社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2005－ARC－164 (14)
2005／8／4

実行時間予測ツールの設計と実装

山本啓二† 石川 裕† 松井俊浩††

リアルタイムシステムで実行されるタスクは，定められたデッドラインまでに実行を完了することが求められる．従って，リアルタイムシステムを設計する際には，各タスクの最悪実行時間を把握し，それがデッドラインを満たすかを確認する必要がある．本論文では，タスクの最悪実行時間を予測する手法について提案を行なう．提案手法はGCCのRTLを用いており，ソースコードを解析するフロー解析部とアーキテクチャごとの実行時間を解析する実行クロック解析，キャッシュ解析部からなる．これらの解析器はRTLを使用しているため，様々な言語およびアーキテクチャに適用できる．また，これら解析器を実行時間予測ツールとして実装を行ない，設計通りに動作することを確認する．

The Design and Implementation of an Execution Time Analysis Tool

KEIJI YAMAMOTO ,† YUTAKA ISHIKAWA † and TOSHIHIRO MATSUI††

A real time task is required to complete execution by its deadline. Therefore, it is necessary to know the worst case execution time of the task to design the real time system, and to confirm whether it satisfy deadline. In this paper, we propose a new technique for predicting the worst case execution time of the task. The proposed technique is based on gcc RTL(Register Transfer Language). It consists of the flow analysis that analyzes the source code, execution time analysis and cache analysis that expects the execution time of an architecture. These techniques can be used as a general technique because it is independent of a specific language and architecture. These techniques are implimented as an execution time analysis tool.

1. はじめに

ロボット制御[1]などに用いられるリアルタイムシステムでは，計算の正確さだけでなく，時間的な正確さも求められる．つまり，ある決められたデッドラインまでに必要なタスクの実行を完了することが要求される．ロボット制御では，タスクの持つデッドラインまでに処理が完了しない場合に，ロボットが倒れたり，他の物体と衝突する可能性がある．このようなデッドラインミスを回避するには，リアルタイムシステムの設計時に，それぞれのタスクの最悪実行時間（Worst Case Execution Time:WCET）を予測することが重要となる．

WCETの予測手法は，動的解析と静的解析の2つに分類することができる．動的解析は，予測対象のプログラムを実機で動かし，その実行時間を統計的に求める手法である．この手法は，予測対象のプログラムをブラックボックスとして扱うため実行時間の測定は容易に行なうことができるが，プログラムが最悪な実行パスを通っているのかを判断することはできない．よって，求められた最悪実行時間は真に最悪な実行時間であるとは言えず，予測した最悪実行時間を超過する可能性がある．マルチメディア処理などの厳密なリアルタイム性が求められないソフトリアルタイムシステムを設計する場合は，多少のデッドラインミスは許容できる．しかし，ロボット制御や自動車制御などの，デッドラインミスがシステムに致命的な影響を及ぼすハードリアルタイムシステムの場合には，予測した最悪実行時間を超えないことを保障する必要がある．

静的解析はプログラムのソースコードを解析することで実行時間を予測する手法である．ソースコードにはプログラムに含まれる全ての動作が記述されているため，実行する可能性のある全てのパスについて解析が可能となる．しかし，キャッシュや分岐予測，アウトオブオーダー実行などの高速化機構を含む実行時間は，実機でプログラムを動作させないと分からない場合が多く，精度の面で難しい．

本論文では，プログラムのソースコードを解析し実行時間を予測する静的実行時間予測手法について提案を行なう．提案手法は，様々なアーキテクチャで実行時間の予測が可能となる汎用的なフレームワークの構築を目標としている．

† 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
Graduate School of Information Science and Technology, The University of Tokyo
†† 産業技術総合研究所デジタルヒューマン研究センター

2. 関連研究

静的実行時間予測の一般的な手法は，プログラムを基本ブロックに分割し，基本ブロックの実行される回数を予測するフロー解析部分と，ターゲットアーキテクチャ上で個々の基本ブロックを実行するのに必要とされるクロック数を予測する実行クロック解析部分から構成される．基本ブロックとは，分岐の入らない一連の命令のまとまりを指す．

2.1 フロー解析

フロー解析では，C 言語やアセンブリ言語などのソースコードを基本ブロック分割し，関数の呼び出し関係や分岐，ループの繰り返し回数の解析などを行なう．一般的なプログラムでは，ループの繰り返し部分が実行時間の大部分を占めるため，繰り返し回数の上限を正確に知ることが WCET の予測精度の向上につながる．

2.1.1 ループ解析

ループの繰り返し回数を求める手法は，ユーザが解析器に対して繰り返し回数を明示する手法と解析器がソースコードを自動解析して繰り返し回数を求める手法に分けられる．

繰り返し回数を明示する手法[2)3)]は，ユーザがソースコードに特別な構文を使用してループの繰り返し回数を記述する．そして，その情報を解析器が読み取ることで繰り返し回数を取得する手法である．この手法は，どのようなループであってもユーザが繰り返し回数を記述する限り繰り返し回数を取得できる．しかし，個々のループに対して繰り返し回数を記述する手間がかかることや，繰り返し回数の記述にプログラミング言語を独自拡張した構文を用いるために，通常のコンパイラを使用してプログラムをコンパイルすることができなくなるという問題もある．

ソースコードを自動解析して繰り返し回数を求める手法[4)5)]は，ソースコードからループ部を判別して繰り返し回数の上限を自動的に求める手法である．この手法では，ソースコードを変更する必要も無く，ユーザが明示的に繰り返し回数を記述する手間も無い．既存のプログラムにも容易に適用することが可能であるが，実行時の状態によって繰り返し回数が決まるループなど，構文によっては適用範囲が限定されるという問題がある．

2.1.2 言語依存

既存のフロー解析手法[2)3)]は特定言語に依存したものとなっている．フロー解析の対象が高級言語の場合は，コンパイラの最適化が原因となりフロー解析の結果と，コンパイルして生成されたバイナリの動作が異なる可能性がある．コンパイラの最適化によってループの強度削減やアンローリングが行なわれたり，到達不能パスが削除されたりする．WCET の予測精度は，解析対象が実行バイナリに近いほど向上する．よって，高級言語をフロー解析する場合は，コンパイラの最適化についても考慮することが必要となる．

アセンブリ言語をフロー解析する場合は，コンパイラの最適化が完了しているため WCET の予測精度は高くなる．しかし，アセンブリ言語からは高級言語に含まれているループや分岐，変数などの情報を十分に得ることができないため，解析が難しくなる．また，解析器は特定のアーキテクチャに依存しているため，解析アルゴリズムの汎用性は低いという問題がある．

2.2 実行クロック解析

実行クロック解析では，ターゲットマシン上で基本ブロックに含まれる一連のコードを実行したときのクロック数を予測する．

近年のプロセッサは複数の実行ユニットを持ち，キャッシュや分岐予測といった実行時の状況によって変化する機構を持つため，厳密な実行クロック数を求めることはできない．しかし，これら機構をある程度モデル化して予測を行なうことは可能である．

クロック数を予測する上で重要になるのは，パイプライン解析とキャッシュ解析である．パイプライン解析では，インストラクションがパイプライン中をどのように流れるのかを予測する．また，キャッシュ解析では，メモリアクセスがキャッシュにヒットするかミスするのかを予測する．アウトオブオーダー実行[6)]や分岐予測[7)]などの機構をモデル化し動作を予測している研究もあるが，本論文では実行クロックに大きく影響を及ぼすパイプライン解析とキャッシュ解析のみについて考える．

パイプライン解析やキャッシュ解析は，ターゲットのアセンブリ命令を解析することで行なう．これらアセンブリ命令は，コンパイラによって生成したり，実行バイナリを逆アセンブルして生成する．

Kirner らは[8)]ループの繰り返し回数をソース中に記述できる `WCET C` という C 言語を拡張した言語を提案している．また，パイプライン解析では，ステージが 4 つで実行ユニットも 1 つの単純な構造のパイプラインを考慮している．キャッシュに関しては考慮しておらずこれら手法の適用範囲は限定的である．

Healy らは，キャッシュ解析に関してはセットアソシアティブ方式のインストラクションキャッシュ[9)]およびダイレクトマップ方式のデータキャッシュ[10)]をモデル化し WCET の予測を行なっている．メモリアクセスがどのようにキャッシュにヒットするのかを always hit/always miss/first hit/first miss の 4 種類に分類してキャッシュミスによるペナルティを見積もっている．しかし，予測のターゲットとしているアーキテクチャが `microSparc` であるなど近年のアーキテクチャとは構造が異るため，これら手法の適用も限定される．

2.3 最悪実行パスの検索

プログラム全体の WCET を求めるには，フロー解

析や実行クロック解析の結果を元にしてプログラムの最悪な実行パスを探索することで行なう．一般的な計算手法として，整数計画法[11][8]やTiming Schema[12]を利用したものがある．

3. 提案手法

本章では，前章で述べた問題点を解決するWCETの予測手法を提案する．

フロー解析で求めるループの繰り返し回数については，ユーザの負担を避けるため可能な限りソースコードの自動解析で求める．自動解析できない構文については，ユーザが繰り返し回数を記述可能な形式とするが，関連研究にあるようなプログラミング言語を独自拡張した構文は避ける．

実行クロック解析では，様々なアーキテクチャにおいて容易にWCETの予測が行なえることを目標とする．このためパイプライン解析では，パイプラインの具体的なシミュレーションを行なわずに実機でインストラクションを実行し，基本ブロックの実行クロックを求める．また，パイプライン解析とは別に基本ブロックで発生するメモリアクセスについてキャッシュ解析を行ない，メモリアクセスを原因とするパイプラインストールのペナルティを計算する．

3.1 WCET予測ツールのフレームワーク

図1にWCET予測ツールのフレームワークを示す．提案手法では，フロー解析で解析対象とするソースコードとしてRTL（Register Transfer Language）[13]を用いる．RTLは，GCCの内部で用いられている中間言語で，アセンブリ言語に近いがアーキテクチャ非依存の低級言語である．

アセンブリ言語をフロー解析する場合は，様々なアーキテクチャごとに解析器を作成する必要があるが，RTLはアーキテクチャに非依存のため同一のフロー解析手法で様々なアーキテクチャに対応することができる．また，GCCでコンパイルできる高級言語はコンパイルの過程でRTLに必ず変換されるため，様々な高級言語に対してもフロー解析が可能となる．さらに，GCCの最適化処理はRTLに対して行なわれるため，コンパイラの最適化を考慮した解析が行なえ，WCETの予測精度は高くなる．

フロー解析の対象にRTLを使用することは，高級言語やアーキテクチャに依存しない汎用的な予測を行なう上で重要な要素である．

3.2 フロー解析

フロー解析ではRTLを解析して基本ブロックへの分割を行ない，関数の呼び出し関係やループの繰り返し回数といったフロー情報を求める．

ループの繰り返し回数は可能な限り解析器による自動解析で求める．ただし，自動解析で解析できない構文にも対応するため，ユーザがソースコードに繰り返し回数を注釈として記述できる手法も用意する．ソースコードに記述した注釈情報は，ソースコードと共にGCCによってRTLに変換された後にフロー解析部で抽出される．

```
while(1) {
  WCET_LOOP_BOUND (30);  /* 注釈 */
}
```

図2 ユーザによる繰り返し回数の注釈情報

図2に，C言語における注釈情報の記述例を示す．while(1)のようなループ構文は，自動解析でループ回数を求めることができない．しかし，ループ部のブロック内へ注釈情報を伝える擬似関数である`WCET_LOOP_BOUND (30)`を記述することにより，このwhileループは最大30回実行されると解析器に対して示すことができる．このような擬似関数は，解析器が繰り返し回数を読み取った時点で破棄するため，これ以降の解析および最適化フェーズで擬似関数の存在は無視することができる．

3.3 実行クロック解析

基本ブロックの予測実行クロックは，基本ブロックを構成するインストラクションを実機で実行して求める．CPUの動作をシミュレートすることで実行クロックを求めることも可能ではあるが，シミュレータの開発にコストがかかり，様々なアーキテクチャに対応することは難しい．

本手法では，個々の基本ブロックを一定回数実行して平均の実行クロックを求める．プログラムのフローを無視して，直接基本ブロックの命令を実行するために，次に示すインストラクションは，フローを無視して実行しても安全な命令へと書き変える．

- スタック操作
- 関数呼び出し
- ジャンプ
- メモリアクセス

さらに，基本ブロックの先頭にスタックを待避するコード，末尾にスタックを復元するコードを挿入する．また，基本ブロックを直接関数として呼び出しできるようにするため，ブロック先頭にラベルを記述する．関数呼び出しやジャンプ命令はnopに変換し，スタック操作に関るレジスタアクセスを別の安全なレジスタアクセスへと置き換える．また，メモリを参照する命令はレジスタを参照するように書き変える．

本手法では，メモリ参照をレジスタ参照へ書き変えているため，実機で発生すると考えられるメモリアクセスを原因とするパイプラインストールの影響を把握することができない．そこで，別途メモリアクセスのペナルティを予測するためにキャッシュ解析を行なう．

クロック解析により測定したクロックとキャッシュ解析で得られるペナルティを加算して，基本ブロックの実行クロックとみなす．

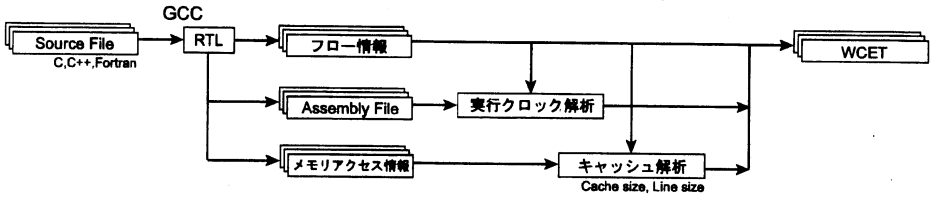

図 1 WCET 予測ツールのフレームワーク

3.4 キャッシュ解析

メモリアクセスを原因とするパイプラインストールはプログラムの性能に大きな影響を及ぼす．Pentium Mプロセッサ[14]の場合には，キャッシュのアクセスレイテンシは，L1 キャッシュが 3 クロック，L2 キャッシュが 9 クロックである．また，キャッシュにヒットせずメインメモリにアクセスする場合には，データの準備ができるまでパイプラインがストールし，数百クロック以上のペナルティが発生する．よって WCET の予測精度を上げるには，メモリアクセスがキャッシュにヒットするかを解析することが重要となる．

キャッシュには，インストラクションキャッシュとデータキャッシュがあるが，本論文ではデータキャッシュについて解析する．インストラクションキャッシュについては考慮せず，常にインストラクションはキャッシュにヒットしペナルティは発生しないと仮定する．また，キャッシュ解析で解析の対象とするのは，ループ内でループ変数を添字とした配列へのアクセスや，ポインタをインクリメントしつつ行なうアクセスのような，線形にメモリアドレスが変化するアクセスパターンを持つメモリアクセスである．

解析の手順としては，まず基本ブロック内で発生するメモリアクセスの抽出を行ない，フロー情報から得られるループ部分およびループ変数の増減パターンから，そのブロック内でアクセスされる仮想的なメモリアドレス領域を求める．キャッシュへのヒットおよびミスは，同一の仮想的なキャッシュラインへ何度アクセスが発生するかを計算することで求める．

本手法は例えば，キャッシュラインが 16byte で 4byte 毎のメモリアクセスが 4 回発生するとき，最初のアクセスは必ずキャッシュミスを起こし，その際にデータはメモリから読み込まれるためキャッシュラインは満たされる．よって，残りの 3 回のアクセスは必ずキャッシュにヒットすると予測する．

4. 実　　装

GCC で用いられている中間言語の RTL をフロー解析の対象とするため，GCC の内部に WCET 予測ツールを実装している．GCC の構造はフロントエンドとバックエンドに分かれており，フロントエンドでは高級言語から RTL へのパースが行なわれる．バックエンドでは，フロントエンドで生成された RTL に対して数十段の最適化を行なった後に，ターゲットアーキテクチャのアセンブリ文を出力している．

WCET 予測ツールのうち，フロー情報とメモリアクセス情報を抽出する部分は，バックエンドで行なわれる最適化処理の一部として実装している．フロー解析のうち関数の呼び出し関係を解析する部分は，高級言語から RTL にパースされた直後に行なう．これは，RTL の最適化が進むと RTL 中に記述される関数の引数などの情報の追跡が難しくなるためである．

ループの繰り返し回数の自動解析および注釈情報の解析は，ループ最適化フェーズの直後に行なう．ループ最適化フェーズでは，ループの強度削減やループ展開の処理が行なわれるため，最適化の前後でループ回数が変化する可能性があるためである．

メモリアクセス情報は GCC の最適化が全て終了し RTL に対応するアセンブリ文を出力する直前に行なう．また，WCET 予測ツールを実装する際にベースとして用いた GCC は，GCC 3.3.3 である．

(i) `for (i = 0 ; i < 10 ; i++) { S; }`

(ii)
```
i = 10;
while (i < 100){ S; i+=2;}
```

(iii)
```
for (i = 0; i < n ; i++) {
  WCET_LOOP_BOUND (25);
  S;
}
```

図 3 評価に用いたソースコード

5. 評　　価

実装を行なったフロー解析・実行クロック解析・キャッシュ解析について評価を行なった．

フロー解析では，**図 3** に示す一般的なループを解析し，ステートメント S を含むブロックが何度実行されるのか調べた．

それぞれのループの解析結果を**表 1** に示す．(i)，(ii) のループは，初期条件および終了条件が定数で与えられているため繰り返し回数は定数で求められる．(iii) のループでは，終了条件が変数のため繰り返し回数も変数となるため，変数を含む式として繰り返し回数を求めることができる．ただし，この場合はユー

表 1 ループ部分のフロー解析結果

	(i)	(ii)	(iii)
ループ変数	reg59	reg59	reg60
初期値	0	10	0
最終値	10	100	reg59
増減	1	2	1
評価値	10	100	reg59
評価関数	reg59<10	reg59<100	reg60<reg59
注釈	-	-	25
繰り返し	10	45	25 , reg59

表 2 ベンチマークプログラムの実行クロック解析結果

	実測クロック	予測クロック	割合 (%)
insertsort	387	497	128
matmul	994	1405	141
fibcall	302	307	101

ザによって注釈情報である `WCET_LOOP_BOUND (25)` が記述されているため，このブロックの繰り返し回数は25回であると解析ツールに知らせることができる．

実行クロック解析では，ベンチマークプログラムをターゲットで実行したときの実行時間と予測実行時間を比較した．測定に用いた CPU は `Pentium M 1.4GHz` で OS は `Linux 2.4.20` である．実行時間の測定には RDTSC 命令を用い，クロックレベルでの時間を計測した．なお，実行クロック解析で考慮していないメモリアクセスによるパイプラインストールを避けるためベンチマークの問題サイズは CPU の L1 キャッシュに収まる大きさとし，データ部は測定前にあらかじめアクセスすることでキャッシュに格納した．

ベンチマークの測定結果を**表 2** に示す．`insertsort`, `matmul`, `fibcall` はそれぞれ挿入ソート，行列の乗算，フィボナッチ数列の加算を行なうベンチマークプログラムである．挿入ソートはソートの際の比較回数がデータセットによって変化するため，最悪な場合のデータセットを用意し実行した．どのベンチマークにおいても，実測値よりも予測値が大きくなっているため安全な実行時間の予測が行なえていると言える．

実測クロックと予測クロックとの差は，実行クロック解析でアセンブリ文の書き変えを行なったために，インストラクション間に依存関係が生じアウトオブオーダー実行などの高速化機構が妨げられるのが大きな要因となっている．インストラクションの依存関係に変化が生じないように書き変えを行ない計測を行なったところ，`insertsort` の場合は 340.02 クロックで，`matmul` の場合は 958.50 クロックであった．この場合は予測クロックの方が実測クロックより小さくなり安全な実行時間の予測とは言えない．よって表 2 の結果は，インストラクションの依存関係によるパイプラインストールがマージンとなり安全な予測実行時間となっていることになる．

```
for(i = 0 ; i < SIZE ; i++)   /* depth 1 */
 for(j = 0 ; j < SIZE ; j++)  /* depth 2 */
  for(k = 0 ; k < SIZE ; k++) /* depth 3 */
   c[i][j] += a[i][k] * b[k][j];
```

図 4 matmul ベンチマークのメモリアクセス部分

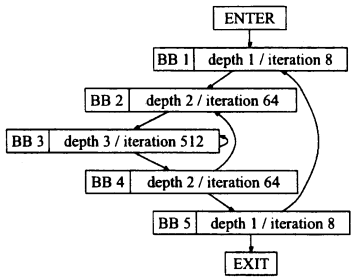

図 5 matmul ベンチマークのフロー解析結果

また，`matmul` ベンチマークに対してキャッシュ解析を行なった．**図 4** に `matmul` のメモリアクセス部分のソースコードを示す．`a,b,c` は `int` 型の変数で 4 バイトの容量を持つ．**図 5** は図 4 をフロー解析した結果をフローグラフとして示したものである．図 4 の `BB` は基本ブロックの ID を表し，`depth` 及び `iteration` はループの深さと繰り返し回数を表す．**表 3** にキャッシュ解析の結果を示す．解析の際に使用したパラメータはキャッシュサイズが 16Kbyte，キャッシュラインが 16byte である．表 3 のミス回数はそれぞれの `depth` 以下で発生すると予測されるキャッシュミスの回数を表している．アドレス数は，それぞれの `depth` 以下で発生するメモリアクセスのユニークメモリアドレス数を表している．

例えば，`depth` 3 に注目したときの繰り返し回数は 8 回となる．ユニークアドレス数は，`a,b` 共に 8 アドレスであるが，`c` のメモリアドレスはループの間に変化することはないため 1 アドレスとなっている．キャッシュミス回数は，`a` は 4 バイト毎にキャッシュラインを線形にメモリアクセスしているため 2 回となる．`b` も線形にメモリアクセスを行なっているが，32 バイト毎のアクセスのため本手法では全てキャッシュミスとして扱っている．

また，`depth` 2 に注目したときの繰り返し回数は 64 ($8*8$) 回であり，ユニークアドレス数は，`a` については `depth` 3 のみに注目した場合と変らず 8 アドレスであり，8 つのメモリアドレスに 8 回づつアクセスしていることになる．また，`b` の場合は 64 個のメモリアドレスに 1 回づつアクセスを行っていることを表している．

表 3 キャッシュミスの予測回数 (matmul , SIZE=8)

	ミス回数			アドレス数		
depth	3	2	1	3	2	1
a	2	2	16	8	8	64
b	8	64	64	8	64	64
c	1	2	16	1	8	64
合計	11	68	96	17	80	192

6. まとめと今後の課題

タスクの最悪実行時間を予測するツールを構築することを目標とし，その予測手法の提案を行なった．既存手法は，特定のアーキテクチャや言語に依存しているため幅広い問題に適応できないが，提案手法は RTL の使用や実機でのインストラクション実行により，様々なアーキテクチャおよび言語に適応することが可能である．

今後の課題としては，L1，L2 キャッシュなどの複数レベルのキャッシュを考慮したキャッシュ解析器の実装や，セットアソシアティブ方式のキャッシュアルゴリズムをモデル化することを考える．また，ヒューマノイドロボット制御を主目的に開発が進められている RMTP[15] や SHx など他の組込み向けプロセッサへの適用を行い，移植性の評価を行う予定である．

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) 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，　齋藤元，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 2004-SLDM-114, pp.1–7 (2004).

2) 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).

3) Li, Y.-T.S., Malik, S. and Wolfe, A.: Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software., *IEEE Real-Time Systems Symposium*, pp.298–307 (1995).

4) 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).

5) 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).

6) Xianfeng Li, Abhik Roychoudhury, T. M.: Modeling Out-of-Order Processors for Software Timing Analysis, *Proc. 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pp.92–103 (2004).

7) 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).

8) Kirner, R., Lang, R., Freiberger, G. and Puschner, P.: Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models, *Proc. 14th Euromicro International Conference on Real-Time Systems*, pp. 31–40 (2002).

9) Healy, C., Arnold, R., Mueller, F., Harmon, M. and Walley, D.: Bounding Pipeline and Instruction Cache Performance, *IEEE Trans. Comput.*, Vol.48, No.1, pp.53–70 (1999).

10) 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).

11) Puschner, P. and Schedl, A.: A Tool for the Computation of Worst Case Task Execution Times, *Proc. 5th Euromicro Workshop on Real-Time Systems*, pp.224–229 (1993).

12) C.Y.P and A.C.Shaw: Experiments with a Program Timing Tool based on a Source-Level Timing Schema, *Computer*, Vol.24, No.5, pp. 48–57 (1991).

13) : GNU Compiler Collection Internal, `http://gcc.gnu.org/onlinedocs/gccint/RTL.html`.

14) Intel Corporation: *IA-32 Intel Architecture Optimization Reference Manual* (2004).

15) 山崎信行: Responsive Multithreaded Processor の全体設計，情報処理学会研究報告 2004-SLDM-114, pp.9–14 (2004).