ディジタル回路自動設計に適した
プログラミングモデルと言語の研究

Practical programming models and languages for digital circuit design automation

照屋 大地

Daichi TERUYA

(2018年度入学, 18834303)

指導教員 中條拓伯 准教授

東京農工大学 工学府 博士後期課程
電子情報工学専攻 知能・情報工学専修

2020年度博士論文

(2021年1月31日提出)

東京農工大学 工学府 博士後期課程
電子情報工学専攻 知能・情報工学専修 2020 年度博士論文 要旨
題目　　ディジタル回路自動設計に適したプログラミングモデルと言語の研究
Practical programming models and languages for digital circuit design automation
学籍番号 18834303　　　　氏名　　照屋 大地 (Daichi TERUYA)
提出日 2021 年 1 月 31 日

ASIC や FPGA で用いるディジタル回路の設計は, ハードウェア記述言語 (hardware descripttion language; HDL) を用いたレジスタ転送レベル (Resister Transfer Level; RTL) による開発が必要となる．記述の性質から HDL はバグが発生しやすく，デバッグにも様々なツールが必要で熟練した技術が必要である．そこで C/C++や Java といったプログラミング言語によって記述されたプログラムやアルゴリズムと同じふるまいをする回路を合成する技術である高位合成（high-level synthesis; HLS）と呼ばれる技術が注目され既に様々な言語による HLS ツールが開発されている．しかしながらソフトウェア開発と回路設計の記述モデルには大きな隔たりがある．ソフトウェアのプログラムには並列性やデータフローに関する情報が含まれておらず，最適化のための情報を付け加えなければスループットが非常に低くなってしまうことや，複雑で巨大なステートマシンが生成されてしまい，自動的な並列化やパイプライン化が難しい問題が発生する．

そこで本研究は，FPGA を主なターゲットとして，ソフトウェア開発の分野で生産性を高めるために用いられているプログラミングモデルによってディジタル回路設計の生産性を高めることを目的とした研究を行った．本研究は大きく分けて 1) フレームワークを用いた最適なアーキテクチャの自動生成ツールの研究，2) ディジタル回路設計に近いプログラミングパラダイムを用いた回路設計ツールの研究の二つのアプローチに分けられる．

まずフレームワークによる設計支援ツールに取り組んだ．センサ入力が必要となる組み込みシステムにおいて，一般的な汎用マイクロコントローラ（MCU）では通信速度やセンサ数が増えるにつれデータハンドリングが困難になる．そこで，ユーザが定義したセンサ入力やデータ処理の一部を FPGA へオフロードしたアーキテクチャを自動的に生成するフレームワーク，PyJer を提案した．生成アーキテクチャを限定的にすることで既存の最適化手法やツールの自動的な適用が行いやすく，ユーザによる専門的知識に基づくコードの最適化を最小限に抑えることができる．評価のため簡易的な 2 次元ビームフォーミングによる音源方位推定システムを構築した．作成したシステムのコード量は，Java が 398 行，Python が 30 行，Verilog HDL が 118 行となった．実機検証の結果，ビームフォーミング処理の動作レートは約 174[Hz] と CPU で動作させた場合の 691[Hz] と比較して低速なものとなったが，センサとの通信は全て並列化することができるためセンサの数が増えたとしても処理レートは安定する．また， FPGA の LUT 使用量が 10%程度，BRAM の使用量が 15%程度と低くなった．CPU-FPGA 間通信のための機構などによるリソース消費のオーバヘッドを小さく抑えながらもセンサとの通信の並列化と実用上十分なデータ処理速度を達成することができた．

PyJer の中では Java 言語ベースの HLS ツールを利用している．HLS ツールを用いてソフトウェアとして利用されてきた既存のプログラムから回路を合成しようとした場合には，性能が低下してしまうことが一般的である．性能低下を回避するためには回路設計の知識と HLS ツールにおける技法を熟知したエンジニアによるコードの最適化が必要となる．筆者は，ほとんどの HLS ツールにおいてプログラムの最適化が必要であることや C や Java の一部の言語機能に制約が必要な理由は，回路設計とプログラミングのパラダイムに大きな開きがあることであると考えた．この違いによって大局的な並列性の解析や複雑なループの並列化など現在のコンパイラ技術でも困難な問題を引き起こし，ユーザによるコードの最適化を行う必要が生まれる．そこで本研究では，

ディジタル回路設計と近いパラダイムを持ったプログラミングモデルである functional reactive programming (FRP) を用いたハードウェア設計ツール Mulvery を提案する．Mulvery は CPU と FPGA が共存するアーキテクチャにおいて利用されることを想定しフレームワーク化することでソフトウェアとハードウェアの協調設計を実現する．画像処理を例に評価を行ったところ，5x128 ピクセルの画像と 5x5 ピクセルのフィルタの畳み込み演算を 100MHz の動作周波数で 1 クロックで実行することができ，アーキテクチャ探索等を行うことなく高いスループットの回路を合成することを確認できた．また 32x32 個の LED マトリクスを用いた実験では，MCU のみでは LED マトリクスの制御さえ困難であったが，Mulvery はハードウェアオフロードによる LED マトリクス制御の高速化に加え，Ruby 言語の既存ライブラリなどを用いて Web サーバを記述することも可能で，ネットワーク経由での画像表示まで可能なものにした．

これらの応用に関する研究として，関数ポインタのサポートとクラウドコンピューティングへの FPGA の応用技術についての研究を行った．HLS においては，ポインタのように空間上の位置が動的に変化するようなプログラムを合成することは困難である．本研究で C 言語を用いる HLS ツールにおいてコードを抽象化し再利用性を高めるために重要な関数ポインタもこのような理由でサポートしコードの再利用性を高め，Mulvery のような FRP による高階関数を多用した抽象的なプログラミングモデルを実現するための研究に取り組んだ．クラウドコンピューティングの文脈では，Mulvery のデータフロー型の記述を生かし複数の FPGA で複数のタスクを分担して処理するための仕組みの基礎研究に取り組んだ．反応閾値モデルを応用した自律分散システムによって，メンテナンス性の高さと高いフォールトトレランスを実現する手法を提案した．

目 次

第 1 章　緒論 1
1.1 ハードウェアによるソフトウェアの高速化 1
1.1.1 ハードウェアアクセラレーション 1
1.1.2 FPGA を用いたアクセラレーション 2
1.2 ディジタル回路設計の現状と課題 3
1.2.1 ハードウェア記述言語を用いたディジタル回路設計 3
1.2.2 ディジタル回路設計の生産性を高めるためのアプローチ 3
1.3 研究目的と本稿の構成 4

第 2 章　FPGA の原理と回路設計の技術 5
2.1 FPGA の技術の概要 5
2.1.1 FPGA の原理 5
2.1.2 FPGA の EDA 技術 7
2.1.3 参考文献 9
2.2 高位合成の原理 9
2.2.1 高位合成の概要と利点 9
2.2.2 動作合成 9
2.2.3 参考文献 11
2.3 高速化技法 11
2.3.1 並行計算 11
2.4 まとめ 12

第 3 章　関連研究 13
3.1 はじめに 13
3.1.1 ソフトウェア開発の技術の有効活用 13
3.1.2 動向調査の意義 14
3.2 カスタム回路設計ツールの種別 14
3.2.1 ドメイン特化言語を用いたレジスタ転送レベル設計 14
3.2.2 汎用プログラミング言語や DSL を用いた動作レベル設計 15
3.2.3 ドメイン特化言語を用いたシステムレベル設計 15
3.3 既存の開発ツールの分類と特徴 15
3.3.1 DSL ベースの RTL 設計ツール 16
3.3.2 HLS ツール 19
3.3.3 システムレベル設計ツール 21
3.4 本研究の位置付け 23

第 4 章　フレームワークを用いたアーキテクチャレベルの最適化　25
4.1 複数のツールを組み合わせた IoT/CPS エッジデバイス向けフレームワーク 25
4.1.1 関連研究 26
4.1.2 PyJer の概要 26
4.1.3 PyJer の設計方針 28
4.1.4 PyJer が実現するシステムのアーキテクチャとデータフロー 30
4.1.5 PyJer の使用法とその実装 32
4.1.6 評価方法 37
4.1.7 アプリケーションとその評価 38
4.1.8 PyJer の有効性の検討 40
4.1.9 まとめ 41
4.2 C 言語ベースの IoT/CPS エッジデバイス向けフレームワーク 41
4.2.1 はじめに 41
4.2.2 新しいフレームワークのコンセプト 41
4.2.3 Mulvery アーキテクチャ 43
4.2.4 Mulvery Core フレームワーク 44
4.2.5 具体的なアプリケーションの実装例 47
4.2.6 データコレクタとしての活用 49
4.2.7 まとめ 50

第 5 章　ハードウェア設計に適したプログラミング・パラダイム　52
5.1 背景 52
5.1.1 前章での研究の問題点 52
5.1.2 関連研究および先行事例 54
5.2 Mulvery フレームワーク 55
5.2.1 Mulvery フレームワークの構成 55
5.2.2 Mulvery フレームワークを用いたシステム開発の手順 56
5.3 Functional reactive programming 57
5.3.1 Reactive Programming 57
5.3.2 Observer Pattern 58
5.3.3 Functional reactive programming 59
5.3.4 基本的な文法と動作 59
5.4 レジスタ転送レベル設計の合成 60
5.4.1 RxRuby による記述からハードウェアを合成する手法 62
5.5 処理の記述の透過性 66
5.6 ハードウェア合成手法の性能評価と予備実験 67
5.6.1 サンプルプログラム 67
5.6.2 ハードウェアの合成 67
5.6.3 ハードウェアの評価 68
5.7 ハードウェア・ソフトウェアの分割とシステムへの統合 69
5.7.1 ハードウェア・ソフトウェアの分割手法 69
5.7.2 ハードウェア・ソフトウェアの連携手法 70
5.8 アプリケーションの開発例とその評価 71
5.8.1 フルカラー LED の制御とアプリケーションソフトウェア 71

5.8.2 評価 . . . 73
5.9 結論 . . . 74

第 6 章 HLS における関数ポインタと高階関数のサポートとその最適化 78
6.1 はじめに . . . 78
6.1.1 HLS と関数ポインタ . . . 78
6.1.2 HLS における関数ポインタのメリット . . . 78
6.2 関連研究 . . . 80
6.2.1 定数畳み込み . . . 80
6.2.2 C++のテンプレートによる関数引数 . . . 80
6.2.3 メモリマップ型インタフェース . . . 80
6.2.4 Point-to 解析によってポインタ変数を取り除く方法 . . . 81
6.3 関数ポインタの削除 . . . 81
6.3.1 Callee の候補値探索 . . . 83
6.3.2 関数ポインタを用いている命令の置換 . . . 85
6.3.3 Void-type pointer . . . 86
6.4 Evaluation . . . 88
6.5 まとめ . . . 89

第 7 章 データフロー型プログラムと分散 FPGA 環境 90
7.1 はじめに . . . 90
7.1.1 FPGA のクラウドコンピューティングへの展開 . . . 90
7.1.2 FPGA 混載ヘテロジニアスシステム上の問題点 . . . 90
7.1.3 社会性昆虫の生態とその応用 . . . 92
7.1.4 分散システムにおける応用 . . . 93
7.1.5 目的と目標 . . . 93
7.2 反応閾値モデルの適用 . . . 93
7.2.1 反応閾値モデル . . . 93
7.2.2 CPU・FPGA の差別化と学習係数 . . . 95
7.3 新たな自律分散システムの構築 . . . 95
7.4 関連研究 . . . 96
7.4.1 ハードウェア・ソフトウェアの協調設計ツール . . . 97
7.4.2 FPGA のクラウドサービスへの導入 . . . 97
7.5 統計的モデルの利用 . . . 98
7.6 提案手法の検証 . . . 99
7.6.1 実装と実験の環境 . . . 99
7.6.2 実験モデル . . . 99
7.6.3 実験 1: 自動スケーリングの検証 . . . 100
7.6.4 実験 2: 優先度の割り当て . . . 101
7.7 クラウドシステムに向けたオートスケール機構 . . . 102
7.7.1 システム構成 . . . 102
7.7.2 PaaS 型データ分析クラウドサービス . . . 103
7.8 節のまとめ . . . 104

第 8 章　結論 **105**
8.1　ディジタル回路設計における課題 105
8.2　フレームワークによるアーキテクチャレベル最適化 105
8.3　ディジタル回路の自動設計に適した言語パラダイム 105
8.4　C 言語における関数ポインタと高階関数のサポート 106
8.5　データフロー型プログラムの分散 FPGA 環境への応用 106

図 目 次

1.1 The architecture of tipical GPU and TPU 2

2.1 Basic island-style FPGA structure [14], [15] 6
2.2 Basic logic element (BLE) [16] 7
2.3 Logic cluster [16] 8
2.4 Optimization of for-statement by loop-unrolling (top) and pilepining (bottom) . 10
2.5 Three types of parallelization 11

3.1 Example of system design using Simulink [30] 16
3.2 The graph of the synthesized hardware [6] 23
3.3 Comparison of tools 24

4.1 Structure where data flow is difficult to be modified 27
4.2 Flowchart of PDF signal DA conversion 29
4.3 The program of time measurement of mic data processing 29
4.4 Measured processing time of mic data processing 30
4.5 Hardware architecture with CoRAM 30
4.6 Arcitecture when data distribution is performed by a module in PL 31
4.7 System architectrue that PyJer generates 31
4.8 Tool hierarchy 32
4.9 Example description of data processing mechanism using Synthesijer 32
4.10 Example description of sensor data processing using Verilog HDL 35
4.11 Example of Top module using Synthesijer 35
4.12 Build flow and tools used for build automation 36
4.13 Position of sound source and its signal 37
4.14 Insertion of delay 38
4.15 Results of beam-forming 40
4.16 Utilization of SoC FPGA in IoT/CPS area 42
4.17 Description languages used in PyJer 42
4.18 Comparison of common design and Mulvery architecture 43
4.19 Example definition of connections of LIS3DHIF 47
4.20 Target of fluentd 50
4.21 Mulvery on fluentd 50
4.22 Schematic diagram of configuration with Mulvery applied to the entire data collection system 51

5.1 Comparison of architecture with MCU-only design and FPGA-based I/O processing 53
5.2 The synthesizing flow of Mulvery 55

5.3 Software architecture of Mulvery framework 56
5.4 Template of MulveryBase class 56
5.5 Example of build_dataflow method 57
5.6 Example of main function and its outputs 57
5.7 Example of counting the number of events using Reactive Programming 58
5.8 Class diagram of Observable Pattern 59
5.9 Example of Subject and Observer for mouce events 59
5.10 Example description of LINQ on C# 61
5.11 Example of event counting using RxRuby 61
5.12 Relationship among the operators, threads, and principal schedulers 62
5.13 Examples of Scedulers on RxRuby 63
5.14 Structure of a dataflow and its container 64
5.15 Workflow that generates the HDL code of a module 64
5.16 Example of Observable 65
5.17 Example of object that synthesizes a lambda abstraction 65
5.18 If method on Mulvery 66
5.19 How to specify hardware offload in SDSoC and Intel FPGA SDK for OpenCL . . 67
5.20 Example when the location where the method is executed differs between CPU and FPGA depending on how it is called 67
5.21 Applying Laplacian filter 68
5.22 Example of folding 68
5.23 Architecture of the program 69
5.24 Pipeline of Fig. 5.22 69
5.25 Synthesized lambda abstraction in Fig. 5.22 70
5.26 Hardware verification using Icarus Verilog 70
5.27 Resource consumption 71
5.28 Pseudo code for counting the number of events 71
5.29 Architecture with CoRAM 72
5.30 Data field for NeoPixel [96] 72
5.31 Signal to control NeoPixel [96] 73
5.32 32×32 LED matrix 74
5.33 Serial connection of NeoPixel 75
5.34 Example of build_dataflow to control NexPixel matrix 75
5.35 The picture used for the experiment 76
5.36 The NeoPixel showing Fig. 5.35 76

6.1 unsinthesizable pointers 79
6.2 Examples of indirect call that cannot be resolved by constant folding: (left) A CFG with a condition node, (right) a CFG with two indirect calls. 80
6.3 (top) Naive solution, (bottom) [99]'s solution 81
6.4 CFG of Listing 6.3, that contains dynamic indirect function call 83
6.5 The CFG that the indirect call has removed 84
6.6 Indirect call with a multiplexer 84
6.7 Hardware generated from the code including indirect call 89

7.1 (top) Original block diagram of `source.opA().opB()` and (bottom) the program devided between `opA` and `opB`. 91
7.2 Usual design method of distributed system which includes FPGAs 91
7.3 Organization structure of ants 92
7.4 Comparison of response curves with different θ_i 94
7.5 Graph of $P_{ij}(t)$ using Equation(7.3) ($\bar{T} = 100$)(upper:$\theta_i = 0.5$, center:$\theta_i = 1$, lower:$\theta_i = 2.0$) 95
7.6 Flow of Job processing 96
7.7 Network structure for the experiment 99
7.8 Number of Workers which is processing task for each 0.1sec of experiment 1 (Upper: a graph when give a Job, Lower: a graph when give two Jobs) 101
7.9 Number of Workers which is processing task for each 0.1sec of experiment 2 (Upper: System-E2A, Lower: System-E2B) 101
7.10 A distributed file system and compute nodes 102
7.11 Source Code division and assignment computing resources 103
7.12 An Example of Architecture of PaaS Type IoT data analytics cloud service . . . 103
7.13 Numbers of assigned workers for each time (Gray line : 1 stage, Black line : 2 stages, Dotted line : average) 104

Listings 目次

3.1 Description example of a hardware that calculates GCD using Chisel [1] 14
3.2 Example of matrix product using Vivado HLS 15
3.3 Calculating the average value of an event using Estrel [2] 16
3.4 Up-down conter by JHDL [3] 17
3.5 Description example of 8-bits up counter using CλaSH [4] 18
3.6 Calculation of $\log_2(N)$ using PyMTL's function-level [5] 19
3.7 Calculation of $\log_2(N)$ using PyMTL's cycle-level 19
3.8 Calculation of $\log_2(N)$ using PyMTL's register-transfer-level 20
3.9 Example of kernel description using MaxCompiler 22
3.10 Example of Manager on MaxCompiler [6] 22
4.1 Sensor data acquisition in Synthesijer program 33
4.2 Example of data transfer to DRAM 34
4.3 Example of Memory Access Controller 34
4.4 TCL script to update IPs for existing Block Design 36
4.5 Java program for performance measurement 39
4.6 Data Processor using Mulvery Core 44
4.7 Source code of Fig. 4.6 aftre macro expansion 45
4.8 Example of Interface on Mulvery Core framework 46
4.9 Example of Sensor Interface 46
4.10 Definition of Sensor Interface that Data Processor uses 47
4.11 Example of implementation of LIS3DH interface (1) 48
4.12 Example of implementation of LIS3DH interface (2) 48
4.13 Example of Data Processor that uses LIS3DH 49
4.14 Mulvery language usage example 51
5.1 Always statement in Verilog HDL 60
5.2 Definition of statement of Mulvery API 60
5.3 An implementation of the *from_array* operator in Mulvery 63
6.1 Examples of high-order functions 79
6.2 sample list 2 79
6.3 Example with an indirect call 82
6.4 Optimized IR of Listing 6.3 82
6.5 Three types of `store` instruction usage regarding function pointers 83
6.6 The recursive function to build adjacency graph 85
6.7 Replacement of the instructions that reference function address/pointer 86
6.8 Replacement of an indirect call 87
6.9 Map function using void pointer 87

6.10 Alternative implementation of map of 6.9 87
6.11 Sample program for the evaluation 88

表 目 次

1.1 Comparison of standard processor (e.g. CPU, MCU), standard accelerator (e.g. DSP, GPU), and ASIC (custom circuit) 2

2.1 The memory table for $f(A, B, C) = (A\&B)|(A\&C)$ 7

4.1 Software-based development environment for mic data processing 28
4.2 Platform used for experiment and verification 37
4.3 Zynq-7000 utilization 39

5.1 Schedulers implemented on RxRuby 62
5.2 Classification of Rx operators 66
5.3 Signal transmission time length and tolerance [96] 73
5.4 Specifications of the board used in the experiment [97] 74
5.5 Software environment used for the experiment 77
5.6 FPGA resource consumption 77

6.1 FPGA resource consumption (on Cyclone V GX-9) 88
6.2 Performance (simulation) 89

7.1 Computers used for the experiment 99

第1章　緒論

1.1　ハードウェアによるソフトウェアの高速化

1.1.1　ハードウェアアクセラレーション

ムーアの法則の終焉によって，ソフトウェアの性能向上は限界を迎えつつある [7]．CPU のような汎用的なプロセッサはプリミティブな命令を組み合わせて処理を行うが，**ハードウェアアクセラレータ (hardware accelerator)** と呼ばれるデバイスを目的に合わせて用いることでシステム全体の処理性能を向上させることができる．このようにハードウェアアクセラレータを用いた処理の高速化を**ハードウェアアクセラレーション (hardware acceleration)** と呼ぶ．このようなアクセラレータには特定の処理に特化した LSI (large-scale integration) が搭載されており，動画のエンコーダ・デコーダといった機能単位のアクセラレータや，DSP (digital signal processor) のようなディジタル信号処理に特化したプログラマブルデバイスなど，様々な粒度のアクセラレータが存在する．一般消費者にも広く利用されているハードウェアアクセラレータとしては，主にコンピュータグラフィクスやゲームの描画処理に利用される GPU (graphics processing unit) が知られている．浮動小数の処理に必要な機能を具えた比較的単純な PE (processing element) を数百〜数千個並べ高い並列度で処理を行うことで高い処理性能を実現している（図 1.1 上）．浮動小数演算は科学分野や機械学習，人工知能の分野でも多く利用されるため，コンピュータグラフィクス以外の分野に GPU を応用する GPGPU (general-purpose GPU) も広く行われている [8]．このようなデバイスでは，機能を特化させるほど電力性能や処理速度が向上するが，用途が限定的となり製造数も汎用的なものと比べて少なくなるため，相対的に開発製造のコストが大きくなり価格も上昇するといったトレードオフがある．

家庭用の電子機器やモバイルデバイスの開発などさらに高い電力性能が要求される場面においては個々のシステムに合わせて設計されるカスタム IC (application specific integrated circuit; ASIC) が利用される．身近な例としては，デジタルカメラの画像補正や，テレビや動画プレーヤーなどで利用される専用の動画エンコーダ/デコーダなどが挙げられる．人工知能分野，特に深層学習 (deep learning) においてはベクトルやテンソルの演算を繰り返し行うため，Google が TPU (Tensor Processing Unit) と呼ばれる高性能な計算を実現するためのハードウェアアクセラレータを発表している [9]．多数の PE を並べた点は GPU と共通であるが，TPU の場合には隣接する PE が接続（図 1.1 下）され PE 間で心臓の収縮 (Systolic) のようにデータを転送し処理を行う特徴を持つ．このようなアーキテクチャは H.T.Kung らが提案したシストリックアレイ (systoric array) として知られており [10]，これを応用して高いスループットでテンソル演算を行うアルゴリズムが知られている[1]．

特定のアプリケーション専用に LSI を開発する ASIC は最も設計の自由度が高く目的に応じて制作するものであるため性能の追及が可能であるが，開発と製造にかかるコストや納品までにかかるターンアラウンドタイム (turn around time) が非常に大きくなってしまう．表 1.1 に，CPU

[1] シストリックアレイを応用したアルゴリズムはシストリックアルゴリズム (systoric algorithm) として多くの研究がなされている [11].

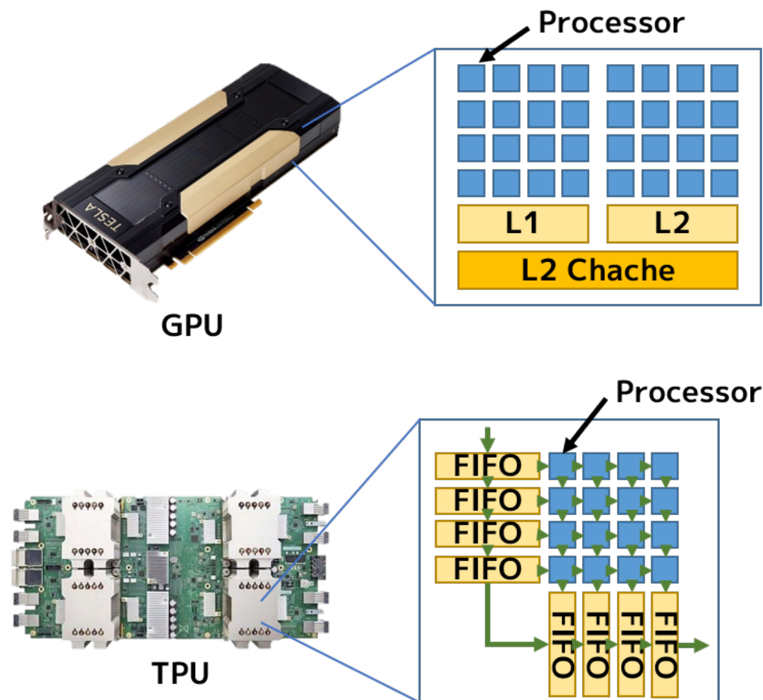


図 1.1: The architecture of tipical GPU and TPU

表 1.1: Comparison of standard processor (e.g. CPU, MCU), standard accelerator (e.g. DSP, GPU), and ASIC (custom circuit)

		Performance	Flexibility	Cost
Better	1	ASIC	Standard processor	Standard processor
	2	Standard accelerator	Standard accelerator	Standard accelerator
Worse	3	Standard processor	ASIC	ASIC

や MCU といった量産されている標準的なプロセッサ (表中 standard processor)，DSP や GPU のような標準的なアクセラレータ (表中 standard accelerator)，ASIC の特徴を比較形式でまとめる．ASIC は処理能力や電力性能について最も優れているものの，他のシステムに利用する柔軟さや開発コスト，価格といったコスト面では汎用品と比較して不利である．

1.1.2 FPGA を用いたアクセラレーション

カスタム回路を利用したいが量産しないためコストが見合わない，システムや技術の更新が頻繁でターンアラウンドタイムを長く取れない，といった場面においては，**再構成可能コンピューティング (reconfigurable computing)** が有効である [12]．**FPGA (field-programmable gate array)** に代表される，目的によってアーキテクチャの構成をカスタマイズできる**再構成可能デバイス (reconfigurable device)** を利用しシステムに応じた専用アクセラレータを開発する．FPGA も LSI の一種であるが，エンドユーザが設計したカスタムディジタル回路を書き込むことができるため，再構成可能デバイスと呼ばれている．FPGA は，同じ回路が動作する ASIC と比較する

と動作クロック周波数が低く性能が劣るという問題があるものの，製造コストやターンアラウンドタイムを大きく節約しながらも専用アクセラレータや専用プロセッサを構成することが可能であるといった利点がある．

今後 FPGA を用いた高性能計算 (High Performance Computing; HPC) の需要が増しさらなる設計開発コストの増大が見込まれており [13]，カスタム回路設計の生産性を向上させるための研究が盛んに行われ，様々なツールが開発されている．

1.2 ディジタル回路設計の現状と課題

1.2.1 ハードウェア記述言語を用いたディジタル回路設計

ASIC や FPGA で用いるディジタル回路の設計は，**ハードウェア記述言語 (hardware description language; HDL)** を用いたレジスタ転送レベル (Resister Transfer Level; RTL) による開発が必要となる．しかしながら HDL による設計はクロックサイクルレベルの低い抽象度の記述が必要であり，高性能なカスタム回路の設計が可能な反面，開発にかかるコストや time-to-market が大きくなってしまう問題がある．

記述の性質から HDL はバグが発生しやすく，デバッグにも様々なツールが必要で熟練した技術が必要である．対してソフトウェア開発はデバッグや解析手法が深く研究され知見が広く共有されていることが特徴であり，ハードウェア設計と比較して開発コストが低く生産性が高いといえる．加えて，ソフトウェア開発ではより多くのツールが利用可能であり，オープンソースソフトウェアやフリーウェアなど入手性の高いツールも数多い．このことから，ASIC の設計や FPGA を用いたシステム開発はコストが高く，電力性能や計算速度の利点にも関わらず利用できる場面は限られている．

1.2.2 ディジタル回路設計の生産性を高めるためのアプローチ

これらの問題を解決するため，大きく二つのアプローチが知られている．ひとつ目はメタプログラミングである．これを用いたツールは DSL (domain specific language) やマクロなどを用いて回路を記述し，HDL を生成する．これによって，HDL と比較してハードウェアの記述量を大きく削減することができ，記述の抽象度を向上させることができる．これらのツールは既存のプログラミング言語を用いた記述である場合もあるが，以前として回路設計の知識や技術を必要とするため，生産性を高めることが可能ではあっても設計のコストは依然として高いままである．

ふたつ目のアプローチは高位合成 (high-level synthesis; HLS) である．HLS は，C/C++や Java といったプログラミング言語によって記述されたプログラムやアルゴリズムと同じふるまいをする回路を合成する技術である．既に様々な言語による HLS ツールが開発されている．FPGA の大手ベンダである Xilinx や Intel は C 言語ベースのツールを公開しており，利用可能である．一般にこれらの HLS ツールは既存のソフトウェアの高速化を意図して設計されており，プログラムの一部を FPGA にオフロードする形で利用することが想定されている．しかしながら，エンドユーザはいくらかのアノテーションをプログラムに書き加える必要がある．これはソフトウェアの記述と回路記述のパラダイムが大きく異なり回路の合成に必要な情報が十分に含まれていないことが原因である．プログラミング言語はコントロールフロー型の記述であり，HDL はデータフロー型の記述である．これは二つの問題を引き起こす．第一に，ソフトウェアのプログラムには並列性の記述に関する情報が含まれていないため，並列プログラミングで用いるような記述が必要となる．第二に，データフローに関する情報が含まれていないため，自動的なパイプライン化が難し

く，最適化のための情報を付け加えなければスループットが非常に低くなってしまうことや，複雑で巨大なステートマシンが生成されてしまう問題がある．

PCIe などのインタフェースを介した拡張カードを利用する場合や単一のチップに搭載した SoC FPGA (system-on-chip) のように，CPU と FPGA が共存した計算機がしばしば用いられるため，ソフトウェアとハードウェアを同時に開発するコデザイン (co-design) 環境も広く用いられている．このようなコデザイン環境においても高位合成が利用されるが，コデザイン環境は CPU と FPGA の通信の記述コストを削減することが主な役割であり，ハードウェア部分の設計については高位合成の抱える問題が依然残されている．

システムの素早い開発のため, ソフトウェアの分野では Python や Ruby といった軽量言語 (light-weight language; LL) がしばしば用いられる．このような言語は動的型付けを応用した拡張性の高い型システムを用いることで開発効率と高い再利用性を実現していることが特徴として挙げられる．ハードウェア開発においては，これらの言語のように開発効率に主眼を置いたツールは研究が進んでおらず，発展の余地が残されているといえる．

1.3　研究目的と本稿の構成

このような背景から，本研究では，**ソフトウェア開発の分野で生産性を高めるために用いられているプログラミングモデルによってディジタル回路設計の生産性を高めることが可能か？**という問いに取り組んだ．FPGA を主なターゲットとして，ディジタル回路設計の生産性を高めるための技術についての研究を行った．本研究は大きく分けて 1) フレームワークを用いた最適なアーキテクチャの自動生成ツールの研究，2) ディジタル回路設計に近いプログラミングパラダイムを用いた回路設計ツールの研究の二つのアプローチに分けられる．

3 章では，既存のディジタル回路設計支援ツールに関する文献調査と本研究の立ち位置について議論する．4 章では，電力効率が重要である IoT/CPS (Internet-of-Things, cyber-phisical system) の分野のための開発環境の研究について説明する．エッジデバイス開発に適したアーキテクチャを構成するためのフレームワークを開発し，エッジデバイスに合った既存のツールや最適化手法を自動的に適用することで，回路設計の専門家でない開発者でも高いパフォーマンスを発揮するシステムを開発した．5 章では，ハードウェア設計言語と近いパラダイムを持つプログラミングモデルを用いて記述したプログラムからディジタル回路を合成するための手法についての研究について述べる．筆者は提案手法を応用しハードウェア・ソフトウェアコデザイン環境 Mulvery を開発し，その評価を行った．6 章では，HLS ツールにおいてサポートされないことがほとんどである関数ポインタの実現のための手法について説明する．関数ポインタをサポートすることによって高階関数などの抽象化のためのテクニックが利用できるようになる．C 言語ベースの高位合成ツールにおいてこれまで実現が困難とされていた関数ポインタを実現手法について説明し，関数ポインタが高位合成にもたらす効果や利益について議論する．続く 7 章では，FPGA を含むヘテロジニアス計算機環境において Mulvery を分散コンピューティングに応用する手法について述べる．そして最後の 8 章で研究全体の総括を行う．

第2章　FPGAの原理と回路設計の技術

本研究では，FPGA を主なターゲットとしたディジタル回路設計の技術に関する研究を行う．具体的な研究内容に触れる前に，FPGA を用いたシステム開発と技術の現状について深く理解するため，この章で FPGA の基本的な原理について説明する．また，本研究と密接に関連しているプログラミング言語を用いて回路を設計するツールの基本的な原理について説明する．

2.1　FPGA の技術の概要

2.1.1　FPGA の原理

FPGA の構造

図 2.1 に，現在の多くの FPGA が採用しているアイランドスタイル FPGA と呼ばれる種類の FPGA の概略図を示す [14]．FPGA は，大きく分けて

- 汎用ロジック (soft logic),
- 配線要素 (switch block, connection block),
- 入出力ブロック (I/O block),
- 専用ロジック (memory block and multiplier)

の四種類の要素で構成される．汎用ロジック (general logic, soft logic) は論理回路を構成するプログラマブルなブロックであり，配線要素はブロックを接続するためのプログラマブルなブロックである．入出力ブロック (I/O block) は外部への I/O を提供するブロックであり，これもプログラマブルである．そして専用ロジックは図中のメモリブロック (memory block) や乗算器 (multiplier) など専用に実装されたブロックである．メモリや乗算器は汎用ブロックの組み合わせでも実現可能であるが，多数のブロックを消費する上に非常に多く用いられるため専用のブロックを用意することで回路面積の節約と処理の高速化に貢献する．汎用ロジックと専用ロジックを対比させて，それぞれ soft logic, hard logic と呼ばれることもある [14].

論理要素と論理クラスタ

汎用ロジックのブロックは図 2.2 に示すような basic logic element (BLE) によって構成される．BLE は主に loop-up table (LUT), D-フリップフロップ (D-FF) で構成され，k ビットの入力に対して 1 ビットの値を出力する．図 2.2 は，6 入力 1 出力の BLE である．LUT は一般には k ビット入力 1 ビット出力のメモリテーブルである．このようなメモリは真理値表の代わりに用いることができ，k 入力のメモリは k 個の変数を持つ任意の論理関数（2^{2^k} 種類）を表現することができる．たとえば論理関数 $f(A,B,C) = (A\&B)|(A\&C)$ に対応する 3 入力 LUT のメモリテーブルは表 2.1

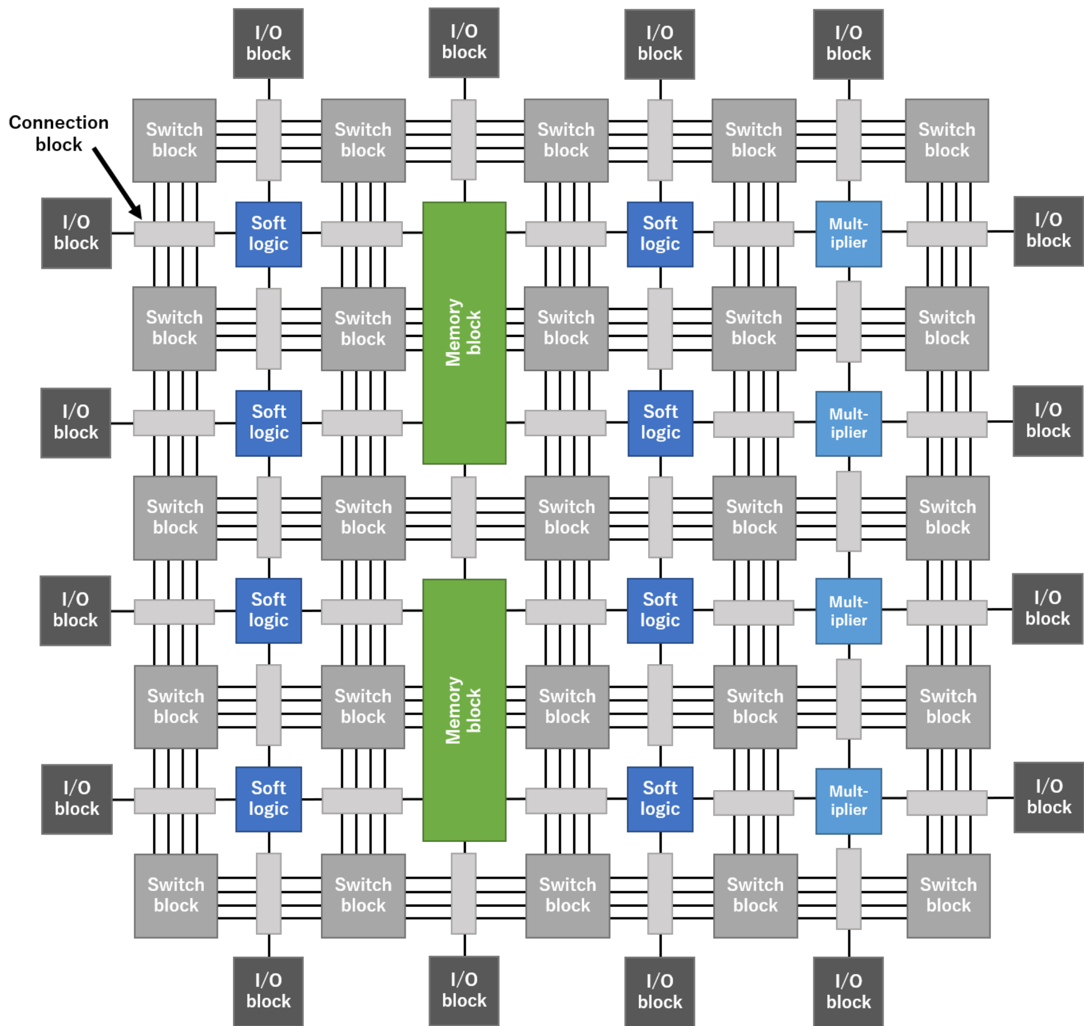


図 2.1: Basic island-style FPGA structure [14], [15]

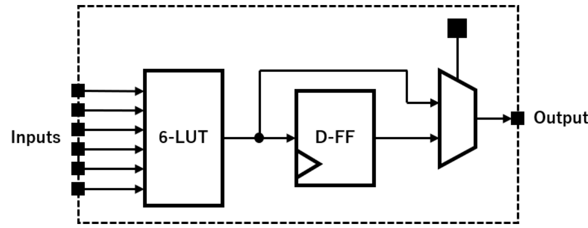


図 2.2: Basic logic element (BLE) [16]

表 2.1: The memory table for $f(A, B, C) = (A\&B)|(A\&C)$

Address	Data	Address	Data
000	0	100	0
001	0	101	0
010	0	110	1
011	1	111	1

に示すものとなる．変数の数が k よりも多い論理関数を表現する場合には複数の論理関数に分解して，複数の BLE を組み合わせて表現する．LUT の入力数 k が大きいほど複雑な論理関数（変数の多い論理関数）を表現できるため，ある論理回路を構成するために必要な BLE の数を減らすことができ，面積や伝送遅延の点で有利になる予測ができる．しかしながら必要になるメモリのサイズは 2^k と指数的に増加し結果的に必要な配線の量も増えるため，面積あたりに実装できる BLE の数が減少するというトレードオフがある．文献 [16] によれば，LUT の入力が 5,6 の場合に面積と遅延の面で良好なパフォーマンスが得られるとされ，実際に近年の FPGA の多くで 6 入力 LUT が採用されている．

入力信号は複数の BLE で共有できることが知られており，図 2.3 のように複数の BLE をまとめた **論理クラスタ (logic cluster)** を構成することで面積や速度の性能を改善することができる．文献 [16] では，論理ブロックの使用効率が最適になるようなパラメータ設定について，論理クラスタへの入力信号の数 I と LUT の入力信号数を k，論理クラスタに含まれる BLE の数を N として

$$I = \frac{k}{2}(N + 1) \tag{2.1}$$

と実験を通して定式化している．論理クラスタを用いない場合には $I = k \times N$ であるから，一方で LUT の場合と同様に入力数や BLE の数が増えるにつれ論理クラスタ内での伝送遅延が大きくなるトレードオフがあり，文献 [16] では N が 4〜10 かつ k が 4〜6 で面積あたりの遅延について最も良い結果が出ることを報告している．

2.1.2 FPGA の EDA 技術

RTL 記述から FPGA に書き込むことができるファイルを生成するためには electric design automation (EDA) ツールが必須であり，FPGA で実行する回路の性能にとって非常に重要な役割を果たす．FPGA に書き込むことができるファイルはビットストリームと呼ばれ，Verilog HDL や VHDL などの HDL による RTL 記述からビットストリームを生成するまでには，次のような処理を経る：

1. 論理合成 (logic synthesis)

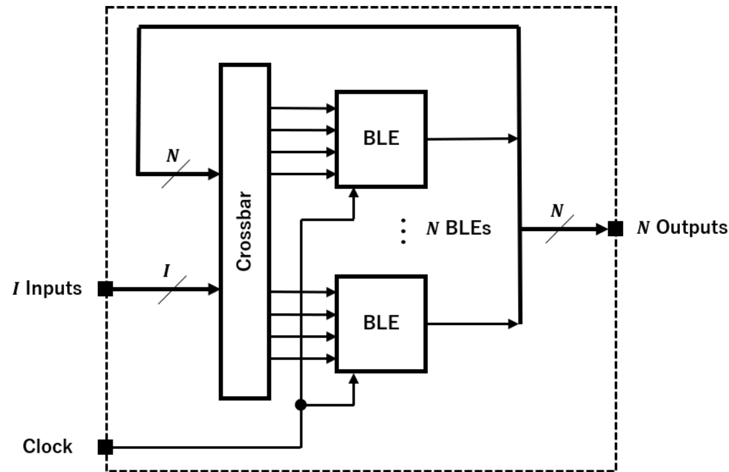


図 2.3: Logic cluster [16]

2. テクノロジーマッピング (technology mapping)
3. クラスタリング (clustering)
4. 配置 (place)
5. 配線 (route)
6. ビットストリーム生成 (generate bitstream)

論理合成 は，RTL 記述からゲートレベルの論理回路[1]に変換する工程である．この際に EDA ツールは論理回路の不要な配線の削除や最適化などを行っていく．

テクノロジーマッピング では，論理回路で使われているゲートの代わりに FPGA の論理セルを用いた回路に変換する工程である．例えば UCLA (University of California, Los Angeles) の研究グループが提案した FlowMap [17] ではブーリアンネットワークで表現された有向グラフ (DAG) において k-入力となるようにカットを加えていき，k-入力 LUT に対応させていく (mapping) といった流れで処理を行う．

クラスタリング では，LUT レベルの回路において LUT をクラスタリングし，論理クラスタをマッピングしていく工程である．論理クラスタ内の配線は論理クラスタ間の配線と比較して高速であるため，単に論理クラスタに多くの LUT を詰め込むのみならず，論理クラスタ間の接続をなるべく短く少ない接続に抑えることも高いパフォーマンスを発揮する回路を合成するにあたって重要な目標となる．

配置配線 では，論理クラスタレベルの回路において各論理クラスタを FPGA 上の物理的な位置に配置し，その配線を確定させていく作業である．実際の処理では配置と配線は別々の工程として行われるが，しばしば配置配線 (place and route) とひとまとまりにして呼ばれる．論理クラスタは 2 次元上に整列しているため，回路中の論理クラスタ $C_1, ...C_n$ と FPGA 上にある利用可能な論理クラスタ $T_1, ..., T_n$ に対して回路中の論理クラスタ C_i, C_k 間のバス幅が w_{ik}，利用可能な

[1]ゲート回路，ネットリストなどともよばれる

論理クラスタ T_j, T_l の間の距離が d_{jl} であるとしたとき，C_i を T_j に設置したときに 1，それ以外で 0 になる変数 x_{ij} によって，論理クラスタの配置は次の目的関数の最小化問題と考えることができる：

$$\sum_{i,j,k,l=1}^{n} w_{ik} d_{jl} x_{ij} x_{kl}. \tag{2.2}$$

これは二次割当問題 (quadratic assignmant problem; QAP) そのものであり，よって計算複雑性クラスが NP-hard に相当するような問題である．また配線では，グラフ上で最短経路探索を行い概略配線を行った後，複数のバスが同じ経路を使うことによる混線を避けるため競合ワイヤにコストをかけ，改めて経路探索を行う．これを競合がなくなるまで繰り返していき，詳細配線を決定する [15]．このように配置配線においては複数の計算量の大きな最適化問題を解く必要があり，回路規模が大きくなるにつれ指数的なオーダーで処理に時間を要するようになることがわかる．

ビットストリーム生成　では，FPGA 内の LUT やデータバススイッチの設定を生成し，FPGA に書き込むことができるファイルを生成する工程である．

2.1.3　参考文献

FPGA のデバイスに関する技術については，I. Kuon らによる *FPGA Architecture: Survey and Challanges* [14] に詳細にまとめられている．日本語で読むことのできる文献としては，天野らによる FPGA の原理と構成 [15] が非常に参考になった．

2.2　高位合成の原理

2.2.1　高位合成の概要と利点

ディジタル回路設計は一般的には HDL を用いて行われるが，近年では C や Java などの高級言語 (high-level language; HLL) から RTL 設計を合成する回路設計も行われるようになっている．このような設計は高位合成 (high-level synthesis; HLS) と呼ばれており，Xilinx や Intel などの大手ベンダからも HLS ツールが提供され実用的な段階に入ってきている．

RTL 設計は設計に時間がかかる上にミスの混入も発生しやすく，またコードの再利用性もあまり高くないために開発効率が非常に悪い．また，RTL 設計を検証するためにはシミュレーションを行う必要があるが，RTL シミュレーションは各レジスタの値などを全てシミュレーションしなければばならないため，非常に低速である．対して HLS を使う場合，RTL シミュレーションを行う前にプログラムをそのまま実行してシミュレーションの代わりとするビヘイビアシミュレーションが可能となり，シミュレーションに必要な時間を削減し検証のコストを大幅に低減することができる．また抽象度の高さや設計のしやすさなどから開発効率も格段に改善する．

2.2.2　動作合成

多くの HLS ツールでは C 言語や C 言語をベースとした独自の言語を用いているため，この節では入力を C 言語であるとして説明する．HLS ツールはまず，入力されたプログラムの制御フローグラフ (control flow graph; CFG) とデータフローグラフ (data flow grpah; DFG) を生成する．その後,

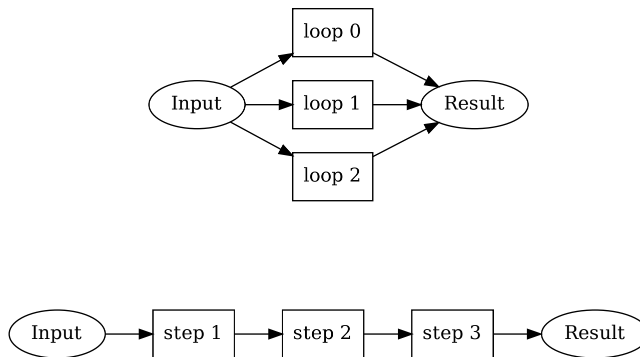


図 2.4: Optimization of for-statement by loop-unrolling (top) and pilepining (bottom)

- アロケーション (allocation)
- スケジューリング (scheduling)
- バインディング (binding)

の 3 つの工程を行う．

アロケーション　の工程では，プログラムが必要としているメモリや DSP などのコンポーネントを解析し確保する作業を行う．FPGA では BRAM と呼ばれるオンチップメモリが多数配置されているため，プログラム全体で共通のメモリを使うよりも，依存関係がない部分を細かく切り分けてそれぞれで使用する方が遅延やデータ幅の観点で有利であり，高速化に寄与する．しかしながら，データ間の依存関係のプログラムから最大限の最適化結果を得られるような解析を行うことは難しいため，人間の手によって指示を与え HLS ツールの推論を手助けする必要がある．

スケジューリング　では，CFG をもとにして処理の流れをステートマシンに変換していく．ディジタル回路ではすべての順序回路や組み合わせ回路のモジュールが並列に動作するため，CFG, DFG を用いて，実行順序やモジュール間のタイミングを合わせる処理が必要となる．この場合にも，C 言語（やその他のプログラミング言語）ではデータや制御の依存関係の表現が不十分な場合があり，人間の手で情報を付け加え最適化を促す必要がある．図 2.4 に for 文についてのスケジューリングの例を示した．配列に対し逐次処理を行うようなループを展開 (loop unrolling) することで配列の全ての要素に対してまとめて処理を実行することができるようになる場合がある．一方で，ループ展開するのではなく処理をパイプライン化することでメモリからデータを読み出し次第即座に処理を開始し，スループットを高める最適化も考えられる．単純なループではどちらが最適化判断できることもあるが，複雑なループやデータ構造である場合にはこれらの自動的な解決は難しく，ユーザが最適化を施す必要がある．パイプライン化などの並列化のテクニックについては次節でより詳細に説明する．FPGA 上の BRAM には 1 クロックでアクセス可能なバスの数に制限があるため，設計者はこのような点も踏まえてアロケーションとスケジューリングが意図通りに行われるようなプログラムを記述する必要がある．

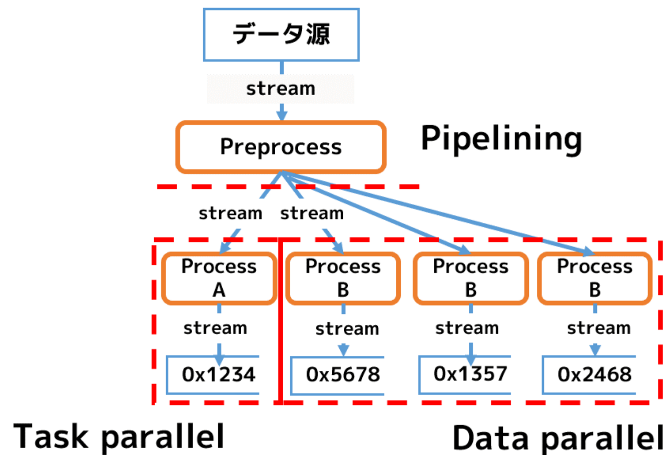


図 2.5: Three types of parallelization

バインディング　バインディングでは，アロケーションで確保した FPGA 上の資源を実際に割り当てていく作業である．より多くのメモリブロックや浮動小数点演算器を割り当てたパートは計算の並列度が高まり処理が高速になるが，すべてのパートでそのようなバインディングを行うとリソースが不足してしまう．このようなリソース割り当てにおいても C 言語の表現力のみでは解析が難しいため，やはり人手による指示やコードの最適化が必要である．

2.2.3　参考文献

数多くの高位合成ツールや回路設計ツールの比較が，M. W. Numan らによる *Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains* [18] にまとめられている．

2.3　高速化技法

この節では，ハードウェアアクセラレーションで用いられる技法の中でも，特に HLS を用いた回路設計において特に重要となる高速化技法について簡単に説明する．システムにデータを入力してから処理結果が出力されるまでの時間を**レイテンシ**，一定の時間中に入力することができるデータの量を「スループット」と呼ぶ．

2.3.1　並行計算

複数の処理を並行に実行する技法として，作業をいくつかのステップに分けそれぞれ別のリソースで計算する**パイプライン化**と，ある作業中で互いに依存関係のない部分を**並列化**するものの大きく二種類に分けて説明できる．

パイプライン化

図 2.5 中で Pipelining で示した分割は, 処理の流れを前後に大きく二分している．図中「Process」で示したブロックは，処理が完了するとデータは次の「Process A/B」のブロックに引き継ぐことで空いたリソースとなり，データ源から新たなデータを取り出して次のデータの処理を開始することができる．すなわち，データが処理されるまでのレイテンシは変化しないものの，計算リソースを効率的に利用し単位時間により多くのデータを投入することができるようになるため，スループットを向上させることができる．

処理の並列化

対して処理の並列化は，レイテンシを短くするための技法である．並列化はさらに二種類に分類することができる．

タスク並列化　たとえば `a = processA(in0, in1); b = processB(in0, in1);` という処理においては，`a` と `b` の計算の間に依存関係がないため，すなわちデータである `in0`，`in1` を受け取った時点で並列に処理を開始しても問題がない．このような並列化は**タスク並列化 (Task parallelization)** と呼ばれる．

データ並列化　`for (data in array) result += processB(data)` といったように配列からデータを逐次取り出して処理する例を考える．この場合，あるデータは他のデータとの依存関係がないため，関数 `process` を実行するリソースを複数用意すれば，その分並列に動作させることができる．つまり，配列の 0 番目から 99 番目の要素を担当する回路，100 番目から 199 番目を担当する回路・・・といったように複数のリソースで分担することで処理を高速化することができる．このような並列化は**データ並列化 (Task parallelization)** と呼ばれる．

2.4　まとめ

この章では，FPGA の原理と FPGA の回路設計で用いられている EDA 技術についての概要をまとめた．また，プログラムからハードウェアを合成する HLS の原理の概要も説明した．本研究では，特に HLS において課題とされている「人手による指示やコードの最適化」について取り組む．この章で説明した通り，人手による指示やコードの最適化はプログラムの中に回路合成に必要な情報が十分に含まれていないことが根本的な原因であり，何かしらの形で情報を与える以外に，今後のコンパイラ技術や HLS 技術の進展によって改善するのは難しい．本研究では，このような不十分な情報をいかに補うかという点に着目し，研究を進めていく．

第3章　関連研究

3.1　はじめに

3.1.1　ソフトウェア開発の技術の有効活用

ソフトウェア開発の世界では，生産性を高めるための開発手順やプログラミング手法などがよく研究されており，多くのカスタム回路設計ツールにおいてもハードウェア開発にこれらの手法を取り入れることを目指している．近年，システムに求められる真の要件を見出すため，リリースまでの期間の短縮とテストの繰り返しを重要視した開発手順やプログラミング手法が注目を集めている．

ソフトウェア開発の開発手順とハードウェア開発　生産性を高めるための開発手順の例として，アジャイルソフトウェア開発フレームワーク [19] が挙げられる．この開発手順では，要求に答えたシステムをできるだけ素早く開発し，評価する手法である．さらに開発と評価を継続的に繰り返すことでシステムの完成度を高めていく．この開発手順は，開発初期には全体像が明確化されず最終的な開発期間が延びる可能性がある反面，テストまでの期間が短く，ユーザの要求やシステム要件の見直しを何度も行うことが可能でありユーザの要求に正確に応えることができるメリットがある．

ASIC や VLSI の開発では莫大な製造コストがかかるため，このような手法は取り入れられてこなかった．しかしながら FPGA を用いたシステムではカスタム回路を何度も修正することができるため，このようなソフトウェア開発手法と同様の開発の技術を取り入れることができる．

ソフトウェア開発のプログラミングモデルとハードウェア開発　生産性を高めるプログラミング手法の例として，軽量言語 (lightweight language; LL) [20] を用いたプログラミングが挙げられる．動的型付けやラムダ抽象の採用など記述の抽象度を高めるものが多く，高い生産性から様々な場面で利用されている．IEEE の 2017 年の記事では，Python が最も人気の言語であるとされている [21]．Python は機械学習や深層学習のシーンで多く利用されており，ハードウェアアクセラレーションの対象としても重要なターゲットである．

FPGA 上ではアプリケーション実行中にバス幅や信号の経路を動的に変更するのは困難である．しかしながら軽量言語には動的プログラミング言語が多く，デジタル回路を合成する際に静的に型情報や呼び出される手続き（メソッド）が決定できないため，カスタム回路設計ツールに利用される例は少ない．

また，型システム [22] を用いた型検証によって記述の正当性を検証する研究も多く行われている．回路設計では，信号線の接続でバス幅が異なると意図しない動作が起こりバグの温床となるため，型安全な言語を使うことによって不正な信号線の接続を回避することができる．

Listing 3.1: Description example of a hardware that calculates GCD using Chisel [1]

```
1  class GCD extends Module {
2    val io = IO(new Bundle {
3      val a = Input(UInt(16.W))
4      val b = Input(UInt(16.W))
5      val e = Input(Bool())
6      val z = Output(UInt(16.W))
7      val v = Output(Bool())
8    })
9    val x = Reg(UInt())
10   val y = Reg(UInt())
11   when (x > y) { x := x − y }
12     .elsewhen (x <= y) { y := y − x }
13   when (io.e) { x := io.a; y := io.b }
14   io.z := x
15   io.v := y === 0.U
16 }
```

3.1.2 動向調査の意義

カスタム回路設計支援ツールは，様々な動機から設計がなされている．例えば，ハードウェア技術者が記述の手間を省くため，特定のアプリケーションに特化させデジタル回路設計の知識を不要にするため，などの理由がある．本研究では，デジタル回路設計の知識を持たないプログラマが FPGA を利用して高い処理能力や電力性能を実現するためのツールの研究・開発を行う．デジタル回路設計の知識を前提としないカスタム回路の設計かつ高い性能を発揮する回路の合成を目指すことを試みる．

そこで，これまでに発表されてきたカスタム回路設計ツールについて調査・分析し，本研究の立ち位置を明確にすることで，これまでの研究と全く異なる新しい試みであることを確かめることを目的とする．

3.2 カスタム回路設計ツールの種別

HDL 記述よりも高い生産性を実現するため設計されたカスタム回路設計ツールは三種別に分類することができる．この節では，代表的なツールを例に挙げながら，それぞれの種別について説明する．

3.2.1 ドメイン特化言語を用いたレジスタ転送レベル設計

第一は，HDL を効率よく記述するための**ドメイン特化型言語 (Domain Specific Language; DSL) を用いた RTL 設計ツール**である．以降，DSL ベースの RTL 設計ツールと呼ぶ．

Verilog HDL や VHDL の生産性の低さの一因は，記述の低い抽象度にある．アプリケーションに応じて入出力バスなどのインタフェースは変化するが，このような差分を言語仕様のレベルで吸収できず，都度 Wrapper を実装したりモジュールの再設計を行う必要がある．そこで，モジュールや入出力の抽象度を高める型システムやプログラミングパラダイムを導入した言語を用いて RTL 設計が可能となるツールが提案されている．

この種類の代表的なツールである Chisel[23] のコード例を Listing 3.1 に示す．Scala を用いて記述されているが，記述の内容は Verilog や VHDL と同様に RTL の記述であることがわかる．

Listing 3.2: Example of matrix product using Vivado HLS

```
1  #pragma HLS ARRAY_RESHAPE variable=a complete dim=2
2  #pragma HLS ARRAY_RESHAPE variable=b complete dim=1
3
4  for (int i = 0; i < MAT_A_ROWS; i++){
5    #pragma HLS PIPELINE
6    for (int j = 0; j < MAT_B_COLS; j++){
7      res[i][j] = 0;
8      #pragma HLS PIPELINE
9      for (int k = 0; k < MAT_B_ROWS; k++){
10       res[i][j] += a[i][k] * b[k][j];
11     }
12   }
13 }
```

3.2.2 汎用プログラミング言語や DSL を用いた動作レベル設計

第二は，C/C++言語や Java 言語などの**汎用プログラミング言語 (General Programming Language; GPL) や DSL を用いた動作レベル設計**である．このようなツールを実現する技術は高位合成 (high level synthesis; HLS) として知られまとめられている [24]．以降は HLS ツールと呼ぶ．

計算に用いられるアルゴリズムはしばしば手続きとして定義される．しかし RTL 設計ではデータフローモデル [25] で記述する必要があり，アルゴリズムが複雑であるほど大きな開発コストを要してしまう問題があった．そこで GPL で記述されたアルゴリズムを解釈し同じふるまいが実装されたデジタル回路を自動的に設計する HLS ツールが数多く提案され [26], [27]，商用利用も行われている [28]．

この種類の代表的なツールである Vivado HLS [23] のコード例を Listing 3.2 に示す．このコード例では，三重ループによる行列演算の記述に対して最適化が行われている．記述のパイプライン化のようなアーキテクチャの制御（5, 8 行目のプリプロセッサ命令）のほか，配列がメモリ上にどのように配置されるかの制御（1, 2 行目の命令）を行っている．

3.2.3 ドメイン特化言語を用いたシステムレベル設計

第三は **GUI や DSL を用いたシステムレベル設計**である．以降，システムレベル設計ツールと呼ぶ．DSL ベースの RTL 設計ツールとよく似ているが，より特定のドメインに特化させることで RTL 設計を必要としないものや，既に用意されたモジュールを組み合わせることで RTL 設計をせず簡単にカスタム回路を設計するためのツールを指す．

ハードウェアの関連する組込みシステムのモデルベース開発と相性がよく，MATLAB/Simlink [29], [30] において，HDL Coder としてハードウェア設計ツールが実現されている．Simulink での GUI 設計の例を図 3.1 に示す．このように GUI や DSL を用いてコンポーネントを組み合わせたデータから，FPGA や ASIC の回路として合成可能な HDL 記述を自動的に合成することができる．

3.3 既存の開発ツールの分類と特徴

この節では，これまでに発表されているカスタム回路設計ツール 18 種について，2 節で説明した三種別に従って分類し，それぞれの特徴についてまとめる．ツール名に付した年は，そのツー

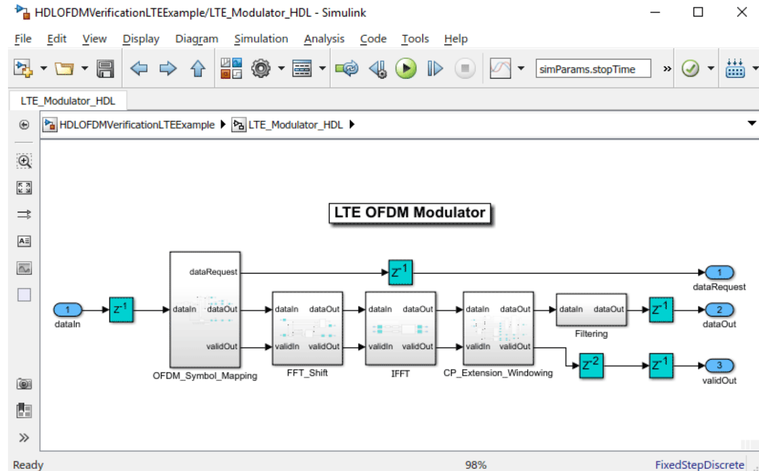


図 3.1: Example of system design using Simulink [30]

Listing 3.3: Calculating the average value of an event using Estrel [2]

```
module AVERAGE:
  # declare part
  input INCREMENT_AVERAGE(integer);
  output AVERAGE_VALUE(integer);

  # body
  ver TOTAL := 0 : integer, NUMBER := 0 : integer in # declare local variables
    every immediate INCREMENT_AVERAGE do
      TOTAL := TOTAL + ? INCREMENT_AVERAGE;
      NUMBER := NUMBER + 1;
      emit AVERAGE_VALUE(TOTAL / NUMBER) # emit signal to output port
    end
  end.
```

ルに関する最も古い論文もしくはツール自体が公開された年を示す．

3.3.1 DSL ベースの RTL 設計ツール

Esterel (1992) Esterel[2] は，リアクティブシステムを記述するために設計された平行プログラミング言語である．リアクティブシステムとは，入力イベントに対し，あらかじめ決められた手続きを適用して出力するようなシステムである．Estrel は，異なる入力イベントはコンカレントに処理されるという思想に基づいており，入力イベント毎に新しいサブプロセスが起動する．インタフェースの変数間の関係（同時に到着する/関係がない）を表す記法を言語仕様として持ち，コンパイラはより正確な決定性状態機械を構築し最適化を施すことができる．コード例を Listing 3.3 に示す．

始めはプログラミング言語として提案された Esterel であるが，後になって回路設計言語としても用いられるようになった [31]．オープンソースで公開されている The Columbia Esterel Compiler は，CPU で実行可能な C 言語のプログラムに加え，Verilog のコードもしくは BLIF 形式の回路情報を合成することができる [32]．

JHDL (1998) JHDL[33] は，Java 言語をベースとした再構成可能システム向けの HDL である．オブジェクト指向言語の抽象化機構を利用することが可能となり，コードの再利用性を高め

Listing 3.4: Up-down conter by JHDL [3]

```
1  public class mod4count2 extends Logic {
2    public static CellInterface[] cell_interface = {
3      in( "reset", 1 ),
4      in( "mod4", 1 ),
5      out( "total_count", "wid" ),
6      param( "wid", INTEGER )
7    };
8
9    public mod4count2(Node parent, int length, Wire reset,
10           Wire mod4, Wire total_count) {
11     super(parent);
12
13     int i;
14     int width = total_count.getWidth();
15
16     bind("wid", width);
17     connect("reset", reset);
18     connect("mod4", mod4);
19     connect("total_count", total_count);
20
21     Wire addem = or(mod4,reset);
22     // Count up if addem is asserted. count down otherwise.
23     new upDownCounter(this, reset, constant(width,length),
24             addem, vcc(), total_count);
25   }
26 }
```

ることができるようになる．Listing 3.4 に示したコード例のように，Verilog や VHDL のようにモジュール間の接続などを全て記述する RTL 設計に近い．JHDL は PRSocket と呼ばれるコンポーネント間のインタフェースを提供しており，このインタフェースを用いることで FPGA の部分再構成 (partial reconfiguration) 機能が利用できるようになる．

Verischemelog (1999) Verischemelog [34] は，Scheme ベースのカスタムデジタル回路設計ツールである．JHDL と同様のツールで，Verilog を抽象的に扱いマクロ言語を通して回路記述を生成することでコードの再利用性と生産性を高めることを目標としている．Scheme がインタラクティブに実行可能な点に注目し，インタラクティブ実行中にも都度エラーや警告を行い設計の不備を発見できるように設計されている．

SystemVerilog (2002) SystemVerilog[35], [36] は，Verilog では抽象的な記述が困難だったため，新たなデータ型やオブジェクト指向，アサーションなどを導入し記述の抽象度を高めた HDL である．

しかしながらこれらの新機能が回路へ合成可能か否かは合成系の実装次第であり，整数型や共用型など一部の機能を除いて多くの機能が合成できない可能性が高く，システム検証機能の強化にとどまる改善となった．

ForSyDe (2002) ForSyDe (formal system design) [37] は，関数型プログラミング言語の考え方をハードウェア設計に応用可能であるか検討したものである．Haslell ベースの回路設計フレームワークとして実装されることを想定している．[37] では，高階関数や多相型，遅延評価など関数型言語の特徴的な機能からデータフローモデルのハードウェア設計に変換する数理モデルを構築

Listing 3.5: Description example of 8-bits up counter using CλaSH [4]

```
1  upCounterLd
2    :: HiddenClockReset domain gated synchronous
3    => Signal domain (Bool,Bool,Unsigned 8)
4    −> Signal domain (Unsigned 8)
5  upCounterLd = mealy upCounterLdT 0
6
7  upCounterLdT s (ld,en,dIn) = (s',s)
8    where
9      s' | ld = dIn
10        | en = s + 1
11        | otherwise = s
```

し，ハードウェア設計の抽象化の方法を説明している．Mulvery やその他の関数型の記述を採用するツールがハードウェアを生成する手法を数理モデルとして厳密に検討した論文であるといえる．

Bluespec (2003)　Bluespec [38] はSystemVerologベースの独自言語であるBluespec SystemVerilog (BSV) を用いる回路設計ツールである．SystemVerilog の機能に加え，パラメトリック多相な function の定義や関数引数を使うことができるため，抽象度を向上させ再利用性の高い回路記述を行うことができるようになった．また型推論による静的型付けを行う言語であるため，例えば適切なバス幅を自動的に決定するなど記述の手間を省くことができる．

CλaSH (2009)　CλaSH [39] は，Haskell 言語ベースの関数型ハードウェア記述言語である．CλaSH 以前のツールでは，if やパターンマッチなどのプリミティブ関数を用いた処理は回路に変換することができず, プリミティブ以外の別の関数を用いる必要があった（case の代わりに multiplexer が用意されているなど）．CλaSH は新たに専用のコンパイラを用意することで，Haskell のプリミティブ関数を回路記述に用いることができるようになった．したがって，多相型（パラメトリック多相，アドホック多相）[22]，ユーザ定義の高階関数，パターンマッチなどの Haskell の特徴的な機能全てを記述に利用でき，このツールならではのアーキテクチャ探索手法なども提案されている [40]．また，ステートの遷移を表現するフレームワークを用いることで，クロックレベルの記述を実現することができるようになった．パターンマッチ文を用いてデコーダを記述しているコード例を Listing 3.5 に示す．

Haskell 言語は純粋関数型言語であるため，次のようなシンプルなルールで網羅的に回路への変換を行うことができる．

- 全ての関数はコンポーネントに変換される
- 全ての関数の引数は入力として変換される
- 関数の返り値は出力として返還される
- 関数適用はコンポーネントのインスタンシエーションとして変換される

Chisel (2012)　Chisel [23] は，Scala 言語ベースのハードウェア設計ツールである．

Chisel 以前から公開されていた JHDL, HML, Verischemelog はマクロによって HDL よりも高い生産性で設計が可能であるが，末端のコンポーネントは HDL で記述されることがあり，Java や scheme はハードウェアの型やセマンティクスとマッチしないため厄介な問題を導いていると指摘している．そこで，ハードウェア用のプリミティブ型（Byte 型や Bool, Float 型）を定義し，ハー

Listing 3.6: Calculation of $\log_2(N)$ using PyMTL's function-level [5]

```
1   # FL implementation for calculating log2(N)
2  @s.tick_fl
3  def fl_algorithm():
4    # put/get have blocking semantics
5    s.out.put( math.log( s.in.get(), 2 ) )
```

Listing 3.7: Calculation of $\log_2(N)$ using PyMTL's cycle-level

```
1   # CL implementation emulates
2   # a 3-cycle pipeline
3  s.pipe = Pipeline( latency = 3 )
4  @s.tick_cl
5  def cl_algo_pipelined():
6    if s.out_q.enq_ready():
7      if s.pipe.can_pop():
8        s.out_q.push( s.pipe.do_pop() )
9      else:
10       s.pipe.advance()
11
12   if not s.in_q.deq_ready():
13     s.pipe.do_push( math.log( s.in_q.deq(), 2 ) )
```

ドウェアモジュールを定義するためのベースクラス（Bundle や Component) を用いたプログラミングフレームワークを提供することで，HDL と GPL の間の型システムとセマンティクスのギャップに由来する問題を解決した．また，Verilog HDL の代わりに C++のコードを生成し，Verilog シミュレータで実装するよりも高速なシミュレーションを実現できる．Listing 3.1 に示したコード例のように HDL に近い低い抽象度の記述が可能であり，末端のコンポーネントの記述も全て Scala 言語のみで記述することができる．

著者らは，Chisel と Verilog HDL を用いて実装された 3-stages RISC-V を用いて Chisel の有効性を比較した．コード量は Verilog による記述の 1/3 となり，動作速度や回路面積はほとんど変化しなかった．また，シミュレーション時間は Verilog シミュレータを用いた場合よりも 7 倍高速となった．

著名な RISC-V のオープンソースソフトコア実装のひとつとして知られる，カリフォルニア大学バークレー校による RISC-V の実装にも利用されている [41], [42].

PyMTL (2014)　PyMTL[5], [43] は, Python ベースのカスタム回路設計ツールである．Function Level (FL), Clock Level (CL), Resister Transfer Level (RTL) の三段階の抽象度でハードウェアを設計でき，システムレベル設計ツールの一種であるとも言える．SystemVerilog のコードを Import する機能も提供しており，シミュレーションを行うこともできる．コード例を Listing 3.6 に示した．FL 記述は HLS に近いが，PyMTL においてハードウェア化される全てのオペレーションは変数 s に対する操作であり，HLS ではなく DSL によるシステムレベル設計である．

PyMTL は Mulvery と近い思想を持つツールであるため，次節で詳しく比較検討する．

3.3.2 HLS ツール

Warp Processor　Warp Processing[44] は，マイクロプロセッサ用にコンパイルされたバイナリを解析し，FPGA 用の回路を合成するツールである．バイナリを解析するため，コンパイル可

Listing 3.8: Calculation of $\log_2(N)$ using PyMTL's register-transfer-level

```
1  # Part of RTL implementation
2 s.N = Reg( Bits32 )
3 s.res = RegEn( Bits32 )
4 s.connect( s.res.out, s.out.msg )
5 ...
6 @s.combinational
7 def rtl_combN():
8   s.res.in_ = s.res.out + 1
9   s.N.in_ = s.N.out >> 1
10  if s.N.out == 0:
11    s.res.en = Bits1( 0 )
12  else:
13    s.res.en = Bits1( 1 )
```

能なプログラミング言語であればふるまい記述に利用することができ，これは他のツールと大きく異なる特徴であるといえる．

Warp Processor は一度マイクロプロセッサ上でプログラムを実行し，プロファイラによって FPGA へオフロードする部分を動的に検出する．オフロードされる部分は一度デコンパイルされ，改めて回路に合成される．回路が FPGA に書き込まれたらプログラムの対応する部分を FPGA に計算を要求するように書き換える．これらの処理は全てプログラムの実行中，動的に行われる．Warp Processor は常にプロファイルを続け，最速に実行できるようにアーキテクチャを調整し続ける．例えば FPGA によって高速化が実現できないプログラムは，最終的に FPGA にオフロードされる部分が無くなり，マイクロプロセッサ上でのみ処理される．

LegUp LegUp [45] は，C 言語ベースのハードウェア/ソフトウェアの協調設計環境である．プログラムを高位合成する際，ハードウェアに適した部分とソフトウェアに適した部分があるため，プログラム全体をハードウェア化するとパフォーマンスが悪化することがある．このため，ハードウェア化に適切な部分のみをカスタム回路に変換し，その他の部分は MIPS ベースのソフトプロセッサ[1]上で実行するカスタム回路を生成する．

プログラムを MIPS プロセッサで実行しプロファイリングすることで，自動的にハードウェアとソフトウェアに分割する．中間表現として，様々なプログラミング言語のバックエンドで用いられている LLVM IR [46], [47] を用いている特徴がある．様々な言語をフロントエンドとして用いることができる可能性に加え，LLVM IR はハードウェアにおけるオペレーションとよく対応している特徴があるため，高位合成のための中間言語として十分な性能を発揮できるとしている．

浮動小数点演算への対応 [48] や複数のクロックドメインへの対応 [49] など，2019 年現在も活発に開発が続けられているツールの一つである．

Darkroom (2014) Darkroom[50] は，画像処理のための DSL とコンパイラを提供する．画像処理では DRAM へのアクセス帯域を大きく消費するため，ラインバッファを用いアクセス帯域を小さく保つようにコンパイルが行われる．Darkroom DSL はデータフロー型のプログラミングモデルを用いるため，容易に有効非巡回グラフ (directed acyclic graph; DAG) を生成できる．Darkroom コンパイラは画像処理の記述から生成された DAG をシリアライズし，ASIC や FPGA のための RTL 記述を生成する．

[1]HDL などで記述された合成可能 (synthesizable) なプロセッサ．FPGA 上で動くプロセッサなどとして利用される．

Darkroom 以前から，C++のマクロを用いて実装された画像処理 DSL の Halide[51] が利用されている．Halide を用いた HDL ツールとして，アカデミックでは [52]，商用では日本の Fixstars 社による実装 [53] などが公開されている．Halide は CPU や GPU 向けに設計された DSL であるため，ウィンドウサイズや画像サイズを抽象的に定義できる．このような特徴を持つ場合，実行時までデータサイズが確定できない問題があるため，Darkroom DSL は画像やフィルタのウィンドウサイズなどが静的で固定なものとなるように定義されている．

Intel HLS, Vivado HLS　現在最も広く使われているといえる HLS ツールとして，FPGA 大手ベンダである Intel 社の Intel HLS [54] および Xilinx 社のの Vivado HLS [55] が挙げられる．両ツール共に C/C++をベースとした HLS ツールで，プリプロセッサ命令 (pragma) を用いて合成される回路のアーキテクチャを制御する．例えば Vivado HLS では，ループを展開し 1 ステートで計算するための命令 (unroll) や，FPGA 上のメモリを複数利用することでメモリアクセスの帯域を増やすための命令 (array_partition) などの命令が用意されている．C/C++の言語仕様では並列実行が考慮されていないため，ユーザはデータアクセスパターンの制御などを自身で管理する必要があり，並行プログラミング技法が必須の知識となっている．

3.3.3　システムレベル設計ツール

Ruby (1990)　1990 年に，Ruby という名の回路設計フレームワークが提案されている [56]．これは Mulvery が利用する Ruby[57] とは異なるものである．Ruby は 2 つの回路の関係性の表現方法を定義したもので，例えば R, S という回路があったとき $R; S$ は R から S にコネクションのある関係を表す．他の関係性の表現を組み合わせて数理論理学的に回路の正当性を検証を行うための技法をまとめたフレームワークである．シストリックアレイ [58] のような，多数の小さな計算要素の並ぶアーキテクチャの設計と検証などに利用されている [59]．

HML (1995)　HML [60], [61] は，ML 言語ベースのハードウェアモデリング言語である．ML 言語の高度な型システムを用いて型安全なプログラミングモデルを提供する．型推論を持つ言語であるため，ユーザは型を明示せずともインタフェースの接続を安全に行うことができる．ふるまい記述を行うこともできるが複雑な記述は困難である．モジュール間の接続を得意としているため，システムレベルの設計ツールとして分類した．2019 年現在，ツールは公開されておらず，利用することができない．

MATLAB/SimuLink の HDL Coder　HDL Coder [62], [63] は MATLAB/SimuLink [29], [30] のプログラムを元にカスタム回路を生成するツールである．信号処理や制御システムの分野では MATLAB/SimuLink がしばしば利用されている．これらのモデリングはデータフロー型のパラダイムで行われるため，回路を合成するツールとしても利用されている．特に SimuLink は様々な要素ブロックを組み合わせたブロック図を作成し，VLSI や DSP システムなどのシミュレーションを行うシステムレベル設計ツールであったため，FPGA 向けの回路の生成も自然な機能である．したがってハードウェアエンジニアが利用するため設計されたツールであり，ソフトウェアエンジニアが利用するためのものではない．

CAOS　CAOS (CAD as an Adaptive OpenPlatform Services)[64] は，FPGA アクセラレーションを用いた HPC システム向け開発ツール（ワークフロー）である．これまでに説明したツールと大きく異なり，マルチノードかつ，各ノードに複数の FPGA が搭載されていることを想定したシ

Listing 3.9: Example of kernel description using MaxCompiler

```
1  HWType flt = hwFloat(8,24);
2  HWVar x = io.input("x", flt );
3
4  HWVar x prev = stream.offset(x, − 1);
5  HWVar x next = stream.offset(x, +1);
6
7  HWVar cnt = control.
8      count.simpleCounter(32, N);
9  HWVar sel nl = cnt > 0;
10 HWVar sel nh = cnt < (− N1);
11 HWVar sel m = sel nl & sel nh;
12
13 HWVar prev = sel nl ? x prev : 0;
14 HWVar next = sel nh ? x next : 0;
15 HWVar divisor = sel m ? 3.0 : 2.0;
16
17 HWVar y = (prev+x+next)/divisor;
18 io.output("y", y, flt ) ;
```

Listing 3.10: Example of Manager on MaxCompiler [6]

```
1  Manager manager = new Manager("MAV", MAX2BoardModel.MAX24412C);
2  KernelParameters p = manager.makeKernelParameters();
3  Kernel k = new MovingAverageKernel(p);
4
5  manager.setKernel(k);
6  manager.setIO(
7      link ("x", DRAM(LINEAR)),
8      link ("y", DRAM(LINEAR))
9  ) ;
10 manager.build();
```

ステムをターゲットとしている．ノード間の接続を記述したシステムテンプレート，各ノードの持つ計算資源を記述したノードテンプレート，FPGA にデプロイされる計算カーネルを記述したアーキテクチャテンプレートの 3 種類の記述によって複雑な HPC システムの構成を記述する．アプリケーションは C/C++や OpenCL を用いて記述し，API を通して FPGA にデプロイされる計算カーネルを呼び出すことができる．CAOS のコンパイラはテンプレート・アプリケーションコード・計算カーネルを元にシステム全体のプロファイリングを行い，最適なスケジューリングとマルチノード環境へのマッピングを自動的に行う．ただし，計算カーネル自体は HDL で記述されている必要があるため，事前に他のツールを用いて設計しておく必要がある．

MaxCompiler　Maxeler 社の開発する商用ツールである MaxCompiler[6] はカスタム回路設計ツールである．命令シークエンスを記述する Kernel，カーネルと I/O を組み合わせ統合する Manager，FPGA 上のハードウェアを制御するソフトウェアの Host Application の 3 層に分けてシステムを記述する．Kernel と Host Application は MaxCompiler 用に作られた Java 言語ベースの DSL：MaxJ を用いて記述され，Host Application は C/C++や Fortrun によって記述される．

Kernel の記述の例と生成されるハードウェアを表現したグラフを Listing 3.9 および図 3.2 に示した．また，Manager の記述の例を Listing 3.10 に示した．Kernel の設計の粒度は非常に細かく，この部分は RTL 設計に近いといえる（ただしクロックとの同期を記述しないため，RTL 設計とは異なる）．

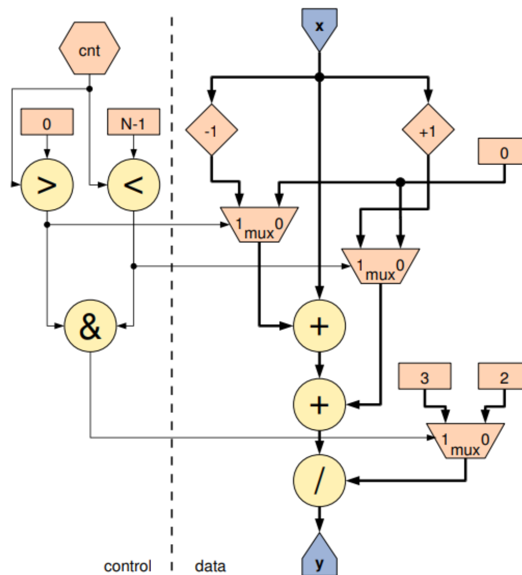


図 3.2: The graph of the synthesized hardware [6]

3.4 本研究の位置付け

図 3.3 に，先行研究・ツールと比較した本研究の位置付けを示す．記述の指向を「ハードウェア設計的」「プログラミング的」「特定用途向け」の三種類に分類した．ハードウェア設計的なツールは HDL の改善を念頭においたツールであり，DSL のツールがほとんどである．プログラミング的なツールは HLS ツールを指しており，C や Java などのプログラミング言語を使って，動作記述を行うものである．特定用途向けのツールはある一定の領域に絞り込んだツールを指し，専用の GUI ツールやフレームワークによって簡易に動作を記述することができるツールである．

これまでの多くの研究やツールは，ソフトウェア開発を行うプログラマ向けのツールではなく，ハードウェア設計者の負担を軽減するためのハードウェア設計者のためのツールとして開発されている．特定用途向けに事前にチューニングを施されたツールを除けば，プログラミング的なツールに分類されたツールであっても，ユーザによってコードや生成物のチューニングが施されることが前提となっている．したがってハードウェアの知識の少ないユーザが実用的な回路を設計するためには，非ノイマン型計算機アーキテクチャへの理解やツール毎の最適化戦略への理解など，プログラミング以外の部分への多大な学習コストを要する．

本研究では，プログラミング的かつ非特定用途向け（つまり様々な領域に適用可能）であるツールでありながらも，ハードウェアの知識に基づくプログラムの設計や生成物の最適化を最低限に低減することを目標とする．本研究は，現在広く用いられている GPGPU のように，FPGA が一般的に広く利用される汎用アクセラレータとして活用されるための重要な技術研究となることを目指す．

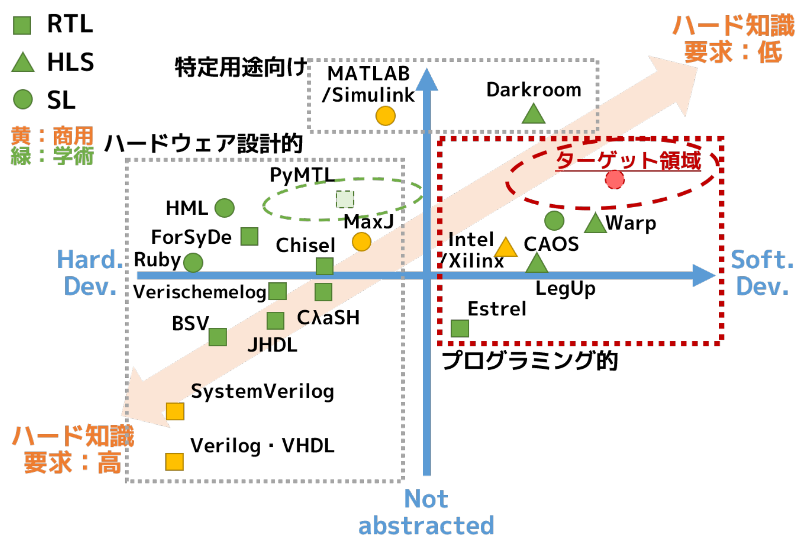


図 3.3: Comparison of tools

第4章　フレームワークを用いたアーキテクチャレベルの最適化

4.1　複数のツールを組み合わせた IoT/CPS エッジデバイス向けフレームワーク

IoT(Internet of Things) が注目を浴び，センサと通信機能を持つ様々なデバイスが開発されている．このようなデバイスでは，センサ入力やデータの出力のための機能を備えた LSI チップも数多く製品化されており，センサデータの種類や量の増加，分解能の向上などに伴いより高度なアルゴリズムを用いたデータ加工によるアプリケーションの実現が期待される．しかしながら，データ入力やデータ加工の処理コストの問題が生じてくる．

一方，目的とする処理をハードウェア回路として並列に実行する FPGA を用いたアクセラレータが，処理能力や電力効率の面で注目されている．スマート家電や自動運転など様々な分野で高い分解能のセンサを多数搭載するシステムが登場し，今後も増加していくことが予想される．このようなシステムにおいて FPGA を利用することは，大まかに分けて 2 種類の利点がある．ひとつは，プロトタイピングのために FPGA を活用するものである．回路を繰り返しアップデート可能な FPGA を用いてコンセプトの実証などを行い，十分大量に生産することが期待できる最終的な製品では ASIC 化するなどしてコストや電力性能を改善する開発プロセスとなる．もう一方は，最終的な製品にも FPGA を利用する場合である．カメラやテレビにおける画像処理のように，処理の高速性が求められかつ顧客毎や技術の変化に応じて処理内容が変化しうるような場合に FPGA を搭載し様々な状況に対応できるように設計されたデバイスが存在する．またネットワーク機器の分野では物理層をセンサとアクチュエータのように捉えることができ，ファームウェア更新をするようにパケット処理回路を更新することが可能となる．しかしながら，一般に FPGA によるアクセラレータの設計は Verilog HDL や VHDL といったハードウェア記述言語を用いてレジスタ転送レベルで行うため，ソフトウェアベースのシステムと比べて開発のコストが大きくなり，また高級言語によって記述された既存のソフトウェアを RTL 記述に変換する場合も多大なコストを要する．

近年では，C 言語をベースとした Vivado HLS が無償で利用でき，さらに JavaRock[65] や JavaRock-Thrash[66][67]，Synthesijer[68] といった Java 言語をベースとした高位合成ツールが開発されている．これらのツールの登場によって，FPGA 回路設計のコストの低減と，これまでのソフトウェア資産の再利用が期待される．しかしながら，センサインタフェースや DRAM といった外部モジュールとの通信ではタイミングが重視され，クロックを意識しない高位合成ではタイミング調整が困難であるという問題があった．

この章では，複数の高速なデータ入力とインターネットへのデータ出力が必要となる組み込みシステムのプロトタイピングのためのフレームワーク PyJer をの研究について説明する．複数の高位合成ツールと RTL 設計を組み合わせることでソフトウェアベースの開発に近いシステム開発の実現を目指した．

なお，本章で説明する PyJer は Github にて公開されている [69].

4.1.1　関連研究

高位合成ツールにおける問題点

高位合成ツールを用い，FPGA によるアクセラレータを実装する研究が数多く存在する．その中で，Vivado HLS を用いて開発した行列積アクセラレータと PyCoRAM および HDL で開発した行列積アクセラレータの性能比較がある [70]．ここでは，行列積での計算よりも FPGA-CPU 間の I/O にかかる比重が大きく，Vivado HLS のみによるアクセラレータよりも NEON 命令を用いて CPU 上のみで行う処理のほうが高速であると報告している．PyCoRAM はメモリアクセスチューニングに特化したツールであるために，計算カーネルの設計は HDL で行う必要があった．PyCoRAM および HDL を用いた開発では，PyCoRAM によってメモリチューニングは達成できるが，PyCoRAM で用いる計算カーネルの HDL による記述について，抽象度の低さに起因する開発の敷居の高さが指摘されている．

センサ入力を必要とするシステムの構築

これまでに，マイコンとセンサを用いたシステムを FPGA 上に移植する研究が行われている．[71] では，マイクロコントローラ上に実装された倒立振子制御のシステムを Java 言語ベース高位合成ツール JavaRock を用いて FPGA 上に専用ハードウェアとして移植を行い，その性能評価を行った．ここでは，センサデータの取得に多くの時間が必要であることが指摘され，センサ値取得とセンサデータの処理が同時に実行できる FPGA によって高速化が可能であると報告している．本研究では IoT に向けたデバイスを意識し，システム全体の FPGA 移植ではなく，容易なインターネットアクセスの実現のため CPU との連携も考慮する．

また，Electronic System Level(ESL) synthesis を用いた高速で低消費なワイヤレスセンサシステムの実装を行っている研究もある [72]．ASIC での実装の前にモデルベースの記述と FPGA によってプロトタイピングとチューニングを行うことで短期間の開発と高い性能の実現が両立できたことを報告している．

4.1.2　PyJer の概要

PyJer は，センサ入力が必要となる組み込みシステムの高速なプロトタイピングを行うための開発フレームワークである．この章ではその具体的特徴について説明する．

目的

センサ入力が必要となる組み込みシステムにおいて，センサの出力が高速であったり入力するセンサが多い場合，一般的な汎用マイクロコントローラではセンサのハンドリングが困難になると考えられる．このため，センサ入力はデータ処理とは並列に行えるシステムが構築できる必要がある．PyJer ではこれを FPGA にオフロードし，大量のデータの処理が必要なシステムのプロトタイピングを効率よく行える環境を実現する．また，サーバなどの外部システムとの通信については，TCP/IP などこれまでのソフトウェア資源を用いた開発が有効であると考えられる．

これらのことから，PyJer の基本的方針として，FPGA ではセンサ入力やデータ処理の一部を，CPU では通信や複雑な処理など既存のソフトウェア資源が活きる部分を受け持つものとした．

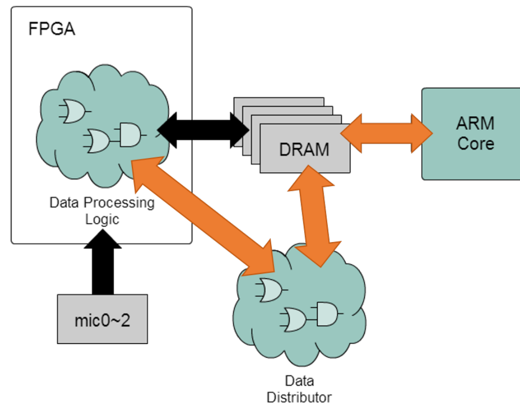


図 4.1: Structure where data flow is difficult to be modified

特徴

インターネットアクセスと SoC の活用：
IoT を意識したデバイスを開発する場合，デバイスのインターネットを通した通信が重要となる．FPGA のみでシステムを構築する場合，インターネットへの接続の実現には大きなコストを要する．そこで PyJer は SoC をターゲットとし，OS が動作する CPU と FPGA の連携を実現する．これによってインターネットを通した通信に関する多くのソフトウェア資源を用いることが可能となり，効率のよいプロトタイピングを行うことができる．
フレキシビリティの高いアーキテクチャ：
データ出力機構を CPU を通したものでなく別の機構に変更する場合，設計次第では変更の影響の範囲が広くなりシステムの変更が複雑なものとなってしまう（図 4.1）．そこで PyJer では CoRAM アーキテクチャを導入し，ユーザはその実装を意識することなく高いフレキシビリティを得ることができる．これはロジックの実装からデータフローの定義を分離して実装を行うことで実現される．
抽象度の高い記述による設計：
高速なプロトタイピングのために，開発全体を通して抽象度の高い設計を行うことができる．これまでの Vivado HLS[55] のような汎用の高位合成ツールのみではハードウェアの詳細なチューニングが行いにくく，PyCoRAM[73] のような詳細なチューニングを支援するようなツールは HDL による設計のチューニングを支援することを想定したものであった．PyJer では，設計の段階に応じて複数のツールを使い分けることで，ツールごとの導入して得られるメリットそれぞれを活用することができる．またビルド全体の自動化によって，複数のツールの使用によって発生する開発の煩雑さを解消した．

開発の手順

PyJer では，FPGA チップベンダの提供する回路合成・配置配線ツールを用いて行うべき作業も含めてビルド作業の多くを自動化した．これによってプロトタイピングがなるべく早い開発サイクルで行える開発環境を目指す．具体的には，以下の手順で開発を行う．

1. Verilog HDL を用いたセンサとのインタフェースの記述

表 4.1: Software-based development environment for mic data processing

CPU	ARM Cortex-A9 667MHz
DRAM	DDR3-1066, 533MHz, 32bit wide, 512MB
OS	Linux 4.0.0-gd94f3f3
実装に用いた言語	C (g++ 4.6.3)

2. Java を用いデータの処理機構の記述
3. Java を用いたサブモジュール間の接続の定義
4. Python を用いたデータフローの定義
5. Vivado TCL を用いた外部との接続の定義
6. make コマンドによるビルド

4.1.3 PyJer の設計方針

PyJer ではセンサ入力処理を FPGA にオフロードし並列化し，効率化を図る．ここでは，センサ入力を必要とする組み込みシステムにおける重要事項を考察し，PyJer の設計方針について説明する．

センサ入出力のハードウェアオフロード

予備実験として，センサ入力処理のソフトウェアのみによる実装と評価を行った．この予備実験においてはフレームワークのターゲットとしている Zynq-7000 においてプログラマブルロジック (PL) 部を使用せず，プロセッシングシステム (PS) 部のみで実装を行う形式とした．ソフトウェア開発環境を表 4.1 に示す．予備実験では，シリコンディジタルマイク SPM0405HD4H を使用した．このマイクは PDM 波形を出力し，本研究ではその出力周波数を 2MHz に設定した．

本研究において，PDM 波形をアナログ値に変換する手順は，信号を 512[bit] のリングバッファにバッファリングし，信号 1[bit] の到達ごとにその総和を求めるものとした．より具体的な処理の手順を図 4.2 に示す．この処理を C 言語で実装し，処理に要する時間を OS の提供する clock 関数を用いて計測した．ただし clock 関数の計測精度は 10[ms] のため，処理を 10^9 回繰り返しその平均値を処理に要する時間とした．また，センサ入力の代用として，変数の読み出しを行っている．

計測に使用したプログラムを図 4.3 に示す．このプログラムによって得られた結果は図 4.4 の通りとなった．処理１回あたりに要した時間は約 0.0511[μs]，すなわち１秒あたりの処理レートは 19.6[MHz] であった．同じマイクを複数接続することを考えた場合，９つまで接続が可能である．

CoRAM アーキテクチャの導入

フレキシビリティの高いアーキテクチャを実現するため，PyJer の生成する IP コアは CoRAM アーキテクチャ [74] の形式をとる．CoRAM アーキテクチャの形式をとる場合，図 5.29 に示すように IP 内部の各モジュールはそれぞれ CoRAM と呼ばれる RAM および Control Thread と接続す

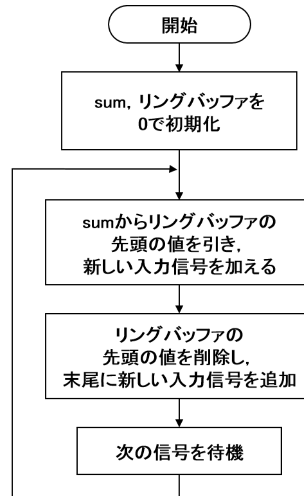


図 4.2: Flowchart of PDF signal DA conversion

```
#define BUFF_SIZE (512)
#define BIAS_LEVEL (256)
#define NUM_ITERATION (1000000000)

int main(void)
{
...
  start = clock();
  for(i = 0; i < NUM_ITERATION; i++){
    buff_tail = (buff_tail + 1) % BUFF_SIZE;

    sum = sum - buff[buff_tail] + input;
    output = sum - BIAS_LEVEL;
    buff[buff_tail] = input;
  }
  end = clock();
...
}
```

図 4.3: The program of time measurement of mic data processing

る FIFO を両方向に持つ．Control Thread と呼ばれるデータフローを制御するモジュールによって CoRAM 間のデータ転送が行われる．このアーキテクチャを用いることで，データフローの変更にロバストなシステムが構築できる [74]．図 4.6 のようにデータ配信機構を SoC の CPU 部から FPGA 内のロジックに変更するような場合，Control Thread によるデータの転送先を変更するのみでよい．

Vivado による合成・配置配線の自動化

開発サイクルを速めるため，Vivado TCL を用い合成・配置配線の自動化を実現する．Vivado TCL は，Xilinx 社が提供する統合開発環境 Vivado(Vivado IDE) をバッチ制御するために利用する言語である．Vivado TCL を用いてあらかじめメタにプロジェクトの設定を記述しておくことが可能である．これを用いて合成・配置配線の制御を行うのは煩雑なものであるため，プロトタイピングのために自動化ツールを書くのはコストが大きい．Vivado IDE の生成するプロジェクト

```
Processing time: 51050000[us]
Average processing time: 0.051050[us]
```

図 4.4: Measured processing time of mic data processing

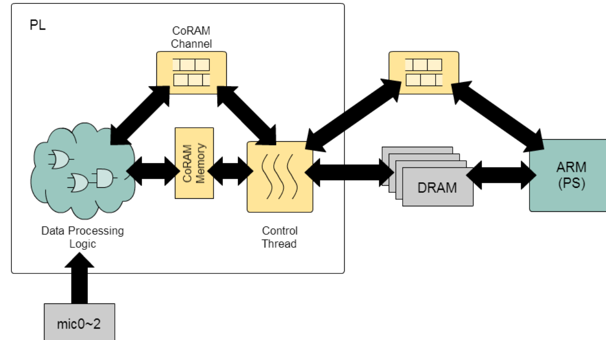


図 4.5: Hardware architecture with CoRAM

フォルダを用いずに開発することで自動的に生成されるファイルの除外の手間が省け，Git などのバージョン管理システムによるプロジェクトの管理を容易に行うことができるメリットも得られるため，構築したフレームワークにおいてこの機能を実現する．

4.1.4 PyJer が実現するシステムのアーキテクチャとデータフロー

アーキテクチャ

PyJer によって実現されるシステムのアーキテクチャを図 4.18 に示す．従来の設計では複数のセンサが 1 つの CPU に接続される形であった．対して PyJer を用いたシステムのアーキテクチャでは，センサ入力に必要な処理はハードウェアにオフロードされ，並列化される．また，データ処理機構の一部もハードウェアにオフロードすることができる．この章では，PyJer の生成するアーキテクチャに含まれる重要なコンポーネントについて説明する．

Sensor Interfaces

センサとの通信と，センサデータのデコード，センサへの命令のエンコードなどを受け持つ．クロックレベルのタイミング制御が必要なため，Verilog HDL で記述されることを想定している．

Computing Logic

各センサから収集したデータの処理を行う．また，Memory Access Controller に指示を送り，CPU-FPGA 間のデータ転送のタイミングの制御なども行う．Java 言語によって記述され，Synthesijer を用いて高位合成される．

Memory Access Controller

FPGA 内の CoRAM と DRAM の間のデータ転送を行う，Control Thread のこと．Python 言語によって転送タイミングの設定などを行ない，PyCoRAM によって生成される．

Computing Program

DRAM を通して PL 部からセンサデータを受け取り，ソフトウェア的に行ないたいデータ処理や，外部システムに出力する前のデータ整形などを行う．C/C++言語で記述されることを想定している．

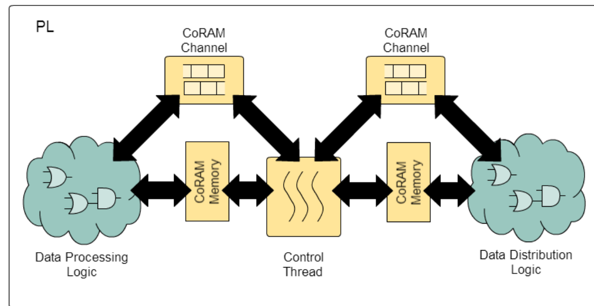


図 4.6: Arcitecture when data distribution is performed by a module in PL

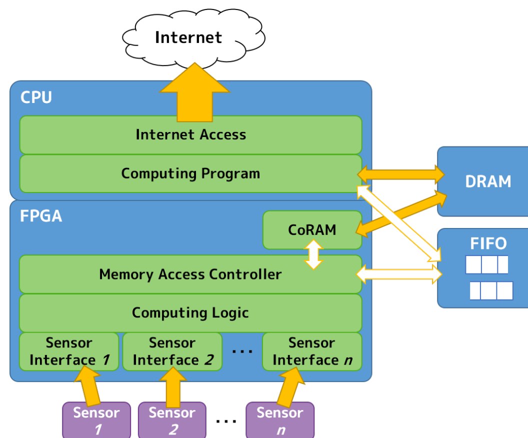


図 4.7: System architectrue that PyJer generates

Internet Access

外部システムとの通信を受け持つ．既存のソフトウェア資源を用いた Ethernet や Bluetooth 等による外部システムとの接続を想定している．

データフロー

4.1.4 にて説明したアーキテクチャ内でのデータの流れについて説明する．

センサデータの取り込み

各 Senser Interface はそれぞれ独立してセンサとの通信を行ない，センサデータの取り込みを行う．例えば PDM 信号であれば，Sensor Interface はバッファリングやフィルタリングを行ない，Computing Logic からの要求に応じて即座にデコード済みの最新のセンサデータを出力することができる．

FPGA から CPU へのデータの受け渡し

Computing Logic は，処理が完了したデータを CoRAM へ蓄積する．Computing Logic は任意のタイミングにて，Memory Access Controller に対して CoRAM のデータを，CPU が自由にアクセスできる DRAM の領域へと転送させる命令を送る．

CPU-FPGA 間の同期

CPU-FPGA 間には，PyCoRAM による Memory Access Controller を通した FIFO が用意され

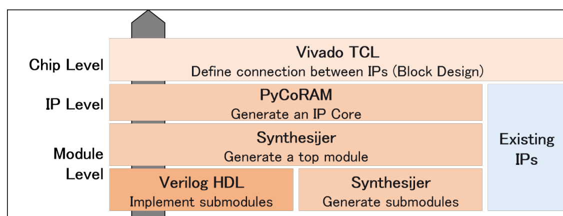


図 4.8: Tool hierarchy

```
@synthesijerhdl
public class DataProcessor{
...
    @auto
    public void controller()
    {
        // データ処理
```

図 4.9: Example description of data processing mechanism using Synthesijer

ている．このため Computing Logic は CPU との簡単なデータのやりとりや，同期的に CoRAM のデータの転送を行うことができる．

4.1.5　PyJer の使用法とその実装

この章では，PyJer の使い方とその実装について述べる．使用したツールの階層構造を示す図を図 4.8 に示し，最下層から詳細を説明していく．

PyJer を用いたシステム設計の手順

データ入力機構，データ処理機構の設計：

データ入力機構およびデータ処理機構について，それぞれサブモジュール単位で設計を行う．また，サブモジュールの接続の定義を行い，トップモジュールを作成する．この段階でデータの入力・処理・CoRAM への出力までの記述を行う．

設計には基本的に Java 言語を使用し (図 4.9)，Java 言語ベース高位合成ツールである Synthesijer によって Verilog HDL へとコンパイルが行われる．センサとの通信などクロックサイクルレベルの設計が必要な部分に関しては Verilog HDL を直接用い設計を行う（図 4.10）．実装・生成されたサブモジュール群は Synthesijer の機能を用いてひとつのモジュールとして結合される（図 4.11）．

Java コードからのセンサデータへのアクセスのサンプルコードを図 4.1 に示す．Sensor Interface と Computing Logic の接続の定義は，Synthesijer の利用方法に従い Top モジュールと呼ばれるコード内で行う．正しく接続の定義を行えば，Computing Logic 側ではクラス変数がクラス外部から書き換えられるような形でセンサデータが入力される．すなわち，Top モジュールにて接続先に設定した変数にアクセスすることでセンサデータを得ることができる．Top モジュールの Computing Logic の I/O の定義にて，`<name>_in`, `<name>_we` というポートを用意し we を常に HIGH にすることで，in に書き込んだ値を Computing Logic の Java コード内のクラス変数`<name>`に反映させることができる．

Listing 4.1: Sensor data acquisition in Synthesijer program

```
1  // −−−モジュール Top−−−
2  public class SampleTop {
3      public static void main(String... args){
4          ...
5          // Computing の LogicI/の定義 O
6          HDLPort core_sensor_data_0_in = sampleCore.newPort( \
7            "i_sensor0_data_in", HDLPort.DIR.IN, \
8            HDLPrimitiveType.genVectorType(32));
9          HDLPort core_sensor_data_0_we = sampleCore.newPort( \
10           "i_sensor0_data_we", HDLPort.DIR.IN, \
11           HDLPrimitiveType.genBitType());
12         ...
13
14         // センサインタフェースの I/の定義 O
15         HDLModule sensorInterface = \
16           new HDLModule("sensor_interface_sample", "clk", "rst");
17         HDLPort sensor_data = sensorInterface.newPort("data", \
18           HDLPort.DIR.OUT, HDLPrimitiveType.genVectorType(32));
19         ...
20
21         // センサインタフェースののインスタンス生成
22         HDLInstance instanceSensorInterface0 = \
23           sampleTop.newModuleInstance(sensorInterface, \
24           "sensorInterface0");
25         HDLInstance instanceSensorInterface1 = \
26           sampleTop.newModuleInstance(sensorInterface, \
27           "sensorInterface1");
28
29         // 接続の定義
30         instanceSample.getSignalForPort( \
31           sample_sensor_data_0_in.getName()).setAssign(null, \
32           instanceSensorInterface0.\
33           getSignalForPort(sensor_data.getName()));
34         instanceSample.getSignalForPort( \
35           sample_sensor_data_0_we.getName()).setAssign(null, \
36           HDLPreDefinedConstant.HIGH);
37         ...
38     }
39 }
40
41 // −−−Computing Logic−−−
42 @auto
43 public void sampling_mic_data(){
44     while (true){
45         for (int i = 0; i < BUFSIZE_MIC_DATA; i++){
46             buffer_sensor0_data[i] = i_sensor0_data;
47             buffer_sensor1_data[i] = i_sensor1_data;
48             wait_sampling_period();
49         }
50     }
51 }
```

Listing 4.2: Example of data transfer to DRAM

```
1  private void send_data_to_dram()
2      // データを転送する
3      for (int i = 0; i < 4096; i++){
4          o_mem_addr = i;
5          o_mem_d = result_data[i];
6          o_mem_we = true;
7          for (int j = 0; j < 1; j++);
8          o_mem_we = false;
9      }
10
11     // Memory Access Controller へトリガ送信
12     o_comm_d = 0; // unused value
13     o_comm_enq = true;
14     for (int j = 0; j < 1; j++);
15     o_comm_enq = false;
16
17     for (int j = 0; j < 10; j++);
18
19     // 受信完了通知を待つ
20     while (i_comm_empty);
21     while (!i_comm_empty){
22         o_comm_deq = true;
23         for (int j = 0; j < 1; j++);
24         o_comm_deq = false;
25     }
26 }
```

Listing 4.3: Example of Memory Access Controller

```
1  iochannel = CoramIoChannel(idx=0, datawidth=32) # ARM side
2  ram = CoramMemory(idx=0, datawidth=DSIZE * 8, \
3    size=RAMSIZE, length=1, scattergather=False)
4  channel = CoramChannel(idx=0, datawidth=32) # FPGA side
5
6  def body():
7      unused_req_1 = channel.read()
8
9      # Translate dataset from BlockRAM to DRAM
10     ram.read_nonblocking(DAT_SRC+(0*SPSIZE), \
11   DAT_DST+(0*ADDR_WIDTH), SPSIZE)
12     ram.read_nonblocking(DAT_SRC+(1*SPSIZE), \
13   DAT_DST+(1*ADDR_WIDTH), SPSIZE)
14     ram.wait()
15
16     channel.write(1)
17
18 while True:
19     body()
```

```
module SensorInterface
  (
    input clk,
    input rst,
    ...
  );
  ...
  always @(posedge clk) begin
    // データ入力の処理
```

図 4.10: Example description of sensor data processing using Verilog HDL

```
@synthesijerhdl
public class DataProcessorTop{
  public static void main(String... args)
  {
    HDLModule dataProcessorTop \\
      = new HDLModule("HataProcessorTop", "clk", "reset");
    // 上位層への I/F の定義
    ...
    HDLModule dataProcessor \\
      = new HDLModule("DataProcessor", "clk", "reset");
    // DataProcessor の I/F の定義
    ...
    HDLModule sensorInterface \\
      = new HDLModule("SensorInterface", "clk", "reset");
    // SensorInterface の I/F の定義
    ...
    /*** インスタンスの生成 ***/
    HDLInstance instanceDataProcessor \\
      = dataProcessorTop.newModuleInstance(dataProcess...
    ...
    /*** I/O の接続の定義 ***/
    instanceDataProcessor.getSignalForPort("clk").set...
    ...
```

図 4.11: Example of Top module using Synthesijer

データ処理機構から DRAM へデータを転送するためのサンプルコードを図 4.2 および図 4.3 に示す．メソッド `send_data_to_bram` が呼び出されると，クラス変数 `result_data` の内容を CoRAM に転送し，その後トリガを受け取った Memory Access Controller が CoRAM メモリの内容を DRAM に転送する．

データフローの定義：

この段階では，Python 言語によって，データフローの定義（Memory Access Controller の定義）を行う．

CoRAM アーキテクチャ実装のためのフレームワークとして，Python 言語ベースのツールである PyCoRAM を用いた．これによって Synthesijer によって生成されたトップモジュールを含めた IP が生成される．

VivadoTCL による IP の接続（Block Design）の定義：

最後の段階として，Block Design を行うための Vivado TCL の記述を行う．PyJer はこの TCL スクリプトのテンプレートを提供し，ユーザは外部ピンへの出力や PyCoRAM が生成した IP への入出力の変更などその一部の修正を行うのみでよい．

make コマンドによるビルドの実行：

ここまでの記述が完了したのち，make コマンドのみで全てのツールを通した回路の合成・配置配線が行うことができる．

Listing 4.4: TCL script to update IPs for existing Block Design

```
1 update_ip_catalog -rebuild -scan_changes
2 report_ip_status -name ip_status
3 upgrade_ip -vlnv <IP Name> [get_ips <Component Name>]
```

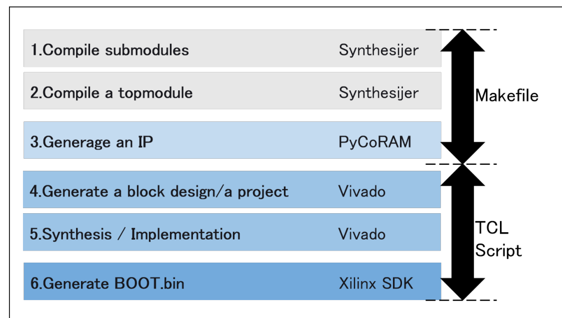


図 4.12: Build flow and tools used for build automation

Vivado IDE における合成・配置配線の全自動化

Vivado IDE の生成するプロジェクトフォルダを用いる場合，1）ソースコードやファイルの管理がしにくい，2）Vivado IDE でバッチ処理を行う場合に IP の更新時に Block Design でエラーが起こりやすい，という問題があった．そこで PyJer では Block Design を生成するための TCL スクリプトを用意するという方法によって解決した．

なお，Vivado IDE の GUI によって構築した Block Design は，Vivado IDE の File ⇒ Export ⇒ Export Block Design から TCL コードとしてエクスポートが可能であるが，これを利用した合成は困難であった．これを用いてバッチ処理によるビルドを行うとき，外部から IP が更新された際に図 4.4 の手順で IP の更新を行うとビルドが停止してしまう問題に直面したためである．図 4.4 のスクリプトは GUI で Upgrade Selected IP を行う際に実行されるものである．

Makefile および Vivado TCL によるビルド全体の自動化

PyJer のユーザは，SD カードブートを行うための BOOT.bin ファイルを make コマンドのみによって自動的に生成することができる．

BOOT.bin を生成するために必要な手順を図 4.12 に示す．複数のツールを取り入れたことによってビルドの手順が複雑なものとなっている．このため，ビルド全体を自動的に実行するための Makefile を作成し，Vivado IDE による合成・配置配線・BOOT.bin の自動生成のための TCL スクリプトを記述した．ツールのプログラムの改変を伴う複数ツールの統合は困難であり拡張性にも欠けることから，各ツールに Verilog HDL ファイルを生成させ，最終的にそれらを 1 つの IP に組み立てるという方針でビルドツールを実装した．

表 4.2: Platform used for experiment and verification

評価ボード	Digilent ZedBoard
SoC	Xilinx Zynq-7000 AP SoC XC7Z020-CLG484-1
DRAM	DDR3-1066, 533MHz, 32bit wide, 1GB
OS	Linux 4.0.0-gd94f3f3
mic	SPM0405HD4H 2MHz PDM

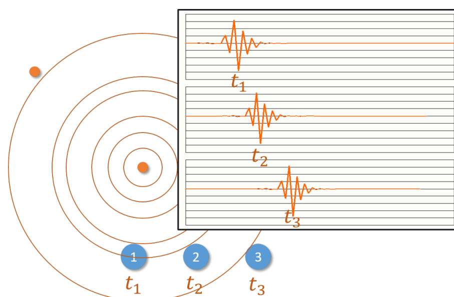


図 4.13: Position of sound source and its signal

4.1.6　評価方法

実験・評価に用いるプラットフォーム

フレームワークは Xilinx 社の SoC である Zynq-7000 を使用することを想定して構築した．このため，フレームワーク構築に必要な実験や評価を行う環境として，Zynq-7000 を搭載した Digilent 社の評価ボードである ZedBoard を使用した．

構築したフレームワークはマイクロコントローラのみではセンサ入力の処理が困難なシステムをターゲットとしている．したがって量の多いデータ入力を行うために，4.1.6 節で述べる評価用アプリケーションで用いるためのセンサとして 2MHz で PDM 出力を行うマイクを選択した．

構築したフレームワークを用いて実装するアプリケーション

評価のために，構築したフレームワークを用いて次のようなアプリケーションを実装するものとした．

音源位置推定システム：

このアプリケーションは，マイクを 3 つ用いたマイクロホンアレイによって音源の位置を推定し，可視化した位置を HTTP を通して配信するシステムである．

図 4.13 に示すように，ある音源からの音に対してそれぞれのマイクから得られる信号にはズレが発生する．この得られた複数の信号に対して図 4.14 に示すように**遅延子**を挿入し特定の方向からの波の位相を揃え和を取ることによって，その方向からの音を増幅することができる．この疑似的な指向性をペンシルビームと呼び，これを空間的に走査することで音圧分布を推定する手法を**ビームフォーミング法**と呼ぶ [75]．2 つのマイクを用いる場合，音源は 1 つの双曲線上に推定される．システム上では，マイクアレイ前方を 0[rad]，右向きを正の角として使い，式 (4.1) によっ

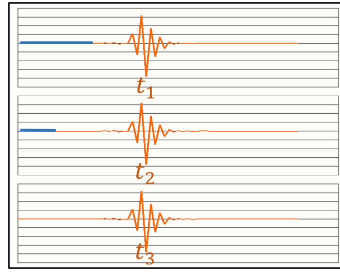


図 4.14: Insertion of delay

て遅延子の大きさを直線で近似している.

$$l_{ab}(\theta) = \frac{2D_{ab}}{v_s}\tan\theta \tag{4.1}$$

ただし，θ は音源の方向，D_{ab} はマイク間の距離，v_s は音速を示す.

本来 3 次元において音源定位を行う場合，複数のマイクの音波を周波数領域で処理する必要があるが，今回は簡単のために 2 次元の方式の簡易的な拡張を行った．正面を見て左右方向の角を θ，上下方向の角を φ としてそれぞれ 2 つのマイクを用いて 2 次元的音場推定を行い，2 本のベクトルを得る．θ 方向に関する横ベクトルを縦に正方行列になるまで並べ，また φ に関する縦ベクトルを横に正方行列になるまで並べる．得られた 2 つの行列の和を簡易的な音場の推定とした.

$$R = \begin{pmatrix} \theta_1 & \theta_2 & \cdots & \theta_n \\ \theta_1 & \theta_2 & \cdots & \theta_n \\ \vdots & \vdots & \ddots & \vdots \\ \theta_1 & \theta_2 & \cdots & \theta_n \end{pmatrix} + \begin{pmatrix} \varphi_1 & \varphi_1 & \cdots & \varphi_1 \\ \varphi_2 & \varphi_2 & \cdots & \varphi_2 \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_n & \varphi_n & \cdots & \varphi_n \end{pmatrix} \tag{4.2}$$

この手法では，正しく音源の位置が推定できるのは音源が 1 つの場合のみである.

ビームフォーミング処理の機構は Java 言語によって記述され，CPU 上で動作する Web サーバによって音源の位置が可視化された画像が配信される.

4.1.7 アプリケーションとその評価

この章では，4.1.6 評価方法の章で述べたアプリケーションを実装の評価およびフレームワークに関する考察を行う.

コードの記述量とハードウェア量

作成したシステムのコード量は，Java 言語が 398 行，Python 言語が 30 行，Verilog HDL が 118 行となった．これに対する Zynq-7000 の PL 部の使用率は表 4.3 の Full の列の通りとなった．また，比較のために，FPGA に Memory Access Controller のみを実装し合成・配置配線した場合 (Empty とよぶ)，および Memory Access Controller も削除し，Zynq ベースシステムのみを合成・配置配線した場合（Base とよぶ) のハードウェア使用量を測定した．その結果も表 4.3 に示した.

表 4.3: Zynq-7000 utilization

	Full	Empty	Base
LUT as Logic	9.13%	8.02%	4.00%
LUTRAM	1.06%	1.05%	0.45%
FF	5.88%	2.91%	1.50%
BRAM	15.0%	8.93%	1.43%
IO	6%	4.50%	4.50%
BUFG	6%	6.25%	6.25%

Listing 4.5: Java program for performance measurement

```
1 @auto
2 public void controller()
3 {
4     while (true){
5         o_signal_2m_indctr = !o_signal_2m_indctr;
6         execute_beamforming();
7     }
8 }
```

実行速度

`execute_beamforming()` メソッドの実際の動作速度を，図 4.5 に示すコードを用いて計測した．`o_signal_2m_indctr` は外部 I/O に接続された 1[bit] 信号線である．この Java コードは，ビームフォーミング処理が１度完了する毎にその値が反転する記述である．この出力をオシロスコープを用いて観測することで動作周期を計測した．

この結果を図 4.15 に示す．オシロスコープの出力は約 87.2[Hz] となっているため，ビームフォーミング処理の動作レートは約 174[Hz]，動作周期は約 5.7[ms] となる．Zynq-7000 の PS 部でセンサ入力処理を除いた同様のプログラムを動作させた場合，処理のレートは約 691[Hz]，処理時間は約 1.45[ms] であった．すなわち PL 部におけるビームフォーミング法の動作速度は PS 部のみの場合と比べておよそ４倍となり，性能が低下していることがわかった．

考察

ハードウェアの消費量について　500 行程度のそれほど多くない記述によってセンサと SoC を活用したシステムのプロトタイピングを行うことが可能であった．またハードウェア使用率が低く抑えられているという結果となった．この結果から，より複雑なアルゴリズムの使用や，より規模が小さく省電力な SoC に変更することができることがわかる．

LUT に関して，Empty から Full への増加量は，1.1%程度となっている．これは，センサ入力のための処理やビームフォーミング処理が単純な処理の繰り返しであったために回路がそれほど複雑とならなかったことが考えられる．BRAM に関しては，Empty から Full への増分が他要素と比べて大きかった．これは，センサからのデータを蓄積するためのデータバッファを Java コード内で宣言したことが要因であると考えられる．Memory Access Controller はデータ転送を行うものであるから，それほど Empty の内容が大きく変化するとは考えにくい．

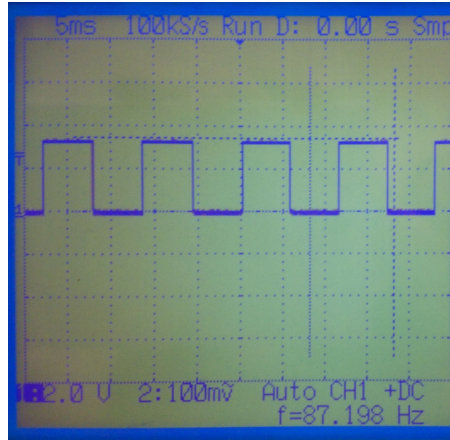


図 4.15: Results of beam-forming

これらのことから，センサの接続数を増やした場合 BRAM がより消費され，処理の内容をより複雑なものとした場合には，処理の内容に応じて各リソースが消費されるものと予想できる．

分解能の向上について　データ処理のレートが PS 部のみの処理と比較して低速となってしまったが，今回の実装では FPGA アクセラレータではビームフォーミング処理の並列処理化は行っておらずソフトウェアベースでの実装と同様の手続きで処理を行うため，単純な繰り返し処理においては動作周波数の高い PS 側が有利だったのではないかと考えられる．これは Altera 社の提供する OpenCL**altera˙cl** のようなチューニングツールを用いることによって改善が可能である．また，今回の実装ではマイクの数が少なく多くのサイドローブによる偽信号が出現した．解決策としてマイクの数を増やすことが挙げられるが，Zynq-7000 の PS 部のみによる実装では 9 個が限界であった．対して本フレームワークによる実装の場合センサ入力は並列に行うため，入力数の制約は外部入出力ピンの数によるものとなる．同様のシステムで商用のものには，36 個のマイクを用いた事例が存在する **onjyo˙sys** ため，9 個の入力ではパフォーマンスの不足が考えられる．

ロジックの並列化について　Synthesijer は Java 言語の仕様である thread 構文をサポートしているため，PyJer を用いた場合でもアルゴリズムの並列処理の記述は可能である．しかしながら，配列をメソッドの引数に指定することができない，int 配列のみのサポート，クラスの実行時インスタンス化が不可など高位合成に伴う制約がある．単純な処理であれば少々のコストで並列処理の記述が可能であるが，ソフトウェアとして Java 言語を記述する場合と比較して大きな計算コストや記述量を要する場合もあるということが予測される．

4.1.8　PyJer の有効性の検討

PyJer を用いた場合システム全体の性能は，PyCoRAM 単体もしくは HDL ベースの開発と比べて低下すると予想される．高位合成ツールを用いる場合，ロジックの最適化を HDL ベースの開発と同程度に行うことは現状困難である．

ところで，CoRAM から DRAM へのデータ転送部分も，Synthesijer のみで記述可能である．しかしツールの特性から，Synthesijer と比較して PyCoRAM を用いたデータ転送の方が高速なものとなる．したがって，SoC をターゲットプラットフォームとして CPU と FPGA の連携を考える

場合，Synthesijer 単体での開発と比較して，PyJer を用いた開発が有効であるといえる．8 章の結果をふまえると，Computing Logic も性能ボトルネックとなっていると考えられる．Computing Logic の性能は高位合成に用いるツールに依存するものとなるため，Synthesijer の記述能力や出力される論理回路の性能の向上に伴って PyJer によって構築されるシステムの最大性能は向上し，より HDL ベースの開発で実現できる性能に近づくこととなる．

4.1.9　まとめ

これまで FPGA を用いた開発において，単一の高位合成ツールでは機能が不足する場合があった．しかし複数のツールを利用して回路設計を行う場合には煩雑な手順が伴った．PyJer を用いた場合，汎用の高位合成ツールとチューニングのための高位合成ツールを組み合わせそれぞれの利点を活かすことが可能となり，SoC を用いたシステムの高速なプロトタイピングが実現できる．PyJer によって設計された SoC アプリケーションは CPU のみによる実装と比較して処理レートの低下が認められたが，プログラムを変更することなくより多くのデータを入力できることが可能となる．

4.2　C 言語ベースの IoT/CPS エッジデバイス向けフレームワーク

4.2.1　はじめに

前の節では，FPGA にハードマクロとして MPU が搭載された **SoC FPGA** と，**高位合成**の技術を組み合わせることで，冒頭で述べた IoT/CPS 技術の課題を解決することについて取り組み [76]．SoC FPGA と複数の開発支援ツールを活用した IoT/CPS のための FPGA 開発フレームワーク：PyJer を提案した [77]．目的を IoT/CPS の領域に特化し設定することで，複数のツールを組み合わせによってソフトウェアベースの場合に近いシステム開発を目指した．PS(Processing System) と PL(Programmable Logic) が協調し，I/O 処理やセンサ値の移動平均の計算などのデータの下処理を PL にオフロードする．これによって，TCP/IP や Bluetooth Low Energy(BLE) の利用などのソフトウェア資源が活用でき，同時に取り扱うことのできるセンサ・アクチュエータ数の増加とデータ処理能力の強化も実現できる．

近年，IoT/CPS の領域には，Microsoft や NTT Data など組み込み開発を専門としない企業も数多く取り組んでいる **mizuho˙iot**．収集したデータの収集・処理は IaaS や PaaS などの形でクラウド化・プラットフォーム化がなされ，Raspberry Pi や Arduino の登場に伴うデバイス開発の容易化によって，今後 IoT デバイス開発にも様々な業種の企業が参加することが予測できる．

この節では，PyJer における問題点を整理し，言語を C 言語に統一した新たなフレームワークについて説明する．

4.2.2　新しいフレームワークのコンセプト

PyJer の概要と問題点

PyJer では，ベースとなる高位合成ツールを Java 言語ベースの Synthesijer[68] とし，PS-PL 間の連携の最適化のために Python 言語ベースのツールである PyCoRAM [73] を用い，センサやアクチュエータとの通信に関しては，HDL によってライブラリが記述できるような構成であったが，この構成では次のような問題点があった．

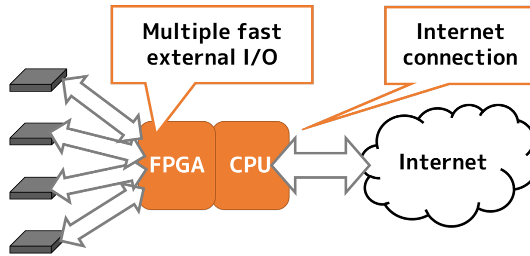


図 4.16: Utilization of SoC FPGA in IoT/CPS area

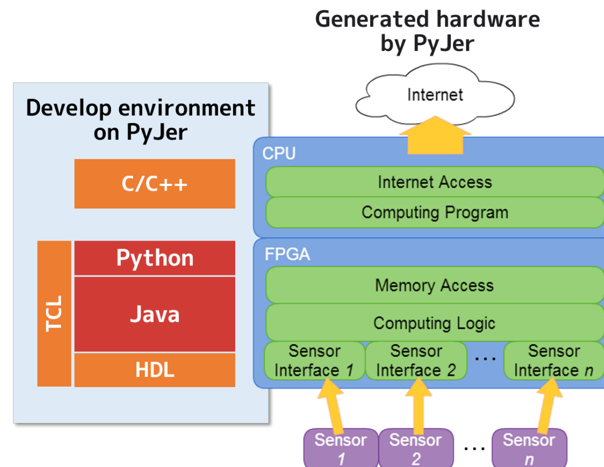


図 4.17: Description languages used in PyJer

- 複数の記述言語の習得が必要
- 個々のツールやフレームワーク全体のアップデートの複雑化
- 速度最適化のためにはハードウェアの知識が必要

複数のツールを用いる構成では，複数の言語の習得が必要となってしまい，学習コストが大きなものとなってしまう．またツール毎にアップデートが必要となるため，フレームワーク全体としてのメンテナンス性も下がってしまう．加えて PyJer は，データ処理は PL で行うよりも PS で行うほうが高速であるという大きな問題を抱えている．多くの高位合成ツールは並列化やパイプライン化などのハードウェアの最適化のための機構を備えるが，このような指示子などを用いたハードウェア化のための詳細な最適化には，ハードウェアに関する知識が少なからず必要となる．

コンセプト

新しいフレームワークは，次の 3 点の実現を目標とする．複数ツールを使うことに起因する問題を取り除くため，C++言語のフレームワークとして統一する．

1. 幅広いプログラマが利用可能である
2. プラットフォームによらない PS-PL 間連携のサポート

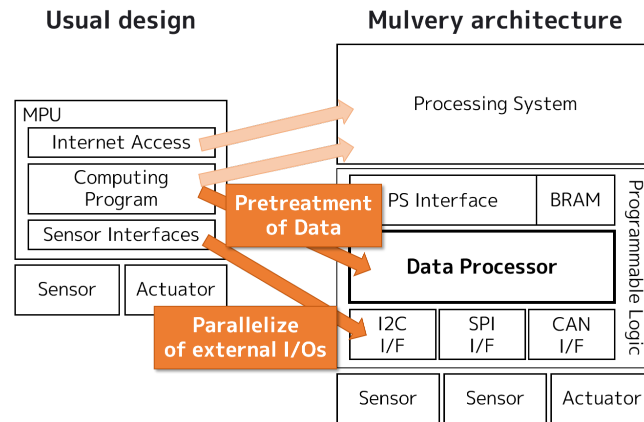


図 4.18: Comparison of common design and Mulvery architecture

3. 速度最適化を行う支援

１の目標は，最適化指示子や並列処理の制御を廃することで，回路設計のための知識を必要としないものとすることで実現する．２の目標は，PS-PL 間連携のための統一的インタフェースとアーキテクチャを提供することで実現する．３の目標は，IoT デバイスにおいて頻繁に用いられる演算に関して最適化済み演算器を提供することと，外部 I/O を用いた通信に関する処理の並列化を実現するアーキテクチャの提供によって実現する．

4.2.3 Mulvery アーキテクチャ

PyJer をもとに，新しいフレームワークの用いるアーキテクチャとして Mulvery アーキテクチャを提案する（図 4.18）．

I/O の並列化によるセンサデータ取得の高速化

従来の MPU のみを用いたデバイス設計では，MPU 単体で１：外部との通信；２：データの処理；３：センサデータの取得をまとめて行うものであった．この構成では外部 I/O が増えるに従いデータ処理に割くことのできる計算リソースが減少するという問題がある．そこで Mulvery アーキテクチャでは，MPU の計算リソースの多くを消費する [71] センサやアクチュエータとの通信に必要な処理を，I/O それぞれに対して独立した回路によって行う．投機的に最新のセンサデータを取得し自身のレジスタに保持しておくことで，必要に応じて即座に値を返すことができる．

データの前処理のオフロードによる CPU 資源の開放

加速度センサのデータなどについて，ノイズ除去のために移動平均を取る前処理がよく行われている．このようなセンサデータを使うために必要な処理や整形などは，PL 上の “Data Processor” で行う．複数のセンサから集めたデータを用いて新たな情報を生成することもできる．したがって PS で動作するプログラムは，データの前処理に計算リソースを割くことなく，必要なデータ処理やネットワークを介した通信を行うことができる．

Listing 4.6: Data Processor using Mulvery Core

```
1  #include "mulvery_core.h"
2
3  MULVERY_CLASS{
4  public:
5    sc_out<unsigned int> addr;
6    sc_in<int> din; sc_out<int> dout;
7    sc_in<bool> trg;
8    MULVERY_MAIN()
9    {
10     psif.write_bram(0xac, i2cifd.get_data(0x12));
11     DEBUG_PRINT("0x%x = %d\n", 100, psi.read_bram(100));
12   }
13
14 private:
15   mulvery_ns::mulvery_core::PSInterface psif;
16   mulvery_ns::mulvery_core::I2CIFDriver i2cifd;
17 };
```

CPU と FPGA の連携方式の定義によるポータビリティの向上

PS と PL の間の通信は，“PS Interface” によって隠ぺいされる．PS Interface が持つ BRAM は PS からアクセス可能なメモリ空間の一部と同期される．したがって，共有メモリを介した通信のようにデータをやり取りすることができる．加えて，PS から PL およびその逆方向の非同期な I/O チャネルを持つ．これによって，割り込み処理のサポートも可能となる．

ソフトウェア資源の活用

プログラム実行が適するデータ処理や TCP/IP，BLE などを用いた外部システムとの通信は，“Processing System” が担う．Mulvery アーキテクチャを用いることで，これまで I/O にあった MPU における処理のボトルネックが改善され，処理能力の多くをデータ処理に割くことが可能となる．このため Mulvery Core を使わない場合と同じアルゴリズムを用いた場合でも，高速化を見込むことができる．

4.2.4 Mulvery Core フレームワーク

本稿では，Mulvery アーキテクチャを実現するためのフレームワークとして，Mulvery Core フレームワークを提案する．並列制御を廃しながら Mulvery アーキテクチャを実現するためのコーディングスタイルとセンサや PS とのインターフェイスの実装方法について述べる．

Mulvery Core フレームワークを用いた開発

Mulvery Core フレームワークを用いた開発では，図 4.18 の I2C Interface や SPI Interface などの Sensor Interface および PS Interface はなるべく再利用し，Data Processor を実装することが中心となる．I^2C を用いたセンサからデータを取得し，BRAM に値を書き出す Data Processor の実装例を図 4.6 に示す．Mulvery Core フレームワークは，コードの可搬性を高めるために C++ のクラスライブラリとして提供する．

Listing 4.7: Source code of Fig. 4.6 aftre macro expansion

```
1  SC_MODULE(Mulvery){
2  public:
3    sc_out<unsigned int> addr;
4    sc_in<int> din; sc_out<int> dout;
5    sc_in<bool> trg;
6    sc_in_clk CLK; sc_in<bool> RST;
7    SC_CTOR(Mulvery){
8        SC_THREAD(mulvery_main);
9    }
10   void mulvery_main()
11   {
12     psif.write_bram(0xac, i2cifd.get_data(0x12));
13   }
14
15 private:
16   mulvery_ns::mulvery_core::PSInterface psif;
17   mulvery_ns::mulvery_core::I2CIFDriver i2cifd;
18 };
```

コーディングスタイル

C/C++コードのままの形では並列実行の概念を扱うのは困難である．この問題を解決しつつコーディング時における並列処理の概念を廃するため，図 4.6 の MULVERY_CLASS や MULVERY_MAIN のようなマクロを用いたクラス定義やメインメソッドの定義などを行う形とする．これはプリプロセッサによって，コンパイルの時点で，図 4.7 に示すようなソースコードへと展開される．フレームワークが並列制御などを適切に行えるよう，SystemC 言語を用いた実装に変換される．

Data Processor の PS Interface の利用

図 4.8 に，PSInterface クラスとして例を示した．PSInterface の持つ BRAM は，PS からアクセス可能な DRAM に対してインコヒーレントである．read/write メソッドを用いることで BRAM へのデータの書き込みを行うことができる．send/recv メソッドを用いて，DRAM へのデータ転送や BRAM へのデータ受信を指示することができる．

Data Processor の Sensor Interface の利用

図 4.8 に，I2CIFDriver クラスとして例を示した．Sensor Interface に合わせて，適当なドライバクラスを実装する．汎用的な通信プロトコルに関しては，標準ライブラリとしてインタフェースクラスを用意することを検討している．

Sensor Interface の実装

汎用的でない通信方式を用いたセンサをサポートするために，ユーザ独自の Sensor Interface のの実装をサポートする．ユーザオリジナルの Sensor Interface の例を図 4.9 に示す．Sensor Interface として機能させるクラスは，SENSOR_INTERFACE マクロを用いて定義する．センサに接続するためのポートおよび Data Processor に接続するためのポートを公開メンバ変数として定義する．

Listing 4.8: Example of Interface on Mulvery Core framework

```
1  class PSInterface{
2    void write_bram(const unsigned int &addr, const int &data);
3    int read_bram(const unsigned int &addr);
4    void send_dram(const unsigned int &addr, const unsigned int &size);
5    void recv_dram(const unsigned int &addr, const unsigned int &size);
6  };
7
8  class I2CIFDriver{
9    void write_data(const unsigned int &addr, const unsigned int &data);
10   unsigned int get_data(const unsigned int &addr);
11 }
```

Listing 4.9: Example of Sensor Interface

```
1  SENSOR_INTERFACE(OrigIF){
2  public:
3    // to sensor
4    sc_in<bool> SPC, DIN;
5    sc_out<bool> START_TRG;
6
7    sc_out<int> data; // to Data Processor
8
9    OrigIF(){
10     data.write(0);
11   }
12
13 private:
14   void polling();
15 };
```

Listing 4.10: Definition of Sensor Interface that Data Processor uses

```
1 <Instance Name>:|}
2   interface: <Sensor Interface Name>|}
3   connection:|}
4     <Sensor Interface Port 1>: <Data Processor Port 1>|}
5     <Sensor Interface Port 2>: <Data Processor Port 2>|}
6     ...|}
```

```
LIS3DH_1:
  interface: LIS3DHIF
  connection:
    x: x1
    y: y1
    z: z1
```

```
LIS3DH_2:
  interface: LIS3DHIF
  connection:
    x: x2
    y: y2
    z: z2
```

図 4.19: Example definition of connections of LIS3DHIF

メンバメソッドとして polling メソッドを実装することで，センサに対して定期的にデータを問い合わせることができる．

Sensor Interface-Data Processor 間の配線

Sensor Interface は様々な種類のものが複数配置されることがあるため，Data Processor との接続の方法を明示的に示す必要がある．この配線の定義のフォーマットを図 4.10 に示した．Mulvery Core フレームワークでは, YAML 形式の記述によって Data Processor が使用する Sensor Interface の種類，数，配線の方法を定義する．Sensor Interface のインスタンス名に対して，使用するインタフェース名を interface に，と配線の接続方法を connection に設定することで定義できる．

4.2.5 具体的なアプリケーションの実装例

FPGA アプリケーション開発の具体例として，２つの３軸加速度センサを取り扱うアプリケーションの実装について説明する．加速度センサのデータは一般に多くのノイズを含むため，本稿では移動平均を取りセンサの値とするものとした．加速度センサとして STM 社の LIS3DH を用いることを想定した．

センサインタフェースの定義：図 4.11 に，LIS3DH のインタフェース定義のサンプルコードを示した．LIS3DH には I^2C および SPI の二種類のインタフェースが用意されているが，本稿では SPI インタフェースを用いた例を用いる．polling メソッドでは，1ms 毎に x,y,z 軸の値を取得し，直近 5ms の平均の値を出力するように実装されている．

Data Processor の実装： LIS3DHInterface クラスのデータを Data Processor で扱う．この Data Prosessor の実装例を図 4.13 に示した．今回 LIS3DHInterface へのアクセスは複雑でないため，ドライバクラスの実装は行わない．２つのセンサを扱うために，LIS3DHIF の出力ポートに対応する入力ポートを２組用意している．メインの処理として，10ms に一度それぞれのセンサの傾きを計算し，結果を DRAM へバースト転送している．

Data Processor と Sensor Interface の接続： Data Processor と Sensor Interface の接続の定義を図 4.19 に示した．YAML 形式で接続する Sensor Interface と接続するポートを指定する．インスタンス名を変更することで複数の同一 Sensor Interface をインスタンス化することができる．

Listing 4.11: Example of implementation of LIS3DH interface (1)

```
1  SENSOR_INTERFACE(LIS3DHIF)
2  {
3  public:
4    // sensor connections
5    sc_out<bool> SPC, SDI, CS;
6    sc_in<bool> SDO;
7
8    // output ports
9    sc_out<int> x, y, z;
10
11   SENSOR_CTOR(LIS3DHIF)
12   {
13     SPC.write(true);
14     CS.write(true);
15   }
16
17 private:
18   void polling();
19   void wait_one_clk();
20   void wait_ms(const int &n);
21 };
```

Listing 4.12: Example of implementation of LIS3DH interface (2)

```
1  // virtual method of SENSOR_INTERFACE
2  void LIS3DHIF::polling()
3  {
4    int tmp_x, tmp_y, tmp_z;
5    int buf_x[5], buf_y[5], buf_z[5];
6
7    while (true){
8      ...
9      tmp_x = 0;
10     for (int pos = 7; pos >= 0; pos--){
11       tmp_x += (SDO.read()?1:0) << pos;
12       wait_one_clk();
13     }
14     ...
15     x.write(average(buf_x, 5));
16     ...
17     wait_ms(1);
18   }
19 }
```

Listing 4.13: Example of Data Processor that uses LIS3DH

```
1  MULVERY_MODULE
2  {
3  public:
4    // for LIS3DHIF
5    sc_in<int> x1, y1, z1, x2, y2, z2;
6
7    MULVERY_MAIN()
8    {
9      while (true){
10       int tilt1 = get_z_tilt(x1, y1, z1);
11       int tilt2 = get_z_tilt(x2, y2, z2);
12       psif.write_bram(0x00, tilt1);
13       psif.write_bram(0x01, tilt2);
14       psif.send_dram(0x00, 2);
15       wait_ms(10);
16     }
17   }
18
19 private:
20   mulvery_ns::mulvery_core::PSInterface psif;
21 };
```

考察： Mulvery Core を用いることで，モジュール間の連携や並列制御を意識することなく複数のセンサからデータを取り出すことができた．MPU 側では移動平均などの下処理の必要がないため，すぐに必要な形のデータを利用することができる．移動平均のような必須となる下処理と傾きの計算のようなアプリケーションによって異なる下処理を Sensor Interface と Data Processor の間で分離して実装することでコードの再利用性を高めることが可能である．等価なシステムを MPU のみで実装することを考えた場合，逐次実行型のモデルとなる．このためさらにセンサを増やす場合などはマルチタスクの実現等も検討する必要がある．

4.2.6 データコレクタとしての活用

多数のエンドデバイスから生成されたデータは分析のために一箇所に収集される．大規模なデータを収集するための研究として，ネットワークのスケジューリングによる最適化の研究 [78] や，IoT のためのツリー構造による最適化の研究 [79] などが行われている．

データコレクタ

複数のデータ源からデータを集め，データストレージやデータ処理機構への転送をコーディネートする**データコレクタ**が注目を集めている．代表的なものとして，オープンソースプロジェクトの fluentd[80] がある．また，このようなデータコレクタをエンドデバイスまで適用できるよう，ハードウェアデバイスに向けたより軽量で，ネイティブコードで動作する Fluent Bit[81] も公開されている．

我々は，Mulvery アーキテクチャを用いたハードウェアをデータコレクタへ適用することによって，高速化や省電力化が期待できるのではないかと考えている（図 4.21）．Mulvery Core によるハードウェアは fluentd/fluent bit のプラグインとして記述されるソフトウェアによって接続され，fluentd/fluent bit のハードウェアアクセラレータとして活用することができる．エンドデバイスではセンサとの通信の並列実行や CPU リソースの開放などのメリットが得られる．エッジコン

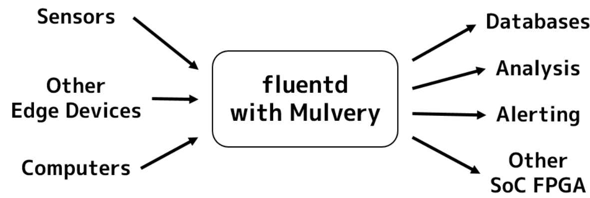


図 4.20: Target of fluentd

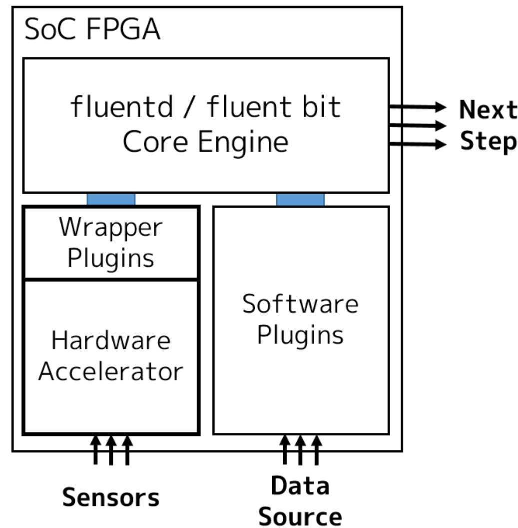


図 4.21: Mulvery on fluentd

ピュータや上位のコンピュータでは，データの受け口の並列化やフィルタ処理のハードウェアオフロードなどの活用が考えられる．

ハードウェアメタ記述言語 Mulvery

今後，fluentd/fluent bit への適用にともなって，Mulvery Core フレームワークを利用した，Ruby 言語ベースのハードウェアメタ記述言語 Mulvery の実装を予定している．Mulvery Core フレームワークのメタプログラミング用 DSL として構築し，fluentd/fluent bit 用のプラグインを生成する機能を付加する．Mulvery 言語の使用イメージを図 4.14 に示した．Mulvery Core を用いる場合，複数のモジュールの実装が必要があったが，Mulvery 言語を用いる場合はなるべく単一のスクリプトで手早くハードウェアが設計できるような仕様を目指す．

4.2.7　まとめ

本稿では，これまでに提案した高位合成フレームワーク PyJer で挙げられた問題点の改善と SoC FPGA 活用の容易化を目的に，Mulvery Core フレームワークを提案した．Mulvery Core フレームワークは Mulvery アーキテクチャを構成することで，IoT/CPS の領域のアプリケーションの開発やプロトタイピングの高速化を目指す．さらに，センサや PS との通信のためのモジュールを

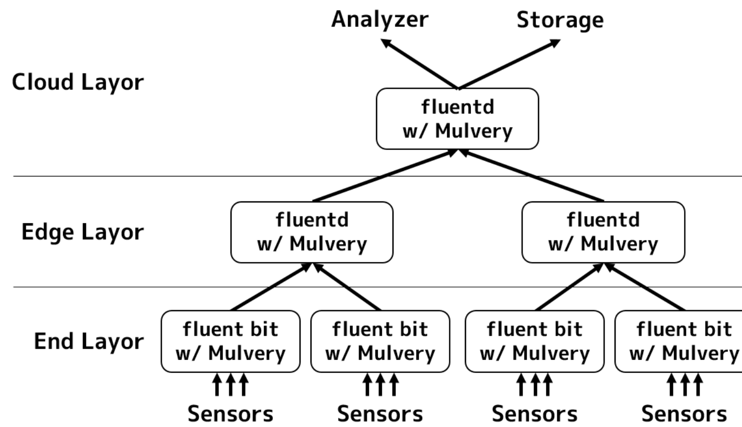


図 4.22: Schematic diagram of configuration with Mulvery applied to the entire data collection system

Listing 4.14: Mulvery language usage example

```
1  m = MulveryModule.new()
2  m.add_module(PSIF.new(), 'psif')
3  m.add_module(I2C.new(), 'sens')
4
5  a = MulveryVal.new()
6  a = 0
7  for (i = 0; i < 100; i++){
8    a += m.module['sens'].read().wait(10)
9  }
10 a /= 100
11
12 m.module['psif'].write_bram(0x00, a.get_val())
13
14 m.gen_verilog()
```

分離することでポータビリティや再利用性の高いソースコードとすることが可能となる．ソフトウェア的な実行が可能となるように，Mulvery Core フレームワークは C++のライブラリとして提供される．SystemC 言語に展開されるため，SystemC で用いるテストベンチや資源の活用が可能である．Sensor Interface と Data Processor の接続は，YAML 形式のファイルによって定義される．これによって SystemC で必要な複雑なモジュール間接続の実装が不要となる．さらなる展開として，データコレクタとしての活用を検討している．エンドデバイスからデータフローの最上流まですべてのデバイスに Mulvery によるハードウェアアクセラレーションによるメリットを提供することを目指す．

第5章　ハードウェア設計に適したプログラミング・パラダイム

5.1　背景

5.1.1　前章での研究の問題点

前章では，組込み環境への FPGA の導入を目指した研究について述べ，センサやアクチュエータのデータ処理を FPGA にオフロードするためのツール：PyJer [69] をについて説明した [76], [77]．しかしながら PyJer は，ハードウェアを合成するために複数のツールを用いており，複数のプログラミング言語を用いる必要がある．加えて，合成プロセスが複雑なものとなり，デバッグが複雑なものとなる問題点があった．この点を踏まえて，C 言語のみで記述可能なハードウェア開発フレームワークを提案した [82]．システムのアーキテクチャを，センサ/アクチュエータを用いるシステムに対して効果的な構造（図 5.1）に定型化することによってハードウェアの合成を容易なものとしながらも効率の良いハードウェアに合成することが可能となった．

しかしながら，C 言語を用いたプログラミングは依然としてモダンな開発環境とは言い難い．近年，プログラミングには Python や Ruby，Scheme に代表されるような軽量プログラミング言語 [20] が多く用いられるようになっている．2017 年の IEEE の記事 [21] では，Python を No.1 のプログラミング言語として挙げている．このような言語には生産性を高めるための工夫が多く施され，またユーザコミュニティによって様々なライブラリが開発・共有されコードの再利用性の高さを伺い知ることができる．プログラミングの技術の高まりにも関わらず，依然多くの高位合成系は C や Java などの言語をベースとしている．これらの言語にも数多くのアップデートが施され，生産性や再利用性を高めるための工夫が数多く施されるようになったが，高位合成系ではこれらの機能をサポートしていないことも多い．

また，ソフトウェアとして利用されていた既存のプログラムから回路を合成しようとした場合には，一般に性能が低下してしまう．性能低下を回避するためには回路設計の知識と HLS ツールにおける技法を熟知したエンジニアによるコードの最適化が必要となる．

回路設計とプログラミングのパラダイムの差異が引き起こす問題

筆者は，ほとんどの HLS ツールにおいてプログラムの最適化が必要であったり C や Java の一部の言語機能に制約が必要な理由は，回路設計とプログラミングのパラダイムに大きな開きがあることであると考えている．多くのプログラミング言語は制御の流れ（コントロールフロー）を記述し，CPU を制御するための命令列に変換する．対して回路設計で必要なのはデータの流れ（データフロー）の記述であり，実際に既存のハードウェア記述言語はいずれもデータフロー型の記述であるレジスタ転送レベルの記述を中心に用いている．

このような違いは，HLS ツールを開発するにあたって深刻な問題を引き起こす．HLS では，制御フローをデータフローに変換する必要があり，スケジューリング問題を解く必要があることであ

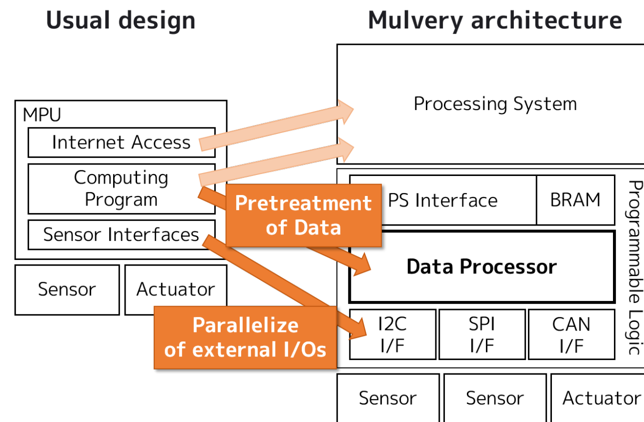


図 5.1: Comparison of architecture with MCU-only design and FPGA-based I/O processing

る．スケジューラは制御フローから制御やデータの依存関係を解析し，依存関係がないと判断された部分を並列に実行するようにスケジューリングしていく．しかしながら，大局的な並列性の解析や複雑なループの並列化などは現在のコンパイラ技術でも困難であり，ユーザによって様々な指示を加えてコードの最適化を行う必要がある．

したがって，前章で説明したような用途を限定することで最適化を最小限に抑えるような工夫はこれらの問題の根本的解決策とはならないものであったといえる．

Functional reactive programming を用いたアプローチ

パラダイムの違いが引き起こす問題を踏まえ，本研究ではこれまでハードウェア設計の文脈では用いられてこなかった言語モデルによるハードウェア設計ツール Mulvery を提案する．本研究の目標は，新しいハードウェア設計手法を提案し，アノテーションによる最適化やプログラミング言語の構文の拡張を行わずに速度や面積の効率のよい回路を生成することである．Mulvery はハードウェア設計と近いパラダイムを持つファンクショナルリアクティブプログラミング (functional reactive programming; FRP) を導入する．Web プログラミングや Android アプリなどの GUI の実装に多く利用されている API 定義である Reactive Extensions (Rx) に準拠する形で Ruby 用ライブラリとして実装する．Mulvery は Rx のオペレータをパイプライン化されたモジュールに変換し，HDL による記述を生成する．ほとんどのオペレータは高階関数であり，あらゆる型のデータの処理に応用可能である．

Rx はソフトウェアのプログラミングのために定義された API であり，Mulvery における使い方もプログラミングする場合と同様であるため，一見 HLS ツールのように見える．しかしながら，FRP はデータフロー型のプログラミングパラダイムであるため，Mulvery はプログラムから直接データフローグラフ (data-flow graph; DFG) を生成することができる．したがって，Mulvery は HLS ツールではなく，メタプログラミングの技法に近いものであるといえる．

この章における中心となる目的は，ハードウェア設計の専門家ではないソフトウェア技術者でも効率のよいハードウェア設計が可能となるプログラミングパラダイムの検証である．この章では，FRP を応用したハードウェア設計手法の提案と，次の項目についての説明をする：

- FRP と LL によってハードウェア設計コストを削減すること，

- ソフトウェアとハードウェアのコデザインのサポートすること,
- 動的型付け言語におけるハードウェア設計技法を提案すること.

軽量プログラミング言語のユーザ層の多くはハードウェアの知識が十分でないことが予想できる．そこで，ユーザによる合成のチューニングのフェーズをなるべく最小限とし，ハードウェアの知識がなくとも高いパフォーマンスを発揮するシステムを設計することが可能な仕組みの実現を目指す.

5.1.2 関連研究および先行事例

軽量言語を用いたメタプログラミング

一般に軽量言語とされる Scala や Haskell などの関数型言語によるハードウェア開発支援ツールの事例 [83], [84] が存在する．また，複数のパラダイムを取り入れ素早くハードウェアを構築することに注目した Python によるツール [85] もある．これらのツールは DSL として利用しハードウェアを組み立てることに注目しているツールであることから，高位合成と異なりビヘイビアをハードウェアへと変換しない.

軽量プログラミング言語による高位合成ツールが生まれにくい背景として，軽量プログラミング言語の多くが型システムに動的型付けを採用していることが挙げられる．実行時まで型が決定しない性質はハードウェア化が困難であるため，このような状況となっている.

PyMTL [5], [43] では，Ruby を用いる Mulvery と同様に，軽量言語である Python を用いている．PyMTL の関数レベル記述は Mulvery と近いものであるので，目的の違いについて整理する.

PyMTL と Mulvery で大きく異なる点として，ターゲットとなるユーザの違いが挙げられる．Mulvery が回路設計の知識を必須としないツールとしていることに対し，PyMTL ではあくまで回路設計技術者の生産性を高めるためのツールとして設計されている．呼び出される関数は全て RTL で実装されている必要があるため，新たな関数を利用するたびに CL 記述や RTL 記述を追加する必要があり，回路設計の知識が要求される．対して Mulvery では，プログラミングモデルとして体系だった Rx の API の RTL 記述があらかじめ用意されており，ユーザが自ら新たな RTL を用意する必要はなく，回路設計の知識を必須としない．裏を返せば，Mulvery ではサイクルレベルやレジスタ転送レベルでの詳細な制御は困難である.

また PyMTL は RTL 設計を意図したツールであるため，アーキテクチャの自動的な最適化等については考慮されておらず，ユーザがアーキテクチャ探索を行う．対して Mulvery は，Rx を用いた Ruby プログラムからデータフローの DAG を抽出し，自動的な最適化を施す.

ハードウェア/ソフトウェアの協調設計環境

CPU と FPGA が共存する関係においては，ハードウェアの開発支援のみならず，ハードウェアシステムとソフトウェアシステムの統合の支援も重要なものとなる．CPU-FPGA 間の通信やデバイスドライバの実装など様々な作業が必要であり，CPU のみの環境に FPGA を導入するには大きな開発コストを要する．現在の主なコデザインのアプローチには，Xilinx 社の SDSoC のようにソフトウェアの一部を選択してハードウェア化するものと，OpenCL のようにヘテロジニアス環境を想定して記述するフレームワークとするものの 2 種類がある．だがこれらのツールはソフトウェアのアクセラレーションに重点を置いているため，外部 I/O を扱うには困難が伴う.

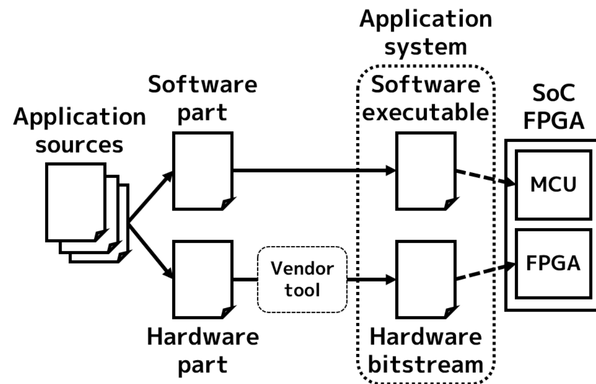


図 5.2: The synthesizing flow of Mulvery

5.2 Mulvery フレームワーク

この節では，Mulvery フレームワークの構成とアプリケーション開発の手順について説明する．

Mulvery が主なターゲットとするデバイスは，プロセッサと FPGA の両方をひとつのチップ上に備えた **SoC FPGA (system-on-chip FPGA)** である．このようなデバイスは，Xilinx 社の Zynq-7000 シリーズや Intel 社の Stratic 10 SoC FPGA など様々なベンダが生産しており，組込みシステムなどの用途に用いられている．図 5.2 はプログラムが SoC FPGA デバイスに書き込まれるまでのフローである．Mulvery は，プログラムを読み込むとプロセッサ上で動作するプログラムと FPGA に書き込む bit ファイルの両方を出力する．つまり，Mulvery はプログラムをソフトウェア部とハードウェア部に分割し，これらの間を取り次ぐためのプログラムやハードウェアを自動的に生成する．

5.2.1 Mulvery フレームワークの構成

Mulvery フレームワークの概念図を図 5.3 に示す．ユーザは Mulvery が提供するライブラリを使ってプログラムを記述する．このライブラリでは，様々なクラスがハードウェアを合成するようにオーバーライドされている．したがって，このプログラムを直接 Ruby インタプリタで実行することでハードウェアとソフトウェアが生成される．言い換えれば，Mulvery ライブラリを読み込んで実行された場合には Ruby インタプリタは実際の処理や計算を行わない．このように既存のクラスやメソッドをオーバーライドして実際の挙動を変更する手法は，Python 言語の計算フレームワーク NumPy[86] の GPU アクセラレーションを行うためのライブラリ CuPy[87] でも利用されている．Mulvery フレームワークでは Ruby 言語の文法や機能の拡張を行っていないため専用の処理系などは用意しておらず，通常の Ruby インタプリタで回路を合成することが可能である．

記述の検証を行うために，記述通りのプログラムとして実行するためのライブラリも提供されている．通常のソフトウェアとしてプログラムを実行する場合には，Mulvery Library の代わりに Mulvery Library For Soft を利用する．このライブラリは記述通りの処理や計算を行うようにプログラムされており，回路合成や，リソース消費量や回路のパフォーマンスの解析などを行うことはできない．

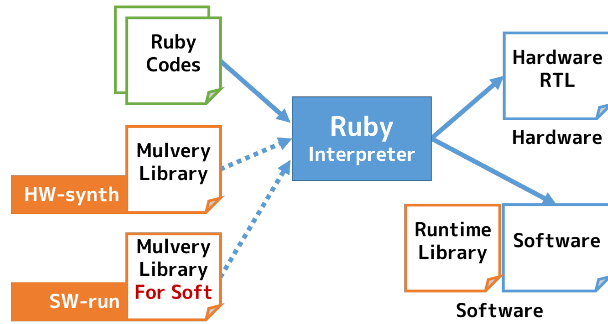


図 5.3: Software architecture of Mulvery framework

```
1: class Mulvery < Mv::MulveryBase
2:   def build_dataflow()
3:     ...
4:   end
5:
6:   def main()
7:     ...
8:   end
9: end
```

図 5.4: Template of MulveryBase class

5.2.2 Mulvery フレームワークを用いたシステム開発の手順

Mulvery では，フレームワーク開発の簡単のために Rx を用いた記述とそうでない記述を分離する方式を取る．図 5.4 に示すように，MulveryBase クラスを実装する形でシステム開発を行う．

MulveryBase クラスを用いず明示的なハードウェア・ソフトウェア記述の分離を行わないためには，ソースコードの抽象構文木を解析すればよい．Mv::Observable クラスのインスタンス化を検出することでハードウェア記述の開始点を抽出することができる．

アプリケーション開発例

サンプルとして，1 秒ごとに 0 から 10 までの数字と，それを 2 倍した値を出力するハードウェアを作成し，ソフトウェアから読み出すアプリケーションを開発しながら具体的な開発手順を説明する．

データフローの定義　データフローの定義は，MulveryBase.build_dataflow にて行う．Mulbery-Base がインスタンス化される際に実行され，ハードウェア化の際には Verilog HDL のソースコードを生成し，ソフトウェア実行の際には Rx ライブラリが呼び出され実行される．インスタンス変数に設定された Observable は，ソフトウェア側から読み出しができるように CPU-FPGA 間の共有メモリに領域が確保され，イベントの到着と同時に共有メモリにイベントの内容が書き込まれる．

図 5.5 は build_dataflow メソッドの記述の例である．タイマとデータを用意し，データフローを記述する．zip メソッドは 2 つの Observable のイベントを待ち合わせる性質を持っているため，_numbers に流れるイベントが 1 秒ずつ出力されるようになる．@numbers2 では，@numbers から 2 の倍数のみを取り出す操作を行うため，イベントは 2 秒ごとに出力される．

```
1: def build_dataflow()
2:   now_time = Mv::Observable.interval(1)
3:   data = (0..10).to_a()
4:
5:   _numbers = Mv::Observable.from_array(data)
6:   @numbers = Mv::Observable.zip(time, _numbers)
7:   @numbers2 = @numbers.select(){ |t, e| e % 2 == 0 }
8: end
```

図 5.5: Example of build_dataflow method

```
1: def main()
2:   @numbers.subscribe() do |e|
3:     print("numbers : #{e}")
4:   end
5:   @numbers2.subscribe() do |e|
6:     print("numbers2 : #{e}")
7:   end
8:   sleep(100)
9: end
```

```
numbers : [0, 0]
numbers2 : [0, 0]
numbers : [1, 1]
numbers : [2, 2]
numbers2 : [2, 2]
numbers : [3, 3]
numbers : [4, 4]
numbers2 : [4, 4]
...
```

図 5.6: Example of main function and its outputs

プログラムの記述　CPU 側で実行されるコードは, MulveryBase.main に記述する．build_dataflow でインスタンス変数として定義された Observable にアクセスすることができる．イベントがアップデートされる度に通知を受けとる際には subscribe メソッドを用い，ある瞬間の最新のイベントを読み出したい場合には last メソッドを用いるなどして値を読み出すことができる．

図 5.6 は main メソッドの記述の例（左）と出力（右）である．インスタンス変数に設定された Observable はソフトウェア側から呼び出すことができる．numbers の出力は 1 秒ごとに行われ, numbers2 の出力は 2 秒ごとに出力される．出力されているタプルの 1 つ目の値は経過秒数で，2 つ目の値は data の内容である．

5.3 Functional reactive programming

この節では, Mulvery ライブラリの API が用いているプログラミングモデル：functional reactive programming について説明する．まず初めに reactive programming の概要について説明し，その後 functional reactive programming の説明と Mulvery への応用の意味について説明する．

5.3.1 Reactive Programming

GUI などの, アプリケーション自身やその外部環境からの様々な種類のイベントによって駆動するアプリケーションの開発に適したプログラミングのパラダイムとして，**Reactive Programming**（以降 RP とよぶ）が研究されている [88]．RP では，Observable と呼ばれる，時間の流れとともに値が変化するオブジェクトに対して宣言的に操作を記述することでプログラムを記述する．命令型のプログラミングモデルでは割り込みなどを用いて変数の値を更新することで動的な値を扱う必要がある．RP では，オブジェクトの値が時間とともに直接変化するため，プログラミングの際に割り込みやデータ更新を考慮する必要がない．

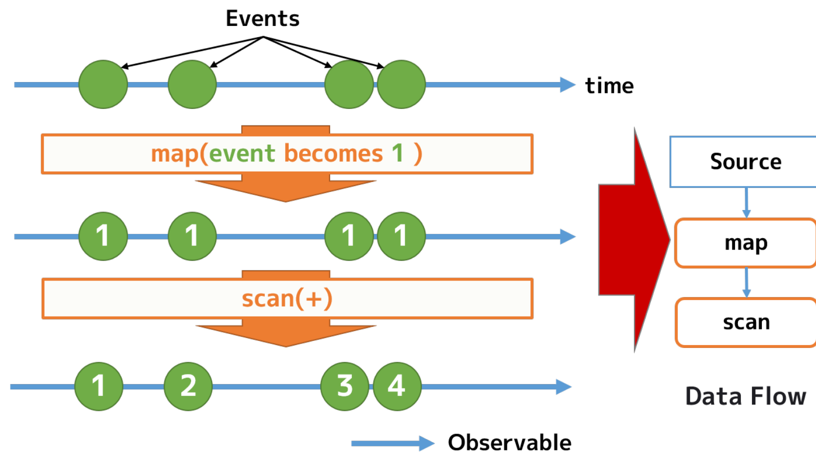


図 5.7: Example of counting the number of events using Reactive Programming

図 5.7 にイベントの発生回数を数える例を示す．Reactive Programming によってイベントの発生回数を数える例．Observable に流れてきたイベントに対して map や scan などのオペレータを適用することで，データフロー型のプログラミングモデルを実現する．

このような記述からは，データフローグラフが容易に構築することができる．これによって，タスク並列や時間並列が可能な部分の抽出が容易なものとなる．センサからのデータの処理やアクチュエータの駆動なども同様の性質を持つことから，ハードウェア合成のために RP のパラダイムを導入するものとした．本提案では，ベースとなる RP の実装として，Microsoft 社の LINQ をもとに作られた ReactiveX[89][90]（以降 Rx とよぶ）を用いる．既に複数の企業でも利用されている共通のスキームであるため，標準的なオペレータセットであるといえる．

この節では，Rx を理解するために重要な Observer Pattern，LINQ，Scheduler の概念について説明する．また，それぞれがどのようなハードウェアに合成されるかを説明する．

5.3.2 Observer Pattern

Observer Pattern は，Subject と呼ばれるイベントを発生させるオブジェクトと，Observer と呼ばれるイベント発生の通知を受け取るクラスによって構成される．Observable パターンのクラス図を図 5.8 に示す．Subject は Observer に通知を行うための notify メソッド，通知を受け取る Observer を追加/削除する Subject には複数の Observable を関連付けることができる．Subject は Observer の追加/削除と，Observer へのイベントの通知を行うメソッドを具する．また，Observer は event の発生時にコールバックされるメソッドを具する．

Observer Pattern を用いた記述は，主に GUI 設計等におけるイベント駆動型のプログラミングモデルに用いられる．たとえば，マウスイベントを処理する例は図 5.9 のように記述できる．マウスイベントが発生すると，登録されている全ての Observer の onNotify を呼び出す（左）．onNotify メソッドを実装した ObserverBase のサブクラスを実装し，Mouse クラスに登録する（右）．

Rx における Observable クラスは，**Observer Pattern** [91] と呼ばれるデザインパターンに基づいて実装される（図 5.8）．

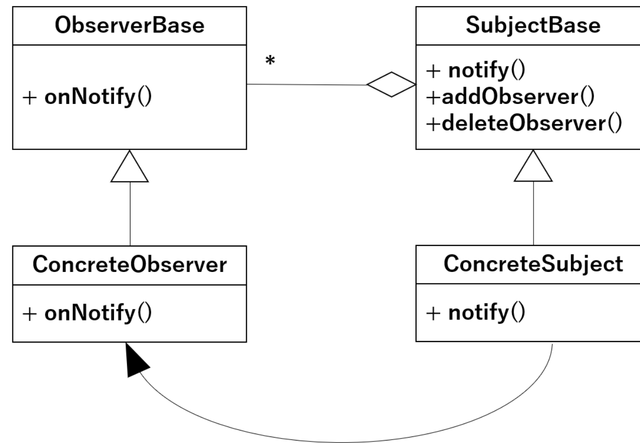


図 5.8: Class diagram of Observable Pattern

```
1: class Mouse < SubjectBase
2:   include Singleton
3:   def notify(event)
4:     @observers.each() do |observer|
5:       observer.onNotify(event)
6:     end
7:   end
8: end
```

```
1: class OnMouse < ObserverBase
2:   def onNotify(event)
3:     do_something()
4:   end
5: end
6:
7: Mouse.instance
8:      .addObserver(OnMouse.new())
```

図 5.9: Example of Subject and Observer for mouce events

5.3.3 Functional reactive programming

RP はイベント駆動をプログラミングモデルに落とし込んだものである．対して回路はイベント駆動型のモデルではなく，モジュール間のデータバスには常に信号が流れ続けている．Verilog HDL では，Always 文と呼ばれる文法を用い，組み合わせ回路を表現する．図 5.1 に示した Always 文の例は，入力 `data_in` の値が変わったときその値に対応する値を `data_out` に出力するデコーダの例である．このように，イベントを伝えるのではなく，モジュール間には常にデータが流れておりその値の変化をイベントとして捉えるプログラミングモデルを Functional reactive programming (FRP) と呼ぶ [92], [93].

筆者は，このプログラミングモデルは回路記述言語の記述モデルと共通したモデルであり，回路設計にマッチしたプログラミングパラダイムであると考えた．したがって，このようなソフトウェア開発とハードウェア設計に共通のプログラミングモデルに着目し，FRP を用いたプログラミングモデルを用いて Mulvery ライブラリの API 設計をすることとした．ベースとなる API とそのプログラミングモデルは RP に基づく API である Rx を参考にしているが，Mulvery ではこれらの API を FRP に適した形へと改変を加えて用いている．

5.3.4 基本的な文法と動作

Mulvery によるプログラムの文は，大まかに Listing 5.2 に示すような構造を取る．

たとえば，図 5.11 のプログラム `count = events.map(){ |event| 1 }.scan(){ |acc, x| acc + x }` では，`events` が observable に相当し，`map(){...}` および `scan(){...}` が operation に相当する．Ruby によって右辺を評価した結果は具体的な値ではなく observable object となるため，変数

Listing 5.1: Always statement in Verilog HDL

```
1 always @(data_in) begin
2   case (data_in)
3     2'b00: data_out = 8'b1011_0100;
4     2'b01: data_out = 8'b1101_0001;
5     2'b10: data_out = 8'b0001_0000;
6     2'b11: data_out = 8'b1101_1001;
7   endcase
8 end
```

Listing 5.2: Definition of statement of Mulvery API

```
1 <observable> ::= <observable object> | <observable statement>
2 <observable statement> ::= <observable>.<operation chain>
3 <operation chain> ::= <operation> | <operation>.<operation chain>
4 <operation> ::= <operator>(<operator args>) <lambda abstraction>
5 <lambda abstraction> ::= { |<variable>| <ruby program> }
```

`count` は observable object を格納する．operator の多くは高階関数であり, operator には operator 自身が受け取る引数 operator agrs のほか，ラムダ抽象 lambda abstraction を受け取る（4 行目）．ラムダ抽象の実装は Mulvery の提供するデーター型の API ではなく，通常の Ruby プログラを記述する．Ruby においては，ラムダ抽象に具体的な値を適用・評価した結果は，最後の文の評価結果であり，すなわち図 5.11 のプログラムでは，`map` に渡されたラムダ抽象は常に 1 を返し，`scan` に渡されたラムダ抽象は `acc + x` を返す．

`<observable>.<operator>` の形の記述部分において,「`observable` は `operator` のレシーバ」という．observable がイベントを送出すると，その observable をレシーバとする全てのオペレータがそのイベントを受け取る．オペレータは，オペレータ自身の定義とラムダ抽象の適用の評価結果に基づいて新たなイベントを生成し，そのオペレータをレシーバとする次のオペレータに対してイベントを送出する．つまり，図 5.11 では `map` によって全てのイベントは値 1 を持つイベントに変換し，それを次のオペレータ `scan` に送出する．

5.4 レジスタ転送レベル設計の合成

動的型付け言語のプログラムは実行時まで変数の型を決定することができないため，HLS ツールはバス幅やレジスタの大きさを決定することができず，ハードウェアを合成することができない．FRP がデータフロー型の記述であるという特徴を用い，Mulvery は動的型付け言語のプログラムからデータフローグラフを構築する．Mulvery は，Mulvery ライブラリのオペレータをそれぞれデータフローグラフのノードに変換する．

Mulvery ライブラリを用いて記述されたプログラムは専用の処理系で処理されるわけではなく，通常の Ruby インタプリタ上で実行される．つまり，Mulvery ライブラリで記述されたプログラムを読み込んだ Ruby のインタプリタは実際の計算を実行する代わりに，データフローグラフを生成する．

LINQ

LINQ は，**Language-INtegrated Query** の略であり，Microsoft 社によって C#言語の機能として開発されたクエリの名称である [94]．データ列を操作するためのクエリを標準化しプログ

```
1: var query = from x in list
2:             where x % 2 == 0
3:             orderby x
4:             select x * 3
```

```
1: var query = list
2:            .Where(x => x % 2 == 0)
3:            .Orderby(x => x)
4:            .Select(x => x * 3)
```

図 5.10: Example description of LINQ on C#

```
1: events = Rx::Observable... # Observableの生成
2: count = events
3:          .map(){ |event| 1 }
4:          .scan(){ |acc, x| acc + x }
```

図 5.11: Example of event counting using RxRuby

ラミング言語上に実装することで，これまで文字列などで扱い間接的に実行していたSQLなどのクエリ言語を隠ぺいして扱えるようにすることを主な目的としたフレームワークである．特徴として，各クエリはラムダ抽象を受け取る高階関数として実装されていることが挙げられる．このような形式にすることによって，イベントのデータ型によらず共通のクエリを使って表現することができるようになる．これらは実装する言語の構文で記述されることが多いが，C#言語ではクエリ言語様の構文も用意されている（図5.10）．

Rxにおいても，Observableに流れるデータを操作するため，LINQクエリをRxオペレータに取り入れている．言い換えると，Reactive ProgrammingのためのLINQ拡張ライブラリと言うことができる．図5.7をRxRubyを用いて実装した例を図5.11に示す．イベントの発生回数を数える例のRxRubyを用いた実装例．Observableに対するOperatorのメソッドチェーンによってデータフローを表現することができる．

Scheduler

Rxでは，マルチスレッドプログラミングを実現するために**Scheduler**と呼ばれる仕組みが用意されている．通常Rxのオペレータはプログラムが動作するスレッド上で動作するが，一部のRxオペレータや明示的に指定した場合には，異なるスレッドでの動作や割り込みのような動作が実現できる．実際にこのような制御を容易に実現するために，RxRubyでは表5.1のような実装されたSchedulerが用意されている．

図5.12にオペレータ，スレッド，スケジューラの間の関係性を示す．LocalSchedulerはキューを持ち，その先頭から順にタスクを処理していく．LocalSchedulerのインスタンスはそれぞれ独立したスレッド上で実行され，これによって並列性を実現している．

これらのSchedulerは，オペレータの引数として指定することができる．多くのオペレータはCurrentThreadSchedulerをデフォルト引数としてとるが，timerオペレータなどはスレッドを長時間ブロックしてしまうため，DefaultThreadSchedulerをデフォルト引数として持つ．

図5.13は，timerオペレータを例に挙げてSchedulerによる動作の違いを示したものである．timerは通常並列に動作するためのSchedulerに登録される（上）．明示的に複数のtimerを同じSchedulerに登録すると処理が逐次的に実行される形になる（下）．

表 5.1: Schedulers implemented on RxRuby

Scheduler 名	機能
CurrentThreadScheduler	現在のスレッド上で処理を逐次実行する Scheduler
ImmidiateScheduler	現在のスレッド上で即座に処理を開始する Scheduler
DefaultScheduler	バックグラウンドスレッドで処理するための Scheduler
LocalScheduler	new すると新しいスレッドが生成されその上で処理が行われる

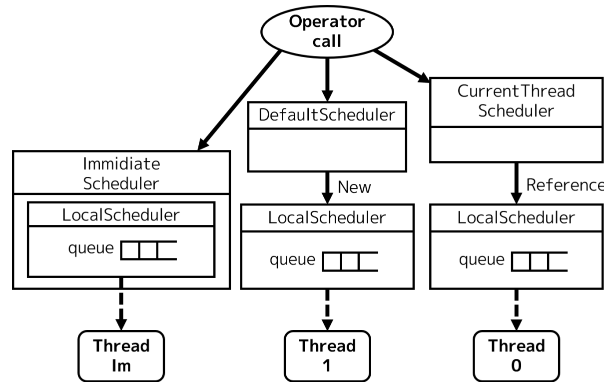


図 5.12: Relationship among the operators, threads, and principal schedulers

5.4.1 RxRuby による記述からハードウェアを合成する手法

本研究は，Rx による記述がデータフローの記述とほぼ等価であることに着目し，動的型付け言語による記述からデータフローグラフの抽出とハードウェアの合成を行うものである．動的型付け言語においては実行時までデータ型が決定しないため，他の合成系のように静的解析によって合成することができない．RxRuby による記述からハードウェアを合成するために考慮すべき点を整理すると，

1. データフローの抽出とハードウェア生成
2. 高階関数に渡すラムダ抽象のハードウェア合成

の 2 点が挙げられる．

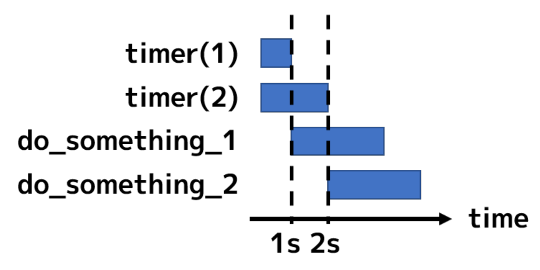
この節では，これらの考慮すべき点について，動的型付けの特徴を利用して実現する手法について説明する．

データフローの抽出

Rx による記述は，イベントのデータ型によらず，全て Observable 型に対するオペレータの呼び出しの形で記述される．そこで，ハードウェアを合成する際には，この Observable 型を工夫して Multi-stage Programming [95] の手法をとることでメタプログラミング的にハードウェア記述を生成することができる．

DFG は Observable と DataflowOperator の組み合わせで構成される．Listing 5.3 は Observable クラスの実装の一部である．`from_array` オペレータは DataflowOperator クラスのインスタンスをタスクとしてスケジューラに登録する．このときラムダ抽象も DataflowOperator に記録される（図 5.14）．すべての DataflowOperator ソフトウェア実行時にはスケジューラによってすぐに実行

```
1: Rx::Observable
2:    .timer(1)
3:    .subscribe(){ do_something_1 }
4:
5: Rx::Observable
6:    .timer(2)
7:    .subscribe(){ do_something_2 }
```




```
1: scheduler = Rx::LocalScheduler.new()
2:
3: Rx::Observable
4:    .timer(1, scheduler=scheduler)
5:    .subscribe(){ do_something_1 }
6:
7: Rx::Observable
8:    .timer(2, scheduler=scheduler)
9:    .subscribe(){ do_something_2 }
```

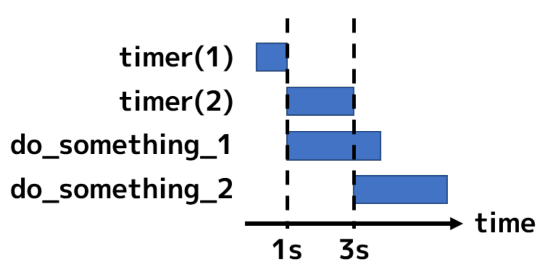


図 5.13: Examples of Scedulers on RxRuby

Listing 5.3: An implementation of the *from_array* operator in Mulvery

```
1  module Observable
2    class << self
3      def from_array(array, \\
4          scheduler=LocalScheduler.new)
5        operator = \\
6            DataflowOperator.new(:from_array)
7
8        scheduler.schedule(operator)
9      }
10     ...
```

が始まるが，ハードウェア合成の際には実行する代わりに DataflowContainer に記録され，次のステップの解析で利用される．

データフローから回路を合成

DataflowOperator から HDL で記述されたモジュールが生成されるまでのフローの概略図を図 5.15 に示す．Operator type ごとに基本となる実装が HDL のテンプレートとして用意されており，ユーザが定義するラムダ抽象の合成結果をこのテンプレートに挿入することで HDL のモジュールが生成される．

ハードウェアを合成するための Observable 型は，図 5.16 のようなコードで定義される．2 行目からは，引数に渡された数だけイベントをバッファするクエリ：buffer を定義している．4 行目の BUFFER_HDL は，バッファを実装した Verilog HDL のコードのテンプレートが含まれており，Ruby 標準のテンプレートエンジン：ERB を用いてバッファする大きさに合わせて buffer のハードウェアを生成している．7 行目では，トップモジュールとなるテンプレートに，それまでに追加されたモジュールの HDL を追加することでトップモジュールを生成している．

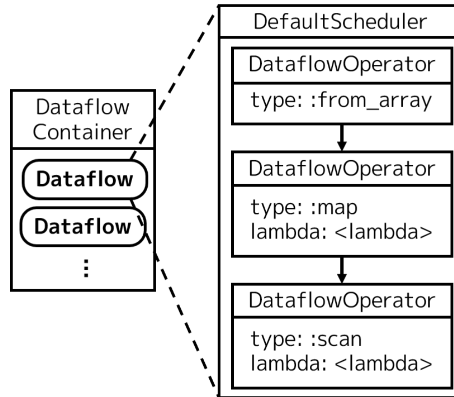


図 5.14: Structure of a dataflow and its container

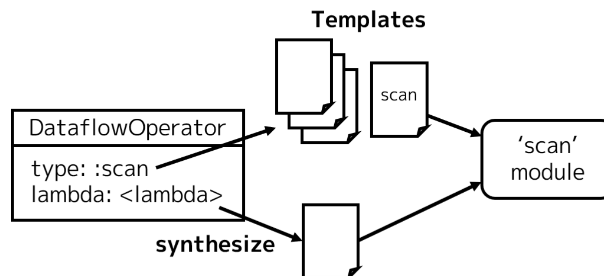


図 5.15: Workflow that generates the HDL code of a module

ラムダ抽象の合成

map や scan に代表されるような高階関数となるオペレータは，イベントを引数にとるラムダ抽象を引数にとる．たとえば，到着したイベントを倍にして返すコードは，次のようなものとなる．

```
sum = observable.map(){ |event|
  event * 2
}
```

Rx におけるラムダ抽象は，オペレータ毎に目的が決まっているため，多くの場合イベントをどのように操作するかの宣言的な記述であると考えられる．したがって，ラムダ抽象の引数に対してストリームの合成の場合と同様に，マクロのようにふるまうオブジェクトを渡すことでハードウェアの合成を行うものとした．つまり，map オペレータの定義は次のようなものとなる．

```
def map()
  process = yield(HDLComposer.new())
  self.append_module( \\
      ERB.new(MAP_HDL).result(binding))
end
```

オブジェクト指向型言語では，演算子をオーバーロードすることができる．これをうまく利用して Multi-stage programming を行う HDLComposer を定義できる（図 5.17）．

ただしこの手法では，Ruby の持つ構文を正しく合成することができない．つまり，if 文や for 文を合成するには少し工夫する必要がある．このために，ハードウェアを合成するための特殊なメソッドを用意することによるサポートが考えられる．if 文の例を図 5.18 に示す．mv_if メソッドは if 文を実現するためのオブジェクトを返す．このオブジェクトには elsif メソッドや else メソッ

```
 1: class Observable
 2:   def buffer(num)
 3:     self.append_module_hdl( \\
 4:         ERB.new(BUFFER_HDL).result(binding))
 5:   end
 6:   ...
 7:   def generate_hdl()
 8:     modules = @modules
 9:     return ERB.new(TOP_HDL).result(binding)
10:   end
11:   ...
```

図 5.16: Example of Observable

```
class HDLComposer
  def *(val)
    case val_type
    when :numeric then
      return <<EOS
result = mv_val_#{@id} * #{val};
EOS
    ...
  end
  ...
```

図 5.17: Example of object that synthesizes a lambda abstraction

ドが含まれており，ハードウェアを合成するためにすべての condition で HDLComposer を通過させることができる．ソフトウェアとして実行する場合には，通常の if 文と等価なふるまいをするように実装すればよい．

このラムダ抽象の合成はいくつかの前提に基づいて実装されているため，合成できない場合が残っている．ラムダ抽象の合成についてはこれまでの通常の高位合成と同じ手法を取ることも可能であるが，本提案では行っていない．なるべくラムダ抽象の内容をシンプルにして，RP 記述を工夫することで，よりハードウェアに適したプログラムになると考えられる．

Scheduler の合成

Mulvery フレームワークでは，FPGA の並列処理性能をより活かすために，Scheduler の運用方法を RxRuby の実装と少し異なるものとしている．5.4 項でも述べたように，オペレータの多くは CurrentThreadScheduler をスケジューラとして用いるが，この実装ではタスク並列となっている処理同士が逐次的に実行されてしまう．そこで Mulvery フレームワークでは，一部のオペレータのデフォルトの Scheduler を DefaultScheduler とした．

Rx オペレータは，表 5.2 に示すように分類することができる．Creation 系のオペレータ群はデータフローの先頭にくるため, これらについて DefaultScheduler を用いてスケジューリングすることによって生成されるハードウェアの並列度を向上させることが可能となる．Timer 系オペレータは時間計測を行うため RxRuby でも DefaultScheduler を用いて新たなスレッドを生成して実行される．Mulvery フレームワークにおいてもこのままとした．データ操作系オペレータは直前のオペレータが実行されるスレッドで実行される場合が多いため，これまでと同様に CurrentThreadScheduler を用いたスケジューリングを行う．

```
def func(t)
  mv_if(value, ->v{v == 1}){
    value * 1
  }
  .elsif(->v{v == 2}){
    value * 2
  }
  .else(){
    value * 3
  }
  .endcheck
end
```

図 5.18: If method on Mulvery

表 5.2: Classification of Rx operators

Creation	Timer	Data manipulation
of_array	timer	select
generate	interval	map
empty		average

図 5.13 下段の例のように，LocalScheduler によって本来並列な処理のシリアライズを行う場合においては，LocalScheduler が空の Observable を用意し，逐次実行されるようなデータフローを構築することで実現する．

5.5 処理の記述の透過性

これまでのハードウェア・ソフトウェアコデザイン環境では，ハードウェアオフロードする対象のコード片を明示的に指定する必要があった（図 5.19）．SDSoC では，C 言語で記述されたプログラム中の関数に対して Toggle HW を指定することでハードウェアにオフロードされる（上）．Intel FPGA SDK for OpenCL では，OpenCL C 言語の__kernel 指定子で指定されたカーネル関数がハードウェアとしてオフロードされる．

この場合の問題点として，指定されたメソッドを呼び出す場合には必ず CPU と FPGA の間でのデータの受け渡しが発生するため，プログラムを十分に解析しないまま不適切なオフロードを行うとパフォーマンスが低下する場合があることが挙げられる．

これに対して，本論文で提案する手法を用いる場合，同じメソッドの呼び出しでも，CPU と FPGA 間でデータに近い場所での実行が実現できるメリットが得られる．その例を図 5.20 に示す．6 行目の func は FPGA で実行されるが，9 行目の func は CPU 上で実行される．

提案手法では，オペレータをハードウェアに合成する際に渡されたラムダ抽象の内容も同時に合成してオペレータ内部に組み込む．このため，あるメソッドが一度ハードウェア化されていたとしても，ソフトウェアとして実行される部分において呼び出された場合にはハードウェアへの受け渡しは発生せず，ソフトウェアとして実行される．

ラムダ抽象に渡されるデータは個々のイベントデータであるため，データサイズが小さいことが多い．小さなデータを繰り返し CPU-FPGA 間でやりとりする場合にはシステム全体のパフォーマンスが著しく低下することが知られており，この問題を避けることができる性質は合成されるシステムにとって大きなメリットとなる．

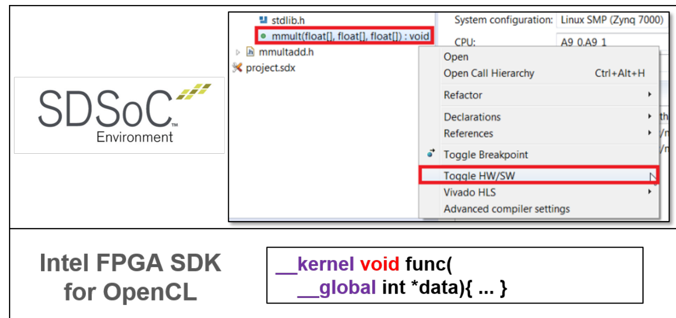


図 5.19: How to specify hardware offload in SDSoC and Intel FPGA SDK for OpenCL

```
1: def func(data)
2:   ...
3: end
4:
5: stream_1 = Stream.new()
6: grouped_1 = stream_1.group_by{|d| func(d)}
7:
8: array_1 = [1, 2, 3, 4, 5]
9: grouped_2 = array_1.group_by{|d| func(d)}
```

図 5.20: Example when the location where the method is executed differs between CPU and FPGA depending on how it is called

これらのような透過性の高い記述は，動的型付け言語であるために得られる利点である．

5.6 ハードウェア合成手法の性能評価と予備実験

5.6.1 サンプルプログラム

性能を評価するために，畳み込み演算を例にハードウェアの合成を試みた．図 5.21 に示すように，128×128 の画像に対して 5×5 の大きさのラプラシアンフィルタを適用するプログラムを用いた．このプログラムの一部を図 5.22 に示す．このプログラムでは，イベントとして画像の 1 行（128 画素）を受け取る．これを 5 行バッファすることで 5×5 のフィルタを 128 回適用できるようになるため，map を用いて畳み込みを行うようにした（図 5.23）．Matrix クラスは標準のクラスであるが，ハードウェア合成のためにオーバーライドした実装となっている．また，畳み込み演算をするための conv メソッドが追加されている．

5.6.2 ハードウェアの合成

合成の結果，図 5.24 に示すようなパイプラインが出力された．Mulvery では，トップモジュールで各オペレータのハードウェアのインスタンス化を行い，図 5.24 に示したパイプラインのようにモジュール接続をおこなう．

このプログラムでは，畳み込み演算を行う map メソッドに渡されるラムダ抽象が複雑なものであったが，合成することができた．この合成の結果の模式図を図 5.25 に示す．図中 data で示す 5×128 のレジスタ配列を 5×5 に区切り，図中 mats に示す $(5 \times 5) \times 128$ の大きなバスとして取

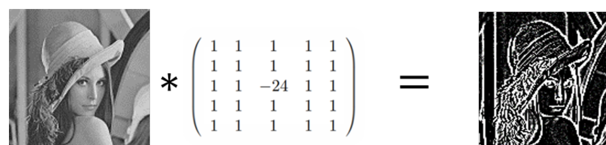


図 5.21: Applying Laplacian filter

```
 1: buffer = input.shift_buffer(5)
 2: line = buffer.map(){ |data|
 3:   mats = Array.new()
 4:   (0...SIZE).each { |i|
 5:     mats.push(Matrix[data[0][i, 5], \
 6:                      data[1][i, 5], \
 7:                      data[2][i, 5], \
 8:                      data[3][i, 5], \
 9:                      data[4][i, 5]])
10:   }
11:   result_row = Array.new()
12:   mats.map() { |mat|
13:     result_row.push(mat.conv(kernel))
14:   }
15: }
16: result = line.buffer(128)
```

図 5.22: Example of folding

り出す．mats の各要素は 128 個の独立した畳み込み演算器それぞれに接続され，別のレジスタに用意されている kernel と畳み込まれる．

5.6.3 ハードウェアの評価

・動作の確認

Icarus Verilog を用いて合成されたハードウェアの動作確認をおこなった（図 5.26）．実際には，5×128 の畳み込み演算は 1 クロックで行われていた．手動のチューニングを行うこともアーキテクチャ空間の探索を行うこともなく，効果的なハードウェアを生成することができている．図中では最初の畳み込みの遅延が 2 クロックかかっているが，クロックの立ち上がりタイミングの影響で最初の信号のみを読み飛ばしてしまっていた．

・リソース消費量

生成されたハードウェアを Xilinx 社の SoC FPGA である Zynq-7000 プラットフォームに配置配線した場合の結果を確認した結果を図 5.27 に示す．小さなリソース消費量で回路を実装することができていた．

・ソフトウェア実行時との比較

最後に，CPU で実行した場合との処理速度の比較を行った．Intel Core-i3 @ 3.4GHz でソフトウェア実行し，CentOS7 上の time コマンドを用いて CPU 時間を測定した．その結果，画像一枚当たりの処理に，CPU 時間で 500ms の時間を要していた．チューニングをくわえていない自作のライブラリを Ruby 上で扱ったため，長い処理時間を要していると考えられる．

対して合成されたハードウェアは，1 枚の画像あたり 135 クロック程度を消費している．動作周波数が 100MHz であることから，処理速度自体は 194GB/s 近い値となっている．しかしながら，

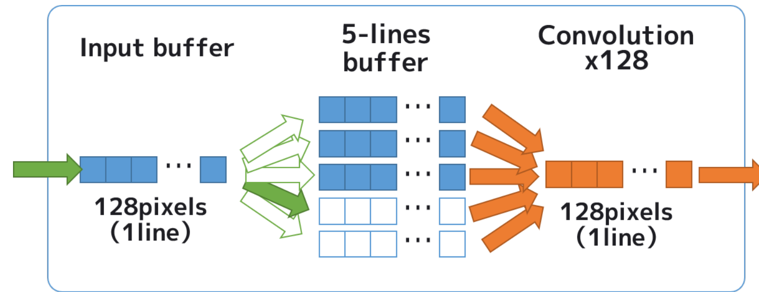


図 5.23: Architecture of the program

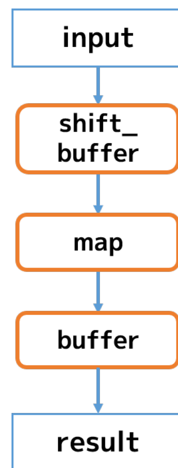


図 5.24: Pipeline of Fig. 5.22

メモリや外部 I/O からこの量のデータを入力するのは困難である．Zynq アーキテクチャにおいてはメモリ読み出しの実測値がおよそ 1.6GB/s 程度であることから，データバスがボトルネックになることが予測できる．メモリを経由してデータ転送する場合には，最大の処理速度としても画像一枚あたり 0.2ms 程度であると予測される．

データバスのボトルネックの問題があるものの，明示的な回路合成のチューニングのフェーズを省略しながらも CPU での処理と比較して効率のよいハードウェアを生成することが可能であることがわかった．

5.7 ハードウェア・ソフトウェアの分割とシステムへの統合

5.7.1 ハードウェア・ソフトウェアの分割手法

Rx を用いた記述を行う場合でも，プログラムのすべての部分が Observable の操作の記述となるわけではない．多くの場合には，データストリームの記述とそうでない部分が混在する．例えば，イベントの回数を表示するようなプログラムを作る場合，図 5.28 のような例が考えられる．Rx による記述 (L1-2) はハードウェアオフロードされ，Observable からイベントを取り出して処理する記述 (L4-5) はソフトウェアとして実行される．この例では，print 関数は Observable そのものではなく，Observable に流れるデータの最新のものを取り出しており，Observable に対する操作の記述ではない．

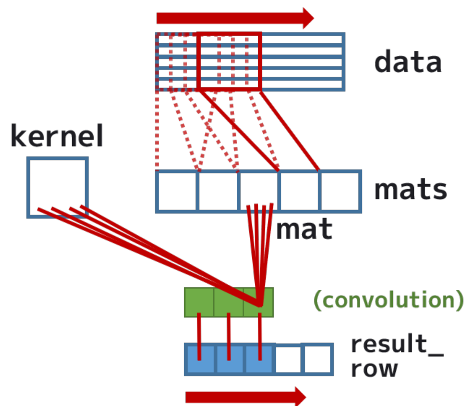


図 5.25: Synthesized lambda abstraction in Fig. 5.22

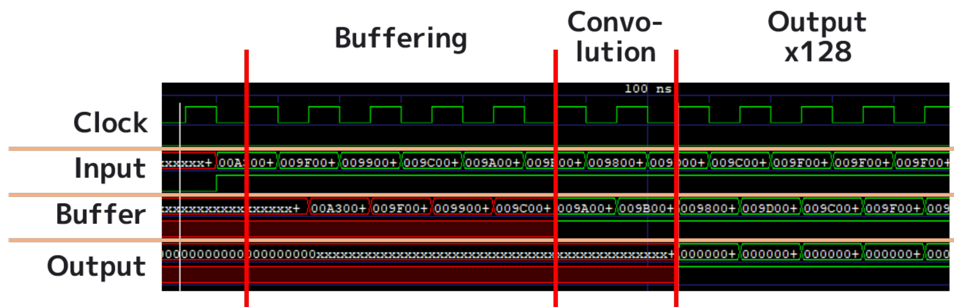


図 5.26: Hardware verification using Icarus Verilog

そこで本提案では，ハードウェア・ソフトウェア分割の戦略として，Rx で記述された部分をハードウェアとし，Rx で記述しにくい（できない）部分をソフトウェアとして分割するアプローチをとる．データフローモデルとして記述しやすい処理は FPGA による処理に適し，記述しにくい処理は FPGA による処理には適さないといえる．したがってこの手法は，機械的に処理可能でありながらも効果的なソフトウェア・ハードウェア分割であることが期待できる．

5.7.2　ハードウェア・ソフトウェアの連携手法

CoRAM アーキテクチャの導入

フレキシビリティの高いアーキテクチャを実現するため，Mulvery の生成する IP コアは CoRAM アーキテクチャ [74] の形式をとる．CoRAM アーキテクチャでは，図 5.29 に示すように合成されたハードウェアは CoRAM と呼ばれる RAM および Control Thread と接続する FIFO を両方向に持つ．Control Thread と呼ばれるデータフローを制御するモジュールによって CoRAM 間や CoRAM-DRAM 間のデータ転送が行われる．

データフロー

FPGA から CPU へのデータの受け渡し

生成されたハードウェアは，イベントの内容を CoRAM へ書き込む．Memory Access Controller

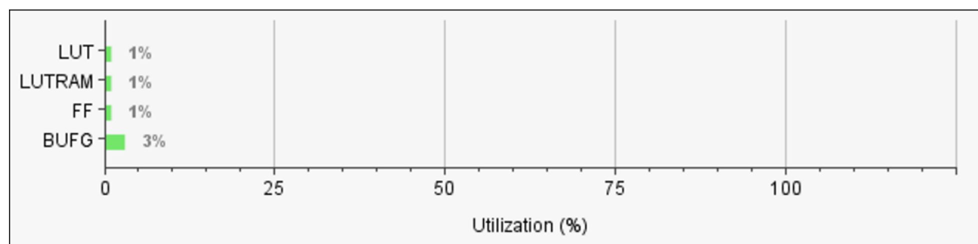


図 5.27: Resource consumption

```
1: events = Observable.new()
2: num_events = events.map(event becomes 1).scan(+)
3:
4: def show_num_of_events()
5:   print num_events.last()
```

図 5.28: Pseudo code for counting the number of events

はCoRAMのデータを，CPUが自由にアクセスできるDRAMの領域と同期し，共有メモリのように見せることでメモリマップドI/Oのようなインタフェースを提供している．書き込まれるイベントやタイミングについての詳細は5.2.2項で説明する．

CPU-FPGA 間の同期

CPU-FPGA間には，Memory Access Controllerを通したFIFOが用意されている．このためハードウェア側はCPUとの数ワード程度の小さなデータのやりとりや，同期的なデータの転送を行うことができる．ハードウェアとCPUはこのFIFOを通してメモリ読み出しのタイミングの通知し，同時に読み出し開始アドレスおよび読み出し長を通知する．ハードウェアは，これらの情報からDRAMに対してDMA (Direct Memory Access) を行い，データの書き込み・読み出しを行う．

5.8 アプリケーションの開発例とその評価

この章では，アクチュエータを扱うデバイス開発を通してMulveryフレームワークの性能の評価を行う．ハードウェアによるLEDマトリクスの点灯制御と，ソフトウェアによるLEDマトリクスの表示内容の制御を行うシステムを開発し，生成されたハードウェアの性能について評価を行う．

5.8.1 フルカラー LED の制御とアプリケーションソフトウェア

Mulveryフレームワークの性能評価のために，Worldsemi社のフルカラーLEDであるWS2812B (以降，NeoPixelとよぶ）を用いたLEDマトリクスの制御システムの設計を行った．このシステムでは，LEDマトリクスで出力する内容をソフトウェアから制御する．

NeoPixel の制御方式と予備評価

NeoPixelの制御信号 NeoPixelは, シリアル信号によってRGB信号を転送する．１つのNeoPixelが受け取るデータは，R,G,Bそれぞれ8bitの計24bitのデータであり，GRBの順に転送される

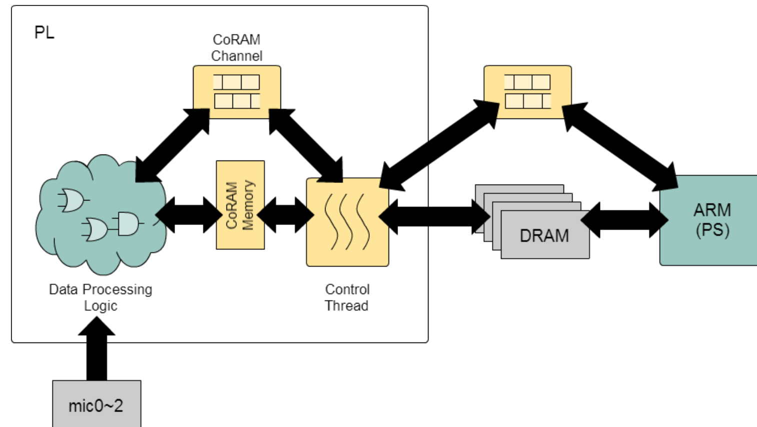


図 5.29: Architecture with CoRAM

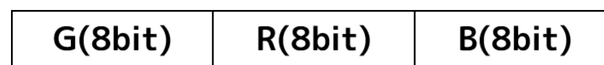


図 5.30: Data field for NeoPixel [96]

(図 5.30)．同期のためのクロック信号線を持たず，1 本の信号線のみでデータを転送する．このため，1.25μs の間の HI と LOW の時間の比で転送データが 0 なのか 1 なのかを区別する方式をとっている（図 5.31，表 5.3)．

この実験では，NeoPixel を用いて作成した 32 × 32 の大きさの LED マトリクスの制御を行う（図 5.32)．この 1024 個の NeoPixel はすべて直列に接続されている（図 5.36)．D1 から D2 にデータが転送されるとき，PIX1 は先頭の 24bit を削除して PIX2 にビット列を転送する．24 × 1024 ビットの信号を連続して出力することで，すべての NeoPixel を 1 本の信号線で出力することが可能である．

MCU による制御の検討　直列に接続した NeoPixel 全体を 30fps で制御することを考えた場合，$\frac{1}{30}\mathrm{s}/1.25\mu\mathrm{s}/24\mathrm{bits} = 1,111.1$ より，1 本の信号線あたり 1,111 個の LED の制御が理論的な限界となる．したがって，実験で使用する 32 × 32 の大きさの LED マトリクスは制御の限界に近い規模となる．MCU によって制御する場合 1 秒あたりに $1,024 \times 24 \times 2 = 49,152$ 回のタイマ割り込みを発生させる必要があり，ソフトウェアによる制御が難しい分類であるといえる．アセンブリ言語を用いベアメタルの MCU で制御することで時間に正確な制御を行うことが考えられるが，加えて開発コストが大きくなってしまうのに加え，計算リソースの多くを LED の制御に割く要求は変わらない．したがって，表示コンテンツを制御するためのソフトウェアを動作させることは困難である．

LED マトリクス制御回路と表示コンテンツ制御ソフトウェアの実装

コンテンツ制御ソフトウェアは DRAM 上の領域に表示させるコンテンツのデータを展開し，LED マトリクス制御回路は DRAM からデータを読み出して内容を LED マトリクスに出力する．LED マトリクスの制御回路の実装の一部を図 5.34 に示す．

表示コンテンツ制御ソフトウェアは Linux OS 上で動作させるものとした．これによって，Ethernet の利用や WebAPI の構築など IoT デバイスに不可欠な機能について既存のソフトウェア資

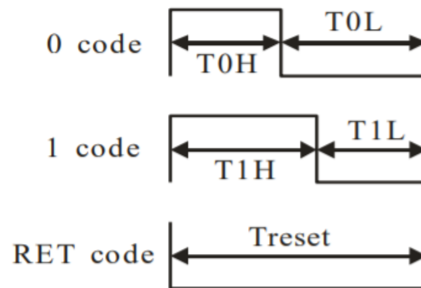


図 5.31: Signal to control NeoPixel [96]

表 5.3: Signal transmission time length and tolerance [96]

信号名	信号の長さ	許容誤差
T0H	0.4μs	±150ns
T1H	0.8μs	±150ns
T0L	0.85μs	±150ns
T1L	0.45μs	±150ns
RES	50μs	±150ns

源を利用して開発することが可能になる．

5.8.2 評価

実行環境

これまでに説明した LED マトリクスを駆動させる実行環境として，Digilent 社の Zybo[97] を使用し，筆者がビルドした Linux Kernel を用いた Debian を OS として使用した（表 5.4，表 5.5）．Zybo 上で動作する Debian にて SSH デーモンを動作させ，全ての操作は SSH 経由で行った．

データ転送に必要な CPU のリソース

データ転送の命令は，C++言語によって記述された Ruby 拡張プラグインを経由して行われる．C 言語の gettimeofday() 関数を用いて，データのメモリへの書き込みからハードウェアへのデータ転送命令が完了するまでの時間を 10 回計測した．結果，実行時間の平均は 163.8μs となった．データ転送命令のために CPU から FPGA に転送されるデータは読み出し開始アドレスと読み出し長さのみであるため，高速な実行時間となった．この命令を受け取った FPGA は DRAM に対して DMA を開始するため，CPU がデータ転送のために CPU 時間を消費することがなく，CPU の計算リソース消費を抑えることを実現している．

LED マトリクスの制御に必要な FPGA のリソース

これまでに説明した手法を用いて設計したハードウェアを合成した結果の FPGA リソース消費量を表 5.6 に示す．xc7z010 は Zynq-7000 シリーズ中でも FPGA リソースの少ないグレードのデ

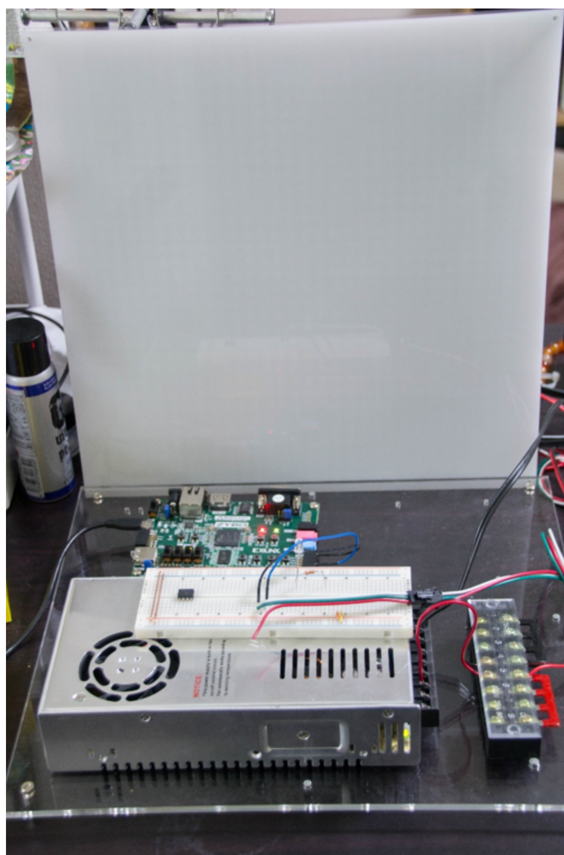


図 5.32: 32 × 32 LED matrix

表 5.4: Specifications of the board used in the experiment [97]

搭載 System on Chip	Zynq-7000 xc7z010clg400-1
CPU@動作周波数	ARM Cortex-A9 MPCore @ 800MHz (Dual Core)
FPGA@動作周波数	Artix-7 FPGA 相当 @ 100MHz

バイスであるが，2 割未満のリソース消費量でハードウェアを合成することができた．BRAM の消費が大きいのは，DRAM とのデータ共有のための共有メモリが占有しているためである．

ただし合成の結果，図 5.34 のプログラムでは 4 行目の interval と 11 行目，14 行目の timer の時間計測を同時に開始する必要があるが，このままでは同期ができないことがわかった．このため，ハードウェア生成にあたって Verilog HDL 出力の一部に手を加えている．

出力される Verilog HDL は，NeoPixel の制御の部分のみでも 300 行近く，1/10 程度の記述量で回路設計ができるのは大きなメリットであるといえる．また，PS-PL 間通信のハードウェア設計も省略できているため，実際の記述の削減量はより大きなものとなる．

5.9 結論

この章では，プログラムから自動的にハードウェア化に適した部分を抽出しオフロードする仕組みを実現するため，Reactive Programming のパラダイムを導入したフレームワーク：Mulvery を提案した．これまでハードウェア合成の困難さから使うことが高位合成ツールとして用いるこ

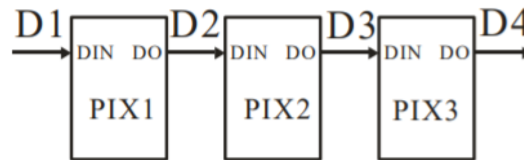


図 5.33: Serial connection of NeoPixel

```
 1: sequence = memory_to_bin(memory, 3072)
 2: Rx::Observable.interval(Mv::time::ms(33)).subscribe(){
 3:   Rx::Observable.zip(
 4:       Rx::Observable.interval(Mv::time::us(1.25),
 5:       Rx::Observable.from_array(sequence))
 6:   .subscribe(){ |no, val|
 7:       send_signal(1)
10:       check(val, ->v{ v == 0 }){
11:         Rx::Observable.timer(Mv::time::us(0.4).subscribe(){ send_signal(0) }
12:       }
13:       .else {
14:         Rx::Observable.timer(Mv::time::us(0.8).subscribe(){ send_signal(0) }
15:       }
16:       .endcheck
17:   }
20: }
```

図 5.34: Example of build_dataflow to control NexPixel matrix

とが避けられてきた動的型付け言語の性質を生かすことで，アーキテクチャ探索や明示的なチューニングを行うことなく効率のよい CPU+FPGA 環境のアプリケーションを開発することが可能となる．

Observer Pattern，LINQ，Scheduler を組み合わせた Reactive Extensions のフレームワークはデータフロー記述に近いパラダイムを持つ．この記述を用いることで，ハードウェア技術者によるチューニングを経ずともソフトウェアよりも性能の高い処理が可能なハードウェアを合成する手法を提案した．動的型付け言語を用いることで，処理の記述に高い透過性を持たせることが可能となり，CPU と FPGA の間で意図しない不必要なデータの通信を削減することも期待できる．メタプログラミングとしての技法を用いているため，Rx の各オペレータはハードウェア技術者による設計がなされた性能の高いハードウェアに合成される．このため，これまでの高位合成と比較して，ハードウェアの知識を持たない技術者でもより高い性能が発揮できることが期待できる．

予備評価および実際のアプリケーション開発事例では，CPU（MCU）のみで開発する場合と比較してより性能の高いシステムを設計・開発することが可能であることが示された．

今後の展開として，マルチ FPGA 環境における開発の支援技術の提案を挙げた．FPGA を用いた IoT デバイスが複数連携する場合の開発コストの低減や，CPU と FPGA を複数混載した分散システムにおける開発コストの低減を目指す．処理の記述の透過性が高く，FPGA で実行されるコードがそのまま CPU 上でも利用できるため，タスクやジョブの負荷の大きさに応じた計算リソースの再配分の仕組みなどの提案が期待できる．

実際の組込み環境を考えた場合，リアルタイム性の保障とハードウェア消費量の制御が課題として挙げられる．ソフトウェアでは実現できない高いリアルタイム性の保障はハードウェアによる

図 5.35: The picture used for the experiment

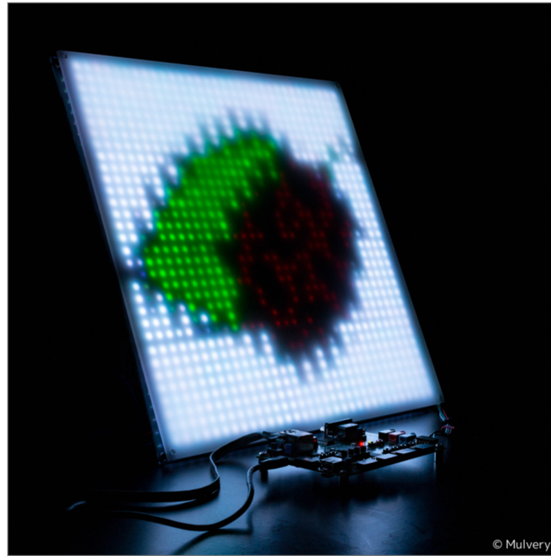


図 5.36: The NeoPixel showing Fig. 5.35

デバイス開発の利点であるといえるため，リアルタイム性の保障は重要な課題となる．また，ハードウェア消費量を削減することで，搭載可能なアプリケーションの増加のみならず，消費電力の削減なども期待することができる．

表 5.5: Software environment used for the experiment

Linux Kernel	Linux version 4.9.0-g8277d20-dirty (gcc 4.9.2) （筆者によるビルド）
ZYBO 上の gcc	g++-4.7 version 4.7.2 (Debian 4.7.2-5)

表 5.6: FPGA resource consumption

リソース	消費量	全体に対する割合 (%)
LUT	3254	18.49
LUTRAM	361	6.02
FF	3298	9.37
BRAM	6.50	10.83
IO	5	5

第6章　HLSにおける関数ポインタと高階関数のサポートとその最適化

6.1　はじめに

6.1.1　HLSと関数ポインタ

広く利用されている Verilog HDL や VHDL では抽象度の高い記述が難しくコードの再利用性の低さが課題とされており，プログラミング言語を用いてディジタル回路設計を行うメリットのひとつとして RTL 設計と比較して抽象度の高い記述を実現することが期待できる．しかしながら HDL ツールにおいても，依然高い抽象度の記述をするための課題が残されている．

C 言語は，Xilinx や Intel などの大手 FPGA ベンダが提供する HLS ツールをはじめとして，C 言語は多くの HLS ツールで記述用言語として利用されている言語である．抽象度の高い記述を阻害している例のひとつとして，C 言語において抽象的に記述するために欠かせない機能のひとつである関数ポインタが多くの HLS ツールでサポートされていないことが挙げられる．

C 言語における抽象化の例として，関数ポインタを用いた高階関数の実現がある．例えば，標準ライブラリに含まれる `qsort` は値の比較するための関数を受け取る (Listing 6.1)．関数ポインタを通して任意の型に対する比較を定義できるようにすることで，ユーザ定義型に対してでも sort アルゴリズムを提供することができるようになる．この他に重要な例として，コールバック関数がある．外部からの入力に対してレスポンスを返すようなユーザインタフェースや通信における開発では，イベント駆動型のプログラミングモデルが適している．このようなシステムの開発フレームワークを実装する際には，イベントに対してユーザが任意の処理を追加できるようにコールバック関数が登録できる必要がある (Listing 6.1).

6.1.2　HLSにおける関数ポインタのメリット

HLS では，配列を除いたポインタ変数はサポートされないことが一般的である．指す先がコンパイル時に静的に決定されず実行時に動的に決まるような場合，間接参照する回路の間でメモリ空間を共通化する必要があるため性能的に不利になってしまう．また，動的に値が決まる関数ポインタの場合には，間接呼び出しを行う回路から型の一致する関数すべてにアクセスできる必要があり，回路面積的に不利になる（Listing 6.2, 図 6.1）．

この章では，C 言語を用いる HLS 言語における関数ポインタの実現手法を提案し，記述の抽象度をより高めることを目的とする．また，提案手法が回路の合成結果に対して与える影響について評価を行う．関数ポインタを利用するプログラムを，関数ポインタを利用しないプログラムに変換する手法として提案する．すなわち，提案手法をコンパイラの前処理や最適化のひとつとして施すことによって，あらゆる高位合成ツールで関数ポインタをサポートすることができるようになる．

Listing 6.1: Examples of high-order functions

```
1  // qsort
2  void qsort((void*)data, size_t len, size_t t_size, int (*cmp)(const void* a, const void* b));
3
4  // Callback function
5  void append_callback(EventType type, void (*user_f)(const Event* event));
```

Listing 6.2: sample list 2

```
1  typedef int (*FnPtr)(int*, int*);
2
3  int func_0(int* a, int* b);
4  int func_1(int* a, int* b);
5  int unused_func(int* a, int* b);
6
7  int caller(FnPtr op, int* a, int* b){
8    return (*op)(a, b);
9  }
```

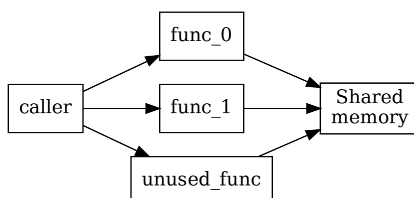


図 6.1: unsinthesizable pointers

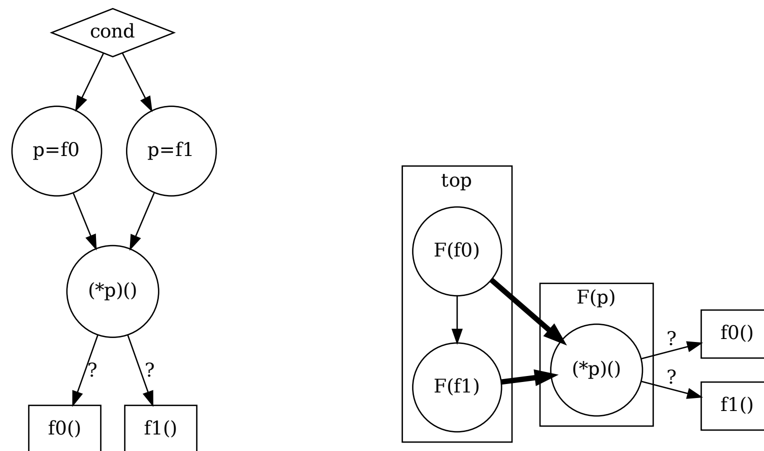


図 6.2: Examples of indirect call that cannot be resolved by constant folding: (left) A CFG with a condition node, (right) a CFG with two indirect calls.

6.2 関連研究

6.2.1 定数畳み込み

最も straight-forwad な方法は，定数畳み込みを用いる手法である．定数畳み込みが適用可能な場合は直接呼出しに置き換えることができる．このような場合に限って関数ポインタが利用可能なツールも存在している．ただし，関数ポインタ変数の指す先が条件分岐などによって変化する場合には定数畳み込みができず，コンパイルすることができない．また，ある高階関数が異なる Callee を用いて複数回呼び出される場合にも定数畳み込みに失敗することがある．図 6.2 に，定数畳み込みで解決できない間接呼び出しの CFG の例を示す．図中のノード（`(*p)()`）は関数ポインタ変数 `p` を通した間接呼出しである．図中の太矢印は関数 `F` の呼出しを表しており，関数 `F` は引数 `p` として関数ポインタを受け取る．

6.2.2 C++のテンプレートによる関数引数

定数畳み込みを用いた手法に似た手法として，C++のテンプレートを用いた手法が提案されている [98]．関数呼び出し演算子を実装したクラスのインスタンスを引数として受け取る関数と関数呼び出し演算子のオーバーライドによって，関数引数を受け取るのと同等な記述を実現している．しかしながら，C++のパラメトリック多相の機能（テンプレートクラスやテンプレート関数）を利用しているため，コンパイルの時点でディスパッチされるメソッドが決定できないプログラムはコンパイルすることができない．したがって，定数畳み込みの場合と同様に動的に呼び出される関数が決まる場合には利用することができない．

6.2.3 メモリマップ型インタフェース

動的な関数ポインタ変数をサポートした事例として, メソッド間でのリソース共有 (inter-procedural resource sharing) が提案されている [99], [100]．この手法では, 関数から生成される全てのモジュー

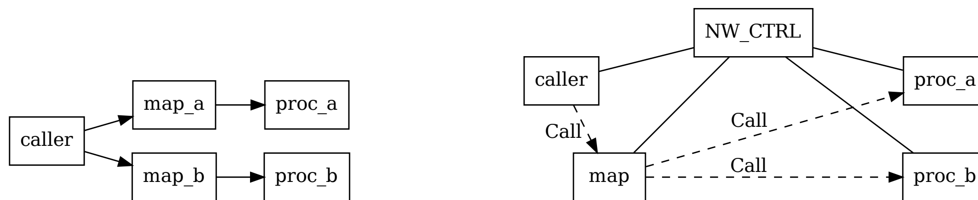


図 6.3: (top) Naive solution, (bottom) [99]'s solution

ルはメモリマップ型のインタフェースを通して互いに呼出しが可能である．図 6.2 にブロック図の例を示す．全てのモジュールがコントローラを通して相互に接続されており，関数ポインタ変数の値が動的に変化する場合でも任意の関数を間接呼出しすることが可能である．しかしながら，モジュールが増えるにつれメモリマップ型インタフェースを実現するためのネットワークが複雑化し，回路面積や遅延の面で不利となりやすいといえる．

6.2.4 Point-to 解析によってポインタ変数を取り除く方法

L. Semeria らの研究グループは，変数へのポインタと動的メモリ確保を実現する手法を提案している [101], [102]．複数のモジュール間で共有されるメモリ空間を用意し，Point-to 解析 [101] によってあらかじめ参照されるメモリ空間を絞り込むことによってメモリ空間を限定する手法をとっている．このため過剰なメモリ空間の確保やモジュール間ネットワークを回避し，効率的な回路合成を実現している．この提案の中では関数ポインタの議論はなされていないものの，この手法は関数ポインタにも応用できると考えられる．

6.3 関数ポインタの削除

本手法では，プログラムの解析を容易にするため，LLVM IR や GIMPLE のような SSA 形式 (static single assignment form) の中間表現を利用する．Listing 6.3 は間接呼び出しを含む LLVM IR の例であり，図 6.4 はその CFG である．

本手法の基本的なアイデアは，次の 3 つの手順となる：

1. 呼び出される可能性のある関数を各間接呼出し命令毎に列挙し，
2. プログラム中の関数ポインタを関数 ID 変数に変換し，
3. 間接呼び出し命令を Switch 命令と直接呼出し命令で置換する．

各間接読み出し命令毎に候補値 (candidate callees) をリストアップすることでモジュール間接続のネットワークの最小化（回路と遅延の最適化）を行う．Listing 6.3 に対して関数ポインタの削除手法を適用した結果を Listing 6.4 に示す．また図 6.5 はその制御フローである．

ハードウェア的な解釈　図 6.3, 6.4 のプログラムを例に，生成されるハードウェアのゴールを図 6.6 に示す．Callee となるモジュールはマルチプレクサを通して呼び出し元に接続され，実行時の文脈に応じて呼び出される関数が変化する回路とすることが目標である．

Listing 6.3: Example with an indirect call

```
1  entry:
2    ...
3    br i1 %cmp, label %if.then, label %if.else
4
5  if.then:
6    store i32 (i32)* @foo, i32 (i32)** %fp
7    br label %indirect_call
8
9  if.else:
10   store i32 (i32)* @bar, i32 (i32)** %fp
11   br label %indirect_call
12
13 indirect_call:
14   %0 = load i32 (i32)** %fp
15   %call = call i32 %0(i32 %arg)
16   store i32 %call, i32* %result
```

Listing 6.4: Optimized IR of Listing 6.3

```
1  ...
2
3  indirect_call:
4    %0 = load i32* %fp
5    switch i32 %0, label %call_foo [
6      i32 0, label %call_foo
7      i32 1, label %call_bar
8    ]
9
10 call_foo:
11   %call = call i32 (i32)* @foo(%arg)
12   store i32 %call, i32* %result
13   br label %next
14
15 ...
16
17 next:
18   %1 = phy i32 [%call, label %call_foo], ...
19   store i32 %1, i32* %result
```

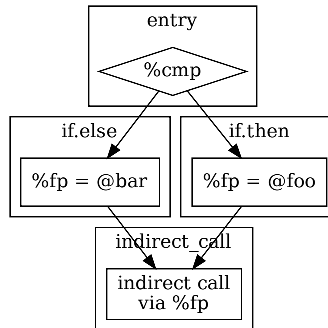


図 6.4: CFG of Listing 6.3, that contains dynamic indirect function call

Listing 6.5: Three types of `store` instruction usage regarding function pointers

```
; 1) Store the addres of function foo
; to the pointer
store i32 (i32)* @foo, i32 (i32)** %fp;

; 2) Store the value of the pointer
; to another pointer
%0 = load i32 (i32)** %pred_fp;
store i32 (i32)* %0, i32 (i32)** %fp;

; 3) Store the value of the phy instruction
; to the function pointer
%1 = phi i32 (i32)* [@foo, %blk0], [@bar, %blk1];
store i32 (i32)* %1, i32 (i32)** %fp;
```

6.3.1 Callee の候補値探索

最初のステップとして，各間接呼び出しの命令について，呼び出される可能性のある関数を探索し，そのリストを作る．全ての関数のアドレスは実行時に変化することはなく定数として扱うことができるため，それぞれの関数に整数値の ID を割り当て，メモリ空間上のアドレスの代わりに用いる．整数型や浮動小数型のような変数は値の範囲を絞り込むような推論となるが，今回は関数ポインタ（変数）に限ったもので不確定の値を使った演算は伴わないため，確定的な値を推論することが可能である．

ある関数ポインタの候補値は，その関数ポインタに関連する `store` 命令を再帰的に辿ることによって列挙することができる．関数ポインタに関連する `store` 命令は，1) 関数のアドレスを関数ポインタ変数に `store` する場合と，2) 関数ポインタ変数から関数ポインタ変数へ `store` する場合と, 3) `phy` 命令から関数ポインタへの `store` の 3 種類に分けられる (Listing 6.5).

1) は関数のアドレスの値を直接 Store するものであるため，対応する関数 ID をがそのまま関数ポインタの候補値のひとつとなる．

2) は読み出し元の関数ポインタの候補値を再帰的に辿り，関数ポインタの候補値を調べる．

3) は直前に分岐命令があり代入される値に複数の候補がある場合である．

図 6.5 の next ノードに含まれる phi 命令は，SSA 形式における ϕ 関数であり，制御が合流した際にこれまでの分岐に応じた値を取り出すための仮想的な命令である

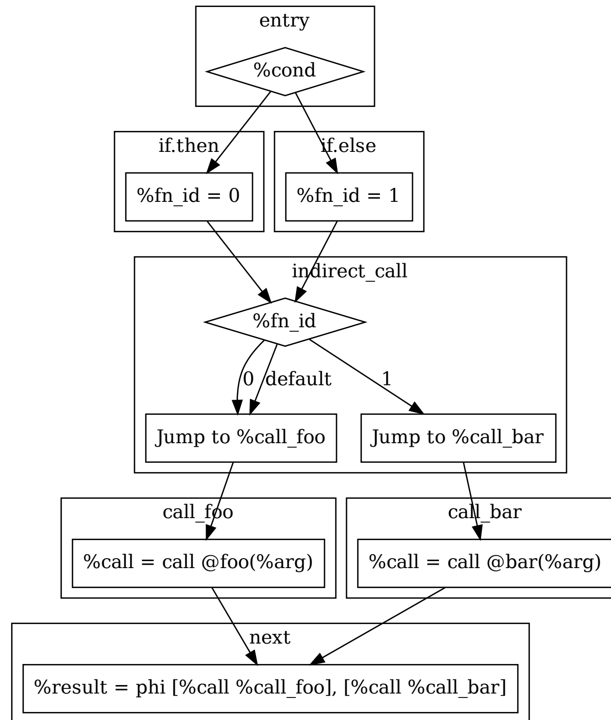


図 6.5: The CFG that the indirect call has removed

Listing 6.6 は，この場合分けに基づいて関数ポインタの候補値を求めるための再帰関数の疑似コードである．LLVM では各変数に対してその変数を使用している命令の一覧：`users` を保存しているので，これを利用している (line 3)．また，`is_store_faddr2fp` (line 4), `is_store_fp2fp` (line 6), `is_store_phi2fp` (line 8) はそれぞれ `store` 命令を種別毎に判定するための関数である．ϕ 命令の場合には内容が関数のアドレスを直接返される場合と他の関数ポインタからのコピーの場合があるため，それぞれの場合に応じて分岐する (line 10-13).

計算量は，間接呼び出しの数を N，それぞれの関数ポインタに対する store 命令の個数の平均値を M として，$O(N \log M)$ 程である． 一般的に M の値は N に対して十分小さいものであるため，概ね線形時間であり高速に処理できる．

ただし C 言語では，別ファイルで定義されている関数の名前解決はリンカが行うため，Callee 探索の段階で他ファイルで定義されている関数の名前解決を行う必要がある．本稿では詳細な議

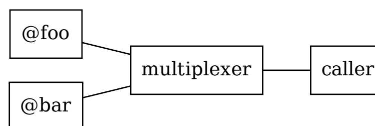


図 6.6: Indirect call with a multiplexer

Listing 6.6: The recursive function to build adjacency graph

```
1  CandList get_candidate(FunctionPointer fp){
2    CandList cand_list;
3    for (inst in fp.users)
4      if (is_store_faddr2fp(inst))
5        cand_list += inst.rhv;
6      else if (is_store_fp2fp(inst))
7        cand_list += get_candidate(inst.rhv);
8      else if (is_store_phi2fp(inst))
9        for (value in inst.values)
10         if (is_from_fp(value))
11           cand_list += get_candidate(value);
12   else (is_faddr(value))
13     cand_list += value;
14   return cand_list;
15 }
```

論は省くが，実際には前処理などによってプログラム全体の関数の ID 一覧を作成し，コンパイル（最適化）時に参照できるようにする必要がある．

また本稿では，関数ポインタおよびその配列，構造体の動的な確保については議論せず，サポートしない．ただし，関数ポインタの配列については次パラグラフで議論する．

関数ポインタ配列のサポート　関数ポインタ配列をサポートするには，関数ポインタへのポインタをサポートする必要がある．変数の動的確保が行われると解析が複雑になるため，本稿においては関数ポインタ及び関数ポインタ配列の動的確保のサポートは行わない．

関数ポインタの配列においては，コンパイラはどの要素の値が読み書きされるのか静的に決定することができない場合もあるため，添え字に関わらず共通の候補関数リストを利用する．たとえば，`void (*ary[])() = {foo, bar};` といった関数ポインタを保持するポインタ配列を使った間接呼び出しは，どの要素を使った間接呼出しであったとしても共通の ID リストを用いた switch 命令を利用する．

したがって，指す先の関数ポインタ全ての候補値を更新することで過不足のない候補値伝播が可能になる．

6.3.2　関数ポインタを用いている命令の置換

置換すべき命令は次の通りである：

1. `alloca` 命令,
2. `load` 命令,
3. `store` 命令,
   (a) 関数アドレスを関数ポインタに `store` する場合,
   (b) 関数ポインタの値を関数ポインタに `store` する場合,
   (c) `phi` 関数の値を関数ポインタに `store` する場合,
4. `call` 命令.

Listing 6.7: Replacement of the instructions that reference function address/pointer

```
1  ; 1) ; %fp = alloca i32 (i32)*, allign 4
2  %fp = alloca i32, allign 4
3
4  ; 2) ; %0 = load i32 (i32)** %fp;
5  %0 = load i32* %fp;
6
7  ; 3-a) ; store i32 (i32)* @foo, i32 (i32)** %fp;
8  store i32 0, i32* %fp;
9
10 ; 3-b)
11 ; %0 = load i32 (i32)** %pred_fp;
12 ; store i32 (i32)* %0, i32 (i32)** %fp;
13 %0 = load i32 %pred_fp;
14 store i32 %0, i32* %fp;
15
16 ; 3-c)
17 ; %1 = phi i32 (i32)* [@foo, %blk0], \\
18 ; [@bar, %blk1];
19 ; store i32 (i32)* %1, i32 (i32)** %fp;
20 %1 = phi i32 [0, %blk0], [1, %blk1];
21 store i32 %1, i32* %fp;
```

Listing 6.7 に，4) 以外の各命令の置換例を示す．1)，2)，3-b) については，単に引数の型を整数値にする．3-a) および 3-c) の場合は引数の型を整数値に変換した上で，関数名を対応する関数 ID に置き換える．

Listing 6.8 に 4) 間接呼び出し命令の置換例を示す．間接呼び出し命令 (L3) は switch 命令に置換 (L.8-11) され，候補となっている callee の ID 毎に直接呼出しを行う block に分岐する．各 block では ID に対応する関数の直接呼出し (L.14,18) が行われ，合流ブロック (L.21) では ϕ 命令によって正しい返り値が保存されるようにする．

6.3.3 Void-type pointer

C 言語において関数ポインタを用いて高階関数のような仕組みを実現するためには，`void` 型ポインタの利用が必須である．Listing 6.9 に `void` 型ポインタを用いた高階関数 `map` の実装例と引数に渡すことができる関数の例を示す．この例の中の `map` 関数は，任意の型のデータを受け取るため，ターゲットとなるデータ配列 `ary` および適用する関数 `f` に `void` 型ポインタが利用されている．関数 `twice` の中で明示的型変換によって `void` 型ポインタを `int` 型ポインタへ変換しているが，HLS においては静的に型が決定していなければバスやレジスタの幅を決定することができずコンパイルができない．

そこで，`void` 型ポインタを静的に解決してしまう実装方法を用いる．Listing 6.9 の `void` 型ポインタを静的に解決できるように実装したプログラムを Listing 6.10 に示す．最も大きな変化は関数 `alt_map` の第一引数が配列への `void` 型ポインタ `void* ary[]` から `void* (*ary_at)(size_t pos)` という関数ポインタへと変化した点である．配列へアクセスするための関数 `my_ary_at` が間接呼び出しになる（Listing 6.10 3 行目）ため，関数ポインタ削除手法を用いることで直接呼出しに変換することができる．この時重要なのは配列へのアクセス関数 `my_ary_at` の返り値が `void` 型ポインタではなく具体的な型になっている点である．したがってコンパイラは候補となる型とその場所について知ることができ，操作の対象となる配列への配線が可能となる．また，コンパイラが型を解決できるようになりユーザ定義関数 `alt_twice` の中で明示的型変換が必要なくなり仮

Listing 6.8: Replacement of an indirect call

```
1  ; 4) Indirect call
2  ; indirect_call:
3  ; %0 = load i32 (i32)** %fp
4  ; %call = call i32 %0(i32 %arg)
5  ; store i32 %call, i32* %result
6  indirect_call:
7    %0 = load i32* %fp
8    switch i32 %0, label %call_foo [
9      i32 0, label %call_foo
10     i32 1, label %call_bar
11   ]
12
13 call_foo:
14   %call = call i32 (i32)* @foo(%arg)
15   br label %next
16
17 call_bar:
18   %call = call i32 (i32)* @bar(%arg)
19   ...
20
21 next:
22   %1 = phy i32 [%call, label %call_foo], ...
23   store i32 %1, i32* %result
```

Listing 6.9: Map function using void pointer

```
1  void map(void* ary[], size_t len, void (*f)(void* val)){
2      for (size_t pos = 0; pos < len; pos++){
3        (*f)(ary[pos]);
4      }
5  }
6
7  void twice(void* val){
8      int *d = (int *)val;
9      *d = *d * 2;
10 }
```

Listing 6.10: Alternative implementation of map of 6.9

```
1  void alt_map(void* (*ary_at)(size_t pos), size_t len, void (*f)(void* val)){
2      for (size_t pos = 0; pos < len; pos++){
3          (*f)(ary_at(pos));
4      }
5  }
6
7  int ary[SIZE] = {...};
8  int *my_ary_at(size_t pos){
9      return &ary[pos];
10 }
11
12 void alt_twice(int* val){
13     *val = *val * 2;
14 }
```

Listing 6.11: Sample program for the evaluation

```
1  /* baseline (no indirect call) */
2  responce = request(add_user, "foo", "bar");
3  responce = request(add_user, "hoge", "fuga");
4  ... // totally five add_user requests
5
6  responce = request(find_user, "foo", "bar");
7  responce = request(find_user, "hoge", "fuga");
8  ... // totally 5 req
9
10 responce = request(del_user, "foo", "bar");
11 responce = request(del_user, "hoge", "fuga");
12 ... // totally 5 req
13
14 /* With indirect call */
15 auto f_list[] = {add_user, find_user, del_user};
16 for (auto q : f_list){
17   responce = request(q, "foo", "bar");
18   responce = request(q, "hoge", "fuga");
19   ... // totally five indilect calls
20 }
21
22 // ‘request‘ will do nothing, just yields process to the passed function
23 void request(Response (*q)(char[], char[]), char[] id, char[] password){
24   (*q)(id, password);
25 }
```

表 6.1: FPGA resource consumption (on Cyclone V GX-9)

	ALM	Registers	Memory bits
No indirect call (baseline)	4,198	3,011	**58,544**
With indirect call (ours)	**1,191**	**1,082**	59,336

引数に型を指定できるようになったため，関数ポインタ f に関しての間接呼び出しの削除を行えば，新しく生成された直接呼出しに void 型ポインタが含まれることはなくなり，こちらも同様にコンパイラが配線やレジスタの幅を静的に決定できるようになる．

6.4 Evaluation

評価として，操作を関数ポインタで指定できる簡易的なデータベースを実装した．評価に用いたプログラムの一部を Listing 6.11 に示す．Baseline として間接呼出しを用いないプログラムを用い，間接呼び出しを用いた同じプログラムと比較した．

評価の結果を Table 6.1 と Table 6.2 に示す．ALU とレジスタの消費はおおよそ 1/3 になり，メモリ使用量は変化しなかった．一方で，実行サイクル数は 3 倍となった．

関数ポインタを使ったプログラムから生成されたハードウェアを図 6.7 に示す．Baseline では処理が inline 展開され多くのリソースを使っていたが，Ours のコードではマルチプレクサを通して関数を指定することでリソース量を削減した．しかしながら処理がシリアライズされるため，処理にかかるサイクル数が増加したと考えられる．

表 6.2: Performance (simulation)

	Cycles
No indirect call (baseline)	**8,225**
With indirect call (ours)	24,140

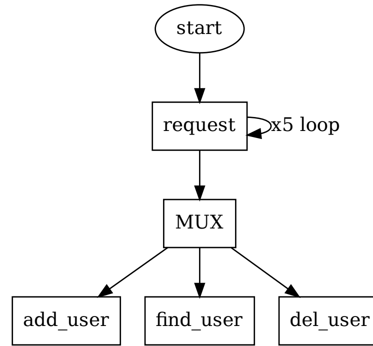


図 6.7: Hardware generated from the code including indirect call

6.5 まとめ

この章では，C 言語プログラムの SSA 形式中間表現から関数ポインタを介した間接呼び出しを削除するための手法について説明した．動的なふるまいをする間接呼び出しは HLS ツールではサポートすることができなかったが，間接呼び出し削除手法によってすべての間接呼び出しが HLS ツールで合成可能となる．間接呼び出しは高階関数のような抽象度が高く再利用性の高い記述には欠かせない機能であり，HLS ツールの生産性の向上に大きく寄与する手法である．関数のアドレスはプログラム実行中は定数であることから，これらを定数として扱い定数伝播によって各間接呼び出しにおける呼び出される可能性のある関数の一覧を作成することで，合成する際の面積効率を改善することができる．また，コード中から関数ポインタ自体を削除する手法であるため，既存の HLS ツールなどにおいてもプリプロセスとして本手法を適用することによって関数ポインタのサポートが可能となる．高階関数の実現には `void` 型ポインタが必須であるが，関数ポインタを用いて `void` 型ポインタの変数を覆い隠し型ヒントを与えるようなテクニックを紹介し，高階関数を実現するための手法についても説明した．

第7章　データフロー型プログラムと分散FPGA環境

7.1　はじめに

7.1.1　FPGA のクラウドコンピューティングへの展開

クラウドコンピューティングサービスが広く普及し，様々な用途で活用されている．クラウドサービスは Software as a Service (SaaS)，Platform as a Service (PaaS)，Infrastructure as a Service (IaaS) の 3 種類に大別できる．SaaS は，Google 社の gmail や Microsoft 社の Office 365 のような，ソフトウェアをインターネットを通じてどこからでも利用できるサービスである．PaaS は，Salesforce 社の Heroku や IBM 社の Watson などのような，ミドルウェアをホストし提供するクラウドサービスである．そして IaaS は，Amazon 社の AWS のような，仮想マシンやネットワークを管理し提供するクラウドサービスである．

このようなクラウドコンピューティングにおいて，FPGA を利用することが注目され始めている．FPGA は電力当たりの処理能力が高く，ビジネス上で利用するにあたって大きな利点となる．Amazon 社は，IaaS 型クラウドサービスの一つである AWS EC2 において，FPGA を利用可能なマシンである F1 インスタンスの提供を開始した [103].

7.1.2　FPGA 混載ヘテロジニアスシステム上の問題点

分散 FPGA の課題と Mulvery　ひとつのアプリケーションを複数の FPGA 上で動作させるためには大きく分けて 2 つのアプローチがある．第一に，論理合成を行った大きな回路を複数の回路に分割する手法である．単一の FPGA に収まらないような HDL で記述された巨大な回路であっても複数の FPGA で動作させることが可能となるアプローチであるが，FPGA 間の通信を最小限に抑えるためのグラフカットや回路を同期させるための機構によって動作周波数が低下してしまうなど，以前様々な課題が残されている分野である．第二に，機能を分割して FPGA 間で役割を分担させる手法である．HDL で記述する場合においては単純にモジュールを適宜複数の FPGA に分割し FPGA 間で情報のやり取りをするためのハードウェアを設計すればよい．しかしながらモジュール間で情報を受け渡す際のストール制御なども都度行う必要があるため，HLS を活用して FPGA 間の通信の設計の自動化などを行う企業の取り組みも存在する．ただし，C 言語のようなプログラミング言語はハードウェアに合成された際のモジュール間の接続を表現するすべを持たないため，C 言語の仕様に含まれない何かしらの異なる手段を以て切断点を明示する必要がある．

これらに対して，複数のモジュールを繋ぎ合わせデータフローを表現した Mulvery の記述は，回路分割を容易に行うことができる特徴がある．たとえば `source.opA().opB()` というプログラムにおいては， `source.opA()` と `.opB()` のように，任意の隣接したオペレータ間のカットは他のオペレータに影響を及ぼさず実行可能である．つまり，カットした部分のデータの受け渡しが FPGA 間通信になるのみである（図 7.1）.

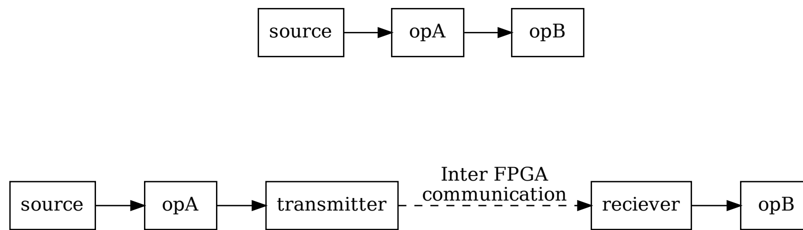


図 7.1: (top) Original block diagram of `source.opA().opB()` and (bottom) the program devided between `opA` and `opB`.

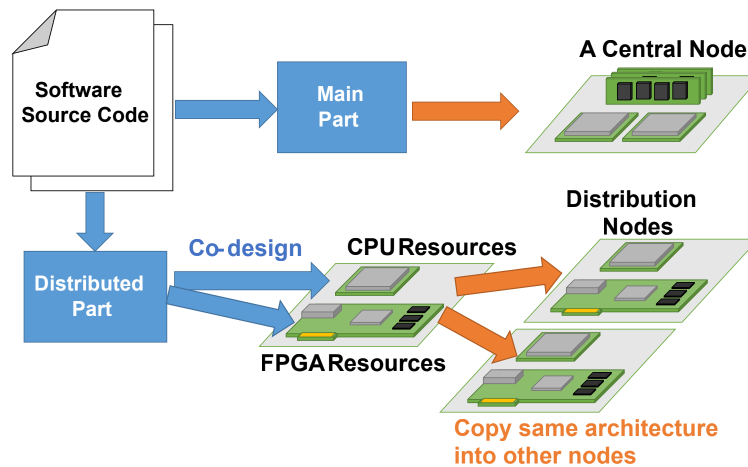


図 7.2: Usual design method of distributed system which includes FPGAs

PaaS 型クラウドコンピューティングで活用する場合の課題　FPGA を PaaS 型のクラウドコンピューティングサービスで活用する場合，どのようなタスクがどの程度の量実行されるのか事前に予測できず，運用中にも CPU や FPGA のリソースの需要が動的に変化する．このため，タスクに対して割り当てる CPU や FPGA のリソース量を動的に変化させることが可能な分散システムが必要となる．しかしながら動的に回路規模を変更することは困難であり，また FPGA を割り当てる数を動的に変更する場合には FPGA の再構成にかかるオーバヘッドが必要となってしまう．

素朴な FPGA 混載分散システムの設計フローとして，あらかじめ各計算ノードに配置する処理を切り出しておき，これを高速化する FPGA 回路の設計や CPU との協調設計を行う設計手法（図 7.2）が考えられる．しかしながらこの手法では動的なリソースの再配分の粒度が CPU+FPGA の計算ノード単位となってしまう．例えば，CPU リソースのみを必要とするタスクに計算リソースを割り当てる場合，拘束されるが処理を行わない FPGA が出現することが予測できる．

異なるアプローチとして，FPGA を必要とするタスクから優先的に FPGA を割り当て，FPGA が不足する場合には CPU で代替するようなリソース配分を行う手法が考えられる．しかしながら，これを実現するためには同じ計算モジュールをプログラムとハードウェアの 2 種類記述する必要があり開発コストに見合う効果は期待できない．

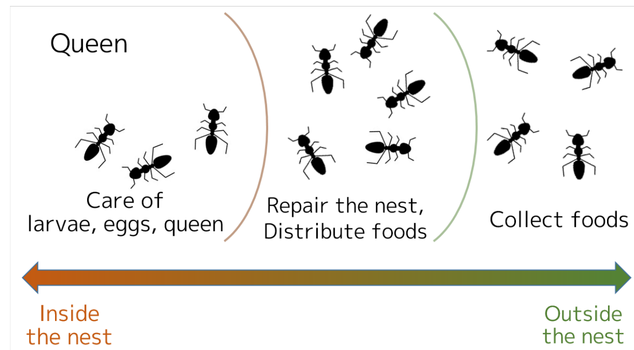


図 7.3: Organization structure of ants

7.1.3 社会性昆虫の生態とその応用

ハチやアリのような社会性昆虫は，非常に多数の個体によってコロニーを構成し，運用する．社会性昆虫のコロニーは指揮命令系統を持たずリーダーや上司といった役割の個体が存在しないにも関わらず，コロニー運営に必要な労働を分担し，協力している．しかしながら，コロニー全体を俯瞰すると，必要なタスクに対し適応的に労働力の割り当てが行われている（図 7.3）[104] [105].

文献 [106] では， シワクシケアリ (*Myrmica kotokui*) のコロニーについて，次のふたつの特徴を発見した：1) 労働しない個体のみを集めて新たなコロニーを形成すると，働き始める個体が出現する．2) 盛んに労働する個体のみを集めて新たなコロニーを形成すると，働かない個体が出現する．これは，コロニーの状況に応じて労働する個体の数が変化する適応的な労働の制御システムがあることを示唆している．Bonabeaw らの研究グループは，この制御システムを **反応閾値モデル** としてモデル化した [105].

ある個体が，「巣の破損」「幼虫からの餌の要求」などの外部からの刺激に対してどの程度の強さで反応を始めるかは個体によってそれぞれ異なり，この「閾値」によって適応的タスク配分を実現するモデルである．例えばアリの成虫から幼虫への給餌を例に挙げると，「N 匹の幼虫が餌を要求すれば餌を運ぶ」という閾値 N を個体ごとに個別に持つということになる．コロニー全体での閾値の分散が十分に大きければ，要求が少ない場合には閾値の低い成虫のみが反応し，要求が増えるほど働き始める成虫が増える．このようにして，指揮命令系統を持たずに適応的な労働力配分を実現するモデルである．

このような社会性昆虫の自己組織性は，スティグマジ *(stigmergy)* とよばれる，環境を介した個体間の情報伝達のもとに実現される．スティグマジを介した情報伝達の例として，ant colony system と呼ばれる短い経路を選び出すための方法が知られている [107]．アリは餌を巣に持ち帰る際に揮発性のフェロモンを経路に残し，他のアリはそのフェロモンをたどることで同じ餌を発見する．ある餌の回収が複数の経路によって行われていたとき，長い経路のフェロモンは短い経路のフェロモンと比較して揮発量が多く，フェロモンの強度が強い短い経路の方が選択されやすくなる．長い経路は選択される回数が減りフェロモン濃度がより低下するため，最終的に淘汰されていく．このモデルは確率的経路最適化アルゴリズムである蟻コロニー最適化 (ant colony optimization) として情報処理の分野での応用もなされている [108] [109].

7.1.4 分散システムにおける応用

コンピュータシステムがより複雑になるにつれて，コンピュータを結ぶネットワークの設定や管理は大きな課題となる．システムが自身の最適化を行う autonomic computing というコンセプトが提案されている [110].

文献 では，資源割り当て問題と反応閾値モデルの共通性に着目し，ネットワーク資源割り当て問題をモデル化し，反応閾値モデルを適用した．この文献では，現実に即したモデルへのさらなる拡張が必要としながらも，シミュレーションを用いて反応閾値モデルの資源割り当てに対する有効性を示した．このほか，TERMES プロジェクトは社会性昆虫の生態の自律分散システムを応用して，複数のロボットが自律的に協力する手法を提案している [111], [112]．TERMOS プロジェクトのロボットは，ひとつの構造物を複数のロボットで協力して組み上げるが，互いに通信はせず独立して自身のセンサデータのみに基づいて行動する．

本稿では，分散システムにおけるホストへのタスク割り当てと社会性昆虫の労働力配分の仕組みの共通性に着目し，本稿では，分散システムと社会性昆虫に共通の「ワーカへのタスク割り当て」問題の共通性に着目し，社会性昆虫の労働力配分の仕組みを分散システムに応用することを目標とする．

7.1.5 目的と目標

この節では，スティグマジと反応閾値モデルに基づく動的な計算資源割り当ての仕組みを提案する．具体的には，次のような三つの性質を実現することを目標とする：

- 制御ノードを必要としない分散コンピューティング環境
- タスクの状態に応じた動的計算資源配分
- 一部計算資源の喪失などに対する耐障害性

7.2 反応閾値モデルの適用

7.2.1 反応閾値モデル

アリの社会におけるタスクは大きく分けて，幼虫の世話といった持続的だが時間と共に要求の大きさが変化するものと，餌の回収など発生タイミングの予測が不可能なものの二種類に分けられる．求められる労働力の量はタスクや時刻に応じてそれぞれ異なっているため計画的に労働力配分を決めることができず，それぞれの時点で適切なワーカの数を配分する必要がある．Bonabeau らによるモデルでは，各個体はタスクそれぞれの要求量に応じて確率的に取り組むかどうか判断する [113].

文献 [105] では，ある個体がタスクから受ける刺激に対して非活動状態から活動状態に遷移する確率 $P(X = 0 \to X = 1)$ を式 (7.1) のようにモデル化している．

$$P(X_i = 0 \to X_i = 1) = \frac{s^2}{s^2 + \theta_i^2} \tag{7.1}$$

ただし，X_i を個体 i の活動状態（$X_i = 0$ ならば非活動で，$X_i = 1$ ならば活動）を表しており，また θ_i は個体 i の反応閾値，s はタスクからの刺激の大きさを表している．

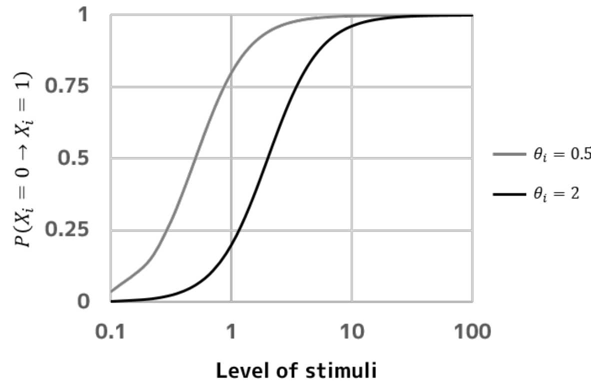


図 7.4: Comparison of response curves with different θ_i

図 7.4 は，ある個体があるタスクに対する反応確率 $P(X_i = 0 \to X_i = 1)$ が要求量に対してどのように変化するのか示したものである．また，反応閾値 θ_i の変化によってグラフがどのように変化するかも示されている．θ_i が大きいほど反応確率の変化が緩やかになる．

本研究では Bonabeau らのモデルに基づいて，時刻 t においてワーカ i がタスク j の処理を始める確率 $P_{ij}(t)$ を式 (7.2) に示すように定義する．

$$P_{ij}(t) = \frac{(\theta_i \cdot S(M_i, t))^2}{(\theta_i \cdot S(M_i, t))^2 + \bar{T}^2} \tag{7.2}$$

θ_i はワーカ i の反応閾値を示す．$S(M_j, t)$ は時刻 t におけるタスク j からの要求の強度であり，M_j はタスク j が固有に持つ情報である．この $S(M_j, t)$ についての詳細は後述する．$\bar{T}$ は P_{ij} の変化の速度を制御するための定数パラメータであり，$\theta_i = 1$ である場合に $P_{ij} = 0.5$ になるまでに必要な時間である．

要求量 $S(M_j, t)$ の例　要求量 $S(M_j, t)$ はタスクの性質に応じて自由に設定できる．例えば，時間に対して線形に要求量を変化させる場合には，式 (7.3) のように設定すればよい．ただし A は全てのタスクで共通の比例定数，T_j はタスク j が発生した時刻である．

$$S(T_j, t) = A \cdot (t - T_j) \tag{7.3}$$

スティグマジによる情報の共有　複数のタスクに対し複数のワーカで取り組むためには，タスクの情報が共有されている必要がある．アリの場合，Peer-to-peer[1]モデルのように直接的なコミュニケーションをとってタスクの情報を伝え合うと情報の伝播に時間を要し非効率的であるため，スティグマジモデルと呼ばれる環境を介した間接コミュニケーションを行うことが知られている [104]．本研究で提案するモデルでは，このような間接コミュニケーションを取り入れる．この節では，タスクの情報やその状態の一覧を環境情報と呼び，それらを共有するためのサブシステムをスティグマジと呼ぶこととする．

[1] P2P; ネットワークにおいてホスト同士が直接コミュニケーションを行うモデル

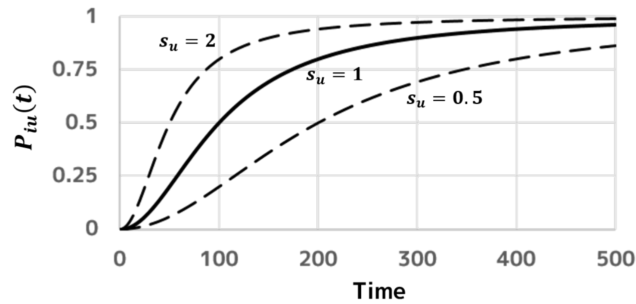


図 7.5: Graph of $P_{ij}(t)$ using Equation(7.3) ($\bar{T} = 100$)(upper:$\theta_i = 0.5$, center:$\theta_i = 1$, lower:$\theta_i = 2.0$)

7.2.2 CPU・FPGA の差別化と学習係数

本研究では，処理するタスクの種類を頻繁にスイッチさせないために，ノード i がタスクを処理している最中は，異なる種類のタスクに対して $\theta_i = \theta_i + \varphi$ とする．FPGA による計算ノードに対しより大きな φ を設定することで，頻繁にタスクの種類が切り替わることを防ぎ動的再構成に必要なオーバヘッドの発生を低減することが可能になる．

7.3 新たな自律分散システムの構築

提案手法の概略図を図 7.6 に示す．

システムは，1) ワーカ，2) スティグマジ，3) メディエータの三つのコンポーネントで構成される．これらのコンポーネントは，単一の計算機の中で個別のプロセスとして動作することも，複数の計算機に分散して配置されることもある．

ジョブとタスク　本稿では，ユーザが実行したい処理の単位をジョブと呼ぶ．ジョブは複数の小さな処理に分割され，この単位をタスクと呼ぶ．タスクとジョブにはそれぞれ ID が振られる．

ワーカ　単一の計算機の中で複数のワーカが個別のプロセスとして動作することもある．各ワーカは一様分布や正規分布従う乱数を用いて独自に自身の反応閾値 $theta_i$ を決める．この乱数の分布の累積分布関数の形によって，あるタスクに対してワーカの増え方をある程度制御することができる．つまり，あるタスクの要求量が時間に対して線形に増えるとき，一様分布であれば概ね線形に，正規分布であればシグモイド曲線様にワーカの数が増える．ただし，システム全体のふるまいを制御しやすくするために，乱数の分布のパラメータ（i.e. 分散や平均など）はシステム全体で共通のものを用いる．

スティグマジ　スティグマジはシステム中に唯一つ存在し，ワーカが情報を共有するための「環境」である．スティグマジは「処理待ちタスク」と「処理済みタスク」の情報を記録している．処理待ちタスクは処理に必要なデータやプログラム，処理済みタスクはその処理結果を保持している．

スティグマジは，次のような機能を提供する：

- 処理待ちタスクの一覧の取得

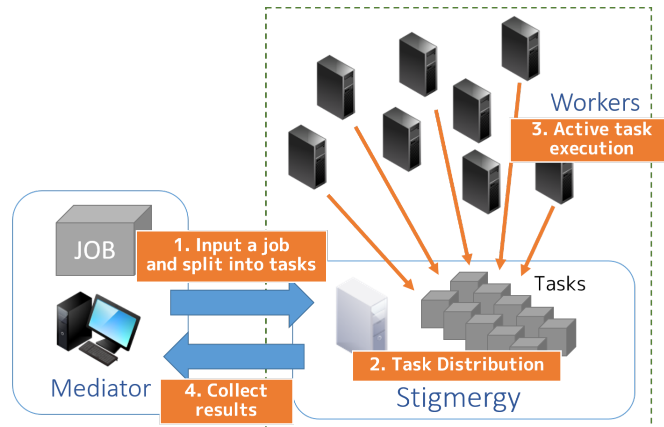


図 7.6: Flow of Job processing

- 処理済みタスクの一覧の取得
- タスクの処理結果の記録

メディエータ　メディエータはシステム中にひとつ以上存在する．メディエータはユーザとシステムのインタフェースであり，ユーザはこれを介してジョブをシステムに登録する．ジョブをタスクに分割する作業はメディエータが行い，すべてのタスクの処理が完了したことを検知してユーザに通知する．

ジョブの処理の流れ

Step 1. ジョブの投入とタスク分割　ユーザからジョブが投入されると，メディエータはジョブをタスクに分割する．スティグマジにタスクが記録され，ワーカから読み書きが可能になる．

Step 2. スティグマジを通したタスク配分　ワーカは定期的にスティグマジに対して処理待ちタスクの一覧を要求する．この後，ワーカはそれぞれのタスクに対し反応確率 $P_{ij}(t)$ を計算する．もし値域 $[0,1]$ 一様乱数 r が $r < P_{ij}(t)$ ならば，ワーカはタスク j の処理を開始する．

Step 3. ワーカによるタスクの処理　スティグマジからタスクを受け取ったワーカは，プロセスの処理を開始する．タスクが完了すると，ワーカはその処理結果をスティグマジに記録する．スティグマジに結果が記録されると，スティグマジは処理待ちタスク一覧から対応するタスクを削除し，処理済みタスク一覧へと結果を登録する．

Step 4. 処理済みタスクの統合　メディエータは定期的にスティグマジに対して処理済みタスクの一覧を問合せ，処理の状況を監視する．全てのタスクが完了したら処理結果の統合を行い，ジョブの処理結果とする．

7.4　関連研究

この小節では，提案システムに関連する分散システムについて調査し，議論する．

7.4.1 ハードウェア・ソフトウェアの協調設計ツール

Xilinx 社の SDAccel[114] や Intel 社の FPGA SDK for OpenCL [115] は FPGA ベンダが提供する協調設計ツールであり広く利用されているが，単一のホストコンピュータ上のリソースを使って開発するためのプラットフォームであるため，複数のホストコンピュータをまたいだ分散システムの構築には向かない．

Rabozzi らの開発した CAOS[116] は，FPGA を含むヘテロジニアスシステム上でアプリケーションを開発するためのプラットフォームである．ハードウェア構成を指定してベンチマークを取りインタラクティブにシステム全体のチューニングを行うワークフローを持つ．しかしながら単一のアプリケーションを実装するためのプラットフォームであり，複数のタスクが同時に実行されるヘテロジニアスシステムを想定しておらず，フレキシブルな SaaS や PaaS のクラウドコンピューティングシステムを実現することは困難である．

7.4.2 FPGA のクラウドサービスへの導入

Fahmy ら [117] は，FPGA を PaaS 型のクラウドサービスに導入するのに必要なフレームワークを提案した．FPGA の領域を分割し複数の仮想 FPGA (vFPGA) として提供することで複数のユーザが同時に FPGA を利用することが可能になった．ハイパーバイザ上でミドルウェアを動作させ，複数の vFPGA や vCPU を利用するアーキテクチャを持つ．しかしながら複数のホストコンピュータを横断したリソース管理を行うアーキテクチャではなく，また耐障害性や拡張性等の検討はなされていないため，実際に大量に導入するにはさらなる研究が必要である．

Chen ら [118] は，リソースプール上で LUT や BRAM 単位で FPGA のリソースを管理し，抽象化している．加えて OpenStack[119] を用いたホストコンピュータを横断したリソース管理を行っているため，IaaS 型のクラウドサービスを提供することが可能なアーキテクチャを持つ．本研究は PaaS 型のクラウドサービスをターゲットとしているため目的が異なるが，FPGA の抽象化の手法や KVM と OpenStack を用いたホストコンピュータの抽象化は非常に参考になる事例である．

これらのことから，次の点を考慮した仕組みを考案する必要がある．

- 複数の FPGA を含む計算ノードにまたがり分散システムを設計できること
- 複数種類のタスクが同時に実行され時間と共にリソース需要が変化することに耐えられること
- 耐障害性や拡張性を持つこと

既存の自律分散システム　これまでにも多くの自律分散型計算システムの研究がなされている．スティグマジのようなデータ共有の仕組みは既に提案されている [120]．そのほかの似た研究として，日立によるデータフィールドアーキテクチャ [121], [122] が挙げられる．このシステム中の「データフィールド」はスティグマジに相当し，「アトム」はワーカに相当する．アトムはデータフィールドで共有されているデータを処理する．この手法は，次のような目標を掲げている：

- 耐障害性
- オンライン拡張
- オンラインメンテナンス

我々の研究は反応閾値モデルに基づいており，データフィールドアーキテクチャのようなシステムに適用することで次のような特性を実現する：

- 自動スケーリング
- 適応的タスク配分
- タスクとジョブへのプライオリティの設定

自動スケーリング 反応閾値モデルを使用すると,「働き者」と「怠け者」のワーカが現れる．これは，θ_i が個体によって異なることに起因する．スティグマジに存在するタスク数が増えタスクの平均的な処理の待ち時間が増えるにつれて，反応閾値 θ_i の値が大きい「怠け者」ワーカも処理に参加するようになる．このようにして，処理を行うワーカを適応的に増減させることで，タスクの量に対して必要最小限のワーカの数を保つことができる．

適応的タスク配分 提案手法は，動的なロードバランシングのシステムである．Eagar らの研究グループは，静的ロードバランスと比較してよいパフォーマンスを発揮する動的ロードバランシングの手法を提案している [123]．対して本提案手法では，単一障害点となりうる制御ホストを排して動的ロードバランシングを実現することが可能である．

タスク/ジョブへの優先度の付与 タスクやジョブ処理の優先度を与えることについて議論する．例えば，式 (7.3) 中の定数 A をジョブの種別に応じて個別に割り当てるとする．このとき，A の値の大きいジョブは $P_{ij}(t)$ の増加が速く，結果的に優先的に処理されることとなる．

あるタスクの要求量が十分に大きい時，未処理の小さな要求量の多寡に関わらず「怠け者」のワーカも処理を開始する．このような設定の場合においては，文献 [121], [122] のような性質と同じような特徴を獲得する．

7.5 統計的モデルの利用

負荷共有アルゴリズム (Load Sharing Algorithm) は，分散システムに応じて優先度を実現する手法のひとつである．この手法では，ヘテロジニアスシステムに適用可能な動的負荷共有の仕組みを用いている．

しかしながら，負荷共有アルゴリズムはワーカ同士で交渉を行うが，システム全体が安定するために何度も交渉を繰り返す必要がある．それに対して，我々のシステムでは各ノードは $P_{ij}(t)$ にのみ基づいてタスクの処理をするかどうか判定するため，ワーカ間の通信がなく各ワーカが独立して動作する．

本手法は M/M/K 型のキューに基づいたモデルであるため解析が行いやすく，したがって実際の仕組みへの応用についてのアドバンテージがあるといえる．

課金額に応じた優先度の付与 テナントの支払う料金に応じて，優先度をタスクに付与することで，ユーザに提供する性能をコントロールすることが可能である．したがって課金体系は優先度に応じた料金と割り当てノード数の積とするのが有力となる．

表 7.1: Computers used for the experiment

	A	B
Machine	Raspberry Pi 2 Model B	Raspberry Pi 3 Model B
CPU	ARM Cortex-A7 4cores@900MHz	ARM Cortex-A53 4cores@1.2GHz
Memory	1GB	1GB
OS	Raspbian (Mar. 2017)	Raspbian (Mar. 2017)

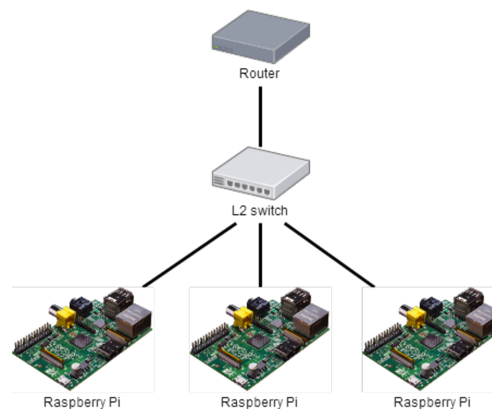


図 7.7: Network structure for the experiment

7.6 提案手法の検証

このセクションでは，実験を通して提案手法の効果検証を行う．この検証では，自動スケーリング，適応的タスク配分，及びタスク/ジョブへの優先度の割り当てについての検証を行う．

7.6.1 実装と実験の環境

実験システムは C++11 を用いて実装した．コンポーネント間の通信は TCP/IP を用いている．検証では，表 7.1 に示す 2 種類の計算機を用いた．この小節では，それぞれの環境を A，B と呼ぶ．

A の計算機をスティグマジとメディエータとして用い，10 個の B の計算機をワーカとして用いた．これらは図 7.7 に示すように接続されている．

7.6.2 実験モデル

システム構成 実験環境は，単一のスティグマジとメディエータ，及び複数のワーカから構成される．各計算機は 4 コアのプロセッサを搭載しているため，各計算機上で 4 つのワーカを個別のプロセスとして動作させている．

ジョブとタスクの設定　ジョブとして，RC4 暗号に対するブルートフォース攻撃を用いた．この実装の中では，ジョブは次に示すコンテンツによって構成される：

- キー長 (Byte) - N_k
- 分割数 - N_d
- 平文
- 暗号文

鍵の探索範囲は N_d 分割され各タスクに配分されることから，各タスクで復号の試行が行われるキーの数は $N_r = N_k/N_d$ となる．タスク j が発行された時刻 T_j がそれぞれのタスクの固有の情報 M_j として記録される．各タスクの内容は次に示す通りである：

- キー長 (Byte)
- 探索範囲の大きさ - N_r
- 平文
- 暗号文

各タスクの要求量の変化の仕方を表す関数 S は時刻に対して線形に変化する関数 $S(\{A_j, T_j\}, t) = t - T_j$ とした．A_j はタスク j に割り当てられた優先度を示している．

ワーカの設定　反応閾値 θ_i はワーカが起動された時点でワーカ自身が決定する．乱数には 32bit 浮動小数点数を用い，$\mu = 1, 0, \sigma^2 = 0.4$ の正規分布に従う，範囲 $[0, 2]$ の値である．

7.6.3　実験 1: 自動スケーリングの検証

実験手法　提案手法を適用しないシステムを **System-E1A** と呼び, 適用したシステムを **System-E1B** と呼ぶ．System-E1A は 30 ワーカ，System-E1B は 50 ワーカで構成される．ジョブが 30 分割され投入されたとき，10 秒後に平均 10 ワーカが「怠け者」として処理を行わず待機するようにパラメータをした．

始めに，50 タスクに分割されるジョブをひとつだけ投入した際にタスクを処理しているワーカの数の推移とジョブの処理に要した時間を計測した．次に，System-E1B において同様のジョブを二つ投入した際にタスクを処理しているワーカの数の推移を計測した．

実験結果　ジョブをひとつだけ投入した場合，ジョブの処理が完了するまでに System-E1A では 20 秒，System-E1B では 26 秒の時間を要した．図 7.8 左のグラフはタスク処理を行っているワーカの数の推移を示している．

図 7.8 右のグラフは，System-E1B において二つのジョブを投入した際の処理を行っているワーカの数の推移である．ジョブの処理は 20 秒の時点で完了しているが，それ以降も 35 のワーカが処理を続けていた．

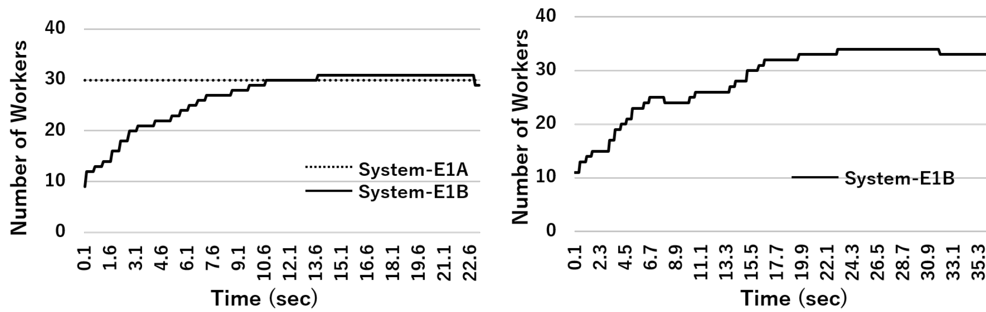


図 7.8: Number of Workers which is processing task for each 0.1sec of experiment 1 (Upper: a graph when give a Job, Lower: a graph when give two Jobs)

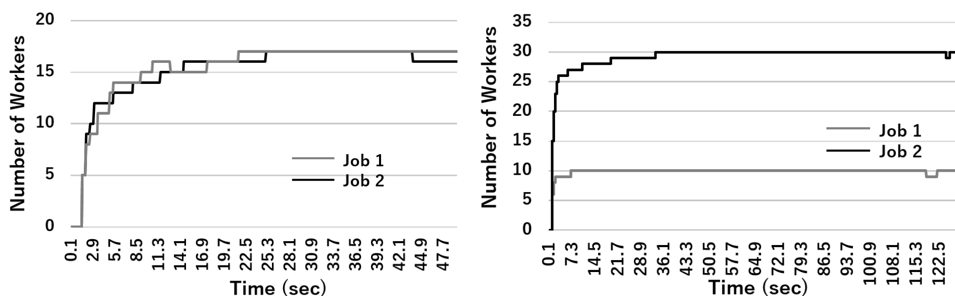


図 7.9: Number of Workers which is processing task for each 0.1sec of experiment 2 (Upper: System-E2A, Lower: System-E2B)

7.6.4 実験 2: 優先度の割り当て

実験方法 **System-E2A** と **System-E2B** は 40 のワーカで構成される．二つのシステムはともに System-E1B と同様に提案手法を適用したモデルであるが, タスクの要求量の関数を $S(\{A_j, T_j\}, t) = A_j \cdot (t - T_j)$ とし，優先度 A を用いて計算するように変更している．毎秒二つのジョブがそれぞれのシステムに投入される．各ジョブは 40 のタスクに分割される．System-E2A においては，毎秒投入される二つのジョブは同じ優先度を持つが，System-E2B においては，2 つめのジョブは 1 つめのジョブの 3 倍の大きさの優先度 A の値を持つ．小さい優先度を持つジョブを Job1，大きい優先度を持つジョブを Job2 と呼ぶ.

実験結果 図 7.9 に実験 2 の結果を示す．これらのグラフは，各ジョブに対して割り当てられたワーカの数の推移を示している．ジョブに同じ優先度を与えた System-E2A においては，それぞれのジョブに割り当てられたワーカ数は同様に推移している．それぞれのジョブに異なる優先度を与えた System-E2B においては，優先度 A の比と同様に割り当てられたワーカ数の比が 3:1 で安定した.

この実験から，優先度の設定が機能していることが確認できた.

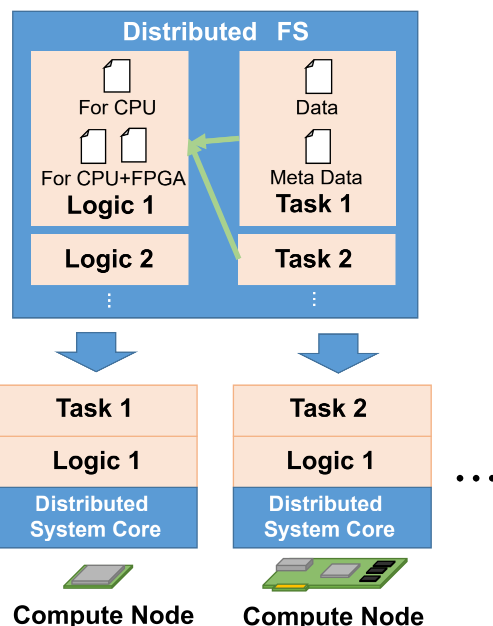


図 7.10: A distributed file system and compute nodes

7.7 クラウドシステムに向けたオートスケール機構

7.7.1 システム構成

システムは，コンパイラ，分散ファイルシステム，計算ノードの 3 種類のコンポーネントを持つ．

コンパイラは，Mulvery フレームワークを用いて記述されたユーザアプリケーションを解析し，アクセラレータを合成しプログラムと結合する．アクセラレータは仮想 FPGA に収まるサイズに分割され，同じ粒度で分割された Ruby プログラムを組として処理ロジックにパッケージ化される．Mulvery では既存の小さなモジュールを組み合わせてデータフローを構築するため，モジュールの回路規模から分割後の回路規模予測を行い効率的に回路分割を行う．この仕組みによって，処理ロジックは CPU のみの計算ノードでも実行可能となり，FPGA を混載した計算ノードではアクセラレータを利用することが可能となる．

分散ファイルシステムには，処理ロジックとは別に，処理されるデータを格納したタスクが格納される．タスクには，処理されるデータに加え，タスクの優先度や必要な処理ロジックの ID を格納したメタデータも格納される．

計算ノードは，それぞれの反応閾値に応じて処理するタスクを選定し，対応する処理ロジックとタスクをダウンロードする．処理ロジックをダウンロードした後は同じ処理ロジックで対応できるタスクを優先して処理するような学習係数 φ を持ち，頻繁に処理ロジックの変更が発生しないようにする．

処理の済んだタスクは，次の処理ロジックに進む必要がある場合には処理結果と共に次に遷移するべきタスク処理機構の ID を組にして新たなタスクとして分散共有ファイルシステムに戻される．

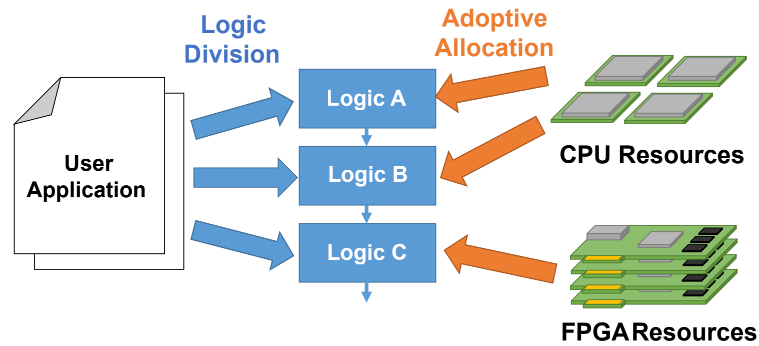


図 7.11: Source Code division and assignment computing resources

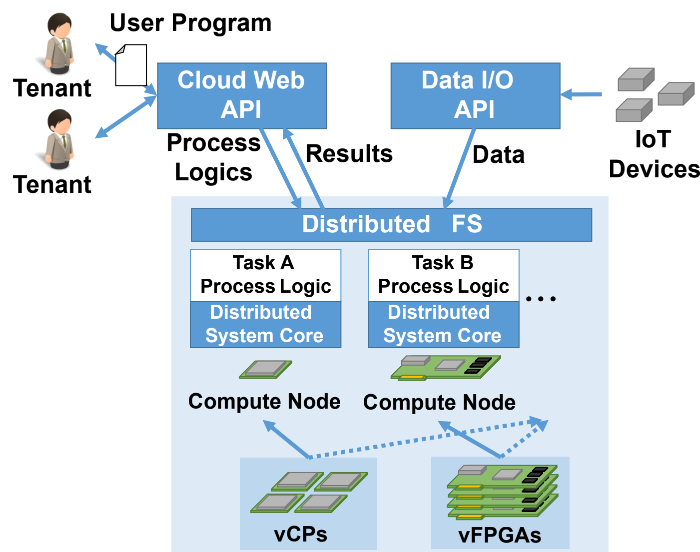


図 7.12: An Example of Architecture of PaaS Type IoT data analytics cloud service

7.7.2 PaaS 型データ分析クラウドサービス

応用例として，データが常に流入する IoT データを分析するためのプラットフォームの実装を例として考える（図 7.12）．プラットフォームはプログラムの実行環境を提供し，プラットフォームのユーザ（テナント）はデータ分析アルゴリズムを登録して処理を実行する．

スループットと課金額の関係　処理ロジックが 1 段で 20 秒かかる処理と，処理ロジックが 2 段でそれぞれが 10 秒で完了する処理に対しそれぞれ 2 秒毎に新しくタスクが発行されるようなとき，1 段の処理ロジックの方に多くのワーカが割り当てられる．この例のシミュレーションを行った結果（図 7.13），計算ノードの割り当て数の平均値の比はおよそ 2.3 倍となった．計算ノードの拘束時間が長いほどデータ損失時のリカバリコストが高まるため，自律分散システムがより冗長度を高く設定する傾向にある．

したがって，割り当てノード数に応じて課金額を決定することで，パイプライン化するほどスループットが高く課金額が安くなる課金体系をテナントに提供することが可能となる．

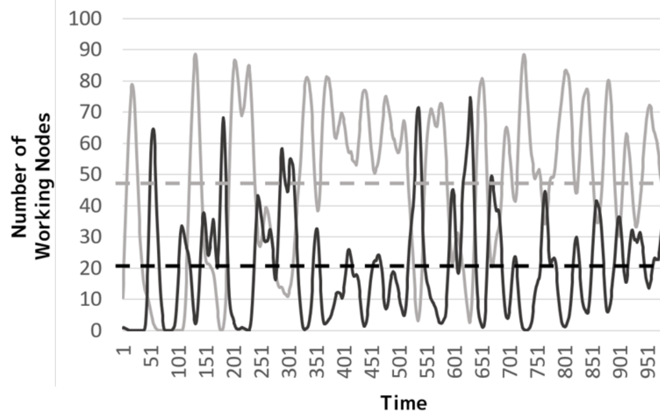


図 7.13: Numbers of assigned workers for each time (Gray line : 1 stage, Black line : 2 stages, Dotted line : average)

7.8 節のまとめ

この節では，CPU と FPGA を混載した分散システムにおいて，PaaS 型のクラウドサービスを提供するために必要となる技術の提案した．これまであまり議論されてこなかった複数種類のタスクが同時に実行される分散システムについて，高い拡張性と耐障害性を備える自律分散システムとして実現する手法を提案した．

ソフトウェア言語によってハードウェアを設計する高位合成の技術の特徴である CPU のみの計算ノードでも CPU+FPGA の計算ノードでも動作する点を応用すると，FPGA を含んだ自律分散システムを実現することが可能となる．

自律的な動的資源割り当ての手法として反応閾値モデルを用いることで，FPGA を計算リソース需要の高い順に割り当てを行うことが実現できることについて述べた．また，反応閾値モデルの工夫によってタスクそれぞれに対して優先度を付与することが可能であることを示した．実験によって，タスク間の優先度の比が割り当てられるワーカ数の比となり，設定が容易であることが示された．さらに，反応閾値モデルに学習係数を適用することで，FPGA の頻繁な再構成を避けオーバヘッドを低減することが可能であることについて述べた．

最後に，オートスケール機構の説明と，応用例についての検討を行った．データフロー型の記述を行う高位合成系を用いることによって回路規模の予測を容易なものとし，回路分割を行うことについて述べた．この回路分割の技術と先行研究 [117] [118] に示された FPGA の仮想化技術を応用することによって，より高い実装密度で FPGA を利用することが可能となる．これらの応用例として PaaS 型データ分析プラットフォームについて検討した．本提案のスループットとワーカ割り当て数の関係の分析と，その特徴を活かした料金体系についての分析を行った．結果，パイプライン化を行ったテナントのアプリケーションは，アルゴリズムに変更を加えることなく高いスループットを得つつ利用料金を引き下げることが可能であるという特徴的な料金体系が実現可能であることを示した．

第8章　結論

8.1　ディジタル回路設計における課題

1章では，これまでのディジタル回路設計における課題を整理し，本研究で取り組むリサーチクエスチョンを示した．FPGAによるハードウェアアクセラレーションが注目され，高性能計算から組み込みシステムまで様々な分野においてディジタル回路設計が必要となりつつある．ディジタル回路設計の生産性を高めるためにソフトウェア開発で用いられているプログラミング言語でビヘイビアレベルの記述をし，そこからRTL設計を合成するHLS技術の研究や実際の利用が増えている．しかしながらHLSツールにおいてもプログラミング言語の言語機能に制約がかかっているなど，生産性には向上の余地が残されていた．そこで本論文では，**ソフトウェア開発の分野で生産性を高めるために用いられているプログラミングモデルによってディジタル回路設計の生産性を高めることが可能か？** という問いに取り組んだ．

8.2　フレームワークによるアーキテクチャレベル最適化

4章では，フレームワークによってディジタル回路設計やSoC FPGAの開発に対する専門的知識を用いず自動的にアーキテクチャレベルの最適化を行う手法について説明した．ツールの用途を限定し開発をモデル化することで既存の自動最適化手法やツールの部分的な適用が容易となり，ユーザが専門的知識による手動の最適化を行う必要がなくなり生産性を高めることができると示した．提案したフレームワークはIoT/CPSにおけるエッジデバイスの開発などに用いることを想定したフレームワークであった．分野や用途が変化すれば都度新たなフレームワークを開発する必要があるため，ディジタル回路設計全体を見渡す視点においてはスケーラビリティに欠いた手法であるといえる．

8.3　ディジタル回路の自動設計に適した言語パラダイム

5章では，GUI設計などの領域で用いられているプログラミングモデルであるFRPを用いたディジタル回路合成について提案した．筆者はCやJavaなどを用いるこれまでのHLSツールにおいて専門知識に基づく最適化やサポートできない言語機能の存在はプログラミング言語とディジタル回路設計のパラダイムが異なるものであるからであると考えている．FRPはHDLと同じデータフロー型の記述であり，そこでプログラミング言語とHDLがパラダイムを一にするモデルがあることに着目した．FRPに基づくプログラムはディジタル回路合成の際の最適化に必要なデータフローや制御フローの並列性の情報を保持しており，プログラムの解析と最適化の手間を削減し生産性を高めることができると示した．FRPに基づくハードウェア・ソフトウェア協調設計環境として筆者が開発したツールMulveryはソフトウェア開発で広く用いられているAPIを提供し，4章で提案したドメイン特化型のフレームワークとは異なり幅広いアプリケーションに対して応用可能なツールとして実現した．

8.4 C 言語における関数ポインタと高階関数のサポート

6 章では，HLS ツールにおいてサポートが困難とされていた C 言語における関数ポインタのサポートの手法について説明した．Xilinx や Intel など大手 FPGA ベンダの HLS ツールをはじめ学術界では LegUp など様々な HLS ツールは記述言語に C 言語を採用している．C 言語において関数ポインタは，高階関数といった抽象度の高いライブラリを提供するために不可欠の言語機能である．候補値探索によって網羅的で過不足のないモジュール間ネットワークを構成することで，回路面積や実行速度に対する影響を最小限に留めた上で動的な関数ポインタのサポートを実現した．

8.5 データフロー型プログラムの分散 FPGA 環境への応用

7 章では，Mulvery のようなデータフロー型プログラムを用いて分散 FPGA 環境を実現するための手法について説明した．単一の FPGA に回路が収まらず複数の FPGA に分割するような場合は，HLS や HDL では FPGA ごとに個別に設計する必要があった．データフロー型モデルで記述されたプログラムは複数 FPGA への分割が容易であるため，このような記述において複数の FPGA で PaaS 型のクラウドコンピューティングを実現するための要素技術検証を行った．自律的な動的資源割り当て手法として反応閾値モデルを用いることで，計算資源の需要の大きい問題に対し優先的に FPGA を割り当てることができるようになった．また，本来プログラムとして記述されているため，FPGA のリソースが枯渇している場合などに CPU で計算を行うなど，ヘテロジニアスな環境でのコードのポータビリティの高い記述もハードウェアとソフトウェアを同じパラダイムのモデルで設計するメリットである．

本研究では，プログラミングモデルの工夫によってディジタル回路設計の生産性を高めることが可能であるか確かめてきた．特に重要な研究成果としては，回路設計と共通のパラダイムを持つプログラミングモデルを用いてプログラミングを行うことで最適化の効率化や再利用性の向上，CPU-FPGA 間でのコードのポータビリティなど様々な利益をもたらすことを示した．FRP のようなモデルは GUI 設計などを中心に用いられており，残念ながら高性能計算や組み込みデバイス開発などの場面で多く用いられているとは言い難い．再構成可能ハードウェアとデータフロー型プログラムを用いた高性能計算のための分散コンピューティング技法などの有用な社会実装の研究を進めていくことによって，より社会にとって有益な研究となることを目指していく．

Bibliography

[1] *The Basics / chisel-tutorial*. https://github.com/ucb-bar/chisel-tutorial/wiki/The-Basics, Referenced Jan. 2019.

[2] Gèrard Berry and Georges Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(92)90005-V. URL: http://www.sciencedirect.com/science/article/pii/016764239290005V.

[3] *Edit Distance / JHDL*. http://www.jhdl.org/documentation/latestdocs/code/JHDL_examples2.html, Referenced Jan. 2019.

[4] *Clash.Examples*. http://hackage.haskell.org/package/clash-prelude-0.99.3/docs/Clash-Examples.html, Referenced Jan. 2019.

[5] Shunning Jiang, Christopher Torng, and Christopher Batten. "An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework". In: *the Proceedings of the First Workshop on Open-Source EDA Technology (WOSET ' 18)* 13 (Nov. 2018). San Diego, CA, USA, pp. 1–5.

[6] "MaxCompiler White Paper". In: Maxeller Technologies, Feb. 2011.

[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. "Dark Silicon and the End of Multicore Scaling". In: *IEEE Micro* 32.3 (May 2012), pp. 122–134. ISSN: 0272-1732. DOI: 10.1109/MM.2012.17.

[8] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". In: *Eurographics 2005 - State of the Art Reports*. Ed. by Yiorgos Chrysanthou and Marcus Magnor. The Eurographics Association, 2005. DOI: 10.2312/egst.20051043.

[9] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *arXiv*. 2017. URL: https://arxiv.org/pdf/1704.04760.pdf.

[10] H.Kung. “Why systolic architectures?” In: *Computer* 15.1 (1982), pp. 37–46. DOI: 10.1109/MC.1982.1653825.

[11] 梅尾博司. “シストリック・アーキテクチャとそのアルゴリズム”. In: **オペレーションズ・リサーチ** 37.8 (1992), pp. 375–381.

[12] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, algorithms and applications*. Kluwer Academic Publishers, Jan. 2007. DOI: 10.1007/978-1-4020-6100-4.

[13] Prasanna Sundararajan. “High Performance Computing Using FPGAs”. In: *Xilinx White Papter: FPGAs* (2010), pp. 1–15.

[14] I. Kuon, R. Tessier, and J. Rose. *FPGA Architecture: Survey and Challenges*. Now Foundations and Trends, 2008. DOI: 10.1561/1000000005.

[15] 天野英晴. *FPGA* **の原理と構成**. オーム社, 2016. ISBN: 9784274218644. URL: https://books.google.co.jp/books?id=yh7CjwEACAAJ.

[16] E. Ahmed and J. Rose. “The effect of LUT and cluster size on deep-submicron FPGA performance and density”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.3 (2004), pp. 288–298. DOI: 10.1109/TVLSI.2004.824300.

[17] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. “Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.11 (2006), pp. 2331–2340. DOI: 10.1109/TCAD.2006.882119.

[18] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner. “Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains”. In: *IEEE Access* 8 (2020), pp. 174692–174722. DOI: 10.1109/ACCESS.2020.3024098.

[19] Scott W. Ambler and Mark Lines, eds. *Disciplined Agile Delivery - A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press, 2012.

[20] MIT Artificial Intelligence Lab Dynamic Languages Group. *Call for Participation of 'LL1: Lightweight Languages Workshop'*. http://ll1.ai.mit.edu/, Referenced Jul. 2018.

[21] Stephen Cass. “The 2017 Top Programming Languages”. In: *IEEE Spectrum*. July 2017.

[22] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[23] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. “Chisel: Constructing hardware in a Scala embedded language”. In: *DAC Design Automation Conference 2012*. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[24] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. “An Introduction to High-Level Synthesis”. In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 8–17. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.69.

[25] Dennis. “Data Flow Supercomputers”. In: *Computer* 13.11 (Nov. 1980), pp. 48–56. ISSN: 0018-9162. DOI: 10.1109/MC.1980.1653418.

[26] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016), pp. 1591–1604. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2513673.

[27] Luka Daoud, Dawid Zydek, and Henry Selvaraj. “A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing”. In: *Advances in Systems Science*. Cham: Springer International Publishing, 2014, pp. 483–492. ISBN: 978-3-319-01857-7.

[28] G. Inggs, S. Fleming, D. Thomas, and W. Luk. “Is high level synthesis ready for business? A computational finance case study”. In: *2014 International Conference on Field-Programmable Technology (FPT)*. Dec. 2014, pp. 12–19. DOI: 10.1109/FPT.2014.7082747.

[29] *Matlab / MathWorks*. https://jp.mathworks.com/products/matlab.html, Referenced Jan. 2019.

[30] *Simulink / MathWorks*. https://jp.mathworks.com/products/simulink.html, Referenced Jan. 2019.

[31] Jerker Hammarberg and Simin Nadjm-Tehrani. “Development of Safety-Critical Reconfigurable Hardware with Esterel”. In: *Electronic Notes in Theoretical Computer Science* 80 (2003). Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS’03), pp. 219–234. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)80820-X. URL: http://www.sciencedirect.com/science/article/pii/S157106610480820X.

[32] *CEC: The Columbia Estrel Compiler*. http://www1.cs.columbia.edu/~sedwards/cec/, Referenced Jan. 2019.

[33] P. Bellows and B. Hutchings. “JHDL-an HDL for reconfigurable systems”. In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*. Apr. 1998, pp. 175–184. DOI: 10.1109/FPGA.1998.707895.

[34] James Jennings and Eric Beuscher. “Verischemelog: Verilog embedded in Scheme”. In: *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)* 35 (Apr. 1999). DOI: 10.1145/331963.331978.

[35] D. I. Rich. “The Evolution of Systemverilog”. In: *IEEE Des. Test* 20.4 (July 2003), pp. 82–84. ISSN: 0740-7475. DOI: 10.1109/MDT.2003.1214355. URL: https://doi.org/10.1109/MDT.2003.1214355.

[36] “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language”. In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (Feb. 2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[37] I. Sander and A. Jantsch. “System Modeling and Transformational Design Refinement in ForSyDe [Formal System Desig]”. In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 23.1 (Nov. 2006), pp. 17–32. ISSN: 0278-0070. DOI: 10.1109/TCAD.2003.819898. URL: http://dx.doi.org/10.1109/TCAD.2003.819898.

[38] Rishiyur Nikhil. “Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions”. In: Jan. 2008, pp. 129–146. ISBN: 978-1-4020-8587-1. DOI: 10.1007/978-1-4020-8588-8_8.

[39] C.P.R. Baaij, Matthijs Kooijman, Jan Kuper, W.A. Boeijink, and Marco Egbertus Theodorus Gerards. “CλaSH: Structural Descriptions of Synchronous Hardware using Haskell”. In: *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*. eemcs-eprint-18376. United States: IEEE Computer Society, Sept. 2010, pp. 714–721. ISBN: 978-0-7695-4171-6. DOI: 10.1109/DSD.2010.21.

[40] Rinse Wester and Jan Kuper. “Design Space Exploration of a Particle Filter Using Higher-Order Functions”. In: *Reconfigurable Computing: Architectures, Tools, and Applications*. Ed. by Diana Goehringer, Marco Domenico Santambrogio, João M. P. Cardoso, and Koen Bertels. Cham: Springer International Publishing, 2014, pp. 219–226. ISBN: 978-3-319-05960-0.

[41] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[42] freechipsproject. *Rocket Chip Generator*. https://github.com/freechipsproject/rocket-chip, Referenced Jul. 2018.

[43] D. Lockhart, G. Zibrat, and C. Batten. “PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 280–292. DOI: 10.1109/MICRO.2014.50.

[44] F. Vahid, G. Stitt, and R. Lysecky. “Warp Processing: Dynamic Translation of Binaries to FPGA Circuits”. In: *Computer* 41.7 (July 2008), pp. 40–46. ISSN: 0018-9162. DOI: 10.1109/MC.2008.240.

[45] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. “LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems”. In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’11. Monterey, CA, USA: ACM, 2011, pp. 33–36. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423. URL: http://doi.acm.org/10.1145/1950413.1950423.

[46] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. *See* `http://llvm.cs.uiuc.` MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[47] Chris Lattner and Vikram Adve. “The LLVM Compiler Framework and Infrastructure Tutorial”. In: *LCPC’04 Mini Workshop on Compiler Research Infrastructures.* West Lafayette, Indiana, Sept. 2004.

[48] S. Bansal, H. Hsiao, T. Czajkowski, and J. H. Anderson. “High-level synthesis of software-customizable floating-point cores”. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE).* Mar. 2018, pp. 37–42. DOI: `10.23919/DATE.2018.8341976`.

[49] O. Ragheb and J. H. Anderson. “High-Level Synthesis of FPGA Circuits with Multiple Clock Domains”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* Apr. 2018, pp. 109–116. DOI: `10.1109/FCCM.2018.00026`.

[50] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. “Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines”. In: *ACM Trans. Graph.* 33.4 (July 2014), 144:1–144:11. ISSN: 0730-0301. DOI: `10.1145/2601097.2601174`. URL: `http://doi.acm.org/10.1145/2601097.2601174`.

[51] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *SIGPLAN Not.* 48.6 (June 2013), pp. 519–530. ISSN: 0362-1340. DOI: `10.1145/2499370.2462176`. URL: `http://doi.acm.org/10.1145/2499370.2462176`.

[52] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. “Programming Heterogeneous Systems from an Image Processing DSL”. In: *CoRR* abs/1610.09405 (2016). arXiv: `1610.09405`. URL: `http://arxiv.org/abs/1610.09405`.

[53] fixstars. *Halide-elements.* `https://github.com/fixstars/Halide-elements`, Referenced Jan. 2019.

[54] Intel. *Intel High Level Synthesis Compiler.* `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`, Reference Feb. 2019.

[55] Xilinx. *Vivado HLS.* `http://japan.xilinx.com/products/design-tools/vivado/integrationesl-design.html`, Reference Feb. 2019.

[56] G. Jones and M. Sheeran. “Circuit design in Ruby”. In: *Formal Methods for VLSI Design.* Elsevier Publishers, 1990, 3a.3.1–3a.3.10.

[57] *Ruby - A Programmer’s best friend.* `https://www.ruby-lang.org/en/`, Referenced Jan. 2019.

[58] H.T. Kung. “Why systolic architectures?” In: *IEEE Computer* 15.1 (1982), pp. 37–46.

[59] W.W.C. Luk. “Systematic serialisation of array-based architectures”. In: *Integration* 14.3 (1993), pp. 333–360. ISSN: 0167-9260. DOI: https://doi.org/10.1016/0167-9260(93)90014-4. URL: http://www.sciencedirect.com/science/article/pii/0167926093900144.

[60] Yanbing Li and M. Leeser. “HML, a novel hardware description language and its translation to VHDL”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.1 (Feb. 2000), pp. 1–8. ISSN: 1063-8210. DOI: 10.1109/92.820756.

[61] Yanbing Li and M. Leeser. “HML: an innovative hardware description language and its translation to VHDL”. In: *Proceedings of ASP-DAC’95/CHDL’95/VLSI’95 with EDA Technofair*. Aug. 1995, pp. 691–696. DOI: 10.1109/ASPDAC.1995.486388.

[62] *HDL Coder / MathWorks*. https://mathworks.com/products/hdl-coder.html, Referenced Jan. 2019.

[63] MathWorks. *Rapid Prototyping Using HDL Coder*. https://mathworks.com/videos/rapid-prototyping-using-hdl-coder-119978.html, Referenced Feb. 2019.

[64] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. D. Santambrogio. “Heterogeneous exascale supercomputing: The role of CAD in the exaFPGA project”. In: *Design, Automation Test in Europe Conference Exhibition （DATE), 2017*. Mar. 2017, pp. 410–415. DOI: 10.23919/DATE.2017.7927025.

[65] 三好 健文 and 船田 悟史. “FPGA 向け高位合成言語としての Java の活用手法の検討”. In: **第** *53* **回プログラミング・シンポジウム予稿集** 2012 (Jan. 2012), pp. 59–68. URL: http://ci.nii.ac.jp/naid/170000071251/.

[66] 小池 恵介, 三好 健文, 船田 悟史, and 中條 拓伯. “Java 言語ベース高位合成ツール JavaRock-Thrash の開発”. In: **組込みシステムシンポジウム** *2013* **論文集** 2013 (Oct. 2013), pp. 41–48. URL: http://ci.nii.ac.jp/naid/170000078511/.

[67] 小池 恵介, 三好 健文, 五十嵐 雄太, 船田 悟史, and 中條 拓伯. “Java 言語ベース高位合成ツールによるアクセラレータ開発環境”. In: **電子情報通信学会論文誌** *D* J98-D.3 (Oct. 2015), pp. 373–383. ISSN: 1881-0225. URL: http://ci.nii.ac.jp/naid/170000078511/.

[68] Takefumi Miyoshi. *Synthesijer*. http://synthesijer.github.io/web/, Referenced Jul. 2018.

[69] Daichi Teruya. *PyJer*. https://github.com/maruuusa83/pyjer, 参照 Jul. 2016.

[70] 菊谷 雄真, Tran Thi Hong, and 高前田 伸也. “高位合成ツール Vivado HLS と PyCoRAM を用いた FPGA アクセラレータの性能比較”. In: **電子情報通信学会技術研究報告** *= IEICE technical report :* **信学技報** 115.342 (Dec. 2015), pp. 27–32. ISSN: 0913-5685. URL: http://ci.nii.ac.jp/naid/40020700462/.

[71] 植竹 大地, 大川 猛, 三好 健文, 横田 隆史, and 大津 金光. “高位合成ツール JavaRock による倒立振子制御処理の高速化 (高位合成と開発環境, リコンフィギャラブルシステム, 一般)”. In: **電子情報通信学会技術研究報告**. *RECONF,* **リコンフィギャラブルシステム** 113.221 (Sept. 2013), pp. 55–60. ISSN: 0913-5685. URL: http://ci.nii.ac.jp/naid/110009783140/.

[72] Julien HENAUT, Daniela DRAGOMIRESCU, and Robert PLANA. “FPGA Based High Date Rate Radio Interfaces for Aerospace Wireless Sensor Systems”. In: *2010 Fifth International Conference on Systems* (Mar. 2009), pp. 173–178. ISSN: 0913-5685. DOI: 10.1109/ICONS.2009.28.

[73] Shinya Takamaeda-Yamazaki, Kenji Kise, and James C. Hoe. “PyCoRAM: Yet Another Implementation of CoRAM Memory Architecture for Modern FPGA-based Computing”. In: *Intersections of Computer Architecture and Reconfigurable Logic (CARL 2013)*. Davis, CA, Dec. 2013.

[74] Eric S. Chung, James C. Hoe, and Ken Mai. “CoRAM: An In-fabric Memory Architecture for FPGA-based Computing”. In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’11. Monterey, CA, USA: ACM, 2011, pp. 97–106. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950435. URL: http://doi.acm.org/10.1145/1950413.1950435.

[75] 浅野太, ed. *2* **群** *6* **編** *3* **章 音源定位**, **知識の森**. 電子情報通信学会, 2011.

[76] Daichi Teruya, Daichi Miyazaki, and Hironori Nakajo. “A Sound Field Visualizer with Java-based High Level Synthesis Tool and CoRAM Architecture Synthesis Framework”. In: *IEICE technical report (in Japanese)* 116.53 (May 2016), pp. 97–102. ISSN: 0913-5685. URL: http://ci.nii.ac.jp/naid/40020849704/.

[77] Daichi Teruya, Daichi Miyazaki, and Hironori Nakajo. “PyJer : A Framework for Prototyping of IoT Devices with High Level Synthesis Tools and SoC”. In: *Transaction D (in Japanese)* J100-D.3 (Mar. 2017), pp. 287–297.

[78] W.C. Cheng, C.-Fu. Chou, L. Golubchik, S. Khuller, and Y.-C. Wan. “Large-scale data collection: a coordinated approach”. In: *IEEE INFOCOM 2003* (Apr. 2003). ISSN: 0743-166X. DOI: 10.1109/INFCOM.2003.1208674.

[79] Kai-Yin Fok Chi-Tsun Cheng Nuwan Ganganath. “Large-scale data collection: a coordinated approach”. In: *IEEE Transactions on Industrial Informatics* (Sept. 2016). ISSN: 1941-0050. DOI: 10.1109/TII.2016.2610139.

[80] fluentd. *fluentd*. http://www.fluentd.org/, 参照 Dec. 2016.

[81] fluent bit. *fluent bit*. http://fluentbit.io/, 参照 Dec. 2016.

[82] Daichi Teruya and Hironori Nakajo. “Overview of an HLS Framework Surpporting IoT/CPS Development”. In: *IEICE technical report (in Japanese)* 116.417 (Jan. 2017), pp. 61–66. ISSN: 0913-5685.

[83] Takefumi Miyoshi. *synthesijer.scala*. http://github.com/synthesijer/synthesijer.scala, Referenced Jul. 2018.

[84] C.P.R. Baaij, Matthijs Kooijman, Jan Kuper, W.A. Boeijink, and Marco Egbertus Theodorus Gerards. “CλaSH: Structural Descriptions of Synchronous Hardware using Haskell”. In: *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*. eemcs-eprint-18376. United States: IEEE Computer Society, Sept. 2010, pp. 714–721. ISBN: 978-0-7695-4171-6. DOI: 10.1109/DSD.2010.21.

[85] Shinya Takamaeda-Yamazaki. “Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL”. In: *Applied Reconfigurable Computing*. Vol. 9040. Lecture Notes in Computer Science. Springer International Publishing, Apr. 2015, pp. 451–460. DOI: 10.1007/978-3-319-16214-0_42. URL: http://dx.doi.org/10.1007/978-3-319-16214-0_42.

[86] *NumPy*. http://www.numpy.org/, 参照 Jan. 2018.

[87] *CuPy*. https://cupy.chainer.org/, 参照 Jan. 2018.

[88] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. “A Survey on Reactive Programming”. In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34. ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: http://doi.acm.org/10.1145/2501654.2501666.

[89] Jessie Liberty and Paul Betts, eds. *Programming Reactibe Extensions and LINQ*. Apress, 2011.

[90] *ReactiveX*. http://reactivex.io/, Referenced Feb. 2019.

[91] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, eds. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[92] Conal Elliott and Paul Hudak. “Functional Reactive Animation”. In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/.

[93] Henrik Nilsson, Antony Courtney, and John Peterson. “Functional reactive programming, continued”. In: *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop* (Jan. 2002). DOI: 10.1145/581690.581695.

[94] Erik Meijer, Brian Beckman, and Gavin Bierman. “LINQ: Reconciling Object, Relations and XML in the .NET Framework”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. Chicago, IL, USA: ACM, 2006, pp. 706–706. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142552. URL: http://doi.acm.org/10.1145/1142473.1142552.

[95] Walid Taha. “A Gentle Introduction to Multi-stage Programming”. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50. ISBN: 978-3-540-25935-0. DOI: 10.1007/978-3-540-25935-0_3. URL: https://doi.org/10.1007/978-3-540-25935-0_3.

[96] LIMITED WORLDSEMI CO. *WS2812B Intelligent control LED integrated light source (datasheet)*. 参照 Jan. 2018.

[97] Digilent. *ZYBO FPGA Board Reference Manual*. Reviced April 11, 2016, 参照 Jan. 2018.

[98] D. Richmond, A. Althoff, and R. Kastner. “Synthesizable Higher-Order Functions for C++”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (Nov. 2018), pp. 2835–2844. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2857259.

[99] M. Minutoli, V. G. Castellana, A. Tumeo, and F. Ferrandi. "Inter-procedural resource sharing in High Level Synthesis through function proxies". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7293958.

[100] M. Minutoli. "An Architecture for Function Pointers and Non-inlined Function Call in High-level Synthesis". MA thesis. Politecnico di Milano, 2013.

[101] L. Semeria and G. De Micheli. "Resolution, optimization, and encoding of pointer variables for the behavioral synthesis from C". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.2 (Feb. 2001), pp. 213–233. DOI: 10.1109/43.908442.

[102] L. Semeria, K. Sato, and G. De Micheli. "Synthesis of hardware models in C with pointers and complex data structures". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.6 (Dec. 2001), pp. 743–756. DOI: 10.1109/92.974889.

[103] Amazon Web Service. *EF2 F1 Instances*. Website https://aws.amazon.com/ec2/instance-types/f1/. Referenced Apr. 2018.

[104] Marco Dorigo, Eric Bonabeau, and Guy Theraulaz. "Ant algorithms and stigmergy". In: *Future Generation Computer Systems* 16.8 (2000), pp. 851–871. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/S0167-739X(00)00042-X. URL: http://www.sciencedirect.com/science/article/pii/S0167739X0000042X.

[105] Eric Bonabeau, Andrej Sobkowski, Guy Theraulaz, and Jean-Louis Deneubourg. *Adaptive Task Allocation Inspired by a Model of Division of Labor in Social Insects*. Working Papers 98-01-004. Santa Fe Institute, Jan. 1998. URL: https://ideas.repec.org/p/wop/safiwp/98-01-004.html.

[106] Yasunori Ishii and Eisuke Hasgeawa. "The mechanism underlying the regulation of work-related behaviors in the monomorphic ant, Myrmica kotokui". In: *Journal of Ethology* 31.1 (Jan. 2013), pp. 61–69. ISSN: 1439-5444. DOI: 10.1007/s10164-012-0349-6. URL: https://doi.org/10.1007/s10164-012-0349-6.

[107] M. Dorigo and L. M. Gambardella. "Ant colony system: a cooperative learning approach to the traveling salesman problem". In: *IEEE Transactions on Evolutionary Computation* 1.1 (Apr. 1997), pp. 53–66. ISSN: 1089-778X. DOI: 10.1109/4235.585892.

[108] T. Liao, K. Socha, M. A. Montes de Oca, T. Stützle, and M. Dorigo. "Ant Colony Optimization for Mixed-Variable Optimization Problems". In: *IEEE Transactions on Evolutionary Computation* 18.4 (Aug. 2014), pp. 503–518. ISSN: 1089-778X. DOI: 10.1109/TEVC.2013.2281531.

[109] Grosan Crina and Abraham Ajith. "Stigmergic Optimization: Inspiration, Technologies and Perspectives". In: *Stigmergic Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–24. ISBN: 978-3-540-34690-6. DOI: 10.1007/978-3-540-34690-6_1. URL: https://doi.org/10.1007/978-3-540-34690-6_1.

[110] J. O. Kephart and D. M. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (Jan. 2003), pp. 41–50. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1160055.

[111] Kirstin Petersen, Radhika Nagpal, and Justin Werfel. "TERMES: An Autonomous Robotic System for Three-Dimensional Collective Construction". In: *Robotics: Science and Systems*. 2011.

[112] Hugh F. Durrant-Whyte, Nicholas Roy, Justin Werfel, Kirstin Petersen, and Radhika Nagpal. "Distributed Multi-Robot Algorithms for the TERMES 3D Collective Construction System". In: 2011.

[113] E. Bonabeau, G. Theraulaz, and J.-L. Deneubourg. "Quantitative study of the fixed threshold model for the regulation of division of labor in insect societies". In: *Proc. Roy. Soc. London B 263* (1996), pp. 1565–1569.

[114] Xilinx. *SDAccel*. Website `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`. Referenced Apr. 2018.

[115] Intel FPGA SDK for OpenCL. *maruuusa83/arisan*. Website `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`. Referenced Apr. 2018.

[116] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. D. Santambrogio. "Heterogeneous exascale supercomputing: The role of CAD in the exaFPGA project". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. Mar. 2017, pp. 410–415. DOI: `10.23919/DATE.2017.7927025`.

[117] S. A. Fahmy, K. Vipin, and S. Shreejith. "Virtualized FPGA Accelerators for Efficient Cloud Computing". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Nov. 2015, pp. 430–435. DOI: `10.1109/CloudCom.2015.60`.

[118] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. "Enabling FPGAs in the Cloud". In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. CF '14. Cagliari, Italy: ACM, 2014, 3:1–3:10. ISBN: 978-1-4503-2870-8. DOI: `10.1145/2597917.2597929`. URL: `http://doi.acm.org/10.1145/2597917.2597929`.

[119] Opan Stack. *Open source software for building private and pulblic clouds*. Website `https://www.openstack.org/`. Referenced Apr. 2018.

[120] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty". In: *ACM Comput. Surv.* 12.2 (June 1980), pp. 213–253. ISSN: 0360-0300. DOI: `10.1145/356810.356816`. URL: `http://doi.acm.org/10.1145/356810.356816`.

[121] Kinji Mori, Hirokazu Ihara, Katsumi Kawano, Minoru Koizumi, Masayuki Orimo, Kozo Nakai, Hiroaki Nakanishi, and Yasuo Suzuki. "Autonomous Decentralized Software Structure and Its Application". In: *Proceedings of 1986 ACM Fall Joint Computer Conference*. ACM '86. Dallas, Texas, USA: IEEE Computer Society Press, 1986, pp. 1056–1063. ISBN: 0-8186-4743-4. URL: `http://dl.acm.org/citation.cfm?id=324493.325044`.

[122] K. Mori. "Autonomous decentralized systems: Concept, data field architecture and future trends". In: *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems*. 1993, pp. 28–34. DOI: `10.1109/ISADS.1993.262725`.

[123] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. "Adaptive Load Sharing in Homogeneous Distributed Systems". In: *IEEE Trans. Softw. Eng.* 12.5 (May 1986), pp. 662–675. ISSN: 0098-5589. URL: `http://dl.acm.org/citation.cfm?id=5527.5535`.

謝辞

本論文は，東京農工大学工学府博士前期課程情報工学専攻，および同学府博士後期課程電子情報工学専攻知能・情報工学専修に在籍した中での研究成果をまとめたものです．指導教官として主査を引き受けて下さった同大学工学研究院先端情報科学部門准教授の中條拓伯先生に深い感謝の意を表します．博士前期，後期課程を通して研究の初歩から博士学位論文の執筆まで，長期にわたって様々な場面でご指導を賜りました．筆者が研究の進め方や論文の構成などに行き詰まった時には中條先生は筆者との議論にとことんお付き合い下さり，また中條先生の幅広い研究交流関係から多くの方を紹介していただき，都度研究の新たな視点や方向性を獲得してきました．中條先生の助言と手厚いサポートによって本研究を遂行することができました．

本論文の提出にあたって，東京農工大学工学研究院先端情報科学部門の教授金子敬一先生，教授近藤敏之先生，教授藤田欣也先生，教授藤波香織先生の四名の先生に副査をお願いし，数多くのご助言を賜りました．深い感謝の意を表します．

4 章ならびに 5 章で説明したツール PyJer, Mulvery の研究と開発にあたっては，研究会や高位合成友の会等々で度々お会いした東京大学情報理工学系研究科情報科学科准教授の高前田伸也先生と，わさらぼ/イーツリーズ・ジャパンの三好健文博士から大変多くのご助言を賜ました．ここに深謝申し上げます．

5 章で説明した Mulvery の開発の一部は，独立行政法人情報処理推進機構 未踏 IT 人材発掘・育成事業に採択され，当事業に係る業務委託として推進しました．東京工業大学情報理工学院准教授の首藤一幸先生はまだアイデア段階だった Mulvery の提案に期待を寄せて下さり，プロジェクトマネージャを担当してくだいました．また，富士通クラウドテクノロジーズ株式会社（当時）の呉屋寛裕氏は，筆者と共に未踏事業でプロジェクトメンバーとして取り組んでくれました．お二方にはここに深謝の意を表します．このほか合宿で共に深い議論をしてくださった PM の方々，未踏'17 年度の同期，OBOG の皆様，未踏事業とプロジェクトを支えていらっしゃる未踏事務局の皆様にも非常に感謝しております．

6 章で説明した関数ポインタの削除に関する研究においては，University of Toronto, Depertment of Electrical and Computer Engineering, Professor の Jason H. Anderson 先生に大変お世話になりました．本研究の一部は日本学術振興会若手研究者海外挑戦プログラムに採択され，Anderson 先生はその受入研究者を引き受けてくださいました．遠い異国の学生でも高位合成研究で著名なツール LegUp の研究に関わる機会を与えてくださった Anderson 先生には感謝の念に堪えません．

7 章で説明した反応閾値モデルをもとにした分散計算の研究においては，独立行政法人国立高等専門学校機構沖縄工業高等専門学校教授（当時）の正木忠勝先生から多大なご指導を賜りました．7 章で説明した研究は，筆者が正木研究室に所属していた際に行っていた研究を基礎に本研究の内容へと発展させたものです．ここに深い感謝の意を表します．

そして，日常から多くの議論を通して様々な知識やアイデアを頂いた中條研究室の皆様にも感謝を申し上げます．筆者と同じく高位合成やハードウェア設計支援に関するテーマについて取り組む同研究室博士前期課程の山下遼太君とは特に多くの議論を交え，より研究を洗練することができたと思っています．さらに，物品の購入など，事務の面から研究活動を支援していただきま

した秘書室の沖田明子さん，金子美津子さんに感謝いたします．皆様のご支援のおかげで研究を大変円滑に推進ことができました．ここにお礼を申し上げます．

本研究の一部は日本学術振興会科学研究費特別研究員奨励費 18J22381 の助成を受けています．また，本研究の一部は「独立行政法人情報処理推進機構未踏 IT 人材発掘・育成事業」に係る業務委託，および「日本学術振興会 若手研究者海外挑戦プログラム」による成果です．経済的なサポートを与えてくださった独立行政法人学術振興会ならびに独立行政法人情報処理推進機構 未踏 IT 人材発掘・育成事業に深謝の意を表します．また研究成果の発表の場を与えて下さいました Okinawa.rb および一般社団法人 日本 Ruby の会，組込みシステム技術に関するサマーワークショップ実行委員会の皆様にも深い感謝の意を表します．

対外発表，活動実績

主論文

[A1] **照屋大地**, 宮﨑大智, 中條拓伯, “Pyjer: 高位合成ツールと SoC を用いた IoT 向けデバイスプロトタイピングのためのフレームワーク,” 電子情報通信学会論文誌 D, Vol. J100-D, No. 3, pp.287–297, 2017. **学生秀逸論文**

[AÎ] **Daichi Teruya**, Hironori Nakajo, “A Ruby-Based Hardware/Software Co-Design Environment with Functional Reactive Programming: Mulvery,” IEICE Transactions on Information and Systems, 2020, Vol. E103.D, No. 9, pp.1929–1938, 2020.

[B1] **Daichi Teruya**, Bipin Indurkhya, Tadakatsu Masaki, Hironori Nakajo, “Autonomous Distributed System Based on Behavioral Model of Social Insects,” The 24th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA ’18), pp. 289–295, CSREA Press, Las Vegas, NV, USA, 2018/7/30-8/2.

国際会議

[B2] Ryota Yamashita, **Daichi Teruya**, Hironori Nakajo, “Parallelization of Recursive Function in Ruby-based High-level Synthesis,” 2019 International Conference on Field-Programmable Technology (ICFPT), IEEE, 2019/12/9-13.

国内研究会

[G1] **照屋大地**, 宮﨑大智, 中條拓伯, “Java 言語ベース高位合成ツールおよび CoRAM アーキテクチャ合成フレームワークを用いた音場の可視化システムの構築,” 電子情報通信学会, 信学技報, Vol. 116, No. 53, RECONF2016-20, pp. 97-102, 富士通研究所, 2016.

[G2] **照屋大地**, 中條拓伯, “IoT/CPS 技術を支援する高位合成フレームワークの構想,” 電子情報通信学会, 信学技報, Vol. 116, No. 417, RECONF2016-61, pp. 61-66, 慶応大学日吉キャンパス, 2017.

[G3] **照屋大地**, 中條拓伯, “Ruby 言語ベースのハードウェア・ソフトウェアコデザイン環境の実現とリアクティブプログラミングの適用,” 電子情報通信学会, 信学技報, Vol. 116, No. 379, RECONF2017-65, pp. 89-94, 慶応大学日吉キャンパス, 2018. **優秀リコンフィギャラブルシステム講演賞**

[G4] **照屋大地**, 中條拓伯, ”自律分散システムのための高位合成ツールを用いたオートスケール機構,” 電子情報通信学会, 信学技報, Vol. 118, No. 63, RECONF2018-9, pp. 45-50, ゲートシティ大崎, 2018.

[G5] **Daichi Teruya**, Bipin Indurkhya, Tadakatsu Masaki, Hironori Nakajo, "Autonomous Distributed System Based on Behavioral Model of Social Insects," 情報処理学会, 第 119 回 数理モデルと問題解決研究会 (MPS 119), Vol. 2018-MPS-119, No. 4, pp. 1–4, Las Vegas, NV, USA, 2018.
[G6] 識名朝彬, **照屋大地**, 中條拓伯, "ルールベースガイドによるドメイン知識活用型機械学習システムの実現," 電子情報通信学会, 信学技報, Vol. 119, No. 286, CPSY2019-44, pp. 23-28, 愛媛県男女共同参画センター, 2019/11/13-15.

講演

[1] **照屋大地**, "Mulvery 新機能（Visualizer と RTL）のご紹介," 高位合成友の会 第六回, May. 11, 2019, 東京農工大学 小金井キャンパス 7 号館 0711, 東京.
[2] Hironori Nakajo, Tomoaki Shikina, **Daichi Teruya**, Masashi Takemoto and Shozo Takeoka, "New AI Architecture with Fusion of Logical Inference and Machine Learning," CANREXI (CANDAR Extreme Infrastrcture) Workshop (CANREXI) in The Sixth International Symposium on Computing and Networking (CANDAR'18), Nov. 28-30, 2018, Takayama Cultural Hall, Hida Takayama, Japan. **招待講演**
[3] **照屋大地** "高位合成/FPGA 活用技術の最前線 / Ruby コードをハードウェアへ - Mulvery で打ち砕くハードウェアとソフトウェアの壁," 第 20 回組込みシステム技術に関するサマーワークショップ (SWEST20), Aug. 29-31, 2018, 水明館, 岐阜. **招待講演**
[4] **照屋大地**, "CPU+FPGA プラットフォームのための Ruby ベースの開発環境," 沖縄 Ruby 会議 02, Mar. 10, 2018, 琉球大学 工学部 1 号館 大教室 321/322, 沖縄. **招待講演**
[5] **照屋大地**, "Rx からハードウェアを合成するフレームワーク Mulvery のご紹介," 高位合成友の会 第五回, Mar. 3, 2018, 東京工業大学 南四号館 2 階 201 講義室, 東京.
[6] **照屋大地**, 呉屋寛裕, "CPU+FPGA プラットフォームのための Ruby ベースの開発環境," 平成 29 年度 未踏事業 成果報告会, Feb. 10-11, 2018, 富士ソフト アキバプラザ 5 階 アキバホール, 東京.
[7] **照屋大地**, "大量のセンサを取り回したい人に贈る PyJer," IoT 縛りの勉強会！ (IoTLT) Vol.17, Jul. 19, 2016, ヤフー株式会社, 東京.

その他の活動

[1] An international visiting graduate student (IVGS) in Depertment of Electrical and Computer Engineering, University of Toronto, 2019.
[2] "動的言語における高位合成技術の研究開発," 日本学術振興会 若手研究者海外挑戦プログラム 採択, 2019.
[3] "リアクティブプログラミングに基づくハードウェア・ソフトウェアの協調設計環境," 日本学術振興会 特別研究員 DC1 採択, 2018.
[4] "CPU ＋ FPGA プラットフォームのための Ruby ベースの開発環境," 平成 29 年度未踏 IT 人材発見・育成プロジェクト 採択, 2017. **スーパークリエイター認定**

[5] “NokodAI,” 第 7 回相磯秀夫杯デザインコンテスト TRAX デザインコンペティション, 2016.