Vol.2020-EMB-55 No.7
2020/12/11

関数型言語ElixirのIoTフレームワークNervesに関するリアルタイム性の評価およびその改善の試み

高瀬 英希[1,2,a)]　河上 晃治[1]　菊池 豊[3]

概要：IoTフレームワークの選定時には，そのリアルタイム性能を把握しておくことが肝要である．本研究では，関数型言語Elixirによる IoTフレームワークであるNervesについて，そのリアルタイム性を定量的に評価する．Nervesの動作する複数の評価ボードを選定し，単純なベンチマークアプリケーションの実行結果によってその基礎データを得た．これらの結果に対する考察を経て，Nervesのリアルタイム性に影響を与えると考えられる幾つかの指標が見つかった．そこでさらに本研究では，リアルタイム性を改善するための試みを検討し，その有効性を議論する．

An evaluation of real-time performance for IoT framework Nerves and attempt to its improvement

1. はじめに

IoT（Internet of Things）は，情報技術の総合格闘技である．ネットワークインフラの急速な整備に後押しされ，あらゆるモノやコトさらにはヒトまでインターネットに接続して情報処理される時代が訪れている．IoTシステムを構成する要素は，外界環境の直近に位置してセンシングやアクチュエーションを担うエンドデバイス，デバイスの近傍に位置して収集したデータを加工しつつ中継するエッジサーバ，および，収集されたデータを統計解析するクラウドサービスからなる．このようにそれぞれの構成要素が持つ計算機特性を集結させることで，新たな社会的価値が創造されることが期待されている [1].

IoTシステムの特にエンドデバイスおよびエッジサーバについては，処理内容の正確さだけでなく，処理時間のリアルタイム性に価値が置かれることが多い．ここで，リアルタイム性とは，システムの処理が定められた時刻（デッドライン）までに完了できる性能のことを指す．ある処理の実行時間が高い精度で見積もられること，実行時間の揺らぎが小さいことが要求される．リアルタイム性は，デッドラインまでに処理を完了できなかった場合に，以降にその処理の価値が減少していくソフトリアルタイム性と，ただちにその価値が失われるハードリアルタイム性に分類される．IoTフレームワークの選定時には，そのリアルタイム性を十分に把握することが重要である．

現状のIoTシステム開発では，各構成要素の開発が個別に行われており，構成レイヤ間の開発手法の相違によって開発生産性の低下が生じることがある．組込み分野では，旧来よりC/C++が採用されることが多い．これは低レイヤのプログラミングや性能要件が厳しい機能の実現には役立つが，習熟容易度や開発生産性については難がある．この解決に向けたひとつの方策として，抽象度の高いプログラミング言語を導入することが挙げられる．これまでの取り組みには，PythonおよびRubyを組込み機器に応用するための開発フレームワークであるmicro-python[2] およびmruby[3] の研究開発が進んでいる．これらは少ないコード行数で所望の機能を実装することに貢献するが，元のプログラミング言語の仕様や機能からの派生であるため，アプリケーションの互換性が担保できないことがある．

我々は，IoT分野におけるElixir[4], [5] の可能性に着目している．Elixirは2012年に登場した関数型言語であり，処理の振る舞いではなくデータの扱いを直接的に操作するためのライブラリや記法が豊富に整備されている．習熟容

1 京都大学
Kyoto University
2 JST さきがけ
JST PRESTO
3 高知工科大学
Kochi University of Technology
a) takase@i.kyoto-u.ac.jp

ⓒ 2020 Information Processing Society of Japan

易性および開発生産性に優れており，並列処理のプログラミングが容易に実現できるという利点がある．また，Elixir アプリケーションは Erlang VM 上で動作するため，プロセスモデルが軽量かつ頑強であり，分散処理に向いていてシステムのスケーラビリティを得やすいという特徴を持つ．これらの特徴は，IoT システム開発における生産性の向上に対して大きな利点になりえると考える．

本研究の目的は，Elixir の IoT フレームワークである Nerves[*1]について，そのリアルタイム性を明らかにすることである．Nerves で構築されたアプリケーションの性能を定量的に評価し，IoT システムへの展開時において Nerves のリアルタイム性に影響を与えうる要因を考察する．Nerves は，Elixir の利点を IoT システムにそのまま展開できる開発フレームワークおよび実行環境である [6], [7]．すなわち，IoT システムのエンドデバイスから収集されるビッグデータを，同じ言語で開発された Web アプリケーションでシームレスに処理することができる．

さらに本研究では，Nerves のリアルタイム性を改善するための試みを検討し，その有効性を議論する．システムのリアルタイム性を実現するにはアプリケーションレベルだけでなく処理系やカーネル，ハードウェアによる協調的な支援が不可欠である．このことから，Erlang VM におけるガーベッジコレクション（GC）やプロセス優先度の設定，CPU の動作周波数の設定がもたらす影響を明らかにする．

本稿の構成は，次のとおりである．まず 2 章では，準備として Erlang，Elixir および Nerves について紹介する．次に 3 章では，Nerves のリアルタイム性の定量的評価のための評価環境を示す．4 章では，基礎的な評価結果を示してこれを考察する．その後，Nerves のリアルタイム性を改善する試みとして，5 章では GC の設定，6 章ではプロセス優先度，7 章では CPU 動作周波数のそれぞれの影響を調査する．最後に 8 章にて本稿のまとめと今後の展望を示す．

2. 準備

2.1 Erlang

Erlang [8] は，1986 年に Joe Armstrong らによって Ericsson 社内で開発され，1998 年にオープンソースとして公開された並列処理指向のイミュータブルな関数型プログラミング言語である．高可用性が求められる大規模かつスケーラブルなソフトリアルタイムシステムを構築することに向いており，電話交換機や銀行口座管理システム，メッセージングサーバなどで採用されている．Erlang は現在も活発に開発が続けられており，Open Telecom Platform（OTP）と呼ばれる配布形式で定期的にリリースされている．

Erlang の処理系である Erlang VM は，アクターモデルを採用しており，非常に軽量かつ頑強なプロセスモデルによって処理が進行する．プロセスの並列動作時には排他制御や同期処理を行う必要があり，プロセス間通信が他プロセスのメモリ領域に影響を及ぼすことがあるため，一般的なプログラミング言語では実装が複雑になる [9]．Erlang ではすべての変数がイミュータブルであり，各プロセスは非共有かつ非同期なメッセージ交換のみでプロセス間通信が行われる．アプリケーションレベルで排他制御や同期処理を行う必要が無いため，並行・並列処理に向いている．また，複数の計算機で実行されている Erlang VM 同士の通信が可能であるため，分散システムの実現にも向いている．

Erlang は監視ツリーという構造化モデルを基本的なコンセプトとしている．計算処理を行うプロセスとそれを監視するプロセスを階層的に配置し，計算プロセスに障害が発生した際にはシステム全体を停止することなく監視プロセスから再起動できるする．つまり Erlang では，`try/catch` のような例外処理ではなく，障害が起きたプロセスを高速に再起動し，外部の監視プロセスで例外処理を行うようにする．このようにすることで，例外処理を簡素化しながら，耐障害性の高いシステムを構築できる．このようなモデルは，IoT システムの構築時にも有用となると考えられる．

Erlang VM は，カーネルが管理するプロセスの 1 つとして稼働する．つまり，Erlang VM が生成および管理するプロセスは，カーネルのレベルにおけるプロセスやスレッドとは異なる．各プロセスには固有のメモリブロックが割り当てられ，不要なメモリ領域の解放や割当量の動的変更の GC 処理もプロセス単位で行われる．また，Erlang プロセスのスケジューリングも Erlang VM によって行われる．

2.2 Elixir

Elixir は，Jose Valim によって 2012 年から開発されている関数型プログラミング言語である [5]．Erlang VM 上で動作するため，並行処理および分散処理が得意である点や高い耐障害性を持つ点などの Erlang の特徴を引き継いでいる．特にイミュータブル特性により，マルチコアによる並列処理ではコア数に比例して実行速度が向上する [10]．また，Elixir アプリケーションは長い歴史を持つ Erlang が備える豊富なライブラリ資源を利用することもできる．

Elixir の言語仕様は Ruby を参考としており，従来の関数型言語と比べて記法や概念がシンプルであるが応用が効くように設計されている．典型的な Elixir コードの例として**図 1** を示す．この処理では，パイプ演算子`|>`と MapReduce モデル [11] に基づく `Flow` を用いてデータを加工しながら

```
1..1000
  |> Flow.from_enumerable()
  |> Flow.map(& foo(&1))
  |> Flow.map(& bar(&1))
  |> Enum.sort
```

図 1 Flow を使ったリスト操作の Elixir 記述

*1 `https://www.nerves-project.org`

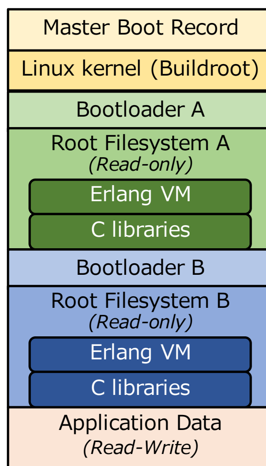

図 2 Nerves ファームウェアのファイル構造

計算が進行する．1 から 1000 までの要素を各コアに分配した後，各要素に関数 `foo/1` と `bar/1` を順に適用し，最後に要素を整列する．このように Elixir は記述性に優れていて習熟も容易であり，開発生産性が向上する．

Elixir の Web フレームワーク Phoenix[12] は，Ruby における同様のフレームワークである Ruby on Rails と同等以上の生産性を誇りながら，レスポンス性能が極めて高い．[13] によれば，Phoenix によって構築されたサーバは Ruby on Rails のそれよりも 10.63 倍のスループットを達成できたという評価結果の報告がある．[10] では，Elixir/Phoenix の IoT サーバにおける性能向上の貢献が示されている．すなわち，デバイス側でも Phoenix を活用することで，リアルタイム性に優れた IoT システムを構築できる潜在能力がある．本稿の著者らは [14] において，Elixir の IoT システムへの導入に向けた基礎評価の結果を示している．

2.3 Nerves

Nerves[6] は，Nerves Project によって開発されている，Elixir で組込みソフトウェアを開発するためのフレームワークである．ターゲットに合わせてカスタマイズされた 12 MB 程度の軽量な Linux ブートローダと，Erlang VM による Elixir の実行環境を提供する．Nerves の活用によって，Erlang および Elixir の特徴である高いスケーラビリティおよび耐障害性を備えた IoT システムを構築できる．さらに，Elixir の特徴である開発生産性を活かした効率的なシステム開発が実現できる．Phoenix と連携して，サーバと IoT デバイスとの通信制御を容易に行うこともできる．

Nerves のファームウェアは，開発用のホスト PC において，カーネルや Erlang VM，ファイルシステムおよびアプリケーションデータなどが一体となってビルドされる．これを microSD カードに書き込んで，ターゲットの IoT ボード上でアプリケーションが実行される．Nerves ファームウェアのファイル構造は，図 2 に示すとおり，ブートローダおよび Erlang VM を含むルート・ファイルシステムが二重化されている．この構造により，システムレベルで障害が発生しても，ファイルシステムの切替えのみでシステム全体の復旧を速やかに行える．なお Nerves では，Linux カーネルおよびカーネル機能の構成ツールに Buildroot[15] を採用している．Nerves Project にて現在公式にサポートされているターゲットは，Raspberry Pi および BeagleBone の各ファミリ，x86_64 および OSD32MP1 である．

表 1 定量的評価に用いた IoT ボードの主な仕様

	SoC	CPU	Memory
rpi3	Broadcom BCM2837B0	Cortex-A53 4 core / 1.4 GHz	1 GB LPDDR2
rpi0	Broadcom BCM2835	ARM1176JZF-S 1 core / 1.0 GHz	512 MB LPDDR2
bbb	TI AM3358	Cortex-A8 1 core / 1.0 GHz	512 MB DDR3

```
def sum(n) do
  1..n
  |> Enum.reduce(fn x, acc -> x + acc end)
end

def fib(0) do 0 end
def fib(1) do 1 end
def fib(n) do fib(n - 1) + fib(n - 2) end

def sleep() do :timer.sleep(5) end
```

図 3 評価に用いるベンチマークアプリケーション

3. 評価環境

本章では，Nerves のリアルタイム性を定量的に評価するために用意した評価環境について示す．

IoT ボードには，Raspberry Pi 3 Model B+，Raspberry Pi Zero WH，および，BeagleBone Green を用意した．以降，本稿では，それぞれの IoT ボードを rpi3, rpi0 および bbb と表記する．表 1 にそれぞれの主な仕様を示す．ファームウェアを書き込む microSD カードは，Gigastone 社の 16 GB SDXC / UHS-I U1 Class 10 のものを使用した．

Nerves の開発環境および実行環境として，次のソフトウェアを使用した．バージョンは本評価の実施時（2020 年 10 月 18 日）の最新のものである．

- Nerves v1.7.0
- nerves_system_br v1.13.2 (Buildroot 2020.08)
- nerves_bootstrap v1.9.0
- Elixir 1.10.4 / Erlang 23.0.4

評価に用いるベンチマークアプリケーションには，図 3 に示す 3 種類の関数を用意した．`sum/1` は，1 から n までの整数値のリストの総和を求める関数である．ローカルデータを集中的に利用するものとなる．`fib/1` は，n のフィボナッチ数を再帰的に求める関数である．スタックメモリを集中的に使用する．`sleep/0` は，Erlang の標準モジュー

ⓒ 2020 Information Processing Society of Japan

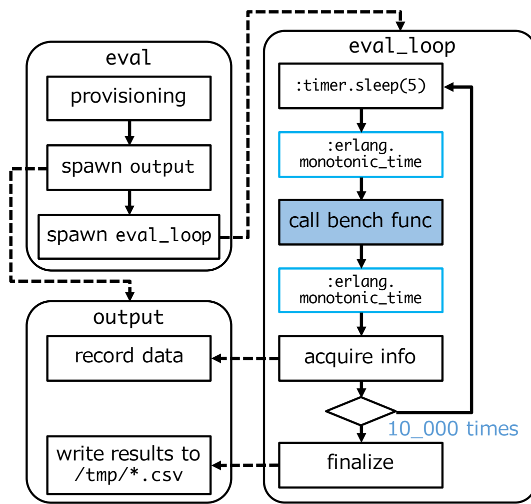

図 4 Nerves アプリケーションの評価方法

ル:timer によって 5 ミリ秒待つ関数である．sum/1 および fib/1 の引数 n は，rpi3 での実行時間がおよそ 5 ミリ秒となるように，4000 および 20 にそれぞれ設定した．

Nerves アプリケーションの評価にあたっては，他プロセス等の影響を極力排除するため，図 4 に示す 3 個のプロセスによるフローで実施する[*2]．本フローでは，まず eval プロセスによって初期設定を行ったのち，output および eval_loop のプロセスを生成する．eval_loop プロセスでは，図 3 のいずれかのベンチマークの関数を呼び出して実行する．関数の実行時間は Erlang 標準モジュールの関数:erlang.monotonic_time/0 によってマイクロ秒単位で計測し，計測ごとに output プロセスに結果を送信する．ベンチマークの関数の実行および計測は 10,000 回繰り返す．繰り返しが全て終了したら eval_loop プロセスから通知し，output プロセスにて全結果を csv ファイルとして Nerves ファイルシステムの書き込み可能領域に書き出す．

1 章で述べたとおり，リアルタイム性の評価においては，ハードリアルタイム性としては処理の実行時間の最大値が，ソフトリアルタイム性としては揺らぎの小ささが重視される．本研究では，後者は標準偏差を求めて評価する．

4. 基礎評価

Nerves のリアルタイム性について，まず本章では基礎的な評価を実施し，この比較結果を考察する．

4.1 IoT ボードごとの比較評価

表 2 および**図 5** に，IoT ボードごとの基礎的な評価結果を示す．表 2 の 2 列目の board が IoT ボードに対応している．また，以降の表は同様に，1 列目の func はベンチマークの関数名，AVG，MAX，VAR および STDEV の列は，

[*2] 本研究で用意したベンチマークアプリケーションおよび評価フローの実装は https://github.com/takasehideki/nerves_rt_perf にて公開している．

表 2 IoT ボードごとの基礎的な評価結果

func	board	AVG	MAX	VAR	STDEV
sum	rpi3	4990	5715	10619	103
	rpi0	6541	8330	86682	294
	bbb	7807	19372	156970	396
fib	rpi3	5051	6650	1264	36
	rpi0	4842	6218	5529	74
	bbb	7045	16386	67759	260
sleep	rpi3	5999	6117	193	14
	rpi0	5997	7221	483	22
	bbb	5999	10357	4776	69

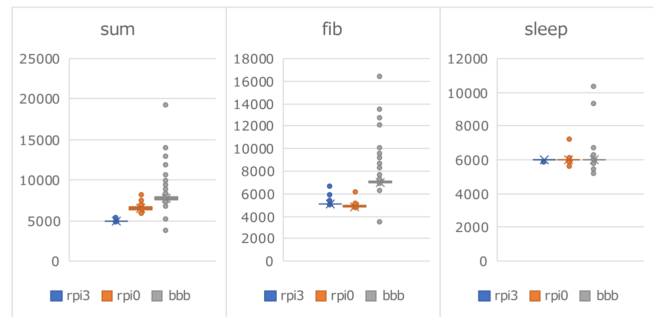

図 5 IoT ボードごとの基礎的な評価結果の箱ひげ図

各関数の実行時間の平均値，最大値，分散および標準偏差をそれぞれマイクロ秒で示している．図 5 は最大値，最小値および四分位数が表現される箱ひげ図であり，データの分布や外れ値の存在を視覚的に読み取ることができる．

この基礎評価では，rpi3 が最大値および標準偏差ともに最も良好な結果を示した．主に CPU の周波数とメモリ量が要因と考えられる．本研究でのベンチマーク関数は単純な処理のみであり，コア数はそれほど影響が無い．また，評価時に得られた注視すべき点として，Erlang VM における GC 処理の発生時には実行時間の揺らぎが確認された．

bbb は rpi0 よりも劣る結果となった．CPU 動作周波数の設定が要因として考えられる．詳細は 7 章で示すが，bbb は 5 段階の周波数設定があるが，rpi0 は 2 段階のみとなる．

4.2 実行環境ごとの比較評価

rpi3 を対象として，他の実行環境との Nerves の比較評価を実施した．比較対象には，Raspberry Pi OS (32-bit) with desktop 2020-08-20[16] および Ubuntu MATE 20.04 beta 1[17] を採用した．Ubuntu MATE は 32-bit および 64-bit の両方を用いた．これらの環境では，ベンチマークの関数は Elixir の REPL である IEx 上で直接実行した．Elixir のバージョンは Nerves と同じく 1.10.4-otp-23 とした．

表 3 および**図 6** に，実行環境ごとの評価結果を示す．表 3 の 2 列目の runtime が実行環境であり，順に Nerves, Raspberry Pi OS，Ubuntu MATE の 32-bit および 64-bit をそれぞれ表す．この比較からは，Nerves はリアルタイム性については良好な結果を示しているといえる．ただし，

表 3　rpi3 における実行環境ごとの評価結果

func	runtime	AVG	MAX	VAR	STDEV
`sum`	Nerves	4990	5715	10619	103
	RPi OS	3102	4525	3874	62
	MATE32	3308	4589	60499	246
	MATE64	2108	3283	11032	105
`fib`	Nerves	5051	6650	1264	36
	RPi OS	3097	4158	759	28
	MATE32	4004	4958	248234	498
	MATE64	2005	2753	8986	95
`sleep`	Nerves	5999	6117	193	14
	RPi OS	5999	6159	104	10
	MATE32	5999	11519	4147	64
	MATE64	5999	6999	1170	34

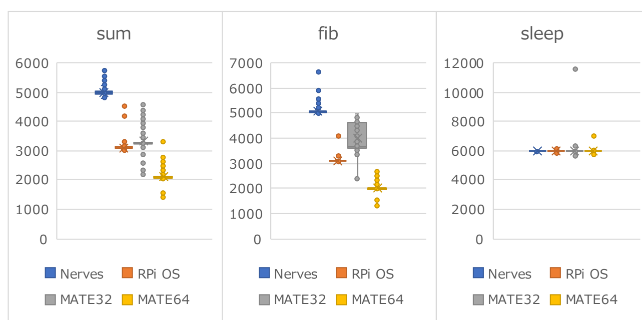

図 6　rpi3 における実行環境ごとの評価結果の箱ひげ図

RPi OS での実行時間も安定しており，平均性能はより高くなっている．平均性能に関しては，Nerves では HiPE[18] がサポートされていないことが影響している．Erlang のネイティブコード向けコンパイラである HiPE は処理性能に大きな効果が得られるが，現在では開発保守体制が継続されていない．MATE については，同じく HiPE により平均性能は高いが，リアルタイム性に関わる最大値および標準偏差は劣る結果となっている．ただし 32-bit および 64-bit の比較は着目すべきである．現時点の Nerves（Buildroot）では 32-bit のカーネルが採用されている．

5.　GC の設定によるリアルタイム性の改善

4.1 節の基礎評価からも分かるとおり，処理系でメモリ管理が行われるプログラミング言語では，GC によるリアルタイム性の低下は不可避である．本章では，Erlang VM における GC の設定によって Nerves のリアルタイム性を改善する試みを検討し，その効果を評価する．

5.1　Erlang VM における GC アルゴリズム

Erlang OTP 20.0 以降で採用されている GC アルゴリズムは，Chenecy のコピー GC[19] およびグローバルラージオブジェクトスペースを用いた，generational semi-space copying collector である [20], [21], [22]．それぞれのプロセスに対して，固有のスタックおよびヒープが割り当てられる．これらのメモリブロックは互いの方向に成長あるいは収縮し，空き容量が存在しない場合に GC が行われる．

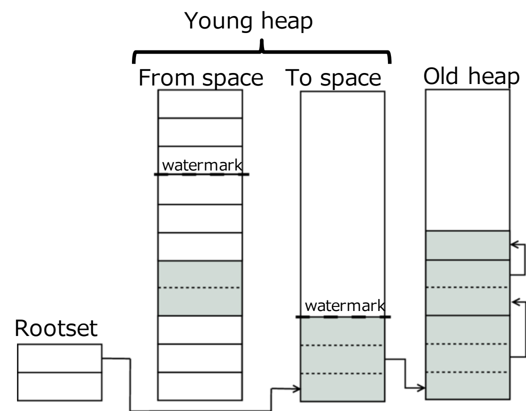

図 7　Erlang VM における GC アルゴリズム（[20] より引用）

図 7 に示すように，Erlang プロセスに割り当てられるヒープ領域は，Young heap および Old heap，さらに Young heap は From space および To space とそれぞれ呼ばれる領域から構成される．Erlang のデータはまず From space に配置され，この領域に空きが無くなった時に GC が行われる．その開始位置の Rootset から到達可能なデータは，semi-space copying によって To space にコピーされる．参照されず From space に残るメモリは回収される．GC 処理時に長期間存在するデータが走査されることを避けるために，Young heap で 2 回参照されたデータは，generational GC によって追加領域の Old heap に移動される．

full sweep GC は，Young heap のメモリを回収できない場合あるいは Old heap に移動できる領域を確保できない場合に，Old heap からメモリを回収するために行われる．また，full sweep GC は，Old heap にまだ余裕がある場合でも，genarational GC を特定回数行った後に定期的に実行される．full sweep GC の処理後にも十分なメモリが確保できない場合は，プロセスに割り当てるヒープサイズをフィボナッチ数に応じて徐々に大きくする．

Elixir コードからは，プロセスの生成時のオプション `spawn_opts()` によって次のパラメータが設定できる [23]．

- `fullsweep_after`: full sweep GC の実行前に generational GC を行う回数の最大値．ゼロにすると generational GC は行われなくなる．デフォルトは 65535．
- `min_heap_size`: ヒープサイズの最小値．単位は語（word）であり，デフォルトは 233．

5.2　`fullsweep_after` の設定による評価結果

`eval_loop` 生成時の `fullsweep_after` に 0, 32767, 65535 および 131071 をそれぞれ設定して評価した．表 4 に結果を示す．3 列目の `Number` が `fullsweep_after` の設定回数となる．紙面の都合上，有意差が見られなかった bbb および `sleep/0` の結果は省略している．

表 4 fullsweep_after の設定による評価結果

func	board	Number	AVG	MAX	VAR	STDEV
sum	rpi3	0	5205	6326	3760	61
		32767	5002	6982	11580	108
		65535	4990	5715	10619	103
		131071	4871	6560	8383	92
	rpi0	0	7397	9057	10617	103
		32767	6100	10405	53882	232
		65535	6541	8330	86682	294
		131071	6013	7616	51682	227
fib	rpi3	0	5045	6806	1946	44
		32767	5049	5738	705	27
		65535	5051	6650	1264	36
		131071	5063	6836	2043	45
	rpi0	0	4834	6504	4692	68
		32767	4839	5249	4737	69
		65535	4842	6218	5529	74
		131071	4845	6640	5597	75

表 5 min_heap_size の設定による評価結果

func	board	Size	AVG	MAX	VAR	STDEV
sum	rpi3	34	4977	6878	10439	102
		233	4990	5715	10619	103
		6765	4907	6577	3273	57
		196418	4821	6673	9752	99
	rpi0	34	6406	8232	156167	395
		233	6541	8330	86682	294
		6765	6639	8211	130514	361
		196418	6823	10669	587460	766
fib	rpi3	34	5073	5518	1271	36
		233	5051	6650	1264	36
		6765	5103	7030	2080	46
		196418	5076	5807	1242	35
	rpi0	34	4838	6414	5307	73
		233	4842	6218	5529	74
		6765	4834	6362	5204	72
		196418	4832	6214	5444	74

sum/1 では，generational GC を抑止する 0 の設定としたときに標準偏差の改善が見られた．このベンチマークは大量のローカルデータを使用するため，頻繁な GC の実行を抑制することの効果が得られていると考えられる．ただし，平均性能は悪化していることに注意が必要である．一方，fib/1 では有意な効果は見られなかった．

5.3 min_heap_size の設定による評価結果

次に，min_heap_size を 34，233，6765 および 196418 にそれぞれ設定して評価した．**表 5** に結果を示す．3 列目の Size がヒープサイズの最小値である．表 4 と同様に bbb および sleep/0 の結果は省略している．

sum/1 では，特に rpi3 において最小のヒープサイズの増大によるリアルタイム性の改善が見られた．この値を大きくすると GC 回数が少なくなり，処理の高速化が図れる．ただしデータの局所性が低下するため，大きくしすぎると性能の悪化に繋がる．一方，fib/1 は，前節と同様に効果は見られなかった．今回の評価では再帰回数が少なく，ヒープメモリがそれほど多く使用されなかったためである．

6. プロセスの優先度の設定による改善

Elixir および Erlang のプロセスは固有の優先度を持ち，Erlang VM によるプロセススケジューラによって実行順序が決定される．優先度を適切に設定することで，リアルタイム性が求められるプロセスの実行時間の揺らぎを抑えられることが考えられる．そこで本章では，プロセスの優先度の設定によって Nerves アプリケーションのリアルタイム性を改善することを試みる．

6.1 Erlang VM におけるプロセスのスケジューリング

Erlang VM のプロセススケジューラは，reduction という値に基づくラウンドロビンスケジューリングを採用している．プリエンプティブな方式であり，reduction によって定められる時間を超えるまでプロセスの処理が続いた場合には CPU 使用権が次のプロセスに移行する．

Erlang プロセスの優先度は，low，normal，high および max の 4 種類が用意されている．同じレベルの優先度のプロセスは CPU 使用権が平等に割り当てられる．low と normal は相対順位であり，前者よりも後者のプロセスの実行頻度が高くなる．high は low および normal に対して絶対順位であり，下位のプロセスより優先して high のプロセスが実行される．ただし，OTP サーバやシステムプロセスの多くは normal で実行されることに注意が必要である．max のプロセスは最も優先して実行されるが，ユーザプロセスに設定することは推奨されない．

優先度継承や優先度上限などのハードリアルタイムシステムを実現するための機能は，Erlang VM では提供されていない．また，優先度の高いプロセスで長時間のビジーループが発生すると，システムプロセスなどの処理が滞りシステム全体で問題が発生する可能性が高くなる．

Elixir コードからのプロセスの優先度の設定には priority_level() を使用する．GC の設定と同様に，プロセス生成時の spawn_opts() にて設定できる [23]．なお，自身のプロセスの優先度を実行時に変更することもできる．

6.2 priority_level の設定による評価結果

eval_loop 生成時の priority_level に low，normal および high をそれぞれ設定して評価した．**表 6** に結果を示す．3 列目の Level が設定したプロセスの優先度である．sleep/0 は自明に有意差が見られないため省略する．

sum/1 では，優先度を変更することで bbb における最大値の改善が見られた．fib/1 でも同様に，優先度を high

表 6 priority_level の設定による評価結果

func	board	Level	AVG	MAX	VAR	STDEV
sum	rpi3	low	4988	6839	8728	93
		normal	4990	5715	10619	103
		high	5018	5707	9256	96
	rpi0	low	6066	7561	56099	237
		normal	6541	8330	86682	294
		high	6047	7774	59242	243
	bbb	low	7763	18147	165747	407
		normal	7807	19372	156970	396
		high	7751	13706	153148	391
fib	rpi3	low	5090	6775	2251	47
		normal	5051	6650	1264	36
		high	5087	5276	640	25
	rpi0	low	4839	6659	5402	73
		normal	4842	6218	5529	74
		high	4836	5975	4924	70
	bbb	low	7057	19445	51255	226
		normal	7045	16386	67759	260
		high	7054	13121	34832	187

表 7 governor の設定による評価結果

func	board	gvnr	AVG	MAX	VAR	STDEV
sum	rpi3	cons	4990	5715	10619	103
		perf	2067	3055	1703	41
		save	5008	6126	9069	95
	rpi0	cons	6541	8330	86682	294
		perf	4457	6051	47818	219
		save	6148	8112	51265	226
	bbb	cons	7807	19372	156970	396
		perf	2354	3500	17872	134
		save	7810	14639	166399	408
fib	rpi3	cons	5051	6650	1264	36
		perf	2158	3151	180	13
		save	5048	6725	2004	45
	rpi0	cons	4842	6218	5529	74
		perf	3386	4413	1705	41
		save	4824	5427	3375	58
	bbb	cons	7045	16386	67759	260
		perf	2084	2943	793	28
		save	7027	13339	25977	161

にすることで最大値の改善が見られた．特に rpi3 ではその効果が顕著である．ただし，標準偏差については有意差が見られなかった．この理由としては，今回のベンチマークでは他のプロセスが存在しないためであると考えられる．

7. CPU 周波数の制御による改善

4.1 節で示唆したとおり，CPU の動作周波数の変動はプロセスの処理時間に影響をもたらし，ひいてはリアルタイム性の低下に繋がりうる．そこで本章では，CPU 周波数の制御による Nerves のリアルタイム性の改善を試みる．

7.1 動的電圧・周波数制御

最近の SoC では動作周波数を動的に変更できる CPU が搭載されており，Linux のカーネルモジュールである cpufreq によってその制御が担われている．CPU の動作周波数の設計は供給電圧に相関があり，この制御技術は情報機器の省エネルギー化のための動的電圧周波数制御（DVFS）[24] として知られている．本研究で使用している rpi3 では 600 MHz および 1.4 GHz，rpi0 では 700 MHz および 1.0 GHz のそれぞれ 2 種類，bbb では 300 MHz, 600 MHz，720 MHz，800 MHz および 1.0 GHz の 5 種類の動作周波数を設定することができる．

周波数制御のための cpufreq は Linux v3.4 から標準機能として搭載されている．Nerves が採用している Buildroot では/sys/devices/system/cpu/cpufreq にて利用可能である．Linux における CPU 周波数制御の戦略を governor と呼び，Nerves では次の 5 種類から選択できる [25].

- performance: 常に最大の周波数で CPU を動作させる
- powersave: 常に最小の周波数で CPU を動作させる
- userspace: ユーザに指定された周波数で動作させる
- ondemand: 負荷に応じて周波数を動的に切り替える*3
- conservative: 負荷に応じて周波数を動的かつ段階的に切り替える（デフォルトの設定）

7.2 CPU 周波数の制御による評価結果

ベンチマークアプリケーションの実行前に，ターゲットの governor を conservative，performance および powersave にそれぞれ設定して評価した．表 7 に結果を示す．3 列目の gvnr は governor の設定であり，cons, perf および save がそれぞれに対応する．sleep/0 は有意差が見られないため省略する．

この結果から，CPU 動作周波数を固定することは，リアルタイム性の改善に効果があることがわかる．最大の周波数に設定すると，平均および最大の性能も大きく改善する．ただし消費エネルギーおよびデバイス発熱の観点からは望ましくない状況であることに注意が必要である．最小に設定しても，平均性能の大きな劣化は見られなかった．

8. おわりに

本研究では，関数型言語 Elixir の IoT フレームワーク Nerves について，ベンチマークアプリケーションによって性能を定量的に評価しながら，そのリアルタイム性の改善を試みた．本研究の成果として，次の知見が得られた．

- Nerves は，他の実行環境と比較しても良好なリアルタイム性を有する．また，IoT ボードに搭載される SoC の性能はリアルタイム性にも直結する．

*3 負荷が高まると周波数を高くして，アイドル時間が増えると元の周波数に戻す．

- Erlang VM における GC 回数の抑制は，ローカルデータを多用する場合にはリアルタイム性の向上に貢献する．最小のヒープサイズの適切な設定も有用である．
- プロセスの優先度を `high` にすることは，実行時間の最大値の改善に寄与する．ただし OTP サーバやシステムプロセスを含む他のプロセスの実行を阻害するため，設定には注意が必要である．
- CPU 動作周波数を固定することはリアルタイム性の向上に重要である．ただし常に最大とする `performance` の設定は，消費エネルギーの増大を招く恐れがある．

今後の展望として，Nerves のリアルタイム性のさらなる定量的評価が挙げられる．本研究で用意したベンチマークは単純な関数のみのものであったため，より実用的なアプリケーションで評価することが不可欠である．Nerves では，デバイスドライバ操作のための Elixir Circuits やネットワークユーティリティ VintageNet を備える．これらのライブラリを活用したより IoT 志向なアプリケーションでリアルタイム性の評価を実施したい．本研究で試行した GC の設定，プロセス優先度および CPU 動作周波数の制御の組合せによる評価も必要であると考えている．

システム全体のハードリアルタイム性を真に確保するためには，Linux カーネルを用いる Nerves だけでは本質的に限界がある．Nerves の公式ターゲットである AM335x および OSD32MP1 には，リアルタイム制御向けの PRU コアおよび Cortex-M4 コアもそれぞれに集積されている．これらを活用した組込みカーネル技術との融合を図り，Erlang/Elixir および Nerves によってリアルタイム IoT システムを構築するための新たな統合開発手法を模索したい．

謝辞　本研究の一部は，JST さきがけ JPMJPR18M8 の支援を受けたものである．Nerves Project の co-author である Frank Hunleth 氏および Tombo Works の衣川亮太氏には，本研究の実施にあたって多くの技術的な助言や協力をいただきました．ここに深く感謝いたします．

参考文献

[1] Lebedev, A.: What is an IoT platform? (online), `https://dzone.com/articles/what-is-an-iot-platform` (2020.11.20).
[2] Geogre, D.: MicroPython - Python for microcontrollers (online), `https://micropython.org/` (2020.11.20).
[3] Tanaka, K., Nagumanthri, A. and Matsumoto, Y.: mruby – Rapid Software Development for Embedded Systems, *Proc. of 15th Int'l Conf. on Computational Science and Its Applications (ICCSA)*, pp. 27–32, (2015).
[4] Thomas, D.: *Programming Elixir ≥ 1.6: Functional |>Concurrent |>Pragmatic |>Fun*, Pragmatic Bookshelf (2018).
[5] The Elixir programming language (online), `https://elixir-lang.org/elixir-lang` (2020.11.20).
[6] Nerves Project: Nerves Platform (online), `https://www.nerves-project.org/` (2020.11.20).
[7] Schneck, J.: Nerves Project: Performant, Scaleable, and Fault Tolerant Embedded Systems, 第 21 回組込みシステム技術に関するサマーワークショップ（SWEST21）, `https://swest.toppers.jp/SWEST21/program/keynote.html#keynote` (2019).
[8] Ericsson OTP Team: Erlang Programming Language, `https://www.erlang.org/` (2020.11.20).
[9] D.Cvetkovic, M. and S.Jevtic, M.: Interprocess Communication in Real-Time Linux, *Proc. of 6th Int'l Conf. on Telecommunications in Modern Satellite, Cable and Broadcasting Service (TELSIKS)*, pp. 618–621 (2003).
[10] Fedrecheski, G., Costa, L. C. P., and Zuffo., M. K.: Elixir programming language evaluation for IoT, *Proc. of IEEE Int'l Sympo. on Consumer Electronics (ISCE)*, pp. 105-–106 (2016).
[11] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM*, Vol. 51, No. 1, pp. 107–113 (2008).
[12] Phoenix: Productive. Reliable. Fast. A productive web framework that does not compromise speed and maintainability (online), `http://phoenixframework.org` (2020.11.20).
[13] Mccord, C.: Elixir vs Ruby Showdown - Phoenix vs Rails (online), `https://littlelines.com/blog/2014/07/08/elixir-vs-ruby-showdown-phoenix-vs-rails` (2020.11.20).
[14] 高瀬英希，上野嘉大，山崎進：関数型言語 Elixir の IoT システムへの導入に向けた基礎評価，情報処理学会研究報告，2018-EMB-48(5) (2018).
[15] Buildroot - Making Embedded Linux Easy (online), `https://buildroot.org/` (2020.11.20).
[16] Raspberry Pi Foundation: Raspberry Pi OS (online), `https://www.raspberrypi.org/software/` (2020.11.20).
[17] Ubuntu MATE Team: Ubuntu Mate |For a retrospective future (online), `https://ubuntu-mate.org/` (2020.11.20).
[18] Petterson, M., Sagonas, K. and Johansson, E.: The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation, *Proc. of Int'l Sympo. on Functional and Logic Programming (FLOPS)*, pp. 228–244 (2002).
[19] Chenecy, C. J.: A nonrecursive list compacting algorithm, *Communications of the ACM*, Vol. 13, pp. 677–678 (1970).
[20] Larsson, L.: Erlang Garbage Collector (online), `https://www.erlang-solutions.com/blog/erlang-garbage-collector.html` (2020.11.20).
[21] Cessarini, F. and Vinoski, S.: *Desigining for Scalability with Erlang/OTP*, O'Reilly (2016).
[22] Stenman, E.: *The BEAM Book – The Erlang Runtime System*, `https://blog.stenmans.org/theBeamBook/` (2020.11.20).
[23] Erlang Run-Time System Application (ERTS) Reference Manual Version 11.1 (online), `https://erlang.org/doc/man/erlang.html` (2020.11.20).
[24] Pillai, P. and Shin, G. K.: Real-time dynamic voltage scaling for low-power embedded operating systems, *Proc. of Sympo. on Operating Systems Principles (SOSP)*, pp. 89–102 (2001).
[25] Freeze, C.: PowerControl: Improve the Performance of Your Nerves Device (online), `https://dockyard.com/blog/2019/02/25/powercontrol-improve-the-performance-of-your-nerves-device` (2020.11.20).

ⓒ 2020 Information Processing Society of Japan