情報処理学会論文誌　プログラミング　Vol. 4　No. 3　1–15 (June 2011)

# Ruby用リアルタイムプロファイラの設計と実装

須永高浩[†1]　笹田耕一[†1]

Rubyは高い生産性を持つプログラミング言語である．これまでもプログラムにあるそれぞれのメソッドの実行時間を計測することができるプロファイラは整備されてきた．しかし，分析対象プログラムの実行と並行してリアルタイムプロファイリングが行えるツールが存在しなかった．そこで，本研究ではRubyで書かれたソフトウェアのプロファイリングをリアルタイムに行える実行時間プロファイラを開発した．本プロファイラは，対象となるRubyで記述されたプログラムの，メソッド単位での実行時間情報を取得することが可能である．この情報取得はネットワークで接続された別のホストからリアルタイムに行うことができる．さらに，それぞれの情報についてプログラムのエントリポイントからの完全なコールパスを関連付けてプロファイリングを行うことが可能であり，対象となるプログラムのパフォーマンスの傾向を詳細に把握することが可能である．本プロファイラは，プロファイリングモジュールとモニタプログラムに分離されており，プロファイリングモジュールで情報を収集し，モニタプログラムがその情報をユーザへリアルタイムに提供する．本稿では，本プロファイラの設計と実装について述べる．また，本プロファイラのオーバヘッドの評価を行い，その結果を述べる．実用的なプログラムによる評価では1.15倍程度の実行時間の増加となり，本プロファイラは実用的であるとの結論を得た．

# Design and Implementation of Real-time Performance Profiler for Ruby

Takahiro Sunaga[†1] and Koichi Sasada[†1]

Ruby is a programming language which has high productivity. Up to now, Ruby's performance profilers that are able to measure execution time of each method have been developed. However, there is no real-time profiler that is able to show the profiling result while a Ruby program is running. We developed a profiler that is able to profile programs written in Ruby language in real time. The profiler can get execution time information of each method in the target Ruby program. It is able to get the information from a foreign host connected by network in real time. Furthermore, the profiler can profile associating each information with the entire call-path from an entry point of the program. With this call-path informathion users can know the trend in program performance in detail. The profiler consists of two parts: a profiling module and a monitor program. The profiling module collects information, and the monitor program provides information for users in real time. In this paper, we describe the design and implementation of our proposed profiler. We also describe the result of overhead evaluation for the profiler. Execution time increases 1.15 times in a practical example. We conclude that our profiler is capable for practical use.

## 1. はじめに

プログラムの性能を評価する1つの指標として，実行時間がある．同等の仕事をするプログラムではより短い実行時間で終了することが好ましい．プログラムの実行時間を短くするために重要なツールとして実行時間プロファイラがある．実行時間プロファイラは一般に，プログラム全体の実行時間のみではなく各実行部分における実行時間を計測することができる．その情報を用いて，実行時間を最も消費している実行部分（ボトルネック）に対して集中的に対策を行うことで，効率的にプログラムの実行時間を縮めることが可能になる．

たとえば試験環境でWebアプリケーションを動作させているとき，ある特定の条件下でリクエストの処理の実行時間が長い，という問題があったとする．この場合，リアルタイムでないプロファイラであれば，再現方法を調査し，再現環境を構築したのち，プロファイリングを行いながら再現させる必要がある．

この例で，Webアプリケーションを実行し，問題発生後に後から対象のアプリケーションに接続し，プロファイリング情報をリアルタイムに閲覧できるプロファイラが利用できれば，性能上の問題が発生した時点でただちにボトルネックの調査が行えるため，問題を再現させるための手順を省略できる．またリクエストを送りながらプロファイル情報を閲覧することで，パフォーマンスの傾向をより詳細に知ることができる．結果，このようなプロファイラを用いればより容易に性能向上が行える可能性がある．

Ruby[1)]は高い生産性を持つプログラミング言語であり，世界中で広く使われているプログラミング言語の1つである．これまで，Rubyプログラムに対してそれぞれのメソッドの実行時間を計測することができる実行時間プロファイラは整備されてきた．Rubyに対する代表的なプロファイラとしてruby-prof[2)]がある．しかし，分析対象プログラムの実行と並

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

行してリアルタイムプロファイリングが行えるツールが Ruby には存在しなかった．

そこで本研究では，Ruby プログラムに対してリアルタイムにプロファイリングができるツールの設計と実装を行った．本プロファイラは，対象プログラムにおいて性能改善に必要な情報，特に実行時間を多く消費している実行部分（ボトルネック）の検出に必要な情報を取得し，それをユーザに提供する．また，本プロファイラは呼び出し元ごとにプロファイルをとるため，ボトルネックの場所の特定が容易である．

プロファイリングの手法としては，大きく分けて，イベントトレースとサンプリングの 2 つのアプローチがある．イベントトレースによる手法はそれぞれの実行単位の突入時と脱出時に時刻を測定し，その実行時間を計測する手法である．サンプリングによる手法は周期的に実行位置の取得を行い，サンプル数を計測することでそれぞれの実行単位のコストを計測する手法である．本プロファイラは，詳細なコンテキスト情報を取得するためにイベントトレースによる手法を基本に用いた．

本プロファイラは，実行開始後にユーザがいつでも対象プログラムからプロファイル情報を引き出せるようにプロファイル情報を Ruby インタプリタから取得するプロファイリングモジュールと，そこから情報を得てユーザに情報を表示するモニタプログラムの 2 つのプログラムから構成される．プロファイリングモジュールは Ruby 組み込みモジュールであり，取得対象の Ruby プログラムへ読み込ませて使用する．モニタプログラムは独立して動作するプログラムであり，プロファイリングモジュールから情報を受け取り，GUI によりユーザに情報を提供する．**図 1** は本プロファイラのモニタプログラムの動作画面例である．プロファイリングモジュールがあらかじめ読み込まれていれば，対象プログラムの実行中に後からモニタプログラムを接続することができる．プロファイリングモジュールとモニタプログラムの接続はネットワーク経由で行える．したがってプロファイリングモジュールとモニタプログラムを別のホストで実行させることができる．

本研究の貢献として，以下の 2 点をあげる．

- Ruby リアルタイムプロファイラの開発を行い，それを利用者へ提供する．利用者が各々の開発するソフトウェアの性能向上を図ることができる．さらにリアルタイムにプロファイリングが行えるようになるため，より容易かつ詳細にプロファイリングを行うことができる．
- リアルタイムプロファイラの設計と実装に関する知見の提供をする．本プロファイラはリアルタイムなプロファイルデータの共有をネットワーク越しに行うプロファイラである．そのようなプロファイラに関して詳細な設計・実装手順を示し，その評価結果を示すことで今後のプロファイラの開発に有効な知見を示す．

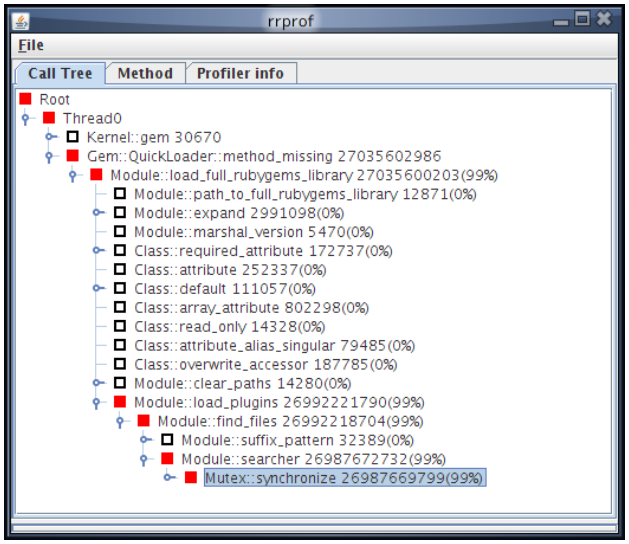


**図 1**　本プロファイラの実行例
Fig. 1　Example of our profiler.

以下，2 章で本プロファイラの要求分析について述べ，3 章で設計の概要を述べる．4 章から 6 章までで本プロファイラの設計と実装について詳細に述べる．7 章で本プロファイラの性能評価について述べ，8 章で関連研究，9 章で本稿のまとめを述べる．

## 2. 要 求 分 析

本章では，本プロファイラのユーザから見た要求分析について述べる．

### 2.1　本プロファイラの使用目的

本プロファイラは，対象プログラムのプロファイル情報をリアルタイムにユーザに提供するためのプロファイラである．本プロファイラで想定される実行環境は一般的な Web サーバやパーソナルコンピュータ等である．

ユーザはプロファイル情報を用いてソフトウェアを改良を行ったり，環境の改善等を行ったりすることで，効率的にプログラムの性能向上を図ることができる．また，プロファイル情報をリアルタイムに得ることができるため，ユーザはより容易にプロファイリングが行

える．

Ruby では，基本的な実行単位としてメソッドがある．本プロファイラではメソッドごとのプロファイル情報をとる．

### 2.2 必 要 要 件

本プロファイラの必要要件として以下のものをあげる．

（1） メソッドの総実行時間と自己実行時間の取得

ボトルネックを検出するためには，プログラムのなかで多く実行時間が消費されているメソッドを検出する必要がある．そこで，メソッドごとの総実行時間を取得する必要がある．またメソッドの実行時間が子呼び出しで消費されているのか，そのメソッド自体で消費されているのか区別することでより正確にボトルネックを把握することができる．そこで，総実行時間から子呼び出しの実行時間を減算した値である自己実行時間が調査できる必要がある．

（2） メソッドの実行回数の取得

あるメソッドが大きなコストを消費していることが分かったとき，そのボトルネックを解消するためには，その実行回数が多いのかメソッド単体での実行時間が長いのか判別する必要がある．これを判別するためにはそれぞれのメソッドの実行回数の情報が必要である．

（3） 呼び出しコンテキスト情報の取得

実際のアプリケーションやライブラリは階層化された実装がされていることが多く，実行時間は呼び出しコンテキストによって大きく左右されることがある[3]．そのようなプログラムのボトルネックの傾向を正確に把握するためには，プロファイル情報に対して，どのような順番でメソッドが呼ばれたかという呼び出されたコンテキスト情報を関連付けて調査する必要がある．

（4） 現在の実行位置の表示

性能上の問題が発生した際，プロファイル時に現在のプログラム上の実行位置を知ることで，素早くボトルネックを発見できる可能性がある．そこで，現在の実行位置をリアルタイムにユーザに表示する必要がある．

（5） プロファイリングモジュールとモニタプログラムの分離

プロファイル対象プログラムを動かしている環境では，プロファイル情報を GUI により表示できない場合がある．たとえば Web アプリケーションを動かすサーバ上では GUI が提供されていない場合が多い．そこで，プロファイリング情報を取得するプログラムであるプロファイリングモジュールと，その情報を表示するプログラムであるモニタプログラムに分離し，それぞれをネットワークで接続された別ホスト上で独立して実行させる必要がある．

（6） 低いオーバヘッド

プロファイラの利用は一般に対象プログラムに対して時間的オーバヘッド（処理性能の悪化）や記憶領域オーバヘッド（使用記憶領域量の増加）をもたらす．

時間的オーバヘッドが大きいと正確な情報が得られない可能性がある．さらに，時間的オーバヘッドによりプログラムの実行時間が大幅に増大するとプロファイルに時間がかかり，ユーザの負担が増加する．記憶領域オーバヘッドによりマシンが持つ記憶領域を消費し尽くしてしまう場合，プロファイリングが不可能になったり，プログラムの実行時間が大幅に増大したりする可能性がある．また，本プロファイラでは，必要要件 (5) で述べたようにネットワークによる接続を行う．このとき，転送量がネットワークで対応しきれなくなると，プロファイリングが不可能になる可能性がある．したがってこれらのオーバヘッドをより低く抑える必要がある．

ただし，記憶領域オーバヘッドと転送量によるオーバヘッドは，現実的に対応できる範囲内であればよい．本プロファイラでは，2.1 節で述べたように最低でも一般的なパーソナルコンピュータ程度の性能があるコンピュータでの実行を想定しており，たとえば使用領域の消費量が 5 MB である場合と 10 MB である場合では近年のコンピュータでは大きな差はないと考えられる．これに対し，時間的オーバヘッドはより少ない方が正確なプロファイル情報が取得できる．したがって，時間的オーバヘッドの削減が最も重要である．

## 3. 設　　計

本章では，本プロファイラの設計について述べる．

### 3.1 プロファイラの構成

本プロファイラの構成を**図 2** に示す．必要要件 (5) で述べたように，本プロファイラはプロファイリングモジュールとモニタプログラムの 2 つの部分に分離されている．

プロファイリングモジュールは，Ruby プログラムのプロファイル情報を取得し，モニタプログラムに対してその情報を送信する Ruby 拡張モジュールである．プロファイリング対象となる Ruby プログラムから読み込むことで使用する．プロファイリングモジュールは，プロファイリング対象と同一プロセスで動作する．

モニタプログラムは，プロファイリングモジュールからプロファイル情報を取得しそれをユーザに対して表示するプログラムである．モニタプログラムは一定間隔おきにプロファイリングモジュールと通信を行いプロファイル情報を得て，リアルタイムにその情報を表示

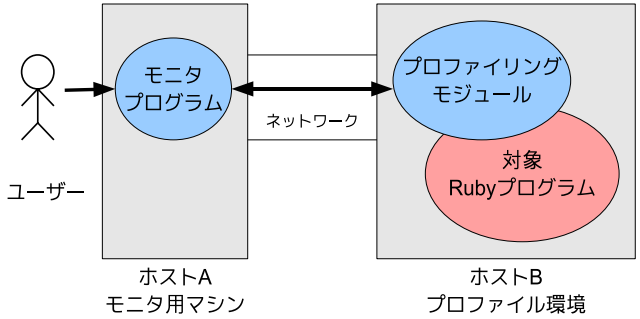


**図 2**　本プロファイラの構成
Fig. 2　Structure diagram of our profiler.

する．

プロファイリングモジュールはモニタプログラムが接続されていなくても単独で動作可能であり，ネットワーク経由で別ホストから接続することができる．後からいつでもモニタプログラムを接続し，情報の取得が行える．

### 3.2　プロファイリングの流れ

本プロファイラの処理の流れを**図 3** に示す．この図を用いて処理の流れを以下で説明する．プロファイリングモジュールは対象となる Ruby プログラムで読み込まれると，そのプログラムの実行と並行して情報送信のための送信スレッドを開始し，モニタプログラムの接続の待ち受けを開始する（図 3 [1]）．さらに，Ruby プログラムの実行を行いながら必要要件 (1)，(2) で述べられているプロファイル情報の取得を行う．プロファイル情報は，モニタプログラムから接続されるまで送信されず，プロファイリングモジュールに蓄積されつづける．

モニタプログラムはユーザから指定された接続先にあるプロファイリングモジュールへ接続し，プロファイリング開始のメッセージを送る（図 3 [2]〜[3]）．プロファイリングモジュールはプロファイリング開始のメッセージを受け取ると，プロファイル情報の送信を行う準備をする．その後，モニタプログラムはユーザによって指定された情報取得間隔でプロファイリングモジュールに情報要求メッセージを送り（図 3 [4]），プロファイリングモジュールはそれに応答する形で蓄積されていたプロファイル情報を送り返す（図 3 [5]）．モニタプログラムは，送られてきたプロファイリング情報をユーザに表示する（図 3 [6]）．

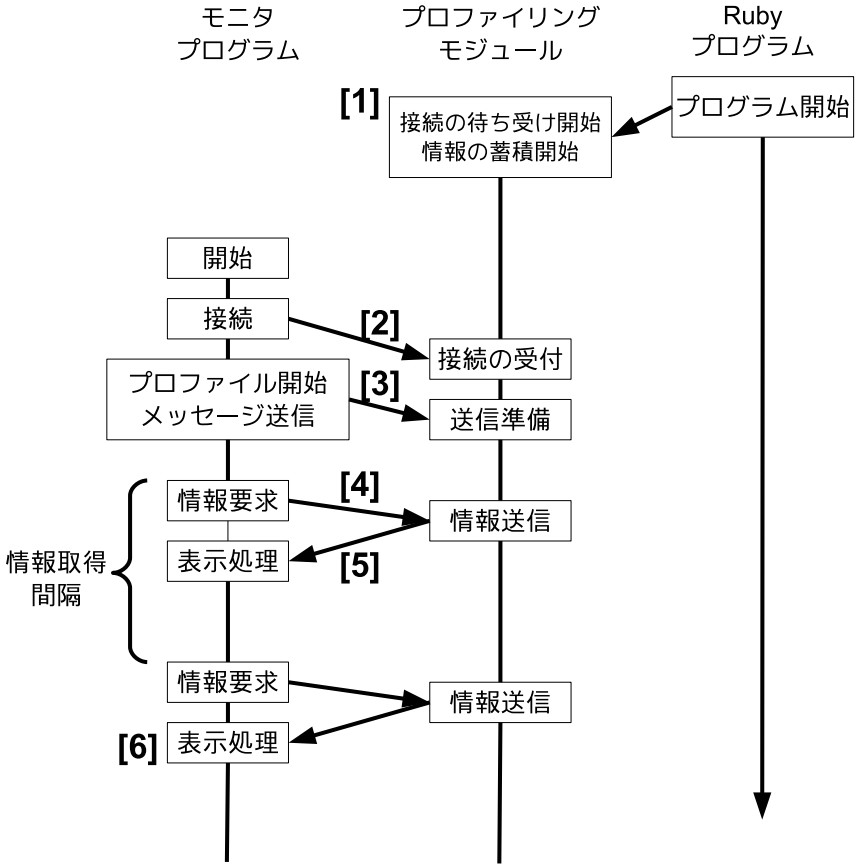


**図 3**　プロファイリングの流れ
Fig. 3　Profiling flow.

### 3.3　呼び出しコンテキスト情報

必要要件 (3) で述べたように，プロファイル情報を呼び出しコンテキストの情報に関連付けて蓄積する必要がある．そのような情報を蓄積する方法としてコールグラフが一般的に用いられるが，コールグラフでは十分な情報が得られないケースがある[4]．

そこで必要要件 (3) を満たすため，本プロファイラではプロファイル情報蓄積のための

```
def a()
  b()
  a()
end

def b()
end

def c()
end

def d()
  b()
  a()
  a()
end

d()
a()
```

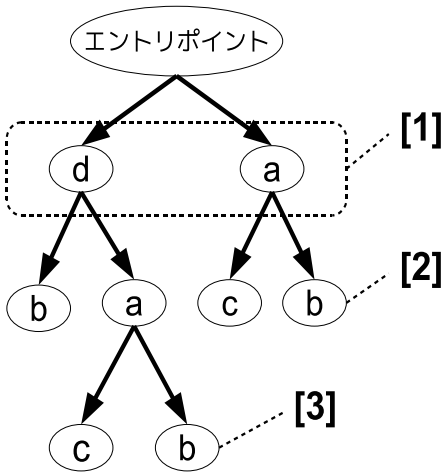


**図 4**　CCT（Calling Context Tree）の例
Fig. 4　An example of the CCT (Calling Context Tree).

データ構造として CCT（Calling Context Tree）[5] を使用した．CCT とは，それぞれのメソッドの呼び出しごとに情報を保持する構造であり，プログラムのエントリポイントをルートノードとしたツリー構造になる．「呼び出し」を呼び出し元と呼び出し先を結ぶエッジとし，呼び出し先のメソッドをそれぞれ別のノードとする構造である．このとき，それぞれのノードをコールノードと呼ぶ．

**図 4** は CCT の例である．図の左側にあるプログラムを CCT として表現すると右側のようなツリー状の構造になる．図 4 のプログラムでは，プログラムのエントリポイントからメソッド d と a が呼ばれる．したがって，エントリポイントの子要素に d と a に対応するノード（図 4 [1] で示した部分）が作成される．ここで重要であるのは，呼び出し元に対応するコールノードによって情報が区別されることである．たとえば，図 4 の [2] と [3] は同じメソッド b を示すが，この 2 つは呼び出し元が「エントリポイント→ a」と「エントリポイント→ d → a」となり異なるため，CCT 上の異なるノードとして記録され，情報が区別される．

ただし本プロファイラの実装では，1 つのメソッドから同一名・同一クラスのメソッドが複数回呼ばれた場合，それらはすべて同じ呼び出しとして扱う．つまり，ある 1 つのコールノードについて，同一名・同一クラスのメソッドに対応する子コールノードはただ 1 つとなる．また，本プロファイラでは再帰呼び出しを行った場合でも循環グラフを作ることはせず，無制限に子コールノードを増やしていく．

本プロファイラではこのように，それぞれの情報を呼び出しコンテキスト情報と関連付けて情報を蓄積することができる．結果としてボトルネックをより正確に把握することができる．

### 3.4　現在位置の取得

本プロファイラは必要要件 (4) を満たすため，スレッドごとに現在実行しているコールノード ID を取得し，プロファイル情報の送信時に一緒にモニタプログラムへ送信を行う．モニタプログラムはこの情報を利用して現在実行位置の表示を行う．

### 3.5　低いオーバヘッドのための設計方針

必要要件 (6) のうち，時間的オーバヘッドに対する要件を満たすため，次のような設計方針を立てた．

- 記憶領域の削減と時間的オーバヘッドの削減とのトレードオフが発生した場合，時間的オーバヘッドの削減を優先する．
- 情報の集約処理や計算処理はできるだけモニタプログラム側で行う．これはプロファイリングモジュール側での処理の負担を減らし，よりオーバヘッドの削減を見込めるためである．また，集約方法や計算方法をモニタプログラム側の変更のみで柔軟に追加や変更できるため，実装の工数が削減されることが見込まれる．
- 冗長なデータ送信を避ける．また，情報の蓄積は送信形式と同様の形式で行う．これは，情報送信による不要な負荷を削減するためである．

記憶領域オーバヘッドと転送量によるオーバヘッドは 7 章で見るように実用的な範囲内に収まる．

## 4.　プロファイル情報の取得と蓄積

本章では本プロファイラでのプロファイル情報の取得方法と蓄積方法について詳細に示す．

### 4.1　情報の取得

本プロファイラでは，メソッドごとに突入時と脱出時に呼ばれるイベントフック関数を用

意し，そのイベントフック関数内で情報の取得処理を行う．これは，Ruby の公開 API である `rb_add_event_hook` を用いた．

実行回数は，カウンタをコールノードごとに保持し，それぞれのメソッドへの突入時にインクリメントすることで記録する．

メソッドの実行時間は，イベントフック関数内で実行開始時刻と実行終了時刻を取得しその値から算出する．時刻の取得方法には以下の 2 つの機能を実装した．

**直接モード**　Ruby の各実行スレッド自身が直接，OS の時刻を取得するモードである．時刻取得には POSIX API である `clock_gettime` を利用する．メソッドの突入と脱出のたびにシステムコールを発行するため，実行取得コストが高いと考えられる．

**間接モード**　一定時間ごとに増加する時刻カウンタを用意し，Ruby の各スレッドはそのカウンタから値を読み出す．整数値 1 つをコピーするだけの処理であるため，直接モードに対してオーバヘッドが低減されると考えられる．本モードでは，時刻カウンタの実現のため専用のタイマスレッドを用意する．このスレッドは，一定間隔ごとにカウンタをインクリメントしていく．

直接モードでは完全にイベントトレース型のプロファイラと同一である．間接モードは，サンプリングは行っていないが，時刻をサンプル数，タイマスレッドのインクリメント処理をサンプルイベントととらえたとき，得られる結果はサンプリングプロファイラと同等である．サンプリングプロファイラは，定期的にサンプルイベントを発生させ，そのサンプルイベントが発生した時点で処理があるメソッドのサンプルカウンタを増加させる．つまり，それぞれのメソッドごとに実行中に起こったサンプルイベントをカウントしている．本プロファイラの間接モードでは，それぞれのメソッドにおいて実行開始時点での時刻と，実行終了時での時刻の差をとるため，その値はタイマスレッドのインクリメント処理が行われた回数となる．つまりそれぞれのメソッドごとに実行中に起こったインクリメント処理の回数をカウントしていることとなるため，時間計測の精度に関してはサンプリングプロファイラと同等の結果が得られる．

### 4.2　情報の蓄積

プロファイリングモジュールは，取得した情報を内部に一時的に蓄積し，情報送信時にその情報を送信する．本節では，その情報の蓄積方法について示す．

#### 4.2.1　プロファイル情報

3.3 節で述べたように，本プロファイラでは呼び出しコンテキストごとにプロファイル情報の区別を行う必要があるため，CCT による情報蓄積を行う．

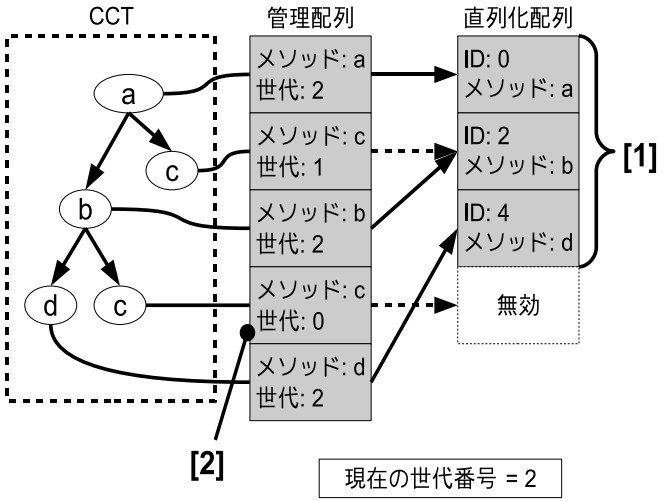


**図 5**　本プロファイラの情報蓄積の例
Fig. 5　An example of accumulating information.

また，3.5 節で述べたように，時間的オーバヘッドが低くなるような設計をする必要がある．本プロファイラでは，時間的オーバヘッドを低くするため，送信が必要な情報とプロファイリングモジュール内部でのみ必要な情報に分けて情報蓄積を行う．送信が必要な情報の蓄積には，情報送信時の直列化が不要であるデータ構造を使用する．また，すでに送信した情報を区別し無駄な再送を避けるため世代番号による情報管理を行う．

**図 5** は，本プロファイラの情報蓄積の例を図示したものである．この図を用いて CCT による情報蓄積の方法を示す．

本プロファイラでは，プロファイリングモジュール自身が必要とする情報を格納する管理配列と，送信する必要がある情報を格納する直列化配列の 2 種類の配列を用意する．それぞれに含まれる情報の詳細は 4.2.2 項で説明する．直列化配列はポインタ値等を含まず，記録したデータ表現をそのまま送信できる．情報送信の際には，図 5 の [1] で示した部分が送信され，その後直列化配列はすべて無効化される．

情報が有効であるかを示すため，情報の世代番号による管理を行う．プロファイラはスレッドごとに現在の世代番号を記録する．この世代番号は情報が送信されるたびにインクリメントされる．また，管理配列のそれぞれの要素ごとに現在指している情報の世代番号を記

録しておく．図 5 では，現在の世代番号は 2 である．たとえば図 5 の [2] で示した要素は，世代番号が 2 ではないため，古い情報を指していると見なされる．

送信時には現在の世代番号が 3 へ更新される．したがって管理配列が指す情報はすべて古い情報となる．このように情報の管理を行うことで，直列化配列には未送信の情報のみが有効な情報として存在することになる．また，送信不要な情報は管理配列側に記録するため，情報送信時には直列化処理を行わずにそのまま直列化配列を送信することができる．

#### 4.2.2　プロファイル情報の詳細

プロファイル情報を格納するそれぞれの配列について詳細に示す．

管理配列はプロファイリングモジュール自身が必要とする情報や，効率化のために必要な情報が含まれる配列である．具体的には以下の情報が含まれる．ここで，コールノード ID とは特定のコールノードを示す，CCT 内で一意な ID である．

- メソッド ID やクラス ID
- 親コールノード ID
- 対応する直列化配列の要素番号
- 世代番号
- 子コールノード ID のハッシュテーブル

コールノード ID は管理配列内のそれぞれの構造体には含まれないが，管理配列の要素番号がコールノード ID となるように格納されるため，コールノード ID によるインデックスアクセスが可能となっている．また，子コールノード ID のハッシュテーブルは，メソッド ID とクラス ID をキー，コールノード ID を値としたハッシュテーブルである．

管理配列の要素はコールノードごとに 1 つ作成され，すべての要素はプログラム終了時まで破棄されない．これは，コールノードとコールノード ID の対応を維持し，モニタプログラム での情報受信時にどのノードの情報かを識別するためである．

直列化配列は，プロファイリングプログラムからモニタプログラムへ送信する必要がある情報が含まれている配列である．具体的には以下の情報が含まれる．

- コールノード ID
- 親コールノード ID
- メソッド ID やクラス ID
- 実行回数カウンタ
- 総時間情報
- 子時間情報

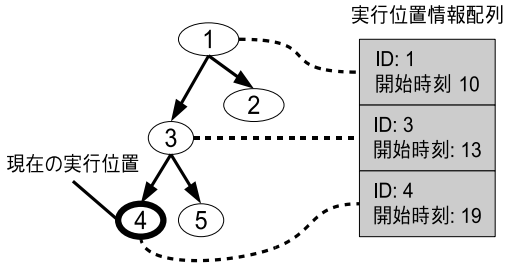


**図 6**　実行位置情報配列の例
Fig. 6　An example of run places array.

直列化配列の要素は送信後に破棄されるため，要素インデックスとコールノード ID は対応しない．

子実行時間は子のコールノードで消費された時間の合計で，総実行時間から子実行時間を減ずることで自己実行時間が算出できる．

それぞれの配列の要素は必ず 1 つのコールノードに関連付けられている．また，それぞれの配列は Ruby スレッドごとに 1 つずつ保持する．

#### 4.2.3　実行位置情報配列

本プロファイラは，Ruby プログラムの実行スレッドごとに 1 つずつの実行位置情報配列を保持している．

実行位置情報配列とは，Ruby プログラムのエントリポイントからのスタック情報を記録している配列である．配列の先頭がエントリポイントとなる．配列のそれぞれの要素には，対応するメソッドの開始時刻とノード ID が含まれている．

この実行位置情報配列は，ユーザに対して現在の実行位置を示すために使用される．また，開始時刻は，現在実行中のそれぞれのノードの実行時間を知るために必要である．本プロファイラは実行時間を実行終了時に計算するため，実行が終了するまで正確な実行時間を知ることができないため，この開始時刻の情報が必要になる．

**図 6** は実行位置情報配列の例を示す．この例のように，現在の実行されているコールノードから，エントリポイントに対応するノードまでのそれぞれのノードの ID と開始時刻が記録されている．

#### 4.2.4　情報蓄積の流れ

情報蓄積の手続きについて示す．ここで，「現在のコールノード ID」とは現在実行してい

```
M: 管理配列（与えられる値）
S: 直列化配列（与えられる値）
T: 実行位置情報配列（与えられる値）
i': 突入元のコールノード ID（与えられる値）
i: 突入先のコールノード ID（処理中に計算）
j: 対応する直列化配列の番号（処理中に計算）

if (M_i' の子ノードハッシュテーブルに突入先に該当するものがある) then
    i ← 該当したノードの ID
else
    M に突入先に対応する要素を追加
    i ← 追加した要素の番号 (これを突入先のコールノード ID とする)
    M_i' の子ノードハッシュテーブルに突入先に該当するものを追加する
end
S への参照をロックする
if (M_i の直列化配列の世代番号 = 現在の世代番号) then
    j ← M_i の直列化配列の要素番号
else
    S に要素を追加する
    j ← 追加した要素の番号
    M_i の直列化配列の要素番号 ← 追加した要素の番号
    M_i の世代番号 ← 現在の世代番号
end
S への参照をアンロックする
S_j の実行回数カウンタをインクリメント
T への参照をロックする
T の終端に要素を追加し，その要素にメソッドの開始時刻を記録する
T への参照をアンロックする
```

**図 7**　擬似言語によるメソッド突入時の処理手順
Fig. 7　Procedure for entering methods in pseudo-language.

```
i: 脱出元のコールノード ID（与えられる値）
i': 脱出先 (呼び出し元) のコールノード ID（処理中に計算）

S への参照をロックする
if (M_i の直列化配列の世代番号 = 現在の世代番号) then
    j ← M_i の直列化配列の要素番号
else
    S に要素を追加する
    j ← 追加した要素の番号
    M_i の直列化配列の要素番号 ← 追加した要素の番号
    M_i の世代番号 ← 現在の世代番号
end
S への参照をアンロックする
t ← (現在時刻) − (T の終端要素に記録された開始時刻)
S_j の総実行時間に t を加算
T への参照をロックする
T の終端から要素を一つ削除
T への参照をアンロックする
i' ← M_i の親コールノード ID
S_i' の子実行時間に t を加算
```

**図 8**　擬似言語によるメソッド脱出時の処理手順
Fig. 8　Procedure for exiting methods in pseudo-language.

る Ruby プログラム上のメソッドに対応するコールノードの ID を示し，「現在の管理配列要素」「現在の直列化配列要素」とはそれぞれ現在のコールノード ID に対応する管理配列・直列化配列の要素を示す．

**図 7** はメソッド突入時の処理を擬似言語で示したものである．ここで，$M_i$ と $S_i$ はそれぞれ $M$ の第 $i$ 要素，$S$ の第 $i$ 要素を示す．蓄積先のノード検索処理を行った後，プロファイリングに必要な情報を記録するという手順で処理を行う．ノードが存在しない場合，それぞれの配列に対応する要素が追加される．

次に，メソッドから脱出しようとしたときの処理手順を**図 8** に示す．この処理では，現

在時刻から実行位置情報配列にある対象コールノードの開始時刻記録を減ずることで実行時間の算出を行っている．また，脱出先のコールノードの子実行時間に計算した実行時間を加算することで，子実行時間の記録をしている．

## 5. 情報のやりとり

### 5.1 情報の送信

本プロファイラでの情報のやりとりは，モニタプログラムがプロファイリングモジュールに対して情報要求等のリクエストを送りプロファイリングモジュールがそれに対して応答を返すという手順で行う．たとえばプロファイル情報は，モニタプログラム側から定期的に情報のリクエストを送り，プロファイリングモジュールがそれに応答するという形をとる．プロファイリングモジュールは自発的にメッセージを送ることはない．

モニタプログラムへの情報の送信は，Ruby プログラムの実行とは別のスレッドである送信スレッドによって行われる．以下でこのスレッドで行われることを記述する．

#### 5.1.1 送信用バッファ

本プロファイラは性能低下を避けるため，ダブルバッファリングを行っている．

**図 9** は，それぞれのスレッドが持つ配列と，送信時の情報の入れ替えを示す．すでに述べたように，Ruby プログラムで実行されているスレッドごとに 1 つの直列化配列と実行位置情報配列を持つ．送信スレッドはさらに，送信用バッファとして直列化配列と実行位置情報配列をそれぞれ 1 つ所有する，これらの配列を送信用と蓄積用とで順次入れ替えていき，情報を送信していく．情報の送信は直列化配列と実行位置情報配列を順次そのまま送信することで実現できる．スレッドごとに順次入れ替え，そのつど送信していけばよいため，送信用のバッファは 1 つで十分である．

送信スレッドは，モニタからの定期的な情報要求メッセージを受け取ると Ruby プログラムの各スレッドが所持する直列化配列と実行位置情報配列に対して順次以下の手続きを行う．ここでスレッド T を処理対象の Ruby プログラム上のスレッドとする．

(1)　スレッド T が所有する直列化配列と実行位置情報配列への参照をロック
(2)　送信スレッドが所有する実行位置情報配列の長さをスレッド T のそれと同一の長さにし，要素をすべて「無効」とマーク
(3)　送信スレッドが所有する直列化配列と実行位置情報配列をスレッド T のそれと交換
(4)　世代番号をインクリメント
(5)　スレッド T が所有する直列化配列と実行位置情報配列への参照をアンロック

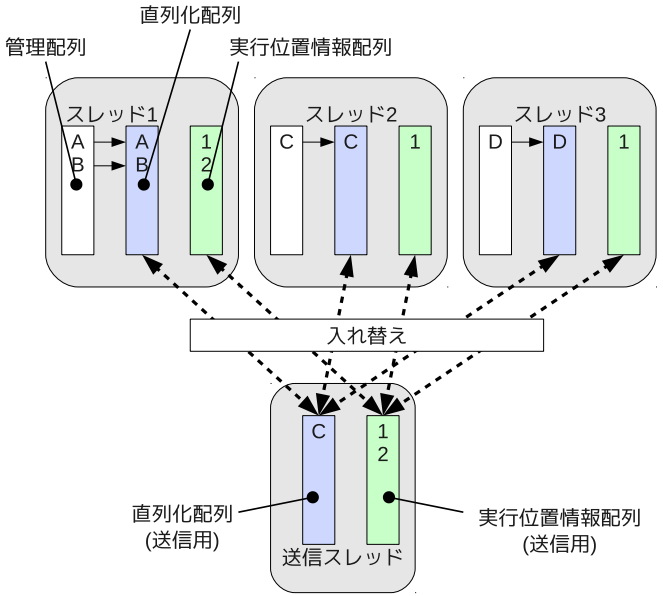


**図 9**　送信時のバッファの入れ替え
Fig. 9　Swapping buffer for receiving.

(6)　送信スレッドが所有する直列化配列と実行位置情報配列をモニタプログラムへ送信

以上の処理をすべてのスレッドに対して行うことで情報が蓄積されているすべてのバッファをモニタプログラムへ送信することができる．

なお，送信される実行位置情報配列の内部に「無効」とマークされた要素が入る可能性がある．その場合，モニタプログラムは前回に送信された実行位置情報配列から要素内容を推定する．

### 5.2 プロトコル

プロファイリングモジュールとモニタプログラムの間では，独自のプロトコルでメッセージをやりとりし，通信を行う．

**図 10** は，メッセージの構造を示す．

それぞれのメッセージの先頭にはメッセージタイプとメッセージサイズが記録されている．メッセージタイプはメッセージに含まれている内容がどのようなものかを示す数値である．

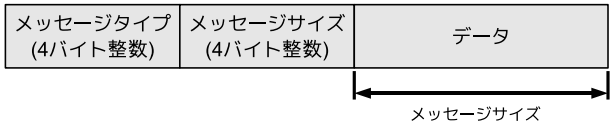


**図 10**　本プロファイラがやりとりするメッセージの構造
Fig. 10　The structure of message for this profiler.

本章ではその独自プロトコルについてどのような情報がやりとりされるかを示す．

#### 5.2.1　構造体オフセット情報

プロファイリングモジュールは，プロファイル開始メッセージを受け取ると，その応答と一緒に構造体のサイズやメンバのオフセット等の情報を送信する．

本プロファイラは，コールノード情報や実行位置情報配列を C 言語の構造体としてそのまま送信する．モニタプログラムに対して，それらの構造体のサイズやそれぞれのメンバのオフセットを知らせる必要があるため，この情報を送信する．

#### 5.2.2　プロファイル情報

本プロファイラでは直列化配列と実行位置情報配列をそれぞれ 1 つのメッセージとしてそのままモニタプログラムに対して送信する．モニタプログラムは 5.2.1 項で述べた構造体オフセット情報によってそれぞれの情報の位置を知ることができる．直列化配列にはプロファイリングモジュールが蓄積したプロファイル情報が含まれるため，そこから各ノードの実行時間情報や実行回数等を知ることができる．

#### 5.2.3　スタック情報

現在の実行位置を表示するためにスタックの情報が含まれる実行位置情報配列を送信する．これにより，モニタプログラムは実行中のノードを表示することができる．4.2.3 項で述べたように，モニタプログラムはこの情報を用いてどのノードが実行中であるかを表示する．また実行中のノードの現在の実行時間の算出を行う．

#### 5.2.4　名 前 取 得

モニタプログラムはユーザに表示するためクラス ID やメソッド ID からクラス名やメソッド名を知る必要がある．

モニタプログラムはクラス名やメソッド名を調べるために内部に名前表を保持しており，プロファイル情報内に名前表に存在しない未知のクラス ID やメソッド ID が現れた場合，プロファイリングモジュールに対して名前情報要求メッセージを送信する．プロファイリングモジュールはモニタプログラムからの要求がくると名前を調べ，モニタプログラムへ名前情報を応答する．

## 6. 表　　示

本プロファイラの情報表示のためモニタプログラムの実装を行った．ただし今回は参考実装として実装を行った．プロファイリングモジュールとの独立性は高いので，本プロファイラ用のプロトコルを用いることでモニタプログラムは本実装以外にも対応可能である．

今回はモニタプログラムは Java と，その GUI ライブラリである Swing を用いて作成した．表示方法としては，メソッドごとに集約し一覧表示を行うメソッドリストビューと，CCT 構造をそのままツリービューに表示する CCT ビューの実装を行った．

**図 11** はメソッドリストビューの表示例である．本ビューは同一のメソッドのコールノードは 1 行に集約されて表示される．リストのカラムヘッダをクリックすることで各項目ごとにソートを行うことができる．このビューを用いることで，ユーザはメソッドごとのプロファイル情報を見ることができる．

**図 12** は CCT ビューの表示例である．ツリービューのそれぞれのノードアイテムが 1 つのコールノードに対応している．ノードアイテムの赤い四角のアイコンは現在対象ノードが実行されていることを示す．ノードアイテムのラベルには，対応するノードのクラス名とメソッド名が表示され，その後に総実行時間，実行時間比率（親の総実行時間に対する自分の総実行時間の割合）が表示される．このビューではコールノードごとに区別されるため，ユーザは呼び出しコンテキストに依存して発生するパフォーマンスの傾向を見ることができる．

## 7. 性 能 評 価

2.2 節の必要要件 (6) で述べられているオーバヘッドに対する要求が満たされているか確認するため，本プロファイラに対して，プロファイリングモジュールのオーバヘッドの評価を行った．

また，今後の時間的オーバヘッドの削減のため，本プロファイラ自身のボトルネックの評価を行った．

### 7.1　評 価 環 境

すべての評価は以下に述べる同一の環境で行った．評価は 2 台のマシンで行い，1 台を対象マシンとし，検証対象 Ruby プログラムとプロファイリングモジュールを実行させる．もう片方をモニタ用マシンとしてモニタプログラムを実行させた．対象マシンは Intel Core

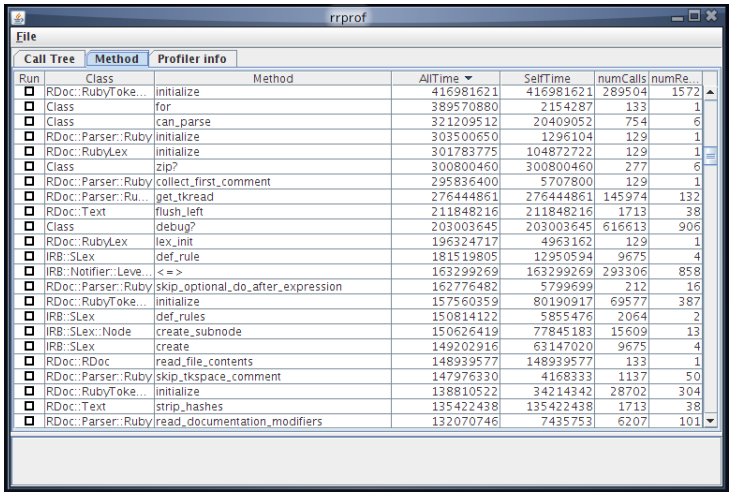


**図 11**　モニタプログラムのメソッドリストビューの表示例
Fig. 11　Example of method list view of monitor program.

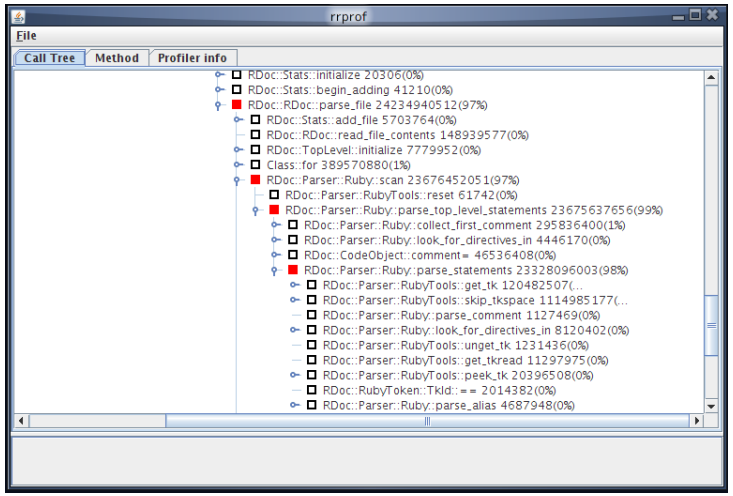


**図 12**　モニタプログラムの CCT ビューの表示例
Fig. 12　Example of CCT view of monitor program.

i5 CPU 650（3.20 GHz）と 4 GB のメモリで，OS は Linux Kernel 2.6.34（64 bit）である．Ruby のバージョンは 1.9.3dev（リビジョン 28556）を用いている．モニタ用マシンは Intel Core 2 Duo CPU E8500（3.16 GHz）と 4 GB のメモリ，OS は Linux Kernel 2.6.34（32 bit）である．

本評価で使用するプログラムは，rdoc，webrick，test-all，竹内関数プログラムの 4 種類である．rdoc は Ruby 用のライブラリドキュメントをソースコードから自動生成するプログラムであり，複雑なアプリケーションである．すべての評価において，Ruby の標準ライブラリディレクトリ内にあるすべてのライブラリを対象として実行する．webrick は，Ruby に標準添付された Web サーバライブラリであり，Web アプリケーションの開発等でよく用いられる．test-all は Ruby に標準添付のすべてのテストプログラムを走らせるものである．600 ほどの Ruby スクリプトからなるものであり，規模が大きく大量のメソッドが存在するため，コールノードが大量に作成される[⋆1]．竹内関数プログラムは，本評価のために作成した，竹内関数を実行する短い Ruby スクリプトである．実行すると，メソッド呼び出しが大量に行われるため，本プロファイラによるボトルネックが大きくなると考えられる．

⋆1 ただし，本プロファイラの実装の何らかのバグにより，test-all 実行中にはモニタプログラムの接続を行うことができなかった．

### 7.2　記憶領域オーバヘッド

複雑なプログラムに対して本プロファイラによるプロファイリングを行ったときの，必要な増加記憶領域量の評価を行った．この評価は，プロファイリングモジュールに使用記憶領域量を表示する機能を実装することで行った．この値は，プロファイリングモジュールが保有する管理配列と直列化配列の必要総要素数とそれぞれの要素 1 つあたりのサイズ，またそれぞれの管理配列の要素が保有するハッシュテーブルのサイズから算出されたものである．

評価は rdoc と webrick，test-all で行った．

**表 1** は，本評価の結果である．使用記憶領域量はコールノード数に管理配列と直列化配列の要素のサイズ（120 バイト）を乗じたものである．ノード情報の記録よる記憶領域の消費は rdoc で 13.29 MB，webrick で 0.19 MB 程度であることが分かった．本評価環境において，rdoc 単独で実行させると，メモリ消費量は 100 MB を超える．この値と比較して，本プロファイラの使用記憶領域は小さいものであり，プログラムの実行に与える影響は小さいと考えられる．

**表 1**　使用記憶領域の評価結果
Table 1　The evaluation result of memory usage.

| プログラム | 使用記憶領域量（MB） |
|---|---|
| rdoc | 13.29 |
| webrick | 0.19 |
| test-all | 425.40 |

また，極端に大量にコールノードが作られる例である test-all でも 425 MB ほどであり，近年の一般的なマシンであれば十分に対応できる使用記憶領域量である．

ただし，意図的に次々に新しいコールノードが作成されるようなプログラムを実行させた場合，本プロファイラは無制限にコールノードを作成してしまい記憶領域不足に陥るため，正常にプロファイリングを行うことはできない．そのような例として，新しいメソッド名のメソッドを動的に作成し呼び出す，といったことを繰り返すプログラム等が考えられるが，実際のプログラムにはあまり存在しないと考えられる．

### 7.3 転　送　量

プロファイル情報をモニタプログラムに転送する際にどの程度の量を転送する必要であるか計測を行った．この計測は使用記憶領域の計測時と同様にモニタプログラムにプロファイリングモジュールからの入力通信量と，出力通信量の表示を行う機能を実装して行った．両数値とも値はバイト数で示され，入力通信量はいままで送られてきたすべての情報の総容量を示し，出力通信量はプロファイリングモジュールへの要求や指示メッセージの合計サイズを示す．本評価は rdoc で行った．情報取得間隔は 50 ミリ秒，100 ミリ秒，500 ミリ秒，1 秒，5 秒と変化させ行った．入出力サイズは情報取得のタイミングで数値が異なる可能性があるため，計測はすべて 5 回行い，その平均値を数値として採用した．

**図 13** の 2 つのグラフは，情報取得間隔によるモニタプログラムの通信量の変化を示す．縦軸はプログラム開始時から終了時までの総通信量（入力サイズは MB，出力サイズは KB）で，横軸は情報取得間隔を示す．

情報取得間隔が短いほど通信量が多くなり，間隔が 50（ミリ秒）のとき入力サイズは 40 MB 程度になることが分かった．rdoc の実行時間は 100 秒程度であり，その値から計算すると，実行時間中の転送量は 50 ms で平均 3.2 Mbps 程度である．近年のコンピュータネットワークでは 100 Mbps 以上の帯域を持つネットワーク構成が一般的であり，本プロファイラの帯域に対して十分対応できると考えられる．

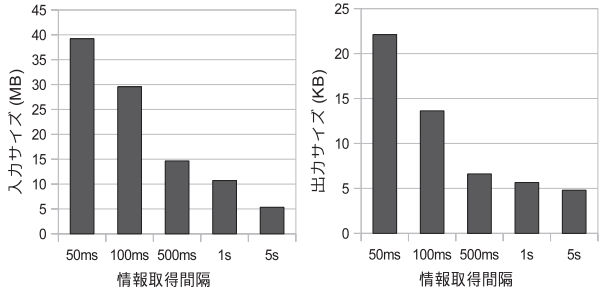


**図 13**　モニタプログラムの入出力通信量
Fig. 13　Input/Output communication volume for monitor program.

### 7.4 時間的オーバヘッド

Ruby のプログラムの実行時間に対して，本プロファイラによる実行時間の増大量（時間的オーバヘッド）の計測を行った．計測は以下の 3 つの状態で行った．

(1)　本プロファイラを使用し，かつモニタプログラムを接続する．情報取得間隔は 100 ミリ秒，時刻取得方法は直接モードとする．
(2)　本プロファイラを使用するが，モニタプログラムは接続しない．時刻取得は直接モードとする．
(3)　本プロファイラを使用し，かつモニタプログラムを接続する．情報取得間隔は 100 ミリ秒，時刻取得方法は間接モードとする．
(4)　プロファイラを使用しない

この計測結果のうち，(1) と (4) の実行時間差を求めることで，本プロファイラによるオーバヘッドを調べることができる．(1) と (3) の差を求めることで，時刻取得方法によるオーバヘッドの変化を調べることができる．また，(2) の場合であっても情報の蓄積は (1) と同様に行われるため，この時間差を求めることで，プロファイリングモジュールがモニタプログラムに送ることによって発生するオーバヘッドを調べることができる．評価は rdoc と竹内関数で行った．計測は各 5 回行い，その数値の平均値を結果として採用した．

**表 2** は本評価の結果を示す表である．直接・増加率の列は (1) を (4) で割った値であり，プロファイラを直接モードで使用したときの実行時間が増加する割合を示す．同様に，間接・増加率の列は (3) を (4) で割った値であり，間接モードにおける実行時間の増加割合を示す．この結果から，時刻取得が直接モードのとき，rdoc は 1.2 倍程度，竹内関数プログラムでは 4 倍程度に増加することが分かった．また間接モードの場合は直接モードに対

**表 2**　本プロファイラのオーバヘッド
Table 2　Overhead of this profiler.

| | (1) | (2) | (3) | (4) | 直接・増加率（倍） | 間接・増加率（倍） |
|---|---|---|---|---|---|---|
| プロファイラ | 使用 | 使用 | 使用 | 不使用 | | |
| モニタプログラム | あり | なし | あり | - | | |
| 時刻取得方法 | 直接 | 直接 | 間接 | - | | |
| rdoc | 100.56 | 100.66 | 98.7 | 86.94 | 1.15 | 1.14 |
| 竹内関数 | 43.68 | 43.19 | 34.45 | 10.90 | 4.01 | 3.18 |

**表 3**　分割したプロファイリングモジュールの機能
Table 3　Divided functions of profiling module.

| | |
|---|---|
| 時刻取得 | OS からの時刻取得処理 |
| 実行情報取得 | Ruby からメソッド ID とクラス ID の情報取得処理 |
| ノード検索 | 管理配列と直列化配列の要素の追加・検索処理 |
| 情報算出 | 実行時間の算出と実行カウンタのインクリメント |
| 実行位置情報配列操作 | 実行位置情報配列に対する情報蓄積のための処理 |

して，rdoc ではあまり変化が見られず，竹内関数プログラムでは増加率が 3.18 倍に低下した．モニタプログラムへの情報送信が実行時間に対して与える影響は，情報送信間隔が 100 ミリ秒の場合ほとんど存在しないことが分かった．

竹内関数プログラムでは増加率がやや高いが，7.1 節でも述べたように竹内関数プログラムは頻繁にメソッド呼び出しが行われるため，最も実行時間が増加するケースである．rdoc は現実的なプログラムであり，実使用環境での増加率は rdoc の値に近いと考えられる．よって，時間的オーバヘッドは現実的なプログラムでは実用的な範囲に収まっているといえる．

### 7.5　プロファイラ自身のボトルネック

本プロファイラの各機能における実行時間を計測し，本プロファイラ自体のボトルネックの評価を行った．本評価を行うためにプロファイリングモジュールの機能を**表 3** に示す 5 つの部分に分割した．

この計測は，それぞれを 1 つずつを無効にした状態でプロファイリングモジュールを構成し，各状態での実行時間を計測し，その計測値から各部の実行時間の推定を行う．計測はトレースモードで行う．また，Ruby インタプリタに対して，いっさいの処理を行わないイベントフック関数を登録し，その実行時間を計測した．この計測値から，本プロファイラによらない，Ruby がイベントを呼び出す際のオーバヘッドを算出することができる．評価は竹内関数で行った．計測はすべて 5 回行い，その平均値を数値として採用した．

**表 4**　オーバヘッド中の各部の割合
Table 4　Percentage of each functions in the overhead.

| 部分 | 実行時間（秒） | 割合（%） |
|---|---|---|
| ノード検索 | 12.52 | 38.29 |
| 時刻取得 | 8.91 | 27.26 |
| トレース | 7.55 | 23.09 |
| 実行位置情報配列操作 | 2.79 | 8.52 |
| 実行情報取得 | 0.57 | 1.74 |
| 情報算出 | 0.36 | 1.09 |

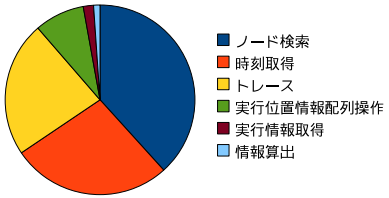


**図 14**　各部の実行時間の割合
Fig. 14　Percentage of each execution time of functions.

**表 4** と**図 14** は本評価の結果を示す表とグラフである．各項目は表 3 に示す各部の，オーバヘッド全体に対する実行時間割合を示す．また「トレース」は Ruby がイベントを呼び出す際のオーバヘッドの割合を示す．消費時間の比率が最も大きいのはノード検索で 38.29%ほどの実行時間を消費している．また，clock_gettime 関数による時刻取得が 27.26%，Ruby がイベントを呼び出す際のオーバヘッドが 23.09%であり，それらで多くの実行時間を消費していることが分かった．

## 8. 関 連 研 究

様々な言語に対してプロファイラは実装されてきた．また Ruby プログラムに対するプロファイラもすでに実装が存在する．ここでは関連した実装や研究を示す．

gprof[6] は，GNU Binutils[7] に含まれるプロファイラである．このプロファイラは，各関数の自己実行時間と総実行時間と実行回数を計測することができる．gprof はサンプリングプロファイラに分類され，サンプルイベントを用いて実行時間を計測する．また各関数の実行回数を調べるために，各関数の先頭部に回数計測用の命令を追加する．本プロファイラは Ruby プログラムに対しても行うことができるが，C 言語レベルでの関数ごとの情報が

表示されるため，Ruby のメソッドごとの情報を得ることはできない．

ruby-prof は，Ruby プログラムを対象にしたプロファイラである．このプロファイラは各メソッドに対して呼び出し回数，メモリ使用率，オブジェクト確保量を計測することができる．gprof と同様のコールグラフ形式での出力に対応している．しかし，リアルタイムプロファイリングには対応しておらず，出力はプロファイリングの終了後にまとめて出力される．

Java Virtual Machine Tool Interface（JVMTI）[8] は Java 仮想マシンをモニタリングするためのインタフェースである．Java Virtual Machine 上で動作するプログラムのためのプロファイラやデバッガ等を実装することを目的としたインタフェースである．JVMTI に含まれるイベントコールバックの設定機能や，バイトコード命令の変更機能を用いてプロファイラを実現することができる．JVMTI を用いたプロファイラである NetBeans profiler [9] 等のプロファイラはリアルタイムにプロファイリングを行うことができる．

Inoue らの研究[10] では，Java VM から低いオーバヘッドで CCT を構築する手法について述べられている．しかし，本手法は本プロファイラとは異なるサンプリングベースのプロファイラである．また，本プロファイラはすべての呼び出しをトレースし正確な CCT を構築可能であることに対して，Inoue らの提案手法では不正確な CCT を作成する可能性がある．

## 9. おわりに

Ruby で書かれたソフトウェアのプロファイリングをリアルタイムに行える実行時間プロファイラの設計と実装を行った．本プロファイラは Ruby プログラムから情報を取得するプロファイリングモジュールと，ユーザへ情報を提示するモニタプログラムの 2 つの部分で構成されている．また，CCT をもとにしたデータ構造を保持し，そのデータをプロファイリングモジュールとモニタプログラム間で通信することで情報のやりとりを行う．

本稿では，本プロファイラについて要件分析を示し，設計と実装を詳細に述べた．また，その実装によるオーバヘッドに関する評価を行い，その方法と結果を述べた．

要求分析では以下の項目を必要要件としてあげた：

- メソッドの総実行時間，自己実行時間，実行回数の取得
- 呼び出しコンテキスト情報の取得
- 現在の実行位置の表示
- プロファイリングモジュールとモニタプログラムの分離
- 低いオーバヘッド

これらの必要要件を満たすプロファイラの設計と実装を行った．

必要要件のうち「低いオーバヘッド」について，要件が満たされているか確認するため評価を行った．評価の結果，記憶領域のオーバヘッドは rdoc で 5.18，webrick で 0.11 MB 程度であった．モニタプログラムとプロファイリングモジュール間の全転送量は rdoc で 40 MB ほどであった．このときの情報取得間隔は 50 ミリ秒であった．本プロファイラによる実行時間の変化率は，極端な例である竹内関数プログラムでは 4.01 倍ほどであったが，現実的な例である rdoc で 1.15 倍ほどであった．以上の結果から，本プロファイラは十分実用的であると考える．

本プロファイラ自身のオーバヘッドを調べたところ，コールノード検索が最も多く 28.76%，OS からの時刻取得が 20.47%，Ruby 自身のイベント処理が 17.34%であり，これらでほとんどの実行時間が占められていた．この分析結果は今後のプロファイラの開発の参考となる結果である．

今後の課題として次のようなものがある．

**実行時間以外のリソース消費の取得**　本プロファイラは現在，実行時間と実行回数のみの情報しか取得できない．これ以外にも取得できる情報の追加を行う．たとえば，各コールノードで行われている入出力量を取得することで，I/O 処理の最適化を行い，対象プログラムのパフォーマンスを向上させることができる．また，各コールノードで行われているオブジェクトの生成の量を取得することで記憶領域効率のボトルネックを分析することが容易となり，効率を向上させることができる．

**オーバヘッドの低減**　今回得られた評価結果，特に 7.5 節で述べたプロファイラ自身のオーバヘッドの評価をもとに，プロファイリングのオーバヘッドの低減を目指す．手法としては，ノード検索アルゴリズムの最適化や Ruby 自身のイベントフック処理の効率化等が考えられる．

**情報取得する実行単位の追加**　本プロファイラでは現在メソッド単位でのみのプロファイリングに対応している．これをメソッドレベルではなく，Ruby のブロック単位や行単位でのプロファイリングに対応することでより詳細な性能情報の取得を可能にする．現在の Ruby では，ブロック単位でのイベントの取得には対応していないため，ブロック単位での情報取得を行うためには Ruby インタプリタへの変更が必要である．

**表示方法の工夫**　現在のモニタプログラムでは，CCT をツリービューのノードに対応させ情報表示する方法と，メソッドごとに集約し情報表示する方法に対応している．これを

ファイルごとやクラスに集約し情報表示を行ったり，視覚効果の工夫を行ったりして，より利便性を高める．

**謝辞**　本研究は科学研究費補助金（課題番号 21700024）の助成を受けたものである．

## 参 考 文 献

1) Ruby コミュニティ：オブジェクト指向スクリプト言語 Ruby．http://www.ruby-lang.org/
2) Maeda, S.: ruby-prof. http://ruby-prof.rubyforge.org/
3) Froyd, N., Mellor-Crummey, J.M. and Fowler, R.J.: Low-overhead call path profiling of unmodified, optimized code, *ICS*, Arvind and Rudolph, L. (Eds.), pp.81–90, ACM (2005).
4) Zhuang, X., Serrano, M.J., Cain, H.W. and Choi, J.-D.: Accurate, efficient and adaptive calling context profiling., *PLDI*, Schwartzbach, M.I. and Ball, T. (Eds.), pp.263–271, ACM (2006).
5) Ammons, G., Ball, T. and Larus, J.R.: Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling, *PLDI*, pp.85–96 (1997).
6) Graham, S.L., Kessler, P.B. and McKusick, M.K.: gprof: A Call Graph Execution Profiler, *SIGPLAN Symposium on Compiler Construction*, pp.120–126 (1982).
7) Project, G.: GNU Binutils. http://www.gnu.org/software/binutils/
8) Microsystems, S.: JDK 5.0 Java Virtual Machine Tool Interface (JVMTI)-related APIs & Developer Guides. http://download.oracle.com/javase/1.5.0/docs/guide/jvmti/
9) Microsystems, S.: NetBeans profiler. http://profiler.netbeans.org/
10) Inoue, H. and Nakatani, T.: How a Java VM can get more from a hardware performance monitor, *Proc. 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, OOPSLA '09*, New York, NY, USA, pp.137–154, ACM (2009).

(平成 22 年 12 月 17 日受付)
(平成 23 年 3 月 29 日採録)

**須永　高浩**

1987 年生まれ．2010 年慶應義塾大学環境情報学部卒業．現在，東京大学大学院情報理工学系研究科修士課程創造情報学専攻在学中．

**笹田　耕一**（正会員）

2004 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．2006 年同大学院工学教育部博士後期課程電子情報工学専攻退学．博士（情報理工学）（東京大学情報理工学系研究科 2007 年）．2006 年東京大学情報理工学系研究科助手，2008 年同講師（現職）．システムソフトウェア，特に並列処理システム，言語処理系に関する研究に興味を持つ．