特別研究報告

題目

ソフトウェア障害分析のための
低侵襲な実行モニタリングツールの試作

指導教員

井上 克郎 教授

報告者

嶋利 一真

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成28年度 特別研究報告

ソフトウェア障害分析のための低侵襲な実行モニタリングツールの試作

嶋利 一真

内容梗概

情報システムは現代の社会の様々な活動を支える重要な基盤であり，ソフトウェアの動作に障害が発生すると，社会的にも大きな影響を及ぼすことがある．

ソフトウェアに障害が発生したときに重要となるのが，ソフトウェアの内部状態を分析し，障害の原因を迅速に把握することである．例えばソフトウェアが応答しないという障害が発生したとき，直接の原因はソフトウェアの内部状態から知ることが出来る．そして，その状況に陥るに至ったソフトウェアの振る舞いを分析することで，ソフトウェアの修正に限らず，異常データの除去，計算機資源の追加などの様々な対処が可能となる．

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の一時停止や再開を制御し，一時停止した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．つまり，ブレークポイント・デバッガを使った分析活動自体がソフトウェアの実行を変化させてしまい，ソフトウェアの正確な内部状態を分析できなくなる．

本研究では，ソフトウェアの正確な内部状態を迅速に分析する新しい方法として，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 JVM Tool Interface を用いた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たとき，開発者が手作業で実行を一時停止してからその情報を読み出すのではなく，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継続したまま，あるいはきわめて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．

主な用語

ソフトウェア障害
低侵襲性
デバッグ
JVM TI

目 次

1 はじめに **3**

2 背景 **5**

2.1 障害分析について 5

2.2 障害分析手法における望ましい要素 6

2.3 既存のデバッグ手法 6

3 提案手法 **8**

3.1 ツールの機能 8

3.2 ツールの設計方針 9

3.3 試作したツールの操作方法 10

3.3.1 試作したツールのコマンド 10

3.3.2 試作したツールの実行例 11

3.4 JVM TI による実装方法 14

3.5 低侵襲性の実現方法 17

4 評価 **18**

4.1 Web アプリケーションのデバッグへの適用 18

4.2 性能評価 18

4.2.1 DaCapo Benchmarks を用いたベンチマークテスト 18

4.2.2 ブレークポイントにおける出力によるオーバーヘッド 21

5 まとめと今後の課題 **26**

謝辞 **27**

参考文献 **28**

付録 **29**

1 はじめに

情報システムは現代の社会の様々な活動を支える重要な基盤であり，ソフトウェアの動作に障害が発生すると，社会的にも大きな影響を及ぼすことがある．しかし，情報システムが運用される期間に発生するあらゆる状況を事前にテストし，すべての欠陥を事前に取り除くことは現実的に難しい．たとえば，2007 年 10 月 12 日朝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，開発者の想定外の実行を引き起こすような入力データがシステムに与えられ，ソフトウェアの欠陥が顕在化したものである [4].

ソフトウェアに障害が発生したときに重要となるのが，ソフトウェアの内部状態を分析し，障害の原因を迅速に把握することである．例えばソフトウェアが応答しないという障害が発生したとき，異常なデータによって処理の完了条件が満たされなくなってしまったのか，計算機資源の不足によって時間がかかっているのか，データベースやネットワークなど外部システムとの接続に問題があるのかといった直接の原因は，ソフトウェアの内部状態から知ることが出来る．そして，その状況に陥るに至ったソフトウェアの振る舞いを分析することで，ソフトウェアの修正に限らず，異常データの除去，計算機資源の追加などの様々な対処が可能となる．

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の一時停止や再開を制御し，一時停止した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．つまり，ブレークポイント・デバッガを使った分析活動自体がソフトウェアの実行を変化させてしまい，ソフトウェアの正確な内部状態を分析できなくなる．また，現在すでにソフトウェアを利用している一般ユーザがいる場合，ソフトウェアの実行を一時停止してしまうとそのユーザまで影響を受けてしまう．

この問題に対して，ソフトウェアの実行履歴を記録して過去の実行状況を復元する Capture and Replay 技術というのも存在しているが，企業の開発者らはこのような高度な技術に対しては「実行に時間がかかるので使いたくない」という立場を取っており [3]，短時間でソフトウェアの正確な動作を分析できる軽量な手法が求められている．

本研究では，ソフトウェアの正確な内部状態を迅速に分析する新しい方法として，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 JVM Tool Interface を用いたデバッグを提案する．この方法において重要視するのは低侵襲性である．低侵襲性とは，元のプログラムの実行にほとんど影響を与えないような性質のことである．本デバッグでは，

元のプログラムの実行速度や実行順序に影響を与えないこと，実行を中断しないことがあげられる．

低侵襲の実現方法としては，既存のブレークポイント・デバッガにおいてブレークポイントが来たら実行を中断している点を，ブレークポイントにおいて実行を中断せずに必要な情報のみを出力する形で行う．また，ブレークポイントの情報はユーザ側で自由に定義出来るものとする．これにより，実行を中断せず，実行時間のオーバーヘッドも最低限で実現するものとする．

本研究ではJavaを題材にした低侵襲モニタリングツールの試作を行った．実装手段としてはC++でJava^{TM} Virtual Machine Tool Interfaceを用いて実装を行った．Java^{TM} Virtual Machine Tool InterfaceはOracleが提供している(以降はJVM TIと呼ぶ)． JVM TIは，Java^{TM} 仮想マシンで動作するアプリケーションの状態検査と実行制御の両方の機能を提供し，プロファイリングツール，デバッグツール，監視ツール，スレッド分析ツール，カバレッジ分析ツールなどのVMの状態その他にアクセスする必要がある各種ツールのVMインタフェースとして機能するものである[7]．Eclipseデバッガにも JVM TIは用いられており，これを用いることによって機能としては従来のデバッガと同等のものは問題なく作成できる．JVM TIを用いることでソフトウェアの実行について調べたい情報が決まったとき，開発者が手作業で実行を一時停止してからその情報を読み出すのではなく，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継続したまま，あるいはきわめて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．

具体的な実現方法として，JVM TIの持つ機能を使い，Javaプログラムの実行を停止せずに指定した変数の情報を取得した．また，外部からtelnetコマンドを用いて通信を行うことで自由にブレークポイントの追加や削除を行い，必要以上のメモリを使用しないようにしている．

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

2　背景

本節では本研究の背景として，デバッグの手順，既存のデバッグ手法についての説明を行う．

2.1　障害分析について

通常，ソフトウェア障害（以下は単に障害と書く）は次の 3 段階を経て発生する [9].

- プログラマがプログラムコード内に欠陥（バグ，故障）を作る
- 欠陥がプログラム状態の中に感染を引き起こす
- 感染が障害（外部から観察できるエラー）を引き起こす

障害を取り除くための作業，デバッグは 7 つの手順に従って進められる [9].

1. 追跡：問題データベースにエントリを作成する
2. 再現：障害を再現する
3. 自動化：テストケースの自動化と単純化を行う
4. 起源の検出：障害を起点に感染の起源となった可能性がある箇所まで，依存関係を遡る
5. 着目：感染起源の候補が複数ある場合，既知の感染，障害原因，異常，あやしいコード，早期の欠陥源を確認する
6. 切り分け：感染の起源を特定し，欠陥から障害に至る感染の連鎖が分かるまで，感染起源の探索を続ける
7. 修正：感染の連鎖を断ち切って欠陥を除去する．そして，修正がうまく行ったことを確認する

デバッグ作業の中で，一番時間がかかるステップは手順 4〜6 を繰り返す欠陥の検出，障害の分析である．これらのステップにおいて用いられるのがロギングやデバッガである．

ロギングは開発者が，プログラムにあらかじめ特定のデータをプログラムの外部に出力するように命令を記述する方式である．デバッガは元のプログラムを何も変更せず，実行中のプログラムと接続し，特定の瞬間の状態を観察・操作するものである．

2.2　障害分析手法における望ましい要素

障害分析に用いる手法において望ましい要素は以下のようになる．

1. ソフトウェアの実行を中断せずデバッグ作業が実施出来る
2. 任意のタイミングで実行ログ出力の可否を切り替えることが出来る
3. プログラム実行におけるオーバーヘッドが小さい
4. プログラムに対する副作用がない

1の前提として，障害分析の際に重要な点として，デバッガ等を用いても通常の実行時と同じように動作することが挙げられる．これが保証されずに実行途中で実行の中断等が起こってしまった場合，通常の実行時と異なる振る舞いを見せてしまう．例えば，ネットワーク系のシステムであれば，不用意に実行を中断した場合にタイムアウトが発生して，正常な動作を行えなくなってしまう．したがって障害分析において,『ソフトウェアの実行を中断せずデバッグ作業が実施出来る』ことは重要である．

2は，ロギングを行う際，実際に必要なログだけを取得することは重要だということである．もし，ログ出力を行いたい部分以外のログを出力してしまった場合，ログの可読性が下がる上に，空間的コストも増大してしまう．したがって，実行ログが膨大になりすぎないように,『任意のタイミングで実行ログ出力の可否を切り替えることが出来ることは重要である．

3の前提として，デバッガは通常のプログラムの実行に加えて処理を行うことで，プログラムの振る舞いを観察・制御することが出来るが，観察や制御を行う際に，プログラム通常の実行に加えて別の処理をしているのでオーバーヘッドが生じることがある．したがって，1の『ソフトウェアの実行を中断せずデバッグ作業が実施出来る』と同様に，オーバーヘッドの影響で通常の実行時と異なる振る舞いを見せる可能性があり，予期せぬ動作を起こしてしまい，正常に障害分析ができない可能性がある．したがって,『プログラム実行におけるオーバーヘッドが小さい』ことは重要である．

4のプログラムに対する副作用とは，ある機能がコンピュータの論理的な状態を変化させ，それ以降で得られる結果に影響を与えることである．いったん副作用が生じた場合，通常実行時と異なる振る舞いを見せる可能性があり，正常に障害分析ができない原因となる．したがって,『プログラムに対する副作用がない』ことは重要である．

2.3　既存のデバッグ手法

2.2節を踏まえて，既存のデバッグ手法について述べる．代表的な手法として，対話的デバッガ，ロギング，AspectJについてそれぞれの特徴をあげる．

対話的デバッガ

デバッグ担当者がプログラムにブレークポイントを設置し，プログラムの実行を一時的に停止して，そのときの変数の値を確認する [8]．その後，シングルステップを用いたり，複数のブレークポイントを設置することで，あらゆる変数を分析可能であるが，一般ユーザが使っている途中のものなど,「動いている」ソフトウェアをそのまま分析する用途には適さない．また，ブレークポイント＋ステップ実行による値の観測は，実行順序などをある程度把握している状態でなければ使えない．この手法は本節で述べた１の『ソフトウェアの実行を中断せずデバッグ作業が実施出来る』の条件を満たさない．

ロギング

開発者が，プログラムにあらかじめ特定のデータをプログラムの外部に出力するように命令を記述する方式で，そのデータを分析することで障害の原因を特定することが出来る．事前に十分なデータが選定できればよいが，予想外の原因による障害が生じた場合や動作中のプログラムの分析には利用することが難しい．詳しいデータを記録すればするほどデータ量が増加するため，記憶媒体の管理などの運用面まで気を配る必要がある．また，ほとんどのデータはシステムに問題が起きない限り利用されないので，計算機資源の無駄遣いにもなる．この手法は本節で述べた２の『任意のタイミングで実行ログ出力の可否を切り替えることが出来る』の条件を満たさない．

AspectJ

ソースコード変換によって観測したい処理を対象プログラムに埋め込む方式であり，デバッグ担当者がロギングを行えるようになったとは考えられる [2]．しかし，今動いているソフトウェアの分析という使い方は，ソースコード変換を用いた場合は副作用が心配され，本節で述べた４の『プログラムに対する副作用がない』の条件を満たさない．

このように全ての要件を十分に満たしたデバッグ手法は現在確立されてない．しかし，このようなデバッグ手法が必要とされることは多い．例えば，稼働中のネットワーク通信を扱うサーバがあげられる．システムは稼働中であるため，一般ユーザが利用しており，実行を中断するのは不可能である．また，想定外の動きをシステムが起こした場合，本節のロギングでも述べたように一般的なログ出力では十分に対応できないことがある．さらに，稼働中のシステムであるため，オーバーヘッドが大きい方法は使用することが出来ない．現在，このようなシステムのバグを取る際には長時間メンテナンスを行わなければならない場合が多く，その際に一般ユーザがサービスを利用できなくなるといった問題がある．

3 提案手法

本研究では，低侵襲モニタリングツールの提案と試作を行った．手法を図解すると，図 1 のようなイメージである．

まず最初に，ユーザから観測命令 $< p, v >$ として観測したい位置 p とデータ v の組の集合を受け取る．次に，それらのデータをツール内で，ブレークポイント未設置集合に追加した後，それぞれの組に対してブレークポイント設置の処理を行う．設置が完了すればブレークポイント設置済み集合へと移す．ユーザからのデータ受信処理は初期段階と，実行途中の任意の段階で出来るものとする．そして，プログラムがブレークポイントに到達する毎に，その位置 p において指定されたデータ v の情報を出力する．本研究における詳細な機能に関しては 3.1 節で説明する．

本研究においては，位置 p の指定にはファイル名と行番号を，データの指定には変数名を用いた．理由としては，一般的な print 文と同様の処理を行いたいからである．一般的な print 文は，特定のファイルの特定の行において，特定の変数の情報を出力することが出来る．これと同様の処理を行うために，位置の指定にファイル名と行番号，データの指定に変数名を用いた．

なお，本ツールの試作の際に JVM TI を用いている．JVM TI は，Java^{TM} 仮想マシンで動作するアプリケーションの状態検査と実行制御の両方の機能を提供し，プロファイリングツール，デバッグツール，監視ツール，スレッド分析ツール，カバレッジ分析ツールなどの VM の状態その他にアクセスする必要がある各種ツールの VM インタフェースとして機能するものである [7]．したがって，本デバッガの実行モニタリングという目的に適していると考えられる．具体的にどのように JVM TI を用いたかについては 3.4 節で説明する．JVM TI を用いて言語は C++でコーディングを行った．ツールは Java 仮想マシンに実行時に読み込まれるエージェントとして実装されており，プログラム実行時の引数としてツールの実行パスを与えることで，ツールの機能が利用可能になる．使用したバージョンは，Java(TM) SE Runtime Environment (build 1.8._121-b13) Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode) である．

3.1 ツールの機能

ツールの機能は，開発者が指定した観測命令 $< p, v >$ について特定の実行位置 p で特定のデータ v を実行を停止せずに出力することである．p はファイル名と行番号で指定し，v としては局所変数の名前あるいは時刻を指定するものとした．変数の中でも局所変数のみの実装となった理由としては，3.4 節で述べる

指定は，起動時にテキストファイルによって行うか，プログラム実行中のソケット通信に

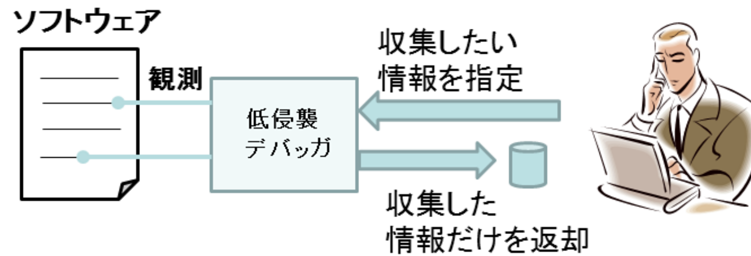

図 1: 低侵襲なモニタリングツールの概観

よって行うことが出来る．これにより，“ファイル名 メソッド名 行番号 変数名 [型名] is 値” の形式で結果が出力される．

これらの機能は 2.2 節に基づき，採用された．具体的に述べると，まず『ソフトウェアの実行を中断せずデバッグ作業が実施出来る』点についてはこのデバッガの特徴の最も大きな特徴であり，一般的なブレークポイント・デバッガとの差異であるため，3.4 節で述べる機能を用いて実装を行った．

二つ目の,『任意のタイミングで実行ログ出力の可否を切り替えることが出来る』については，3.3 節で，外部からコマンド入力を行うことで実装を行った.『プログラム実行におけるオーバーヘッドが小さい』については，4.2 節にて詳しく記載するが，小さいオーバーヘッドで実行を行うことが出来た．

最後の,『プログラムに対する副作用がない』については，本プログラムは図 1 で示したようにソフトウェアに対して観測ポイントを設置することのみで実行の制御を行っていない．したがって，副作用は生じないと考えられる．

3.2 ツールの設計方針

3.1 節で説明した機能を実装するために，今回採用したデータ取得の戦略を説明する．
まず，起動時にテキストファイルを読み込み，ブレークポイントを設置するファイル名・変数名・行番号のリストを作成する．次に，実際にブレークポイントの設置・処理を行うスレッドとは別に，外部からの通信を受け入れるスレッドを作成する．このスレッドでは “行いたい命令 ファイル名 変数名 行番号” を観測命令として指定することで，それをブレークポイント未設置集合に追加する．次に，この集合を用いてブレークポイントの設置を行う．このブレークポイントの設置が，本モニタリングツールを扱う際にオーバーヘッドが生じる大きな原因となっている．

本ツールではオーバーヘッドを小さくするために，ブレークポイントの設置はクラスの準備時に行う．指定される情報が，ファイル名，変数名，行番号のため，最初に該当するファ

イルを探索し，クラスファイルが準備されるたびに指定されたリストを検索して該当するファイルが存在するかどうかの確認を行う．こうしてすべてのクラスに対して確認を行い，ブレークポイントの設置を終了する．

次に，プログラムがブレークポイントに到達した際に情報出力の処理を行う．具体的に行う処理は，実行を止めずに指定した変数の値を出力すること，あるいは指定した行番号での時刻を取得することのみである．ブレークポイントをプローブポイントとしてデータの出力に用いることは，ブレークポイントの副作用を効果的に利用していると言える [8].

この手法により，実行を止めずにデバッグを行うことが出来る．

3.3 試作したツールの操作方法

今回の研究で試作したツールの操作方法について以下で順に述べる．

1. Java プログラムの実行時に，VM 引数に対して本研究で作成したデバッガの dll ファイルを，ブレークポイントを設定したい場所を "ファイル名 変数名 行番号" の形式で指定した設定ファイルと共に読み込ませる

2. Java プログラムの実行状況に合わせて，設定ファイルで指定された変数の情報が Eclipse の標準エラー出力に出力される

3. ブレークポイントを追加・削除したい場合は，外部プロセスから telnet コマンドを用いて，実行中の dll にファイルに接続を行い，"行いたい命令 ファイル名 変数名 行番号" の順に入力を行う．

3.3.1 試作したツールのコマンド

外部プロセスから telnet コマンドを用いて実行できるコマンドは以下の 4 つがある．

以下にコマンド名とその使用法をまとめる．

Set コマンド

ブレークポイントを Java プログラムに対して設置することが出来るコマンドである．設定ファイルに記述されている "ファイル名 変数名 行番号" に対してはこのコマンドが実行される．Cygwin から実行する場合は "Set ファイル名 変数名 行番号" の形式で入力を行う．この際に指定できる変数名は，配列型，String 型，オブジェクト型を除く局所変数のみである．また，変数名の代わりに "-time" という入力をすることも出来る．このコマンドが実行された場 合は変数の情報の出力の代わりに，プログラムが指定された地点を実行した時刻を，"月/日 時:分:秒. ミリ秒" の形式で出力する．

Clear コマンド

本ツールで Java プログラムに設置されたブレークポイントを削除することが出来るコマンドである．Cygwin から実行する際は，“Clear ファイル名 変数名 行番号”の形式で入力を行う．Clear コマンドで削除されるブレークポイントはすでに設置されているブレークポイントのみで，存在しないブレークポイントを入力として与えてもプログラムには何も起こらない．

LogOn コマンド

本ツールで Java プログラムに設置されたブレークポイントによって出力されるログの出力をオンにするコマンドである．デフォルトではログの出力機能はオンになっている．Cygwin から実行する際は，“LogOn”のコマンド名のみを入力として与える．

LogOff コマンド

本ツールで Java プログラムに設置されたブレークポイントによって出力されるログの出力をオフにするコマンドである．デフォルトではログの出力機能はオンになっている．Cygwin から実行する際は，“LogOff”のコマンド名のみを入力として与える．

3.3.2 試作したツールの実行例

Cygwin と Eclipse を用いた実際の実行例をいくつか挙げる．まず，実行開始時に実行位置 p とデータ v の組を指定して，それらの組で指定されたデータの情報を出力する処理の実行例を挙げる．ソースコードの全文は付録にて記載した [6]．このプログラムのファイル名は “Main.java”で，プログラムの出力機構は 39 行目と 41 行目の print 文である．通常実行を行うとこのプログラムの出力は以下のようになる．

```
0
10
20
0
11
22
30
33
34
40
44
```

次に，このプログラムに対して以下のように設定テキストファイルを記述する．

```
Main.java delete1 42
Main.java delete2 48
Main.java -time 45
```

この設定ファイルは行ごとに処理される．順に，Main.java の 42 行目の delete1，Main.java の 48 行目の delete2 のそれぞれの行頭での情報と Main.java の 45 行目の先頭を通過した時刻情報を標準エラー出力に出力するよう設定したことを意味する．そして，実行時の VM 引数に "-agentpath:'作成した.dll ファイルへのパス'='作成した設定.txt ファイルへのパス' " を指定する．実例は以下のようになる．

```
-agentpath:C:/Users/Owner/Source/Repos/ProbeJ/x64/Debug/ProbeJ.dll
=C:/Users/Owner/Source/Repos/ProbeJ/x64/Debug/options.txt
```

この指定を終えた後，Java プログラムを実行すると，Eclipse の標準エラー出力に以下のように出力される．

```
Main.java main line42 delete1[I] is 0
Main.java main line42 delete1[I] is 10
Main.java main line42 delete1[I] is 20
Main.java main line45 02/13 13:53:22.358
Main.java main line48 delete2[I] is 0
Main.java main line48 delete2[I] is 11
Main.java main line48 delete2[I] is 22
Main.java main line48 delete2[I] is 30
Main.java main line48 delete2[I] is 33
Main.java main line48 delete2[I] is 40
Main.java main line48 delete2[I] is 44
```

結果から，プログラムに元々あった print 文と同様の動作を行えていることが確認できた．したがって，指定ファイルの指定行番号の行頭における変数の値，型名，時刻情報が正確に出力されていることが分かる．この実験より，設定ファイルによる記述でブレークポイント設置とブレークポイントにおける情報出力を本ツールにおいて行えることが分かった．

次に，実行途中において実行位置 p とデータ v の組を指定して，それらの組で指定されたデータの情報を出力する処理の実行例を挙げる．本実行例における Java プログラムのソースコードを行番号と共に抜粋し以下に記載する．

テキストファイルの設定まで先程の同様の処理を行う．このまま実行を行えば，順に Main.java の 42 行目の delete1，Main.java の 48 行目の delete2 のそれぞれの行頭での情報と Main.java の 45 行目の先頭を通過した時刻情報を標準エラー出力に出力するよう設定したことを意味する．

本実行例では，このメソッドの実行を行う前に外部プロセスから本ツールに対して telnet コマンドを用いて通信を行った．以下に行ったコマンドを記述する．

```
$ telnet
telnet> open home-pc 39876
Set Main.java j 47
Clear Main.java delete2 48
```

これにより外部プロセスからプログラムを実行しているホストにアクセスすることが出来た．Set コマンドにより，新たに Main.java の 47 行目における j の情報を出力するようにブレークポイントが設置され，Clear コマンドにより 48 行目に指定されていた delete2 を出力するためのブレークポイントが削除された．この処理の後に実行を行うと以下のようになる．

```
Main.java main line42 delete1[I] is 0
Main.java main line42 delete1[I] is 10
Main.java main line42 delete1[I] is 20
Main.java main line45 02/13 13:53:22.358
Main.java main line47 j[I] is 0
Main.java main line47 j[I] is 1
Main.java main line47 j[I] is 2
Main.java main line47 j[I] is 3
Main.java main line47 j[I] is 4
Main.java main line47 j[I] is 5
Main.java main line47 j[I] is 6
```

実行結果より，実行途中においてコマンドで指定・削除されたブレークポイントが更新されて，最終的に指定されたファイル名・行番号の行頭における変数の値，型名，時刻情報が正確に出力されていることが分かる．この実験より，実行途中におけるコマンドでブレークポイント設置・削除と，最終的に設置されているブレークポイントにおける情報出力を本ツールにおいて行えることが分かった．

3.4　JVM TI による実装方法

JVM TI は，JavaTM 仮想マシンで動作するアプリケーションの状態検査と実行制御の両方の機能を提供する．JVM TI の関数については大きく分けて，プログラムの様々な情報を取得出来る JVM TI 関数とプログラム内で発生する多くのイベントについての通知を受けることが出来るイベント・コールバック関数の 2 つの種類がある．本ツールでは，この 2 つを用いて以下の順に操作を行う．

1. JVM TI のクライアント (エージェント) を起動する
2. エージェントがイベントの通知を受け取り，イベントごとに対応するイベント・コールバック関数を呼ぶ
3. イベント・コールバック関数内で JVM TI 関数を呼び出して，JVM TI 機能にアクセスする

本研究で使用したイベント・コールバック関数の一覧と，それぞれに割り当てた処理を以下に記す．

Agent_OnLoad イベント

エージェント起動時に呼ばれる関数である．ここで行う処理は，テキストファイルに対する字句分割の処理と，他のイベント・コールバック関数の設定，JVM TI 関数への権限付与である．この際に使用しているデータモデルは入力された実行位置 p とデータ v による観測プローブ $< p, v >$ のリストと観測プローブの配備済み・未配備のフラグである．これらを参照し，後述の ClassPrepare イベント でブレークポイントを設置していく．

ここで，表 1 に示した関数を用いて初期設定を行う．具体的には GetCapabilities 関数で現在の権限を取得し，権限を設定後に AddCapabilities で権限を追加する．そして SetEventCallback 関数と SetEventNotificationMode 関数で各イベント・コールバック関数に対してコールバックの許可を行う．

Agent_OnUnload イベント

エージェントのシャットダウン時に呼ばれる関数である．今回は特に処理を行っていない．

ClassPrepare イベント

プログラムであるクラスの準備が完了した時点で生成される関数である．本ツールではこのイベント・コールバック関数内で，ブレークポイントの設置を行っている．具体

的には，ブレークポイントの設置情報が書かれたリストのファイル名と一致する Java クラスの準備が完了した時点で，そのクラスの全メソッドを調べてブレークポイントを設置したい行番号を含むメソッドを特定する．そして，そのメソッドに対してブレークポイント設置の処理を行う．これを全クラスの準備が完了するまで続ける．

MethodEntry イベント

プログラムがメソッドに入った時点で呼び出される関数である．本ツールではこの関数を，プログラムの実行準備が整って初めて何らかのメソッドの実行が開始されたときのみ呼び出す．呼びだした際に，ソケットを開き接続を待ち受けるスレッドを生成する．このスレッドに対して外部から通信を行って観測プローブ $<p,v>$ のリストを操作することで，ブレークポイントの設置･除去を行うことが出来る．また，Breakpoint イベント実施の可否についても外部からの通信により設定することが出来る．

Breakpoint イベント

ClassPrepare イベントで設置したブレークポイントに対して処理を行う．具体的には設置されたブレークポイントとその際にリストで指定された変数や時刻に対して，その情報の出力処理を行う．

また，利用した JVM TI 関数の一覧とその処理内容を表 1 に簡潔にまとめる．

これらの JVM TI 関数やイベント・コールバック関数を用いて以下の手順で本ツールの処理を行う．

1. プログラム実行と同時にツールの実行が開始され，Agent_OnLoad イベントが呼ばれ，その内部でブレークポイント未設置リストを作成し，実行権限を付与する

2. ClassPrepare イベントでブレークポイント未設置リストに対し，ブレークポイント設置処理を行い，ブレークポイント設置済みリストに移す

3. プログラムがブレークポイントに達した際，指定された変数に対して出力処理を行う

本ツールで基本データ型の局所変数と時刻情報のみの出力とした理由は，局所変数における情報取得とフィールド変数における情報取得で用いる JVM TI 関数が異なることがあげられる．局所変数情報のためには，GetLineNumberTable 関数でその位置を，GetLocalInt 関数等でその値を取得する必要がある．それに対して，フィールド変数はヒープを辿ってオブジェクトを特定する必要がある．JVM TI の仕様上，指定されたオブジェクトから直接的または間接的に到達可能なオブジェクトの情報を取得できる FollowReferences 関数の使用が有効であると考えられるが，API が難解であるため，今回は機能としての実装は見送った．

表 1: 利用した JVMTI 関数の一覧とその処理内容

JVM TI 関数	処理内容
AddCapabilities	使用が許可された JVM TI 機能に対する権限の追加
ClearBreakpoint	指定された命令に対するブレークポイントの除去
GetCapabilities	現在所有している任意の JVM TI 機能を返却
GetClassMethods	指定されたクラスに含まれるメソッドの数とその ID のリストを返却
GetLineNumberTable	指定されたメソッドについてソース行番号のエントリから成るテーブルを返却
GetLocalVariableTable	指定されたメソッドの局所変数の情報を返却
GetMaxLocals	指定されたメソッドによって使用される局所変数のスロット数の返却
GetMethodDeclaringClass	指定されたメソッドを定義するクラスを返却
GetMethodName	指定されたメソッドの名前とシグネチャを返却
GetSourceFileName	指定されたクラスについてソース・ファイル名を返却
GetLocalInt	型が int，short，char，byte，boolean のいずれかである局所変数の値を取得
GetLocalLong	型が long である局所変数の値を取得
GetLocalFloat	型が float である局所変数の値を取得
GetLocalDouble	型が double である局所変数の値を取得
SetBreakpoint	指定された命令に対するブレークポイントを設定
SetEventCallbacks	イベントごとに呼び出される関数を設定
SetEventNotificationMode	イベントの有効化または無効化の設定

3.5 低侵襲性の実現方法

本節では主に，2.3 節で述べた対話的デバッガとの違いである実行を止めない方法について述べる．具体的な結果については 4 節で述べる．

従来の対話的デバッガにおいては，ブレークポイントで実行を停止していた．例えば，今回のデバッグ対象である Eclipse でコーディングされた Java プログラムについても，Eclipse デバッガを用いてデバッグするならばブレークポイントで停止してデバッグを行う．本研究では，ブレークポイントで実行を停止するという操作を行う代わりに，その地点での指定された情報を出力した．方法としては，JVM TI のイベント・コールバック関数である Breakpoint 地点において実行を停止する処理を行う代わりに，情報を出力する処理を行うだけである．これにより，ソフトウェアの実行を停止することなくソフトウェアの内部情報を取得することが出来る．

4 評価

本節では，作成したデバッガを実際のシステムに対して使用した結果と，デバッガの性能評価について記載する．

4.1 Web アプリケーションのデバッグへの適用

本研究で作成したデバッガを，Tomcat 上で動作する Servlet[5] に常駐させて実験を行った．このプログラムは，検索クエリに対して検索結果の一覧を返すというプログラムである．しかし，検索クエリが 1 日程度実行されなかった場合に応答時間が非常に遅くなる場合があるという問題を抱えていた．そこで，本研究で作成したデバッガを用いてデバッグを行った．使用目的として，障害の原因と考えられた実行開始時に一度だけ実行される準備の処理が複数回行われている可能性を調べるため，また各検索処理の応答時間の計測のために用いた．具体的にはある日に検索応答を評価して一度ログ収集を停止してから，3 日後に再びログを収集するという作業を行った．

その結果として，ログは正常に出力され目的としていた情報について収集することができ，障害の原因について分析することが出来た．今回の実験では，本ツールの特徴である長期間実行が続いている間ログ出力を止めておくこと，ログがサーバのディスク容量に影響を与えないこと，サーバプログラムにログ出力を加える必要がないことを確認することが出来た．

4.2 性能評価

実際に低侵襲性を実現できたかどうかを確認する．

4.2.1 DaCapo Benchmarks を用いたベンチマークテスト

本研究で作成したデバッグ用ファイルを，DaCapo Benchmarks[1] の DaCapo-9.12- bach のバージョンに用いて性能評価を行った．今回，DaCapo Benchmarks の 14 個のベンチマーク中，Windows 用にコンパイルした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．評価の方法としては，DaCapo Benchmarks のそれぞれのベンチマークをそのまま実行したものとデバッグ用ファイルを付随させて実行したものを比較し，実行時間のオーバーヘッドを確認した．今回の実験では出力によるオーバーヘッドではなく，ツールを引数に与えて実行させた際のオーバーヘッドを求める．そのため，ブレークポイントの設定ファイルは空のファイルを与える．

計測時間については，ベンチマークが出力した時間を用いた．DaCapo Benchmarks のそれぞれのベンチマークをそのまま実行したもので計測した結果を t_i，デバッグ用ファイルを

付随させて実行したもので計測した結果を t'_i とし，それぞれ 10 回ずつ実行する．オーバーヘッド $O(\%)$ を求める計算式は以下のようになる．

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

また，この計測は表 2 の環境で行った．用いたベンチマークとオーバーヘッドについての結果を表 3 に，箱ひげ図を図 2 に記載する．

表 2: 性能評価を行った環境

CPU	Intel(R) Xeon(R) CPU E5-1603 0 @ 2.80GHz 2.80GHz
OS	Windows10 Pro
メモリ	16.0GB
ディスク	HDD
Java のバージョン	Java(TM) SE Runtime Environment (build 1.8._121-b13) Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)

表 3: DaCapo Benchmark に対する実行のオーバーヘッド

ベンチマーク名	オーバーヘッド
avrora	2%
batik	3%
fop	1%
h2	3%
jython	4%
luindex	6%
lusearch	15%
pmd	2%
sunflow	24%
tradebeans	2%
tradesoap	8%
xalan	18%

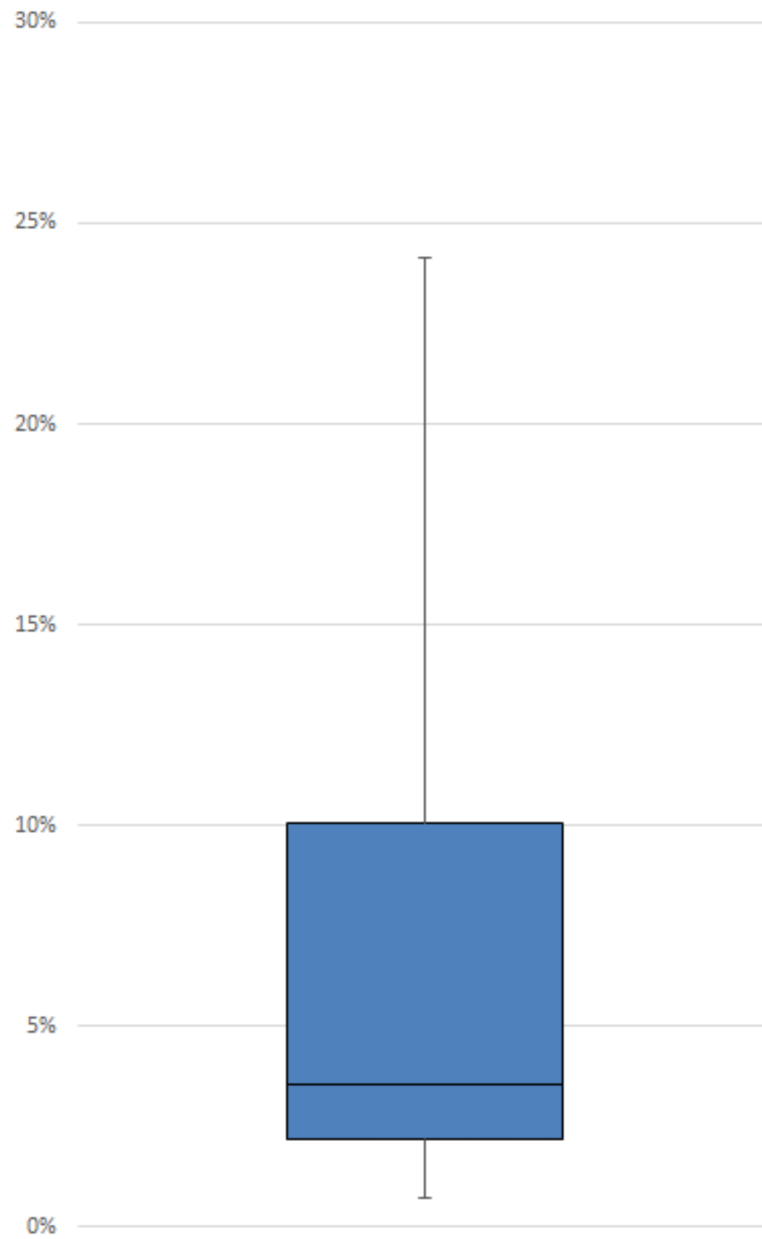

図 2: DaCapo Benchmarks に対する実行のオーバーヘッド

通常実行時との比較結果，オーバーヘッドは平均 7.4%，最大 24%であることが分かった．

平均を大きく超えた結果を示したベンチマークについて考察する．今回の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は，使用が許可された JVM TI 機能に対する権限の追加を行う AddCapabilities 関数によるものであると考えられる．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とが判明した．

4.2.2 ブレークポイントにおける出力によるオーバーヘッド

次にブレークポイントにおける出力行った際に，どの程度のオーバーヘッドが生じるかを計測した．計測の方法としては，付録に載せたソースコードを一部変更して用いる．まず，ソースコードを以下のように書き換える．

```
30      public static void main(String[] args) {
31           int N=1;
32           int A[]=new int[20*N];
33           int SIZE=20*N;
34           int a1[]=new int[5*N];
35           int a2[]=new int[5*N];
36           for(int i=0;i<5*N;i++) a1[i]=(int)(Math.random()*100000);
37           for(int i=0;i<5*N;i++) a2[i]=(int)(Math.random()*100000);
38           for(int i=0;i<5*N;i++) insert(A,SIZE,a1[i]);
39           int j=0;
40             long t1 = System.currentTimeMillis();
41           while(j<3*N){
42                int delete1=delete(A);
43               j++;
44           }
45           long t2 = System.currentTimeMillis();
46           for(int i=0;i<5*N;i++) insert(A,SIZE,a2[i]);
47           j=0;
48           long t3 = System.currentTimeMillis();
49           while(j<7*N){
50                int delete2=delete(A);
51                j++;
52           }
53           long t4 = System.currentTimeMillis();
54           System.err.println(t4-t3+t2-t1);
55      }
```

書き換えは出力回数を増減させるためと挿入ソートをある程度機能させ，出力にかかった時間を計測するために行った．まず，ソート前の配列に 0 から 100000 のランダムな整数が入るように 36 行目と 37 行目を変更した．そして，40 行目，45 行目，48 行目，53 行目のそれぞれ出力処理の開始時と終了時での時刻を取得し，それらを加減算することで 54 行目で出力にかかった時間を求めた．出力文の実行回数については，31 行目で定義されている N の値を 1000 から 5000 まで 1000 ずつ増加させて計測を行う．

まず，Java 通常実行時の出力時間の計測の仕方を説明する．出力文を 42 行目，50 行目の

表 4: ブレークポイントにおける出力によるオーバーヘッド

出力回数	10000	20000	30000	40000	50000
デバッガ未使用 [ms]	314.7	651.5	1118.5	1538.8	1942.6
デバッガ使用 [ms]	454.9	936.7	1415.2	1930	2432.9
オーバーヘッド [%]	45%	44%	27%	25%	22%

末尾に行番号を変えないようにそれぞれ以下のような形式で挿入する．

```
System.err.print("Main.java main line43 delete1[I] is ");
System.err.println(delete1);
```

```
System.err.print("Main.java main line51 delete2[I] is ");
System.err.println(delete2);
```

これにより測定を行い，54 行目の出力文を見て計測を行った．

次に，本ツール使用時の出力時間の計測の仕方を説明する．先程の Java の print 文の代わりに 43 行目と 51 行目の行頭にブレークポイントを仕掛ける．設定テキストファイルを以下のように記述，実行を行った．

```
Main.java delete1 43
Main.java delete2 51
```

これにより測定を行い，54 行目の出力文を見て計測を行った．計測した結果を t_i，デバッグ用ファイルを付随させて実行したもので計測した結果を t'_i とし，それぞれ 10 回ずつ実行し，オーバーヘッドも同時に求める．オーバーヘッド $O(\%)$ を求める計算式は以下のようになる．

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

また，この計測は表 2 の環境で行った．二つの実行の結果を表 4 と，図 3，図 4 にまとめる．

図 3 より，実行時間と実行回数は通常実行時，デバッガ使用時共に，線形であることが分かる．また，デバッガ使用時と通常実行時を比較したオーバーヘッドは実行回数と共に減少していることが分かる．

出力を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．したがって，このオーバーヘッドは出力に大きく依存していると言える．本ツールは出力を

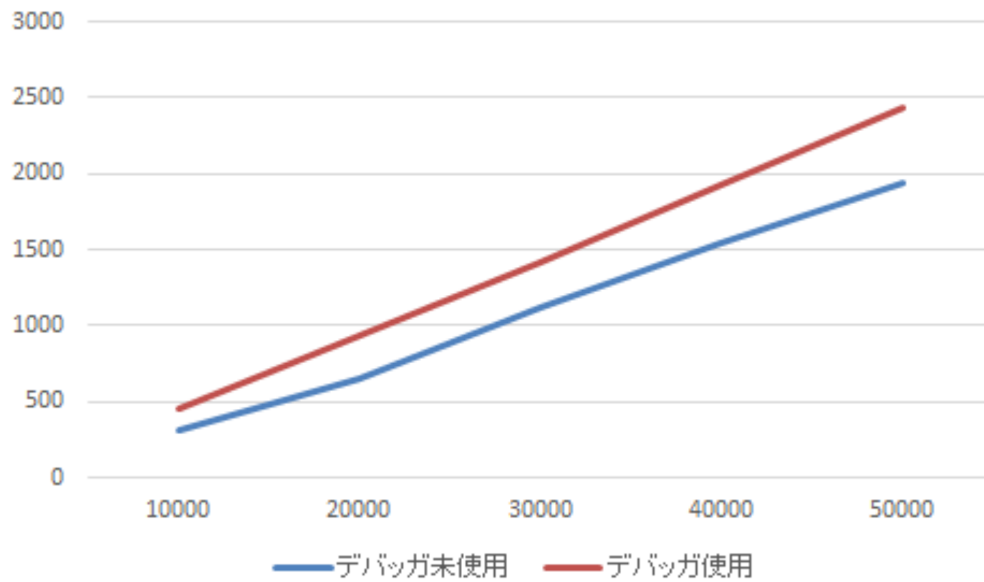

図 3: ログ出力命令の実行回数と実行時間 (ms)

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．

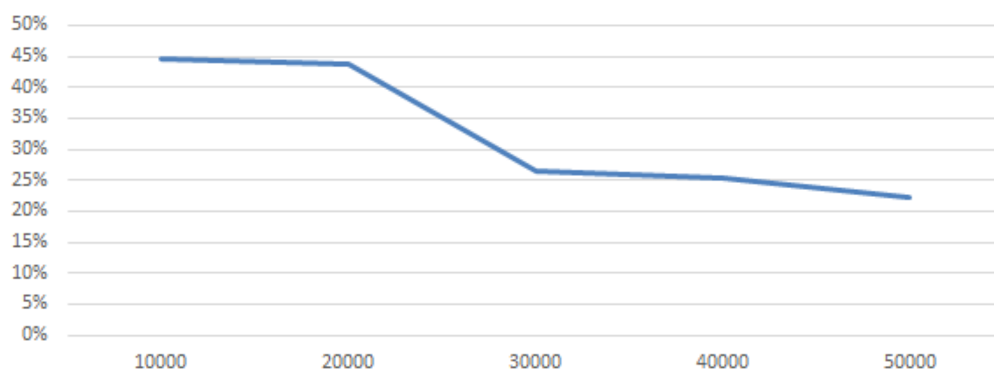

図 4: ログ出力命令の実行回数とオーバーヘッド (%)

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．実際に稼働している Web アプリケーションに対しても評価を実施した，また実行時間のオーバーヘッドを，DaCapo Benchmark を用いて計測したところ，平均 7.4%，最大 24%のオーバーヘッドでプログラムの実行を止めずに，デバッグを実行できることが確認できた．

今後の課題としては，機能面の充実はもちろん， データ収集に時間がかかった場合に収集を取りやめるといった影響の上限を定める方式を考え，ロギングよりも効率的に実行ログを取れるようにしていきたい．また，他の課題として，セキュリティを意識した開発があげられる．具体的には，本研究の特徴である Java プログラム実行時に本デバッグ用プログラムを引数として与えれば，外部のプロセスから自由に任意の変数にアクセスできる点は，セキュリティの観点から大きな問題がある．それは任意の変数情報の取得の際に，個人情報や機密情報が含まれるファイルにアクセスが行われる可能性があるという点である．したがって，今後の課題として，外部からのアクセスの際にパスワードを要求したり，アクセスできるファイルに制限をかけるなどの機能を実装する必要がある．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授には，研究においてたくさんの貴重な御意見を賜りました．多くの御助言を頂いた 井上 教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授には，研究において多くの御助言を賜りました．多くの御助言を頂いた 松下 准教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教授には，研究に関する適切な御指導及び御助言を賜りました．石尾助教授の御指導及び御助言のおかげで本論文を完成させることができました．石尾 助教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科 春名 修介 特任教授には，常に適切な御指導及び御助言を賜りました．多くの御助言を頂いた 春名 特任教授に心より深く感謝いたします．

島根大学大学院総合理工学研究科情報システム学領域 神谷 年洋 教授には，研究において多くの御助言を賜りました．多くの御助言を頂いた 神谷 教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 伊藤 薫 氏には，研究に関する相談に乗って頂き，また，評価実験の補助を行って頂くなど研究の様々な場面でご協力していただきました．心より深く感謝いたします．

最後に，御指導，御助言を頂き私を支えてくださった，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に，心より深く感謝いたします．

参考文献

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190, New York, NY, USA, October 2006. ACM Press.

[2] 千葉滋. アスペクト指向入門. 技術評論社, 2005.

[3] B. Siegmund. M. Perscheid. M. Taeumel. R. Hirschfeld. Studying the advancement in debugging practice of professional software developers. *In Proceedings of International Workshop on Program Debugging*, 2014.

[4] 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.

[5] K. Ito, T. Ishio, and K. Inoue. Web-service for finding cloned files using b-bit minwise hashing. *IWSC*, 2017.

[6] 大阪大学大学院情報科学研究科. 平成 26 年度 博士前期課程 入試問題 (A) 情報工学 http://www.ist.osaka-u.ac.jp/japanese/admission/files/2014mc_jyouhou.pdf, 2013.

[7] Oracle. JVM TI Reference. WWW page, 2007. http://download.oracle.com/javase/6/docs/platform/ jvmti/jvmti.html.

[8] 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.

[9] Andreas Zeller. *Why programs fail.* 第 2 版. O’Reilly Japan, 2012.

付録

二分探索を用いた挿入ソートプログラム

```
1  public class Main {
2       static int front=0,rear=0;
3       public static void insert (int A[],int SIZE,int d){
4            int p,left,right,m,i;
5            if(rear>SIZE-1){ System.out.println("Overflow !!\n");
6                 System.exit(1); }
7            p=-1;
8            if(front==rear)   p=front;
9            else{
10                left=front;right=rear-1;
11                while(left<right){
12                     m=(left+right)/2;
13                     if(A[m]==d) { p=m+1; break; }
14                     if(d<A[m]) right=m-1; else left=m+1;
15                     }
16                     if(p==-1){ if(A[left]>d)p=left; else p=left+1;
17                }
18           }
19           i=rear;
20           while(i>p){     A[i] = A[i-1];i--;}
21           A[p]=d; rear++;
22      }
23      public static int delete(int A[]){
24           int x;
25           if(front==rear){ System.out.println("Underflow !!\n");
26                System.exit(1); }
27           x=A[front]; front++;
28           return x;
29      }
```

```
30      public static void main(String[] args) {
31           int N=1;
32           int A[]=new int[20*N];
33           int SIZE=20*N;
34           int a1[]=new int[5*N];
35           int a2[]=new int[5*N];
36           for(int i=0;i<5*N;i++) a1[i]=10*i;
37           for(int i=0;i<5*N;i++) a2[i]=11*i;
38           for(int i=0;i<5*N;i++) insert(A,SIZE,a1[i]);
39           int j=0;
40           while(j<3*N){
41                int delete1=delete(A);
42                j++;
43           }
44           for(int i=0;i<5*N;i++) insert(A,SIZE,a2[i]);
45           j=0;
46           while(j<7*N){
47                int delete2=delete(A);
48                j++;
49           }
```