コンピュータ科学科情報システム系卒業論文

DTVMを用いたオフチェーン型スマートコントラクト実行基盤の組込みシステムへの適用

2026 年 2 月

102230045　岩見 一輝

概要

近年,IoT 技術の普及に伴い, 組込み機器間での自律的な取引やデータ管理を実現する手段としてブロックチェーンおよびスマートコントラクトの活用が期待されている [1]。しかし, すべてをノード上で実行する従来のフルオンチェーン実行方式では, 膨大な計算コストや実行遅延, およびガスコストの発生が課題となり, 計算資源や電力に制約のある組込み機器への適用は困難であった。また現在のブロックチェーンはクラウドサーバ等の外部リソースに依存した構成が一般的であるが, これは通信遅延や単一障害点のリスクを増大させる要因となっている。本研究では, 組込み機器上でもスマートコントラクトを実行できるようにするために, スマートコントラクトの計算処理自体をデバイス近傍のオフチェーン環境で実行し, その実行結果および正当性の証跡のみをブロックチェーンに記録する手法を提案する。この構成により, ブロックチェーンの持つ透明性と改ざん耐性を維持しつつ, 組込み機器における計算負荷の大幅な低減とリアルタイム性の向上を図る。またオフチェーン実行には DTVM（DeTerministic Virtual Machine）というスマートコントラクト専用言語である solidity を実行可能かつ高い実行速度を誇る仮想マシンを使用する。本論文では, 提案手法に基づいたプロトタイプシステムを構築し, 一般的な PC を用いた評価実験を行う。既存のフルオンチェーン実行の実行方式と比較し, 処理時間, メモリ使用量, ガスコストの観点から組込み機器上に適用可能であるか, またどのような問題点が発生するか検討する。

目次

図目次

第 1 章

はじめに

1.1 研究背景

近年,IoT 技術の急速な普及により, センサーデバイスやアクチュエータなどの組込み機器がネットワークを介して相互に接続される社会が実現しつつある。IoT 技術は, スマートシティ, 自動運転, サプライチェーン管理, 産業用ロボットなど, 多岐にわたる分野で基盤技術として活用されている。これらのシステムでは, 膨大な数のデバイス間でデータの授受や価値の取引が自律的に行われることが期待されており, その信頼性を担保する技術としてブロックチェーンおよびスマートコントラクトが注目を集めている [1, 2, 3]。スマートコントラクトは, あらかじめ定義された契約条件をプログラムとして記述し, 特定の条件が満たされた際に自動的に実行する仕組みである。中央集権的な管理者を介さず, 分散ネットワーク上で実行されるため, プロセスの透明性や高い改ざん耐性を有している。しかし, 現在のスマートコントラクトの実装, 特に Ethereum に代表されるパブリックブロックチェーンにおいては, 契約の実行（計算処理）とデータの記録（合意形成）が不可分な「オンチェーン実行」が主流である。IoT 技術とブロックチェーンを組み合わせるには, 大きく分けて 3 つの課題が存在する。第一に, 計算資源の制約である。ブロックチェーンに参加する全ノードが同一の計算を重複して行う必要があるため, 計算コストが極めて高い。IoT 機器上でオンチェーン実行を行うには限られたストレージや計算資源といった厳しいリソース制限から実装が難しいと考えられている [3]。これに対して IPFS 等を用いてデータ保存をオフチェーンへ逃がすオンチェーン/オフチェーン併用モデルが提案されている [4]。第二に, 処理性能の限界である。現在主流である Ethereum のブロックチェーン上ではスマートコントラクトを実行する EVM という仮想マシンが存在しているが, ブロックチェーンの普及が進むにつれその処理性能はより高いものが求められる。ブロッ

クチェーン上ですべての処理を完結させてスケールさせることには限界があり, 高スループットが要求される用途ではオフチェーンで大部分の処理を行う必要性が指摘されている [5]。加えて, オンチェーンでの重複計算を避けるために計算をオフチェーンで実行し, オンチェーンでは結果検証のみを行う枠組みが提案されている [6]。第三に, 経済的コストである。実行のたびに「ガス代」と呼ばれる手数料が発生し [7], 頻繁にデータを更新する IoT デバイスにとって大きな負担となる。これらの課題から, 現状の IoT システムは, 潤沢な計算リソースを持つクラウドサーバ上で代理実行されるケースが一般的である。しかし, クラウド依存の構成は, 通信遅延の増大や, 特定のサーバがダウンした際にシステム全体が停止する単一障害点の問題を内包している [3]。真の意味で自律的な IoT エコシステムを構築するためには, ネットワークの末端に位置する組込み機器そのものが, スマートコントラクトの実行能力を持つことが不可欠である。

1.2 研究目的

本研究の目的は, リソース制限の厳しい組込み機器上において, スマートコントラクトを効率的に実行可能とするオフチェーン実行基盤を構築することである。本研究では, スマートコントラクトの複雑な論理演算をブロックチェーンの外側（オフチェーン）で実行し, その実行結果のみをブロックチェーンへ接続・保存する機構を提案する。またオフチェーンでの実行には EVM と比較して実行速度の速い DTVM を採用することで, より処理性能を向上させる。これにより, ブロックチェーンが持つ高いセキュリティ性を維持しつつ, オンチェーン実行のボトルネックを解消することを目指す。本研究を通じて, 組込み機器が自律的に契約を履行できる環境を示すことで,IoT 社会における高度な分散型自動化システムの実現に寄与する。

1.3 本論文の構成

第 2 章では本研究に必要な前提知識について詳しく述べる。第 3 章ではオフチェーン実行とオンチェーンでの保存を組み合わせる機構について述べる。第 4 章では作成した接続機構を使用して比較して評価実験を行い, 組込み機器への適用性を評価する。第 5 章では本研究のまとめと今後の課題について述べる。

第 2 章

前提知識

本章では, 本研究に関連する専門用語について解説する。

2.1 ブロックチェーン

ブロックチェーンとは, 分散型台帳技術の一種であり, 複数の参加者によって取引履歴を共有・管理する仕組みである。ブロックチェーンの特徴としてまず図 2.1 のようなデータの保存方法である。取引データはブロックと呼ばれる単位にまとめられ, 各ブロックは暗号学的ハッシュ関数によって直前のブロックと連結される。この構造により, 過去のデータを改ざんすることが極めて困難となり, 高い耐改ざん性が実現されている。

ブロックチェーンのもう一つの特徴として, 図 2.2 のように中央管理者を必要としない点が挙げられる。従来の中央集権型システムでは, 単一の管理主体がデータを管理していたが, ブロックチェーンではネットワーク参加者が全員取引を実行して確認することで取引の正当性を検証する。また中央集権型システムでは中央のサーバがエラーで止まってしまった場合, システム全体が停止してしまうが, ブロックチェーンの場合は一部のシステムが停止しても, システム全体は動き続けることができる。

このような特性から, ブロックチェーンは暗号資産だけでなく, サプライチェーン管理や IoT 分野など, 改ざん耐性や透明性が求められる分野への応用が進められている。ブロックチェーンプラットフォームの代表例としては Ethereum が有名である [7]。

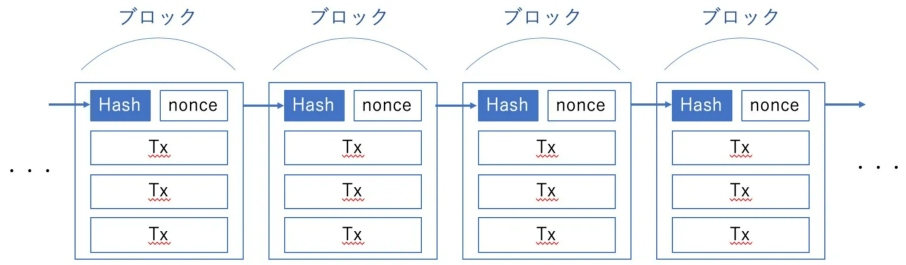


図 2.1: ブロックチェーンのデータの保存方法

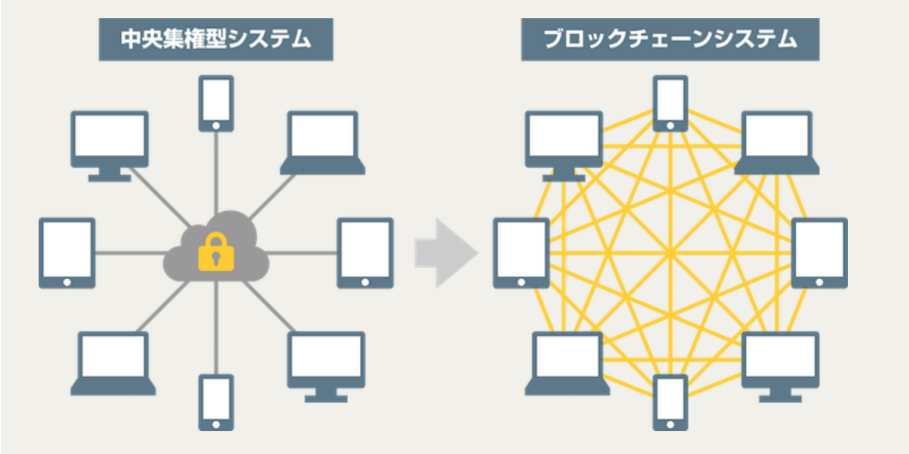


図 2.2: ブロックチェーンの概念図

2.2 スマートコントラクト

スマートコントラクトとは, ブロックチェーン上で実行されるプログラムのことであり, あらかじめ定義された条件が満たされた場合に, 自動的に処理を実行する仕組みである。Ethereum に代表されるブロックチェーンプラットフォームでは, スマートコントラクトが EVM という仮想マシン上で実行され, 取引の自動化や信頼性の高い処理を実現している。開発にはスマートコントラクト専用言語である Solidity が使われている。

従来の中央集権型のようなシステム上での契約処理では, 第三者機関やサーバが仲介する必要があったが, スマートコントラクトを用いることで, 契約条件の検証から実行までをプログラムによって自動化できる。これにより, 契約の間で他の人物が入り込むことがなく人為的ミスの削減やコスト削減が可能となる。また契約が直接行われるため, 契約の透明性も高くなる。

一方で, スマートコントラクトは通常, ブロックチェーン上のノードやサーバ環境で実行されるため, 計算資源や実行コスト（Gas）に制約がある。実行コスト (Gas) とはスマートコント

ラクトの命令ごとにあらかじめ設定された数値であり, 複雑な処理ほどそのコストは高くなっている。これによってネットワーク上で一部の処理を無限ループさせたり, 複雑で重い処理を複数回実行させたりといった悪意のあるスマートコントラクトの実行を防ぎ, ネットワーク利用者の公平性を維持している。このガスコストを計算する処理はスマートコントラクトを実行する際に自動的に追加される。そのため組込み機器のようなリソース制約の厳しい環境で直接実行することを考えた場合, このガスコスト計算処理によってより必要なリソース量が増加し, 実行するのが困難となる可能性がある。また組込み機器が高頻度で外部のデータを取得する場合, スマートコントラクトもそれだけ高頻度で実行する必要があるため, ガスコストによって処理が実行されない場合がある。これらの点が本研究の課題背景の一つとなっている。

2.3 DTVM

DTVM（Deterministic Trusted Virtual Machine）は, 決定論的な実行を保証する WebAssembly（Wasm）ベースの仮想マシンであり, 主にスマートコントラクトの実行性能向上を目的として提案されている [8]。

DTVM の重要な特徴の一つは, 異なるハードウェア環境においても同一の入力に対して同一の実行結果を得られる決定論的実行を保証する点である。図 2.3 より DTVM は入力された Wasm コードを dMIR という独自の中間表現に変換しており, これによって入力された Wasm コードを決定論的なコードに変換している。ブロックチェーンでは複数のノードが同一の計算結果を得ることを前提としてシステム全体の正当性が保たれるため, 実行環境の決定論性は不可欠な要件である。DTVM はこの要件を満たす実行環境として設計されている。

また dMIR からマシンコードへのコンパイル方式も二種類備えており,FLAT モードではプログラムの最適化をほとんどせず, 呼び出されてからの応答性を重視しており, 関数の呼び出された回数に応じて FLAS モードという最適化レベルの高いコンパイル方式を行うことで, 実行速度を向上させている。

二つ目の特徴は,DTVM は WebAssembly を実行形式として採用しており, 軽量かつ高速な実行が可能である。DTVM 論文では,EVM と比較してスマートコントラクトの実行時間が大幅に短縮されることがベンチマークにより示されている [8]。さらに,c++ や Rust などの言語で記述されたプログラムに加えて,Solidity で記述されたスマートコントラクトも Wasm へ変換して実行可能とする互換性を備えており, 既存の EVM 向けスマートコントラクト資産を活用できる点も DTVM の研究成果として示されている。その際のガスコスト計算については

EVM の模倣または簡易モデルを使用している。

一方で,DTVM は実行結果の正当性検証や悪意ある実行に対する検出機構を提供しない。また, 実行結果をブロックチェーン上に保存する機能や, ネットワークを介した合意形成機構も備えていない。これらの機能は DTVM 論文においても実行基盤の範囲外として明確に分離されており, DTVM はあくまで決定論的かつ高速なプログラム実行マシンとして設計されている。

一般に WebAssembly（Wasm）ベースの仮想マシンはブラウザ環境に組み込まれ, クライアント側での安全かつ高速なコード実行に利用されることが多い。しかし本研究ではブラウザ用途ではなく, 仮想マシンを独立したオフチェーン実行基盤として利用する。すなわち, ブロックチェーンの合意層とは分離された形で, 高速なスマートコントラクト実行環境として DTVM を用いる。

Wasm ベースの高速実行環境は他にも存在するが, DTVM は Solidity で記述されたスマートコントラクトを Wasm へ変換して実行可能である点に特徴がある。この特性により, 既存の Ethereum 向けコントラクト資産を活用しつつ, その一部をオフチェーンで実行する構成が可能となる。この互換性の観点から, 本研究では他の Wasm 仮想マシンではなく DTVM を採用した。

また, 高速なスマートコントラクト実行エンジンとして, EVM の高速実装である evmone が存在する [9]。evmone は EVM 仕様に準拠した実装であり, オンチェーン実行モデルそのものを変更するものではない。DTVM 論文では evmone との性能比較が行われており, DTVM が実行時間の観点で優れた結果を示している [8]。

表 2.1 に示すように, EVM および evmone はスマートコントラクト実行とブロックチェーン上の状態更新・合意形成を一体として扱う実行環境である。一方で DTVM は, 合意形成や状態管理機構を持たず, 決定論的かつ高速なプログラム実行のみを担う実行基盤である。そのため DTVM を利用したオフチェーン実行を達成するためには DTVM での実行結果をブロックチェーンへ送信し, 保存する機構が必要である。

本研究ではこの DTVM をオフチェーン実行基盤として利用し, DTVM で得られた実行結果をブロックチェーンへ送信し, 状態として保存する接続機構を構築する。これにより,EVM によるフルオンチェーン実行では困難であったリソース制約環境におけるスマートコントラクト実行を実現することを目指す。

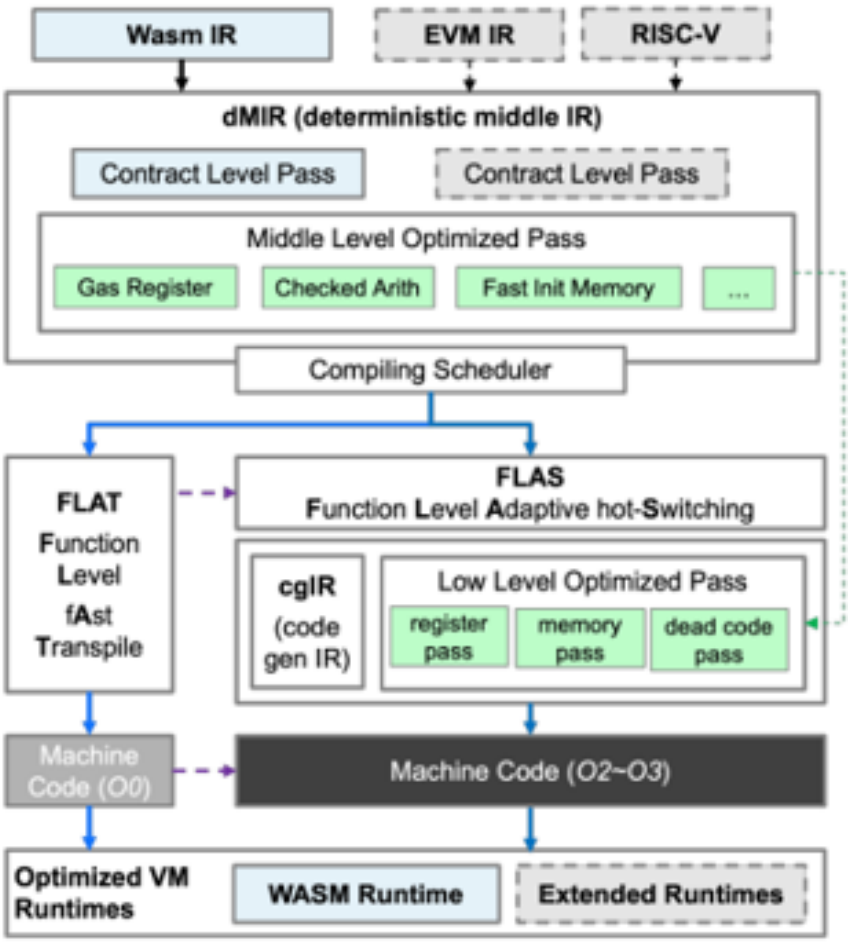


図 2.3: DTVM のコンパイル方式図

表 2.1: EVM,evmone,DTVM における提供機能の比較

項目	EVM	evmone	DTVM
主な目的	ブロックチェーン上での SC 実行と状態更新	EVM 命令の高速実行	決定論的・高速なプログラム実行
実行場所	ブロックチェーン上 (オンチェーン)	ブロックチェーン上 (オンチェーン)	ブロックチェーン外 (オフチェーン)
合意形成との関係	合意形成・状態遷移と一体	合意形成・状態遷移と一体	合意形成を提供しない
実行結果の扱い	実行結果がそのまま状態として保存	実行結果がそのまま状態として保存	実行結果の保存は外部に委ねる
決定論性	保証される	保証される	保証される
実行形式	EVM バイトコード	EVM バイトコード	WebAssembly（Wasm）
実行性能	比較的低速	高速 (EVM 比)	高 速　(EVM,evmone 比)
ガスコスト計算	あり	あり	あり（EVM の模倣または簡易モデル）
セキュリティ（検証）	ネットワーク全体で検証（合意に組込まれる）	ネットワーク全体で検証（合意に組込まれる）	実行結果の正当性検証は未提供

第 3 章

提案手法

本章では, 本研究で提案する手法について述べる。

3.1 オフチェーン実行 + オンライン同期:DTVM とブロックチェーンの接続機構

本研究ではスマートコントラクトの処理をすべてブロックチェーン上で実行するのではなく, オフチェーンで処理を行い, その結果のみをブロックチェーンに送信して保存する方式を採用する。

従来のフルオンチェーン型の場合, 実行命令であるトランザクションを受け取るとブロックチェーンにつながっているすべてのノード上で対象のスマートコントラクトが実施される。しかし本研究ではその実行するスマートコントラクトの処理をオフチェーン, いわゆるローカルで DTVM を用いて実行し, その結果のみをブロックチェーン上に保存する手法を提案する。現状 DTVM は決定論的なプログラム実行環境であり, ブロックチェーンに接続してデータを保存する機能は持ち合わせていない。そのため DTVM を呼び出し, その実行結果を取得してブロックチェーンに送信する仕組みを作成する必要がある。

この手法によるメリットはスマートコントラクトで実行される部分が少なくなるため, ガスコストを減らすことができる。ガスコストが減ることで, 高頻度でプログラムを呼び出すことができる。またガスコストを計算する処理の数も減るためプログラム全体の実行時間が短くなる可能性がある。

一方でデメリットとして考えられることは, セキュリティ性の低下である。本来ブロックチェーンとはスマートコントラクトの実行をすべてのノードで行い, その結果をお互いに監視しあう

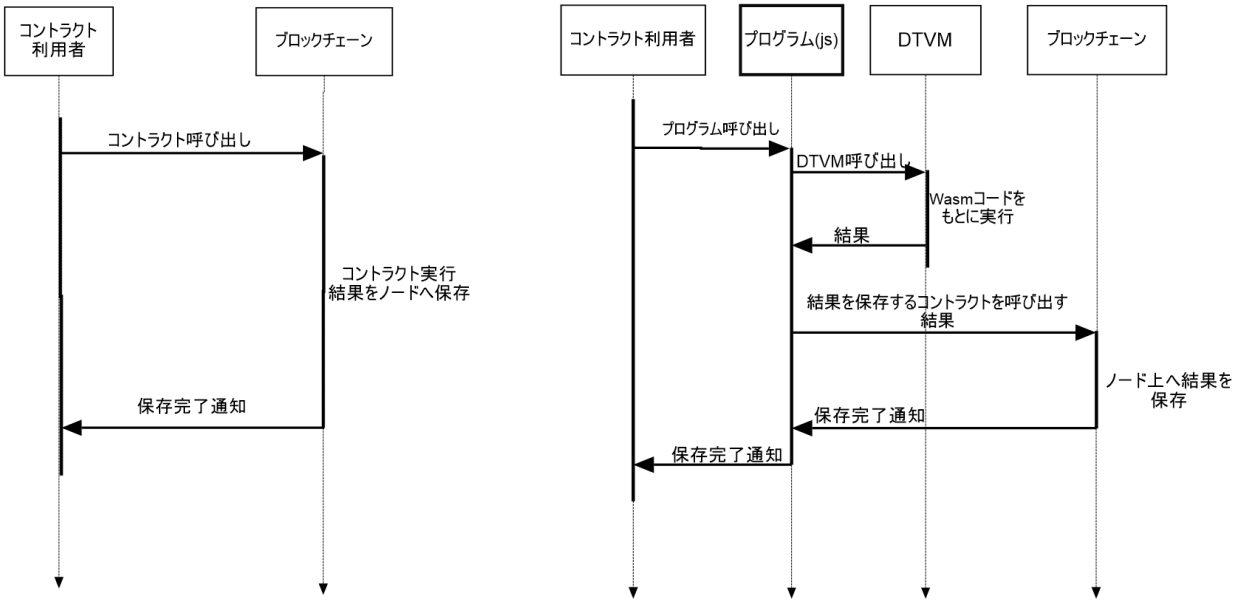
ことでその正当性を検証し, データをブロックにして保存する。しかしこの仕組みではコントラクトの処理をローカルで実行し, その結果のみをブロックチェーン上で保存する。そのため仮にローカルで処理を実行する部分を改ざんされたとしても, ブロックチェーン上では改ざんされたプログラムから送信される結果のみ確認することになるため, IoT 機器内でプログラムが改ざんされてしまっていることに気づくことができなくなっている。また, ローカルで実行した結果をブロックチェーンに接続する際のオーバーヘッドの大きさによっては実行時間全体が逆に長くなってしまう可能性も考えられる。
本研究では実行時間に焦点を当てて, この手法について評価する。セキュリティ面については第 5 章で今後の課題として扱う。

3.2 具体的な仕組みについて

本研究では hardhat というブロックチェーン開発環境を使用する。javascript で作成したプログラム内で DTVM に実行させる Wasm ファイルを指定して実行させ, あらかじめブロックチェーンにデプロイしておいたデータを保存するコントラクトを呼び出すことで,DTVM の実行結果のみをブロックチェーンに保存する。図 3.1 の 10 行目にある runDTVM という関数の中で,DTVM の実行バイナリに Wasm ファイルを渡し, 実行する関数とその引数を渡すことで DTVM を呼び出している。22 行目以降の main の中でこの runDTVM を呼び出し, その結果をブロックチェーンにデプロイしておいた保存用コントラクト storeResult を呼び出すことで, ブロックチェーン上に保存している。従来のフルオンチェーン型である図 3.2(a) と今回提案する手法である図 3.2(b) を比較すると, ブロックチェーンで動作する部分が少なくなっていることが分かる。本研究ではこのブロックチェーン上の動作が少なくなる事によるメリット, デメリットを評価する。

```
console.log("cwd =", process.cwd());

const { ethers } = require("hardhat");
const { execSync } = require("child_process");

// Hardhat ローカルノードにデプロイ済みのコントラクト
const CONTRACT_ADDRESS = "0
    x5FbDB2315678afecb367f032d93F642f64180aa3";

//DTVMに実行するWasmファイルを渡して実行させ,結果を取得する関数
async function runDTVM(n) {
  const cmd = `../dtvm ./fibo.wasm -f run --args ${n}`;
  const output = execSync(cmd).toString().trim();

  console.log("DTVM output:", output);

  if (output.includes(":")) {
    return Number(output.split(":")[0]);
  }
  return Number(output);
}

async function main() {
  const Store = await ethers.getContractFactory("ResultStorage");
  const store = await Store.attach(CONTRACT_ADDRESS);

  //DTVM呼び出し
  const r = await runDTVM(5);

  //保存用コントラクトを呼び出して,結果をブロックチェーン上に保存
  const tx = await store.storeResult(r);
  const receipt = await tx.wait();
}

main().catch(console.error);
```

図 3.1: DTVM を呼び出して結果をオンチェーン保存する実験スクリプト



(a) フルオンチェーン　　(b) オフチェーン実行＋オンチェーン同期

図 3.2: 実行方式の比較

第 4 章

評価実験

本章では, 本研究で実施した評価実験の内容, 実験環境, そしてその結果と考察について述べる。

4.1 実験概要

本実験ではフィボナッチ数列を計算するプログラムを対象として, 2 つのパターンでその実行時間, メモリ使用量, そして必要なガスコストを計測した。実行は組込み機器上ではなく PC 上で行い, 実行結果をもとに組込み機器への適用性を評価した。実行するパターンは以下の 2 つである。

4.1.1 パターン 1 :EVM によるフルオンチェーンでの実行

従来のフルオンチェーン型の例として,Solidity で書いたフィボナッチ数列のスマートコントラクトをブロックチェーン上で実行する。

4.1.2 パターン 2 :DTVM によるオフチェーン実行 (実行する Wasm ファイルの元の言語は c++)

DTVM を用いた提案手法の仕組みを使い,c++ で書いたプログラムを Wasm バイトコードにコンパイルして実行する。

表 4.1: 実験環境（ハードウェア）

項目	内容
実行環境	PC（WSL2 上で実行）
OS	Windows 11 Home
プロセッサ	Intel Core Ultra 5 125U
メモリ	16 GB
アーキテクチャ	x86_64

表 4.2: 実験環境（ソフトウェア）

ソフトウェア	バージョン
Ubuntu (WSL2)	24.04.3 LTS
Linux Kernel	6.6.87.2-microsoft-standard-WSL2
DTVM	v1.0.0
Hardhat	v2.28.3
Node.js	v22.22.0
npm	v10.9.4

4.2 実験環境

本研究で使用した実験環境を, ハードウェア環境とソフトウェア環境に分けて表 4.1 および表 4.2 に示す。

4.3 実験内容

フィボナッチ数列３０から４０を順番に計算するプログラムを対象にプログラム全体の実行時間, それぞれのプログラムの異なる部分である仮想マシンの呼び出し部分のみの実行時間, 最大メモリ使用量, また数列４０を計算して保存する際のガスコストを計測した。実行するフィボナッチ数列のプログラムはパターン１は Solidity で作成した図 4.1, パターン２は c++ で作成した図 4.2 を Wasm にコンパイルしたものを使用した。またフィボナッチ数列の引数を３０から４０に変化させて呼び出すためのプログラムはどちらのパターンも javascript で作成して図 4.3,4.4 のようになった。プログラム全体の実行時間はこの二つのプログラムの全体の実行時間のことで, 仮想マシンの呼び出し部分のみの実行時間というのは main 内の for ループで囲

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract FibonacciStorage {
uint256 public lastResult;
uint256 public lastInput;

  function fibonacci(uint256 n) public pure returns (uint256) {
      if (n == 0) return 0;
      if (n == 1) return 1;

      uint256 a = 0;
      uint256 b = 1;

      for (uint256 i = 2; i <= n; i++) {
          uint256 c = a + b;
          a = b;
          b = c;
      }

      return b;
  }

  function computeAndStore(uint256 n) external {
      uint256 result = fibonacci(n);
      lastInput = n;
      lastResult = result;
  }
}
```

図 4.1: パターン 1 用フィボナッチ数列 (Solidity)

まれている範囲のみの実行時間のことである。実行結果の値についてはそれぞれのパターンで 30 回ずつプログラムを実行してその結果の平均値で比較した。

それぞれの測定方法についてはプログラム全体の実行時間とメモリ使用量については /usr/bin/time -v コマンドを使用して計測した。仮想マシンの呼び出し部分の実行時間については Node.js が提供する高精度タイマである performance.now() を用いて計測した。ガスコストは, 各トランザクション実行後に取得されるトランザクションレシートに含まれる gasUsed を用いて計測した。この値は Ethereum 仮想マシンにおける命令実行量を表す指標であり,Hardhat のローカルネットワーク上でも Ethereum と同等の形式で取得可能である。

```
extern "C" unsigned long long run(unsigned long long n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    unsigned long long prev = 0;
    unsigned long long curr = 1;

    for (unsigned long long i = 2; i <= n; ++i) {
        unsigned long long next = prev + curr;
        prev = curr;
        curr = next;
    }

    return curr;
}
```

図 4.2: パターン 2 用フィボナッチ数列

```
import pkg from "hardhat";
import { performance } from "node:perf_hooks";

const { ethers } = pkg;

// Hardhat ローカルノードにデプロイ済みのコントラクト
const CONTRACT_ADDR = "0x5FbDB2315678afecb367f032d93F642f64180aa3
    ";

async function main() {
  const Fib = await ethers.getContractFactory("FibonacciStorage");
  const fib = await Fib.attach(CONTRACT_ADDR);
  const start = performance.now();

  for (let n = 30; n <= 40; n++) {
    const tx = await fib.computeAndStore(n);
    const receipt = await tx.wait();

    const result = await fib.lastResult();

    console.log("n =", n);
    console.log("result =", result.toString());
    console.log("gasUsed =", receipt.gasUsed.toString());

  }
  const end = performance.now();
  console.log("wallTime(ms) =", (end - start).toFixed(2));
}

main().catch(console.error);
```

図 4.3: パターン 1 用実行プログラム

```
console.log("cwd =", process.cwd());

const { ethers } = require("hardhat");
const { execSync } = require("child_process");


// Hardhat ローカルノードにデプロイ済みのコントラクト
const CONTRACT_ADDRESS = "0
    x5FbDB2315678afecb367f032d93F642f64180aa3";

async function runDTVM(n) {
  const cmd = `../dtvm ./fibo.wasm -f run --args ${n}`;
  const output = execSync(cmd).toString().trim();

  console.log("DTVM output:", output);

  if (output.includes(":")) {
    return Number(output.split(":")[0]);
  }
  return Number(output);
}


async function main() {
  const Store = await ethers.getContractFactory("ResultStorage");
  const store = await Store.attach(CONTRACT_ADDRESS);
  const start = performance.now();

  for (let n = 30; n <= 40; n++) {
    const r = await runDTVM(n);
    const tx = await store.storeResult(r);
    const receipt = await tx.wait();
    const result = await store.lastResult();

    console.log("n =", n);
    console.log("result =", result.toString());
    console.log("gasUsed =", receipt.gasUsed.toString());
  }
  const end = performance.now();
  console.log("wallTime(ms) =", (end - start).toFixed(2));
}

main().catch(console.error);
```

図 4.4: パターン 2 用実行プログラム

4.4　実験結果

結果は図 4.5,4.6,4.7,4.8 のようになった。プログラム全体の実行時間とメモリ使用量は両パターン間でほとんど差は見られなかったが, 従来のフルオンチェーンで実行するパターン１に対して, オフチェーン実行 + オンチェーン同期型であるパターン２のほうが実行時間は約０.１秒ほど長く, メモリ使用量は約２０００ kbyte 多くなった。またパターンごとに異なる部分である仮想マシン呼び出し部分の実行時間は約０. １秒ほどパターン２の方が長くなっており, この差はプログラム全体の実行時間の差とほとんど一致していた。そしてプログラムの実行に必要なガスコストについてはパターン２がパターン１の約半分となった。

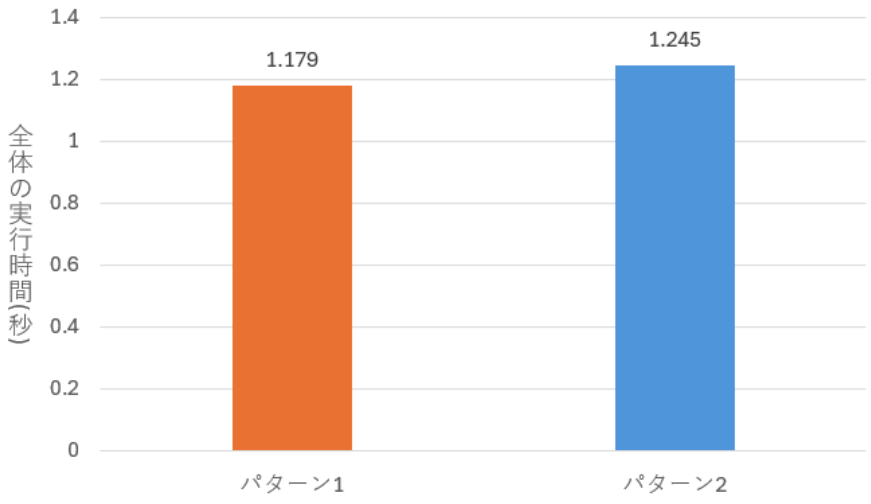


図 4.5: プログラム全体の実行時間の比較

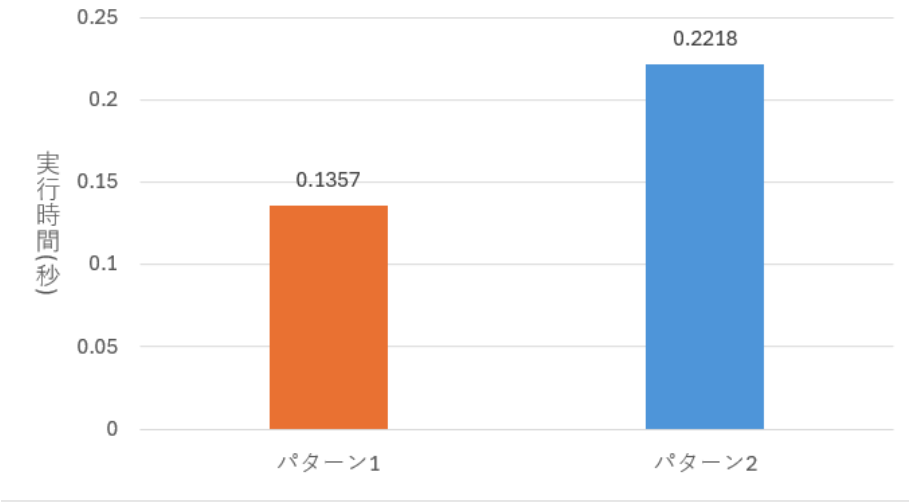


図 4.6: 仮想マシン呼び出し部分の実行時間の比較

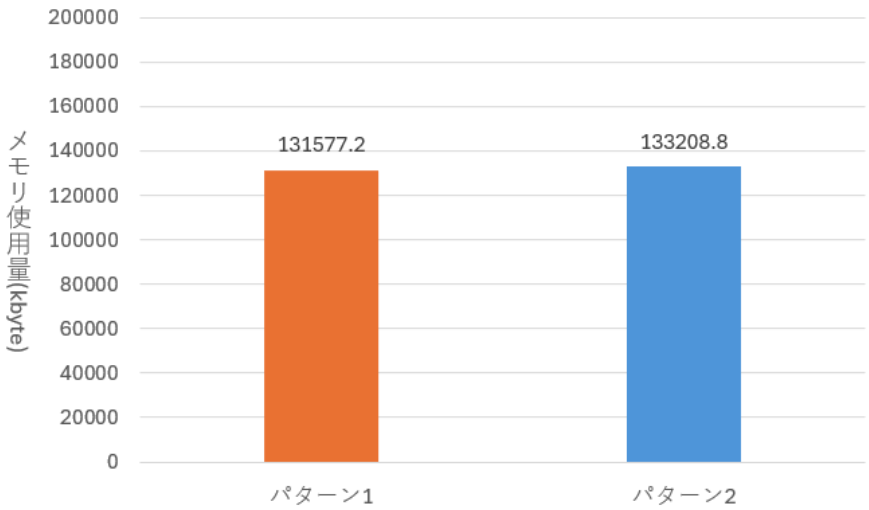


図 4.7: メモリ使用量の比較

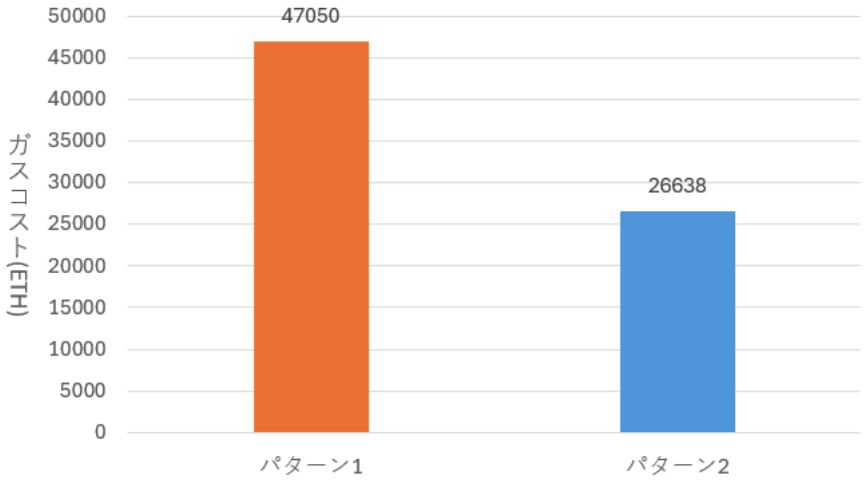


図 4.8: ガスコストの比較

4.5 考察

実行前の予想では少なくとも実行時間とガスコストの２つはパターン２のオフチェーン実行の方が優れた結果になると予想していたが, 実行時間についてはパターン１の方が短くなった。これは呼び出し部分の実行時間の比較から, パターン１はオンチェーン上のコントラクトを一度呼び出すとフィボナッチの計算からチェーン上に保存するまでをまとめて実行するのに対して, パターン２ではローカルにある DTVM を呼び出した後にオンチェーン上の保存用コントラクトを呼び出すという二回の手順を行っているオーバーヘッドが大きいためであると考えられる。それによって実行時間がわずかに増加し, 必要なメモリ使用量も大きくなっていると考えられる。

この問題を解決するには DTVM のローカル実行呼び出しとブロックチェーンへの接続を担う間の機構を一括で実行できるようなシンプルで単純なものにする必要があるだろう。例えば本実験では図 3.2(b) のように DTVM からの実行結果を一度プログラムに渡して, プログラムがブロックチェーンにその結果を送信したが, DTVM が実行する Wasm ファイルにブロックチェーンに結果を送信する部分までを含ませることが出来れば,DTVM から直接ブロックチェーンに結果を保存することが可能になり, 本実験で 2 回の手順で行っていた部分が一回で済み, オーバヘッドが減少すると考えられる。

本研究ではオフチェーン実行＋オンチェーン同期型を組込み機器上でも実行できるようにすることが目的である。実験結果より本実験の環境下において今回作成した接続機構はラズパイ程度のリソース量があれば十分実行可能であることが確認できた。しかし今回使用したフィボナッチ数列のプログラムは比較的単純であり, より本格的な複雑な処理を行うプログラムの場合はどれほどオーバーヘッドが大きくなるのか検証する必要がある。その大きさの度合いによっては組込み機器への適用は難しいだろう。そのため実行時間, メモリ使用量の２つを改善するためにも新たな DTVM とブロックチェーンの接続機構を作成する必要がある。

ただガスコストについては大きく差がでており, オフチェーン実行を実現することが出来れば今までブロックチェーン上ではできなかった複雑な処理を実行することができたり, 高頻度でプログラムを実行出来たりするといったメリットが出てくると思われる。

第 5 章

おわりに

5.1 まとめ

本研究では DTVM という Wasm ベースで開発言語の多い仮想マシンを使用して, 従来のフルオンチェーンによるスマートコントラクト実行に代わる, オフチェーンでの DTVM 実行 + 結果のみブロックチェーンに保存する仕組みを作成し, 組込み機器への適用性を評価するために実行時間, メモリ使用量, そしてガスコストを計測して比較した。実験結果では実行時間とメモリ使用量が予想とは異なり, 本研究で提案した手法よりも従来のフルオンチェーン型の方が優れた結果となった。この結果から本研究で作成した機構はラズパイ程度のリソース量があれば十分実行可能であるが, より複雑な処理を行うプログラムの場合にオーバーヘッドがどれほど増加するか再検証する必要があることを確認した。またオーバヘッドの増加量が大きい場合に備えて, 作成方法を変えて新たな接続機構を作る必要があることを確認した。またそれと同時にガスコストについてはオフチェーン実行の方が大きく優れていたため, ガスコスト削減の観点でのオフチェーン実行 + オンチェーン同期型の有用性を確認した。

5.2 今後の課題

本論文の実験では簡単なプログラムを対象にシンプルな比較実験を行ったが,IoT 機器でのブロックチェーン動作を検証するためにも今後はより具体的なユースケース (例:IoT 機器でのデータのやり取りを目的としたブロックチェーンシステム [10]) に対しても検証する必要がある。そして本実験では PC 上で比較実験, 評価を行ったが, 実際にラズパイや IoT 機器等で同様の実験を行い, 本実験での結果との差異がどの程度あるのかも検証する必要がある。リソース量によって実行性能やネットワーク接続が不安定になる可能性があるだろう。

またオフチェーン実行によるデメリットであるセキュリティ性の低下と, 状態保存がされないことから結果を再利用することができないという二つの点も解決する必要がある。セキュリティ面について本研究の方式では, オフチェーン実行部分の改ざん検知が課題となる。従来のフルオンチェーンの場合, 全ノードでコントラクトを検証し, 実行の正当性を確認することができる。しかし本研究の方式では実行部分であるオフチェーンを改ざんされた場合に, 直接実行部分を確認しない限り発見することが出来ない。これに対し, 実行環境を信頼実行環境（TEE）で保護し, 合意層と実行層を分離するアーキテクチャ [11] や, あるいは zk-SNARK 等によるオフチェーン計算の証明生成とオンチェーン検証 [12] といったアプローチが考えられる。

状態保存については,DTVM はあくまでも軽量かつ高速なプログラム実行環境であり, ブロックチェーン的な動作は未提供である。そのため前回の実行結果を利用する場合はその結果をローカルで保存しておく必要がある。フルオンチェーンの場合は前回の結果をチェーン上に状態として保存しておき, 再利用が可能だが,DTVM はローカル実行でありかつ状態保存機能は未提供のため, 前回の結果をローカルに保持しておき, 再利用できるような仕組みを考案, 作成する必要がある。

参考文献

[1] K. Christidis and M. Devetsikiotis, *Blockchains and Smart Contracts for the Internet of Things*, IEEE Access, vol. 4, pp. 2292–2303, 2016. https://ieeexplore.ieee.org/document/7467408

[2] Ali Dorri, et al. *Blockchain for IoT Security and Privacy: The Case Study of a Smart Home*, Conference Paper · March 2017 https://people.cs.pitt.edu/~mosse/courses/cs3720/Blockchain_for_IoT_Security_and_Privacy.pdf

[3] H.-N. Dai, Z. Zheng, and Y. Zhang, *Blockchain for Internet of Things: A Survey*, https://arxiv.org/pdf/1906.00245

[4] Pooja Khobragade *On-chain Off-chain Blockchain Model for IoT using IPFS*, IET Conference Proceedings, July 2023 https://www.researchgate.net/publication/373003375_On-chain_off-chain_blockchain_model_for_IoT_using_IPFS

[5] J. Poon and T. Dryja, *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*, 2016. https://lightning.network/lightning-network-paper.pdf

[6] J. Teutsch and C. Reitwießner, *A Scalable Verification Solution for Blockchains*, https://arxiv.org/pdf/1908.04756

[7] G. Wood, *Ethereum: A Secure Decentralised Generalised Transaction Ledger*, Ethereum Yellow Paper, 2014. https://ethereum.github.io/yellowpaper/paper.pdf

[8] Wei Zhou, et al. *DTVM: REVOLUTIONIZING SMART CONTRACT EXECUTION WITH DETERMINISM AND COMPATIBILITY*, June 10, 2025 https://arxiv.org/pdf/2504.16552v2

[9] Ipsilon *evmone : Fast Ethereum Virtual Machine implementation*, https://github.com/ipsilon/evmone

[10] Kenta Kawai, Wu Yuxiao, Yutaka Matubara, and Hiroaki Takada *BLOCKCHAIN-*

BASED DEMAND-SUPPLY MATCHING SYSTEM FOR IOT DEVICE DATA DISTRIBUTION , 2024 hhttps://aircconline.com/csit/papers/vol14/csit142408.pdf

[11] R. Cheng, et al., *Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution*, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8806762

[12] Alvaro Alonso Domenech, Jonathan Heiss, Stefan Tai *Servicifying zk-SNARKs Execution for Verifiable Off-chain Computations*, https://arxiv.org/pdf/2404.16915