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

2008－ARC－179 (25)
2008／8／7

トランザクショナルメモリのための性能評価手法

中島貴裕[†1] 菅原豊[†1]
入江英嗣[†1] 平木敬[†1]

ハードウェアトランザクショナルメモリの性能評価において、共有カウンタのインクリメントなどの非常に小規模なベンチマークや、共有メモリ並列計算用ベンチマークであるSPLASH-2内の各アプリケーションのロックで囲まれた部分をトランザクションに変えた形でのベンチマークが広く使われている。しかし、これらのベンチマークでは各トランザクション内での実行命令数が少ないため、プログラミングの労力が細粒度ロックを用いた場合と同程度に大きくなってしまい、並列プログラミングをしやすくするというトランザクショナルメモリの特徴を反映していない。本稿では、プログラミングの容易さをトランザクション中の命令数と考え、SPLASH-2のベンチマークの共有データへの排他的アクセス部分をロックからトランザクションに変えた上で、その大きさを変化させた上で実行時間を評価した。このベンチマーク上で、これまでに提案されているいくつかのハードウェアトランザクショナルメモリ実装をシミュレーションにより評価した。

A Benchmark for Coarse Grained Transactional Memory

Takahiro Nakajima,[†1] Yutaka Sugawara,[†1] Hidetsugu Irie[†1]
and Kei Hiraki [†1]

Microbenchmarks such as shared counter and modified SPLASH-2 are widely used to evaluate hardware transactional memory. However, these benchmarks only replace fine grained locks into transactions.As a result, the number of instructions in one transaction is very low, and this means is does not ease programmablity of parallel programs. In this paper, we introduce a new evaluation benchmark which can evaluate coarse grained transaction.First, we replace locks in applications in SPLASH-2 into transactions and vary the size of them. We compared the performance of conflict detection policy using this benchmark.

1. はじめに

近年、半導体技術の進歩により、チップ内に多くのトランジスタを集積できるようになった。その結果、1チップに複数のプロセッサコアを搭載したチップマルチプロセッサ (CMP) が普及している。並列化可能なプログラムに対して、適切にプログラムを分割、並列化した上でCMP上で複数スレッドを同時に走らせることにより、単一スレッドでの実行時を上回る性能を上げることができる。

しかし、細粒度ロックを用いて正しく効率的なマルチスレッドのプログラムを組むことは経験と時間を要する作業である。マルチスレッドプログラミングの難しさを緩和しつつ性能を確保するために、並列プログラミングの1方式としてのトランザクショナルメモリが注目されており、ソフトウェアによる実装や、ハードウェア支援による実現方式が既にいくつか提案されている。

ハードウェア支援トランザクショナルメモリを評価する方式としては、共有カウンタのインクリメントや双方向キューなどのマイクロベンチマークが多く使われている。また、共有メモリ並列計算用のベンチマークであるSPLASH-2のクリティカルセクションの部分をトランザクションに置き換えたものをベンチマークとして広く使用されている。これらを用いることで、トランザクション間のコンフリクトの回数や全体の実行性能などの評価を行うことができる。

しかし、先ほど挙げたベンチマークでは1つのトランザクション内の命令数が少ないので、プログラミングの労力が細粒度ロックを使用した場合と等しくなってしまい、トランザクショナルメモリを採用する利点が薄れてしまう。プログラミングの容易さというトランザクショナルメモリの性質を活かしつつ現実のアプリケーションへの応用を考える場合には、これらの細粒度なトランザクションに加えて、トランザクション

†1 東京大学大学院情報理工学系研究科

内の命令数の多いベンチマークアプリケーションを用いての評価が必要になる。

本稿ではそれを実現するための第一歩として、SPLASH-2 ベンチマークをベースとした上で、トランザクション中の実行命令数の変化が実行時間に与える影響を調べた。

本稿の構成は以下の通りである。2 節では背景について述べ、3 節で提案する性能評価方式を述べる。4 節では提案評価方式の 1 例として、トランザクション間のポリシーやプロセッサ数を変えた上でのシミュレーションによる評価方式を説明し、5 節で結果の考察を行う。6 節で関連研究について述べ、7 節で本稿をまとめる。

2. 背　　景

2.1 ロックの粒度

マルチスレッドによって実現される共有メモリ並列プログラムにおいては、排他処理を行うためにロックが用いられてきた。ロック自体にかかるオーバーヘッドを無視した場合、ロックを用いて効率的な並列アプリケーションプログラムを組むためには、できるだけロック内の命令数を必要最小限に抑えることが重要と考えられている。排他制御をする必要のない部分までロック内に配置をしてしまうと、スレッドが逐次化される頻度が上がるためである。粗粒度なロックを用いた場合には、プログラミングの容易さを得るかわりに多くの場合性能を犠牲にしてしまう。

トランザクショナルメモリを用いた場合にも粒度による影響は受けると予想できる。しかしロックの場合と違い、トランザクション間のコンフリクトが無い限り同時に複数のトランザクションを実行可能である。

2.2 トランザクション粒度とベンチマーク

ハードウェアトランザクショナルメモリを評価するにあたって、共有カウンタのインクリメントや双方向キューの操作などのマイクロベンチマークや、マルチスレッドでの並列アプリケーションの細粒度ロックをトランザクションに変えたベンチマークが広く使われている。これらはトランザクショナルメモリ実装の性能をロックの場合と比べる場合には有用だが、並列プログラミングの負担を減らすというトランザクショナルメモリの利点を評価することができない。

3. 提案する評価方式

本稿では粒度を変えた場合にはロックを用いた場合とは違った挙動を示し、かつそれがトランザクショナルメモリの実装方法に依存すると予想した上で、同一アプリケーションに対してトランザクション内の命令数を変化させた上でシミュレーションを行い、命令数の変化に対する実行時間を評価する。まずはベースとなる SPLASH-2 について説明し、その上でトランザクション内の命令実行数を変化させる方法について述べる。

3.1 SPLASH-2

SPLASH-2[9] は共有メモリ並列アプリケーションのためのベンチマークであり、7 つのアプリケーションからなっている。プログラムの流れとしては、プログラムの起動後に、スレッドもしくはプロセスを指定された数だけ生成し、バリア同期をしながらマルチプロセッサのマシン上で並列で動作する。SPLASH-2 は共有メモリ用のアプリケーションであるため、プログラムの文面上に明示的にデータ通信のコードは現れない。

3.2 SPLASH-2 のトランザクショナルメモリへの変換

SPLASH-2 では共有資源へのアトミックなアクセスのためにはロックによる同期を行う。トランザクショナルメモリへの対応は、LOCK と UNLOCK というマクロで囲われた部分が該当する箇所であり、LOCK をトランザクション開始、UNLOCK をトランザクション終了に置き換えることによって実現できる。SPLASH-2 の並列実行部分は barrier をいくつかはさむ形になっており、LOCK と UNLOCK で囲われた部分をトランザクションにする場合は、barrier を越えることはできない。なお、barrier に関しても、SPLASH-2 では BARRIER というマクロで自由に定義できるようになっている。

3.3 トランザクションの拡大

対象とするアプリケーションとして barnes と fmm を採用した。barnes、fmm を前節の方式でトランザクションに変えた上で実行した場合の動的実行命令数は表 2 の通りであった。なお、この実行は図 1 における L-F の方式で行った。

これに対し、トランザクション内の実行命令数を増やす操作を行った。今回は細粒度のロックを行うプログラミングの難しさの緩和を目標としているため、直感的にわかりやすい機能単位である関数とトランザクションの 1 つの単位とした。具体的には、ロックを含む最も内側の関数全体をトランザクションとして実行するようにプログラムを変更した。

対象とするアプリケーションは barnes と fmm である。barnes は N 体問題に対して Barnes-Hut 法を用いてシミュレートするプログラムであり、では複数のプロセッサがツリーに対しての操作を行い、その際に

ロックが必要になる。load.C の loadtree 関数がそのツリー操作部分であり、ノードのつけかえの際にロックが必要となっている。

fmm も N 体問題に対して、Fast Multipole Method を用いてシミュレートするプログラムである。FMM 内ではセルの操作中にロックが必要になっている。interaction.C の ShiftLocalExp、ComputeMPExp、ShiftMPExp の各関数内でロックを取っている。データセットのサイズとして、barnes、fmmt ともに物体数は 256 とした。トランザクションの単位をこれらの関数にした場合の実行命令数を表 3 に示した。barnes ではトランザクション内の平均実行命令数が 252.8 インストラクションから 1027.1 インストラクションに、fmm では 219.0 から 5797.2 インストラクションに増加している。

4. 評　　価

4.1 CMP 構成

今回のシミュレーションに用いた CMP の構成を述べる。全体で 1 チップの構成にしてあり、プロセッサ数は 4 である。各プロセッサはインオーダのシングルイシューであり、32KB のインストラクション及びデータキャッシュをそれぞれ持つ。各プロセッサは L2 キャッシュを共有している。キャッシュはディレクトリ方式であり、コヒーレントプロトコルは MESI を拡張したものである。

4.2 実装方式

ハードウェアトランザクショナルメモリの実装方式としては、logTM[2]、TCC[3]、VTM[4] などが提案されている。実装方式をトランザクションにおけるバージョンの管理方法とコンフリクト発見のポリシーによって大きく分類することができる[5]。今回は、バージョン管理に関しては、コミット時にメモリへの値の反映を行う Lazy version managiment 方式で評価を行った。L1 キャッシュの他にトランザクション中の値を記憶するバッファを用意しておき、コミット時にそのバッファの値をメモリ上に反映させる。

他の方式としては、トランザクション中の値の書き込みはメモリに直接行い、その際に書き込み先のアドレスのデータを一時バッファに記録しておき、失敗時のロールバックのために使う方式もある。[2]

トランザクション間のコンフリクト発見のポリシーは大きく 2 つに分かれる。

4.2.1 Eager Conflict Detection

Eager Conflict Detection では、トランザクション間のコンフリクトの判定を各メモリアクセス時に行うものであり、VTM[4] などで採用されている方式である。あるプロセッサがトランザクション中にメモリアクセスを行った場合に、他のプロセッサに対してアクセスをしたという通知を行う。他のプロセッサがこれを受信した場合に、そのプロセッサがトランザクション中に参照したアドレスの集合と、書き込んだアドレスの集合に対して照合した上でコンフリクトの判定を行う。コンフリクトが発生した場合の動作として、常に受信側のトランザクションを中止させる方式 (BASE) と、受信側のほうがより古いトランザクションだった場合に送信側をストールさせる方式 (TIMESTAMP) がある。[5] 今回はどちらの方式を選ぶかによって性能に変化が出ると考え、両方の場合に対して評価を行った。

4.2.2 Lazy Conflict Detection

Lazy Conflict Detection では、トランザクション間のコンフリクトの判定はコミット時に行うものであり、TCC[3] などで採用されている方法である。あるプロセッサがトランザクションのコミットする時は、他のプロセッサに対してトランザクション中に一時バッファに書き込んだ値のアドレスを通知し、受信したプロセッサはトランザクション中に参照したアドレスの集合と、書き込んだアドレスの集合に対して照合を行い、もし集合内に通知されたアドレスがあるならば、受信側がトランザクションを中止する。

4.3 評価環境

シミュレーション環境としては Simics シミュレータと[10]GEMS ツールセットを用いた。Simics は命令セットとして SPARCV9 をサポートしたシミュレータであり、OS 機能を含めたファンクショナルシミュレーションを担当する。それに加えて、GEMS ツールセット中の Ruby モジュールを用いてメモリシステムのシミュレーションをする。トランザクショナルメモリ部分のシミュレーションは、GEMS 内の logTM[2] 用シミュレータの一部分を用いた。

今回は Simics 上で Solaris10 を起動し、その中で本稿で提案したベンチマークアプリケーションを実行した。スレッドは POSIX スレッドを利用し、アプリケーション内で `pthread_create` によりスレッドを発行後、`pset_bind` で各プロセッサに割り付け、マイグレーションが起こらないようにしてある。トランザクションの開始、終了は Simics の magic インストラクションという、一種の nop 命令をアプリケーションが呼び出すことでシミュレータに通知している。

命令セットは SPARC-V9 であり、gcc ver 3.4.3 を用い、最適化オプションは-O3 を用いた。メモリシミュレータである Ruby に与えたパラメタに関しては表 1

パラメタ	値
キャッシュラインサイズ	64B
L1 キャッシュ レイテンシ	1cycle
L1 キャッシュ Way 数	4
L1 キャッシュ サイズ合計	32KB
L2 キャッシュ レイテンシ	20cycle
L2 キャッシュ Way 数	8
L2 キャッシュ サイズ合計	8MB
メインメモリレイテンシ	200cycle

表 1 メモリシミュレータのパラメタ設定

ベンチマーク	平均実行命令数	readset	writeset
barnes	252.8	6.58	4.67
fmm	219.0	8.77	4.70

表 2 トランザクション内の平均実行命令数

ベンチマーク	平均実行命令数	readset	writeset
barnes(ext)	1027.1	17.0	7.64
fmm(ext)	5797.2	38.5	12.0

表 3 トランザクションの拡大後の実行命令数

に示した。今回注目するのは並列実行部分であるので、シミュレーション時における実行時間を含めたプロファイルに関しては、スレッド発行後の並列処理部分だけにしており、プログラム開始直後の初期化ルーチンは含まれないようにしてある。

4.4 実行命令数

SPLASH-2 における各種ベンチマークのロック、アンロックで囲まれた部分をトランザクションに変換した場合の平均実行命令数及びトランザクション中にアクセスしたアドレスの集合、は表 3 の通りとなっている。readset はトランザクション中に load アクセスした集合であり、

5. 結　　果

各種実現方式に対して、barnes を用いて評価した結果を図 1 に示した。また、fmm を用いて評価した結果が図 2 である。

図の略語の意味は以下のとおりである。縦軸は E-F(TIME) を基準とした実行性能である。最初の文字はコンフリクト発見ポリシーであり、Eager なものは E と、Lazy なものに対しては L と表記した。次の文字がトランザクション内のインストラクション数の違いであり、細粒度ロックと同じにした場合には F と、か関数単位で拡大した場合には C と表現した。Eager のもので BASE は前節で説明したように、メモリのリクエストが来たら無条件でアボートをする方式であり TIMESTAMP はリクエスト元のトランザクションがより古かったらアボートする方式である。

表 4 は barnes を実行したときのトランザクションのプロファイル、表 5 は fmm を実行したときのトランザクションのプロファイルである。上段 3 つの方式ははロック部分をそのままトランザクションにしたもので、下段 3 つの方式はトランザクションのサイズを関数に拡大したものであり、abort はトランザクションのアボートの回数である。

barnes のロックをトランザクションに変換した場合 (図 1 の左側) には、タイムスタンプを用いない場合の Eager な方式の性能が他の場合より低い値を示している。表 4 これは元々にあるように、トランザクションの中止 (abort) が他の方式に比べて非常に多く起こっているためである。

これに対し、トランザクション内の命令数を関数単位まで増加させた場合には、図 1 の右側にあるように、Eager(BASE) の性能低下がより顕著になっている。それに対し、他の 2 つの方式に関しては性能に対して Eager (BASE) ほどの影響はなく、Eager (TIMESTAMP) が 6.5 %程度の性能低下、Lazy では 1 %程度の性能低下となっている。表 4 にあるように、どの実装方式でもトランザクションの abort 回数は多くなっているが、絶対数がそこまで多くならない場合には全体の実行時間への影響は少ない。

fmm を用いて評価した場合の結果を図 2 に示した。fmm の合では、ロックをトランザクションに変換した場合にはほとんど実行性能に差が出ていない。それに対し、関数単位まで拡大した場合には Eager(BASE) が 38 %程度まで性能が低下している。

fmm の場合では、表 5 にあるように、トランザクションの拡大を行わない場合には Eager (BASE) の方式ではほとんどコンフリクトを起こさないが、トランザクションの拡大を行った場合には非常にコンフリクト (abort) を多く起こした上で性能低下につながっている。この 2 つのベンチマークだけを考えた場合には、トランザクションのコンフリクト発見ポリシーで Lazy を採用するときには、関数単位までのトランザクションの拡大は性能への影響はほとんどないと言える。

6. 関 連 研 究

トランザクショナルメモリの性能評価に関する関連研究を以下に挙げる。STMBench[1] はソフトウェアトランザクショナルメモリ (STM) のためのベンチマークであり、トランザクションのサイズを変えた上でいくつかの STM 実装に関して評価を行っている。

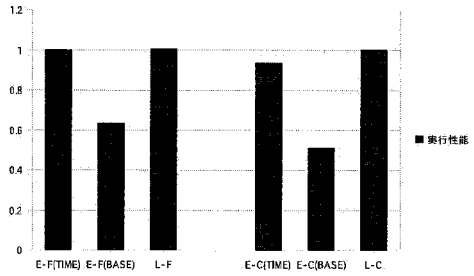

図 1 4 プロセッサでの barnes の実行性能. 縦軸は E-F(TIME) を基準とした実行性能. 左側がトランザクション内の命令数が少ない場合であり、右側がトランザクション内の命令数が多い場合である。

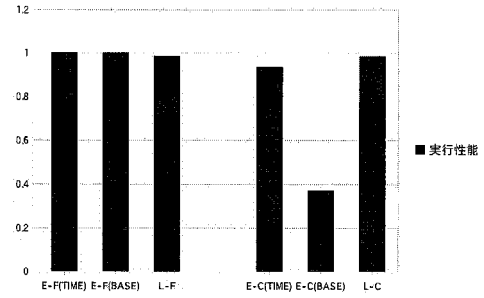

図 2 4 プロセッサでの fmm の実行性能. 縦軸は E-F(TIME) を基準とした実行性能.

実現方式	abort	stall
E-F(TIME)	90	42351
E-F(BASE)	3256	142
L-F	99	22133
E-C(TIME)	707	375222
E-C(BASE)	4347	98
L-C	154	17287

表 4 barnes におけるトランザクショナルメモリの統計情報.

実現方式	abort	stall
E-F(TIME)	0	0
E-F(BASE)	2	0
L-F	4	412
E-C(TIME)	707	375222
E-C(BASE)	6318	0
L-C	206	2628

表 5 fmm におけるトランザクショナルメモリの統計情報.

logTM[2] のチームでは異なるハードウェアトランザクショナルメモリの実装に対して、トランザクション間に起こる実行性能向上を妨げる原因を列挙した上で評価を行っている。[5]

7. まとめと今後の課題

ハードウェアトランザクショナルメモリのプログラミングの容易さとトランザクション中の命令数に関係があると仮定した上で、SPLASH-2 内の barnes, fmm をベースに実行命令数を増加させたベンチマークを作成し、シミュレーションにより評価を行った。トランザクション内の命令数を増加させているために実行速度に関しては細粒度のトランザクションを行うものよりも低下しているが、速度低下率に関しては実装方式によって変わるが、Lazy Conflict Detection ポリシーを採用した場合にはこれら 2 つのアプリケーションでの速度低下が 1 %未満であるとわかった。

以上の評価から、Lazy Conflict Detection ポリシーを採用した場合には、この 2 つのアプリケーションに関しては最内関数レベルまではトランザクションの拡大に対して性能の低下が少なくてすむとわかった。今度の課題としては、実験のためのベンチマーク種類を増やすこと、より大きなトランザクションにした場合に対する性能の変化を評価することなどが挙げられる。

参 考 文 献

1) Rachid Guerraoui, Michal Kapalka, Jan Vieek. STMBench7: A Benchmark for Software Transactional Memory(EuroSys'07), 2007.
2) Kevin E. Moore and Jayaram Bobba and Michelle J. Moravan and Mark D. Hill and David A. Wood. LogTM: Log-based Transactional Memory. Proceedings of the 12th International Symposium on High-Performance Computer Architecture. pp. 254–265.
3) Lance Hammond and Vicky Wong and Mike Chen and Brian D. Carlstrom and John D. Davis and Ben Hertzberg and Manohar K. Prabhu and Honggo Wijaya and Christos Kozyrakis and Kunle Olukotun.Transactional Memory Coherence and Consistency.Proceedings of the 31st Annual International Symposium on Computer Architecture. pp. 102. IEEE Computer Society.
4) Ravi Rajwar and Maurice Herlihy and Konrad Lai.Virtualizing Transactional Memory.Proceedings of the 32nd Annual International Symposium on Computer Architecture. pp. 494–505. IEEE Computer Society.

5) Jayaram Bobba and Kevin E. Moore and Luke Yen and Haris Volos and Mark D. Hill and Michael M. Swift and David A. Wood.Performance Pathologies in Hardware Transactional Memory.Proceedings of the 34th Annual International Symposium on Computer Architecture.
6) Michael L. Scott and Michael F. Spear and Luke Dalessandro and Virendra J. Marathe. Delaunay Triangulation with Transactions and Barriers.PROC of the 2007 IEEE INTL Symposium on Workload Characterization. Boston, MA. Benchmarks track.
7) Maurice Herlihy and J. Eliot B. Moss .Transactional Memory: Architectural Support for Lock-Free Data Structures.Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 289–300.
8) Sean Lie.Hardware Support for Unbounded Transactional Memory. Masters thesis, Massachusetts Institute of Technology.
9) Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proceedings of the 22nd International Symposium on Computer Architecture, pages 24-36.
10) Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood.Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News (CAN), September 2005.