計算機アーキテクチャ　103－21
設　計　自　動　化　69－21
(1993. 12. 17)

ASIP向きハードウェア／ソフトウェア・コデザインシステム PEAS-Iにおけるデータパス部の最適化手法

本間啓道† 塩見彰睦† 今井正治† 引地信之††

†豊橋技術科学大学情報工学系 †† SRA

†〒441 愛知県豊橋市天伯町字雲雀ヶ丘1-1
††〒102 東京都千代田区平河町1-1-1

あらまし

PEAS-Iは，ASIPを開発するためのハードウェア／ソフトウェア・コデザイン・システムであり，特定分野の応用プログラムをより高速に実行するCPUコアとそのCPUコアをターゲットとする応用プログラム開発環境を同時に生成するシステムである．本稿では，レジスタの個数を考慮した命令セットの最適化の手法について述べ，本手法を用いて生成したCPUコアが従来のCPUコアより高速に応用プログラムを実行できることを示す．また，応用プログラムから静的解析及び動的解析を用いて最適化に必要なデータを抽出する方法について述べる．

和文キーワード　VLSI, ASIP, ハードウェア／ソフトウェア協調設計，高位論理合成

Datapath Optimization Methodology in PEAS-I : A Hardware/Software Codesign System for ASIP

Yoshimichi HONMA† Akichika SHIOMI† Masaharu IMAI† Nobuyuki HIKICHI††

†Toyohashi University of Technology †† SRA

†1-1 Hibariga-oka, Tenpaku-cho, Toyohashi, Aichi,441 Japan
††1-1-1 Hirakawa-cho, Chiyoda-ku, Tokyo, 102 Japan

Abstract

This paper describes the datapath optimization method in PEAS-I, which is a hardware/software codesign system for ASIP development. The PEAS-I system analyzes a set of application programs and associated data set, determines optimal instruction set, and then generates CPU core and application program development tools. A datapath of the generated CPU core consists of register file and computational modules. This method optimizes both the size of register file and the combination of computational modules. Using this method, a better design of the CPU core.

英文 key words　VLSI, ASIP, Hardware/Software Codesign, High-level synthesis

1 はじめに

近年の半導体の集積度の向上等に伴い，CPU コアと周辺回路を 1 チップに集積した ASIP (Application Specific Integrated Processor) の開発が行われるようになった．図 1に ASIP の構成例を示す．ASIP は 1 チップでシステムを実現するため，汎用 CPU に周辺回路を組み合わせたシステムよりも小さく，組み込み用途に多くの需要があると思われる．

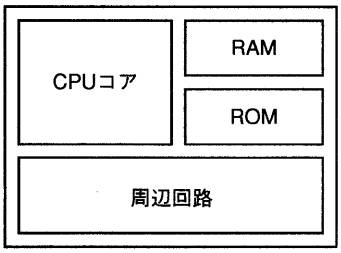

図 1: ASIP の構成例

ASIP システムの構築のためには，ASIP のハードウェア設計だけでなく，その ASIP のための応用プログラム開発ツール（コンパイラ，シミュレータ等）も実現する必要がある．数万ゲート規模の ASIP の設計を従来のゲートレベルでの設計手法で行うと，長期の設計期間を必要とする．また，応用プログラム開発ツールの設計も，ハードウェア・アーキテクチャの決定をうけて開発が開始されるため，これも ASIP システム設計の全体のパスが長くなる要因となる．設計期間の長期化は，ハードウェア，ソフトウェアの設計結果をフィードバックしてシステムを再設計することを困難にする．

近年，HDL（ハードウェア記述言語）や CAD 技術の発達により，論理合成等の作業が自動化され，簡単な CPU ならば数人日程度で設計が完了するようになってきている．しかし，チップのアーキテクチャは設計者の経験にもとづいて決定されており，数理的最適化手法などの形式的な方法はあまり用いられていない．

ASIP の設計期間を短縮し，ハードウェアとソフトウェアのバランスがとれたシステムを設計するためには，ハードウェアとソフトウェアの協調設計が必要である．この協調設計の目的となるのは，ハードウェアとソフトウェアの最適な機能分割と，設計期間の短縮である．著者らは，ASIP 開発のための協調設計システム PEAS (Practical Environment for ASIP Development) を提案し，PEAS-I を試作した [1].

本稿では現在の PEAS システムにおけるハードウェア・アーキテクチャの最適化手法の限界と新しい最適化手法について述べる．また，応用プログラムから静的解析及び動的解析を用いて最適化に必要なデータを抽出する方法について述べる．

2 PEAS-I システム

2.1 概要

PEAS-I システムは，C 言語で記述された応用プログラムとデータを入力し，与えられた制約条件のもとで最大の性能を持つ CPU コアを自動設計すると共に，その CPU コアのための C コンパイラ，シミュレータ等の応用プログラム開発ツールを自動生成するシステムである．

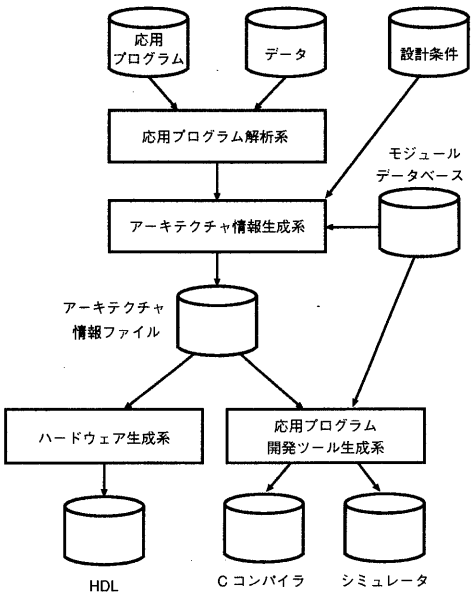

図 2: PEAS-I システムの構成

2.2 構成

PEAS-Iシステムの構成を図2に示す．PEAS-Iシステムは以下の4つのサブシステムから構成されている．

(1) 応用プログラム解析系
C言語で記述された応用プログラムとそのプログラムへの入力データの動的解析を行い，命令の実行頻度等を求める．

(2) アーキテクチャ情報生成系
(1)で得られた解析結果をもとに，チップ面積等の制約条件のもとで性能を最大にする命令セット等のアーキテクチャ情報を決定する．

(3) ハードウェア生成系
(2)で得られたアーキテクチャ情報にもとづいて，CPUコアをHDLの形で生成する．

(4) 応用プログラム開発ツール生成系
(2)で得られたアーキテクチャ情報にもとづいて，Cコンパイラ，シミュレータ等の応用プログラム開発ツールを生成する．

現在のバージョンではHDLとしてNTTで開発された高位合成システムPARTHENONのSFL (Structured Function description Language) を用いている[2]．ハードウェア生成系で生成されたCPUコアのHDL記述は，PARTHENONを用いて論理合成され，ネットリストの形式で出力される．また，Cコンパイラは，GNU Cコンパイラ[3]を利用して自動生成される．

3 アーキテクチャモデル

3.1 命令セット・アーキテクチャ

命令セットはGNU Cコンパイラの中間コードである RTL (Register Transfer Language) に対応している．この命令セットを次の3つに分類し，命令の実現方法を変化させる事により，特定分野の応用プログラムに適したCPUコアの生成を行う．

(1) *Primitive RTL (PRTL)*
応用プログラムの実行のために絶対に必要とされ，必ずハードウェアで実現されるRTL

(2) *Basic RTL (BRTL)*
ハードウェアで直接実現できるが，PRTLの組み合わせによっても実現できるRTL

(3) *Extended RTL (XRTL)*
ライブラリ関数，ユーザ定義関数に対応するRTL

現在のバージョンではXRTLはサポートしておらず，BRTLの実現方法だけを変化させている．

3.2 ハードウェア・アーキテクチャ

ハードウェア生成系で生成されるCPUコアは以下の特徴を持つ．

- 32ビットのハーバードアーキテクチャ
- 命令は3アドレスの汎用レジスタ方式
- ロード／ストア・マシン
- 命令セットはGCCのRTLに対応している
- パイプライン制御を行う

3.1節で述べたPRTLを実現するためのハードウェアであるALUや1ビットシフタ等は，PEAS-Iが生成するCPUコアの最小構成であり，これをkernelと呼ぶ．kernelによってハードウェアで実現されている命令群，すなわちPRTLを使用すれば，この命令を組み合わせることによりBRTL，XRTLに相当する動作を構成することができ，任意のCプログラムの実行が可能である．

4 ハードウェア・アーキテクチャの最適化に関する考察

PEASでは与えられた応用プログラムに合わせてCPUコアのハードウェア・アーキテクチャを最適化している．ここでいう最適化とは与えられた設計条件（ゲート数と消費電力）のもとで，応用プログラムの実行サイクル数が最小になることをいう．

4.1 現在の最適化手法

現在のPEAS-Iでは，データパス中の演算器の構成を応用プログラムに合わせて最適化している．

最適な演算器の構成を求めるために IMSP (Instruction set implementation Method Selection Problem)[4] を組み合わせ最適化問題として定式化し，これを解くことによって演算器の構成を決定している．IMSP の入力は応用プログラムを実行したときのファンクショナリティ[1]の実行頻度である．この入力から，与えられた設計条件のもとで，応用プログラムの実行サイクル数が最小になるように BRTL を実現するための演算器を取捨選択する．IMSP を解くことによって，実行サイクル数を予測することが出来る．

4.2 現在の PEAS での最適化の限界

現在の PEAS では，レジスタの個数を最適化する手法がないため，レジスタの個数はユーザが指定している．レジスタの個数を考慮せずに演算器構成を最適化した場合，以下の 2 つの点が応用プログラムの実行サイクル数を増加させる要因となり，データパス全体を見ると必ずしも最適とは言えない．

- 応用プログラムが要求する個数より CPU コアの持つレジスタの個数が少ない場合，レジスタからメモリへ変数を退避するための命令（ストア命令）やメモリからレジスタへ変数を復帰するための命令（ロード命令）が必要になる．その結果，実行サイクル数が増加する．

- 応用プログラムが要求する個数より CPU コアの持つレジスタの個数が多い場合，CPU コアはレジスタにゲートを消費され，高速な（実行サイクル数の少ない）演算器を持つことができない．その結果，実行サイクル数が増加する．

4.3 応用プログラムの実行サイクル数に関する考察

応用プログラムの実行サイクルは大きく分けて，レジスタの個数に影響される部分：MI(r) と命令セットの実現方法（データパス部の演算器構成）に影響される部分：MD(r,D) に分けられる．応用プログラムの実行サイクル数 Cy(r,D) は次の式で表せる．

$$Cy(r,D) = MI(r) + MD(r,D) \qquad (1)$$

[1]C 言語における演算子と関数を総称する概念

ここで，r はレジスタの個数を，D は設計条件（データパス部のゲート数，消費電力）をそれぞれ表す．また，MI(r) は CPU コアの演算器の構成を固定したとき，レジスタを十分に持つ CPU コアで応用プログラムを実行したときの実行サイクル数とレジスタが r 個の CPU コアで応用プログラムを実行したときの実行サイクル数との差分を表す．MI(r) は図 3のようにレジスタの個数に対して単調に変化すると考えられる．また，レジスタの個数 r を増加させていったとき，MI(r) が 0 となる最小の r の値を R_{max} とする．要するに，レジスタの個数をそれ以上に増やしても実行サイクル数が減少しないレジスタの個数を R_{max} とする．レジスタウインドウ等の特殊なレジスタ周辺のハードウェアを用いない場合，R_{max} は一意に求められる．

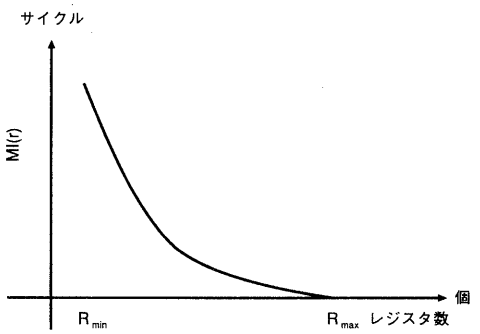

図 3: MI(r) の変化

次に MD(r,D) は，演算のみに必要なサイクル数を表す．MD(r,D) は演算器の構成によって変化するが IMSP を解くことによって予測することが可能である．

設計条件（データパス部の総ゲート数のみを考慮）を一定としたとき，MD(r,D) は図 4のように，レジスタの個数に対して単調に増加すると考えられる．MD(r,D) は演算器の構成に大きく影響を受ける部分で，演算器に使用できるゲート数が多ければそれだけ高速な演算器（通常，高速な演算器ほどゲート数が大きい）を持つことが出来る．図 4は横軸にレジスタの個数を取っているため，演算器に使用できるゲート数は左側（r の値が小さい場合）の方が多く，グラフは単調増加となっている．

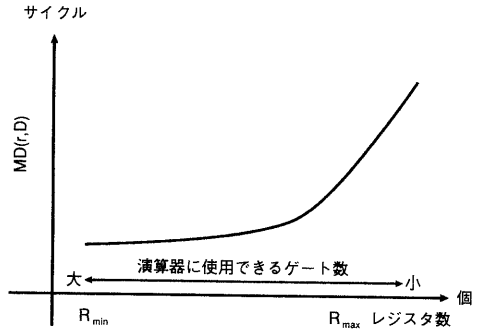

図 4: 演算にかかるサイクル MD(r,D) の変化

以上の考察から，設計条件（データパス部の総ゲート数のみを考慮）を一定にしたときの応用プログラムの実行サイクル数 Cy(r,D) とレジスタ数の関係は図5のようになると考えられる．すなわち，レジスタの個数が多すぎても，少なすぎても実行サイクル数 Cy は増加し，レジスタの個数 r=R_{opt}のときに Cy は最小となると考えられる．

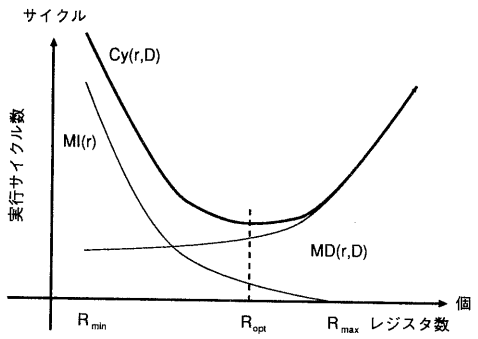

図 5: 実行サイクル数の変化

5 新しい最適化手法

5.1 最適化の手続き

本論文で提案する方法では，レジスタファイルも含めたデータパス部全体を最適化する．この手続きでは，応用プログラムの実行サイクル数が最小となる R_{opt}を求め，レジスタの個数を R_{opt}個にした場合の演算器の最適な構成を求める．

レジスタと演算器に割り当てる設計条件のトレードオフを考慮して，最適なレジスタの個数を決めることで，従来の最適化手法に比べて，さらに応用プログラムの実行サイクル数を小さくすることが可能となると考えられる．

R_{opt}を求める手続きを図 6に示す．手続き中の R_{min}はCコンパイラが最低限必要とするレジスタの個数を表す定数，R_{max}は 4.3 に述べたように MI(r) が0になるときの最小のレジスタ数である．手続き中で，ステップ2の (3) に現れる MD(r,D) は IMSP を解いて予測する．また，このとき，IMSP に与えるゲート数に関する制約条件は設計条件 D から r 個のレジスタが消費するゲート数を差し引いたものを使用する．図6の手続きで応用プログラムの実行サイクル数が最小になる R_{opt}を見つけた後，IMSP を解く（あるいは，手続き中で IMSP を解いているので，その結果を保存しておく）ことによって最適なレジスタの個数と演算器の構成を求めることができる．

図 7は従来の手法と本手法を用いた時の実行サイクル数の比較を概念的に示したものである．本手法を適用して得られる ASIP の実行サイクル数の曲線は，レジスタの個数 (r) を変化させて得られる実行サイクル数の曲線の包絡線となる．

5.2 実行例

図 8に従来の最適化手法との比較を示す．この図は 5 次の FIR フィルタをサンプルプログラムに用いて，それぞれの手法で最適化された CPU コアでサンプルプログラムを実行したときの実行サイクル数を示している．同一の設計条件のもとで本手法適用すると，実行サイクル数の削減が可能であることが知られる．同図では，レジスタの個数を 8 個としたときと比較すると，実行サイクル数が最大で 68.6 %，平均で 14.3 %短縮されている．

6 応用プログラムの解析方法

6.1 最適化手続きで必要な情報

第 5 節で述べたように，データパス部の最適化を行うためには次の 2 種類の情報が必要となる．

- 入力：MI(r)，R_{max}，設計条件 D
- 出力：R_{opt}
- 方法：

ステップ 1：[初期化]

(1) r := R_{min}

(2) Cy_{min} := ∞

ステップ 2：[繰り返し]

(3) レジスタの個数が r 個の時の実行サイクル数 Cy(r,D) を次式で求める．
Cy(r,D) := MI(r) + MD(r,D)

(4) Cy(r,D) < Cy_{min} の場合

- R_{opt} := r
- Cy_{min} := Cy(r,D)

(5) r := r + 1

r > R_{max} になるまでステップ 2 を繰り返す．

ステップ 3：[終了]

(6) R_{opt} を出力する．手続き終了．

図 6: R_{opt} を求める手続き

(1) 各ファンクショナリティの実行頻度

(2) MI(r)

これら 2 種類の情報を応用プログラムとその入力データから抽出する作業が応用プログラムの解析である．

単純な方法としては，応用プログラムをシミュレータ上で実行し，各ファンショナリティの実行回数と応用プログラムの実行サイクル数を計測すればよい．ただしこの方法では，MI(r) を計測するためには，全ての BRTL に対応するハードウェア演算器を持ち，レジスタの個数だけが異なるコンパイラを多数（R_{max}-R_{min}種類）用意し，それぞれのコンパイラで目的コードを生成し，それら全てをシミュレータ上で実行しなければならない．したがって，この方法では 1 つの応用プログラムを解析するのに

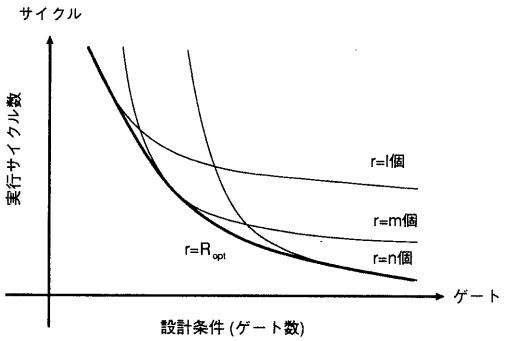

図 7: 最適化の手法と実行サイクル

その応用プログラムが大きければ大きいほど，また，R_{max}-R_{min}が大きければ大きいほど，解析時間が長くなる．現在用いているシミュレータの上での目的コードのシミュレーション時間は，ターゲット CPU で目的コードを直接実行した場合の 5000～6000 倍必要である．したがって，この方法は効率が良くない．解析時間を短縮するためには，応用プログラムの実行回数は極力減らしたい．以下ではその方法について考察する．

6.2 レジスタの個数と実行サイクル数の差に関する考察

今，同じ命令セットを持ち，レジスタの個数だけが異なる 2 つの CPU コアがあると仮定する．これら 2 つの CPU コアをそれぞれターゲットとする 2 種類のコンパイラで，同一のソースプログラムから目的コードを生成し比較すると，変数をレジスタの上に載せるためのロード命令や，レジスタ上におけなくなった変数をメモリ上に退避するためのストア命令の個数が変化しているが，演算に使用する命令の種類や順序は本質的に変化していない．したがって，レジスタの個数が異なる CPU コアで応用プログラムを実行したときの実行サイクル数の差は，変数の退避／復帰を行うロード／ストア命令の実行回数の差を計測することで，求めることができる．

6.3 基本ブロックの実行回数に関する考察

応用プログラムの動作を基本ブロックを単位に考える．基本ブロックとは連続した文の列からなり，

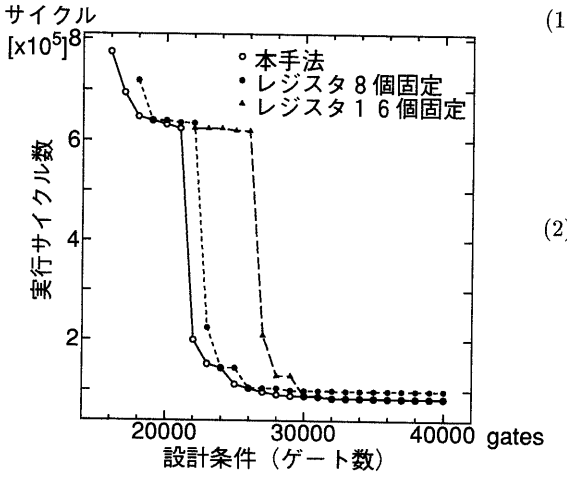

図 8: 設計条件と実行サイクル数の関係

制御は先頭の文に与えられ，そのあと，途中で停止したり，途中から分岐したりしないで，最後の文から制御が離れるものをいう [5]．基本ブロックは制御の推移の最小単位である．

レジスタの個数が変化しても，プログラムの制御の流れは変わらない．したがって，レジスタの個数が変化しても，基本ブロック間の制御の推移や，個々の基本ブロックの実行回数は変化しない．そこで，応用プログラムを１回だけ実行し，各基本ブロックの実行回数を測定すれば，基本ブロックの実行回数とその基本ブロックを１回実行したときに増減するロード／ストア命令の個数からその基本ブロックでおきるロード／ストア命令の増減を知ることが出来る．したがって全ての基本ブロックについて増減を求め，合計することで応用プログラム全体の増減を求めることが出来る．

6.4 MI(r) の算出方法

以上の考察をまとめると，MI(r) の値を計算するためには次のようにして応用プログラムの解析を行えばよいことがわかる．

6.4.1 静的解析部：

(1) 応用プログラムをコンパイルする．

十分な個数のレジスタを持ち，演算器の構成も最大な CPU コアをターゲットとするコンパイラを用いて目的コードを生成する．この目的コードは動的解析に使用する．

(2) (1) で生成した目的コードから R_{max} を発見する．

R_{max} は 4.3 に示すように，もうそれ以上レジスタを増やしても実行サイクル数が減少しないレジスタの個数を示している．例えば，256 個のレジスタを用意した CPU コア用のコンパイラが，ある応用プログラムをコンパイルした結果，30 個のレジスタしか使わないコードを生成したとする．このとき R_{max} は 30 である．たとえ，256 個のレジスタを持っていたとしても，31 個目以降のレジスタを使うコードをコンパイラが生成しないのであれば，残りのレジスタは実行サイクル数の減少には貢献しない．したがって，レジスタを十分に持つ CPU コア用のコンパイラでコンパイルした目的コードを調べることによって，R_{max} を発見できる．

(3) 各基本ブロックごとに増加するサイクル数を求める．

レジスタの個数 r を R_{max} から R_{min} まで変化させ，各 r について，各基本ブロックでの命令数の増加を調べる．

6.4.2 動的解析部：

シミュレータを用いて目的コードをシミュレートし，各基本ブロックの実行回数を計測する．

6.4.3 MI(r) の算出：

それぞれの基本ブロックにおいて，増加するサイクル数と基本ブロックの実行回数との積をとり，その合計を MI(r) とする

6.5 MI(r) の算出例

図 9に示すように，応用プログラムが 2 つの基本ブロックに分割され，それぞれループを構成すると

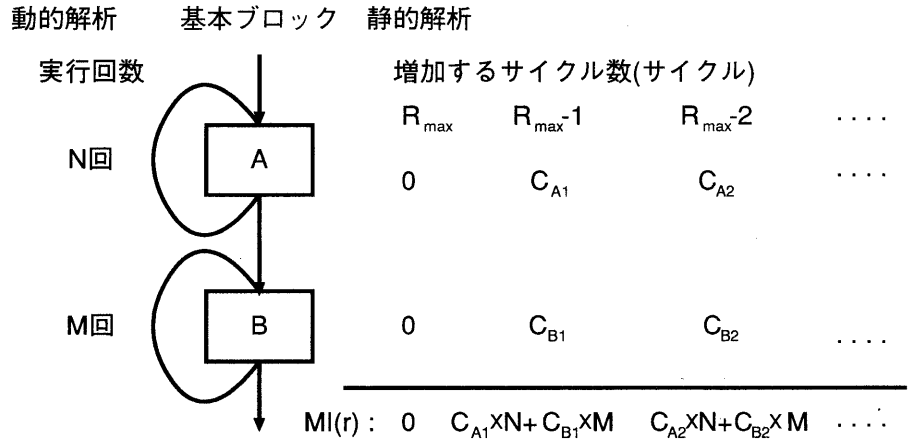

図 9: MI(r) の算出方法

仮定する．図の静的解析の部分はレジスタの個数が R_{max},R_{max}-1,R_{max}-2 のとき基本ブロック A ではそれぞれ，0，C_{A1},C_{A2}だけサイクル数が増加し，基本ブロック B ではそれぞれ，0，C_{B1},C_{B2}だけサイクル数が増加する事を示している．また，動的解析の方は基本ブロック A は N 回実行され，基本ブロック B は M 回実行されたことを示している．この場合，MI(r) は次のようになる．

$MI(R_{max}) = 0$

$MI(R_{max}\text{-}1) = C_{A1} \times N + C_{B1} \times M$

$MI(R_{max}\text{-}2) = C_{A2} \times N + C_{B2} \times M$

7 結び

本稿では，PEAS-I におけるハードウェア・アーキテクチャの最適化の現状と限界について述べ，その解決策として，レジスタファイルを含めたデータパス全体の最適化手法について述べた．サンプルプログラムを用いた実験では，この手法を用いることで従来の最適化に比べて，応用プログラムの実行サイクル数が最高 68.6 %，平均 14.3 %短縮できることを示した．

また，最適化に必要なデータを収集する応用プログラムの解析方法とその際の時間短縮法について述べた．現在，応用プログラム解析ツールを製作中であり，近日中に完成し評価を行う予定である．

謝辞

本研究を進めるにあたり御討論いただいた鶴岡高専の佐藤淳助手，豊橋技術科学大学大学院博士課程の A.Alomary 氏，木下貴史氏，ならびに豊橋技術科学大学 VLSI 設計研究室の諸兄に感謝いたします．また，設計ツール及びライブラリ等を提供して頂いた NTT コミュニケーション科学研究所，VLSI テクノロジー社，ならびに研究をご支援頂く（株）SRA に感謝いたします．

参考文献

[1] Sato,J.,et al.: "PEAS-I: A Hardware/Software Codesign System for ASIP Development," IEICE Trans.,to apper

[2] NTT データ通信:「PARTHENON User's Manual」, 1989

[3] Stallman,R.: Using and Porting GNU C Compiler, Free Software Foundation, Inc., 1991

[4] Alomary,A.,et al.: "An ASIP Instruction Set Optimization Algorithm with Function Module Sharing Constraint," IEICE Trans., Vol.E76-A. No.10, pp1713-1720, 1993.

[5] Aho, A.V.,Sethi, R and Ullman, J.D. : Compilers, principles, techniques, and tools., Bell Telephone Laboratories, Inc., 1986