2005－ARC－165（2）
2005／11／30

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

請　園　智　玲†　　田　中　清　史†

近年，ソフトウェア開発手法は多様化し，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一般的になってきた．このような技術を利用する場合，プログラムのバイナリコードは実行時に確定する特性を持つため，コンパイル時に行われる静的コード最適化技術では，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プログラム全体にスコープを当てた最適化を施すことができない．

本稿では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を支援するハードウェアの提案を行い，それにより実現するアルゴリズムを併せて提案する．

Dynamic Binary Optimization Using Hardware Analysis System

TOMOAKI UKEZONO† and KIYOFUMI TANAKA†

Recently, software development methods have been diversified and use of dynamic link library and dynamic class loading techniques has become common practice. In those styles of development, programs have a characteristic that the whole binary codes are fixed at runtime. Therefore, conventional techniques for static optimization are inadequate for the binary codes dynamically fixed, since optimization that targets the whole program, e.g. function inline expansion and code layout optimization, cannot be used, which could be applied to conventional monolithic programs.

In this paper, we propose the hardware mechanism that supports dynamic optimization of binary codes and algorithms achieved by using the hardware.

1. はじめに

近年，ソフトウェア開発手法は多様化し，ソフトウェア実行するプラットホームも多様化してきている．この変化はこれまでのコンパイル時の最適化処理にとって大きな問題となる．本節ではこの問題点を明らかにし，その解決手段である動的最適化技術を紹介する．

1.1 ダイナミックリンク問題

近年，ソフトウェアの大規模化，複雑化に伴い，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，また JAVA JIT コンパイラのような動的コード生成技術が一般的に使われるようになってきた．これらの技術は，ソフトウェアの差分アップデートや差分配布，また，プログラムコードがマルチプラットホームに対応するために有用である．このような技術を利用する場合，プログラムのバイナリコードが実行時に確定する特性を持つ．このため，コンパイル時に行われる静的コード最適化技術では，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プログラム全体にスコープを当てた最適化を施すことができない．これはダイナミックリンク技術を用いるプログラム共通の問題である．

1.2 パスプロファイリング問題

静的最適化の根本的な問題として，分岐における実行パスやループ回数など，対象プログラムの実行時の振る舞いを完全に解析することができない点が挙げられる．これは入力データによりプログラムの振る舞いが変わるために生じる問題である．これにより，静的最適化処理は効果的なハードウェア資源管理や投機的な実行を行う積極的な最適化を施したプログラムコードを生成することが難しくなる．

1.3 機種依存問題

最後に挙げられる問題は機種依存最適化である．通常，静的最適化は特定の ISA（Instruction Set Architecutre）を対象に行われる．しかしながら，ISA を変更せず，長期に渡り開発された CPU は同じ ISA が使用されていても，計算資源や実装が個体により異なる．これらに対応するためにコンパイラは機種依存最適化を提供するが，ALU やキャッシュ構成など CPU 間の細かな計算資源や実装の違いに重点を置いたものではない．また，ソフトウェアベンダは数多くある CPU 構成に対応したバイナリをそれぞれ作成し，配布することが難しいため，機種依存最適化を積極的に利用していない．

† 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

1.4 動的最適化と本研究の目的

これら問題を根本的に解決するためには，プログラムを実行する計算機環境にソースコードが存在し，プログラムに対する入力データが特定されていなければならない．この条件を満たすことは現在の計算機の利用方法から考えて極めて難しい．この問題を解決するために動的最適化技術が研究されている．

本研究ではこれら問題に対処するため，プログラム実行時に，ユーザに対し透過的に実行中コードを最適化し，バイナリコード再生成を行うシステムを提案する．本システムでは対象プログラム実行中にハードウェアとシステムソフトウェアによって実行バイナリが最適化される．このことから，実行確定したバイナリに対する最適化を行うことが可能になるため，ダイナミックリンクやパスプロファイリング問題を解決することが可能となる．また，システムソフトウェアは自己のハードウェアプロファイルを知ることにより，機種依存に関係する最適化を行うことが可能となる．更に，本システムは最適化処理に専用ハードウェアを用意することにより，ソフトウェアのみで実現する動的最適化と比べ，少ないオーバヘッドで動的最適化の実現を可能とし，また，CPU ストールサイクルやストール原因などソフトウェアが通常知ることのできない情報を最適化処理に反映させることが可能となる．

提案システムを実現するために，CPU の実行時情報ログを収集するためのハードウェアと対象プログラム実行中に最適化ルーチンを起動するためのトラップ発生ハードウェアが必要となる．本稿では本研究における提案ハードウェアの一つであるユーザ定義トラップを紹介し，そのハードウェアを使った最適化としてトレース生成とトレース配置最適化を併せて提案する．これら最適化は命令キャッシュの効率利用を目的としたものである．このため，本稿における目的は提案ハードウェア／最適化アルゴリズムを用い，命令キャッシュアクセス回数削減と命令キャッシュミス回数削減にある．

2. 関 連 研 究

これまでの研究では，実行中コードに最適化処理を施す手法として，インタプリタや VM を使った動的最適化技術[1)3)]，また，一度実行した結果をフィードバックして再度，最適化を施すプロファイルベース最適化技術が研究されてきた[4)]．本節では代表的なバイナリ変換技法を挙げ，本研究との違いを明らかにする．

2.1 Dynamo

Dynamo[1)] はインタプリタ上でバイナリコードを実行することで，最適化のための情報収集と最適化タイミングの取得を実現している．主な最適化アルゴリズムとしてオリジナルコードからのトレース生成が挙げられる．トレースを抽出することによって，トレース実行中の命令キャッシュミスを減らすことが可能となる．しかしながら，オリジナルコードをインタプリタ実行するオーバヘッドは動的最適化による実行速度向上のポテンシャルを下げる結果をもたらしている．また，ハードウェアのサポート無しに，ユーザプロセスにおけるプログラムコード解析を行うため，キャッシュメモリやバッファなどのハードウェア資源の利用状況が解析できない．そのため，Dynamo は実行時最適化を行うが，その最適化はコードスケジューリングとレジスタ割り当てに特化している．

2.2 Software Trace Cache

Software Trace Cache[4)] はプロファイルベースの最適化手法である．この手法の狙いは Dynamo と同じくトレースの生成にある．しかしながら，Dynamo と違い，実行パスプロファイリングをより正確に行っている．Software Trace Cache は Greedy Chaining Algorithm を用いてパスプロファイリングを行う．Greedy Chaining Algorithm は予備実行結果から各分岐命令における分岐方向に重み（edge profile）をつけるとともに，各基本ブロックに実行回数を加味して，最も多く実行される基本ブロック群（HOT トレース）を生成する．この HOT トレースはサブルーチンや関数を超えて集められる．また，Software Trace Cache は HOT トレースの上位を特権トレースエリアと呼ばれるメモリ領域に配置し，他のトレースを特権トレースエリアの命令キャッシュ上のインデックスに重ならないように配置することで，HOT トレースにアクセスした際にキャッシュミスが発生しないことを保証している．しかしながら，この技法はプロファイルベースであるため，事前にアプリケーション毎にトレーニングインプットを用意して予備実行を行い，ポピュラーパスを算出しなければならない．また，Greedy Chaining Algorithm はプログラム中の全ての基本ブロックに対しての評価を行うため，計算コストが大きく，実行時に行う動的最適化には向かない．

2.3 PMU-Based Profiling

これまで研究された動的最適化の手法は，上記の様なソフトウェアによるパスプロファイリングの他に，Pentium-4，Itanium，Power PC 970 といったプロセッサに内蔵されているオンチップパフォーマンスモニタリングユニット（PMU）を使ったハードウエアパフォーマンスモニタリング（HPM）情報をフィードバックした最適化に焦点を当て行われてきた.[5)]Itanium-2 における PMU はユーザ指定のパフォーマンス情報を PMU レジスタに保存し，レジスタがオーバフローする際に PMU 割り込みを発生させ，レジスタ内容を主記憶に書き出す方式を取っている．Shye は Itanium-2 の BTB（Branch Trace Buffer）を主記憶に書き出す機能を利用して，分岐のプロファイル（Branch Vector）を残し，その情報を基にオフラインでパスプロファイリングを行う手法を提案している．しかしながら，この手法はハードウェア機構を利用するが，実行

中プログラムに干渉するものではない．そのため，この最適化は本質的にプロファイルベース最適化と変わらない．

2.4 本研究における手法

本稿ではこれまでに紹介した研究と同様にトレース生成による命令キャッシュの効率化利用を目的とするが，異なるアプローチで動的最適化を実現する．ユーザに対し透過的にバイナリの最適化を行うため，予備実行／オフライン最適化を必要とするプロファイルベースの最適化は行わない．実行時情報の収集は専用のハードウェアとソフトウェアによって実現する．これは CPU の内部機構に追加した専用ハードウェアとオペレーティングシステム（OS）の持つトラップハンドラによって実現される．本機構は情報収集と最適化処理がプログラム実行中にバックグラウンドで行われるため，ユーザ／プログラマ側から見た場合に透過である．

3. 最適化サポートハードウェア

適化をサポートするハードウェアとして PMU-Based Profiling と同様に近年の CPU に内蔵される PMU を使う手段がある．PMU は CPU の実行パフォーマンスを計測，プログラマがソフトウェアチューニングを行う情報提供のために存在する．そのため，動的最適化を行うためのハードウェアとしては機能的に不十分である．本研究では最適化のために十分なプロファイル情報を軽いオーバヘッドで得るために，PMU とは別途，最適化処理専用ハードウェアを提案する．本節では提案ハーウェアの 1 つであるユーザ定義トラップハードウェアを紹介し，その用途を論じる．

3.1 ユーザ定義トラップ

トレース生成とトレース配置最適化を対象プログラム実行中に行うためには，プグラム中に存在する各分岐命令の振る舞いを記録しつつ，最適化処理を開始するタイミングを見つけなければならない．本研究では分岐命令解析（パスプロファリング）とトレース生成／トレース配置処理を OS のトラップハンドラで実装する．このため，任意の動作条件でトラップハンドラを起動するハードウェアが必になる．この機能を実現するためのハードウェアを図 1 に示す．

図 1 は指定命令の特定動作からトラップハンドラを駆動するためのハードウェアのブロック図である．OS はプログラム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し，該当エントリがテーブル中に存在した場合，

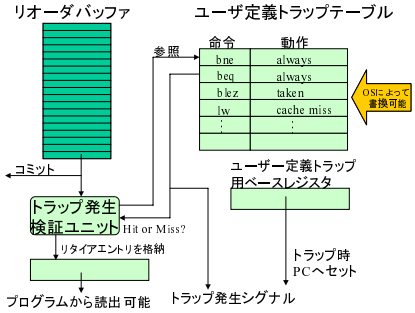

図 1 ハンドラを駆動するためのハードウェア

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．

4 節で紹介するトレース生成，トレース配置の最適化を実現するために，ユーザ定義トラップテーブルに登録が必要な命令は分岐命令である．表 1 に命令を識別するビットパターンと表 2 に分岐命令における動作条件を示すビットパターンを示す．

表 1 命令識別ビットパターン

ビットパターン [opcode]	命令種
00 [000000]	Control
01 [000000]	Load/Store
10 [000000]	Integer Arithmetic & Logical
11 [000000]	Miscellaneous

命令を識別するビットパターンは 8 ビット分用意されている．上位 2 ビットが表に対応する命令の種類を意味し，下位 6 ビットが命令を識別する命令セットアーキテクチャ上のオペコードを意味する．尚，このテーブルは MIPS 命令セットアーキテクチャを基に作られているため opcode フィールドは 6 ビットである．また，命令個々にエントリを作成することが可能だが，opcode フィールドを 0 にセットすることに

表 2 分岐命令における動作条件ビットパターン

ビットパターン	動作条件
0000 0001	not-taken
0000 0010	taken
0000 0100	forwad
0000 1000	backward
0001 0000	unconditional （Branch Type）
0010 0000	indirect （Branch Type）
0100 0000	return （Branch Type）
1000 0000	reserved

よって指定カテゴリを代表する集約エントリを作成できる．これによりエントリ使用数を削減することが可能となる．この他にも，エントリを無効化する無効化ビットフィールドが存在する．

動作条件を示すビット幅は 8 ビット分用意されている．表 2 は個々のビット位置が分岐命令の動作条件と分岐命令のタイプを示している．分岐タイプビットフィールドは分岐に関する命令の動作条件を 1 エントリに集約した際に使用される．ソフトウェアはハンドラを起動したい条件に該当するビットフィールドの位置に 1 をセットする．

これらハードウェアの導入により，最適化を行うハンドラは必要な時にだけ呼び出されるように，細かくトラップを制御することが可能となる．例えば，本稿で説明するトレース生成最適化は実行フェーズによって後方分岐数をカウントするモードと出現する分岐命令全てを収集／記録するモードが存在する．後方分岐カウントモードでは後方分岐以外でハンドラに遷移する必要が無いため，全ての分岐でハンドラを呼び出す必要が無い．例外を発生させるとパイプラインを通過中の命令を全てキャンセルしなければならないため，無用な例外発生を避けた方が最適化処理導入による実行時性能低下を抑えられる．

4. 最適化アルゴリズム

本節では 3 節で詳細を述べたハードウェアを用いて実現する最適化アルゴリズムを示す．提案するアルゴリズムはトレース生成アルゴリズムとトレース配置アルゴリズムである．実行トレースを主記憶上に置くことはスーパースカラプロセッサにとって，命令キャッシュアクセス回数とキャッシュミス数を削減することにつながる．また，トレース配置アルゴリズムによってトレース間のインデックス衝突を回避し，更なるキャッシュミス数削減が可能となる．以降，本研究が用いるパスプロファイリング法とトレース生成／トレース配置の 2 つのアルゴリズムの詳細について述べる．

4.1 ループ解析

本研究ではパスプロファイリングとトレースの生成をループに関してのみ行う．本システムは実行時に最適化処理を行う理由で，パスプロファイリングに時間的コストを必要とするアルゴリズムを適用できない．そのため，比較的コストの小さい MRET（Most Recently Executed Tail）アルゴリズム[2] を採用し，ループ解析を行う．これは，ループ回数の多いループボディ上の基本ブロックは比較的実行回数が多いという予測に基づいた方針である．パスプロファイリングの基本的なアルゴリズムは Dynamo のアルゴリズムを採用した．しかしながら，Dynamo のアルゴリズムは過度に単純化されているために，ループが入れ子状の場合に，長いトレースを生成することができない．この結果，ループに関するトレース抽出が分断され，キャッシュアクセス回数やキャッシュミス数が増加する場合がある．このことから提案するアルゴリズムは Dynamo のアルゴリズムを改良し，インナーループ解析を追加した．

例として，インナーループが存在する場合のパスプロファイリング手順を以下に示す．

(1) システムは後方分岐カウントモードでハンドラが起動されるまで待機する．
(2) 後方分岐が実行されるとハンドラが起動し，後方分岐（trace tail）のコレクションを作成する．
(3) trace tail が既にコレクションに存在する場合は実行回数をカウントしていく．
(4) カウント数がシステムの設定した閾値を超えた場合，システムはトレース生成モードに変更し，全ての分岐命令でハンドラが起動されるようになる．
(5) トレース生成モードでは後方分岐に遭遇するまで，全ての分岐の履歴を収集する．
(6) 後方分岐に遭遇した場合，その分岐が trace tail かインナーループかを判断する．インナーループの判断は飛び先が既に生成したトレースに在るか調べることで可能である．
(7) インナーループだと判断された場合，その分岐が not-taken で実行されるまで，分岐履歴の収集を中断する．
(8) インナーループを抜けた後，また trace tail に辿り着くまで分岐履歴の収集を行う．
(9) trace tail に辿り着いたら分岐履歴の収集を止め，収集した分岐履歴からトレースを作成する．
(10) 作成が終了したら，後方分岐カウントモードに戻り待機する．

パスプロファイリングアルゴリズムの異常系処理等を省き簡略化したフローチャートを図 2 に示す．

以上のパスプロファイリングを繰り返し行い，オリジナルコードから複数のトレースを抽出する．生成したトレースは主記憶上に配置後，次のトレース生成時のインナーループの検索に使用するため，管理情報をハンドラが保持する．

4.2 トレース生成

パスプロファイリングと分岐履歴収集が終了した後，システムはトレースの生成を開始する．トレース生成

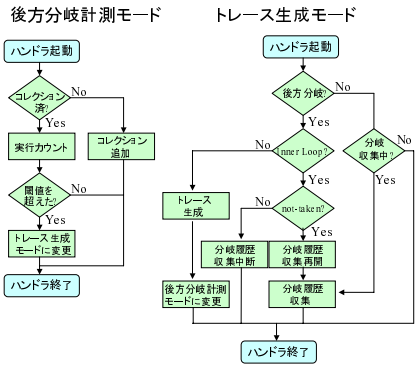

図 2 パスプロファイリングアルゴリズムフローチャート

は以下の手順で行われる．

(1) 命令拾い出し
trace tail のループバック先から再度 trace tail に辿り着くまで，収集した分岐履歴を使い，前回ループ実行の軌跡を追いかけながら命令を拾い出していく．

(2) 他のトレースへのリンク
命令拾い出しを行う過程で，インナーループを見つけた場合，インナーループ部は既に作成したトレース中に存在するので，該当するトレースの主記憶上の位置をトレース管理テーブルから検索しその場所へ飛ぶ無条件分岐命令を挿入する．

(3) トレース配置位置の決定
拾い出しが終了したら，生成したトレースの主記憶上の配置位置を決定する．配置アルゴリズムは後述する．

(4) リロケーション解決
配置位置が決定した後，トレース中の全ての PC 相対分岐命令の飛び先へのオフセットを再計算して，命令の即値を変更する．

(5) 他トレースからのリンク
リンク先のトレースの末尾のオリジナルコード復帰命令の飛び先を自トレースのインナーループ終了直後の命令に設定する．

(6) 分岐条件反転
トレース中に taken で実行した条件分岐が存在した場合，条件反転を行う．(例：bne を beq に変換)

(7) オリジナルコード復帰命令追加
作成したトレースの末尾（ループバック分岐命令の次）にオリジナルコードに復帰する無条件分岐命令を追加する．

(8) トレース貼り付け
生成したトレースを配置アルゴリズムで決定した主記憶上の位置に貼り付ける．

(9) オリジナルコード修正（自トレースへのリンク）
トレースの先頭命令のオリジナルコード上の位置にトレースへ無条件分岐する命令を書き込む．

(10) トレース管理情報更新
作成したトレースをトレース管理情報テーブルに追加する．

例として，生成トレースが他のトレースにリンクされる様子を図 3 に示す．

図 3 の（a）は 3 重入れ子構造のループをトレースに生成する順序を示している．生成アルゴリズムに従い，トレースが生成される順番は閾値を超えた順である．図の様な単純な入れ子構造のループの場合，いかなる場合も最内側のループからパスプロファイルングが開始される．図の表記では inner-loop1 ，inner-loop2，outer-loop の順である．よって生成されるトレースの順番は trace1 ，trace2 ，trace3 の順である．

図 3 の（b）は既存トレースへのリンクの図を示している．inner-loop2 がパスプロファイリングされ，トレース生成されるとき，既に trace1 は生成済みであるため trace 2 内の trace1 のコピーの部分は削除され，代わりに trace1 へのリンクのための無条件分岐命令が挿入される．trace3 も同様に trace1 ，trace2 へのリンクが行われる．しかしながら，trace3 の場合 trace1 へのリンクを行っても trace2 へのリンクで削除されてしまうため，結果的に trace2 へのリンクのみしか残らない．

図 3 の（c）はリンク先のトレースのオリジナルコード復帰命令を変更している図である．trace2 を生成中，インナーループ部を trace1 にリンクした後，trace1 のオリジナルコード復帰命令をリンクした直下の命令に帰ってくるように飛び先を変更している．これにより trace1 に実行が遷移しても trace2 に実行が戻ってくるようになる．

4.3 トレース配置

本システムはトレースを配置する際に，他のトレース配置位置を参照し，命令キャッシュインデックス衝突が起き難い位置を算出するアルゴリズムを使用している．しかしながら，実行中に多量に生成されるトレースを全て計算対象にすることは，本システムにおいて重大なオーバヘッドとなる．このため，参照するトレースの数を限定したアルゴリズムを提案する．考案アルゴリズムの配置位置インデックス算出要件を以下に示す．

- 前回生成したトレースの最後の命令の命令キャッシュインデックスに衝突していない．
- 最後に実行したトレースとインデックスが衝突していない．
- 全てのリンク先トレースの末尾のインデックスに衝突していない．

本アルゴリズムは，以上の条件を満たすインデックスを計算する．この要件は前回生成したトレース，前回実行したトレース，リンク先トレースのみを注意するものである．自分のリンク元（親）のトレースと，

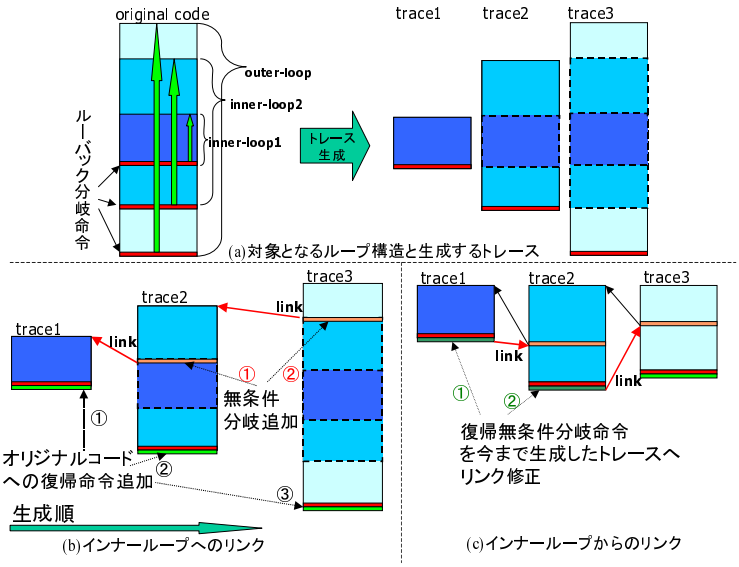

図 3 トレース加工工程

自分のリンク先（子）のトレースと，同一ループ内に存在する（兄弟）トレースのインデックスを避けるように考案されている．これは自分に対して近い続柄のトレースが局所的に連続して実行される予測と，1 つのトレースが十分に小さく，想定された続柄を超えた多段の入れ子構造がプログラム上で頻繁に登場しない予測に基づいている．

5. おわりに

本稿ではハードウェアサポートを用いたバイナリコードの動的最適化の一例を示した．提案ハードウェアは命令の振る舞いを検知しトラップを発生させる機能を有している．これにより，プログラム実行中の任意タイミングでトラップハンドラを起動させることが可能となり，ソフトウェアの振る舞いを解析することが可能となった．

提案最適化アルゴリズムはハードウェアが提供する情報を利用し，実行トレース生成とトレースの配置を行う．本アルゴリズムを採用することにより命令キャッシュアクセス数と命令キャッシュミス数を削減することが可能となる．

これらハードウェアと最適化アルゴリズムを備えたシステムはユーザに対して透過に実行されるため，通常の計算機使用方法を変更することなく，実行を重ねる毎により高速に実行するバイナリを生成することが可能となる．また，コンパイルを行う計算機ではなくバイナリを実行する計算機上での最適化であるため，命令キャッシュのブロックサイズ等の機種依存部分にも対応できる．この他にも，実行中にダイナミックリンクやダイナミッククラスローディングにより新たにバイナリに追加／修正が発生した場合にも対応できる．

現在シミュレーションにより，提案手法の予備評価を行ってる．予備評価は提案アルゴリズムのポテンシャルを計測するもので，CPU シミュレータに直接最適化アルゴリズムを実装し，生産物の性能を評価する．

参 考 文 献

1) Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia. Dynamo：A Transparent Dynamic Optimization System. SIGPLAN Conference on Programming Language Design and Implementation, pages 1–12, 2000.
2) 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.
3) Stephen J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive. Recompilation with On-Stack Replacement. Proceedings the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization,pages 241–252, 2003.
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.