Vol.2017-UBI-54 No.3
Vol.2017-CDS-19 No.3
2017/5/25

IoT 向け Computation Offloading を用いた 画像処理システムの提案

古都哲生[†1]　峰野博史[†2]

概要：IoT デバイス群で稼働する画像処理システムにおいて，処理プログラムの分割及びデバイス実行配分に関する効率化手法である“IoT 向け Computation Offloading”を提案する．具体的には，高負荷な画像認識アプリケーション・プログラムをパフォーマンスの高くない IoT デバイス群で実行するにあたり，予めアプリケーション・プログラム，デバイス群のパフォーマンス情報を獲得することにより，最適配置を決定，実行し，実行時のログ情報をフィードバックすることで，次回の実行の最適配置を行う．

キーワード：IoT，画像処理，画像認識，組込みシステム，コンピューテーション・オフローディング

Proposal of IoT Image Processing System with Computation Offloading

TETSUO FURUICHI[†1]　HIROSHI MINENO[†2]

Abstract: In the image processing system on the some IoT devices, we propose an efficiency improvement method as "Computation Offloading for IoT" on division of process and allocation of processing on each device. Specifically, in the case of processing a heavy image recognition program with some low-performance embedded system, the program and device performance information is acquired in advance. Based on this information, the optimum arrangement is determined and executes. Log information acquired at the time of execution is used as information for optimal arrangement at the next execution.

Keywords: IoT, Image Processing, Image Recognition, Embedded System, Computation Offloading

1. はじめに

これまでの単独動作の組込み機器に対して，IoT はセンサーをネットワーク経由でクラウド・サーバに接続することで，応用を広げてきている．集められた膨大な量のモノの情報により，新しい価値が生み出され，生活，ビジネス，社会を大きく変えることが予想されている[1]．半導体技術の進歩により，センサーも高性能化し，IoT の高機能，高性能，高信頼性への期待が高まっている．しかしながら，バッテリー長寿命化，低価格化，低消費電力・低価格化を要求される IoT デバイスは，能力・機能不足，遠隔地設置の為の故障発生時の代替え策が無い，等，様々な制約を持っている．これらの制約を回避するために，今まで組込み機器が単体で行っていたデータ処理や記録をできるかぎりクラウド・コンピュータ側に処理させる方法も検討されてきた[2][3]．さらに通信障害回避や通信コスト低減のため，センサーが配置されている現場や，通信する前の段階での処理要求なども高まってきた．

電池駆動の電子機器としては，携帯電話やスマートフォン，タブレットに代表されるスマートデバイスも目まぐるしく発展している．通信インフラやクラウド・コンピュータの発達で，さらに利便性が向上している．スマートデバイスのアプリケーション・プログラムも，単独動作のタイプから，ネットワークやサーバ連携のタイプなど多様化している．しかしながら，アプリケーション・プログラムのさらなる複雑化により，スマートデバイスのハードウェアの能力不足によるユーザーインターフェースの不具合や，高負荷アプリケーション・プログラムの実行に伴う消費電力の増大による，バッテリー保持時間の短時間化などユーザー満足度を低下させる事象が発生してきている．その一つの解決方法として，スマートデバイスの処理自体をクラウド・コンピュータで肩代わりさせる Computation Offloading の技術が注目されてきている．

2. 従来研究と課題

IoT は，エッジ側にあるセンサーデバイスや，処理システム，通信線路と複数の技術の相互連携で成立している．現状の IoT は温度，湿度，気圧，加速度，ジャイロ，地磁気，等の情報を，記録もしくは，有線又は無線の通信手段でサーバに転送し，サーバ側で目的に応じた処理を行うことで，サービスが実現されている．現状はシンプルなセンサーで構成され，電池駆動，低価格化，等の要求要件ため，

†1 イークラウド・コンピューティング／静岡大学 大学院 自然科学系教育部 情報学専攻
e-Cloud Computing&Co. / Graduate School of Science and Technology, Shizuoka University

†2 静岡大学学術院情報学領域／情報学部／総合科学技術大学院 情報学専攻
Graduate School of Science and Technology, Shizuoka University

Vol.2017-UBI-54 No.3
Vol.2017-CDS-19 No.3
2017/5/25

末端のエンドデバイスは，軽量なマイクロ・コントローラで構成されることが多かった．

IoT が使われている現場では，頻繁な仕様追加・修正・変更があり，アプリケーション・プログラムの処理や構造が複雑化してきている．そのため，アップデートによるシステムのパフォーマンス不足障害や，ハードウェア的な故障やソフトウェア的なシステム障害，情報セキュリティ上の問題等が発生した場合，即座に手動での対応，修理，交換ができないことも多く，システム・ダウンを自律的に回避する仕組みも要望されている．

2.1 画像処理ソフトウェア

コンピュータの処理能力の向上とともに，またオープンソースソフトウェアの一つである OpenCV の技術の向上により，提供されている API を使うことで，画像処理・認識アプリケーション・プログラムが容易に作成でき，製品にも組み込まれている．

例えばクルマのナンバープレート認識プログラムの処理フローは図 1 の通りである[4]．まず，処理は大きく，プレート抽出と，ナンバー認識の 2 つに分かれる．プレート抽出は，セグメンテーションと特徴判別，プレート識別の流れで処理する．ナンバー認識は，画像の切り出し，特徴判別，文字認識（OCR）の流れで処理する．プレート抽出のセグメンテーションでは，入力されたカラー画像データをグレースケール化し，ソーベル・フィルタで垂直エッジを検出し，二値化し，クローズ・モルフォロジー処理でイメージ領域を特定する．特徴判別は，単一色領域でのマスクキングを行い，幾何学的形状でナンバープレートの可能性の高い領域を判別する．判別された領域は，機械学習(ML)アルゴリズムの一つである SVM(Support Vector Machine)で，プレート画像を識別する．得られたプレート画像は，ナンバー認識の処理に移行する．すなわち，位置情報を元に画像領域を切り出し，位置，大きさ，縦横比情報から特徴判別し，それぞれの画像領域はニューラルネットワーク(NN)で実装された光学的文字認識（OCR）アルゴリズムで数字，記号の「0～9・」のいずれかの文字認識を行う．最終的にそれぞれの画像領域から認識された文字を並べ，ナンバープレート番号列を得る．

2.2 画像処理ハードウェア

画像処理ソフトウェアは， CPU 能力や大容量のメモリを要求するため，これまで，実行ハードウェアとしては，PC 単体システムが一般的であった[図 2(a)]．近年では，CPU もマルチコア化することで，並列実行による高速処理化が可能となり，また，GPU も汎用利用可能な PC+GPU システムとしてさらに高速化することができるようになってきた[図 2(b)]．また，直接データをクラウド・サーバに上げ，処理するクラウド連携システムも高速化の一手法としてある[図 2(c)]．組込み機器においても，高機能 SoC による組込みシステムで実行されることが多く，もっとも単純な Single Core システム[図 3(a)]や，CPU がマルチコア化し，タスク，スレッド屋プロセスの並列実行を可能にした Multi-Core システム[図 3(b)]，CPU と汎用ハードウェア・エンジンの GPGPU（General Purpose Graphics Processor） や DSP（Digital Signal Processor）を組み合わせて高速処理を行う GPGPU/DSP 連携システム[図 3(c)]，さらに特定用途に特化し高速な処理を行うための専用ハードウェア・エンジンを持った Hardware Engine システム[図 3(d)]と，その目的に応じたハードウェアの選択が可能となってきている．また，エンドデバイス側の CPU 負荷を軽減し，通信手段でクラウド・サーバに上げ処理をする SoC-Cloud 連携システムもある[図 4(a)]．さらに，エンドデバイス側に複数の SoC とクラウド・サーバと連携することで効率を追究したシステムも考えられる[図 4 (b)]．

開発時間，開発コスト，等の理由で，ハードウェアの多くは，汎用処理ができるように設計されている．しかしながら実装されるアプリケーション・プログラム，アルゴリズムの負荷が想定より重くなったとしても，ハードウェア完成後の性能向上は，オーバークロックや電源電圧アップなど，ハードウェア信頼性を損なう方法しかなかった．従って．現行のハードウェアでの性能実現は，ソフトウェアエンジニアのアルゴリズムの改善に頼っていた．

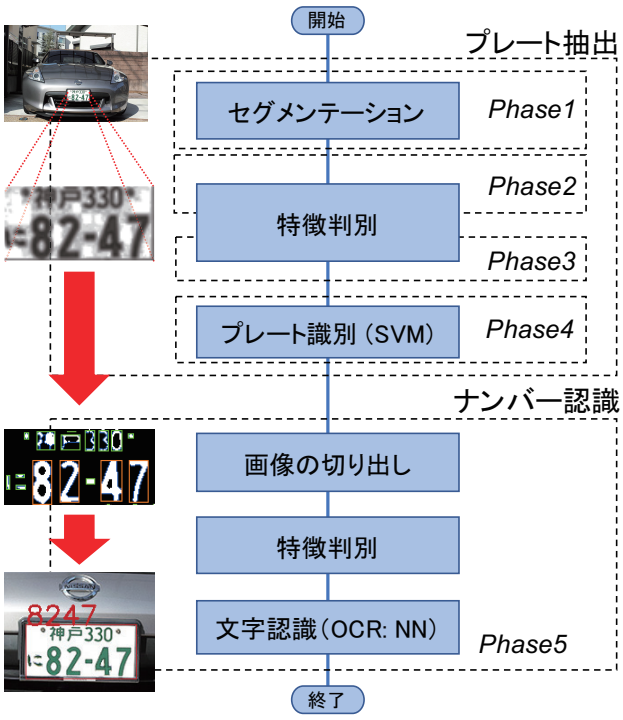
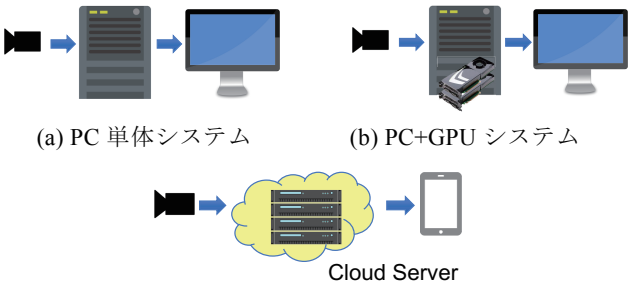


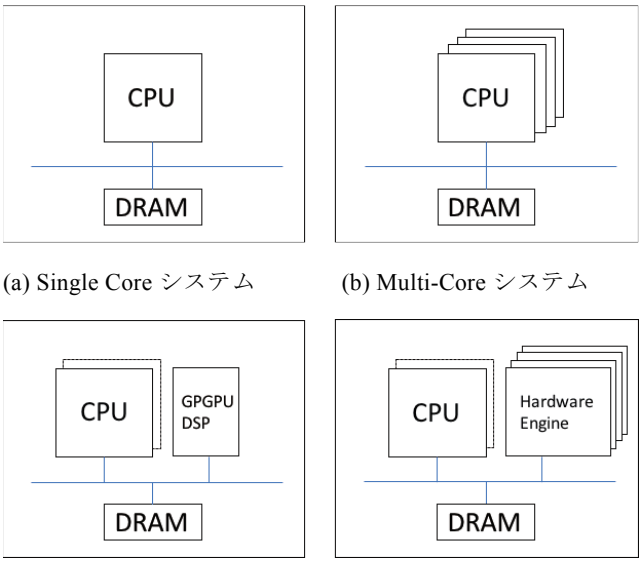
図 1 ナンバープレート認識フロー図



(a) PC 単体システム　(b) PC+GPU システム

(c) クラウド連携システム

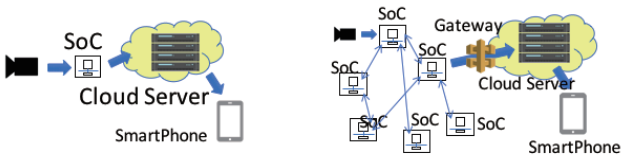
図 2　画像処理システム



(a) Single Core システム　(b) Multi-Core システム

(c) GPGPU/DSP システム　(d) Hardware Engine システム

図 3　SoC の画像処理システム



(a)　SoC – Cloud 連携　(b)　複数 SoC と Cloud 連携

図 4　SoC クラウド連携システム

2.3 クラウド・コンピューティング

近年，インターネットの高速化やサーバシステムの仮想化技術の進化により，これまでモジュール単独で実行してきた処理を，ネットワーク経由のサーバ・サプライヤによるサービスであるクラウド・コンピュータの利用も一般的になってきた．このサービスにより，コンピュータリソースを投資する設備から利用するサービスとして扱い，運用の考え方が変わってきている．また，これまでのデータセンター的なコンピュータ・システムであるプライベート・クラウドだけでなく，スケーラブルな構成が簡単に設定できるパブリック・クラウドもコスト面でメリットがあり，サービスのスタート時のビジネスリスク低減にメリットがある．クラウド・コンピュータのサービスも，マシンのインフラとして提供される IaaS や，あらかじめ OS がインストールされたプラットフォームとして提供される PaaS，アプリケーション・プログラム・レベルの API が提供される SaaS 等がある．

2.4 Computation Offloading について

近年バッテリー駆動のスマートデバイスでの，パフォーマンス不足，メモリ不足，消費電力削減，コスト削減等の目的のために，クラウド・コンピューティングとタスク調整機能使った Computation Offloading が注目されている．手動でアプリケーション・プログラムを Offloading し，基本的なサスペンド／再開機能を実装した MAUI[5]や,スレッド細分化を実現しプログラムの自動パーティショニング機能を持った CloneCloud[6]，クラウド・コンピュータの仮想マシン・インスタンス確保から対応し，経済的なインスタンス生成を行う COSMOS[7]など，Offloading のための効率の良い仕組みを実装したフレームワークが多く提案されている[8]．現状の Computation Offloading は，モバイル・アプリケーション・プログラムの一部分を，クラウド・コンピュータで自動的に分散処理させることで，スマートデバイス自身の CPU 単体では負荷が大きかった処理を行うことが可能になる．既存のプログラムを分散化させるために，仮想マシンレベルでの処理分散を考慮しているのが特徴的である．多くの場合，処理プロファイル情報を元に，分散部分の分離，分散方式を決めている．

2.5 本研究が解決する課題

スマートデバイスの Computation Offloading は，スマートデバイスと，さらに高性能な CPU 群であるクラウド・コンピュータを利用することで，ユーザーインターフェースと処理が実現されている． IoT システムにおいては，センサー，通信，処理，記録と機能的に明確に分割されて構成されている場合が多く，負荷のかかる処理が必要な場合は，現場で高価な高性能ハードウェアを利用することが多かった．充分なハードウェアを用意できない場合は，能力の低いハードウェアで，長時間演算や近似解の算出等の暫定的な方法の選択肢しかなく，本来の目標精度，スペックを満たせなくなる．さらに，ネットワークに常時接続できず，直接クラウド・サーバをアクセスできない場合もあり，通信が不可欠なスマートデバイス向け Computation Offloading システムの適用も難しい場合が多かった．

3. 提案システム

我々は，これまで予めインストールされた固定プログラムで稼働していた IoT デバイスを，外部システムから自由に制御・実行できる環境を提案する．本環境では，今後高度化が期待される画像情報を扱う次世代 IoT デバイスに特化したフレームワークを想定している．

3.1 着目点

スマートデバイスが，フューチャーフォンからスマート

フォンに変わってきているのと同様にIoTデバイスも現状のOSのないシステムから，RTOS（Real Time OS）ベースのシステムやLinux kernelを持った高機能・高性能なシステムが一般的になることは容易に推測できる．半導体プロセスの進化により，ハードウェアのコストや消費電力は低下するとともに，スマートフォンと同様にマルチコアのCPUも使えるようになってくると考えられる．ソフトウェアの実装においては，情報セキュリティの脆弱性対策やプログラミング容易性の観点で，組込みシステムで一般的であったC/C++言語環境から，Python, Ruby, Lua, JavaScript, 等のスクリプト言語が注目されてきており，組込み機器向けの実装もされている．また，現状のIoTデバイスは，個々のデバイスのパフォーマンスは高くないものの，数多くのセンサー設置が期待できるため，それらをIoTデバイス群として，総合的に潤沢なプロセッシング能力を持つ環境が実現できると考えられる．

そこで我々はスマートデバイスとクラウド・コンピュータで実現したComputation Offloadingの手法の一部を，IoTデバイス群を対象としたIoT向けComputation Offloadingのフレームワークに適用し，アプリケーション・プログラム，実行デバイスのモデル化，実行時間算定方法，実行環境，最適実行アルゴリズムを検討した．

3.2 IoT向けComputation Offloadingについて

スマートデバイスのComputation Offloadingの主目的は，スマートデバイスでの負荷の重いアプリケーション・プログラムを実行した際の，消費電力の増大に伴うバッテリー寿命の低下や，スマートデバイス自身のパフォーマンス不足を解消するため，パフォーマンスが高く，充分な容量，帯域，速度を持つ通信と，スケーラブルなクラウド・コンピュータで補うことにある．

本研究では，IoTデバイスが設置されている現場での処理を効率的に行うための方法として，今後一般的になると考えられるSoCで構成された複数のIoTデバイスを想定し，ゲートウェイ経由又は，直接インターネットに接続し，クラウド・サーバにデータを転送するシステムにターゲットを絞った．これまでIoTセンサーデバイス単体に要求される機能や性能を，IoTセンサーデバイスを含めた，IoTデバイス群にComputation Offloadingする機構に焦点を定めた．

スマートデバイスのComputation Offloadingは，対象となるアプリケーション・プログラムの全て，または一部を，クラウド・サーバ側のインスタンスに受け渡し，処理するが，我々のIoT向けComputation Offloadingは一連の処理を，同系のセンサーデバイスやアクセラレータとして利用できるIoTデバイス群で分散処理を行い，IoTデバイス同士が独立してデータ転送を行うことで実現することに大きな違いがある．さらに，今回のOffloading対象のアプリケーション・プログラムの分割粒度を細かくすることで，スマートデバイスのCPUより非力なIoTデバイスのCPUでの実行を可能にした．

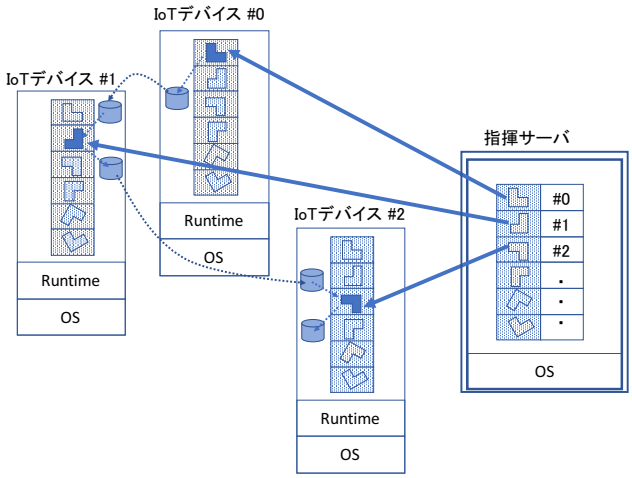


図 5　IoT向けComputation Offloading 実行モデル

図 5にIoT向けComputation Offloadingの実行モデルを示す．ハードウェアとして，1台の指揮サーバと複数個のIoTデバイスを考える．指揮サーバはサーバ・レベルのパフォーマンスを想定しているが，IoTデバイスは，中から小規模のLinuxシステムを想定している．IoTデバイスの負荷を軽くするために，本モデルでは，IoTデバイス側の機能としては，センシング又はプログラムの実行とデータ転送しか割り付けていなく，スケジューリングやIoTデバイス同士の調停は行わない．アプリケーション・プログラム自体の処理の制御の主体は指揮サーバが持っており，アプリケーション・プログラムは分散処理される単位である構成エレメントに分けられた状態でシナリオとして指揮サーバに格納され,随時解釈される．なお，構成エレメントは事前もしくは随時，指揮サーバからIoTデバイスに転送されるものとする．指揮サーバは，各構成エレメントをどのIoTデバイスで実行するかのスケジューリング情報も持っており，指揮サーバがシナリオを実行・解釈すると，あらかじめスケジューリングされたIoTデバイスに，構成エレメントの起動指示が随時送られる．起動を指示されたIoTデバイスは独立して，起動後の結果を別のIoTデバイスに直接転送を行うため，指揮サーバの負荷を軽くすることができる．

3.3 IoT Computation Offloading処理フロー

IoT Computation Offloadingフレームワークでは，処理フローを，①前処理，②実行前，③実行時，④実行後，の4段階で考える．

(1) 前処理

本処理を行う前の初期設定処理である．対象となるアプリケーション・プログラムを分散処理できる形に変換する．さらに，各IoTデバイスのパフォーマンス情報（IoTデバイス単体，IoTデバイス間転送速度），アプリケーション・

プログラム負荷情報を取得し，各 IoT デバイスに対しアプリケーション・プログラムを事前に配送し，実行可能状態にしておく．

(2) **実行前**

あらかじめ取得している各 IoT デバイスのパフォーマンス情報，アプリケーション・プログラム負荷情報を元に分散実行経路を探索し静的にスケジューリングする．

(3) **実行時**

予め静的スケジューリングした経路に従って随時処理が実行される．実行時には実行時間やデータ転送時間がプロファイル情報として格納される．

(4) **実行後**

実行時に取得したプロファイル情報から，IoT デバイスのパフォーマンス情報やアプリケーション・プログラム負荷情報を再抽出し，次回の最適分散経路探索のための情報として更新する．

3.4 実行時間モデル＆実行配置アルゴリズム検討

アプリケーション・プログラムを，複数個の IoT デバイスで実行する際の負荷粒度を合わせた機能単位のブロックを，構成エレメントとして定義する．構成エレメントを効率良く配分実行させるために，IoT デバイス実行時間モデルを作り，そのモデルに基づいてシステム総実行時間を算出する方法を検討した．

3.5 IoT デバイス実行時間モデル

IoT デバイスの実行時間モデルは，IoT デバイス自身のパフォーマンスと実行されるアプリケーション・プログラムの構成エレメントの負荷情報，各 IoT デバイス間の転送速度から構成される．

(1) **初期パフォーマンス・パラメータ値の取得**

あらかじめそれぞれの IoT デバイスで，それぞれのアプリケーション・プログラムの構成エレメントを実行させ，実行時間を測定し，各 IoT デバイスのパフォーマンス情報，構成エレメント負荷情報の初期パラメータ値として設定する．

(2) **IoT デバイス実行時間のモデル**

アプリケーション・プログラムの構成エレメントが並列動作可能な形式で記述されている場合は，複数の IoT デバイスで実行させることが可能である．今回検討している環境は，複数個の IoT デバイスでアプリケーション・プログラム構成エレメントを実行させることができるようになっている．まず，単一 IoT デバイスで実行する場合を考える．アプリケーション･プログラムの構成エレメント i の，指定 IoT デバイス j での実行時間を DI_{ij} とすると，構成エレメント i の CPU の負荷度合いを AE_i，IoT デバイス j の演算能力を DP_j より次式で求めることができる．

$$DI_{ij} = AE_i \times DP_j \quad \cdots \quad (1)$$

対象となる IoT デバイス j と次 IoT デバイス k へのデータ転送時間 DN_{jk} は，構成エレメント i から次構成エレメントへの転送データ量を S_i，IoT デバイス j から IoT デバイス k への IoT デバイス間転送速度を TL_{jk} とすると，次式のとおりとなる．

$$DN_{jk} = \frac{S_i}{TL_{jk}} \quad \cdots \quad (2)$$

構成エレメントにおける次構成エレメントへのデータ転送を含めた実行時間 DE_{ijk} は，次のとおりとなる．

$$DE_{ijk} = DI_{ij} + DN_{jk} = AE_i \times DP_j + \frac{S_i}{TL_{jk}} \quad \cdots \quad (3)$$

複数 IoT デバイスで実行する場合は，上記 DE_{ijk} の経路が複数個存在するため，その最大時間が，該当構成エレメントの実行時間となる．単一経路の *DEijk* 値を *DE0* とし，複数経路の *DEijk* 値を $DE0, DE1 \ldots DEn$ とすると，該当経路の実行時間 *DEmax* は，（4）式となる．

$$DEmax = \max(DE0, DE1, \ldots DEn) \quad \cdots \quad (4)$$

3.6 システム総実行時間

システム総実行時間は，アプリケーション・プログラム構成エレメントそれぞれの処理時間の総数となる．総実行時間 *TD* は，アプリケーション・プログラム構成エレメントが 1 から m まであるとすると，次式で表すことができる．

$$TD = \sum_{n=1}^{m} DEmax_n \quad \cdots (5)$$

3.7 最適実行配置アルゴリズム

アプリケーション・プログラムのそれぞれの構成エレメントに対して，実行可能な IoT デバイスが複数個あると，経路可能性が増え，すべての構成エレメントでは相当数の経路が存在することになる．実行配置のために効率的な配置を見つける必要があり，その探索方法としては，①全経路探索，②特定条件での経路探索（処理時間，転送時間），が考えられる．

4. プロトタイプ実装と評価

画像処理向け IoT Computation Offloading 実現の為の基礎評価として，画像処理の一つであるナンバープレート認識プログラムを IoT デバイス群で処理をする場面を想定し，プロトタイプ実装及び評価を行った．

4.1 プロトタイプの実装について

IoT Computation Offloading フレームワークをプロトタイプとして構成するために，全体の実行制御を行う指揮サーバと実際に処理を行う IoT デバイスを二種類のハードウェアに実装した．指揮サーバはクラウド・コンピューティン

グ・サービスの一つである VPS（Virtual Private Server）に実装し，IoT デバイスとしては，有線 LAN もしくは無線 LAN で接続された複数種類の Raspberry Pi モジュールを用いた[図 6]. また，すべてのサーバ，モジュールは Linux 上で稼働する Python 言語でプロトタイプ・プログラムを記述した.

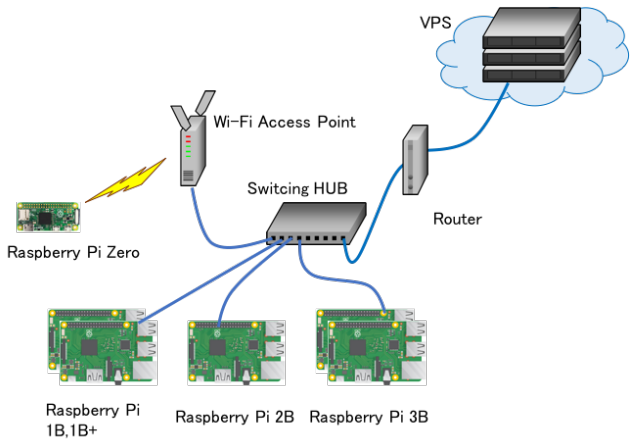


図 6　プロトタイプ構成図

(1) 画像アプリケーション・プログラム

今回のナンバープレート認識プログラムは C++言語で実装されていたが，Python3 で稼働する OpenCV3.1 環境のスクリプトとして再実装した．その際に，複数個の IoT デバイスで並列分散実行のために，Phase 別に構成エレメントとして分け，関数として記述した.

Phase1 はプレート識別のセグメンテーションの処理で，今回の評価ではあらかじめカメラで取り込んだ画像データを JPEG フォーマットのファイルとして読み込み，フィルタリング処理を行う．Phase2 は，プレート認識の特徴判別の前半の処理である．Phase3 はプレート認識の特徴判別の後半の処理で，プレート識別を行う．Phase3 の処理は最も処理負荷が重く，並列に実行することで処理時間を短縮することができる部分である．今回は最大 5 並列で処理ができるようにした．Phase4 は識別されたプレート画像を SVM でナンバープレートかどうかを識別する．Phase5 はプレート画像から数字領域画像を切り出し，それぞれの数字領域画像データを NN（OCR）で文字認識することでナンバープレート番号を得る[図 1].

(2) クライアント・ランタイム・スクリプト

各 IoT デバイスで実行されるランタイム・プログラムであるクライアント・ランタイム・スクリプトは，起動されると，指揮サーバに接続を試みる．指揮サーバと接続が成立すると，指揮サーバからのコマンド待ち状態となる．指揮サーバからは，1 行の Python スクリプトが送られ，クライアント・ランタイム・スクリプトは送られてきたスクリプトをそのまま実行する．また，クライアント・ランタイム・スクリプトは，実行スクリプトの変数を含めたオブジェクトを IoT デバイス間で直接転送する API も持っており，指揮サーバの指示に基づいて，送信側の IoT デバイスから受信側の IoT デバイスに直接転送を行う.

(3) 指揮サーバ・スクリプト

指揮サーバは，各 IoT デバイスで実行されるクライアント・ランタイム・スクリプトと通信し，アプリケーション・プログラムから変換された Offload スクリプトに従って，各 IoT デバイスにコマンドを送り出す．現在は，各 IoT デバイス間のデータ転送はあらかじめ Offload スクリプトに記述する必要はあるが，今後のバージョンでは自動的に変数オブジェクトの転送を発生できるようにする予定である.

(4) IoT デバイス群構成

画像アプリケーション・プログラムを実行する環境は，IoT デバイス群であるが，予めクライアント・ランタイム・スクリプトが稼動している.

(5) 経路探索プログラム

今回は構成デバイスが最大 6 個と比較的少ない個数での探索であったため，全経路探索を採用した．それぞれの IoT デバイス単体で測定した IoT デバイスのパフォーマンス情報やアプリケーション・プログラム負荷情報を元に，構成エレメントをそれぞれ全 IoT デバイスで実行する経路での実行予想時間を求め，最短実行時間経路を選び出すプログラムを作成した．並列実行できる構成エレメント Phase3 の経路部分に関しては，予め経路の最適解は求めて，探索経路の実時間での絞り込みをすることで，全体経路における組み合わせ数の激増化を押さえた.

4.2 画像 IoT 向けフレームワーク

画像アプリケーション・プログラムは，これまで PC 上のオープンソースの OpenCV もしくはそれに準ずる API を使った開発が主流である．PC レベルのパフォーマンスゆえに，多くの場合 1 本のプログラムで書かれる場合が多く，複数のハードウェア機器に対応する画像アプリケーション・プログラムを作ることは，新規に新たなプログラムを開発する事と同様の労力が必要で，現実的でなかった．我々は，Python で稼働する OpenCV アプリケーション・プログラムを利用し，画像アプリケーション・プログラム，クライアント・ランタイム・プログラム，指揮サーバ・プログラムの 3 項目を決め，環境を構築した[図 7]. IoT デバイスは，Linux ベースの OS 環境で稼動する Python スクリプトであるランタイムスクリプトが稼動する．また，指揮サーバは，Linux 環境下の Python スクリプトで稼動する，Offloading スクリプトのインタープリタが実行し，IoT デバイス群と通信する．IoT デバイス群は，必要に応じて，それぞれの IoT デバイスのファイルシステム上にあるライブラリ・スクリプトを読み込むことで，新しい機能を持ったルーチンを実行することが可能である.

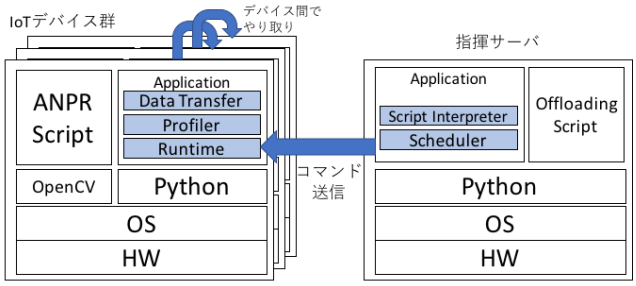


図 7　画像 IoT Computation Offloading フレームワーク

4.3 評価内容及び結果

(1) パフォーマンス評価(個別・転送)

あらかじめ今回利用する IoT デバイスのパフォーマンスを測定し，基準となる IoT デバイスを元に，今回の対象となる画像認識プログラムの実行時間より，各 Phase の処理の負荷情報を求める．基準 IoT デバイスとして，Raspberry Pi 1B を設定し，各 IoT デバイスの相対パフォーマンス値を実測した．また，各 IoT デバイス間において，数種類のデータサイズのデータの転送を実際に行い，その転送時間を測定する．今回の評価は 6 個の IoT デバイスなので，1 つのサイズにおいて ${}_6P_2$(=15 通り)の組み合わせがある．図 8 に IoT デバイス単体パフォーマンスとそれぞれの通信速度の実測値を示す．円で書かれたノードは，今回の IoT デバイスであり，円内数字は Raspberry Pi 1B を 1 としたときのパフォーマンス比率を表し，円の大きさでパフォーマンスを表している．また，各 IoT デバイス間の線は太さで転送速度比率を表している．IoT デバイスのデバイスドライバ負荷や，アクセスポイント，WPA の暗号化処理等のオーバーヘッドにより，無線接続は有線接続に比べて約 1/6 の伝送速度となっている．

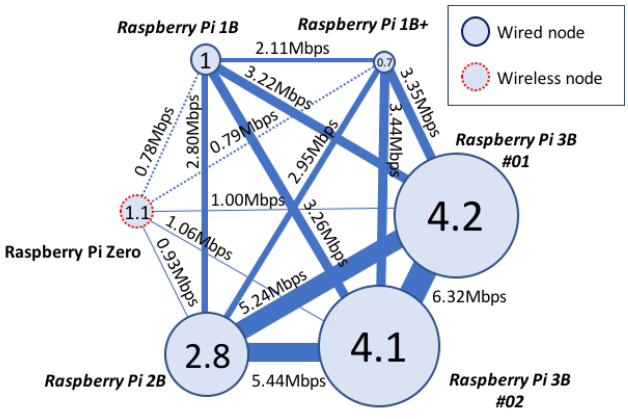


図 8　IoT デバイス単体・転送パフォーマンス結果

(2) 構成エレメントの重み付け値

今回基準とした Raspberry Pi 1B モジュールでの実行時間を元に，Phase1～5 までの構成エレメントのプログラム負荷値を設定した．また，各構成エレメント間のデータ転送サイズの目安値も設定した[図 9].

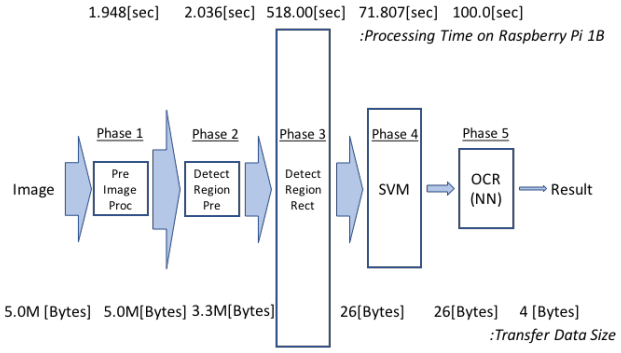


図 9　構成エレメントの重み付け値

(3) 全経路探索

今回は並列実行可能部分の候補として最速経路から 6 通りのパターンを選び，全体経路を探索し，それぞれの予想実行時間を算出した[図 10]. 四角内の数字は割り付けられた Raspberry Pi の型番号で，1 は Raspberry Pi 1B, 2 は Raspberry Pi 2B，3 は Raspberry Pi 3B となっており，数字の値が高いほどパフォーマンスが高いモジュールである．

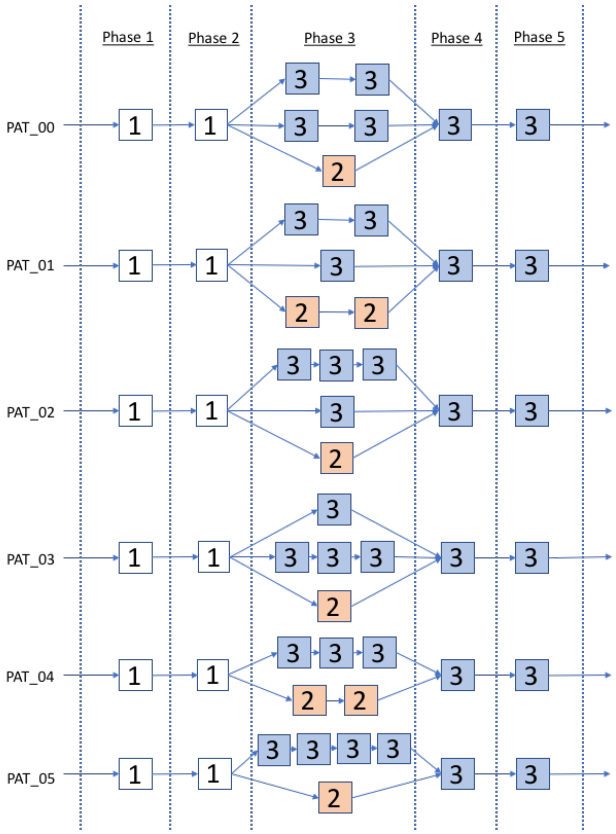


図 10　全探索結果図

(4) 実機評価

全経路探索で求められた経路を実機システムで実行するスクリプトを手動で書き，実際に実行させ実行時間を計測し，全経路探索時に計算された予想実行時間と比較した[図 11].

実行時間に最も影響があるフェーズが 5 並列実行可能な Phase3 の構成エレメントである．全経路探索での最速経路

は，図 10 の PAT_00 で，最もパフォーマンスの高い 2 台の Raspberry Pi 3B に，Phase 3 の 2 タスクがそれぞれ割り付けられ，残りの 1 タスクは次のパフォーマンス順位の Raspberry Pi 2B に割り付けられている．全経路探索時の予想総実行時間は 94.28 秒で，実機での総実行時間は 105.70 秒となり実機比-10.8%の誤差となった． 全経路探索での二番目の経路は，図 10 の PAT_01 で，Phase 3 の 2 タスクが Raspberry Pi 3B と，Raspberry Pi 2B に割り付けられ，残りの 1 タスクは，Raspberry Pi 3B に割り付けられている．全経路探索時の予想総時間は，117.36 秒で，実機総実行時間は 123.81 秒となり，実機比 -5.21%となった．最も誤差が大きかったのが図 10 の PAT_04 で，Phase 3 の 3 タスクが Raspberry Pi 3B に，2 タスクが Raspberry Pi 2B に割り付けられ，全経路探索時の予想総時間は，117.63 秒であったのにかかわらず，実機総実行時間は 154.70 秒となり，実機比-23.96%となった．

いずれの結果も，各フェーズの結果は誤差が不均一で，特に Phase5 の実行時間のモデル結果は大きな値となり，少なめに見積もられたモデル結果の底上げをしている傾向もある．この点において，モデル及びモデルパラメータの再調整が必要と考える．

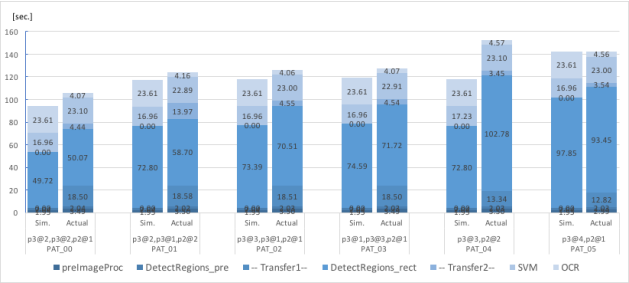


図 11 最適経路上位 6 位の予想・実績実行時間

5. おわりに

並列処理部分があるプログラムにおいて，並列処理制御機構があれば，低消費電力な能力の低いデバイスでも数を増やすことで，パフォーマンスを向上させることができることができることがわかった．

また，今回作り上げたモデルをベースに経路探索し，実行時間予測した値と，実機で実測した値については，経路順位としてはほぼ一致する傾向が認められ，画像認識アプリケーション・プログラムとしての IoT システムにおける Computation Offloading の機能を実現するスケジューラ向けモデリングとしては充分使用できるものであると結論付けた．

並列化されている処理は，全体処理時間短縮に貢献しており，アプリケーション・プログラムの自体の並列度合改善や，接続 IoT デバイスの追加で PC と同等レベルの処理速度まで改善できるものと思われる．さらに，実機システムにおいて，一つの IoT デバイスが何らかの故障，障害で使用不可となっても，残りの IoT デバイスを再配置することで，性能は低下するが，機能を実現できることも確認できた．また，通信においては通信速度だけの情報でモデル化を行ったが，転送サイズに大きな差がある場合は，レーテンシー時間が大きく係わる傾向があり，今後のモデル化に追加していきたい．

今回の評価においては，IoT デバイスのパフォーマンスを 1 つのアプリケーション・プログラムでの実行時間だけで評価し，モデリングを行ったが，現実には IoT デバイスに実装されている CPU の世代，クロック周波数，コア数，等，パフォーマンスに係わる条件があり，実機動作においては，アプリケーション・プログラムの処理だけでなく通信速度にも大きく影響を及ぼすことがわかった．そのため，今後は実機に近い値に近づけるため IoT デバイス自体の処理時間モデルを改良し，実行するアプリケーション・プログラムの構成モジュールの並列性の確保とそのやり取りデータの粒度を合わせることで，IoT 向け Computation Offloading 効果をさらに高める改善も進めていく予定である．

参考文献

[1] Joseph Bradley, Jeff Loucks, James Macaulay, Andy Noronha , Inc.: “Internet of Everything (IoE) Value Index (White Paper)” , Cisco Systems, 2012.
[2] Tetsuo Furuichi: “Lightweight image sensor node for next generation IoT”, IEEE 4th Global Conference on Consumer Electronics (GCCE), pp. 36-37, 2015.
[3] Tetsuo Furuichi, Hiroshi Mineno: "Distributed Data Processing Method for Next Generation IoT System", IEEE 5th Global Conference on Consumer Electronics (GCCE), pp. 357-358, 2016.
[4] Daniel Lélis Baggio, Shervin Emami, David Millán Escrivá, Khvedchenia Ievgen, Naureen Mahmood, Jason Saragih, Roy Shilkrot: “Mastering OpenCV with Practical Computer Vision Projects” , PACKT PUBLISHING, pp. 161-188, 2012.
[5] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman,S. Saroiu, R. Chandra, and P. Bahl: “MAUI: Making Smartphones Last Longer with Code Offload”, In ACM MobiSys, 2010.
[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti: "CloneCloud: elastic execution between mobile device and cloud, in: Proceedings of the Sixth Conference on Computer Systems", ACM, pp. 301–314, 2011.
[7] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, Ellen Zegura: "COSMOS: Computation Offloading as a Service for Mobile Devices", MobiHoc '14 Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing, pp. 287-296, 2014.
[8] Khadija Akherfia, Micheal Gerndta, Hamid Harroudb: "Mobile cloud computing for computation offloading: Issues and challenges", Production and hosting by Elsevier, 2016.