並列リコンフィギャラブル計算機システムに関する研究

A Study on Parallel Reconfigurable Computer System

2021

岡山理科大学大学院

工学研究科

システム科学専攻

高野 恵輔

序文

本論文は, 著者が岡山理科大学大学院工学研究科 (修士課程情報工学専攻および博士後期課程システム科学専攻) 在学中に, 並列リコンフィギャラブルシステムの実現に対して行った研究をまとめたものである.

本文は, 6 章から成り, 第 1 章の緒論では，本研究の背景である近年の並列計算機の基本概念およびリコンフィギャラブルデバイスを利用した計算機システムの開発動向から本研究の方向性について述べる.

第 2 章では, 緒論に続き並列計算機の構成技術についてまとめ, それぞれの利点, 欠点を述べた後にそれらの運用状況, および既存の開発システムについて説明しシステムの設計方針を定める.

第 3 章では, 実装の容易性を重視して設計・試作したリングネットワークによる PC-FPGA Hybrid System (PFH System) のプロトタイプマシンの構成, 通信機構について述べ, 幾つかシチュエーションを想定してアプリケーションを実装し, その性能評価と有効性の検討を行う. また, 電力効率を向上させる方法として, CPU から FPGA へプロセスを移行するマイグレーション手法を提案し, その有効性を示す.

第 4 章では, 第 3 章で評価した PFH System をベースに, Ethernet 通信を利用することで, より実装の容易性, 拡張性を向上させた PFH System を設計・実装する. 設計した PFH System に 3 つのアプリケーションを実装し, その性能評価から分散処理への有効性を示す.

第 5 章では, FPGA を利用したアプリケーションを開発するために, Rust プログラミング言語を使用したハードウェア設計環境を設計・実装し, その評価を行う. Rust プログラミング言語のコミュニティで定義されるコーディング規約を元に, 提案するハードウェア設計環境におけるコーディング規約の警告・提案を行うシステムを実装し, その性能評価を行う.

最後に, 第 6 章の結論では本研究で得られた結果をまとめ, 今後の課題と将来の展望について述べる.

目 次

表 目 次

図 目 次

第1章　緒論

近年, 半導体製造プロセスの微細化やシステムアーキテクチャの進化により, 高性能かつ高機能なプロセッサや大容量かつ高速なメモリが搭載された計算機システムが一般に供給されるようになった [1]. しかし, システムの性能向上とともに計算機で解決する問題も大規模化しておりさらなる性能向上が求められている. 問題に対して効率の良い計算機システムを実現するために, 並列・分散処理アーキテクチャ, オペレーティングシステム, 並列分散言語やその処理系などについてさまざまな研究が行われている [2][3]. また計算機システムの性能向上は今も続いており, システムの応用範囲も多岐にわたるようになってきている. 科学技術計算においては, スーパーコンピュータを用いた自動車設計や人流解析などへの応用が行われ, またデータ解析においては, データセンタなどに流入してくる多量のデータに対して用いられるなど実社会での運用がされている [4][5]. このように, 計算機システムは形態を変えて多方面へ導入されている.

並列計算においてはクラスタシステムの利用が求められている. 過去にはベクトル並列計算機を用いていたが, プロセッサの性能向上や汎用ネットワークの高速化・規格化によりクラスタシステムへの移行が進んだ. それだけでなく CPU のみの性能で不足がある場合に外部アクセラレータを用いる高速化手法も取り入れられている. 画像処理のように単純な演算を多量のデータに対して行う処理では GPU を用いる方法が最も性能が良いとされ, 計算モデルとして類似する流体や電磁波などの計算にも用いられている. また 2000 年代中旬頃から, NVIDIA 社の GPU を汎用計算に利用できる CUDA が発表されたことを皮切りに, GPU のアクセラレータ利用が盛んになってきた [6]. しかし, 高性能な GPU は消費電力が大きいため, GPU を用いたクラスタシステムの電力制約に対する議論も見られる [7].

そこで近年, FPGA (Field Programmable Gate Array) を代表とする内部回路の構成を変更が可能なデバイスをアクセラレータとして利用するリコンフィギャラブルシステムが盛んに研究されるようになってきた. FPGA は, 従来 ASIC (Application Specific Integrated Circuit) の設計における試作評価等に使用されることが多かったが, その柔軟性と, 実装次第で CPU や GPU と比較して演算性能や電力効率の高い処理を行うことができる特性から, 並列計算機への導入や応用研究が盛んに行われている [8][9][10]. さらに, FPGA には, 通信回路など

のアプリケーション以外の回路を搭載することもでき, 多方面で利用できるネットワークを構築するための研究も行われている [11][12].

さて一方, 研究室や事務所などの中小規模の環境では, 先に示すような大規模計算機の導入は困難である. 性能の高いワークステーションを 1 台導入するか, 安価な計算機を複数用いて小規模のクラスタシステムを構築することがコスト面から選択されることが多い. また先に述べたような GPU や FPGA などのアクセラレータを使用するのも選択肢に入る. しかし, 中小規模の環境では使用できる電力や計算機の設置場所等様々な面で制約がある. このため, 想定する環境に点在する余剰資源を有効に活用でき, 用途に合った構成がとれ多様な形態で構築できるシステムの提案が望まれる.

FPGA を含むシステムの運用を考えた時, CPU は C 言語を代表とするプログラミング言語を用いて処理を実現するのに対し, FPGA ではプログラミング言語とは根本的に言語のコンセプトが違うハードウェア記述言語を用いて処理を実現することになる. 近年は, 高位合成技術の発展により, 一部のプログラミング言語でハードウェアの設計を行うことができるようになってきた [13][14][15]. また, いくつか提案されているドメイン固有言語を使用して開発する例もある [16][17][18]. これは, FPGA での開発に長年携わっている人でも Verilog などの言語で複雑なアルゴリズムを必要とするアプリケーションを設計するのは困難であるため, より高い抽象度でハードウェアの設計を行うためである. これらの例のどちらにもいえることはハードウェア開発のコストを低減することを目的としているということである. そこで, ハードウェアについて専門的な知識を持たない技術者が FPGA によるハードウェアアクセラレーションを扱うことのできる手法が求められている.

このような背景から, 本研究では CPU や GPU, FPGA などの資源を複合して柔軟に, 用途に応じたシステムを構築する手法として PC と FPGA を複合したシステムを提案する. この方針に基づき構築した実機のシステムが用途に応じて利用できることを実証することを目的とする. また, 本研究で提案するシステムにおいて, FPGA アプリケーションを実装することを想定した Rust プログラミング言語を用いたハードウェア設計手法を提案し, 本手法の有用性を示すことを目的とする.

第2章　並列・分散計算機システムの構成技術

2.1　緒言

並列・分散計算機システムとは, 一つの問題を複数の細かい問題に分割し複数の計算機で処理を行う, またある問題において各問題が時間軸上で依存の無いものを遊休状態にある計算機に割り当て高速化, 高効率化を図ることのできるシステムである. このようなシステムにおいてネットワークの結合方式や制御方法, ならびに資源の利用方法に様々な形態が存在し, 各問題の性質や運用手法と深く関係している. 本章では, このようなシステムを設計・実装する上で重要なシステムのネットワーク構造とその特性についてまとめる. また, 運用するシステムの構造に応じた制御や通信方式の形態について説明し整理する. さらに, 幾つかの既存実装例とそのシステムの特性について述べ, 後の計算機設計の方針を定める.

2.2　相互結合網

並列・分散計算機システムにおけるノード間の接続方式は多数存在するが, 全てに対応する接続を網羅することは出来ない.

ここでは, 実際の実装されてきた計算機の構造を示し, ネットワークトポロジーの種類をまとめる. また, 実際に実装されている計算機でも基本構成からの派生形については今回取り扱わないものとする. 現在, 一般的に用いられている基本的なネットワークトポロジーを図 2.1 に示す.

(A) の共有バス結合は, 複数のノードを共通のバスに接続するネットワークトポロジーであり, CPU, メモリを接続する最も基本的な方式である. ノードが互いに通信を行う場合, バスに接続されている共有メモリを経由した間接的な通信を行うか, あるいは通信先のプロセッサやノードの持つキャッシュやメモリに対して直接的な通信を行う. 近年の多くの単独で駆動する計算機の内部がこの形式で構築されている.

この方式は, ノード数が増加するごとにアクセス競合を起こしやすくなるため, 通信の効率が落ちることが問題点である. 単純なバスを用いた構成では大規模なシステムの実装に対応することができない.

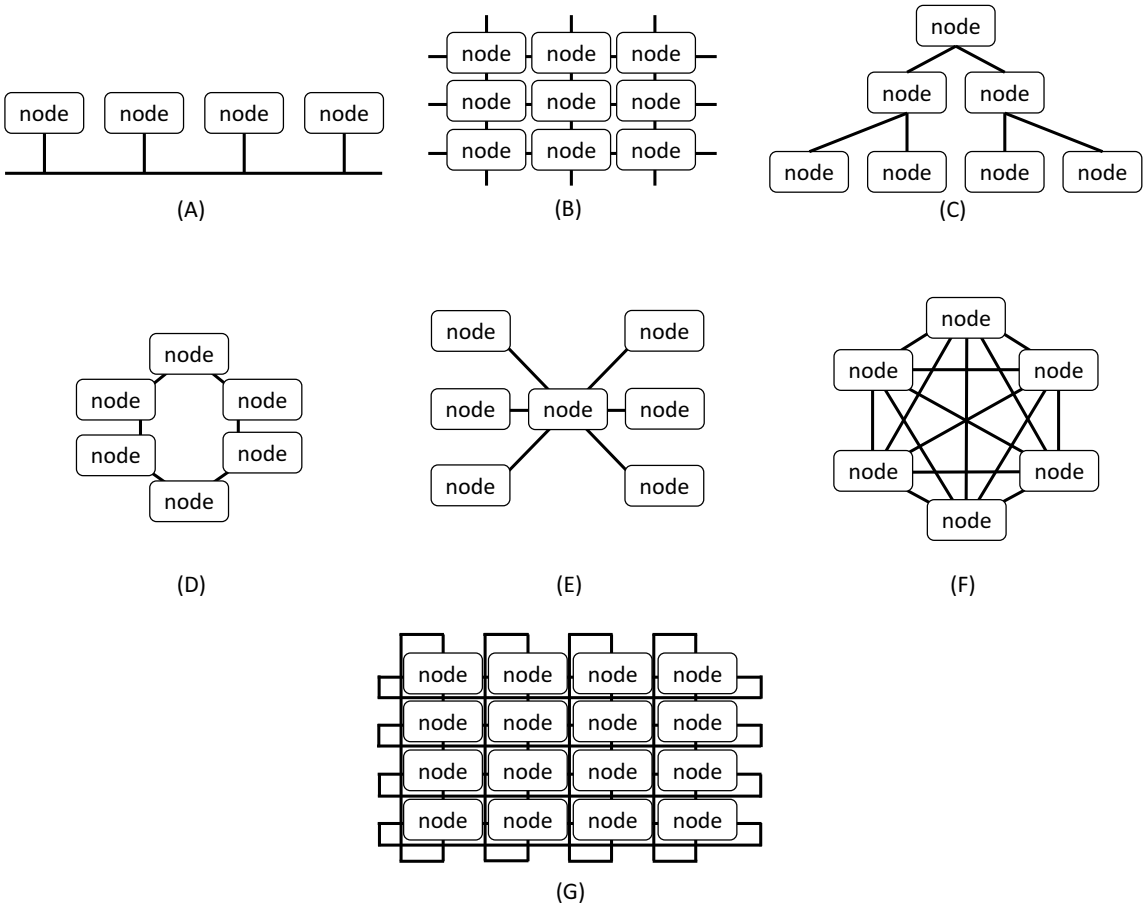

図 2.1: ネットワークの基本結合方式

(B) のメッシュ結合は, 各ノードが 1 つ以上の他のノードとポイントツーポイントで接続されたネットワークトポロジーである. 2 次元の場合, 各ノードは 4 リンクを持ち, アレイ状に配置し接続した方式である. 3 次元の場合, 各ノードは 2 次元の時からさらに 2 リンク増加し 6 方向での接続となる. 1 辺が N の 2 次元メッシュの直径は $2N$, また 3 次元のメッシュの直径は $3N$ である. また, メッシュネットワークのエッジ部分を結合する接続網を構築すると, (G) トーラス結合と呼ばれるネットワークトポロジーになり, 直径を約半分にすることができる. 近接するノード間での通信は各所で独立して行うことができるため, データ移動に局所性のある問題に対して適している.

(C) のツリー結合は, 最上位のノード以外はノード毎に 3 リンクを持ち, 上位のノードに対し 1 リンク, 下位のノードに対し 2 リンクで接続されるネットワークトポロジーである. データ収集に適した構造となっているが, 不規則な通信パターンではトラフィックが一部に集中する欠点をもち, また中間のノードが故障するとそこを経由する全てのネットワークが遮断される. そのため, スター型ほどではないが対故障性に劣る. 多重化されたネットワークで対故障性を向上させた Fat Tree 等が提案されている.

(D) のリング結合は, 各ノードは 2 リンクを持ち, ノードが円状に配置・接続されたネットワークトポロジーである. これは, 1 次元の直列接続のエッジ同士を接続したような構造を取り, ネットワークの直径は, 直列であった場合の半分の直径で構築できる. この形式はリンクが 2 本しかないため比較的容易に実装が可能であるが, ネットワークの信頼性は高くなく, また多数の接続を行う場合は直径が大きくなる問題点を持つ.

(E) のスター結合は, 各ノードが交換設備に接続されたネットワークトポロジーである. 先の共有バス結合に比べて, ノード毎の接続線は独立しているため障害耐性は高いが, 交換設備となるノードが故障した場合に全ての通信が断絶するため障害耐性は交換設備の対故障性能に依存する.

(F) の完全結合は, 全てのノードが互いに接続された形式で, どのノードに対しても 1 ホップで通信が可能な方式である. 1 つのノードが故障しても他のノードとの通信には関係なく接続できるため, 耐障害性はネットワークの中で最も高い. ネットワークの直径は 1 であるが, ノードが増加するだけネットワークの本数も増え実装が困難になるのが本ネットワークの特徴である.

以上のネットワークは, ノード間を直接接続することから直接網や静的網とも呼ばれている. その他, クロスバスイッチや多段スイッチなどの通信時に回線そのものが動的に交換されるものを間接網, 動的網と呼ばれている. 本研究では, FPGA を含む並列・分散システムを構築することを目的としている. FPGA 内には, ネットワーク制御回路以外にも複数の基本機能モジュールとアプリケーション回路を実装するため, ネットワーク回路は軽量であることが望ましい. また, 中小規模での環境を想定しているためノード数が多数になることはない. そのため, 容易に実装が可能なリング結合, または外部にスイッチを設置するスター結合が適切であると考えられる.

2.3 通信方式

並列・分散計算機システムでは, 複数のプロセス間で処理に必要なまとまったデータの通信, また制御信号を対応するノードへ正確に転送する必要がある.

ここでは, ノード間通信の主要な通信方式をまとめる.

マスタ・スレーブ方式は, 複数のノードからデータの送受信を行う場合, 特定のノード (マスタ) から対象のノード (スレーブ) に対して制御信号を送信し, スレーブからの応答を待つ方式である. マスタとスレーブの関係がネットワークの構築時で固定されている場合と, 制

御ごとにマスタとスレーブの関係が切り替わる場合がある.

近年では, マスタを「プライマリー」「ペアレント」, スレーブを「レプリカ」「ワーカー」といった語句に置き換える場合もある.

トークンパッシング方式は, トークンと呼ばれる発信権をネットワーク内に順番に回していくことで, 送信データの衝突を回避するよう管理する方式である. トークンを周遊させる時間を管理することでリフレッシュサイクル時間を規定することができる. トークンを受け取った機器が離脱している場合は規定された時間が経過すると自動的に次の機器にトークンが移動する. このため, 安定した通信ができる反面, システム構築時にノード数を決定して置かなければ性能低下を招く.

プロデューサ・コンシューマ方式は, プロデューサ (生産者) とコンシューマ (消費者) の間でデータをやり取する方式で, プロデューサはデータを作成して次々に送信する方式である. コンシュマー側は, 受信したデータの中に格納された ID を確認し, コンシューマ自身に関係のある ID のデータのみを順番に処理する. この方式では, スレーブ毎の個別の制御信号を作成する必要がなく, 関連する ID をコンシューマ側が確認して処理を進めるため送信するデータを削減することができる.

オンザフライ方式は, それぞれのノードに対して通信データ内の領域が割り振られており, 各ノードはデータを受信すると自分のアドレスに対して割り当てられた読み出し領域のデータを読み出し, 書き込み領域にデータを書き込んで次の機器にデータを送る方式である. 一つのデータの情報を最大限に使用でき, 効率よく通信できる.

以上が, 通信方式の一例である. ノード間通信では, マスタ・スレーブ方式が安定した通信を行うのに最適である. しかし, リングやメッシュ型のネットワークを構築する際に必要な制御信号を減らすことができるため, 通信経路における制御をプロデューサ・コンシューマ方式, ノード間の制御にはマスタ・スレーブ方式を使用するハイブリットな方式を適用する.

2.4 リコンフィギャラブルシステムの分類

リコンフィギャラブルシステムは, アーキテクチャを柔軟に変更可能なシステムの総称である. これらは, 体系的に分類して整理することは難しい. 本節では, 末吉らにより提唱される分類 [8] を用いる. 表 2.4 にリコンフィギャラブルシステムの分類を示す. リコンフィギャラブルシステムは, その柔軟性の観点から汎用型と専用型に分類される. 専用型は, 名の通り特定のアプリケーションに特化させたアーキテクチャを実装したものである. 対して汎

表 2.1: リコンフィギャラブルシステムの分類

リコンフィギャラブルシステム	汎用型	ハードウェアエンジン型
		コプロセッサ型
		プロセッサ型
	専用型	-

用型は, プログラマブルデバイス, メモリ, プロセッサ等の汎用的な部品から構成され, 原理的にはどのようなアプリケーションでも実行することができる.

次に, 汎用型は実行するアプリケーションの実行形態からハードウェアエンジン型, コプロセッサ型, プロセッサ型の 3 種類に分類される. ハードウェアエンジン型は, 計算処理に外部のプロセッサを必要とせずシステムの可変構造部に処理構造を実装しそこのみで処理を行う. コプロセッサ型は, 可変構造部とプロセッサ部にシステムが分割されており, 主要な処理を可変構造部で行い, その他の処理をプロセッサ部で行う. プロセッサ型は, コプロセッサ型から更に可変構造部との結合が密結合になった構造であり, 可変構造部からプロセッサの持つレジスタ等に直接アクセスできる.

本研究では, 限りなく汎用的な利用を可能とするシステムの構築を目的とする. ノードには, 汎用計算機や汎用 FPGA 評価ボードを混在させることより, 物理的にプロセッサ型を取ることは出来ない. よって, ハードウェアエンジン型とコプロセッサ型の 2 種類を可変で実現できることを目標としシステムの設計を行う.

2.5 複合システムのアーキテクチャ

ノードの FPGA 混在に際して, 文献 [19] を元に簡易的なアーキテクチャの分類を行う. 図 2.2 に示すように, 複合システムは 3 つの分類できる. (A) に示す構造は, FPGA が通信を行う場合にプロセッサ側のネットワークを使用する. 最も基本的な構成であり実装は簡単である. ただし, この構造では FPGA 間の通信性能の低さが問題となる.

次に (B) に示す構造では, FPGA 側に別のネットワークを設けたものである. 形状としては (A) の発展形態であり, この構造では (A) の問題点である FPGA 間の通信性能低下の問題を解決できる. ただし, FPGA から他のプロセッサに対してアクセスを行う場合には (A) と同様にプロセッサ側のネットワークを利用する必要がある.

(C) は, 異なるデバイスを含むアーキテクチャであるプロセッサや FPGA, その他のデバイスを一つのネットワークで結合したものである. 共通のネットワークを利用するため通信

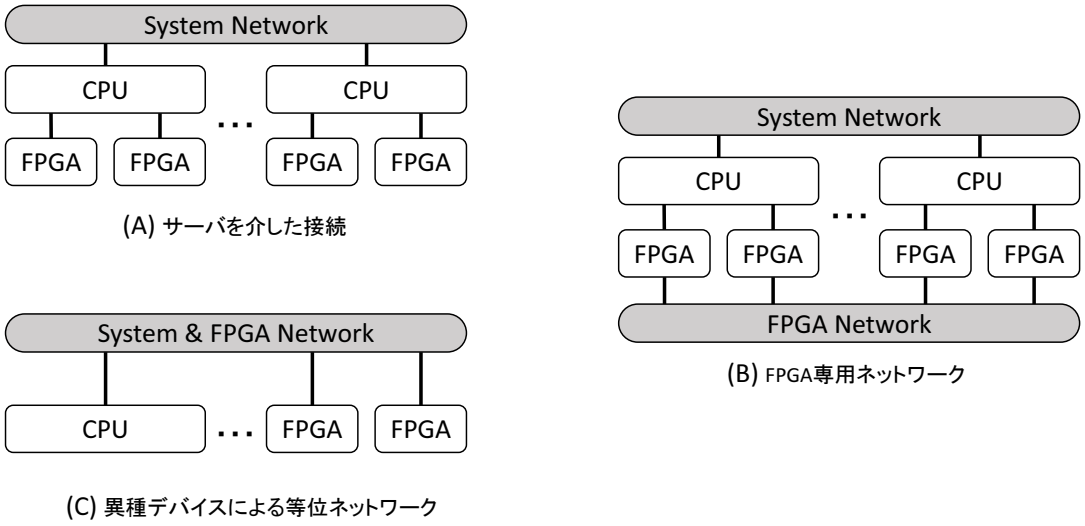

図 2.2: FPGA を含む複合並列計算機のネットワーク構造

性能を最も高く実装できる. しかし, それぞれのデバイスで共通して実装できるネットワークを選択する必要があり, また最低性能のデバイスに性能が寄せられる問題もある.

本研究では, これらのネットワークを可能な限り可変で利用できる形式を取る.

2.6 既存実装例

リコンフィギャラブルデバイスを用いた既存の実装例は多数提案されている. FPGA を利用した並列計算機の例を示す.

流体専用並列計算ハードウェア

東北大学にて, 佐野らにより実装された流体計算専用の計算機である [20][21]. この計算機は, 有限要素法に基づくステンシル計算を利用して津波の伝搬波や水深を計算する. 計算はストリーム計算要素 (SPE) で行う構造を取っている. SPE は 8 つの 32bit・word からなる計算格子セルを入力とし, これらを利用してパイプライン計算を行う. 実機として実装されたものは Intel FPGA Arria 10 を用いて行われている. 分類では, ハードウェアエンジン型に該当するシステムである.

PEACH3

慶應義塾大学にて, 天野らにより設計, 実装されているシステムである [22]. FPGA に搭載したスイッチを用いて遠隔の GPU を制御することができる. このシステムに使用する FPGA

には, 計算部分を搭載しておらず, GPU クラスタを構築し使用するためのものである. これにより密結合なクラスタシステムが構築できる. 実機では, Intel (旧 Altera) FPGA Stratix IV を用いている.

OpenFC

琉球大学にて, 長名らにより開発が進められている CPU と FPGA を複合させたシステムである [23][24]. ネットワークに Xilinx 製の Aurora IP Core を使用しているが, 加えて Intel 製の FPGA にも対応させたシステムとなっている. これによりベンダの垣根を超えた拡張可能なネットワークシステムを構築できる. ユーザーが作成したアクセラレータは, C++を用いて Vivado HLS(またはインテル HLS) を使用して, ボードに依存しないスタイルでシステムを実装できる. フレームワークを使用するためのホスト C++コード用 DMA API も提供されている.

その他の例

東京大学の牧野らは, GRAPE と呼ばれる並列計算機の開発を進めていた. これは, 元は重力多体問題に対する専用計算機として開発されたが, 派生して分子動力学や多粒子系への応用が進められている.

このように, リコンフィギャラブルシステムの並列計算機において適用されておりネットワーク, 計算性能, 汎用性と広く有用性があることが示されてきている. しかし, 問題に対して専用設計を行うと柔軟性や拡張性において低下し, 対して汎用性を重視すると性能面で向上しづらくなるなどこれらはトレードオフの関係にある.

また, 性能を求める場合はノード間の密なデータ通信を必要とするため必然的に密結合になるが, 先に述べたように性能に対して柔軟性などが低下する側面もある. 本研究では, 多様な形態での利用を想定しているので疎結合な計算機システムの構築を行う.

2.7 ハードウェア設計言語

FPGA におけるアプリケーション開発において性能の点で最も効率的な実装を行う場合, ハードウェア記述言語を用いて, Register-Transfer Level(RTL) での実装が行われる. 対して,

ある程度複雑なアプリケーションの実装には高位合成を用いるという使い分けがなされている. しかし, 第 1 章で述べたようにハードウェア記述言語による設計コストは高く, また高位合成では, ハードウェアから入出力される信号のタイミングを考慮した実装は難しい. このような場合に, DSL ベースによる, RTL 記述と比較して抽象的で効率的なハードウェア設計方法が必要とされる.

高位合成や DSL などの処理系から出力されたハードウェア記述言語は, 人が読むことを前提としていない RTL 記述が出力される場合が多い. 回路の再利用性を考慮すると, 出力される回路記述に対して可読性を考慮する必要がある.

以上を考慮し, ハードウェア設計手法に求められる機能を示すと

- 効率の良い記述方法
- 可読性のある RTL 記述の出力
- 設計に対してコーディング規約を設ける

であることが望まれる.

本研究では, これらの方針を元に Rust を用いてハードウェア設計環境を実現する.

2.8 方針

これまでに挙げてきたことを元に, 本研究の方針を改めて示す.

まず, FPGA 側のネットワークにおいては, FPGA に搭載する他の回路を考慮し軽量であることが望ましい. そのため, 第 3 章ではリングネットワーク, 第 4 章では Ethernet をベースとしたネットワークを構築することのできる回路を FPGA に実装する.

次に, FPGA 間の通信においては, ネットワークの先に存在するノード間の通信を行う場合, マスター・スレーブ方式が安定する. しかし, ノード間の接続は少ない信号線を用いてネットワーク構築を行いたいためプロデューサ・コンシューマ方式であることも望ましい. よって, 各方式のハイブリッドな方式で通信を行う方式とする.

さらに, 設計するシステムは FPGA の機種を特定のものに限定せずリコンフィギャラブルシステムとして, 可変的に利用できることを目標にしている. そのため, 搭載するデバイスが限定される専用型, また汎用型分類の中ではプロセッサ型は選択肢から外れる. よって, 汎用型分類の中のハードウェアエンジン型, コプロセッサ型, これら 2 種類として利用できるシステムとしての設計方針とする.

また, これまでに開発されてきたシステムでは性能を重視し密なデータ通信を想定した密結合な方式をとって実装を進められていたため柔軟性をある程度犠牲にしている. 柔軟性を重視するため, これらのシステムとは逆に疎結合なネットワークでのシステム設計を行う.

最後に, 並列リコンフィギャラブル計算機システムに対して, FPGA アプリケーションを実装することを想定し, 効率の良い記述が可能で, 可読性のある RTL 記述を出力でき, 設計に対してコーディング規約を設けた FPGA 設計手法を実装することを方針とする.

以上を次章より進めるシステムの設計方針とする.

2.9 結言

以上, 本章では緒論に続き, 並列・分散計算機システムを設計するために必要な基本的なネットワークの結合方式や制御方法を示し, その特性を述べ整理した. また, 幾つかの既存実装例を挙げ, その特性を元に本研究におけるシステムの設計方針を決定した.

第3章　リングネットワークによるPC-FPGA複合システム

3.1　緒言

多数のPCをネットワークによって接続し並列分散処理を行うPCクラスタの特徴と, マルチFPGAシステムによる高い電力効率を得られる特徴を複合して持つシステムとして, PC-FPGA複合クラスタを提案する. このシステムの実現には, 先行研究 [33] で要求されている従来のクラスタシステムを統合することだけでなく, 実装の容易性を獲得する必要がある. そのためには, FPGAを一つのノードとして自立できるようにしたうえで, PC, FPGAを問わず任意の構成単位 (以下ノードと呼ぶ) がホストとなって他のノードを同一手順で使役できるような手段を提供することが望まれる. 本章では, 画像処理や数値流体計算などの隣接情報のデータ通信を伴う処理を応用範囲とし, 実装の容易性と汎用性, 電力効率を重視したPCとFPGAの複合した並列・分散計算機システムの構築する手法について述べる. 上記を元にシステム設計, 実装, 応用について検討を行う. また, システムを用いた電力効率の良い運用方法の一つとして処理のマイグレーションが挙げられる [34]. マイグレーションとは, 一般に移行や移動という意味を持ち, 本文ではデータ処理を実行中のプロセスから別のプロセスへ移行するプロセスマイグレーションを指すこととする. 設計したシステムにおいて, 新たに提案するFPGAマイグレーション手法を実行し, その性能を元に考察を行う. これにより提案するシステムの有効性を示す.

3.2　システム構成

PC-FPGAを複合したシステムを実現するために, 以下3つの通信における必要な機能・要件を示す.

1. 細粒度・低レイテンシ
2. ハードウェアによる通信制御
3. 軽量な通信ハードウェア

これは, ノード間でユーザ機能のデータやユーザプロセス起動のためのコマンド等を含む数百ビット以内のブロックデータを高頻度で転送することを可能とし, FPGA にアプリケーションを実装するための回路空間を圧迫せずアプリケーションが自立して通信の要求できる条件である. これらを元に実装されたシステムは PC と FPGA の協調処理を実現する.

しかし, 近年のシステムの運用傾向から,

1. 特定の FPGA にとらわれず多機種の FPGA に対応できる
2. 大量のデータを取り扱う

ことが要求されている. したがって, これらの要求を解決する PC と FPGA を複合したシステムが必要とされていると考える. これに基づき PC-FPGA 複合システム (PFH System) を提案する.

3.2.1 全体構成

PFH System の全体構成と FPGA 回路のブロック図を図 3.1 に示す. システムは，PC 本体とその拡張スロットに装着された FPGA ボードからなる内蔵 FPGA ノードと自立して動作する FPGA ボード単体の独立 FPGA ノードで構成されるノードには複合クラスタ外部との通信の中継を担当するゲートウェイの機能を持たせたものもある．図 3.1 の構成例は, 4 台の内蔵 FPGA ノードを備えたシステムとなる. システムは, ノード数が 16 個以内の中小規模での実装を対象にするためリングネットワークでシステムを構築する. これにより, FPGA のみでのネットワーク構築を可能とする.

回路の各モジュールは共通の内部バスで接続されているため，任意のモジュール同士の相互通信が可能である．PCI DMA Controller は, PCIe コネクタに接続された PC との通信に使用されるモジュールである. Control Registers は, 内部バスに接続されたモジュール間の DMA 制御等を行う機能にアクセスできるレジスタである. Router は, ノード間通信を行うための通信回路である. DRAM は, FPGA ボード上に搭載する DRAM と内部バス間の変換制御を行う回路である. Configuration Module は, 動的部分再構成を行うための回路である. 最後に App Module は, アプリケーションを搭載するための回路である. また内部バスの規格に適合する回路であれば容易に接続・機能拡張することが可能である.

このシステムでは, 回路からの要求に応じて通信および DMA のシーケンスを行うことを可能にするため, 基本的な制御の主導権は FPGA にある構造を取る. 例えば, PC から FPGA の DRAM に対して DMA 転送を行う場合, PC は Control Registers を経由して Bus DMA

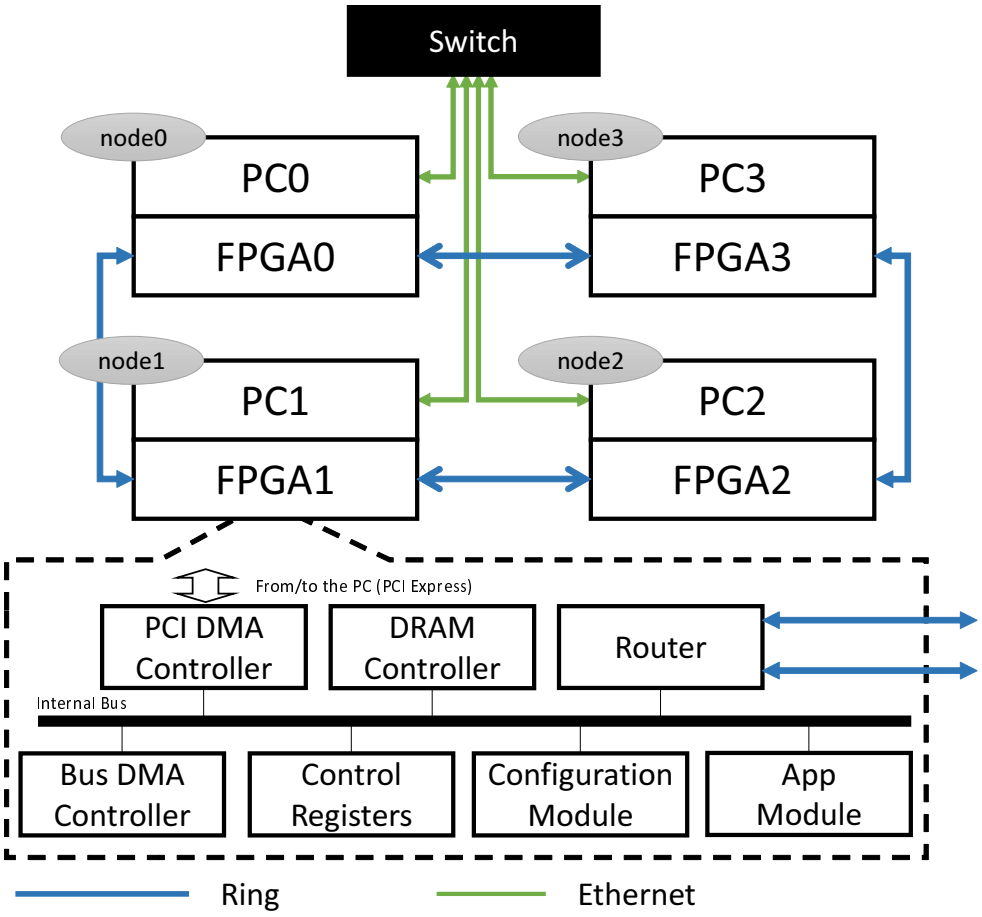

図 3.1: PFH System の構成例と FPGA 回路のブロック図

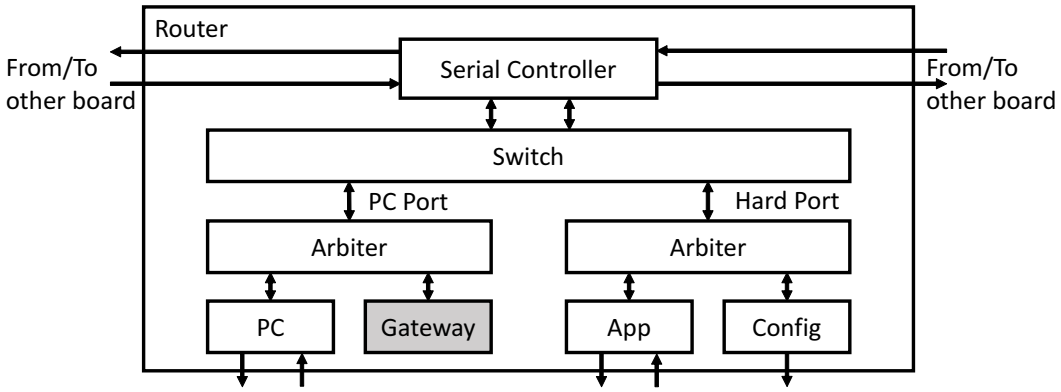

図 3.2: ネットワークルータ

Controller を駆動させる. Bus DMA Controller は, PCI DMA Controller と DRAM を制御して DMA 転送を実現する方式を取る. PC は, PCI DMA Controller からの割り込み処理を受けて DMA 転送の前処理および後処理を行う.

3.2.2 ネットワークルータ (Router)

PFH System の FPGA ネットワークでは，ユーザモジュールで扱う入出力データ，動的部分再構成用のデータ，FPGA および PC の DMA 転送データ，他ボード制御用の通信が必須であり，これらのデータ転送のためには数十〜数百ビット単位のブロックデータの低遅延な通信機構の実装が必要である．またネットワークを経由して他の FPGA 内の各モジュールへ直接アクセスできることも望まれる．本研究では, Switch に SPIN[35] の通信機構を拡張した回路を用いることで上記の用途に適応できる通信機能を実現する.

データリンク層

PFH System のネットワークは, 同軸通信ケーブルによるシリアル通信を用いたネットワークの構築を想定している. ネットワークルータのブロックダイアグラムを図 3.2 に示す. 通信には, Switch 内に備えた Send/Recv バッファを用いて Serial Controller のデータ制御を行っている. Switch は, 通信にストアアンドフォアード方式を採用しており, Serial Controller から受信したデータを中継および PC/Hard Port から来たデータの転送制御を行う. PC および Hard Port は, 元のポートから数を増やすために Arbiter を間に介している. PC Port 側には PC, Gateway ポートを備え, Hard Port 側には, App, Config ポートを備える. PC は, PCIe で接続された PC が FPGA ネットワークを利用するためのポートである. App は, App Module が FPGA ネットワークを利用するためのポートである. Config は, Configuration Module を

制御し動的部分再構成を行うためのポートである. Gateway は, ゲートウェイ機能を持つモジュールに接続することを想定しているが, 本研究では使用しない. PC および App ポートは, 対応する各モジュールが後に示す転送シーケンスを制御することで通信を行う仕組みとなっている. Config ポートは, PC や App ポートと違いデータの受信機能のみを持ち, 受信した動的再構成回路のデータを Configuration module に転送し, 動的再構成を行う.

本システムのデータリンク層における通信には以下に示す構成のパケットを用いる.

- word0 : ヘッダ
- word1 : 送信元アドレス (内部バスアドレス空間)
- word2 : 送信先アドレス (内部バスアドレス空間)
- word3 : データサイズ
- word16-32 : データ（Type:REG のみ）
- word1-32 : データ（Type:DMA DATA のみ）

パケットは, 1word(32bit) のパケットヘッダと 32word のパケットデータからなる．ヘッダには, 表 3.1 に示す内容が含まれている. ネットワーク内の信号には K キャラクターと呼ばれる制御信号が存在し, この制御信号の一部を受信バッファの状態に応じて送信側の信号に割り当てることでネットワーク内のフロー制御の実現をする.

ネットワーク層

パケットには接続要求 (REQ)，接続許可 (ACK)，DMA 制御 (DMA CTRL)，DMA データ (DMA DATA)，レジスタ制御 (REG) の 5 種類が存在する．これらのパケット情報は, 先に示した表 3.1 のヘッダ内容で指定される. DAD, SAD はボード間の相対アドレスを示す. 各ノードは, DAD を使用して宛先に対してのデータの中継, およびデータの受信を行う. SAD は, 受信したノードが ACK など送り返す必要のあるパケットを生成する際に使用する. 使用時に DAD および SAD は, 同じ値を指定する. DPT, SPT は宛先および差出元が, PC および FPGA のどちらかを示す. 例えば, PC が PCIe を経由して対象の FPGA に対してデータを送信する場合, DPT を FPGA に設定し SPT を PC に設定する. TYP は, パケットの種類を示す. 今回のシステム設計においては使用しない. VCH は, パケットが通過している仮想チャネルを示す. DPA, SPA は FPGA ボード内の宛先, 差出元を示す. 例えば, FPGA の App Module

表 3.1: ネットワーク通信において使用されるパケットヘッダ

Field	Description
[8:0]	Reserved
PTYP [2:0]	Packet Type 0 Connection request packet (REQ) 1 Connection acknowledgement packet (ACK) 2 DMA control packet (DMA CTRL) 3 DMA data packet (DMA DATA) 4 Register control packet (REG)
SPA [1:0]	Source port 0 PC 1 Gateway 2 App 3 Config
DPA [1:0]	Destination port
VCH [0]	Virtual channel
TYP [2:0]	Data type
SPT [1:0]	Source node
DPT [1:0]	Destination node
SAD [3:0]	Source address
DAD [3:0]	Destination address

が差出元で, 宛先のモジュールが Config の場合, DPA に Config, SPA に App Module を設定する. 最後に PTYP は, 先に述べたパケットの種類を示す.

次に, ヘッダの次に続くパケットの内訳について説明する. パケットの word1, word2 は FPGA の内部バスにおけるアドレスを示す. PFH System は内部バスにより各モジュールが相互に接続されているシステムを実装する. 全てのモジュールにはアドレスが割り当てられており DRAM や App Module には備えたメモリにも同様にアドレスが存在する. これらに対して PC や App Module が Router を経由してアクセスする際に設定が必要となる.

word3 は, パケットタイプが REG のときに機能するデータのサイズを示す. 例えば, FPGA に対し 128bit 分のデータをレジスタに書き込む場合, word3 には 128bit=16byte を示すために 16 を設定する.

上記のパケットを元に通信を行うシーケンスの例を図 3.3,3.4 に示す. 図 3.3 は, 図 3.1 の構成を対象に例を示すと, ホストとなる PC0 のシステムメモリからネットワークを経由して FPGA1 の DRAM に対してデータを転送する手順である. データパケットの反復回数が REQ パケットに記述されているため, 終了処理は必要としない. 図 3.4 は, 図 3.1 における FPGA0 内の App Module からネットワークを経由して FPGA1 の DRAM に対してデータを要求し, 受信を行う手順である. 最初に App Module が要求パケットを送信した後に Router

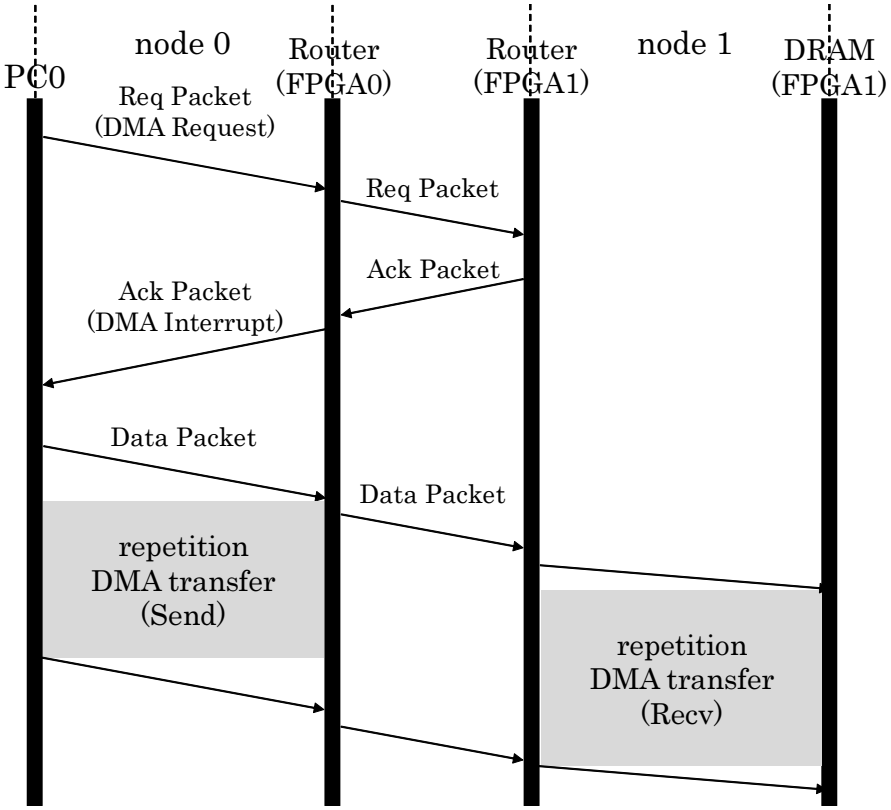

図 3.3: 送信シーケンス

間で行われている Req-Ack のシーケンスは, 送信要求のパケットに対する接続要求と応答の処理である. その直後に受信要求を行ったノードの Router から生成される Req-Ack のシーケンスは, データ転送に対する接続要求と応答の処理である.

このシステムでは, PC, DRAM, Configuration Module, および App Module の間で同様のシーケンスが定義されている. また, Configuration Module は DRAM や App Module と同様に内部バスアドレス空間にマッピングされており, ルータの Config ポートにおける制御では, この Configuration Module に対して内部バスを経由して制御する方式をとる. これにより, 書き込み先アドレス・ポート (word2 とヘッダ DPA) を Configuration Module に指定して書き込みデータを部分再構成データにすることで他のノードより FPGA を動的部分再構成を実行することができる. 図 3.5 に Config ポートの動作手順を示す. 簡易的な制御のため, この処理はハードウェア実装を行っている.

3.2.3 アプリケーションモジュール (App Module)

図 3.6 にアプリケーションモジュールのブロック図を示す. アプリケーションモジュールは, FPGA に実装するアプリケーション用の回路空間である. アプリケーションは, PC のソ

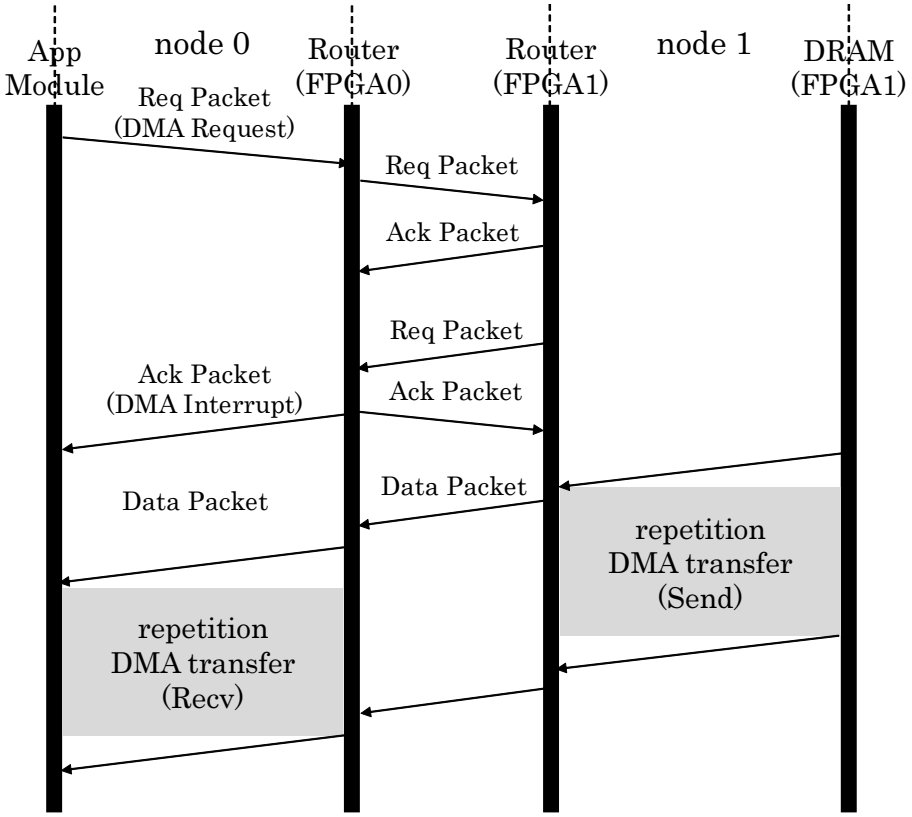

図 3.4: 受信シーケンス

フトウェアと同様に変更できることが望ましい. そこで, 動的部分再構成を用いて, アプリケーション回路を書き換える構成とする. 通信などの共通で持つ必要のある機能 (以下, 基本機能と呼ぶ) は, どのアプリケーションにおいても変更する必要はない. そのため, アプリケーションモジュールは基本機能は静的回路とし, アプリケーションの本体となる動的回路を User Module として定義し実装する.

基本機能は FPGA 基板上に搭載されている外部メモリと自ボード, もしくは他ボードのモジュールとのデータ通信を行う簡易 DMA 機能を有する回路となっている. 他ボードとデータ通信を行う場合, 先に示したネットワークルータにおけるシーケンスを基本機能が実行し, データの転送を行う. 制御レジスタによりデータ転送, 簡易 DMA 機能は制御することができる.

制御レジスタは, PC やネットワークの先の他ノードが内部バスを経由して制御することが可能となっている. User Module がこれらの機能を利用する際, Frontend からレジスタの書き込みを行うだけで実現できる. また Local buffer へ格納したデータの読み出しや書き込みは FIFO を介しているので App module の静的回路との同期を気にせず処理を実装できる.

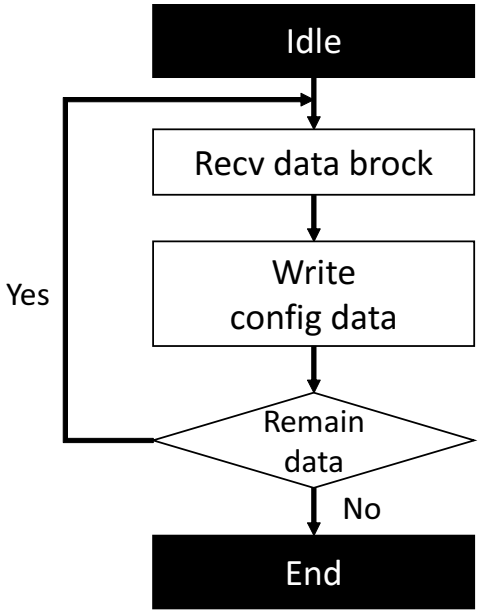

図 3.5: Config ポートの動作手順

3.3 試作システム

前節で固めた設計に基づき, 本研究ではリングネットワークによる PFH System を開発した [36][37]. 以下に 4 台の PC と 4 台の FPGA で実装した試作機の詳細について述べる.

3.3.1 リングネットワーク PFH System

表 3.2 に試作機を構成する要素を示す. PC 間のネットワークは, 1000BASE-T を介しルータ (WZR-900DHP) に接続されている構成となっている. FPGA ネットワークは, 各 FPGA が同軸通信ケーブルを備えた双方向ネットワークで接続されている. FPGA ネットワークの

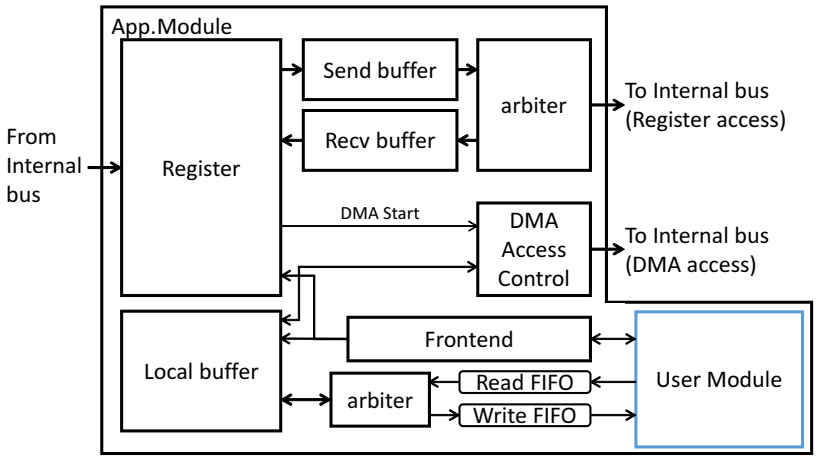

図 3.6: アプリケーションモジュールのブロック図

表 3.2: 試作機の構成要素

Component	Node No.	Version
CPU	PC0	Corei7-3820
	PC1 to PC3	Corei7-4770S
OS	PC0 to PC3	CentOS7
FPGA	FPGA0 to FPGA3	Xilinx KC705 (XC7K325T)

物理層は GT Wizard により生成した IP で制御しており, 伝送速度は 4*Gbps* である. FPGA ボードに使用されている DRAM は DDR3-1600 を使用する. DRAM の制御には Memory Interface Generator (MIG) を用いて生成した IP を用いる. PCI DMA Controller には, 株式会社 SYSTEC により開発された PCI Express インタフェース IP である SYPCIE を Gen2x4 で使用する. Configuration Module には Xilinx 製の HWICAP IP を使用する. 内部バスには, Advanced eXtensible Interface (AXI) を採用しており, これに基づき Xilinx 社の提供する AXI interconnect IP (バス幅 128bit) を用いて相互に接続する. AXI アドレス空間におけるメモリマップと PCI Express におけるアドレスを表 3.3 に示す構成で実装する. ホスト PC から見

表 3.3: FPGA 内メモリマップ

(a) PCI Express Address

Usage	Start Address	End Address	Size
DMA	0x00000000	0x000003FF	1 KB
System Control	0x00000000	0x000003FF	1 KB
Internal Bus	0x00000000	0x07FFFFFF	128 MB

(b) Internal Bus Address

Module(Unit)	Start Address	End Address	Size
HWICAP	0x00000000	0x00000FFF	4 KB
BAR2 Register	0x00100000	0x0010FFFF	64 KB
Router(PC)	0x00200000	0x0020FFFF	64 KB
Router(App Module)	0x00400000	0x0040FFFF	64 KB
App Module(Reg)	0x00600000	0x0060FFFF	64 KB
App Module(User)	0x00610000	0x0067FFFF	448 KB
DRAM	0x40000000	0x7FFFFFFF	1 GB

た FPGA デバイスは, DMA, System Control, Internal Bus の 3 つのアドレス空間にアクセスできる状態となっている.

DMA は, 先に示した Bus DMA Controller からの割り込み (Interrupt) によって PC が制御を行うためのレジスタである. このアドレスの制御は, 基本的にユーザのプロセスが行うことはない.

次に System Control は, FPGA 内の各システムの状態を確認することのできるレジスタである. 例えば, Router の送信バッファが Full になっているかどうか, DRAM の初期化処理が終了しているかどうかなどの情報が確認できる. また, FPGA 内の内部バスや Router のリセットもこのレジスタの制御により可能にしている.

Internal Bus は, 表 3.3 の (b) アドレスをマッピングしており, PCIe を経由して各モジュールを直接制御することを可能にしている.

以上を元に実装した試作機の全景を図 3.7, ボードの実装を図 3.8 に示す.

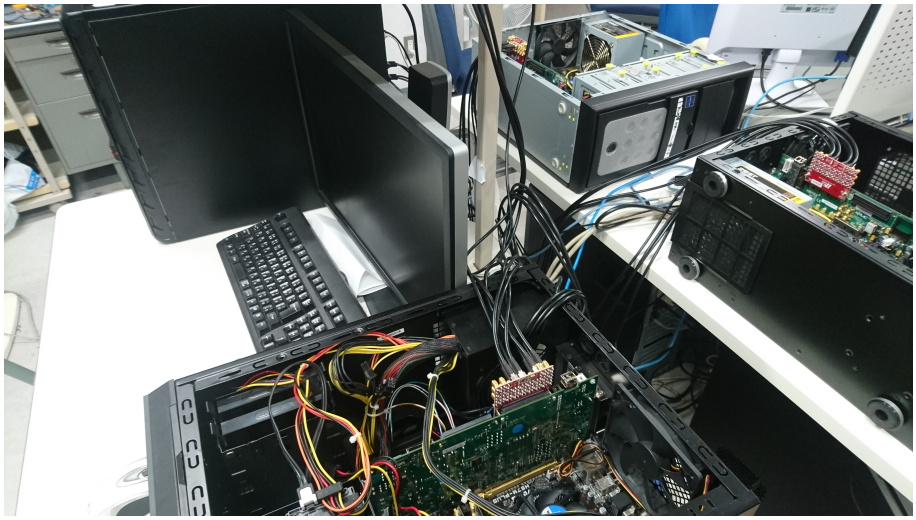

図 3.7: PFH System の全景

3.3.2 ソフトウェア

システム上のホスト PC が FPGA を利用するためには, PCIe の制御や FPGA 上に実装した各モジュール間のシーケンスを実装しなければならない. それらの制御を考慮せずアプリケーションに組み込むことのできる仕組みが必要であるため, Application Programming Interface (API) を実装する.

ホスト PC からは, 通信ライブラリと制御関係を統合したライブラリとして使用する. クラスタ全体の制御としては, 分散処理フレームワークの Spark を使用することを前提としている. PC および FPGA のアプリケーション本体は C 言語, ノードへの分散やデータの回収等の制御には Python を開発に用いている.

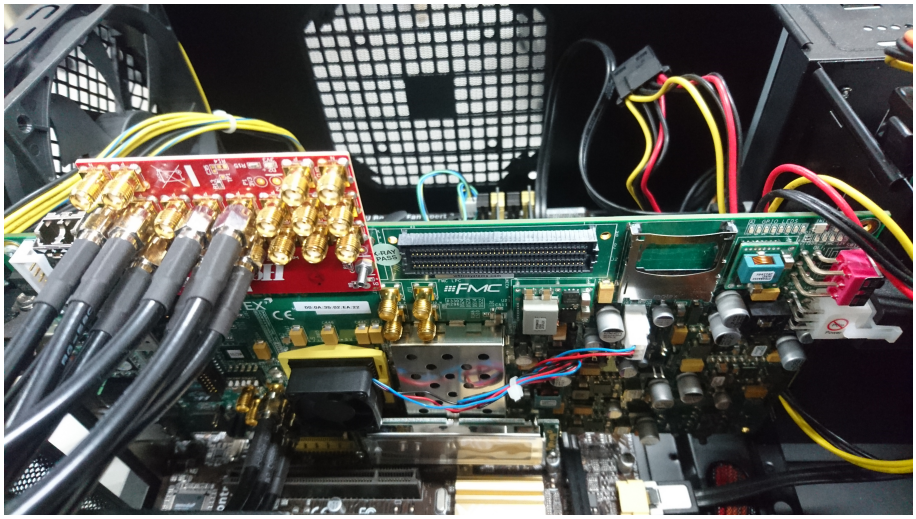

図 3.8: PFH System のボード実装状況

表 3.4: 実装 API の一覧

Function	API
Data transfer	dma_pc_to_pc dma_pc_to_fpga dma_fpga_to_fpga
Register control	register_read register_write
FPGA control	partial_reconfig batch_start

表 3.4 に, FPGA を用いて分散処理を行うための API を示す.
dma_pc_to_pc はホスト PC メモリ内のデータを他の PC メモリに転送し, dma_pc_to_fpga はホスト PC メモリ内のデータを他の FPGA DRAM に転送するのに利用する. また, dma_fpga_to_fpga 各 FPGA 間でデータ転送を行うために利用する. register_read および register_write はローカル FPGA のレジスタ値を制御するために使用される. partial_reconfig を使用することで動的部分再構成を実行でき, batch_start は App Module に実装したアプリケーションを実行することができる. これらの API が PFH System に実装されている.

3.4 性能評価

本節では, 実装した PFH System の試作機において基本性能の評価, および種々の問題を実装した性能評価を行う. ここでは, 5 種類の画像フィルタ, JPEG エンコーダを扱う. 画像フィルタにおいては, 並列処理, マイグレーションでの評価を行い [38], JPEG エンコーダに

おいては, 分散処理における評価を行う [39].

3.4.1 基本性能の評価

基本性能の評価については，以下のデータ転送, または, FPGA 上の回路／ PC 上のソフトウェアの呼び出し (以下, 単に呼出という) を実行すれば十分であろうと考えられる. なお, PC および FPGA の名称は, 図 3.1 のものに対応する.

(1) PC0 から FPGA0 への呼出／データ転送 (PC から自ボード FPGA)

(2) PC0 から FPGA1 への呼出／データ転送 (PC から他ボード FPGA)

(3) FPGA0 から FPGA1 への呼出／データ転送 (FPGA から FPGA)

(4) FPGA0 から PC0 への呼出／データ転送 (FPGA から PC)

(1)〜(4) の機能を確認するため，以下の手順で実験を行った.

(i) PC0 から, アプリケーションを FPGA0〜FPGA3 に送信して動的部分再構成

(ii) PC0 上で HD 画像を読み込み, 1280*180pixel の大きさに 4 等分して, 各分割画像を FPGA0〜FPGA3 に送信

(iii) PC0 が FPGA0〜FPGA3 に呼出を行ってフィルタ指示を行い, 自身のプロセスを休止

(iv) FPGA0〜FPGA3 からの呼出に応じて PC0 のプロセス休止を解除し, フィルタ後画像を受信

(v) PC0 は 4 枚の画像を受け取った後, これを単純結合して PC0 上に表示

この時の FPGA 回路のリソースは, 静的回路が全体の 66.50%, 動的回路の User Module が全体の 8.78%であった. User Module には, Verilog で記述したグレースケールフィルタを実装した.

これら実験の結果, PFH System において基本的な処理を行うための機能を備えていることを確認した. PC-FPGA 間通信速度を測定した結果, FPGA0 への転送には 332[MB/s], FPGA1 への転送には 305[MB/s] となった. PCIe DMA Controller は, 通信対象のモジュールからの割り込みにより駆動する仕組みで実装している. そのため, 割り込みの待ち時間により帯域に対して低速な通信となっていると考えられる.

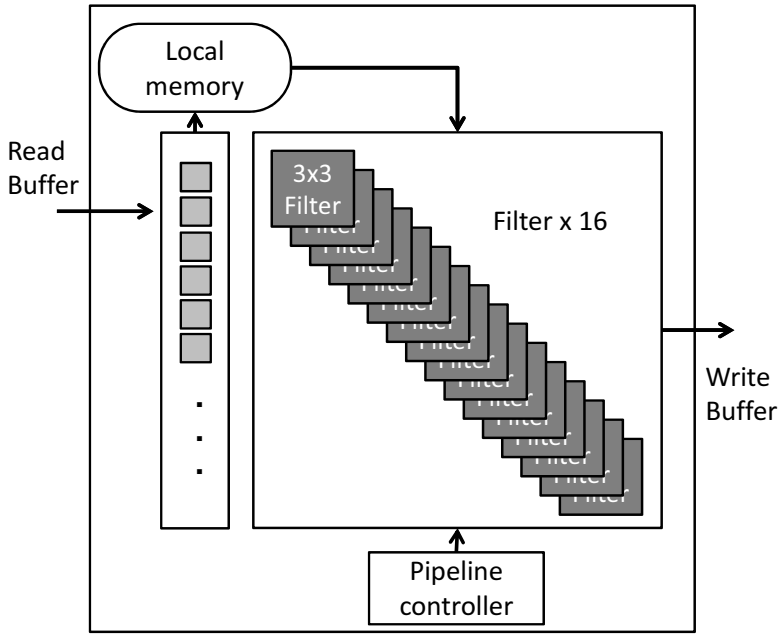

図 3.9: 画像フィルタ回路のブロック図

3.4.2 並列処理

本節では, 並列処理の実行を検証する. User Module に実装するアプリケーションとして以下の 5 種類の画像フィルタを用意する.

- 平滑化フィルタ
- ラプラシアンフィルタ
- ガウシアンフィルタ
- ソーベルフィルタ
- メディアンフィルタ

これらは, 全て App Module のバッファをリングバッファとして利用するように実装している. 画像フィルタ回路のブロック図を図 3.9 に示す. フィルタ回路は, App モジュールの Read FIFO から 1 行ずつ読み出し, 処理済みのデータを 1 行ずつ Write FIFO に対し書き込む. Read FIFO から読み出したデータは, 内部の Local memory に一度格納される. Pipeline Controller は, 16 個の 3*3 フィルタに対してパイプライン制御を行いながら Local memory 内のデータに対してフィルタリングを行う. 3*3 フィルタは, 1 画素 (8bit*3ch) に対する畳み込み処理を 1 クロックで実行する.

このような実装を行ったフィルタ回路を用い, 1280*720 の画像に対して供給クロック 160MHz で駆動させた各フィルタを実行した. 時間をクロック単位で計測した結果, 18.17msec で実行できることを確認した.

また, このフィルタを用いて基本性能の評価で行った (i)〜(iV) を実行し, 並列処理を行った. 各 FPGA によるフィルタの実行時間は DMA 転送時間によって隠蔽される．このため, 画像処理時間としては, 4 つの FPGA に対する画像の送信／受信と, それらの処理の間に必要なインターバル等のオーバヘッドの合計となり, 143.0[ms] となった.

3.4.3 マイグレーション

マイグレーションにおける実験では, 1000 枚の HD 画像に対してラプラシアンフィルタを実行することを想定する. 実行は以下の手順で行う.

1. PC0 と FPGA0〜FPGA3 を起動し, それ以外のノードの電源を停止する
2. PC0 でフィルタソフトウェアを起動し, PC0 で 500 枚の画像のフィルタ処理を行う
3. FPGA0〜FPGA3 の HWICAP にラプラシアンフィルタの部分再構成データを転送する
4. 残り 500 枚の画像データを各 FPGA0 FPGA3 へ転送しフィルタ処理を移行する
5. PC0 の電源を停止し, 一定時間後に PC0 を起動する
6. UserModule からの割り込みにより PC0 は DRAM からデータを受信する

加えて, これらの処理実行中の電力を計測する.

この実験の結果, 全ての画像は正しくフィルタリングされていることを確認できた. 実験における PC0+FPGA0 と FPGA0 単体の消費電力はそれぞれ 60.0[W] と 21.0[W] であった. このことから, PC0 から FPGA0,1,2,3 に処理を要求した後, PC0 の電源を切ることで, 消費電力を 31.7[%] 削減できることが示された. 全体として, プロセスマイグレーションの意図した動作が実現されていることがわかった. 実験で実証されたように, プロセスが FPGA に委託された後, PC を一時的にシャットダウンすることが可能であり, 結果は後で得られる. この機能は, 従来のシステムにはない特徴であるといえる.

3.4.4 分散処理

本節では, 分散処理の実行を検証する. 実験には, C 言語で記述された JPEG エンコーダのコードを用い, FPGA には高位合成を用いて実装を行う. 対象とする画像は, 320*240 の

表 3.5: JPEG エンコーダのリソース使用量 (%)

	User Module
FF	5.69
LUT	15.29
Memory LUT	0.16
BRAM	21.68
DSP48	33.81

Algorithm 1 root_process における疑似コード

```
for Data_iter = 1 to number of images do
    if FPGA execution image then
        set FPGAData(image)
    else
        set SparkData(image)
    end if
end for
set exedataS ⇐ executor.submit(Spark_process(SparkData))
set exedataF ⇐ executor.submit(fpga_process(FPGAData))
set Synchronize(Spark_process, FPGA_process)
save exedataS, exedataF
```

HVGA とする. 表 3.5 に User Module のリソース使用量を示す.

JPEG エンコーダにおいては, Spark を用いた分散処理を適用する. 分散処理には, 疑似コード 1,2 のアルゴリズムを元に分散処理を実行する.

`root_process` は, fpga 制御用のプロセス (`fpga_process`) と Spark 駆動用のプロセス (`Spark_process`) を管理するためのプロセスであり, 準備したデータを立ち上げた各プロセスに分配する処理と, 各プロセスから処理データを回収する処理を行う. `fpga_process` は, システムに参加する FPGA に対してデータを分配・処理の要求・回収を行うプロセスである.

この疑似コードに基づき分散処理を行った結果を表 3.6, 3.7 に示す. 表 3.6 は, 画像データの転送時間の実験結果を示している. 転送する画像数の増加に伴い, 伝送効率が向上していることが明らかになった. ただし, PC0 はローカル FPGA の DRAM を介して他の FPGA の DRAM に画像を転送するため, FPGA ネットワークを介してリモート FPGA の DRAM に画像を送信するには, ローカル FPGA の DRAM へのデータ転送と比較して実行時間を要する結果となった.

表 3.7 は, JPEG エンコーダにおける分散処理の実験結果を示している. PC-FPGA 比率は, PC と FPGA によって処理される画像の数の比率を示している. たとえば, 10:0 の比率は PC

Algorithm 2 fpga_process における疑似コード

```
set FPGAData
for Data_iter = 1 to Number_of_image do
    set dma_pc_to_fpga(Data[Data_iter])
    if board_address ! = LOCAL_BOARD_ADDRESS then
        set dma_fpga_to_fpga(board_address)
    end if
    set batch_start(board_address)
    if board_address ! = LOCAL_BOARD_ADDRESS then
        set dma_fpga_to_fpga(board_address)
    end if
    set buffer ⇐ dma_pc_to_fpga()
end for
return buffer
```

表 3.6: 枚数毎のデータ転送時間

Conditions	1 image [ms]	4000 images [s]
PC (Spark)	3.65	7.48
FPGA (Local board)	0.45	7.19
FPGA (Remote board)	0.80	12.14

のみでの処理, 0:10 の比率は FPGA のみでの処理に対応する. 図 3.10 は, 処理対象の画像数と実行時間の関係を示している. 比率が 10:0 の場合, 画像数の増加に伴い, 1 つの画像の実行時間が減少した. PC での処理については, 通信と計算の重複は Spark によって管理されているため, プロセス数が少ない場合に比べてプロセス数が多いほど処理効率が向上すると考えられる. FPGA での処理については, PC0 はすべての FPGA に画像を分配してから処理を実行し, 終了時に結果を収集している. このため, 通信と計算が重ならず, 画像数に比例して実行時間が長くなる結果となったと考えられる.

図 3.11 では, PC と FPGA の比率が 10:0 と 0:10 の場合, 実行時間は 9.0[s] と 80.1[s] であった. それぞれ, 画像の数は 4000 枚であり, PC と FPGA の性能比が 8.9:1 であることを示している. 一方, PC-FPGA の比率が 9:1 の場合, 計測した実行時間は最も性能が良く, これは先に

表 3.7: 分散比率毎の実行時間

	Distribution ratio of PC:FPGA					
number of images	0 : 10	1 : 4	1 : 1	4 : 1	9 : 1	10 : 0
1000	19.7	16.6	12.6	4.0	3.2	3.7
2000	39.9	38.5	20.5	8.2	5.0	5.6
3000	59.2	46.2	29.9	12.2	6.8	7.7
4000	80.1	71.2	38.3	16.3	8.0	9.0

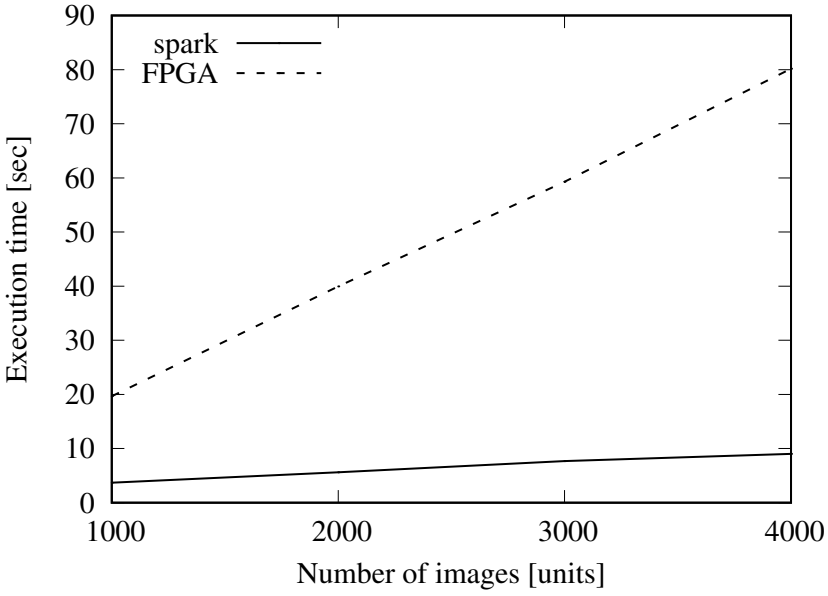

図 3.10: PC と FPGA における実行時間

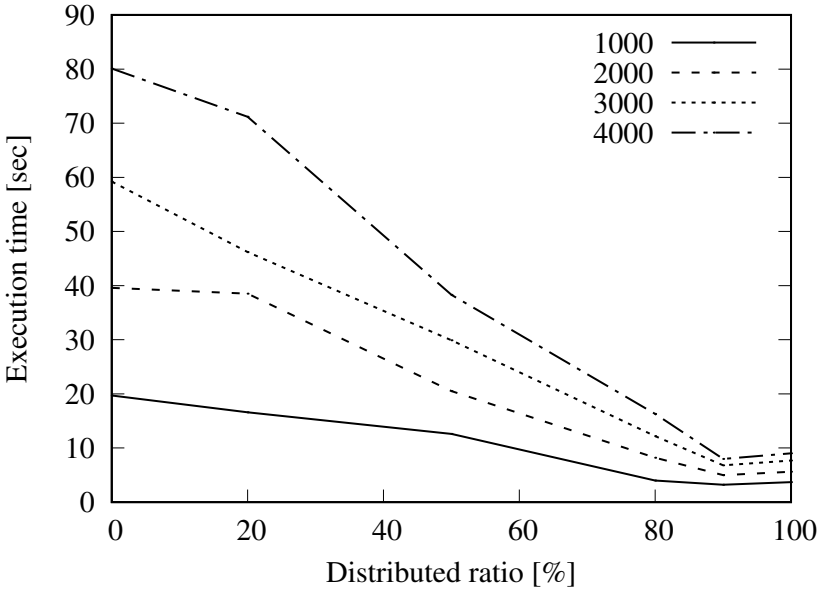

図 3.11: 実行時間と分配比率の関係

示した PC と FPGA の性能比である 8.9:1 と近い値である. したがって, PC と FPGA のどちらかだけで処理を行った実行時間から最適な分散比率を割り出すことができると考えられる. また, この状況において, PC と FPGA の性能比率が 1:1 の場合が実行時間は最も短くなると考えられる. 加えて, PC と FPGA のどちらかに性能差が偏っていても, 少しでも性能の劣る側のシステムに処理を分配することで性能向上を図ることができると考えられる.

このことより, 最適な分散比は, PC と FPGA の処理が互いに干渉せず, PC と FPGA 間の

分散処理が成功したと言える.

3.5 結言

本章では, リングネットワークによる PC-FPGA 複合システムの設計と試作機の実装について述べ, 画像処理フィルタ, JPEG エンコーダといった各問題について並列・分散処理, マイグレーションを実行し, 性能評価を行った.

本章で得られた評価結果を以下にまとめる.

基本性能の評価において, FPGA 回路のリソースは, 静的回路が全体の 66.50% で実装できることを確認した. また, 予備実験によりシステムの要求する (1) PC からローカル FPGA への呼出／データ転送 (2) PC からリモート FPGA への呼出／データ転送 (3) ローカル FPGA からリモート FPGA への呼出／データ転送 (4) ローカル FPGA から PC への呼出／データ転送の機能を有していることを示した. PC-FPGA 間通信速度を測定した結果, ローカル FPGA への転送には 332[MB/s], リモート FPGA への転送には 305[MB/s] で通信できることを示した.

画像フィルタにおいては, 5 つのフィルタを実装し, 並列処理の実行, および処理のマイグレーション手法を提案し, システムの有用性を示した. マイグレーションにおいて, 1 台の PC から 4 台の FPGA に処理を要求した後, PC の電源を切ることで消費電力を 31.7[%] 削減できることを示した.

JPEG エンコーダにおいては, 提案する分散処理手法を適用し, システムの性能を評価した. 4000 枚の処理において PC-FPGA の比率が 9:1 の場合に実行時間が 8.0[s] となり, 実行時間は最も性能が良い結果となった. このことから, PC と FPGA の比率が 10:0 と 0:10 の場合の実行時間 9.0[s] と 80.1[s] から, PC と FPGA の性能比と近い値であることより, 処理の実行時間から最適な分散比率を割り出すことができると考えられる.

以上の結果より, FPGA により構築されたリングネットワークを持つ PC と FPGA を複合したシステムの実現ができ, 並列・分散処理において要求される状況に応じて性能向上できることを示した. 使用した FPGA の機種が一種のみだったが, 内部に実装した回路は可能な限り汎用のモジュールを使用しており, 用途に応じて拡張ができ, また FPGA の機種にとらわれない柔軟な分散処理システムを構築できる可能性を持っている. 以上より, 提案するシステムの有効性を示した.

第4章　EthernetベースのPC-FPGA複合システム

4.1　緒言

PC だけを用いた並列計算機と同様に, PC と FPGA を複合したシステムにおいても容易な拡張性が求められる. 第 3 章で示したリングネットワークで実装した PC-FPGA 複合システムは, FPGA 同士の接続においては実装容易性を担保できているとは言えない.

本章では, 第 3 章で実装したシステムの内部構造に大幅な変更は行わず, ネットワークにおいて Media independent interface (MII) と呼ばれる規格に基づいた, より汎用性を重視した並列・分散計算機システムの構築を目標とし, 設計, 実装, および評価を行う.

4.2　システム設計

PC と FPGA を組み合わせたシステムはこれまでに多数の研究報告がなされている [40]. 多くのシステムは密結合による並列処理を志向して設計されているが, 筆者の提案するシステムは疎結合による分散処理を指向している. 前章であげたシステム構成から, 実装容易性や性能を考慮して PFH System の実装方針を改めて 3 つ定義する.

1. 問題の性質や電力要求に応じて、PC と FPGA の柔軟な分散利用が可能な構成
2. Ethernet MII(Media Independent Interface) を持つ FPGA に広く対応する FPGA 間接続
3. 分散処理フレームワーク Spark[41] と HLS による分散処理

これらの方針を元に構成した PFH System の構成を図 4.1 に示す. PFH System は, PC および FPGA で構成されたクラスタシステムである. 本章のシステムでは, PC と FPGA はそれぞれ L2 スイッチングハブを介してネットワークを形成する. 第 3 章のシステムでは, リングネットワークによる独自規格の通信方式を用いていたため, 通信可能なノードの機種が限られる. Ethernet のプロトコルをベースに通信する方式にすることにより, ネットワークインタフェースの速度差があるノードが存在しても, スイッチングハブが速度差を吸収でき, ま

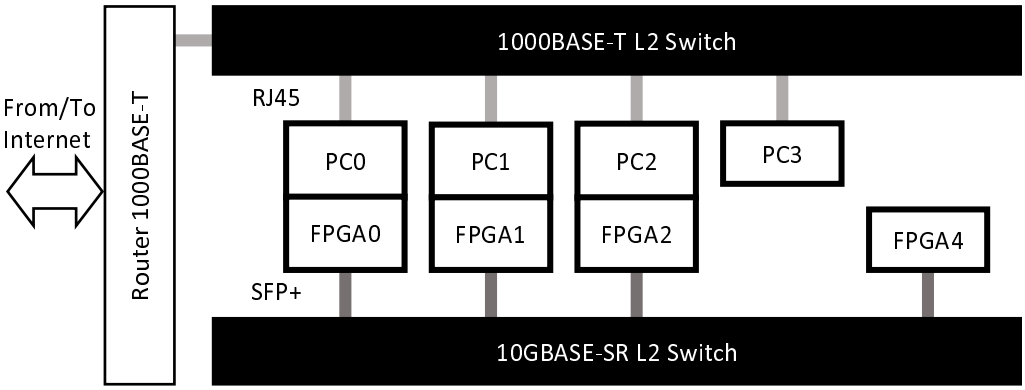

図 4.1: Ethernet 通信をベースとした PFH System の構成

た通信において PC と FPGA を同じスイッチングハブに接続するような構成でも実装できる. 第 3 章で示したものと同様に, FPGA は PC に必ずしも内蔵されている必要はなく, ネットワークに単独で存在することもできる. PC に直接接続されて筐体内部に格納されている FPGA を内蔵 FPGA, ネットワークに単独で存在する FPGA を独立 FPGA と定義する. また通信の起点となるノードから見て同じ筐体内部に格納されている PC および FPGA はローカル, ネットワークの先に存在する FPGA とそれに接続された PC をリモートと定義する. 図 3.11 で示される一例は 4 台の PC と 4 台の FPGA で構成されおり, 3 台の内蔵 FPGA と 1 台の独立 FPGA で構成されているものになる. この構成において, PC0 を通信の起点として見ると, FPGA0 はローカルとなり, それ以外の FPGA1,2,4,PC1,2,3 はリモートとなる.

4.2.1 FPGA 内部構成

FPGA 内部に実装している回路のブロック図を図 4.2 に示す. FPGA 内の回路は PCIe DMA Controller、DRAM Controller、App Module、Network Interface、Configuration Module で構成されている. 各モジュールは, 内部バス (Interconnect) を介してお互いにアクセスが可能な構成になっている. PCIe DMA Controller は, PC がローカルの FPGA 上の DRAM やリモートの FPGA 上の DRAM と DMA を実行するためと, レジスタ制御機能により FPGA 内部の各モジュールを制御するためのモジュールである. DRAM Controller は, FPGA ボード上に搭載されている DRAM にアクセスするためのモジュールとなっている. App Module は, FPGA 内に実装するアプリケーションを搭載するための回路である. 最後に Configuration Module は, App Module 内に実装されたアプリケーションを動的に再構成するためのシステム回路である. これは, 各 FPGA ベンダの提供する専用モジュールを利用する, もしくはこの専用モジュールを内包するモジュールを製作することによって実現する. FPGA の回路は

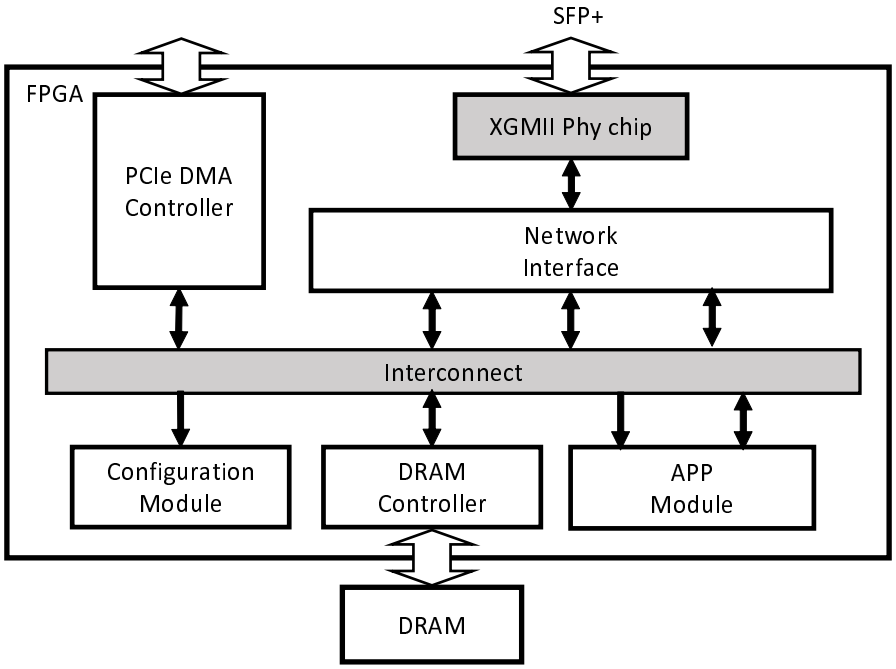

図 4.2: FPGA に実装した回路のブロック図

上記のモジュールにより構成されている. 本システムにおける回路においても, 内部バスの規格に適合する回路であれば容易に接続・機能拡張することが可能である.

4.2.2 FPGA ネットワーク

データリンク層

FPGA のネットワークは Ethernet L2 スイッチングハブを介して通信するネットワークとなっている. ネットワーク回路 (Network Interface) の構造を図 4.3 に示す. ネットワーク回路は PC と HW のそれぞれ Send/Recv バッファを持ち, HW 側の通信の挙動においては HW Controller により管理されている. PC は, PCIe を経由した通信を行うためのものである. HW は, FPGA に実装した回路が FPGA ネットワークを利用するためのものである. `NI HW Ctrl Port` は HW Controller にアクセスでき, `NI PC Port` は PC が PCIe を経由して PC Send/Recv buffer にアクセスできる構造となっている. `NI HW R/W Port` は, HW Controller からの制御信号により起動し, DRAM へのデータ転送やレジスタアクセスを行う機能を持つ. FPGA に実装したアプリケーションは, `NI HW R/W Port` にアクセスすることでリモート FPGA やリモート PC と通信が可能となっている.

本システムの通信には, 図 4.4 において定義するフレームを使用する. Ethernet II+ frame は, IEEE にて定義される Ethernet II フレームにネットワーク回路で使用する制御ビットを

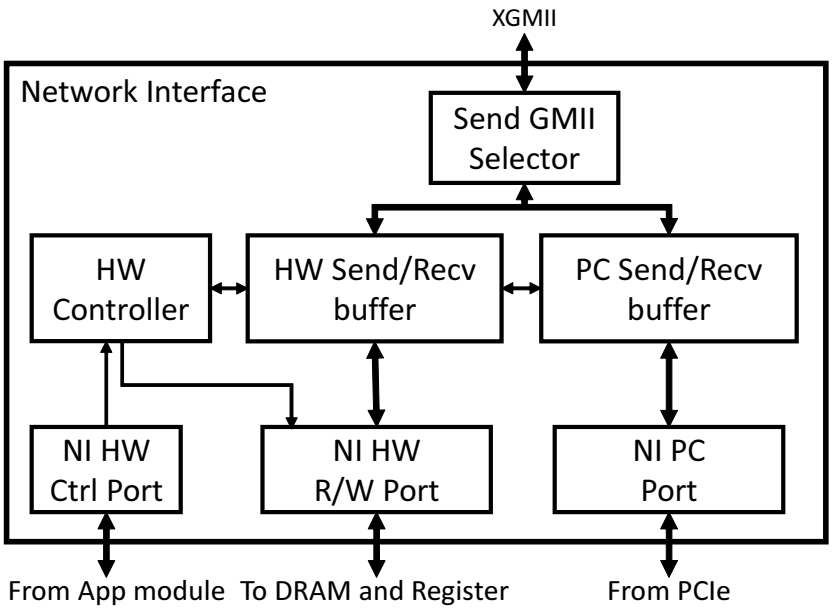

図 4.3: ネットワーク回路のブロック図

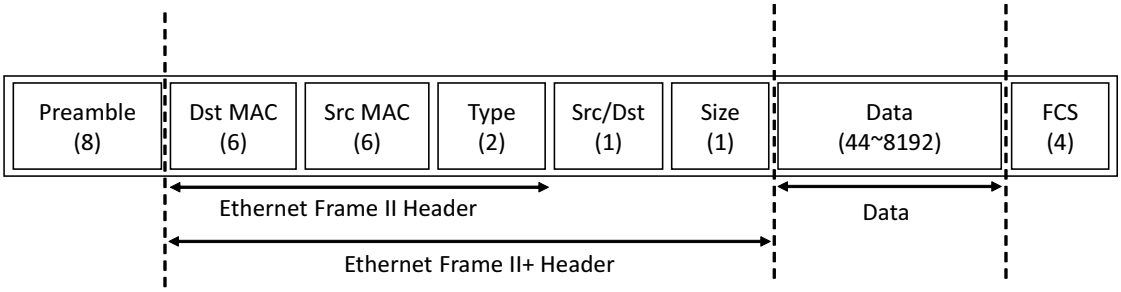

図 4.4: Ethernet II+ フレームの構成

付加したフレームである. 通常の Ethernet II フレームと同様に宛先アドレスと送信元アドレスは MAC アドレスを用いて送受信の制御を行う. フレームのタイプは, Ethernet II 規格と同様にフレームの種類を PFH System が判断するときに利用され, Ehternet II において定義されていない区間から表 4.1 に示す 6 つのタイプを定義している. ネットワーク回路は, 通信の種類ごとにタイプを選択して通信を開始する. フレームの Src/Dst は, HW 側と PC 側の送受信バッファを区別するためのデータである. この 1 オクテットの下 2bit が送信元, 上 2bit が宛先に対応し, かつそれぞれ 2bit のうち 0bit 目を PC, 1bit 目を FPGA として定義している. 例えば, 送信元が PC で宛先が HW の場合はこの 1 オクテットを 1001=0x9 に設定し, また送信元が FPGA で宛先が PC と HW 両方の場合はこの 1 オクテットを 1110=0xE と設定する. Size は, Frame Type が 0x0110 と 0x0111 のときのみに使用するデータサイズを指定する区画でありそれぞれのフレームで違う意味を持つ. Type がデータフレームを示す 0x0110

表 4.1: Ethernet II フレームタイプ

Type Code	Frame Type	Frame size
0101H	Register Write	72
0102H	Register Read data	72
0103H	Register Read Request	72
0104H	Start Address	72
0110H	Data frame	1052-8220
0111H	Data Request	72

の場合は, ペイロードのサイズを示す. ペイロードに格納するデータを 1kbyte 単位で指定する. ペイロードが 1kbyte より大きく 2kbyte 以下の場合, Data Size には 0x2 が入り, 7kbyte より大きく 8kbyte 以下のサイズの場合は 0x8 が入る. Type がデータ要求を示す 0x0111 の場合は, 要求するデータのサイズを示す. この場合, 8kbyte のフレームを 16 個まで要求できるので, 一度に最大で 128kbyte 要求できる.

またこれらに加えて, ネットワーク回路ではフロー制御のフレームも使用する. ネットワーク回路は, バッファの状態に応じて Ethernet II の規格に基づいたフロー制御フレームを自動生成する. ここで, Send/Recv buffer が保存できる最大フレーム数を M と定義する. PC Send/Recv buffer, HW Send/Recv buffer のどちらかの受信バッファに $M-1$ 以上のフレームが格納されている状態になったときに, 待ち時間を最大に設定したフロー制御フレームをネットワークに対して送信する. 設定した待ち時間の終了時に, まだ受信バッファの格納フレーム数が $M-1$ 以下でなければ, 再度待ち時間を最大にしたフロー制御フレームを送信する. 格納フレーム数が $M-2$ 以下になったら, 待ち時間を 0 に設定したフロー制御フレームを送信し, ネットワークおよび通信相手の待ち状態を解除する.

ネットワーク層

通信には, データリンク層のプロトコルを利用して各通信の方式に利用する. ホストとなる PC からリモート FPGA のレジスタに対して書き込みを行うシーケンスを図 4.5, また R リモート FPGA のレジスタに対して読み出しを行うシーケンスを図 4.6 に示す. 最初に PC は `NI PC Port` を制御し, 通信先の設定および内部バスのアドレスデータをペイロードの先頭に書き込む. これは, レジスタ書き込みおよび読み出しに共通して同じ動作である. 次に, 送信開始の信号を生成し通信相手に対してパケットを送信する. この時, レジスタ書き込み

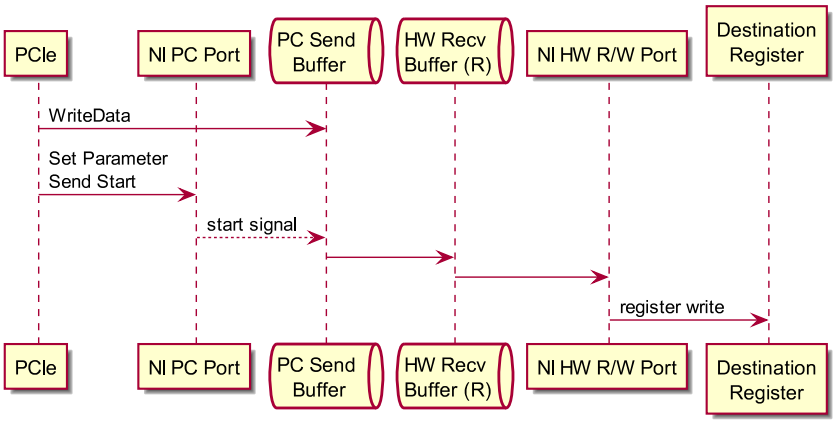

図 4.5: Register write シーケンス

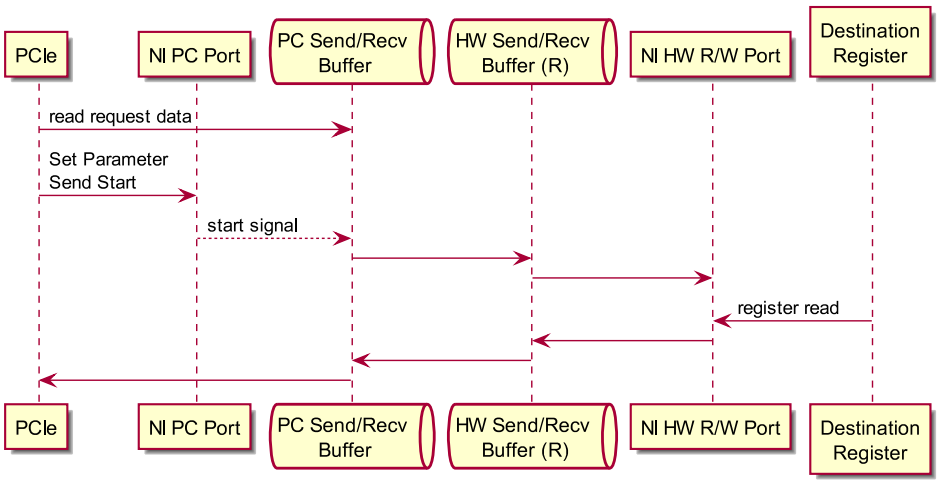

図 4.6: Register read シーケンス

であればその時点で処理を終了し, レジスタ読み出しの場合は PC 側はフレームの受信を待機する.

次に, PC の主記憶装置からリモート FPGA の DRAM にデータ送信するシーケンスを図 4.7 に, またリモート FPGA の DRAM からデータを要求し受信するシーケンスを図 4.8 に示す. データ書き込みのシーケンスにおいては, 最初に PC は `NI PC Port` を制御し, 書き込み先のアドレスを設定するフレームを送信する. 次に PC は, 送信するデータを `NI PC Port` を経由して Send Buffer へ書き込み続ける. バッファに書き込んだデータが最大サイズ (8kbyte) を超えると自動的に Send Buffer は, 対象の FPGA へデータの送信を始める. 最大サイズ以下のデータや最大サイズで割り切れない端数のデータは, `NI PC Port` を制御することで送信を行うことができる.

データ読み出しのシーケンスにおいては, PC は `NI PC Port` を制御し、読み出し先アドレスと要求データサイズを格納したデータ要求フレームを送信する. その後 PC はデータ

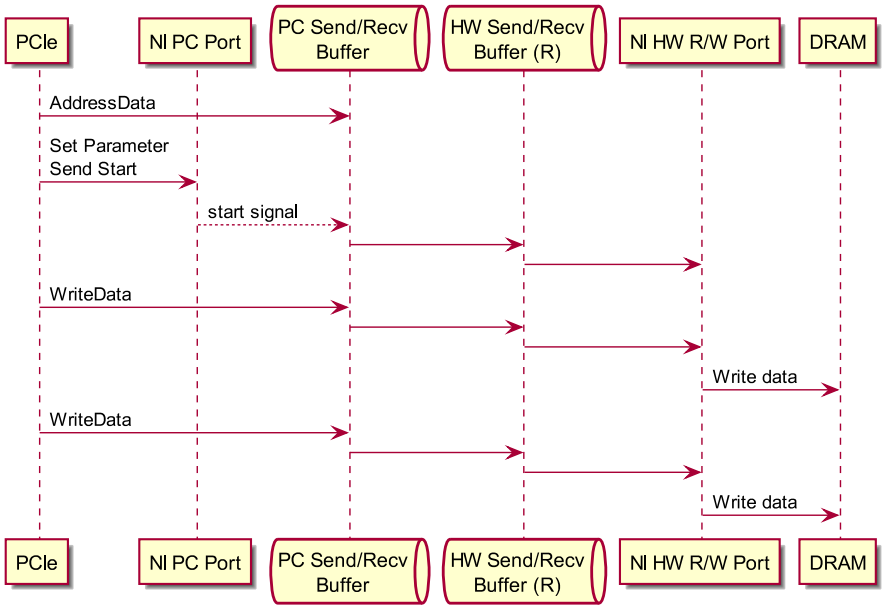

図 4.7: Data write シーケンス

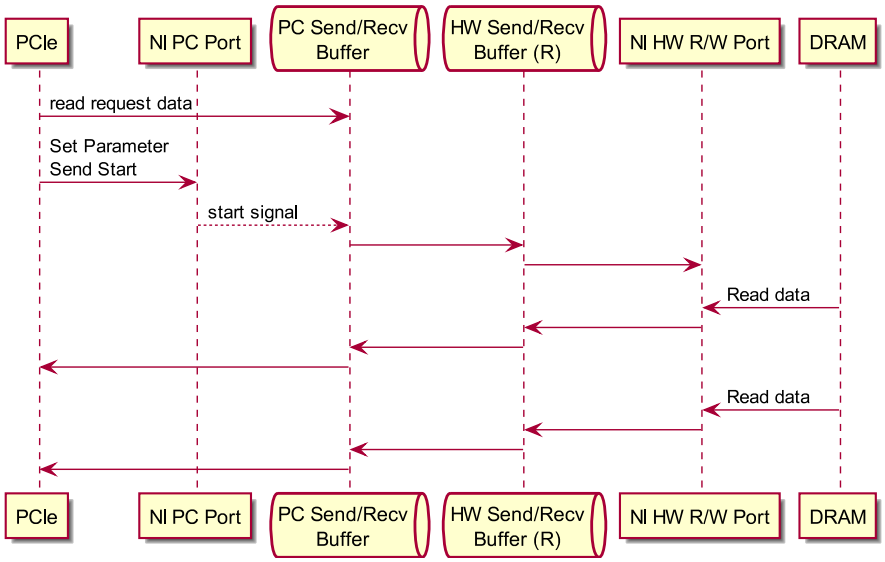

図 4.8: Data read シーケンス

が送信されてくるのを待つ．パケット送信後に PC は, Recv Buffer に受信データを `NI PC Port` と PCIe を経由して PC の主記憶に格納する．データ要求フレームを受信した FPGA は、フレーム内の読み出し先アドレスから要求データサイズ分読み出しを行う．最大サイズ (8kbyte) 毎にフレームに読み出したデータを格納してデータ要求フレームを送信してきたノードに対してデータフレームの送信を開始する．

FPGA の DRAM 同士でデータ通信を行う場合は `NI HW Ctrl Port` を制御することで実行できる．

4.2.3 API

PFH System において, PC がローカル/リモート FPGA を利用するための API を提供している. 表 4.2 に各言語によって提供している API の一覧を示す. API は, C 言語用と Python

表 4.2: PFH System の API 一覧

Function	C API	Python API	
Lib Control	HS_Init	Init	Initializing API library.
	HS_Finalize	Finalize	Close API library.
	HS_Comm_Size	-	Get the number of nodes joining in the network.
PC Data Transfer	HS_Send	Send	Write data to remote FPGA or local FPGA DRAM.
	HS_Recv	Recv	Read data from remote FPGA or local FPGA DRAM.
Register Control	HS_RgRead	regread	Read Remote/Local FPGA register.
	HS_RgWrite	regwrite	Write Remote/Local FPGA register.
App Control	HS_App_run	app_run	Start executing App module.
	HS_App_param	app_param	Set parameter to App module.
	HS_App_wait	app_wait	Wait for App Module to end.

用を実装している. C 言語での API 仕様は, 送受信とライブラリ制御を MPI ライクに記述ができるで実装できるようにしている. Python での API は, より簡易に利用できるように C 言語 API を Wrap して実装している.

Lib Control に属する API は, 初期化や終了処理などのライブラリを利用するために呼び出す必要のある API である. PC Data Transfer に属する API は, ライブラリを使用する PC の主記憶メモリと FPGA の DRAM とのデータ通信を行うための API である. Register Control は, FPGA のレジスタを制御するための API である. 最後に API Control は, FPGA に実装したアプリケーション (後述の App module) の制御を行うための API である.

提供する API は, 動的ライブラリ, ヘッダファイルおよび Wrapper として用意している. C 言語および Python は, これらのヘッダおよび Wrapper を利用することで FPGA システムにアクセスできるようになっている.

次に, API を使用した記述例について説明する. 実行を想定したアプリケーション回路の仕様について説明する. アプリケーションは引数を 1 つだけ受け取る. 処理を行うデータはアドレス 0x40000000 から 1024byte 読み出し, 処理が終わり次第アドレス 0x41000000 へ 1024byte 書き戻すものとする. 上記のような仕様のアプリケーションを FPGA に実装したとき, API を用いて実行するコードは図 4.9 のようになる. ホストとなる PC は, 1024byte の buf にデータを用意・また空の 1024byte の受信用バッファrecvbuf を用意しておきこのコードを実行する.

```
1: HS_Init(); {// Initializing library}
2: HS_Send(buf, 1024, 0x40000000, SELF); {// Write local }
3: HS_App_param(PARAM, 0); {// Setting parameter}
4: HS_App_run(SELF); {// Executing application}
5: HS_App_wait(); {// Wait for ending to application}
6: HS_Recv(recvbuf,1024, 0x41000000, SELF); {// Read local}
7: HS_Finalize(); {// Closing library}
```

図 4.9: PFH System API (C API) を利用したアプリケーションの記述例

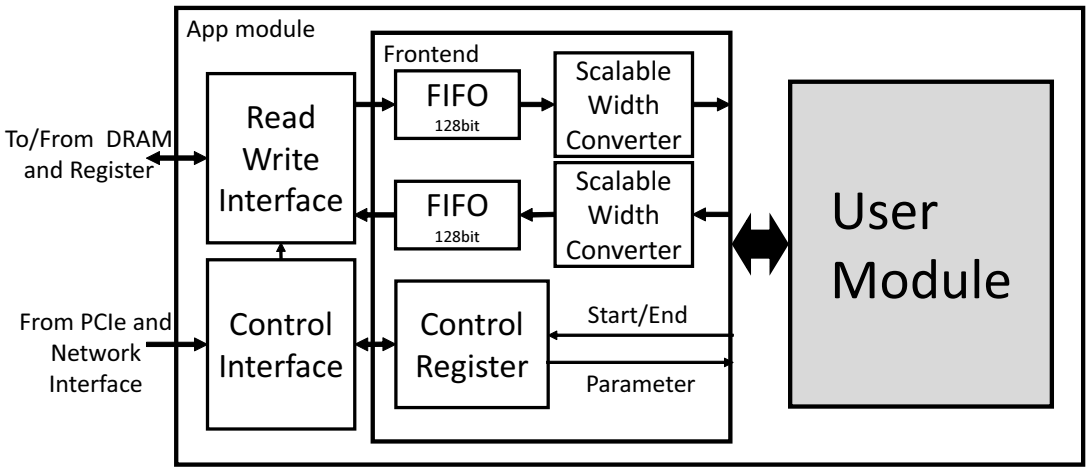

図 4.10: アプリケーションモジュールのブロック図

4.2.4 アプリケーションモジュール (App module)

図 4.10 に App module の内部構造を示す. この回路は, フロントエンドにあたる Frontend とアプリケーション本体の User Module の二種類の回路を含む. 前章で実装したアプリケーションモジュールは, メモリベースのバッファを内包する構造にしていた. 本回路では, FPGA に実装するアプリケーションと読み出し・書き込み対象との区間を短くするため, メモリベースのバッファを経由しない FIFO ベースのインタフェースとしている. Frontend は, DRAM 等のモジュールからデータの読み出し・書き込みを行うための機能を持つ. 読み出し・書き込みするデータの bit 幅は実装する User Module で決めることができるようになっている. User Module は Read,Write それぞれの Scalable Width Converter と Control Register に接続されている. User Module から指定したパラメータにより, Frontend 内の Scalable Width Converter が fifo 側の 128bit 幅と User Module 側の任意の bit 幅へ変更ができる. 例えば、User Module が 3ch カラー画像を 1pixel 毎に取り出したい場合は, User Module から Scalable

Algorithm 3 Pseudocode of the root_process

```
for Data_iter = 1 to number of data do
    if FPGA execution data then
        set FPGAData(data)
    else
        set SparkData(data)
    end if
end for
exedataS ⇐ executor.submit(Spark_process(SparkData))
exedataF ⇐ executor.submit(FPGA_process(FPGAData))
Synchronize(Spark_process, FPGA_process)
save exedataS, exedataF
```

Width Converter に対して, 24bit 幅で読み出すことを示す信号を与えることで, 24bit 毎に fifo のインタフェースで読み出し・書き込みができるようになる. Frontend による DRAM との読み出し・書き込み処理は, 実装したアプリケーションから自発的に行えるだけでなく, PC や他のノードから FPGA 内の内部バス経由での指示を行うこともできるようになっている. PC からの指示は, 先に示した `App_param`, `App_run` などの API により実現している.

4.3 分散処理手法

PFH System を用いて分散処理を行う手法について説明する. PC と FPGA の両システムを用いてそれぞれに処理を分散させる. PC 側では, オープンソースの分散処理フレームワークである Spark を用いてホストとなっている PC やネットワーク上の他の PC に処理を分散させる. 対して FPGA では, ホストとなる PC の 1Core を FPGA 管理用に割り当て, 各 FPGA に処理を分散させる. 本手法において, 分散処理に使用される FPGA には, 連続したネットワークアドレスが割り当てられる.

PFH System の動作を図 4.11 に示す. ホストとなっている PC は, Algorithm 3 における `root_process` により, FPGA 資源を管理する `FPGA_process` と PC 側の処理を管理する `Spark_process` を開始する. PFH System は, 分散処理の実行前に処理対象のデータをホスト PC 上のメインメモリに展開する. 次に, PC で処理を行う分のデータを `Spark_process` に委託する. Spark は, `root_process`, `FPGA_process`, および `Spark_process` が駆動する CPU コア以外の使用可能な資源を全て使用する. `FPGA_process` は, 図 4 に示す手順に従い, データの分配, 処理の開始, 処理の終了状態確認, およびデータの回収を行う.

FPGA で実行するアプリケーションは, 処理の実行前に PC から回路データを Configuration Module に送り実行可能な状態にしておく必要がある.

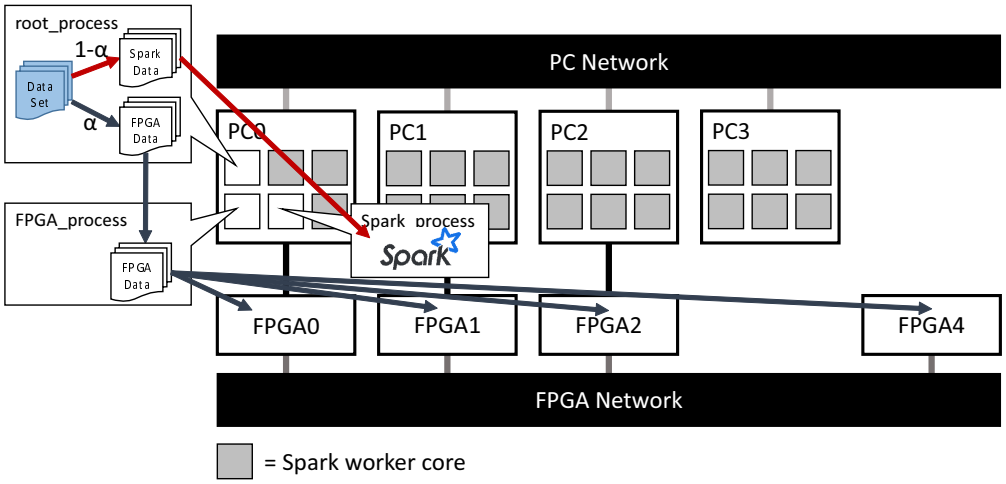

図 4.11: PFH System における分散処理手順

Algorithm 4 Pseudocode of the FPGA_process

```
set Data from FPGAData
for Data_iter = 0 to Number_of_data do
    board_address = Data_iter % NUMBER_OF_BOARD
    if board_address != LOCAL_BOARD_ADDRESS then
        Send(Data[Data_iter],board_address)
        app_run(board_address)
    else
        Send(Data[Data_iter],SELF)
        app_run(SELF)
    end if
    if board_address == NUMBER_OF_BOARD − 1 then
        app_wait()
        buffer.Append(Recv(SELF))
        for raddress = 1 to NUMBER_OF_BOARD − 1 do
            buffer.Append(Recv(raddress))
        end for
    end if
end for
return buffer
```

FPGA_process と Spark_process は, プロセスの終了時に処理結果を root_process に受け渡す.

以上の手順により,PFH System における分散処理を実現している.

表 4.3: 実装環境

Component	Node No.	Version
CPU	PC0 to PC3	Corei5-9600
OS	PC0 to PC3	Ubuntu 18.04
FPGA	FPGA0 to FPGA2 and FPGA4	Xilinx KC705 (XC7K325T)
10GBASE-SR L2 Switch	–	QSW-804-4C

4.4 実装

本章のシステム設計の節で定義した仕様を元に PFH System の実機を実装する [42]. 表 4.3 に全体の構成要素を示す. PC は, Intel Core-i5 9600 かつ OS は Ubuntu 18.04, FPGA は, Xilinx KC705 で構築する. FPGA に搭載する回路は, Vivado 2020.1.1 を用いて生成する. PC のネットワークは, Ethernet(1000 BASE-T) によりスイッチングハブ、ルータ WZR-900DHP (Buffalo Technology, Nagoya, Japan) へ接続されている. FPGA ネットワークは, 10 GBASE-SR により QSW-804-4C(QNAP Systems, New Taipei City, Taiwan) に接続して構築している. FPGA ボード上の DRAM は DDR3-1600 を搭載している. PCIe DMA Controller は, Gen2x4 の XDMA IP Core[43] を利用しており, 先に示した API はこれのデバイスドライバに付属するライブラリを用いて実装している. XGMII Phy chip は 10G Ethernet PCS/PMA IP Core[44] を使用して実装する. これらを元に実装した実機の全景を図 4.12 に示す.

図 4.12: PFH System の全景

予備実験として DRAM read/write 性能の評価を行った. PC を起点として FPGA の DRAM に対してデータの送受信を行う際のスループットを計測した. 起点となる PC を PC0 とし, Remote の FPGA を FPGA1 とする. 図 4.13 にそれぞれのスループットを示す. Local は, 起

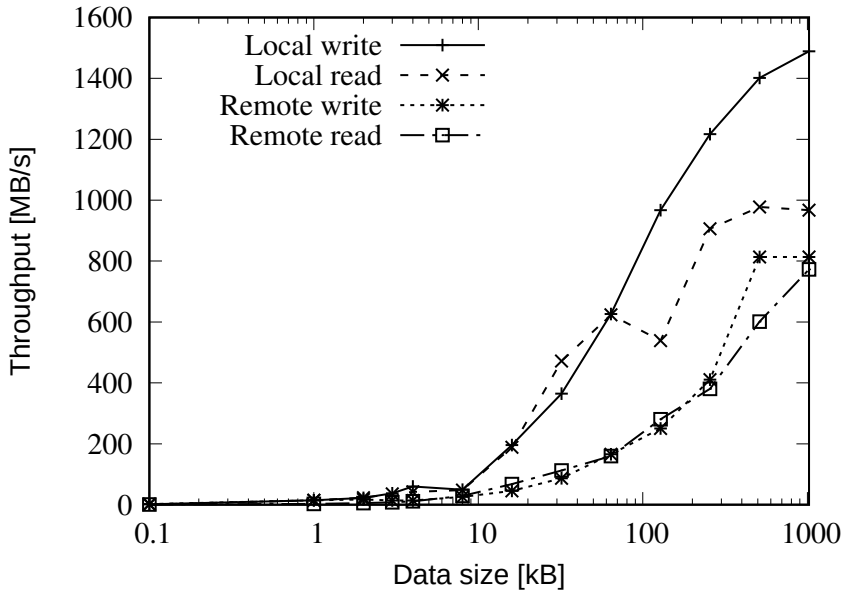

図 4.13: DRAM read/write のスループット

表 4.4: 対象 FPGA における資源使用率 [%]

	Static circuit	Maximum resources of User Module	Jpeg encoder	Median filter	Dijkstra
FF	16.40	33.56	7.71	1.65	0.12
LUT	30.60	33.56	13.65	7.75	0.43
Memory LUT	1.11	32.81	0.16	0.10	0.01
BRAM	50.11	33.71	21.24	0.79	28.99
DSP48	0.00	35.71	33.81	0.36	0.0

点となる PC に内蔵された FPGA の DRAM に対してデータの読み書きを行ったものである. Remote は, ネットワークの先にある FPGA の DRAM に対してデータの読み書きを行ったものである. スループットの計測を行った結果, ネットワークの物理層 10Gbps:1250[MB/s] に対して送信は 813.52[MB/s], 受信は 772.99[MB/s] であった. 実装した結果, 表 4.4 に示すリソースの使用量であった. 静的回路のリソースは, FF: 16.40[%], LUT: 30.60[%], Memory LUT: 1.11[%], BRAM: 50.11[%] となり, 動的部分再構成に確保できる領域 (`User Logic`) は, リソース全体の 33.56[%] であった.

PCIe や SFP+の通信帯域に対して, 62〜65%の速度しか出ていないのは通信時の制御がボトルネックとなっていると考えられる. 図 4.7,4.8 で PCIe から行う制御の切り替わりに時間がかかっている. 例えば, Address Data の制御に 0.12us かかるのに対し, Address Data と Set Parameter Send Start 間の待ち時間に 46〜75us 程度要する.

4.5 実験と結果

本実験では, PC と FPGA で実行するアプリケーションは同じ C 言語のソースコードを利用する. アプリケーションは, JPEG エンコーダ, 7*7 メディアンフィルタ, 最短経路探索 (ダイクストラ法) の三種類を対象に実行する. JPEG エンコーダは, 画像データを取り込み JPEG 圧縮したデータをバイナリで出力する. 7*7 メディアンフィルタは, 3ch(24bit) の画像データを取り込み 7*7 のピクセルでメディアンフィルタを適用した同サイズの画像データを出力する. 最短経路探索は, ノード間距離を記述したデータをを取り込みスタートからゴールまでの最短のノード番号を順に並べたデータを出力する. JPEG エンコーダと 7*7 メディアンフィルタは, 320*240*3ch のサイズの画像 1000–4000 枚を対象に実行する. 最短経路探索は, 512 ノードの経路距離データ (512*512*2byte) を対象に実行する. PC 側は, ソースコードを動的ライブラリとしてコンパイルしたものを python で呼び出せるようにしたものを実行する. FPGA 側は, Vivado HLS 2020.1 で IP 化したものを先に述べた User Module に実装し, 先に述べた API を用いて実行する. PC アプリケーションのコンパイルには, 最適化のオプションを標準のままで gcc 7.5.0 を使用する. FPGA のアプリケーション生成時には, dataflow 指示子を全体に使用する. PFH System により上記 3 つのアプリケーションを, 先に示した分散処理方法に基づき実行する. 加えてシステムを稼働させているときの電力を計測する.

実装した各アプリケーション回路のリソースは, 先に示した表 4.4 に示す.

JPEG エンコーダ, 7*7 メディアンフィルタ, 最短経路探索の 3 種類のアプリケーションにおいて分散処理を実行した時間を図 4.14, 4.15, 4.16 に示す. Distribution ratio は, PC:FPGA の分散比率を示している. 0 はすべて PC で処理をしている状態であり, 100 は全て FPGA で処理している状態である. この数値は図 4.11 の α にあたる.

これら三例において JPEG エンコーダでは, PC での処理が FPGA と比較して速く PC へ 9 割, FPGA へ 1 割の分配比率の時に一番性能が高くなった. メディアンフィルタでは, FPGA での処理が PC と比較して速く, PC へ 1 割, FPGA へ 9 割の分配比率の時に一番性能が高かった. 最後にダイクストラ法では, FPGA での処理が PC と比較して速いが, 先のメディアンフィルタの場合と違いデータの数により最高性能となる分配比率に変化があった.

PC1 台の稼働時電力は 55[W] で idle 時は 24[W], FPGA は 1 台 21[W] であった. 全ての資源を処理に使用した場合, 最大電力は 300[W] であった. また PC0 以外の PC を停止させ FPGA4 台のみで処理で実行したところ最大電力は 110[W] であった.

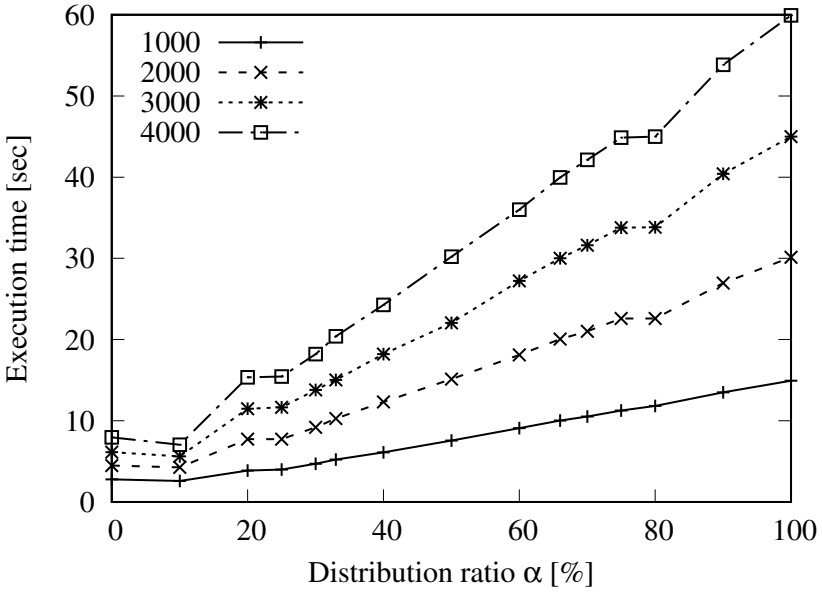

図 4.14: JPEG エンコーダの実行時間

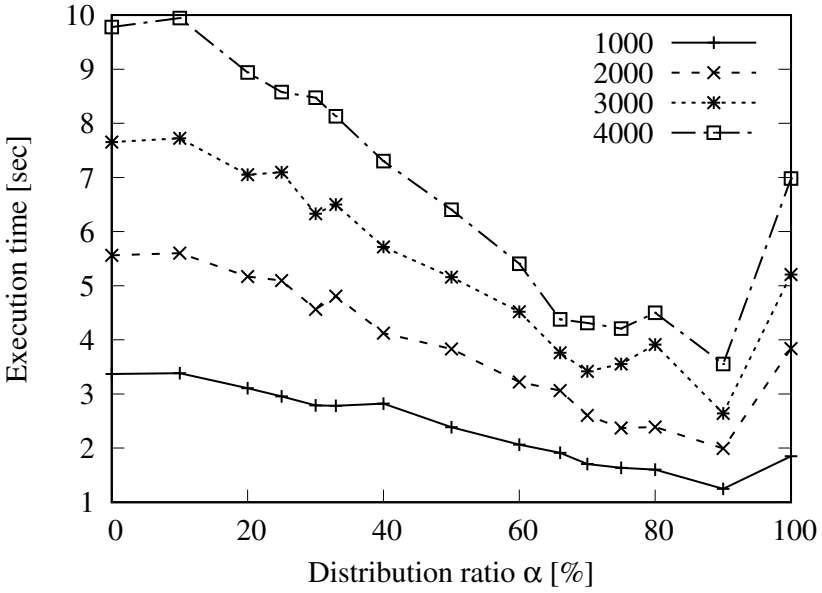

図 4.15: メディアンフィルタの実行時間

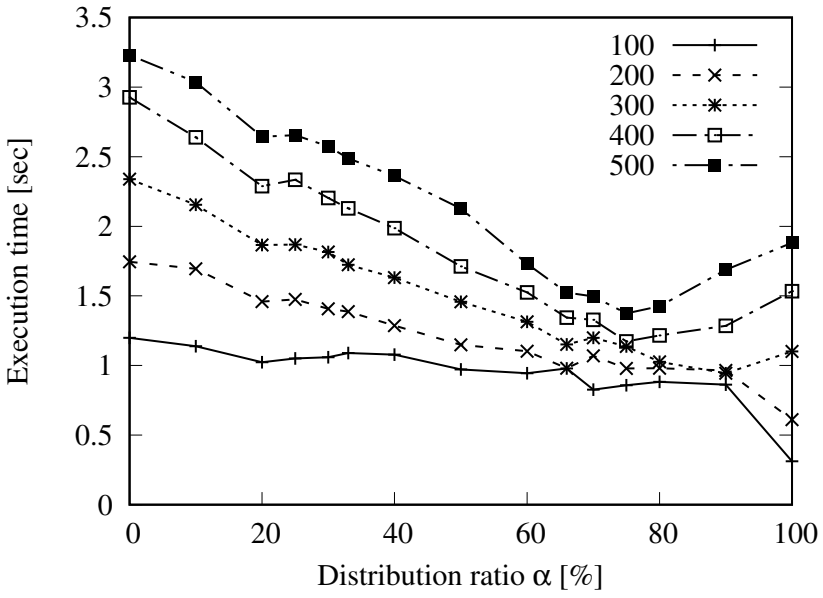

図 4.16: ダイクストラ法の実行時間

4.6 考察

PC だけもしくは FPGA の片方のシステムによる処理速度が速い場合においても, 速度において劣勢のシステム側に少しでも処理を分散することで処理時間を短縮することができる. これは図 4.14 の JPEG エンコーダの実験結果により明らかであり, 前章のリングネットワークで構成した PFH System と同様の傾向を持つ. 4000 枚の処理において PC だけの処理では 7.951[s] の実行時間に対して, PC と FPGA に 9:1 の割合で分散させた場合は 7.039[s] の実行時間で処理を終えた.

また図 4.16 のようにデータ数によって処理時間が変わる処理の場合, それに伴い最適な分散比率も変化する. 今回のダイクストラ法の実装では, 500 ケースの処理と 300 ケースの処理では最適な分配比率が変わる結果となった.

1 台の管理用の PC と 4 台の FPGA を駆動させることで電力効率を優先した構成を取ることもできる. 実験結果より, 最短経路探索における 500 ケースの実行結果は, PC と FPGA の両方を使用した最大性能時の電力量は 412.20[Ws], 管理用の PC1 台と FPGA だけを利用する構成での電力量は 206.91[Ws] であることが言える. これは, 実行時間 [s] × 最大電力 [W] による計算結果である. これにより, 電力効率は FPGA だけを利用する場合の方が 2 倍効率が良いことになる. またこの運用の延長線上として, 前章で有効性を示したマイグレーションがある. システムの構造上, 前章のマイグレーション処理が実行できるため本システムにおいても前章と同様の有効性が示される.

PFH System は柔軟な構成での実装が可能であるので, 最大性能を発揮させたい場合は PC と FPGA を複合させて構成し, 電力効率を考えたい場合は管理用の PC1 台と複数台の FPGA で構成するなど設置・運用環境に合わせた構成を取ることができる. これは, リングネットワークで実装した PFH System と同様に用途に応じて柔軟にシステムの構成を変更できる特徴を持ちながら, PC もしくは FPGA だけの処理と比較して性能の向上が得られ, また拡張性に富んだ設計であるといえる.

4.7 結言

本章では, Ehternet を FPGA ネットワークに利用した PC と FPGA を複合させた分散処理に利用できるシステムの提案を行い, 以下の 3 つの方針に基づきシステムの実装を行った.

1. 問題の性質や電力要求に応じて, PC と FPGA の柔軟な分散利用が可能な構成
2. Ethernet MII(Media Independent Interface) を持つ FPGA に広く対応する FPGA 間接続
3. 分散処理フレームワーク Spark と HLS による分散処理

JPEG エンコーダ, メディアンフィルタ, 最短経路探索の 3 つの処理を, 構築した PFH System を用いて分散処理を行った.

実験の結果, JPEG エンコーダの分散処理実行において 4000 枚の画像を処理する場合, PC だけの処理では 7.951[s] の実行時間に対し, PC と FPGA に 9:1 の割合で分散させた場合は 7.039[s] となり実行時間を終えた. これにより, PFH System は PC だけもしくは FPGA だけで処理を行うよりも一部でも片方の資源に処理を分散させることで実行時間が短縮されることを確認した. また, 分散比率をアプリケーション毎に変更することで最短の実行時間で実行できた. さらに, 管理用の PC と 4 台の FPGA だけの電力効率を重視する構成では 206.91[Ws] で処理を実行することができ, その場合の構成は元の構成で処理を実行したときの電力 412.20[Ws] と比較して電力効率に優れることを確認した.

(1) 実験より最高性能で実行できる構成と電力効率を重視する構成を使い分けることができた点, また (2)MII を使用したネットワークインタフェースが PFH System に実装できた点, さらに (3) 分散処理を行うにあたり, HLS で実装した FPGA アプリケーションと Spark を併用して実行できた点より本章の緒言で示した実装方針を満たしたシステムの提案ができたといえる. これにより, システムの運用状況に応じて電力や性能などを考慮したシステムが構築でき, 実装する FPGA ボードに依存しない内部回路構成をもち, アプリケーション

の実装に同じソースコードを用いて運用ができるという 3 つの利点が得られるシステムが実現できたといえる.

今後の課題としては, PC から FPGA を利用する際の簡易化のために API を充実させる必要がある. また, PFH System では決まった資源を利用しているため資源管理を行っていない. PFH System の FPGA の資源管理が行える仕組みを実装する必要がある. さらに、PFH System に他機種や他ベンダーの FPGA を利用し実験を行う必要がある.

第5章　Rustによるハードウェア設計記述手法

5.1　緒言

1 章でも述べたように, FPGA はアプリケーション搭載において, 電力効率の良さから研究者, エンジニアなど多方面から注目され, あらゆる分野で使用され始めている. ある実装では, CPU や GPU と比較して, 電力効率, 処理性能共に優れた論理回路ベースのシステムが実装できることが示されている [45] [46]. FPGA には, 論理回路として実装を行うため, 内部の資源が許す限りの並列性を持たせたアプリケーションが実装できる. したがって, これらに有効な画像処理や深層学習などの分野において近年使用されており, 有効性が示されてきている.

一般に, ソフトウェア開発の場合, 開発者の方針や運用状況など, 用途に応じてプログラミング言語やフレームワークを選択して開発することができる. 例えば, サーバアプリケーションの実装には, PHP や Java が広く用いられている. 一方、論理回路の開発に使用できる言語は限られているのが現状である. ハードウェア開発には, Verilog [47] や VHDL [48] などのハードウェア記述言語（HDL）が使用される. しかし, HDL は低レイヤな言語であり, 開発には長い時間がかかる傾向がある. 近年の FPGA における研究 [14] [49] [15] では, 既存のプログラミング言語を使用した論理回路開発の効率向上を目的とした HDL の研究開発が行われている.

FPGA を用いたシステム開発において, ソフトウェアのように目的や開発者の趣向に応じて開発プログラミング言語や手法を選択できることで, 論理回路開発の効率の向上, ならびに開発範囲の拡大に貢献できると考えられる. 本研究では, プログラミング言語である Rust を使用したハードウェア設計ドメイン固有言語 (DSL) を提案・実装する. 前章までの複合システムにおいては, 既存の High Level Synthesis (HLS) または, Verilog 等の RTL 記述でアプリケーションの実装を行ってきた. アプリケーションの実装に使用したい言語の HLS 処理系やそれに代わる実装方法が存在しない場合は, 変換元の言語を HLS に対応する言語に変換するトランスパイラなどを使用する必要がある.

本章では, プログラミング言語である Rust を用いたハードウェア設計環境を設計・実装し従来のハードウェア記述と比較し評価を行う. また Rust の特徴である強力なコーディン

グチェック機能を複合システムにおけるハードウェア設計においても適用する手法を提案・実装し, これにおいても評価する.

5.2 DSL および Rust プログラミング言語における概要

5.2.1 ドメイン固有言語 (DSL)

DSL とは, 特定の記述・用途において特化させた言語である. DSL の例としては, 統一モデリング言語 (UML) [50] や SQL [51] が挙げられる.

これまで, ハードウェア設計用の DSL は, 記述の簡素化と設計の効率向上を目的としてが積極的に開発が行われている. 既存のプログラミング言語を対象にハードウェア設計 DSL の開発が行われており, 例えば, Java や Python, Scala などのプログラミング言語を使用したハードウェア設計 DSL が提案されている.

Veriloggen [18] は, Python の内部 DSL として実装されたハードウェア設計 DSL である. この DSL の仕組みは, Python の抽象構文木 (AST) モジュールを使用せず, 階層化された Verilog HDL の AST を直接構築する仕組みとなっている. 元々のハードウェア設計言語である Verilog と比較して, 抽象的な設計ができるよう実装されていることが特徴である.

Chisel [16] は, Scala のハードウェア設計 DSL であり, 先に述べた Veriloggen と同様に内部 DSL として実装されたハードウェア設計 DSL である. この言語は, RISC-V や Edge TPU 等の実機の設計において既に利用された実績がある.

また Java によるハードウェア設計 DSL である JHDL [52] や, 論理合成システム PARTHENON で使用される HDL の SFL [53] などが存在する.

このように, 既存言語をベースのものや, 新たに言語を提案し実装したものが多数提案されている.

5.2.2 Rust プログラミング言語

Rust プログラミング言語は, Mozilla がサポートをするオープンソースのシステムプログラミング言語である [55] [56]. Rust は速度, 安全性, および並行性を実現することを目標に開発が進められている. 以下に Rust の特徴の詳細を示す.

1. コンパイラは, ターゲットプラットフォームまで最適化されたバイナリが生成するため, C/C++と同等またはそれ以上の処理速度で計算が可能である.

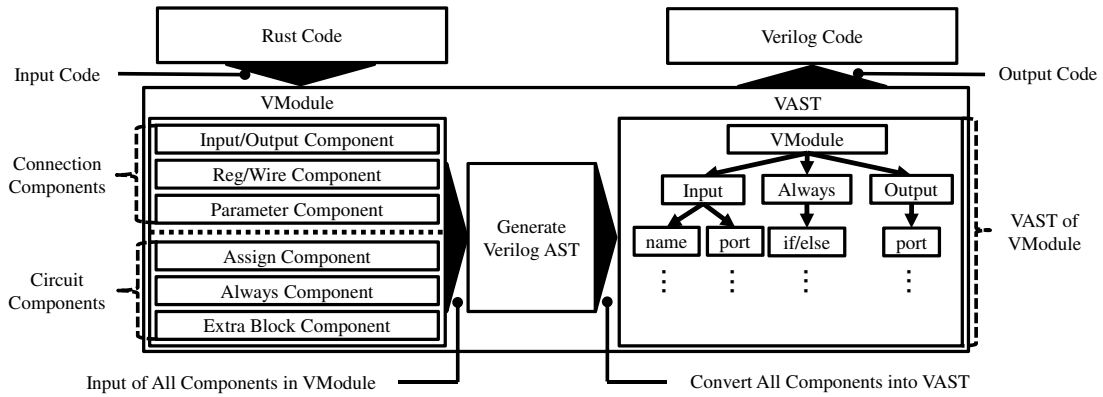

図 5.1: 提案システムにおける AST 構築手順

2. コンパイラが事前にリソースの静的検証を実行し, 不正なメモリ領域へのアクセスを防止することによって安全なバイナリを生成する.

3. 言語標準ライブラリに並行性を実現する関数を含めることによって実現されている.

5.3 システムアーキテクチャ

提案システムは, Rust プログラミング言語を RTL に変換するための DSL とその処理系である [57]. システムのアーキテクチャを図 5.1 に示す.

提案システム内の Verilog Module(`VModule`) は, ハードウェア設計データを格納する構造体である. `VModule` 内に Verilog AST(`VAST`) を構築し, 提案システムはその構築データを元に論理合成可能な RTL 記述を出力する. AST とは, 対象のプログラミング言語に必要な情報のみが抽出されたツリー構造のデータである. 一般に, コンパイラやインタプリタの中間処理として使用されており, 提案システムでは `VAST` として AST を構築している. `VAST` は, Verilog 構文と同等の構造を Rust の構造体と列挙型によって構築している. 生成する RTL 記述は Verilog2001 に準拠し, 論理合成可能な構文を主としたライブラリとして実装している. 次項より Listing 5.1 に示す LED 回路の記述例より生成する RTL 記述とコード生成のフローついて説明する.

Listing 5.1: LED 回路記述の例

```
1     let mut m = VModule::new("LED");
2
3     let clk = m.Input("CLK", 1);
4     let rst = m.Input("RST", 1);
5     let btn1 = m.Input("BTN1", 1);
6     let btn2 = m.Input("BTN2", 1);
7     let mut led = m.Output("LED", 8);
```

```
8
9     let mut fsm = Clock_Reset(clk.clone(),rst.clone())
10       .State("State")
11       .AddState("IDLE").goto("RUN", F!(btn1 == 1))
12       .AddState("RUN").goto("END", F!(btn2 == 1))
13       .AddState("END").goto("IDLE", Brank!());
14    let run = fsm.Param("RUN");
15    let fstate = m.FSM(fsm);
16
17    m.Assign(led._e(_Branch(F!(fstate == run), _Num(8), _Num(0))));
18    m.endmodule();
```

5.3.1 基本機能

モジュールの作成と入出力・内部配線・レジスタの生成

提案システムでハードウェア記述を行う場合, 最初に先に述べた VModule を作成する. Listing 5.1:1 行目のように VModule::new() のメソッドを使用することで VModule を作成する. メソッドの引数にモジュール名を文字列で渡すことにより作成することができる.

モジュールに入出力・内部配線・レジスタを追加するための trait を実装している. Listing 5.1:3–7 行のように記述することで入出力を定義できる. Input, Output は入力および出力を定義するメソッドである. 第一引数に文字列型で信号線名, 第二引数には i32 型で信号幅を要求する.

パラメータおよびローカルパラメータを記述するための Param, LParam メソッドを実装している. また, 内部コンポーネントのワイヤとレジスタを記述するための Wire, Reg メソッドも実装している. これらの引数も入出力と同じ内容を要求する.

組み合わせ回路と順序回路

提案システムには, 組み合わせ回路と順序回路を作成するためのメソッドを実装している. Assign メソッドと Function メソッドは, 組み合わせ回路の作成を想定して実装している.

Listing 5.1:17 行のように記述することで Verilog の assign 構文と同等の記述を行うことができる. Assign メソッドは, Assign_AST を引数に取る. Assign_AST は, 先に述べた入出力・内部配線・レジスタ作成の際に生成される構造体に実装している _e メソッドを利用することで生成できる. _e メソッドは assign における代入文として機能する.

Function メソッドは, Func_AST を引数に取る. Func_AST は, func 関数を使用することで生成でき, If, Else_If, Else メソッドおよび, Case メソッドによる分岐の実装ができる.

Always メソッドは, 順序回路の作成を想定して実装している. Listing 5.2 に, このメソッドを使用した順序回路の記述例を示す. Always メソッドは Always_AST を引数に取る. Always_AST は Posedge, Negedge, Nonedge 関数により生成される. Nonedge 関数は引数を必要とせず, 順序回路の駆動信号を指定していない Always_AST を生成する. Posedge, Negedge 関数は, 引数に順序回路の駆動信号となる信号を取る. Listing 5.2:1 行のように, それぞれと同名のメソッドをメソッドチェーンで実装できるようになっている. Listing 5.2:2–8 行のように先に述べた Function メソッドと同様に分岐文の記述ができるようになっている.

Listing 5.2: 順序回路の記述例

```
1 m.Always(Posedge(clk.clone()).Posedge(rst.clone())
2     .If(F!(rst != 1), Form(F!(done =0)))
3     .Else_If(F!(btn0 == 1), Form(F!(data = 10)))
4     .Else(Form(F!(data = 20))
5           .Form(F!(done = 1)))
6     .Case(select.clone())
7           .S(_Num(1),Form(F!(o_1 = 1)))
8           .S(_Num(2),Form(F!(o_1 = 2))));
```

5.3.2 拡張機能

提案システムは, Verilog を元にした記述の他に, 複雑な回路の生成を補助する記述手法を実装している.

Finite State Machine (FSM)

提案システムは, 有限オートマトンの作成を補助する機能を実装している. FSM の実装例を Listing 5.1:9–15 行に示す. この例では, IDLE, RUN, END の 3 つの状態を作成し, その状態遷移について定義している. FSM は, Clock_Reset 関数の第一引数に駆動信号, 第二引数にリセット信号を指定することで FSM 構造体を作成することができ, 作成した FSM 構造体は VModule に実装した FSM メソッドの引数として指定することで組み込むことができる. FSM 構造体の State メソッドは, 引数に文字列型を要求する. これにより FSM 内部の状態遷移レジスタの名前を変更することができる. Listing 5.1:10 行ではレジスタの名前を “State” にしている. AddState メソッドは, 引数に指定した文字列の状態を追加する. goto メソッドは二つの引数を取る. 第一引数は直前に作成した状態から移動する先の状態名を要求し, 第二引数は状態を遷移する条件を要求する. from メソッドは, 第一引数は直前に作成した状態へ

移動する遷移元の状態名を要求する. 第二引数は goto と同様である. また, 作成された状態名を取得したい場合は, Param メソッドに取得したい状態名を指定することで取得できる.

AXI ポート作成

提案システムには, 複雑な既存インタフェースの作成を補助する機能を実装している. ここでは, AXI ポートの特に Full および Lite のスレーブポート作成の補助を行う機能を実装する.

Listing 5.3 に Lite-Slave ポートの記述例を示す.

Listing 5.3: axi-lite インタフェースの記述例

```
1    let mut al = VModule::new("axi_interface");
2        let clk = al.Input("clk", 0);
3        let rst = al.Input("rst", 0);
4
5        let mut axi = AXIS_Lite_new(clk, rst);
6        axi.NamedRegSet("Calc_A");
7        axi.NamedRegSet("Calc_B");
8        axi.NamedRegSet("Output_calc");
9
10       let a = al.Output("o_A", 32);
11       let b = al.Output("o_B", 32);
12       al.Assign(a._e(axi.NamedReg("Calc_A")));
13       al.Assign(b._e(axi.NamedReg("Calc_B")));
14
15       let w = al.Wire("write_en_cdata",0);
16       al.Assign(w._e(_Num(1)));
17
18       let calc = al.Input("i_Calc", 32);
19       axi.RegWrite(w, calc);
20
21       al.AXI(axi);
```

Listing 5.3:5 行に示すように, Lite-Slave ポートは AXIS_Lite_new 関数により作成される. この関数は, 先に述べた Clock_Reset 関数と同じものを引数として要求する. Listing 5.3:6–8 行の記述に示すように NamedRegSet メソッドにより, インタフェースの内部レジスタを追加する. 引数には, 作成するレジスタの名前を文字列型で要求する. レジスタを呼ぶには Listing 5.3:12,13 行のように NamedReg メソッドを使用する. またレジスタを AXI インタフェースの外から内容を書き換える場合, Listing 5.3:19 行のように RegWrite メソッドを使用する.

Listing 5.4 に Full-Slave ポートの記述例を示す.

Listing 5.4: axi-full インタフェースの記述例

```
1     let mut al = VModule::new("axi_full_interface");
2         let clk = al.Input("clk", 0);
3         let rst = al.Input("rst", 0);
4
5         let mut axi = AXIS_new(clk, rst);
6         axi.OrderRegSet(64);
7
8         al.AXI(axi);
```

Listing 5.4:5 行に示すように, Full-Slave ポートは AXIS_new 関数により作成される. Listing 5.4:6 行のように OrderRegSet メソッドによりアドレスサイズを設定することができる.

5.3.3 Verilog コード出力

VModule に構築された VAST は, endmodule メソッド呼び出すことにより, VModule 内のバッファに Verilog コードの文字列を生成する. Listing 5.5 に, 先の Listing 5.1 のソースコードから生成された Verilog コードを示す.

Listing 5.5: LED 回路記述の出力結果

```
1  module LED (
2      input CLK,
3      input RST,
4      input BTN1,
5      input BTN2,
6      output [7:0] LED
7  );
8      // ----Generate local parts----
9
10     localparam IDLE = 0;
11     localparam RUN = 1;
12     localparam END = 2;
13     reg [31:0] State;
14     reg [31:0] State_Next;
15
16     // ----Generate assign component----
17
18     assign LED = (State==RUN)? 8: 0;
19
20     // ----Extra component set----
21
22     always@(posedge CLK or posedge RST) begin
23         if (RST == 1) begin
24             State <= IDLE;
25         end
26         else begin
```

```
27              State <= State_Next
28          end
29      end
30      always@(posedge CLK) begin
31          case(State)
32              IDLE : begin
33                  if(BTN1==1&&RST!=1)
34                      State_Next <= RUN;
35              end
36              RUN : begin
37                  if(BTN2==1)
38                      State_Next <= END;
39              end
40              END : begin
41                  State_Next <= IDLE;
42              end
43          endcase
44      end
45  endmodule
```

生成した Verilog コードは, genPrint メソッドでターミナル上に, getFile メソッドで引数に指定した文字列のファイルに出力する.

5.3.4 FPGA 設計における警告と提案

Rust プログラミング言語におけるコーディング規約を元にした警告と提案のコーディングチェッカーを実装する. 入出力と内部コンポーネントの命名規則, 長すぎる算術式, 順序回路でのエッジ信号における立ち上がりおよび立ち下がりの混合, および同じ構文での組み合わせ回路と順序回路の混在において警告と提案を行うように実装する.

図 5.2 は, 先に示した提案システムにコーディング規約に基づき警告と提案を行う方法を追加した提案システムの構造を示している. 提案システムの処理系は, この VModule 内の VAST を走査して Checking list を生成する. 警告・提案生成器は, Checking list を使用しコーディング規則に基づいて警告と提案を出力する. コーディング規約の検出方法は, VModule のメソッドとして実装する.

5.4 実験

5.4.1 実験 1

提案されたシステムを使用して生成したソースコードの行数と出力した VerilogHDL コードの行数を比較する. 生成された Verilog HDL の空白行とコメント行は, 行数として計測

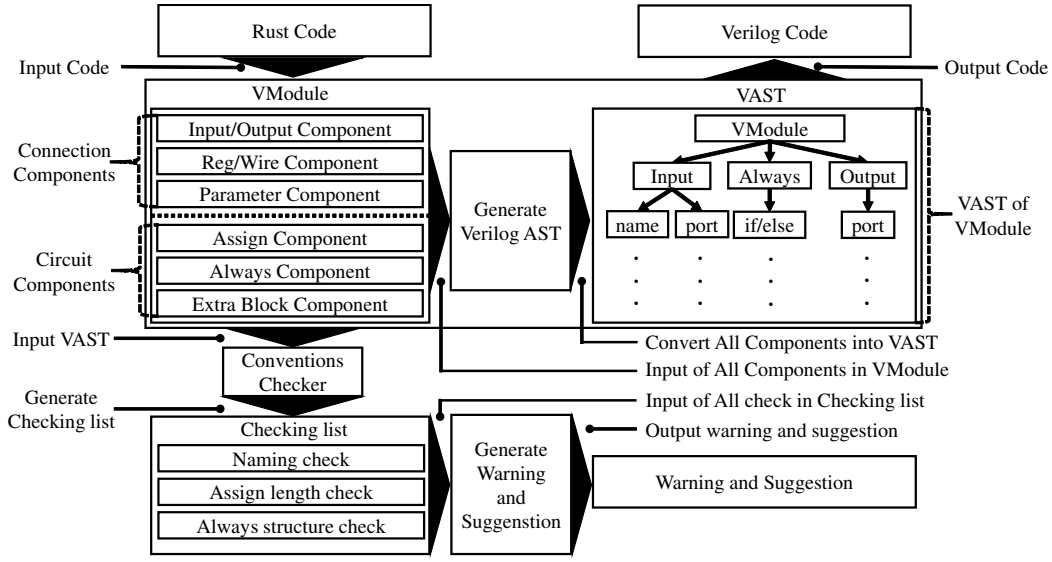

図 5.2: コーディング規約チェッカを追加した提案システムの構成

しない. Listing 5.1 に示した LED 回路に加えて, First-In First-Out (FIFO) および, Universal Asynchronous Receiver/Transmitter (UART) の回路を提案システムを用いて実装した [58].

図 5.3 および表 5.1 に提案システムと Verilog のソースコード行の数と行数削減率の実験結果を示す. 図 5.3 に示すように, LED 回路には FSM 実装機能を用いているため, コードの

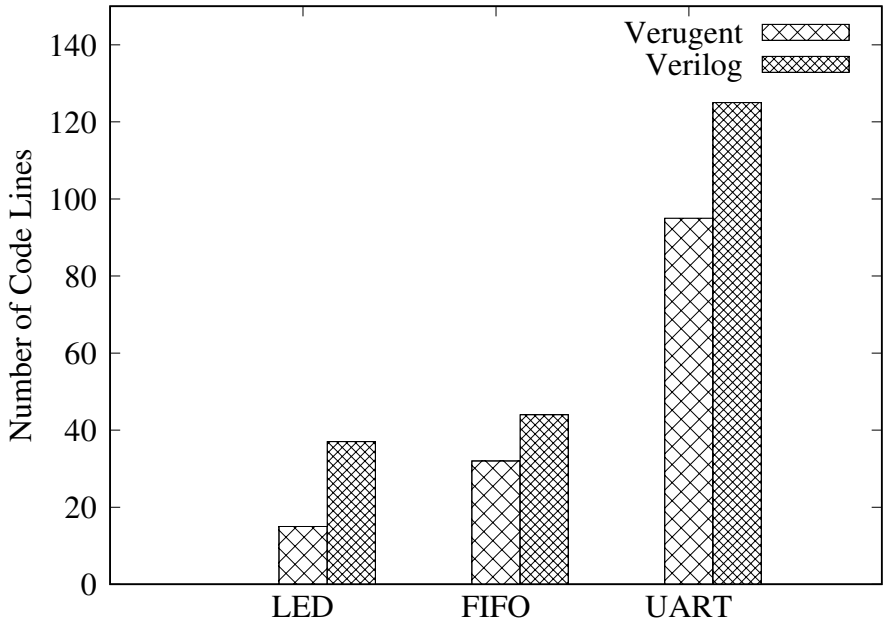

図 5.3: 提案システムの記述と生成記述におけるコード行数

短縮がされており生産性が高いことが示される. FSM 構造体は, コード出力時にステートの定義パラメータなどを自動生成するため, より少量の記述で効率良く回路の実装ができる.

表 5.1: 提案システムでの記述行数と生成コードの行数における比率

Circuit	$\frac{ProposedSystem}{Verilog}$ [%]
LED	40.5
FIFO	72.7
UART	76.0

表 5.2: FPGA 内部の資源使用量

Resource / Process	FF	LUT	Memory LUT	BRAM
Synthesis	230 (0.49 [%])	262 (0.22 [%])	0 (0.0 [%])	0 (0.0 [%])
Implementation	230 (0.49 [%])	262 (0.22 [%])	0 (0.0 [%])	0 (0.0 [%])

FIFO や UART の回路の記述量は, LED 回路ほどではないが, およそ 30%程度の行数を削減ができている. これにより Verilog の記述よりおよそ 2/3 の行数に短縮した記述で回路の設計ができることが示された.

5.4.2 実験 2

この実験では, 提案システムが FPGA へアルゴリズムを搭載した回路の実装が可能であるかを評価する. 実装には, ザイリンクス FPGA ZyboZ7-20 と開発ソフトウェア Vivado2018.2 を使用する. 本実験では, 山登り法を実装対象のアルゴリズムとしこれを提案システムにより記述し実装する. この記述により出力された Verilog に対して論理合成・配置配線を行った結果, 表 5.2 に示すリソースの使用量であった. この回路では, 論理合成と配置配線で使用される FPGA リソースは, どちらも FF は 230, LUT は 262 を使用する. Memory LUT と BRAM は, メモリを参照する回路ではないため使用されない. また, 生成した回路を実機に搭載し動作を確認することができた. これにおいては, 提案システムでの記述で 82 行, 出力された Verilog のソースコードで 125 行と Verilog で記述する場合と比較して 65.6%の記述で効率的にアルゴリズムを実装することができた.

したがって, 提案システムにより記述されたアルゴリズムの回路記述が実機に搭載できることが明らかになった.

5.4.3 実験 3

この実験では, 提案システムが行う 4 つの “ 警告と提案 ” に対する事例を示す [59].

List 5.6 は, register と parameter の命名における検出の例について示している. module, register, wire および Input/Output (I/O) ports の名前は, コーディング規則に従って記述を検出する. Rust のコーディング規約では, 変数と構造体の名前を snake_case で記述し, 型と trait の名前を CamelCase で記述することを推奨している. 提案手法において, Rust プログラミング言語のコーディング規則に基づいて警告と提案を実行する. Rust コンパイラでは, 変数が snake_case で記述されているのが望ましいが CamelCase で記述されている場合において, 警告と提案を出力する. 提案システムにおいても, レジスタまたはワイヤの名前が snake_case でない場合は警告を表示し, CamelCase の場合は snake_case への変換例示す.

Listing 5.6: Warning and suggestion of naming.

```
V-Warning: Register is not snake_case. (Register name :DATA)

V-Warning: Parameter is not CamelCase. (Parameter name :numwords)
This is better for coding. --> Parameter name: Numwords

*Note*: It is recommended that you write your code based on a specific
    naming convention.
The naming convention for the programming language Rust (RFC 430) is as
    follows:
** Modules, functions, and variables must be written in snake_case.
** Types and traits must be written in CamelCase.

In accordance with this, it is appropriate to describe the hardware based
     on the following rules.
The naming convention for the programming language Rust (RFC 430) is as
    follows:
** Modules, functions, and variables must be written in snake_case.
** Types and traits must be written in CamelCase.

In accordance with this, it is appropriate to describe the hardware based
     on the following rules.
** Modules, functions, wires, regs and input / output must be written in
     snake_case.
** Parameters and local parameters must be written in CamelCase.
```

次に Listing 5.7 では, 長い記述における検出例を示す. 提案手法は, Assign 構文において 8 つよりも多いオペランドの計算を記述した場合において長い記述であると検出する. これは, 可読性に劣る, 信号の遅延を引き起こす回路を生成するなどの理由により長い記述を抑制するためである.

Listing 5.7: Warning and suggestion of too long Assign syntax.

```
V-Warning: Too long assignment. (length: 10)
```

```
*Note*: Arithmetic operations that are too long significantly reduce
    readability.
        Therefore, the arithmetic operation must be divided into
            multiple short descriptions.
For example:
--Not recommended case--
module.Assign(smooth._e( (img_data_0_0 + img_data_0_1 + img_data_0_2 +
    img_data_1_0 + img_data_1_1 + img_data_1_2 + img_data_2_0 +
    img_data_2_1 + img_data_2_2) / 9 );
--Better case--
module.Assign( line0._e(img_data_0_0 + img_data_0_1 + img_data_0_2) );
module.Assign( line1._e(img_data_1_0 + img_data_1_1 + img_data_1_2) );
module.Assign( line2._e(img_data_2_0 + img_data_2_1 + img_data_2_2) );
module.Assign( smooth._e( (line0 + line1 + line2) / 9 );
```

次に Listing 5.8 は, Always 構文を使用して組み合わせ回路を記述している場合, 提案システムでは警告と提案が表示されることを示している. Always メソッドでは, Verilog での記述と同様に順序回路と組み合わせ回路の両方の記述を行うことができる. しかしコードスタイルの統一のため, 提案システムは順次回路と組み合わせ回路の混合を避ける記述を提案する.

Listing 5.8: Warning using the Always syntax for combinatorial circuits.

```
*Note*: It is recommended to use ‘‘always’’ as a sequential circuit.
When generating a combinational circuit, readability is improved by using
     ‘‘function’’ or ‘‘assign’’.
```

最後に List 5.9 は, 信号混在についての警告を示している. ハードウェア構造から, 立ち上がり信号と立ち下がり信号を混在させることは推奨されていない. 提案手法は, Always において立ち上がり信号と立ち下がり信号が混在していることを警告する.

Listing 5.9: Warnings in cases of mixed rising and falling signals.

```
V-Warning: A description that mixes ‘‘positive’’ and ‘‘negative’’ is not
     recommended.(always number: 0)
*Note*: The driving edge positive and negative are mixed.
It is recommended that the driving edge of ‘‘always’’ be either positive
     or negative.
```

これらのケーススタディより, 提案手法はコーディングを改善するために, コーディング規約からの逸脱に対する警告と提案を行う能力があること示した.

5.5 結言

本章では, Rust プログラミング言語を用いたハードウェア設計環境を設計・実装し評価を行った. また Rust の特徴である強力なコーディングチェック機能を提案するハードウェア設計環境においても適応する手法を提案・実装しこれにおいて評価を行った.

提案システムは, Verilog と比較して 2/3 ほどの行数での記述で実装できることを示し, これにより効率の良い回路記述ができることが示した. また, 山登り法のアルゴリズムを実装し, 提案システムにより出力した回路記述が FPGA の実機に実装できることを確認した.

さらに提案システムにおいて, 4 つのケーススタディにより, Rust プログラミング言語におけるコーディング規約を元にした警告と提案を行うことのできる能力があることを確認した.

これらより, 複合システムにおけるアプリケーション記述に十分な機能を持つシステムが実現できたといえる.

今後は提案システムを効率的に利用するために, 実アプリケーションの実装を行い, 性能評価を行う必要がある. また, より効率的な記述や性能の良い回路を生成するためのメソッドや関数の実装が必要である. さらに, 本システムを基盤に HLS ツールの実装を行うことも検討する.

コーディングチェック機能においては, 警告と提案における広範囲なシミュレーションシナリオを作成する必要がある.

第6章　結論

本論文では, 2 章で示した指針をもとに中小規模の環境で使用することを想定した並列リコンフィギャラブル計算機システムの提案を行った. 加えて, 提案した並列・分散システムでの FPGA アプリケーション記述を想定して Rust による新たなハードウェア設計環境の提案を行った.

3 章にて FPGA ネットワークをリングネットワークで構築した PFH System を設計し, 並列処理, 分散処理, マイグレーションの 3 種類のシチュエーションで運用することが可能であることを示した. 分散処理による JPEG エンコーダの実行では,4000 枚の処理において PC-FPGA の比率が 9:1 の場合に実行時間が 8.0[s] となり, 実行時間は最も性能が良い結果となった. マイグレーションにおいて, 1 台の PC から 4 台の FPGA に処理を要求した後, PC の電源を切ることで消費電力を 31.7[%] 削減できることを示した. PC-FPGA 複合システムが多様な用途で利用できることを示した.

4 章にて FPGA ネットワークを Ethernet ベースに変更した PFH System を設計, 実装し, 利用用途を分散処理に限定し性能を評価した. 結果, 最高性能で実行できる構成と電力効率を重視する構成を使い分けることができ, さらに分散処理を行うにあたり, HLS で実装した FPGA アプリケーションと Spark を併用して実行できるシステムの提案し, 運用が可能であることを示した. 管理用の PC と 4 台の FPGA だけの電力効率を重視する構成では 206.91[Ws] で JPEG エンコーダの処理を実行することができ, その場合の構成は元の構成で処理を実行したときの電力 412.20[Ws] と比較して電力効率に優れることを確認した. これにより, PFH System は, 1 章で示した“想定する環境に点在する余剰資源を有効に活用でき, 用途に合った構成がとれ多様な形態で構築できるシステムの提案”を満たしたシステムの提案が出来たといえる.

5 章では, PFH System に FPGA アプリケーションを実装することを想定した, Rust プログラミング言語を用いてハードウェア実装を行う手法を提案した. 実験により, 提案する設計環境による記述が Verilog の RTL 記述と比べ 2/3 行程度の少ない行数で記述できること

を確認した. また, Rust のコーディング規約をベースにした規約を本設計環境にも適用することで, 実装した処理系が, 想定したシナリオにて“ 記述における警告 ”および“ 推奨する記述を提案 ”を行う能力があることを確認した. これにより, 提案システムが FPGA アプリケーションの記述において有効であることを示すことができたといえる.

今後の課題として, 他シリーズもしくは他ベンダーの FPGA を用いて PFH System を実装することで更に柔軟に利用できることを示す必要がある. これにより, FPGA 評価ボードを機種問わず複数持っている場合, 容易にシステムの構築ができるようになるであろう.

また Rust プログラミング言語を用いたハードウェア設計手法については, 実アプリケーションを実装して更に性能評価を行う必要がある. さらに本研究で実装した処理系を基盤に, 高位合成処理系の実装を行うことも視野に入れる必要がある.

現状よりも簡単にシステムの拡張, つまりはノードの数を容易に増やすことができ, Rust に限らずあらゆるプログラミング言語において並列リコンフィギャラブル計算機システムを利用できる未来が来ることを切に願う.

謝辞

本研究の一部は, Xilinx University Program による支援によります. 本研究の全般にわたり, 研究の機会を賜ると同時に直接の御指導, 御教示をいただき, 長期に亘る在学期間においてあらゆる点で支えていただきました岡山理科大学工学部情報工学科小畑正貴教授に心から感謝致します. まだ研究室に配属されていない学部 1 年生からの学生生活だけでなく, 研究室配属後の生活においても様々な助言をいただきました. 私のように, 興味に流されがちな学生の面倒を 6 年間も見ていただいたことに, 改めて感謝申し上げます.

並列計算機システムの論文作成に関しまして, 貴重なご意見をいただき, また実験環境の実現において有益なご意見, ご助言をいただきました岡山理科大学工学部情報工学科 上嶋明准教授, 尾崎亮准教授に感謝致します. 技術も知識も不足していた私が, このように研究が続けられたのは, 共同研究者として研究の機会をいただけたからだと感じています. また, FPGA 開発環境の開発とその論文化, およびその過程において広く協力をいただき, 加えて公私ともに暖かい激励をくださった岡山理科大学工学部情報工学科 小田哲也講師に心から感謝致します. 小田先生の人望のおかげで, 研究会や国際会議など国内外問わず, 多くの研究者と知り合うことができ, 非常に充実した研究生活を送ることが出来ました. ここに改めて御三方に感謝申し上げます.

本論文を提出するにあたり主査の小畑正貴教授、副査の上嶋明准教授に加え 3 名の先生に副査になっていただきご助言を賜りました. 岡山県立大学情報工学部 佐藤 洋一郎教授, 岡山理科大学工学部 信吉輝己教授, 岡山理科大学工学部 島田英之教授には厚くお礼申し上げます.

Rust による FPGA 開発環境の実現に関して, 有益な御助言と議論の機会をいただいた岡山大学工学部情報系学科 乃村能成准教授に感謝致します. また, 同分野の研究者との議論の機会をいただいた電子情報通信学会リコンフィギャラブルシステム研究会, および高位合成友の会の皆様に深く感謝の意を表します.

並列計算機システムの研究を行う上で, 暖かく見守っていただいた岡山理科大学工学部情報工学科の教職員および学生, 卒業生の諸氏に深く感謝します.

博士後期課程在籍時において同学年としてともに切磋琢磨し研究を進めてきた宮本直輝

氏, 高谷健太氏, また修士・博士後期課程の在籍時に心の支えとなってくれた友人各位に感謝します.

最後に, 本研究を進めるうえで自身の学生生活における心身両面において支えてくれた家族に感謝の言葉を送ります.

2021 年 9 月吉日

高野 恵輔

参考文献

[1] John L. Hennessy, David Andrew Patterson, “Computer Architecture 6th Edition”,“ コンピュータアーキテクチャ 定量的アプローチ 第 6 版 ”, 中條拓伯, 天野英晴, 鈴木貢 訳, 2019.

[2] 天野英晴,“ 並列コンピュータ 非定量的アプローチ ”, オーム社, 2020.

[3] Douglas E. Corner, “Essentials of Computer Archtecture 2nd Editio”,“ コンピュータアーキテクチャのエッセンス 第 2 版 ”, 吉田邦夫 訳, 2020.

[4] P. Andrew, C. Adrian, C. Eric, C. Derek, D. John, E. Hadi, F. Jeremy, G. Jan, H. Michael, H. Scott, H. Stephen, K. Joo-Young, L. Sitaram, P. Eric, S. Aaron, T. Jason, X. P. Yi, B. Doug, G. G. Prashanth and P. Simon, “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”, In Proc. 41th Annual International Symposium on Computer Architecture(ISCA), pp.13–24, 2014.

[5] E. Ghasemi and P. Chow, “Accelerating Apache Spark Big Data Analysis with FPGAs”, 2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), pp.737–744, 2016.

[6] 深沢圭一郎,“ HPC における CPU, アクセラレータの動向 ”, 日本シミュレーション学会論文誌, Vol.36, No.2, pp.76–78, 2017.

[7] 稲富雄一, 垣深悠太, 小野貴継, 井上弘士,“ 電力制約型スーパーコンピュータにおける性能モデリング ”, 情報処理学会, 2016-HPC-155, No.17, pp.1–9, 2016.

[8] 末吉敏則, 天野英晴,“ リコンフィギャラブルシステム ”, オーム社, 2005.

[9] 天野英晴, 尼崎太樹, 飯田全広, 泉知論, 長名保範, 佐野健太郎, 柴田裕一郎, 末吉敏則, 中原啓貴, 張山昌論, 丸山勉, 密山幸男, 本村真人, 山口佳樹, 渡辺実,“ FPGA の原理と構成 ”, オーム社, 2016.

[10] Y. Kono, K. Sano and S. Yamamoto, “Scalability analysis of tightly-coupled FPGA-cluster for lattice Boltzmann computation”, In Proc. 22nd International Conference on Field Programmable Logic and Applications (FPL 2012), pp.120-127, 2012.

[11] 田中大智，Antoniette Mondigo，佐野健太郎，山本　悟，“密結合 FPGA クラスタのための直接網の設計と評価”, 信学技報，vol.117，no.379，RECONF2017–62，pp.71–76，2018.

[12] 高木大智，趙謙，久我守弘，尼崎太樹，飯田全広，末吉敏則，“FPGA の高速シリアル通信を用いたクラスタコンピューティング環境の一検討，” 火の国情報シンポジウム 2018，no.A6–1，pp.1–6，2018.

[13] 三好 健文，“FPGA 向けの高位合成言語と処理系の研究動向”, コンピュータソフトウェア, Vol.30, No.1, pp.76–84, 2013.

[14] S. Windh et. al., “High-Level Language Tools for Reconfigurable Computing”, Proc. of IEEE, Vol.103, No.3, pp.390–408, 2015.

[15] B. Saravanakumaran and M. Joseph, “Survey on Optimization Techniques in High Level Synthesis”, Proc. of CCSIT-2017, pp.11–21, 2017.

[16] J. Bachrach et al., “Chisel: Constructing Hardware in a Scala Embedded Language”, Proc. of DAC-2012, pp.1212–1221, 2012.

[17] N. Watanabe and A. Nagoya, “Design of Hardware Description Language FSL Based on Object-Oriented/Functional Programming Languages”, IEICE-RECONF, Vol.115, No.228, pp.27-32, 2015.

[18] “Veriloggen”, `https://github.com/PyHDI/veriloggen`, [Available Online] 15 May, 2021.

[19] 上野知洋, 佐野健太郎，“FPGA クラスタとその相互結合網の研究動向”, 電子情報通信学会誌, Vol.103, NO.4, pp.421–425, 2020.

[20] Antoniette MONDIGO, Tomohiro UENO, Kentaro SANO, Hiroyuki TAKIZAWA, Scalability Analysis of Deeply Pipelined Tsunami Simulation with Multiple FPGAs, IEICE Transactions on Information and Systems, Vol.E102.D, No.5, p.1029–1036, 2019.

[21] Kohei Nagasu, Kentaro Sano, Fumiya Kono, Naohito Nakasato, “FPGA-based Tsunami Simulation: Performance Comparison with GPUs, and Roofline Model for Scalability Analysis”, Journal of Parallel and Distributed Computing, Vol.106, pp.153–169, 2016

[22] Kaneda Takahiro, Sakai Ryotaro, Nishikawa Naoki, Hanawa Toshihiro, Tsuruta Chiharu, Amano Hideharu, “Performance Evaluation of PEACH3: Field-Programmable Gate Array Switch for Tightly Coupled Accelerators”, the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART 2017), pp.9:1–9:6, 2017.

[23] Y. Osana and Y. Sakamoto, “Performance Evaluation of a CPU-FPGA Hybrid Cluster Platform Prototype,” the 8th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2017), pp. 22:1–22:6, 2017.

[24] Y. Osana, T. Imahigashi, and A. Tomori, “OpenFC: a portable toolkit for custom FPGA accelerators and clusters,” 2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW), pp. 185–190, 2020.

[25] “Apache Hadoop,” http://hadoop.apache.org/

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” In Proc. 2nd USENIX conference on Hot topics in cloud computing (HotCloud’10). USENIX Association, pp.10-10. June. 2010.

[27] J. Hou, Y. Zhu, L. Kong, Z. Wang, S. Du, S. Song and T. Huang, “A Case Study of Accelerating Apache Spark with FPGA,” 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), pp.855-860, Sep. 2018.

[28] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, “Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale,” In Proc. Seventh ACM Symposium on Cloud Computing (SoCC ’16), pp.456-469, Oct. 2016.

[29] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache Hadoop YARN: yet another resource negotiator,” In Proc. 4th annual Symposium on Cloud Computing (SOCC ’13), pp.5-16, Oct. 2013.

[30] Xilinx Inc., DMA/Bridge Subsystem for PCI Express v4.1 Product Guide v4.1(PG195), Sep. 2020.

[31] Xilinx Inc., 10 Gigabit Ethernet PCS/PMA v6.0 LogiCORE IP Product Guide (PG068), Feb. 2020.

[32] Xilinx Inc., Vivado Design Suite User Guide High-Level Synthesis (UG902), June. 2020.

[33] 尾崎亮, 上嶋明, 小畑正貴, “PC-FPGA 複合クラスタの実現と評価”, 電子情報通信学会論文誌, Vol.J96-D, No.5, pp.1313–1320, 2013.

[34] D. Duolikun, A. Aikebaier, T. Enokido, and M. Takizawa, “A process migration approach to energy-efficient computation in a cluster of servers”, Proc. of the BWCCA-2014, 2014.

[35] 小畑正貴, “FPGA における差動信号入出力を用いた PC クラスタ用ネットワークインタフェース”, 情報処理学会論文誌, Vol.44, SIG06(ACS1), pp.87–95, 2003.

[36] 高野恵輔，上嶋明，尾崎亮，小畑正貴，”汎用 FPGA ボードによる PC-FPGA 複合クラスタシステムの構想，” 信学技報，Vol.116，No.210，pp.39-44，Sep. 2016.

[37] 高野恵輔，上嶋明，尾崎亮，小畑正貴，”PC-FPGA 複合クラスタ上での動的部分再構成による画像処理，” 信学技報，Vol.117，No.46，pp.57-62，May. 2017.

[38] Keisuke Takano, Tetsuya Oda, Ryo Ozaki, Akira Uejima, Masaki Kohata, “PC process migration using FPGAs in ring networks”, IEICE Communications Express, vol.9, no.5, pp.141-145, 2020.

[39] Keisuke Takano, Tetsuya Oda, Ryo Ozaki, Akira Uejima, Masaki Kohata, “Implementation of Distributed Processing Using a PC-FPGA Hybrid System”, Proc. of FPT-2019, pp.387-390, 2019.

[40] M. Owaida, D. Sidler, K. Kara, and G. Alonso, “Centaur: A Framework for Hybrid CPU-FPGA Databases”, 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.211–218, 2017.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets”, In Proc. 2nd USENIX conference on Hot topics in cloud computing (HotCloud’10). USENIX Association, pp.1–7. 2010.

[42] Keisuke Takano, Tetsuya Oda, Ryo Ozaki, Akira Uejima, Masaki Kohata, “Implementation of Process Migration Method for PC-FPGA Hybrid System”, Lecture Notes in Networks and Systems, Vol.159, pp.204–210, 2020.

[43] Xilinx Inc., DMA/Bridge Subsystem for PCI Express v4.1 Product Guide v4.1(PG195), Sep. 2020.

[44] Xilinx Inc., 10 Gigabit Ethernet PCS/PMA v6.0 LogiCORE IP Product Guide (PG068), Feb. 2020.

[45] S. Asano, “Performance Comparison of FPGA, GPU and CPU in Image Processing”, Proc. of FPL-2007, pp. 126-131, 2009.

[46] J. Gomez-Pulido, et. al., “Accelerating Floating-point Fitness Functions in Evolutionary Algorithms: A FPGA-CPU-GPU Performance Comparison”, Genetic Programming and Evolvable Machines, Vol.12, pp. 403-427, 2011.

[47] IEEE Std. 1364-2001, “IEEE Standard for Verilog Hardware Description Language”, IEEE SA, pp. 1-590, 2001.

[48] ModelSim, “Foreign Language Interface”, User Manual, Version 5.6d, [Available Online] 13 Nov., 2019.

[49] N. Kapre and S. Bayliss, “Survey of Domain-specific Languages for FPGA Computing”, Proc, of FPL-2016, pp. 1-12, 2016.

[50] “The Unified Modeling Language”, https://www.uml-diagrams.org/, [Available Online] 13 Nov., 2019.

[51] C. Chuan, et. al., “A Survey of SQL Language”, Journal of Database Management (JDM) Vol. 4, No. 4, pp. 4-16, 2019.

[52] P. Bellows and B. Hutchings, “JHDL - An HDL for Reconfigurable Systems”, Proc. of IEEE FCCM-1998, pp. 175-184, 1998.

[53] PARTHENON HOME PAGE, http://www.parthenon-society.com/archive/NTT/, [Available Online] 15 May., 2021.

[54] S. Jocelyn, B. François, A. Sameer, “Implementing Stream-Processing Applications on FPGAs: A DSL-Based Approach”, Proc. of FPL-2011, pp.130-137, 2011.

[55] N. Matsakis and F. Klock II, “The Rust Language”, ACM SIGAda HILT-2014, Vol. 34, No.3, pp.103-104, 2014.

[56] “The Rust Programming Language”, https://www.rust-lang.org/, [Available Online] 15 May., 2021.

[57] “Verugent”, https://github.com/RuSys/Verugent/, [Available Online] 15 May., 2021.

[58] Keisuke Takano, Tetsuya Oda, Masaki Kohata, “Design of a DSL for Converting Rust Programming Language into RTL”, Lecture Notes on Data Engineering and Communications Technologies, Vol.47, pp.342–350, 2020.

[59] Keisuke Takano, Tetsuya Oda, Masaki Kohata, “Approach of a Coding Conventions for Warning and Suggestion in Transpiler for Rust Convert to RTL”, Proc. of 9th IEEE Global Conference on Consumer Electronics(GCCE) 2020, pp.789–790, 2020.