Swift 処理系における不可分命令のオーバーヘッドとその解決案

竹内 幹雄　河内谷 清久仁

参照カウント方式のメモリ管理では，個々のオブジェクトの被参照数 (参照カウント) を不可分命令を用いて更新し，それがゼロになったらそのオブジェクトを回収する．不可分命令は複数スレッドによるメモリアクセスの競合を避けるために必要であるが，それが起きないことが事前にわかっている場合，あるいは起きたことが事後にわかる場合は，通常の命令と補償コードを用いることでそのオーバーヘッドを削減できる．今回我々は Swift 処理系の通常の命令による単一スレッド向けのコード生成および実行時ライブラリを値型にまで拡張した．マイクロベンチマークによる評価では，複数スレッド向けのコードと比べて最大 87%の実行時間が削減されることが確認された．さらにトランザクショナルメモリによるオブジェクト回収時の子オブジェクトの参照カウントの更新の最適化についても述べる．

Automatic memory management based on reference counting updates reference count of each object with atomic operation and collects the object when the count becomes zero. Atomic operation is needed to avoid conflicts of memory access with multiple threads, however, if it is known in advance that the conflict won't happen or if it is possible to detect the conflict when it happens, we can eliminate the overhead by using non-atomic operation and compensation code. In this paper, we enhanced the Swift compiler and runtime library to use non-atomic operations for generating single-threaded code for value types. With micro benchmark suite, we confirmed that the execution time with single-threaded code was reduced by at most 87% of the original multi-threaded code. We also introduce an idea for using transactional memory to optimize decrements of reference count of child objects when the parent is about to be collected.

1　はじめに

自動メモリ管理は，プログラミング言語の歴史における最も重要な発明の一つである．データの生存期間はアプリケーション全体に関わる大域的な性質であるにも関わらず，手動メモリ管理はそれを局所的に判断しなければならず，本質的に危険である．自動メモリ管理はその重責からプログラマを解放する．

自動メモリ管理の種類には，大きく分けて，トレーシング方式と参照カウント方式の二種類がある．

トレーシング方式は，時折，大域変数などから再帰的に到達可能な全てのオブジェクトに印を付け，次のステップで印の付いていない全てのオブジェクトを一度に回収する．メモリ管理の平均オーバーヘッドが低く，アプリケーションのスループットが高い反面，アプリケーションに比較的長い停止時間が存在する弱点があり，レスポンスが重視されるユーザインタフェースなどの対話型アプリケーションに使いづらい面がある．

参照カウント方式は，個々のオブジェクトの参照を，変数に保存または変数から削除する際に，そのオブジェクトの被参照数 (参照カウント) を増減し，それがゼロになったらそのオブジェクトを回収する．アプリケーションに長い停止時間が存在しないものの，メモリ管理のオーバーヘッドが全体の実行時間に占める割合が比較的大きく，スループットが重視されるサーバーなどの非対話型アプリケーションには無駄が多く不向きである．

---

* The overhead of atomic operations and its solution for the Swift programming language.

Mikio Takeuchi, Kiyokuni Kawachiya, 日本アイ・ビー・エム (株) 東京基礎研究所, IBM Research - Tokyo.

2 Swift 処理系における参照カウント方式のメモリ管理

Swift [3] は米国 Apple 社によって設計実装されたプログラミング言語である．対話型アプリケーションのためのプラットフォームである iOS, watchOS, tvOS, macOS 向けの主要言語に位置付けられており，その処理系は参照カウント方式のメモリ管理を採用している．2014 年に発表された比較的新しい言語であるが，プログラミング言語の人気ランキング TIOBE Index [14] で 2017 年に過去最高の 10 位に入っており，世界中で多くのプログラマに利用されている．

2015 年にオープンソースとして公開されて以降，我々は Swift をハイブリッドクラウドプラットフォーム Bluemix 向けの言語の一つに位置付け，開発者に提供している．クラウドで動作するアプリケーションの多くは非対話型アプリケーションであるため，参照カウント方式のメモリ管理の特徴である，長い停止時間が存在しないことのメリットが生かせず，高いオーバーヘッドのデメリットが目立つ事がある．

参照カウント方式のメモリ管理のオーバーヘッドの多くの部分は，不可分命令による参照カウント更新に起因することがわかっている．不可分命令は，複数のコアを持つプロセッサ上で複数のスレッドが同時に同じオブジェクトの参照カウントを更新する際の競合を避けるために必要であるが，なんらかの手段でそれが起きないことを保証できれば，通常の命令で参照カウントを更新することでそのオーバーヘッドを削減できる．

そのような手段の一つで既に実用化されているものにエスケープ解析 [13] がある．エスケープ解析はプログラムのコンパイル時に行う静的な解析であり，オブジェクトを作成したスレッドがそれを他スレッドからもアクセス可能な状態にするコードを検出し，オブジェクトを複数スレッドからアクセスされる可能性のあるものとないものに分類する．オブジェクトを他スレッドからアクセス可能な状態にするコードが実際に実行されるとは限らず，またオブジェクトが他スレッドからアクセス可能な状態になっても，実際に他スレッドからアクセスされるとは限らないため，エスケープ解析の結果は保守的であり，その効果は限定的である．

本論文では，別の手段として，オブジェクトが複数スレッドからアクセスされないための条件として，アプリケーションが単一スレッドで動作するという性質を用いる．これはエスケープ解析より厳しい条件であるが，オブジェクトごとの判断が不要というメリットがある．アプリケーションが単一スレッドで動作することを保証するには，処理系がアプリケーションのコンパイル時または実行時に，スレッドの作成を検出するか，あるいはプログラマがアプリケーションのコンパイル時に単一スレッドで動作することを宣言すればよい．Swift コンパイラは後者のためのオプションを既に持っている．

今回我々はそのオプションの適用範囲を値型にまで拡張した．値型 (value type) は参照型 (reference type) と異なり複数の参照を持たない型であるが，現在の Swift 処理系はコピーオンライト (図 1) という仕組みで値型を実装しており，実装上は複数の参照を持ち不可分命令で参照カウントが更新されているため，メモリ管理のオーバーヘッドは参照型と変わらない．そのため通常の命令による単一スレッド向けコード生成の適用範囲を値型まで拡張することで大きな性能向上が期待された．実際にマイクロベンチマークで効果を測定したところ，図 2 に示すように，複数スレッド向けのコードと比べて最大 87%の実行時間が削減されることが確認できた．

3 Swift アプリケーションを単一スレッド向けにビルドする手順

Swift 処理系は，Swift もしくは C++で書かれた実行時ライブラリのバイナリを処理系のビルド時に作成している．2017 年 9 月の時点では通常のビルド時には複数スレッド向けのバイナリしか作成しないため，まずは Swift 処理系をビルドし直し単一スレッド向けのバイナリを作成する必要がある．その後 Swift で書かれたアプリケーションを単一スレッド向けにコンパイルし，作成した単一スレッド向けの実行時ライブラリのバイナリとリンクする．

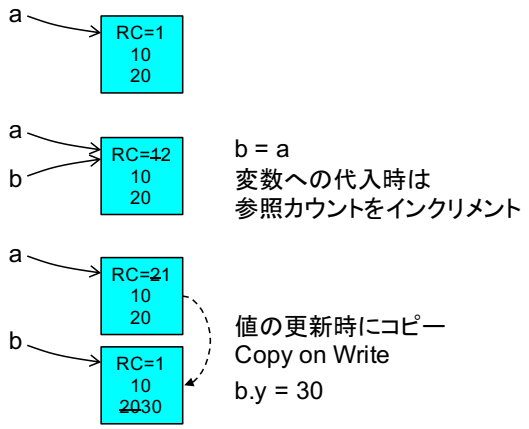

図 1　コピーオンライトによる値型の実装

3.1　Swift 処理系を単一スレッド向けにビルドする

本論文の読者は，Swift 処理系のビルドに慣れていると仮定する．もしまだの場合はビルド手順 [2] を一読し，`build-script` による通常のビルドができるようになってから進めて欲しい．以下では単一スレッド向けのビルドに必要な差分だけを示す．

3.1.1　macOS の場合

Swift 処理系を単一スレッド向けにビルドするには，ビルド時に以下のオプションを指定する．

```
swift-stdlib-use-nonatomic-rc=true
```

ビルド時に指定可能なオプションは多岐に渡るため，煩雑になるのを避けるため，Swift のビルドシステムは複数のオプションをまとめて表現するための preset という仕組みを持っている．

標準で用意されている macOS 向けの preset の一つが上記オプションを含むため，それを指定すれば単一スレッド向けにビルドされる．

```
$ build-script --preset=buildbot,tools=RA,\
stdlib=RD,single-thread
```

3.1.2　Linux の場合

残念ながら，標準で用意されている Linux 向けの preset に単一スレッド向けのものは存在しない．そこで Swift のビルドシステムが用意する仕組みを用いて自分用の preset を定義する．それには以下の内容からなるファイル `~/.swift-build-presets` を作成する．

`~/.swift-build-presets`

```
[preset: buildbot_linux,single-thread]
mixin-preset=buildbot_linux
swift-stdlib-use-nonatomic-rc=true
```

そして定義した preset を指定すれば単一スレッド向けにビルドされる．

```
$ build-script --preset=buildbot_linux,\
single-thread
```

3.2　Swift アプリケーションを単一スレッド向けにコンパイルする

Swift アプリケーションを単一スレッド向けにコンパイルするには，コンパイル時にオプション `-Xfrontend -assume-single-threaded` を指定する．

```
$ swiftc -Xfrontend -assume-single-threaded \
MyApplication.swift
```

4　トランザクショナルメモリによる参照カウントの更新

2, 3 節では，不可分命令の代わりに通常の命令でオブジェクトの参照カウントを増減することでそのオーバーヘッドを大きく削減できることを，単一スレッドで動作するアプリケーションを題材として示した．本節では同手法の適用範囲を，複数スレッドで動作するアプリケーションにまで広げる方法について議論する．

複数スレッドで動作するアプリケーションにおいても，実際に競合する参照カウントの更新は全体の一部であり，競合する確率も参照カウントの更新ごとにまちまちである．不可分命令による参照カウント更新のオーバーヘッドは，それが競合しなかった場合は結果的に無駄になるため，全ての参照カウントの更新に対する競合する参照カウントの更新の割合が比較的低い場合，競合に備えてあらかじめ全ての参照カウントの更新を不可分命令で実行する代わりに，まず通常の命令で参照カウントを更新し，競合が起きたら不可分命令でその参照カウントの更新をやり直した方が最

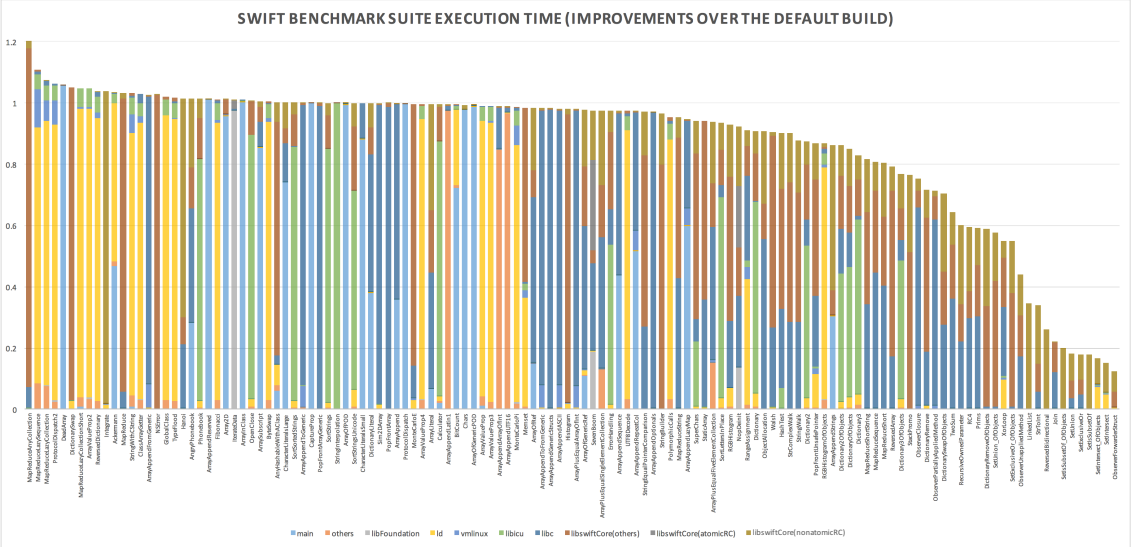

図 2　通常の命令による単一スレッド向けコード生成による性能向上

終的なオーバーヘッドが低くなる可能性がある．

メモリ更新の競合の発生はトランザクショナルメモリ †1 [5] を用いて検出することができる．トランザクショナルメモリは IBM Blue Gene/Q [15], IBM zEnterprise EC12 [9], IBM POWER 8 [10] などの商用プロセッサでサポートされている．ここでは米国 Intel 社の Transactional Synchronization Extensions [11] が提供する Restricted Transactional Memory (RTM) 命令セットを例題に，その適用可能性を検討する．

RTM では，XBEGIN, XEND 命令でトランザクションを開始または終了する．トランザクションの実行中に，通常の命令によるメモリ更新が競合すれば，そのトランザクションは失敗し，XBEGIN の引数として指定した補償コードに制御が移される．したがって補償コードとして不可分命令で同じメモリを更新するコードを書けば，ひとまず目的は達成できるが，XBEGIN, XEND にはそれ自身の実行コストがあるため，一つのメモリ更新を不可分命令から通常の命令に置き換えただけではそのコストに見合うだけのオーバーヘッドの削減効果が得られない．

Intel Core i7-4770 3.40GHz (4cores x 2HT) 上で，

†1　本論文ではトランザクショナルメモリはハードウェアトランザクショナルメモリを指すものとする

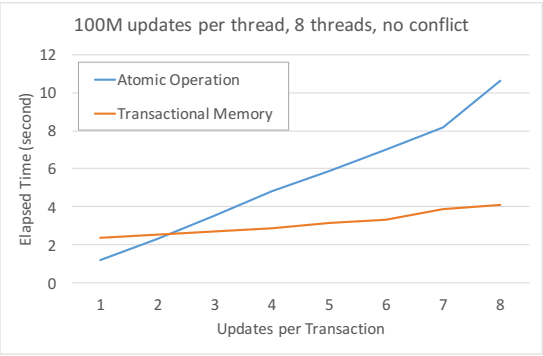

図 3　RTM の効果

g++ 5.4.0, GNU assembler 2.26.1 でコンパイルしたコードを，8 スレッドでメモリ更新を 1 億回ずつ並列実行した時間を比べたところ，競合が起きない場合，一つのトランザクションで三つ以上のメモリ更新を行えば，それに見合う効果が得られることがわかった (図 3)．

5　トランザクショナルメモリによるオブジェクト回収時の子オブジェクトの参照カウントの更新

4 節で，トランザクショナルメモリで性能を改善するには，一つのトランザクションで三つ以上のオブ

ジェクトの参照カウントを更新する必要があることがわかった．本節では，その可能性を探るため，参照カウント方式のメモリ管理の動作を検討する．

参照カウント方式のメモリ管理では，各オブジェクトが持つ参照カウントが，そのオブジェクトが他から指される時にインクリメント，指されなくなる時にデクリメントされる．デクリメントにより参照カウントがゼロになると，そのオブジェクトは回収される．オブジェクトの回収は，そのオブジェクトが参照する全ての子オブジェクトの参照カウントをデクリメントして (図 4)，ゼロになればそれを回収した後に，そのオブジェクトが占めるメモリを解放する．子オブジェクトは一般に複数あるため，それらの参照カウントのデクリメントを一つのトランザクションで実行すれば，性能向上のための条件を満たすことが可能である．

5.1 Swift 処理系におけるオブジェクトの回収

現在の Swift 処理系は，オブジェクトの回収を，同じ手続きの子オブジェクトへの再帰呼び出しにより 1 パスで実現している．それを仮想コードで示すと以下のようになる．

Procedure: Collect_Object(O)
  for all object P referenced from O **do**
    Decrement P 's RC with *atomic (heavy)* operation
    if the RC reached 0 **then**
      call Collect_Object(P)
    end if
  end for
  Free the memory area used by O

5.2 トランザクショナルメモリによるオブジェクトの回収

5.1 節のコードにトランザクショナルメモリを直接適用すると，トランザクションが入れ子になる．RTM はトランザクションの入れ子をサポートするが，トランザクションの失敗は常に最も外側のトランザクションを失敗させるため失敗時のペナルティが大きすぎて使いづらい．この問題を避けるため，子オブジェクトの参照カウントのデクリメントと，子オブジェクトの回収を分割し，前者を一つのトランザクションにまとめる．この分割により，前者と後者の間で，後に回収する子オブジェクトを覚えておく必要が生じるが，これは，回収しない子オブジェクトの参照をクリアすることで表現できる．補償コードではトランザクションを使わず，同じ手続きの子オブジェクトへの再帰呼び出しにより 1 パスで実現する．回収対象のオブジェクトが，性能向上に必要な数の子オブジェクトを持たない場合は，最初からこの補償コードを実行すればよい．以上を仮想コードで示すと以下のようになる．

Procedure: Collect_Object_TM(O)
  N is the number of objects referenced from O
  T is a threshold value decided from CPU specs, average failure ratio, etc.
  if $N \geq T$ **then**
    // Proposed method
    XBEGIN // Start a transaction
    for all object P referenced from O **do**
      Decrement P 's RC with *non-atomic (light)* operation
      if the RC reached 0 **then**
        remember P (e.g. by clearing P that has non-zero RC)
      end if
    end for
    XEND // End the transaction
    if the transaction failed **then**
      goto TRADITIONAL
    end if
    for all object Q whose RC reached 0 **do**
      call Collect_Object_TM(Q)
    end for
    Free the memory area used by O
    return
  end if
  TRADITIONAL:
  // Original method, for small N or transaction-failure cases

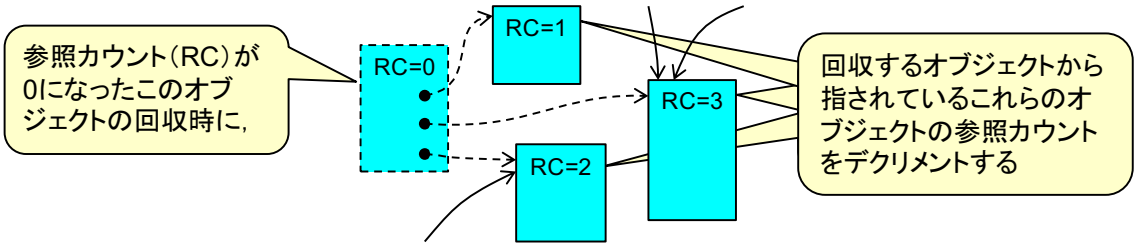

図 4　オブジェクト回収時の子オブジェクトの参照カウントのデクリメント

for all object P referenced from O **do**
  Decrement P 's RC with *atomic (heavy)* operation
  if the RC reached 0 **then**
    call Collect_Object_TM(P)
  end if
end for
Free the memory area used by O

6　関連研究

参照カウント更新のオーバーヘッドを削減する試みとしては，次の 2 つが代表的である．

延期型参照カウント法 (deferred reference counting) [4] は，ローカル変数から参照されるオブジェクトを，参照カウントを更新する代わりにゼロカウント表に登録して管理する方法である．参照カウントがゼロであることが，そのオブジェクトがどこからも参照されていないことを意味しないため，オブジェクトの即時回収ができず，回収のために時々全体のアプリケーションを止める必要がある．メモリの利用効率も多少下がる．それらはいずれもメモリの少ない携帯端末で動作する対話型アプリケーションの実行に影響があるため，Swift には採用されていない．

併合型参照カウント法 (coalesced reference counting) [12] は，オブジェクトのフィールドから参照されるオブジェクトを，参照カウントを更新する代わりに，前回の回収以降の初回のフィールドへの書き込み時に，そのオブジェクトの書き込み前のフィールドの値をスレッドごとのログに登録して管理する方法である．延期型参照カウント法と同様に，オブジェクトの即時回収ができず，回収のために時々全体のアプリケーションを止める必要がある．メモリの利用効率も多少下がるため，Swift では採用されていない．

2017 年に発表予定の Swift 4 では，memory ownership model [1] が導入される予定になっている．ownership を明示的に指定することで，オブジェクトの参照の変数への代入時の挙動をプログラマがきめ細かく制御できるようになる予定である．

7　おわりに

本論文で，我々は Swift 処理系の通常の命令による単一スレッド向けのコード生成および実行時ライブラリを値型にまで拡張した．マイクロベンチマークによる評価では，通常の複数スレッド向けのコードと比べて最大 87%の実行時間が削減されることが確認された．Swift アプリケーションを単一スレッド向けにビルドする具体的な手順についても解説した．さらにトランザクショナルメモリによるオブジェクト回収時の子オブジェクトの参照カウントの更新の最適化についても述べた，

我々は他にも Server Side Swift に関する研究開発を行なっている．例えば Linux 上での効率のよいスレッドのスケジューリング方法や，複数スレッドアプリケーションでの効率の良い参照カウントの実装方法，Swift による web サーバフレームワークの Kitura [6] などがある．我々の Swift 関連技術の全般については Swift@IBM web site [8] や Swift@IBM Slack channel [7] を参照されたい．

参 考 文 献

[1] Apple Inc.: Ownership manifesto, http://github.com/apple/swift/blob/master/docs/OwnershipManifesto.md.

[2] Apple Inc.: Swift GitHub repository, http://github.com/apple/swift.

[3] Apple Inc.: Swift web site, http://swift.org/.

[4] Deutsch, L. P. and Bobrow, D. G.: An Efficient, Incremental, Automatic Garbage Collector, *Commun. ACM*, Vol. 19, No. 9(1976), pp. 522–526.

[5] Herlihy, M. and Moss, J. E. B.: Transactional Memory: Architectural Support for Lock-free Data Structures, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, New York, NY, USA, ACM, 1993, pp. 289–300.

[6] IBM Corporation: Kitura web site, http://www.kitura.io/.

[7] IBM Corporation: Swift@IBM Slack channel, http://swift-at-ibm-slack.mybluemix.net/.

[8] IBM Corporation: Swift@IBM web site, http://developer.ibm.com/swift/.

[9] IBM Corporation: z/Architecture Principle of Operations. SA22-7832-10, http://www.ibm.com/support/docview.wss?uid=isg2b9de5f05a9d57819852571c500428f9a, February 2015.

[10] IBM Corporation: IBM Power ISA^{TM} Version 3.0B. Book II. Chapter 5: Transactional Memory Facility, http://openpowerfoundation.org/?resource_lib=power-isa-version-3-0, March 2017.

[11] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1. Chapter 16: Programming with Intel® Transactional Synchronization Extensions, 253665-063US, July 2017.

[12] Levanoni, J. and Petrank, E.: A Scalable Reference Counting Garbage Collector, Technical Report CS-0967, Technion - Israel Institute of Technology, Haifa, Israel, November 1999.

[13] Park, Y. G. and Goldberg, B.: Escape Analysis on Lists, *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, New York, NY, USA, ACM, 1992, pp. 116–127.

[14] TIOBE Software: TIOBE Index, http://www.tiobe.com/tiobe-index/.

[15] Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., and Michael, M.: Evaluation of Blue Gene/Q Hardware Support for Transactional Memories, *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, New York, NY, USA, ACM, 2012, pp. 127–136.