超並列プログラミング言語処理系 micro Elixir / ZEAM のビジョンを語ろう

山崎 進 [1], 森 正和 [2], 久江 雄喜 [1], 高瀬 英希 [3]

[1] 北九州市立大学
zacky@kitakyu-u.ac.jp

[2] カラビナテクノロジー
kiritsugu.mori@karabiner.tech

[3] 京都大学
takase@i.kyoto-u.ac.jp

概 要 本論文では超並列プログラミング言語処理系 micro Elixir / ZEAM の研究構想について紹介する．micro Elixir / ZEAM の機能的な特徴は，Elixir マクロを用いたメタプログラミング解析器，LLVM を用いたコード生成系，命令並列性に基づく静的命令スケジューリング，CPU / GPU を統合する超並列高速実行処理系 Hastega，I/O バウンド処理を高速化する省メモリ並行プログラミング機構 Sabotender，実行時間予測に基づく静的タスクスケジューリングとハードリアルタイム性，プロセス間通信を含む超インライン展開，超インライン展開や静的タスクスケジューリングを前提にした大域的なキャッシュメモリと I/O の最適化などである．これらを前提に，今までの常識を打ち破るようなカーネルや VM，プロセッサアーキテクチャ，並列/分散コンピューティング，特定ドメイン向け最適化などの研究に取り組む．

1 はじめに

Elixir [23] は，並行・並列プログラミングに長けており，耐障害性が高いという特長を備えている．さらに，Elixir で書かれたウェブサーバーフレームワークである Phoenix [17] を用いることで，極めてレスポンス性の高いウェブサーバーを構築できる [5]．Elixir User's Survey 2016 [1] の調査対象で Elixir を採用していると回答した企業数は，全世界で 2014 年に 191，2015 年に 405，2016 年に 1109 である．現在はさらに急速に普及が進んでいる．

Elixir の高い並行・並列プログラミング能力と耐障害性を支えているのは，Elixir の実行系である Erlang VM である．Erlang VM は Erlang [4] のために開発された VM で，Elixir の他にもいくつかのプログラミング言語が Erlang VM 上で動作する．

我々が本発表で提案する Elixir のサブセット言語であるプログラミング言語 micro Elixir とその処理系 ZEAM (ZACKY's Elixir Abstract Machine) では，そのような Erlang VM 互換の実現とは異なるアプローチで Elixir の処理系を開発する．

micro Elixir / ZEAM の基本構成は次の通りである:

- Elixir マクロを用いたメタプログラミング解析器
- LLVM を用いたコード生成系

我々が micro Elixir / ZEAM で掲げる野心的な研究目標は次の通りである:

- 並列性に関わる最適化
  - 命令並列性に基づく静的命令スケジューリング

  - CPU / GPU を統合する超並列高速実行処理系 Hastega
  - I/O バウンド処理を高速化する省メモリ並行プログラミング機構 Sabotender
- 大域的な最適化
  - 実行時間予測に基づく静的タスクスケジューリングとハードリアルタイム性
  - プロセス間通信を含む超インライン展開
  - 超インライン展開や静的タスクスケジューリングを前提にした大域的なキャッシュメモリと I/O の最適化

本論文の目的は micro Elixir / ZEAM の研究構想を提示し，各研究目標に対する技術的課題を明らかにすることである．

本論文の以下の構成は次の通りである: 第 2 章にて Elixir の特長を生み出す基礎となる MapReduce プログラミングスタイル [3] について説明した後，第 3 章にて Erlang VM の特長を詳述し，互換処理系について概説する．第 4 章で，新たな処理系をわざわざ研究開発することについての問いに対する答えを説明する．第 5 章で，micro Elixir / ZEAM の基本構成について説明する．第 6 章に我々が考える Elixir の並列性の 3 分類を提案し，それぞれの並列性について研究目標を掲げる．第 7 章で，いくつかの要素技術を提案した上で，それらを統合して大域的な最適化を行うことについて述べる．さらに第 8 章で，他の Erlang VM 互換の処理系について概観し，これらと micro Elixir / ZEAM の関係性のあり方について述べる．最後に，本論文のまとめと将来課題を第 9 章で述べる．

2 Elixir における MapReduce プログラミングスタイル

Elixir は MapReduce プログラミングスタイル [3] を採用している．このような Elixir のプログラム例を図 1 に示す．

```
1..1_000_000
|> Enum.map(foo())
|> Enum.map(bar())
|> IO.inspect
```

図 1. MapReduce プログラミングスタイルの Elixir コード例

まず，1 行目の `1..1_000_000` は，1 から 1,000,000 までの要素からなるリストを生成する．なお，数字の間の_ によって，数字を分割するコンマを表す．

次に，2,3,4 行目の先頭にある |> はパイプライン演算子である．パイプライン演算子の前に書かれている記述の値を，パイプライン演算子の後に書かれた関数の第 1 引数として渡す．すなわち，次のような記述と等価である．

```
IO.inspect(Enum.map(Enum.map(1..1_000_000, foo()), bar()))
```

パイプライン演算子により，左から右へ，上から下へ，自然な流れで処理を記述することができ，Lisp のように括弧の複雑なネストにより可読性を損ねることがなくなる．

次に 2,3 行目に書かれている `Enum.map` は，第 1 引数に渡されるリスト等の要素 1 つ 1 つに，第 2 引数で渡される関数を適用する．ここでは，関数 `foo` を各要素に適用した後，関数 `bar` を各要素に適用する． 最後に，4 行目の `IO.inspect` は第 1 引数に渡された値の内部表現を表示する．

もし，関数 `foo` が元の値を 2 倍する関数で，関数 `bar` が元の値に 1 加える関数であった場合には，図 1 のプログラムにより，2 倍してから 1 加える処理を 1 から 1,000,000 までの要素に適用したリスト，すなわち `[3, 5, 7, ...]` を生成して表示する．

3 Erlang VM

現在，Elixir は Erlang VM という仮想機械 (VM) 上で動作する．Erlang VM の特長は，並行プログラミングに優れていることと，耐障害性が高いことである．

並行プログラミングに優れる理由は次の通りである:

- アクターモデル [10] に基づく並行プログラミングモデルを採用していること
- プロセスの生成がコアごとに独立していて軽量であること
- 資源を直接操作するプロセスを 1 つに限定して資源の利用をそのプロセスへのメッセージパッシングで実現していること
- これらにより同期・排他制御を行う必要性を大幅に削減していること

また，耐障害性に優れている理由は，次の通りである:

- プロセスごとに分離したメモリ空間であること
- そのことにより Full GC が働いていわゆる Stop the world すなわち処理系を利用するすべてのプログラムの動作が一斉停止する事態に陥ることがないこと
- 基本的にプログラム中に例外処理を記述せずにプロセスごと異常終了するように設計しておき監視プロセスにより該当プロセスを再起動させることで復旧させるアプローチを採用していること
- これらのことにより障害が発生してもメモリのリークや不整合が生じることがないこと

Elixir は，このような Erlang VM の特長を継承している．

4 micro Elixir / ZEAM の実装戦略

おそらく新たな処理系をわざわざ研究開発することについて，次のような疑問を持つと考えられるので，各節でそれぞれの問いに答えた:

- 現在主流の Erlang VM から円滑に移行することはできるのか？(4.1 節)
- 優れた Erlang VM よりもさらに優れた処理系を作れる勝算はあるのか？(4.2 節)
- さらに micro Elixir / ZEAM を研究開発していくことに意義はあるのか？(4.3 節)

4.1 Erlang VM からの円滑な移行戦略

まずは「現在主流の Erlang VM から円滑に移行することはできるのか？」という問いに答える．

我々は Erlang VM 互換の処理系を一から研究開発するアプローチの問題点を次のように考えた:

- 現行の優れた言語処理系である Erlang VM と両立しない
- そのことにより，少なくとも Erlang VM 同等以上のパフォーマンスを安定して得られるようにならないと実運用に用いる利点がない
- そのような状態になるまでに，多大な時間を必要とする

そこで，我々が採用した戦略は次の通りである:

1. Elixir のサブセットのプログラミング言語である micro Elixir を新たに定義する
2. ZEAM は Elixir プロジェクト中のコードの一部を，NIF (Native Implemented Function: ネイティブコードで実装された関数) にコンパイルして Erlang VM から呼び出せるようにする処理系として，当面の間研究開発を進める

3. ZEAM は，与えられたコードを解析して，micro Elixir の言語仕様の範囲内のコードと範囲外のコードに分割する．前者はネイティブコードにコンパイルして NIF として定義する．NIF 呼出しコードと後者のコードを繋ぎ合わせた Elixir コードを生成し，Erlang VM で実行するようにする
4. ZEAM の解析部は， Elixir マクロを利用することで，パーサーをフルスクラッチで開発しないで済ませる (第 5.1 章)
5. ZEAM の生成部は， Rust [11] 経由で LLVM [6] を利用することで，対応可能な ISA を最大化し，かつ最適化器などのツールチェーンを活用できるようにする (第 5.2 章)
6. ZEAM の最初のアプリケーションを超並列高速実行処理系 Hastega (第 6.2 章) とすることで，最初から micro Elixir / ZEAM を利用する動機を作る

このうち戦略 3 について説明する．例えば図 1 のコードのうち，1〜3 行目は micro Elixir の範囲内で，4 行目の `IO.inspect` は範囲外であるとする (最初期にはこのようにデザインする予定である)．この時，図 2 のように分割し，関数 `compile_to_nif` をネイティブコードにコンパイルして NIF として定義する．なお，`def func do ... end` は引数のない関数 `func` を `...` で示されるコードを実行する関数として定義することを意味する Elixir の構文である．

```
def compile_to_nif do
  1..1_000_000
  |> Enum.map(foo)
  |> Enum.map(bar)
end

def rest_elixir_code do
  compile_to_nif
  |> IO.inspect
end
```

図 2. コード分割例

すなわち，micro Elixir / ZEAM は Erlang VM 互換を目指すのではなく，Elixir に特化してより高度に最適化したプログラミング言語処理系を目指す戦略を採っている．Erlang VM 互換を捨てる代わりに，最初のうちは Erlang VM から呼出して利用しやすい仕組みとして研究開発を進めることで，Erlang VM 互換戦略よりも早期に実務で利用できるようにした．

4.2 Erlang VM を超えていくロードマップ

次に「優れた Erlang VM よりもさらに優れた処理系を作れる勝算はあるのか？」という問いに答える．

4.1 節で述べた戦略 6 で示した Hastega (第 6.2 章) は，現行の Erlang VM には備わっていない機能である．しかも，Erlang VM に機能追加する形で実現することので，実現した暁には，Elixir / Phoenix から Hastega の機能を自由に利用できるようになる予定である．この時点で，Erlang VM より優れた処理系を作っていることになると考えている．

さらに第 1 章でも提示したように次のような野心的な研究目標を掲げている:

- 命令並列性に基づく静的命令スケジューリング (6.1 節)
- I/O バウンド処理を高速化する省メモリ並行プログラミング機構 Sabotender (6.3 節)
- 実行時間予測に基づく静的タスクスケジューリングとハードリアルタイム性 (7.1 節)

- プロセス間通信を含む超インライン展開 (7.2 節)
- 超インライン展開や静的タスクスケジューリングを前提にした大域的なキャッシュメモリと I/O の最適化 (7.3 節)

4.3 Erlang VM との関係性

次に「さらに micro Elixir / ZEAM を研究開発していくことに意義はあるのか？」という問いに答える．

4.1 節，4.2 節で述べたように，当面は Erlang VM に機能を後付けする形で研究開発を進めていくことから，Erlang VM の性能を高めたり，機能性を増したりする意義が充分あると考えている．

研究開発が進み，micro Elixir の言語仕様の範囲が十分に広がって Elixir の処理系として実用上使える状態になった段階で，Erlang VM から離れた独立した処理系を提供することを考えている．その主な目的の 1 つは，Erlang VM では実現できないような研究目標を達成することである．その段階まで達成できれば，さらに意義があると言えるだろう．

5 micro Elixir / ZEAM の基本構成

micro Elixir / ZEAM は次のような構成である:

- Elixir マクロを用いたメタプログラミング解析器 (5.1 節)
- LLVM を用いたコード生成系 (5.2 節)

5.1 Elixir マクロを用いたメタプログラミング解析器

Elixir には Elixir マクロという強力なメタプログラミング機構が備わっている．Elixir マクロを用いることで，Elixir の言語仕様を容易に拡張できるだけでなく，既存の言語仕様のパースを記述することなく Elixir プログラム中で Elixir プログラムの 抽象構文木 (Abstract Syntax Tree: AST) を参照・操作できる．

例えば図 1 のコードは，Elixir マクロにより図 3 のような AST に変換される:

```
{:|>, [context: Elixir, import: Kernel],
 [
   {:|>, [context: Elixir, import: Kernel],
    [
      {:|>, [context: Elixir, import: Kernel],
       [
         {:.., [context: Elixir, import: Kernel], [1, 1000000]},
         {{:., [], [{:__aliases__, [alias: false], [:Enum]}, :map]}, [],
          [{:foo, [], Elixir}]}
       ]},
      {{:., [], [{:__aliases__, [alias: false], [:Enum]}, :map]}, [],
       [{:bar, [], Elixir}]}
    ]},
   {{:., [], [{:__aliases__, [alias: false], [:IO]}, :inspect]}, [], []}
 ]}
```

図 3. コード分割例

図 3 の AST の詳説は割愛するが，この AST は Elixir の基本データ構造であるアトム `:atom` とタプル `{a, b, c, ...}`，リスト `[a, b, c, ...]` を組合わせて表現されているので，通常の Elixir のプログラムにより変換や解析を行うことができる．

micro Elixir / ZEAM では Elixir マクロを解析部に用いることで，通常のプログラミング言語処理系の実装で必要になるパースを記述する必要性が無くなり，AST を解析して中間コードを生成する本質的なプログラミングに集中できるようにした．

5.2 LLVM を用いたコード生成系

近年のコンパイラではコード生成系として LLVM [6] が採用されていることが多い．LLVM の利点は，実に多様なアーキテクチャのコードを生成できることと，豊富な最適化器のツールチェーンを活用できることである．

したがって，我々も micro Elixir / ZEAM のコード生成系として LLVM を採用したい．しかし，2019 年 1 月現在，Elixir から LLVM を利用するためのバインディングと呼ばれる API はまだ提供されていない．

そこで，我々は Elixir から Rust [11] を呼出し，Rust の LLVM バインディングを利用することで，Elixir から LLVM を用いてコード生成できるようにする．Elixir と Rust のインタフェースは Rustler [16] を用いる．

コード生成系を Elixir で記述するか，Rust で記述するかについては，次のような得失がある:

- Elixir で実装した場合には，図 3 のような Elixir コードの内部表現をネイティブに扱えることから，実装が容易である可能性が高い．しかし，第 6.2 章で後述するようなクライアントサイドでの Hastega の実行をする際に，クライアントサイドでコード生成できるようにするために，自己反映的 (self-reflective) な処理系である必要性が出てくる．自己反映的な処理系を実装するためには，注意深く設計する必要があることから，設計の難易度が増す可能性がある．
- Rust で実装した場合の利点は，既存の LLVM バインディングをそのまま利用できるので，クライアントサイドでのコード生成が容易になる．しかし，図 3 のような Elixir コードの内部表現をパースするプログラムを Rustler [16] を使って実装する必要があるため，実装が複雑になると考えられる．

双方の得失を総合判断した結果，現時点では，Elixir で記述した方が利点が大きいと考えている．

6 並列性に関わる最適化

Elixir コードから読取れる並列性には，大きく分けて次の 3 種類があると我々は考えている:

- 命令並列性: Elixir コードをデータフローで表したときに現れる並列性
- Hastega 型並列性: 例えば図 1 のコードで示されるような MapReduce プログラミングスタイルによって示唆される並列性
- Sabotender 型並列性: アクターモデル [10] にしたがった並行プロセスモデルで現れる並列性

それぞれの並列性を考慮して，次のように並列化・高速化を行う:

- 命令並列性: 6.1 節
- Hastega 型並列性: 6.2 節
- Sabotender 型並列性: 6.3 節

6.1 命令並列性に基づく静的命令スケジューリング

典型的には図 1 に示されるように，Elixir は与えられたデータに対し，パイプライン演算子でデータフローを形成して次々とデータ変換を行うことで，計算が進行する，いわばデータ変換パラダイムあるいはデータフローパラダイムとでも言うべきプログラミングパラダイムであると解釈することもできる．

MapReduce スタイルで書かれた Elixir コードでは，データフローを容易に読取ることができるため，命令並列性を読取ることも容易である．このことを利用して，命令並列性を最大限生かす形で，静的に命令スケジューリングすることが可能となる．

現代の CPU の多くは，アウト・オブ・オーダー実行方式とパイプライン実行方式を併用しており，命令並列性を利用してレジスタや ALU などの資源割当てを最適化している．Elixir コードから抽出される命令並列性を用いて，アウト・オブ・オーダー実行による最適化が働きやすいように命令スケジューリングを行うことがまず考えられる．

アウト・オブ・オーダー実行方式には，コアサイズと消費電力が大きく増加するという弱点がある．一方，Elixir の持つ並列処理の性能特性を考慮すると，アウト・オブ・オーダー実行方式を採用せずに，代わりにコア数を増やしたほうが性能が向上する可能性が大いにある．組込みシステム向け ARM ではアウト・オブ・オーダー実行方式を採用しないことで消費電力を減らしている CPU が存在する．このような場合に，Elixir コードから静的に抽出される命令並列性を元に命令スケジューリングを行うことで実行効率を高めることができるだろう．あるいは命令並列性の抽出が容易であるのであれば，アウト・オブ・オーダー実行方式の代わりに VLIW アーキテクチャを採用することを再検討する価値もあると考えられる．VLIW 方式の方がアウト・オブ・オーダー実行方式よりコアサイズを抑えることが可能である．

そこで我々は RISC-V [8] をベースとして Elixir の高速実行にふさわしいプロセッサアーキテクチャを検討する研究もスタートさせる．かつて RISC が生み出された時に，コンパイラ側の要望や制約を基にプロセッサアーキテクチャをフルスクラッチで検討していた歴史をなぞりたいと考えている．

なお，この命令並列性に基づく静的命令スケジューリングのアイデアの源泉は，Guarded Horn Clauses (GHC) [22] や KL1 [21] に由来する．

6.2 CPU / GPU を統合する超並列高速実行処理系 Hastega

図 1 のプログラムコードは容易に並列実行が可能である．1 から 1,000,000 の各要素に対する関数 `foo` と関数 `bar` の適用は他に影響されることなく完全に独立して実行することが可能である，すなわちこのプログラムは 1,000,000 の並列性を持つことが容易に推測できる．

Flow [24] という並列処理ライブラリは，このことを利用してマルチコア CPU による並列プログラミングを簡便にする方法を実現した．

また，我々も Hastega という CPU / GPU を統合する超並列高速実行処理系を提案し，研究開発している [30] [28]．Hastega のプロトタイプ実装は，Python [25] 上の GPGPU ライブラリである CuPy [19] を用いた場合と比べて 3 倍以上の速度向上を達成した [30]．

Hastega が目標としているのは，次のことである:

- CPU バウンドな処理を並列化・高速化する
- MapReduce スタイルで書かれた Elixir コードから，CPU の SIMD 命令や RISC-V [8] のベクター機能拡張 RV32V，GPU 命令を用いたネイティブコードを生成する
- サーバーサイドでの実行だけでなく，Phoenix [17] から生成する WebGL 2.0 [9]，WebGPU [26]，WebAssembly [27] などを用いてクライアントサイドでも実行できる
- Sabotender と連携して計算負荷分散・スケジューリングを行う

- Hastega の主要なアプリケーションとして，行列計算，線形回帰，機械学習，データ分析，レイトレーシングなどを想定している．これらのアプリケーションはベクタ計算を基調としており，Hastega による高速化を図りやすい．

現在イメージしている Hastega のコード例を図 4 に示す．

```
defmodule SomeModule do
  require Hastega
  import Hastega

  defhastega do
    def func do
        1..1_000_000
        |> Enum.map(foo())
        |> Enum.map(bar())
        |> IO.inspect
    end
    hastegastub
  end
end
```

図 4. Hastega のコード例

`require Hastega` は Hastega の処理系を有効にし，`import Hastega` は `Hastega.defhastega` などと書かずに `defhastega` とだけ書けばいいようにする．`defhastega` と `hastegastub` は Hastega に定義されているマクロである．`defhastega` は，続く `do` ブロック内を Hastega 処理系に渡す．`hastegastub` は，スタブコードを生成するマクロで，`defhastega` に続く `do` ブロック内で定義されている関数 (ここでは `func`) それぞれについて，ネイティブコードと，Elixir からネイティブコードの呼出しをするスタブコードを生成する．

`defhastega` にはオプションをつけることができ，CPU と GPU のどちらで実行するかやコードを実行するコア数などを指定することができる．オプションを指定しなかった場合には，適切に負荷分散するようにする．

我々は micro Elixir / ZEAM が実現する最初の機能として，Hastega を選んだ．その理由は，実現した場合の訴求力が高いこと，Erlang VM から NIF を呼出す形で実現しやすいことが挙げられる．

6.3 I/O バウンド処理を高速化する省メモリ並行プログラミング機構 Sabotender

I/O バウンドであるような処理を高速化するためには，高速にコンテキストスイッチでき，省メモリ性の高い並行プログラミング機構と，非同期 I/O の採用が有効である．Node.js [7] は Node プログラミングモデル [20] に基づくコールバックを用いた新しい並行プログラミング機構と非同期 I/O を全面的に採用することで，I/O バウンド処理を高速化することに成功した．

我々は Node プログラミングモデル [20] に基づく省メモリの並行プログラミング機構を C++ と Elixir で実装して省メモリ性能を比較した [31]．Node プログラミングモデルを用いると従来のマルチプロセス方式やマルチスレッド方式に比べて，スタックメモリを使用せずにマルチタスクを実現できるので，特にウェブサーバーの同時セッション最大数やレイテンシを大幅に改善できる [18] [29]．

我々がメモリ消費を計測したところ，C++ による実装で 1 スレッドあたり約 200 バイト，Elixir による実装で 1 スレッドあたり約 1.3KB であった [31]．さらに追試したところ，単方向リスト方式

の採用により C 言語による実装で 1 スレッドあたり 50 バイト強の省メモリ性能を達成できる可能性もあることがわかった．

このプロトタイプ実装の結果を受けて，micro Elixir / ZEAM における省メモリ並行プログラミング機構としては，並行プログラミング機構をネイティブコードで実装して Elixir から利用できるようにした方が，より省メモリ性能を追求できることが明らかになった．Node プログラミングモデルに基づく micro Elixir / ZEAM における省メモリ並行プログラミング機構を Sabotender と呼称する．

Sabotender の研究目標は次の通りである:

1. 省メモリ: 1 タスクあたり 200 バイト程度もしくは 200 バイトを切る省メモリ性能を備える
2. 同時セッション最大数: Phoenix に適用した時に従来の 10 倍程度以上の同時セッション接続数を達成する
3. コンテキストスイッチの効率性: 従来のマルチスレッド方式よりも高速にコンテキストスイッチできるようにする
4. マルチコア CPU の効率性: Erlang VM で探求されてきたようなマルチコア CPU における同期・排他制御を徹底的に排除する設計思想を踏襲して，マルチコア CPU で効率よく実行できるような設計にする
5. 耐障害性: GC を含むメモリ管理機構との連携と，Elixir の並行プロセス API である GenServer と互換性のある API を提供することで，従来の Erlang VM で達成していた高い耐障害性を維持する
6. イベント処理能力: アクターモデル [10] に基づく並行プログラミングスタイルを維持したまま，従来の Erlang VM で頻発していたイベントキューが「詰まる」問題を抜本的に改善する
7. 安全性: モデル検査と型検査などを駆使して，処理系の不具合によるデッドロックや不公平性，型不一致を防止する

3 について，従来方式では，コンテキストスイッチをするときにレジスタの退避やメモリ空間の切り替えなどを行う必要があるのに対し，Sabotender では，コンテキストが関数へのコールバックとして表現されること，プリエンプションが無くコールバック単位での擬似プリエンプションであることから，各コールバックでレジスタ生存期間が完結するのでコンテキストスイッチのためにレジスタを退避する必要がないと考えられる．また，Sabotender では，メモリ空間の切り替えは関数の実行に必要な実行時環境の切り替えとして表現される．このようなことを利用して，コンテキストスイッチにかかる時間を削減する．

4 について，現状でも数コア〜数十コアのオーダーのマルチコア CPU が市販されており，将来的には数百コア〜数千コアのオーダーのマルチコア CPU が実現する可能性もある．Sabotender は，このようなマルチコア CPU での運用を前提とし，効率よくマルチコア CPU を活用できるような設計，すなわち，従来の Erlang VM の設計で追求されてきたように，コア間の同期・排他制御を極力しないで済むような設計を追求する．

たとえば Erlang VM では，プロセスごとに独立したメモリ管理を行う分散メモリ方式や，コアごとに独立したスケジューラーを採用することで，コア間の同期・排他制御を行う状況を排除している．Sabotender でも，このような設計思想を継承し，徹底的にコア間の同期・排他制御を削減する．またコア数が数十以上になってくると，計算量も問題になる．$O(n)$ 以上の計算量のアルゴリズムではなく，たとえば $O(\log n)$ のアルゴリズムの採用も必要になるだろう．どうしても必要なコア間の同期・排他制御については，計算量の少ないアルゴリズムの採用を進める．

なお，Elixir ではプロセスごとに独立したメモリ管理を行うことから，分散メモリモデルを採用することが可能である．分散メモリ方式を採用できることで，MIMD 方式でありながら数千コアのオーダーのマルチコア CPU を実現できる可能性が高まる．この場合，アクターモデル [10] に基づ

くプロセス間通信を模す形でコア間の同期・排他制御機構を設計すると良い．

5 について，プロセスごとに独立したメモリ管理が高い耐障害性につながる理由については第 3 章で述べた．Elixir の並行プロセス API である GenServer によるプロセス管理機構により，各プロセスが正常に機能しているのか異常状態なのかを監視し，プロセスに異常が起こった時にエラーログに記録してプロセスを再起動するなどの適切な例外処理を行うことができる．Sabotender でもこのような設計の思想と方針を継承し，Sabotender の導入に合わせて改良を重ねることで，高い耐障害性を維持・向上する．

6 について，第 3 章に述べたように，Elixir では，アクターモデル [10] の採用により，同期・排他制御の必要性を大幅に削減している．しかし，その代償として，通信が集中する資源管理プロセスには慢性的に処理待ちのイベントが蓄積されやすく，イベントキューが「詰まって」処理が滞る不具合がしばしば発生する．

Sabotender で実現しようとしているのは，イベントキューが「詰まる」前に，アクセスが集中する資源管理プロセスを Sabotender によって多重化することで，処理が滞ることを未然に防止することである．しかも，アクターモデルに基づくプログラミングスタイルを堅持したままで，処理の多重化を実現することを目指している．

そのためには，Node [20] のように，資源への同期アクセスを徹底的に廃し，すべて非同期アクセスで統一することが肝要である．非同期アクセスを徹底するために構文を用意し，非同期アクセスの構文を活用した静的解析と最適化を進める方針を採る．

また，アクセスの集中をできるだけ早く，できれば事前に予測して，適切にコアを割り当てて負荷分散を図ることも有効である．アクセス集中の予測のため，第 7.3 章で述べる I/O 制御の最適化を考えている．さらに，資源管理プロセスと，資源管理プロセスへの通信を対応づけて印づけをすることで，プロセスのスケジューリングを行う際にアクセスが集中することが予測されるプロセスを優先的に多めの実行時間が割り当てられるようにスケジューリングすることも考えられる．

7 について，このような高度な処理系を信頼できるものにするためには，処理系自体に不具合が織り込まれない仕組みに設計することが肝要である．想定される不具合としては，データ型の不一致による問題のほかに，並列プログラミングに起因するデッドロックや不公平性といった不具合が考えられる．データ型の不一致などの問題については，型理論によって型安全性を追求するアプローチや，Alloy [15] のような反例を例示することによるアプローチが有効である．また，並列プログラミングに起因する問題については，SPIN [12] などのモデル検査のようなアプローチが有効である．Sabotender の研究開発にあたり，このような理論的アプローチを積極的に導入して進める．

7　大域的な最適化

本章では，まず次の要素技術を提案する．

- 実行時間予測，静的タスクスケジューリング，ハードリアルタイム性の実現 (7.1 節)
- 超インライン展開 (7.2 節)

さらに，それらを統合して大域的な最適化を行うことについて述べる (7.3 節)．

7.1　実行時間予測に基づく静的タスクスケジューリングとハードリアルタイム性の実現

停止性問題，すなわち「任意のプログラムが有限時間内で終了 (停止) するかどうかを，アルゴリズムによって証明することはできない」という定理が存在することにより，任意のプログラムについて，そのプログラムを実行するのにどのくらいの時間がかかるのかを推定することは不可能であるとされてきた．

しかし，MapReduce プログラミングスタイル [3] を徹底し再帰呼出しを禁止することで，有限の長さのデータを走査するプログラム片を帰納的に構成したプログラムに限定できる．そのようなプログラムは数学的帰納法により有限時間内で終了することを証明できると考えている．これにより，実用的な範囲で停止性問題の限界を超えることができる．

また，Elixir が提供しているライブラリである Stream を用いると MapReduce プログラミングスタイルでも無限長のデータを操作するプログラムを記述できる．しかし，Stream とライブラリ名を明示して使用することから，無限長のデータを扱う可能性があることを容易に認識できる．このことを利用して，有限時間内で終了するのかについて安全かつ近似的な判定をすることは比較的容易ではないかと考えている．さらに有限時間内で終了しない場合についても，有限時間で終了する部分区間に区切ることは可能である．

これらの考え方を応用すると，機械語命令の実行時間の数理モデルとデータ長が与えられた時に，プログラムの実行時間をデータ長の関数として推定することが可能なのではないかと考えている．実行時間予測を静的にかつより厳密に行える可能性が拓ける．

実行時間予測により，Sabotender で実行されるコールバック関数で表される各タスクの所要時間を推定できる．これを元に静的スケジューリングを行うことで，今まで以上にスケジューリングを最適化できる可能性がある．さらに Hastega と連携することで，CPU コアだけでなく GPU も統合してスケジューリングを最適化できるだろう．

実行時間予測に基づく静的スケジューリングにより，従来のマルチプロセス／マルチスレッド方式の動的スケジューリングより容易にハードリアルタイム性の保証ができるようになると期待している．

従来の優先度スケジューリングでは，例えばデッドラインが遠くに設定されたタスクのデッドラインを確実に保証するには，デッドラインが近づくにつれて優先度を高めるといった動的な制御を行う必要があった．また，優先度が低いタスクが実行可能になった後で，優先度の高いタスクの実行が割込む場合に，優先度逆転の問題が起こってしまうため，優先度の低いタスクの優先度を高める動的な制御が必要である．人間であれば，タスクの閑散期に優先度の低いタスクや整理整頓などを集中的に実行することで繁忙期に備えることもできるが，従来の動的スケジューリングでタスクの閑散期に優先度の低いタスクや GC などを集中的に行うようなことは出来なかった．

我々はデッドラインを元に静的タスクスケジューリングを行う方式を考えている．これにより，デッドラインが遠くに設定されているタスクも，優先度逆転の問題も，より自然な形で実現できると考えている．またデッドラインで管理することで，閑散期・繁忙期を認識しやすくなるので，閑散期に優先度の低いタスクや GC などを集中的に実行する制御を行うこともできるようになるだろう．

実行時間予測と静的タスクスケジューリングを前提とすると，ハードウェア割込みについても，CPU に組込まれた割込み回路を実装せずに，実行時間予測によって保証されるデッドライン以内に実行するポーリングによってソフトウェア的に割込みを実現することも可能になる．この利点は，割込みに伴うレジスタ退避やメモリ空間の切替えが単純・省力化できることと，パイプラインやキャッシュメモリなどが割込みによって乱れないことによる高速化が期待できることが利点として考えられる．

静的タスクスケジューリングを用いた他の最適化のアイデアについては，7.3 節で述べる．

7.2 プロセス間通信を含む超インライン展開

Elixir は，オブジェクト指向プログラミング言語と異なり，関数の呼出先は静的に決定されていることがほとんどで，動的になるのは，変数として保持している関数をドット演算子で呼出す時と，プロトコルを用いる時のみである．

したがって，やろうと思えば各プロセスで実行する関数群を全てインライン展開することもできる．動的になる場合についても，変数の静的に解析したり，プロトコルとその実装を定義する

`defprotocol` `defimpl` の関係性を解析したりすれば，インライン展開することは可能であると考えられる．

このようなインライン展開を超インライン展開 (super inlining) と呼ぶことにする．

超インライン展開をするにあたって，Elixir が採用しているアクターモデル [10] に基づくプロセス間のメッセージ通信が，各プロセスの主要な外部入出力の 1 つとなっているので，このプロセス間のメッセージ通信をいかに静的に解析できるかが，超インライン展開を行った際の性能向上につながる重要な位置付けになると考えている．

したがって，プロセス間通信の静的解析を含めた大域的な静的解析を行って超インライン展開を行うことを研究目標の 1 つとする．

超インライン展開を行った後の最適化については，7.3 節で述べる．

7.3 大域的なキャッシュメモリと I/O 制御の最適化

7.1 節で述べた静的タスクスケジューリングと，7.2 節で述べた超インライン展開を活用した最適化として，大域的なキャッシュメモリと I/O 制御の最適化を提案する．

プロセッサ技術の発展に伴って，CPU とメモリや I/O との速度差は大きく開いている．したがって，他のどのような最適化よりも，キャッシュメモリや I/O 制御の最適化の方が大きな速度向上の効果がある．

キャッシュメモリや I/O 制御の最適化の基本は，できるだけ早い段階でプリフェッチなどの準備を行い，できるだけ遅い段階で結果を取得したりライトバックを行ったりするようにして，その間に結果に依存しない計算を粛々と行ったり，結果を予測した投機的実行を行ったりするような命令スケジューリングを行うことである．

従来のプログラミング言語処理系では，命令スケジューリングは関数単位でしか行ってきていなかった．インライン展開を行えば関数を超えて命令スケジューリングができるようになるが，せいぜい 1〜2 関数をまたがってインライン展開するのみであった．命令スケジューリングを行う範囲が狭いと，メモリや I/O を充分な時間待つことができないため，結果を取得するために CPU をストールさせてしまうことになる．

そこで，超インライン展開を行うことによって，プロセスが呼出す全ての関数群にまたがった命令スケジューリングを行えるようになる．

また，静的タスクスケジューリングと組合わせることにより，タスクのコンテキストスイッチの前からメモリや I/O のプリフェッチを行い，コンテキストスイッチ後にメモリや I/O へのライトバックを行うような命令スケジューリングも可能になる．Sabotender によるコールバックでタスクが構成されるので，プリフェッチとライトバックをあらかじめ分離したコールバックとして定義しておけば，このようなスケジューリングは容易になる．

さらに，I/O 制御をデータベースのように，アトミックな処理を単位として記述し，場合によってはロールバックを可能にするというアイデアも考えている．これにより，デッドラインの関係上，どうしてもタスクの優先度逆転が生じてしまう場合に，優先度の低いタスクの I/O 制御をアボートしたりロールバックしたりすることで，速やかに優先度の高いタスクへ制御権を渡すことが可能になる．これにより，よりデッドライン制約が厳しい場合でもデッドライン制約を満たすことが可能になると考えている．

8 他の Erlang VM 互換処理系との関係性

本章では次のことについて述べる:

- 関連研究: 他の Erlang VM 互換処理系 (8.1 節)
- 他の Erlang VM 互換処理系との関係性 (8.2 節)

8.1 関連研究

他の Erlang VM 互換の処理系の概要については以下の通りである:

- AtomVM [2] は組込みシステム向けに開発された Erlang VM 互換の処理系である．マイコンボードでの実行を想定している．我々の実測では，AtomVM のバイナリイメージ (elf) のサイズは約 676KB に収まっている．インタプリタ実行のため，低速である．
- core_erlang [14] は Rustler [16] の作者による Erlang VM 互換処理系である．Rust [11] を用いて実装しており，Core Erlang と名づけられた Erlang のサブセットプログラミング言語から低レベル中間表現 (LIR) へのコンパイルと，LIR によるインタプリタ実行が行える．現時点では `unsafe` を用いていないため，メモリ安全・型安全である．インタプリタ実行のため，低速である．
- Starlight [32] も Rust [11] による Erlang VM 互換処理系である．Rust の特性を生かして，メモリ安全・型安全と高速性を両立し，かつできる限り GC が無くなるように設計・実装を進めている [32] [33].
- Enigma [13] も Rust [11] による Erlang VM 互換処理系である．Erlang 標準のコンパイラである `erlc` が生成する BEAM バイトコードを実行する．`unsafe` を使用しているため，完全にはメモリ安全・型安全ではない．

8.2 他の Erlang VM 互換処理系との関係性

4.2 節に述べたように，当面は既存の Erlang VM に機能付加する形で micro Elixir / ZEAM の研究開発を進める．この戦略は他の Erlang VM 互換のプログラミング言語処理系についても同様に考えることができる．すなわち，それらの処理系についても，Hastega (6.2 節) のような micro Elixir / ZEAM の機能を提供しうる．

一方で，Sabotender (6.3 節) のように，NIF のような形で外付けしたり，現行の Erlang VM を修正して機能を取り入れることが難しいものも存在する．このような機能については，Erlang VM から独立した処理系を構築する際に取り入れる．

9 まとめと将来課題

本論文では次のことを述べた:

- 第 2 章では，Elixir における MapReduce プログラミングスタイルについて述べた．
- 第 3 章では，Elixir を支える処理系である Erlang VM とその優れた点である並行プログラミングと耐障害性について説明した．
- 第 4 章では，micro Elixir / ZEAM の実装戦略について,「現在主流の Erlang VM から円滑に移行することができるのか？」「優れた Erlang VM よりもさらに優れた処理系を作れる勝算はあるのか？」「すでにたくさんの Erlang VM 互換のプログラミング言語処理系が数多く提案されている中で，さらに micro Elixir / ZEAM を研究開発していくことに意義はあるのか？」の 3 つの問いに答える形で説明した．
- 第 5 章で，micro Elixir / ZEAM の基本構成と実装方針について解説した．
- 第 6 章では Elixir コードから読取れる 3 種類の並列性について紹介し，この 3 分類に沿って，6.1 節で命令並列性に基づく静的命令スケジューリングについて，6.2 節で CPU / GPU を統合して CPU バウンド処理を高速化する超並列高速実行処理系 Hastega について，6.3 節で I/O バウンド処理を高速化する省メモリ並行プログラミング機構 Sabotender についてそれぞれ紹介した．

- 第 7 章で，実行時間予測に基づく静的タスクスケジューリングとハードリアルタイム性の実現 (7.1 節) と，プロセス間通信を含む超インライン展開 (7.2 節) について提案した後，それらを前提とした大域的なキャッシュメモリと I/O の最適化の可能性について議論した (7.3 節)．
- 第 8 章で，他の Erlang VM 互換の処理系について概観した (8.1 節)．また，それらに対しても我々の提案手法を可能な限り提供する方針であることを示した (8.2 節)．

将来課題として，まず着手するのは Hastega の実装である．`Enum.map` に対し，CPU の SIMD 命令の発行によるソフトウェア・パイプラインによる最適化を実装することから始める．そのためには，Elixir マクロを使って LLVM のコードを生成する実装を確立する必要がある．

次に，多くの提案機能を支えているのは実行時間予測である点が明らかになったことから，実行時間予測の実現に向けて，有限長のデータの走査を含むプログラム片を帰納的に構成したプログラムが必ず有限時間内で終了することを証明し，リアルタイム性の実現や，静的解析・形式検証技術などの綿密なサーベイを行った上で，実行時間予測を理論的に体系づけて設計・実装する．

以上の実装と理論体系を整備した後であれば，各論の中から着手しやすい部分から順番に実装して研究開発を進めることが可能になると考えている．

以上を前提に，今までの常識を打ち破るようなカーネルや VM，プロセッサアーキテクチャ，並列/分散コンピューティング，各種数学・機械学習等特定ドメイン向け最適化等の研究に取り組む．

謝辞

本研究の一部は，北九州産業学術推進機構（略称：FAIS）の新成長戦略推進研究開発事業「シーズ創出・実用性検証事業」の支援を受けた．デライトシステムズの上野 嘉大氏， rust-jp Slack チームの特に Tatsuya Kawano 氏，クックパッドの笹田耕一氏をはじめとする第 120 回情報処理学会プログラミング研究会での発表 [30] での質疑応答をしていただいた方々，さくらインターネットの松本亮介氏，北陸先端科学技術大学院大学の青木利晃先生，平成 30 年度の北九州市立大学のディジタルシステム設計と組込みソフトウェアの受講者の方々にはとくに有益な助言を多数いただいた．

参考文献

[1] Adams, J.: Elixir Users' Survey 2016 Results, 2016. https://www.dailydrip.com/blog/elixir-users-survey-2016-results.html.

[2] Bettio, D.: AtomVM: how to run Elixir code on a 3 $ microcontroller, 2018. https://medium.com/@Bettio/atomvm-how-to-run-elixir-code-on-a-3-microcontroller-b414773498a6.

[3] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM*, Vol. 51, No. 1(2008), pp. 107–113.

[4] Ericsson: Erlang: build massively scalable soft real-time systems, 1998. http://www.erlang.org.

[5] Fedrecheski, G., Costa, L. C. P., and Zuffo, M. K.: Elixir programming language evaluation for IoT, *2016 IEEE International Symposium on Consumer Electronics (ISCE)*, Sept 2016, pp. 105–106.

[6] Foundation, L.: The LLVM Compiler Infrastructure, 2002. http://llvm.org.

[7] Foundation, N.: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine., 2009. `https://nodejs.org/`.

[8] Foundation, R.: RISC-V: The Free and Open RISC Instruction Set Architecture, 2010. https://riscv.org.

[9] Group, K.: WebGL 2.0 Specification, 2018. https://www.khronos.org/registry/webgl/specs/latest/2.0/.

[10] Hewitt, C., Bishop, P., and Steiger, R.: A Universal Modular ACTOR Formalism for Artificial Intelligence, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[11] Hoare, G.: Rust: Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety., 2006. https://www.rust-lang.org/.

[12] Holzmann, G. J.: Verifying Multi-threded Software with Spin, 1989. http://spinroot.com/spin/whatispin.html.

[13] Hrastnik, B.: Enigma: A clean and simple implementation of the Erlang VM in Rust., 2018. `https://github.com/archSeer/enigma`.

[14] J., H. E.: Core Erlang compiler implemented in Rust, 2017. `https://github.com/hansihe/core_erlang`.

[15] Jackson, D.: Alloy is an open source language and analyzer for software modeling., 1997. http://alloytools.org.

[16] Josephsen, H. E. B.: Rustler: Safe Rust bridge for creating Erlang NIF functions, 2015. https://github.com/hansihe/rustler.

[17] MacCord, C.: Phoenix: A productive web framework that does not compromise speed and maintainability., 2014. https://phoenixframework.org.

[18] McCune, R. R.: Node.js Paradigms and Benchmarks, 2011. STRIEGEL, GRAD OS F'11, PROJECT DRAFT.

[19] Preferred Networks: CuPy: A NumPy-compatible matrix library accelerated by CUDA, 2005. https://cupy.chainer.org.

[20] Tilkov, S. and Vinoski, S.: Node.js: Using JavaScript to Build High-Performance Network Programs, *IEEE Internet Computing*, Vol. 14, No. 6(2010), pp. 80–83.

[21] Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *The Computer Journal*, Vol. 33, No. 6(1990), pp. 494–500.

[22] Ueda, K.: Guarded Horn Clauses, Technical report, ICOT, June 1985. http://www.ueda.info.waseda.ac.jp/ ueda/pub/tr103new2.pdf.

[23] Valim, J.: Elixir: Elixir is a dynamic, functional language designed for building scalable and maintainable applications., 2013. https://elixir-lang.org.

[24] Valim, J.: Flow: Computational parallel flows on top of GenStage, 2017. https://github.com/elixir-lang/flow.

[25] van Rossum, G.: Python: Python is a programming language that lets you work quickly and integrate systems more effectively., 1991. https://www.python.org.

[26] W3C: GPU FOR THE WEB COMMUNITY GROUP, 2018. https://www.w3.org/community/gpu/.

[27] WebAssembly Community Group: WebAssembly specification, reference interpreter, and test suite., 2015. https://github.com/WebAssembly/spec/tree/v1.0.

[28] Yamazaki, S.: Hastega: Challenge for GPGPU on Elixir, *Lonestar ElixirConf 2019, Austin, TX, USA*, March 2019.

[29] 山崎進, 森正和, 上野嘉大, 高瀬英希: Elixir の軽量コールバックスレッドの実装と Phoenix の同時セッション最大数・レイテンシ改善の構想, 第 *2* 回 *Web System Architecture* 研究会, 福岡, 2018. The paper and the presenation are available at `https://zeam-vm.github.io/papers/callback-thread-2nd-WSA.html` and `https://zeam-vm.github.io/zeam-WSA-20180512/#/`, respectively.

[30] 山崎進, 森正和, 上野嘉大, 高瀬英希: Hastega: Elixir プログラミングにおける超並列化を実現するための GPGPU 活用手法, 第 *120* 回情報処理学会プログラミング研究会, 熊本, 情報処理学会プログラミング研究会 (PRO), Vol. 2018, No. 2, 東京, 8 月 2018, pp. (8). The paper and the presentation are available at `https://zeam-vm.github.io/papers/GPU-SWoPP-2018.pdf` and presentation is available at `https://zeam-vm.github.io/GPU-SWoPP-2018-pr/`, respectively.

[31] 山崎進, 森正和, 上野嘉大, 高瀬英希: Node プログラミングモデルを活用した C++および Elixir の実行環境の実装, 研究報告システムソフトウェアとオペレーティング・システム（*OS*）, Vol. 2018-OS-144, No. 1, 東京, 7 月 2018, pp. 1–6. presentation is available at `https://zeam-vm.github.io/LCB-SWoPP-2018-pr/`.

[32] 鈴木鉄也: Rust で Erlang 処理系を実装してみている, 2018. https://medium.com/@szktty/rust-で-erlang-処理系を実装してみている-d5e3edb25b82.

[33] 鈴木鉄也: 年末年始 Rust 振り返り, 2018. https://medium.com/@szktty/年末年始-rust-振り返り-7e0601f5aa3d.