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

2005－ARC－162 （8）
2005－HPC－101 （8）
2005／3／7

プログラムの信頼性向上を目的とした補助スレッドの効率的利用

本山　英典[†] **古関　聰**[‡] **小松　秀昭**[‡] **深澤　良彰**[†]

概要 複雑化したシステムでは、プログラムのエラーの動的な検出が重要である。このためには、C 言語における Assertion のなどの動的エラー機構は有効であるが、それによる実行速度の遅延は少なくない。一方、高速キャッシュメモリを複数のスレッドで共有するマルチスレッドプロセッサが広く利用されるようになってきている。マルチスレッドプロセッサでは、スレッド間でデータ通信を高速に実行することができる。本稿では、プログラムからエラー検出コードを抜き出し、補助スレッドで実行させることにより、エラー検出にかかる実行速度の遅延を隠蔽する手法を提案する。この手法をサンプルプログラムとベンチマークプログラムに適用した結果、それぞれ遅延の 46%, 68%を隠蔽することができた。

Efficient Useage of Helper Thread for Software Reliability Improvement

Hidenori MOTOYAMA[†], Akira KOSEKI[‡], Hideaki KOMATSU[‡] and Yoshiaki FUKAZAWA[†]

Abstract The dynamic detection of program errors has becoming important in complicated systems. Although dynamic error detections such as Assertion in C language are effective, the execution latency caused by them is not negligible. On the other hand, multithreaded processors in which several threads share the high-speed cache memory have been widely used. In these processors, data can be transfered very fast between threads. In this paper, we propose a technique which hides such latency by allocating error detection code extracted from the original program to a helper thread. Applying this technique to sample programs(sorting programs) and the benchmark programs, 46 % and 68 % of the latency for error detections could be hidden respectively.

1　始めに

SMT(Simultaneous Multi-Threading) 技術 [1] やマルチコア化などにより、CPU リソースは今後余ると考えられる。本稿ではこのリソースをソフトウェアの信頼性にあてる手法を提案する。信頼性の向上にあたり、システムの実行速度の遅延を招かないように、システムを実行する主スレッド MT（Main Thread）とエラーを発見する補助スレッド HT(Helper Thread) によって構成し、並列実行を行う。HT はエラー検出コードに対して依存関係が張られているコードを MT から複製・切り取ることで生成するが、これによりスレッド間のコードには依存関係（真依存・出力依存・制御依存）が発生し、これを解消するためにデータ通信・同期が必要となる。本手法では、PDG(Program Dependency Graph) でこれらのデータの共通先行節を見つけ、スレッド間の通信量を削減し、出力依存関係のあるデータの通信にバッファを用いることで依存を緩和させ、同期命令を削減させる。また、エラー検出コードの実行コストと通信コストを比較し、MT/HT に割り当てを行うことで、MT の実行コストを最適化する。さらに、複雑な制御依存に対して、投機的にエラー検出コードを実行させる。これにより HT の実行は MT の制御フローに依存されないため、HT をより最適することができる。以上により、MT のエラー検出コードのためにかかる実行コストを削減し、かつ HT の実行も効率よく実行させることができる。

[†]早稲田大学大学院理工学研究科
Graduate School of Science Engineering, Waseda University
[‡]日本 IBM（株）東京基礎研究所
Tokyo Research Laboratory, IBM Japan Ltd.

2　従来研究と本手法の特徴

2.1　従来研究

複数のプロセスを用いたソフトウェア信頼性の手法として、シャドウプロセッシング [2] がある。この手法は、エラー検出コードが挿入されているオリジナルプログラムから、エラー検出コードの実行のみを行う HT とエラー検出コードを削除した MT から構成されている。HT はエラー検出コードに依存関係のあるコードをスライシングすることによって生成する。そのため、HT はエラー検出に必要な最小限のコードで構成し、オリジナルプログラムの制御フローを再現してエラー検出を行うことができる。この手法を用いると、HT と MT 間でデータ通信を行う必要がなくなり、プロセス間の同期にかかるオーバーヘッドがほとんど発生せず、MT ではエラー検出コードにかかるオーバーヘッドのほとんどを削除することができる。

しかし、相互入力やシステムコールなどについては HT では正確に再現することができないため同期を取る。また、データベースの書き込みのチェックなど、出力に対するエラー検出が必要な場合は同期をとる必要がある。このように同期を取る必要があるプログラムで、複雑な制御フローをスライシングで再現しようとすると、元のプログラムとほとんど変わらない HT となり、最適化も掛かりにくい。そのため、HT は MT より実行が大幅に遅れてしまい、同期で MT が HT の実行を待つ必要がでてくる。

また、SMT 環境で複数のスレッドを用いて CPU のリソースを有効に活用する手法として、キャッシュミス先行実行手法 [3] がある。この手法は、プログラム内のメモリ参照を引き起こすロード命令を、補助スレッドで先行してロードしキャッシュにデータを移す、プリフェッチ技術である。この手法は HT が MT に対して、常に一定距離範囲で先行して実行しなければプリフェッチの効果が得られない。そのため、HT と MT の間に細かく同期を張る必要がある。しかし、この同期はオーバーヘッドが大きく、現在のような、同期をハードウェア機構で高速に実現できる環境でないと、HT でプリフェッチの効果を得るのは難しい。

2.2　本手法の特徴

本稿では、文献 [2] と同様に、エラー検出コードを HT で実行することで、その実行にかかるオーバーヘッドを隠蔽することを目的とする。文献 [2] のプログラムスライシングによって生成された HT は、MT と HT の実行を分離し同期を極力とらないことで、MT の実行速度の遅延を隠蔽した。本手法の特徴は、以下の 3 つである。

- 各種依存関係 (制御依存関係, 真依存関係, 出力依存関係) の緩和し、通信コストの低い箇所でのデータ、MT のデータ通信量・同期のオーバーヘッドの削減
- MT の実行コストを考慮した、Assertion の割り当て
- Assertion の投機的実行による HT の最適化

これにより、MT のエラー検出にかかるオーバーヘッドを最小化し、HT の実行時間を最適化し、MT/HT の同期によって引き起こされる遅延も改善することができる。

例えば、図 1(a) のノード 5-7 の制御依存関係は、(b) のように変形して真依存関係に変換し、Assertion 判別の要素とする。よって、プログラムスライシングを行うと、制御依存関係の複雑さから最適化がかけにくいが、本手法では制御フロー情報と Assertion 内の判別式は独立して実行することができ、最適化を効果的にかけることができる。

そして、スレッド間に真依存関係のあるノード

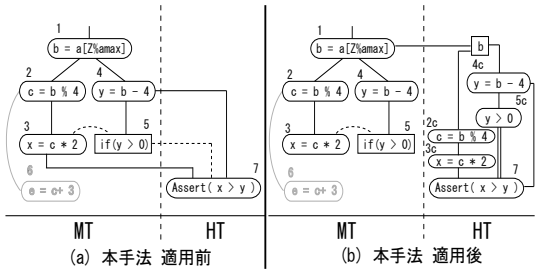

図 1: 本手法の適用例

は依存関係が最小になるように変形を行い、通信量を削減する。

また、残った依存関係はバッファを用いて出力依存関係を緩和させる。文献 [3] のような HT をデータプリフェッチに利用する場合には。スレッド間の実行距離を一定範囲に保つために細かな同期命令が必要だった。しかし、本研究で使用する Assertion は偽と判断されるとプログラムを停止するため、スレッド間の距離を一定に保つ必要ない。そのため、同期にかかるオーバーヘッドは少なく、既存の同期方法で十分効果を得ることができる。

また、Assertion の割り当てについては、各種依存関係の緩和で、通信量が最小化することができな

かった場合は、Assertion を MT で実行させ、MT のオーバーヘッドを最小化する。

そして、各種依存関係の緩和された各 Assertion は実行順序に依存関係が存在しなく、Assertion と制御フロー情報も依存関係がなくなる。そのため、Assertion の実行順序の変更や、制御フロー情報が決定する前に投機的に Assertion を実行させることが可能となる。投機的に実行させることで、HT の無駄な停止時間を有効に活用することができる。

3　本手法の概要

本章では、まず、本手法の概要を提示し、次に HT 生成のステップに関して説明する。

本手法では、MT と HT のコード間に発生する依存関係（真依存・出力依存・制御依存）を緩和する方法として以下の 1〜2 を適用し、Assertion のスライスを作成する。

1. 制御依存関係の緩和（3.1 節)
2. 真依存関係の緩和 (3.2)

そして、生成された複数のスライスに対して以下の 3〜6 を適用し、HT で実行する Assert 文の選択・変形を行う。

3. 出力依存関係の緩和 (3.3)
4. データ通信 (3.4 節)
5. Assertion の割り当て (3.5 節)
6. Assertion の投機的実行 (3.6 節)

HT で Assertion を実行するためには、MT/HT の間に張られた依存関係を考慮する必要がある。そのため、スレッド間でデータをやり取りすることで、プログラムの正確性を保証する。本手法では、Assertion のスライスに真/制御依存関係の緩和を施すことによって、データのやり取りによって生じる MT のオーバーヘッドを最小限にする。

そして、MT/HT のコード間に張られた依存関係を最小化した後、残った依存関係をスレッド間で共有する必要がある。本手法では、このデータに対して生存グラフを作成し、出力依存関係を調べる。そして、出力依存が存在する場合には、MT がそのデータをバッファに保存させデータ通信を行う。これにより、スレッド間で細かい同期を取る必要がなくなり、MT の同期による遅延を回避できる。また、HT では Assertion 実行に必要なデータがそろった時点で実行ができるため、より積極的に最適化をかけることができる。

そして、以上で生成された各 Assertion のスライスを HT で実行するか否かを判別する。本稿では MT にかかるオーバーヘッドを最小限にするため、データ通信によるオーバーヘッドと Assertion を実行するのにかかるオーバーヘッドを比較する。そして、HT でスライスを実行させるか、MT で Assertion を実行するかを決定する。

さらに、制御フローが複雑な Assertion に対して、HT で投機的に実行させ、MT で制御フロー情報を後から通信し、実行した Assertion の有効の有無を判別する。これにより変形されたスライスは、MT の制御フローに影響を受けないため、より効率よく Assertion を実行させることができる。

以上のように生成されたスライスを最適化し、HT にする。以下の節では、各手法について補足する。

3.1　制御依存関係の緩和

本稿では、MT/HT のコード間に張られた制御依存関係をすべて真依存関係に変換を行う。HT に割り当てた Assertion と MT のコードで制御依存のあるノードにエッジを張る。そして、エッジの張られたノードの先行節を見つけ、そのノードの後続節を HT にコピーする。

例えば、図 2 の (a) のように、ノード 5 と 7 には

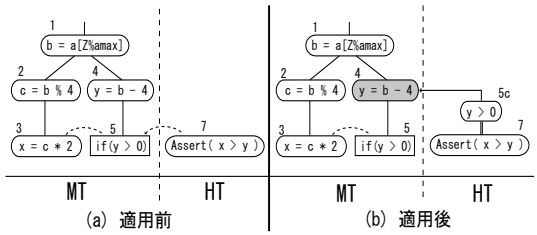

図 2: 真依存関係への変換

制御依存のエッジが破線で示されている。そこで、エッジの張られたノード 5 の先行節であるノード 4 を見つける。そして、ノード 4 の後続節ノードであるノード 5 を HT にコピーし、ノード 5c とする。これにより、MT/HT のコード間に張られていた制御依存関係は、ノード 4 と 5c の真依存関係に変換することができる。そして、ノード 5c, 7 の実行順序は依存せず、最後に 2 つの結果の論理和をとる。

また、真依存関係に変換したエッジは 3.2 節で示しすように、共通先行節を見つけ、MT/HT のコード間に張られた依存関係を減少させる。このとき、MT/HT のコード間に張られた真依存関係を減少できない場合には、MT から制御フロー情報の通信を行わせる。

3.2 真依存関係の緩和

Assert(EXP) の EXP を構成する要素について PDG を作成し、真依存関係のあるノードに対して新たにエッジを張る。エッジが張られたノードの先行節の中で共通なものを探し出し、そのノードの Sucessor ノードを HT にコピーをする。図 3 の (a)

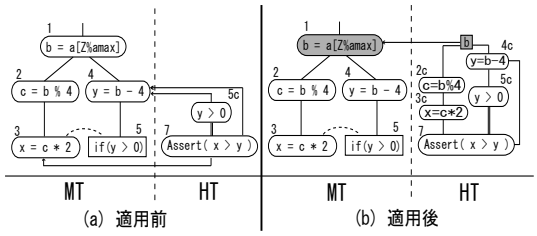

図 3: PDG による共通先行節の検索

で、Assertion と MT のコードに張られたエッジは 3-7, 4-7, 4-5c がある。そこで、ノード 3 とノード 4 の先行節を検索する。ノード 3 の先行節はノード 2 とノード 1 で、ノード 4 はノード 1 となる。このことから、ノード 3 とノード 4 の共通先行節はノード 1 と決定する。決定後は、ノード 1 の Sucessor ノードを HT にコピーする。コピーの範囲は、共通先行節ノードから最初にエッジが張られていたノードまでとする。つまり、図 3 では、ノード 2, 3, 4 をコピーし、図 3 の (b) のようにノード 2c, 3c, 4c とする。以上のようにすることで、MT/HT のコード間に張られた真依存関係を減少させることができる。

3.3 出力依存関係の緩和

3.1 節で見つけた共通先行節ノードや制御フロー情報をスレッド間でで共有するためには、MT のコードの出力依存関係を考慮し、プログラムの正確性を保証しなければならない。図 4 の (a) のように、ノー

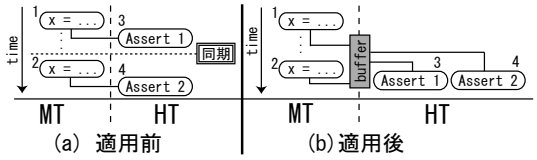

図 4: 出力依存関係の緩和

ド 1, 2 は出力依存関係を持つ。ここで、ノード 3, 4 がそのデータを必要とするときに、ノード 2 で x の値が再定義される前に同期を取る必要がある。このノード 1, 2 の時間の間隔が狭すぎる場合 (変数 x の生存区間が短い場合) には、HT が遅延を起こすと、MT の実行にも影響を及ぼす。

ここで、出力依存の時間間隔が狭い場合には、図 4 の (b) のように、MT でデータをバッファに保存させる。これによって、HT の遅延による MT の遅延を回避することができる。HT ではデータさえそろっていれば、ノード 3, 4 の実行順序に制約がなく、HT をより積極的に最適化をすることができる。

3.4 データ通信

スレッド間でデータを共有する際には通常バッファと循環バッファの 2 種類を用いる。MT が HT に受け渡すデータ数が判定可能でメモリに十分入りきる大きさの場合は通常バッファを用いる。

また、データ通信量が判定不可能か容量が多すぎ

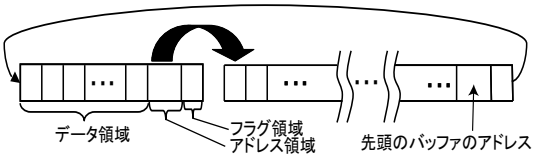

図 5: 循環バッファ

る場合には、図 5 に示すような循環バッファを用いる。循環バッファは通常バッファを複数列並べて構成し、各バッファにはデータ領域, アドレス領域, フラグ領域を設定する。データ領域には MT が HT に通信する値を保存する。アドレス領域は、次のバッファの先頭アドレスを保管する。ただし、最後のバッファのアドレス領域にはバッファの先頭アドレスを保管する。そして、MT/HT がデータの書き/読みの終了を判別するために、バッファの最後に 1bit のフラグ領域を設ける。このフラグが 1 のときは MT のデータ書き込みが終了、0 のときは HT の読み込みが終了したことを意味する。

このようにバッファを構成することで、データ通信の量が判断不可能でも量が多い場合でも、対応することができる。また、各スレッドで他方のスレッドの進行状況の確認回数を削減することができ、効率的にデータ通信を行うことができる。

3.5 Assertion の割り当て

本研究の目的は MT の実行速度の遅延を低減することにある。よって、MT のデータ通信による遅延が Assertion 実行による遅延を上回る場合、MT に Assertion を実行させる。そこで、各 Assertion に対して判定にかかる実行コストとデータ通信 (同期・バッファ) にかかる実行コストを算出し、Assertion の割り当て先を HT にするか否かを判別する。

$Assert_k$ を実行するか否かを判定するに

は、MT から通信する必要があるデータを $\{Data_1, Data_2, ..., Data_j, ..., Data_n\}$ とし、$Data_j$ を必要とする Assertion の集合を $\{Assert_{j1}, Assert_{j2}, ..., Assert_{jl}, ..., Assert_{jm}\}$ とする。そして、$Assert_{j1}, \sim, Assert_{jm}$ と $Data_j$ について、実行/通信回数の比率を $ratio(Assert_{j1}), \sim, ratio(Assert_{jm}), ratio(Data_j)$ とし、実行 1 回にかかるコストを $Cost(Assert_{j1}), \sim, Cost(Assert_{jm}), Cost(Data_j)$ とする。このとき、$Assert_k$ を実行するコスト (AEC) を

$$AEC = Cost(Assert_k)$$

とし、データ通信にかかる実行コスト (DCC) を、

$$ratio_{ASS}[j] = \sum_{k=1}^{m} ratio(Assert_{jk})$$

$$DCC = \sum_{i=1}^{n} \left(\frac{ratio(Data_i)}{ratio_{ASS}[i]} \times Cost(Data_i) \right)$$

と求める。そして、以下のように Assertion を MT/HT のどちらに割り当てるかを判別する。

$AEC - DCC > 0$: HT で $Assert_k$を実行 …(1)
$AEC - DCC \leq 0$: MT で $Assert_k$を実行 …(2)

(2) の場合は $Data_j$ を必要とする Assertion の集合の中から $Assert_k$ を除去する。

実行比率が静的に解析できない場合は、プログラムのプレ実行を行い平均値を求めて利用する。また、実行コストの見積もりはプロセッサのパフォーマンスガイドに従い行う。

3.6 投機的な Assertion 実行

投機的な Assertion の実行とは、制御フロー情報の入手に時間がかかる場合、Assertion の判別式の計算ができるものから先に実行し、HT の無駄な待機時間を削減するものである。

ここでは、3.1 節で述べたような、各 Assertion に対して行うのではなく、すべての Assertion に対して適用する。つまり、$Assert_1, ..., Assert_n$ の制御フロー情報 $Flag_1, ..., Flag_n$ が存在するとき、$Assert_1, ..., Assert_n, Flag_1, ..., Flag_n$ を実行可能な順番に置き換える。そして、Assertion の判別式と制御フロー情報の計算の両方が終わったものから、その結果の論理和をとりエラーか否かを確認する。この方法により、HT の実行を最適化し、無駄な待機時間を削減することができる。 また、ループに

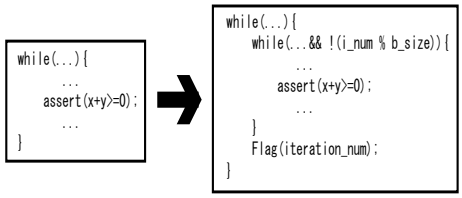

図 6: ループ変形による投機的実行の効率化

適用する場合は、図 6 に示すようにループを二重に変形する。そして、最内ループの外側でイタレーション情報をフラグとして MT にバッファリングさせる。これにより、MT の通信にかかる実行コストを大幅に減少させることができる。HT では、MT からイタレーション情報を入手するまでに、先行して Assertion を実行させる。そして、情報が入手したときに、投機的な実行を停止し、投機的に実行した Assertion とイタレーション情報を照らし合わせる。

以上のように、Assertion を投機的に実行させることで、HT の最適化を図ることができる。しかし、制御フロー情報と Assertion の実行を入れ替えたため、エラーが発覚した場合に、入れ替える前のどの Assertion で最初にエラーが検出されるかがわからなくなる。そこで本手法では、順序を入れ替える前のコードを保証コードとして保存しておく。そして、Assertion が偽と判定した場合に、先行節にある Assertion を保証コードから見つけ出し、評価済みか否かを確認する。評価がされていない Assrtion については、MT から情報を入手し、必ず評価を行うようにする。このように、Assertion がエラーを検出したときに、エラーの位置をより正確に特定することができる。

4 実験

本章では、以上で述べたアルゴリズムを境界チェック, Null ポインターチェック, 0 除算チェックを施したサンプルプログラム（ソート）とベンチマークプログラムに適用した結果を示し、本手法の有効性を示す。

4.1 実験環境

実験環境は表 1 に示す。実行時間の測定には clock() 関数を用いて計測を行った。実験では図 2 に示すように、各種依存関係の緩和を行ったもの、Assertion の割り当てをしたもの、Assertion の投機的実行を行ったもの 3 グループに分け、MT ですべ

表 1: 実行環境

CPU	Xeon 2.4GHz
Memory	512MB
OS	Windows XP Professional
Compiler	VisualC++6.0
Compiler Option	/Zi /Ox
Link Library	winmm.lib

ての Assertion を実行したものと比較した。Assertion 割り当て判定に必要な演算命令にかかる実行コストは IA-32 命令のレイテンシとスループット [4] を用いた。対象としたプログラムは、6 種類のソートプログラム、Spec2000 の地震波伝播のシミュレーションを行う Equake、圧縮アルゴリズムの Bzip2 の 2 種類のベンチマークである。

表 2: 分類表

分類	意味
A	各種依存関係の緩和
B	Assertion の割り当て
C	Assertion の投機的実行

4.2 実験結果

表 3: 遅延削減率

プログラム	本手法		
	A	B	C
183.Equake	32.2 %	65.6 %	65.6 %
256.Bzip2	-270 %	-70.8 %	70.2 %
bubble	-210 %	21.4 %	24.9 %
insertion	64.0 %	1.11 %	41.0 %
merge	-281 %	-22.3 %	48.6 %
quick	-151 %	2.61 %	39.9 %
shell	44.1 %	44.1 %	44.1 %
selection	78.5 %	78.5 %	78.5 %

表 3 で示すように、Assertion をすべて実行した場合は平均 103%の遅延が増えている。この原因は、スレッド間の依存関係を減少させることができない箇所で、MT が HT に対してすべて通信を行うためである。また、HT が多くの Assertion を実行するため、MT よりも実行時間がかかってしまっている。Assertion を選択して実行する場合は平均 15.1%の遅延を削減することができた。通信コストが大幅に削減され、HT の実行時間も短縮されたため速度向上が得られたと考えられる。

Assertion の投機的実行を行った場合には平均 51.6%の遅延を削減できた。これは、Assertion の割り当てに加え、通信量をさらに削減することができた。また、HT の最適化がより効率よくかけられ、HT の実行時がんが短縮され、MT に CPU リソースが十分に分配されたためだと考えられる。

5 終わりに

本研究では、HT の生成時に、MT/HT のコードに張られる真依存関係・制御依存関係を共通先行節を見つけることにより減少させ、MT のデータ通信にかかるオーバーヘッドを削減した。そして、Assertion 実行にかかるコストとデータ通信にかかるオーバーヘッドを比較し、MT/HT への割り当てを行うことで MT にかかる Assertion 実行コストの最適化を図った。そして、スレッド間のデータ通信を循環バッファを使うことにより、出力依存関係を緩和し、スレッド間で発生する同期による遅延を最小限にすることができた。また、補助スレッドで Assertion の投機的実行を行うことにより、HT の待機時間を縮小し効率よく最適化をかけることができるようになった。結果として、サンプルプログラムで 46%, ベンチマークプログラムで 68%の遅延を隠蔽することができた。

参考文献

[1] Tullsen, D.M. et al:"Simultaneous Multithreading:Maximizing On-Chip Parallelism", In 22nd Intl. Symposium on Computer Architecture, June 1995.

[2] H. Patil, C. N. Fischer:"Efficient run-time monitoring using shadow processing", Proc. of 2nd Intl. Workshop on AADEBUG, 1995.

[3] Dongkeun Kim, Donald Yeung:"Design and Evaluation of Compiler Algorithms for Pre-Execution", In Proc. of 9th Intl. Conference on ASPLOS, pp62-73, 2002

[4] IA-32 命令のレイテンシとスループット
http://www.intel.co.jp/jp/developer/download/