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とその初期評価

小野喬史[†]　大津金光[†]
横田隆史[†]　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は困難である。そこで、本研究は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、バイナリ変換による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自動生成を行うことでアプリケーションの実行性能を向上させることを目標とする。本稿では、試作したバイナリトランスレータについて述べると共に、シミュレータを用いた評価結果により、スレッドの処理サイズが十分大きい場合、ほぼ線形に近い性能向上が可能であることを示す。

Preliminary Evaluation of Binary-Level Multithreading

Takafumi ONO,[†] Kanemitsu OOTSU,[†] Takashi YOKOTA[†]
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 always accessible. Therefore, it is difficult to raise the execution performance of these application programs when the source codes are not available. In order to solve this problem,we have designed and developed a prototype system that performs both binary translation and multithreading at the same time. We evaluate the performance of the translated codes by using a thread pipelining simulator.

1. はじめに

従来の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として、その性能を向上させてきた。しかし、その高速化には限界が見えて来た。そのため、この問題の解決策として、プロセッサシステムのマルチスレッド実行によりスレッドレベル並列(TLP: Thread-Level Parallelism)を利用する事が最も有効である。しかし、マルチスレッド実行を前提としたプログラミングを行うことは容易ではないため、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生成するシステムが必要となる。

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る。しかし、ソースレベルにおいてマルチスレッド化することによるアプリケーションの高速化には、ソースコードの参照が必要となる。しかし、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いため、高速化を行いたいアプリケーションのソースコードが参照できない場合は、高速化できないという問題が発生する。本研究はこの問題の解決策として、ソースコードではなく実行バイナリコード自体を対象とする、**バイナリ変換** (**Binary Translation**)を用いる。一般的に、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っているため、最適化処理を行いにくいという問題があるが、バイナリコードにおいて失われた情報の復元を試みる研究[2]も行われており、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と同等の性能を出せる可能性は十分にあると考えられる。

Compaq社のFX!32[3]、IBM社のBOA[4]、DAISY[5]、HP社のDynamo[6]は、バイナリ変換と、実行時最適化により性能の向上を行っている。このように、バイナリ変換を用いてシングルスレッドプロセッサを高速化する目的の研究は盛んに行われているが、マルチスレッド化により実行性能の向上を目指す研究はまだ行われていない。

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

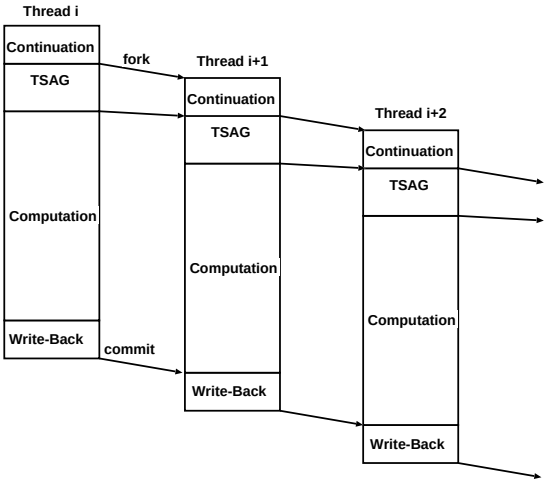

図1 スレッドパイプライニング

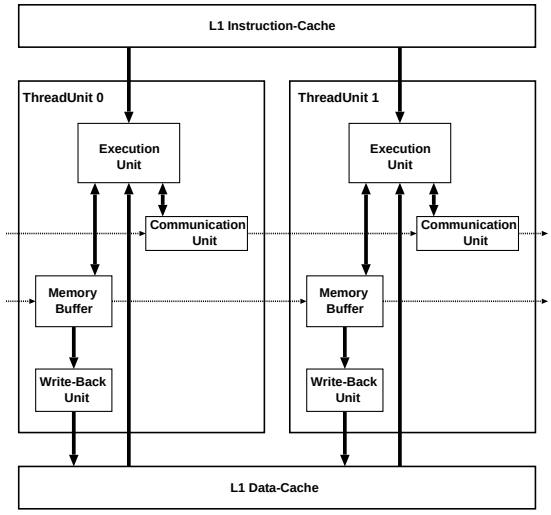

図2 スレッドユニットの構成

以上を背景として、本稿ではバイナリ変換に基づいたマルチスレッドコードの自動生成[1]の有効性を確認するために行った初期評価について述べる。以降、2節ではマルチスレッド実行モデルについて、3節ではマルチスレッドコード生成のシステム概要、4節では試作したマルチスレッドツールで用いたマルチスレッド化手法、5節では初期評価結果について述べる。

2. マルチスレッドモデル

本研究で前提とするシステムは、マルチスレッド実行を支援する機能を持つプロセッサによる実行を前提とし、シングルスレッドコードからバイナリ変換によってマルチスレッドコードを生成しそのバイナリを実行する。マルチスレッドモデルとして各スレッドが4つの単純なステージで構成されマルチスレッド化を行いやすいスレッドパイプライニングモデルを用いる。以下、本研究で用いるスレッドパイプライニングモデルとスレッドパイプライニングを実現するアーキテクチャについて説明する。

2.1 スレッドパイプライニング

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えて、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。

本研究ではスレッドパイプライニング実行モデル[7]に基づいてマルチスレッド化を行う。図1にスレッドパイプライニングの実行状態を示す。なお、スレッド間データ依存については、後述する**Memory Buffer**を用いて解決する。

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1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と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。

2.2 対象アーキテクチャ

前節で述べたスレッドパイプライニング実行モデルを実現するために、本研究では、**Superthreaded Architecture**[7]を用いて実現する。

図2に、Superthreaded Architecture で用いるスレッドユニットの構成を示す。各スレッドユニットは従来の汎用プロセッサに、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る。1プロセッサは、スレッドユニットを複数個並べた構成となる。各スレッドユニットでL1キャッシュは共有とする。

各スレッドユニットは、計算を行う**Execution U-**

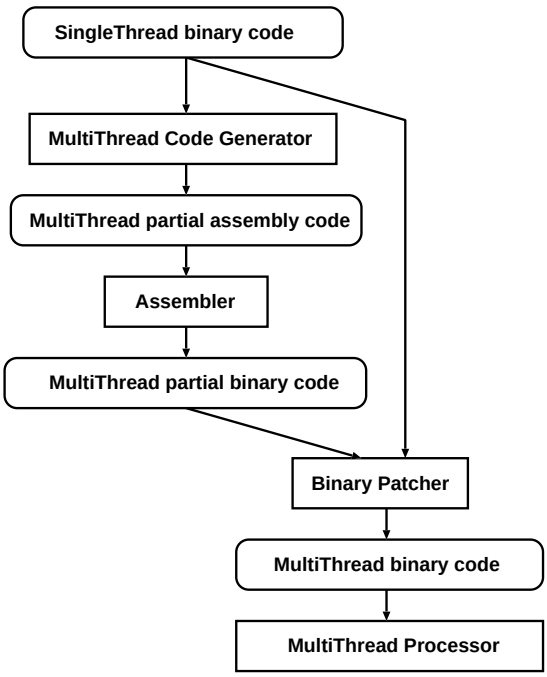

図3 システム概要

nit、スレッド間通信のための **Communication Unit**、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**、メモリへの Write-Back 処理を行う **Write-Back Unit** により構成される。

3. システム概要

図3に、本研究で試作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生成するシステムの概要を示す。

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ために、まず対象アプリケーションであるシングルスレッドバイナリコードを逆アセンブルし、ループを含み高速化が可能と予想されるサブルーチンを決定する。

次に、本研究で試作したバイナリレベルマルチスレッド化ツール **MultiThread Code Generator** にマルチスレッド化するサブルーチンを指定し、シングルスレッドバイナリコードを入力することで、サブルーチンのマルチスレッド化したアセンブリコードを生成する。このアセンブリコードをアセンブルすることにより、部分マルチスレッドバイナリコードを生成する。

この部分バイナリコードとシングルスレッドバイナリコードを **Binary Patcher** を用いて結合し、最終的なマルチスレッドバイナリコードを生成する。この Binary Patcher は、実行時に対象サブルーチンが呼び出された場合、マルチスレッド化されたサブルーチン側が呼び出されるように、オリジナル側のサブルーチンの先頭にマルチスレッド化された側のコードへのジャンプ命令を挿入する。

以上の手順で作成したマルチスレッドバイナリコードをマルチスレッド実行機能を持つ **MultiThread Processor** により実行することにより、対象アプリケーションを高速化する。

ここで、対象となるアプリケーション内のコードからマルチスレッド化を行うべきサブルーチンを人手で指定しなければならないが、将来的にはマルチスレッド化することで、高速化する可能性のあるサブルーチンを自動判定する。

4. マルチスレッド化手法

4.1 マルチスレッド生成アルゴリズム

本研究で試作した MultiThread Code Generator における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変換アルゴリズムは以下の通りである。

(1) 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
(2) 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
(3) データフロー解析により、ループ変数とイテレーション間依存の検出
(4) 中間表現レベルでループ構造をスレッドパイプライニングに基づきマルチスレッド化
(5) 中間表現から対象となる命令セットへの変換

まず、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い、中間表現へと変換していく。

その後、中間表現に変換した対象アプリケーションを基本ブロック単位に分割し、制御フローを解析しループ構造の検出を行う。そして、データフロー解析を行うことによって、ループにおけるループ変数とイテレーション間依存を検出する。このループ変数が 2.1 節で述べた Continuation ステージで処理され、イテレーション間依存変数が TSAG ステージにおいて処理される。シングルスレッドコードの静的な解析が終了した後は、これまでの情報を用いてスレッドパイプライニングモデルに基づいてマルチスレッド化を行う。

マルチスレッド化による高速化のためには、(i) 速やかに次スレッドを生成し処理を始める事と、(ii) イテレーション間依存による処理の中断時間を小さくする事が重要である。

(i) については、Continuation ステージで、次スレッド開始時に必要となるループ変数を計算してから、次スレッドを生成することで行う。つまり、ループ変数の計算はループの先頭以外のスレッドは前スレッドで計算を行い、先頭スレッドはループ突入前に計算する。この結果、生成されたスレッドは速やかに処理を行う

ことができる。

(ii) については、Comptionステージにおいて、イテレーション間で依存するロード命令をできるだけ後方へ、ストア命令をできるだけ前方へ移動するという最適化を行うことで対処する。

Continuationステージ終了後、TSAGステージにおいて、イテレーション間依存からスレッド間依存となるメモリアクセスのアドレスをMemory Bufferに登録する。なお、TSAGステージはメモリアクセスのアドレスを計算する関係上、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ステージを開始できず、TSAGステージ終了後にはステージの終了を次スレッドに通知する必要がある。

Computationステージでは、そのスレッドにおけるメインとなる計算本体のコードを実行する。このとき、スレッド間で依存のあるメモリアクセスはMemory Bufferが監視を行う。Memory Bufferに登録されたアドレスへのメモリアクセスが発生した場合、そのデータが前スレッドで書き込み済みならばそのまま処理を続け、まだならばスレッド実行は前スレッドでの書き込みが終了するまで待機状態となり書き込み終了後に実行を再開する。逆に、Memory Bufferに登録したメモリアドレスでの書き込みが終了したら速やかに次スレッドへ通知する。

スレッドパイプライニング実行において性能を出すためには、Computationステージにおいてスレッド制御にかかるオーバーヘッドを隠すために十分な処理サイズを必要とするが、本研究では後述するループアンローリングを用いて対処する。

Writebackステージでは、スレッドを終了させ、スレッド内処理の結果をメモリに書き戻す。プログラムオーダでメモリに値を書き込むことを保証するため、前スレッドのWritebackステージが完了するまで、本ステージを開始することができず、ステージ終了後にはステージの終了を次スレッドに通知する必要がある。

以上の手法を用いて中間表現上でマルチスレッド化を行い、実行対象となる命令セットに変換する。

4.2 適 用 例

前節で述べた方法で用いてにシングルスレッドコードをマルチスレッド化した例として、図4 に変換前後のコードをアセンブリ言語を用いて示す。図の左側が変換前のシングルスレッドコード、右側が変換後のマルチスレッドコードである。変換コードでは、実行モデルとなる Superthreaded Architecture 固有のスレッド制御命令(表1)を用いる。

表1 スレッド制御命令

命令	処理
bstr	マルチスレッド実行領域の開始地点のアドレスを示す。
estr	マルチスレッド実行領域の終了地点のアドレスを示す。
lfrk	次スレッドを生成する。
tsagd	次スレッドに TSAG ステージ終了のフラグを送る。
wtsagd	前スレッドからのTSAGステージ終了フラグを待つ。
altsd	指定するアドレスを**Memory Buffer**に登録する。
sttsw	指定する値をメモリに書き込み、次スレッドに通知する。

```
$L0:                                /* Continuation Stage */
    lw      $2,16($fp)                  bstr
    lw      $3,20($fp)                  lw      $2,16($fp)
    slt     $2,$2,$3                    sw      $2,-8($sp)
    bne     $2,$0,$L2                   lw      $2,-8($sp)
    j       $L1                         addi    $2,$2,1
$L2:                                    addi    $3,$fp,16
    l.s     $f0,16($fp)                 sttsw   $3,$2
    cvt.d.w $f0,$f0
    mov.d   $f12,$f0                    lw      $2,-8($sp)
    jal     sin                         lw      $3,20($fp)
    l.d     $f2,48($fp)                 slt     $2,$2,$3
    add.d   $f0,$f2,$f0                 bne     $2,$0,$L3
    s.d     $f0,48($fp)                 j       $ST_END
    lw      $3,16($fp)              $L3:
    addu    $2,$3,1                     lfrk
    move    $3,$2
    sw      $3,16($fp)              /* TSAG Stage */
    j       $L0                         wtsagd
$L1:                                    addiu   $2,$fp,48
                                        altsd   $2
                                        tsagd

                                    /* Compution Stage */
                                        l.s     $f0,-8($sp)
                                        cvt.d.w $f0,$f0
                                        mov.d   $f12,$f0
                                        jal     sin
                                        l.d     $f2,48($fp)
                                        add.d   $f0,$f2,$f0
                                        s.d     $f0,48($fp)
                                        addiu   $3,$fp,48
                                        addiu   $4,$fp,48
                                        lw      $2,0($3)
                                        sttsw   $4,$2

                                    /* Write-Back Stage */
                                    $ST_END:
                                        estr
```

図4 マルチスレッド変換の前後

スレッドパイプライニングによるスレッド領域の開始と終了地点はそれぞれ **bstr** と **estr** 命令で指定する。

Continuationステージでは、現スレッドのループ変数の値(アドレス:($fp+16))から、次スレッド開始時に必要となるループ変数値を計算し、その結果を **sttsw** 命令を用いて書き込み、次スレッドを生成する **lfrk** 命令を用いて次スレッドを生成する。その際、ループ変数値はsttsw命令により次スレッドの値を書き込むので、現スレッド内の計算においてループ変数値を用いる場合、以後はアドレス: ($sp-8) を用いて処理を行う必要がある。

TSAGステージでは、スレッド間依存となるメモリアクセス(アドレス: ($fp+48))のアドレスを **altsd** 命令によってMemory Bufferに登録する。前節で述べたステージの切替え処理のため、TSAGステージの最初と最後に、**wtsagd** 命令と **tsagd** 命令をそれぞれ挿入する。

Computationステージでは、計算本体を実行する。処理中、Memory Bufferに登録されたアドレス: ($fp+48)にメモリアクセスが発生すると前述の通り、前スレッドでの書き込みの終了を待つ。

Write-backステージでは、estr命令によりメモリへ

の書き戻しが行われ、その後は次スレッドへWrite-backステージの終了を通知し、現スレッドの実行を完了する。

4.3 ループアンローリング

マルチスレッドコードにはシングルスレッドコードには存在しないスレッド制御命令が存在する。そのため、スレッド制御にかかるコストに注意する必要がある。もし本来の計算部分に対してスレッド制御にかかるコストが大きい場合は、スレッド制御におけるオーバーヘッドのために性能が出ない。そこでループアンローリングを行い、1スレッド当たりの処理サイズを大きくし、スレッド制御にかかるオーバーヘッドの影響を小さくする必要がある。

前節の例で用いた図4では命令数が約2倍に増えており、スレッド制御によるコストが大きく性能が出ない。そこで、ループアンローリングを行うことでComputionステージの処理量を大きくすることによって、スレッド制御によるコストの影響を小さくする必要がある。

また、ループアンローリングを用いる事によってメモリアクセス数を減少できる可能性がある。メモリアクセスの減少は実行性能の向上の上で有効である。

なお、ループアンローリングするのは、対象バイナリコードの中間表現への変換後、マルチスレッド化処理前に行う。

5. 評　　価

バイナリレベルにおいてマルチスレッドコードを生成し、その実行性能について評価する。

5.1 評価環境

実行性能を評価するための評価環境について説明する。評価には、スレッドパイプライニングモデル[7]のシミュレータ**SIMCA**[8]を用いる。

まず、前述した図3に準じ、対象となるシングルスレッドバイナリコードからMultithread CodeGeneratorにより評価プログラムの計算本体部分のサブルーチンのマルチスレッドアセンブリコードを生成する。その後、SIMCA用アセンブラにより部分マルチスレッドバイナリコードを生成する。最後に、Binary Patcherにより両者を結合して最終的なマルチスレッドバイナリコードを生成する。以上の手順により生成したマルチスレッドコードをSIMCA上で実行する。

なお、対象となるシングルスレッドコードのバイナリは、SIMCA用gccクロスコンパイラ(version 2.7.2.3 最適化オプション -O2)を用いて生成したコードを用いる。

評価用のプログラムとして、台形公式を用いたsin関数の積分値計算と内積値計算を用いる。

評価は、評価用プログラムをシングルスレッド実行とマルチスレッド実行により、それぞれの計算本体のクロック数を計測し、速度向上比(= シングルスレッドコードのクロック数 / マルチスレッドコードのクロック数)を求めることで行う。このとき、マルチスレッドコードは、ループアンローリングを行わない場合と行う場合(回数は4回、8回、16回)の4種類のコードを用い、SIMCAのスレッドユニット数を4台、8台、16台と変化させた場合での評価を行う。

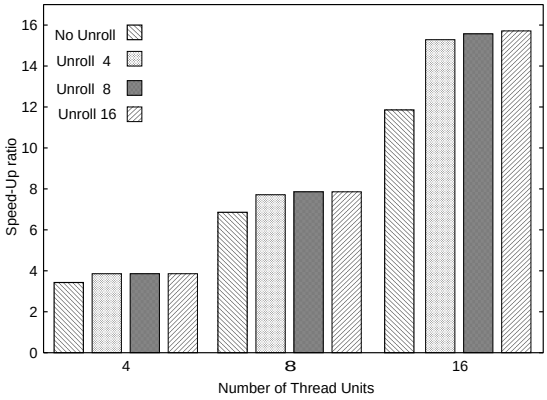

図5　積分値計算プログラムの速度向上比

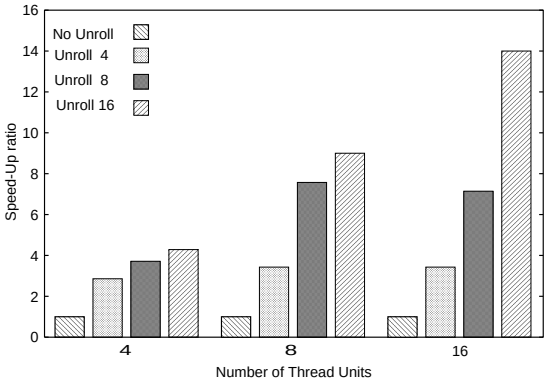

図6　内積値計算プログラムの速度向上比

5.2 評価結果

図5にsin関数の積分値計算の速度向上比を示す。今回の条件内でのアンローリングの回数では、ほぼ回数に関係なく、スレッドユニット数が4、8、16と増えるにしたがって、速度向上比も4倍、8倍、16倍となる。アンローリングを行わずとも、3.5倍、6.7倍、12.3倍の速度向上が得られている。

このことから、積分値計算におけるマルチスレッド化においては、ループアンローリングを行わずとも高速化は可能であるが、スレッドユニット台数が増えた場合にはループアンローリングが必要であり、アンローリングによりほぼリニアな性能向上比が得られることがわかった。よってスレッドユニット台数分の速度向上が得られることがわかる。

図6に内積値計算の速度向上比を示す。速度向上比はスレッドユニット数4、8、16台においてアンローリングを行わなかった場合は、ほぼ0.9倍であった。アンローリング回数が4の場合は、それぞれおよそ2.8倍、3.5倍、3.5倍。同様にアンローリング回数が8の場合は、それぞれおよそ3.8倍、7.6倍、7.1倍。アンローリング回数が16の場合は、それぞれおよそ4.6倍、9.1倍、13.9倍となり、アンローリングを16回行わなかった場合、スレッドユニットが4台と8台の向上比はほぼ同じであった。これは、1イテレーションあたりの処理サイズが小さいため、スレッド制御命令のオーバーヘッドにより性能が出ず、高い向上比を得るためには、&たとえアンローリングを行っても、スレッドユニット台数が8台では8回、16台では16回のアンローリングと、スレッドユニット台数の回数だけアンローリングする必要がある。

これらの結果より内積計算は、スレッドユニット台数に比例した回数分アンローリングすることによって、スレッドユニット台数に近い速度向上比を出すことできることがわかる。

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を前提として、バイナリ変換を用いてシングルスレッドからマルチスレッドコードへ自動変換を行うツールMultithread Code Generatorを試作し、マルチスレッド実行による速度向上の初期評価として積分値計算と内積計算を用いて評価を行った。

アプリケーション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変換処理を行った。その際、ループアンローリングによって、1スレッドあたりの処理サイズを大きくすることで高速化することを示した。

今回の初期評価から、スレッドパイプライニングモデルによるマルチスレッド実行により、各スレッドあたりの処理サイズが十分に大きければ、アプリケーションの実行性能を向上できることを示した。

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(C)課題番号12680328)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作成手法, 情報処理学会研究報告(ARC-141),p41-46.2001.
2) C. Cifuentes and M. Van Emmerik, "UQBT: Adaptable Binary Translation at Low Cost," *Computer*, Vol 33, No 3, pp. 60–66, 2000.
3) 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.
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. Banerji, "Dynamo: A Transparent Dynamic Optimization System," *Proceedings of Programming Language Design and Implementation*, 2000.
7) J. Y. Tsai, J. Huang, and et al, "The Superthreaded Processor Architecture," *IEEE Transactions on Computers*, Vol. 48, No. 9, 1999.
8) J. Huang. The SImulator for Multi-threaded Computer Architecture(SIMCA),Release 1.2. http://www-mount.cs.umm.edu/Research/Agassiz/simca.html