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

2006－ARC－169 (29)
2006／8／2

ソフトウェアトレース生成による動的最適化の予備評価

請園智玲† 田中清史†

近年，ダイナミックリンクライブラリやダイナミッククラスローディング技術がソフトウェア開発において一般的に使用されている．これら技術はソフトウェアの差分配布や別プログラム間のコード共有化に有用であるが，実行中にプログラムロード行うため，関数のインライン展開やコード配置最適化などモノリシックプログラムに適用できた広いスコープの最適化を行うことが難しい．また，通常のコンパイラによる最適化では，実際の実行において通るパスとその頻度を知ることができず，この情報を利用した最適化を行うことが難しい．本稿では，上述の問題を解決するための手法として，プログラム実行時に実行中コードに対し最適化を行う動的最適化に着目し，その最適化アルゴリズムの一つとしてソフトウェアトレース生成を提案し予備評価を行う．

Preliminary Evaluation for Dynamic Optimization by Software Trace

TOMOAKI UKEZONO† and KIYOFUMI TANAKA†

Recently, Dynamic link library and Dynamic class loading techniques are generally used to develop software. Those techniques are effective for software differential distribution and commoditizing of program code. However, Those techniques are loaded to the main memory in execution time. Consequently, a compiler is dufficult to perform wide scope optimization as function inline expansion and code layout optimization, which could be applied to conventional monolithic programs. Additionally, conventional optimizer which is executed by a compiler can't obtain run-time informations e.g. program pass and it's execution frequency. Therefore, a compiler can't execute optimization that utilizes those infromations.

In this paper, we focused on the dynamic optimization which a target program at execution time, and we propose and evaluate the algorithm generating software trace which can resolve those problem descrived above.

1. はじめに

近年，ソフトウェア開発手法・実行形態は多様化し，それに合わせたコード最適化技術が求められている．本節では従来のコンパイル時に行う最適化の問題点を挙げ，その解決策としての動的最適化を議論する．

1.1 静的最適化の問題

近年，ソフトウェアの大規模化が進み，ダイナミックリンクライブラリやダイナミッククラスローディング技術を利用し，ソフトウェア間のプログラム共有化による開発効率向上が行われている．また，これらソフトウェアを分割・共有する考え方は，コンピュータネットワークによりソフトウェアの配布・更新をする場合に有効な手段となる．このため，これら技術は大規模なソフトウェア開発において頻繁に利用される．しかしながら，これら技術はコンパイル時にプログラム全体の実行コードを確定しないため，関数のインライン展開やコード配置最適化などのプログラム全体をスコープに当てた最適化を行うことができない．

† 北陸先端科学技術大学院大学 情報科学研究科
Japan Advanced Institute of Science and Technology

通常のアプリケーションソフトウェアは，入力データを入力し，出力を作成する．入力データが無い又は固定で，出力が一定のプログラムである場合，コンパイラは作成するプログラムコードがどの様に実行されるかを完全に解析することが可能になるが，入力データによってプログラムの振る舞いを変え，出力を変化させるプログラムの場合，この解析は不可能になる．この問題はコード配置最適化やレジスタ割り当てにおいて，最適化問題を困難にしている．

1.2 動的最適化

これら問題はダイナミックリンクライブラリや入力データなど，実行時に確定する要因が原因となっている．この問題を解決するために実行時に得られる情報をフィードバックしてバイナリコードに最適化を施す技術が研究されてきた．この技術を本研究では実行時最適化と位置づける．また，実行時最適化には大きく分けて 2 つの手法がある．

(1) 動的最適化
(2) プロファイルベース最適化

動的最適化は実行中のバイナリコードを実行中に変更する方法である．この方法を実現するにはハード

ウェア又はソフトウェアの実行環境に対するサポートが必要となる．最適化が実行中に行われる利点として細かな入力データの変化などで生じる非効率に即座に対応できる利点がある．加えて，CPU 内の実行状況を知るための情報は膨大なものとなるが，その情報を状況に応じ取捨できる利点がある．しかしながら，実行中のコードを修正することは難しい問題である．例えば，単純にプログラム上に命令を追加／削除すると，リロケーション問題が発生し，プログラムコードの主記憶上の移動，それに伴うプログラム中の関係する PC 相対分岐命令の再計算が必要となる．これは実行中に行う最適化のオーバヘッドとしては膨大なものとなる．本手法を採用する関連研究として Dynamo[2] や PMU-Based Profiling[5] が挙げられる．

プロファイルベース最適化はプロファイルと呼ばれる実行時に得られる情報をシミュレータなどであらかじめ実行して取得し，そのプロファイルを基にバイナリに最適化を施す手法である．この利点は実行時でなくオフライン時に最適化を行うため，最適化のためのオーバヘッドを考える必要が無い．例えば，実行時最適化で問題となったリロケーション問題は同様に起こりえるが，オフラインで行われるため，問題とならない．しかしながら，予備実行を行い最適化を行うプロセスは入力データが随時変更される場合に実行環境にとって大きな負担となる．また，CPU の動作は複雑であることから，全ての振る舞いをプロファイルとして取得すると膨大なデータとなる．そのため，取得するプロファイルは統計的なものとなる．これは最適化の効果に大きく影響する．本手法を採用する関連研究として Software Trace Cache[4] や Hot Path Prediction[3] が挙げられる．

これら 2 つの手法のどちらも，実行環境で実行時情報をフィードバックして最適化を行うため，上述のコンパイル時の最適化の問題を解決可能である．本研究は実行時最適化のアプローチとして動的最適化を採用し，動的最適化の問題点の一つである実行オーバヘッドを小さくする実行環境と最適化アルゴリズムの提案を行う．

1.3 最適化アルゴリズム

多くの実行時最適化の研究ではパスプロファイリングによるコード配置最適化が研究されてきた．これはプログラム実行時に実行パスの偏りをフィードバックしてプログラムコードの配置を変更することで，i-cache ミス数などを削減し実行効率向上をめざすアルゴリズムである．本研究では動的最適化のアルゴリズムとしてソフトウェアトレース生成アルゴリズムを提案する．このアルゴリズムは実行時情報を基にバイナリコード中の頻繁に実行される基本ブロックとその実行順序を特定し，頻繁に実行されるプログラム実行トレースを生成して主記憶上に配置する．この時生成するプログラム実行トレースを本研究においてソフトウェアトレースと呼ぶ．ソフトウェアトレースパス上の基本ブロックが主記憶上に連続に置かれるため，スーパースカラマイクロプロセッサの命令フェッチ効率向上につながる．

本稿は以下の節で構成される．2 節では動的最適化を実現する実行環境として提案するハードウェア及びシステムソフトウェアの概要説明を行う．3 節では最適化アルゴリズムとして提案するソフトウェアトレース生成アルゴリズムについての概要説明を行う．4 節では提案したソフトウェアトレース生成アルゴリズムの潜在効果を説明するための予備評価を示す．5 節では 4 節で示した評価を基に考察行い，提案した手法のまとめについて述べる．

2. 動的最適化環境

動的最適化を実現するためには実行中のプログラムを必要に応じて実行／停止して最適化処理を実行する実行環境が必要になる．Dynamo[2] は最適化対象バイナリコードをインタプリタ上で実行することでインタプリタが動的最適化のための情報収集と最適化を実行するタイミングの取得を実現している．本研究はこの機能をハードウェアとシステムソフトウェアで実現した．[1] これにより Dynamo で実現された実行環境より軽い実行環境の実現を目指す．本節ではこの動的最適を実現する実行環境の概要を説明する．

2.1 ユーザ定義トラップ

パスプロファイリングを行うためにはプログラム上の分岐命令の振る舞いを調べなければならない．また，最適化対象コード実行中に最適化処理を実行するタイミングを見つけ，最適化処理に遷移する仕組みを提供しなければならない．本研究において，これらの要求はハードウェアで実現する．提案する実行環境は実行時情報収集／最適化処理を OS のトラップハンドラで実装する．この場合，呼び出すトラップの発生条件を任意に指定できる必要がある．ユーザ定義トラップはこの機能を提供するハードウェアである．図 1 にユーザ定義トラップの概要を示す．

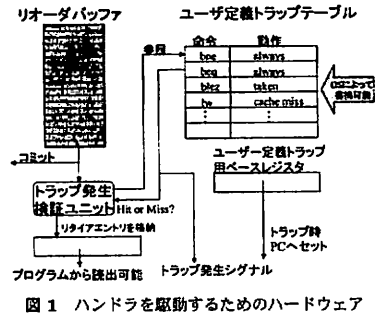

図 1 ハンドラを駆動するためのハードウェア

OS は最適化対象バイナリコードを実行する以前に

ユーザ定義トラップテーブルにトラップを起こしたい条件を記述しておく．記述は命令のオペコードとハードウェアにより定められた動作条件を示すビットパターンの組み合わせにより行う．一方，ハードウェアは命令コミット時にリオーダバッファからリタイアするエントリを解析し，コミットされる命令がテーブルに記述されていないか探索する．該当エントリがテーブル中に存在した場合，例外を発生させる．この例外は通常のプロセッサが持つ例外と異なり，事前にセットしてあった専用のトラップベースレジスタの先にジャンプし，あらかじめ登録されているハンドラ (この場合は実行時情報収集／最適化処理ルーチン) が実行される．また，ハードウェアは例外を発生させたと同時に，リタイアしたリオーダバッファエントリをソフトウェアから読み出し可能なレジスタに格納する．ハンドラはこのレジスタを参照することにより，最適化に必要な情報を得ることが可能となる．コミット時のリオーダバッファのエントリには完了した命令の種類と結果が格納されているため，プログラムの振る舞いを解析する情報としては最適である．例えば，リオーダバッファは正確な例外復帰に対応するため，全ての実行中命令の PC を保持している．また，分岐命令の場合は，分岐予測の成立／不成立を命令コミット時にチェックするため，必ず飛び先アドレスの計算結果を保持している．最適化処理はこの 2 つの情報から対象とする分岐命令の，taken，not-taken，後方分岐，前方分岐を知り，パスプロファイリングが可能となる．

3. ソフトウェアトレース生成

本研究はユーザ定義トラップを使用し，ソフトウェアトレース生成するアルゴリズムを提案した.[1] 最適化トラップハンドラは大きく分類して 4 つの処理を行う．

(1) ループ解析
実行中プログラムからループボディを見つけ出す．

(2) パスプロファイリング
ループ内の実行パスのプロファイリングを行い，最も実行される可能性の高いパスを見つけ出し，ソフトウェアトレース生成に必要な分岐命令リストを作成する．

(3) トレース生成
生成された分岐命令リストからそれぞれの分岐命令を含む基本ブロックを抽出しソフトウェアトレースとして実行可能なように加工する．

(4) トレース配置／トレースリンク
生成されたトレースを主記憶上に配置し，オリジナルコード上のトレースの先頭命令をトレースへの JUMP 命令に変換し，作成したソフトウェアトレースが実行されるようにオリジナルコードを修正する．

本節では上述のユーザ定義トラップにより呼び出される実行時情報収集／最適化処理の概要を 4 つの処理毎に説明する．また，新しく提案するパスプロファイリングの手法を併せて紹介する．

3.1 ループ解析

最適化処理はソフトウェアトレースを生成するが，プログラム中に存在する全ての実行トレースを生成することは非現実的である．そのため，最適化処理はまず，プログラム中の頻繁に実行される基本ブロックの特定を行う．頻出基本ブロックを特定する足がかりとして，ループ解析を行う．ユーザ定義トラップを用い，ループバックする分岐命令でトラップを発生させその履歴を取る．最適化処理は一定回数以上のループバック (後方分岐) を行った分岐命令を見つけ，その分岐命令の taken 方向の分岐先から辿って，再度，元の分岐命令に到達するまでのパスを頻出する基本ブロックでできたトレース (HOT トレース) と認識しする．

3.2 パスプロファイリング

最適化処理はループ区間を特定した後，一定回数のループバックを実行する間，パスプロファイリングモードとなり，ユーザ定義トラップを全ての分岐を実行した際にトラップを発生させるように設定する．パスプロファイリングモード中はループ区間内に存在する分岐命令の解析を行う．

ここでは以前に提案したパスプロファイル手法に加えて新たにグラフを作成しより精細にパスプロファイルを行う手法を提案する．

以前に提案したパスプロファイルは 1 度だけループバック分岐命令の飛び先から再度ループバック分岐命令に到達するまで実行する分岐命令のリストを作成する手法である．パスプロファイリングモード中は全ての分岐命令でトラップが発生するため，パイプラインフラッシュが頻発し，プログラム実行が非効率になる．これを最小限にするにはパスプロファインリングモードの時間をできるだけ短くしなければならない．そのためにはこの手法は大変有用である．しかしながら，分岐方向の実行割合に偏りが少ないループではソフトウェアトレースの精度が極端に低下する．このため新しく分岐命令グラフを利用したパスプロファイリング手法を提案する．本手法で解析時に作成される分岐命令グラフの例を図 2 に示す．

図 2 はパスプロファイリングモードをループバック 10 回として，その間に作成した分岐命令解析状態である．図中 A〜F に対応する節はループ内に存在する分岐命令を表し，H がループバック分岐命令を表す．2 方向に伸びる枝は，分岐命令の飛び先方向を表す．それぞれの枝に t と nt と示されるのは taken 方向に分岐したか，not-taken 方向に分岐したかを表している．またこの枝に () 付けで示される値は実際に計測中にこの方向へ分岐した回数を表している．まずプロファイリングを開始したときには H の節のみがあり，分

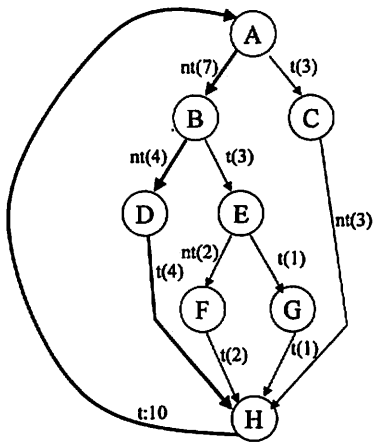

図 2　パスプロファイリング中に作成される分岐解析グラフ

岐を実行する度に発生するトラップで節の追加，枝の重み付けを行っていく．手順は，トラップが発生した分岐命令の PC を元にグラフに検索を掛けてグラフ中に同一分岐命令があるかを調べる．無い場合は新たに節を作成し前回実行した節につなぐ．存在した場合はそのパスは一度実行したパスなので，前回実行節と今回実行節 (ヒットした節) の間の枝の実行回数をインクリメントする．この処理を H から伸びる t の枝の重みが 10 になるまで続ける．解析が終わると，A から H までで実行回数が多い枝を選びながらグラフを辿り，分岐を拾い出していく．図では A，B，D，H が選ばれ，この分岐がトレース生成用の分岐リストとして採用される．

本手法を採用することにより，1 度だけ辿るパスプロファイリングに比べ，より精度の高いパスプロファイリングが可能となる．

3.3　トレース生成

パスプロファイリングで得られた分岐命令リストからソフトウェアトレースを生成するためにはいくつかの手順がある．ここでは詳細の説明は省略するが，その手順を示す．

(1)　各基本ブロックの抽出
(2)　taken 分岐の taken 条件の反転
(3)　既存トレースへのリンク
(4)　オリジナルコードへの復帰命令追加
(5)　トレース配置位置決定
(6)　トレース内のリロケーション解決

以上の処理を行うと主記憶上に配置することが可能なトレースが生成される．

3.4　トレース配置／トレースリンク

トレースを生成した段階では生成したトレースはトラップハンドラのバッファ内にあり，それを最適化対象バイナリがアクセス可能な範囲に移動し配置しなければならない．また，ただ配置しただけでは生成したトレースは実行されないので，オリジナルコード上のトレースの先頭に相当する命令をトレースへ JUMP する命令で上書きしなければならない．これがトレースリンクである．

4.　予備評価

予備評価は最適化処理のオーバヘッドを除き，理想状態で実行することによりソフトウェアトレースによる性能向上の潜在能力を検討するために行った．本節では実験方法論と評価に用いた 2 つのプログラムの実行結果を示す．

4.1　シミュレーション環境

予備評価の計測は CPU シミュレータ上で行った．シミュレータは MIPS64 命令セットアーキテクチャで作成された実行バイナリをインオーダ・スーパースカラ実行シミュレーションを行うことが可能である．主なシミュレーションパラメータを表 1 に示す．

表 1　主なシミュレーションパラメータ

設定項目	設定値
命令発行幅	2,4,8,16
分岐予測	完全分岐予測
命令ウインドウサイズ	∞
L1 i-cache サイズ	64KB (block size 64B)
L1 i-cache 構成	4 way set-associative
L1 d-cache サイズ	64KB (block size 64B)
L1 d-cache 構成	4 way set-associative
L2 unified-cache サイズ	2MB (block size 64B)
L2 unified-cache 構成	8 way set-associative
L1 cache ミスペナルティ	10 クロックサイクル
L2 cache ミスペナルティ	120 クロックサイクル

本研究で提案する最適化アルゴリズムは命令フェッチ効率向上を目的とするため，その潜在性能を明確に示すために，分岐予測はミスの無い完全分岐予測にし，また命令間依存と資源競合による命令フェッチストールを回避するため，命令ウインドウのサイズを∞にパラメータを定めシミュレートした．これにより，性能決定要因を L1 i-cache からのフェッチスループットとキャッシュミスによる CPU 全体のストールに絞る事が可能となる．L1 i-cache は 1 サイクルで 1 ブロックからの読み出しが可能であり，命令フェッチ幅分読み出すことが可能なポートが備わっている．

4.2　ワークロード

シミュレータ上で実行するプログラムとして SPEC INT95 の 099.go と 130.li を実行した．それぞれのプログラムは peak で生成し，実行時に ref のデータを指定した．コンパイラには gcc を用い，最適化オプションには `-static-O2-funroll-loops` を指定した．評価に示すデータは 20 億命令実行した状態のデータである．

4.3 動的最適化パラメータ

予備評価では最適化実行オーバヘッドを隠蔽するために最適化ルーチンをシミュレータ上に実装した．最適化には3つのパターンを用意してそれぞれの振る舞いを計測した．また，最適化ルーチンがループを特定するためのループバック回数閾値を1000回とし，パスプロファイルを行うためのループバック回数を100回として計測を行った．

(1) パスプロファイルを行わない (ver1)
(2) パスプロファイルを行いヒット精度優先でトレースを作成 (ver2)
(3) パスプロファイルを行いトレース作成数優先でトレースを作成 (ver3)

ver1は一定閾値 (1000回) のループバックが認められるループを発見した場合に，パスプロファイルを行わず，1001回目のループ実行でループバック分岐命令に到達するまで実行する全ての分岐命令をコレクションしトレースを作成する方針である．ver2は一定閾値のループバックが認められた後，パスプロファイルモードに移行し，更に100回のループバックの間，パスプロファイルを行う．しかしながら，100回のループバックが終わった後，プロファイル時に作成したグラフを辿っている際にレジスタジャンプなどの出現でプロファイルが寸断された状態やグラフ内で上方のノードにtaken分岐する場合などループバック分岐命令に到達できないパスが選択された場合に生成を中断する．ver3はver2同様，パスプロファイルを行うが，パスを辿り選択する際にver2のように生成を中断せず，再帰的に一つ上の2つ枝の存在する節に立ち戻り中断せざるを得なかった枝と反対側の枝を選択肢し，再度グラフを辿りパスを形成する．

3つのパターンにはそれぞれ期待するプログラムの振る舞いがある．ver1はパスプロファイルを行うver2, ver3と比べ最適化のためのオーバヘッドが軽い．しかしながら，1001回目の実行で収集される分岐命令の分岐先がより実行されやすい基本ブロックに分岐している確証が無いため，実行時に分岐方向が偏る分岐命令を含んだトレース作成に有効である．ver2, ver3はパスプロファイルを行うため，分岐方向を選択する根拠があるが，パスプロファイリング中は全ての分岐命令でトラップが発生するため100回のループバック中の命令スループットが極端に低下すると予想される．また，ver2はプロファイルの結果，最も実行されやすいパスのみを生成する方針であるのに対し，ver3はできるだけトレースを作成し，トレース実行の機会をver2に比べ向上させる目的がある．

4.4 評　　価

まず，評価プログラムを各最適化で実行しtaken方向への分岐した回数とnot-takenへ分岐した回数を比較した．計測したtaken方向への分岐比率を表2に示す．トレース実行中はループバック分岐命令以外

表2 taken 遷移率

	最適化	taken 遷移率 (%)
099.go	なし	57.4253
099.go	ver1	60.9875
099.go	ver2	55.7906
099.go	ver3	57.5886
130.li	なし	59.4778
130.li	ver1	61.9404
130.li	ver2	60.9521
130.li	ver3	58.8217

ではtaken方向への分岐が発生しないため，トレース生成アルゴリズムはtaken比率を減少させる傾向にあると考えられるが，評価に使用した2つのアプリケーションはtakenとnot-takenの比率がバランスしているプログラムであり，このようなアプリケーションでは本最適化はトレースの先頭からループバック分岐命令まで実行される可能性の高いトレースを作成することが困難である．この場合，トレースを作成し，トレース実行を開始してもループバック分岐命令に到達するまでにオリジナルコードに復帰する可能性が上がり性能差が生じにくくなる．また，taken比率の低い場合もプログラムは命令フェッチ効率が最適化以前の状態で高いので性能差が生じにくい．本アルゴリズムが有効なプログラムはtaken比率が高いプログラムである．表2ではtaken比率の減少は最大で2%弱となっている．

3にトレース作成数とトレース平均長，トレース実行割合を示す．

表3 トレース作成結果

	最適化	作成数	平均長 (命令)	実行割合 (%)
099.go	ver1	108	45.2314	30.33
099.go	ver2	67	67.4179	26.77
099.go	ver3	76	68.9210	24.12
130.li	ver1	16	30.6875	25.39
130.li	ver2	19	31.1052	36.41
130.li	ver3	19	20.7894	33.76

130.liは極端に小さいプログラムのため，変化が無いが，099.goではver2とver3の間でトレース作成数に9個の差が生じている．また，トレース実行割合は両プログラム通して30%前後であり，まだ70%の割合でオリジナルコードを実行していることから，トレースのヒット精度が全体的に低いことがわかる．

表4に最適化による性能の変化を示す．

予備評価では命令間依存や資源競合による命令フェッチストールが起きない状況で計測を行っているが，実際の実行，特にフェッチ幅の大きい構成では性能の頭打ちになる状況が多く，表で示される性能は達成できない．しかしながら，最適化による性能変化が明確に出ている．IPCは単純に命令フェッチスループットにキャッシュミスペナルティを引いたもので換算できる．

表 4　最適化による性能の変化

	最適化	命令フェッチ幅	IPC	i-cache ミス率	命令フェッチ率
099.go	なし	2	1.8176	0.2538	1.9029
099.go	ver1	2	1.8194	0.2906	1.9101
099.go	ver2	2	1.8237	0.2967	1.9164
099.go	ver3	2	1.8205	0.2983	1.9131
099.go	なし	4	3.1230	0.4516	3.3858
099.go	ver1	4	3.0913	0.5116	3.3625
099.go	ver2	4	3.1434	0.5310	3.4293
099.go	ver3	4	3.1257	0.5315	3.4091
099.go	なし	8	4.8765	0.7393	5.5427
099.go	ver1	8	4.7144	0.8175	5.3756
099.go	ver2	8	4.9149	0.8751	5.6517
099.go	ver3	8	5.5614	0.8672	5.5614
099.go	なし	16	6.5743	1.0465	7.8455
099.go	ver1	16	6.2858	1.1441	7.5199
099.go	ver2	16	6.6891	1.2592	8.1319
099.go	ver3	16	6.5321	1.2327	7.9054
130.li	なし	2	1.7934	6.5335E-07	1.8801
130.li	ver1	2	1.7823	1.0519E-06	1.8635
130.li	ver2	2	1.8135	7.3614E-07	1.8972
130.li	ver3	2	1.8115	1.0958E-06	1.8956
130.li	なし	4	3.0438	1.1476E-06	3.3024
130.li	ver1	4	3.0121	1.8355E-06	3.3516
130.li	ver2	4	3.0568	1.2813E-06	3.3023
130.li	ver3	4	3.0951	1.9363E-06	3.3500
130.li	なし	8	4.2610	1.6630E-06	4.7856
130.li	ver1	8	4.2119	2.6505E-06	4.6954
130.li	ver2	8	4.3391	1.8824E-06	4.8517
130.li	ver3	8	4.4919	2.9186E-06	5.0495
130.li	なし	16	5.2371	2.1032E-06	6.0525
130.li	ver1	16	5.1210	3.3045E-06	5.8539
130.li	ver2	16	5.2638	2.3433E-06	6.0367
130.li	ver3	16	5.5669	3.7278E-06	6.4496

表では 2,4,8,16 命令フェッチ幅を変化させ，最適化無しと上記で示した最適化 3 パターンの性能の変化を示している．ver2 と ver3 のパスプロファイルを行う最適化はいずれも僅かながら最適化なしの実行より IPC における実行性能が向上している．最大性能差は 130.li の ver3 で IPC が約 0.33 向上している．しかしながら，ver1 に関して性能が低下する例が見られる．前述したとおり，これは実行パスが偏らないために 1001 回目のループ実行で選択するパスが正確で無いために not-taken パスを選択すべきポイントで taken を選択していることが原因となっている．また，今回のプログラムではトレース実行は必ずループ最中で，オリジナルコードへ実行を移していた．このため，ループ内処理をオリジナルコードとトレースの両方を i-cache にマップしているため，i-cache ミス率が増大している．しかしながら，ver2 と ver3 に関してはトレース実行による命令フェッチスループットの向上により，　このペナルティが隠されいる．

本研究は予備評価としてソフトウェアトレースを生成する最適化の潜在性能を明らかにするための評価を行ったため，最適化アルゴリズムを計算機で実行するオーバーヘッドを評価に導入していない．そのため，実行時に最適化アルゴリズムが実行に及ぼす性能的影響がアルゴリズム適用による性能向上を上回る可能性がある．しかしながら，その実行でオリジナルコード実行より性能が劣った場合でも，次回以降のプログラムロード時にはオーバヘッド無しで最適化済みのプログラムコードを実行することが可能なため，ver2，3 の最適化は無用にはならない．

5. おわりに

本稿では動的最適化の一手法であるソフトウェアトレース生成を提案し，その予備評価を行った．提案したアルゴリズムを用いることで最大で約 0.33 の IPC 向上が得られた．

今後の課題として命令間依存や資源競合を入れた評価とアウトオブオーダ実行時の評価を行っていく．

本研究で提案した最適化はプロファイルベース最適化を採用しても同様に実現可能である．しかしながら，トレースを作成することにより，リロケーション問題を気にせずに容易にプログラムに対して追加，修正，削除が可能になったことから，動的最適化環境で多様な最適化を行うことが可能となる．本研究ではこのソフトウェアトレースの特性を元に作成したトレースの基本ブロック内部の最適化を行うことで，より性能向上を引き出せる最適化アルゴリズムを提案していく．

参考文献

1) 諏圀 智玲，田中 清史，"ハードウェア解析システムによるバイナリコードの動的最適化"，情報処理学会研究報告，ARC, Vol.2005, No.120, pp.7–12, 2005.
2) Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia. Dynamo : A Transparent Dynamic Optimization System. SIGPLAN Conference on Programming Language Design and Implementation, pages 1–12, 2000.
3) Evelyn Duesterwald and Vasanth Bala. Software Profiling for Hot Path Prediction: Less is more. Proceedings International Conference on Architectural Support for Programming Languages and Operating Systems, pages 202–211 2000.
4) Alex Ramirez, Josep L. Larriba-Pey, Carlos Navarro, Josep Torrellas, Matero Valero. Software Trace Cache. Proceedings International Conference on Supercomputing, pages 119–126, 1999.
5) Alex Shye. Exploring the Potential of Performance Monitoring Hardware to Support Run-Time Optimization. Master thesis, Univ of Colorado. 2005.