動的バイナリ変換処理のためのユーザーレベルホットスポット検出システム

大津 金光 † 横田 隆史 † 馬場 敬信 †
† 宇都宮大学工学部情報工学科

1 はじめに

近年，1 個のチップ上に複数のプロセッサコアを集積したマルチコアプロセッサが広く普及しており，その潜在的な演算性能は飛躍的に向上している．しかしながら，その性能を活かすためにはプログラムコードの並列化 (マルチスレッド化) という一般的には困難とされる作業が必要であり，それがマルチコアプロセッサ本来の性能を発揮させる上での足枷となっている．

この状況を打開するために，我々は既存のプログラムコードの実行中に (動的に)，その機械命令コード (バイナリコード) を直接的に変換することで，プログラムの改良を行なう方法が有効であると考え，そのシステムの実現を目指している [1].

プログラムのバイナリコードを動的変換により改良して高速化を図る方式を採る場合，変換処理にかかる時間がプログラム本来の実行時間にそのまま上乗せされ，実行時間の増大すなわち性能低下を招くという問題が発生する．この問題を解決するためには，プログラム中で繰り返し頻繁に実行される部分 (以降，ホットスポットと呼ぶ) に限定したコード変換処理を行なう必要がある．たとえコード変換に時間を費したとしても，変換により改良されたコードが繰り返し実行されることで，全体として高速化が達成されることは十分に期待できる．

本稿では，動的なバイナリ変換処理によりプログラム性能の改善を行なうシステムの実現を前提として，プログラムのホットスポットをユーザーレベルで検出するシステムについて述べる．なお，本研究では一般に広く普及している x86 プロセッサをターゲットとし，実装実験の容易さの観点から OS として Linux が動作している環境を対象とする．

2 設計方針

本稿で述べるホットスポット検出システムの設計目標は，文献 [1] と同様に汎用性と低コストな実現を第一とする．そのため，OS が提供する標準的な機能を利用してユーザーレベルコードのみにより実現する．

ホットスポットを検出するためには，プロファイラによりプログラムの実行時の情報を収集する必要がある．ホットスポットを検出するためのプロファイリング方式として従来からよく利用されるものはサンプリングにより統計的に調べる方式である．この方法ではタイマーによる定期的な割込み (シグナル) により，割込み発生時点でのプログラムカウンタ (PC) の値を取得し，それを手掛りとして頻繁に実行されている部分を見付けだす．これをユーザーレベルコードのみで実現する場合，標準的な Linux OS 環境では最高で毎秒 100 回程度のサンプリングが可能である．しかしながら近年のプロセッサは高クロック化しており，毎秒 100 回程度のサンプリングでは数千万命令毎に一度の PC の値を取得することになるため，精度が極めて低い．

サンプリング方式に基づいたプロファイラの一つとしてよく知られている *Oprofile*[2] では主要な機能をカーネルモジュールとして実現することにより，大幅に短かい間隔でのサンプリングを実現している．しかしながら，この方式の場合，割込み処理のためにプロセッサコンテクストを頻繁に切り替えることになるため，その時間的コストが問題となる．そのため，たとえサンプリング間隔を短かく出来たとしてもカーネル内の割込みハンドラとユーザーコードの間を頻繁に行き来することになり，それに伴なうオーバヘッドにより性能が大きく低下することになる．このことから，プロファイル情報の精度を高める上ではプロセッサの動作モード (特権モード等) やコンテクストの切り替えを伴なわない方式が有利であると考えられる．

そこで，本システムではサンプリングによる方式を採らず，プログラムに対して計数を行なうためのコードを埋め込むコードインスツルメンテーション方式を採用する．計数のためのコードが実行されることで実行回数に計数し，その回数に基いてホットスポット検出を行なう．

通常，コードインスツルメンテーション方式はプログラムのコンパイル時に計数用のコードをオブジェクトコードに埋め込むことで実現されることが多い [3, 4] が，本システムではバイナリコードへのパッチ当て [1] により計数用コードをプログラムに対して付加することで実現する．

3 全体の処理の流れ

図 1 に本ホットスポット検出システムの全体の処理の流れを示す．本システムは，文献 [1] と同様に共有ライブラリの形で実現 (以降，本システムを *libdynprof.so* と呼ぶ) され，プロファイル対象となるプログラム (以降，ユーザーコードと呼ぶ) の実行開始時に動的リンクされることで起動される．

libdynprof.so は起動後すぐに自身の内部データ構造の初期化を行ない，ユーザーコードをバイナリコードレベルで解析を行なう．コード解析によりコード中に含まれる各サブルーチンのアドレス領域情報を取得し，ユーザーコードに含まれる全てのサブルーチンに対してそのエントリポイントを文献 [1] の手法によりジャンプ命令に置き換えて，実行回数を調査するためのコード (以降，計数コードと呼ぶ) を付加する．その後，ユーザーコード側に実行を戻す．

ユーザーコードの実行中にサブルーチン呼び出しが起こる度に，サブルーチン先頭に書込まれたジャンプ命令を介して計数コードが呼び出され，サブルーチンの実行回数が計数される．このとき，実行回数が一定

A User-Level Hot-Spot Detection System for Dynamic Binary Translation
†Kanemitsu Ootsu, Takashi Yokota, and Takanobu Baba
Department of Information Science, Faculty of Engineering, Utsunomiya University (†)

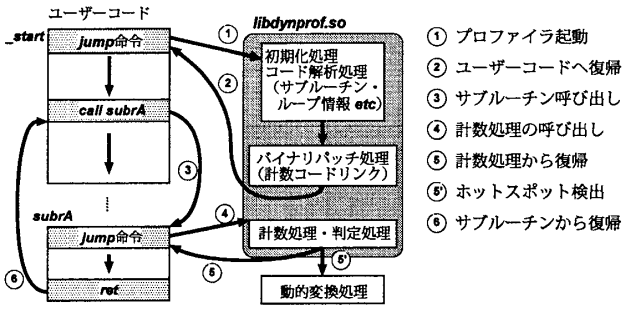

図 1: プロファイルシステムの処理の流れ

回数以上になったサブルーチンをホットスポットとして検出し，コード改良のための動的バイナリ変換処理の対象とする．

4 計数コード

図 2 に計数コードの構造とサブルーチン呼び出し時の処理の流れを示す．計数コードはサブルーチン毎に 1 対 1 で対応して動的に生成され，プロセスのヒープ領域上に確保された動的生成コード用バッファに格納される．

計数コードには，対応するサブルーチンが何回呼び出されたかを計数するコード (計数処理)，その回数が閾値以上であるかどうかを判定するコード (判定処理)，元のサブルーチンの先頭にあり計数コードへのジャンプ命令によって置き換えられた命令コード，サブルーチンに戻るためのジャンプ命令が含まれ，サブルーチン呼び出し時にはこの順に実行される．本システムでは，この計数コードを各サブルーチンに付加するため，サブルーチン先頭のいくつかの命令を上書きして計数コードへのジャンプ命令を書き込んでいる．その際に上書きされたいくつかの命令を計数コードの中に取り込み，計数コードから元のサブルーチンに戻る直前に実行することで，元のコードと同じ処理結果となるように対処している．

また，計数用に固有のデータ領域を持つ．これは，実行回数を記録する際のメモリアドレス計算を簡略化し，計数処理のコストを低減するためである．計数処理を行うために，通常はサブルーチンのアドレスからその実行回数を記録するためのメモリアドレスを求める計算を行うことになるが，この計算は例えば配列要素のアドレス計算やハッシュ関数の計算などを伴うため，そのコストは小さくない．計数コード毎に固有のデータ領域を持たせることで，これらのアドレス計算処理を省くことが出来る．なお，図中では計数用のデータ領域をコードと隣接した形で記述しているが，これは論理的な関係を示したものであり，物理的には隣接する必要はない．

計数コード中で実行回数と閾値の比較を行い，閾値を越えた場合にはそのサブルーチンをホットスポットとして検出し，ジャンプ命令により直接同一プロセス内の動的バイナリ変換処理ルーチンに実行を移し，ユーザーコードの改良の手続きを開始する．このように，同一プロセス内でジャンプ命令により変換処理を開始できるため，特権モードへの切り替えやプロセス切り替えなどの高コストの処理を伴うことなく速やかにホットスポットコードの改良のための動的バイナリ変換処理に移行することが出来る．

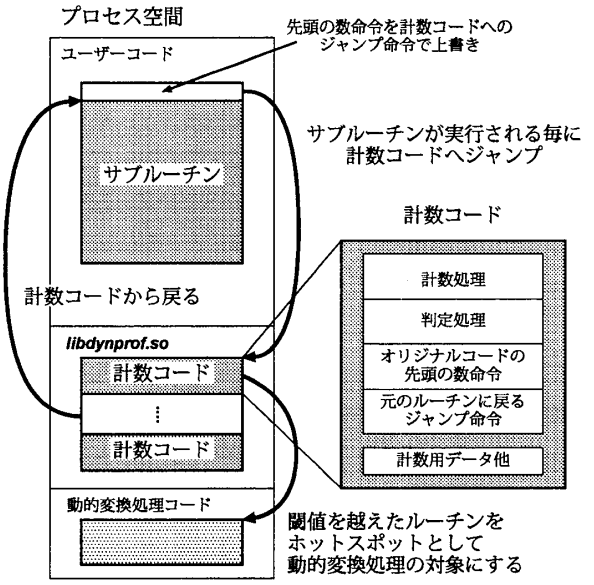

図 2: 計数コードの構造と処理の流れ

5 おわりに

動的なバイナリ変換処理によりプログラム性能の改善を行なうシステムの実現を前提として，プログラムのホットスポットとなるサブルーチンをユーザーレベルで検出するシステムについて述べた．

今後は，本システムにより検出したホットスポットコードの性能を改善する上で極めて有用となる，ループや基本ブロック等の実行時情報を取得できるプロファイラを開発する予定である．

謝辞 本研究は，一部日本学術振興会科学研究費補助金（基盤研究 (B)18300014，同 (C)19500037，若手研究 (B)17700047）および宇都宮大学重点推進研究プロジェクトの援助による．

参考文献

[1] 大津金光, 横田隆史, 馬場敬信, “動的リンク機構を利用したバイナリコードパッチ機能の設計”, 情報処理学会第 69 回全国大会講演論文集, 講演番号 3A-3, pp.1-31–1-32, 2007.

[2] John Levon, “OProfile internals”, 2003. http://oprofile.sourceforge.net/doc/internals/, (参照 2008-01-15).

[3] Susan L.Graham, Peter B.Kessler, and Marshall K.McKusick, “gprof: a Call Graph Execution Profiler”, Symposium on Compiler Construction, pp120-126, 1982.

[4] Amitabh Srivastava and Alan Eustace, “ATOM: a system for building customized program analysis tools”, Proc. of Programming Language Design and Implementation (PLDI1994), pp.196-205, 1994.