Vol.2009-ARC-183 No.7
Vol.2009-OS-111 No.7
2009/4/22

移植性の高い最悪実行時間予測ツール RETAS の設計と実装

山本啓二[†1]　石川裕[†1]　松井俊浩[†2]

信頼性の高い実時間システムを構築するためには，実時間システム上で動く実時間タスクの最悪実行時間を見積もり，それがデッドラインを満たすことを保証することが重要である．本研究では，様々なアーキテクチャへの移植に優れた最悪実行時間予測ツールである RETAS の設計と実装について述べる．RETAS は，タスクの実行時間をメモリアクセス時間とメモリアクセスを除いた命令実行時間に分けて計算する．メモリアクセス時間は中間表現を解釈し実行できるシミュレーターを使って求める．メモリアクセスを除いた命令実行時間は，実機上でコードを部分的に実行し，その時間を計測して求める．提案手法を，Pentium-M, XScale, SH アーキテクチャ上で実装し評価する．結果，どのアーキテクチャでも安全に最悪実行時間を予測できることを示す．

Design and Implementation of Portable Worst-Case Execution Time Analysis Tool: RETAS

Keiji Yamamoto ,[†1] Yutaka Ishikawa [†1]
and Toshihiro Matsui[†2]

To design a reliable real-time system, it is important to know the worst-case execution time of a real-time task, and to confirm whether it satisfies deadline. In this paper, we propose a new portable worst-case execution time analysis tool named RETAS. Execution time is predicted by combining the partial execution of the code and memory access time calculated using a simulator. We demonstrate that RETAS predicts the execution time safely in different environments, Pentium-M, XScale and SH.

1. はじめに

信頼性の高い実時間システムを構築するためには，その上で動く実時間タスクがデッドラインを必ず満たすことを保証することが重要である．通常，実時間タスクには条件分岐によって様々な実行パスが存在する．デッドラインを満たすことを保証するには，最長となる実行パスを見つけ最悪実行時間（Worst-Case Execution Time:WCET）を見積もる必要がある．

一般には，様々な条件の下で実時間タスクの実行を繰り返し，その実行時間を測定し統計的に WCET を見積もっている．しかし，このような手法では真に最長となる実行パスを通って得られた WCET であることを保証できない．そこで，タスクのコードを解析して実行可能経路を求め WCET を予測する静的最悪実行時間予測手法が研究されている．静的最悪実行時間予測手法は，実行パスやループ回数などのフロー情報をコードから求めるフロー解析部分と，命令実行時間やメモリアクセス時間などのアーキテクチャに依存した要素を解析する部分に分かれている．

本研究では，移植性を考慮し，複数のアーキテクチャで容易に利用可能な最悪実行時間予測手法について述べる．我々が従来開発してきた最悪実行時間予測ツール RETAS[1] について，移植性において不十分であった機構について再設計を行なう．アーキテクチャ依存部分と非依存部分を切り分け，アーキテクチャ依存部分を小さくすることで移植性を高める．Intel 社の Pentium-M[2] や XScale[3]，Renesas 社の SH[4] アーキテクチャ向けに提案手法を実装し移植コストについて述べる．また，ベンチマークプログラムの実行時間を予測し実測値との比較を行ない，提案手法で安全に最悪実行時間が予測できることを示す．

2. 静的最悪実行時間予測

一般に静的最悪実行時間予測では，プログラムを解析して実行可能経路を求め，その経路に対するプログラムの実行時間を計算し最悪値を求める．実行可能経路は，ソースコードや実行ファイル（アセンブリコード）を解析して求める．解析のターゲットとするコードを高級言語か低級言語にするかによって解析精度や実装コストや移植性が変わる．実行時間は，

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology (AIST)

© 2009 Information Processing Society of Japan

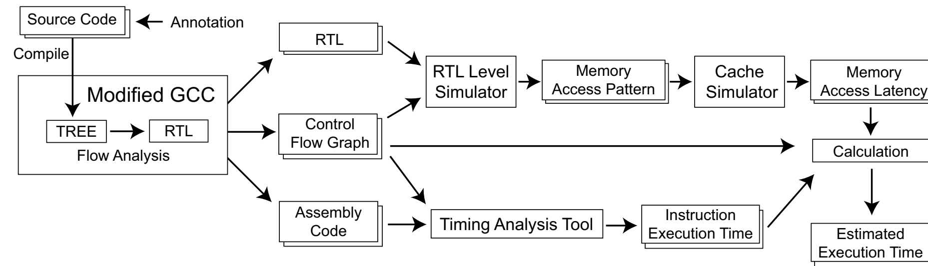

図 1 (旧) 最悪実行時間予測ツール RETAS

プロセッサのパイプラインモデルやキャッシュモデルを用いて予測を行なう．モデル自体の精度が最終的な予測精度に繋がるため，モデルの実装者はプロセッサの内部動作に精通している必要がある．実装されたモデルは特定のアーキテクチャに強く依存しているため，他のアーキテクチャへの移植コストは非常に大きい．

2.1 旧 RETAS の構成

RETAS[1] は幅広いアーキテクチャへの移植を考え設計と実装を行なっている最悪実行時間予測ツールである．旧 RETAS の構成を**図 1** に示す．RETAS はコンパイラの中間表現と，実機を用いた実行時間の測定結果を利用して最悪実行時間を求める．中間表現の利用により，アーキテクチャ毎に異なるアセンブリコードを個別に解析する解析器を用意する必要はない．中間表現を解析する解析器をひとつ実装するだけでよい．命令の実行時間は，メモリアクセス時間とメモリアクセスを除いた命令実行時間に分けて考える．メモリアクセス時間は中間表現をシミュレーター上で実行しメモリアクセスパターンを求め，そのパターンからキャッシュへのヒット回数やメインメモリへのアクセス回数を予測する．メモリアクセスを除いた命令実行時間は，実機を用いてメモリアクセスを除いたコードの実行時間を測定することでアーキテクチャのモデル化を行なわずに実行時間を求める．この測定は基本ブロック単位に行なう．

シミュレーターは中間表現を解釈し実行するため特定のアーキテクチャに依存せずにメモリアクセスパターンを求めることができる．アクセスパターンからキャッシュへのヒット回数を求めるために，ターゲットアーキテクチャのキャッシュアルゴリズムやキャッシュサイズ等を元にキャッシュのシミュレーションを行なう．このキャッシュシミュレーターはキャッシュアルゴリズムごとに実装が必要である．ただし，Pentium や XScale や SH 等の多くのアーキテクチャはセットアソシアティブ方式で置き換えアルゴリズムに Least Recently Used(LRU) または Round Robin(RR) を用いるため，キャッシュサイズなどのパラメータの変更で対応できる．

コードの実行時間を測定するには，測定する基本ブロックをアセンブリコードから切りだし，ブロックの前後に時間測定用のコードを挿入する必要がある．基本ブロック内に含まれるメモリアクセス命令は，そのまま実行するとセグメンテーションフォールトが発生するため実行しても安全な命令へと変更する．アセンブリコードから基本ブロックを容易に切り出すために，コンパイラを修正しアセンブリコード内に基本ブロックの切れ目となる印を挿入する．

メモリアクセス命令の変更には，どの命令がメモリアクセス命令なのかといったアーキテクチャに依存する知識が必要である．また，時間測定用のコードについてもどのようなコードで時間が測定できるか知っている必要がある．

図 1 の旧 RETAS[5)1)] では，コンパイラに Gnu Compiler Collection (GCC) を用いて，GCC 内部でフロー解析を行ないフロー情報と GCC の中間表現である Register Transfer Language (RTL)[6] を出力していた．またメモリアクセス命令実行時のレイテンシを予測するため，`RTL Level Simulator` を使ってメモリアクセストレースを求め，その結果を `Cache Simulator` で計算しキャッシュのヒット回数やメインメモリへのアクセス回数を求めていた．

旧 RETAS では GCC を修正するコストが大きいという問題点があった．GCC のバージョンアップに伴い RETAS の GCC に対するパッチも随時修正する必要がある．実際に，GCC 3.3.3 に対する実装[7] から GCC 4.0.2 に対する実装[1] の際にはメジャーバージョンアップの影響もあるが移植に時間を要した．また，`RTL Level Simulator` と `Cache Simulator` はデータキャッシュが実行時間に与える影響を求める目的で設計しているため，命令キャッシュや分岐予測器の影響を予測するにはシミュレーターの再設計が必要である．

3. 設 計

従来の問題点をふまえて本研究で提案する新 RETAS を**図 2** に示す．破線部分が従来の RETAS との変更部分である．大きな違いは，アノテーション解析およびフロー解析を GCC の外部で行なうようにすること及び `RTL Level Simulator` への機能統合である．ここでは変更部分のみについて述べる．

3.1 フロー解析

フロー解析器は 7) で述べた機能を GCC の外部に実装する．GCC へはアセンブリコード出力部を変更し，RTL の出力機構を加える修正を行なう．これは GCC のバージョンアッ

© 2009 Information Processing Society of Japan

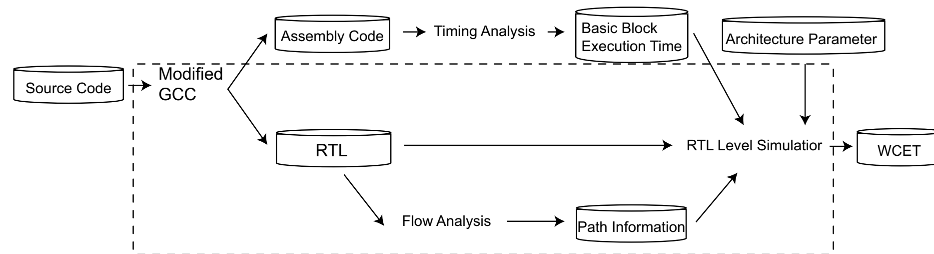

図 2 最悪実行時間予測ツール RETAS

プへ容易に追随するためである．アセンブリコード出力部は，基本ブロックの境界にコメントを挿入するように修正する．RTL の出力フォーマットは，新たに実装するフロー解析器が RTL をパースしなくても扱えるようにするため JSON[8] を用いる．

3.2 アノテーション

フロー解析を GCC の外部で行なうことにより，アノテーションの記述および解析方法も変更する．旧 RETAS[5] では，アノテーションは pragma を用いて記述し，これら pragma は RETAS のプリプロセッサで特別な関数呼び出しへと置き換えていた．フロー解析の過程で，これら特別な関数呼び出しを発見すると，関数名およびパラメータである引数を抽出しアノテーションとして扱っていた．フロー解析を GCC の外部で行なう場合は，これらの特別な関数呼び出しがコードに残ってしまうという問題がある．

新 RETAS では，アノテーションを特別な関数呼び出しへ置き換える代わりに，インラインアセンブリとして記述する．たとえば，以下のコードは for ループの繰り返し回数が最大で 16 回であることを示している．

```
for (c = 0; c < N; c++) {
  asm("# retas iteration 16");
  function();
}
```

アノテーションはアセンブリコードのコメントとして記述する．インラインアセンブリはコンパイル過程で RTL にそのままのアセンブリコードで表現される．よって，GCC の外部でもアノテーションの抽出が可能である．

3.3 RTL Level Simulator

旧 RETAS の `RTL Level Simulator` は RTL をパース，実行しメモリアクセストレースを出力するシミュレーターであった．メモリアクセスレイテンシは，得られたメモリアクセストレースを元に `Cache Simulator` を動かし求めていた．新 RETAS では，`RTL Level Simulator` に `Cache Simulator` の機構も含め，`RTL Level Simulator` でデータキャッシュ以外に命令キャッシュや分岐予測器の影響を予測できるようにする．

4. 実 装

4.1 アーキテクチャ非依存部分

ここでは，アーキテクチャに非依存な部分の実装について述べる．アーキテクチャに非依存なのは次の部分である．

- GCC の修正
- フロー解析
- RTL Level Simulator

GCC 4.1.1 に対して RTL 出力機構およびアセンブリコード出力部の修正を実装する．GCC 内部でフロー解析を行なう場合には GCC 4.0.2 に対して約 2500 行の修正を要した．新 RETAS の実装では GCC 4.1.1 に対し約 700 行の修正となっている．外部に実装したフロー解析器は python で記述し，コード行数は約 1000 行である．`RTL Level Simulator` は python で実装し，コードは約 900 行である．

4.2 アーキテクチャ依存部分

ここでは，個々のアーキテクチャ毎に実装が必要な部分について述べる．アーキテクチャに依存しているのは主に次の部分である．

- キャッシュメモリ機構
- メインメモリの特性
- 基本ブロックの時間測定手法

4.2.1 キャッシュメモリ機構

キャッシュやメインメモリの特性はパラメータの変更で対応できる．キャッシュに関してはプロセッサのデータシートを参照し，キャッシュサイズやラインサイズ等をパラメータとして `RTL Level Simulator` を動かす．キャッシュミス時にはメインメモリへのアクセスが発生する．メインメモリのアクセスレイテンシはチップセットやメモリの特性によって変化するため，計測によって求める．

4.2.2 メインメモリのアクセスレイテンシ

メインメモリのアクセスにかかるレイテンシは，**図 3** に示すコードを使って測定してい

© 2009 Information Processing Society of Japan

```
 1 p = memory_allocation();
 2 sequential_memory_access(p);
 3 clear_cache();
 4 for (c = 0; c < TRIAL; c++) {
 5   i = rand();
 6   tsc0 = read_time_stamp_counter();
 7   temp = p[i];
 8   tsc1 = read_time_stamp_counter();
 9   tsc[c] = tsc1 - tsc0;
10 }
11 print_result(tsc);
```

図 3 メモリレイテンシ測定用コード

る．まず，ある程度のアドレス空間を確保し，そのアドレス空間全体にアクセスする．次にキャッシュサイズ以上の別のアドレス空間を確保し，そこにアクセスすることでキャッシュをクリアする．乱数を使ってアクセスするメモリアドレスを決定し，メモリアクセスの前後にタイムスタンプカウンターを取得する命令を実行しメモリアクセス命令の実行時間を得る．乱数を使用するのは，シーケンシャルにアクセスする場合のメモリプリフェッチ等の影響を避けるためである．前後のタイムスタンプカウンターの単純な差分には，タイムスタンプカウンター命令自身の実行時間も含まれているため考慮する必要がある．

4.2.3 基本ブロックの時間測定手法

基本ブロックの時間測定を行なうには，クロック単位での時間測定と基本ブロック内の一連の命令を実行できる環境が必要である．クロック単位での時間測定は，近年のアーキテクチャであればパフォーマンスカウンターがプロセッサに備わっているためこれをタイムスタンプカウンターとして利用できる．

基本ブロック内の命令にはメモリアクセス命令等が含まれているため，一つの基本ブロックを単純に実行しようとすると無効なアドレスへのメモリアクセスが発生し実行時間を測定することができない．よって基本ブロック内のアセンブリを無効にしたり書き変えたりして基本ブロックのみでも実行できるコードに変換する．無効にする命令には，ジャンプ命令やスタック操作命令がある．時間測定中にジャンプやスタックを操作されると基本ブロックから抜けれなかったり，基本ブロックから抜けた後のスタックポインタが不正な値となり時間測定ができないためである．書き変える命令には，メモリアクセス命令，特定のレジスタの操作命令等がある．該当する命令をプロセッサのデータシートを参照し，適切に処理する必要がある．

4.2.4 Pentium-M アーキテクチャ

Pentium-M[2] アーキテクチャでは `rdtsc` 命令を使ってタイムスタンプカウンターを取得することができる．`rdtsc` 命令を発行するために必要なコード行数は数行である．

基本ブロックの実行時間を測定するには，基本ブロックの実行前に，スタックの確保，浮動小数点ユニットの初期化，浮動小数点レジスタへの値の代入を行なう．その後，タイムスタンプカウンターを取得し，基本ブロックを実行する．メモリへのアクセスは，スタック以外の場合には無効なアドレスにアクセスすることになるため，レジスタアクセスとグローバル変数へのアクセスに置き換える．スタックポインタ (`esp,ebp`) への代入は他のレジスタ (`eax,ebx` 等) に置き換える．これらは python で実装しておりコード行数は約 200 行となっている．

浮動小数点命令の実行時間は 1) ではデータシートから得た命令とサイクル数の表を元に実行時間を決定していた．本研究では，基本ブロック実行前に浮動小数点レジスタに値をあらかじめ設定しておくため，実行時間の測定が可能となっている．

4.2.5 XScale アーキテクチャ

XScale[3] には命令の実行回数やサイクル数等のパフォーマンス測定用のレジスタ，`Performance Monitoring Register` があり，これを設定することでタイムスタンプカウンターを取得できる．ただし，Pentium-M アーキテクチャと違ってレジスタへのアクセスが特権モードに限られるため OS にこれらレジスタを操作する命令を追加する必要がある．本実装では，`Performance Monitoring Register` を操作するシステムコールを XScale の評価環境である Linux 2.4.20 に追加する．コード行数は約 40 行である．また，ジャンプ命令やスタック操作命令を無効化するのに必要なコード行数は約 160 行である．

4.2.6 SH アーキテクチャ

本研究でターゲットとした SH7780[4] には XScale と同様のパフォーマンス測定用のレジスタがあり，これを用いてタイムスタンプカウンターを取得できる．レジスタへのアクセスは XScale 同様に特権モードに限られるため，OS にこれらレジスタを操作する命令を追加する必要がある．本実装では，パフォーマンス測定用レジスタを操作する専用のデバイスをカーネルモジュールとして作成する．ユーザーレベルからは専用デバイスへの `ioctl` を使用してタイムスタンプカウンターを取得する．XScale の場合と異なりカーネルモジュールとして実装するため，カーネルの再構築の手間はない．このデバイスのコード行数は約 100 行となっている．また，ジャンプ命令やスタック操作命令を無効化するのに必要なコード行

© 2009 Information Processing Society of Japan

表 1 Pentium-M Architecture

プロセッサ	Pentium-M 1.6GHz
L1 キャッシュ/レイテンシ	32KB/ 3 cycle
L2 キャッシュ/レイテンシ	1024KB/ 9 cycle
データキャッシュ	8 way Set Associative
キャッシュ置き換えアルゴリズム	LRU
キャッシュラインサイズ	64 bytes

表 2 XScale Architecture

プロセッサ	XScale PXA270 416MHz
Cache	32KB + 32KB Inst/Data
Instruction Cache	32 way set associative
Data Cache	32 way set associative
Replacement policy	Round-Robin
Line Size	32 bytes

表 3 SH Architecture

プロセッサ	SH7780 400MHz
Cache	32KB + 32KB Inst/Data
Instruction Cache	4 way set associative
Data Cache	4 way set associative
Replacement policy	LRU
Line Size	32 bytes

数は約 190 行である．

5. 評　　価

提案手法をベンチマークプログラムを用いて評価する．評価環境として Pentium-M，XScale，SH アーキテクチャを用い，それぞれのアーキテクチャの詳細を**表 1**, **表 2**, **表 3** に示す．ベンチマークプログラムの詳細を**表 4** に示す．

5.1 実測値の測定

RETAS で求める予測値の精度を調べるため，ベンチマークタスクが最悪な実行パスを通る時の実行時間を測定する．ベンチマークプログラムの前後でタイムスタンプカウンターを読み実行時間を測定すると，OS の処理時間 (プログラムのロード時間や，メモリアクセス時のページ割り当ての時間) も測定結果に含まれる．RETAS はこのような OS の処理時間は無視しているため，OS の処理以外のベンチマークプログラムの実行時間を計測する必要がある．

表 4 ベンチマークの内容

プログラム名	処理内容
Matmul	16 x 16 整数値の行列乗算 3 重ループ
Fibonacci	フィボナッチ数 30
Servo	ヒューマノイドロボット HRP2[9] で使われているセンサー・サーボループプログラム HRP2 に置けるハード・リアルタイム処理部分
Stereo	左右のカメラ入力から得られるステレオ画像から距離画像を生成するプログラム ロボットの行動計画時に必要となる処理部分

```
 1 memory_allocation();
 2 benchmark();
 3 for (c = 0; c < 100; c++) {
 4   clear_cache();
 5   tsc0 = read_time_stamp_counter();
 6   tsc0 = read_time_stamp_counter();
 7   benchmark();
 8   tsc1 = read_time_stamp_counter();
 9   tsc[c] = tsc1 - tsc0;
10 }
11 print_result(tsc);
```

図 4 実測値の測定用コード

図 4 に実測値の測定プログラムを示す．まずベンチマークプログラム内で使用されるメモリの割り当てを行なう．次に、ベンチマークプログラムを一通り動かして OS のページ割り当てを実行する．キャッシュメモリのサイズ以上の別のアドレス空間をアクセスすることによって，ベンチマークを動かした際のキャッシュをクリアする．何度かタイムスタンプカウンターの値を取得する．何度か実行するのはタイムスタンプカウンターを取得する関数自体のキャッシュミス時間の影響を避けるためである．ベンチマークの計測結果を printf 等で逐次出力していると入出力の処理で実行時間が乱れるため，ベンチマークが終わるまで結果は出力しない．ここでは 100 回の計測を行なった最悪値を実測値として採用している．

© 2009 Information Processing Society of Japan

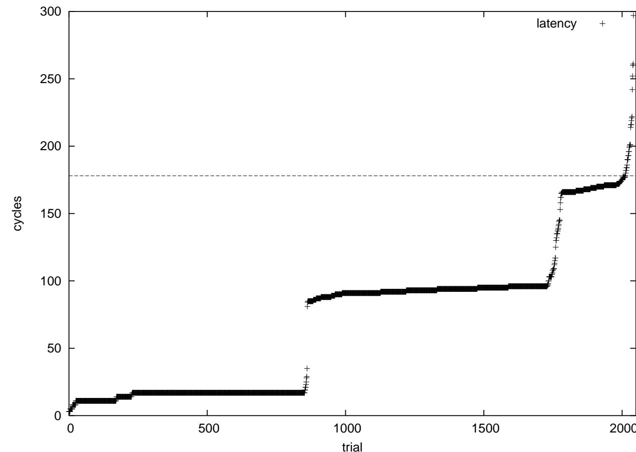

図 5 Pentium-M メモリアクセスレイテンシ

表 5 メモリアクセスレイテンシの測定結果

アーキテクチャ	レイテンシ (cycles)
Pentium-M	180
XScale	121
SH7780	80

5.2 メインメモリのアクセスレイテンシ

メインメモリのアクセスレイテンシ以外の値はプロセッサーのデータシートを参照している．メインメモリのアクセスレイテンシは評価環境でメインメモリのアクセス時間を測定した値を用いている．メインメモリのアクセス時間はバスのタイミングによってばらつくため常に一定の値とはならない．ここでは，最悪実行時間を求める観点からメインメモリアクセス時間の測定値から予測される最悪値を用いている．**図 5** は Pentium-M プロセッサの場合にメモリアクセス時間を計測したものである．図 5 は 2048 回メモリアクセスを行なったときの時間を短い順にプロットしたものである．図 5 より，メモリアクセスレイテンシは 180 サイクル程度であるとわかる．**表 5** に評価で用いたアーキテクチャのメモリアクセスレイテンシの測定結果を示す.

表 6 ベンチマークプログラムの予測結果と最悪値

アーキテクチャ	ベンチマーク	実測値 (cycles)	予測値 (cycles)	誤差
Pentium-M	Matmul	28398	32507	+14%
	Fibonacci	146	164	+12%
	Servo	23372	31857	+36%
	Stereo	131.51 M	165.71 M	+26%
XScale	Matmul	48184	51277	+6%
	Fibonacci	190	204	+7%
	Servo	1526 K	1803 K	+18%
	Stereo	297.78 M	316.72 M	+6%
SH	Matmul	51307	54604	+6%
	Fibonacci	220	252	+14%
	Servo	34187	38728	+13%
	Stereo	336.89 M	342.81 M	+2%

5.3 ベンチマークプログラムの予測結果

ベンチマークプログラムの実測値と予測値の結果を**表 6** に示す．誤差を見ると全てのアーキテクチャとベンチマークにおいてプラス誤差となっているため安全に予測できていることがわかる.

XScale の Servo タスクは他のアーキテクチャに比べて数十倍の実行サイクルとなっている．これは XScale に浮動小数点ユニットが無いため Servo タスクの浮動小数点命令をエミュレートしているためである.

6. 関連研究

WCET 予測は様々な要素技術の組み合わせで求められる．これら要素技術は大きくフロー解析と実行時間解析の 2 つに分けられる.

フロー解析の目的は，プログラムのすべての実行可能パスを求めることである．実行可能パスを求めるには，アノテーションを利用する手法とソースコードを自動解析する手法がある．アノテーションはプログラマ自身がループ回数や分岐情報などのフロー情報を記述する手法[10] である．自動解析はソースコードの条件文やループ構造を解析しアノテーションに頼らずに自動的にループ回数等を求める手法[11][12] である.

アセンブリ言語をフロー解析する場合は，言語自体がアーキテクチャに依存しているため解析器の移植は困難である．高級言語を解析する場合は，コンパイラの最適化が考慮されないため実際の実行フローとは異なる可能性がある．本研究ではコンパイラの中間表現を用い

ⓒ 2009 Information Processing Society of Japan

ることで，コンパイラの最適化に対応し，また移植性にも優れている．

実行時間解析の目的は，ターゲットアーキテクチャ上である命令列が実行される時間を求めることである．このため，プロセッサを構成する様々な機構のモデルについて研究されている．例えば，命令キャッシュ[13)14)]やデータキャッシュ[15)16)]，分岐予測機構[17)]，パイプライン[18)19)]などのモデルがある．一方，プロセッサの内部動作に関する詳細な情報は公開されていないことが多いため，正確なモデルを作成することが困難であるという問題がある．また，個々のモデルはアーキテクチャに強く依存しているため移植は困難である．

モデルを用いずに実行時間を解析する手法として，実際にコードを実機で実行し，実行時間を計測する手法が挙げられる．モデルを用いて考えていたプロセッサの様々な機構は，実機での実行によりすべて計測結果に含まれる．ただし，単純に計測するだけではブラックボックステストになるため最悪値を保証することができない．本研究では，実行時間を命令実行時間とメモリアクセス時間に分けて考えることで，実行フローを無視して命令実行時間を計測することや，シミュレーターを利用してメモリアクセスレイテンシを求めることが可能である．モデルを用いず計測を行なうため，移植性にも優れている．

Petters ら[20)]はコンパイラの生成すフローグラフを利用して実行不可能な経路を削減して実行回数を減らし，実行時間を測定する手法を提案している．Wenzel ら[21)]はモデル検査手法を利用してテストデータを自動生成し，様々な経路の実行時間計測を可能としている．

最悪実行時間を予測する製品として RapiTime[22)] や aiT[23)] がある．RapiTime は様々な実行経路でプログラムを動かして実行時間を測定し，コードのカバレッジを元に最悪実行時間を計算するツールである．精度を上げるにはカバレッジを大きくする十分なテストデータが必要である．aiT はパイプラインやキャッシュ等のアーキテクチャモデルを元に実行時間を予測するツールである．特定のアーキテクチャに依存しているため広く利用することはできない．

7. まとめ

本研究では，移植性の高い最悪実行時間予測ツール RETAS の設計について述べ，実際に Pentium-M，XScale，SH アーキテクチャ上に移植し予測精度と移植性の評価を行なった．実行時間予測ツールのアーキテクチャ依存部分と非依存部分を切り分け，依存部分を小さくすることによって移植性を高めている．依存部分を小さくするため，アーキテクチャに特有の命令の実行時間については，アーキテクチャのモデル化を行なわず実行時間の測定によって求めている．またキャッシュアルゴリズム等もアーキテクチャに依存する部分ではあるが，代表的なキャッシュアルゴリズムをあらかじめ実装しておくことで，アーキテクチャの違いをパラメータの変更のみで対応している．

移植に必要なコストは，XScale に対応させる場合に約 200 行のコード量，SH に対応させる場合に約 300 行のコード量である．予測精度については，Pentium-M アーキテクチャが他のアーキテクチャに比べて精度が悪いものの，誤差がプラス方向のため安全に予測できている．Pentium-M の予測誤差は，メインメモリへのアクセス時間が大きくばらついていることが原因である．

謝辞　本研究の一部は，科学技術振興機構 (JST) の戦略的創造研究推進事業 (CREST) の支援を受けた．

参考文献

1) Yamamoto, K., Ishikawa, Y. and Matsui, T.: Portable Execution Time Analysis Method, *the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA06)*, pp.267–270 (2006).
2) Intel Corporation: *IA-32 Architecture Software Developer's Manual* (2004).
3) Intel Corporation: *Intel XScale Core Developer's Manual* (2004).
4) Renesas Technology: SH7780 ハードウェアマニュアル (2006).
5) 山本啓二，石川　裕，松井俊浩：移植性の高い実行時間予測手法の設計と実装，情報処理学会研究報告 (ARC-169, SWoPP 2006)，pp.127–132 (2006).
6) Free Software Foundation: GNU Compiler Collection Internals, `http://gcc.gnu.org/onlinedocs/gccint/`.
7) 山本啓二，石川　裕，松井俊浩：実行時間予測ツールの設計と実装，情報処理学会研究報告 (ARC-164, SWoPP 2005)，pp.79–84 (2005).
8) Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON), Internet RFC 4627.
9) 松井俊浩，比留川博久，石川　裕，山崎信行，加賀美聡，堀　俊夫，金広文男，斎藤　元，稲邑哲也：ヒューマノイド・ロボットのための実時間分散情報処理，情報処理学会研究報告. SLDM, [システム LSI 設計技術]， Vol.2004, No.33, pp.1–7 (2004).
10) Kirner, R. and Puschner, P.: Transformation of Path Information for WCET Analysis during Compilation, *Proc. 13th Euromicro International Conference on real-Time Systems*, pp.29–36 (2001).
11) Healy, C., Södin, M., Rustagi, V. and Whalley, D.: Bounding Loop Iterations for Timing Analysis, *Proc. 4th Read-Time Technology and Applications Symp.*, pp. 12–21 (1998).
12) Gustafsson, J., Lisper, B., Snadberg, C. and Bermudo, N.: A Tool for Automatic Flow Analysis of C-programs for WCET Calculation, *Proc. 8th IEEE International*

© 2009 Information Processing Society of Japan

Workshop on Object-Oriented Real-Time Dependable Systems (2003).
13) Lim, S.-S., Bae, Y.H., Jang, G.T., Rhee, B.-D., Min, S.L., Park, C.Y., Shin, H., Park, K., Moon, S.-M. and Kim, C.S.: An Accurate Worst Case Timing Analysis for RISC Processors, *IEEE Trans. Softw. Eng.*, Vol.21, No.7, pp.593–604 (1995).
14) Ottosson, G. and Sjödin, M.: Worst-Case Execution Time Analysis for Modern Hardware Architectures, *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)* (1997).
15) White, R.T., Healy, C.A., Whalley, D.B., Mueller, F. and Harmon, M.G.: Timing Analysis for Data Caches and Set-Associative Caches, *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, IEEE Computer Society, p.192 (1997).
16) Lundqvist, T. and Stenström, P.: An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution, *Real-Time Syst.*, Vol.17, No.2-3, pp.183–207 (1999).
17) Colin, A. and Puaut, I.: Worst Case Execution Time Analysis for a Processor with Branch Prediction, *Real-Time Syst.*, Vol.18, No.2-3, pp.249–274 (2000).
18) Schneider, J. and Ferdinand, C.: Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation, *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, ACM Press, pp.35–44 (1999).
19) Stappert, F. and Altenbernd, P.: Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-time Programs, *J. Syst. Archit.*, Vol.46, No.4, pp. 339–355 (2000).
20) Petters, S. and Farber, G.: Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible (1999).
21) Wenzel, I., Rieder, B., Kirner, R. and Puschner, P.: Automatic Timing Model Generation by CFG Partitioning and Model Checking, *Proc. Conference on Design, Automation, and Test in Europe* (2005).
22) Rapita Systems Ltd.: RapiTime, `http://www.rapitasystems.com/rapitime`.
23) AbsInt: aiT WCET Analyzers, `http://www.absint.com/`.

© 2009 Information Processing Society of Japan