一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2018-88,CPSY2018-98,RECONF2018-62(2019-01)

高位合成による専用命令実装手法の予備的評価

岩本　凌大†　藤枝　直輝††　市川　周一††　坂本　譲二†

† 豊橋技術科学大学大学院 電気・電子情報工学専攻
†† 豊橋技術科学大学 電気・電子情報工学系
〒 441-8580 愛知県豊橋市天伯町雲雀ケ丘 1-1
E-mail: †iwamoto@ccs.ee.tut.ac.jp, ††ichikawa@ieee.org

あらまし　ソフトウェアの処理手順などの知的財産権や技術的ノウハウの保護は重要な課題である．坂本ら (2018) は，ソフトウェアの秘匿性を向上するため，応用プログラムの一部の関数を命令セットシミュレータに組み入れ，専用命令をもつソフトプロセッサとして高位合成することを提案した．本稿では，この手法の適用可能性を示すため，CHStone ベンチマークの各プログラムにこの手法を適用する．秘匿対象の関数を選択する際の基準として，処理時間が大きな関数を優先して選択し，実行時間への影響を評価する．
キーワード　FPGA，特殊化，ソフトプロセッサ

Preliminary evaluation of special instruction implementation methods by high level synthesis

Ryodai IWAMOTO†, Naoki FUJIEDA††, Shuichi ICHIKAWA††, and Joji SAKAMOTO†

† Electrical and Electronic Information Engineering Course, Graduate School of Toyohashi University of Technology
†† Department of Electrical and Electronic Information Engineering, Toyohashi University of Technology
1-1 Hibarigaoka, Tempaku-cho, Toyohashi, Aichi, 441-8580 Japan
E-mail: †iwamoto@ccs.ee.tut.ac.jp, ††ichikawa@ieee.org

Abstract　Protection of intellectual properties and technical know-how is an important issue. In our previous work, we proposed implementing some functions of software as special instructions of a soft processor generated from an instruction set simulator by using high level synthesis. In this research, we examine this method by applying it to each program of CHStone benchmark. As a criterion for selecting functions to be concealed, a function with a long processing time is preferentially selected, and the influence on the execution time is evaluated.
Key words　FPGA, specialization, soft-processor

1. はじめに

組込みシステムにおいて，ソフトウェア解析の防止は重要な課題である．一般にハードウェアはソフトウェアよりも解析が困難であり，ソフトウェアの一部を論理回路として実装することで，ソフトウェアを秘匿することができる [1] [2] ．

一方で組込みシステムではハードウェア記述言語（Hardware Description Language，HDL）で記述されたソフトプロセッサが多く利用されており，主に FPGA（Field Programmable Gate Array）上に実装されている．ソフトプロセッサは，システムやアプリケーションの要求に応じて，機能や構成を柔軟に変更できる．近年では C 言語などから HDL を生成する高位合成技術 [3] が成熟しており，C 言語で記述されたソフトプロセッサ [4] の研究や，命令セットシミュレータからのソフトプロセッサ [5] を合成する研究が行われている．

我々の先行研究 [6] では高位合成により命令セットシミュレータからソフトプロセッサを生成し，秘匿対象となるソフトウェアの一部の関数を専用命令として組み入れ，特殊化されたプロセッサを生成することを提案した．専用命令を実装するアプリケーションとしては，CHStone ベンチマーク [7] を対象としている．しかしながら，専用命令化する関数の選択方法がアドホックで，応用として示されたアプリケーションが少ない（3 種類）ことが問題であった．

本研究では CHStone の 11 種のアプリケーションに専用命令

を実装する．また，秘匿対象として選択する関数に明確な基準を設け，専用命令化した際の傾向を確認する．CHStone のうち MIPS のみ実装の対象から除外したが，これは MIPS が main 関数のみで構成されているためである．この main 関数を秘匿対象として専用命令化すると，ソフトウェア全体がハードウェア化されるため，本研究の目的であるソフトウェアの一部を秘匿するという目的に沿わない．

2. 背　　景

2.1 ASIP

特定用途向けにカスタマイズされた命令セットを持つプロセッサを ASIP （Application-Specific Instruction-set Processor）という [8] ．ASIP の設計ではハードウェア/ソフトウェア協調設計について古くから研究がなされている [9] ．本研究で生成する特殊化されたプロセッサもある種の ASIP であるといえるが，本研究の目的はハードウェア化による処理の秘匿である．消費電力や実行時間の削減が大きな目的である研究 [8] [9] とは目的が異なる．本研究では関数ごとに単一の専用命令を用いることで，専用コンパイラが必要無いように専用命令を実装する．

2.2 アーキテクチャ記述言語

DSP（Digital Signal Processor） も含めたプロセッサの開発においては，LISA [10] や SIM-nML [11] のようなアーキテクチャ記述言語が用いられることがある．これらはプロセッサの記述に特化した設計がなされているため，性能の良いハードウェアを生成できるが，独自言語であるため学習コストが高い．

2.3 高 位 合 成

高位合成とは C 言語など動作レベルの記述言語から，HDL により記述された回路を自動的に生成する技術である．商用の高位合成ソフトウェアとして，例えば Mentor の Catapult [12] や NEC の CyberWorkBench [13] がある．再帰関数や動的ポインタなどのハードウェアによる実装が困難な処理はツールによって制限・禁止される可能性がある．

本研究では Xilinx が提供する Vivado HLS [14] を用いる．Vivado HLS は C/C++ 言語からの高位合成をサポートしている．しかし，システムコールや動的なメモリ割り当て，再帰を含む関数は合成できない．また，ポインタの型変換について C 言語ネイティブな型の間での一定ケースに限るなどの成約がある．

2.4 高位合成を用いたソフトプロセッサ

Skalicky ら [5] は C 言語で記述された命令セットシミュレータを高位合成にかけることで直接プロセッサを生成することを提案した．さらに彼らは対象のアプリケーションで利用されない命令の実装を省略することによるハードウェア使用量の削減を提案した．これはプリプロセッサにより不要な命令に関する記述を削除することで行われ，実装されない命令は未定義の命令として扱われる．

Ahmed ら [4] は C 言語で記述された MIPS 命令セットベースのソフトプロセッサ HIRUNDO を提案し，オープンソースで公開している．HIRUNDO はいくつかの命令をプロセッサの実装から省くことでハードウェア使用量を削減する機能がある．しかしながらこのプロセッサは subleq（減算してその結果が 0 以下なら分岐する） という単一の命令だけを実行できるコプロセッサをもっており，プロセッサに実装されなかった命令はコプロセッサでのソフトウェア処理でエミュレートされる．

本研究では先行研究 [6] 同様 Skalicky ら [5] の手法をベースとし，命令セットシミュレータの高位合成により得られたソフトプロセッサに専用命令を組み入れる．命令の一部を実装しないため，プログラム実行における汎用性は失われるが，組込みシステムのように実行する応用が事前に決まっているならば許容されうる．必要があればは Ahmed ら [4] の手法を適用することで若干のハードウェアオーバヘッドと引き換えに汎用性を取り戻すことが可能である．

3. 専用命令の組入れ

3.1 組入れの手順

本研究では，坂本 [6] の提案手法を用いて専用命令を組み入れる．図 1 に専用命令組入れの流れを示す．先行研究 [4] [5] と同様に MIPS アーキテクチャを対象とし，専用命令化は C 言語の関数レベルで行う．専用命令化された関数は，シミュレータから生成されるプロセッサとともにハードウェア化される．関数のローカル変数の一部も回路中に組み入れられ，解析が困難となり，セキュリティ性が向上する．また，プログラムとデータを一括で保護できる．

アプリケーション側では秘匿対象の関数を，専用命令を呼び出すためのコードに書き換える．秘匿対象の関数は専用命令のためのラッパ関数となる．専用命令はインラインアセンブラによって直接記述する形で呼び出す．このアプリケーションをコンパイルして実行バイナリを生成する．

シミュレータ側では，まず秘匿対象の関数およびその関数から呼び出される関数の記述を，アプリケーション側からシミュレータ側に移す．シミュレータの命令実行に関するコードに対

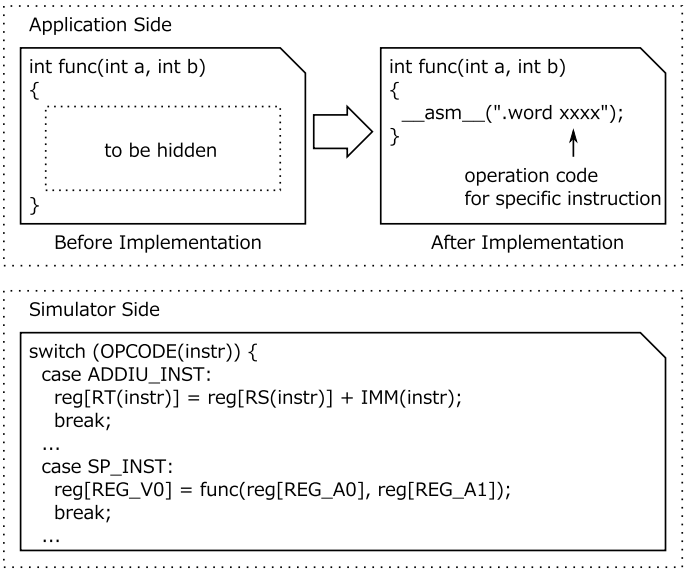

図 1　専用命令の実装方法 [6].

Fig. 1　A method to implement special instructions.

し，専用命令が実行されたときに秘匿対象の関数を実行するための記述を追加する．MIPS の ABI （Application-Binary Interface）の規定に従い，関数の引数にはレジスタ a0〜a3 を，返り値には v0，v1 を用いる．これらのレジスタを経由して引数や返り値の受け渡しを行う．シミュレータを Vivado HLS により高位合成することでプロセッサを生成し，アプリケーション側で生成されたバイナリを実行する．

先行研究 [6] では CHStone ベンチマーク [7] のうち，ADPCM，AES，MOTION の 3 つの応用について，専用命令化する関数の引数を渡す方法を比較，検討している．本研究では CHStone のその他のアプリケーションについてこの手法を用いて専用命令を実装する．

3.2 組入れにおける制約

専用命令の組入れには，主に検討するべき制約が 3 つある．1 つ目は使用する高位合成ツールの制約事項の影響である．秘匿対象の関数内に，使用する高位合成ツールで制限・禁止された記述がある場合，それを回避するようにソフトウェアを修正する必要がある．本研究では Vivado HLS を用いるため，システムコール・動的なメモリ割り当て，再帰を含む関数は使用できない．また，ポインタの型変換も制限される． CHStone ベンチマークでは，近年の高位合成ツールで合成可能とするために，これらを用いない記述をしている．

2 つ目は，専用命令化する関数の選択に関する制約である．図 2 に専用命令の組入れによって関数の呼び出し関係が影響を受ける場合の例を示す．図 2（a）に示すプログラムでは，関数 func1 が func2 と func3 を呼び出し，これらが共通の func4 を呼び出す．ここで func2 を専用命令化したものが図 2（b）である．このとき func2 が呼び出す func4 もシミュレータ側に移す必要がある．しかし func4 をアプリケーション側から削除すると，func3 から func4 を呼び出すことができなくなる．func4 はアプリケーション側に残しておくか，func4 単体で別途専用命令化する必要がある．しかし，前者では func4 に関する秘匿化が達成できず，後者ではハードウェア使用量のオーバヘッドが増大する可能性がある．よって専用命令化する関数は，その関数以下の関数が他で使用されていないものであることが望ましい．

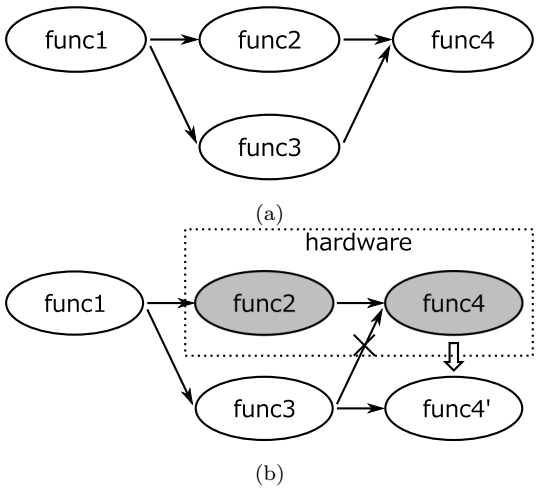

図 2　関数の呼び出し関係における専用命令組入れの影響 [6].

Fig. 2　Effect of special instruction incorporation in function call relation.

3 つ目は，関数に含まれる変数や，引数の受け渡しに関する制約である．図 1 に示したように関数が少数の値渡しの引数をもち，グローバル変数を使用しなければ，レジスタを経由した引数や返り値の受け渡しが可能である．そうでない場合，メインメモリにあるデータの読み書きが必要である．先行研究 [6] では CHStone の ADPCM，AES，MOTION についてケーススタディを行った．

3.3 専用命令の参照渡しの実装手法

先行研究 [6] では専用命令への参照渡しの手法として 3 つの実装手法を提案した．1 つ目はメインメモリのポインタを渡す手法である．この手法では，関数にメインメモリへのポインタを渡し，関数内部でメインメモリを直接操作する．

2 つ目はスクラッチパッドメモリ（SPM）を利用する手法である．この手法では，プロセッサ内にソフトウェアから自由に読み書き可能なローカルメモリを搭載し，関数にはそのメモリを操作させる．シミュレータ側では関数に対応する専用命令のほか，SPM へのロード・ストア命令をメインメモリへのロード・ストア命令とは別に実装する．アプリケーション側のラッパ関数ではメインメモリから SPM への読み出し，関数に対応する専用命令の呼び出し，SPM からメインメモリへの書き戻しを順番に実行する．

3 つ目は一時変数を用いる手法である．この手法では，関数ごとに引数の配列の内容を渡すための専用のローカルメモリをプロセッサ内に用意し，関数にはそのメモリを操作させる．SPM との違いはローカルメモリが関数ごとに独立である．また，アプリケーション側のラッパ関数に行わせていた読み出し・書き戻しをシミュレータ側で行うため，アプリケーション側のラッパ関数への変更は不要であり，生成されるバイナリはメインメモリのポインタを渡す手法と同一となる．

秘匿対象の関数がグローバル変数を操作する場合，秘匿対象の関数に出現するグローバル変数の操作を，参照渡しされたポインタ引数を経由した操作に変換することで，参照渡しの引数と同様にして対応できる．

4. 秘匿対象の関数の選択

4.1 秘匿対象の関数の選択方法

評価対象のプログラムは CHStone 1.11 [7] の MIPS を除いた 11 種とする．MIPS は main 関数のみで記述されるプログラムであり，main 関数を専用命令化すると，プログラム全体がハードウェア化される．これは処理の一部をハードウェア化することで秘匿する本研究の目的とは異なる．

先行研究 [6] では，秘匿対象の関数の選択に明確な基準が設定されていなかった．本研究では，処理時間の大きな関数を優先して秘匿対象とする．本研究の目的は処理の秘匿であるが，今回は試験的に処理時間の大きなものを選択した．関数ごとの処理時間の測定は，Core-i5 4440 を搭載した PC 上の仮想マシン（Ubuntu16.04）でプロファイルを取ることによって行った．

表 1　秘匿対象として選択した関数.
Table 1　Functions selected as hidden target.

application	function name	time[%]
DFADD	propagateFloat64NaN	8.63
GSM	gsm_mult_r	8.92
AES	AddRoundkey_InversMixColumn	53.57
ADPCM	upzero	24.18
JPEG	YuvToRgb	14.64
BLOWFISH	BF_encrypt	60.79
DFDIV	mul64To128	15.14
DFMUL	mul64To128	9.72
DFSIN	mul64To128	11.36
MOTION	Flush_Buffer	96.60
SHA	sha_transform	87.25

CHStone ベンチマークは処理時間がとても短く，GNU プロファイラ (gprof) では測定精度の関係で正確な測定ができなかった．そのため，関数の処理時間の測定には KCachegrind [15] を用いた．秘匿対象の関数の選択は処理時間の大きな関数を優先して選択するが，3.2 節で述べた組入れの制約に従う．選択する関数以下の関数が他で使用されていないもので，最も処理時間が大きな関数を選択する．

4.2　秘匿対象の関数

4.1 節で述べた方法で選択した関数を表 1 に示す．time は各関数のプログラム内で占める処理時間の割合を示す．また，以下のように選択された関数の特徴によってプログラムを分類した．

- DFADD・GSM：引数の参照渡しが無く，返り値を持つ
- AES：引数の参照渡しが有り，返り値を持つ
- ADPCM･JPEG･BLOWFISH･DFDIV･DFMUL･DFSIN：引数の参照渡しが有り，返り値を持たない
- MOTION・SHA：引数の参照渡しが有り，返り値を持たず，グローバル変数を直接操作する

プロファイルの結果，処理時間の大きな関数の多くは，多くの関数を呼び出していた．そのため，図 2 のような制約にかかるものが多く，今回選択した関数で，実際に最も処理時間が大きいものを選択できたアプリケーションは ADPCM，BLOWFISH，MOTION の 3 つであった．SHA の sha_transform 関数はプログラム全体の処理時間の 87.25% を占めるが最も処理時間の大きな関数ではない．実際には sha_transform 関数を呼び出す関数が最も処理時間が大きな関数であるが，制約にかかり選択されなかった

5.　評 価 方 法

プロセッサの高位合成環境，およびその上で動作するソフトウェアの開発環境を表 2 に示す．評価指標は専用命令を組入れて合成されたプロセッサのハードウェア使用量，およびプログラムのサイズと実行時間について定める．ハードウェア使用量については，シミュレータ記述を Vivado HLS により高位合成，生成されたハードウェア記述を Vivado により論理合成し，報告された論理ブロック（Slice）数を指標とする．合成オプションはともにデフォルトとし，動作周波数は 100MHz とする．

表 2　高位合成およびソフトウェアの開発環境.
Table 2　High level synthesis and software development environment.

HLS Tool	Vivado HLS v2018.1
Logic Synthesis Tool	Vivado v2018.1
Target Device	Artix-7 XC7A100TCSG324-1
Benchmark	CHStone v1.11
Compiler	gcc v5.4.0 (mipsel-linux-elf-gcc)
Instruction Set	MIPS32 Release 1

プログラムのサイズは，コンパイルされたバイナリに対して readelf コマンドを実行し計測する．報告された命令 (.text) と読み出し専用データ (.rodata) の大きさの合計を指標とする．実行時間の見積もりは，メインメモリにプログラムをロードした状態でプロセッサを Vivado HLS 上でシミュレートすることで行う．ロードされたプログラムはその終端で exit システムコールを呼び出すため，システムコール命令の実行をもってプロセッサはメインループを抜けるものとする．Vivado HLS により見積もられた実行サイクル数を実行時間の指標とする．また，入力は各プログラム中にあらかじめ埋め込まれているものを用いる．

評価対象は以下の 3 つに大別される．

- Reduce：Skalicky ら [5] の先行研究にならい，プログラムの実行に最小限必要となる命令を実装したプロセッサ
- Special：選択された関数を専用命令化した上で，特殊化されたプログラムの実行に最小限必要となる命令を実装したプロセッサ
- HLS：プログラム全体を高位合成した専用回路

special は更に 3 つの実装に分別され，直接メモリのポインタを渡す手法を ptr，SPM を使用する手法を spm，一時変数を用いる手法を tmp で示す．専用命令化された関数はアプリケーション側から削除されるため，Reduce と Special で実行に必要な命令は異なる場合がある．

HLS では対象のプログラムをすべてハードウェア化している．プログラムのサイズの評価は行わない．生成された回路の実行サイクル数の見積もりを実行時間の指標とする．

6.　専用命令の実装および評価

先行研究 [6] の手法を用いて各アプリケーションに専用命令を実装した． DFDIV・DFMUL・DFSIN（mul64To128 関数）の実装において，先行研究では考慮されていなかった，64 ビット変数の引数の受け渡しが発生した．この場合，MIPS の ABI の規定では，レジスタ 2 つに渡って受け渡される．以下に mul64To128 関数のプロトタイプ宣言を示す．

```
typedef unsigned long long int bits64;
void mul64To128 (bits64 a, bits64 b, bits64 * z0Ptr,
bits64 * z1Ptr);
```

このとき，引数 a の下位 32 ビットが a0 レジスタに，上位 32

ビットが a1 レジスタに格納されている．したがって，これら 2 つのレジスタの値を読み出し，1 つの 64 ビット変数としてシミュレータ側の関数に渡すように操作する必要がある．引数 b についても同様である．

表 3 に各アプリケーションの高位合成結果を示す．AES において，一時変数を用いた実装ではシミュレーション時にエラーが発生し，実装できていない． BLOWFISH・MOTION における，直接メモリのポインタを渡す手法と SPM を用いる手法および，DFDIV・DFMUL・DFSIN の SPM を用いる手法は高位合成時にエラーとなる．これはビット幅の異なる変数のポインタ型の間のキャストが原因である．DFSIN において，一時変数を用いる手法ではシミュレーションの結果が reduce と異なっており，正しく実装できていない．専用命令化された mul64To128 関数の実行回数が本来のアプリケーションでの実行回数と異なっており，原因は調査中である．

図 3 に各アプリケーションのコードサイズの削減率を示す．reduce のコードサイズを基準とする．special は参照渡しの無いアプリケーションを，ptr は直接メモリのポインタを渡す手法，spm は SPM を用いる手法，tmp は一時変数を用いる手法を示す．ptr および tmp の方が spm よりも削減率が大きい．spm はアプリケーションでも SPM へのロードおよびストアに関する処理を行うためである．

図 4 に各アプリケーションの reduce を基準としたレイテンシーの削減率を示す．prof は KCachegrind でプロファイルした各プログラムにおける秘匿対象の関数が占める処理時間の割合である．言い換えれば，prof は対象の関数の実行時間をゼロと仮定した場合の処理時間の見積もりである．プロファイリングとシミュレーションとの間の環境の違いはあるものの，レイテンシーの削減率はおおむねこれよりも少し小さくなると予測できる．DFADD・GSM・AES・ADPCM・JPEG（spm）・DFDIV・MOTION・SHA では prof より小さな削減率となった．最も差が大きなのは ADPCM の SPM 実装であり，prof よりも 15% ほど小さな値となった．JPEG（ptr・tmp）・DFMUL・DFSIN（ptr）では prof よりも大きな削減率となっ

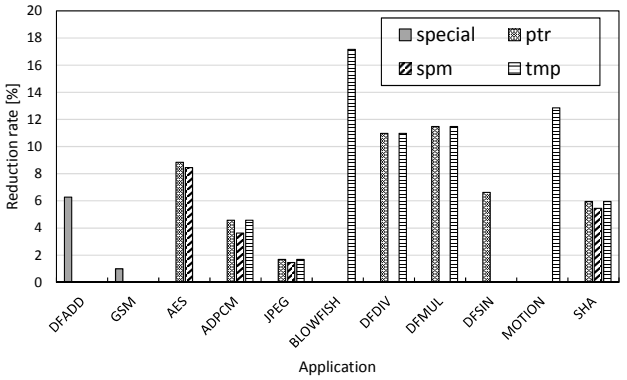

図 3 各アプリケーションにおける reduce を基準としたコードサイズの削減率．
Fig. 3 Code size reduction rate based on reduce in each application.

表 3 各アプリケーションのそれぞれの実装による合成結果．
Table 3 Synthesis result by each implementation of each application.

App.	Impl.	SLICE	LUT	FF	DSP	BRAM
DFADD	reduce	376	1,193	397	0	2
	special	360	1,224	399	0	2
	HLS	1,501	4,812	2,440	0	0
GSM	reduce	411	1,356	580	3	2
	special	439	1,415	636	4	2
	HLS	1,244	3,604	3,127	51	8
AES	reduce	583	1,748	1,065	7	2
	ptr	799	2,325	1,753	7	4
	spm	704	2,098	1,658	7	6
	HLS	633	1,553	1,810	0	10
ADPCM	reduce	273	898	346	3	2
	ptr	513	1,532	787	7	2
	spm	469	1,430	596	7	4
	tmp	556	1,655	775	7	4
	HLS	2,473	7,395	4,660	128	11
JPEG	reduce	389	1,283	652	3	2
	ptr	403	1,338	652	3	2
	spm	406	1,335	653	3	3
	tmp	501	1,494	857	3	5
	HLS	5,596	16,761	11,472	15	53
BLOWFISH	reduce	246	794	381	0	2
	tmp	1,247	3,383	3,825	0	4
	HLS	1,413	4,084	4,821	0	15
DFDIV	reduce	457	1,522	488	7	2
	ptr	545	1,754	724	16	2
	tmp	485	1,563	700	15	2
	HLS	1,244	3,365	3,210	24	0
DFMUL	reduce	457	1,522	488	7	2
	ptr	614	1,919	881	23	2
	tmp	612	1,950	840	22	2
	HLS	643	1,867	1,031	16	0
DFSIN	reduce	414	1,374	490	7	2
	ptr	578	1,790	776	19	2
	HLS	3,566	10,249	7,400	43	0
MOTION	reduce	356	1,192	513	0	2
	tmp	791	2,283	1,568	0	4
	HLS	837	2,377	1,802	0	4
SHA	reduce	329	1,072	360	0	2
	ptr	774	2,447	1,730	0	4
	spm	670	2,221	1,635	0	6
	tmp	781	2,583	1,745	0	6
	HLS	473	1,462	1,525	0	18

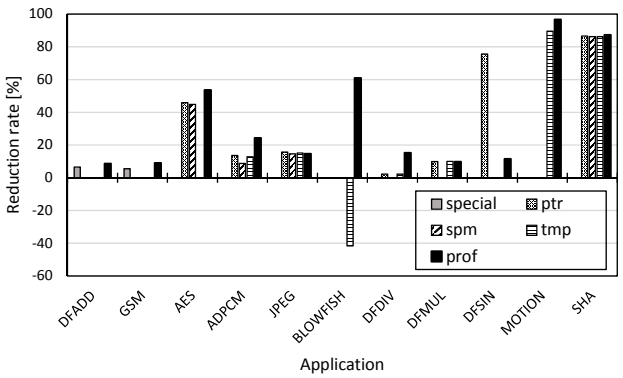

図 4　各アプリケーションにおける reduce を基準としたレイテンシーの削減率.

Fig. 4　Latency reduction rate based on reduce in each application.

た．JPEG（ptr・tmp）・DFMUL・DFSIN（ptr）ではその差は 1% 未満である． DFSIN（ptr）では 64.2% の差が発生した．BLOWFISH(tmp) は reduce よりもレイテンシーが大きくなるという結果となった．BLOWFISH で専用命令化される BF_encrypt 関数はベンチマークのテストパターンで 1171 回実行される．また，一時変数を用いた実装の場合，関数の実行前後にメモリと一時変数との間で計 2088 個の配列要素のコピーが行われる．これによりレイテンシーが reduce よりも増加したのではないかと考えられる．

7. おわりに

本研究では先行研究 [6] の手法を用いて，CHStone ベンチマークの MIPS を除くすべてのアプリケーションに専用命令を実装した．先行研究では応用が少なく，秘匿対象の選択がアドホックであった．本研究では CHStone ベンチマークの MIPS を除いたすべてのアプリケーションに専用命令を実装し，専用命令化する関数の選択は処理時間の大きなものを優先するという基準を設けて選択した．本研究の目的は処理の秘匿であり，コードサイズの大きなものを優先する方法も考えられる．これは今後の課題とする．

処理時間の大きな関数を選択して専用命令化し，実際に実行時間へ与える影響を評価した．ほとんどのアプリケーションでは，専用命令化を施したプロセッサでのプログラムの実行時間は，対象関数の実行時間をゼロと仮定した見積もりに近い値を示した．一方で，一時変数を用いた実装を BLOWFISH の BF_encrypt 関数のような大きな配列を引数とする関数に適用すると，プログラムの実行時間が専用命令化する前よりも長くなる場合があることも確認された．

謝辞

本研究は，JSPS 科研費 16K00072 の補助を受けたものである．

文　献

[1] S. Ichikawa et al., “An FPGA implementation of hard-wired sequence control system based on PLC software,” IEEJ Trans. Electrical and Electronic Engineering, vol.6, no.4, pp.367–375, March 2011.

[2] 松岡佑海, 藤枝直輝, 市川周一, “難読化ツール oLLVM を用いたハードウェア難読化手法の評価,” 電気学会論文誌 D，vol.139，no.2，2019.

[3] R. Nane et al., “A Survey and Evaluation of FPGA High-Level Synthesis Tools,” IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol.35, no.10, pp.1591–1604, Oct. 2016.

[4] T. Ahmed et al., “Synthesizable-from-C Embedded Processor Based on MIPS-ISA and OISC,” Proc. IEEE 13th International Conference on Embedded and Ubiquitous Computing (EUC 2015), pp.114–123, Oct. 2015.

[5] S. Skalicky et al., “Designing customized ISA processors using high level synthesis,” Proc. 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp.1–6, Dec. 2015.

[6] 坂本譲二，“プロセッサの高位合成および特殊命令の実装・評価,” Master’s thesis，豊橋技術科学大学 電気・電子情報工学専攻，2017.

[7] Y. Hara et al., “Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis,” Journal of Information Processing, vol.17, pp.242–254, 2009.

[8] C. Galuzzi and K. Bertels, “The Instruction-Set Extension Problem: A Survey,” ACM Trans. Reconfigurable Technology and Systems, vol.4, no.2, pp.18:1–18:28, May 2011.

[9] J. Sato et al., “PEAS-I : A Hardware/Software Codesign System for ASIP Development,” IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences, vol.E77-A, no.3, pp.483–491, 1994.

[10] S. Pees et al., “LISA-machine description language for cycle-accurate models of programmable DSP architectures,” Proc. 36th annual ACM/IEEE Design Automation Conference (DAC 1999), pp.933–938, 1999.

[11] S. Basu and R. Moona, “High level synthesis from Sim-nML processor models,” Proc. 16th International Conference on VLSI Design (VLSID 2003), pp.255–260, Jan. 2003.

[12] Mentor Graphics Corp, “Catapult HLS Platform”. http://www.mentorg.co.jp/hls-lp/catapult-high-level-synthesis/index.html.

[13] NEC, “ASIC･FPGA 設計向け C 言語ベース高位合成ツール CyberWorkBench”. http://jpn.nec.com/cyberworkbench/index.html.

[14] Xilinx Inc, “Vivado Design Suite ユーザーガイド：高位合成,” User Guide UG902 (v2018.1)，April 2018.

[15] J. Weidendorfer, “KCachegrind”. http://kcachegrind.sourceforge.net/html/Home.html.