WebAssemblyにおける必要メモリ量の区間推定を用いたインスタンスの再起動によるメモリ使用量の削減

中川 翔太[1]　串田 高幸[1]

概要：Web ブラウザ上で計算を高速に行うために登場した WebAssembly は，ブラウザ外でも利用され始めている．WebAssembly のインスタンスに割り当てられるメモリは，拡張可能であるが縮小はできない性質がある．そのため，WebAssembly を長期間運用をする際に，1 度利用した最大量のメモリを占有し続ける．また，プログラムにメモリリークが含まれていた場合，メモリの割り当て量が拡大し続け，動作環境のメモリを使い果たしてしまう．これによってプロセスが強制終了させられると，その上で動作する Web サービスも停止する．本研究では，インスタンスの動的な再起動によりメモリの割り当て量を削減する手法を提案する．再起動は，区間推定を用いて適切なメモリの割り当て量を推定し，区間の最大値を越えたインスタンスに対して実行する．これによって，インスタンスの再起動の回数を最小限にすると同時に，ホストマシンのメモリ使用量の削減を図る．評価では，実行環境のメモリ使用量とインスタンスの稼働時間について，提案手法の有無で比較をする．

1. はじめに

背景

WebAssembly は，Web ブラウザ上でネイティブ水準の速度で動作する低レベルコードであり，コンパクトな表現，効率的な検証とコンパイル，オーバーヘッドのない安全な実行を提供する [1]．WebAssembly は，ポータビリティ，セキュリティに優れるという特徴から，ブラウザ外でも利用され始めている．例えば，ブラウザ外のシステムとのインタフェースの標準化を図る WebAssembly System Interface (WASI)[*1] の策定が進められている．活用事例として，WebAssembly for Proxies[*2] では，L4/L7 プロキシのプラグインとして WebAssembly を利用している．また，Krustlet[*3] では，WebAssembly のインスタンスを Kubernetes の Pod として，ネイティブ実行させている．Fastly 社のプロダクトの 1 つである Compute@Edge[*4]では，ネットワークのエッジでのリクエストの処理に WebAssembly を活用している．

WebAssembly では，1 つの WebAssembly バイナリ (.wasm) あるいは WebAssembly テキストフォーマット (.wat) を 1 つのモジュールとして扱う．WebaAsembly は，モジュールをランタイム上に展開して実行する形式である．また，モジュールが展開されステートフルで実行可能なオブジェクトをインスタンスと呼ぶ．

WebAssembly は線形メモリモデルを採用している．線形メモリモデルでは，メモリはプログラムから 1 つの連続したアドレス空間として見える [2]．各モジュールはメモリ空間を 1 つ定義できる．モジュールからロードされたインスタンスは，起動時にモジュールにある定義に基づいてメモリを確保する．また，必要に応じてメモリをページサイズ (64KB) 単位で拡張できる [1]．しかしその一方で，拡張されたメモリを部分的に解放，あるいは縮小する機能はサポートされていない．したがって，インスタンスを解放することによってのみ，そのインスタンスに結び付いたメモリを解放できる．

課題

インスタンスが利用するメモリは，一度拡大されると縮小することができない．したがって，一度大きな容量のメモリが確保されると，その後の利用実態にかかわらず，そのメモリが紐づけられたインスタンスが終了するまで解放されない．また，メモリリークを起こす WebAssembly アプリケーションを，インスタンスとしてデプロイし長期間稼働させた場合のサーバー環境のメモリ使用量の推移を図 1 に示す．最初のランタイム起動直後の様子では，OS や

[1] 東京工科大学大学院バイオ・情報メディア研究科コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404–1

[*1] https://wasi.dev/
[*2] https://github.com/proxy-wasm
[*3] https://krustlet.dev/
[*4] https://docs.fastly.com/products/compute-at-edge

その他の何らかのプロセスと WebAssembly のランタイムがサーバーのメモリを使用している．また，サーバーのメモリ使用量にはまだ十分な空き容量がある．次の複数インスタンス起動後の様子では，展開された複数のインスタンスがそれぞれメモリを使用し始めている．そのため，サーバーのメモリの空き容量が減少している．最後の OOM Kill が発生する状況の様子では，複数インスタンスが確保したそれぞれのメモリの容量がメモリリークによって拡大されている．そして，サーバーのメモリの空き容量が無くなっている．

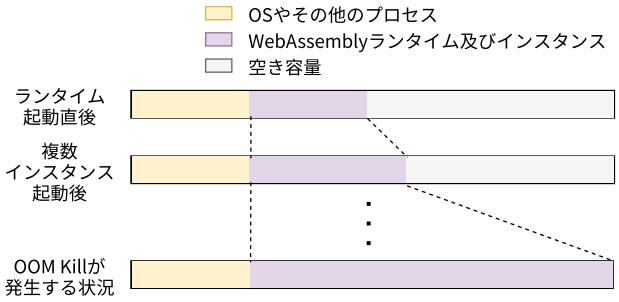

図 1　サーバーのメモリ使用量の推移

多くの場合でメモリリークは，動的に確保したメモリを自動で解放できる，ガベージコレクターの導入によって解決可能であるが，強参照が残る場合では有効に働かない．更に，WebAssembly においてガベージコレクションの利用の有無はユーザに委ねられているため，ガベージコレクションによる走査があることはシステムに対して保証することはできない．加えてガベージコレクションは，インスタンスの確保したメモリ空間の内部に対して働くものであるため，インスタンスのメモリ全体の容量の縮小はできない．

通常 WebAssembly のインスタンスは，1 つの OS のプロセス内のスレッドとして多数デプロイがされる．そのため，1 つのインスタンスによる実行環境全体のメモリ使用量への影響が小さくとも，インスタンスの数に比例してメモリへの影響も大きくなる．図 1 に，WebAssembly を実行しているサーバー環境のメモリ使用量の推移を示す．WebAssembly ランタイムを機能した後，複数のインスタンスを起動している．そして，インスタンスを長時間稼働させ続けると，図 1 の下部のように，複数のインスタンスによってメモリ使用量が増加し続け，最終的に 100%に達する．これによって，Out Of Memory (OOM) Kill が発生する状況になる．その結果，そのホストマシン上で動作するプロセスのいずれかが強制終了する．強制終了させられたプロセスが，インスタンスが動作するプロセスであった場合，そのプロセス上で動作しているすべてのインスタンスもすべて停止する．したがって，インスタンスに紐づいたメモリの不必要な肥大化によって，システムのメモリを使い切らないように工夫をする必要がある．

各インスタンスに紐づいたメモリの容量は，拡大することができても縮小することができない．そのため，インスタンスに紐づくメモリの容量を小さく抑えるためには，インスタンスの再起動によって一度 WebAssembly と共に紐づいたメモリを解放する必要がある。しかし，頻繁なインスタンスの再起動はインスタンスを長時間稼働させるという目的と相反する．したがって，どのインスタンスをいつ再起動するかが重要な課題となる．

各章の概要

2 章では，関連研究を紹介する．3 章では，提案方式とそのユースケース・シナリオについて説明する．4 章では，提案方式に基づいたソフトウェアの実装と，その実験方法について説明をする．5 章では，提案方式とそのソフトウェアの評価方法及び，得られるデータの分析手法について説明をする．6 章では，本研究の提案方式について議論をする．7 章では，本研究のまとめを述べる．

2. 関連研究

Gackstatter は，サーバーレスフレームワークで，Docker ベースである Apache OpenWhisk と，WebAssembly ランタイムの実装 3 種のコールドスタート時間・実行性能・メモリ使用量の観点で比較している [3]．提案 WebAssembly ランタイムは Docker と比較して，Rasberry Pi 上で 99.5%，サーバー上で 94%早いコールドスタートを実現しており，スループットでは Docker の 2.4 から 4.2 倍のスループットを実現している．この結果から，WebAssembly はサーバーレスプラットフォームでの利用に非常に適していることが分かった．また，クラウドネイティブの観点から Krustlet についても言及しており，Pod の管理・操作がレイテンシのボトルネックとなり，WebAssembly のコールドスタートの早さが生かせない可能性を指摘している．

Lehmann らは，WebAssembly を動的解析するための汎用フレームワーク Wasabi を提案している [4]．事前に JavaScript で実装された解析するためのコードをバイナリに挿入をして実行することにより，実行時のパフォーマンスのボトルネック，エラーやセキュリティギャップの検出が可能である．動的解析はプロダクション環境のアプリケーションに対しては利用するためのものではない．そのため，安定して長期間 WebAssembly アプリケーションを運用をするためには Wasabi のような動的解析機構の他に，恒常的な監視のための機構を検討する必要がある．

Mäkitalo らは，ブラウザ外でクロスプラットフォームのモジュラーアプリケーションを構築するために WebAssembly を利用することを検討している [5]．WebAssembly の実行時のダイナミックリンクの実装と測定を行った結果，

モジュール化によってアプリケーションの起動時間が約98%短縮されている．また，ダイナミックリンクによる実行時間への影響はごくわずかであり，モジュールから 10 分の 1 だけ機能をロードする度に実行時間が約 90%短縮されている．今後の方向性では，モジュールのプリロードによる起動時間のオーバーヘッドの最小化や，使用されていないインスタンスを検出してアンロードすることによるシステムリソースの管理，複数アプリケーション間におけるモジュールのキャッシュの共有を挙げている．

Elliott らは，リソースに制約のある IoT やフォグデバイスで WebAssembly を実行する OS である Wasmachine を紹介している [6]．この研究では，WebAssembly のネイティブコンパイルを行い，カーネルモードで実行することで Linux と比較して最大 11%高速化している．また，WebAssembly のサンドボックス機能のほか，Rust 言語によるメモリ安全性の担保によって高いセキュリティ性能を維持している．現時点ではリソースプロビジョニング機能が欠けていることに言及しており，マルチタスクを実行する際のプロセスの実行時間や最大メモリ量の制限の制限が実行できない．また，全てのプロセスがインメモリに収まることを前提としているため，インメモリに収まらない場合を考慮したメモリに関するスケジュール機構を検討する必要がある．

3. 提案方式

提案方式

本研究の提案では，同一モジュール由来のインスタンスで類別し，動的に閾値を決定する．そして，決定した閾値より大きいメモリを持つインスタンスを再起動させる．これにより，一度メモリを解放することで，実行環境全体のメモリ使用量を削減する．

提案機構全体の処理の流れを図 2 に示す．各コンポーネントはそれぞれ以下の役割を担っている．

Collector Server で実行されているインスタンスの所有するメモリの容量を Agent を経由して取得する．取得した最新のメモリの容量を Store に格納し，同時に Estimator にも送信する．本研究におけるメトリクスの収集頻度は，Julian. M の Practical Monitoring を基準に 10 秒毎とした [7].

Store 過去の各インスタンスごとのメモリの容量が保存される．

Estimator Store に格納されたメトリクスを利用して，インスタンスに必要なメモリの容量を推定する．推定値は信頼区間で算出する．

Executor メモリの容量と Estimator が推定した信頼区間の最大値を比較する．メモリの容量の方が大きい場合，Server の Agent が再起動を実行する．

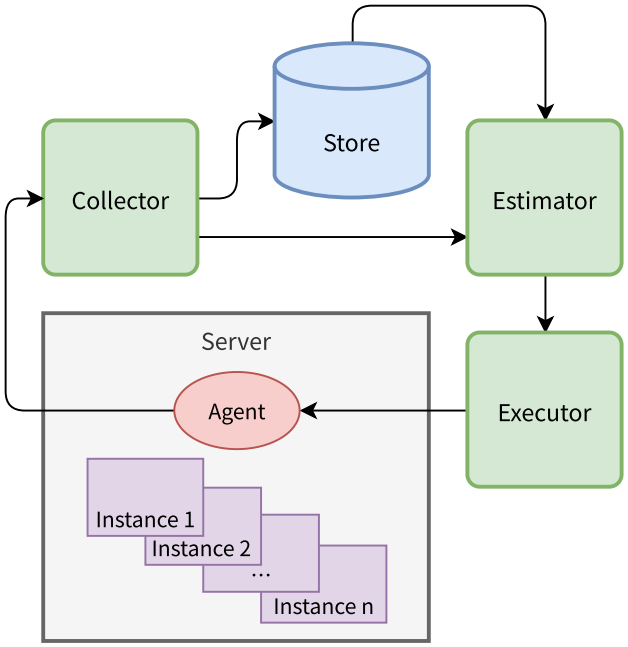

図 2 提案機構の処理の流れ

Server インスタンスを実行している環境である．Agent では，Collector へのメトリクスの送信と Executor の指令にしたがってインスタンスの再起動処理を実施する．

推定アルゴリズム

本項では，再起動するインスタンスの選択に用いる閾値の決定手法を説明する．

はじめに，同一モジュール由来の，過去から将来にわたって動作する，インスタンスのメモリの容量の代表値 a の集合を母集団 P とおく．これは，$P = \{a_1 a_2, \ldots, a_n\}$ と表記でき，n は，インスタンスの総数である．各インスタンスについて，代表値 a は起動以後の時系列のメモリの容量の中央値を利用する．

ここで，P は正規分布に従うと仮定して，P の母平均 μ を推定する．推定値は，μ をインスタンスのメモリの容量の必要最小量として，区間推定を行う．本研究の提案では，インスタンスが生成されてから計算時点までで得られている m 個からなる a の集合を標本 $x = \{a_1, a_2, \ldots, a_m\}$ として利用する．母平均 μ の信頼区間を求める際は通常，母分散 σ^2 利用する．しかし，σ^2 は (1) 式で求める必要があるが，未知数である母平均 μ に依存しているため，利用できない．

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2 \qquad (1)$$

したがって，σ^2 を利用する代わりに，推定する時点で判明しているメモリの容量のデータから求められ，期待値が σ^2 により近くなる分散である，不偏分散 u^2 を利用する．不偏分散 u^2 は，(2) 式で求めることができる．

$$u_2 = \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \overline{x})^2 \qquad (2)$$

区間推定に u^2 を利用する場合の求まる分布は正規分布ではなく t 分布になる．t 分布は標本数の増加につれ，正規分布に近似していく特徴がある．そのため，P が確かに正規分布に従っている場合，メトリクスの数が多くなるにつれ推定の精度が向上する．また，区間推定に用いる標本平均 $\overline{x}$ は，(3) 式で求まる．

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad (3)$$

更に，t 分布における区間推定では，統計量である t 値を予め計算しておく必要がある．自由度が $m-1$ で，信頼水準を $1-\alpha$ とおいたときの上側確率が $\alpha/2$ となる t 分布を考える．このときの t 値は $t_{\alpha/2}(n-1)$ と表現できる．母分散 σ^2 が未知である場合の母平均 μ の信頼区間は，

$$\overline{x} - t_{\alpha/2}(n-1)\sqrt{\frac{u^2}{n}} \leq \mu \leq \overline{x} + t_{\alpha/2}(n-1)\sqrt{\frac{u^2}{n}} \quad (4)$$

の式で求めることができる [8]．

本研究の提案手法では，推定された区間の最大値である $\overline{x} + t_{\alpha/2}(n-1)\sqrt{\frac{u^2}{n}}$ を閾値として利用する．この閾値を超えたメモリの容量を確保している WebAssembly のインスタンスを再起動することで，各インスタンスのメモリの容量の肥大化を抑制する．

ユースケース・シナリオ

WebAssembly バイナリにコンパイルされたアプリケーションインスタンスをデプロイし，長期間サービスを停止させずに運用することを考える．アプリケーションとして，24 時間 365 日稼働しているオンラインストレージサービスを想定する．Web ブラウザベースでのファイルのアップロード・ダウンロード，ファイルの一覧とメタデータの表示をサポートする．取り扱うファイルは，数 KB のテキストファイルから数百 MB の動画ファイルまで，大小は様々なサイズが存在する．

サーバー側は，図 3 の構成であり，HTTP リクエストを受け取って以下の処理をする．

`GET /path` `path` に対応するファイルのメタデータを Volume から取得する．

`POST /path` リスエストボディの書き込まれた，`path` に対応するファイルを Volume に作成する．

`DELETE /path` `path` に対応するファイルを Volume から削除する．

本研究の提案機構を上記のユースケースに適用する場合，初めにオンラインストレージサービスの動作環境に配置しストレージサービスのインスタンスを監視する．そして，メモリリークや巨大なファイルの読み込みによってメモリが肥大化したインスタンス検出し，再起動する．インスタンスの再起動にしたがって，インスタンスに紐づくメモリも 1 度解放される．これによって，インスタンスとそのメモリによってホストマシンのメモリの空き容量が不足する事態を回避できる．

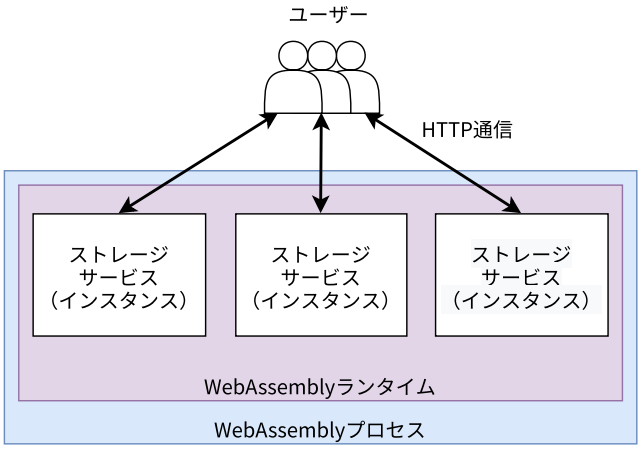

図 3 アプリケーションインスタンスの配置

4. 実装と実験方法

実装

本研究における実装では，機能を以下の 3 つに大分できる．

- 各インスタンスの起動と削除をする機能．
- 各インスタンスのメモリの容量を一定間隔で取得する機能．
- 各インスタンスのメモリ統計をもとに再起動対象を決定する機能．

図 4 に実装の全体の構造を示す．2 つのプロセスに分かれており，2 つのプロセス間は HTTP リクエストの送信と HTTP レスポンスの受信を行う．Worker プロセスでは，WebAssembly ランタイムである wasmtime を利用しており，インスタンスの起動と停止を行う．また，Julian. M の Practical Monitoring を基準にメモリ容量を 10 秒毎に取得する [7]．

Controller プロセスでは，各インスタンスのメモリの容量を収集し，時系列データを構築する．そして，そのインスタンス毎の時系列のメモリの容量の推移をもとに提案アルゴリズムを用いて再起動する対象のインスタンスを選定する．再起動対象のインスタンスが決定されると，Worker プロセス経由でインスタンスの削除と起動を実行する．

実験環境

実験環境を図 5 に示す．ハイパーバイザ (VMware ESXi) 上の仮想マシンを利用する．仮想マシンに Ubuntu 20.04 をインストールし，その上で 4 の Worker プロセスと Con-

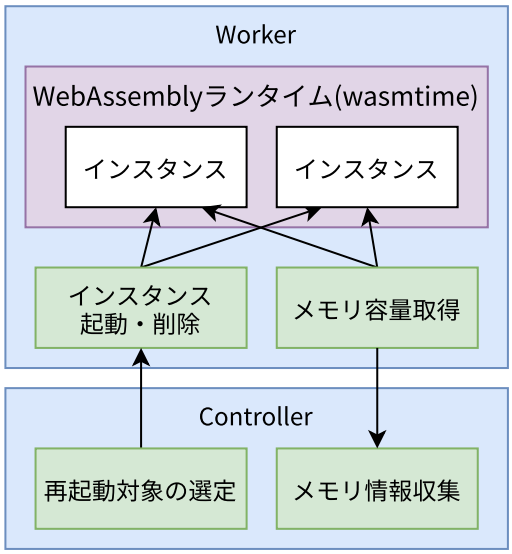

図 4　実装の概要図

troller プロセスを実行する．そして，Prometheus[*5]を実行する仮想マシンをもう 1 台用意し，Worker, Controller プロセスの動作するノードの CPU 使用率とメモリ使用量のメトリクスを収集する．実験で用いる WebAssembly のアプリケーションには，プログラム中に意図的にランダムな大きさでメモリリークの発生させ，メモリの使用量の増加を再現したものを利用する．

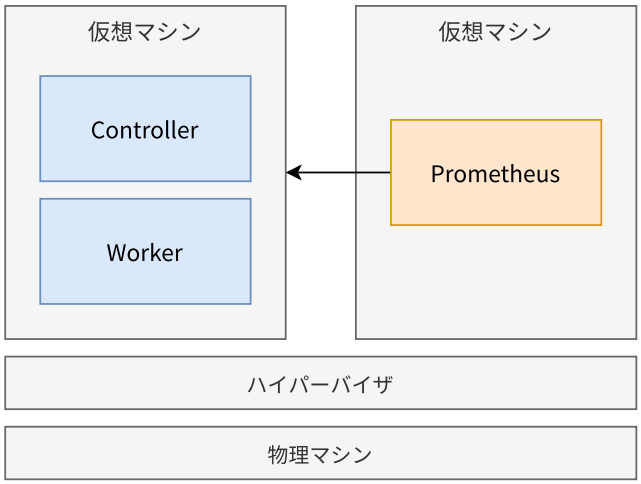

図 5　実験環境の全体像

5.　評価手法と分析手法

評価方法

評価手法では，以下の手法をそれぞれ WebAssembly のワークロードに適用して比較をする．

- 本研究の提案手法を用いて動的にインスタンスを再起動させた場合
- ホストマシンのメモリの空き容量が不足したら，最もメモリの容量が大きいインスタンスを再起動させた場合
- ホストマシンのメモリの空き容量が不足したら，最も長時間動作しているインスタンスを再起動させた場合
- 再起動処理を行わなかった場合

また，以下のメトリクスについて実際に計測する．

- インスタンスのメモリ使用量
- サーバー（実行環境）のメモリ使用量
- インスタンスの起動時間

評価項目

評価に用いる項目を下記に示す．

- インスタンスの必要なメモリ容量の推定精度
- ホストマシンのメモリ使用量とその変化量
- 各インスタンスの稼働時間
- 同一モジュールのインスタンス毎の再起動の頻度

インスタンスの必要なメモリ容量の推定精度は，予め必要なメモリ容量が設定されたインスタンスを利用し，推定される区間とどれだけ近似するかを計算し，評価する．ホストマシンのメモリ使用量は，それぞれの条件で同数のインスタンスが動作させて比較する．提案手法の適用によって，平均のメモリ使用量が削減されるか確認する．また，インスタンスの稼働時間も，同一モジュールのインスタンス毎で比較をする．

6.　議論

本研究の提案では，同一モジュール由来のインスタンスについて，計測を開始してから推定時点までに取得したメモリの容量の推移から，母平均の信頼区間を求める．そして，求めた信頼区間の最大値を閾値として利用する．閾値より大きいメモリを確保しているインスタンスを検出し，再起動することでインスタンスのメモリの容量を削減する．

再起動対象の判定では，新しくメトリクスが追加される毎に信頼区間を再計算しているが，インスタンスのメモリの容量が緩やかに増加し続けた場合に，信頼区間の値も追従して増加し続ける．この場合，メモリの容量が肥大化しても再起動の対象外となる．そのため，信頼水準の値の設定を信頼区間を小さく抑えるという工夫が必要になる．しかし，信頼区間が非常に小さく設定されている場合，メモリの拡大が一切許されず，直ちに再起動の対象となる．更に，インスタンスに割り当てられるメモリの増加の傾向はアプリケーションに依存する．したがって，信頼水準の値の決定についてもインスタンス毎に動的に決定する必要があると考える．この問題の解決方法としては，インスタンス毎のメモリの増加傾向の傾きを利用する．単位時間当たりのメモリの増加量の多いインスタンスに対しては信頼水準の値を下げることで許容する区間の幅を拡大する．反対

[*5] https://prometheus.io/

に，単位時間あたりのメモリの増加量が僅かなインスタンスに対して，信頼水準の値を上げることで許容される区間を縮小する．これにより，インスタンスのメモリの増加傾向に対応した信頼水準の値が決定できると考える．

また，現時点では WebAssembly および WASI について，エコシステムが未成熟であり実装されていない機能が複数存在する．例えば，ネットワークインタフェースの仕様が決まっておらず，実装で利用する予定のリファレンス実装である Wasmtime にもその機能が実装されていない．他にも，マルチスレッドやアトミック処理といった機能も仕様が決定していない [9]．そのため実装と実験の際に，ネットワークとのインタフェースをホスト側への部分的な実装が必要になる．あるいはリファレンス実装に加えて，ネットワークインターフェースの実装された別の WebAssembly ランタイムの実装を利用する必要がある．

7. おわりに

本研究は，WebAssembly を長期間運用をする際に，各 WebAssembly のインスタンスに紐づけられるメモリが肥大化すること課題としている．特にメモリリークが生じるインスタンスが実行環境に含まれていた場合，メモリの割り当て容量が拡大し続け，インスタンスが動作している実行環境のメモリを使い切る．それに対するアプローチとして，インスタンスを再起動をすることによってメモリの容量を縮小する手法を提案した．はじめに，各インスタンスに紐づくメモリの容量を監視する．そして，得られるメモリの容量の推移をもとに，そのインスタンスに適切なメモリの容量を推定し再起動の対象を動的に決定する．これによって，インスタンスの再起動を最小限に抑えつつ，ホストマシンのメモリ使用量の削減を図る．評価では，実行環境のメモリ使用量とインスタンスの稼働時間について，提案手法の有無で比較をする．

参考文献

[1] 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, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200 (2017).

[2] González, A., Latorre, F. and Magklis, G.: *Processor Microarchitecture: An Implementation Perspective*, Synthesis lectures on computer architecture, Morgan & Claypool Publishers (2010).

[3] Gackstatter, P.: A WebAssembly Container Runtime for Serverless Edge Computing, *Software Engineering*, p. 116.

[4] Lehmann, D. and Pradel, M.: Wasabi: A framework for dynamically analyzing webassembly, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1045–1058 (2019).

[5] Mäkitalo, N., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O. and Mikkonen, T.: Bringing WebAssembly up to Speed with Dynamic Linking, *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, New York, NY, USA, Association for Computing Machinery, p. 1727–1735 (online), DOI: 10.1145/3412841.3442045 (2021).

[6] Wen, E. and Weber, G.: Wasmachine: Bring IoT up to Speed with A WebAssembly OS, *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 1–4 (online), DOI: 10.1109/PerComWorkshops48775.2020.9156135 (2020).

[7] Julian, M.: *Modern Monitoring* (2017).

[8] Mood, A. M., Graybill, F. A. and Boes, D. C.: *Introduction to the Theory of Statistics* (1974).

[9] Rossberg, A.: WebAssembly Core Specification. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.