情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 1　77–87 (Mar. 2010)

高い移植性を持つ最悪実行時間解析手法

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．本論文では，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手法を提案し，それを実行時間予測ツール RETAS として実装する．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コンパイラの中間表現を解釈し実行するシミュレータを使って求める．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．提案手法の移植性を評価するため，Pentium-M, XScale，SH アーキテクチャに提案手法を実装する．移植に要したコード量はそれぞれ 200～300 行である．また，ベンチマークプログラムの実測値と RETAS の予測値とを比較した結果，+2 から +36%のプラス方向の誤差であることを示し，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．

Portable Worst-Case Execution Time Analysis Method

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 method. Based on this approach, an execution time analysis tool named RETAS is implemented. 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. Porting RETAS to those architectures requires about 100 to 200 code lines to describe architecture dependent features. Comparing with actual and predicted execution times in benchmark programs, the predicted execution times are from +2% to +36% errors against the actual execution times. The results show that RETAS safely predicts the worst case execution times in those CPU architectures.

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必要がある．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，最長の実行時間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て統計的に 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[1]．

静的最悪実行時間予測手法は，実行パスやループ回数等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等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．フロー解析部分は，高級言語を解析する場合にはコンパイラの最適化をどのように考慮するのかという問題があり，低級言語を解析する場合には解析器がアーキテクチャに依存しているため移植性に乏しいという問題がある．アーキテクチャ依存部分の解析には，対象となるアーキテクチャの詳細なモデルを実装する必要があり，精度を求めると実装コストが大きく移植性も乏しくなるという問題がある．

本論文では，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．コンパイラ内部の中間表現を利用することで，特定のアーキテクチャに依存せずにフロー解析を行う．また，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についてはアーキテクチャモデルを用いずに実機を使った計測により実行時間を予測する[2),3]．

評価として Intel 社の Pentium-M[4] や XScale[5]，Renesas 社の SH[6] 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．移植に際しては，各アーキテクチャともに 200～300 行のコード記述量で済むことを示す．また，ベンチマークプログラムの実行

†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)

© 2010 Information Processing Society of Japan

時間を予測し実測値との比較を行う．予測誤差は Pentium-M の場合に +14 から +36%で，XScale の場合に +6 から +18%で，SH の場合に +2 から +14%であることを示す．誤差がすべてプラス方向で，実測値より予測値がつねに大きいため，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. 設　　計

2.1 概　　要

本論文で設計，実装する最悪実行時間予測ツールを RETAS と呼ぶ．RETAS は C 言語のソースコードを入力として受け取り，解析の後にそのコードの最悪実行時間を出力する．RETAS はコンパイラ，シミュレータ，解析器から構成されるツールで，その動作概要を**図 1** に示す．RETAS 内部では 4 つのタスクが実行される．それぞれのタスクの動作を以下に示す．

(1) ソースコードを，修正した Gnu Compiler Collection（GCC）を用いてコンパイルし，アセンブリコードと Register Transfer Language（RTL）[7] と呼ばれる GCC 内部の中間表現を得る．

(2) RTL をフロー解析し，すべての実行可能パスを求める．

(3) アセンブリコードを，最後尾を別として内部に分岐を含まない命令列のブロックに分割する．このブロックを基本ブロックと呼ぶ．各基本ブロックが独立して実行できるようにコードを変換し，実機上で基本ブロックを実行する．この実行時間は，メモリアクセス遅延を考慮しない命令実行スループットである．2.4 節に詳細を述べる．

(4) (2) の実行可能パスをもとにプログラムの実行およびキャッシュの振舞いを `RTL Level Simulator` を用いてシミュレーションし，メモリアクセス時間を予測しながら，(3) の命令実行スループットとともに最悪実行時間を求める．アーキテクチャによってキャッシュの振舞いは異るため，キャッシュアルゴリズムやキャッシュサイズ等のアーキテクチャ情報をもとに `RTL Level Simulator` はキャッシュのシミュレーションを行う．2.5 節に詳細を述べる．

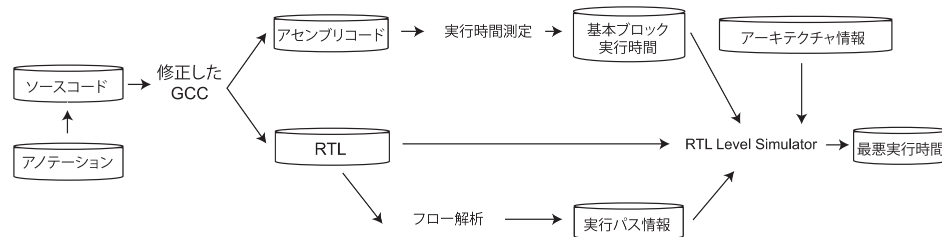


図 1　最悪実行時間予測ツール RETAS
Fig. 1　Worst-Case execution time analysis tool: RETAS.

命令の実行時間はアーキテクチャに強く依存している．RETAS は移植性を高めるために，命令の実行時間をメモリアクセスに要する時間 (4) と，メモリアクセスを除いた命令実行時間 (3) に分けて考えている．これにより，アーキテクチャのパイプラインモデルを実装することなく実行時間の予測が可能となっている．

2.2 フロー解析

静的最悪実行時間解析を行う場合には，コードを解析して実行フロー情報を求めることが必須である．コードを解析する場合には，C 言語等の高級言語を解析する手法とアセンブリ言語等の低級言語を解析する手法が考えられる．高級言語を解析する場合は，アーキテクチャ非依存で実行フローを求めることができる．しかし，コンパイラの最適化によってソースコードと実行コードとの間の対応関係が失われるため，正確な実行フローであるとはいえない．一方でコンパイラが生成する低級言語を解析する場合はコンパイラの最適化も考慮されているため，正確な実行フローを求めることができる．ただし，低級言語はアーキテクチャに依存しているため，解析器の移植性は乏しい．

RETAS は，中間表現を用いることで高い移植性を保持しつつ正確な実行フローを解析する．中間表現には Gnu Compiler Collection（GCC）で使用されている Register Transfer Language（RTL）[7] を用いる．GCC のコンパイル過程は，高級言語のソースコードを RTL へ変換し，その RTL に様々な最適化を施し，最終的に RTL をターゲットアーキテクチャのアセンブリコードに変換する．アセンブリコードに変換する直前の RTL は，コンパイラの最適化やレジスタ割付けが行われているため，アセンブリコードと RTL はほぼ等価の関係がある．アセンブリコードに変換する直前の RTL を GCC から取り出し，この中間表現に対してフロー解析を行う．中間表現の利用により，様々なアーキテクチャ上でコンパイラの最適化も考慮しつつフロー解析が可能であるため移植性は高い．

2.3 アノテーション

動的な実行フロー，たとえば実行時のデータの値によってループ回数が変化する場合は，フロー解析を行ってもループ回数が不明でフロー情報を求めることはできない．このような場合には，フロー情報をアノテーションとして記述できる機構が必要である．RETAS は，アノテーションをソースコード中に記述できる機構を持つ．アノテーションはアセンブリ

© 2010 Information Processing Society of Japan

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

図 2　アノテーションの挿入例
Fig. 2　Example of annotation.

```
func_basic_block:
    /* reserve stack area */
    /*
     * Code for BASIC BLOCK
     */
    /* release stack area */
    /* return instruction*/
```

図 3　基本ブロック実行関数の構造
Fig. 3　Structure of basic block execution function.

コードのコメントとして記述する．**図 2** のコードは for ループの繰返し回数が最大で 16 回であることを示している．

アノテーションは，

`# retas` コマンド　パラメータ...

のフォーマットで記述する．アノテーションを含んだソースコードを修正した GCC でコンパイルすると，RTL にもアノテーションがアセンブリコードとして含まれる．RETAS のフロー解析器は RTL を解析し，アノテーションコマンドに従ってパラメータを解釈し，フロー情報の補足や修正を行う．現在 RETAS が使うアノテーションコマンドを以下に示す．

iteration　このアノテーションが含まれるブロックの繰返し回数を設定する

cycle　このアノテーション通過時の実行時間の加算値を設定する．これは I/O の処理時間を設定する場合等に使う．

2.4　基本ブロック実行時間

基本ブロックは，関数として実行できるように**図 3** ようなコードを生成し，関数の実行時間を実機上で計測する．

測定用コードは，次の手順で生成される．

(1)　アセンブリコードを基本ブロックに分割
(2)　基本ブロック内のメモリアクセス命令，スタック命令，分岐命令を書き換え，計測用関数を生成
(3)　時間測定用コードを使って測定プログラムを生成

以下詳細に述べる．

2.4.1　基本ブロック切り出し

RETAS では GCC を修正し，アセンブリコードを出力する際に基本ブロックの境界にコメントを挿入する．これにより，ターゲットアーキテクチャのアセンブリ命令に関する知識なしに挿入したコメントをもとに基本ブロックへの分割が可能である．これにより移植性が高まっている．

2.4.2　基本ブロック内の命令置換

基本ブロックは 1 つの関数として定義されて実行される．関数を正常に実行できるように，メモリアクセス命令，スタック操作命令，分岐命令の 3 つに関して以下のとおり命令置換を行う．

基本ブロック単位に分割するとレジスタの初期値が不定となる．このため，レジスタ間接によるメモリの領域は不定となる．よって，実行中に必要とするメモリ領域を定義し，そのアドレスをレジスタに代入する命令を挿入する．

基本ブロック内でスタック操作が行われていると，基本ブロック実行後にスタックポインタが適切なメモリ領域を指されないことにより，呼び出し元に戻れなくなる可能性がある．そこで，スタック操作命令はロード/ストア系の命令に変換し，スタックポインタの値そのものは変更しないようにする．また，単純にスタックポインタの値を変更する命令は，別のレジスタの値を変更する命令に置換する．

これらの置換により基本ブロック内で参照されるメモリ領域は固定される．複数回基本ブロックを実行して実行時間を求めると，参照されるメモリはキャッシュ上に存在することになる．キャッシュの振舞いを含めたメモリアクセス遅延は，2.5 節で述べるシミュレーションにより求められる．実際の実行とは異なり同一メモリ領域を参照することになり，データハザードが生じる可能性がある．これは予測実行時間を増大させるが，最悪実行時間を求める観点では安全である．

分岐命令は基本ブロックの出口に現れる．これら分岐命令はコメントアウトする．分岐命令の実行に 1 クロックかかるとして，分岐命令が含まれている基本ブロックの実行時間は，基本ブロックの実行時間測定値に 1 クロック加算している．

本節で述べた命令置換あるいはコメントアウトはアーキテクチャ依存である．新しいアーキテクチャへの移植時には，アーキテクチャのデータシートから命令をリストアップし適切な置換方法を決める作業が必要となる．

© 2010 Information Processing Society of Japan

```
1 main () {
2   tsc0 = read_time_stamp_counter();
3   null_funcall();
4   tsc1 = read_time_stamp_counter();
5   null_funcall_cost = tsc1 - tsc0;
6   func_basic_block();
7   for (i = 0; i < COUNT; i++) {
8      tsc0 = read_time_stamp_counter();
9      func_basic_block();
10     tsc1 = read_time_stamp_counter();
11     func_cost[i] = tsc1 - tsc0;
12     func_cost[i] -= null_funcall_cost;
13  }
14  result(func_cost);
15 }
```

図 4　実行時間測定用のコード
Fig. 4　Execution time measurement program.

2.4.3　実行時間測定プログラム生成

実行時間測定プログラムは，おおむね**図 4** のようなプログラムとなる．

サイクル単位の正確な時間を取得するため，時間の測定には近年のプロセッサに搭載されているタイムスタンプカウンタ取得命令を用いる．関数が呼ばれたらすぐに戻る関数（null 関数）を作り，関数呼び出しの実行コスト $null_funcall_cost$ を計測しておく．上記の関数呼び出しを含んだ基本ブロックの計測時間を $func_cost$ とすると，基本ブロックの実行時間 T_b は

$$T_b = (func_cost - null_funcall_cost)/COUNT$$

で求まる．なお，時間測定用のコード生成はアーキテクチャに依存している．移植の際には，どのようなコードでタイムスタンプカウンタを取得できるかという知識が必要となる．

2.5　RTL Level Simulator

2.2 節で説明したフロー解析結果を用いて，キャッシュの振舞いを含めたメモリアクセスシミュレーションを行う．中間表現の RTL はアセンブリコードとほぼ等価な関係がある．よって，RETAS では移植性を考慮して RTL を解釈し実行するシミュレータである `RTL Level Simulator` を設計し実装する．図 1 に示すように `RTL Level Simulator` は，実行パス情報，RTL，基本ブロック実行時間，アーキテクチャ情報のパラメータを受け取り，その実行パスでの実行時間をシミュレーションによって求める．実行パス情報は，基本ブロックを実行する順序を表したものである．たとえ RTL 中に分岐命令が存在したとしても，与えられた実行パス情報に従って RTL を実行する．アーキテクチャ情報は，`RTL Level Simulator` がシミュレートするキャッシュアルゴリズムやキャッシュサイズ等のパラメータである．

`RTL Level Simulator` 単体では，特定のアーキテクチャに依存せずにインストラクションレベルで実行のシミュレーションが可能である．ここでは，機械語命令そのものは模擬せずにメモリのアクセスパターンによるメモリアクセス時間を求めていることに注意してほしい．いい換えれば `RTL Level Simulator` は機械語命令を模擬するものではない．

メモリアクセス時間には，メインメモリへのアクセス時間とキャッシュへのアクセス時間がある．キャッシュへのアクセス時間を求めるには，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もとにしたメモリアクセスのシミュレーションが必要で，これはアーキテクチャに依存する要素である．これらについては後述する．

`RTL Level Simulator` は，実行パスに従って，各基本ブロックのメモリアクセス時間を求め，2.4 節で求めた基本ブロックの実行時間を加算していく．同じ基本ブロックでも実行のパスによってメモリアクセス時間は異なるために，実行パスに従って基本ブロックのメモリアクセス時間を求めていく必要がある．`RTL Level Simulator` は，実行可能パスすべてについて実行時間を計算し，その最大値が最悪実行時間となる．

2.5.1　キャッシュアクセス時間

参照するメモリアドレスに依存してキャッシュの振舞いが変わる．参照されるメモリアドレスは，RTL レベルでは決定されていない．さらに，動的にメモリが確保されるようなプログラムでは，メモリアドレスは実行時に決定される．

シミュレーション中に生じたメモリアクセスは次のような方針でメモリアドレスを割り当ててキャッシュの振舞いを模擬する．あるレジスタの値が不定で，そのレジスタを使って間接メモリ参照するコードが現れた場合，そのレジスタには仮想的に 4 GB 境界となるメモリアドレスを割り当ててシミュレーションする．これにより，32 ビットアドレス空間を有するアーキテクチャでは，異なるメモリ領域を参照しているコードは必ず異なるメモリ領域を参照することになる[*1]．本手法では，実際には 2 つの整数変数が連続したメモリ上に割り当てられたことにより同一キャッシュブロック内に値が格納されるような状況を模擬できない．これにより実行時間を多めに予測してしまうが，最悪実行時間を求める観点では安全で

*1 64 ビットアドレス空間を有するアーキテクチャでも同様に異なるメモリ領域を参照しているコードは必ず異なるメモリ領域を参照するような仮想アドレス領域を設定すればよい．

© 2010 Information Processing Society of Japan

ある．

シミュレーションの結果，L1 キャッシュに保存されている場合には，L1 キャッシュアクセスコストは無視する．これは，2.4 節で述べた基本ブロック実行時間に含まれているからである．L1 以外のキャッシュを利用していると解釈される場合は，その実行コストがメモリアクセス時間として扱われる．これらキャッシュアクセス時間は通常データシートに示されている．RTL Level Simulator によって，データアクセス時にキャッシュからはずれていると判断するとメインメモリのアクセス時間がコストとして加算される．

Pentium や XScale や SH 等の多くのアーキテクチャではキャッシュにセットアソシアティブ方式が採用され，置き換えアルゴリズムに Least Recently Used（LRU）または Round Robin（RR）が用いられている．よって，シミュレータに 1 度実装すれば，キャッシュサイズ，Way 数，ラインサイズやアクセス時間等のパラメータの変更のみで対応できる．

2.5.2　メインメモリのアクセス時間

キャッシュヒット時のキャッシュアクセス時間はデータシートから得られるが，メインメモリへのアクセス時間はアーキテクチャやマザーボードによって異なるため計測する必要がある．メインメモリのアクセス時間は，**図 5** に示すコードを使って測定する．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．これは，OS のページ割当て処理を実行することを意図している．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の単純な差分には，タイムスタンプカウンタ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．

```
 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 5　メインメモリのアクセス時間測定用コード
Fig. 5　The code for measuring the access time of main memory.

3. 実　　装

ここでは移植の際に実装が必要な部分を，Pentium-M，XScale，SH アーキテクチャそれぞれについて述べる．アーキテクチャに依存しているのは，主に基本ブロックの時間測定部分でその他のキャッシュ等は RETAS へのパラメータ設定のみで対応できる．

3.1　Pentium-M アーキテクチャ

Pentium-M [4] ではタイムスタンプカウンタを rdtsc 命令を使って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 3 行である．

Pentium-M アーキテクチャ上で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う．その後，タイムスタンプカウンタを取得し，基本ブロックを実行する．Pentium-M アーキテクチャの場合は mov 命令を用いてレジスタアクセスやメモリアクセスが行われる．メモリアクセスは，無効なアドレスにアクセスすることになるため，グローバル変数へのアクセスに書き変える．スタックポインタである esp，ebp への代入は，他のレジスタ（eax，ebx 等）に置き換える．pop，push，jmp 命令等はコメントアウトする．これらのアセンブリコードの書き変えには，python を利用したスクリプトで実装しておりコード行数は約 200 行である．

3.2　XScale アーキテクチャ

XScale [5] 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を取得できる．ただし，Pentium-M の場合と異なりパフォーマンス測定用のレジスタへのアクセスが特権モードに限られる．よって，OS にこれらのレジスタを操作する命令を追加する必要がある．RETAS の実装では，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 に追加し，そのコード行数は約 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．XScale の場合は，レジスタアクセス命令（mov）とメモリアクセス命令（ldr，str）と分離されているため，Pentium-M のように mov 命令がレジスタアクセスなのかメモリアクセスなのか判定することがないため，スクリプトのコード行数が小

© 2010 Information Processing Society of Japan

さくなっている．

3.3　SH アーキテクチャ

ターゲットとした SH7780 [6] 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．RETAS 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評価環境の Linux 2.6.20 のカーネルモジュールとして作成する．ユーザ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数は約 190 行である．

4．評　　価

提案手法の移植性と予測精度を評価する．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．また，Pentium-M と XScale は Linux 2.4.20 を，SH は Linux 2.6.20 を用いて評価する．

4.1　移　植　性

各アーキテクチャへの移植に際し，記述したコード行数を**表 4** に示す．各アーキテクチャとも，カウンタ取得とアセンブリ書き変えの合計で 200〜300 行程度で移植できることが分かる．ただし，アーキテクチャによってはカウンタを取得するにもカーネルへの知識が必要になる．また，アセンブリを書き変えるにもアーキテクチャの命令セットへの十分な理解が必要であったり，GCC の出力するアセンブリコードの特徴を知ったうえではじめて書き変えスクリプトの記述が可能になる．

一方で，アーキテクチャのモデルを実装し実行時間を求める場合には，すべての命令に対し，その命令がパイプラインをどのように使うのかという記述が必要になる．この場合はモデルの移植は非現実的であり，RETAS の移植性の高さが分かる．

移植が必要な部分以外の RETAS のコード行数は，RTL を出力するための GCC へのパッチが約 900 行，`RTL Level Simulator` が約 500 行，フロー解析器が約 1000 行である．GCC へのパッチの約 900 行のうち約 800 行が RTL の出力関数で，残りの約 100 行がその出力関数を呼び出す部分やアセンブリコードへのコメント挿入部となっている．GCC のバージョンアップに追随する場合でも RTL の仕様に変更がない限り RTL の出力関数はそのまま利用できるので，実質的に記述が必要なコード行数は約 100 行である．なお，本論文では GCC 4.1.1 に対してパッチを実装している．

表 1　Pentium-M architecture
Table 1　Pentium-M architecture.

マシン	IBM Thinkpad X31
プロセッサ	Pentium-M 1.6 GHz
L1 キャッシュ/レイテンシ	32 KB/3 cycle
L2 キャッシュ/レイテンシ	1,024 KB/9 cycle
データキャッシュ	8 way Set Associative
置き換えアルゴリズム	LRU
ラインサイズ	64 bytes

表 2　XScale architecture
Table 2　XScale architecture.

マシン	Sharp Zaurus SL-C3000
プロセッサ	XScale PXA270 416 MHz
命令キャッシュ	32 KB/32 way set associative
データキャッシュ	32 KB/32 way set associative
置き換えアルゴリズム	Round-Robin
ラインサイズ	32 bytes

表 3　SH architecture
Table 3　SH architecture.

マシン	Hitachi SH7780 Solution Engine
プロセッサ	SH7780 400 MHz
命令キャッシュ	32 KB/4 way set associative
データキャッシュ	32 KB/4 way set associative
置き換えアルゴリズム	LRU
ラインサイズ	32 bytes

表 4　移植に要するコード行数
Table 4　The number of lines for porting.

アーキテクチャ	カウンタ取得	アセンブリ書き変え
Pentium-M	3	200
XScale	40	160
SH	100	190

© 2010 Information Processing Society of Japan

4.2　ベンチマーク

ベンチマークプログラムを用いて最悪実行時間の予測値と実測値を比較し，予測精度を評価する．マイクロベンチマークとして静的最悪実行時間予測手法の研究で一般に用いられている Matmul と Fibonacchi を使用する．実アプリケーションとしてヒューマノイドロボット HRP2[8] 上で実時間タスクとして動作しているプログラムである Servo と Stereo を使用する．

4.2.1　Matmul ベンチマーク

Matmul ベンチマークは 16×16 の整数値の行列乗算プログラムである．ループの繰返し回数はプログラム中に定数で与えられている．3 重のループが 1 カ所あり，分岐は存在しない．実行可能経路は 1 通りで，コード行数は 10 行である．演算は整数演算のみである．

4.2.2　Fibonacci ベンチマーク

Fibonacci ベンチマークはフィボナッチ数 30 を求めるプログラムである．ループの繰返し回数はプログラム中に変数で記述されているため，アノテーションの挿入によってループの繰返し回数を記述する必要がある．1 重のループが 1 カ所あり，分岐数は 1 である．実行可能経路は 2 通りある．コード行数は 10 行である．演算は整数演算のみである．

4.2.3　Servo タスク

Servo タスクは，センサ値からアクチュエータ目標値を計算するセンサ・サーボループプログラムで，センサ・サーボの I/O 処理部分はコメントアウトしている．1 重のループが 11 カ所あり，分岐数は 15 である．ループの繰返し回数はプログラム中に定数で与えられてる．実行可能経路は 3456 通りある．コード行数は 300 行である．演算には浮動小数点演算を含む.

4.2.4　Stereo タスク

Stereo タスクはロボットの行動計画時に必要な，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である．カメラ入力部分はコメントアウトしている．5 重のループと 4 重のループがそれぞれ 1 カ所あり，分岐数は 2 である．実行可能経路は 4 通りある．コード行数は 115 行である．演算は整数演算のみである．

4.3　予 測 精 度

4.2 節で述べたベンチマークプログラムを用いて RETAS の予測精度を評価する．

4.3.1　実測値の測定

ベンチマークタスクが最悪な実行パスを通るときの実行時間を測定するため，ベンチマークプログラムの分岐条件を最悪な実行パスを通るように修正する．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 の処理時間は考慮していないため，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．

```
 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 6　実測値の測定用コード
Fig. 6　Measurement code of actual timing.

図 6 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当てを行う．次に，ベンチマークプログラムをひととおり動かして OS のページ割当てを実行する．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 回の計測を行った最悪値を実測値として採用している．

4.3.2　メインメモリのアクセス時間の測定

メインメモリのアクセス時間は，2.5.2 項に示したコードを使い，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つね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．**表 5** に評価に用いたアーキテクチャのメインメモリアクセス時間の測定結果を示す．

4.3.3　ベンチマークへのアノテーションの挿入

RETAS はループの繰返し回数がプログラム中に定数で与えられている場合には，コード

© 2010 Information Processing Society of Japan

表 5　メモリアクセス時間の測定結果
Table 5　Result of main memory latency.

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

表 6　ベンチマークプログラムの最悪実行時間の予測値と実測値
Table 6　Estimated and actual WCET of benchmark programs.

アーキテクチャ	ベンチマーク	コードサイズ (bytes)	計算時間 (秒)	実測値 (cycles)	予測値 (cycles)	誤差 (%)
Pentium-M	Matmul	105	4	28398	32507	+14
	Fibonacci	51	1	146	164	+12
	Servo	1108	461	23372	31857	+36
	Stereo	478	8243	131.51 M	165.71 M	+26
XScale	Matmul	108	4	48184	51277	+6
	Fibonacci	60	1	190	204	+7
	Servo	1156	489	1526 K	1803 K	+18
	Stereo	504	9212	297.78 M	316.72 M	+6
SH	Matmul	104	4	51307	54604	+6
	Fibonacci	40	1	220	252	+14
	Servo	582	494	34187	38728	+13
	Stereo	370	9475	336.89 M	342.81 M	+2

の自動解析によってアノテーションなしにループの繰返し回数を求めることができる．4.2 節で述べたベンチマークのうち，`Fibonacci` のみループ回数が変数で与えられているためアノテーションが 1 行挿入されているが，それ以外のベンチマークプログラムにはアノテーションが含まれない．

4.3.4　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 RETAS の予測値の結果を**表 6** に示す．コードサイズは，ベンチマークの実行コードのバイナリサイズを示している．計算時間は，`RTL Level Simulator` のシミュレーションに要した時間を示している．

予測誤差は Pentium-M の場合に +14%から +36%で，XScale の場合に +6%から +18%で，SH の場合に +2%から +14%である．誤差を見るとすべてのアーキテクチャとベンチマークにおいてプラス方向の誤差となっている．最悪実行時間予測では，実測値よりも予測値がつねに大きいことが求められるため安全に予測できていることが分かる．

Pentium-M の予測誤差が他のアーキテクチャに比べて大きくなっているのは，実際にはメインメモリのアクセス時間にばらつきがあるが，測定したメインメモリのアクセス時間の最悪値を一律に用いてメモリアクセス時間を計算しているためである．

`Servo` タスクに注目すると，実測値，予測値ともに XScale の値が Pentium-M や SH の値に比べて数十倍大きくなっている．これは XScale に浮動小数点ユニットがないため浮動小数点命令をソフトウェアでエミュレートしているためである．

5.　関 連 研 究

最悪実行時間予測は様々な要素技術の組合せで求められる．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[1]．

5.1　フロー解析

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等のフロー情報を記述する手法[9] 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[10),11] 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解析した実行フローとが異なる可能性がある．

RETAS は，コンパイラの中間表現を用いることでコンパイラの最適化を考慮したフロー解析を行うことができる．中間表現に対するフロー解析器は 1 つのため，1 度解析器を実装するだけで済み，移植性に優れている．また，RETAS のフロー解析器はループ構造の自動解析やアノテーションによるフロー情報の補完機能を持っている．

5.2　実行時間解析

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．たとえば，命令キャッシュ[12,13] やデータキャッシュ[14,15]，分岐予測機構[16]，パイプライン[17,18] 等のモデルがある．一方で，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

© 2010 Information Processing Society of Japan

時間を計測する手法があげられる．パイプラインやキャッシュといったプロセッサの様々な機構での実行時間は，実機でコードを実行することによりすべて計測結果に含まれている．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．Petters ら[19] はコンパイラの生成する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[20] 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．

RETAS 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．モデルを用いず実行時間を計測するため，移植性にも優れている．ただし，RETAS は命令キャッシュや分岐予測機構については考慮していない．本論文で用いたベンチマークプログラムは命令キャッシュに収まる大きさで，分岐数も少ない．RETAS で命令キャッシュに収まらない大規模なタスクの実行時間を予測すると，キャッシュのスラッシングの影響で安全でない予測結果になる可能性がある．

5.3　精　　度

産業界で利用可能な最悪実行時間を予測する製品として RapiTime[21] や aiT[22] 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もとに確率的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もと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やコンパイラに依存しているため広く利用することはできない．

ツールの精度は同一のアーキテクチャやベンチマークプログラムを用いない限り比較することはできないが，aiT は PowerPC MPC755 上で +30〜+50%の予測精度[23] がある．RapiTime は確率的に最悪実行時間を計算するため精度が変化するが，SimpleScalar Simulator 上で +14〜+366%の予測精度[24] がある．RETAS は様々なアーキテクチャ上で +2〜+36%の予測精度がある．これは既存のツールと比較しても少くとも同等の精度があり，移植性の利点を考えると提案手法は有用であるといえる．

6. ま　と　め

本論文では，高い移植性を持つ最悪実行時間予測手法について述べた．提案手法を実行時間予測ツール RETAS として Pentium-M，XScale，SH アーキテクチャ上に実装し予測精度と移植性の評価を行った．実行時間を予測する際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．アーキテクチャに特有の命令の実行時間については，パイプラインのモデル化を行わず実行時間を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に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，Pentium-M に対応させる場合に約 200 行のコード量，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．ただし 200〜300 行のコード記述量であっても，カーネルや，アーキテクチャの命令セットに関する十分な知識が必要である．予測誤差は Pentium-M の場合に +14 から +36%で，XScale の場合に +6 から +18%で，SH の場合に +2 から +14%である．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，誤差がすべてプラス方向のため安全に予測ができている．

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．

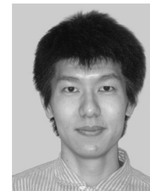
参　考　文　献

1) Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. and Stenström, P.: The worst-case execution-time problem — overview of methods and survey of tools, *ACM Trans. Embed. Comput. Syst.*, Vol.7, No.3 (2008).
2) 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).
3) 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).
4) Intel Corporation: *IA-32 Architecture Software Developer's Manual* (2004).
5) Intel Corporation: *Intel XScale Core Developer's Manual* (2004).
6) Renesas Technology：SH7780 ハードウェアマニュアル (2006).
7) Free Software Foundation: GNU Compiler Collection Internals. http://gcc.gnu.org/onlinedocs/gccint/
8) 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

© 2010 Information Processing Society of Japan

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).
9) 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).
10) 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).
11) 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 Workshop on Object-Oriented Real-Time Dependable Systems* (2003).
12) 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).
13) 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).
14) 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: Proc. 3rd IEEE Real-Time Technology and Applications Symposium* (*RTAS '97*), p.192, IEEE Computer Society (1997).
15) 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).
16) 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).
17) Schneider, J. and Ferdinand, C.: Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation, *LCTES '99: Proc. ACM SIGPLAN 1999 workshop on Languages, compilers and tools for embedded systems*, pp.35–44, ACM Press (1999).
18) 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).
19) Petters, S. and Farber, G.: Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible (1999).
20) 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).
21) Rapita Systems Ltd.: RapiTime. http://www.rapitasystems.com/rapitime
22) AbsInt: aiT WCET Analyzers. http://www.absint.com/
23) Souyris, J., Pavec, E.L., Himbert, G., Borios, G., Jégu, V. and Heckmann, R.: Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation, *5th Intl. Workshop on Worst-Case Execution Time* (*WCET*) *Analysis* (2007).
24) Bernat, G., Colin, A. and Petters, S.M.: WCET Analysis of Probabilistic Hard Real-Time Systems, *Proc. 23rd Real-Time Systems Symposium RTSS 2002*, pp.279–288 (2002).

(平成 21 年 7 月 24 日受付)
(平成 21 年 12 月 15 日採録)



山本　啓二（正会員）

2004 年同志社大学工学部知識工学科卒業．2006 年東京大学大学院情報理工学系研究科コンピュータ科学専攻修士課程修了．現在，同博士課程在学中．実時間システムに興味を持つ．

石川　　裕（正会員）

1982 年慶應義塾大学工学部電気工学科卒業．1987 年慶應義塾大学大学院工学研究科電気工学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．1993 年技術研究組合新情報処理開発機構出向．2002 年より東京大学大学院情報理工学系研究科コンピュータ科学専攻．教授．次世代高性能コンピュータシステム，高信頼システムソフトウェア開発技術，実時間処理等に興味を持つ．

© 2010 Information Processing Society of Japan

松井 俊浩（正会員）

1980年東京大学計数工学科卒業．1982年東京大学大学院情報工学専門課程修士修了，工業技術院電子技術総合研究所に入所．1991年工学博士．マルチメディアディスプレイ，オブジェクト指向言語EusLisp，幾何モデラー，移動型オフィス情報サービスロボット等の研究を通じてロボット工学に貢献．1991年から1999年にかけて，米国スタンフォード大学，マサチューセッツ工科大学，オーストラリア国立大学等の客員研究員．2003年より，産総研デジタルヒューマン研究センター副センター長．現在は，産総研情報通信エレクトロニクス分野の研究コーディネータ．日本ロボット学会，計測自動制御学会から論文賞，ISRR国際会議から研究賞，工業技術院成績優秀者賞等を受賞．日本ロボット学会，情報処理学会，米国IEEEの会員．

© 2010 Information Processing Society of Japan