

マルチスレッド化のためのサンプリング情報に基づくホットループ検出

青木 政人[†] 大津 金光[†]
横田 隆史[†] 馬場 敬信[†]

我々は、シングルスレッドコードからマルチスレッドコードへのバイナリ変換システムを提案し、実現に向けての研究を進めている。一般的にコンパイラなどは実行前にコード最適化を行っている。しかし、実行時に解析を行うことで更なる最適化が可能となる。実行時最適化では、その対象をプログラムにおいて実行割合の高い部分であるホットスポットに限定することでオーバーヘッドの軽減が可能となり、最大限の性能向上が可能となる。本研究では、実行時最適化処理の一環として、ホットスポットの特定をするために、サンプリングによるプロファイリングシステムを試作した。サンプリングによる情報を基にホットループを検出、マルチスレッド化を前提として、ホットループの解析を行い性能向上可能性を検討する。その結果1%程度のオーバーヘッドでプロファイル情報の取得が可能であり、またホットループを解析することで、ループの高速化可能性を判定することができることを示す。

Hot-loop Detection based on Sampling Information for Multithreading.

MASATO AOKI,[†] KANEMITSU OOTSU,[†] TAKASHI YOKOTA[†]
and TAKANOBU BABA[†]

We have proposed a binary translation system that can translate single-thread application codes into multithreaded ones. This translation system optimizes application codes statically. However, we can achieve further optimization by using run-time information. To reduce overheads, the run-time optimization should be performed only on the hot-spot codes, that are the frequently executed codes in the application.

We have developed a profiling system, that is a part of our proposed run-time optimization system. It utilizes sampling information to detect hot-spot codes. We evaluate the detected hot-spots, the overhead caused by sampling processes, and the possibilities of the speedup by run-time optimization. The results show that the profiling system can detect hot-spot codes at less than 1 % overhead.

1. はじめに

近年、マルチスレッド実行によるスレッドレベル並列性 (TLP: Thread-Level Parallelism) の活用により、性能を向上させる事が必須となりつつある。また一般的には、シングルスレッドコードからマルチスレッドコードを生成する方法としては、ソースコードレベルにおいて行うことが最も有効であると考えられている。しかしながら、ソースコードを参照不可能なアプリケーションにおいては、マルチスレッド化は行なうことができない問題が現われる。そこで我々は、シングルスレッドコードからマルチスレッドコードへのバイナリ変換システムを提案し、実現に向けての研究を進めている¹⁾。

一般的なコンパイラでは、主に実行前にコードの静的な解析を行い最適化を行っている。しかし静的な解析ではプログラムのバイナリの全てを解析することに

限界があるため、本バイナリ変換システムでは実行時においても、解析を行い最適化を施すことを目標としている。

実行時最適化を行う際にはマルチスレッド化による性能向上を最大限に得るために、マルチスレッド化にかかるオーバーヘッドを低く抑える必要がある。実行時に最適化を行う場合、その最適化にかかる作業時間全てがオーバーヘッドとなる。そのためプログラム全体に対して実行時にマルチスレッド化処理を行ったのでは、最適化処理自体による実行時のオーバーヘッドが大きくなってしまい、ゆえにホットスポットのみに限定したマルチスレッド化を行う必要がある。

しかし一般的にプログラムは実行時に挙動が変化するため、実行前の静的な解析ではホットスポットの検出は困難である、そのため実行時に挙動の解析を行うことで、ホットスポットが検出できる。

ホットスポットはプログラム内のある箇所が定められた回数以上実行された場合に検出される、その回数が閾値である。ホットスポットを検出するためには、判定のための閾値が問題となってくる。ホットスポット

[†] 宇都宮大学工学部情報工学科

Department of Information Science, Faculty of Engineering, Utsunomiya University

検出の際、プログラムは様々な挙動を示すため、適切な閾値を設定することによって、プログラム内で多くの実行割合を占めるホットスポットが検出でき、検出されたホットスポットがマルチスレッド化による高速化が可能であれば、その効果はより大きいものとなる。

さらに、マルチスレッド化を行った際に、各スレッドが小さな作業量しか持たない場合、スレッドの制御にかかるコストの割合が大きくなってしまい、大きな性能向上が望めない。そのため各スレッドに適度な作業量を持たせる必要がある。そのためマルチスレッド化の対象はバイナリを解析することで検出が可能であり、あるていど大きな作業量を持った、ループを単位として検出する。

本稿ではサンプリングにより実行プロファイル情報を収集するシステムを試作した。サンプリングシステムでは低オーバーヘッドでプロファイル情報を検出でき、ホットループの検出が可能である。本システムはハードウェアクロックを利用してプログラムの実行に割り込みをかけ、その度にPCの値をサンプリングすることで実行プロファイルデータを収集する。

マルチスレッド実行において、検出されたホットループが各イテレーション間に大きな依存を持つ場合、マルチスレッド実行の際の各スレッド実行のオーバーラップの度合であるオーバーラップ度が十分に得られず速度向上が達成できないという問題が現れてくる。そのためプログラム内のループについての解析を行い、レジスタの書き込みや読み出しを手がかりにして、ループの実行オーバーラップ度を調査することでループのマルチスレッド化による高速化の可能性についての判定を行う。

2. システムの概要

図1に、本研究で提案しているシステムの全体構成を示す。本システムは、シングルスレッドコードで実

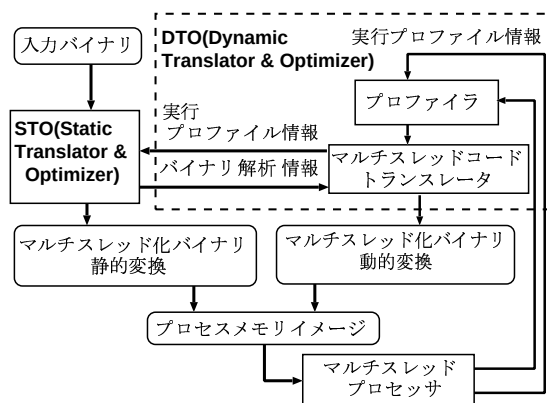


図1 システム全体構成

行されるプログラムを静的・動的に解析を行い、マルチスレッド化を含む最適化を施し、マルチスレッド実行することにより実行性能を向上させる。

このシステムは大きく分けて、プログラム実行前に動

作するSTO(Static Translator and Optimizer)、プログラム実行時に動作するDTO(Dynamic Translator and Optimizer)、の2つから構成される。

まずプログラム実行前に、オリジナルのシングルスレッドバイナリコードに対して、STOが静的に基本ブロック・ループ情報などを解析し、ループのイテレーションのサイズ、イテレーション間のレジスタ依存を調査し、イテレーション間のオーバーラップ度を算出した上で、マルチスレッド化による高速化の可能性についての解析を行う。その結果マルチスレッド化による高速化が可能と判定された場合のみマルチスレッド化を行う。

DTOはマルチスレッドプロセッサの実行時に、その実行プロファイル情報を収集する。そして定期的にまたは特定のイベントにより、DTOに実行が遷移する。

DTOは実行が開始されると、プロファイラがサンプリングによって収集したプロファイル情報を基に、プログラムのホットスポットを検出する。検出されたホットスポットに対してDTOは、オリジナルコードをSTOが解析した情報をもとに、非投機マルチスレッド化による高速化の可能性の有無を判定する。高速化が可能であればマルチスレッド化による高速化を図る。検出されたホットスポットが非投機マルチスレッド化では高速化しない場合、投機スレッド実行を含めた最適化による高速化を目指した処理を行うために、パスプロファイルなどの詳細なプロファイリングを行う。

このようにして、本システムはシングルスレッドバイナリコードをマルチスレッド実行することにより、実行性能を向上させる。

2.1 実行時最適化の問題点

本システムのDTOはアプリケーション実行中に動作するため、STOには存在しない問題がある。実行前における静的な解析の場合、解析やマルチスレッド処理にかかる時間を考慮する必要は無いため、静的に出来る限りのあらゆる処理を行うことが出来る。しかしDTOではマルチスレッド化処理及びプロファイリング処理にかかる時間がオーバーヘッドとなるため、処理時間及び処理量を極力小さくする必要がある。たとえば、DTOがマルチスレッド化したコードが、マルチスレッド化前のコードより実行時間が短かったとしても、マルチスレッド化するための時間が長ければマルチスレッド化を行う意味がなくなるからである。そのためSTOより受け取った情報を基にDTOはホットループのなかでもマルチスレッド化処理によるオーバーヘッドを隠す程度には高速化が見込まれるホットループのみに絞ったマルチスレッド化をする必要がある。

本システムではDTOが実行時に最適化を行う際に必要となる実行バイナリの命令コード、制御フロー、データフローの各種解析情報について、STOからDTOに受け渡すことで、DTOの実行時による解析量を軽減する事で動作コストを削減する。

検出されたホットループが非投機のマルチスレッド実行で高速化が望めない場合、特定のパスに沿って投機的にマルチスレッド実行を行うことでさらなる性能

向上の可能性を引き出すことが出来ると考えられる。しかし投機的マルチスレッド実行のためには、パスの挙動の解析が必要となるため、パスプロファイルシステムによる挙動解析が必要となる。

2.2 マルチスレッド実行モデル

本研究ではスレッドパイプラインモデル³⁾に基づいてマルチスレッド化を行う。図2にスレッドパイプラインの実行状態を示す。

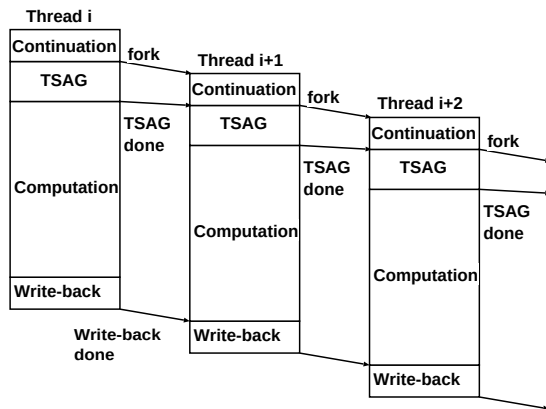


図2 スレッドパイプライン

マルチスレッド実行において、各スレッドはプログラムの制御フローに沿って分割されたコードを実行する。コードの分割は、各スレッドによって実行されるコードサイズの均等化を考慮して、プログラム内のループ構造に着目し、ループの各イテレーションを単位としてマルチスレッド化処理を行う。ただし、1イテレーションが必ずしも1スレッドになるわけではない。

なお、スレッド間データ依存については、メモリアクセスにおける依存が起こった場合に同期をとる機能を持つ**Memory Buffer**に対して通知する。Memory Buffer には、自スレッドと先行するスレッドとの間でデータ依存となるメモリアクセスのアドレス全てのエントリのテーブルが存在し、実行時に Memory Buffer は、自スレッドでのメモリアクセスについて常に監視を行い、必要時にはスレッドの実行を停止させる。

ループのイテレーション単位により対象コードを分割後、マルチスレッド実行時の各スレッド内は図2に示すように、Continuation, TSAG(Target Store Address Generation), Computation, Write-backの4つのステージに分割される。

Continuation ステージでは、次スレッド開始時に必要となるループ変数値の計算を行い、その後に次スレッドを生成する。

TSAG ステージでは、静的に解析されたスレッド間のデータ依存となるメモリアクセスのアドレスを、**Memory Buffer**に対して通知する。このとき、直前スレッドのTSAG ステージが終了するまで、自スレッドのTSAG ステージを開始してはならない。

Computation ステージでは、計算本体のコードを実行する。この際に発生するメモリアクセスは、Memory Bufferによって監視され、スレッド間で依存があるメ

モリアクセスが発生した場合にはスレッド間で同期を取りながら処理を行う。

最後のWrite-back ステージでは、スレッド終了処理として前スレッドの終了処理が完了した後、現スレッドの処理結果の書き出しを行う。

3. サンプリングシステム

本研究では、実行プロファイル情報を取得・解析し動的最適化に必要な情報を取得しつつ、ホットループ検出のためのオーバーヘッドを低く抑えるため、一定期間毎にPCをサンプリングすることによりホットループを検出するサンプリングシステムを試作した。

一定時間毎にプロファイル情報を取得することで、常時プロファイルを取得する手法に比べ大幅にオーバーヘッドが削減できる。また、本システムではサンプリング間隔の設定が可能であり、これによってプロファイル情報の量やプロファイル情報の取得にかかるオーバーヘッドを調節することができる。サンプリングの結果、取得したプロファイル情報を基に検出されたホットループに着目し、そのホットループが高速化可能であれば大きな速度向上を得ることが出来る。

3.1 ホットループ検出アルゴリズム

サンプリングによって実行プロファイル情報(PCの値)を取得し、ホットループの検出を行うまでの流れを図3に示す。まずハードウェアクロックを利用して

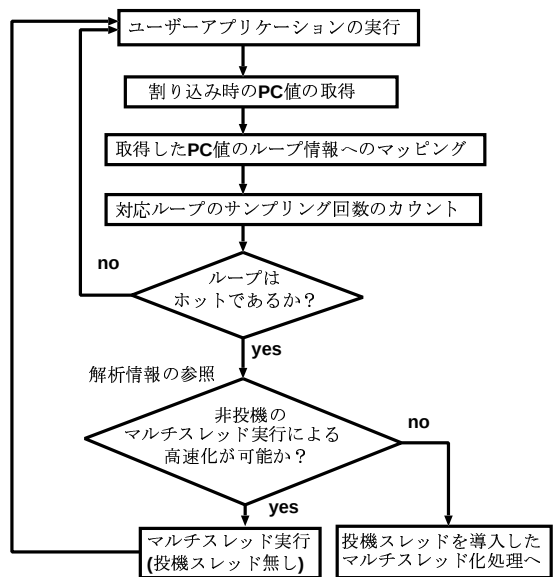


図3 ホットスポット検出フロー

タイマ割り込みをかけ、そのときのPCの値を取得する。そしてそのPCの値がSTOにより解析された情報から、どこのループに属するかを検索し、サンプリングシステムが管理しているループ情報を格納したデータベース内のループのサンプリング回数をインクリメントする。ループ情報としては、ループの開始アドレス、終了アドレス、ループのイテレーション内のレジ

スタ依存によるスレッド間のオーバーラップ度、ループのサイズ、そしてループがサンプリングされた回数を記録するカウンタがある。

対応するループが発見され、サンプリング回数がカウントされた後に、あらかじめ設定された閾値を越えて実行された場合、そのループをホットループとして検出する。

検出されたホットループは、実行前に解析されたループ情報を基に高速化可能か否かが判定される。非投機マルチスレッド実行による高速化が可能であると判断すると、マルチスレッド化コードの実行を開始し高速化を図る。非投機のマルチスレッドでは高速化が望めないと判断された場合、さらなる最適化のためにパスポファイルを行い、投機スレッドを導入したマルチスレッド実行へと移る。

3.2 サンプリング処理の流れ

本サンプリングシステムの処理は、静的にアプリケーションコードを解析した結果の情報を取得する実行前処理と、ユーザアプリケーションの実行に割り込みをかけサンプリングを行い、動的に解析を行う実行時処理とに分けられる。サンプリング処理の流れを図4に示す。

一連の流れは以下ようになる。

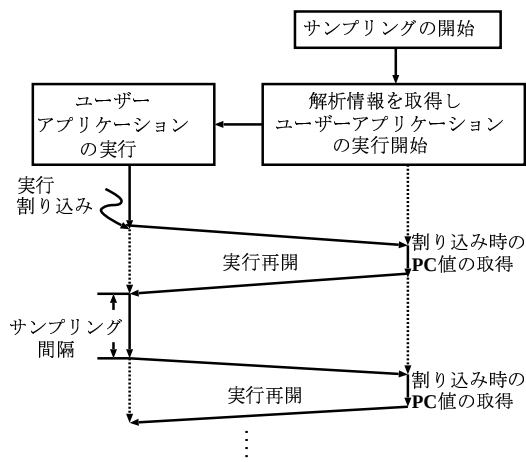


図4 PC サンプリングの流れ

実行前処理

- 入力バイナリを制御フロー解析し、基本ブロックに分割しループ情報を検出する。
- 分割した基本ブロック及びループの開始アドレスや終了アドレス、そしてループのイテレーション間の依存度やオーバーラップ度を記録したデータベースを生成。
- ホットスポットを検出するための閾値およびサンプリング間隔を算出する。

実行時処理

- (1) 実行前処理により生成したデータベースを読み込む。
- (2) 実行前処理により算出された、サンプリング間隔を設定し、ユーザアプリ側へ実行を移す。

- (3) 設定したサンプリング間隔が経過した後、割り込みがかかり、その時のPCの値を取得・登録。
- (4) 再び割り込みの間隔を設定しユーザアプリ側へ実行を移す。
- (5) 以後(2)～(4)を繰り返す。

3.3 ホットループ検出閾値決定手順

PC値をサンプリングにより取得し、ループ情報に記録した際にサンプリングされた回数が閾値以上になったとき、そのループがホットループであると検出する。この閾値が大きすぎる場合、検出されるホットループの数が少なくなり、性能向上に貢献するループを見逃してしまう可能性が増える。逆に小さすぎる場合、総実行回数の少ないループをホットループとして検出してしまい、実行回数の少ないループをマルチスレッド化することになり、結果として実行性能が向上しないという問題が発生する。

この閾値を各アプリケーション毎に適切な値に設定する事で、マルチスレッド化が有効なループを可能な限り検出すると同時に性能向上に寄与しないループをホットループとして検出しないようにする事が可能となる。その際、静的に分かる情報で簡潔に求める事が必要となる。実行時にしか分からない情報を用いてしまうと、実行時に閾値を算出することが必要となる。その算出作業が全てオーバーヘッドに繋がるためである。

静的に分かる情報としてコードサイズがあり、コードサイズが大きい場合、ループの数が多くなると仮定する。ループが多数存在する場合、1ループ当たりの実行頻度が下がるので、閾値を低く抑えなければホットループとして検出されるものが少なくなり、性能向上の可能性があるループを見逃してしまう可能性がある。またサンプリング間隔が大きくなるに従い、サンプリング回数そのものが少なくなる。本稿では

$$\text{閾値} = \left\lceil \frac{C}{\text{コードサイズ}} \times \frac{1}{\text{サンプリング間隔}} \right\rceil \quad (1)$$

C: 定数

により閾値を算出する事とした。ここで定数Cは、SPECint95ベンチマークのサンプリングの実験結果から、サンプリングのオーバーヘッドを総実行サイクル数の1%程度に抑えて、適切なホットループを検出できる閾値を導き出すために、実験的に 22×10^{10} に設定した。

3.4 ループ間依存解析

検出されたホットループでは、レジスタの読み書きに対し図5のような逆フロー依存が高速化を妨げる要因となるため、これを調査することで、イテレーション間のオーバーラップ度を知ることができる。図5内の1つの箱が1スレッドに対応している。

ループ間依存について図6に例を示し、説明する。命令セットはSimpleScalar⁴⁾の命令セットであるPISAである。レジスタの3番が 00400f88 l-hu \$v1[3],0(\$v0[2])の書き込みと 00400f60 addu \$v0[2],\$v0[2],\$v1[3]の読みだしでイテレーション間での依存となっており、レジスタの16番が 00400f90 addiu \$s0[16],\$s0[16],1の書き込みと 00400f70 sb

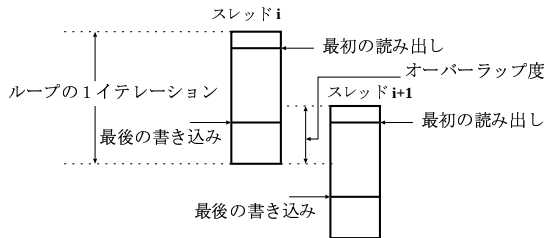


図5 逆フロー依存

\$v0[2],0(\$s0[16])の読み出しで依存となっている。しかしレジスタの3番は値を変更された後、ループの脱出判定に用いられているレジスタ2番に格納されていることから、ループ変数であると推測される。

ループ変数の値はあらかじめContinuationステージで計算されるため、速度向上を妨げる依存とならない、したがって図6の場合ではレジスタ16番が速度向上を妨げる依存を持つことになる。つまり図6の場合では図5に当てはめて、書き込み位置が00400f90読み出し位置が00400f70であることから、ループサイズ10命令中5命令程度のオーバーラップ度であると算出される。

```

00400f58 lui    $v0[2],4097
00400f60 addu   $v0[2],$v0[2],$v1[3] ← レジスタ3番からのイテレーション内での最初の読み出し
00400f68 lbu    $v0[2],11424($v0[2])
00400f70 sb     $v0[2],0($s0[16]) ← レジスタ16番からのイテレーション内での最初の読み出し
00400f78 sll     $v0[2],$v1[3],0x1
00400f80 addu   $v0[2],$v0[2],$s4[20]
00400f88 lhu    $v1[3],0($v0[2]) ← レジスタ3番へのイテレーション内での最後の書き込み
00400f90 addiu   $s0[16],$s0[16],1
00400f98 sli    $v0[2],$v1[3],256
00400fa0 beq    $v0[2],$zero[0],00400f58 ← レジスタ16番へのイテレーション内での最後の書き込み

```

図6 依存のあるループの例

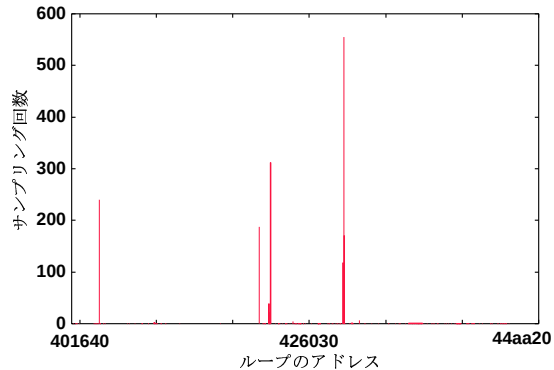
4. 評価

本サンプリングシステムの評価には、シングルスレッドプロセッサのシミュレータ SimpleScalar をベースとし、4命令同時発行可能、スレッドパイプラインモデルのシミュレーションが可能であり、キャッシュメモリのシミュレートも可能なシミュレータ SIMCA⁵⁾ をベースに、一定時間毎にアプリケーションコードの実行に対し割り込みを行い、指定したコードへと実行を移すことが可能になるよう機能を拡張した。対象となるバイナリはSIMCA用gccクロスコンパイラ (version2.7.2.3) を用いて生成したコードを用いた。評価用プログラムとして、SPECint95のm88ksim、データセットとしてtrainを用いた。

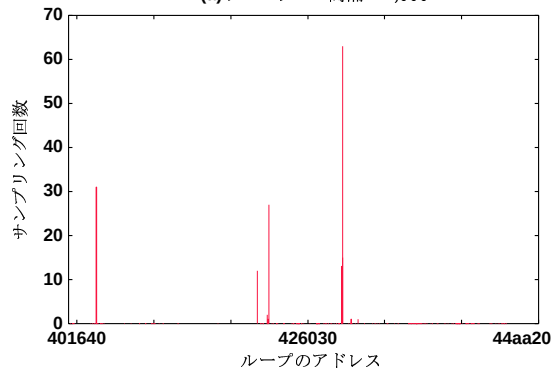
サンプリング結果

図7にm88ksimのサンプリングの実行結果を示す。横軸がループのアドレスに対応し、縦軸がループ内の命令がサンプリングされた回数を示している。実行頻度の高い部分がホットループとして検出され、マルチスレッド化による速度向上の可否が検討される。

このm88ksimでは各ループ間に頻繁に実行されるループと、そうではないループが顕著な差で存在する



(a) サンプリング間隔100,000



(b) サンプリング間隔1,000,000

図7 m88ksimのサンプリング結果

ことが分かる。プロファイリング結果の傾向としてm88ksimでは、実行回数の多いループがプログラム内の様々なアドレスに散在している。m88ksimにおいてはサンプリング間隔をある程度広げていっても、ループの間に実行回数の差が見られることがわかる。

m88ksimのコードサイズ290976バイトと、サンプリング間隔100,000を用いると式(1)より閾値8が得られる。図7の結果にこの閾値を適用すると、8つのホットループが検出できた。

サンプリングオーバーヘッド

次に、図8にはm88ksimにおいてプロファイル情報を取得するためのオーバーヘッドを示す。横軸がサンプリングを行う間隔であり、縦軸がサンプリングによるオーバーヘッドの割合を示す。

m88ksimにおいては、図8から、サンプリング間隔が100,000サイクルでは、サンプリングに要するオーバーヘッドは、総実行サイクル数の1%以下であることがわかる。しかし、さらにサンプリング間隔を広げていくとオーバーヘッドは減少するが、サンプリングを行う回数も減少し、ホットループ選別が困難になった。そのため、サンプリングによるオーバーヘッドと、ホットループの検出精度がトレードオフとなる。

ホットループ解析

関数killtime内のホットループのコードを図9に示す。m88ksim内のホットループについてレジスタの依存解析により、イテレーション間のオーバーラップ度

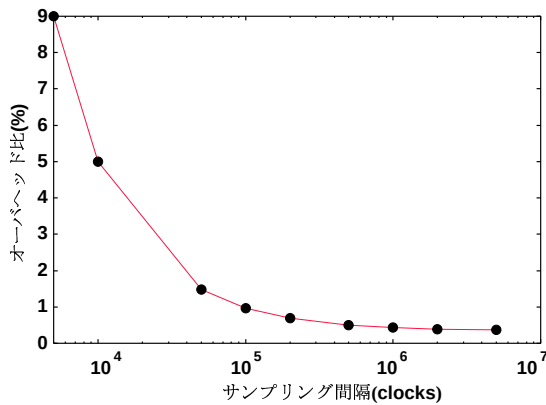


図8 m88ksimのサンプリングオーバーヘッド

を算出する。例えば関数 killtime 内のループでアドレス 42b860 より開始されるループでは、サブルーチンコールはなく解析は容易であり、依存はレジスタ 5 番のみが持つ。しかしこのレジスタは 0042b8a0 addiu \$a1[5], \$a1[5], 1 のように値が 1 ずつインクリメントされていること、その後 0042b8a8 slti \$v0[2], \$a1[5], 32 のようにレジスタの 2 番に値が格納されていること、そのレジスタの 2 番が 0042b8b0 bne \$v0[2], \$zero[0], 0042b860 のようにループの脱出条件になっていることから、レジスタの 5 番はループ変数であると推測されるため、高速化の妨げにはならない。ゆえにこのループは依存がなく、マルチスレッド化による効果が高いと考えられる。しかしこのループは総命令数が 12 命令の小さなループであるため、ループアンローリング等を施して、1 イテレーションの作業量を増やすことで、更なる速度の向上の可能性が考えられる。

```

0042b860 sll $v0[2], $a1[5], 0x2  ← イテレーション内での
0042b868 addu $a3[7], $v0[2], $t0[8]  最初の読み出し
0042b870 lw $a2[6], 0($a3[7])
0042b878 addu $v1[3], $zero[0], $zero[0]
0042b880 situ $v0[2], $a2[6], $a0[4]
0042b888 bne $v0[2], $zero[0], 0042b898
0042b890 subu $v1[3], $a2[6], $a0[4]
0042b898 sw $v1[3], 0($a3[7])
0042b8a0 addiu $a1[5], $a1[5], 1  ← イテレーション内での
0042b8a8 slti $v0[2], $a1[5], 32  最後の書き込み
0042b8b0 bne $v0[2], $zero[0], 0042b860

```

図9 m88ksim内の関数killtime内のホットループ

5. おわりに

本稿では、一定周期毎にPCの値をサンプリングしホットループを検出するシステムを実装し、サンプリングの結果についての評価を行った。評価結果より、100,000 サイクルのサンプリング間隔でプロファイル情報の取得を行った場合、各ループ毎の実行回数に顕著な違いがみられ、プログラムの実行全体に対し、高い実行割合を占めるループの検出が可能であることが示された。さらに、サンプリングによって生じるオーバーヘッドも間隔を調整することで、総実行サイクル数の1%以下に抑えることが可能であることが分かった。

本稿ではホットループ検出のための閾値を静的に分

かるコードサイズとサンプリング間隔から求めることを提案し、実際に適用した。その結果m88ksimのようにループ数が少ないアプリケーションでは、頻繁にサンプリングされるループとその他のループを明確に分けることができた。

検出されたホットループについて、ループのイテレーションサイズ、イテレーション間のオーバーラップ度によって、マルチスレッド化による高速化可能性の検討を行った。その結果、ホットループはイテレーション間のレジスタの依存を調査することでマルチスレッド化により高速化するかどうかの検討が可能であることが分かった。

今後の課題として、

- 本稿のシステムによってホットループの検出を行い、結果をもとに実行時にホットループに対し、マルチスレッド化を行いその性能向上を評価。
- 今回導出した閾値をもとに検出したホットループをマルチスレッド化し、その閾値の妥当性を高速化するかどうかにより評価。
- 検出されたホットループを解析した結果、どの程度のループのオーバーラップ度・ループサイズであればマルチスレッド化した際に高速化が可能であるかの指標が定まっていないため、指標の確立。以上のことが挙げられる。

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)14380135, 同(C)14580362, 若手研究14780186)の援助による。

参考文献

- 1) 大津金光, 小野喬史, 馬場敬信. バイナリレベルにおけるマルチスレッド化コード生成手法, 情報処理学会研究報告(ARC141), Vol.2001, No.10, pp.41-46, 2001.
- 2) 青木政人, 大津金光, 小野喬史, 横田隆史, 馬場敬信. バイナリレベルでの実行時最適化を支援するランタイムシステム, 情報処理学会第64回全国大会, 第1分冊 pp.101-102, 2002年.
- 3) J. Y. Tsai, J. Huang, and et al., "The Superthreaded Processor Architecture," *IEEE Transactions on Computers*, pp.881-902, Vol. 48, No. 9, 1999.
- 4) D. Burger, T. Ausim and S. Bennett, "Evaluating Future Microprocessors", The simple Scalar Tool Set. <http://www.cs.wisc.edu/~m-scalar/simplesscalar.html>
- 5) J. Huang. The Simulator for Multi-threaded Computer Architecture(SIMCA), Release 1.2. <http://www.mount.cs.umm.edu/Research/Ag-assiz/simca.html>.