再現コンパイル手法を用いたJava JITコンパイラの問題判別

緒方 一則[†] 小野寺 民也[†] 河内谷 清久仁[†]
小松 秀昭[†] 中谷 登志男[†]

Java JIT コンパイラは，実行時情報を利用した高度な適応型最適化を行い，Java プログラムをより高速に実行する．反面，適応型最適化は，適用する最適化の組合せの増加と，最適化動作が入力データやスレッド実行順序などに依存することで動作の再現性を低下させ，JIT コンパイラの問題判別作業を困難にしている．この問題は，入力データ量や実行スレッド数の多い基幹システムで顕著になる．JIT コンパイラの問題判別には，通常，デバッグ用トレースを用いるが，基幹システムなどで，本番運用中に常時トレースを生成するのは非現実的である．そのため，デバッグ用トレースを取得するには，同一アプリケーションを別のマシンで再コンパイルする必要があるが，適応型最適化の動作は入力データやスレッド実行順序などに依存するので，再コンパイルしても同一のデバッグ用トレースが得られない．本論文では，再コンパイルでも本番運用中と同じ結果を再現する，再現コンパイル手法を提案する．本手法は，記録コンパイラと再現コンパイラと呼ばれる 2 種類のコンパイラを使用する．記録コンパイラは本番運用中に使用され，レポジトリと呼ばれるメモリ領域とシステムダンプを用いて，コンパイラへの入力をすべて保存する．我々の実験では，入力を保存することによる速度低下は無視できる程度，レポジトリに必要なメモリ量はコンパイラ生成コードの 10%以下であった．

Problem Determination for a Java JIT Compiler Using Replay Compilation

Kazunori Ogata,[†] Tamiya Onodera,[†] Kiyokuni Kawachiya,[†]
Hideaki Komatsu[†] and Toshio Nakatani[†]

The JIT compiler for Java has been greatly improved by supporting more advanced optimizations that are based on the adaptive optimization technique. On the other hand, the adaptive optimization increases variety of combinations of optimizations to be applied to a method and decreases the possibility to reproduce the same compilation process again because it depends on the input of the Java program and the execution order of threads. Thus, it makes the JIT compiler difficult to debug particularly in the mission-critical environment, where it usually processes large amount of input with many threads. The typical approach to debug the JIT compiler is to generate the trace of the compilation process, but it is impractical in the mission-critical environment. The only way to get the trace is to run the same Java application again remotely, but this does not necessarily generate the same output because the adaptive optimizations depend on the runtime environment that can be different from that at the time of the failure when it is compiled again remotely. In this paper, we propose a new technique, called replay JIT compilation, to recreate the same JIT compilation process offline remotely using two compilers, the state-saving compiler and the replaying compiler. The state-saving compiler is designed to run in the customer environment as part of the Java runtime and save into the repository in the memory all the state information necessary for the replaying compiler to reproduce the same compilation process later offline. Our experiment has shown that the overhead of state saving is negligible, and the size of the additional memory area for state saving is only 10% of the compiled code.

1. はじめに

Java は，企業の基幹アプリケーションの開発にも広く使われるようになってきた．Java JIT コンパイラは，実行時情報を利用する適応型最適化により，実行中の環境に最適なコードを生成して実行速度向上を図る[6),12),18)]．適応型最適化には，たとえば，頻出値のプロファイル結果に基づく特殊化や，コンパイラのメソッド解析結果を別メソッドのコンパイルに再利用するメソッド間解析などがある．これらの最適化の動作は，プログラムへの入力データ，プロファイル結果，

† 日本アイ・ビー・エム（株）東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

スレッド実行順序などに依存する．

コンパイラの問題判別では，通常，適用した最適化の動作を記録するデバッグ用トレースを用いる．しかし，基幹システム上でつねにトレースを生成しながら動作させることは，アプリケーション実行速度を大幅に低下させ，トレースの保存領域がアプリケーションで使用可能なメモリやディスク領域を減少させるので，非現実的である．したがって，デバッグ用トレースを取得するには，本番環境とは別の実行環境で同じアプリケーションを再実行する必要がある．しかし，適応型最適化は，最適化の組合せパターンを増加させると同時に，最適化動作が入力データやスレッド実行順序などに依存することで再現性を低下させるので，再実行によるコンパイラの問題判別はできない．

Java JIT コンパイラは Java 仮想マシン（Java VM）のコンポーネントとして動作し，Java バイトコードや，プロファイル結果など適応型最適化に使用する値など，コンパイラへの入力データを Java VM から取得する．Java VM は，コンパイラへの入力を Java VM の内部状態としてメモリ上に保持する．プロファイル結果やスレッドの実行順序の変化が Java VM の内部状態を非決定的に変更し，JIT コンパイラへの入力データが変化するので，JIT コンパイラの最適化動作が変化する．たとえば，今回作成したプロトタイプのベースとなる Java 実行時環境では，今回の評価に使用した Java プログラムすべてにおいて，10 回の実行のうち少なくとも 1 回は，プロファイル情報が変化したために，適用する最適化が変化した．したがって，再コンパイルによる問題判別を行うには，JIT コンパイラが使用した内部状態を復元して，コンパイラの動作を再現する手法が必要である．

本論文では，JIT コンパイラの動作を，本番環境とは別の環境で正確に再現する，**再現コンパイル**手法を提案する．この手法では，Java VM の内部状態を保存する**記録コンパイラ**と，保存された内部状態を用いて，記録コンパイラの動作を再現する**再現コンパイラ**の 2 種類のコンパイラを用いる．記録コンパイラは本番環境で実行され，JIT コンパイラへの入力データをすべて，**レポジトリ**と呼ぶメモリ領域とシステムダンプを用いて保存する．再現コンパイラは，本番環境とは別の環境で，問題判別のために使用される．JIT コンパイラ生成コードの問題で Java VM が異常終了すると，オペレーティングシステム（OS）は，プロセスメモリの内容を含むシステムダンプを自動的にディスクに保存する．本手法は，システムダンプを用いることで，レポジトリに保存する入力データ量を削減し，プログラム実行中のレポジトリのディスク書き出しを避けた．JIT コンパイラへの入力がすべてシステムダンプに含まれるので，システムダンプのみを受け取れば，本番環境とは隔離された環境でも問題を再現できる．

今回，AIX オペレーティングシステム上で動作する Java 実行時環境を用いてプロトタイプを作成し，記録コンパイラと再現コンパイラが同一のデバッグ用トレースを生成することを確認した．また，入力データをレポジトリに保存することによる実行速度低下は無視できる程度で，レポジトリのサイズは JIT 生成コードのサイズの 10%以下であった．

2. 関 連 研 究

実行時トレースを用いて，プログラムの非決定的動作を再現する手法は，実行順序を記録する手法と，入力データを記録する手法に分類される[15]．

実行順序を記録する手法は，スレッド間の同期イベントの順序を記録・再現することで，プログラムの動作を再現する手法である[8]．同期イベントとは，たとえばロックの取得・解放や，メッセージの送受信である．実行順序の記録を少ないオーバヘッドで実現する手法は，従来から広く研究されている[1),3),8),11)]．

入力データを記録する手法は，プログラムへの入力を記録・再生することで，再実行時の動作環境が記録時と違っても，同じ入力を取得して同じ動作をさせる手法である．Recap[13] は，プログラムへの入力データを記録・再生したが，当時の測定で使われた VAX-11/780 上でも，記録領域のサイズが膨大になった．jRapture[17] は，Java API のうちオペレーティングシステムとやりとりを行うものの動作を，引数と戻り値の記録・再生で再現するシステムである．彼らのプロトタイプでは，実行速度が 1/3 から 1/10 に低下した．ODB[9] は，記録・再生手法をデバッガに組み込んだ．このシステムは記録・再生手法が主目的ではないが，実行速度が最大 1/300 まで低下した．入力データを記録する手法は，このような大きなオーバヘッドがあるために，従来あまり使用されていなかった．

Java JIT コンパイラの場合，コンパイラ自体は，決定的な動作をする，シングルスレッドプログラムである．JIT コンパイラの非決定的動作の原因は，入力データである実行環境の内部状態が，プログラムの実行にともなって非決定的に変化することである．したがって，実行順序を記録する手法を採用するには，Java VM の内部状態を変化させる動作の実行順序を記録する必要がある．しかし，内部状態を変化させる動作とは，たとえば，オブジェクト生成やインスタン

ス変数へのアクセスにともなうクラスロードや外部参照の解決など，プログラム実行中に頻繁に実行される動作なので，それらをすべて記録するのはオーバヘッドが大きく非現実的である．そこで我々は，入力データを記録する手法を採用し，同時に，記録すべきコンパイラへの入力を削減してオーバヘッドを減らす手法を提案する．

Dynamic deoptimization [4] や Tikir ら [19] も，JIT コンパイラを使用する実行環境における，プログラムのデバッグ手法を提案している．これらの手法は，デバッグしやすい JIT 生成コードを得るために，プログラムを再コンパイルしている．これらの手法では，JIT コンパイラには問題がないと仮定して，プログラム自体のデバッグを容易にする手法なので，JIT コンパイラのデバッグには適用できない．

システムダンプを問題判別部門に送付すると，プログラム実行中のメモリに保持されていた機密情報や個人情報などの重要情報が漏洩する可能性がある．このような問題を回避するために，システムダンプから重要情報を除去する方法 [2] なども研究されている．重要情報はデータの一部として保持されている場合が多いので，Java プログラムでは，Java オブジェクトに保持されている可能性が高い．JIT コンパイラの動作再現では，通常，バイトコードやプロファイル結果など，Java VM の内部状態のみを使用し，Java オブジェクトの内容は不要なので，既存の手法を適用してシステムダンプから Java オブジェクトの内容を除去しても，JIT コンパイラの動作を再現できる．別の回避方法としては，ユーザに再現コンパイラを実行してもらい，出力されたデバッグ用トレースのみを送付してもらうこともできる．

3. 再現コンパイル手法

再現コンパイル手法は，JIT コンパイラへの入力データをレポジトリに保存する記録コンパイラと，レポジトリに保存されたデータを使用してコンパイラの動作を再現する再現コンパイラの，2 種類のコンパイラを使用する．これらのコンパイラの典型的な使用例は，記録コンパイラを，ユーザアプリケーションが稼働する本番環境で使用し，再現コンパイラを，JIT コンパイラの問題判別を行う製品サポート部門で使用する．別の使用例として，JIT コンパイラの開発者が，再現性の低い問題のデバッグに使用することもできる．テストケース実行は記録コンパイラを使用し，エラーになったテストケースのデバッグに再現コンパイラを使用すれば，再現性が低い場合でも，問題再現のためにテストを何度も再実行する必要がなくなる．

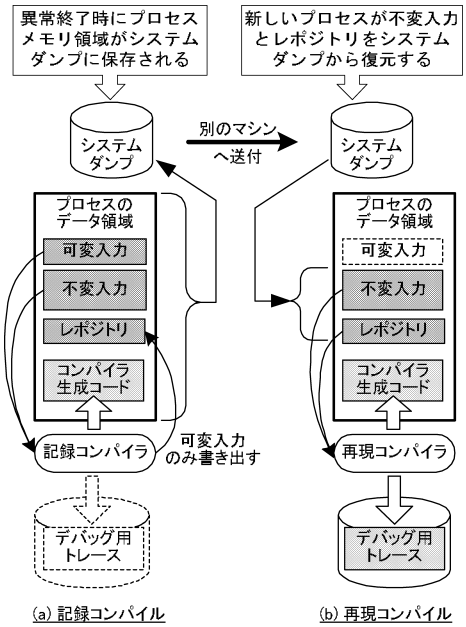

図 1 再現コンパイルの概要
Fig. 1 Replay JIT compilation.

記録コンパイラは，メソッドごとにレポジトリを実行環境プロセスのアドレス空間内に割り当て，Java VM プロセスが異常終了したときに，レポジトリが自動的にシステムダンプに保存されるようにする．システムダンプを使用することで，ユーザアプリケーション実行中にレポジトリをディスクに書き込むことによるオーバヘッドを避けられる．

一方，レポジトリは実行環境プロセスのアドレス空間内に割り当てるので，十分小さくなければならない．そこで，JIT コンパイラが最適化に使用した内部状態のうち，コンパイル後に変更される可能性がある状態だけをレポジトリに保存し，変更されない状態はシステムダンプに保存された値を使うことで，レポジトリのサイズを削減する．JIT コンパイル後に変更される可能性のある内部状態を**可変入力**，変更されない内部状態を**不変入力**と呼ぶ．

再現コンパイラは，システムダンプから，不変入力とレポジトリを復元し，そのレポジトリから可変入力のコンパイル時の値を取得する．**図 1** に，再現コンパイル手法の概要を示す．

3.1 JIT コンパイラの入力データ

Java JIT コンパイラの入力データは，ソースコードである Java バイトコードを含む，Java VM の内部状態である．Java バイトコードや文字列定数などの不変入力は，プログラムの実行中に値が変化しないの

表 1 Java JIT コンパイラの可変入力とその値が変化する要因
Table 1 The variable input used by the Java JIT compiler and the reasons for which its value may change.

要因	入力データが変化する理由	可変入力	可変入力を利用する最適化
組み込みのマルチスレッドサポート			
	クラスは最初にアクセスされたときに初期化される．また，実行回数の多いメソッドから JIT コンパイルされる．これらの順序は，スレッドの実行順序の影響を受ける．	• 初期化済みクラスの集合 • JIT 生成コードのアドレス • JIT コンパイラの解析結果	• コンパイル時にすでに初期化されていたクラスのイニシャライザを呼び出すコードは生成しなくてよい． • コンパイル中のメソッドからの呼び出し先メソッドがすでにコンパイルされていれば，その JIT 生成コードへ直接分岐するコードを生成できる． • JIT コンパイラがメソッド間解析を行い，それを保存している場合，その解析結果を利用して，より最適化されたコードを生成できる．
クラスの動的リンク			
	クラスは実際にアクセスされたときにはじめてロードされる．ロードされたクラスからの外部参照も，参照されたときにはじめて解決する．	• クラス階層情報 • 解決済み外部参照の集合	• クラス階層情報を利用して，仮想呼び出しのデバーチャライズを行う[6]． • 参照する外部参照がすでに解決済みならば，その外部参照を解決するためのコードを生成しなくてよい．また，メソッドへの参照が解決済みならば，呼び出し先メソッドをインラインの対象にできる．
オンライン・プロファイラ			
	オンライン・プロファイラを用いる場合，結果は Java VM 実行中更新され続ける．最適化レベルも，プロファイル結果を考慮して決定する．	• オンライン・プロファイラ収集データ • 最適化レベル	• 実行頻度の高いパスをより最適化したり，出現頻度の高い変数値に特殊化したコードを生成する． • 最適化レベルは，適用する最適化の集合を選択する．
実行環境の構成			
	マシンのハードウェア，ソフトウェア構成を含む，実行環境の構成情報で，起動時に決まる．	• プロセッサの構成（種類，数，キャッシュサイズなど） • OS の種類とバージョン • 実行環境起動オプションと環境変数の値	• JIT コンパイラは，実行環境に合わせて最適化したコードを生成する．たとえば，単一プロセッサのマシン上では，ロックの取得・解放コードは，ほかのプロセッサの動作を考慮しない，単純で速いコードを生成する．

で，コンパイラが使用した後で Java VM から削除されなければ，コンパイル時と同じ値がシステムダンプに自動的に保存される．記録コンパイラは，使用した不変入力のアドレスを記録して，再現コンパイラがシステムダンプから探せるようにすれば，不変入力の値を保存する必要はない．

Java の仕様[10] では，Java VM は，ロードしたクラスをアンロードしてもよい．しかし，クラスをアンロードしないように Java VM を修正しても，メモリ不足などの問題が起きる可能性は低い．なぜならば，クラスをアンロードできるのは，そのクラスをロードしたクラスローダを用いてロードした，すべてのクラスがアンロード可能な場合だけ[10] なので，実際のアプリケーションでは，クラスのアンロードはほとんど起きないからである．

Java VM の内部状態は，実行環境の起動ごとだけではなく，プログラム実行中にも値が変わる可能性がある．したがって，JIT コンパイラの動作を正確に再現するには，JIT コンパイラのコード中の，可変入力を取得したすべての場所で，その値をレポジトリに保存する必要がある．

Java JIT コンパイラなどの動的コンパイラの可変入力は，変化の要因ごとに**表 1** に示す 4 種類に分類できる．このうち，最初の 2 種類は Java 言語の仕様から生じるものであるが，Java 以外でも同じ特徴を持つ言語の動的コンパイラの入力ならば同様に変化する．残りは動的コンパイラの特性によるもので，言語によらない．プログラムが非決定的動作をする原因には，ほかに，プロセス間通信，デバイスからの割込み処理，および，現在時刻に依存する動作があるが，それらは動的コンパイラでは使用されないので，コンパイラの入力が変化する要因はこれですべてである．

本手法では，不変入力をシステムダンプを用いて保存することで，コンパイラへの入力データのうちサイズの大きい部分の保存を避け，入力データ記録のオーバヘッドを削減した．

3.2 動作再現のスコープ

バイトコードなどの不変入力は，そのアドレスを再現コンパイラが参照可能なように保存する．逆に，アドレスを保存できない場合，不変入力であってもレポジトリに保存する必要がある．たとえば，コンパイラがバイトコードや文字列定数を関数呼び出しで取得している場合，関数呼び出しの戻り値は永続的なメモリ上には残らないので，戻り値をレポジトリに記録する必要がある．さらに，関数呼び出しの戻り値をレポジトリから検索可能にするためには関数呼び出しの引数も保存する必要があるので，保存サイズはコンパイラの入力データのサイズより大きくなる．このように，レポジトリサイズは入力データの取得方法に依存する．

一般に，決定的に動作するコンポーネントならば，同じ入力データ列を与えれば，同じ出力データ列を生成する．JIT コンパイラが別のコンポーネントから入力を取得している場合，JIT コンパイラとそのコンポーネントをグループ化し，そのグループ全体の動作を再現して，JIT コンパイラの動作を再現することもできる．このグループを**再現スコープ**（Replay Scope）と呼ぶ．もし，JIT コンパイラ単体への入力のサイズより，このグループへの入力の総和の方が小さければ，グループへの入力を保存・復元することで，レポジトリサイズを削減できる．

図 2 に，JIT コンパイラが Java VM の関数を呼び出して入力データを取得し，Java VM は JIT コンパイラが必要とする内部状態をメモリから直接ロードする場合に，再現スコープを用いてレポジトリのサイズを減らす例を示す．この例では，RS1 と RS2 の 2 通りの再現スコープが考えられる．RS1 では，すべての入力を Java VM から関数呼び出しで取得しているので，可変入力と不変入力の両方を，引数とともに保存する必要がある．RS2 では，Java VM は可変入力を保存するだけでよい．したがって，RS2 を再現スコープとすることで，レポジトリのサイズを削減できる．

一般に，関数呼び出しの結果を保存するような再現スコープを選択すると，不変入力も保存が必要になり，レポジトリサイズが大きくなる．一方，再現スコープを広げて多くのコンポーネントを再現スコープに含めると実装のコストが増える．

3.3 再現コンパイル

再現コンパイラは，システムダンプに保存された，Java VM のデータ領域とレポジトリから，再コンパイルに必要な入力データを取得し，記録コンパイラと同一の動作を再現して，同一のデバッグ用トレースを生成する．再現コンパイラは記録コンパイラと同一のコンパイル処理を行うので，デバッガでステップ実行し，変数の内容を調査することもできる．

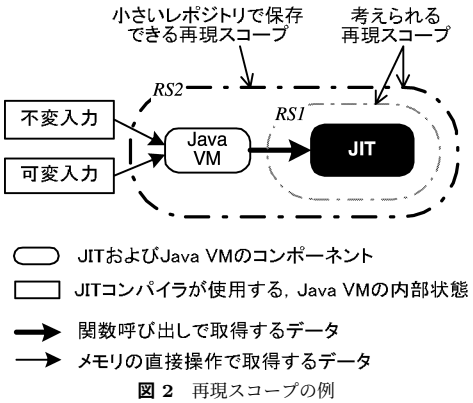

図 2 再現スコープの例
Fig. 2 An example of the replay scopes.

3.3.1 再現コンパイラの起動

再現コンパイラは，ブートストラップ用のプログラム内で動作する．このプログラムは，通常の Java VM でも，必要なインタフェース関数を実装した別のプログラムでもよい．通常の Java VM を使用する場合，再現コンパイラは，ブートストラップ用 Java VM の実行環境の内部状態は使用しない．

Java VM 以外のプログラムを使用する場合，そのプログラムは，再現コンパイラが入力データを取得するために使用するインタフェース関数をすべて実装する必要がある．この方法の利点は，記録コンパイラと再現コンパイラを，異なる OS 上で動かすことができることである．たとえば，問題判別チームは，サポートしている全 OS の実行環境を用意する必要がなくなる．ただし，2 つのコンパイラは，OS の違いによらず，同じ動作で最適化やコード生成を行う必要がある．

再現コンパイラは，指定されたメソッドごとに再コンパイルを行い，デバッグ用トレースを生成する．メソッドを再現コンパイルする順序は任意でよい．たとえば，本番実行時のコンパイル順の逆順でもよい．

3.3.2 コンパイラへの入力データの復元

記録コンパイラが入力データを関数呼び出しで取得する場合，再現コンパイラを実装するには，その関数を，コードを実行するのではなく，与えられた引数に対応する結果をレポジトリから検索するように修正する．記録コンパイラが，メモリ上のデータを直接操作して取得している場合は，レポジトリや，システムダンプに保存されたデータ領域から変数を復元し，それを操作するように，コンパイラを修正する．

復元された変数にポインタが格納されている場合，そのポインタは，すでに終了した，記録コンパイラを

実行したプロセス（記録プロセス）内のアドレスを指しているので，再現コンパイラを実行するプロセス（再現プロセス）内で同じアドレスに同じデータが存在する保証がない．したがって，ポインタ変数を復元した場合は，再現プロセス内の対応するアドレスに変換してから，メモリを参照する必要がある．

あるいは，記録プロセスのヒープ領域を，再現プロセス内で同じアドレスに復元して，ポインタ参照の修正を不要にする方法もある．バイトコードやレポジトリはヒープ領域に保存され，スタック領域に再現コンパイルに必要なデータは存在しないので，ヒープ領域だけを同じアドレスに復元すればよい．記録プロセスと同じアドレス範囲を確保するために，ブートストラップ用プログラムの起動初期の，ヒープ領域がほとんど使用されていない状態で復元作業を行う．この方法の問題点は，実現可能性が対象 OS のプロセスモデルに依存することである．また，状態保存と再現コンパイルを，原則として同じ OS 上で行う必要がある．

4. 実装方法

本章では，記録コンパイラと再現コンパイラを，AIX 上で動作する次期バージョンの Java VM をベースに実装した場合の詳細について述べる．

4.1 記録コンパイラ

使用した JIT コンパイラは，Java VM や，JIT コンパイラのサポート関数から入力データを取得し，適応型最適化を行う．入力データの取得は，関数呼び出しか，メモリ上のデータの直接参照のいずれかで行う．

このプロトタイプでは，コンパイルするメソッドごとにレポジトリ用のメモリ領域を確保し，サイズが最小となるデータ構造で入力データを保存する．**表 2** に，入力データごとの，レポジトリに保存する値の例を示す．レポジトリは，入力値のキャッシュとしても働き，そのコンパイル中は偶然値が変わらなかった場合に，同じ入力値を重複して保存することを避ける．

レポジトリは，JIT 生成コードのアドレスと対応づけて管理する．これにより，実行頻度が非常に高いメソッドを，最適化レベルを上げて再コンパイルする場合でも，その最適化レベルでコンパイルしたときのレポジトリを，生成コードのアドレスから検索できる．

バイトコードがすべてシステムダンプに保存されるように，Java のクラス・アンロード機能は停止した．別の解決法としては，クラスをアンロードして，使用可能なクラスのリストからは削除するがものの，メモリは解放しないでおく方法もある．

表 2 JIT コンパイラの入力データの例
Table 2 An example of the values to be saved in a repository.

コンパイラの入力	レポジトリに保存する値
バイトコード	メソッドのアドレス
文字列定数	文字列定数のアドレス
クラスごとの，初期化が完了したかどうかの状態	クラスのアドレスと，初期化が完了したかどうかのフラグ．
コンパイル済みメソッドの生成コード・アドレス	コンパイル中メソッドから直接呼び出した，呼び出し先メソッドの生成コード・アドレスのリスト
キャッシュされた，メソッド間解析結果	脱出解析結果をキャッシュしているクラスのリスト
解決済み外部参照のリスト	外部参照の解決済み/未解決を示すビットマップ
クラス階層情報	指定した仮想メソッドが，その時点で Java VM 中に存在する実装が 1 個だけかどうかを調べる関数の，引数と結果．結果は，単一の実装の場合はそのメソッドのアドレスを返すものとする．
サンプリング・プロファイラの結果	プロファイル結果のデータ構造から取得した値を検索するためのキーと，取得した値
最適化レベル	その値
実行環境の構成	Java VM と JIT コンパイラのバージョン，プロセッサのモデルと数，OS の種類とバージョン
Java VM 起動オプションと環境変数	起動オプションや環境変数の解析結果を保持するデータ構造のアドレス

4.2 レポジトリサイズの削減

本節では，レポジトリのサイズを削減するために組み合わせて適用した手法の詳細について述べる．

4.2.1 再現スコープの選択

図 3 に，我々が使用した JIT コンパイラの構成の概要を示す．この構成のコンパイラで，レポジトリサイズが最小となる再現スコープの選択について述べる．各コンポーネントの概要を**表 3** に示す．JIT コンパイラ COMP は，CLSMGT および CHA から関数呼び出しを用いて入力を取得する．PROF からは，プロファイラが更新するデータ構造を直接参照して入力データを取得する．COMP が取得する入力は，バイトコードや文字列定数などの不変入力，クラスとメソッドの内部状態（表 2 の，クラスの初期化済み状態から解決済み外部参照のリストまで），クラス階層情報，プロファイラの結果とその結果に依存する最適化レベル，実行環境の情報（構成や Java VM の起動オプションなど）の，5 種類の入力データを使用する．再現スコープの境界となりうる点は，計 8 カ所（Bv1, Bv2, Bv3, Bh1, Bh2, Bp1, Bp2, Be1）である．

まず，JIT コンパイラ本体（COMP）のみを含む再現スコープを RS1 とし，RS1 に CLSMGT を加え

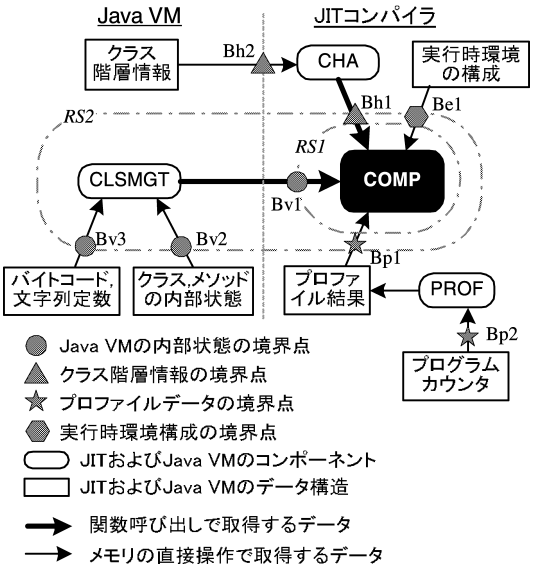

図 3 JIT コンパイラを構成するコンポーネント
Fig. 3 The components in our JIT compiler.

表 3 各コンポーネントの概要
Table 3 Description of each component.

コンポーネント	概要
COMP	コンパイラ本体（最適化とコード生成部分）
CLSMGT	ロード済みクラスを管理する Java VM のコンポーネント．初期化済みクラスや解決済み外部参照の記録も行う．
PROF	サンプリング・プロファイラ
CHA	クラス階層解析を行い，仮想メソッド呼び出しのデバーチャライズに必要な適用集合を算出するコンポーネント

た再現スコープ RS2 とレポジトリサイズを比較する．RS1 では，COMP が Bv1 を通過する入力をすべて CLSMGT から関数呼び出しで取得するので，不変入力であるバイトコードや文字列定数もレジストリに保存する．RS2 では，Bv2 を通過する，可変入力であるクラスとメソッドの内部状態のみを保存し，Bv3 を通過する不変入力はシステムダンプから復元する．その結果，5 章に示すように，RS1 のレポジトリサイズはコンパイラ生成コードのサイズの約 1.6 倍であったが，再現スコープを RS2 とすることで，レポジトリサイズがコンパイラ生成コードの約 20%になった．

次に，RS2 と，RS2 に CHA を加えた場合の再現スコープ RS2+CHA を比較する．これらの再現スコープのレポジトリサイズの差は，Bh1 および Bh2 で保存するクラス階層情報の差である．COMP は，クラス階層情報を利用した最適化のために，指定したメソッドがデバーチャライズ可能かどうかを CHA の関数を呼び出して調べる．Bh1 で入力を保存する場合，関数呼び出しの引数と戻り値を保存し，Bh2 の場合は，クラスロードと，関数呼び出しの順序を記録する．関数呼び出しの順序を記録するための，引数とポインタまたはタイムスタンプの保存サイズは，引数と戻り値の保存サイズと等しいので，RS2+CHA は，クラスロードの順序を記録する分だけレポジトリサイズが大きくなる．Bh2 で記録する場合のもう 1 つの問題は，クラス階層情報復元のために，すべてのクラスロードの記録を保持する必要があり，次項の方法による古いデータの削除が不可能になることである．したがって，RS2 の方が再現スコープとして適当である．

最後に，RS2 と，RS2 に PROF を加えた場合の再現スコープ RS2+PROF の，を比較する．この場合，レポジトリサイズ，Bp1 および Bp2 で保存するプロファイル結果のサイズの差である．プロファイラは，短い間隔（たとえば 10 msec など）で，プログラムカウンタを参照し，実行中のコードを解析して，プロファイル結果を更新する．したがって，Bp2 で入力を保存する場合，レポジトリサイズが膨大になる．一方，コンパイラがプロファイル情報を参照するのは，実行頻度が非常に高いメソッドをより最適化レベルを上げて再コンパイルする場合だけなので，Bp1 で保存する入力は小さい．したがって，Bp1 を境界とする RS2 の方が再現スコープとして適当である．

以上の考察より，図 3 の構成でレポジトリサイズが最小になる再現スコープは RS2 であることが分かる．一方，RS2 では COMP に加えて CLSMGT も修正する必要があり，RS1 よりも実装の作業量は多い．

4.2.2 信頼度による選択

レポジトリサイズの削減が重要である場合，コンパイラの問題が顕在化する可能性はメソッドごとに偏りがあることを利用し，コンパイラの問題が存在する可能性が低いと判断したメソッドはレポジトリを破棄しても問題判別への悪影響はないと想定して，レポジトリを選択的に破棄することもできる．コンパイラ生成コードに問題が存在する可能性の低いメソッドを**信頼度**の高いメソッドと定義し，そのメソッドのレポジトリを破棄すれば，レポジトリの総量を削減，あるいは，適当に設定した制限値以下にできる．

信頼度を決める要因には，たとえば，コンパイル対象メソッドの複雑さや，そのメソッドがコンパイルされてからの時間がある．コンパイラの問題が起きる可能性は複雑なメソッドをコンパイルする場合の方が高く，また，Java VM を異常終了させるような問題は，ほとんどの場合，コンパイル後数回の実行のうちに発

生するからである．複雑さの尺度には，メソッドのサイズやコンパイル時間などを利用できる．また，信頼度を決定する要因には，メソッドのカテゴリもある．多くのプログラムで共通に使われるメソッドは，JITコンパイラの開発中に十分にテストされていると考えられる．たとえば，`java.lang` パッケージに含まれるシステムクラスのメソッドは多くのプログラムから呼び出されコンパイルされているので，本番運用中に新たな問題が起きる可能性は低いと考えられる．

これらの要因で決定した信頼度が適当なしきい値より高いメソッドのレポジトリを順次破棄することで，レポジトリの総量を削減できる．破棄する順序は，信頼度を決定する複数の要因を考慮して決めてもよい．

ただし，コンパイルされてからの時間で信頼度を決める場合のように，コンパイラ生成コードが何度か実行されたと想定してそれ以降の信頼度を決める場合，異常時のみ実行されるような例外処理パスに存在する問題の場合，想定がはずれて必要なレポジトリを破棄し，問題判別が難しくなる可能性がある．したがって，信頼度によるレポジトリの破棄を行うかどうか，どの要因を用いて破棄を行うかは，確実な問題判別への要求と，そのために消費されるメモリ量の削減への要求とのトレードオフとなる．

4.2.3 レポジトリの圧縮

入力データのデフォルト値を決めておくと，そのデフォルト値と同じ入力データをレポジトリから削除して，レポジトリサイズを削減できる．再現コンパイラが，必要なデータがレポジトリに保存されていない場合はデフォルト値を取得したものとして動作すれば，記録コンパイラと同じ入力値を使用することになる．

このデフォルト値を，コンパイル処理がより安全な最適化を行う値に決めておくと，本手法を拡張した場合に不要な問題を避けられるので便利である．本手法では，記録コンパイラと再現コンパイラが同一の動作をすることを目標にしているが，再現コンパイル時にコンパイラのオプションを変更して問題の絞り込みを行う手法も考えられる．この場合，記録コンパイラが取得しなかった入力データを，再現コンパイラが必要とする可能性がある．デフォルト値をより安全な最適化を行う値にすれば，オプションを変えたことで新たな問題が発生する可能性が低くなる．

記録コンパイラは，ここまでの手法を適当に組み合わせて適用した後，通常の圧縮技術（zlib[7] など）を適用すれば，レポジトリサイズをさらに削減できる．

4.3 再現コンパイラ

今回のプロトタイプは，ブートストラップ用 Java VM の初期化中に，システムダンプから状態保存プロセスのデータ領域を復元してレポジトリを検索し，必要最小限の初期化後，再現コンパイラを呼び出す．

システムダンプに保存されたプロセスのデータ領域は構造を持たないバイナリデータ列で，ブートストラップ用プログラムは，その中からレポジトリを発見し取り出す必要がある．そのため，記録コンパイラは，すべてのレポジトリを 1 個のポインタ（ルートポインタ）から到達可能になるような，たとえば連結リストなどのデータ構造で管理する．そして，ルートポインタを，ヘッダとフッタにシグニチャを持つ構造体（アンカ構造体）に保存する．再現コンパイラは，システムダンプの中でヘッダとフッタのシグニチャを検索する．見つかれば，構造体のサイズが正しいことなどを検証し，ルートポインタを取り出すことで，すべてのレポジトリを取り出すことが可能になる．

アンカ構造体は，記録コンパイラが作成中のレポジトリ（現行レポジトリ）へのポインタも保持する．このポインタは，コンパイル中は現行レポジトリのアドレスを保持し，コンパイル終了時に NULL にする．このポインタにより，プロセスの異常終了がコンパイル中か否かをシステムダンプだけで判定できるので，問題判別をより容易にする．

5. 評　　価

再現コンパイル手法を，レポジトリのサイズと，実行時間に対するオーバヘッドで評価した．測定に使用したコンパイラは，3 章で述べたプロトタイプコンパイラである．すべての評価において，レポジトリサイズはコンパイラ生成コードに対する比率で評価した．**表 4** に実行環境を，**表 5** に測定したプログラムを示す．

また，記録コンパイラを，表 4 のそれぞれのマシンで実行して取得したシステムダンプを用いて，再現コンパイラを表 4 のそれぞれのマシンで実行して最適化動作を再現した場合の，9 通りの組合せすべてにおいて，記録コンパイラと再現コンパイラで，同一の生成コードを出力することも確認した．

5.1 レポジトリのサイズ

図 4 に，再現スコープの選択とレポジトリサイズの関連の評価結果を示す．評価した再現スコープは，4.2.1 項の RS1 と RS2 である．信頼度の高いメソッドのレポジトリの削除や，zlib による圧縮はしていない．

レポジトリサイズは，RS1 では最大でコンパイラ生成コードのサイズの 2.7 倍，幾何平均で 1.6 倍であった．RS2 は，最大でコンパイラ生成コードの 33%，幾何平均で 20.0%であった．RS1 では，不変入力も，関

表 4 評価環境
Table 4 Configuration of the tested machines.

	マシン 1	マシン 2	マシン 3
CPU	POWER3, 2-way SMP	POWER3, uni-processor	POWER4, 4-way SMP
RAM	768 Mbytes	768 Mbytes	2 Gbytes
OS	AIX 4.3.3	AIX 4.3.3	AIX 5.1

表 5 評価した Java プログラム
Table 5 Evaluated programs.

Embedded Caffeinemark	CaffeineMark [14] のテストのうち，グラフィックを使わないベンチマークのスコアの幾何平均
jBYTEmark	jBYTEmark の INT index と FP index の幾何平均
SPECjvm98	SPECjvm98 [16] のベストスコア
XMLparser	XMLparser for Java [5] で，パッケージに含まれるサンプル XML ファイルの解釈にかかる時間
SPECjbb	SPECjbb-2000 [16] のスコア

数呼び出しの引数とともにレポジトリに保存するので，レポジトリサイズが大きい．レポジトリサイズの削減には，関数呼び出しの戻り値の記録を避けるように，再現スコープを選択することが重要である．

図 5 に，信頼度によるレポジトリの破棄と，zlib による圧縮を適用した場合の評価結果を示す．この測定の再現スコープは RS2 である．“no filter” は，信頼度によるレポジトリの破棄を行わない場合，“filter system classes” は，システムクラス（`java.lang`, `java.util`, `java.math`, `java.io` で始まる名前のクラス）のメソッドは信頼度が高いものとして，レポジトリを破棄した場合の結果である．いずれの場合も，コンパイルされてからの時間など，クラス以外の要因での信頼度の判定は行っていない．信頼度による破棄により，レポジトリサイズを 20%削減できた．レポジトリ数は平均で 30%削減できたが，レポジトリサイズの削減率がそれより少ないのは，システムクラスのメソッドのレポジトリサイズが，全メソッドの平均サイズよりも小さいためと考えられる．

zlib による圧縮により，レポジトリサイズはさらに約半分になった．信頼度による破棄と組み合わせることで，幾何平均で 60.4%の削減ができた．その結果，多くのユーザで受け入れられると考えられる，コンパイラ生成コードの 10%を下回ることができた．

5.2 コンパイル時間

図 6 に，記録コンパイラのコンパイル時間を，修正前のベース・コンパイラのコンパイル時間との比率で示す．“No zlib” は圧縮を行わない場合の結果，“zlib” は圧縮を行った場合の結果を示す．どちらの場合も，信頼度による破棄は行わなかった．

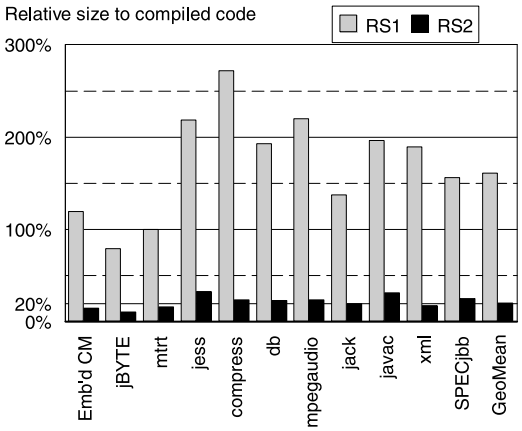

図 4 再現スコープごとのレポジトリサイズ
Fig. 4 The size of reporitories by replay scopes.

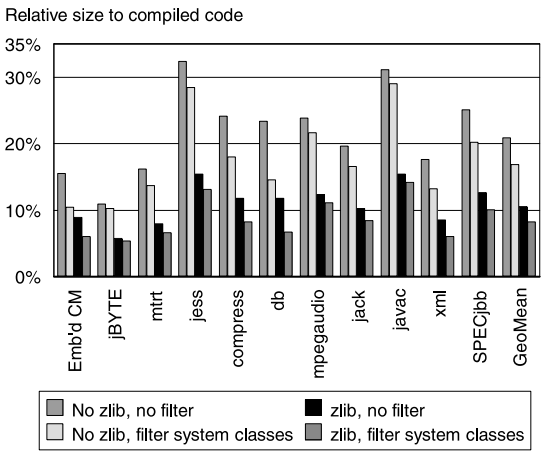

図 5 信頼度による破棄と圧縮によるレポジトリサイズの削減
Fig. 5 The size of repositories with compression and filtering.

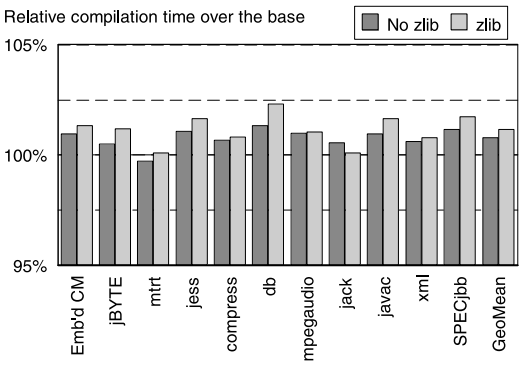

図 6 コンパイル時間
Fig. 6 Compilation time increases when saving inputs.

コンパイル時間の増加は，最大でそれぞれ 1.3%と 2.3%，幾何平均は，それぞれ 0.8%と 1.2%で，無視できるほど小さかった．

5.3 実行速度

記録コンパイラのオーバヘッドは，入力データをレポジトリに保存する処理だけである．そのオーバヘッドは，コンパイル中だけで発生し，コンパイラ生成コードの実行中のオーバヘッドはない．

コンパイル時間の増加が小さかったことから予測できるように，実行速度の低下もごくわずかだった．速度低下の幾何平均は約 1%であった．

6. おわりに

本論文では，記録コンパイラと再現コンパイラの 2 つのコンパイラを用いて，本番環境でのコンパイル処理を別の環境で正確に再現する，再現コンパイル手法を提案した．本手法では，記録コンパイラを本番環境で常用し，コンパイラへの入力データをすべて，レポジトリと呼ぶメモリ領域とシステムダンプを用いて保存する．再現コンパイラは，保存された入力データを用いて，記録コンパイラの動作を再現し，問題判別を容易にする．システムダンプを用いることで，レポジトリには可変入力のみを保存することでレポジトリサイズを削減し，さらに，レポジトリを明示的に保存する必要をなくした．我々は，AIX 上で動作するプロトタイプを作成し，本手法によりコンパイル処理を正確に再現できることを確認した．また，そのプロトタイプを用いて評価を行い，実行時間に対するオーバヘッドは無視できる程度に小さく，入力データを保存するために必要なメモリは，コンパイラ生成コードの約 10%以下になった．

謝辞 再現コンパイル手法の考案にあたって貴重なご意見をいただいた，日本 IBM の砥上章彦氏とカナダ IBM の Trent Gray-Donald 氏，本手法考案の発端となる問題提起をしていただいた日本 IBM の清水敏正氏に深く感謝します．また，本論文に対して助言をいただいた，日本 IBM 東京基礎研究所システムズ・グループのメンバに感謝します．

参考文献

1) Bacon, D.F.: Hardware-Assisted Replay of Multiprocessor Programs, *Proc. 1991 ACM/ ONR Workshop on Parallel and Distributed Debugging* (*PADD*), pp.35–49 (1991).
2) Broadwell, P., Harren, M. and Sastry, N.: Scrash: A System for Generating Secure Crash Information, *Proc. 12th USENIX Security Symposium*, pp.272–284 (2003).
3) Choi, J.D. and Srinivasan, H.: Deterministic Replay of Java Multithreaded Applications, *Proc. SIGMETRICS Symposium on Parallel and Distributed Tools* (*SPDT*), pp.48–59 (1998).
4) Hölzle, U., Chambers, C. and Ungar, D.: Debugging Optimized Code with Dynamic Deoptimization, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (*PLDI*), pp.32–43 (1992).
5) IBM Corp.: XML Parser for Java. http://www.alphaworks.ibm.com/tech/xml4j
6) Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. and Nakatani, T.: A Study of Devirtualization Techniques for a Java Just-In-Time Compiler, *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA*), pp.294–310 (2000).
7) Jean-loup, G. and Adler, M.: zlib. http://www.gzip.org/zlib/
8) LeBlanc, T.J. and Mellor-Crummey, J.M.: Debugging parallel programs with Instant Replay, *IEEE Trans. Comput.*, Vol.C-36, No.4, pp.471–482 (1987).
9) Lewis, B.: Debugging Backwards in Time, *Proc. 5th International Workshop on Automated and Algorithmic Debugging* (*AADEBUG*), pp.225–235 (2003).
10) Lindholm, T. and Yellin, F.: The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspec/
11) Miller, B.P. and Choi, J.D.: A Mechanism for Efficient Debugging of Parallel Programs, *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (*PLDI*), pp.135–144 (1988).
12) Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Proc. Java Virtual Machine Research and Technology Symposium* (*JVM*) (2001).
13) Pan, D.Z. and Linton, M.A.: Supporting Reverse Execution of Parallel Programs, *Proc. 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (*PADD*), pp.124–129 (1988).
14) Pendragon Software Corporation: CaffeineMark. http://www.benchmarkhq.ru/cm30/
15) Ronsse, M., Bosschere, K.D. and Kergommeaux, J.C.: Execution replay and debugging, *Proc. 4th International Workshop on Automated and Algorithmic Debugging* (*AADEBUG*) (2000).

16) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks and SPECjbb-2000. http://www.spec.org/osg/jvm98/ and http://www.spec.org/osg/jbb2000/
17) Steven, J., Chandra, P., Fleck, B. and Podgurski, A.: jRapture: A Capture/Replay Tool for Observation-Based Testing, *Proc. International Symposium on Software Testing and Analysis* (*ISSTA*), pp.158–167 (2000).
18) Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just-In-Time Compiler, *IBM Systems Journal, Java Performance Issue*, Vol.39, No.1, pp.175–193 (2000).
19) Tikir, M.M., Lueh, G.Y. and Hollingsworth, J.K.: Recompilation for Debugging Support in a JIT-Compiler, *Proc. Workshop on Program Analysis for Software Tools and Engineering* (*PASTE*), pp.10–17 (2002).

(平成 17 年 2 月 21 日受付)
(平成 17 年 6 月 27 日採録)

緒方　一則（正会員）

1967 年生．1990 年東京工業大学工学部電気電子工学科卒業．同年日本アイ・ビー・エム（株）入社．現在，同社東京基礎研究所において，Java 処理系の実行時システムの研究に従事．ACM 会員．

小野寺民也（正会員）

1959 年生．1988 年東京大学大学院理学系研究科情報科学専門課程博士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にて，オブジェクト指向言語の設計および実装の研究に従事．現在，同研究所シニア・テクニカル・スタッフ・メンバー．第 41 回（平成 2 年後期）全国大会学術奨励賞，平成 7 年度山下記念研究賞，平成 16 年度論文賞，平成 16 年度業績賞各受賞．理学博士．日本ソフトウェア科学会，ACM 各会員．

河内谷清久仁（正会員）

1963 年生．1987 年東京大学大学院理学系研究科情報科学専門課程修士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にて，オペレーティングシステムやマルチメディア処理システム，携帯情報システム，Java 処理系ランタイム等の研究に従事．現在，同研究所専任研究員．1994 年全国大会奨励賞，2005 年論文賞各受賞．ACM 会員．

小松　秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本アイ・ビー・エム（株）東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

中谷登志男（正会員）

1975 年早稲田大学理工学部数学科卒業．同年日本アイ・ビー・エム（株）入社．1985 年米国プリンストン大学大学院コンピュータ・サイエンス学科より M.S.E. および M.A.，1987 年 Ph.D. を各取得．同年より同社東京基礎研究所においてプログラム最適化技術の設計・実装・評価の研究に従事．2000 年よりディスティングイッシュト・エンジニア．現在同研究所システムズ担当．基礎研究部門におけるコンパイラ・ファームウエア技術のストラテジストを兼任．MICRO-22（1989）および MICRO-23（1990）にて，Best Paper Award を連続受賞．ACM，IEEE 各会員．