Vol.2019-ARC-240 No.41
Vol.2019-SLDM-191 No.41
Vol.2019-EMB-53 No.41
2019/2/28

ハードウェア抽象化記述SHIMによる性能見積のための LLVM-IR命令実行時間計測手法

鳥越　敬[1,a)]　枝廣正人[1,b)]

概要：組込みシステムの大規模・複雑化に伴い、モデルベースで並列度の高いソフトウェアを開発する需要が高まっている。モデル設計時に各ブロックの実行サイクル数を見積り、モデル設計に活用することで、ソフトウェアの並列度や性能設計精度を向上させることができる。そこで、SHIM と呼ばれるハードウェア特徴記述を用いてソフトウェアの性能を見積る、SHIM 性能見積が研究されている。SHIM においては LLVM-IR 命令ごとにターゲットプロセッサでの実行時間を対応付けるが、本論文では、この実行時間を計測する手法を提案する。ある LLVM-IR 命令から生成されるターゲットプロセッサのアセンブリ命令列は一通りではないため、実行時間計測は簡単ではない。提案手法では、見積に適切な実行サイクル値を回帰分析によって求める。回帰分析の際の変数となるパラメータの適切な設定を考察するとともに、実際に SHIM 性能見積に用いて精度を評価し、本手法の妥当性を検証した。

LLVM-IR instruction latency measurement in performance estimation with software-hardware interface for multi-many-core

1. はじめに

近年，組込みマイコンに求められる作業量が増加しており，組込みソフトウェアの大規模・複雑化が進行している．その結果，開発時間短縮，人的ミス削減が可能な開発プロセスとして，モデルベース開発 [1] が普及してきている．モデルベース開発では，仕様書にある制御ロジックや制御対象をブロック線図で表されるモデルで表現し，それを基に開発を進める．これにより，設計段階でのシステムの振る舞い検証，コード自動生成，効率的なテストが可能となり，開発プロセスが効率化できる．

また，組込みソフトウェアの大規模・複雑化は，ハードウェアの要求性能の向上も引き起こしている．これまでは１つのプロセッサの性能を向上させることで性能要件を満たしてきたが，消費電力などの制約が厳しい組込みシステムでは，シングルコアプロセッサでは性能要求を満たすことが困難になっており，プロセッサのマルチ・メニーコア化が進行している．

[1] 名古屋大学大学院情報学研究科
Graduate School of Informatics, Nagoya University
[a)] numakuro620@ertl.jp
[b)] eda@ertl.jp

これらの要因により，モデルベース開発のマルチ・メニーコア対応が求められている．モデルベース開発のマルチ・メニーコア対応の手法の一つに，コードベースの並列化がある．これは，モデルから生成された逐次コードをもとに，並列コードを作成する手法である．しかし，この手法は，システムの設計段階で，タスク分散などを考慮していないため，並列度を向上させにくいという課題がある．また，並列化後に性能目標に達しなかった場合に，システム設計から見直す必要が発生し，手戻りコストがかかってしまう．

この課題の解決策として，我々は，モデルベース並列化（MBP：Model Based Parallelization）[2] を提案している．モデルベース並列化では，システムの設計段階で並列度が高くなるように，タスク分散を考慮した設計を行う．また，モデルから逐次コードではなく，並列コードを自動生成する．並列度の高いモデルを設計するためには，モデルの各ブロックの処理時間を把握し，各コアに負荷を均等に割り当てることが重要である．そのためには，モデル設計段階で各ブロックの処理時間を計測する性能見積手法が必要である．

モデルベース並列化における性能見積では，実機や命令

セットシミュレータ（ISS）を必要とせずに実機の特徴を考慮した見積を行うことが要件となっている．これは，組込みシステムでは，ソフトウェアとハードウェアの開発が同時に進められることがあり，設計初期では実機が存在しない中でソフトウェア設計を行う場合があるためである．そこで，ハードウェア抽象化記述である SHIM[3] を用いた性能見積である，SHIM 性能見積の研究が行われている．SHIM にはハードウェアの特徴をパラメータ化したものが記述されており，SHIM 性能見積ではこれらを利用することで，実機を用いずに実機の特徴を考慮した性能見積が可能である．

本研究は，SHIM 性能見積を複数のターゲットハードウェアに適用するために，先行研究の課題であった，ターゲットハードウェアごとに異なる計測プログラムを作成しなければならないという課題を解決した，回帰分析を用いた SHIM 計測手法を提案する．

2. 準備

2.1 SHIM

マルチ・メニーコアの普及により，様々な特徴を持つプロセッサができ，アプリケーションに最適なプロセッサを選択することができるようになった．しかし同時に，ツール開発者がマルチコアデバイスを効率的に利用するためには，膨大なマニュアルを読んで，マルチコアデバイスの特徴を理解することが必要となり，ツール開発者の負担が増大している．

そこで，マルチコアデバイスを利用するためのツール開発支援を目的として，The Multicore Association[4] は，SHIM（Software-Hardware Interface for Multi-many-core）と呼ばれる国際的な標準規格を策定した．SHIM には，ボードやチップの特徴をパラメータ化したものが，XML 形式で記述される．SHIM は，ソフトウェア開発者に向けてデザインされているため，IP コンポーネント同士がどのように接続されているかなどソフトウェア開発に重要でない情報は記載されていない．その代わり，レイテンシ，FIFO レジスタ通信のようなマスタ同士の通信といったメモリアクセスの情報や，クロック，命令セット，キャッシュサイズといった基本的なプロセッサ情報などの，ソフトウェア開発上で重要となる事柄が記載されている．

SHIM には，ComponentSet という要素があり，これがボード上のプロセッサやメモリを表現している．ComponentSet は，プロセッサやアクセラレータといったマスター型構成要素に相当するハードウェアを表現する MasterComponent と，メモリなどのスレーブ型構成要素に相当する SlaveComponent を持っている．MasterComponent には，キャッシュサイズや実行可能な命令，その実行サイクル数などの情報が記載されている．SlaveComponent には，メモリのサイズや読み書き属性などが記載されている．

なお，SHIM は現在 SHIM2.0 がリリースされ，パイプラインやメモリアーキテクチャ，ヘテロジニアス対応の強化など様々な点が改善されているが，本研究は前バージョンである SHIM1.0 を対象としている．SHIM2.0 を用いれば，詳細なアーキテクチャシミュレーションを併用することにより，高精度の見積が可能となる．しかし，我々が目的とする開発初期段階でのモデルレベル見積には高速性が要求されるため，アーキテクチャシミュレーションを利用せず，SHIM1.0 レベルの情報を用いた高速見積が期待されている．

SHIM の活用方法としては，SHIM 性能見積の他にも，SHIM の情報を OS ミドルウェアが活用し，自動コンフィグレーションを行うことなどが期待されている．

2.2 LLVM

LLVM[5] とは，任意のプログラミング言語の静的コンパイルと動的コンパイルの両方をサポートできる SSA ベースコンパイル基盤を提供することを目的としているオープンソースのコンパイラプラットフォームである．LLVM Core ライブラリは，LLVM 特有の中間表現である LLVM-IR を中心に構成されており，ソースやターゲットに依存しないオプティマイザと多くの一般的な CPU のコード生成をサポートするライブラリである．LLVM Core ライブラリは，よくドキュメント化されているため，新たに作成した言語のコンパイラに，LLVM を容易に利用できる．LLVM を用いたコンパイラは，フロントエンド，ミドルエンド，バックエンドの３つから構成されているが，これらはすべて独立しているため中間生成物をそれぞれ個別に利用することが可能である．

LLVM で使用される LLVM-IR は，表現力，十分な型システム，および拡張性を同時に備えながら，軽量かつ低レベルであることを目指して設計されている．さらに，機械語がすべての高級言語を表現可能な”universal IR”であるのと同様に，LLVM-IR もすべての高級言語を表現するのに十分な能力を持つことも目指している．また，LLVM-IR は無限の仮想レジスタを持つと仮定されているため，レジスタスピルによるレジスタ退避命令などが存在しないという特徴がある．

2.3 SHIM 性能見積の概要

SHIM 性能見積とは，ターゲットハードウェアから計測した性能値をもとに SHIM を作成し，それを用いてターゲットハードウェア上でソフトウェアを実行した際の性能を予測することである．SHIM には，ターゲットハードウェアの性能情報として，プロセッサで実行可能な命令が LLVM-IR 命令セットを利用して記載されている．SHIM の LLVM-IR 命令セットの性能情報は，Performance オブジェクトにおいて表現されており，この性能情報を利用し

Vol.2019-ARC-240 No.41
Vol.2019-SLDM-191 No.41
Vol.2019-EMB-53 No.41
2019/2/28

て SHIM 性能見積を行う．SHIM で，LLVM-IR 命令セットが用いられている理由は，SHIM に汎用性を持たせるためである．SHIM 性能見積においては，命令名を用いて SHIM の性能情報を取得するが，仮に SHIM の性能情報が，ターゲットの機械語命令セットで書かれていたとすると，ターゲットによって機械語は異なる可能性があるため，同じ動作の命令であるにもかかわらずターゲットごとに異なる命令名でアクセスしなければならなくなる．しかし LLVM-IR 命令セットを利用することで，異なる機械語のターゲットでも同じように SHIM の性能情報にアクセスできるようになる．しかし，機械語命令ではなく LLVM-IR 命令を用いるために SHIM 性能見積において難しくなる問題もある．その一つがレジスタミス，キャッシュミスの扱いである．LLVM-IR では無限の仮想レジスタを想定しているため，レジスタスピルが発生することがなく，一度ロードしてレジスタに配置されたデータはなくなることがない．しかし，通常はレジスタの数は有限であり，レジスタスピルによって一度レジスタに配置されたデータがなくなっているという事態が発生する．その場合，LLVM-IR では必要なかったメモリアクセスが発生し，さらにそのメモリアクセスでキャッシュミスによるペナルティなども発生する可能性がある．このような機械語と LLVM-IR の間の差にどのように対応するかは，SHIM 性能見積において重要な要素である．

2.4 SHIM 性能見積の課題

SHIM 性能見積の大きな課題の 1 つに，SHIM に記載する Performance 値の測定がある．上記の通り，Performance は性能情報を表しており，Latency と Pitch という値を持っている．Latency は，1 命令の実行サイクル数で，Pitch は，連続して命令が実行された場合の 1 命令あたりの実行サイクル数である．現代のハードウェアは，複数命令を同時に実行するなど，連続して命令が実行される場合に効率よく実行する機能が搭載されている．Pitch は，このようなサイクル数を表現している．Latency と Pitch は，Best，Worst，Typical の 3 つのサイクル数からなる．これらは，それぞれ，最短，最悪，最頻のサイクルを表している．しかし，SHIM 作成者がこれらの値を測定するには，ターゲットハードウェアにおいて，どのような場合に最短，最悪，最頻の実行サイクルになるかを理解する必要がある．このためには，ターゲットハードウェアの仕様を詳細に理解する必要があり，ユーザーによる完全な SHIM の作成は困難である．

2.5 既存研究

SHIM 性能計測の既存研究として，佐合ら [8] による研究がある．同研究では，最短の実行状況は最頻・最悪の実行状況より定義しやすいことに着目し，インラインアセンブラを用いて最短の実行状況を表現する計測プログラムを作成し，SHIM 計測を行っていた．しかし，この手法では，ハードウェアごとに個別の計測プログラムを用意する必要があるため，複数ハードウェアに対し見積を適用することが難しいという課題があった．本論文で提案する SHIM 計測手法は，この課題を解決する手法となっている．

3. SHIM 計測提案手法

本研究では，先行研究での SHIM 作成方法の課題解決として，回帰分析を用いた SHIM 計測手法の提案を行う．

異なるアーキテクチャに対して，同一の計測プログラムを用いて SHIM 性能値を測定するために，性能値を変数とする多変量の最小二乗法を用いて性能値を予測する．本手法の計測フローは図 1 の通りである．

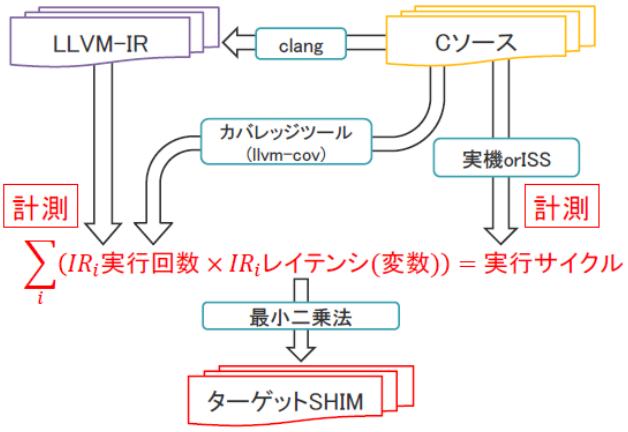

図 1 提案 SHIM 計測フロー

まずサンプルプログラムとなる C 言語プログラム（以下，C ソース）を用意し，それを clang などの LLVM フロントエンドを用いて LLVM-IR にする．その際にデバッグ情報を付与しておくことで，C ソースと LLVM-IR の対応を取ることができる．そして，C ソースを llvm-cov などのカバレッジツールで実行することで，C ソースのカバレッジ情報を取得する．そして，LLVM-IR と C ソースの対応情報と，C ソースのカバレッジ情報を利用して，LLVM-IR のカバレッジ情報を取得する．一方で，C ソースを実際に実機や ISS で実行することで，実機上での実行サイクル数を取得する．そして，これらの情報を式 1 に代入する．

$$\sum_i (IR_i Counts \times IR_i Latency) = TotalCycle \quad (1)$$

なお，$IR_i Counts$ は LLVM-IR 命令 i の実行回数，$IR_i Latency$ は LLVM-IR 命令 i のレイテンシ，$TotalCycle$ は，実機上での実行サイクル数である．

サンプルプログラムを複数用意することで，式 1 が複数生成できる．これらの式を，$IR_i Latency$ を変数とする多

変量の最小二乗法で解き，LLVM-IR 命令のレイテンシを決定する．この手法において特に重要となる点は次の 2 点である．

- サンプルプログラムの用意
  サンプルプログラムを十分な数用意しなければならないが，処理の傾向や出現する LLVM-IR 命令が偏らないように用意する必要がある．
- 変数設定
  この手法では，LLVM-IR 命令のレイテンシを最小二乗法の変数に割り当てるが，単純に 1 対 1 で割り当てると変数の数が多くなりすぎてしまい，サンプルプログラムの必要数が多くなってしまう．しかし，同じ変数内でのレイテンシの差が大きいと，最小二乗法の精度が低下してしまうため，大まかすぎる割り当ては精度低下に繋がってしまう．

本研究では，本手法において重要な問題である LLVM-IR 命令の変数への割り当てについて，4 つの分類方法を提案する．

3.1 命令の処理による分類

この手法は，LLVM-IR 命令が行っている処理に着目し，処理が似ている命令ごとに分類を行う．LLVM-IR 命令の処理については，LLVM Language Reference Manual[7] を参考にした．LLVM-IR には様々な処理の命令があるが，変数が増えすぎないように大まかに表 1 のように分類した．

整数演算命令を arithmetic, 浮動小数点演算命令を float, load 命令を load,store 命令を store に割り当て，残りの命令を others とした．メモリアクセス命令は大半のプログラムにおいて出現回数が多く，精度が重要となるため，load や store を 1 対 1 対応にした．便宜上，この分類を origin 分類と呼ぶ．

表 1 命令の処理による分類（origin）

変数名	命令名
arithmetic	add
	sub
	sdiv
	srem
	urem
	mul
float	fadd
	fsub
	fdiv
	fmul
	fcmp
load	load
store	store
others	その他の命令

3.2 先行研究の SHIM を参考にした分類

この手法は，佐合 [8] で作成された RH850/E1M-S2 の SHIM のサイクル値を参考にし，表 1 の分類において，変数内でサイクル差が大きいものを分離するものである．

この方法で作成された分類が，表 2 である．便宜上，この分類を RH850 分類と呼ぶ．

表 2 先行研究の SHIM を考慮した分類 (RH850)

変数名	命令名
arithmetic	add
	sub
	mul
div	srem
	urem
	sdiv
float	fadd
	fsub
	fmul
fdiv	fdiv
	fcmp
load	load
store	store
callret	call
	ret
others	その他の命令

3.3 オペランド依存関係の有無による分類

この手法は，佐合 [8] で定義された，命令間のオペランド依存関係を考慮した分類である．オペランド依存関係とは，LLVM-IR において，前の命令の結果のレジスタを次の命令がオペランドに使用していた場合に，その 2 つの命令間の依存関係と定義されているものである．命令間にオペランド依存関係が存在する場合，後の命令は，先の命令の結果を待たないといけないため，先の命令の実行が終わるまで実行できない．SHIM の pitch の値は，複数命令を同時に実行した際の 1 命令あたりの実行サイクル数であるため，オペランド依存関係が存在する場合，先の命令は pitch の値では実行できず，latency の値で実行される．

そこで，その pitch と latency の差を考慮して変数内のサイクル差を小さくすることを目的とした分類がこの手法である．この手法を反映した結果，分類は表 3 のようになった．なお，arithmetic_d や add_d など，_d と付いている変数や LLVM-IR 命令は，_d が付いていない同名の変数や LLVM-IR 命令のオペランド依存関係がある場合を表現している．便宜上，この分類を operand 分類と呼ぶ．

3.4 キャッシュミスを考慮した分類

この手法は，メモリアクセス命令のキャッシュミスの有無で変数を分けることで，メモリアクセス命令の変数内の

Vol.2019-ARC-240 No.41
Vol.2019-SLDM-191 No.41
Vol.2019-EMB-53 No.41
2019/2/28

表 3 オペランド依存関係の有無を考慮した分類 (operand)

変数名	命令名	
arithmetic	add	mul
	sub	
arithmetic_d	add_d	mul_d
	sub_d	
div	srem	sdiv
	urem	
div_d	srem_d	sdiv_d
	urem_d	
float	fadd	fmul
	fsub	
float_d	fadd_d	fmul_d
	fsub_d	
fdiv	fdiv	fcmp
fdiv_d	fdiv_d	fcmp_d
load	load	
load_d	load_d	
store	store	
callret	call	ret
others	その他の命令	

サイクル差を減らすことを目的とした手法である．2.3 節で述べたように，SHIM 性能見積において，LLVM-IR が無限の仮想レジスタを想定していることによる実機とのレジスタスピルの差を考慮することは重要である．よって，ここでは，ここまで変数としていた load/store 命令の代わりに，キャッシュアクセス回数，メモリアクセス回数を変数として分類する．LLVM-IR の load/store 命令は，レジスタミスが発生しないため，実機での load/store 回数は，式 2 で表される．

$$Target\_load/store = LLVM\text{-}IR\_load/store + LLVM\text{-}IR\ Reg\ Access \times RegMissRate \quad (2)$$

これが，キャッシュアクセス回数となる．そして，メモリアクセス回数は，キャッシュアクセス回数にキャッシュミス率を掛けたものになる．その結果，分類は表 4 のようになった．なお，cacheaccess はキャッシュアクセス回数で，memaccess はメモリアクセス回数である．

4. 精度評価実験

ここでは，3 章で述べた各変数設定に対し，精度評価実験を行う．

4.1 評価対象

本実験の SHIM 作成ターゲットハードウェアは，Raspberry Pi3 Model B+である．基本性能は以下の通りである．

- CPU : Cortex-A53
- コア数 : 4 コア

表 4 キャッシュミスを考慮した分類

変数名	命令名	
arithmetic	add	mul
	sub	
arithmetic_d	add_d	mul_d
	sub_d	
div	srem	sdiv
	urem	
div_d	srem_d	sdiv_d
	urem_d	
float	fadd	fmul
	fsub	
float_d	fadd_d	fmul_d
	fsub_d	
fdiv	fdiv	fcmp
fdiv_d	fdiv_d	fcmp_d
callret	call	ret
cacheaccess	cacheaccess	
memaccess	memaccess	
others	その他の命令	

- CPU クロック : 1.4GHz
- メモリ : 1GB
- キャッシュ : 32KB L1 キャッシュ, 512KB L2 キャッシュ
- OS : linux ver4.14.68

4.2 評価手法

表 5 のような 20 個のサンプルプログラムを用意し，それぞれの実機での実行サイクル数を計測した．また，LLVM-IR カバレッジ取得ツールを利用し，それぞれのサンプルプログラムの，LLVM-IR 命令実行回数を取得した．

そして，これらの情報から回帰分析によって各変数設定について変数の値を求めた．その際，実行サイクル数の大きさによる重みの差が生まれないように，実行サイクル数を 1000 とする正規化を行ってから回帰分析を行った．精度評価としては，サンプルプログラム 20 個と，それとは別の 5 つのテストプログラム（表 6）に対し，以下の式で誤差率を算出し評価した．

$$EstimateResult = \sum_i (IR_i Counts \times IR_i Latency) \quad (3)$$

$$ErrorRate = (EstimateResult - TotalCycle) \div TotalCycle \quad (4)$$

なお，$EstimationResult$ は見積実行サイクル，$IR_i Counts$ は LLVM-IR 命令 i の実行回数，$IR_i Latency$ は LLVM-IR 命令 i のレイテンシ，$ErrorRate$ は，見積の誤差率，$TotalCycle$ は実機上で測定した見積目標値となる実行サイクルを表す．

表 5　サンプルプログラム一覧

名称	概要	実機上での実行サイクル数
{Bubble/Merge/Quick}Sort.c	要素数 100 の int 型配列に対する { バブル/マージ/クイック } ソート	{214453/72974/40717}
Euclidean Algorithm.c	疑似乱数の組の最大公約数をユークリッド互除法で求める	231123
Gaussian Elimination.c	10 個の連立方程式の解をガウスの消去法で求める	36060
NumberPlace.c	数独を解く	15036946
{Fibonacci.c/Fibonacci_F.c}	{int/float} の変数を用いて 17 番目のフィボナッチ数を求める	{104719/114517}
NapierNumber.c	ネイピア数を求める	2799
{Perfect Number.c/Perfect Number_F.c}	1 から 1000 までの整数の完全数の数を {int/float} の変数で求める	{9965781/10008956}
FunctionCallTest.c	再帰関数や大域変数を利用する関数の呼び出しを行う	1677
SimpleFilter Process.c	疑似乱数による要素数 1000 の int 型配列にフィルター処理を行う	155399
Celsiusto Fahren.c	0.01 から 109.89 までの 0.01 刻みの数字を摂氏から華氏に変換する	24020598
sum.c	0 から 999 までの合計を求める	18748
{Exchange.c/Exchenge_F.c}	{int/float} の値を入れ替える関数を 100 回呼び出す	{3964/6450}
{fadd.c/fmul.c}	float の { 足し算/掛け算 } を 2000 回行う	{61241/59175}
PrimeNumber.c	2 から 999 の整数の間の素数の数を求める	1601278

次に，計測に必要な，実機での実行サイクル数等の取得方法を説明する．

4.2.1　実機での実行サイクル数の取得

Cortex-A53 プロセッサには，Performance Monitor Unit(以下 PMU) と呼ばれる，サイクル数を測定するサイクルカウンタと，特定のイベントを測定するイベントカウンタで構成された機構が搭載されている．この機構を利用して，各サンプルプログラムを 100 回実行し，その平均の実行サイクル数を実行サイクル数とした．なお，100 回実行の各実行の最初に，そのとき保持しているキャッシュを無効にすることで，他実行の影響を受けないようにした．また，計測は，割り込みの影響を極力少なくするために，4 番目のコアに固定して行った．

表 6　テストプログラム一覧

名称	概要	実機上での実行サイクル数
hanoi.c	3 枚のハノイの塔を解く	864
SeriesSum.c	1 から 10 までの和を等差数列の和の公式から求める	56
fcmp.c	浮動小数点数の比較を 8 回行う	893
{RadixConv.c/RadixConv_F.c}	{int/float} の変数で 10 進数を 2 進数に変換する	{733/1062}

4.2.2　実機でのキャッシュミス率の取得

3.4 節の分類を適用するためには，実行の際のキャッシュミス率の取得が必要である．そこで，サイクル数と同様に，PMU を用いてキャッシュミス率を取得した．なお，実際に取得したイベント回数は以下の通りである．

- L1 キャッシュアクセス回数
- L1 キャッシュリフィル回数
- L2 キャッシュアクセス回数
- L2 キャッシュリフィル回数

これらの情報を用いて，式 5，6 で L1,L2 キャッシュミス率を求めた．なお，キャッシュリフィル回数を用いたため，厳密なキャッシュミス率とは異なっていることには注意が必要である．

$$L1MissRate = L1Refill \div L1Access \quad (5)$$

$$L2MissRate = L2Refill \div L2Access \quad (6)$$

また，3.4 節では，キャッシュアクセスとメモリアクセスの 2 変数としていたが，今回のターゲットハードウェアは L1 キャッシュ，L2 キャッシュを持っているため，L1 キャッシュアクセス，L2 キャッシュアクセス，メモリアクセスの 3 変数まで分けられる．しかし，細かく分類しすぎても逆に精度が落ちることも考えられるため，L1 キャッシュアクセスのみ（分類しない）の 1 変数，L1 キャッシュアクセスと L2 キャッシュアクセスの 2 変数，L1 キャッシュアクセス,L2 キャッシュアクセス，メモリアクセスの 3 変数の分類をそれぞれ実験した．なお，便宜上これらの分類をそれぞれ L1 分類，L2 分類，mem 分類と呼ぶ．

4.3　実験結果

各分類の変数の値は表 7 から表 12 のようになった．また，サンプルプログラム 20 種に対する各分類の見積結果は表 13 となり，テストプログラム 5 種に対する各分類の見積結果は表 14 となった．

表 7 origin 分類の変数結果

変数名	出現回数	結果
arithmetic	5096734	-2.31523
float	6037294	-0.919885
load	14912172	2.39813
store	5291153	4.20962
others	21279085	1.57163

表 8 RH850 分類の変数結果

変数名	出現回数	結果
arithmetic	3787427	-5.33061
div	1309307	-3.94779
float	5027608	-12.0627
fdiv	1009686	24.5291
load	14912172	-1.66483
store	5291153	10.8402
callret	118382	3.59353
others	21161003	4.49605

表 9 operand 分類の変数結果

変数名	出現回数	結果
arithmetic	380357	-5.66667
div	21	3493.62
arithmetic_d	3407070	-11.3863
div_d	1309286	3.23792
float	1009445	-19.7286
fdiv	95	97.1356
float_d	4018163	-6.75864
fdiv_d	1009591	26.2952
load	3125670	2.2746
load_d	11786502	-9.46369
store	5325309	19.2001
callret	118382	0.132956
others	21161003	5.76572

表 10 mem 分類の変数結果

変数名	出現回数	結果
arithmetic	380357	-0.566965
div	21	4413.04
arithmetic_d	3407070	-2.57112
div_d	1309286	-3.41529
float	1009445	-31.0224
fdiv	95	79.4298
float_d	4018163	-4.46409
fdiv_d	1009591	32.5699
callret	118382	2.58379
others	21161003	3.00336
L1Access	25646774	1.29356
L2Access	2169.662621	-2581.73
memAccess	299.4673132	4852.49

表 11 L2 分類の変数結果

変数名	出現回数	結果
arithmetic	380357	-6.55964
div	21	2831.29
arithmetic_d	3407070	1.06102
div_d	1309286	-0.241797
float	1009445	-27.5794
fdiv	95	82.6498
float_d	4018163	-0.701167
fdiv_d	1009591	21.0866
callret	118382	7.08065
others	21161003	2.33114
L1Access	25646774	0.696007
L2Access	2169.662621	1738.49

表 12 L1 分類の変数結果

変数名	出現回数	結果
arithmetic	380357	-6.36335
div	21	4296.6
arithmetic_d	3407070	-4.63254
div_d	1309286	-4.70536
float	1009445	-36.33
fdiv	95	80.8465
float_d	4018163	-3.59042
fdiv_d	1009591	29.8441
callret	118382	6.3432
others	21161003	3.64502
L1Access	25646774	1.16774

表 13 サンプルプログラム 20 種への見積

分類名	誤差平均	誤差絶対値平均
origin	-10.2%	25.1%
RH850	-5.9%	18.3%
operand	-0.8%	18.5%
mem	-1.3%	9.3%
L2	5.4%	11.4%
L1	-1.6%	10.8%

表 14 テストプログラム 5 種への見積

分類	誤差平均	誤差絶対値平均
origin	-73.8%	73.8%
RH850	-75.0%	75.0%
operand	905.0%	980.7%
mem	1343.2%	1363.0%
L2	1894.2%	1894.2%
L1	1209.3%	1253.3%

変数の結果では，マイナスの値や，大きすぎる値など 1 命令のサイクル数としては不適切な値が散見された．この原因としては，次のようなことが考えられる．

- 変数内の誤差が大きい

表 1 の分類など，大まかな分類のときは，1 つの変数内に様々なサイクルの命令が入っており，それらを正の値 1 つで表すことができず，マイナスの値が出現していると考えられる

- 変数の出現回数が少ない
  operand 分類以降の，細かい分類を始めたところ，1000 を超えるような大きな値が出現していることがわかる．そして，div など値が大きくなっている変数の出現回数を見ると，他の変数に比べて出現回数が少なくなっており，これが原因で大きな値となってしまっていると考えられる．

次に見積精度について見ると，回帰分析に利用したサンプルプログラムへの見積精度は，比較的高いが，回帰分析に利用していないプログラムへの見積は精度が低くなっている．また，分類の粒度が小さいものほど回帰分析に利用したサンプルプログラムへの見積精度が高いが，回帰分析に利用していないプログラムへの見積は精度は低くなっていることがわかる．粒度が小さいほどサンプルへの精度が高くなっていることは，分類を細かくした成果の表れであり，期待通りの結果である．しかし，サンプル以外のプログラムへの精度の低下は期待と異なる結果である．この原因を調査したところ，サンプル数が足りず，不適切な値となってしまっている変数が見積対象内に出現した結果，大きな誤差を生み出していることがわかった．よって，十分なサンプル数を確保することで，変数の値をサイクル数として適切なものとなり，それに伴いサンプル以外への見積精度も向上すると考えられる．

5. おわりに

本研究では，モデルベース並列化において，モデル設計段階でソフトウェアの性能を見積もることができる見積手法である SHIM 性能見積で用いられる，SHIM の性能値を計測する手法の提案を行った．先行研究での SHIM 計測の課題であった，ターゲットハードウェアごとに異なる計測プログラムを用意しなければならないという課題の解決のために，回帰分析を用いた SHIM 計測手法を行った．そして，回帰分析における変数に LLVM-IR 命令を割り当てる変数設定の手法を 4 つ提案し，Raspberry Pi3 Model B+ をターゲットハードウェアとして，それらの設定に対する精度評価を行った．その結果，変数の割り当てを細かくしたほうが，回帰分析の精度は向上するが，サンプル数が足りなかったため，変数の値としては不適切でない値が出現した．

6. 今後の課題

SHIM 計測手法に関しては，サンプルプログラムを増やし，出現数が少ない命令がなくなるようにする必要がある．また，現在のサンプルプログラムが，数値計算のプログラムに偏ってしまっているため，それ以外のプログラムを増やす必要がある．具体的なサンプルプログラムを増やす方法としては，ベンチマークの利用などが考えられる．また，変数設定の見直しを行っていく必要もある．今回は，変数の数を抑えるために，多くの命令を others としたが，今後サンプルプログラムの数が確保できれば，others の細分化を考慮していくべきである．また逆に，今回 div の出現数が少なかったが，サンプル数を増やしても，出現数が増えないような変数があれば，その変数の精度は見積に大きな影響を与えないということであるため，others など他の変数とまとめることも考えられる．

参考文献

[1] AUTO PROVE Change information into Knowledge ,2020, https://autoprove.net/supplier_news/dspace/30169/
[2] eSOL eMBP(Model Based Parallelizer),2020, https://www.esol.co.jp/embedded/mbp.html
[3] Software-Hardware Interface for Multi-Many-Core(SHIM),2020, https://www.multicore-association.org/workgroup/shim.php
[4] The Multicore Association,2020, https://www.multicore-association.org/
[5] The LLVM Compiler Infrastructure,2020, http://llvm.org/
[6] Clang: a C language family frontend for LLVM,2020, https://clang.llvm.org/
[7] LLVM Language Reference Manual,2020, https://llvm.org/docs/LangRef.html
[8] 佐合惇，枝廣正人，ハードウェア抽象化記述 SHIM と SHIMulator によるソフトウェア動的性能見積手法，組込み技術とネットワークに関するワークショップ（ETNET2019），2019.