論文紹介: Language Support for Lightweight Transactions

久野　靖*

2004.6.3

1　はじめに

□ 紹介論文: Tim Harris, Keir Fraser, Languate Support for Lightweight Transactions, OOPSLA 2003, pp.388-402.

□ 並行言語、並行制御→昔からある研究分野

- 並行制御の機能とかある程度出尽くした感もあるし…
- 現在は「スレッドがあればいいや」であまり顧みられていない

□ この論文→ CCR+トランザクション。

- 実装のセンスもいいし実用性高い。なるほどと思わせるところがある
- 一番すごいのは「ブロックしなくてよい」こと。ウソ?!って感じ

2　INTRODUCTION

□ 1970 年代以降、主流言語では「並行」の分野の発展は止まっている

□ たいていは→マルチスレッド+排他+条件変数 (Java がそう)

□ これには問題点がある:

```
public synchronized int get() {
  int result;
  while(items == 0) wait();
  items--;
  result = buffer[items];
  notifyAll();
  return result;
}
```

- while で繰り返し wait() がなぜ必要なのか忘れられたり理解されてなかったりする
- データアクセスが保護されているかどうかを検査する仕組みなし
- get() と put() はある程度並行できるはずなのにこの方法ではできない

□ 本論文では Hoare の CCR(Conditional Critical Region) に立ち帰る

- CCR では「どの操作群を並行アクセスから保護するか」を指定可能 (?)
- ガード (「どの条件が満たせたら入っていいか」) が指定可能

```
public int get() {
  atomic(items != 0) {
    items--;
    return buffer[items];
  }
}
```

□ 問題はよい実現方法が知られていないことだった

- CCR 中で何と何にアクセスするかは分からない
- いつガードが空くか分からない
- → CCR を全部排他実行しガードを毎回再評価→当然ながらのろい→現在のような条件変数による制御が行われるようになった

□ 今日の技術進歩→非停止並行データ構造に基づく新しい実装

□ CCR を STM(Software Transaction Memory) に対応させる

- 2〜106CPU のマシンで評価の結果、単純な排他制御より優れているとの結果。十分工夫された排他制御と同程度

□ 本論文の 3 つの貢献:

- CCR ではじめて (i) 動的に衝突のない操作を並行実行可能、(ii) 条件の再評価は共有変数が更新された場合のみ、(iii) 非停止な実現→デッドロックしない
- 現代の OOPL と STM をはじめて融合。さまざまな興味深い問題

*筑波大学大学院経営システム科学専攻

- word サイズのデータをそのまま利用可能 (予約等不要)、かつ原子的更新だけでなくスレッド同期まで考慮した最初の実現

3 MOTIVATION

□ cc-NUMA や SMP の普及→並行性は当り前のものに

- しかし多くのプログラマは考えたくないと思っている
- ロック単位の粒度→大きいと性能低下、小さいと構築が大変

□ 内部でロックを含むデータ構造の構成→難しい問題

- SPECjbb の get()、put()、remov() →並行利用可能
- では removeLeast() は→自分でロックを操作しないとできない、全体をロックしてしまうとせっかくの並行性がなくなる

□ デッドロックの防止→システム全体の状況を知らないとできない

□ 優先順の場合→ priority inheritance →やはり全体知識必要

□ 手動でロックを掛けたり外したりするスタイルに問題がある

4 RELATED WORK

□ 並行制御のためのプログラミング言語機能

□ 非停止アルゴリズム、STM

4.1 Language features

□ JVM、CLR、POSIX スレッド→排他+条件変数

□ CCR 型の機能→ DP、Edison、Lynx の await 文

□ Rem --- セマフォで CCR を実現→前述の弱点。Schmid →静的解析による再評価の軽減、ただし制約が大

□ Argus --- トランザクション機能 (enter/leave) を導入した言語

□ Flanagan, Quader --- Java 上で atomic 構文

□ Lomet --- aciton 文。本論文に最も近い。本論文はこれに (i)2PL を不要にしデッドロックをなくした、(ii) 同期に用いる変数を明示しなくて済むようにした、という改良を行っている

4.2 Non-blocking algorithms

□ 非停止アルゴリズム→排他の問題を避けるために多く研究

- 定義: スレッドいくつが死んでも残りのシステムが停止することはないような構成→ nonblocking
- 当然、ロックは除外されることに

□ obstruction-free: nonblocking の 1 つのクラス。他のスレッドと競合しない限り進捗が保証される。本論文もこれを前提とする

- livelock はあり得る→ exponential backoff などの回避手段

□ メモリ上で直接非停止なシステムを作るのは大変→抽象化の方向を模索

- transactional memory →一群のメモリアクセスをトランザクションに (commit/abort)。ハードウェアを想定。
- Shavit、Touitou がソフトウェア TM を提唱。ただし制約が強い
- Herlihy らが CAS(compare and swap 命令) で済む実用的な方法を考案。Java への実装も。ただし open とトランザクションに参加するオブジェクトの同定などが必要→本論文ではこれらの制約も解消

5 LANGUAGE INTEGRATION

□ Java に前述の atomic 文を入れた。設計方針:

- CCR は内部のコードにできるだけ制約を課さないように←単一スレッド用のコードでも簡単に CCR で囲める
- CCR が使われてない部分に大きなオーバヘッドがないように。たとえば全オブジェクトに追加のフィールドを設けるなどは避ける

5.1 Identifying CCR

□ 構文は次の通り

```
atomic(条件) { 文… }
```

- 実行するスレッドにとっては通常の単一スレッド実行と同じ
- 他のスレッドにとっては atomic に起こる

- exactly once セマンティクス。単純のため。at most once や timeout は避けた
- 中からでも普通に例外を投げてよい。

5.2 Data accessible to CCRs

□ CCR 内からはどのクラスのどのフィールドでもアクセスしてもよい

- 特定の継承階層のオブジェクト等だとコードの再利用ができない
- この方針から→ word-based STM

5.3 Native methods

□ CCR 中での native method 実行は禁止している。実行すると例外

- 標準クラスのものもなのでやや不便→問題ないものは許可できるようにしたい

5.4 Nested CCRs

□ 動的に CCR が入れ子になってよい (CCR の中で CCR を含むものを呼ぶ等)

- 動くように設計することはプログラマの責任

5.5 Existing synchronization mechanisms

□ CCR と既存の同期機構の係わりを検討しておく必要

□ CCR 中でロックを獲得する場合→ CCR 入口で atomic に獲得。従って CCR どうしの通信に排他オブジェクトを使うことは可能

□ CCR 中では wait() は無意味→ wait() と notify()、notifyAll() を禁止

5.6 Class loading

□ Java では初めてクラスにアクセスしたときローディング/初期化

- CCR 中で初めてアクセスした場合は→ CCR 中で初期化等が実行されるのは問題あり
- CCR 中で初めてアクセスした場合、まずローディング等が起き、その後で CCR の atomic 部分が実行

5.7 Consistency model

□ Java でのメモリモデルとの関係も検討しておく必要

□ Java ではスレッド間で共有するメモリは (i) 排他領域に入れるか、(2)volatile 指定、によって順序が保証

□ これに CCR も自然な形で追加

6 SOFTWARE TRANSACTIONS

□ CCR 用の STM の実装について

- ソフトによる実装だがハードでも同様
- word サイズの CAS 命令 (ないし同等のもの) が前提

□ STM の特徴

- 特別な場所を予約する必要がない。32 ビット全部使える
- word 幅の atomic access だけ保証されれば double 等も問題なし
- 制御用データ構造は静的に割り当て可能。作業領域は普通のヒープでよい
- read は共有メモリの書き込みなしで実現できる→キャッシュが有効

6.1 STM interface

□ メモリアクセス群に対し nest しないトランザクション機能を提供

```
void STMStart() --- trans 開始
void STMAbort() --- アボート
boolean STMCommit() --- コミット
boolean STMValidate() --- コミット可能?
void STMWait() --- 競合 trans 完了を待って abort
```

□ atmic ブロック (入れ子なし) の実現

```
boolean done = false;
while(!done) {
  STMStart();
  try {
    if(条件) {
      本体動作;
      done = STMCommit();
    } else {
      STMWait();
    }
  } catch(Throwable t) {
    done = STMCommit();
    if(done) throw t;
  }
}
```

□ 入れ子 CCR は 1 つの trans で実現
□ メモリ読み書き用のインタフェースも用意

```
stm_word STMRead(addr a) --- メモリ読み
void STMWrite(addr a, stm_word w) --- 書き
```

6.2 Heap structure

□ データ構造は 3 つある (図 2)

□ application heap --- 普通のヒープ

□ orecs (ownership records) --- trans 制御のため

- ownership 関数でヒープアドレスから orecs へのマッピング定義
- orec はバージョン番号または current owner を格納
- ヒープ内容が更新されるごとに番号は増える (当面再利用なしと仮定)

□ transaction descriptors

- 生きているトランザクションごとに 1 つ (当面再利用なしと仮定)
- ヒープ上の更新を記録: transaction entry = (アドレス, 旧バージョン, 旧値, 新バージョン, 新値)
- 状態: ACCTIVE、COMMITED、ABORTED、ASLEEP
- well formed: (i) エントリが 1 つだけ、または (2) 全エントリで新旧バージョン番号がすべて一致

□ ヒープ上のアドレスの logical state = (値, バージョン) を考える。次の 3 つの状況がある (排他)

- LS1: バージョン番号は orec、値はヒープ
- LS2: ディスクリプタにそのアドレスのエントリが含まれる
- LS3: そのアドレスのエントリがない。この場合は同じ orec に対応する他のアドレスのエントリを探し、バージョン番号はそのアドレスのバージョン番号 (ないし COMMITED ならそれ+1)、値はヒープ。well formed であれば LS3 の値も一意に決まる

□ あるアドレスの logical state は直接 LS1〜LS3 で求められる

- 再利用なしなので、計算後変更がないか再チェックで大丈夫

```
do {
  orec = orec_of(addr);
  アドレスの logical state を orec から求める
} while(orec_of(addr) != orec);
```

- LS1: orec が変化してなければヒープ上の値も変わっていない
- LS2〜LS3: エントリは一度用意されたら変わらない。状態は ACTIVE からその他の 1 つに 1 回だけ変化し得る
- 以上から読み出した snapshot は整合性がある

6.3 STM operations

□ orec は通常はバージョンのみ保持

□ orec がディスクリプタを参照するのは trans が COMMIT または SLEEP になろうとする時だけ。つまり STMCommit、STMWait の直前までは trans は独立に動いている (変更内容はディスクリプタに累積)

□ commit で multi word compare and swap algorithm を適用

6.4 STMStart

□ 新しいディスクリプタを割り当て ACTIVE に

6.5 STMAbort

□ 状態を ABORTED にするだけ

6.6 STMRead

□ 既にアドレスのエントリがある→新しい値を返す

□ ない→エントリを作る。ディスクリプタに同じ orec のエントリがあればそのバージョン番号をコピー、そうでなければ旧バージョンを使う

6.7 STMWrite

□ アドレスのエントリがなければ作る (read すればよい)。書く値は new-value に、new-version は old-versin + 1 (他のエントリも同じに更新する --- well formed を維持)

6.8 STMCommit

□ 必要なすべての orec を acquire →成功したら→すべて COMITTED に→ヒープ内容を更新→すべて release

□ 鍵は acquire と release。いずれもディスクリプタ td とその中のエントリの番号 i を受け取る

```
acquire(trans_desc *td, int i) {
  tarns_entry te = td.entries[i];
  orec seen = CAS(orec_of(te.addr),
                  te.old_version, td);
  if(seen == te.old_version || seen == td)
    return TRUE; /*1 成功 or 既に所有*/
  else if(holds_version_number(seen))
    return FALSE; /*2 バージョン相違:失敗*/
  else
    return BUSY; /*3 他人が所有*/
}
```

- CAS(a, x, y) : アドレス a の内容が x であるなら、それを y に変更 (もちろん atomic に)。返値はアドレス a の内容

□ acquire で FALSE に遭遇→状態を ABORTED にしてすべて release

□ acquire で BUSY に遭遇→誰かが owner になって使用中

- owner を ABORT する
- owner が STMWait() で停止中なら起こす
- 自分は ABORT して再試行に

□ すべて acquire 成功→状態を COMMITED にする (論理的にはこれによってすべての値が更新される)

- 値をヒープに書き戻す (この間にアクセスしてきた人はディスクリプタ側の値を見るのでタイミングは問題にならない)
- 最後に release

```
release(trans_desc *td, int i) {
  trans_entry te = td.entries[i];
  if(td.status == COMMITED)
    CAS(orec_of(te.addr), td, te.new_version)
  else
    CAS(orec_of(te.addr), td, te.old_version)
}
```

6.9 STMValidate

□ 現在の trans のアクセス対象がすべて想定するバージョン番号を持つかチェック

6.10 STMWait

□ すべて acquire するまでは COMMIT と同じ。その後、自分の状態を ASLEEP にして寝る。他人が STMCommit() しようとして BUSY なので分かる→起こしてもらえる

6.11 Optimization

□ ここまでの方式はいくつか不経済な点

- (i) 1 つの orec で最大 1 人しか寝られない
- (ii) read も write もエントリを探索する必要
- (iii) 読み出しのみアクセスでも orec を 2 回更新
- (iv) BUSY で再試行では non-blocking でない

6.12 Multiple sleeping threads

□ 眠る場所を 1 つでなく連結リストにすればよい

6.13 Read sharing

□ cache を汚さないために STMCommit の acquire/release での書き込みを何とかしたい

□ たとえば次のようにする? (i) 更新する場所だけ acquire する、(ii) 読む場所が old_value と old_version のままであることを確認、(iii)COMMIT にする←実際は atomic にならない

□ 新しい状態 READ_PHASE を追加し、その状態で (ii) を実行

□ 他の trans は READ_PHASE に遭遇したらその trans をアボート (実際には READ_PHASE は極めて短くほとんど遭遇できない)

6.14 Avoiding Searching

□ 各トランザクションのアクセスには時間局所性→ orec からエントリをマップする表に最近アクセスしたものをキャッシュ。読んで書く場合に効果的

6.15 Non-blocking commit

□ STMCommit() はこれまでのところ、他人が orec を取っていたらそれが終わるのを待つしかなかった→所有権を「盗む」ことで non-blocking 化。問題は次の 2 点

□ 盗人がどの場所の論理的な値も変更しないようにすること

- 持ち主を ABORT して持ち主のディスクリプタの内容を自分のものにマージする

□ 持ち主が COMMIT していた場合は書き戻し終わってから新しい持ち主が書くことを保証すること

- 全員の書き込みが終わって値が最後のトランザクションのものになっていることを確認してから release する→ orec にカウンタを用意して acquire で up、release で down する。自分が所有権を盗まれたことが分かった時は「新しい」所有者の更新を実行する

□ STMWait() は non-blocking でない (今のところ)。変更も可能

7 IMPLEMENTATION AND EVALUATION

□ Sun JVM 1.2.2 を改造して実装。比較対象は JVM でも改良された版

7.1 Modifications to the JVM and compilers

□ バイトコードへの翻訳とバイトコードから native への変換の 2 つを併せて実装 (.class ファイルのフォーマットは変更なし)

- atomic ブロックはメソッド単位となっている (メソッドになってないものは翻訳時にメソッドとして抽出)。名前に印をつける
- この中でローカル変数は通常通りアクセス。フィールドアクセス、配列要素アクセスは STMRead()、STMWrite() に。
- 各クラスに 2 つ目のメソッドテーブルを追加。トランザクション版のメソッド (必要に応じて変換) をキャッシュ。
- native へのトランスレータはループ中に STMValidate() チェックを挿入 (図 4 の例参照)
- volatile フィールド読み書きは小さい trans として実行

7.2 Memory management

□ ディスクリプタは普通に GC されるヒープに。

- 評価実験では再利用可能なディスクリプタをスレッドローカルにプールする方式 [13]。ディスクリプタの割り当て/開放は STM 操作に掛かる時間の 2%。
- orec は静的割り当て。
- 評価実験では 65535 個を割り当てアドレスのビット 2〜18 を対応させた。4096 に減らしても 1%未満の違いだった。

7.3 Version number

□ 奇数の整数を使用 (orec へのポインタと区別のため)。

- オーバフローは考慮しなかったが、たとえば全スレッドを止めてトランザクションは ABORT させ 1 に戻すとか (GC でもどうせ止めるし)
- CAS は 2 ワード (カウンタも必要なため)

7.4 Performance

□ 共有メモリ MP ではキャッシュの競合を避けることが性能向上のポイント。アプリケーションコードについてはプログラマの責任 (CCR であってもなくても)。

□ 実装に際してはよくあるケースを特別化。

- 例: STMCommit() はまず楽観的に競合なしと思って実行し、他人のディスクリプタが見つかったら通常の方法に変更 (1%未満)

□ コミットまではどのトランザクションもローカルに動けることに注意

□ 競合がない場合、r 箇所を読み w 箇所を書く trans は ← CAS × w、読み× r、ステータス更新× 2、書き× w、CAS × w。

7.5 Experimental set-up

□ Hashtable、Compound、Wait の 3 つの設定で実験

□ CCR と JVM の条件変数とを比較

□ 3 秒間実行し回数計測、5 回のメディアンを取る

7.6 Small systems

□ 4CPU SunFire v480

- ConcurrentHashMap(複雑な実装) がよいが、CCR も悪くない。競合が増えると CCR の方がよくなってくる

7.7 Large sytems

□ 106CPU SunFire e15k (ccNUMA)

- ConcurrentHashMap がよい場合もわずかにあるが CCR が一貫してよい。

7.8 STM performance summary

□ 競合しないトランザクションは完全に並行

□ 競合が少なければ単一ロックが「その場で操作」できるので速い

□ 細粒度ロックはロックの数が増えすぎなければ速い

□ CCR は STM インターフェースを常に使うから最初はコストがかさむが動的な競合がない限り並列に実行可能なので最もスケーラブル

- 「likely to be dynamically non-conflicting」なものに向く
- hot spot を避けるプログラミングに該当

7.9 Ease of programming

□ CCR 型のプログラミングスタイルはいいか?

- 昔からよく使われている、分かりやすい
- データベースのトランザクションとの類似性→簡潔なセマンティクス
- Java で考えると…wait/notify 不要なものは同じ (性能は CCR が上)、必要なものはずっと分かりやすい

8 DISCUSSION AND FUTURE WORK

□ Benchimarking and evaluation

- さまざまなデータ構造、ベンチマークで…

□ Language-level interface

- atomic 文以外…reflective (トランザクション中か、何にアクセスしているか、など見たりする)
- I/O のようなもの (副作用) →トランザクションの中断/再開とか callback とか

□ Implementation-level interface

- word level vs object level とか入れ子トランザクションとか

□ Alternative STM implementations

- 現在の実装が不得手な場面でうまく動く実装とか

□ Hardware support

- どんなハードがあるといいかとかソフトとのインタフェースとか

□ Summary

□ Acknowledgement