Vol.2021-CSEC-93 No.14
Vol.2021-IOT-53 No.14
2021/5/14

FPGAによるソフトウェア解析環境「Iana」の提案

金谷 延幸[1,a)]　津田 侑[1]　高野 祐輝[1,2]　藤原 吉唯[1]　伊沢 亮一[1,b)]　井上 大介[1]

概要：
組み込み機器開発では，実機やソフトウェアエミュレーションなど，テストやデバッグなどの内容に応じ複数の実装形態を使い分ける．各実装形態には一長一短があり，実機では解析手段が限られるため不具合発生時の状態が把握できない場合があり，エミュレーションでは詳細に追跡したくてもその不具合が再現せず追跡できない場合がある．そこで我々は，複数の実現環境における CPU の任意時点における状態を再現し，相互にマイグレーション可能なソフトウェア解析環境 Iana（アイアナ）を提案する．Iana の特徴は CPU やメモリ，I/O の状態をすべて記録し，任意時点の状態と挙動を再現・変更可能な解析専用 CPU にあり，本稿では CPU を FPGA にて実装した．さらに，任意時点での実行状態を QEMU や評価ボードに書き戻し，デバッギング機能とともに解析を再開することができる．Iana を用いることで，解析対象における重要なイベントの絞り込みを行い，またその前の状態に遡ることが可能であり，組み込み機器や IoT 機器などのファームウェア解析にて適切な手段の選択が可能となる．

キーワード：FPGA, ソフトウェア解析，組み込み機器，IoT

FPGA-based Software Analyzer: a Design Proposal

Abstract:
Embedded-software developers choose real embedded devices or software emulation as a debugging platform for their developing programs, depending on their purpose of debugging at the time. This is because those platforms have both advantages and drawbacks (e.g., *real devices basically do not possess good debugging functionality* and *a software emulator could cause some semantic gaps between a real device and emulation*). To fully leverage advantages of every platform, this paper presents *Iana*, a system featuring a CPU implemented on FPGA for migrating execution-state of software from a debugging platform of FPGA to another (QEMU or a real embedded device) and vise versa. The execution-state includes CPU registers, memory, and I/O. As a usage example of Iana, developers debug their programs on a real device until a bug is found, and then they continue the debugging on FPGA with better debugging functionality after the migration. Namely, Iana enables developers to select any execution-state in migration (e.g., the execution-state after 500 instructions executed from the beginning). This is achieved by tracing all executed instructions of software. The experiments in this paper using two software samples confirm that a prototype of Iana correctly migrated execution-states from a RISC-V implemented on FPGA to QEMU and a real RISC-V.

Keywords: FPGA, Software Analysis, Embedded computer, IoT

1. はじめに

組み込み機器や IoT 機器を悪用したサイバー攻撃の被害が顕著となっている．組み込み機器や IoT 機器による攻撃では，対象機器のファームウェア（組み込み機器で実行されるソフトウェア）に存在する脆弱性を狙い拡散するマルウェアを利用することで行われるため，それらの機器におけるデバッグ・テストや，ファジングなどの脆弱性検査，マルウェア挙動分析など，ファームウェアの動的な解析環境が非常に重要となる．

一般的な PC やサーバにおけるデバッグ・テストや，脆

[1] 国立研究開発法人 情報通信研究機構
National Institute of Information and Communications Technology
[2] 大阪大学 大学院工学研究科
Graduate School of Engineering, Osaka University
[a)] kanaya@nict.go.jp
[b)] isawa@nict.go.jp

弱性検査，マルウェアの挙動分析などには，それらの対象となるソフトウェアを実行し，その実行状態を収集する動的解析環境は，解析目的に応じた複数の実装アーキテクチャが提供される．例えば，実機上の OS デバッグ機構を利用したり，ソフトウェアエミュレーションを使う，仮想マシンを使うなど様々な実装が提供されてきた．一方の組み込み機器における動的解析環境は，実機もしくはソフトウェアエミュレーションによる解析など，実装手段が限定される．また，用いられている CPU は非常に安価で低速かつメモリも限られ，HDD や SSD のような二次記憶装置を持たない場合も多いためログを残すこともできず，実機での動的解析は非常に制限される．また，組み込み機器のファームウェアで発見された不具合の調査では，不具合発生時の状況だけでなく前後の実行履歴の把握が必要となる．不具合をエミュレーションにて実行履歴を追跡した場合，微妙なタイミングの違いにより，不具合が再現せず詳細な解析が不能となる場合がある．

そこで我々は，複数の実装環境における CPU の任意時点における状態を再現し，相互にマイグレーション可能な Iana（アイアナ）を提案する．我々は CPU，メモリ，I/O の状態をすべて記録し，任意時点の状態と挙動を再現・変更可能な解析専用 CPU を FPGA （Field-Programmable Gate Array）にて実装した．さらに，この任意時点での実行状態を QEMU や評価ボードに書き戻し，実行を再開することができる．Iana による解析は，解析対象における重要なイベントの絞り込みを行い，またその前の状態に遡ることが可能であり，組み込み機器や IoT 機器などのファームウェア解析にて適切な手段の選択が可能となる．さらに，実際の機器や QEMU など他の実装形態に対して相互にマイグレーション可能であり，FPGA で収集された状態を元に QEMU にてその先の再実行も可能であり，また実機で発生した不具合からの FPGA での再実行も可能である．このような，実機に準じたハードウェア解析環境を含む，相互にマイグレーション可能なハイブリッドな動的解析環境を実現することで，組み込み機器における不具合や脆弱性の解析を容易にし，セキュリティリスクを軽減することができると考える．

我々の貢献を以下に示す．

- 実際の CPU における任意時刻の状態を復元し，状態を書き戻すことで再実行する機構を提案した．この機構には，異なる実装形態に対して汎用性がある．
- 上記を実証するため, 異なる実装形態（FPGA, QEMU, 実機評価ボード） にて実現し，相互マイグレーションを確認した．

本論文の構成は，まず 2 章にて関連研究について紹介し，3 章にて我々の手法 Iana におけるマイクロプロセッサに対する状態のモデル化について述べ，次に 4 章にて Iana システムの設計を説明する．さらに，5 章にて今回試作した Iana システムの PoC 実装について述べ，さらに 6 章にて実際にマイグレーションのケーススタディとその評価について記述し，7 章にて結論と今後の課題について述べる．

2. 関連技術

本章では，我々が対象とする組み込み機器におけるソフトウェア解析技術として，小規模な組み込み機器での動的解析を実現するソフトウェアデバッグ技術について述べる．また，動的解析として現在主流であるソフトウェアエミュレーションに関する技術について述べ，さらに FPGA におけるソフトウェア・ハードウェア統合開発環境におけるデバッグ技術について述べる．

2.1 実機におけるデバッグ

より小規模な CPU におけるソフトウェアの挙動を扱う組み込み機器のデバッグ・テストでは，デバッグ専用ハードウェア In-Circuit Emulator （ICE）を使う方法，JTAG 等のデバッグインタフェースを使う方法などがある．ICE は，CPU 等のソケットに差し込み，CPU の代わりとして動作する，CPU ベンダーから提供されるデバッグ機器である．CPU をハードウェアレベルでエミュレートする必要があるため，非常に高価で利用者は限定され，一般的な利用が困難であった．最近の組み込み用のプロセッサの多くは，CPU 内にデバッグ機構を備え，Joint Test Action Group が制定する JTAG インタフェース [1] やソフトウェアデバッグ向けに最適化した Serial Wire Debug （SWD） [2] にて接続しデバッグが可能である．JTAG や SWD を使ったデバッグ機構は非常に安価であるが，CPU に備わるデバッグ機構に依存し，また通信速度等に制限があるため，実行のトレース情報をリアルタイムで常時出力するようなことは難しい．

2.2 エミュレーションを用いたデバッグ

マルウェア解析で用いられるツールとして QEMU がある．これは，様々な CPU やその周辺機器，マスク ROM（BIOS） 等をソフトウェアエミュレーションし，得られた実行トレースを保存し挙動解析に利用することができる．エミュレーションによる解析は非常に強力であり，すべての挙動に関する情報を取得できるため，独自のマルウェア検出アルゴリズムや解析機構の実装に用いられる．また，エミュレーションは標準的な PC で実行可能であり，専用のハードウェアを必要とせず，解析環境も容易に構築できる．当初は，実行速度の面で劣っていたが，近年の PC の高性能化によって，十分な許容範囲での速度を確保することができるようになった．エミュレーションでは再現しない不具合もあり，実機でのデバッグと併用する必要がある．

2.3 FPGA 開発でのデバッグ

特に，近年のハードウェア開発を伴う組み込み機器では，ハードウェアとソフトウェアの 2 つの領域における開発を統合的に利用できるデバッグ環境に関する研究開発が行われている．Cho らは FPGA が実装された PCI-e カードとそのドライバの開発の際に，FPGA のシミュレーションと PC 側のドライバのエミュレーションを統一的に行う手法を提案している [3]．また，Lee らは QEMU を利用した SoC の 3D グラフィックファームウェア開発における開発手法を提案している [4]．SoC 開発において，初期段階ではエミュレーションを用い，ハードウェア開発が進展すると実機によるデバッグを行い，見つかった不具合を修正するためエミュレーションによるデバッグに戻すサイクルが繰り返される．この開発サイクルを円滑にするため，実機とエミュレーションという 2 つのデバッグ手段の統合が試みられており，Nakamura らは， SoC におけるエミュレーションと FPGA シミューレータをシームレスに取り扱う統合環境を提案している [5]．この開発サイクルにおいて，一方で見つかった不具合を他方で再現しようとしても，実行を最初から行う必要があり，その不具合が再現するとは限らない．このため，実行の再現とマイグレーションを実現した我々の手法により，開発サイクルの効率化を期待できる．

3. 状態と遷移のモデル化

我々の Iana では，組み込み機器におけるソフトウェア動的解析に必要となるデータを，マイクロプロセッサの状態とその遷移のモデルにて表現する．この状態と遷移のモデルに基づき，任意時点での挙動を再現する機構を実現し，これに対しさまざまなデバッグツールや解析ツールを適用することで，不具合発生時の詳細な解析が可能となる．さらに，この状態と遷移のモデルに基づき，実装形態に対して任意の時点に再現した状態の復元と相互マイグレーションを実現する．本章では，Iana における状態と遷移のモデルとトレース情報について議論する．

3.1 対象

まず，Iana において再現およびマイグレーションの対象となるマイクロプロセッサについて定義する．一般に，マイクロプロセッサはひとつのパッケージの中に CPU だけでなくメモリや Memory Mapped I/O として直接アクセスできる内部接続された周辺機器を含む．また，マイクロプロセッサはパッケージ外部に接続された二次記憶や外部の周辺機器には，外部バスコントローラを介しアクセスする．これらの CPU やメモリなどの具体例を以下に示す．

- CPU（演算器，汎用レジスタ，浮動小数点レジスタ，ステータスレジスタ， PC）
- メモリ（RAM，フラッシュメモリ）
- 内部接続された周辺機器・回路（MMU，割り込みコントローラ，UART）
- 外部バスコントローラ（SPI， USB， PCI， SATA）
- 2 次記憶（HDD， SSD， SD カード）
- 外部接続された周辺機器（ネットワークインタフェースカード）

本稿では，CPU 及び直接アクセスできるメモリ，内部接続された周辺機器（以下単に周辺機器）を対象として状態と遷移をモデル化する．

3.2 内部状態

Iana では，以下の状態を再現し書き戻すことで，任意時点での再実行とマイグレーションを実現する．

(1) CPU の内部状態
(2) メモリの状態
(3) 周辺機器の状態

まず我々はインストラクション[*1]の実行だけに最低限必要となる CPU の状態に着目し，CPU の内部状態はすべてレジスタとして抽象化する．具体的にはインストラクションの読み込み先アドレスを示す Program Counter （PC）や演算結果にて設定される Condition Flag もレジスタとして扱う．また CPU は，アドレスによって決まるメモリもしくは周辺機器の状態に値を書き込み，また対象となるメモリもしくは周辺機器の状態に応じたデータを読み込む．

メモリの状態は単純にアドレスと値の組の集合によって表される．同様に，基本的には周辺機器の状態もアドレスとその値で表現される．どの周辺機器のどの状態であるかはアドレスにより区別される．例えば，あるマイクロプロセッサにおいて「0x10013000 番地は UART0 の送信データ」を「0x10023018 番地は UART1 の通信速度設定」の状態を表している．つまり，Iana では，状態はレジスタ番号もしくはアドレスで区別された独立したレジスタとメモリと周辺機器の値の集合であり，ある時点での状態を書き込むことで再実行やマイグレーションを実現する．

3.3 状態の遷移

遷移とは状態の変化を表し，時刻 t_i における CPU の状態を CPU_{t_i} としたとき，CPU_{t_i} から 1 クロック後の $CPU_{t_{i+1}}$ における遷移 TR_{t_i} は以下の 2 つの組で表される．

- 対象（レジスタ番号もしくはアドレス）
- 値

例えば，時刻 t_i に加算命令が実行され，その結果として x9 レジスタに 2211 が書き込まれたとすると，「対象は x9 レジスタ，値は 2211」の遷移となる．

n クロック後の CPU_{t_n} を得るには，初期状態 CPU_{t_0} に

[*1] インストラクションとは RISC-V であれば `ADD rd, rs1, rs2` や `JAL rd` $\texttt{imm}_{21}$ などのオペコードとオペランドの組を指す（rd, rs1, rs2: CPU レジスタ，$\texttt{imm}_{21}$: 21 ビット幅の即値）．

対して遷移 TR_{t_i} を 1 から n まで順次適用すればよい．副作用がある命令によって同一時刻 t_i に複数の状態遷移が発生する場合もある．例えば，「加算命令が実行され，その実行結果がレジスタに格納され，その結果に応じてオーバフローフラグがセットされる」は，加算結果をレジスタに格納するという $TR_{t_{i_1}}$ という遷移と，オーバーフローフラグをセットする $TR_{t_{i_2}}$ という 2 つ遷移で表現できる．

メモリの状態 M_{t_i} と周辺機器 P_{t_i} の状態遷移は対象（アドレス）とその値で表現し，遷移 TR_{t_i} を適用することでそれぞれ $M_{t_{i+1}}$ と $P_{t_{i+1}}$ に遷移する．例えば UART1 の通信速度の初期設定が実行されると，「対象は 0x10023018 番地，値は 216」の遷移が発生する．ただし周辺機器の状態遷移はより複雑であり，CPU からの書き込み以外の要因によっても遷移が自発的に発生する．例えば UART の場合ある時点で送信データを書き込むと，データ送信中フラグのアドレスを対象として値が 1 となる遷移が発生し，送信が完了すると値が 0 となる遷移が発生する．

3.4 トレース情報から遷移の生成

遷移を得る手段として，Iana ではイストラクションの実行に伴うトレース情報を取得することで遷移を得る．時刻 t_i にインストラクション I_{t_i} を実行したとすると，RISC CPU におけるほとんどのインストラクションは以下のように表すことができる．

$$INST_{t_i} = op(source1, source2) \Rightarrow dest$$

このインストラクションを実行した際に以下のトレース情報を取得する．

- インストラクションコード
- source1 の対象と値
- source2 の対象と値
- dest の対象と値

これは source1 と source2 を入力として，演算 op を実行した結果を dest に書き込むことを表している．source1，source2，dest はレジスタ，メモリ，周辺機器を表す．遷移はこのトレース情報からアルゴリズム 1 によって得られる．アルゴリズム 1 中の $TR_{t_i}.Target$ と $TR_{t_i}.Value$ はそれぞれ遷移の対象と値を示し，各インストラクションに対する処理は次の通りである．source1，2 の値として即値が使われる命令に対しては，インストラクションコードから値を計算する（10 行目）．また，ロード命令においてアドレスが周辺機器であった場合，周辺機器から読みだした値が周辺機器の状態を表すため，格納先のレジスタだけでなく周辺機器の遷移を生成する（17 行目）．条件付き分岐命令は，分岐条件の再評価を行い分岐する場合は分岐先を値とする遷移を生成する（22 行目）．副作用がある命令は，すべて複数の遷移としてとして分解することで表現する（25 行目）．なお，このアルゴリズムには暗黙的な PC のインクリメントは含まれていない．周辺機器に対して複雑でかつ依存関係がある状態に対する復元とマイグレーションを実現するには，トレース情報から生成された周辺機器の遷移だけでは不十分であり，その周辺機器に特化した固有の状態とその遷移を生成する必要がある．本論文ではより単純な状態遷移によって表現される周辺機器のみを取り扱う．

アルゴリズム 1 トレース情報から遷移の生成

```
1: 入力: インストラクション INST_{t_i}
2: 出力: 遷移 TR_{t_i} もしくは (TR_{t_{i_1}}, TR_{t_{i_2}})
3:
4: CodeType ← インストラクションのタイプ取得
5: if CodeType is レジスタ間演算命令 then
6:     TR_{t_i}.Target ← INST_{t_i}.dest.Target
7:     TR_{t_i}.Value ← INST_{t_i}.dest.Value
8: else if CodeType is 即値演算命令 then
9:     TR_{t_i}.Target ← INST_{t_i}.dest.Target
10:    TR_{t_i}.Value ← インストラクションコードから即値を取得
11: else if CodeType is STORE 命令 then
12:    TR_{t_i}.Target ← INST_{t_i}.dest.Target
13:    TR_{t_i}.Value ← INST_{t_i}.dest.Value
14: else if CodeType is LOAD 命令 then
15:    TR_{t_{i_1}}.Target ← INST_{t_i}.dest.Target
16:    TR_{t_{i_1}}.Value ← INST_{t_i}.dest.Value
17:    if Source1.Target is 周辺機器 then
18:        TR_{t_{i_2}}.Target ← INST_{t_i}.source1.Target
19:        TR_{t_{i_2}}.Value ← INST_{t_i}.dest.Value
20:    end if
21: else if CodeType is（条件付き）分岐命令 then
22:    if 分岐条件を再評価する then
23:        TR_{t_{i_1}}.Target ← PC の値
24:        TR_{t_{i_1}}.Value ← 分岐先のアドレス
25:        if CodeType is ジャンプ and リンク命令 then
26:            TR_{t_{i_2}}.Target ← INST_{t_i}.dest.Target
27:            TR_{t_{i_2}}.Value ← リターン後の戻り先アドレス
28:        end if
29:    end if
30: end if
```

4. 設計

本章では，Iana の設計について論じる．

4.1 Iana のシステム構成

Iana のシステム構成を図 1 に示す．Iana は，CPU として解析対象プログラムを実行し，遷移の収集・状態の書き戻しを行う．Iana-CPU と，収集された遷移を保存し状態

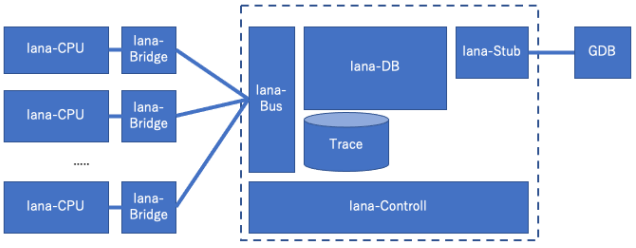

図 1 Iana のシステム構成

の再現を行う Iana-DB が主なコンポーネントであるが，さらに異なる複数の Iana-CPU のマイグレーションを実現し，異なる解析手段をユーザに提供するための以下のコンポーネントから構成される．

- Iana-CPU の違いを吸収する Iana-Bridge
- 複数の Iana-Brige を接続するための Iana-Bus
- Iana の外部インタフェースである Iana-Stub
- 全体をコントロールする Iana-Control

次に主要コンポーネント Iana-CPU と Iana-DB を中心に各コンポーネントの詳細を説明する．

4.2 Iana-CPU と Iana-Bridge

本節では，遷移の収集と状態の書き戻しの主要なコンポーネントとなる Iana-CPU とその周辺コンポーネントである Iana-Bridge について説明する．

多様な CPU アーキテクチャへの対応を実現するため，Iana-CPU は以下によって構成される．

- CPU コア （デコーダ，演算器，レジスタを含む）
- メモリ
- 周辺機器
- トレース情報コレクタ
- 状態書き込み器

Iana-CPU の CPU コア，メモリ，周辺機器は一般的な CPU と全く同じ構成であり，ハードウェア（実機，FPGA 等）もしくはソフトウェアエミュレーション（QEMU 等）により実現される．また，この CPU コア実装に応じて，トレース情報コレクタと状態書き込み器を実装する．状態書き込み器による復元の対象となるのは，CPU コアのレジスタ，メモリ，周辺機器となる．CPU コアのレジスタとメモリは単に再現された状態に基づき書き込まれる．

FPGA 等でハードウェア実装した場合，CPU コアがパイプラインとして実装されるため [6]，注意が必要である．あるクロック t_i における，フェッチされたインストラクション，演算器への入力，演算器の出力，レジスタへの書き込み，メモリへの書き込み結果は，異なるトレース情報 Tr_{i-1}，Tr_{i-2}，Tr_{i-3}，Tr_{i-4} を表している．クロック t_i で収集されたデータから，パイプラインを考慮し収集して，トレース情報として合成する必要がある．また収集と同様に書き込みにもパイプラインに対する考慮が必要となる．周辺機器の書き込みには，周辺機器毎に個別の書き込み方法が必要となる．

Iana-Bridge はトレース情報から生成した遷移を Iana-Bus に送り，また Iana-Bus から取り出した CPU 状態を Iana-CPU に渡し書き戻しを行う．この際に Iana-CPU の実装形態毎に異なる得られたトレース情報の違い，書き戻し方法の違いを吸収する．例えば，QEMU から得られるトレース情報にはレジスタやメモリ演算の対象が含まれないため，インストラクションを再評価し対象を求めることで遷移を生成する．

4.3 Iana-DB とその他のコンポーネント

次に，Iana-DB と残るコンポーネントについて説明する．

Iana-DB は，Iana-Bus 経由で得た遷移を保存し，任意の時点での状態を再現する．再現の際は，まず対象となるプログラムを読み込み，初期状態のメモリを再現する．次に，指定された時刻になるまで遷移を順次適用し，その時刻における内部状態を再現する．

Iana-Stub は解析者に異なる複数のインタフェースを提供する．例えば，Iana-Stub を GDB のスタブとして実装することで，Iana-DB にて再現された CPU の状態に対して，GDB 経由でステップ実行，ブレークポイントの設定，再実行などの操作を行う事ができる．この時，Iana-Stub は仮想的な CPU として振る舞うことになる． Iana-Stub に対して GDB が接続されると，まずリセット直後の初期状態を Iana-DB で再現し，ステップ実行が要求されると次の時刻の状態を再現する．また，Iana-DB は GDB からの要求に従い，復元された状態からレジスタやメモリの値を提供もしくは上書きする．利用者は，GDB 経由にて（仮想的な）CPU を実際に動作させているかのように，ステップ実行やレジスタ・メモリへのアクセスができる．また，別の Iana-Stub を実装すれば，別の解析ツールと連携することができる．

5. 実装

本章では，Iana システムの PoC 実装について説明する．まず，RISC-V を使った Iana-CPU について説明し，次に他コンポーネントの実装について説明する．

5.1 Iana-CPU と Iana-Bridge の実装

PoC 実装における CPU として, 以下の理由により RISC-V を対象とすることにした．

(1) 今後 IoT 機器や FPGA における SoC 向けに利用拡大が期待される．
(2) 仕様がオープンソースとして利用可能である．
(3) IP の権利関係が明確な CPU コアの実装が開発・提供されている．

RISC-V を対象とする以下の 3 実装形態にて Iana-CPU 実装した．

- オープンソースの RISC-V コア Poyo-v [7] をベースとした，FPGA による実装
- QEMU をベースとしたソフトウェアエミュレーションによる実装
- RISC-V 評価ボードを使った実機評価ボードによる実装

FPGA による実装では, Xilinx 社の Zynq UltraScale+ [8] 上に Verilog にて記述された Poyo-v ベースとし，トレース

情報コレクタと状態書き込み器を追加し Iana-CPU を実現した．この Poyo-v は 3 段パイプラインにより構成されるため，各ステージにて収集した情報をパイプラインに沿って後段のステージに渡し，最終ステージの次のステージにて収集した情報を結合し，CPU 外部に書き込む 4 段パイプライン構成にすることで，パフォーマンスの低下を抑えた．また，CPU に書き込みモードを追加し，レジスタやメモリ，PC を自由に書き換え，各パイプラインの中間状態を初期化する仕組みを実装した．さらに Zynq の特色である，FPGA 上の ARM プロセッサを使った FPGA 回路連携 Linux PYNQ [9] を使い，Iana-Bridge を実装した．PYNQ を使うことで，FPGA の同一チップ上に Linux と FPGA 回路を共存させ，かつ Linux 上のドライバ経由で高速かつ簡単に FPGA 回路を利用することができる．PYNQ 上に Python にて実装した Iana-Bridge は，収集したトレース情報を，PYNQ の Linux ドライバ経由で読み出し PYNQ 上のファイルシステムに書き込む．

ソフトウェアエミュレーションによる実装では，QEMU をベースとし，QEMU のトレース機構を用いトレース情報コレクタを実装し，状態書き込み器は QEMU の GDB スタブを利用した．GNU Debugger は，実機の外部で動作するリモートデバッグの際に，ターゲットとなる CPU に実装された GDB スタブと GDB Remote Serial Protocol（GDB RSP）[10,11] に従って通信し，CPU のメモリやレジスタを参照・書き換えし，ステップ実行やブレークポイントの設定を行う．QEMU では GDB スタブを用意することで，エミュレーションしている CPU の動作を GDB から制御すること可能としている．この GDB スタブに対して GDB RSP にて通信するクライアントを実装することで書き込み器を実現した．

実機評価ボードによる実装では，RISC-V CPU FE310-G002 を搭載した評価ボード HiFive1 Rev B [12] を利用し，OpenOCD [13] と JTAG によるトレー情報コレクタと状態書き込み器を実装した．OpenOCD は JTAG を使った CPU のデバッグやメモリ書き込み等を可能とするツールであり，GDB RSP によるリモートデバッグ機能を提供する．OpenOCD に対して，GDB RSP にて指令する GDB クライアントを実装することで状態の書き込み器が実現できる．ステップ実行によるトレース情報の収集ではリアルタイム実行は難しいため，任意の時点で CPU 実行を止め，レジスタやメモリの状態を収集するスナップショット取得機能にて，トレース情報コレクタの代用とする．

QEMU と実機評価ボードで用いた GDB RSP による実装は汎用性があり，CPU の種類やアーキテクチャに依存しない．従って，他の CPU に対しても GDB によるリモートデバッグがサポートされていれば，実機と QEMU によるマイグレーションが実現できる．また，OpenOCD も ARM や MIPS 等多数のアーキテクチャに対応しているため，OpenOCD による実機のマイグレーションを容易に拡張可能である．

5.2 Iana-DB とその他コンポーネントの実装

Iana-DB を中心とする他のコンポーネントは Perl にて実装した． Iana-Bridge にて収集されたトレース情報は，Iana-Bus を経由し Iana-DB によりファイルとして保存される．状態の復元が要求されると，ファイルを読み込み指定された時刻における内部状態を復元する．復元された状態は Iana-Bridge により一旦ファイルとして保存され，保存されたファイルを読み込み状態を書き戻す．

さらに，GDB 向けの Iana-Stub を実装し，GDB スタブに必要な必要最低限の機能を，Iana-Control として実装した．GDB にて変更したレジスタ値や，メモリ値を反映してた新たな CPU 状態を書き戻しマイグレーション可能である．この時に使うデバッガは専用ではなく実機の開発に用いられる GDB に準拠した Eclipse や Visual Studio Code 等の統合開発環境でも利用可能である．

6. ケーススタディと評価

2 つの評価対象プログラムに対し，3 形態の実装について，トレース情報の取得，状態の復元と GDB による参照，再実行，マイグレーションを実験し，評価を行った．ここでは，まず評価環境について説明し，次に各々のプログラムについてのケーススタディとその評価を行う．

6.1 実験環境

本節では，評価実験の環境と手順について説明する．評価実験は，3 つの実装形態（FPGA，QEMU，実機評価ボード）に対して，2 つの評価対象プログラムを用いて行う．

FPGA 実装は，評価ボードとして Avanet Ultra96-V2 Zynq UltraScale+を用い，Windows 10 上で動作する FPGA 開発環境 Xilinx Vivado 2020.1 にて開発した．QEMU 実装及び各評価対象プロうグラムは，x86-64 上の Ubuntu 20.04 LTS 上にて QEMU バージョン 5.2 を利用し開発した．また，各評価対象プログラムは RISC-V コミュニティが公開する公式ツールチェーン[*2]（gcc バージョン 10.2.0）にて最適化オプションを「-g」としてコンパイルした共通のバイナリを用い各実装形態にて実行した．

評価には，2 つのプログラムを用いた．1 つ目は小規模な自作プログラムであり，2 つ目は様々なアーキテクチャにおける評価等に用いられきた Dhrystone ベンチマークプログラムである．なお，プログラムの詳細については 6.2 にて説明する．

3 実装形態にて対象プログラムのトレース情報（実機からはスナップショット）を取得し 3 つの状態を復元し，こ

[*2] https://github.com/riscv/riscv-gnu-toolchain

```
1 void aaa(){
2   register int i;
3   register int j=0;
4   register int k=0;
5
6   for(i=0;i<500;i++){
7     j = j+i;
8     k = k+j+i;
9   }
10  puts("result:");decimal(j);
11  puts(",");decimal(k);puts("\n");
12  return ;
13 }
```

図 2 自作小規模プログラム（抜粋）

れをそれぞれ 3 実装形態に書き戻し，9 通りの組み合わせにて実行の再開とマイグレーションを行った．

6.2 各評価プログラムの説明と評価

本節では評価対象のプログラムの説明と以下の項目に基づいた評価について述べる．

(1) 各実装形態より取得した遷移の一致
(2) 復元された時刻 T_i での状態の一致
(3) 復元時刻から j ステップ後 T_{i+j} での各実行形態における状態の一致

また，各評価プログラム特有の観点からも評価する．

6.2.1 自作の小規模プログラム

最初のケーススタディでは，挙動がわかりやすく，データの改変による効果の推測が容易で，周辺機器とのインタラクションがない図 2 に示すプログラムにて，検証した．

このプログラムにて，3 実装形態における取得した遷移の一致を確認した．また，マイグレーション後の全ての UART 出力が一致することも確認できている．次に，gdb を使って，マイグレーションの確認を行う．図 3 は，トレース情報取得時にマイグレーション元の CPU に対し GDB にて状態を確認した際のログである．まず 500 ステップ目のレジスタ（7〜12 行目）と全メモリー（13〜18 行目）を確認した．ループ終了後の j と k は 124750（27 行目）と 20958000（29 行目）であった．また，29 行目以降はレジスター状態を表示している．次に，500 ステップ目の CPU 状態を復元しマイグレーションした CPU に対して確認した際のログを図 4 に示す．4 行目，6 行目，8 行目はマイグレーション直後の変数を，10 行目からはレジスタを，16 行目では全メモリを表示させた．これらの値は，マイグレーション前の値と一致していた．22〜29 行目はステップ実行にて，正しく動作していることを確認した．次に，実行を再開しループを抜けた行にて停止させた時（35 行目）の j，k の値は先ほどのマイグレーション元の値に一致している（37〜42 行目）．また，全てのレジスタの値も（43〜48 行目）マイグレーション元と同じことを確認した．以上の結果より，マイグレーションの正確性を確認できた．

さらに，遷移から復元された CPU 状態に対してレジスタの値を書き換え，書き換えた状態を書き戻し，レジスタ

```
1  # gdb app.elf
2  GNU gdb (GDB) 10.1
3  _start () at boot.s:6
4  6               li  x1, 0x10
5  (gdb) si 500
6  94          k = k+j+i;
7  (gdb) info register
8  ra 0x42         0x42
9  sp 0x80003fe0   0x80003fe0
10 s1 0x8a3            2211
11 s2 0xc382           50050
12 pc 0x20010304   0x20010304 <aaa+48>
13 (gdb) x/4096w 0x80000000
14 0x80000000: 1970496882 3830892 44 10
15 0x80000010: 0 0 0 0
16 ...
17 0x80003fe0: 11259375 144 -2147467264 536937348
18 0x80003ff0: 0 0 128 16
19 (gdb) b notmain.c:96
20 Breakpoint 1 at 0x20010314: file notmain.c, line
       96.
21 (gdb) c
22 Continuing.
23
24 Breakpoint 1, aaa () at notmain.c:96
25 96        puts("result:");decimal(j);
26 (gdb) print j
27 $1 = 124750
28 (gdb) print k
29 $2 = 20958000
30 (gdb) info register
31 ra 0x1f4            0x1f4
32 sp 0x80003fe0   0x80003fe0
33 s1 0x1e74e          124750
34 s2 0x13fcb30    20958000
35 pc 0x20010314   0x20010314 <aaa+64>
```

図 3 gdb によるマイグレーション元の状態確認 (抜粋)

書き換えが実行結果に正しく影響されるかを検証する．図 5 は，書き換え結果を gdb により確認した際のログである．この検証では，500 ステップ目 （i=66 に相当する） の状態に対して，変数 j に相当するレジスタを「4321」に書き換え，この状態を書き戻した．gdb にて書き戻し直後の変数 i が 4321 であり（6 行），ループ終了後に j，k の値がの値がそれぞれ 126860（15 行），21873740 （17 行）となっていることを確認した．これらの結果より，書き換えた状態が正しくマイグレーションされたことが確認できた．

6.2.2 Dhrystone

より実践的なプログラムを対象とするため，Dhrystone ベンチマークを対象としてケーススタディを行った．ただし，Dhrystone ベンチマークプログラムは，Unix Like OS 上での動作を前提として OS のシステムコールを呼び出すように書かれているため，以下の改変を行っている．

- printf() 関数を使わないように，すべての出力を消去した．MIPS 値の確認は，gdb を使い直接値を参照することで計測する．
- time() 関数を使わないようにし， CPU のカウンタを利用するように書き換えた．

3 実装形態から収集された遷移より 3 実装形態へマイグレーションし，9 通りの組み合わせにて全ての遷移の一致と実行終了後の CPU 状態の一致を確認できた．この結果は，同一バイナリの Dhrystone ベンチマークを同一 ISA の CPU で実行させ途中でマイグレーションしたとしても，

```
# gdb app.elf
aaa () at notmain.c:94
94          k = k+j+i;
(gdb) print i
$1 = 66
(gdb) print k
$2 = 50050
(gdb) print j
$3 = 2211
(gdb) info register
ra 0x42         0x42
sp 0x80003fe0   0x80003fe0
s1 0x8a3            2211
s2 0xc382           50050
pc 0x20010304   0x20010304 <aaa+48>
(gdb) x/4096w 0x80000000
0x80000000: 1970496882 3830892 44 10
0x80000010: 0 0 0 0
...
0x80003fe0: 11259375 144 -2147467264 536937348
0x80003ff0: 0 0 128 16
(gdb) s
92        for(i=0;i<500;i++){
(gdb) s
93          j = j+i;
(gdb) s
94          k = k+j+i;
(gdb) s
92        for(i=0;i<500;i++){
(gdb) b notmain.c:96
Breakpoint 1 at 0x20010314: file notmain.c, line
    96.
(gdb) c
Continuing.

Breakpoint 1, aaa () at notmain.c:96
96        puts("result:");decimal(j);
(gdb) print i
$4 = 500
(gdb) print j
$5 = 124750
(gdb) print k
$6 = 20958000
(gdb) info register
ra 0x1f4            0x1f4
sp 0x80003fe0   0x80003fe0
s1 0x1e74e          124750
s2 0x13fcb30    20958000
pc 0x20010314   0x20010314 <aaa+64>
```

図 4 マイグレーション後の状態確認（抜粋）

```
# gdb app.elf
aaa () at notmain.c:94
94          k = k+j+i;
(gdb) print j
$1 = 4321
(gdb) b notmain.c:96
Breakpoint 1 at 0x20010314: file notmain.c, line
    96.
(gdb) c
Continuing.

Breakpoint 1, aaa () at notmain.c:96
96        puts("result:");decimal(j);
(gdb) print j
$2 = 126860
(gdb) print k
$3 = 21873740
```

図 5 レジスタ書き換えの確認（抜粋）

CPU のクロック周波数（1 秒間のクロック数）と関係なく全く同じ CPU の状態遷移を辿り全く同じベンチマーク結果となり，同じクロック数で終了することを表している．

7. おわりに

本稿では組み込み機器向けソフトウェア解析環境 Iana を提案した．Iana は任意時点での挙動を再現し，さまざまなデバッグツールや解析ツールを適用することで，不具合発生時の詳細な解析が可能となる．また FPGA や QEMU，実機などの異なる実装形態間で相互にマイグレーションすることができ，例えば不具合を実機で確認したときに FPGA や QEMU にマイグレーションしデバッグを継続するなど，最適な解析手段の選択による効率化を可能とする．

今後の課題は，周辺機器の充実である．今回 FPGA にて実装した CPU の周辺機器は，UART のみである．IoT 機器のファームウェア解析やマルウェア解析には Unix Like OS 上での実行が要求され，より複雑な周辺機機器，特に外部バスコントローラ経由で接続される二次記憶や周辺機器に対するモデルの拡張と実装が必要となる．これらを FPGA にて実現し，解析可能なソフトウェアの対象を広げたい．

参考文献

[1] IEEE 1149.1-1990 - IEEE Standard Test Access Port and Boundary-Scan Architecture. https://standards.ieee.org/standard/1149_1-1990.html/.

[2] ARM Developer Serial Wire Debug. https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug.

[3] Shenghsun Cho, Mrunal Patel, Han Chen, Michael Ferdman, and Peter Milder. A Full-System VM-HDL Co-Simulation Frameworkfor Servers with PCIe-Connected FPGAs . In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*, pp. 87–96, 2018.

[4] Shye-Tzeng Shen, Shin-Ying Lee, and Chung-Ho Chen. Full System Simulation with QEMU: an Approach to Multi-view 3D GPU Design . In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 3877–3880, 2010.

[5] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, Ko Yoshikawa, and Takeshi Yoshimura. A Fast Hardware/Software Co-Verification Method for System-on-a-Chip by Using a C/C++ Simulator and FPGA Emulator with Shared Register Communication. In *Proceedings of the 41st annual Design Automation Conference (DAC '04)*, pp. 299–304, 2004.

[6] ジョン・L・ヘネシー, デイビッド・A・パターソン. コンピュータアーキテクチャ [第 6 版] 定量的アプローチ. 星雲社, 2019.

[7] poyo-v. https://ourfool.github.io/poyo-v/.

[8] Avnet Ultra96-V2. https://www.xilinx.com/products/boards-and-kits/1-vad4rl.html.

[9] PYNQ - Python productivity for Zynq. http://www.pynq.io/.

[10] Debugging with GDB, Appendix E GDB Remote Serial Protocol. https://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html.

[11] Howto: GDB Remote Serial Protocol, Writing a RSP Server. https://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.html.

[12] HiFive1 Rev B. https://www.sifive.com/boards/hifive1-rev-b.

[13] Open On-Chip Debugger. http://openocd.org/.