修士論文

題目

# 組込みプロセッサ向け小規模トランザクショナルメモリの実装手法に関する研究

指導教員

近藤 利夫

2017年

三重大学大学院 工学研究科 情報工学専攻
コンピュータ・アーキテクチャ研究室

櫻田 賢大(415M508)

# 内容梗概

現在，共有メモリ型マルチコアプロセッサが広く普及しており，並列処理に利用されている．また，一般に並列プログラムの実行順序が非決定的であるため，共有メモリは並行する複数プロセスの実行順序に依存しないデータの一貫性を保持する必要がある．多くのプロセッサでは，その方法として，共有資源をアクセス権で制御するロック手法（以下，単に「ロック」と呼ぶ）が採用されている．しかし，ロックではオーバヘッドが大きく，細粒度ロックの場合，プログラムの複雑化を招く．また，粗粒度ロックの場合，処理に無関係な共有変数に対する操作も逐次実行となるため性能低下の問題がある．そこで，並列に処理を行うために投機的なメモリアクセスを許すトランザクショナルメモリ (Transactional Memory:TM) と呼ばれる手法が提案されている．TM とは，共有資源内のデータに対する読込み，書込みといった一連の処理をトランザクションと定義し，処理する手法である．複数のトランザクションでアクセス先の衝突がなければ並列的に処理を継続するが，衝突がある場合は，トランザクションを中断し，再実行する．特にハードウェアにより実現される TM は，いくつかの商用プロセッサにおいても採用され，ハードウェア TM と呼ばれる．しかし，それらのプロセッサでは，トランザクション処理を担う回路の規模が大きい．そのため，組込み分野での利用が困難である．そこで，扱うトランザクションサイズを限定し，TM をコア内で留め，組込みプロセッサ等の小規模なシステム向けのハードウェア TM の実装手法を提案する．提案手法では，現在スマートフォンなどで利用されている組込み向け ARM プロセッサのようなアウトオブオーダー型スーパースカラコアへの実装を想定している．そのため，実装 TM は，容易な実装及び，回路規模を抑える必要がある．そこで，既存回路資源を活用し対処する．既存資源とは例外処理機構であり，トランザクション処理のために流用される．また，衝突時の復元時間の見積もりがプロセッサの例外回復時間に依存するため，容易となる．さらに，複数のトランザクション処理間の衝突検知手法として，アクセスアドレスをビット列に変換し，コア間通信を行う手法を提案する．このような手法を用いて，衝突検出機構によるコアの回路規模，消費電力は，共に 0.01%程度の増加に抑えられた．したがって，組込み分野での利用に現実的である．また，性能面では，ロックに比べ最大 57%の実行サイクル数となることを確認した．

# Abstract

Shared memory multicore processors are widely used in parallel processing. Also, since the execution order of parallel programs is generally non-deterministic, it is necessary for the shared memory to maintain consistency of data that does not depend on the execution order. In many processors, "lock" is adopted as a method. However, overhead is large in lock, and in fine grain lock, the program is complicated. Also, in coarse grain lock, operations on shared variables unrelated to processing are also sequentially executed, so there is a problem of performance degradation. Therefore, transactional memory (TM) has been proposed that allows speculative memory access in order to perform parallel processing. TM performs processing on a transaction basis. A transaction refers to a series of processing such as reading and writing to data in a shared resource. If there are no collisions of access destinations in multiple transactions, processing continues in parallel, but if collision is occured, the transaction is suspended and reexecuted. In particular, the TM realized by hardware is adopted in some commercial processors and is called hardware TM. However, in those processors, the scale of the circuit for transaction processing is large. Therefore, it is difficult to use in the embedded systems. So, we propose a hardware TM implementation method for small scale systems. The method assumes implementation on an out-of-order superscalar such as embedded processor being used in smartphones. Therefore, the TM needs to be easily implemented and reduce the circuit scale. So, we utilize existing circuit resources. Existing resources are exception handling units and are diverted for transaction processing. Also, it is easy to estimate the recovery time at the time of collision because it depends on the exception recovery time of the processor. Furthermore, as a collision detection method, we propose a method to convert access addresses into bit strings and perform core-to-core communication. By using this method, both the circuit scale and power consumption of the core by the collision detection mechanism were suppressed to about 0.01% increase. Therefore, it is practical for use in the embedded systems. In terms of performance, we confirmed that it is up to 57% of execution cycles compared with lock.

三重大学大学院　工学研究科

# 目 次

# 図 目 次

# 表 目 次

# 1　はじめに

現在，高性能化のために並列処理の必要性が高まっており，その実行環境として共有メモリ型マルチコアプロセッサが広く研究されている．一般に並列処理プログラムの実行順序は非決定的であるため，共有メモリは並行する複数のプロセスが実行順序に依存せず，データの一貫性を保持するような排他制御が必要である．そのため，現在の多くのプロセッサの排他制御には共有メモリへのアクセス権を用いたロックが採用されている．しかし，ロックではデッドロックの問題や並列処理のオーバヘッドが大きい問題がある．そこで，ロックに変わる手法としてトランザクショナルメモリ (Transactinal Memory)[1] が提案されている．また，TM は実装手法によりソフトウェアトランザクショナルメモリ (Software TM:STM) とハードウェアトランザクショナルメモリ (Hardware TM:HTM) に区別される．前者はハードウェアへの拡張が不要であるが，ソフトウェアによる処理であることからオーバヘッドが大きい．一方，後者はハードウェアにより処理を規定するために STM に比べオーバヘッドが抑えられ，速度性能が高くなると一般に言われている．特に，HTM は商用となっているものを含み多数の研究 [2, 3, 4, 5, 6, 7, 8, 9] がなされている．

しかし，それらの HTM ではトランザクション処理を汎用的に利用可能とするため，プロセッサコアだけでなく，キャッシュメモリへの拡張が必要であり，ハードウェアの大規模化・複雑化，それに伴う設計の長期化という問題がある．そこで，扱うトランザクションサイズに制限を掛け，コア内部の既存資源を利用することでトランザクショナルメモリをコア内で留める組込み用プロセッサ等の小規模なシステム向けの HTM 実装手法を提案する．以降，本稿は次のように構成する．まず，次章でトランザクショナルメモリの概要について，第 3 章でトランザクショナルメモリの分類について，第 4 章ではハードウェアトランザクショナルメモリに関する関連研究について議論する．第 5 章では提案手法について詳細に説明し，第 6 章で評価，第 7 章で結論を述べる．

# 2　トランザクショナルメモリの概要

トランザクショナルメモリの分類は，1)，競合検出，2) データの管理，3) 実装手法により分類できる．本章では，まずトランザクショナルメモリの処理について概括し，その後，各分類について述べる．

ロックに代わる排他制御手法である TM は，データベース上で行われるトランザクション処理をメモリアクセスに適用した手法であり，クリティカルセクションを含む一連の命令列で定義されるトランザクションという単位で投機的に実行する．この投機的な実行においてプログラムの実行結果の整合性を保証する必要があるため，TM はトランザクション内のメモリアクセスを監視する．そして，監視している複数のトランザクション間で同一アドレスへのメモリアクセスを確認すると，整合性を保つため競合と検出する．検出後，共有資源の一貫性を保証するため，トランザクションの取り消し (Abort) を行う．Abort した際は再実行を行うため，トランザクション内で扱うデータをトランザクションの開始時点の状態に復元 (Rollback) する．また，競合が発生せず，トランザクションの実行が完了した場合にはトランザクション内でのデータの変更をメモリ上に反映 (Commit) する．以上の動作により，トランザクショナルメモリは競合が検出されない限りトランザクションと定義された箇所を投機的に並列実行が可能となり，一般に従来のロックに比べ処理性能が向上する．

トランザクション処理の例を図 2.1 を用いて説明する．今，二つのコアのそれぞれでスレッドが実行されていると仮定する．それぞれのスレッドはトランザクション Tx.A，Tx.B を実行する．時刻 t0，t2 において，互いのトランザクションがアドレス a からロードを行う．そして，時刻 t3 では，Tx.A がアドレス a に対してストアを試みるが，Tx.B によりアクセスがあったことを確認し，トランザクションアボートが起きる．その後，時刻 t4 にて競合対象であった Tx.A が Abort したため，Tx.B がアドレス a への処理を安全にできるので Commit される．その後，Tx.A は，再実行が行われる．また，Abort するトランザクションの選択及び再実行するタイミングは，スケジューリング手法により様々な方法 [14, 15] がある．

# 3　トランザクショナルメモリの分類

本章では, 前章で述べた分類について詳しく説明する．これらの分類は, 文献 [6] に基づいており，トランザクショナルメモリの実現手法が確立されていないことに起因する．各分類中で，適宜，提案手法の分類について述べ，本研究の位置づけを示す．

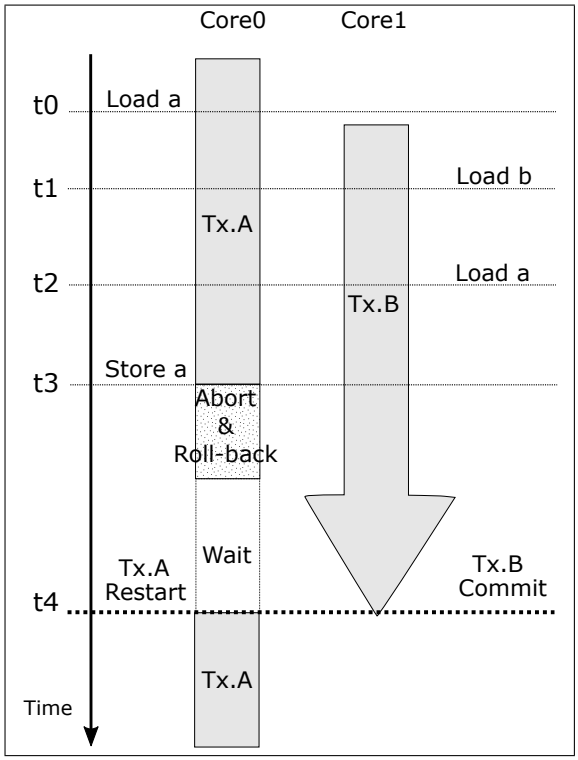


図 2.1: トランザクション処理

## 3.1 競合検出方法による分類

文献 [2, 3, 4, 5] で用いられている HTM での競合検出について述べる．競合検出には，キャッシュメモリの各ラインに設けられる Read/Write ビットを用いる．トランザクション内のメモリアクセスが発生すると，アクセスされたキャッシュラインの Read/Write ビットがセットされ，コミットまたはアボート時にクリアする．HTM において，競合検出は実行するタイミングにより 2 つに分けられる．

**Eager Conflict Detection**

トランザクション内でメモリアクセスが発生した時点で，そのアクセスに関する競合の発生を確認する．

**Lazy Conflict Detection**

トランザクションのコミット直前で，トランザクション内のすべてのアクセスに対して, 競合の確認を行う．

多くの HTM では前述した Read/Write ビットを利用した Eager 方式が採用されている．これはトランザクション内で競合が発生し，検出されるまでの時間が Lazy 方式に比べ短く，実行効率の面で優位なためである．また，トランザクション内でのアクセスアドレスを保持する領域が不要となり，実装コストが抑えられると考えられるためである．したがって，本稿における HTM においても Eager 方式を想定している．

## 3.2 データの管理方法による分類

トランザクションの投機的実行では，アボートにより実行結果が破棄される可能性があるため，トランザクション内で更新した値と更新前の値を併存させる必要がある．そのため，副次的メモリ領域が追加で必要となる．一般的なキャッシュメモリを利用した手法では，一部を副次的メモリ領域として活用する．その利用手法により，以下の分類がある．

**Eager Version Management**
: トランザクション処理開始前の状態を副次的メモリ領域に保持し，トランザクション処理中は従来通りの動作をとる．そのため，Abort 時にはメモリ領域の情報を移す処理が必要となる．

**Lazy Version Management**
: トランザクション処理中は副次的メモリ領域を利用し，処理を行う．トランザクション処理前の状態は元の領域に保持されたままである．そのため，Commit 時，メモリ領域の情報を移す処理が必要となる．

実際には，多くの HTM で Eager 方式が採用されている．これは，高頻度の Abort は性能劣化に繋がることから，Abort の発生に関しては抑制することを前提として考えられたためである．本研究においては，対象プロセッサのアーキテクチャを利用するため，このような専用領域は不要であり，分類上いづれにも属さない特徴がある．

## 3.3 実装手法による分類

トランザクショナルメモリは実装手法により，ソフトウェアトランザクショナルメモリ [11]，ハードウェアトランザクショナルメモリの 2 種類に区分される．

**ソフトウェアトランザクショナルメモリ**

ソフトウェアトランザクショナルメモリ（STM）は TM のアイデアをソフトウェアのみで行う方法である．STM では TM 用の特殊なハードウェア機構を用いることなくトランザクショナルメモリを実現でき，また，ハードウェア資源の制限を事実上受けないため，ユーザーはトランザクションの開始・終了を任意に指定できる．一方で，命令セットは TM 専用の命令をサポートせず，いくつかの命令を組み合わせることで TM 機構を実現する．そのため，生じるオーバーヘッドが大きくなるという問題点がある．主に API として Clojure[12] などの様々な高級言語で利用可能である．

**ハードウェアトランザクショナルメモリ**

Intel 社 [9],IBM 社 [10] により汎用コンピュータやサーバ向けで実現されている．ハードウェアによる制限があるため，プログラムの記述効率は低下する．しかし，メモリアクセス頻度がソフトウェアトランザクショナルメモリに比べ抑えられるため，性能向上が見込まれる．

# 4 関連研究

現在，キャッシュメモリを利用した，汎用性を目指したトランザクショナルメモリの研究が主流である．文献 [7] では，ハードウェアトランザクショナルメモリの制約に対する解決のため，キャッシュメモリ以降のストレージまでもトランザクション処理に利用できるようトランザクションに関わる情報のデータ構造を規定し，実質的に無制限のトランザクショナルメモリの実現をを目指している．しかし，前提ハードウェアの規模が大きいため，組込み用途という点では，消費電力，回路規模共に相応しくない．文献 [8] では，メモリ管理を容易にするために仮想メモリが採用された経緯に倣い, トランザクショナルメモリの仮想化を行い，ハードウェアでの実装や制約に対する問題への解決を試みている．しかし，この試みの前提にあるハードウェアトランザクショナルメモリもまたキャッシュメモリを用いた汎用性のあるトランザクショナルメモリの実装を対象としている．文献 [13] は組込み向けのトランザクショナルメモリに対するフレームワークについて議論しているが，キャッシュメモリを利用することを想定しており，設計コストの観点からの議論は行われていないが，キャッシュメモリの状態保証のためのコヒーレンスプロトコルに手を

加える必要があることが記述されている．また，組込み用途に適応したトランザクション専用キャッシュメモリを用いた場合，消費電力が増加したという記述があり，組込み用途でのトランザクショナルメモリにはキャッシュメモリの利用が効果的でないことが伺える．したがって，キャッシュメモリを用いない組込み用途を目的としたハードウェアトランザクショナルメモリに対する本研究の手法は有用であると考えられる．

# 5　提案手法

本節では，実装が容易な小規模 HTM の仕様及び，実装手法について述べる．提案手法では一般的なキャッシュメモリの拡張による実装ではなく，コア内部のストアバッファへの拡張，コア間の専用通信により実装を行う．また，スーパスカラプロセッサに標準的に備わっている例外回復機構を利用することを想定している．そのため，例外処理機構の流用方法に先だってスーパスカラについて説明する．

## 5.1　スーパスカラプロセッサの概要

本節では，提案手法で利用するプロセッサアーキテクチャについて述べる．図 5.2 及び図 5.3 は，それぞれ標準的なパイプライン構造と本研究に関連のある部分のスーパースカラプロセッサの構造を示している．まず，図 5.2 及び図 5.3 を用いてスーパースカラプロセッサの基本動作について説明する．

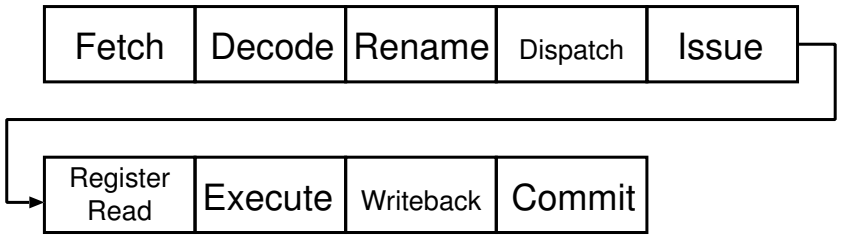


図 5.2: 標準的なパイプライン構造

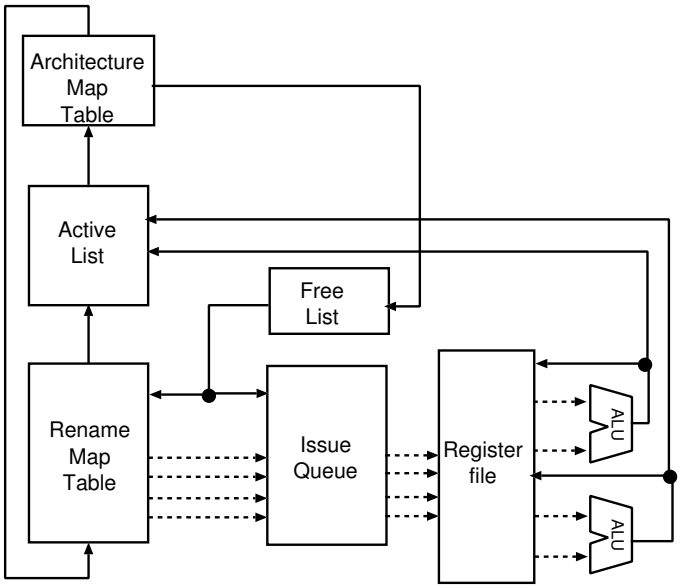


図 5.3: スーパースカラプロセッサの構造

### 5.1.1 パイプライン処理

一般に高性能なプロセッサは，命令メモリからの読み出し，命令の解析，実行などの処理を同時に 1 サイクル内で行うことはせず，各処理要素を直列に連結し，ある要素の出力が次の要素の入力となるようにして，並行に処理させている．よって，1 サイクル以内に処理すべき要素が減ることで動作周波数の向上につながり，またスループットの向上にもつながる．これをパイプライン処理と呼ぶ．以下に各パイプライン要素の動作の説明をする．

**Fetch**

図 5.2 に示す Fetch ステージでは，Program Counter（PC）の値に従って命令キャッシュにアクセスし，命令を読み出す．その後，PC を更新する．

**Decode**

図 5.2 に示す Decode ステージでは，Fetch ステージで読み出した命令を解析し，命令の種類やオペランドといった情報を得る．

**Rename**

図 5.2 に示す Rename ステージでは，デコードされた命令のソースレ

ジスタ番号が図 5.3 に示す Rename Map Table（RMT）を参照し物理レジスタ番号に変換される．同時にその命令のデスティネーションレジスタとして新しい物理レジスタ番号が図 5.3 に示す FreeList から割り当てられる．フリーリストとは未使用の物理レジスタを管理している FIFO である．Free List から割り当てられた物理レジスタ番号はデスティネーションレジスタ番号に対応する RMT に登録される．この一連の操作をレジスタ・リネーミングと呼び，レジスタ・リネーミングによって命令間の偽の依存が取り除かれる．

**Dispatch**

図 5.2 に示す Dispatch ステージでは，命令のコミットを命令順通りに行うためリネームされた命令が図 5.3 に示す Active List に命令順通り登録される．このとき，命令のデスティネーションレジスタ番号と，現在割り当てられている物理レジスタ番号が Active List に登録される．この情報は，例外発生時にアーキテクチャステートを復元するために使われる．また，全ての命令は図 5.3 に示す Issue Queue に書き込まれる．

**Register Read**

図 5.2 に示す Register Read ステージでは，発行された命令のソースオペランドに割り当てられたレジスタを読み出す．

**Execute**

図 5.2 に示す Execute ステージでは，図 5.3 に示す ALU などの演算器を用いて演算を行う．

**Writeback**

図 5.2 に示す WriteBack ステージでは，演算結果をデスティネーションに割り当てられたレジスタに書き込み，Active List に命令が完了したことを知らせる．

**Instruction Commit**

図 5.2 に示す Commit ステージでは，完了した命令が Active List から読み出され，その命令のデスティネーションレジスタ番号に対応した図 5.3 に示す Architecture Map Table（AMT）のエントリに割り当てられた新しい物理レジスタ番号を登録し，該当エントリの物理レジスタを解放し Free List に戻す．

### 5.1.2　スーパースカラ

プロセッサにはスカラ型とベクトル型がある．スカラ型とは原理的に1つの命令で1つのデータを操作する特徴があり，ベクトル型とは1つの命令で複数個のデータを扱える特徴がある．スーパースカラでは，スカラ型の流れを継承しつつ，同時に複数命令を実行可能とすることで，結果的に複数個のデータを同時に扱うことが可能である．そのため，扱うデータの依存関係の影響を受け，次節で述べる例外処理機構が必要となる．また，本稿における例外処理とは，投機的実行の失敗に対する回復処理を指す．

### 5.1.3　例外処理機構

まず，例外処理について図5.4を用いて説明する．例では，2並列のスーパースカラでアセンブリ命令I0∼I3の実行を行う．Cycle1では，I0，I1の命令がフェッチされる．続くCycle2では，本来I1の分岐命令の結果が出るまで，実行すべき次命令が確定できないため，命令フェッチを待機すべきである．しかし，それでは，I1の結果が得られるまで資源利用に無駄が発生する．そのため，分岐処理の結果を予測し，命令を実行する．今回の例では，分岐先に飛ぶと予測して実行するとする．よって，Cycle2以降，再度I0，I1をフェッチし続ける．そして，Cycle3にて，$1の値が計算され，Cycle4にて分岐先が確定する．この時，本来実行すべきであった命令は，I2，I3であったことが確認され，予測が誤っていたことが確認される．したがって，誤って実行したパイプラインを無かったこととし，その過程でのレジスタ等の計算状態を復元する．そして，正しい分岐先の命令を実行しなおす．このような処理が例外処理である．提案手法における例外処理では，トランザクション開始命令を前述の例の分岐命令と見なすことができる．また，例外の検出，計算状態の復元のため，提案手法での例外処理機構は図5.3のActive List，Free Listにより実現される．Active Listはレジスタの情報を含む実行中の命令群を保存し，分岐予測ミスなどの例外が発生した際に，データの状態とPCを例外発生前の状態に復元するために利用される．また，Free Listは命令を並列に実行するためにリネームされるレジスタのリストであり，値の変更の遷移情報が確認できる．本研究は，以上のアーキテクチャ内部の既存資源を流用し，トランザクション処理を実現することで設計コストを抑える手法である．

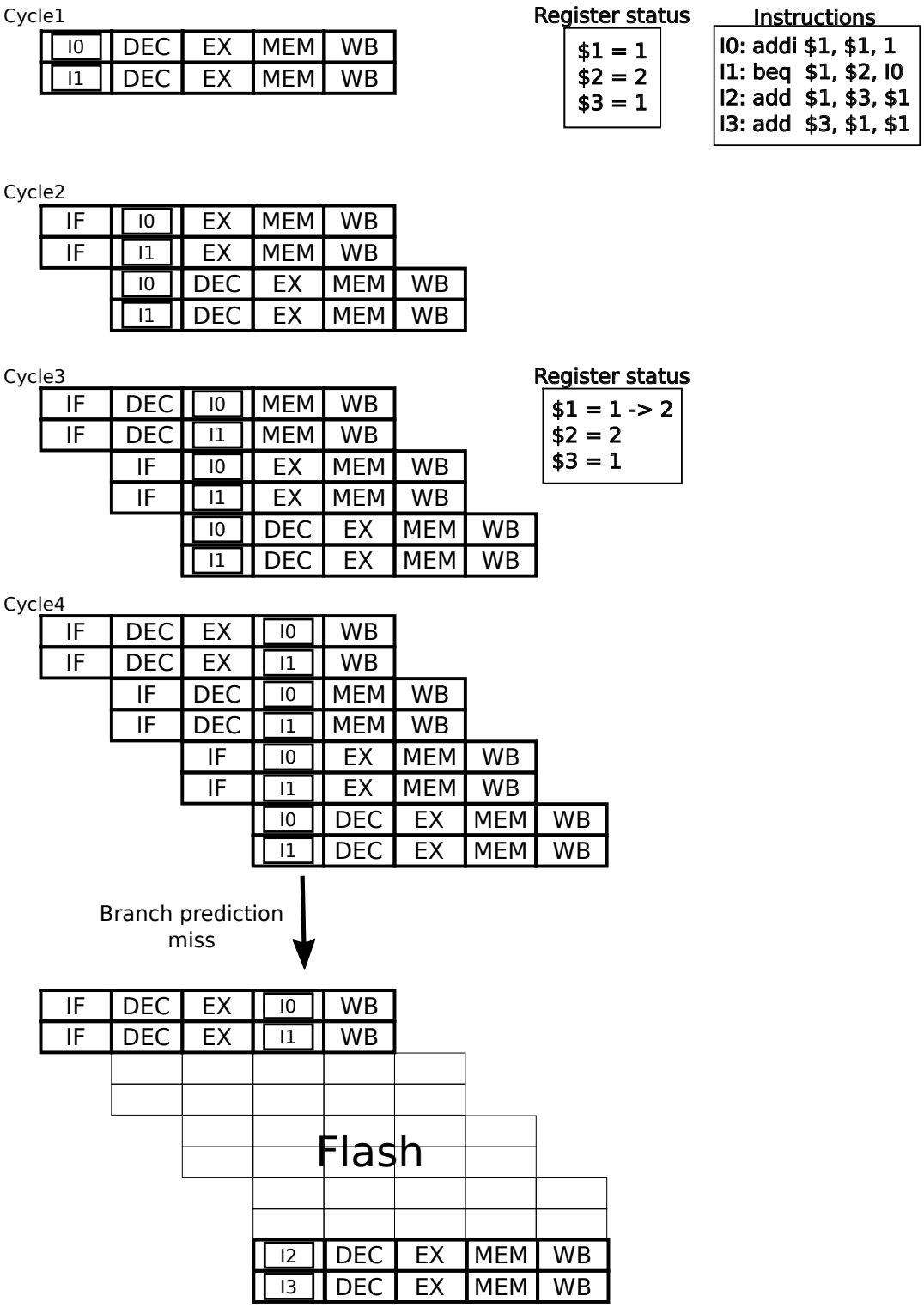
Cycle1
I0 DEC EX MEM WB
I1 DEC EX MEM WB
Register status
$1 = 1
$2 = 2
$3 = 1
Instructions
I0: addi $1, $1, 1
I1: beq $1, $2, I0
I2: add $1, $3, $1
I3: add $3, $1, $1
Cycle2
IF I0 EX MEM WB
IF I1 EX MEM WB
I0 DEC EX MEM WB
I1 DEC EX MEM WB
Cycle3
Register status
$1 = 1 -> 2
$2 = 2
$3 = 1
IF DEC I0 MEM WB
IF DEC I1 MEM WB
IF I0 EX MEM WB
IF I1 EX MEM WB
I0 DEC EX MEM WB
I1 DEC EX MEM WB
Cycle4
IF DEC EX I0 WB
IF DEC EX I1 WB
IF DEC I0 MEM WB
IF DEC I1 MEM WB
IF I0 EX MEM WB
IF I1 EX MEM WB
I0 DEC EX MEM WB
I1 DEC EX MEM WB
Branch prediction miss
IF DEC EX I0 WB
IF DEC EX I1 WB
Flash
I2 DEC EX MEM WB
I3 DEC EX MEM WB

図 5.4: 例外処理の例

## 5.2 追加命令と仕様

一般的な TM はトランザクションの開始・終了を規定するため，それぞれを指す 2 つの命令を使用する．

**XBEGIN**

トランザクションの開始命令である．

1. ロールバック後の再実行のため，プログラムカウンタ (PC)，各レジスタを待避する．これらの処理は，アーキテクチャの流用により行われる．
2. トランザクションの開始を示す信号を LSU, ストアバッファに通知する．

**XEND**

トランザクションの終了命令である．

1. 競合が発生しなかった場合に実行される．
2. トランザクションの実行結果であるストアバッファ内のデータをメモリに書き出す．この時，他のコアからのアクセスは禁止にし，優先的にメモリへの書き出し処理を行う．

また，競合発生時は，PC を XBEGIN の PC に再設定し，例外処理機構によるレジスタの復元，ストアバッファ内の初期化を行う．

## 5.3 競合検出手法

競合検出はアクセスアドレスに対し，ハッシュ関数を用いることで行う．図 5.5 では，あるトランザクション中におけるアクセスアドレスが競合検出用レジスタ (TxReg) へ格納されるまでの流れを示している．図中の HashFunction でアクセスアドレスに対しハッシュ関数を適用し，結果のハッシュ値をシフト量として続くシフタにより，1 に対するシフトを行う．このような過程により，アクセスアドレス毎に高々１ビットの１が立つようなビット列が生成される．この変換後のビット列の格納には，各アクセスアドレス毎に格納するのではなく，各アドレスの変換後のビット列の OR をとることで，一本のレジスタでアクセスアドレスの管理を行う．また，メモリアクセスが発生する度に TxReg の更新を行えば良いため，

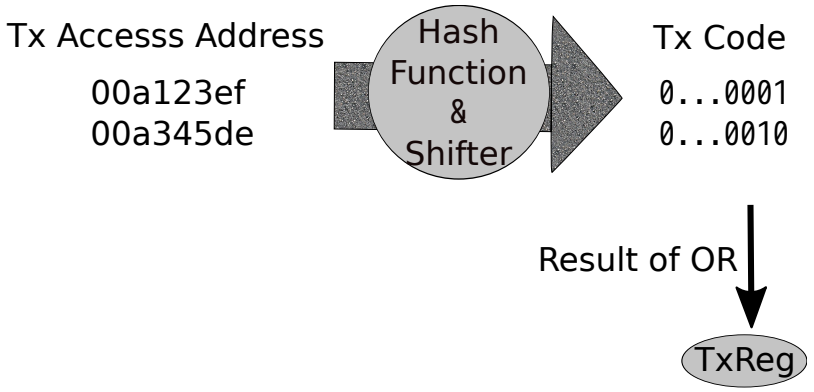


図 5.5: 競合検出手法

図 5.5 の Tx Access Address，Tx Code のような専用の保存領域はハードウェアとしての実態は不要である．さらに，各コアが同じ変換関数を利用し，トランザクション中のアクセスアドレスをビット列として専用レジスタ (TxReg) に格納するため，競合検出時にはレジスタ間の論理演算のみで検出が可能となる．しかし，実現するためには TxReg を各コア間で通信する必要がある．そのため，図 5.6 に示すように Wired-OR を通信線として設ける．このような実装により，あるコアは各他コアの TxReg の OR をとったビット列をレジスタ (OTxReg) で受け取り，自身の TxReg との AND の結果により，競合発生を検出できる．TxReg の通信は，コア間のトランザクションのタイミングによる競合検出の失敗を回避するため，全てのトランザクション中のメモリアクセス発生時に行う．また，本手法ではハッシュ値に度々衝突が起こることが想定される．そのような場合，競合検出が偽陽性を持つ処理となり，動作が保証される．競合検出処理の具体的な例として，Core-A，Core-B の 2 コアでの競合検出過程を以下のプログラムを用いて説明する．また，本例中の TxReg,OTxReg は簡単化のため，5 ビットとする．

| Core-A | Core-B |
|---|---|
| 1: XBEGIN | 1: XBEGIN |
| 2: SW $1, 0x400 | 2: SW $1, 0x100 |
| 3: LW $1, 0x400 | 3: LW $1, 0x400 |
| 4: XEND | 4: XEND |

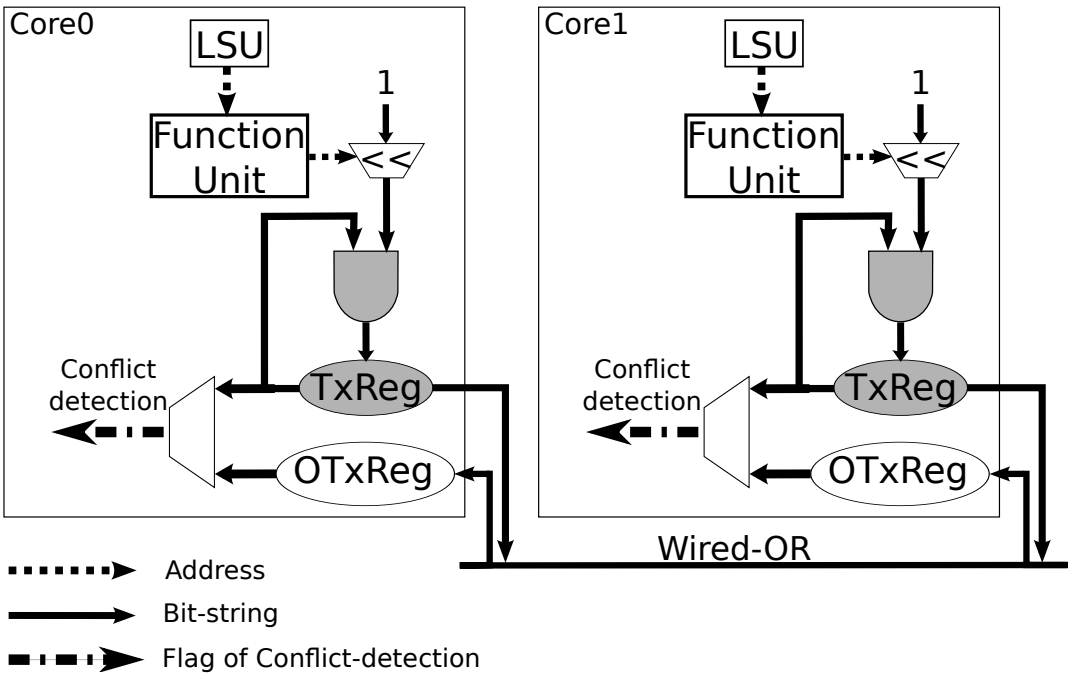


図 5.6: 競合検出の全体像

各コアが 2 行目まで実行している段階では，それぞれのレジスタ TxReg は，図 5.7 に示されるビット列への変換と互いの TxReg の論理演算により，競合が無いことが確認される．しかし，3 行目の実行段階で，図 5.8 の通りとなり，競合が確認される．各ステップでの TxReg，OTxReg の状態を図 5.1 に示す.

表 5.1: 各ステップでの TxReg，OTxReg の状態

| | Core-A | | Core-B | |
|---|---|---|---|---|
| | TxReg | OTxReg | TxReg | OTxReg |
| 1 行目 | 00000 | 00000 | 00000 | 00000 |
| 2 行目 | 01000 | 00010 | 00010 | 01000 |
| 3 行目 | 01000 | 01010 | 01010 | 01000 |
| 4 行目 | 01000 | 01010 | 01010 | 01000 |

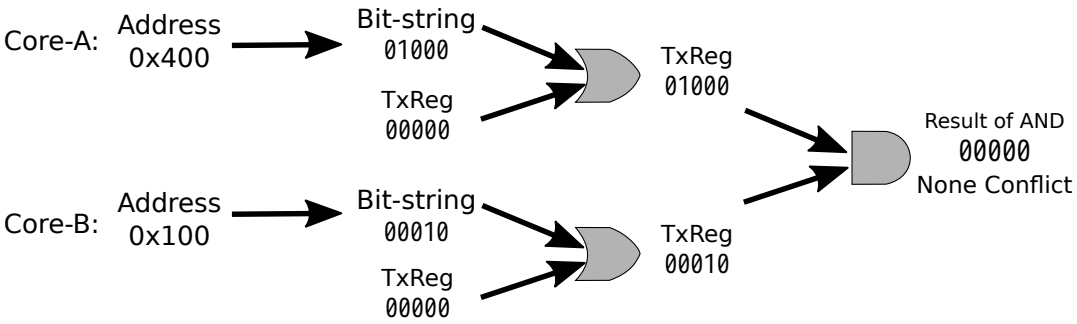


図 5.7: 競合なし

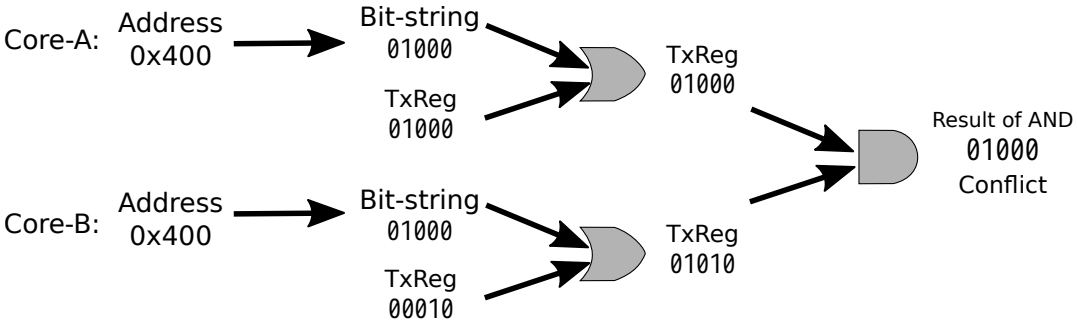


図 5.8: 競合あり

## 5.4 ロールバックと再実行

提案手法ではロールバック処理がストアバッファ内の初期化とレジスタの復元のみで行われるめ，既存の例外回復の仕組みを流用する．ロールバック後は PC が XBEGIN 命令を指し直し，再実行が行われる．提案手法における再実行時のスケジューリングは，再実行が必要なトランザクションを記録しているキューの状態から，対象トランザクションを選択する仕様である．図 5.9 に提案手法を適用した箇所の詳細なブロック図を示す．

StoreBuffer はトランザクション処理に合わせ，動作を変更する必要があるため，トランザクション命令により，以下の状態が追加される．

**Store Stop**

トランザクション処理中の状態．メモリへの書き出しをせず，メモリへの書き込みが発生した場合は，バッファ内に溜め込む．一方で，

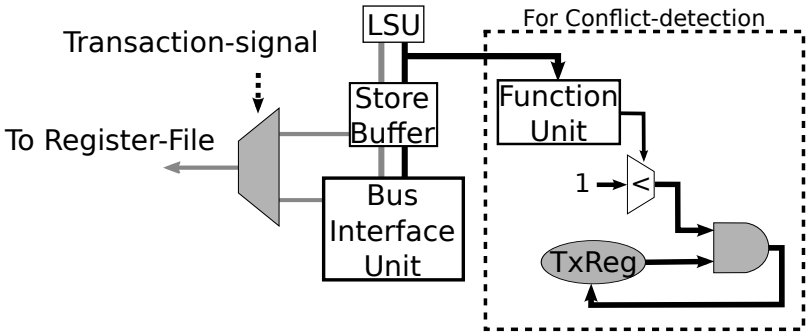


図 5.9: 提案手法適用後のブロック図

読み出しが発生した場合は，StoreBuffer 内にあるデータを優先的に利用し，見つからない場合は，通常のロード命令の処理が行われる．

**Store Commit**

トランザクション処理終了時の状態を指す．StoreBuffer 内のデータをメモリへ書き出す．この時，全ての書き出しが終わるまで，他のコアによるメモリアクセスは禁止にする．

**Buffer Clear**

トランザクションのアボートが発生した際の状態である．ストアバッファをクリアする．

これらの状態遷移は，XBEGIN 命令，XEND 命令により行われる．XBEGIN 命令により，Store Stop となり，XEND 命令実行時の競合の有無により，Store Commit，または，Buffer Clear の状態をとる．また，XBEGIN 命令を条件分岐命令，XEND を先行する条件分岐命令 (XBEGIN) の条件確定命令とみなせるため，投機的命令実行を行うスーパスカラの例外処理機構を活かすことができる．

## 5.5 トランザクション処理に対する制限

前述の基本方針より，提案手法におけるトランザクションには 2 つの制限が発生する．

**トランザクションサイズ**

トランザクション内のアクセスアドレスをストアバッファで管理す

るため，メモリアクセスの頻度が高いトランザクションをサポートできない．そのため，ループ処理や I/O などを除く，比較的短いトランザクションのみサポートしている．また，ロールバック時のレジスタの復帰には例外処理機構を用いているため, トランザクション内の最大命令数は命令ウィンドウのサイズに依存する．

**ネストトランザクションの利用**

ネストトランザクションとは，トランザクション内部でトランザクションを定義する入れ子構造をとるトランザクションである．ネストトランザクションは，ネストの階層管理が必要であり，ハードウェア資源がさらに必要となるため，回路規模及び設計コストの観点から，提案手法ではサポートしていない．

# 6 評価

本章では，ベンチマークプログラムの実行時間，競合検出精度，追加実装となるハードウェアの面積及び電力を元に提案手法の有効性について述べる．

## 6.1 評価環境

HTM の実現に提案実装手法を適用した場合の性能評価環境について述べる．評価には，マルチコアプロセッサ検証向けの機能シミュレータを利用した．また，実行環境は，表 6.2 の通りである．

表 6.2: 実行環境

| | |
|---|---|
| CPU | Core i7-2600 |
| 動作周波数 | 3.4GHz |
| メモリ | 16GB |

## 6.2 性能評価

実行時間及び競合検出精度を基にした性能評価及び評価に用いたプログラムについて述べる．

### 6.2.1　評価用プログラムについて

プログラムは，常に共有資源に対して計算処理をしているスレッドを 1 つ作成し, 他のスレッドはランダムなタイミングで共有資源への計算処理を行う．評価項目として競合発生率, 実行ステップ数, スレッド数, トランザクションサイズが対象である．各評価項目について述べる．競合発生率とは各スレッドのそれぞれで排他制御が発生する確率であり，実行サイクル数は提案手法を動かしたときに実際に発生した命令数である．また，スレッド数は同時に動作しているスレッド数であり，トランザクションサイズはトランザクション間の命令の実行サイクル数である．スレッド数は 2，4，8，16，32 スレッドのそれぞれで実行し, 最大競合発生率は 1 %から 100 %まで 5 %刻みでロックと TM のそれぞれについて評価を行った．トランザクションサイズは (A) トランザクション間の実行サイクル数が最小のもの，(B)) 各スレッドのトランザクション外のサイクル数と同程度のもの，(C)) それの 2 倍のものにより評価する．プログラムの動作は, 例えばスレッド数 2 の場合は 1 つのスレッドは常に共有変数に対して計算処理をし,2 つ目のスレッドはランダムなタイミングで共有変数に対して演算を行う．スレッド数 4 の場合は 3 つのスレッドがそれぞれに別の変数に対してランダムなタイミングで演算を行い,1 つのスレッドが先の 3 つのスレッドで扱っている両方の変数に対して交互に計算処理を行う．スレッド数が 8 以降に関しても同様で，この一連の処理の 10 万回ループ時の実行サイクル数を性能評価の項目として用いた．図 6.10, 図 6.11 は作成したプログラムのフローチャートである．

### 6.2.2　評価結果

図 6.12 から図 6.14 が前項で述べた評価結果を示すグラフである．結果として，ロックより最大 64.4%の実行サイクル数の削減を確認した．また，トランザクションサイズの全てのパターンで，スレッド数 2 の時がロックに比べて実行ステップ数が大幅に改善されている．他のスレッド数の際はスレッド 2 の時ほどの大きな差は見られないが，ロックと同等かそれ以上の性能が確認できる．また，競合発生率においても，全体的に競合発生率が上昇するほどロックに比べ，高速であることが確認できる．さらに，トランザクションサイズとの相関では，3 パターンのいづれもグラフの傾向に大きな変化は見られなかった．以上より，評価対象の HTM は競合発生率が大きくなることで頻繁に発生するトランザクション

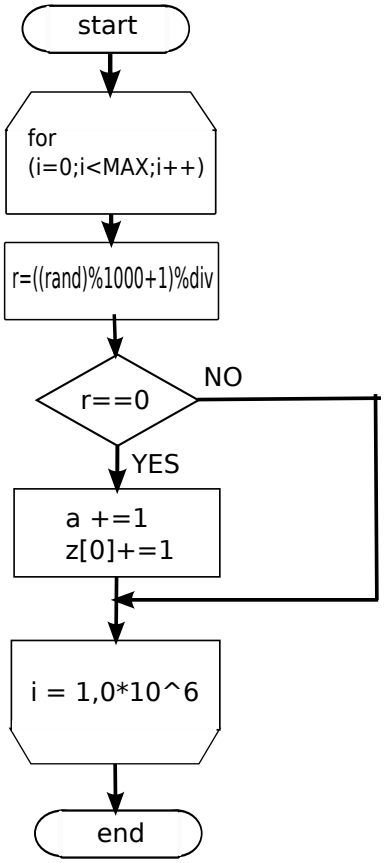


図 6.10: ランダムなタイミングで演算を行うスレッド

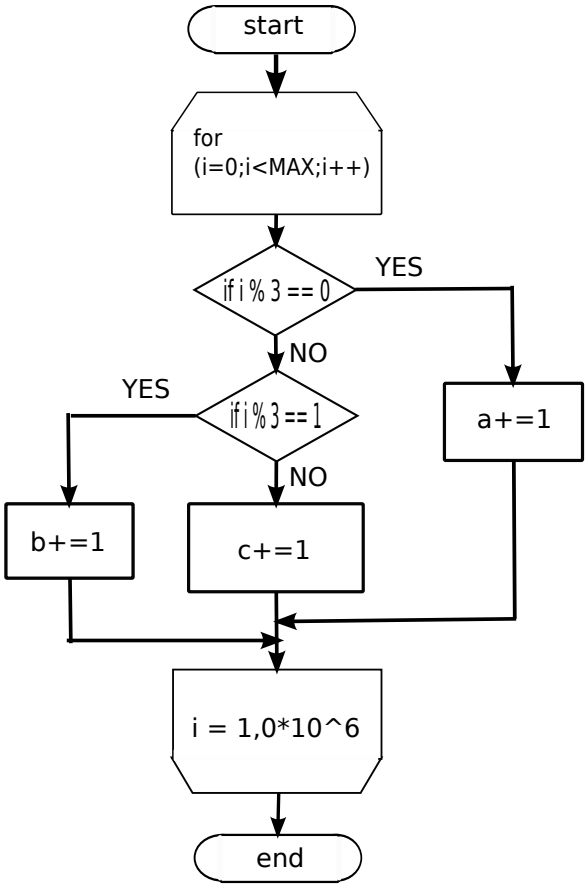


図 6.11: 扱っているすべての変数に順番に演算を行うスレッド

アボートによるオーバーヘッドがロックのオーバーヘッドと比べ同程度かそれ以下であることが確認できる．そのため，トランザクションサイズが小さいものや，ネストトランザクションの存在しないもの，トランザクション間にループ処理が入っていないものならロックと比べ，高い性能が期待できると考えられる．また，トランザクションサイズについては，今回のような高々数十ステップの命令であれば 5.5 節で述べた制限に触れず，問題なく利用可能であることが確認できた．

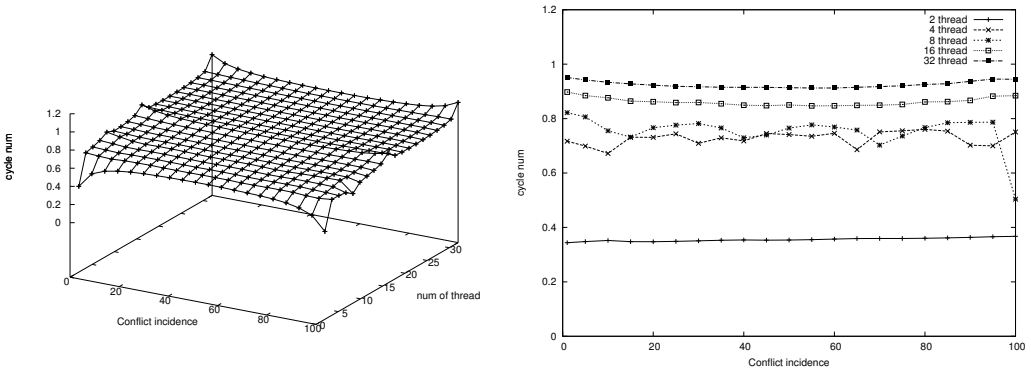


図 6.12: トランザクションサイズ (A)

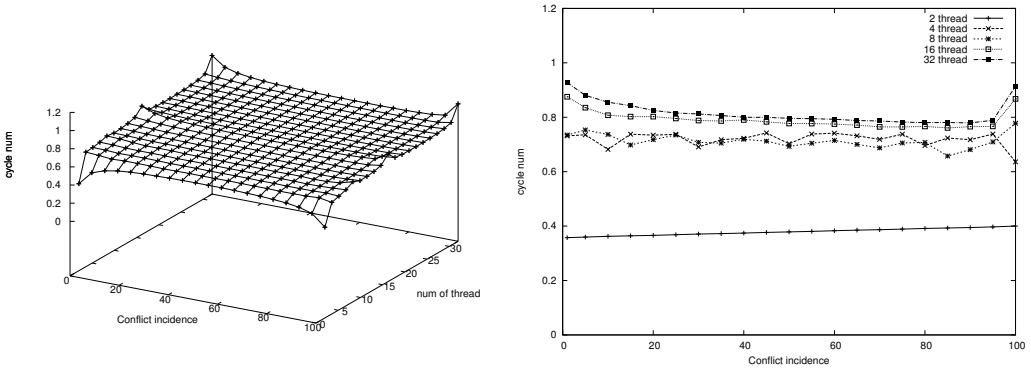


図 6.13: トランザクションサイズ (B)

## 6.3 面積及び電力評価

本節では，提案手法により追加が必要な回路のによる面積と電力の評価について述べる．表 6.3 にプロセッサの構成を示す．また，表 6.4 に示

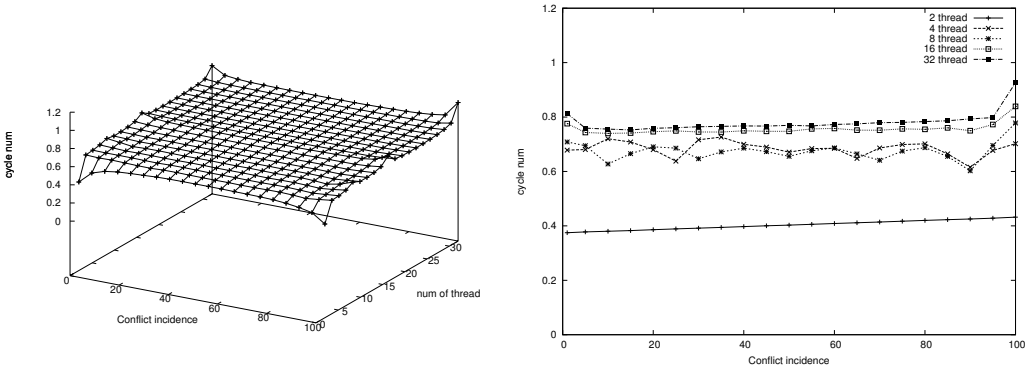


図 6.14: トランザクションサイズ (C)

表 6.3: プロセッサの構成

| | |
|---|---|
| Data path width | 32 bit |
| Fetch, Dispatch, Issue, Commit width | 4 |
| Branch prediction | 16K entry, bimodal |
| BTB | 1024sets |
| Issue Queue size | 32 |
| Register File | 96 entry |
| L1 I-Cache | 32KB, 16B/line, 4way 1cycle latency |
| L1 D-Cache | 32KB, 16B/line, 4way 2cycle latency |

される EDA/CAD ツールにより面積及び電力の評価を行った．評価結果より，0.01%程度の回路規模及び電力増加が確認できる. しかし, いずれもプロセッサコアの内部機構としては無視できる範囲内の増分であるため, 性能向上に対する損失として妥当であると考えられる.

# 7　結論

本稿では，組み込み向けの小規模 HTM の実装を目的として，既存のプロセッサアーキテクチャの設計資源を利用する手法を提案した．本手法により，プロセッサに対し 0.01%程度の面積増加で HTM を実現することが可能である．また, 性能評価より，従来のロックに比べ，35.6%の実行時間を記録し，性能面においても有効であることを確認した．今後の展望として，性能劣化の要因となるトランザクションの競合の誤検出を抑

表 6.4: 使用した EDA/CAD ツール

| | |
|---|---|
| Functional verifiation | Cadence NC-Verilog 12.20 |
| Synthesis | Synopsys Design Compiler 2013.03-SP2 |
| Power estimation | Synopsys PrimeTime PX D-2010.06 |
| Technology | Rohm CMOS 0.18 μ m |
| Standardcell library | Kyoto University standard cell library |

表 6.5: 追加回路とプロセッサコアの評価

| | プロセッサ | 追加回路 |
|---|---|---|
| 面積 ($\mu m^2$) | 7666973.83 | 7973.68 |
| 電力 ($\mu W$) | 9.582e-06 | 1.796e-10 |

えるアクセスアドレス変換関数の改良，及び，プロセッサへ実際に提案手法を実装し，電力面の評価を行うことを検討している．

## 謝辞

本研究を行うにあたり，多数のご指導を頂きました近藤利夫教授，佐々木敬泰助教，並びに深澤研究員に深く感謝いたします．また，コンピュータアーキテクチャ研究室の学生には常に刺激的な議論を頂き，精神的にも支えられました．また，本研究は日本学術振興会の科学研究費補助金，Synopsys 社，Rohm 社，東京大学 VDEC の支援により実施されたことを並びに感謝します．

## 参考文献

[1] M.Herlihy, J. Eliot and B. Moss.: Transactional Memory: Architectural Support for Lock-Free Data Structures. Proc. of ISCA., pp. 289–300,May 1993

[2] L.Hammond,B.D.Carlstrom,V. Wong,M.Chen,C.Kozyrakis,and K.Olukotun.Transactional coherence and consistency:Simplifying

parallel hardware and software.Mico 's TopPicks,IEEE Micro, 24(6),nov/dec 2004.

[3] L.Hammond, V.Wong, M.Chen, B.D.Carlstrom, J.D.Davis, B.Hertzberg, M.K.Prabhu, H.Wijaya, C.Kozyrakis, and K. Oluk otun. Transactional memory coherence and consistency. In Proceedings of the 31st Annual International Symposium on Computer Architecture , page 102. IEEE Computer Society , Jun 2004.

[4] A.McDonald, J.Chung, H.Chafi, C.Cao Minh, B.D.Carlstrom, L.Hammond, C.Kozyrakis, and K.Olukotun. Characterization of tcc on chip-multiprocessors. In Proceedings of the 14th International Confer ence on Parallel Architectures and Compilation Techniques , Sept 2005.

[5] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Oluk otun. The common case transactional behavior of multithreaded programs. In HPCA '06: Proceedings of the 12th International Symposium on High-P erformance Computer Architectur e , Washington, DC, USA, 2006. IEEE Computer Society.

[6] K. E. Moore, J. Bobba, M. J. Mora van, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory . In HPCA '06: Proceedings of the 12th International Symposium on High-P erformance Computer Architectur e , Washington, DC, USA, 2006. IEEE Computer Society.

[7] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In Proc. of the 11th International Symposium on High-Performance Computer Architec-ture, 2005.

[8] R.Rajwar, M.S.LAM, and K.Lai. Virtulizing transactional memory.SIGARCH Comput.Archit.News,33(2):494-505,2005

[9] Intel. Intel 64 and IA-32 Architectures Software Developer 's Manual, June 2013.

[10] Christian Jacobi, Timothy Slegal et. al. Transactional Memory Architecture and implementation for IBM System z,2012 45th Annual IEEE/ACM International Symposium on Microarchitecture

[11] N. Shavit, and D. Touitou. Software Transactional Memory. Distributed Computing,Special Issue, 10(1997),page 99-116.

[12] Stuart Halloway(著), 河合志朗 (訳), プログラミング Clojure, 株式会社オーム社 ,2010.

[13] Cesare Ferria, Samantha Woodb, Tali Moreshetc, R. Iris Bahara, Maurice Herlihyd, Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems, Journal of Parallel and Distributed Computing Volume 70, Issue 10, October 2010, Pages 10421052 Transactional Memory

[14] Keisuke MASHITA, et al.:Yet Another Waiting Mechanism based on Conflict Prediction for Hardware Transactional Memory, CANDAR.2015.57

[15] Yoo, R. M. and Lee,H.-H. S.: Adaptive Transaction Scheduling for Transactional Memory System, Proc.20th Annual Symp. on Parrallelism in Algorithms and Architecture (SPAA'08),pp.169-178(2008).

[16] MIPS Computer Company. The MIPS RISC architecture.