ソフトウェアの実行を分析するための低侵襲なモニタリングツールの試作

嶋利 一真[1,a)]　石尾 隆[2,b)]　井上 克郎[1,c)]

概要：本研究では，ソフトウェアの内部状態を迅速に分析する方法として，開発者が指定した内部情報に限定して情報収集を行う低侵襲デバッグを提案する．ソフトウェアの実行について調べたい情報が決まっているとき，デバッガが開発者に指示された情報だけをソフトウェアの実行を短時間の停止だけで読み出すことで，ソフトウェアの実行への影響を最小化すると同時に，開発者への迅速な応答を実現する．

1. はじめに

情報システムは現代社会の様々な活動を支える重要な基盤であるが，情報システムのあらゆる実行状況を事前にテストし欠陥を取り除くということは現実には難しい．例えば，2007 年に首都圏で発生した自動改札機の動作障害は，1 年間のテストと半年間の運用を経た後に初めてソフトウェアの欠陥が顕在化したものである [1]．

ソフトウェアに障害が発生したとき，そのソフトウェアの内部状態を調査するための重要な道具がデバッガである．現在主流のデバッガはブレークポイント・デバッガと呼ばれており，開発者がソフトウェアの実行を一時停止し，ソフトウェアの内部状態を自由に分析することを可能としている．しかし，開発者による実行状況への干渉は，例えばソフトウェア内部の並行処理の実行順序が入れ替わる，一定時間内に完了するべき処理の実行が間に合わなくなるなどの実行の変化を引き起こす可能性がある．また，サーバ上で運用されているソフトウェアなどで，既にそれを利用している一般ユーザがいる場合，開発者による分析活動がユーザにまで影響を与えてしまう．ソフトウェアを停止させるかわりに，ソフトウェアの実行履歴を詳細に記録して実行状況を再現するといった方法も存在しているが，企業の開発者らは「実行に時間がかかるのであれば使いたくない」という立場を取っている [2]．

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University.
[2] 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology.
[a)] k-simari@ist.osaka-u.ac.jp
[b)] ishio@is.naist.jp
[c)] inoue@ist.osaka-u.ac.jp

本研究では，ソフトウェアの内部状態を迅速に分析する方法として，開発者が必要であると指定した内部情報に限定して情報収集を行う低侵襲デバッグを提案する．ここで低侵襲とは，元のプログラムの実行に与える影響が小さいことを意味しており，特にプログラムの実行を中断しないこと，実行速度に影響を与えないことを重視している．低侵襲性を実現する方法として，ブレークポイントによる実行中断を許さず，そのかわりに開発者が指定した任意の時点において情報の収集，出力のみを行う形式を採用する．ソフトウェアの実行を一時停止してから変数の一覧を閲覧するのではなく，収集した値を開発者が閲覧している最中にも実行を継続することで，実行性能への影響を軽減する．

本研究では，低侵襲デバッグの実現可能性を評価するために，Java に対する低侵襲なモニタリングツールを試作した．そして，ローカル変数とフィールドの観測処理を行った場合の Java 仮想マシンの実行性能への影響と，変数の値を観測するために生じる実行オーバーヘッドを調査した．

以降，2 章では本研究の背景と関連研究について説明し，3 章では本研究の提案手法を説明する．4 章では評価の結果を説明し，最後に，5 章で本研究のまとめと今後の課題を述べる．

2. 背景

デバッグとは，外部から観察できるソフトウェア障害の症状から，その原因となるソースコード内の欠陥を突き止める作業である [3]．その分析作業では，障害につながる命令の実行順序や変数の値を観測，調査することが重要である [4]．

開発者がソフトウェアの実行の様子を観測する手段として，ソフトウェアの処理の進行状況を示すメッセージや重

要なデータをプログラムの外部に出力するロギング処理が広く用いられている．実装の方式はプログラムによって異なるが，いわゆる print 文や，様々なロギングライブラリが活用されている [5]．しかし，ロギングによって記録されるのは開発時点で選定されたデータのみであり，デバッグの時点で任意の変数の値が閲覧できるわけではない．事前に多くの変数の値を記録するようにロギングを設計することも可能であるが，記録するデータ量が増加すると記憶媒体の管理などの運用面まで気を配る必要がある．

デバッグのために一時的なロギング命令が追加される場合もあるが，運用環境で発生した障害の分析などでは，開発者がシステムの状態を勝手に変更できない場合もある．AspectJ [6] のように対象プログラムに観測したい処理を実行時に追加することが可能な処理系も存在するが，例えば Java においては 1 メソッドあたりの大きさに制限があるなど，任意のプログラムに対してプログラム変換が可能なわけではない．

デバッガは，ソフトウェアそのものに手を加えず，外部から任意の時点での変数の観測を可能にするツールである．現在主流のデバッガはブレークポイント・デバッガと呼ばれており，対象となるソフトウェアの実行の制御とメモリの状態の観測を可能としている [7]．ただし，実行の一時停止は，システムの動作のタイムアウトなど，予期せぬ影響を引き起こす可能性がある．本研究では，システムの実行への影響を最小限とするため，状態の観測に特化した手法を提案する．

3. 提案手法

本研究では，ソフトウェアの内部状態を迅速に分析する方法として，開発者が必要であると指定した内部情報に限定して情報収集を行う低侵襲デバッグを提案する．デバッグのための観測処理が低侵襲である，つまりデバッグ対象プログラムの実行への影響を抑えるために，開発者からは情報収集の指示のみを受け付けるものとし，また，開発者から情報が要求された場合のみ出力を行う．

3.1 モニタリングツールの試作

低侵襲デバッグの実現可能性を評価するために，Java を対象とした低侵襲モニタリングツールを試作した．このツールは，プログラムに対するデータ観測命令の集合を保持し，その命令に従って情報収集を行う．1 つの観測命令はプログラムで観測を行う命令の位置 p と，観測対象データ v の組 $< p, v >$ である．プログラムの実行が命令 p に到達するたびにデータ v の観測を実行し，得られたデータをプログラムの実行スレッド th，観測時刻 t とともに内部のバッファに記録する．バッファは固定長であり，最新 k 個（現在の実装では 100 とした）の観測結果のみを保持する．これにより，例えばプログラムのクラッシュなどの特殊な動作が確認されたとき，バッファにはその特殊な動作の際に観測された値が残る可能性が高い．命令はそれぞれ実行される頻度が異なるため，バッファは観測命令ごとに用意する．

開発者はソケット通信を介して低侵襲モニタリングツールにアクセスし，データ観測命令の編集や，収集した情報の読み取りを行う．開発者が実行できる操作は，以下の 3 つである．

- 観測命令 $< p, v >$ の追加．
- 観測命令 $< p, v >$ の削除．
- 観測命令 $< p, v >$ に対応するバッファの中身の出力．

観測命令の位置 p にはソースファイル名と行番号を用いる．Java では実装上はバイトコード単位も利用可能であるが，記述の容易性の観点から行番号を採用した．観測する変数 v としては，その位置 p で利用可能なローカル変数および `this` 参照と，呼び出し元のメソッドで定義されている変数，それらの変数を基点としたフィールド参照の式を指定可能とした．また，変数を指定せず，単に通過時刻のみを記録することも可能である．

モニタリングツールの機能を，Java 仮想マシンに追加するエージェントライブラリとして実装した．観測命令 $< p, v >$ の処理は，内部的には Java 仮想マシンの持つブレークポイント機能を使用して実現した．実行スレッド th の指定された命令位置 p への到達が Java 仮想マシンから通知されるので，p に対応して観測するべきデータ v をバッファに記録し，元のプログラムの処理に復帰させる．通常のデバッガであればここで Java 仮想マシンに対して実行スレッドの停止を指示するところであるが，本研究で構築したモニタリングツールはスレッドの停止を行わない．

モニタリングツールは開発者との対話に専用のスレッドを使用する．CPU 資源が十分にあれば，対話そのものがプログラムの実行に影響を与えることはない．また，いわゆるサーバ・クライアントモデルに相当し，複数の開発者が同じプログラムに接続し，観測命令等を独立に編集することも可能である．開発者の認証の仕組みなどは本研究のスコープ外とした．

プログラムの起動直後などは，対話的操作での観測指示が難しいため，モニタリングツールはプログラム起動時に設定する観測命令のリストを受け付ける．また，プログラム終了時に，内部バッファに蓄積したすべてのデータをファイルに出力する．これにより，実行がすぐに終わるようなプログラムでもデータの観測は可能である．

3.2 モニタリングツールの利用例

モニタリングツールの利用例として，ANTLR によって生成された構文解析器のエラーメッセージの分析方法を示す．ANTLR によって生成された解析器は, 認識できない文字列が入力として与えられたとき，`Lexer` クラス中の以

下のソースコードで警告メッセージを通知する（行頭の数字は行番号である）．

```
359 public void notifyListeners
      (LexerNoViableAltException e) {
360    String text = _input.getText
         (Interval.of(_tokenStartCharIndex,
          _input.index()));
361    String msg = "token recognition error at:
          '"+ getErrorDisplay(text) + "'";
362
363    ANTLRErrorListener listener
          = getErrorListenerDispatch();
364    listener.syntaxError
          (this, null,_tokenStartLine,
          _tokenStartCharPositionInLine, msg, e);
365 }
```

このエラーメッセージはトークンの文字列と行番号を含んでいるが，そのトークンに対応するファイル名は含んでいない．ファイル名自体は構文解析器には渡されておらず，この構文解析処理を呼び出したメソッドが持つ情報を参照する必要がある．

ブレークポイントデバッガであれば，ブレークポイントを 363 行目に設定して変数のビューを確認し，呼び出し元のメソッドを確認することができる．ただし，この方法ではエラーが発生する都度，実行の中断と目視での確認が必要となってしまう．

これに対して，本研究で試作したモニタリングツールでは，たとえば以下のように観測命令を 1 つ作成することになる．

```
Lexer.java <main>.filename 363
```

この観測命令は，`Lexer.java` ファイルの 363 行目において，現在実行中の `main` メソッドで有効な変数 `filename` を参照することを指示する．モニタリングツールは，当該命令位置にスレッドが到達したとき，コールスタックを辿り，main メソッドにおける filename の値を記録する．呼び出し元となっているメソッドの名前や変数名は事前に調査が必要であるが，デバッガの能力を用いた変数の値の収集が可能である．

4. 評価

作成したモニタリングツールがプログラムの実行に与える影響の評価を行う．デバッグ機能を有効にすると，Java 仮想マシンがデバッグのために何らかの最適化などを抑制する可能性がある．そのため，ベンチマークを用いて，モニタリングツールを組み込んだことによって生じるオーバーヘッドを評価する．また，デバッグ機能によって実際に変数の値を観測し，出力を行った場合，観測や出力の実行回数に比例して実行時間が増加すると考えられる．この実行オーバーヘッドについては，実験のための小規模なプログラムを用いて観測する．

オーバーヘッドは実行ごとに変動するため，プログラムを各条件でそれぞれ 10 回実行し平均値を求める．プログラムをそのまま実行した場合の計測結果を t_i，モニタリングツールを有効にした場合の計測結果を t'_i とすると $(1 \leq i \leq 10)$，オーバーヘッド $O(\%)$ は以下の通りである．

$$O = (\frac{\sum_{i=1}^{10} t'_i}{\sum_{i=1}^{10} t_i} - 1) \times 100$$

性能計測の環境は，CPU として Intel(R) Xeon(R) CPU E5-1603（2.80GHz），メモリ 16GB，ストレージとして SATA HDD を搭載したワークステーションである．OS として Windows 10 Pro，Java 仮想マシンには Oracle Java SE の build 1.8._121-b13 を用いた．

4.1 DaCapo Benchmarks への適用

Java 仮想マシンが持つデバッグのための機能を有効にしたことによって生じるオーバーヘッドを確認するために，モニタリングツールを観測命令なしの状態で使用したときの実行性能への影響を，DaCapo Benchmarks[8] のバージョン 9.12-bach を用いて計測した．DaCapo Benchmarks に収録された 14 個のベンチマークのうち，ツールに対して動作が確認できた 12 個のベンチマークを用いた．

使用したベンチマークごとのオーバーヘッドの分布を箱ひげ図として**図 1** に示す．通常実行を基準としたオーバーヘッドは平均 7.9%，最大 21%であった．この値は，例えば企業の業務アプリケーションなど，リアルタイム性の低い環境では影響は小さいと考えらえる．

実験で相対的に大きなオーバーヘッドを示したベンチマークについて，その原因を分析したところ，JVM TI の利用する機能を指定する `AddCapabilities` 関数の実行の有無によって実行速度に大きな差が出ることが判明した．具体的には，ブレークポイントを設置するための権限を付与した場合に，大きなオーバーヘッドがかかっている．これは，ブレークポイントを設置しない場合にのみ実行できるような最適化がこれらのベンチマークに大きく寄与していた可能性がある．

4.2 観測および出力のオーバーヘッド

観測命令の実行に対して，どの程度のオーバーヘッドが生じるかを計測した．計測に使用したコードの基本形は以下のようなものである．

```
1  var1=0;
2  for(int i=0; i<N; i++){
3    var1++;
4  }
```

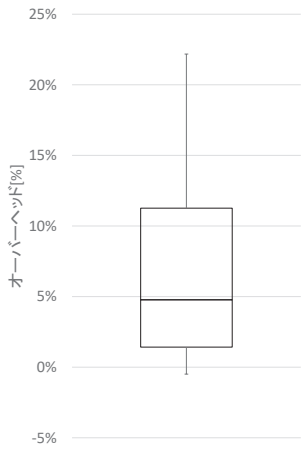

図 1 DaCapo Benchmarks に対する実行のオーバーヘッド

表 1 ログ出力命令の実行回数と実行時間 [ms]

出力回数	10000	20000	30000	40000	50000
通常実行	787.3	1616.1	2432.5	3246.8	4063.8
局所変数	579.0	1142.9	1889.3	2656.7	3387.1
クラス フィールド	362.8	711.1	1232.0	1772.2	2307.2
インスタンス フィールド	1135.6	2251.9	3546.2	4859.8	6167.3

通常実行として，変数 `var1` の値とスレッド，現在時刻を出力するような println 呼び出しを埋め込んだものを用いた．また，モニタリングツールを使用した場合については，3 行目における `var1` を観測対象として選んだ．観測対象となる変数が局所変数，クラスフィールド，インスタンスフィールドのいずれかによって本ツールが内部的に利用する Java 仮想マシンの API が異なり，実行時間も変わってしまうため，それぞれの種別に対してループ文の実行にかかる時間の測定を行った．ループの実行回数 `N` は最大 50000 とし，通常実行時と出力回数を合わせるために，モニタリングツールも最新 50000 件までのデータを保存する設定とした．そして，実行終了時に一括でそれらのデータの出力を行った．

ログ出力命令の実行回数と実行時間を**表 1** に示す．実行時間と実行回数は通常実行時，モニタリングツール使用時共に，ほぼ線形に増加する．出力もモニタリングツールによる値の取得も行わずに実行時間を計測した結果は，5 万回実行しても両者共に高々 2ms であった．つまり，ログ出力命令の実行時間は通常実行では出力に，モニタリングツールでは値の取得と出力に大きく依存している．

局所変数とクラスフィールドにおいて，値の取得と出力に大きなコストがかかっているにもかかわらず通常実行時よりも実行時間が短くなった原因としては，ログの出力量が減少したことが原因であると考えられる．具体的には，通常実行時は毎回ファイル名，メソッド名，行番号，変数名，型名について出力する必要があるが，本モニタリングツールではこれらの情報が同じ場合，一度だけの出力で済むようにログの出力が制御されているため，実行時間が短くなっている．

また，本ツールは出力や値の取得をコマンドで切り替えられるため，適切なログのみを出力するように指示を行えば，オーバーヘッドはほぼ無視できるものと考えられる．

5. まとめと今後の課題

本研究では，既存のデバッガで十分にできない低侵襲デバッグを実現するために，JVM TI を用いて低侵襲モニタリングツールの試作を行った．

今後の課題としては，ロギングを用いた有効なデバッグのために必要なデータの量を調査し，本モニタリングツールのバッファへの記録の仕方やそのデータ量を変更したり，必要なオプションを追加することが挙げられる．なお，モニタリングツールが情報を自由に収集できる点はセキュリティ面での懸念を伴うため，何らかのアクセス制御などの対策を施したうえでツールの公開を目指す予定である．

謝辞 本研究は JSPS 科研費 JP25220003, JP26280021 の助成を受けたものです．

参考文献

[1] ITMedia ニュース：「260 万人の朝の足を直撃　プログラムに潜んだ“魔物”」，`http://www.itmedia.co.jp/news/articles/0710/12/news117.html` (2007).

[2] Siegmund, B., Perscheid, M., Taeumel, M. and Hirschfeld, R.: Studying the Advancement in Debugging Practice of Professional Software Developers, *Proceedings of International Workshop on Program Debugging*, pp. 269–274 (2014).

[3] Zeller, A.: *Why programs fail, the 2nd edition*, O’Reilly Japan，second edition (2012).

[4] Spinellis, D.: *Effective Debugging: 66 Specific Ways to Debug Software and Systems*, Addison-Wesley Professional (2016).

[5] Kabinna, S., Bezemer, C.-P., Shang, W. and Hassan, A. E.: Logging Library Migrations: A Case Study for the Apache Software Foundation Projects, *Proceedings of International Conference on Mining Software Repositories* (2016).

[6] 千葉　滋：アスペクト指向入門，技術評論社 (2005).

[7] Jonathan B. Rosenberg 著 吉川 邦夫訳：How Debuggers Work デバッガの理論と実装，アスキー (2007).

[8] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D. and Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *Proceedings of ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190 (2006).