Haskellによる非破壊的木構造を用いたCMSの実装

當眞 大千　　河野 真治　　永山 辰巳

ウェブサービスではユーザの増加に対応し、容易に拡張できるスケーラビリティが求められる。コンテンツマネージメントシステムにおいてスケーラビリティを実現するためには、並列にデータへアクセスできる設計が必要となる。本研究では、編集元の木構造を破壊することなく編集できる非破壊的木構造を利用する。非破壊的木構造では、少ない排他制御で変更を行えるためスケーラビリティを確保できる。コンテンツマネージメントシステムの実装にはプログラミング言語 Haskell を用いた。Haskell で書かれた HTTP サーバ Warp を用いて簡易掲示板システムを開発し、既存の Java の実装と比較して短い開発期間やコード行数で、同程度の性能を達成できた。

1 はじめに

ブロードバンド環境やモバイル端末の普及により、ウェブサービスの利用者数は急激に伸びている。リクエスト数の増加を予想することは困難であり、負荷が増大した場合に容易に拡張できるスケーラビリティが求められる。ここでいうスケーラビリティとは、利用者や負荷の増大に対し、単なるリソースの追加のみでサービスの質を維持することができる性質のことである。コンテンツマネージメントシステムにおいてスケーラビリティを実現するためには、並列にデータへアクセスできる設計が必要となる。

本研究では、並列にデータへアクセスする手法として、非破壊的木構造を利用する。非破壊的木構造では、少ない排他制御で変更を行えるためスケーラビリティを確保できる。

Implementation of the CMS using Nondestructive Tree Structure and Haskell

Daichi TOMA, Shinij KONO, 琉球大学大学院理工学研究科情報工学専攻並列信頼研究室, Dept.Concurrency Reliance Laboratory, Information Engineering Course, Faculty of Engineering Graduate School of Engineering and Science, University of the Ryukyus.

Tatsumi NAGAYAMA, 株式会社 Symphony, Symphony Co., LTD.

非破壊的木構造を用いたデータベースとして、本研究室による Java による実装が既に存在する。しかしながら、Functional Java を多用しており、それならば純粋関数型言語である Haskell のほうが相性がいいのではないかと考え、本研究では Haskell による実装を行った。

Haskell による実装では、Java と比較して開発期間およびコード行数が非常に短くなるといったメリットがあった。

また、性能比較のために Haskell で書かれた HTTP サーバ Warp を用いて簡易掲示板システムを開発し、既存の Java の実装と同程度の性能を達成できた。

2 Haskell

Haskell は、純粋関数型プログラミング言語である。関数型プログラミング言語では、引数に関数を作用させていくことで計算を行う。変数への代入は一度のみで、書き換えることはできない。遅延評価や、強い静的型付けも Haskell の特徴である。

3 Warp

Warp[1] は、軽量、高速な HTTP サーバである。Haskell の軽量スレッドを活かして書かれている。Haskell の ウェブフレームワーク である Yesod の

バックエンドとして用いられており、現在も開発が続けられている。

3.1 Warp を用いたウェブアプリケーションの構築

Warp を用いてウェブアプリケーションを構築する方法について考察する。

```
application counter request = function
    counter
  where
    function = routes $ pathInfo request

routes path = findRoute path
    routeSetting

findRoute path [] = notFound
findRoute path ((p,f):xs)
    | path == p = f
    | otherwise = findRoute path xs

routeSetting = [(["hello"], hello),
                (["hello","world"],
                    world)]

notFound _ = return $
    responseLBS status404 [("Content-
        type", "text/html")] $ "404"

hello _ = return $
    responseLBS status200 [("Content-
        type", "text/html")] $ "hello"

world counter = do
    count <- lift $ incCount counter
    return $ responseLBS status200 [("
        Content-type", "text/html")] $
      fromString $ show count

incCount counter = atomicModifyIORef
    counter (\c -> (c+1, c))

main = do
  counter <- newIORef 0
  run 3000 $ application counter
```

ソースコード 1　Warp を用いたウェブアプリケーションの例

ソースコード 1 は、URL によって出力する結果を変更するウェブアプリケーションである。/hello/world へアクセスがあった場合は、インクリメントされる counter が表示される。

main

HTTP サーバを起動するには、Warp の run 関数を利用する。run 関数は、利用する Port 番号と、application というリクエストを受けて何かしらのレスポンスを返す関数の 2 つを引数として受け取る。

関数型言語では、関数を第一級オブジェクトとして扱える。また、今回は Haskell のカリー化された関数の特性を利用し、main 内で作成した IORef 型の counter を部分適用させている。

IORef を用いることで、Haskell で更新可能な変数を扱うことができる。参照透過性を失うようにみえるが、Haskell は IO モナドを利用することで純粋性を保っている。IORef 自体が入出力を行うわけではなく、単なる入出力操作の指示にすぎない。IO モナドとして糊付けされた単一のアクションに main という名前を付けて実行することで処理系が入出力処理を行う。

application 及び routes , findRoute

application の実装では、routes という関数を独自に定義して、URL によって出力を変更している。application に渡されるリクエストはデータ型で様々な情報が含まれている。その中のひとつに pathInfo という、URL から hostname/port と、クエリを取り除いたリストがある。この情報を routes という関数に渡すことで、routeSetting というリストから一致する URL がないか調べる。routeSetting は、URL のリストとレスポンスを返す関数のタプルのリストである。

notFound 及び hello

レスポンスを返す関数は、いくつか定義されている。その中で利用されている responseLBS は文字列からレスポンスを構築するためのコンストラクタである。

world 及び incCount

world は、インクリメントされる counter を表示するための関数である。IORef 内のデータは直接触ることができないため、incCount 内で atomicModifyIORef を利用してデータの更新を行なっている。atomicModifyIORef は、データの更新をスレッドセーフに行うことができる。また、responseLBS

で構築したレスポンスは、Resource T というリーソスの解放を安全に行うために使われるモナドに包まれている。lift 関数を用いて、incCount の型を持ち上げ調整している。

実際にプログラムを例にして説明したが、Warp は容易にプログラムに組み込むことができる。我々のシステムでは Warp を用いて開発を行う。

4　コンテンツマネージメントシステムの設計

コンテンツマネージメントシステムのデータ構造としては木構造を用いる。スケーラビリティのある CMS の実現のために非破壊的木構造 [2] を利用した。非破壊的木構造とは、編集元の木構造を破壊することなく新しい木構造を構成することで木構造を編集する方法である。

破壊的木構造と異なりロックせずに並列に読むことができるため、自由にコピーを作成することが可能である。コピーを複数作成し、アクセスを分散させることで性能を維持することができる。

4.1　非破壊的木構造

非破壊的木構造は、木構造を書き換えることなく編集を行うため、読み書きを並列に行うことが可能である。図 1 では、ノード 6 をノード A へ書き換える処理を行なっている。

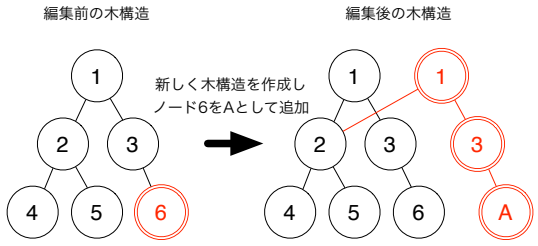

図 1　木構造の非破壊的編集

非破壊的木構造の基本的な戦略は、変更したいノードへのルートノードからのパスを全てコピーする。そして、パス上に存在しない (編集に関係のない) ノードはコピー元の木構造と共有することである。

編集は以下の手順で行われる。

1. 変更したいノードまでのパスを求める。
2. 変更したいノードをコピーし、コピーしたノードの内容を変更する。
3. 求めたパス上に存在するノードをルートノードに向かって、コピーする。コピーしたノードに一つ前にコピーしたノードを子供として追加する。
4. 影響のないノードをコピー元の木構造と共有する。

この編集方法を用いた場合、閲覧者が木構造を参照してる間に、木の変更を行っても問題がない。閲覧者は木が変更されたとしても、保持しているルートノードから整合性を崩さずに参照が可能である。ロックをせずに並列に読み書きが可能であるため、スケーラブルなシステムに有用であると考えられる。元の木構造は破壊されることがないため、自由にコピーを作成しても構わない。したがってアクセスの負荷の分散も可能である。

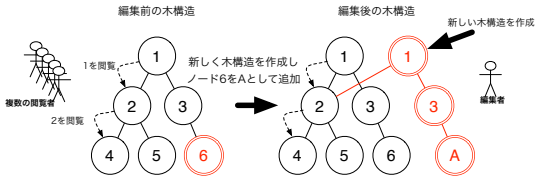

図 2　並列に読み書きが可能な非破壊的木構造

非破壊的木構造を用いて、コンテンツマネージメントシステムの開発を行う。

5　コンテンツマネージメントシステムの実装

コンテンツマネージメントシステムのデータ構造としては木構造を用いると述べた。我々が開発している木構造データベース Jungle [3] について説明する。

Jungle は、非破壊的木構造を扱う木構造データベースで、既に Java による実装が存在する。本研究では、Haskell を用いて実装を行った。コンテンツマネージメントシステムに組み込んで利用しているが、他のシステムに組み込むことも可能である。

5.1 データベースオブジェクトと木の作成

木の作成

Jungle は複数の木を保持することができる。木には名前がついており、名前を利用して判別を行う。また、作成・編集・削除を行うことができる。

```
jungle = createJungle
new_jung = createTree jungle "new_tree"
```

ソースコード 2　データベースオブジェクトと木の作成

createTree 関数を利用して木構造を作成する。

木と木を構成するノード

データベースオブジェクトを作成し、木構造とルートノードを取得するためには以下のように記述する。

```
tree = getTreeByName new_jung "new_tree"
node = getRootNode tree
```

ソースコード 3　木とノードの取得

getTreeByName 関数で名前を指定することで木構造を取得できる。getRootNode 関数でルートノードを取得できる。

木の編集

addNewChildAt 関数で、ノードに新しい子を追加することができる。また、putAttribute 関数で、ノードが持つ連想リストを編集できる。どのノードを編集するかという情報は、ルートノードからのパスを渡すことで解決する。木を編集したあと、updateTree 関数を用いて既存の Jungle に変更を加え新しい Jungle を作成する。

```
new_tree  = addNewChildAt tree [0,1] 0
new_tree2 = putAttribute new_tree
    [0,1,0] "key" "value"
new_jungle = updateTree jungle new_tree2
```

ソースコード 4　木の編集

5.2 木の取り扱い

Jungle の 木 の取り扱いには、Haskell の Map を用いている。Map は、平衡木を使った Haskell の連想リストである。連想リストを用いて、名前と木を結びつけている。

5.3 データ構造

木のデータ構造は、データ型で定義されている。

```
data Node = Empty
          | Node
          { children   :: Children
          , attributes :: Attributes
          } deriving (Show)
```

ソースコード 5　データ構造

各ノードは、Children としてノードを複数持つことができる。Children および Attributes も、Map を用いて定義されている。

5.4 ルートノードの取り扱い

非破壊的木構造であっても、どのノードが最新のルートノードなのかという情報が必要である。スレッドセーフに取り扱う必要があるため、Haskell のソフトウェア・トランザクショナル・メモリを用いて管理している。

5.5 開発期間の短縮

Java 版の Jungle の実装と比較すると、コード行数は約 3000 行から約 150 行へ短くなった。また開発期間は Java 版の実装で、3 ヶ月程度かかったが、Haskell 版の実装は 2 週間程度であった。これにより、関数型プログラミングではコードは短くなり、生産性が向上することが分かった。

6 木構造データベース Jungle を用いた CMS の検証

木構造データベース Jungle 及び Warp を用いて簡単な掲示板ウェブアプリケーションを作成した。同様のウェブアプリケーションを、Java による Jungle 実装 及び Cassandra[4] 上でも動かし性能比較を行う。Cassandra は、Facebook が自社のために開発した分散 Key-Value ストアデータベースであり、Dynamo[5] と BigTable[6] を合わせた特徴を持っている。Java 版では、組み込みウェブサーバである Jetty を利用する。

6.1 実験方法

複数のクラスタを利用して、サーバに対して並列にアクセスを 5000 回行い、それぞれクラスタの実行平均時間をとる。クラスタ台数を増やすことにより並列度を上昇させ、並列度と実行時間の平均をグラフ化する。測定するのは書き込みと読み込みであり、掲示板のメッセージの取得と掲示板のメッセージの編集を行う。

6.2 実験環境

負荷をかける対象であるサーバは、マルチコア環境が生かされているか確認するためにコア数の多いマシンを用いる。サーバの仕様を表 1 に示す。

表 1　検証に使用するサーバーの仕様

名前	概要
CPU	Intel®Xeon®X5650 @2.67GHz * 2
物理コア数	12
論理コア数	24
Memory	132GB
OS	Fedora 16
JavaVM	1.6.0_39-b04
GHC	7.6.3

Warp 及び Cassandra, Jetty は表 2 のバージョンを利用した。

表 2　Warp と Cassandra, Jetty のバージョン

名前	バージョン
Warp	1.3.9
Cassandra	1.2.6
Jetty	6.1.26

サーバに並列に負荷をかけるクラスタの仕様を表 3 に示す。クラスタは最大で 45 台使用する。

6.3 実験結果

読み込みの実験結果を図 3、書き込みの実験結果を図 4 に示す。

Haskell 版および Java 版の Jungle は、ほぼ同程度の速度が出ていることが分かる。また、Cassandra と比較して、僅かながら Jungle が速く処理を終えている。

表 3　検証に利用するクラスタの仕様

名前	概要
CPU	Intel®Xeon®X5650 @2.67GHz
Memory	8GB
OS	CentOS 5.6
HyperVisor	VMWare ESXi

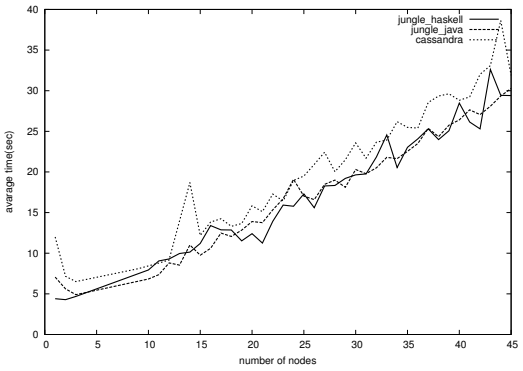

図 3　読み込みの実験結果

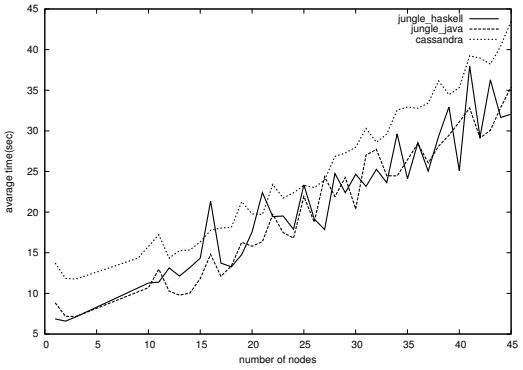

図 4　書き込みの実験結果

6.4 遅延評価

Haskell は遅延評価を行うが、書き込みの際に問題が生じる。何かしらの結果を表示するまで、簡約可能な式の状態で積まれたままとなる。その際メモリを消費し、効率のよい領域に入りきらないサイズになると非常に実行結果が遅くなる。

図 6 では、メモリ消費量を表している。実行中メ

モリ消費量は増えていくが、木の表示を行う 22 秒前後まで簡約化は行われない。表示を行った際に簡約化され、メモリ消費量が急激に下がっている。

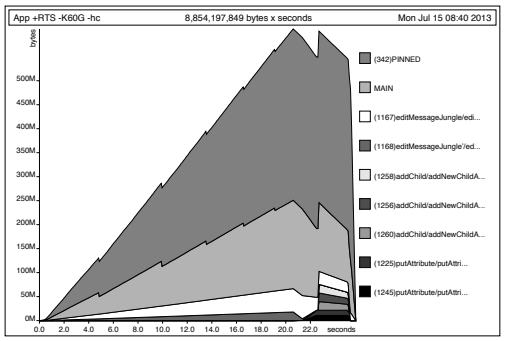

図 5　遅延評価のメモリ消費

図 4 の結果では、オプションで推奨されるヒープ領域のサイズを変更してある。推奨されるヒープ領域のサイズを変更しない場合の実験結果を図 6 に示す。

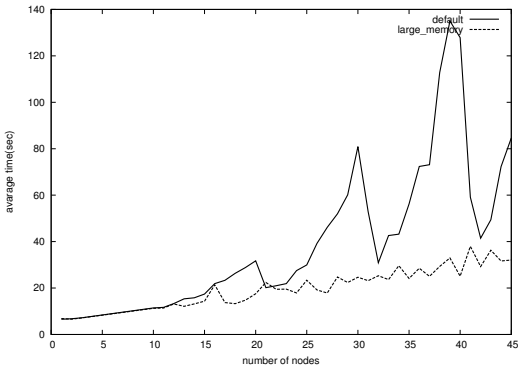

図 6　推奨されるヒープ領域のサイズを変更しない場合の実験結果

評価を行ったあとに実行時間がどのように変わるかを示すため実験方法を変更してある。実行時に書き込みだけではなく、クラスタ台数を変更する際に 10 台増やすごとに一度読み込みを挟むようにした。書き込みを繰り返すと実行時間が悪化し、読み込み後、急激に実行時間が下がる。読み込みの際には、数万回以上の書き込みを処理するため数秒から数十秒かかる。書き込みは、インクリメントしている値を書き込んでいるが順序などは正しく処理できている。

この問題を解決するために、全て遅延評価するのではなく適切な箇所で即時評価を行うことで領域効率を改善する必要がある。

6.5　並列処理

Haskell 版 Jungle では、並列実行に問題を抱えている。複数のスレッドが立ち上がり、並列実行していることは確認したが、シングルコアで実行した場合と比較して実行結果が遅くなる。図 3 や図 4 の結果では、Haskell 版はシングルコアで実行している。

並列に動かした場合の実験結果を図 7 に示す。

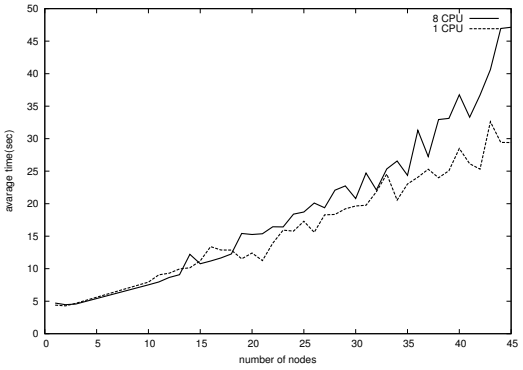

図 7　並列に動かした場合の実験結果

本研究のウェブアプリケーションとは別に、簡単な例題を並列で動かした場合でも実行速度の向上を確認することはできなかった。並列処理で速度向上を達成することは今後の課題である。

7　おわりに

7.1　本研究のまとめ

本研究では、Haskell による非破壊的木構造を用いた CMS の実装を行った。Haskell は生産性が高く、本システムの実装においても開発期間およびコード行数は非常に短くなった。

木構造データベース Jungle と HTTP サーバ Warp を用いて簡易掲示板システムを開発し、既存の Java の実装と同程度の性能を達成できた。

7.2 今後の課題

書き込みの際に、遅延評価のためにメモリを多く使用する問題がある。いくつかの式の評価を正格に切り替え、領域効率を向上しなければならない。

また、並列処理を行った際に実行速度が向上するよう再設計を行う必要がある。

現在の Jungle には木構造を永続化する仕組みが備わっておらず、実装しなければならない。

分散環境で Jungle を効率よく利用するために、木構造をマージする仕組みを実装する必要がある。マージにはお互いの木の情報が必要になる。どのようにマージすべきなのかは、ユーザが知っていると考えられるが、データベース間で過度の情報をやり取りを行うと負荷が上昇するおそれがある。どの程度の情報が必要であるのか検討しなければならない。

7.3 DEOS プロジェクト

2006 年に独立行政法人科学技術機構の CREST プログラムの 1 つとして始まったプロジェクトである。DEOS プロジェクトは、変化しつづける目的や環境の中でシステムを適切に対応させ、継続的にユーザが求めるサービスを提供することができるシステムの構築法を開発することを目標としている。DEOS プロセスにおいて、D-ADD (DEOS Agreement Description Database) [7] と呼ばれるデータベースがある。本論文で説明する 木構造データベース Jungle は、D-ADD への応用を目指しており、DEOS プロジェクトの一環として行なっている。

参 考 文 献

[1] The warp package. `http://hackage.haskell.org/package/warp`. [Online; accessed 19-July-2013].

[2] 玉城将士, 河野真治. Cassandra を使ったスケーラビリティのある cms の設計. 情報処理学会, March 2011.

[3] 玉城将士, 河野真治. Cassandra と非破壊的構造を用いた cms のスケーラビリティ検証環境の構築. 日本ソフトウェア科学会, August 2011.

[4] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, March 2003.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly avaliable key-value store.

[6] Fay Changand Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : A distributed storege system for structured data.

[7] Deos. `http://www.dependable-os.net/osddeos/data.html`. [Online; accessed 19-July-2013].

[8] 玉城将士, 河野真治. Cassandra を使った cms の pc クラスタを使ったスケーラビリティの検証. 日本ソフトウェア科学会, August 2010.

[9] Nancy Lynch and Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.