdge は，Instruction クラス内に Wasm モジュールにおけるオフセットを保持しており，これは WAMR におけるオフセットと同等であるため，これを使用した．WAMR から WasmEdge へのオフセットの変換では，関数の先頭アドレスから対象の命令アドレスまでの間に存在する命令数を計算した．

4.3 スタック

値スタックとフレームスタックは，WasmEdge と WAMR 双方に存在するが，制御スタックは WAMR にのみ存在する．また，WasmEdge と WAMR 間で値スタックとフレームスタックの表現形式が異なるため，全てのスタックにおいてマイグレーション先のランタイムに合わせた相互変換機能が必要である．

制御スタックの相互変換を実現するために，制御スタックを WasmEdge で再現する機能を実装した．制御スタックは，ジャンプ先のアドレスとジャンプ後戻すスタックの位置を管理している．WAMR は，ブロック命令を読むと，制御スタックに push し，ブロックの終わりを示す end 命令を読むと pop する．対して，WasmEdge は，検証フェーズで，ジャンプ命令とブロック命令に対するジャンプ先を事前に計算しておくことで，制御スタックを省略している．WasmEdge で制御スタックを保存するために，必要な分のコードを走査することで，WAMR の制御スタックへの操作を WasmEdge で再現可能にした．ジャンプ先のアドレスは命令の位置を指すポインタであるため，プログラムカウンタと同様の表現形式と変換を使用した．ジャンプ後に戻すスタックの位置は，スタックの要素数で表現されているため，そのままの値を使用した．

値スタックにおける相互変換を実現するために，型スタックを実装した．WasmEdge と WAMR の値スタックは，どちらも複数の Wasm の値から構成されるリストである．しかしながら，WasmEdge は，それぞれの値を値スタックの要素として扱うのに対し，WAMR では，値の 4byte ごとに値スタックの要素として扱うため，WAMR の値スタックから値の境界を把握することができない．我々は，値スタックに対応する値の型情報を管理する型スタックを実装することで，値の境界を把握可能にした．値スタックの操作に合わせて，型スタックに値の型サイズの情報を管理することで，型情報から 1 つの値を 4byte ごとに分解，もしくは 4byte ごとの要素を 1 つの値に復元し，値スタックの単位系の総合変換を実現した．

フレームスタックにおける相互変換を実現するために，それぞれのランタイムにおけるスタック内要素の変換を実装した．WasmEdge のフレームスタックは，リターンアドレス，呼び出し側関数の値スタックポインタ，呼び出し先関数の引数の個数，呼び出し先関数の返り値の個数を一つの要素としたリストで表現される．対して，WAMR のフレームスタックは，

リターンアドレス，その関数の値スタックポインタ，制御スタックポインタを一つの要素としたリストで表現される．WasmEdge と WAMR のフレームスタックに共通している要素は，リターンアドレスと値スタックのポインタである．WasmEdge のリターンアドレスは，Instruction のポインタ，WAMR のリターンアドレスは，命令ポインタであるため，プログラムカウンタと同様の変換を実装した．WasmEdge における値スタックポインタは，値スタックのベースアドレスからの相対位置であるが，WAMR の値スタックポインタは絶対アドレスである．そのため，WAMR における値スタックポインタを先頭アドレスからのオフセットに変換することで，相互変換を可能にした．WasmEdge のみ持つ要素は，呼び出し先関数の引数の個数と，返り値の個数である．WAMR は，関数インスタンスがこれらの引数の個数と返り値の個数の情報を管理しているため，関数インスタンスから引数の個数と返り値の個数を保存することで，引数の個数と返り値の個数を WasmEdge で復元することができる．WAMR のみ持つ要素は，制御スタックポインタである．WasmEdge は制御スタックを持たないが，本実装では，制御スタックを再現可能にしたため，制御スタックの先頭からのオフセットを使用した．

第 5 章

評価

本研究では，エッジデバイスとクラウド同等の性能のマシンで実験を行った．エッジデバイスの仕様を表**??**に示す．クラウドの仕様を表**??**に示す．

5.1 異種ランタイム間の性能比較

WasmEdge と WAMR の性能に特徴があるのかを調べるため，WasmEdge と WAMR で，SQLite のベンチマークを計測する sqlite-bench を使用した．評価する項目は，実行時間，メモリ使用量，sqlite-bench が計測する SQLite の各項目の実行速度である sqlite-bench は，SQLite の次の項目の実行速度について計測した．

- fillseq: 非同期モードでシーケンシャルキー順に N 個の値を書き込んだときの実行速度
- fillseqsync: 同期モードでシーケンシャルキー順に N/100 個の値を書き込んだときの実行速度
- fillseqbatch: 非同期モードで N 個の値を逐次キー順にバッチ書き込んだときの実行速度
- fillrandom: 非同期モードでランダムなキー順で N 個の値を書き込んだときの実行速度
- fillrandsync: 同期モードでランダムなキー順で N/100 個の値を書き込んだときの実行速度
- fillrandbatch: 非同期モードでランダムなキー順で N 個の値をバッチ書き込んだときの実行速度
- overwrite: 非同期モードでランダムなキー順で N 個の値を上書きしたときの実行速度
- fillrand100K: 非同期モードでランダムなキー順で N/1000 個の 100K 個の値を書き込んだときの実行速度
- fillseq100K: 非同期モードで，N/1000 個の 100K 値を順次読み込んだときの実行速度

表 5.1　実験に用いたエッジデバイスの仕様

OS	Ubuntu 22.04
Linux kernel	6.5.6-76060506-generic
CPU	11th Gen Intel i5-1135G7 (8) @ 4.200GHz
RAM	16 GB

表 5.2　実験に用いたクラウドの仕様

OS	Ubuntu 22.04
Linux kernel	5.15.0-86-generic
CPU	Intel Xeon Silver 4208 (16) @ 3.200GHz
RAM	32 GB

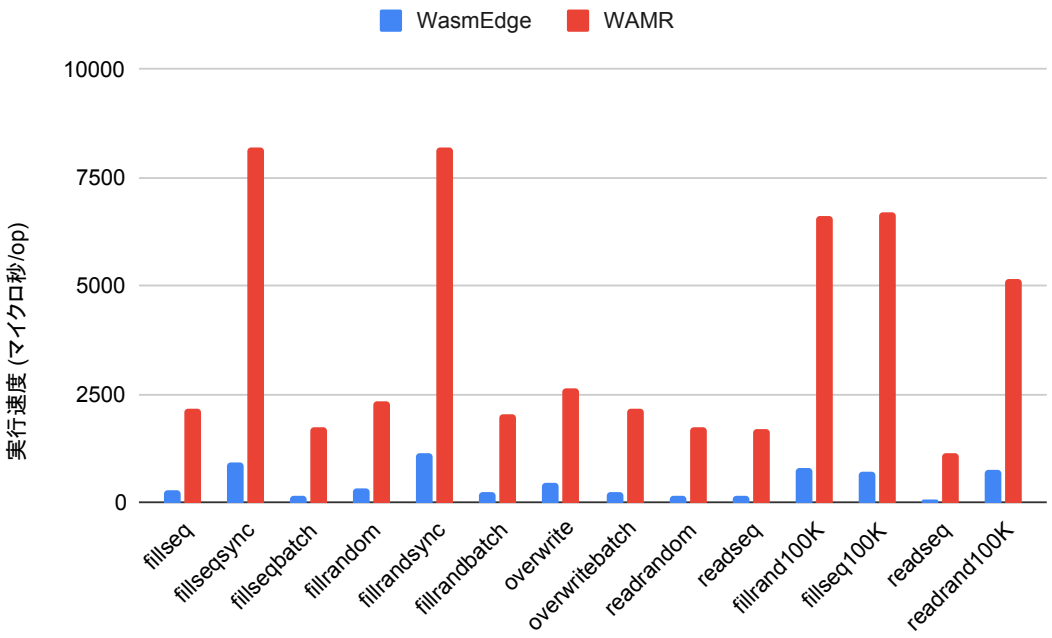

図 5.1　sqlite-bench の各項目の実行速度 (マイクロ秒/op)

- readseq: N 回連続して読み込む込んだときの実行速度
- readrandom: ランダムに N 回読み込んだときの実行速度
- readrand100K: 非同期モードで N/1000 個の 100K 値を順次読み込んだときの実行速度

本研究は，sqlite-bench を Wasm 化し，表**??**で示したマシン上で，WasmEdge と WAMR

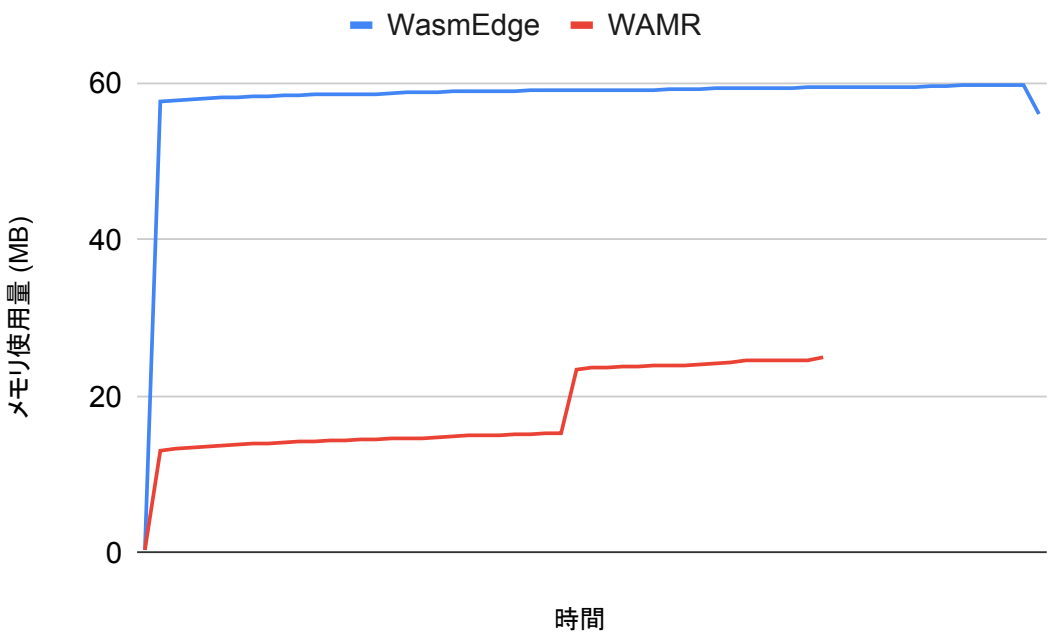

図 5.2　sqlite-bench 実行時の物理メモリ使用量

を用いて実行した．sqlite-bench のエントリ数を 10, 100, 1000, 10000, 100000 に変化させた．計測した値は，実行時間，実行中の物理メモリ使用量 (RSS)，sqlite-bench で計測した値である．実行時間は time コマンドを利用し，sqlite-bench を起動してから，終了するまでの実時間を計測した．実行中の物理メモリ使用量 (RSS) は，Python の psutil ライブラリを利用して，計測した．

図**??**は，sqlite-bench が計測する各項目の実行速度である．グラフを見ると，WasmEdge で動作させた方がすべての項目について，WAMR よりも実行速度が速い．例えば，fillseq は，WasmEdge で動作させると 1 エントリあたり 328.898 マイクロ秒であるが，WAMR で動作させると 1 エントリあたり 2868.891 マイクロ秒であった．これは，およそ 8.7 倍の実行速度の差がある．

図**??**は，sqlite-bench をエントリ数 100 で実行したときのメモリ使用量である．WAMR のメモリ使用量はおよそ 20MB であるが，WasmEdge のメモリ使用量はおよそ 60MB である．WAMR で動作させた方が，WasmEdge で動作させるよりも物理メモリ使用量が少ないことがわかる．

図**??**は，sqlite-bench の実行時間である．エントリ数が 1000 までは WAMR の方が実行時間が短いが，10000 を超えると急激に長くなっている．エントリ数が小さい時，実行速度が速い WasmEdge が WAMR よりも実行時間が長くなっている理由としては，WasmEdge が WAMR よりも起動時間が長いことが考えられる．

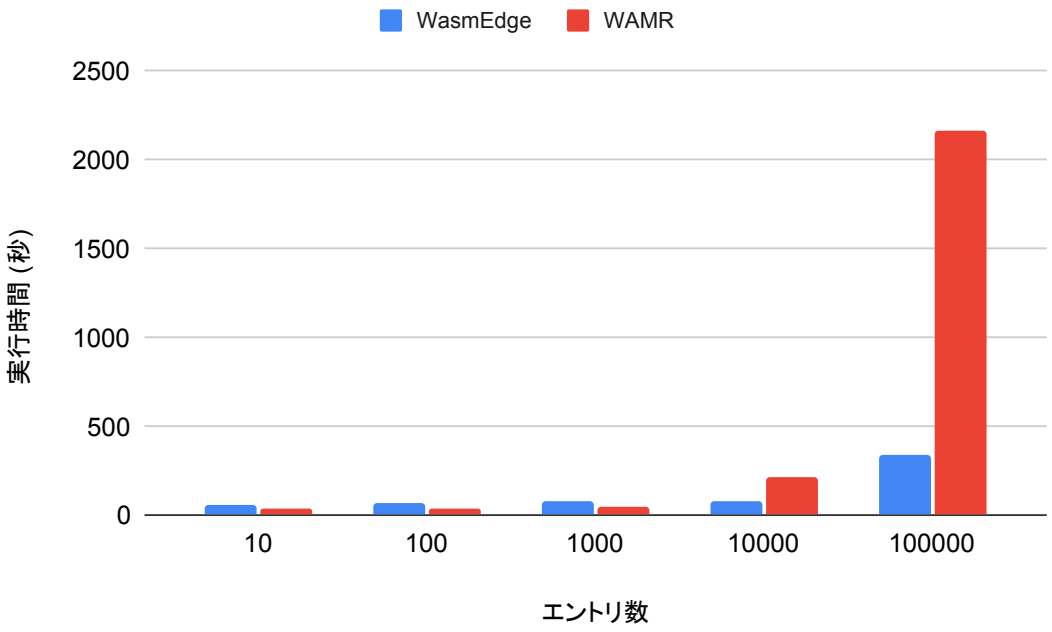

図 5.3　sqlite-bench のエントリ数ごとの実行時間

これらの結果から，WasmEdge は起動時間が遅く，実行速度が速く，メモリ使用量が大きい．一方，WAMR は起動時間が速く，実行速度が遅く，メモリ使用量が小さい．このように，ランタイムごとに特性が異なり，トレードオフの関係になっている．したがって，環境や要求に合わせて，異種 Wasm ランタイム間でマイグレーションをすることは，性能や可用性の向上が期待できる．

5.2　ライブマイグレーション性能

本研究で実装した WasmEdge，WAMR の Wasm ライブマイグレーションの性能と，既存のプロセスマイグレーション技術の CRIU のライブマイグレーションの性能を比較した．評価項目は，保存時間，復元時間，実行状態を復元するためのファイルサイズの合計である．表**??**のマシンで，各実験を 30 回繰り返し，各項目の平均を求めた．使用したアプリケーションは，n-body というベンチマークプログラムである．n-body のソースコードは，The Computer Language Benchmarks Game のものを使用し，wask-sdk でコンパイルした．

図**??**に，WasmEdge，WAMR，CRIU それぞれの保存にかかる時間，復元にかかる時間 (積み立て棒グラフ) に示す．また，図**??**に，WasmEdge，WAMR，CRIU それぞれの復元に必要なファイルのサイズ (積み立てグラフ) を示す．

復元時間，ファイルサイズいずれも，CRIU マイグレーションに比べ，Wasm マイグレー

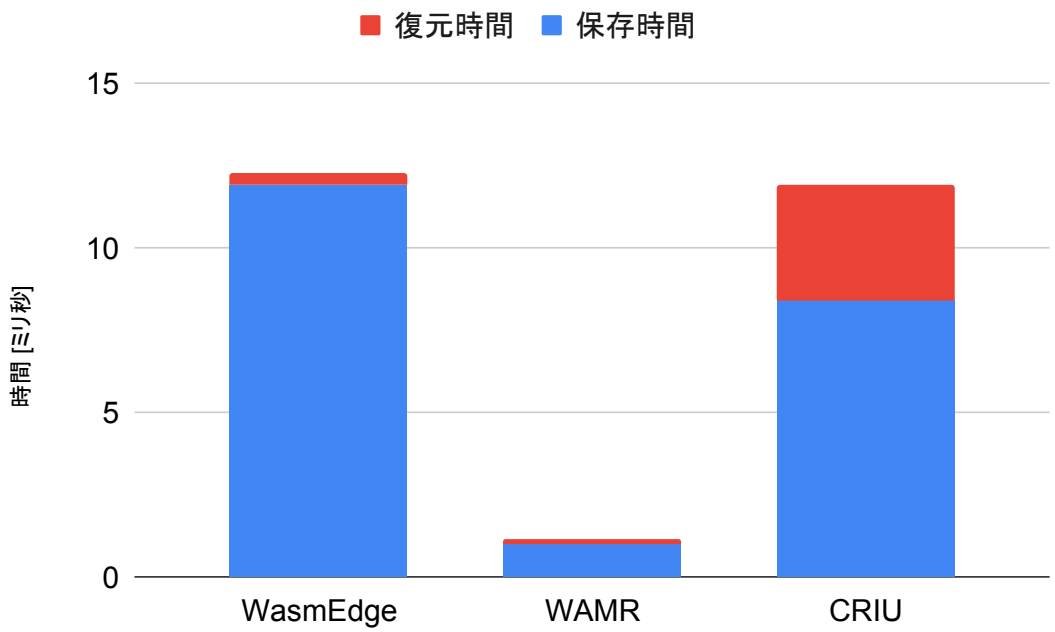

図 5.4　マイグレーションの保存・復元にかかる時間 (ミリ秒)

ションの方が性能が優れていた．保存時間は，CRIU と比較して，WasmEdge が約 1.4 倍遅く，WAMR が約 8.3 倍速かった．復元時間は，CRIU と比較して，WasmEdge が約 10 倍，WAMR が約 20 倍速かった．ファイルサイズは，CRIU と比較して，WasmEdge と WAMR ともに約 3.8 倍小さかった．

エッジコンピューティングは，データセンターに比べ通信帯域が大きくないため，ファイルサイズが小さい Wasm マイグレーションは，マイグレーション時に少ないデータ転送量で済む．また，Wasm マイグレーションは，保存・復元時間が小さいため，ダウンタイムが短い．さらに，Wasm マイグレーションは CRIU マイグレーションに比べ，多様なプラットフォーム間でマイグレーション可能である．これらの 3 つの点に関しては，エッジコンピューティングでは，Wasm マイグレーションの方が，CRIU よりも適していると言える．

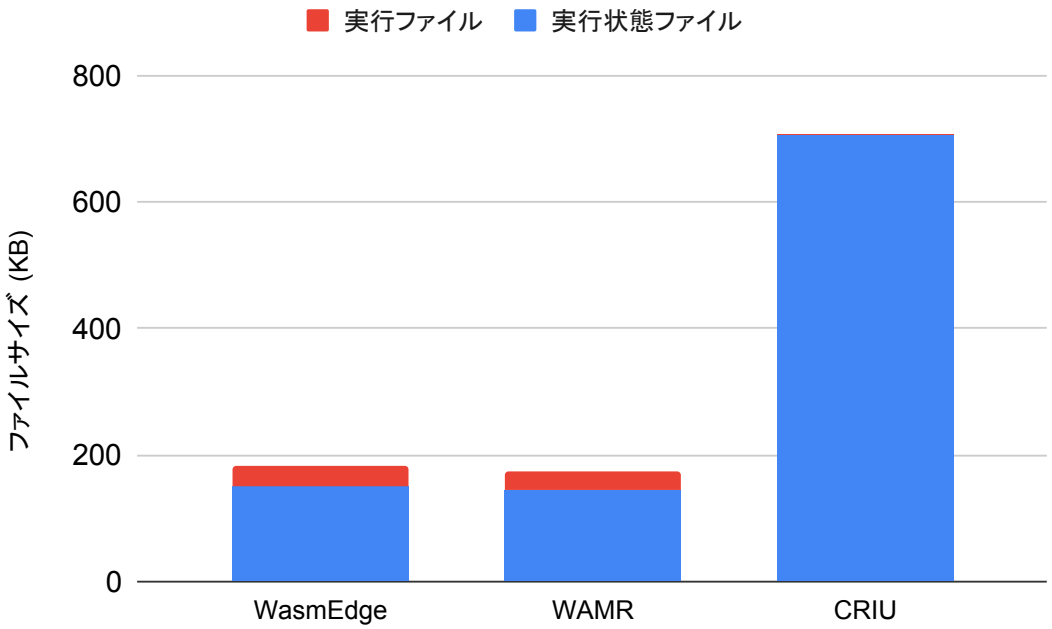

図 5.5　マイグレーションの実行状態のファイルサイズ (キロバイト)

第 6 章

関連研究

Nomad[?] は，Wasm3 という既存のランタイムにライブマイグレーション機構を追加することで，異種 OS や異種 ISA アーキテクチャ間でのライブマイグレーションを可能を実現している．Wasm ランタイム上のアプリケーションを中断せずに，移動させるという点は類似している．Nomad では，移動先のマシンの特性に合わせて，そのリソースや特性を有効に利用できるランタイムを選択することができないが，本研究の提案手法では，異なる特性を持つランタイム間でライブマイグレーションが可能であるため，移動先のマシンの特性に合わせてランタイムを選択でき，特性やリソースを有効に使うことができる．特にエッジコンピューティングにおける，クラウド・エッジ・デバイスのような，特性や性能が異なるマシンが混在する環境において有効である．

Edgedancer[?] は，エッジコンピューティングを対象に，隔離実行環境である Trusted Execution Environment (TEE) 上で Wasm モジュールを実行し，エッジの TEE 間でライブマイグレーションを実現している．エッジコンピューティングを対象にしている点は，本研究と類似している．しかしながら，Edgedancer は，マイグレーションによるモバイルユーザに対するサービスの提供と TEE によるサービスの機密性の維持が主目的であり，本研究と目的が異なる．

Faasm[?] は，Wasm モジュールの実行状態に関するスナップショットを作成できる．しかしながら，主な用途はアプリケーション起動の高速化のためであり，スナップショットを作成するタイミングは，アプリケーション初期化終了時のみである．そのため，本研究のようなアプリケーションの実行途中でのライブマイグレーションでは，目的が異なるため応用できない．

Mobile Web Worker[?] は，HTML5 におけるマルチスレッドを実現する Web Worker を拡張することで，モバイルアプリの処理の一部をクラウドや別モバイルデバイスにオフロード・ライブマイグレーションできる．特性の異なるコンピューティングリソースを組み合わせるという考え方は本研究と類似している．しかしながら，対応する実行環境が Web ブラウザなどに限られてしまうため，実行可能なアプリケーションに制約がある．

第 7 章

おわりに

本研究では，WasmEdge と WAMR 間での，異種ランタイム間ライブマイグレーションの手法について提案した．Wasm の仕様からランタイム構造や実行状態を整理し，実行状態の保存機構，復元機構を作成した．また，ランタイムごとの内部構造が異なる部分は，変換機構を実現することで解決した．

本手法の課題として，シグナルを受け取ってから，命令が終了して保存処理が始まるまでにタイムラグがあることである．とくにホスト関数呼び出しは，ユーザがホスト関数として任意の関数を定義できるため，非常に長い時間のタイムラグが発生する可能性がある．これを解決するために，Wasm 外の実行状態のマイグレーションの実現を行い，ホスト関数実行中でのマイグレーションを実現を目指す．

今後の課題では，より多くの要求に対応するために，すべての Wasm ランタイムで，ライブマイグレーション機構を実現する手法を検討する．現実装では，ランタイムの実装に手を加えることで，ライブマイグレーション機構を実現したが，すべてのランタイムに対応させるには，この手法は現実的に困難である．したがって，ランタイムの実装に依存しないような手法で，すべてのランタイム間でのライブマイグレーション機構の実現を目指す．

謝辞

謝辞を記入する.

参考文献

[1] Reference1
[2] Reference2