計算システム工学分野

題　　目　ハードウェアトランザクショナルメモリにおける競合パターンに応じた競合再発抑制手法の適用

学籍番号 23413514 氏名　江藤　正通
指導教員名　津邑　公暁

1 はじめに

マルチコアを用いた並列プログラミングでは，一般的に共有リソースへのアクセス制御にロックが用いられているが，この場合デッドロックの発生や並列性の低下等の問題がある．そこで，ロックを用いない並行性制御機構として**トランザクショナル・メモリ**[1](以下，**TM**)が提案されている．このTMをH/W上に実現したものは**HTM**と呼ばれ，一般的にフラグを用いたアボート対象の選択方法を採用している．

しかし，この方法ではスケジューリングの効率が悪く，トランザクション(以下，Tx)のアボートやストールが頻発する場合がある．そこで本研究では，この問題を解決する2つの手法を提案する．

2 ハードウェア・トランザクショナル・メモリ

HTM では，Tx として定義された命令列の直列可能性と不可分性を保証する為，Tx 内でアクセスされるメモリアドレスを監視する．そして複数の Tx 間で同一アドレスへのアクセスが発生した場合，少なくとも一方が Write アクセスならば**競合**として検出される．競合が発生した場合，片方の Tx の実行を中断する(**ストール**)．更に複数の Tx がストールしている状態でデッドロックの可能性があると判断された場合，それに関わるどれか 1 つの Tx の途中結果を破棄する(**アボート**)．そしてアボートされた Tx の開始時点の状態を復元し，Tx を再実行する．

なお，Tx のアボート後即座に再実行した場合，競合が再発しやすい為，2 つの待機時間を設ける技術(exponential backoff，magic waiting)がある．前者はアボートが発生する毎に待機時間を指数関数的に増大させる技術で，後者は競合相手の Tx がコミットするまで待機する技術である．後者を用いた場合，並列実行可能な Tx も待機する場合がある為，一般的な HTM では前者が用いられている．しかし前者を用いた場合でも，性能に悪影響を及ぼす競合パターンが存在する．本研究では starving writer の発生および futile stall の発生という 2 つの競合パターンに注目しそれらを解決する手法を提案する．

3 Starving Writer 解消手法

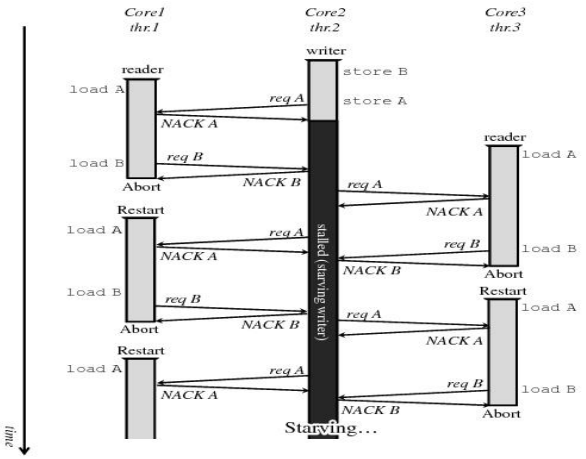

図 1: Starving writer の発生

Starving writer とは，図 1 のようにあるアドレスに対して書き込もうとする Tx(writer)が，そのアドレスから値を読み出す複数の Tx(reader)により妨げられ続ける事により発生する．starving writer が発生する場合，次の2つの特徴見られる．(1) Write after Read(WaR)競合が存在．(2) 同一アドレスに対するアクセスでアボートが発生．そこで，これら 2 つの動作が発生した場合に，reader に magic waiting を適用する事で writer を優先的にコミットさせる．本研究では，この動作を WaR アクセスの有無を記憶するフラグおよび，競合アドレスを記憶するレジスタを追加する事で実現する．

4 Futile Stall 防止手法

競合を頻繁に引き起こす複数のTxが並列実行されるとき，図2のように開始時刻の早いTxがストールさせられる状況が頻繁に発生する．この競合後，これらのTx間で再度競合が発生した場合，開始時刻の遅いTxがアボートされる．この場合，結果的には開始時刻の早いTxをストールさせる必要がなかった事になる．このようなストールをfutile stallとい

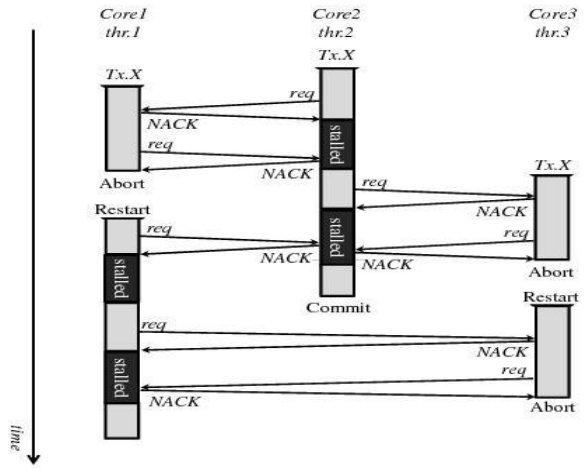

図2: Futile stallの発生

う. Futile stallが多く発生する場合, 並列実行されるTx間で競合が頻発していると考えられ,図2のようにアボートの繰り返しが発生する. このため, アボートを一定回数繰り返したTxを逐次的に実行する事でアボートを防ぐ事ができ, futile stallも抑制できると考えられる. なお, 逐次実行の対象は実行命令数の少ないTxに限定することで性能低下を防止する. これらの動作は, その対象Txを示すフラグや順序制御用のフラグ, 命令数やアボート回数をカウントするカウンタを追加する事で実現する.

5 評価

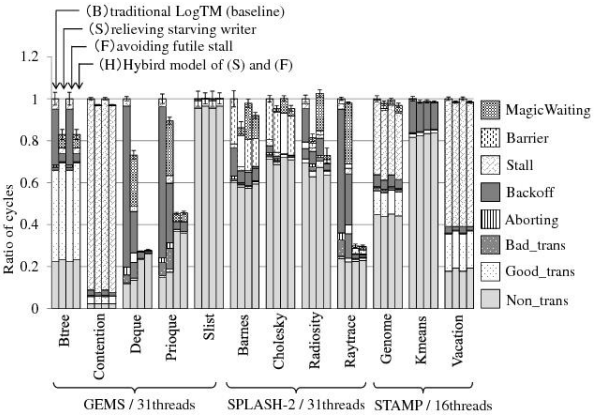

図3: 評価結果

2つの提案手法をHTMの研究で広く用いられているLogTMに実装し, 評価を行った. 評価にはGEMS付属のmicrobench, SPLASH-2, STAMPの3種のベンチマークから12種類のプログラムを用いた. 評価結果を図3に示す. 図中の凡例はサイクル数の内訳を示しており, 下から順にTx外の実行サイクル数, コミットされたTxの実行サイクル数, アボートされたTxの実行サイクル数, アボート, backoff, ストール, バリア同期, magic waitingに要したサイクル数を示している. また,4本のバーは左から順に(B) 既存のLogTM, (S) starving writer解消モデル, (F) futile stall防止モデル, (H) 2つの提案手法を組み合わせたモデルの結果を示しており, (B)のサイクル数を1として正規化している.

まず,(S)では多くのプログラムで性能向上が得られたことが分かる. これは, starving writer解消手法が解決すべき対象とした競合の再発, およびそれに伴うアボートの頻発を含んでいたためである. 最もstarving writerの影響を受けるBtreeでは, (S)により既存モデルよりもアボートが86.8%も削減できており, 最長のアボート繰り返し回数についても約1/4程度に削減された.

次に, (F)ではDeque, Prioque, Raytraceで大きく性能向上が得られたことが分かる. これらのプログラムでは, 特定のTxでアボートが頻発しており, これらのTxのみを逐次実行した事が性能向上に繋がった. また, 競合があまり起きないTxを含むプログラムでは, ほとんど性能低下が発生していないため, 適切に逐次実行の対象が選択された事が分かる.

最後に,(H)ではBarnes以外のプログラムにおいて, 2つの提案手法の内の結果が良い方と同等の結果となった事が分かる. なおBarnesでは, (S)によって十分に高速化が得られる一部のTxを早期に逐次実行の対象としてしまったため(S)ほど性能が向上しなかったと考えらえる. また、RadiosityとGenomeでは, Tx毎に適切な手法が適用されたため, (H)の結果が(S)と(F)の結果を上回った.

6 おわりに

Tx実行時に発生する2つの競合パターンについて注目し各問題を解決する手法を提案した.2つの手法を組み合わせたモデルは既存モデルに対して平均で28.4%, 最大で72.2%の実行サイクル数を削減した.

今後の課題として, 逐次実行対象のTxをより適切に選択する事や, 遊休状態のコアを有効活用する事が挙げられる.

7 参考文献

[1] Herlihy, M. et al.: Transactional Memory: Architectural Support for Lock-Free Data Structures, Proc. of 20th Int'l Symp. On Computer Architecture (ISCA'93), pp. 289-300 (1993)