低消費電力プロセッサのための限定的動的再構成アーキテクチャの提案

平尾岳志[†] 安達琢[†]
浅井哲也[†] 本村真人[†]

高性能・高エネルギー効率のプロセッサを実現するアプローチとして、対象プログラムのホットパスを可変構造のデータパスにマッピングしアクセラレートするリコンフィギュラブルプロセッサが注目されている。本稿では、処理の内容が多岐にわたり、かつ低消費電力性が強く求められる組込み用途をターゲットとし、Control-Flow Driven Data-Flow Switching (CDDS) 可変データパスアーキテクチャを提案する。このアーキテクチャは、(1) 動的再構成を必要最小限な範囲に限定することで、柔軟性と低消費電力性の両立を目指す、(2) 既存命令列をそのままデータパスにマッピングすることで、既存アーキテクチャからスムーズに移行可能なリコンフィギュラブルプロセッサを目指す、の 2 点をその特徴とする。予備的な評価として、小規模なプログラムを手動マッピングし、その基本的特性を調査した。

A Restricted Dynamically Reconfigurable Architecture for Low Power Processors

Takeshi Hirao,[†] Taku Adachi,[†] Tetsuya Asai[†]
and Masato Motomura[†]

Reconfigurable Processor (RP) has attracted wide attention as an approach to realize high-performance and highly energy-efficient processors by mapping target program's hotpath to a reconfigurable data path. In this paper, we propose Control-Flow Driven Data-Flow Switching(CDDS) variable data path architecture for embedded applications that demand extremely low power consumption and wide-ranging of usage. This Architecture is characterized by following two features. (1)Aiming to achieve both flexibility and low power consumption by limiting the scope of dynamic reconfiguration, (2)Aiming to smooth migration from the existing architecture by mapping the existing instruction sequence to the data path. As a preliminary evaluation, we have manually mapped small programs to understand fundamental characteristics of the proposed architecture.

1. はじめに

近年、センサーネットワーク、モバイル用プロセッサに低消費電力な組み込みプロセッサが求められている。既存のプロセッサは消費電力の問題により、性能を上げることが難しくなっており、性能電力比が高いプロセッサが求められている。今までにプロセッサのコア数を増やし、並列に実行することで性能を上げる、マルチプロセッサが提案された。しかし、従来の構造を引きずっているために、性能電力比の大幅な向上は見込めない。汎用プロセッサの消費電力内訳の一例[1] より、(1) 命令読み込み、(2) 制御、(3) レジスタアクセス、(4) パイプラインレジスタ、(5)Arithmetic Logic Unit(ALU) で全体電力の 2/3 以上を消費している。ALU の電力を演算に必要な電力と考えると、その他の電力は演算をプロセッサで実行するために必要な電力であり、演算が目的であるならば削減可能な部分である。そこで、プログラムをハードウェア実行することで、従来のプロセッサにあった削減可能な電力を削減し、並列実行により性能を上げることで性能電力比向上を達成するプロセッサが提案されている。

コンフィギャラブルプロセッサはプログラム内のホットバス部を Hardware(HW) 化し、プログラム実行時に HW で処理を行う。プログラムを HW 化して実行するため、汎用プロセッサで実行するよりも大幅な性能電力比の向上を見込める。しかし、HW 化しているために、汎用性がないという問題がある。そこで、Reconfigurable Hardware(RH) を持つ、Reconfigurable Processor(RP) が提案された。RP は動作前に RH を再構成することが可能で、汎用性を持ちつつ、性能電力比の向上を狙う。しかし、RH を多数の

† 北海道大学
Hokkaido University

Processing Element(PE) で実装しても、大規模なプログラムを一度にマッピングすることはできない。この問題を解決するのが Dynamically Reconfigurable Processor(DRP) である。DRP は、アプリケーションを時分割し、RH を実行時に再構成することで大規模なプログラムを実行することができる。しかし、再構成するときに電力が必要となり、再構成を頻繁に行う場合には多くの電力が必要になるという問題がある。この問題に着目して、加速対象とする部分を限定し、動的再構成しないことで低消費電力化を達成する、CMA[4] が提案された。

本研究では広範囲なアプリケーションに対して性能電力比の向上を目指した Control-Flow Driven Data-Flow Switching(CDDS) 可変データパスアーキテクチャを提案する。はじめに、CDDS では、RH 実行に柔軟性を持たせつつも、可能な限り動的再構成の範囲を減らすことを目指す。コントロールフローの分岐点で、一部のデータパスのみを切り替えることで、従来の DRP に比べ、再構成を減らすことができ、低消費電力化を見込む。

本論文の構成は以下の通りである。2 章にこれまで提案された性能電力比の高いプロセッサの説明をし、利点と問題点を述べる。3 章で提案する CDDS アーキテクチャの概要を説明する。4 章で CDDS アーキテクチャの詳細な設計を説明する。5 章で複数のプログラムを用いた机上評価の結果について述べる。6 章で総括する。

2. 関 連 研 究

本章で性能電力比の高い、組み込み用途向けプロセッサのこれまでの研究を説明し、提案する CDDS との違いを示す。

Green Droid[2] はモバイル用途に向けたコンフィギャラブルプロセッサである。Android OS のホットパスを HW 化したものを多数用意し、その部分の実行を HW に任すことで、性能電力比の向上を狙った。この HW は一部変更可能だが、大きな変更はできず、Android OS 専用プロセッサとなっている。

ADRES[3] は VLIW プロセッサに FU をアレイ状に並べて構成した RH が密結合している DRP である。ホットパスのループ部分を VLIW と RH で同時に実行することで大幅に性能を上げることができる。しかし、単純なループ以外の制御部分は VLIW で実行するため、複数の分岐を含むプログラムは RH で実行することはできない。

CMA[4] は PE アレイを組みあわせ回路で構成した粗粒度の RP である。従来の Dynamically Reconfigurable Processor である Muccra[5], DRP[6] は動的再構成に多くの電力が必要になる。そこで、RH を動的再構成しないことで、再構成時の電力を削減する。また、レジスタを介さず、PE をスイッチで接続し、組み合わせで実行することで電力削減を達成する。しかし、RH では組み合わせ回路のみの実行となり、ループ実行などでデータをレジスタに保持する必要がある場合は、別途マイクロコントローラを使用することが必要となる。また、動的再構成をしないために分岐に対応できず、柔軟性が低くなり、大規模なプログラムの実行は難しいと思われる。

上述の研究と比較して、CDDS は柔軟性を持ちつつ、可能な限り動的再構成を減らすことで、より広範囲なプログラムに対して性能電力比の向上を目指す。はじめに、CDDS はデータパスを動的再構成する可変部と動的再構成しない固定部に分け、分岐命令の結果により、可変部のみ再構成する。そのため、ADRES とは異なり、制御命令実行後に RH 全体の再構成せず、より低消費電力化を狙う。また、CDDS は、RH 上で分岐を多く含むホットパスも後述のコンテキスト数が許す限り実行可能である。よって、CMA とは違い、コンテキストを多数用意すれば、2 重ループなどの複雑な分岐にも柔軟に対応できる。

また、上述の研究では構成情報を生成するためにプログラムを変更、または新たに記述する必要があった。我々は構成情報をバイナリファイルから生成することにより、過去のプログラム資産を引き継ぐことができる。また、バイナリファイルを使うことで、制御情報を生成するのに、小規模な変換ツールで済むという利点がある。

3. CDDS アーキテクチャ概要

本章では我々の提案する CDDS アーキテクチャの概要について述べる。また、構成情報生成手法について説明する。

3.1 Control-Flow Driven Data-Flow Switching

一般的にプログラムをコントロール/データフローグラフで表したときに、コントロールフローの分岐点で、次に実行するデータフローが選択される。このとき、実際に変更されるデータフローは実行中のデータフローから、選択したデータフローへの接続部分のみである。よって、プログラムのデータフローを RH にマッピングしたとき、コントロールフローの分岐点では接続部分を構成している部分のみ再構成すればよ

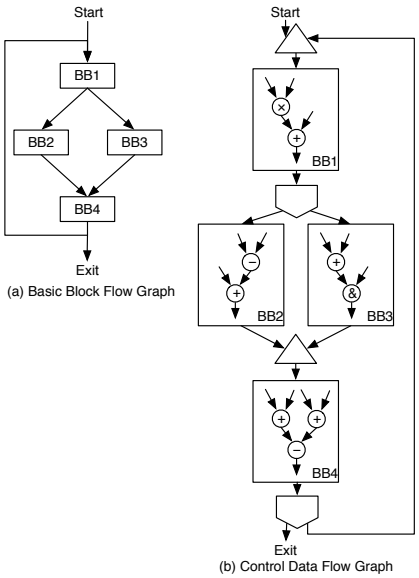

図 1 基本ブロックの流れ
Fig. 1 Basic block flow

い。我々はこの観点より、データパスを可変部と固定部に分割し、実行時に可変部のみ再構成することを考えた。これにより、全体を再構成するのに比べ、可変部のみで済むために、動的再構成する部分を減らすことができる。また、固定部は切り替えないために組み合わせ的に実行することができる。そこで、既存の命令列を固定部に配置し、組み合わせ的に実行する。このように動的再構成を必要最小限な範囲に限定することで低消費電力化を狙う CDDS を提案する。

一例として図 1(a) にプログラムの Basic Block(BB) の流れを示す。BB1 から実行し、BB2 または BB3 を実行し、BB4 を実行する。そして再度、BB1 から実行するか完了する。この BB の流れをコントロール/データフローグラフで表したものを図 1(b) に示す。BB 内のデータフローには分岐がなく、動作中にデータの流れが変わることはない。一方、BB 間では分岐の結果によって、データの流れが変わる。この例の場合では図 2 に示すように BB1 が r2 という値を必要としているときに、はじめは入力から値を得る (図 2a)。次に BB2 または BB3 を実行後、BB4 を実行し、r2 の値を書き換える。ループして、BB1 を実行するときには r2 の値は書き換わっているために、入力から BB1 へのデータの流れから BB4 から BB1(図 2b) のへのデータの流れ (点線の部分) へ変更する必要がある。 よって、すべての BB 内のデータフロー (二重□の部分) を RH 上に構成すれば、BB 内のデータフローを構成する部分は切り替える必要はなく、BB 間のデータフローを構成する部分のみ切り替えればよい。そこで図 3 に示すように BB 内のデータフローを構成する固定部と BB 間のデータフローを構成する可変部に分ける。はじめに、全 BB 内のデータフローの構成情報を一つ保持する。この構成情報により、RH の Switching Element(sw) と Processing Element(PE) で実装した固定部上に BB 内のデータフローを構成する。次に、BB 間でデータを受け渡すために必要なデータフローの構成情報を表 1 に示すようにコンテキストとして複数用意する。固定部には BB 内のデータフローが構成されており、可変部はコンテキストに従い、BB にデータを供給する。例えばコンテキスト 2 の場合、可変部は BB1 から BB2 へ、BB2 から BB4 へデータを供給する。プログラム実行はコントロールフローの分岐点でコンテキストを切り替え、可変部を再構成することで進める。 可変部には演算器はないため、コンテキストに演算情報は含まない。そのため、DRP のように演算情報を含むコンテキストよりもサイズを小さくすることができる。

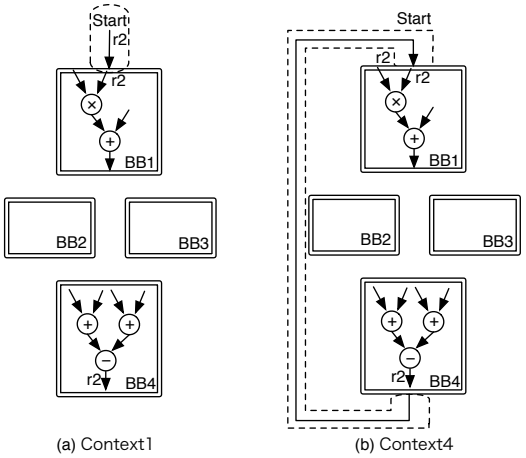

図 2 基本ブロックの流れの切り替え
Fig. 2 Switch over basic block flow

プログラムから、コンテキストを生成するためには、すべてのコントロールの入り口から求める。これは、

表 1 コンテキスト
Table 1 Contexts

コンテキスト	BB 間のデータフロー
1	Start ⇒ BB1
2	BB1 ⇒ BB2 and BB2 ⇒ BB4
3	BB1 ⇒ BB3 and BB3 ⇒ BB4
4	BB4 ⇒ BB1
5	BB4 ⇒ Exit

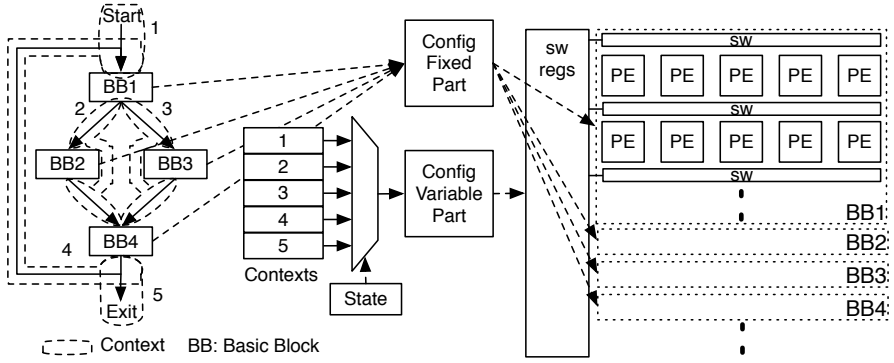

図 3 固定部と可変部に分割
Fig. 3 Dividing into fixed part and variable part

プログラム内へのある一つのコントロールの入り口から、分岐命令または、最後の命令までの命令列が一つのコンテキストを生成するからである。よって、命令列内部への分岐命令一つにつき、分岐命令の次の命令と分岐先の命令の二つの入り口があるため、コンテキストは 2 つ増える。また、プログラム外部からの入り口一つにつき、コンテキストは 1 つ増え、内部から外部への分岐命令もコンテキストは 1 つ増える。これらの総和が全コンテキストとなる。最大コンテキスト数を超えない限り、分岐命令が複数あっても実行することが可能である。ここで、従来からの手法である前方分岐命令を条件付き実行に変えることで、コンテキスト数を削減することができる。

3.2 構成情報の生成

次に、構成情報の生成フローを説明する。はじめに、図 4 に示すようにコンパイラが生成したバイナリコードを逆アセンブルする。次に、そのアセンブリコードからホットパスを抽出し、その部分から構成情報を生成する。プログラムの先頭には、RH を構成する命令 (rhc) を置き、もとのホットパスの位置には呼び出し命令 (rhr) を置く。本体プロセッサは rhc をデコードすると、構成情報を読み込み、マッピングする。次に呼び出し命令をデコード次第、RH が実行をはじめる。プロセッサ起動時に RH を構成するために、実行時に構成のためのオーバーヘッドはない。

バイナリコードを使うことにより、ソースコードを手に入れることができなかった場合でも構成情報を生成することができる、コンパイラの最適化結果が使用できる、様々な言語のソースコードから、実行ファイルが生成できるなどの利点がある。また、部分的に互換性を持たすことで、既存のコンパイラを使用することができ、制御情報を生成する機構を簡単化すること

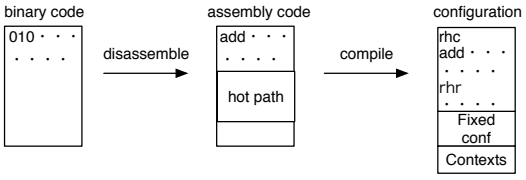

図 4 構成情報生成
Fig. 4 Generate configuration information

ができる。

4. CDDS アーキテクチャ

本章では CDDS プロセッサの詳細な構造について説明する。

4.1 CDDS プロセッサ全体図

はじめに、図 5 にプロセッサの全体ブロック図を示す。プロセッサは本体プロセッサと CDDS アクセラレータで構成する。本体プロセッサは既存のプロセッサを使う。CDDS アクセラレータはデータパスを構成するサブスイッチとメインスイッチ、コンテキストを切り替えるコンテキストコントローラ、構成情報を読み込むコンフィギュレーションローダで構成する。本体プロセッサが rhc 命令をデコードするとコンフィギュレーションローダが命令メモリから構成情報読込み、データパスを構成する。rhr 命令をデコードすると、コンテキストコントローラがコンテキストを切り替え、CDDS アクセラレータが実行を始める。

CDDS アクセラレータではプログラム中のホットパスを実行する。それ以外の計算は本体プロセッサで行うため、ホットパスまでの計算結果は RF、データメモリに保存されていおり、RF とデータメモリからデータを読み書きすることが必要である。そのため、本体プロセッサの RF の一部を共有するすることで、

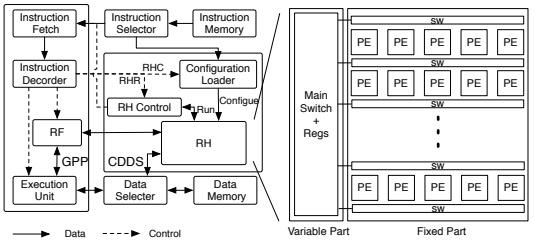

図 5 全体ブロック図
Fig. 5 Block diagram

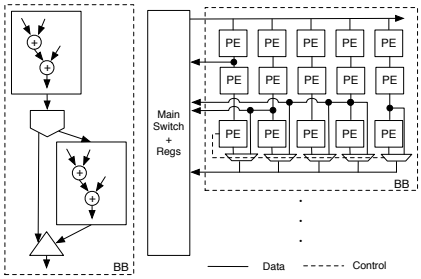

図 6 サブスイッチ概要
Fig. 6 Sub switch abst

必要なデータの読み書きを達成する。

4.2 CDDS アクセラレータのデータパス実装

データパスを構成するためには PE の出力を PE の入力に接続する機構が必要となる。この接続をサブスイッチとメインスイッチを用いて実装する。

はじめに、サブスイッチは図 6 に示すように、BB 内のデータフローを実装するために、PE 間の接続を構成する。また、サブスイッチには条件付き実行が可能な機構を取り入れる。一般的に知られているように、プログラムは短い分岐命令を複数含む。これらの分岐のためにコンテキストを使い、動的に切り替えるのは無駄がある。そのため、短い分岐を条件付き実行に変換し、サブスイッチ上に構成して実行することで、コンテキスト数を削減する。

メインスイッチは BB 間のデータフローを構成する。RF からのデータをサブスイッチの BB を構成する部位に送り、その BB からの演算結果を次の BB に送る。マッピングしたプログラムの実行が完了し、本体プロセッサに制御を戻すときに演算結果をメインスイッチを介して RF に格納する。コンテキストを実行し、その演算結果を移行先のコンテキストが使用するときには、コンテキストを切り替えると演算結果は失われるため、演算結果を保存する必要がある。また、逐次、演算結果を RF に保存し続けるようにすると、データの大規模な移動が頻発し、消費電力、性能の点で問題

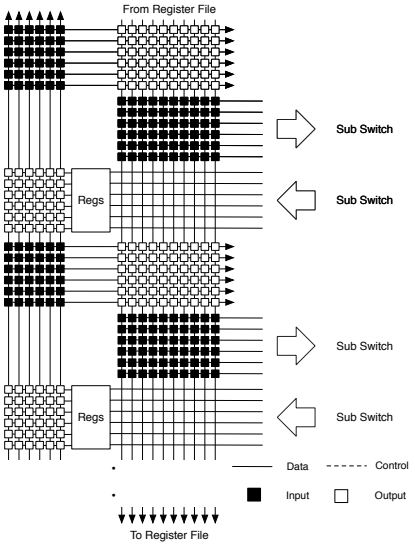

図 7 メインスイッチ
Fig. 7 Main switch

がある。そのため、演算結果を一時的に保存するためのレジスタをメインスイッチに配置する。よって、メインスイッチは図 7 に示すように RF、レジスタ、固定部上に構成した BB との接続をスイッチを使って構成する。

多くのリコンフィギャラデバイスでは RH の自由度を高めるために、PE の周りをスイッチ、配線で囲み、データを自由に送受する方法が取られていた。しかし、多くのスイッチ、配線に多くの面積必要であり、PE の面積に比べ、全体の大部分を占めてしまう。そこで、我々はサブスイッチでは限られた方向にのみデータを流すようにし、メインスイッチでのみ、データを自由に送るようにした。また、並列に実行可能なユニットを制限する。これらにより、スイッチ、配線の構造を簡単化し、PE の面積の占める割合の増大を狙う。

4.3 サブスイッチ

サブスイッチを図 8 に示す。一般的なアプリケーションの並列性は 5 程度[7] であり、1 ステージで並列実行するため、1 ステージは 5 ユニットとした。また、5 命令に 1 度は分岐、LD/ST 命令実行するため、サブスイッチには 1 ステージ毎に 5 つの ALU があり、これらのユニットのうち、一つは LD/ST ユニットを併用し、また別のユニットは一つの分岐ユニットを併用する。PE の出力を次段の PE が使うときには、その出力を次段の PE の入力にスイッチを使って接続する。これにより、組み合わせ的に命令を実行することができる。各ユニットの入力はメインスイッチからの

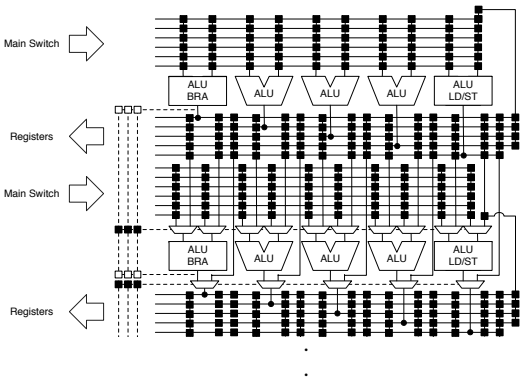

図 8 サブスイッチ
Fig. 8 Sub switch

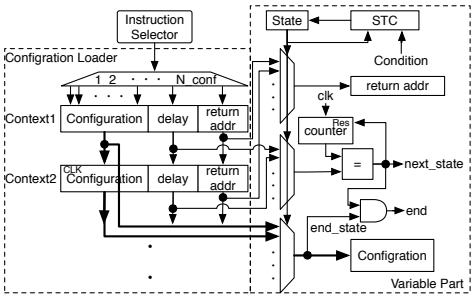

図 9 コンテキストコントローラ
Fig. 9 Context controller

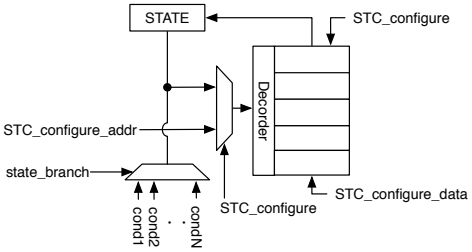

図 10 状態遷移コントローラ
Fig. 10 State transition controller

データか、前段の出力かを選択できる。原則的に PE の演算結果は次のステージのユニットの入力またはメインスイッチ内のレジスタに送るが、LD/ST ユニットには同じステージからでも結果を送ることができるようにした。これは、ALU でメモリアドレスを生成して、その次に LD/ST を行うことが多いためである。分岐ユニットで条件付き実行やコンテキスト切り替えのための条件判定結果を生成する。条件付き実行を行うためにサブスイッチの条件判定ユニットの条件判定結果を下段に送れるように構成する。また、各段のユニットの出力にセレクタを配置し、条件判定結果により、前段の出力か、そのユニットの出力か、選ぶようにした。

4.4 コンテキストコントローラ

次にメインスイッチの構成を制御するコンテキストコントローラについて説明する。コンテキストコントローラを図 9 に示す。本体プロセッサが呼び出し命令をデコードすると、STATE に初期状態として、命令で指定する状態を格納する。すると、コンテキストが切り替わり、メインスイッチのデータパスを構成する。データパスを構成するとサブスイッチに RF のデータが流れ演算が始まる。コンテキストごとの実行時間はそれぞれのコンテキストにサイクル数として付随しており、そのサイクル数まで実行する。実行時間になると、そのコンテキストの末尾の分岐命令の条件判定結果により、次に進む STATE を決める。最後のコンテキストは本体プロセッサの戻り先アドレスを保持しており、実行時間になると、このアドレスを PC に書き込む。

状態の遷移は図 10 に示す状態遷移コントローラで実行する。現在の状態とコンテキストで選択した条件判定の結果より、次の状態を選択する。状態遷移関数はメモリ内に格納される。見るべき条件判定結果はコンテキストにより、指定する。

5. 予備評価結果

CDDS アーキテクチャの評価方法を述べる。本体プロセッサとして、命令拡張が可能な Lattice 社の Lattice Mico32(LM32)[8] を利用した。LM32 の開発環境 Lattice Diamond を使い、プログラムをコンパイルし、一度バイナリコードを生成する。次に、そのバイナリコードを逆アセンブリし、アセンブリコードから構成情報を手動で生成した。この構成情報を CDDS アクセラレータにマッピングし、並列度、ユニット占有率、PE 間接続の切り替え回数を調べた。評価アプリケーションとして sepia filter, 2 値化, crc32, AES で用いる sbox を使用した。これらのアプリケーションは問題なく、全命令をマッピングすることが可能であった。一例として、図 11 に sepia filter をマッピングした図を示す。プログラムの命令をサブスイッチの PE に割り当てる。サブスイッチ部分は切り替えないため、サブスイッチの構成は一つのみである。一方、メインスイッチは切り替えるために、sepia filter の場合、3 コンテキストを保持している。斜線部分は命令をマッピングすることができなかった部分である。

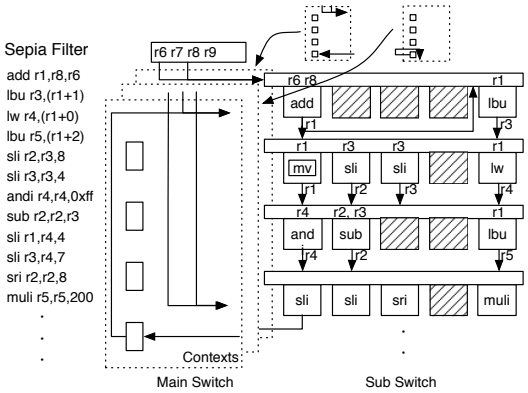

図 11 sepia filter 評価
Fig. 11 Sepia filter evaluation

5.1 ユニット占有率、コンテキスト数

次にユニット占有率の評価を行った。命令列をマッピングした図より、プログラムの命令で埋まっているユニットを使用する全ステージのユニット数で割ることでユニット占有率求めた。結果を表 2 に示す。四つのプログラムで 50〜60%という結果になった。これは、LD/ST 命令が固まり、そのロードしたデータを使う命令が多く、1, 2, 3 ステージの ALU が埋まらなかったこととデータ依存関係が多いことが原因である。コンテキスト数は表 2 に示すように、実行するプログラム内の分岐命令数の 2 倍に 1 を足した数が少なくとも必要になる。

CMA と比較して、sbox, crc32 のような複数の分岐命令を含むプログラムをコンテキストを複数用意することで、柔軟にプログラムを実行することが可能となっている。

表 2 アプリケーション評価結果
Table 2 Evaluation results

	sepia filter	2 値化	crc32	sbox
ユニット占有率	57	53	52	50
コンテキスト数	3	3	5	5
チェーンの最大長 (サイクル数)	14	15	11	11
並列度 (LM32 実行時間/最大長)	25/14=1.8	27/15=1.8	17/11=1.5	19/11=1.7

5.2 性 能 評 価

次に性能評価として CDDS アクセラレータと LM32 と比較し、並列度を調べた。LM32 の実行時間は LD/ST, 乗算命令の実行時間を 2 サイクル、他の命令を 1 サイクルとして、コンテキスト毎に計算した。また、CDDS アクセラレータの実行時間は命令列をマッピングした図から、コンテキスト毎に最大のチェーンとなる部分を探し、そのチェーンを構成する命令を LM32 と同じように計算した。並列度は LM32 の実行時間を CDDS アクセラレータの実行時間で割ることで求めた。表 3 に sepia filter の結果を示す。コンテキストの実行回数はコンテキスト 1, 3 は一回で、コンテキスト 2 は画像サイズの大きさだけループ実行を行う。画像サイズは十分大きいので、全体の並列度としてはコンテキスト 2 の並列度すなわち、1.8 になる。

評価アプリケーション全体の結果を表 2 に示す。四つのプログラムで 1.5〜1.8 という結果になった。並列度が低い原因として、プログラムにデータ依存関係が多いこと、LD/ST を並列に実行しないことが原因である。しかし、PE 同士をレジスタを介さずにスイッチを使って組み合わせ的に接続し、PE で処理が完了しだい次の PE がその結果を使うことで、遅延時間が短くなり並列度以上の性能向上を見込む。

表 3 並列度
Table 3 Parallelism

コンテキスト番号	命令数	LM32 実行時間	CDDS 実行時間	並列度	コンテキスト実行回数
1	22	27	14	1.9	1
2	22	27	14	1.9	ImageSize
3	1	1	1	1	1

5.3 PE 間接続の切り替え回数

次に消費電力削減の予備評価として、PE 間接続の切り替え回数を調べた。はじめに、命令列のデータフローの総アーク数を求める。これはコンテキスト切り替え時に PE 間接続をすべて切り替える場合の回数である。一方、CDDS アクセラレータでは PE 間の接続を構成するメインスイッチのアークのみ切り替える。このメインスイッチのアーク数を命令列をマッピングした図より求めた。表 4 に sepia filter の結果を示すように、メインスイッチアーク数の方が総アーク数に比べて小さいことがわかる。これはデータパスを可変部と固定部に分けたときに、全体に比べ可変部の範囲が小さいことを意味する。

CDDS アクセラレータはコンテキストをループ実行するときは PE 間接続を切り替えない。よって、sepia filter のようにループ回数が多いプログラムでは、表に示したメインスイッチアーク数と総アーク数から求まるアーク数の比率よりも切り替え回数の比率は限りなく小さくなる。総アーク数は従来プロセッサでのレジスタアクセス総数とみなせるため、CDDS アクセラレータにより、演算器間で直接データを受け渡しすることでレジスタアクセス回数を大幅に削減できることがわかる。

表 4 切り替えるアーク数
Table 4 Number of switch over arc

コンテキスト番号	命令数	総アーク数	メインスイッチアーク数	コンテキスト実行回数
1	22	27	6	1
2	22	27	6	ImageSize
3	1	0	0	1

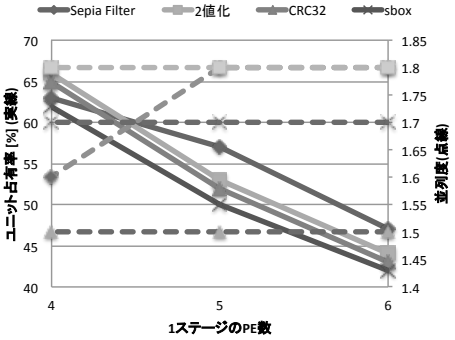

図 12 スケーラビリティ
Fig. 12 Scalability

5.4 スケーラビリティ

ここまでは、1 ステージに PE 数が 5 の場合の評価である。次に 1 ステージの PE 数を変更したときの評価結果について調べた。PE 数を変更したときのユニット占有率、並列度を図 12 に示す。PE 数を 6 にするとユニット占有率は低下する。これはステージに空きユニットがあった場合でも、データ依存関係により、これ以上命令を割り当てることができないからである。PE 数を 4 にした場合、命令が割り当てられていない ALU が複数あったために、PE 数を少なくすることで、ユニット占有率は向上した。しかし、sepia filter ではステージ当たりのユニット数が減り、ステージ数が長くなり、チェーン長が長くなったことで並列度が下がった。PE 数を 3 以下にした場合もユニット占有率は向上するが、コンテキスト数は増える。これは、PE 数を減らすとサブスイッチで下段に送れるデータ数が減少するため、メインスイッチを介す必要があり、コンテキスト数が増えるからである。

結果より、PE 数を減らすことでユニット占有率の向上を見込める。しかし、sepia filter の場合はコンテキスト数が増え、並列度が下がった。コンテキスト数が増えると再構成の回数が増加し、消費電力が増えため、PE 数を 4 以下にするのは避けたい。よって、この予備評価では PE 数はユニット占有率がより大きい 5 が最適であった。

6. ま と め

柔軟性と性能電力比の向上を目指した CDDS アーキテクチャについて述べた。従来の RP はプログラムを HW 実行することで、汎用プロセッサに比べ、命令読み込み、デコードの電力削減を達成する。また、並列に実行することで高性能、高エネルギー効率を達成する。我々はさらに、データパスを固定部と可変部とに分割し、実行時に可変部に限定して動的再構成することで、柔軟性と低消費電力性の両立を目指す。

今後は詳細な消費電力、性能評価を行うためのプロセッサのシミュレータ環境を整える。バイナリコードから構成情報を生成するコンパイラを作成する。

参 考 文 献

1) Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. *SIGARCH Comput. Archit. News*, 38(3):37–47, June 2010.
2) S. Swanson and M.B. Taylor. Greendroid: Exploring the next evolution in smartphone application processors. *Communications Magazine, IEEE*, 49(4):112 –119, april 2011.
3) Francisco-Javier Veredas, Michael Scheppler, Will Moffat, and Bingfeng Mei. Custom implementation of the coarse-grained reconfigurable adres architecture for multimedia purposes. In *FPL*, pages 106–111, 2005.
4) N. Ozaki, Y. Yasuda, Y. Saito, D. Ikebuchi, M. Kimura, H. Amano, H. Nakamura, K. Usami, M. Namiki, and M. Kondo. Cool megaarrays: Ultralow-power reconfigurable accelerator chips. *Micro, IEEE*, 31(6):6 –18, nov.-dec. 2011.
5) Yoshiki Saito, Toru Sano, Masaru Kato, Vasutan Tunbunheng, Yoshihiro Yasuda, and Hideharu Amano. A real chip evaluation of muccra-3: A low power dycamically reconfigurable processor array. In *ERSA'09*, pages 283–286, 2009.
6) M. MOTOMURA. A dynamically reconfigurable processor architecture. *Microprocessor Forum, Oct. 2002*, 2002.
7) DavidW. Wall. Limits of instruction-level parallelism. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):176–188, April 1991.
8) LatticeMico32 開発ツール, `http://www.latticesemi.co.jp/products/designsoftware/micodevelopmenttools/index.cfm`.