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

2006－ARC－169 (22)
2006／8／2

移植性の高い実行時間予測手法の設計と実装

山本啓二† 石川 裕† 松井俊浩††

信頼性の高いリアルタイムシステムを構築するためには，実行するタスクの最悪実行時間を把握し，デッドラインを超えないことを保障することが重要である．本論文では，タスクの最悪実行時間を予測する手法を提案する．提案手法は，GCC の中間表現である RTL を使用して実行フローの解析を行なう．また，アーキテクチャ毎の実行時間を，実機上でコードを部分的に実行および計測し，メモリアクセスレイテンシをシミュレータを用いて予測する．本手法はアーキテクチャ依存部分がほとんど無く高い移植性を持つ．これら手法を実行時間予測システム RETAS として実装し評価する．提案手法が，SimpleScalar シミュレータや XScale，Pentium-M において安全な最悪値を予測することを示す．

Design and Implementation of Portable Execution Time Analysis Method

Keiji Yamamoto ,† Yutaka Ishikawa † and Toshihiro Matsui††

To design a real-time system, it is important to know the worst case execution time of a task, and to confirm whether it satisfies deadline. In this paper, we propose a new method for predicting worst case execution time of a task. The proposed method analyzes execution flow based on RTL (Register Transfer Language) of GCC. Execution time is predicted by combining partial execution of the code and memory access latency calculated using a simulator. This approach has high portability, because there are few architecture dependent parts. Based on this approach, an execution time analysis tool named RETAS is implemented. We demonstrate that RETAS predicts the execution time safely in different environments, SimpleScalar simulator, XScale and Pentium-M.

1. はじめに

信頼性の高いリアルタイムシステムを構築するには，タスクがデッドラインを超えないことを保障することが重要である．そのためには，それぞれのタスクの最悪実行時間 (Worst Case Execution Time:WCET) を知ることが必要である．

一般に WCET を求めるには，様々な条件の元でプログラムの実行を繰り返し，その実行時間を測定し統計的に見積もる．しかし，このような手法はプログラムをブラックボックスとして扱うため，真の WCET を得ていると保障することはできない．そこで，プログラムのコードを解析して実行可能経路を求め，WCET を予測する静的実行時間予測について研究されている．

静的実行時間予測はフロー解析，実行時間解析，計算の 3 つの部分から構成される．フロー解析はプログラムのコードを解析し実行可能経路を求める．実行時間解析は，コードの一部分の実行時間を求める．計算は，フロー解析と実行時間解析の結果から最悪となる実行経路を求め WCET を計算する．

† 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

フロー解析および実行時間解析については過去に様々な手法が提案されている．中でも実行時間解析は WCET を求める上で特に重要な技術で，アーキテクチャの様々な機構を一つ一つモデル化したものや実機を用いた測定手法などが提案されている．

本論文では，従来の WCET 予測手法に代わる新しいフロー解析手法および実行時間解析手法を提案する．提案手法は，コンパイラの中間表現を用いてフロー解析を行なう．また，実機上でコードの実行時間を測定し，シミュレータを用いたメモリアクセスレイテンシ予測と組み合わせて実行時間解析を行なう．プロセッサシミュレータである SimpleScalar を使用して，提案手法で安全な最悪実行時間が予測できることを示す．また，実環境へも適用し XScale，Pentium-M プロセッサにおいてベンチマークプログラムの実行時間の予測を行ない，最悪値を保障できることを示す．

2. 関連研究

WCET は様々な要素技術の組み合わせで求められる．これら要素技術はフロー解析および実行時間解析の 2 つに分けられる．

フロー解析では，プログラムのコードを基本ブロックを最小とするグラフ構造で表す．この構造は Control Flow Graph(CFG) と呼ばれる．既存研究はアセンブ

リコードを解析してCFGを生成するものが多い．最低限のCFGはアーキテクチャごとにアセンブリコードの解析器を用意するだけで求めることができる．しかし，WCETの予測には各基本ブロックが最大で何回実行されるのかといった追加の情報が必要である．このような情報を自動的に求めるには，データフローの解析を行なわなければならない．

データフロー解析についてはC言語において自動的にループ回数を計算する手法[1]やC言語を拡張しソースコード中に解析器へヒントを与える手法[2]などがある．

一般に，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依存のため解析器の移植性は乏しい．また，変数などの意味情報が失われているため，データフロー解析が難しいという問題がある．C言語のような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フロー情報を求めることになる．

実行時間解析では，コードの一部分の実行時間を求める．つまり，あるプロセッサ上でどのように命令が実行されるのかを解析することになる．このため，プロセッサの様々な機構のモデル化について研究が行なわれている．例えば，命令キャッシュ[3][4]やデータキャッシュ[5][6]，分岐予測機構[7]，パイプライン[8][9]などのモデルである．しかし，プロセッサの内部動作に関する詳細な情報が公開されていないため，正確なモデルを構築することはできない．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性に乏しい．

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障することができない．

Pettersら[10]はコンパイラの生成するCFGを使い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．同様に，Lindgrenら[11]は，様々な実行経路で実行時間を測定して連立方程式を立て，これを解くことで基本ブロックの実行時間を求めている．Wenzelら[12]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．ただし，これら手法は動的なメモリアクセスを含むプログラムの場合にメモリアクセス違反が問題となり，適用は静的なプログラムに限定される．

提案手法は，コンパイラ内部で利用される中間表現を利用することにより，アーキテクチャへの依存性をなくしコンパイラの最適化を考慮した実行可能経路を求める．また，中間表現でデータフローを解析し，基本ブロックの実行回数を計算する．実行時間解析は，複雑なアーキテクチャに対応するため実行時間の測定により求める．動的メモリアクセスを含むプログラムへも対応するため，実行時間をメモリアクセスレイテンシと命令実行時間に分けて考え，シミュレータを用いてメモリアクセスのレイテンシを計算する．測定およびシミュレーション共にアーキテクチャ非依存とし，様々なアーキテクチャで共通に使用できる予測手法を提案する．

3. 提案手法

図1に提案する実行時間予測システムであるRETASを示す．提案手法の特長はコンパイラの中間表現を用いたフロー解析および，実機を用いた測定とシミュレーションを組み合わせた実行時間解析である．

実行時間の予測対象プログラムはC言語で記述されているとする．C言語のソースコードはコンパイラのGCCを用いてコンパイルする．このGCCはRETASシステム用に，GCC内部でフロー解析を行ないRTL[13](Register Transfer Language)とControl Flow Graph(CFG)を出力するように変更されている．RTLはGCCの内部で使用される中間表現の一種である．

実行時間の解析は，RTLとCFGを用いたシミュレーションによるメモリアクセスレイテンシの予測と，実行時間測定ツールを用いた命令実行時間の測定から構成される．

RTLレベルシミュレータ上でRTLを実行しメモリアクセスパターンを取得する．このパターンをキャッシュシミュレータへ入力し，キャッシュヒット回数やメインメモリのアクセス回数を計算する．これらシミュレータは，キャッシュサイズやアルゴリズムなどのパラメータ以外はアーキテクチャに非依存である．

実行時間測定ツールは，アセンブリコードを実行時間の測定単位に分割し，その前後に測定用コードを挿入する．このコードを実機で動かし実行時間を計測する．時間測定用コードはアーキテクチャ毎に命令が異なるため，対象のアーキテクチャごとに用意する必要がある．

3.1 中間言語を用いたフロー解析

C言語のソースコードはGCCの内部でTREEおよびRTLと呼ばれる中間表現へ変換される．GCCは，これら中間表現に対して各種の最適化を行ない，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し出力する．中間表現においてフロー解析を行なうため，既存手法のアーキテクチャへの依存性やコンパイラの最適化に関する問題点を解決できる．また，中間表現においてデータフローを解析し静的なループの繰り返し回数を自動的に計算する．

一方で，動的な実行フローはソースコードの静的解析で求めることは不可能である．これら動的フローもCFGへ含めるためには，人間が動的フロー情報を記述する必要がある．提案手法はこれら情報をAnnotationとしてソースコードへ記述する．GCC内部に実装したパーサでこれらAnnotationを解析し，情報をCFGへ出力する．Annotationの例を以下に示す．

- `#pragma WCET("range", n, 0, 20)`
  変数nの取る範囲が $0 \leq n \leq 20$ であることを

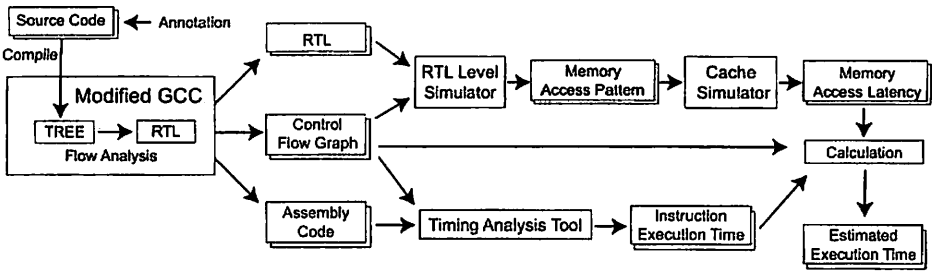

図 1 実行時間予測システム RETAS

表す.

- `#pragma WCET("loop", 10)`
  この pragma が含まれるブロックのループ回数が最大で 10 回であることを表す.

Annotation は pragma を用いて記述する. 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る. 上記のループ回数が 10 回を示す pragma は `WCET_LOOP(10)` という関数呼び出しへと変換する. GCC 内部ではこのような WCET_をプレフィックスに持つ関数呼び出しは, Annotation を示す偽の関数呼び出しとして扱う. フロー解析の過程でこのよう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する. 抽出後は中間表現から偽の関数呼び出しを削除してコンパイルを継続する. これら解析器は GCC 4.0.2 に実装している.

3.2 実行時間の予測

提案手法は複数のアーキテクチャで実行時間の予測を可能とするため, モデルの代わりに実行時間の測定による実行時間の予測を行なう. 本手法の利点は, アーキテクチャの様々な機構の影響を測定結果にすべて含むことが可能な点にある. 反対に, 実行可能経路を網羅しなければ最悪値は保障することができない. 一般に実行可能経路を全て実行することは不可能なので, コードを分割して個々の実行時間を測定することになる. ただし, 通常の実行フローを無視して任意の部分から実行するため, メモリアクセス違反が発生し実行時間が測定できないという問題がある. そこで, プログラムの実行時間をメモリアクセスの時間とメモリアクセスを除いた命令の実行時間に分けて考える.

メモリアクセスに要する時間は, シミュレータを用いてメモリアクセスパターンを取得し, そのパターンをキャッシュシミュレータで解析しキャッシュへのヒット回数やメインメモリのアクセス回数を計算する. シミュレータは中間表現の RTL を直接パースして実行可能である. RTL を利用するため, アーキテクチャに依存することなくメモリアクセスパターンの取得が可能である.

メモリアクセスを除いた命令実行時間は, 実機を用いて命令実行時間を計測して求める. アセンブリコードと CFG から実行時間測定部分を抜きだし, 前後に計測用コードを挿入する. このコードを実機で動かし実行時間を得る.

例えば関数の実行時間は提案手法では以下の式で求める. T_{inst} は命令の実行時間で T_{mem} はメモリアクセスレイテンシを表す.

$$T_{func} = T_{inst} + T_{mem}$$

命令実行時間 T_{inst} は次式で与えられる. N_{block} は関数に含まれる基本ブロックの数で, T_{B_i} は i 番目の基本ブロックの実行時間を表し, N_{B_i} は基本ブロック B_i の実行回数を表す.

$$T_{inst} = \sum_{i=1}^{N_{block}} T_{B_i} * N_{B_i}$$

メモリアクセスレイテンシ T_{mem} は各レベルのキャッシュアクセスレイテンシとメインメモリのアクセスレイテンシを足し合わせる. C_i はキャッシュのレベルを表し N_{C_i} は C_i レベルのキャッシュへのヒット回数, L_{C_i} は C_i レベルのキャッシュのアクセスレイテンシ, N_{main} はメインメモリへのアクセス回数, L_{main} はメインメモリのアクセスレイテンシを示す. L1 キャッシュヒット時のレイテンシは既に T_{inst} に含まれると考えられるため, この式には L1 ヒット時のレイテンシを含めない.

$$T_{mem} = \sum_{i=2}^{N_{cache}+1} N_{C_i} * (L_{C_i} - L_{C_1}) + N_{main} * L_{main}$$

3.2.1 メモリアクセスレイテンシの予測

一般的なプログラムには複数の分岐が含まれるため, 複数の実行可能経路が存在する. 最悪な経路を求めるには全ての経路について実行時間を得なければならないが次の点で不可能である. まず, 組み合わせ爆発により多くの時間を必要になること. また, 全ての経路を通るテストデータを生成することが難しいことである. その他に, 実行前にアクセスされるメモリ領域が既知でない場合にはメモリアクセス違反となる可能性もある.

提案手法では, メモリアクセス部分をシミュレータを用いて解析することで, これらの問題を解決する. シミュレータではコードを正確に実行せず, 一部の分岐命令を任意に操作することで様々な経路の実行を可能にする. よって, テストデータを用意する必要もない. また, メモリアクセスはシミュレータの内部で完結するため, どのようなメモリアクセスを行なってもメモリアクセス違反にはならない.

提案手法ではRTLの実行シミュレーションが可能なRTLレベルシミュレータを実装している．本シミュレータを用いると，GCCの出力するRTLを用いて関数単位のメモリアクセスパターンが取得可能である．シミュレータの内部では無限大のメモリ空間を扱うことができる．メモリアクセス発生時には，独立した4GByteのメモリアドレス空間をポインタに動的に割り当てる．任意の2つのポインタは独立しているため，代入されない限り同じメモリ空間を指すことはない．

メモリアクセスパターンが取得されると，このパターンを用いてキャッシュシミュレーションを行ないキャッシュのヒット回数やメインメモリのアクセス回数を求める．これらの回数はキャッシュサイズやアルゴリズムに依存するため，パラメータとしてキャッシュシミュレータへ与える．現在は，セットアソシアティブ方式でライン置き換え方式はラウンドロビンおよびLRUアルゴリズムを実装している．キャッシュサイズやWay数やラインサイズなどは任意に設定可能である．

3.2.2 命令実行時間の予測

実行時間は基本ブロック単位で求める．GCCによって生成されたアセンブリコードを基本ブロックに分割しメモリ確保のためのコードを挿入する．また，ブロックの前後に実行時間測定用のコードを挿入する．基本ブロック内のフローを変更する可能性のある命令，たとえば分岐命令などは "nop" 命令へ書き変える．

クロック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CPUに搭載されているCPU内部のタイムスタンプカウンタを用いる．Pentium[14]ではRDTSC命令を用いることができる．XScale[15]ではパフォーマンス測定用に使用されるPerformance Monitoring Registerを用いることができる．しかし，このレジスタへのアクセスは特権レベルのみに許可されているため，ユーザレベルからは操作することができない．そこで，XScaleの評価環境のOSであるLinux 2.4.20へ新たにPerformance Monitoring Registerを操作するシステムコールを追加する．

3.3 提案手法の制限

提案する実行時間測定手法は全ての環境に適用できるとは限らない．まず，実行時間の測定はクロック単位で命令の実行時間が取得可能であることに依存している．近年のCPUにはこの様な時間取得命令が存在するが，古いアーキテクチャなどは提案手法を適用することができない．また，実行時間測定前にメモリを確保しメモリアクセス違反の発生を防ぐが，広大なメモリ空間を測定部で扱う場合にはメモリアクセス違反が発生する可能性がある．これらは，新たにメモリサイズなどを指定するpragmaを追加することで解決することは可能である．

予測値が真のWCETよりも早くなる問題として，分岐予測や命令キャッシュを考慮していないことがある．分岐予測は，全て分岐予測ミスがないものとして扱っている．よって，頻繁に分岐予測ミスの発生する環境では予測値が最悪値より短く予測される．また命令キャッシュについても，現時点では全てヒットしレイテンシは無いと仮定しているため同様のことが発生する．データキャッシュに関してはロード命令のみを考慮しており，ストア命令は考慮していない．つまりストア命令でのキャッシュ汚染が考慮されないため，ロード時のキャッシュヒット回数が大きくなり結果として予測値が最悪値よりも短くなる．

予測値の精度の問題として，基本ブロック間のパイプラインを考慮していないことが挙げられる．提案手法は個々の基本ブロックの実行時間を元に全体のWCETを予測する．計測によって基本ブロック内のパイプラインの影響は考慮されるが，基本ブロック間のパイプラインオーバーラップは現状では考慮していない．よって，より実行時間が長くなる方向，つまり安全な方へ実行時間は予測される．また，ロード命令はLSQ(Load Store Queue)を考慮しておらず全てキャッシュへアクセスしに行くと仮定している．このため，LSQが使用された場合より実行時間は長く予測される．

4. 評　　価

実行時間予測手法の信頼性を評価するためには対象とするアーキテクチャの内部動作について詳細な情報が必要となる．しかし，この様な情報の多くは公開されていない．本論文では内部動作が全て判明可能なソフトウェアシミュレータを用いて予測手法を評価する．使用するのは，SimpleScalarツールセット[16]に含まれる`sim-outorder`である．`sim-outorder`は，スーパースカラー，out-of-order実行，分岐予測やキャッシュなどの動作をクロックレベルでシミュレーション可能である．

4.1 SimpleScalarシミュレータを用いた評価

提案手法での予測値が実際の最悪値より大きくなることが，安全なWCETを求める上で必要である．`sim-outorder`は，6つのパイプラインステージ(Fetch, Dispatch, Scheduler, Execution, Writeback, Commit)を持つ．分岐予測ミスが無い場合は，Fetch, Dispatchステージは命令で満たされている．Schedulerステージにおいて，命令が必要とする実行ユニットやメモリアクセスポートが確保されればExecutionステージへ進み命令が実行される．資源の確保ができない場合は，後続命令がout-of-orderで実行されることになる．

提案手法は，まず命令のみの実行時間を実機上で測定する．これは，キャッシュミスによるハザードが最小限のため，パイプラインが満たされていると考えることができる．しかし，実際にはキャッシュミスのハザードは測定時より多くなる．よってメモリアクセスのレイテンシを別途シミュレーションし測定結果に加えている．メモリアクセスレイテンシはデータシート記載の値をそのまま使う．実際にはキャッシュミスの時のメモリアクセス時間には後続命令を実行することができるが，提案手法では後続命令を実行せずパイプ

表 1 SimpleScalar 評価パラメータ

Processor Model	SimpleScalar 3.0/PISA
Data Cache Size	L1 4 KB / L2 64 KB
Cache Access Latency	1 cycle / 6 cycles
Way	4 Way, Set Associative
Replacement Policy	LRU
Line Size	32 bytes / 64 bytes
Memory Access Latency	18 cycles

表 2 ベンチマークプログラムの予測結果と最悪値

ベンチマーク	最悪値	予測値	割合%
matmul	30044	32459	108
insert sort	10139	10514	103
fibonacci	2135	4102	192

表 3 matmul におけるメモリアクセス比較

	SimpleScalar	RETAS
Load Instruction	12288	12288
L1 Hit	12193	12192
L2 Hit	44	48
Main Memory	51	48

ライン全体がメモリアクセス時間停止していることになる．よって，実際よりも提案手法で求めた予測時間は大きくなる．ただし，現在の実装では 3.3 章に示す制限が存在している．

4.2 sim-outorder シミュレータの予測結果

実装を行なった実行時間予測システム RETAS を評価するため，いくつかのベンチマークプログラムを用いて実行時間を予測する．予測対象として sim-outorder シミュレータ[16] を用いる．シミュレータのパラメータを表 1 に示す．

ベンチマークプログラムは全て最適化オプションの "-O2" を指定してコンパイルする．SimpleScalar で求めた最悪値と RETAS の予測値を表 2 に示す．最悪値はベンチマーク毎に最悪な実行パスとなるデータセットを与え実行時間を測定している．

どのベンチマークも最悪値よりも予測値の方が大きいため安全に予測されている．これらベンチマークの中で，fibonacci のみ誤差が大きい．fibonacci はメモリアクセスを含まないレジスタのみの演算を行なうベンチマークである．fibonacci の最内周ループに含まれる命令数は 5 命令で測定した実行時間は 4 クロックであり，このループは 1023 回繰り返される．最悪値を見ると 1 ループあたり約 2 クロックで実行されており，測定結果である 4 クロックとは誤差が大きい．この差はループ時の基本ブロック間のパイプラインオーバーラップにあると考えられる．

表 3 に表 2 の matmul ベンチマークについて，SimpleScalar シミュレータで得られたメモリアクセス統計と RETAS の求めたメモリアクセス解析結果を示す．ロード命令の回数は，RTL レベルシミュレータで RTL を実行した結果と，SimpleScalar でバイナリコードを動かした結果と厳密に一致している．キャッシュのヒット回数は多少の誤差はあるもののほぼ一致しているといえる．この誤差は，動的なメモリ確保時のメモリアドレスの違いや，スタックポインタのアドレスの違いなどに起因する．これら結果より，RTL を用いてメモリアクセス解析を行なっても，アーキテクチャ依存のアセンブリコードを実行するのとほぼ同じ結果が得られると言える．

表 5 ベンチマークプログラムの予測結果と最悪値

プロセッサ	ベンチマーク	最悪値	予測値	割合%
XScale	matmul	48184	51277	106
	insert sort	19873	19967	100
	fibonacci	5110	5124	100
Pentium-M	matmul	24741	33057	133
	insert sort	14442	23658	164
	fibonacci	3334	3911	117

表 6 matmul ベンチマークのメモリアクセス予測結果

	XScale	Pentium-M
Load Instruction	8448	9844
L1 Hit	8351	9794
L2 Hit	-	0
Main Memory	97	50

4.3 実機の予測結果

評価対象とするアーキテクチャは XScale[15]，Pentium-M[14] である．評価するプロセッサの特性を表 4 へ示す．

予測値および最悪値を表 5 に示す．XScale については RETAS システムはそれぞれのベンチマークに対して安全に予測している．また，予測誤差は数%程度であり，厳密に予測していることがわかる．Pentium-M の場合も安全に予測されているが，XScale と比較すると予測誤差は大きくなっている．XScale と Pentium-M との予測精度の違いは，基本ブロック間のパイプラインのオーバーラップにあると考えられる．XScale は ARM アーキテクチャで Pentium-M と比較すると構成が単純である．このため，XScale ではパイプラインの影響が少なく，Pentium-M ではパイプラインの影響が大きくなり誤差が広がったと考えられる．

matmul ベンチマークに関して，実行時間の予測の計算に用いたメモリアクセスシミュレーションの結果を表 6 に示す．matmul ベンチマークのワーキングセットは約 3KBytes のため全て L1 キャッシュに収まる．よってメインメモリへのアクセスレイテンシのみが表 5 には加算されている．

5. ま と め

本論文では最悪実行時間を予測するために，フロー解析と実行時間解析について移植性の高い新しい手法を提案した．フロー解析ではコンパイラの中間表現を用いることで，アーキテクチャへの依存性を解消しコンパイラの最適化も考慮された正確な CFG を取得できる．実行時間解析では，実機を用いてコードの一部の実行時間を測定し，メモリアクセスレイテンシは別途シミュレーションによって求める．本手法は，複雑なアーキテクチャをモデル化することなく実行時間を取得でき，様々なアーキテクチャへの移植も容易である．

提案手法を実行時間予測システム RETAS として実装を行ない，ベンチマークプログラムの最悪値を

表 4 評価環境

Processor	XScale PXA270	Pentium-M
Frequency	416MHz	1.4GHz
Data Cache Size	32 KB	L1 32 KB / L2 1024 KB
Cache Access Latency	2 cycles	3 cycles / 9 cycles
Way	32 Way, Set Associative	8 Way, Set Associative
Replacement Policy	Round-Robin	LRU
Line Size	32 bytes	64 bytes
Memory Access Latency	121 cycles	191 cycles

予測することで評価を行なった．SimpleScalar シミュレータでの評価の結果，安全に，つまり最悪値よりも予測値が大きく予測できた．また，実環境の XScale, Pentium-M を用いた評価も行ない，それぞれのアーキテクチャにおいて安全に予測できた．複数のアーキテクチャへ適用したことで移植性の高さも示すことができた．

予測誤差に関しては，基本ブロック間のパイプラインのオーバーラップを考慮していないため誤差の広がるものもあった．しかし，オーバーラップによる高速化が無視されているため，より実行時間の長い方へ予測される．今後の課題として，提案手法の制限に示した分岐予測や命令キャッシュなどを考慮した WCET の予測手法について考える．

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) 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).
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) 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).
4) 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).
5) 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).
6) 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).
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) 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).
9) 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).
10) Petters, S. and Farber, G.: Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible (1999).
11) Lindgren, M., Hansson, H. and Thane, H.: Using Measurements to Derive the Worst-Case Execution Time, *Proceedings of RTCSA 2000*, Cheju Island, South Korea, IEEE Computer Society (2000).
12) Wenzel, I., Kirner, R., Rieder, B. and Puschner, P.: Measurement-Based Worst-Case Execution Time Analysis, *Proc. 3rd IEEE Workshop on Future Embedded and Ubiquitious Systems*, pp.7–10 (2005).
13) : GNU Compiler Collection Internal, `http://gcc.gnu.org/onlinedocs/gccint/`.
14) Intel Corporation: *IA-32 Architecture Software Developer's Manual* (2004).
15) Intel Corporation: *Intel XScale Core Developer's Manual* (2004).
16) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol.35, No.2, pp.59–67 (2002).