バイナリレベルマルチスレッド化コード生成手法とその評価

大津金光[†] 小野喬史[††]
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あずかることは困難である．この問題に対して，我々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，バイナリレベルでシングルスレッドコードから最適化されたマルチスレッド化コードの生成を行い，実行性能の向上を図るシステムを提案してきた．本稿では，提案システムにおいて基盤技術となるバイナリ変換によるマルチスレッド化のコード生成手法の説明を行う．さらに，変換手法の有効性を検証するための実験的なシステムの試作を行い，試作システムにより生成したマルチスレッド化コードをシミュレータ上で実行して評価を行う．評価により，スレッド制御のオーバヘッドの影響を抑えられる程度のサイズのスレッドコードを生成すれば，十分に性能向上が可能であることを示す．

A Methodology for Binary-level Multithreading and Its Evaluation

Kanemitsu Ootsu,[†] Takafumi Ono,[††] Takashi Yokota[†]
and Takanobu Baba[†]

Recently, various studies have been performed for realizing the high performance processor systems with the multithreaded execution models. However, these studies require the source codes of application programs in order to translate single threaded codes into the multithreaded ones. However, the source codes are not always available. In this case, it is difficult to raise the execution performance of these programs. To solve this problem, we have proposed the binary translation system which inputs single threaded binary codes, and produces the optimized, multithreaded ones using a newly developed binary translation and optimization technology. In this paper, we explain the methodology for binary-level multithreading that is the basic technology of the proposed system. In order to evaluate the effectiveness of the methodology, we have built an experimental binary translation system, and evaluated it by using a cycle base simulator. The evaluation results show that the performance can be greatly improved when the produced thread code size is large enough to reduce the effect of the thread management overhead.

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，現実に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コンパイラの研究開発が行われてきたが，それらはソースコードレベルでマルチスレッド化を行うため，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条件となる．しかしながら，現実問題としてソースコードはそのすべてが参照可能なわけではなく，高速化し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には，それらを高速化することはできない．

この問題に対して，バイナリコード自体を処理の対

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University
†† 東芝 IT ソリューション株式会社
Toshiba IT-Solutions Corporation

象とする**バイナリ変換**（**BT**: **Binary Translation**）技術が解決法となる．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っているため，最適化処理を行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，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．

しかしながら，バイナリコードを変換処理する場合，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発生する．一般に，バイナリコードは命令とデータの区別がなく，たと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の存在や，自己書き換えコードの存在などの理由により，実行前にコードのすべてを解析し変換することは困難である．そのため，バイナリコードを処理の対象とするシステムでは実行時に処理する部分が必要となる．この実行時に処理を行う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，その情報を使った最適化処理を施すことでさらに実行性能を高めることが可能である．これは実行プロファイルに基づくコンパイル手法（**PGC: Profile-Guided Compilation**）と類似しているが，プログラムの各実行局面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[2]．

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プロセッサシステムを前提に，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実行前および実行時に自動的に生成するシステムの構築を行い，アプリケーションの実行性能の向上を目指す[3]．本稿では，システムの基盤技術となるバイナリ変換に基づいたマルチスレッド化コードの自動生成手法についての説明を行い，その有効性を評価するための実験的なバイナリ変換システムの構築を行う[4)~6)]．実際にアプリケーションのシングルスレッドバイナリコードからマルチスレッド化コードを生成し，サイクルベースの命令駆動シミュレータによる性能評価を行う．

2. 基本概念

本研究で開発中のシステムは，マルチスレッドプロセッサシステムを前提とし，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標としている．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 つがあげられる．

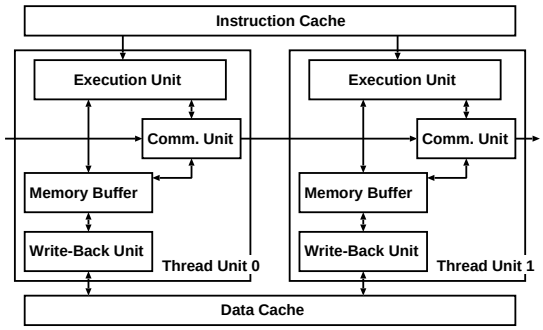

図 1 スレッドユニットの構成
Fig. 1 Configuration of thread units.

2.1 バイナリレベルでのマルチスレッド化

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制御構造に沿った形で，後述のスレッドパイプライニング実行モデル[7]に基づいて行う．

本システムが仮定するマルチスレッド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（図中 **Execution Unit**）に，スレッドの制御，スレッドユニット間の通信・同期を行うための機構（図中 **Comm. Unit**）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ることで，1 つのプロセッサを構成する．対象とするマルチスレッドプロセッサモデルの簡単化のため，スレッド間でのデータ通信はメモリアクセスを介してのみ行うものとする．

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すべ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えない．そのため，メモリアクセスを追跡し，依存が存在する場合に同期をとる機構が必要である．本研究では，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，メモリアクセスごとのチェックにより，真に依存が発生する場合に同期をとる機能を備えたメモリバッファ（図中 **Memory Buffer**）を仮定する．スレッド間依存のあるメモリ参照を行ったスレッドは，依存解消待ち状態に入る．依

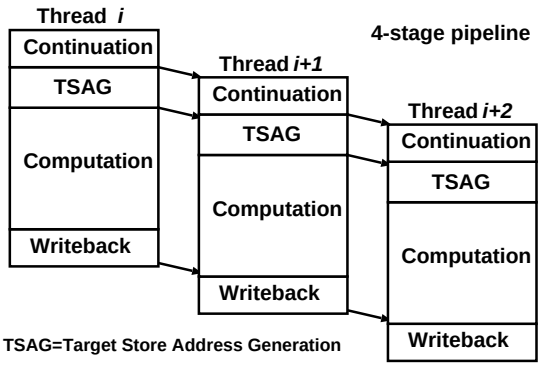

図 2 スレッドパイプライニングモデル
Fig. 2 Thread pipelining model.

存解消待ちのスレッドは，先行スレッドからのデータ転送により依存が解消し次第，その実行を再開する．

また，制御投機のスレッド実行を支援するため，投機実行中のメモリへの書き込みアクセスはすべて Memory Buffer に保持され，スレッドの実行が投機状態でなくなった際に，**Write-Back Unit** により実際に計算結果のメモリへの書き込み処理が行われるものとする．

本システムでは，スレッドパイプライニングの概念を用いたマルチスレッド化を行う．**図 2** にスレッドパイプライニングの概念を示す．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．

スレッド分割後，各スレッドで実行する命令コードを，図に示す **Continuation**，**TSAG**（**Target Store Address Generation**），**Computation**，**Writeback** の 4 つのステージに分ける．

Continuation ステージは，たとえばループにおける誘導変数の更新相当の計算を行うステージで，後続のスレッドが実行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，**TSAG** ステージに入り，ここでスレッド間でデータ依存の可能性があるメモリアクセスのアドレスを，前述の Memory Buffer に対して登録する．

TSAG 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はすべて，Memory Buffer で監視され，スレッド間で真に依存が発生した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う Writeback ステージに入り，スレッド実行の結果の書き戻しを行う．先頭のスレッド以外は，その実行が取り消される可能性があるため，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，結果を正しい順でメモリへと書き戻す．

2.2 バイナリ互換性の確保

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．いかに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．本システムでは，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．中間表現は命令セットアーキテクチャ中立なものを採用し，命令コードから中間表現へのマッピングを行うことで，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しつつ，マルチスレッド実行による高速化を実現する．

2.3 実行時最適化

1 章で述べた理由により，バイナリ変換システムでは実行時に変換処理を行う必要がある．この実行時処理を行う際には，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．たとえば，実行されるパスによってスレッド間の依存関係が変化するコードに対して，実行前変換では，依存関係を保証するために，実行される可能性のあるすべてのパスについて依存関係保証の処理を行う保守的なスレッドコードを生成する必要があるが，実行時処理では，実行局面に応じて，実行される可能性が最も高いパスだけに注目したコードを生成することで，不要な依存関係保証の処理を省き，結果として性能向上を達成することが可能である．

3. システム構成

本研究で提案するバイナリ変換システムのフレームワークについて説明する．**図 3** にシステムの全体構成を示す．本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**）と，プログラム実行時に動作する **DTO**（**Dynamic Translator and Optimizer**）から構成される．以下，STO と DTO について述べる．

3.1 STO

STO は，(1) 実行前のマルチスレッド化コードへの

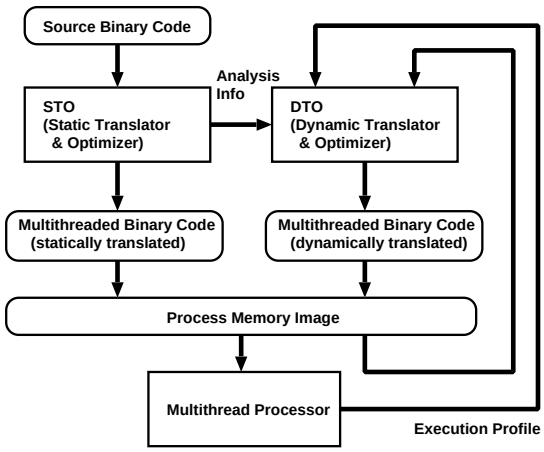

図 3 システム構成
Fig. 3 System diagram.

バイナリ変換処理，(2) DTO が行う変換処理の実行時オーバヘッドを軽減するための解析情報の提供，の大きく 2 つの役割を果たす．

STO は対象となるアプリケーションのバイナリコード（図中 **Source Binary Code**）を入力として，プログラムの実行前に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を出力する．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うことで，実行性能を向上させる．

STO は，プログラム開始地点から制御フロー上追跡可能な範囲にあるコード部分に関して，マルチスレッド化および最適化処理を行う．間接ジャンプ命令などにより追跡不能なコードや，自己書き換えコードなどの実行前に変換処理できない部分については実行時に変換する必要があるが，その処理を後述の DTO が担当する．

また，本システムでは DTO により実行時最適化が行われるが，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情報を STO が作成し，DTO に受け渡す．これにより，実行時最適化処理にかかるオーバヘッドを削減する．

3.2 DTO

DTO は実行時にバイナリコードのマルチスレッド化および最適化を行う．DTO は変換対象となるアプリケーションの実行中に，タイマ割込みによる定期的イベント，および，プログラム中のホットスポット/パスの検出などのイベント発生により起動される．

DTO は STO が実行前に変換できなかったコードに関して，実行時に変換処理を行うが，変換処理自体にかかる時間がアプリケーション全体の実行時間に上乗せされるため，そのすべてを変換処理するのは性能上不利になる可能性がある．そのため，DTO の変換対象を，プログラムのホットスポット/パスとして認識されたコード部分のみに限定する．

マルチスレッドプロセッサ上で実行中のアプリケーションプログラムはその実行と同時に実行時プロファイル情報を収集しており，その情報を基にして，DTO はホットスポット/パスの検出を行う．ホットスポット/パスの検出は，タイマ割込みによるサンプリングと，プロファイルコードの実行の 2 段階の処理で行う．サンプリングによりプロファイルすべき範囲を狭めてから，実際にプロファイルコードを挿入したコードを実行することで詳細なプロファイリングを行う方式をとる．ホットスポット/パスを検出した場合，DTO はマルチスレッド化，および，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う．最適化の結果，生成されたコードをプロセスイメージに反映（生成されたコード側を実行するように命令の書き換えなどを行う）した後，DTO からアプリケーション側に実行を戻す．

4. バイナリ変換によるマルチスレッド化

本章では，前述の STO，DTO の双方に共通する基本的な処理であるバイナリ変換によるマルチスレッド化について説明する．

本研究おけるマルチスレッド化は，プログラムの制御フローに沿った形で，スレッドパイプライニングモデルを基に行う．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よいが，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う．また，ループが多重ループの場合は，その最内ループを変換の対象とする．マルチスレッド化の処理は，以下の 4 段階の作業で行う．

- 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，イテレーション間依存と誘導変数の検出
- 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

4.1 バイナリコード解析と中間表現への変換

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

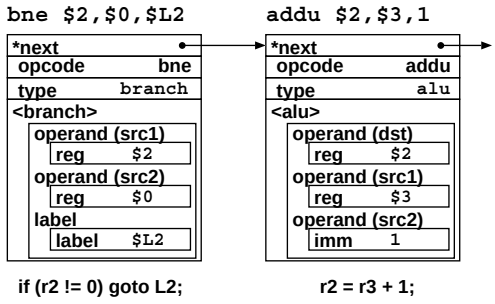

図 4 中間表現の例
Fig. 4 Example of intermediate representation.

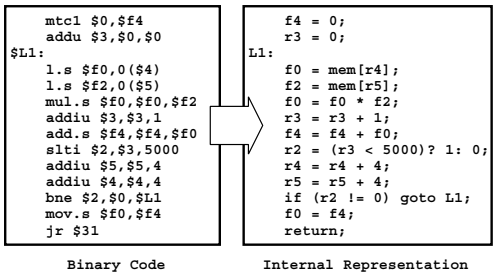

図 5 中間表現への変換例
Fig. 5 Example of conversion to intermediate representation.

現へと変換していく．**図 4** に中間表現の例を示す．この中間表現は命令セット独立な形式であり，この形式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り，中間表現レベルでマルチスレッド化処理を行う．**図 5** にバイナリコードから中間表現への変換例を示す．

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 1 つの操作（あるいは中間言語の 1 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．

バイナリコードによっては，コンパイラの最適化処理などによりトリッキーなコードを含む可能性があり，そのままでは以降の処理が行いにくい場合がある．その場合，コードのイディオム解析や，コードの脱最適化処理により，処理しやすい形に変換する必要があるが，本稿ではこの問題については扱わない．

4.2 基本ブロック分割とループ検出

中間表現への変換後は，基本ブロック分割を行う．基本ブロックの先頭として，(1) コードの開始点，(2) 分岐命令の飛び先，(3) 分岐命令の次の命令，の 3 つの条件を手がかりにして，中間表現レベルで分割する．

本研究ではループ構造に着目したマルチスレッド化を行うため，バイナリコード中からループ構造を検出する必要があるが，ソースコードレベルでの for ループ，while ループは一般的に条件分岐命令に変換されているため，その分岐命令に注目しただけでは if 文などの制御構造と区別がつかない．そのため，ループ構造の検出には制御フロー解析が必要となる．

プログラムのループ構造の検出には，後方分岐（backward branch）を手がかりとして制御フロー解析を行い，制御フロー上，循環している基本ブロック列をループとして検出する．

また，多重ループの場合は最内ループに着目してマルチスレッド化するため，ループの包含関係を調査し，最内ループを探し出す．

4.3 イテレーション間依存解析と誘導変数の検出

ループの検出後は，ループボディ内でのデータフロー解析により，イテレーション間依存のある変数を検出する．変数の使用・定義（use-def）状況を調査し，イテレーションコード内で，変数の定義（def）より先に使用（use）を発見した場合には，その変数をイテレーション間依存があるものとして扱う．

イテレーション間依存変数検出後は，依存変数を，(1) 変数の値の更新に，ループ内不変値（定数など）のみを使用するもの（以下，**ループ不変値更新変数**），(2) それ以外のもの（以下，**ループ変化値更新変数**），の 2 種類に分類する．前者は，Continuation ステージ内で，次スレッドの生成前に計算を行う．後者は，TSAG ステージ内で，そのアドレスを Memory Buffer に登録後，Computation ステージ内でその最終値を次スレッドへ転送する．

図 5 の右側に示す中間表現の場合，**r3**，**r4**，**r5**，**f4** の 4 つの変数が，定義より前に使用されているため，イテレーション間依存変数となる．そのうち，**r3**，**r4**，**r5** の 3 つの変数は，その値の更新に定数（ループ不変である）のみを参照しているので，ループ不変値更新変数となる．一方，変数 **f4** は，その値の更新にループ内で変化する値を参照しているのループ変化値更新変数となる．

ループ不変値更新変数に分類される変数のうち，ループの脱出判定に使用されるものをループの誘導変数として扱う．誘導変数として検出された変数について，値の範囲を検査することで，ループのイテレーション回数を判定する．

図 5 の右側に示す中間表現の場合，ループ不変値更新変数である **r3**，**r4**，**r5** の 3 つの変数のうち，**r3** の

値がループの脱出判定に使用されているので，**r3** をループの誘導変数として検出する．その値の範囲を調べることで，このコードは，初期値 0，増分値 1，終値 5000 の for ループであることが分かる．

4.4 マルチスレッド化コード生成

以上の解析情報を基にして，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，マルチスレッド化コードの生成を行う．

ループの誘導変数を含むループ不変値更新変数の更新コード，ループ脱出条件判定コード，スレッド生成コードなどを Continuation ステージの処理コードとして生成する．

TSAG ステージ内では，ループ変化値更新変数のアドレスを Memory Buffer に登録するコードを生成する．ここで，スレッド間通信はメモリを介して行われるため，変数がレジスタに割り当てられている場合には，スレッド間データ転送用にアドレスを確保して，Memory Buffer に登録する．

Computation ステージに本来の計算を行うコードを生成する．Computation ステージ内では，ループ変化値更新変数の最終更新時点において，スレッド間通信命令を生成することで，その最終的な値を次スレッドに転送する．

最後に，Writeback ステージに，スレッド終了のための命令を生成する．

4.5 コード生成例

コード生成の例として，**図 6** に内積値計算プログラムの変換前後のアセンブリコードを示す．図の左が変換前のシングルスレッドコード，右が変換後のマルチスレッドコードである．使用されている命令セットはスレッドパイプライニングモデル[7]に基づいたアーキテクチャシミュレータ **SIMCA**[8]のものである．マルチスレッド化により，**表 1** に示すスレッド制御命令が付加されている．

図左のバイナリコードを解析することで，レジスタ 3，4，5 がループ不変値更新変数，レジスタ f4 がループ変化値更新変数であることが判明する．この変数情報を基にして，各ステージのコードを生成していく．

Continuation ステージにおいては，ループ脱出判定コードと，ループの誘導変数を含むループ不変値更新変数（図中のレジスタ 3，4，5）更新コードに続いて，次スレッドの生成命令が生成される．

スレッド間のデータ転送はメモリを介して行うため，データ転送用にメモリアドレスを確保する必要がある．本稿では，スタック空間上のアドレスで，使用されていないアドレスを使用することとした．図 6 では，ループ変化値更新変数（図中左のレジスタ f4）の値の転送用のアドレスとしてレジスタ 28（スタックポインタ）で指すアドレスを使用している．TSAG ステージで，このアドレスを **altsw** 命令により Memory Buffer に登録する．

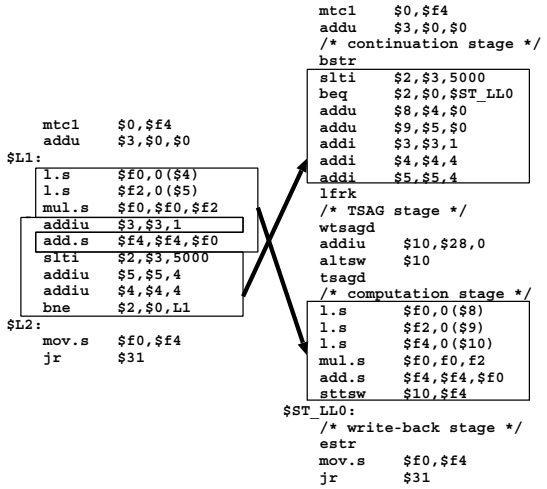

図 6 バイナリレベルでのマルチスレッド化の例
Fig. 6 Example of binary-level multithreading.

表 1 スレッド制御命令
Table 1 Thread management instructions.

命令	処理内容
bstr	マルチスレッド実行領域の開始地点の設定
estr	マルチスレッド実行領域の終了地点の設定
lfrk	次スレッドの生成
tsagd	次スレッドへの TSAG ステージ終了の通知
wtsagd	前スレッドの TSAG ステージ終了まで待機
altsw	指定アドレスの Memory Buffer への登録
sttsw	指定した値のメモリ書き込みと次スレッドへの通知

Computation ステージでは，イテレーション本来の計算を行い，ループ変化値更新変数については，**sttsw** 命令を用いることで，TSAG ステージで登録したアドレス（レジスタ 10 で指すアドレス．つまりレジスタ 28 で指すアドレス）を介して次スレッドへ値を転送する．

最後に，Writeback ステージの実行を行う **estr** 命令を生成して，スレッドコードの生成を完了する．

4.6 ループアンローリング

図 6 において，マルチスレッド化前後の 1 イテレーションの命令数を比較すると，変換前が 9 命令であるのに対して，変換後では 20 命令に増えている．このように，変換前のイテレーションコードのサイズが小さい場合，本来の計算部分（Computation ステージ）に対して，スレッド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可能性が大きい．そのため，1 スレッドあたりの処理コードのサイズを大きくすることで，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．

本研究では，ループアンローリング処理を施すことで，この問題を解決する．ループアンローリング処理は，4.1 節から 4.4 節で説明したマルチスレッド化処理を行う前のループに対して施す．具体的には，中間表現レベルで表現されているループ構造に対して，そのループボディ部分のコードを複製することで実現する．その後，アンローリング後のループに対して，前述のマルチスレッド化処理を行う．

ループアンローリングを行う場合，元のループの総イテレーション回数が，アンローリング後の総イテレーション回数の倍数にならず，端数が出る場合を考慮しなければならないが，ループ脱出後に，端数回数分のイテレーションを実行するループコードを生成することで対処する．

5. 評　　価

5.1 実験システム

3 章で説明した STO と DTO の双方に共通する基本的な処理で，提案システムでの重要な基盤技術となるバイナリレベルでのマルチスレッド化処理の実現性および有効性を検証するため，実験的なバイナリ変換システムを試作した．**図 7** に実験システムの構成を示す．

図中の **MultiThread Code Generator** が対象であるシングルスレッドバイナリコード（図中 **Single-thread binary code**）とマルチスレッド化の対象とするコード領域の指定（サブルーチン単位で指定）を入力とし，4 章で述べた手順に従って自動的にマルチスレッド化処理を行い，マルチスレッド化されたアセンブリコード（図中 **Multithreaded partial assembly code**）を出力する．

現在のところ，本実験システムでは，マルチスレッド化の変換対象とするコード部位の指定とループアンローリング回数のみは手動で与える必要があるが，これらの指定を除けば，4 節で述べたバイナリレベルでのマルチスレッド化処理はすべて自動化している．

このうち，変換対象コード部位について手動で指定する必要があるのはサブルーチンの開始アドレスのみであり，マルチスレッド化の対象とするループの選定は指定されたサブルーチンの開始アドレスからの制御フロー解析により自動的に行われる．この処理は将来的には，プロファイル情報から自動的に変換対象コードを選定することにより，完全自動化する予定である．

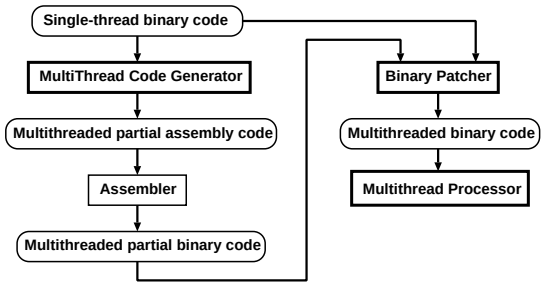

図 7 実験システムの構成
Fig. 7 Configuration of pilot system.

また，アンローリング回数についても，アンローリング回数を変えて生成したコードのサイズなどの情報を基に，最も性能が向上する可能性のあるものを選択するなどの処理により自動化する予定である．

出力されたアセンブリコードをアセンブルすることにより，部分バイナリコード（図中 **Mutithreaded partial binary code**）を生成する．この部分バイナリコードとオリジナルのシングルスレッドバイナリコードを図中の **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（図中 **Multithreaded binary code**）を生成する．この Binary Patcher は，対象サブルーチンが呼び出された際，マルチスレッド化されたコード側が呼び出されるように，オリジナル側にマルチスレッド化コード側へのジャンプ命令を挿入する．

本実験システムにより，バイナリレベルでシングルスレッドコードからマルチスレッド化コードの生成を行い，その実行性能について評価を行う．

5.2 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，SimpleScalar をベースにした，スレッドパイプライニングモデル[7]を実現するアーキテクチャシミュレータ **SIMCA**[8]を用いる．SIMCA の内部構成は前述の図 1 に示すとおりであり，**表 2** に示すシミュレーションパラメータを用いて評価を行った．

前述した実験的なバイナリ変換システムにより，対象となるシングルスレッドバイナリコードから評価プログラムのマルチスレッド化コードを生成し，SIMCA 上で実行する．対象となるシングルスレッドバイナリコードは，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．評価用アプリケーションとして，台形公式を用いた sin 関数の積分値計算プログラム，内積値計算プログラム，SPECint95 の ijpeg を用いる．前二

表 2 シミュレーションパラメータ
Table 2 Simulation parameter.

スレッドユニット	4 命令同時実行 out-of-order 実行
1 次キャッシュ （命令）	direct-map 16 KB（ラインサイズ 32 B） レイテンシ 1 クロック
1 次キャッシュ （データ）	4-way set-associative 16 KB（ラインサイズ 32 B） レイテンシ 1 クロック
2 次キャッシュ （命令・データ混在）	4-way set-associative 256 KB（ラインサイズ 64 B） レイテンシ 6 クロック
メモリ	レイテンシ 18 クロック
メモリバッファ	512 エントリ
スレッドユニット間 通信ポート	レイテンシ 1 クロック

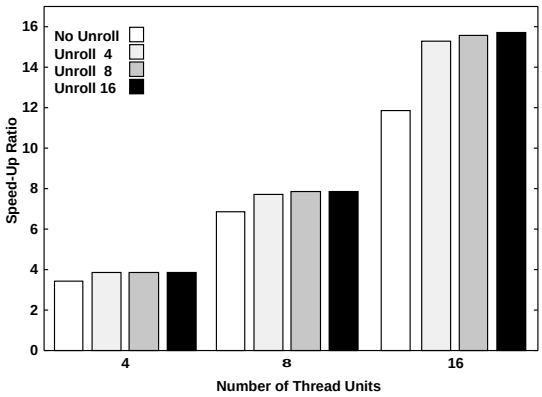

図 8 積分値計算プログラムの速度向上率
Fig. 8 Speed-up ratio in integral calculation.

者は数値計算を目的とした比較的小規模のプログラムであり，ijpeg は実用アプリケーションプログラムである．

評価は，各アプリケーションをシングルスレッド実行とマルチスレッド実行により，それぞれの変換対象コード部分についての実行クロック数を計測し，速度向上率（=シングルスレッド実行クロック数/マルチスレッド実行クロック数）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，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評価を行う．

5.3 評価結果

図 8 に sin 関数の積分値計算の速度向上率を示す．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 と増えるに従い，速度向上率は 4 倍，8 倍，16 倍となる．アンローリングを行わない場合でも，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なくても高速化は可能であるが，ループアンローリングによりさらに性能が上がり，スレッドユニットの台数に対してほぼリニアな性能向上率が得られることが分かった．

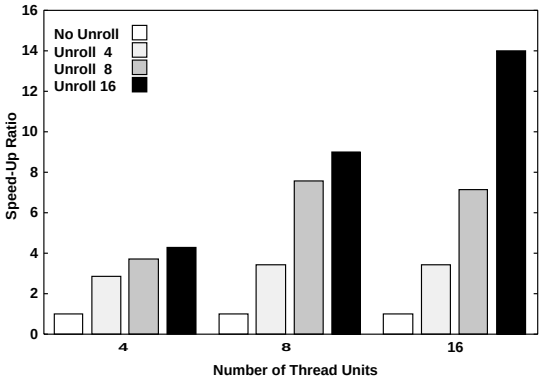

図 9 内積値計算プログラムの速度向上率
Fig. 9 Speed-up ratio in inner product calculation.

図 9 に内積値計算の速度向上率を示す．速度向上率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い場合は，どの台数の場合でもほぼ 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 2.8 倍，3.5 倍，3.5 倍．アンローリング回数が 8 の場合は，それぞれ 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 4.6 倍，9.1 倍，13.9 倍となった．アンローリングを行わない場合に性能が出ない原因は，1 イテレーションあたりの処理サイズが小さいため，スレッド制御にかかるオーバヘッドの影響を受けるためである．オーバーヘッドの影響を軽減するためにアンローリングを行う場合，スレッドユニット台数が 8 台で 8 回，16 台で 16 回のアンローリングと，スレッドユニット台数に応じた回数分のアンローリングが必要である．

これらの結果より内積計算は，スレッドユニット台数に応じた回数分のアンローリングを行うことで，スレッドユニット台数に比例した速度向上率を得ることができることが分かった．

図 10 に SPECint95 の ijpeg の速度向上率を示す．速度向上率は，ijpeg で頻繁に実行されるコードの中から変換対象として抽出したサブルーチン部分の実行時間を計測して算出したものである．プログラム中で頻繁に実行されるコード部分の特定には，PC（プログラムカウンタ）の定期的なサンプリングによるプロファイリングを行った結果[9]を用いた．サンプリング

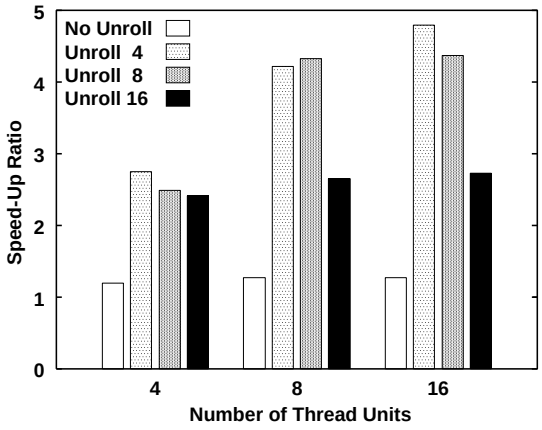

図 10 ijpeg（対象部分コード）の速度向上率
Fig. 10 Speed-up ratio in ijpeg.

したPCの値を基本ブロック単位で集計し，上位のサブルーチンを変換の対象コードとした．今回は，ijpeg中の上位2つのサブルーチン **forward_DCT()** および **rgb_ycc_convert()** を変換の対象として選定し，それに含まれるループ構造を実験システムでマルチスレッド化し，その性能向上率を評価した．データセットは train を使用している．

速度向上率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い場合，1.2 倍から 1.3 倍であった．アンローリング回数が 4 の場合は，それぞれ 2.8 倍，4.2 倍，4.8 倍，回数が 8 の場合は，それぞれ 2.5 倍，4.3 倍，4.4 倍，回数が 16 の場合は，それぞれ 2.4 倍，2.7 倍，2.7 倍となった．

アンローリングをまったく行わない場合に，スレッドユニットの台数を増しても性能が上がらないのは，スレッド制御のオーバヘッドの影響を受けているためと考えられる．アンローリングを行うことで，スレッド制御のオーバヘッドの影響を軽減することができ，実際に 4 回アンローリングの場合は性能が向上している．しかしながら，アンローリング回数を大きくしすぎると性能が下がっている．これは，マルチスレッド化の対象であるループの総イテレーション回数が 100 回以下と少ないため，アンローリング回数を大きくすると，スレッドユニットごとに実行されるイテレーション回数が大きくなり，結果として並列実行できるスレッドユニット数が少なくなってしまうからである．

6. 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．その例として，Compaq 社の FX!32 [10]，IBM 社の DAISY [11]，BOA [12]，Transmeta 社の Crusoe プロセッサ [13]，HP 社の Dynamo [14]，ハーバード大学の Morph [15]，同じく，ハーバード大学の Deco [16] があげられる．以上にあげたシステムはいずれも基本的に変換前および変換後の命令セットの組合せが固定であるが，これを任意の組合せで行う研究として，UQBT [1] があげられる．

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．

また，バイナリ変換によるマルチスレッド化の類似研究として，文献 17) があげられる．これはプログラムループ構造に着目してバイナリコードをマルチスレッド化する点で類似するが，実行時にハードウェアにより行うものである．本研究は，ソフトウェア処理である点と，実行前に可能な限り対象バイナリの解析および変換処理を行うことで，実行時処理のオーバヘッド軽減を目指す点で大きく異なる．

バイナリ変換技術によるものではないが，自動ベクトル化および自動マルチスレッド化の関連技術として，Intel の最適化コンパイラ [18] があげられる．

7. 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について述べた．本システムではソースコードの参照が必要なく，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．

本稿では本システムの基盤となるバイナリレベルでのマルチスレッド化コード生成手法の評価を行うために，実験的なバイナリ変換システムの構築を行い，アプリケーションプログラムを用いて評価を行った．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スレッド制御にかかるオーバヘッドの影響を小さくし，高速化可能であることを示した．しかし，マルチスレッド化の対象であるループのイテレーション回数が少ない場合はアンローリング回数の増加

は性能向上に寄与しないため，適切な回数のアンローリングを行う必要があることが分かった．

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（B）課題番号 14380135，基盤研究（C）課題番号 14580362，若手研究（B）課題番号 14780186 の援助による．

参 考 文 献

1) Cifuentes, C. and Emmerik, M.V.: UQBT: Adaptable Binary Translation at Low Cost, *Computer*, Vol.33, No.3, pp.60–66 (2000).
2) Smith, M.D.: Overcoming the Challenges to Feedback-Directed Optimization, *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization* (*Dynamo'00*), pp.1–11 (2000).
3) 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），pp.41–46 (2001).
4) 小野喬史，大津金光，横田隆史，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とその初期評価，情報処理学会研究報告（ARC144），pp.183–188 (2001).
5) Ootsu, K., Yokota, T., Ono, T. and Baba, T.: A Binary Translation System for Multithreaded Processors and its Preliminary Evaluation, *5th Workshop on Multithreaded Execution, Architecture and Compilation* (*MTEAC-5*), pp.13–21 (2001).
6) 大津金光，小野喬史，横田隆史，馬場敬信：バイナリ変換によるマルチスレッド化コード生成手法とその評価，並列処理シンポジウム JSPP2002，pp.261–268 (2002).
7) Tsai, J.-Y., Huang, J., Amlo, C., Lilja, D.J. and Yew, P.-C.: The Superthreaded Processor Architecture, *IEEE Trans. Computers, Special Issue on Multithreaded Architectures*, Vol.48, No.9, pp.881–902 (1999).
8) Huang, J.: *The SImulator for Multi-threaded Computer Architecture* (*SIMCA*), *Release 1.2*.
9) 青木政人，小野喬史，大津金光，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 回全国大会論文集，第 1 分冊，pp.101–102 (2002).
10) Hookway, R.J. and Herdeg, M.A.: DIGITAL FX!32: Combining Emulation and Binary Translation, *Digital Technical Journal*, pp.3–12 (1997).
11) Ebcioglu, K. and Altman, E.R.: DAISY: Dynamic Compilation for 100% Architectural Compatibility, *24th Annual International Symposium on Computer Architecture*, pp.26–37 (1997).
12) Sathaye, S., Ledak, P., LeBlanc, J., Kosonocky, S., Gschwind, M., Fritts, J., Filan, Z., Bright, A., Appenzeller, D., Altman, E. and Agricola, C.: BOA: Targeting Multi-Gigahertz with Binary Translation, *Workshop on Binary Translation* (*Binary99*) (1999).
13) Klaiber, A.: The Technology Behind Crusoe Processors, Technical report, Transmeta (2000).
14) Bala, V., Duesterwald, E. and Banerjia, S.: Dynamo: A Transparent Dynamic Optimization System, *Programming Language Design and Implementation* (2000).
15) Zhang, X., Wang, Z., Gloy, N., Chen, J.B. and Smith, M.D.: System Support for Automatic Profiling and Optimization, *16th Symposium on Operating Systems Principles* (2000).
16) Feigin, E.J.: A Case for Automatic Run-Time Code Optimization, Master's thesis, Senior thesis, Harvard College, Division of Engineering and Applied Sciences (1999).
17) Hiraki, K., Tamatsukuri, J. and Matsumoto, T.: Speculative execution model with duplication, *International Conference on Supercomputing*, pp.321–328 (1998).
18) Bik, A., Girkar, M., Grey, P. and Tian, X.: Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems, *Intel Technology Journal* (2001).

(平成 14 年 6 月 7 日受付)
(平成 14 年 8 月 30 日採録)

大津 金光（正会員）

1993 年東京大学理学部情報科学科卒業．1995 年東京大学大学院修士課程修了．1997 年東京大学大学院博士課程退学．同年より宇都宮大学工学部助手となり現在に至る．高性能計算システム，マルチスレッド処理，実行時最適化システム等の研究に従事．理学修士．FPGA/PLD Design Conference 審査委員特別賞（2002）受賞．

小野 喬史

2000 年宇都宮大学工学部情報工学科卒業．2002 年宇都宮大学大学院博士前期課程修了．現在，東芝 IT ソリューション株式会社勤務．工学修士．

横田　隆史（正会員）

1983 年慶應義塾大学工学部電気工学科卒業．1985 年慶應義塾大学大学院電気工学専攻修士課程修了．同年三菱電機（株）に入社．1993 年 12 月から 1997 年 3 月まで新情報処理開発機構（RWCP）に出向．2001 年 4 月より宇都宮大学工学部助教授．計算機アーキテクチャ，設計方法論等の研究に従事．工学博士．ICCD Outstanding Paper Award（1995），FPGA/PLD Design Conference 審査委員特別賞（2002）受賞．電子情報通信学会，IEEE 各会員．

馬場　敬信（正会員）

1970 年京都大学工学部数理工学科卒業．1975 年京都大学大学院博士課程単位取得退学．同年より電気通信大学助手，講師を経て，現在宇都宮大学工学部教授．工学博士．1982 年より 1 年間メリーランド大学客員教授．計算機アーキテクチャ，並列処理等の研究に従事．1992 年情報処理学会 Best Author 賞，2002 年 FPGA/PLD Design Conference 審査委員特別賞受賞．著書「Microprogrammable Parallel Computer」（MIT Press),「コンピュータアーキテクチャ（改訂 2 版）」（オーム社）等．電子情報通信学会，IEEE 各会員．