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

2008－HPC－117 (7)
2008／10／15

ループ並列化のためのループ階層構造を検出する実行時プロファイリング手法

佐藤幸紀[†1] 鈴木健一[†2] 中村維男[†3]

近年、HPC の分野のアプリケーションプログラムは年々その規模と複雑さを増してきている。同時に、それらのプログラムを HPC システムのこれからの方向性である超並列プロセッサやアクセラレータ上で適確かつ効率的に並列処理を行う必要がある。本論文においては、ループ並列化を支援するためにプログラム中のループ部分を検出し、そこから入れ子となるループ構造と関数の関係を把握する手法を提案する。実行時プロファイリングツール上に本手法を構築した場合、コンパイル後のバイナリコードからプログラム中に現れるループとその構造を検出可能であること明らかにし、既存の手法と差異を示す。

Run-time profiling technique to detect nested loop structure for loop level parallelization

Yukinori Sato,[†1] Ken-ichi Suzuki[†2] and Tadao Nakamura[†3]

Recently, application programs of the HPC area become larger and larger scale with ever-increasing complexity. At the same time, we have to perform parallel processing of those programs using massive parallel processors and accelerators. In this paper, we propose a profiling technique that detects the structure, with functions, of nested loops in order to support loop-level parallel processing. We demonstrate the proposed loop profiling and show the advantage compared with other profiling methods.

1. はじめに

長い間、HPC システムは世界のスーパーコンピュータ TOP500 のランキングが示すようにムーアの法則と歩調を合わせるように指数関数的な性能向上が達成されてきた。今後もこのペースを維持するという目標の下に様々な技術が生まれ成熟していくことが望まれている。しかしながら、これまでの HPC システムの性能向上は半導体のスケーリングに伴い向上を続けた CPU の動作周波数に大きく依存していた。近年は消費電力や信頼性の観点から CPU の動作周波数を以前のペースで向上させるのは困難といわれており、これまでの CPU の動作周波数の向上に由来しない革新的な技術が求められている。

CPU の動作周波数向上に由来しない性能向上を実現する技術として、多数のプロセッサコアを単一チップ上に持つマルチコアプロセッサを大規模かつ高速に接続した超並列処理技術や、SIMD や FPGA といった強力な演算能力をもつアクセラレータを利用してプログラムの特定部分を高速化する技術などが考えられている。超並列マルチコアプロセッサやアクセラレータにより性能向上を目指した場合は、そのようなハードウェアを実現することより、むしろその理論性能に近づけるように効果的に利用することを支援する技術が重要になると予想される。現時点でも Cell プロセッサや FPGA は非常に強力な演算能力を実現しているが、それらを引き出すためには、非常に高度で時間のかかるチューニングを行う必要がある。

一方、HPC の分野のアプリケーションプログラムは年々コードの規模や複雑さを増しており、並列化やチューニングを適確かつ効率的に行うことが困難となってきている。科学技術計算の分野のシミュレーションは精度の向上や様々な物理過程を盛り込むためにコード量とデータ量共に大規模化の一途を辿っている。また、大規模化に対応するために多くの場合において他の人が書いたサブルーチンやライブラリが利用される。従って、プログラムの全体構成を把握したり、それらを修正したりすることが年々困難となってきている。将来的には、自ら手がけたプログラムといえども、その規模と複雑さが増すにつれて並列化や最適化を行う

†1 北陸先端科学技術大学院大学 情報科学センター
Center for Information Science, Japan Advanced Institute of Science and Technology
†2 東北工業大学
Tohoku Institute of Technology
†3 慶應義塾大学
Keio University

ことは非常に困難となることが予想される。

並列化や最適化が行いにくい原因のひとつとして、ソフトウェアとハードウェアの両者における考え方のギャップがあげられる。一般的にプログラマは関数あるいはサブルーチンを処理の基本単位と考えて開発を進める。すなわち、ソフトフェアから見るとプログラムは関数の集合と考えられている。一方で、ハードウェア上でコードが実行される場合においては、その処理の大半がプログラムのループで占められることが一般的である。すなわち、ハードウェアから見るとプログラムは関数ではなくループの集合とみなすことができる。この両者のギャップは現在広く利用されているプロファイリングツールにも見られる。例えば、代表的なプロファイリングツールである gprof は実行時間に占める割合が高い関数を表示する[1]。しかしながら、実際に並列化や最適化を考える上では、コード中のループ構造を実際に実行する並列なハードウェアに対応付けることが重要なポイントとなるため、プロファイリングで得られる情報が関数レベルのみというのでは不十分と感じる場面が多い。

本研究では、ループレベルの並列化を適確に行うことを支援する目的で、プログラム中に現れるループとその入れ子構造と関数の関係をバイナリコードを実行させることにより検出する手法を提案する。実行時プロファイリングツールである Pin[2] を利用してコンパイル済みのコードを実行させる際にプロファイリングを行う。実行時プロファイリングにおいて関数呼び出し以外の成立する後方分岐命令に着目し、ループ部分を検出する。このような原理に基づくループレベルのプロファイリングと関数レベルのプロファイリングで得られる情報を比較し、ループレベルのプロファイリングの利点を示す。

本論文の構成は以下の通りである。2 節では関連する既存のプロファイリング手法を説明する。3 節ではループと関数の位置関係を正確に検出するプロファイリング手法を提案する。4 節では提案するループレベルプロファイリングのその基礎的評価を行った様子について述べる。5 節は結論である。

2. 既存のプロファイリング手法

2.1 プロファイリング手法の概要

プログラムのプロファイリングとは特定のイベントがプログラムの実行中に起こった頻度をカウントすることによりプログラムの特徴を探りプログラムの全体像を把握することである。伝統的にプロファイリングは本来の実行結果に影響を与えないようにソフトウェアレベルでコードのフロー解析や頻度計測のためのコード (Insrumentation) をプログラムに付加することにより実現されてきた。

プロファイリングにより得られる結果はどのようなイベントに着目したかにより様々な特徴がある。プログラム中に実行された関数呼び出しに着目する場合、gprof[1] のように各関数が実行された時間の全体における割合が把握可能となる。プログラム中のコントロールフローの遷移に着目するエッジプロファイリングは、その遷移を引き起こすプログラム中の条件分岐命令やジャンプ命令を観測することにより実現可能である[3]。エッジプロファイリングは比較的単純なプロファイリングではあるが、エッジの実行頻度をカウントするだけではその時間的遷移が分からないため、プログラム中の頻繁に実行される部分である Hotspot を検出するのには適していないといわれている。

Ball らはプログラム中の Hotspot を的確に把握するために特定の区間内で実行される複数のエッジを包含した部分を関数内の非循環パスに着目して検出するパスプロファイリングを提案している[4]。関数内の非循環パスとは成立する後方分岐命令や関数からのリターン命令で終わる区間であり、そのパスの始まりは後方分岐命令の飛び先の命令や関数の始まりの命令である。Ball らのパスプロファイリングは関数を超えた場合にも拡張することが可能であり、その場合においてパスは成立する後方分岐命令の飛び先の命令から始まり、パスの終端は関数呼び出し以外の成立する後方分岐命令と定義される[5]。

以上のようなエッジプロファイリングやパスプロファイリングはコンパイラがコードを最適化する際の情報として利用するためにプログラムの情報を詳細にそして正確に抽出しすることを主な目的としている。エッジプロファイリングやパスプロファイリングを用いループを検出することは可能である。しかしながら、ループのみを検出するためには詳細なパスの情報は必要ないため、ループ検出に特化した手法が独立して提案されている。

2.2 ループ検出手法とプロファイリング

ループを検出する手法としてプログラムの実行時に ssb(short backward branch) に着目する手法を Gordon-ross らが提案している[6]。この手法は命令列を循環させる成立する後方分岐命令がループの構成要素として本質的なものであるという事象に基づいている。ここで、short の意味するところは関数呼び出しのための後方分岐命令以外は分岐元と分岐先のオフセットは比較的小さな範囲に収まるであろうということに由来している。

ループを検出する他の方法として、ループの先頭命令に着目するという手法が提案されている[5]。ループは後方分岐命令が実行された後に必ず飛び先であるループの先頭の命令が実行されるという事象に基づき、ループの先頭命令を認識することによりループを検出する。

図 1 はループ検出の方法の例を示す。図中の箱は基本ブロックを示し、矢印はコントロールフローを示し

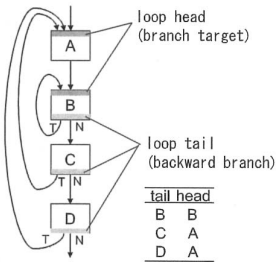

図 1 ループ検出の方法

ている。ここで、基本ブロックはプログラムの実行時に得られる動的基本ブロック (DBBL) を仮定している。ループ検出は図中において灰色で強調されている基本ブロックの先頭と最後尾の命令に着目し行われる。後方分岐命令 sbb[6] に着目したループ検出方法を用いると成立する後方分岐命令に対応する B、C、D を愁嘆とする 3 つのループが検出される。一方、ループの先頭命令に着目する検出方法[5] を用いると A と B から始まる 2 つのループが検出される。ループの先頭命令に着目した場合、同一の飛び先を持つ後方分岐命令が複数個存在する場合は同一ループとして分類されるため sbb に着目した場合と比べて検出される個数が少なくなる。

Moseley らはループを中心としたプロファイリングを提案している[7]。Moseley の手法はループを検出するために DBBL を単位として関数毎に DBBL のスタックを作る。DBBL が実行される毎にスタックに DBBL をプッシュしていく。現在実行されている DBBL が既にスタックに入っている DBBL と同一である DBBL が出現した場合には既に入っていた同一の DBBL までの区間をループとみなし、既に入っている DBBL を含みそれ以降の要素をスタックからポップする。本手法はループの先頭命令に着目した場合とほぼ同等の意味を持つと考えることができる。

3. 提案するループ階層構造検出法

3.1 従来からの並列化手法におけるギャップ

HPC システムにおいては、システムの理論性能を引き出すプログラムが開発できるかが成功の鍵を握ると考えられる。その一方で、HPC 分野のアプリケーションプログラムは年々コードの規模や複雑さを増しており、プログラムの全体構成を把握したり、それらを修正したりすることが年々困難となってきている。

プログラムの大規模化や複雑化の問題はエンタープライズ系や組込み系のアプリケーションで顕著であると指摘されてきている。このような背景の下、プログラマの生産性を高めることのみが目的であるならばソフトウェアエンジニアリングを実践することによりある程度は達成可能である。ソフトウェアエンジニアリングの観点では、ソースコードは可読性が高く、デバッグが容易で、メンテナンス性に富み、再利用性が高いことが要求されており、これらを注意することで生産性の問題にはある程度は対処可能であると考えられる[8]。

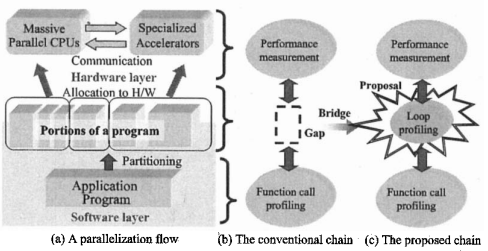

図 2 プロファイリングによる並列化支援

しかしながら、このような要求は結果として頻繁な制御フローの遷移や小さな関数を多用するということにつながる。複雑な制御フローの元で処理速度を向上させるための最適化を施すことは困難である。また、生産的に大規模化を行うためには他の人が書いたサブルーチンやライブラリを利用する必要がある。さらには、エラー処理などの部分のためにめったに使われない部分がコードの大半を占めるようになる。このような複雑なプログラムから並列性を抽出し並列処理することは依存関係を解析する点で非常に困難である。

複雑なプログラムの並列化や最適化が行いにくい理由として、ソフトウェアとハードウェアの間に存在するギャップが原因として挙げられる。一般的にプログラマは関数を処理の基本単位としてプログラムを作成するが、ハードウェアで実際のコードが実行される場合は処理時間の大半をループ部分が占めるためループを処理の単位と考えるのが妥当である。すなわち、注目するべきポイントが関数とループというギャップが存在することになる。

図 2 にプロファイリングによる並列化支援の全体像を示す。図 2(a) には並列化を行う際のフローを、図 2(b) には従来からの並列化支援のためのツール群を示す。従来からのフローは gprof などに代表される関数レベルのプロファイラにより実行時間に占める割合が高い関数を把握することから始まる。関数レベルのプロファイラの結果を踏まえて、ソースコード中の関数とループの関係がつかめている場合はプログラムを的確に分割し、ハードウェアリソース間の通信も考慮しながら適切なリソースに割り付けることを行う。ハードウェアに割り付けられたプログラムは各フェーズにかかる時間を実測することによりプログラムが適切にマッピングされているかという目安を得ることができる。

しかしながら、プログラムの規模と複雑さが増すにつれて並列化や最適化を適切に行うために必要十分なだけループと関数の関係を理解することは非常に困

難となりつつある。そこで、本論文では図 2(c) に示すように提案するループプロファイリングによりこのギャップを埋めることを目指す。

3.2 ループ階層構造の検出と関数の関係

目標としている並列化を支援しつつループと関数のギャップを埋めることが可能なループプロファイリングを構築するためには、(1) ループの入れ子構造を把握可能であること、(2) プログラム中のループの位置と関数が呼び出される位置の関係が把握可能なことの 2 点を実現することが必要である。

本論文においては、プログラム中のループの位置と関数が呼び出される位置の関係が把握できるように 2.2 節で議論した関数呼び出し以外の成立する後方分岐命令に着目したループ検出法[6] とループの先頭命令に着目する検出方法[5] の両者を融合した手法によりループを検出する。具体的にはループが開始される位置である成立する後方分岐命令の飛び先の命令とループが終了する位置である成立する後方分岐命令の両者の位置を記録することにより、ループの階層構造を正確に検出する。ここでプログラム中の命令の位置とはその命令が格納されるアドレスに対応する。また、ループの位置に加えて、関数が呼び出される位置も正確に把握するために、関数が呼び出される位置も記録し、ループ内部で関数が呼び出されているのかといった相対的な位置情報も検出する。さらに、ループプロファイリングの段階である程度の精度でプログラムの実行時間が見積もれるように各ループにて実行される命令数やループの出現頻度をカウントする。

コンパイル後のバイナリコードを利用してループ階層を検出するために、本論文では実行時プロファイリングを実現するツールを用いる。近年の複雑化するアプリケーションプログラムにおいてはライブラリで提供されるコードを利用するケースが増えていること、OpenMP などコンパイル時に特別なオプションをつけることが必要な環境ではコンパイル時に Instrumentation のためのコードを挿入することにより実行性能の予測や実行結果に予想外の影響を与える可能性もあることを考慮に入れて、実際に本番環境で利用するライブラリや並列化されたコードを用いてループ階層構造を検出できる環境として設計を行った。

Moseley ら[7] も実行時プロファイリングによりループを検出する方法を提案しているが、実行時に基本ブロック (DBBL) を単位として解析を行うためループの入れ子構造についての考慮は行われていない。また、関数ごとに DBBL のスタックを設けているが、ある関数内におけるループの位置と関数が呼ばれる位置の情報についても同様に考慮されていないため検出することができない。

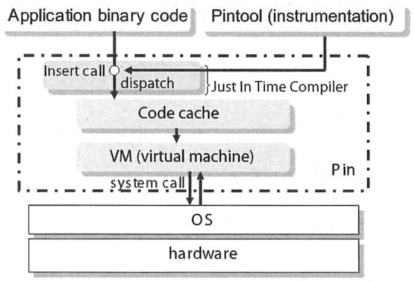

図 3 実行時プロファイリングツール Pin の概要

4. 評価実験

本節では提案するループ階層構造の検出手法を実現するための基盤となるツールとその実行環境を説明した後に、評価実験としてループレベルのプロファイリングと関数レベルのプロファイリングで得られる情報を比較し、ループレベルのプロファイリングの利点を議論する。

4.1 Pin の概要

本論文で提案するループ階層構造の検出手法を実現するために、実行時プロファイリングの代表的なツールである Pin[2] を用いた。図 3 に Pin のソフトウェア構成要素の概要を示す。Pin は図中の点線で囲まれた VM(仮想マシン) と VM に命令を提供する機構から構成される。プロファイリングされるバイナリコードはコードキャッシュと呼ばれる領域に保持されてから VM で実行される。バイナリコードがコードキャッシュに格納される前に JustInTime コンパイラの原理でバイナリコードに Instrumentation のための任意のコードを挿入することにより様々な粒度で任意のプロファイリングが可能である。Instrumentation のための任意のコードを挿入するためにユーザーは API を通して Pintool と呼ばれる Instrumentation のための任意のコードを事前に用意しておく必要がある。本実験においては Instrumentation のための任意のコードとしてループ階層構造の検出と関数の関係の把握するために必要な情報を抽出するコードを生成した。

4.2 実験環境

提案するループプロファイリング手法を SGI Altix350 上に構築した。Altix350 の構成は Intel Itanium2 CPU、Intel Compiler v8.1、RedHat Linux + SGI ProPack 3SP6 である。また、利用した Pin のバージョンは 2.3(17159) であり、Itanium 用の構成を用いた。評価実験を行うためのベンチマークプログラムとして、NAS Parallel Benchmark v3.3 の Serial 版にある BT、CG、DC、EP、FT、IS、LU、MG、SP、UA を利用した。

```
ID     InstAddr          B?  [%]    cnt
0 4000000000001f2f0
2 4000000000001f5c0
1 4000000000001f6e0
1 4000000000001f8d2    B   1.8     56100
2 4000000000001f8e2    B   0.1     4590
4 4000000000001fe10
3 4000000000020240
3 4000000000021352     B  13.5     51000
4 4000000000021422     B   1.2     4590
0 40000000000214e2     B   0.2     510
```

図 4 LU の関数 buts_におけるループ検出結果

```
ID     InstAddr          B?  [%]    cnt
3 4000000000007740
2 40000000000078e0
1 4000000000007a70
2 4000000000007a82     B   0.0     180
3 4000000000007ac2     B   0.0     18
0 4000000000007b40
C 4000000000007c62         exact_
C 4000000000007c82         exact_
C 4000000000007ca2         exact_
C 4000000000007cc2         exact_
C 4000000000007ce2         exact_
C 4000000000007d02         exact_
1 4000000000008212     B   0.2     200
0 4000000000008232     B   1.7     1800
```

図 5 LU の関数 setiv_におけるループ検出結果

4.3 評価結果

図 4 と図 5 に提案するループ検出手法を用いてループプロファイリングを行った中で特徴的であった 2 つの関数の結果を示す。入力としたバイナリコードは LU でデータセットは S を使用した。図中の InstAddr は昇順に並んだ命令アドレスであり、ID はループの個別番号を示す。その行にある命令が後方分岐命令の場合、B?の列に **B** と表示され、関数呼び出しの場合は ID の列に C と表示される。ID を持ち **B** の表示がない命令がループの先頭命令であり、それと同一の ID を持ち **B** の表示がある命令がそのループの最後尾の命令となる。図中の割合 [%] は前回成立した後方分岐命令から今回成立した後方分岐命令までに実行された命令数のループ毎の総和とプログラムの全実行命令中に占める割合である。列の右端の値はそのループが反復された回数の総和であり、後方分岐の所に表示した。

図 4 より buts_という関数内には 5 つのループから構成されることが分かる。また、ID=0 のループが最外ループであり、連続してその ID が検出される ID=1 と ID=3 のループは最内ループであり、ループの反復回数がループ中で多いことが分かる。図 5 より setiv_という関数内には 4 つのループがあり、最内ループは ID=0 であることが分かる。他のループは ID=0 のループのパラメータを更新するためのループであると考えられる。また、ループの位置だけではく関数が呼び出される位置も表示している。ループと関数の位置を同時に出力することにより、最内ループにおいて関数が呼び出されることも分かる。

表 1 LU におけるループプロファイリングのオーバーヘッド

Data set	Native T_{ex}[s]	LoopProf. T_{ex}[s]	Overhead
S	0.05	8.34	167x
W	11.00	263.43	24x
A	112.43	1690.18	15x

表 2 関数毎のプロファイリング結果の比較 (LU.S 実行時)

関数レベル		ループレベル		
Name	Inst.[%]	Name	Inst.[%]	#loops
exact_	2.1	setiv_	1.9	4
ssor_	1.3	ssor_	1.6	16
rhs_	26.0	rhs_	25.9	28
jacld_	19.7	jacld_	19.5	2
blts_	15.6	blts_	15.7	4
jacu_	17.3	jacu_	17.1	2
buts_	16.8	buts_	16.9	5
Sum	98.8	Sum	98.6	61

次に、本論文で提案するループ階層構造の検出方法についての実行時間の面でのオーバーヘッドの評価を行う。評価は Pin を用いてループ検出を行いつつプログラムを実行する時間とループ検出を行わないでバイナリコードを直接ネーティブで実行したときの処理時間を比較することにより行う。

表 1 にバイナリコードを直接ネーティブで実行したときの時間とループプロファイリングを行ったときの時間を LU を用いて比較した結果を示す。各データセットのサイズは S、W、A の順で大きくなる。結果より、データサイズが大きくなるほどループプロファイリングのオーバーヘッドは相対的に小さくなることが観測された。データセットが A のときでループプロファイリングは 15 倍程度の速度低下により可能であることが分かった。

次の評価実験として、ループレベルと関数レベルのプロファイリングにより得られる情報の比較を行った。関数レベルのプロファイリングも同様に Pin を用いて Instrumentation のためのコードを付加することにより実行時に実施した。ループレベルのプロファイリング結果を関数レベルと比較するために、ループレベルのプロファイリング時に関数毎に関数内にある各ループで実行された総命令数の和を求めて関数毎の命令の総和とし、関数レベルのプロファイリングにより求めた関数毎に実行された命令の総和と比較した。

表 2 に LU を実行した場合の関数レベルとループレベルのプロファイリングにおける関数毎に検出された実行命令数の割合を示す。この比較においては実行割合が 1%を超える関数のみを比較した。LU を実行した場合は両者で検出された関数の数は 7 つで等しくなった。また、各関数の実効命令の差は最大でも 0.3%であり、ほぼ等しいといえることが分かった。しかしながら、関数レベルの exact_という関数はループレベルのプロファイリングでは setiv_として検出された。この理由は図 5 で示したように setiv_にある最内ループ

表 3 各レベル毎の検出した関数の数

Name	Func level	Loop level
BT	10	6
CG	2	2
DC	16	14
EP	2	2
FT	5	5
IS	4	3
LU	7	7
MG	9	9
SP	10	10
UA	9	9

においてexact_が呼び出されるためである。

ループレベルのプロファイリングにおいてループとして定義される部分は前回成立した後方分岐命令から今回成立した後方分岐命令までであった。従って、関数exact_の内部にループ構造がない限り関数呼び出しが行われたループのある関数における実行命令数としてカウントされるため、主要な関数として検出される関数の名前が異なるという結果となる。この事象は提案するループプロファイリングの特徴を非常によく表している事象といえる。

表3に各プロファイリングレベルにより検出した実行割合が1%を超える関数の数をベンチマークプログラム毎に集計した結果を示す。結果よりループレベルのプロファイリングを行ったほうが実行時間に影響を与える関数の数が少なく検出されることが観測される。この理由はループ内部で関数呼び出しがされた場合、呼び出された関数の内部にループがない限り呼び出したループのある関数における実行命令数としてカウントされているためである。ループプロファイリングを用いるとループ内部に関数があることは図5のように検出することができるため、主要なループの構成を関数を含めて適切に把握できるツールであるといえる。したがって、提案するループプロファイリング手法はハードウェアとソフトウェアの考え方のギャップを感じることなく並列化や最適化をスムーズに行うことを支援するツールであるといえる。

5. 結　　論

本論文では、並列化やチューニングを適確に行うことを支援する目的で、コンパイル後のバイナリコードを用いてその実行時にプログラム中に現れるループとその入れ子構造、そして関数呼び出しの位置を検出する手法を提案し、基礎的評価を行った。評価実験の結果より、提案するループレベルのプロファイリングは既存の関数レベルのプロファイリングで得られる情報よりとくらべて、ハードウェアとソフトウェアの考え方のギャップを埋める役割を果たし、並列化や最適化をスムーズに行うことを支援するツールであることを示した。

参 考 文 献

1) SusanL. Graham, PeterB. Kessler, and MarshallK. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pp. 120–126, 1982.
2) Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, VijayJanapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 190–200, 2005.
3) JamesE. Smith and Ravi Nain. *Virtual Machines: Versatile Platforms For Systems And Processes.* The Morgan Kaufmann Publishers, 2005.
4) Thomas Ball and JamesR. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pp. 46–57, 1996.
5) Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. In *Proceedings of the Symposium. on Architectural Support for Programming Languages and. Operating Systems*, pp. 202–211, 2000.
6) Ann Gordon-Ross and Frank Vahid. Frequent loop detection using efficient nonintrusive onchip hardware. *IEEE Transactions on Computers*, Vol.54, No.10, pp. 1203–1215, 2005.
7) Tipp Moseley, DanielA. Connors, Dirk Grunwald, and Ramesh Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th international conference on Computing frontiers*, pp. 143–152, 2007.
8) Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th annual international symposium on Computer architecture*, pp. 174–185, 2007.