計算機アーキテクチャ 141－8
（2001．1．26）

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

大津金光† 小野喬史† 馬場敬信†

従来、高性能プロセッサシステムのアーキテクチャとしてマルチスレッド実行モデルを支援するものがさかんに研究されてきた。しかしながら、現状のマルチスレッド化はソースコードレベルで行なわれており、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マルチスレッド化による高速化の恩恵に預かることは困難である。本稿では今後主流になると考えられるマルチスレッド実行を支援する機能を備えたプロセッサシステムを前提として、バイナリレベルでシングルスレッドコードのマルチスレッド化を行ない、バイナリ互換を保持しつつ実行性能の向上を目指したシステムを提案する。

A Methodology for Multithreading with Binary Translation

KANEMITSU OOTSU,† TAKAFUMI ONO † and TAKANOBU BABA†

Recently, various studies have been performed about the high performance processor systems with the multithreaded execution models. However, these studies require the source codes of application programs in order to translate the single threaded codes into the multithreaded ones, and all the source codes cannot be accessible. Therefore, it is difficult to raise the execution performance of these application programs without the source code. This paper proposes the software system architecture that translate the single threaded codes into the optimized multithreaded ones using the binary translation technology.

1. はじめに

従来、高性能プロセッサシステムはシングルスレッド実行を前提にして、その性能を向上させてきたが、その高速化には限界が見えてきた。この問題に対して、マルチスレッド実行モデルが最も有効な解決策であり、実際に次世代の汎用プロセッサシステムの一部にはこれを採用するものが現れてきている。しかしながら、ユーザーがマルチスレッド実行を前提としたプログラミングを行なうことは容易ではなく、シングルスレッドコードからマルチスレッドコードを自動的に生成するシステムの存在が今後重要かつ不可欠になると考えられる。

そのため、これまでに種々の自動マルチスレッド化(あるいは自動並列化)コンパイラの研究開発が行なわれてきた。それらはソースコードレベルでマルチスレッド化を行なうため、実際にアプリケーションの高速化にはソースコードを参照できることが必要となる。しかしながら、現実問題としてアプリケーションプログラムのソースコードはその全てが参照可能なわけではなく、もし実際に高速化したいアプリケーションプログラムのソースコードが参照できない場合には、ソースコードレベルでのマルチスレッド化手法ではそれらのアプリケーションを高速化することができない。これはソースコードの参照を前提とする手法には不可避の問題である。

この問題に対して、ソースコードではなくバイナリコード自体を処理の対象とするバイナリ変換 (**Binary Translation (BT)**) 技術が解決法となりうる。一般に、バイナリコードはソースコードが本来持っていた情報の一部を失なっているため、最適化処理を行ないにくいという性質を持つが、バイナリコードにおいて失なわれた情報の復元を試みる研究[1] もあり、バイナリコードを対象とした場合においても、ソースコードを用いた最適化処理と同等の結果を出せる可能性は十分にあると考えられる。

さらにバイナリ互換性の問題は重要である。当然のことながら、ある命令セットで表現されたバイナリコードは、その命令セットと異なるアーキテクチャ上では全く使えない。そのため、あるアプリケーションを使いたい時に、使用しているプロセッサの用のプログラムバイナリが提供されずソースコードもなく、しかもアプリケーションの提供者が対応しない場合には、そのアプリケーションを使うことは通常は不可能となる。この問題は異機種命令セット間のバイナリ変換を行なうことで対処可能であり、既に実用システムとして活用されている[2]。

また共通の命令セットアーキテクチャを持っているが、拡張命令群が存在する場合に、配布されるアプリケーションが幅広いプラットホームで実行されることを想定し、共通部分の命令セットのみを使用することを強

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University

いられる場面が少なくない。特にその拡張命令群が性能向上に有効な命令 (例えばSIMD 命令など) であった場合には性能向上の機会を自ら閉ざしていることにすらなりかねない。この問題も、バイナリ変換技術により対応可能である。もし拡張命令を使った方が性能が良くなると判断できれば、バイナリコードを拡張命令を使ったコードに書き換えることで、性能向上が見込める。

以上に示す通りに、これらの問題は全てバイナリ変換技術によって解消することが可能となる。しかしながら、バイナリ変換による並列化あるいは最適化処理には、ソースコードを処理の対象としていた場合には存在しなかった問題がある。一般にバイナリコードは命令とデータの区別がなく、静的にコードの全てを解析し変換することは困難である。そのため実行時に変換を支援するシステムが必須となる。この実行時に動作するシステムはプログラムの実行時の挙動に関する情報を取得することができるため、それを使った最適化処理を施すことで実行性能を高めることが可能性である。これは実行プロファイルに基づくコンパイル手法 (**Profile-Guided Compilation**) と類似している点があるが、各実行フェーズでの詳細な情報を収集可能であるため、その潜在的な最適化能力は大きいと考えられる[3]。

以上を背景として、本研究では、今後主流になると考えられるマルチスレッド実行を支援する機能を有するプロセッサシステムを前提にして、ユーザーが記述したシングルスレッドコードからマルチスレッドコードをバイナリ変換により自動的に生成するシステムを構築を行ない、バイナリ互換を維持しつつ、アプリケーションの実行性能の向上を目指す。

2. 基本概念

本研究で提案するシステムは、マルチスレッド実行を支援する機能を有するプロセッサを前提とし、バイナリ変換によるシングルスレッドコードのマルチスレッド化と、実行時最適化を行なうことで、既存アプリケーションとの互換性を維持しつつ、実行性能を高めることを目的としている。本システムの重要な概念として、(1) バイナリレベルでのマルチスレッド化、(2) 既存アプリケーションとのバイナリ互換性、(3) 実行時最適化処理の3つが挙げられる。

2.1 マルチスレッド化

本研究は既存のアプリケーションを制御フローに沿った形でマルチスレッド化を行なうことで、実行性能の向上を目指すが、従来のソースコードレベルではなく、バイナリコードレベルでマルチスレッド化を行なうところに特徴がある。マルチスレッド化処理はプログラムのループ構造に着目し、スレッドパイプライニング実行モデル[10] に基づいて行なう。

マルチスレッド実行を行なうプロセッサの基本構成を図1に示す。各スレッドユニットは従来の汎用プロセッサ相当のものに、スレッドユニット間での通信・同期を行なうための機構を追加したものであり、このスレッドユニットを複数個並列に並べたもので1つのプロセッサを構成する。

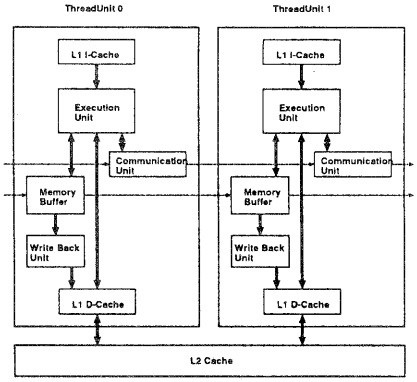

図1 スレッドユニットの構成

マルチスレッド実行の際に問題となるのはスレッド間のデータ依存の全てを事前に把握することが困難なことである。メモリアクセスにはポインタアクセスが存在するため、ポインタを介したメモリアクセスがスレッド間で依存しているかどうかの判断は実行時にしか行なえない。そのため、プロセッサにはメモリアクセスを追跡し、依存が存在する場合に対処できる機構が備わっているものが必要である。ここでは、スレッド間の依存が発生する可能性のあるアドレスを事前に登録しておき、メモリアクセス毎にチェックを行ない、依存がある場合には同期をとる機能を備えたメモリバッファ (図中 **Memory Buffer**) を配置する。

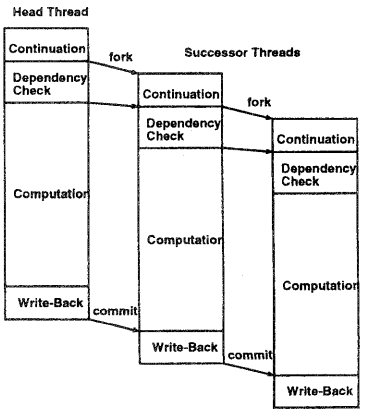

図2 スレッドパイプライニング

図2にスレッドパイプライニングの実行状態を示す。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の断片を実行する。コードを分割する際、理論

的には制御フローに沿っていれば制御フローグラフ上どのような形であっても良いが、各スレッドで実行されるコードサイズの均等化と分割の処理自体の簡略化の観点から、プログラム中のループ構造に注目し、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なう。ここで、1イテレーションが1スレッドになる必要はなく、1スレッドで複数のイテレーションを実行する場合もありうる。

コードの分割後、各スレッドで実行する命令コードを、図に示す通り **Continuation, Dependency Check, Computation, Writeback** の4つのステージに分類する。**Continuation** ステージはループの誘導変数相当の計算を行なうステージで、後続のスレッドが実行を担当するループイテレーションの開始時に必要となる値を算出する。

Continuation ステージ終了後、次のスレッドを起動する。次のスレッドを起動した後、**Dependency Check** ステージに入り、ここでスレッド間のデータ依存をチェックする。前述の **Memory Buffer** の場合、**Memory Buffer** に対して、後続のスレッドと依存が発生する可能性のあるメモリアクセスのアドレスを通知する。

Dependency Check ステージ終了後、**Computation** ステージに入る。ここで、計算本体のコードを実行する。その際に発生するメモリアクセスを、**Memory Buffer** が監視し、スレッド間で依存がある場合にはスレッド間で同期をとり、正しい値を受け渡す。

Computation ステージ後、スレッドは終了処理を行なう **Writeback** ステージに入り、結果の書き戻しを行なう。先頭のスレッド以外は、実行自体の取り消しの可能性があるため、先行するスレッドの実行が完了するまではメモリにその実行結果を反映してはならない。**Writeback** ステージにおいて、それらの同期をとることで、結果を正しい順にメモリへ書き戻す。

2.2 バイナリ互換

計算機システムの性能の向上は重要な問題であるが、互換性の問題も重要である。高性能なシステムであっても、従来使用してきたアプリケーションが使えなければその意味は半減する。互換性を維持する戦略として、ソースコードレベルでの互換性維持と、バイナリコードレベルでの互換性維持の2つのやり方がある。本研究では、後者のバイナリレベルで互換性を維持する戦略を採る。これは、全てのアプリケーションのソースコードが参照可能なわけではなく、たとえソースコードが参照可能であっても、そのコンパイル作業は環境依存であるため、生成したコードが実行可能であるとは限らないからである。

本研究では、対象アプリケーションの命令コードを一度中間表現に変換した後、この中間表現においてマルチスレッド化および各種最適化処理を施し、マルチスレッドコードを生成する。そのため、命令セットアーキテクチャ中立な中間表現を採用し、命令コードから中間表現へのマッピングを行なうことで、様々な命令セットに対応することができる。これにより、既存の命令セットと互換性を維持することが可能となり、過去蓄積されてきた豊富なアプリケーションバイナリコードをそのまま利用することができる。

2.3 実行時最適化

本研究ではアプリケーションプログラムのバイナリコードを処理の対象としている。一般に、バイナリコードは命令とデータの区別がなく、また、例えば間接ジャンプ命令の飛び先といった実行時になって初めて判明する情報が存在するため、バイナリコードを処理の対象とするシステムには実行時に処理する部分が必要となる。この実行時処理を行なう際には、プログラム実行時の挙動に関する情報を利用することができるため、実行の各局面に応じた最適化手法を用いることが可能である。例えば、最も実行されうるトレースが刻々と変化するコードに対して、時間毎にトレースを変えたスケジューリングを行なうことで、性能が改善する可能性がある。

3. システム構成

本稿で提案するシステムのフレームワークについて説明する。図3に本稿で提案するシステムの全体構成を示す。本提案システムは大きくわけて、プログラム実行時に動作する **DTO(Dynamic Translator and Optimizer)** と、プログラム実行前に動作する **STO(Static Translator and Optimizer)**、**DTO, STO** を含め、実際にプログラムを実行するマルチスレッドプロセッサの3つの部分から構成される。ここで、マルチスレッドプロセッサに関しては、現時点では Superthreading モデル[10] を支援する機能を備えたアーキテクチャを仮定しているが、論理的にはマルチスレッド実行が可能なアーキテクチャであればよい。

アプリケーションプログラムは **STO** によりマルチスレッド化されたコードにバイナリ変換され、必要ならばダイナミックリンクライブラリ **(DLL)** とリンクされて、プロセスイメージを生成する。これをマルチスレッドプロセッサ上でマルチスレッド実行を行なうことで、実行性能の向上を図る。

実行前にマルチスレッド化処理ができなかった部分のコードに関しては実行時にこれらの処理を行なうことになるが、そのために処理対象となっているバイナリコードを実行時に参照できる必要がある。よって、プロセスイメージとしては解析対象のバイナリコードと既にマルチスレッド化済みのコードが混在したものとなる。

マルチスレッドプロセッサは実行と同時にプロファイル情報を収集しており、定期的あるいは、特定のイベントの発生などのタイミングで **DTO** に実行を遷移させる。

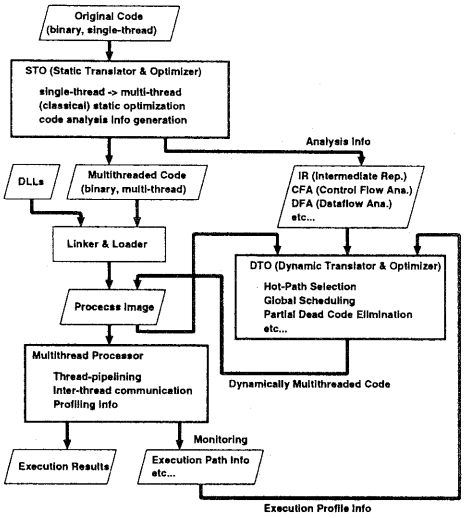

図3　システム構成

DTO に制御が移されると、**DTO** は実行時に収集されたプロファイル情報からプログラムのホットスポットを選別し、該当部分のコードに最適化の余地があれば最適化を行なう。

また、まだ**STO** によりバイナリ変換が行なわれていない部分のコードであった場合は、命令コード解析から始めて、マルチスレッド化とともに、バイナリ変換を行ない、結果をプロセスイメージに反映する。**DTO** での処理が完了した後は、マルチスレッドプロセッサは **DTO** からアプリケーションコードに実行を遷移する。

この流れを繰り返すことで、オリジナルのシングルスレッドバイナリコードをマルチスレッド化および実行時最適化し、マルチスレッド実行で実行性能を高める。

3.1 STO

図において、**STO** は対象なるバイナリコード (図中 **Original Code**) を入力としてバイナリ変換を行い、マルチスレッド化されたバイナリコード (図中 **Multithreaded Code**) を出力する。**STO** は入力バイナリの命令コードの意味解析を行ない、一度中間表現に変換する。この中間表現は命令セット独立な形式であり、これに変換した後は一般の最適化コンパイラと同様の処理が可能である。本提案システムでは、この中間表現レベルでマルチスレッド化を行なう。

入力コードがバイナリコードであり、**STO** はプログラムを実行する前に動作するため、**STO** は全ての処理を完了することはできない。これは、一般にバイナリコードは命令・データの区別がないこと、実行時になって初めて分かる情報の存在などの理由による。例えば、間接ジャンプ命令を含むコードは、一般的に実行時にならなければ命令の飛び先が分からないため、そこから先の命令コードの解析ができない。また、自己書き換えコードは実行時になって初めて命令コードが判明するため、実行前には解析不能である。そのため、バイナリ変換処理の一部は実行時に行なう必要があるが、その処理を後述の **DTO** が担当する。

また、本システムでは実行時最適化を行なうが、その際に必要となる命令コードの解析情報、制御フロー情報、データフロー情報の解析結果を **STO** から **DTO** に受け渡す。これにより、実行時最適化の処理にかかるオーバーヘッドを削減する。さらに、実行時最適化処理に必要となる実行のプロファイル情報を収集するためのコードの追加を行なうことも **STO** の役割である。

3.2 DTO

図において、**DTO** は実行時にバイナリコードの最適化を行なう。**DTO** は対象となるプログラムの実行中の適当なタイミングにより実行を開始する。**DTO** の主な役割は実行時最適化処理である。**STO** により付加されたプロファイル情報収集コードによる結果を基にしてホットパスの選択後、大局スケジューリング、部分冗長コード削除などの最適化処理を行なう。最適化の結果のコードをプロセスイメージに反映した後、対象アプリケーション側に実行を戻す。この際、実行プロファイル収集用のコードが再度付加され、次の機会の最適化処理のための実行プロファイル情報の収集が行なわれる。

DTO は自己書き換えコードが存在した場合に備えて、**STO** が果している役割の一部を兼ね備えている。具体的には、命令コードの解析、制御フロー解析、データフロー解析、コードのマルチスレッド化などの最適化処理である。ただし、**DTO** は実行時にそれらの処理を全て行なうことになるため、処理にかかる時間を可能な限り短くする必要がある。そのため、**STO** よりは処理内容が簡略化されたものとなる。

4. バイナリ変換によるマルチスレッド化

プログラム中のループ処理部分に着目し、スレッドパイプライニングモデルを基にしたマルチスレッド化を行なう。論理的には、以下の4段階の作業が行なわれる。

- 対象アプリケーションのバイナリコードの解析と中間表現への変換
- 基本ブロックへの分割および制御フロー解析とループ構造の検出
- データフロー解析により、誘導変数とイテレーション間依存の検出
- スレッドパイプライニングに基づくマルチスレッド化コードの生成

先ず、対象アプリケーションのバイナリコードの実行開始アドレスから命令コードの解析を行ない、中間表現へと変換していく。中間表現への変換は一般のコンパイラのコード生成の逆の操作に相当する。一般に、複数の命令コードが高水準言語上での一つの操作 (あるいは中間言語の一操作) に対応する。理想的に、この多対一のマッピングを復元できれば、解析対象のコードを生成し

たコンパイラが使用していたものと意味的に等価な中間表現を得ることができ、最適化処理において同等の処理を施すことが可能となる。

ここで、対象がバイナリコードであるため、実行前に全てのコードを解析し尽くすことは困難である。実行時に変換すべき処理を少なくするため、バイナリコード中の命令コード部分を推測し、可能な限り変換処理を実行前に済ませておく。例えば、間接ジャンプ命令によって制御フローが追えなくなった場合も、飛び先が不明なだけで命令コード自体は既に含まれている場合が多く、この部分の命令コードば実行前に変換処理が可能である。

中間表現への変換後は一般の自動マルチスレッド(並列化)化コンパイラと行なうべき処理は本質的に同じである。本研究ではコードのループ部分をマルチスレッド実行を行なうため、制御フロー解析によりループを検出後、誘導変数とイテレーション間依存の検出を行なう。2.1節に述べたスレッドパイプライニングモデルに基づき、誘導変数の更新コードを **Continuation** ステージに生成する。その直後にスレッド生成コードを挿入する。イテレーション間の依存関係にあるデータは可能性のあるものを含めて **Dependency Check** ステージ内で、それらのアドレスを登録するコードを生成し、**Computation** ステージにオリジナルのイテレーションコードで処理コードを生成する。

```
;; continuation stage
        slti $v0[2],$v1[3],5000
        beq $v0[2],$zero[0],$ST_LL0

        addu $t0[8],$a0[4],$zero[0]
        addu $t1[9],$a1[5],$zero[0]

        addi $v1[3],$v1[3],1
        addi $a0[4],$a0[4],4
        addi $a1[5],$a1[5],4

        lfrk            ;; <= thread fork
        wtsagd

;; target-store address generation stage
        addiu $t2[10],$sp[28],$zero[0]
        altsw $t2[10]
        tsagd

;; computation stage
        l.s $f0,0($t0[8])
        l.s $f2,0($t1[9])
        l.s $f4,0($t2[10])
        mul.s $f0,$f0,$f2
        add.s $f4,$f4,$f0
        sttsw $t2[10],$f4

$ST_LL0:
        estr            ;; <== thread end
```

図4 変換後のループボディコード

コード生成の例として、図4に内積計算プログラムのマルチスレッド化後のコード(ループ部分のみ)を示す。生成コードに使用した命令セットはSuperthreadingモデル[10]のシミュレータ **SIMCA** のものを用いている。図からも分かる通り、計算部分に対してスレッドの制御にかかるコストが大きいため、このままでは性能が出ないことになる。そのため、実際にはloop unrollingを行ない、1スレッド当りの処理内容サイズを大きくし、スレッド制御のオーバヘッドの影響を小さくする必要がある。なお、loop unrolling処理を施した場合でのシミュレーション評価により、ほぼスレッドユニットの台数に比例した高速化を達成できることは確認済みである。

5. 関連研究

バイナリ変換技術は、異機種命令コードの実行や命令コードの最適化などの目的のために使用されている。

Compaq社のFX!32[2]はAlphaプロセッサ版WindowsNT上でx86版のwin32アプリケーションを動作させるためのNTのサブシステムで、x86命令のエミュレーションとその高速化のためのAlpha命令セットへのバイナリ変換を行なう。この場合のバイナリ変換はエミュレーション実行を行なった際に取得したプロファイル情報を基にし、システムがプログラムを実行していないアイドル時間の間に行なわれる。一度変換されたコードは次回以降はプロセッサのネイティブコードで実行することで性能が向上する。

HP社のDynamo[6]もFX!32と同様のシステムで、x86用のバイナリを異機種命令セットであるPA-RISC用のバイナリに変換する。FX!32とは異なり、バイナリ変換は実行時に行なわれる。最初はエミュレーション実行を行ないながらプロファイル情報を取得し、プログラムのホットスポットを検出するとその部分コードをバイナリ変換しキャッシュする。変換後はキャッシュ内のプロセッサネイティブコードを実行するため高速に実行できる。

IBM社のDAISY[5]は、独自VLIWプロセッサ上でPowerPCやx86などの異機種命令コードを実行するために、実行時にバイナリ変換を行なっている。この際、複数パスの命令スケジューリングによる命令レベル並列性(ILP)を引き出す最適化を行なうことで性能向上を図っている。DAISYでは、実行対象とする命令コードのエミュレーション実行を行なわず、必ずVLIWコードへのバイナリ変換した後に実行を行なう。

DAISYと似たシステムとして、Transmeta社のCrusoeプロセッサ[7]が挙げられる。Crusoeは **CMS (Code Morphing Software)** と呼ばれるソフトウェアにより、x86命令セットで記述されたバイナリコードを独自命令セットを持つ内部VLIWプロセッサの命令コードに実行時に変換を行なう。DAISYとは異なり、バイナリ変換はプログラムのホットスポット部分に限定され、インクリメンタルに実行時最適化が行なわれる。

以上のシステムは主として(特にx86命令セットとの)バイナリ互換性を重点にしたものであるが、最適化の目的でバイナリ変換を行なうシステムもある。

ハーバード大のMorph[9]は、Alphaプロセッサ上で、OSとの協調作業により、プロファイル取得とそれを基にしたオフライン最適化とバイナリ変換を自動的に

行なう。同じく、ハーバード大のDeco[8]は全て実行時最適化によりバイナリ変換を行なうシステムである。Decoでは、一度最適化によりバイナリ変換されたコードも最適化の対象となっており、プログラムの挙動が変化し、前回の最適化が有効に機能しなくなった場合には再度実行時最適化によりバイナリ変換が行なわれる。

IBM社のBOA[4]はPowerPCアーキテクチャのout-of-orderのスーパースカラ実行に必要な複雑なハードウェアを廃し、ハードウェアの簡素化によるクロック速度の向上で性能を上げるEPICスタイルのアプローチを採ったシステムである。その際に各命令のスケジューリングの効率を上げる必要があるが、BOAではソフトウェアによる実行時バイナリ変換を行なうことでそれを達成している。DAISYがあるコードについてバイナリ変換を一度しか行なわないのに対し、BOAはプログラムの実行パスに関する挙動を追跡し、最適化の機会があれば何度でも実行時最適化を行なう。

JavaのHotSpotVM[11]は、インタプリタ実行時に実行プロファイルを集計し、プログラムのホットスポットを検出した場合は該当コードをVMコードからプロセッサネイティブコードにバイナリ変換を行ない、VMの実行性能の向上を図っている。

以上に挙げたシステムはいずれも基本的に変換前あるいは変換後の命令セットが固定であったが、これを任意の組み合わせで行なう研究として、**UQBT**[1]が挙げられる。UQBTはリターゲッタブルなバイナリ変換システムのフレームワークで、現在のところSPARC, x86, JavaVMの3種の命令セットのみ実装されているが、理論的には任意の異なる命令セット間のバイナリ変換が可能となっている。

これらのシステムはどれもシングルスレッドプロセッサの高速化を目的としており、マルチスレッド化による高速化は考慮されていない。そのため、バイナリレベルでのプログラムのマルチスレッド化による実行性能の向上を目指す本研究の方向性とは大きく異なる。

6. おわりに

今後主流になると予想されるマルチスレッドプロセッサの普及を前提として、バイナリレベルでのシングルスレッドからマルチスレッドコードへの変換を行ない、さらに実行時最適化との組み合わせることで、従来の逐次処理型のアプリケーションプログラムの高速化を目指すシステムの提案を行なった。プログラムのマルチスレッド化は、プログラム中のループ構造に着目し、スレッドパイプライニングモデルを基にした変換処理を行なう。また、本システムではソースコードの参照が必要ではなく、従来アプリケーションとの互換性を維持しつつ、マルチスレッド化により実行性能の向上を達成する。さらに、バイナリ変換処理には必須となる実行時モジュールに最適化処理機構を組み込むことで、実行時最適化処理を行なうものである。

謝辞 本研究は、一部文部省科学研究費 基盤研究(B)課題番号10558039, 基盤研究(C)課題番号12680328の援助による。

参考文献

1) C. Cifuentes and M. Van Emmerik, "UQBT: Adaptable Binary Translation at Low Cost," *Computer*, Vol 33, No 3, pp. 60–66, 2000.
2) R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, Vol.9, No 1, pp. 3–12, 1997.
3) M. D. Smith, " Overcoming the Challenges to Feedback-Directed Optimization," *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, 2000.
4) S. Sathaye, P. Ledak, and et al, "BOA: Targeting Multi-Gigahertz with Binary Translation," *Workshop on Binary Translation (Binary99)*, 1999.
5) K. Ebcioglu, E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proceedings of 24th Annual International Symposium on Computer Architecture*, pp. 26–37, 1997.
6) V. Bala, E. Duesterwald, S. BanerjiS, "Dynamo: A Transparent Dynamic Optimization System," *Proceedings of Programming Language Design and Implementation*, 2000.
7) A. Kaliber, "The Technology Behind Crusoe Processors," 2000, URL: http://www. transmeta.com/crusoe/download/pdf/crusoetechwp.pdf.
8) E. Feigin, "A Case for Automatic Run-Time Code Optimization," Senior thesis, Harvard College, Division of Engineering and Applied Sciences, 1999.
9) X. Zhang, Z. Wang, and et al, "System Support for Automatic Profiling and Optimization," *Proceedings of 16th Symposium on Operating Systems Principles*, 2000.
10) J. Y. Tsai, J. Huang, and et al, "The Superthreaded Processor Architecture," *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, vol. 48, no. 9, 1999.
11) Sun Microsystems, "Java HotSpotTM Technology," URL: http://java.sun.com/products/hotspot/