Vol.2022-SE-210 No.11
2022/3/11

WebAssemblyを対象としたCode Compactionツールの提案およびCode Compactionの有効性検証

齋藤 優太[1,a)]　坂本 一憲[1,b)]　鷲崎 弘宜[1,c)]　深澤 良彰[1,d)]

概要：ウェブブラウザにおける WebAssembly（Wasm）の利用場面ではダウロード時間とコンパイル時間の短縮が課題となっている．短縮の手段の一つとして Code Compaction が考えられ，既に LLVM はコンパイル時とリンク時における Function Merging を含む Code Compaction 最適化を行っているが，リンク後の最適化に対するサポートは十分ではない．リンク後のプログラムを対象とする場合，ビルド構成の工数を必要としないため適用対象を大幅に広げることが可能だが，Wasm を対象としてリンク後に Function Merging 可能な Code Compaction ツールは存在しない．本研究では Wasm を対象としてリンク後に Function Merging 可能な Code Compaction ツールの実装と 1200 件の実行ファイルを対象とした有効性検証を行った．検証の結果，従来の Wasm を対象としてリンク後に最適化可能な Code Compaction ツールと比較して，コードサイズ削減率が 5.1%から 7.6%へ 49%程度改善された．

キーワード：code compaction, function merging, code size reduction, WebAssembly

Proposal of a Code Compaction Tool for WebAssembly and Validation of Code Compaction

Abstract:
In the use of WebAssembly (Wasm) in web browsers, reducing the download time and compile time is an issue. Code Compaction Code Compaction is one of the possible ways to shorten the time. LLVM already performs Code Compaction optimization including Function Merging at compile time and link time, but there is not enough support for post-link optimization. Post-link optimizer can optimize already linked executable files, and it can optimize much more binaries. However, there is no code compaction tool for Wasm that can perform Function Merging after linking. In this study, we implemented a code compaction tool that can perform Function Merging after linking for Wasm, and verified its effectiveness using 1200 executable files. As a result of the verification, the code size reduction rate was improved from 5.1% to 7.6%, an improvement of 49%, compared to the conventional Code Compaction tool that can optimize Wasm after linking.

Keywords: code compaction, function merging, code size reduction, WebAssembly

1. はじめに

WebAssembly はバイナリ形式の命令セットである．さまざまなプログラミング言語のコンパイルターゲットであり，クライアントおよびサーバーアプリケーションでの利用を目的とした技術である [1].

[1] 早稲田大学
Waseda University
[a)] ysaito@suou.waseda.jp
[b)] exkazuu@gmail.com
[c)] washizaki@waseda.jp
[d)] fukazawa@waseda.jp

WebAssembly 形式のプログラムはウェブページの読み込み時に必要に応じてネットワーク越しにダウンロード，JIT コンパイル，実行される実行モデルを持つ．一般的なウェブページは，ロード時間の 80〜90 %をリソースのダウンロードの待機に費やすことが知られており [2]，JavaScript や画像ファイルの圧縮，遅延読み込みなどの技術が開発されている．WebAssembly プログラムに関しても同様に読み込み時間の短縮が課題となっており，動的リンクの実装 [3] やプログラムサイズの削減などが検討されている．

プログラムサイズの削減技術として，プログラムデー

タの圧縮を適用する Code Compression と，プログラムとしての形を保ったまま，プログラムからプログラムへの変換を行う Code Compaction が知られている [4]．Code Compaction はソースコードレベルや中間表現レベル，実行ファイルレベルなど複数のレイヤで実装が行われている．

既に，LLVM には IR レベルで同一の関数を一つに纏める最適化が実装されており，完全には一致していないが類似する複数の関数を一つの関数に纏める Function Merging も提案されている [5]．LLVM IR のような中間表現レベルでの最適化を適用するためには，プログラムの再コンパイルが必要となる．適用対象のプロジェクトごとにビルド方法が異なるため，最適化の有効性を評価する際に，機械的に実行ファイルを収集して最適化するといった調査が困難である．

一方，リンク後のプログラムファイルに対する最適化である Post-link Optimization（PLO）では，WebAssembly を対象とした場合，最適化の適用にプログラムの再コンパイルを必要としない．また，WebAssembly を対象とした Binaryen[*1]などの既存の Post-link Optimization ツールは，類似する関数を一つの関数にマージする Function Merging の機能を有しておらず，WebAssebly を対象とした Function Merging の有効性が検証されていない．

本研究では WebAssembly を対象とした既存の最適化ツール Binaryen に Function Merging の実装を追加し，合計 1200 件の実行ファイルを対象として，Function Merging の有効性を検証した．本論文における研究課題（RQs）は以下の 4 つである．

RQ1 Wasm として配布されたプログラムを対象として Function Merging を適用できるか？

RQ2 Function Merging 機能を搭載した Binaryen によってどの程度のファイルサイズを削減できるか？

RQ3 Function Merging 機能の追加により，Binaryen の削減率をどの程度改善できるか？

RQ4 最適化対象となる Wasm を出力するツールチェインと削減率にどのような関係性があるか？

本論文の貢献は以下の通りである

- 既存ツールである Binaryen において，Wasm を対象とした Function Merging を実装，プロジェクトへ寄贈した[*2]．
- インターネットで配布されている 1200 件の Wasm 実行ファイルに対して Function Merging を適用して有効性を検証した．
- Wasm を生成するツールチェインと Function Merging の削減率の関係性を明らかにした．

[*1] https://github.com/webassembly/binaryen/
[*2] https://github.com/WebAssembly/binaryen/pull/4414/

2. Post-link Optimization に関する研究

リンク後のプログラムファイルに対する最適化は Post-link Optimization（PLO）として研究が行われている．PLO は Link-time Optimization と同様にプログラム全体の解析が可能な最適化である．最適化実装がアーキテクチャやファイル形式に依存するが，プログラムの再コンパイルを必要とせず，より少ない計算量で様々なプログラムに対して最適化が可能な場合がある．

しかし，一般的なコードセクションのアドレス指定によるジャンプを採用するアーキテクチャのプログラムでは，PLO の適用に困難を伴う．なぜなら，コードセクション中の命令数の増減によって，ジャンプ先のアドレスが変化するためである．PLO ツールは関数のサイズを変化させないように調整するか，最適化後のジャンプ先を元の意味と合わせるために，命令中の即値やデータセクション中のコードセクションのアドレスを更新する必要がある．しかし，プログラムファイルのどの位置に，どのシンボルのアドレスが書き込まれているか，といった再配置情報は通常リンカによって取り除かれ，最終的な実行ファイルには出力されない．そのため，例えばリンク後のプログラムに対する PGO（Profile-Guided Optimization）ツールである BOLT ではオブジェクトファイルの持つ再配置レコードをリンク後に保持させるリンカオプションを用いて，関数のサイズ増減によるシンボルアドレスの変更を反映している [6]．つまり，ジャンプ先をコードセクション中のアドレスで表現するアーキテクチャでは，PLO を前提とせずに生成された実行ファイルに対しては PLO の適用が困難であり，プログラムを再配置情報付きで出力するように再リンクする必要がある．

一方，WebAssembly は関数内のジャンプを実現するために，構造化された制御フローに基づいたジャンプ命令を採用しているため，プログラムのサイズによってジャンプ先が変化しない．また，関数呼び出しを表現する際に，コードセクション内のアドレスではなく，関数のインデックスを用いて呼び出し先の関数を指定するため，ジャンプと同様にプログラムのサイズの影響を受けない．加えて，コード空間に対する直接のアドレス参照がデータ空間に存在せず，コードセクションとは分離された関数テーブル上のアドレスが間接的に参照されている．このような特性から，リンク後のプログラムに対するコード領域の変更が容易となっている．コード領域の変更の容易さにより，プログラムを再コンパイルすることなく Binaryen を始めとした WebAssembly の PLO ツールを適用できるため，従来のアーキテクチャのプログラムと比べ PLO の適用範囲を大幅に広げることが可能である．

3. WebAssembly を対象とした Function Merging の提案

プログラムサイズを削減するために，WebAssembly 上で定数のみが異なる類似関数をマージする PLO として，Function Merging を提案する．提案する Function Merging のアルゴリズムは，マージ可能関数の抽出，マージ可否の判定，類似関数のマージの 3 ステップに分かれる．本節ではそれぞれのステップについて説明する．

3.1 マージ可能関数の抽出

素朴にモジュール内の全ての関数を比較することで，マージ可能な関数の抽出を試みると $O(N^2)$ の計算量を要する．そこで，比較する前にマージ可能な関数群が同一のハッシュ値を持つようなハッシュ関数を定義することで，マージの見込みのない関数同士の比較を回避して，計算量を削減できる．具体的には，関数の引数と返り値の型，パラメータ化可能な命令の即値部分のみを除いた命令列を元にハッシュ値を求めるようなハッシュ関数を採用する．ここで，パラメータ化可能な定数項は，スタックに定数をプッシュする`{i32,i64,f32,f64}.const` の 4 命令と，関数直接呼び出しの `call` 命令である．ハッシュ値が等しい関数をグループ化して，グループ内の全ての関数の組み合わせに対して，パラメータ化可能な命令を除いて全て等価であることを検証することで，マージ可能な関数群を抽出する．

3.2 マージ可否の判定

関数のマージ操作は，常にサイズ面での利得がある訳ではなく，マージ時に生成されるサンク関数（関数を呼び出すラッパー関数）によって損失が発生している．よって，実際にマージを適用する前に，マージによって得られる利得と損失を計算し，利得が損失を上回る場合のみマージを行う．

3.3 類似関数のマージ

上記ステップで，即値部分を除くと，マージ可能な関数群に含まれる関数の命令列は同一のオペコードと命令数を持つことが保証されている．以上を踏まえたマージの手順は以下の通りである．

(1) 関数間で異なる即値を持つ命令の位置をパラメータ化予定の命令として抽出する．
(2) 関数群中のある関数を共通関数 S としてコピーする．
(3) 共通関数 S にパラメータの数だけ仮引数を追加する．
(4) S 中のパラメータ化対象の命令を，引数からの値の取得する命令に置換する．
(5) 関数群に含まれるすべての関数を，与えられたパラメータとパラメータ化された値を引数として共通関数 S を呼び出すサンク関数に置換する．

図 1 の左側の関数 a と関数 b のマージを例とすると，`call $f` と `call $g`，`i32.const 42` と `i32.const 24` がパラメータ化される命令となる．関数 a を元に，パラメータ化される 2 命令分の仮引数を追加し，`call` 命令と `i32.const` 命令を引数の値を取得する `local.get` 命令に置換し，共通関数 ab を生成する．そして，関数 a と関数 b をそれぞれの定数パラメータである `i32.const` と `ref.func` を引数として，共通関数 ab を呼び出すサンク関数に置換する．

本手順において，1 つの関数グループのマージによって増加する命令数は，生成されたサンク関数内の命令数であり，パラメータ数 × 関数数である．

なお，共通関数の引数に関数 id を渡して，共通関数の中で id を元に switch-case によって挙動を切り替える手法も考えられるが，以下の理由により採用しなかった．まず，WebAssembly の PLO ではコードセクションの変更は容易だが，データセクションへのデータの追加は，ヒープ用に確保されたメモリ空間と衝突してしまう．そのため，switch-case で用いるジャンプテーブルを作ることが困難であることが理由に挙げられる．また，if-else による分岐や，`block` 命令と `br_table` 命令による構造化されたジャンプを用いても実現は可能だが，マージパラメータ数 × 関数数 × 構造化命令の数だけ命令数が増加してしまうため採用しなかった．

3.4 Binaryen への実装

本研究では WebAssembly に対応した最先端の PLO ツールとして Binaryen を採用した．Binaryen は WebAssembly 用のコンパイラ基盤と，WebAssembly プログラムを入力として受け取り，WebAssembly プログラムを出力する Post-link Optimizer の wasm-opt を提供する．我々は wasm-opt を改良することで，Function Merging による最適化機能を実装した．

Function Merging は Binaryen の最適化パスとして実装されており，関数ごとの最適化が適用された後，実行される．Function Merging を後半に実行する理由は，Dead Code Elimination や Inlining，Locals Reordering によって関数の類似性が高まる可能性があること，モジュール内の関数が削減され，マージ可能性の検証が安価になるためである．なお，Locals Reordering は WebAssembly 上の呼び出しフレームのみで有効なレジスタである Local の割当を正規化する最適化パスである．これを Function Merging の直前に実行することで，プログラムの意味は完全に一致するが，Local の使われ方が異なる関数群がマージ可能になる場合がある．

また，Binaryen の最適化パスの実行中では正確な削減サイズとサンク関数の生成による増加サイズを求めることが出来ない．これは，関数が最終的にどの順序でバイナリに

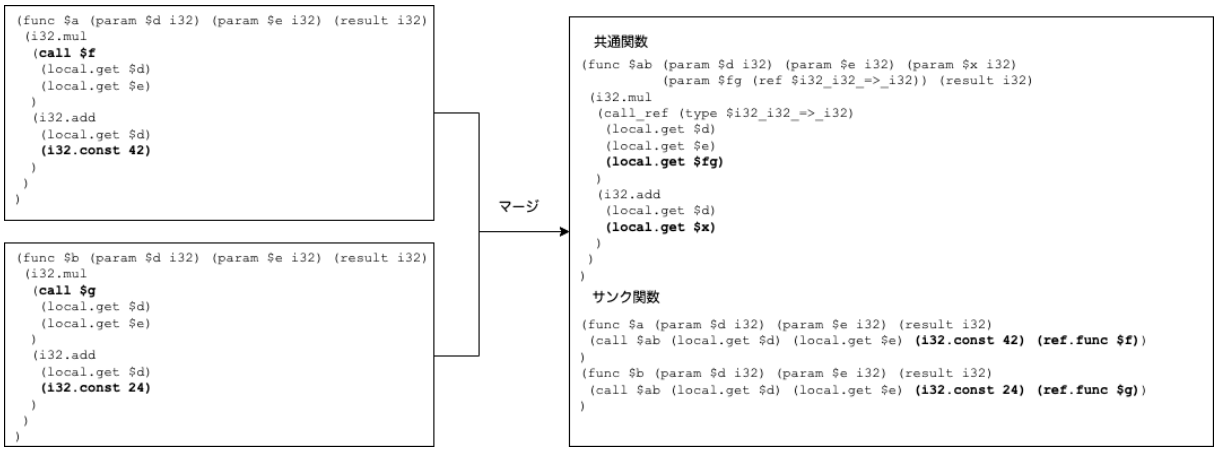

図 1 類似関数のマージの例

書き出されるか決定できないことと，`ref.func` の即値部分や関数エントリの命令列サイズフィールドなどが可変長エンコーディングの LEB128 となっているためである．そのため，多少保守的に増減における要素に対して重みを付け，最終的な出力時の利得を推定し，それぞれのマージグループに対して，実際にマージを適用するか否かを決定している．

4. Function Merging の有効性検証

本研究で提案する Function Merging の有効性検証のために，Node.js のパッケージレジストリである npm[*3]から 1080 件，Unity 製フリーゲーム投稿サイト unityroom[*4]から 120 件のプログラムを収集し，Binaryen が対応する拡張仕様のみを使うプログラム 1200 件に対して実験を行った．なお，明らかにテストやデモンストレーション用に作成されたプログラムは，実際のアプリケーションの特性を反映しないものとして検証対象のプログラム群から除外している．

それぞれのプログラムに対して，Function Merging 実装前と実装後の PLO ツール wasm-opt を適用し，プログラムサイズの削減率の測定と Function Merging 適用中の統計情報の収集を行った．さらに，npm で配布されているプログラムに関しては，各種プログラミング言語から WebAssembly を生成する，Rust や Emscripten などのツールチェインが様々であるため，ツールチェインの推定を行った．推定は，producer セクションの情報や，各ツールチェイン固有のシンボルが存在するかを元に行われている．なお，ここでの有効性検証とは，直接的な効果としての削減率に着目するものであり，実行時間への影響については最終節で議論する．

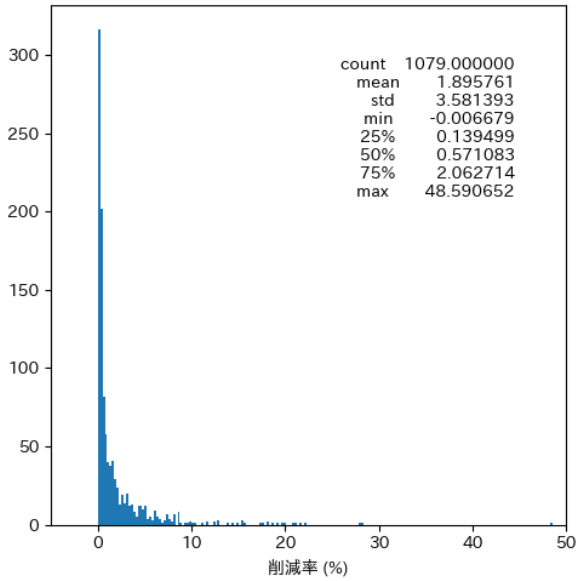

図 2 Function Merging 導入による削減率の度数分布 (npm)

4.1 コードサイズの削減率

図 2 は npm で配布されているプログラム 1080 件を対象とした，Function Merging の導入によって新たに得られた削減率の増加分の度数分布である．

図 3 は unityroom で配布されている Unity アプリのプログラム 120 件を対象とした，図 2 と同様の度数分布である．npm を対象とした結果では，平均削減率は 5.4%から 7.3%まで（改善率は 37.3%），Unity を対象とした結果では 3.2%から 10.7%まで（改善率は 230%），npm と Unity 全てのプログラムを対象とすると，5.1%から 7.6%まで（改善率は 49.0%）向上している．

削減率が最大となったのは，zstd の 48.9%の削減であった．zstd については 4.3 節にて考察を行う．

実行ファイル 1200 件中 990 件，つまり 82.5%のプログラムでサイズの減少が見られた．

*3 https://npmjs.com
*4 https://unityroom.com/

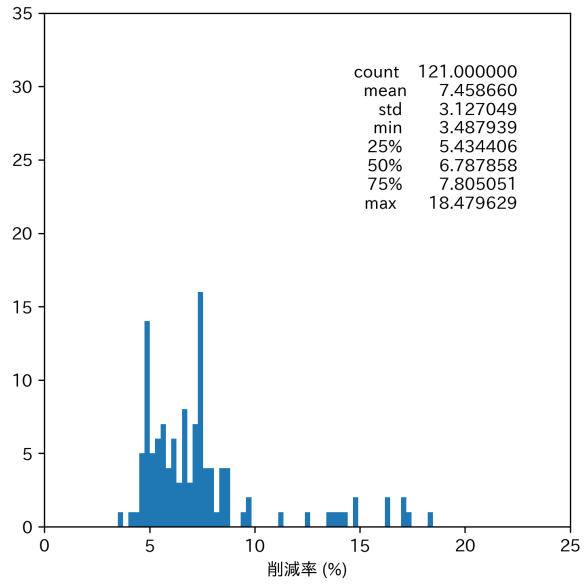

図 3 Function Merging 導入による削減率の度数分布 (Unity)

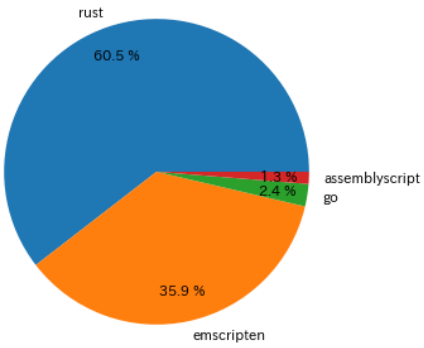

図 4 npm パッケージ内の各ツールチェインの使用率

4.2 ツールチェインごとの特性

図 4 に npm で配布されているプログラムの生成元ツールチェインの使用率を示す．95%以上のプログラムは Rust または Emscripten によって生成されているため，以降 Rust と Emscripten に注目した分析を行う．

図 5，**図 6** に Emscripten，Rust それぞれのプログラムに対する Function Merging の統計情報を示す．

まず，言語レベルでの特性について考える．C++のプログラム中でマージ可能な関数として，テンプレートを具体化した関数群が考えられる．Rust のプログラム中でマージ可能な関数としては，単相化されたジェネリック関数群が考えられる．

図 5 によると，C と C++を入力とする Emscripten によって生成されたプログラムのほうが，Rust プログラムに比べ，関数のマージ機会が少ない．テンプレートとジェネリックスといった類似した機能を持つ言語間でこのような差が現れるのは，それぞれの処理系の単相化方法の差が原因と考えられる．C++の具体化されたテンプレートは，具体化される度に関数が生成され，全く同じパラメータから生成された関数は全く同じシンボル名を持ち，リンク時に COMDAT セクションの情報を用いて重複が取り除かれる．一方で，Rust の単相化されたジェネリック関数は，クレート[*5]の名前をシンボル名にエンコードするため，異なるクレートでは同じパラメータが与えられたとしても異なるシンボル名を持ち，リンク時に定義が衝突すること無く，実行ファイルに同一の意味を持つ関数が複数含まれてしまう．

つまり，C++の場合，異なるパラメータがテンプレートに与えられた時のみマージ可能な関数群が生成されるが，Rust の場合，与えられたパラメータに依らず，マージ可能な冗長な関数群が生成されているため，マージ機会が多くなっている．このように，Rust や C++などのコンパイル時に多相な関数を単相化する言語では，Function Merging は単相化された関数群を多相化し直すような操作と言える．

次に，図 6 のマージによって得られる利得を元にマージをスキップした割合に注目する．マージがスキップされるケースは，マージ対象の関数が小規模であり，サンク関数を生成する追加サイズの方が，削減サイズより大きくなってしまうケースに該当する．そのため，Emscripten のプログラムと比べ，Rust のプログラムのほうが，小さい類似関数が多くなっていると言える．

マージをスキップした回数が最も多いパッケージである ditto はソースコードが公開されていないため，2 番目に多いパッケージである sane-fmt に対して調査を行った，スキップされた関数の総数は 883 件であり，そのうちの 309 件は**図 7** のような，ある trait X を実装するジェネリックな型 T の参照型&T に対する trait X の実装となっている．このような trait のインターフェースに適合するためだけのジェネリックなアダプタ実装は，実装自体が小さく，更に Debug や Clone など頻繁に使われる trait の場合，多数の単相化された関数が生成されるため，図 6 のような結果となっている．

5. ケーススタディ

npm から収集したプログラムの中で最大の削減率 48.9%となった zstd と，2 番目に大きい削減率 22.5%の swc を対象としたケーススタディを行う．

5.1 zstd

zstd は C で実装された圧縮アルゴリズム実装ライブラリである．zstd の実装には，圧縮速度を犠牲にして圧縮率

[*5] Rust におけるライブラリもしくはプログラム単位を総称する

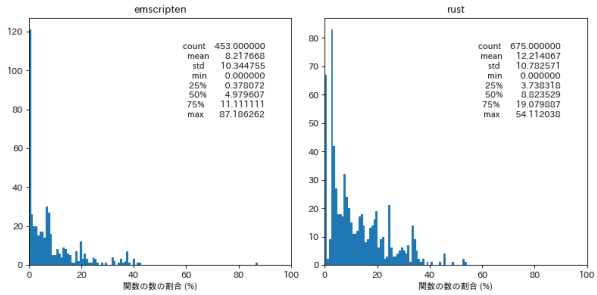

図 5 プログラム内の関数の内でマージされた類似関数の割合

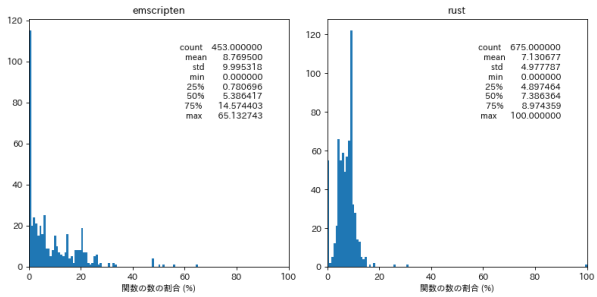

図 6 プログラム内の関数の内でサンク生成コストが利得を上回った類似関数の割合

```
impl<T: ?Sized + Display> Display for &T {
  fn fmt(&self, f: &mut Formatter<'_>) -> Result {
    Display::fmt(&**self, f)
  }
}
```

図 7 ジェネリックな型に対する trait 実装の例

表 1 パッケージ個別の削減結果 (KB).
プログラム列中の O0 および Oz は C コンパイラの最適化レベルを意味する．列名の FM とは Function Merging である．

プログラム	元のサイズ	FM 無し	FM あり
zstd (O0)	1149	1145 (-0.32%)	587 (-48.9%)
zstd (Oz)	534	533 (-0.13%)	527 (-1.28%)
swc	23625	22127 (-0.32%)	17145 (-22.5%)

を向上させるパラメータがあり，インライン展開によって一般的な圧縮実装を特定のパラメータにスペシャライズさせている．スペシャライズの為のインライン展開を強制する`__attribute__((always_inline))`属性を使用しており，最適化レベル 0 の定数伝搬しないモードであっても，インライン展開される．npm で配布されていたプログラムは最適化レベルが 0 であったため，単純にパラメータが展開されただけの圧縮アルゴリズムの実装のコピーが生成されていた．この状態のプログラムに対して，Function Merging を行うと，`__attribute__((always_inline))`によって強制的に行われたインライン化が打ち消されるような操作となり，冗長でサイズの大きいコピー実装を削減できていた．

5.2 swc

swc は Rust で実装された ECMAScript のコンパイラである．特にサイズ削減に貢献していたパーサの実装は，字句解析器から生成されたトークン列のイテレータをジェネリックパラメータとして受け取り，字句解析器以外から生成されたトークン列に対してもパースできるよう一般化されている．また，パーサには 100 件以上のメソッドが実装されており，単相化される毎に全てのメソッドがコピーされる．Function Merging は単相化を打ち消すように，コピーされた関数群をマージすることで，ライブラリの核となるパーサ実装のサイズを削減できていた．

6. 議論

ここで，研究課題に関連付けて得られた結果について考察する．

RQ1 Wasm として配布されたプログラムを対象として Function Merging を適用できるか？

npm と unityroom から 1200 件のプログラムを収集し，再ビルドすること無く全てのプログラムに適用可能であることが示された．

RQ2 Function Merging 機能を搭載した Binaryen によってどの程度のファイルサイズを削減できるか？

評価の結果，サンプル全体平均で 7.6%の削減率を達成した．

RQ3 Function Merging 機能の追加により，Binaryen の削減率をどの程度改善できるか？

評価対象全体の平均削減率は，5.1%から 7.6%へ 49.0%向上した．

RQ4 最適化対象となる Wasm を出力するツールチェインと削減率にどのような関係性があるか？

ツールチェイン毎に結果を分析したところ，多相性を持つ関数の単相化プロセスに依存して，マージ機会が大きく増減することが判明した．

7. 妥当性への驚異

外部妥当性の脅威として，適用対象が npm で配布されていたプログラムと Unity 製のプログラムに限定されていることから，偏りがあった可能性がある．今後，実際のウェブサイトで使われいる Wasm プログラムを直接収集することで，実際のユーザ体験に近い領域での有効性を検証したい．

8. おわりに

ウェブブラウザ上のユーザ体験の向上を目的として，WebAssembly を対象としてプログラムサイズの削減について検討してきた．特に，プログラム中に含まれる類似関数をマージする Function Merging を，適用範囲の広い Post-link Optimization として実装，評価した．なお，本

研究で実装，評価した Function Merging 実装は OSS である Binaryen に寄贈されている[*6]．

今後の展望としては，コスト推定時に用いた重み値のチューニングや，削減率と実行時パフォーマンス，JIT 最適化速度のトレードオフ関係の調査が挙げられる．削減率とのトレードオフ関係について，本研究の最適化はインライン展開されていたものを逆に展開前の状態に戻す操作と考えられるため，JIT 時の最適化器に余分な仕事をさせ，実行までの待機時間が伸びる可能性がある．近年のウェブブラウザに搭載されている WebAssembly 実行エンジンは，ベースラインコード生成と最適化コード生成を共に行うため，最適化時間が伸びたとしても，ベースラインコード生成の速度が変動しない限り，ユーザ体験への影響は少なく済むのではないか，という仮説が立てられるため，検証の必要がある．

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. F.: Bringing the Web up to Speed with WebAssembly, *SIGPLAN Not.*, Vol. 52, No. 6, p. 185–200 (online), DOI: 10.1145/3140587.3062363 (2017).

[2] Manhas, J.: A study of factors affecting websites page loading speed for efficient web performance, *International Journal of Computer Sciences and Engineering*, Vol. 1, No. 3, pp. 32–35 (2013).

[3] 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).

[4] Beszédes, A., Ferenc, R., Gyimóthy, T., Dolenc, A. and Karsisto, K.: Survey of Code-Size Reduction Methods, *ACM Comput.Surv.*, Vol. 35, No. 3, p. 223–267 (online), DOI: 10.1145/937503.937504 (2003).

[5] von Koch, T. E., Franke, B., Bhandarkar, P. and Dasgupta, A.: Exploiting function similarity for code size reduction, ACM, pp. 85–94 (online), DOI: 10.1145/2597809.2597811 (Jun 12, 2014).

[6] Panchenko, M., Auler, R., Nell, B. and Ottoni, G.: BOLT: a practical binary optimizer for data centers and beyond, CGO 2019, IEEE Press, pp. 2–14 (online), DOI: 10.1109/CGO.2019.8661201 (Feb 16, 2019).

[*6] https://github.com/WebAssembly/binaryen/pull/4414/