Replay Compilation: Improving Debuggability of a Just-in-Time Compiler

Kazunori Ogata Tamiya Onodera Kiyokuni Kawachiya
Hideaki Komatsu Toshio Nakatani
IBM Research, Tokyo Research Laboratory
1623-14 Shimo-tsuruma, Yamato-shi, Kanagawa 242-8502, Japan
ogatak@jp.ibm.com

Abstract

The performance of Java has been tremendously improved by the advance of Just-in-Time (JIT) compilation technologies. However, debugging such a dynamic compiler is much harder than a static compiler. Recompiling the problematic method to produce a diagnostic output does not necessarily work as expected, because the compilation of a method depends on runtime information at the time of compilation.

In this paper, we propose a new approach, called *replay JIT compilation*, which can reproduce the same compilation remotely by using two compilers, *the state-saving compiler* and *the replaying compiler*. The state-saving compiler is used in a normal run, and, while compiling a method, records into a *log* all of the input for the compiler. The replaying compiler is then used in a debugging run with the system dump, to recompile a method with the options for diagnostic output. We reduced the overhead to save the input by using the system dump and by categorizing the input based on how its value changes. In our experiment, the increase of the compilation time for saving the input was only 1%, and the size of the additional memory needed for saving the input was only 10% of the compiler-generated code.

Categories and Subject Descriptors D.2.5 [**Software Engineering**]: Testing and Debugging – *debugging aids*; D.3.4 [**Programming Languages**]: Processors – *compilers, debuggers.*

General Terms Reliability.

Keywords Replay compilation, deterministic replay, problem determination, Java, JIT compiler, dynamic optimization, debuggability.

1. Introduction

Over the last decade, the performance of Java has been tremendously improved. Undoubtedly, advances in dynamic compilation technologies have significantly contributed to these improvements. Java JIT compilers perform increasingly more advanced, and thus more complicated, optimizations [9,15,20], and can even generate more efficient code than static compilers by taking advantage of runtime profiles.

However, a dynamic compiler is much harder to debug than a static compiler. Assume that an application crashed in a production environment, and analysis suggests that the code generated for a certain method may be causing the crash. What will then be the next step? If the application was developed with a static compiler, we can simply recompile the method with an option to produce *diagnostic output*. The diagnostic output contains all the details of what the compiler does, including what optimizations are applied and how each optimization transforms the code. This greatly helps a compiler writer analyze a bug in the compiler, and the compiler writer can often recognize the bug without re-executing the compiler using a debugger. Without a diagnostic output, it is very difficult to associate each generated machine instruction with the source code.

We could do the same thing when the application is written in Java. More precisely, we could again JIT compile the problematic method by rerunning the application with an option specified to produce diagnostic output. However, this does not necessarily work, because the method may not be compiled in exactly the same way. The reason is that the compilation of a method depends not only on the method's bytecode but also on the runtime information at the time of compilation, such as the resolution status [6] of classes referenced in the method, the class hierarchy, and the runtime profile. This runtime information is not necessarily the same from run to run because the Java application is multi-threaded, and non-determinism in execution is unavoidable. We actually observed that the combination of applied optimizations had changed in at least one out of ten executions for each of the Java programs we evaluated because of changes in the execution order of threads and the runtime profiles.

A straightforward solution would be to run an application with the diagnostic option specified even in a production environment. However, this significantly increases the compilation time and thus the execution time of the application. In addition, forcing the compiler to always generate the diagnostic output would require prohibitively large amounts of disk space. For example, the diagnostic output for a single execution of a SPECjvm98 benchmark [19] can be in the hundreds of megabytes.

In this paper, we propose a new approach for debugging a JIT compiler, *replay JIT compilation*, which allows methods to be recompiled in exactly the same way as in a previous run. Our approach uses two compilers, *the state-saving compiler* and *the replaying compiler*. The state-saving compiler is used in a normal run, and, while compiling a method, records into a *log* all of the runtime information referenced during the compilation. The log is in the main memory, and automatically included in the system dump when the application crashes. The replaying compiler is then used in a debugging run with the system dump, to recompile a method with the options for diagnostic output. This technique is a kind of trace-and-replay technique, but we successfully minimized its overhead by saving in memory (not on hard disk) only the additional trace information that will otherwise not appear in the system dump. As a result, the system dump will always include all the information required for replaying the JIT compiler to recreate the failure.

It is worth noting that using a system dump to reproduce a problem that crashed a mission-critical application is much better than trying to reproduce the problem by recreating the environment in which the application crashed at a remote site. Such an application tends to be very complicated to install, configure, and deploy, and may demand substantial hardware resources. Thus, it would be laborious to set up the same environment at a remote site to reproduce the observed problem. In addition, it may be impossible to obtain the data to run the application if the data includes highly confidential or sensitive information such as credit card numbers. Without the same data, the application will run differently and the problem may not be reproduced.

We implemented our prototype based on the J9 Java VM [7] and the TestaRossa (TR) JIT compiler [21,23] for AIX. This prototype also implements *confidence*-based filtering to save only the logs that are likely to be needed for debugging. Our experiment showed that the time overhead for saving the input is only 1%, and the space overhead for saving the input is only 10% of that of the compiled code. To our knowledge, this is the first report describing how to successfully replay the JIT compilation offline.

This paper makes the following contributions:

- **Replay JIT compilation:** We reduced the overhead for saving the input for the JIT compiler by using a system dump so that it can always be enabled in a production environment.
- **Confidence-based filtering:** We can reduce the size of the input to save by considering the likelihood that a method may cause an error in the JIT compiler.

The rest of this paper is organized as follows. Section 2 summarizes related work. Section 3 explains the behavior of a dynamic compiler. Section 4 discusses our approach to replaying the JIT compilation using a system dump. Section 5 describes the implementation of our prototype. Section 6 shows how small the overhead of the replay JIT compilation is in terms of the size of the saved input and the time to save the input. Section 7 offers concluding remarks.

2. Related Work

Trace-and-replay is a common technique for cyclic debugging of multi-threaded programs. There are two approaches for trace-and-replay, the *ordering-based* and *content-based* [17] approaches.

The ordering-based approach is to record and replay the order of synchronization events [12], such as locking and message passing. For this approach, many techniques [2,4,12,14] have been developed and discussed to trace and replay the program execution with small overhead. For a Java JIT compiler, the compiler itself operates deterministically, but the compilation results may change because the input for the compiler may change non-deterministically during the execution of the Java program. The input for the JIT compiler is runtime information from the Java VM, and that data is changed by many of the Java operations, such as object allocation, access to a field variable, or method invocation. The changes made by other threads immediately change the input for the compiler. Thus, it is impractical to use the ordering-based approach for a Java VM because recording the order of all of those operations is needed to reproduce the input for the JIT compiler.

The content-based approach is to save and restore the values of the input. Recap [16] records the input for a program. However, it is estimated to generate 1 Mbyte of trace per second, even on a slow VAX-11/780 machine, and a faster machine could generate an unacceptably huge amount, such as 1 Gbyte per second. The jRapture system [22] records the parameters and the return values of a Java API that interacts with the underlying system. However, their prototype was three to ten times slower than normal execution. The idea of the content-based approach is simple and easy to adopt, but it tends to cause prohibitively large overhead in size and speed for use in practical systems. Using a system dump, we successfully minimized the overhead of recording the information required for replaying the JIT compiler.

The first error data capture (FEDC) concept [11] also aims to improve the debuggability of a mainframe system used in a production environment. It uses special hardware, firmware, and software that continuously record information about each component in the system by using a separate computer. When an error occurs, support personnel can analyze the recorded information to debug the error. This is an effective approach for problem determination without replaying the system, but it needs special support by hardware and firmware.

Non-determinism in execution can also cause problems in performance analysis. For example, we cannot discriminate between the causes of performance improvements because of the non-determinism. OOR [8] and PEP [3] solved this problem by using advice files produced by the JIT compiler in the previous best run. Those files record the compilation level and the results of profilers, and they are used by the complier in a performance measurement run. Their motivation is very close to ours, but they did not describe the details of their implementation.

3. Behavior of a Dynamic Compiler

The user invokes a static compiler by specifying one or more source files and zero or more options as command line arguments. Since these are all of the inputs for the static compiler, the user can reproduce the same compilation simply by supplying the same set of the command line arguments. In addition, the user can obtain the details of the compilation by adding an option for the compiler to generate diagnostic output.

In contrast, the user does not directly invoke a dynamic compiler. The dynamic compiler is not a standalone tool, but a component of a virtual machine. The user invokes the virtual machine to run an application, and the virtual machine invokes the dynamic compiler while executing the application. The virtual machine determines at runtime which methods to compile with what set of options. In addition, as we will soon explain, the dynamic compiler fully exploits information available at runtime to generate highly efficient code.

Reproducing the compilation is much harder for a dynamic compiler for the following reason. The inputs for the dynamic compiler are all provided as data structures in the virtual machine, which will be lost when the virtual machine terminates. For example, the runtime information mentioned above is kept as runtime data structures. Furthermore, this is true even for the bytecode of the method to be compiled. The dynamic compiler does not directly obtain the bytecode from an external file such as the `.class` file in java, but does so from runtime data structures. Note that dynamic bytecode generation is becoming popular in the recent versions of Java [5], and there may be no corresponding external files for some compiled methods.

3.1 Runtime Information as Inputs

While compiling a method, the dynamic compiler exploits runtime information which can be categorized into three types, the system configuration data, the virtual machine states, and the runtime profiles.

An example of system configuration data is whether the underlying system is uniprocessor or multiprocessor. The dynamic compiler generates faster code for synchronization for a uniprocessor system. Another example is the processor architecture of the underlying system. The dynamic compiler exploits the information to generate instructions only available in a specific processor architecture. Note that, with the static compiler, the user can specify system configuration data as command line options. This implies that the user must prepare different executables for different configurations, and pick up the right executable based on the actual execution platform.

A common example of the virtual machine states is the resolution status for external references [6]. The bytecode of a method contains external references to classes, fields, and methods. When the virtual machine loads a class, all of the external references are *symbolic*. During the execution, references will undergo resolution and become *direct* references. The dynamic compiler generates faster code for direct references, and code to force the resolution for symbolic references. Another example is the hierarchy of classes loaded into a virtual machine. The dynamic compiler analyzes the hierarchy to devirtualize method invocations [9]. Devirtualization is one of the most important optimizations for object-oriented programs.

The dynamic compilation system is considered to be best positioned for the profile-guided optimizations since it can be made transparent to collect runtime profiles. The runtime profile of a method could be based on block profiling, edge profiling, or path profiling. The dynamic compiler generates optimized code in favor of frequently executed basic blocks, edges, or paths [1,25,27]. The runtime profile may even include value profiles, or distributions of values of arguments and variables. The dynamic compiler then creates specialized versions of a method based on values frequently observed.

3.2 Difficulty in Reproducing the Compilation

As we mentioned earlier, the user invokes a virtual machine to run an application, which in turn invokes a dynamic compiler. However, it is almost impossible to reproduce the compilation of a method simply by re-running the application. The reason is that the modern virtual machine runs an application in a non-deterministic manner. It creates system threads for compilation and garbage collection, and runs an application in a multi-threaded environment. Thus, different runs of an application may result in different methods being compiled. Even if a method is compiled in two runs of an application, the compiled code may not be the same because the inputs for the dynamic compiler may not be identical. For example, the system configuration data will be different if the virtual machine runs the application on different execution platforms. The virtual machine states and runtime profiles may be different because of the non-determinism in the way the virtual machine runs the application.

To reproduce the compilation of a method, we need to be able to do the following things. First, we need to be able to invoke a dynamic compiler as if it were a stand-alone tool, since different runs of an application may result in different methods being compiled. Second, we need to be able to reproduce the inputs including the runtime information exploited at the compilation. This can normally be done by saving the values of those inputs, and restoring them for the reproducing compilation.

3.3 Variable and Fixed Inputs

We can categorize the inputs for a dynamic compiler into two types, *variable inputs* and *fixed inputs*. If the inputs may change after a compilation of a method, we call them variable inputs. Otherwise, we call them fixed inputs. For example, the resolution statuses of the external references are variable inputs. Also, the class hierarchy is variable input since new classes may be loaded after the compilation. On the other hand, the system configuration data is fixed input. Also, the bytecode of the method can be a fixed input if the virtual machine prohibits the rewriting of the bytecode.

Table 1 of Section 4.2 shows a more detailed summary of inputs for the dynamic compiler, including whether they are variable or fixed. Among the four types of input (the target method, the system configuration, the virtual machine states, and the runtime profiles), the first two types are fixed inputs and the others are variable inputs.

Note that, while the values of the variable inputs must be saved at the time of compilation, those of the fixed inputs can be saved at the arbitrary time.

4. Overview of Replay Compilation

We use the content-based trace-and-replay technique to reproduce the compilation by a dynamic compiler. Concretely, we create two versions of a dynamic compiler, the *state-saving compiler* and the *replaying compiler*. In our approach, the virtual machine invokes the state-saving compiler to dynamically compile methods. This compiler saves all of the inputs for each compilation into a *log*. Later, the user invokes the replaying compiler by specifying a method to be replayed. The replaying compiler reproduces the compilation of the method by restoring all of the inputs for the compilation from the corresponding log.

Since the virtual machine invokes the state-saving compiler while running an application, the overhead of the compiler must be sufficiently small both in terms of time and space. As we will show, we employ a cascade of techniques to reduce the overhead.

Note that the replay compilation is meant to support the debugging of a dynamic compiler. Assume that a problem occurred while a Java application is running. The replay compilation is not a tool for analyzing the problem in general. Instead, it should be used when the analysis of the problem *suggests* that an execution of a compiled method caused the problem, and that the dynamic compiler failed to generate the code correctly.

4.1 System Dump Based Approach

We assume that the virtual machine is configured to generate a *system dump* at crashes and user interrupts. As a `core` file in UNIX systems [24], a system dump contains the memory image of an OS process. Exploiting this assumption, the state-saving compiler creates the logs for compilations in the main memory, not explicitly writing them into a file. The logs are then automatically saved into a system dump when the operating system creates one. Avoiding expensive I/O during compilation significantly helps reduce the time overhead of the state-saving compiler.

Note that this system-dump-based approach has a significant advantage. It does not require the virtual machine to be invoked to rerun an application. It suffices to invoke the replaying compiler with the system dump. Furthermore, the platform where the replaying compiler is invoked does not have to be identical to the platform where the system dump was generated.

Thus, one of the scenarios which are only made possible by our approach is as follows. The customer invokes the virtual machine in a production environment which runs a mission critical application, invoking the state-saving compiler. The virtual machine crashes, and a system dump is generated. The customer sends the system dump to the support personnel at a different site. They invoke the replaying compiler in their environment to fix a problem in the compiler.

Developers of a dynamic compiler can also benefit from our approach. Assume that a test case is highly multi-threaded and thus runs in a very non-deterministic manner. Obviously, it is hard to reproduce an error in such a test case by rerunning it. With replay compilation, developers do not have to run the test case repeatedly. They only have to invoke the replaying compiler.

4.2 Log Structure

In general, the dynamic compiler gets the inputs for a compilation, whether variable or fixed, by accessing data structures and calling functions. For an input by data structure access, the state-saving compiler records a pair of the address and the value into a log. Later, the replaying compiler retrieves the value from the log, by using the address as the key. For an input in a function call, the state-saving compiler records into a log the return value together with a list of function identifier and zero or more parameter values. The replaying compiler retrieves the return value from the log by using the list as a key.

The system-dump-based approach makes an optimization possible for a fixed input by data structure access. As mentioned in Section 3.3, while the values for variable inputs must be saved at the time of the compilation, the values for fixed inputs can be saved at the arbitrary time. In particular, the values for fixed inputs can be saved at the time of creating the system dump. Thus, the state-saving compiler does not have to save anything for fixed inputs during compilation, simply relying on the system dump that will

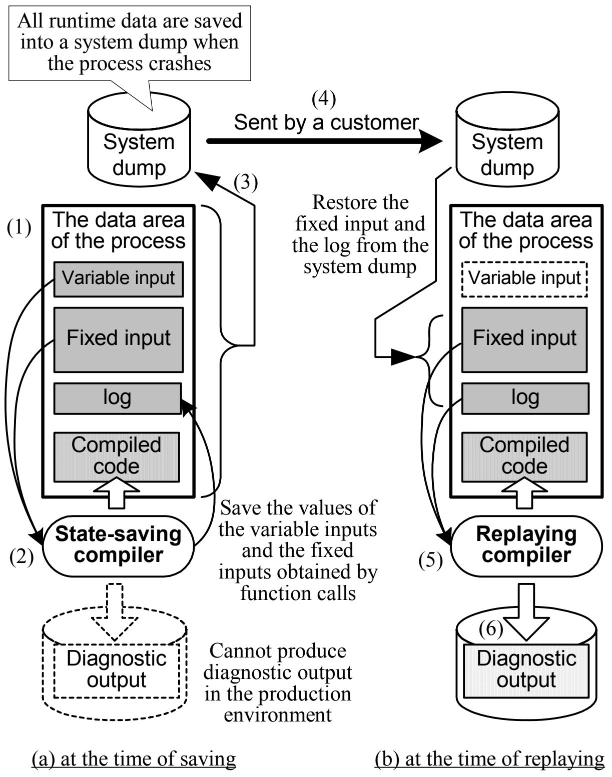
All runtime data are saved into a system dump when the process crashes
(4)
Sent by a customer
System dump
System dump
(3)
(1)
The data area of the process
Variable input
Fixed input
log
Compiled code
(2)
State-saving compiler
Save the values of the variable inputs and the fixed inputs obtained by function calls
Diagnostic output
Cannot produce diagnostic output in the production environment
Restore the fixed input and the log from the system dump
The data area of the process
Variable input
Fixed input
log
Compiled code
(5)
Replaying compiler
(6)
Diagnostic output
(a) at the time of saving
(b) at the time of replaying

Figure 1. Replay JIT compilation

later be generated. Since fixed inputs in data structures access occupy the largest share in inputs for a Java JIT compiler, this also contributes to reducing the overhead of the state-saving compiler.

Figure 1 summarizes the discussion so far by using a typical scenario. (1) The virtual machine is running a Java application at a customer site (left figure). (2) The virtual machine invokes the state-saving compiler, which saves the inputs for compilations into logs. (3) The virtual machine then crashes, causing the operating system to automatically create the system dump. (4) The customer sends the system dump to the service personnel at a remote site. (5) The service personnel invoke the replaying compiler with the system dump to reproduce a problematic compilation. (6) The replaying compiler, running at their site, restores the inputs for the compilation from the corresponding log, generating detailed diagnostic output for the compilation.

Table 1 summarizes the discussion so far for the inputs for a dynamic compiler. It shows the types of inputs for a dynamic compiler together with values to be saved into logs. It also describes how a Java JIT compiler uses the inputs for optimizations.

4.3 Building State-saving and Replaying Compilers

We describe how the state-saving and the replaying compilers are developed. We first build a compiler in such a way that it uses macros to get the inputs for a compilation. Since our base JIT compiler gets the inputs by accessing data structures and by calling functions, we converted those codes getting the inputs to use macros. We then provide two sets of definitions for the

Table 1. Types of input used by the Java JIT compiler and the values to be saved in a log

Type of input	Input for the JIT compiler	Value to be saved into a log	How the JIT compiler uses the input
Target method (fixed input)	Bytecode, literals, and the metadata for external references	Address of the data structure associated with the target method	The JIT compiler reads these inputs as source code.
System configuration (fixed input)	Configuration of the hardware and the software	Model, cache size, number, and specifications of the processors in the machine, and the type and version of the operating system	The JIT compiler can generate code that can run faster in a specific environment than generic code.
	Command line options and the environment variables	Address of the data structure that holds the parsed command line options and the environment variables	Those options may change the compilation process for all or particular methods.
Virtual machine states (variable input)	Set of classes that are referred to and that have been initialized	A flag for each class indicating if the class has been initialized	When the class has already been initialized, the JIT compiler can generate faster code, since the generated code need not handle the initialization.
	Address of the compiled code	Addresses of the compiled code invoked from the method being compiled	The JIT compiler can generate code that directly calls the compiled code of the callee method if it is already compiled.
	Saved results of the JIT optimizations	Addresses of the classes that hold the results of the inter-procedural analysis	The JIT compiler can reuse the saved results of inter-procedural analysis to reduce compilation time.
	Class hierarchy of the loaded classes	A set of the parameters and the return value of each function call for devirtualization	The JIT compiler can use the class hierarchy analysis to devirtualize the method invocation of virtual and interface methods.
	Resolution statuses	A bitmap indicating which of the external references have been resolved	For each resolved reference, the JIT compiler can generate faster code, since the generated code need not handle the resolution. The JIT compiler may be able to inline the callee method when the reference to it has been resolved.
Runtime profiles (variable input)	Runtime profiler output	The values of the runtime profiles	The JIT compiler will apply more aggressive optimizations to the frequently executed path, or can generate code that is specialized for the frequently appearing values.
	Optimization level	A value of the optimization level that is determined based on the runtime profile	The JIT compiler selects the set of optimizations to apply based on the optimization level that was determined based on the profiler output.

macros, one for the state-saving compiler and the other for the replaying compiler. The definitions for the state-saving compiler will store the inputs into a log as well as return the inputs, while the definitions for the replaying compiler will retrieve the inputs from a log. In this way, we can derive the two compilers from a single source.

Figure 2 illustrates the details. Figure 2 (a) shows the code of the conventional compiler. Receiving the name of the target method, the function `compile` obtains the number of available processors, a data structure `md` associated with the target method, the addresses of the bytecode, and a bitmap that holds the resolution status. We assume that the number of the available processors and the bytecode are the fixed inputs, while the others are the variable inputs. Figure 2 (b) presents the macro version for replay compilation. We wrap with a macro every piece of the code which obtains an input. We use different macros, depending on whether inputs are variable or fixed and whether inputs are obtained by accessing data structure or by calling functions.

Figure 2 (c) lists the definitions of the macros for the state-saving compiler. Because of the optimization mentioned in Section 4.2, the `getFixedInputByDataAcc` macro is simply defined to do the same operation as in Figure 2 (a), and not to save anything into a log. Figure 2 (d) shows the definitions for the replaying compiler. The `getFixedInputByDataAcc` macro is defined to get the input from the system dump, not from the log. As we will explain in Section 4.4, we may need to adjust the address appropriately.

The replay compilation assumes that both of the state-saving and the replaying compilers apply the same set of optimizations for the same inputs. Thus, the versions of the source code for the state-saving and the replaying compilers must be synchronized. Deriving the two compilers from a single source greatly simplifies this task of synchronization.

4.4 Replaying the Compilation

While the virtual machine invokes the state-saving compiler, the user invokes the replaying compiler by specifying one or more compilations to be reproduced. This is usually done by specifying method signatures or addresses of compiled code. Note that, like a static compiler, the replaying compiler independently reproduces the specified compilations. There is no restriction on the replay order, and we can even replay all of the compilations in the reverse of the order in which the sate-saving compiler did.

As in Figure 1, it is common that the replaying compiler and the virtual machine invoking the state-saving compiler run on different platforms. Using the replaying compiler, the support

```
compile(char *signature) {
  int      numCPU;
  Method  *md;
  char    *bytecode;
  BitMap  *map;


  numCPU = getNumProcessor(); //Fixed input

  md = getMethod(signature);  //Variable input

  bytecode = md->bytecode;    //Fixed input

  map = md->resolveMap;       //Variable input

  ...
```

(a) an example of the code in a base compiler

```
compile(char *signature) {
  int      numCPU;
  Method  *md;
  char    *bytecode;
  BitMap  *map;
  Log     *log = setupLog(signature);

  getFixedInputByFunc0Arg(int,
                   numCPU, getNumProcessor);
  getVariableInputByFunc1Arg(Method *,
                  md, getMethod, signature);
  getFixedInputByDataAcc(Method,
                    bytecode, md, bytecode);
  getVariableInputByDataAcc(BitMap*, Method,
                       map, md, resolveMap);
  ...
```

(b) the code modified for the replay compilation

```
#define getFixedInputByFunc0Arg(            \
                    destTpye, dest, func) { \
  dest = func ## ();                        \
  putLogForFunc(dest, getFuncID(func));     \
}

#define getVariableInputByFunc1Arg(         \
              destType, dest, func, arg1) { \
  dest = func ## (arg1);                    \
  putLogForFunc(dest,                       \
                    getFuncID(func), arg1); \
}

#define getFixedInputByDataAcc(             \
              baseType, dest, ptr, field) { \
  dest = (ptr)->field;                      \
}

#define getVariableInptByDataAcc(destType,  \
             baseType, dest, base, field) { \
  int  offset = offsetof(baseType, field);  \
  dest = (base)->field;                     \
  putLogFodDataAcc(dest,                    \
        getTypeID(baseType), base, offset); \
}
```

(c) the definitions of the macros (state-saving compiler)

```
#define getFixedInputByFunc0Arg(            \
                    destTpye, dest, func) { \
  dest = (destType)getLogForFunc(           \
                          getFuncID(func)); \
}

#define getVariableInputByFunc1Arg(         \
              destType, dest, func, arg1) { \
  dest = (destType)getLogForFunc(           \
                    getFuncID(func), arg1); \
                                            \
}

#define getFixedInputByDataAcc(             \
              baseType, dest, ptr, field) { \
  dest = adjustAddr(baseType, ptr)->field;  \
}

#define getVariableInptByDataAcc(destType,  \
             baseType, dest, base, field) { \
  int  offset = offsetof(baseType, field);  \
  dest = (destType)getLogFodDataAcc(        \
        getTypeID(baseType), base, offset); \
                                            \
}
```

(d) the definitions of the macros (replaying compiler)

Figure 2. An example of the code to get the input for the compiler

personnel do not have to create exactly the same platform as the state-saving compiler used. This significantly reduces the cost for the support division.

When it is invoked, the replay compiler first loads the system dump into its address space. It then attempts to reproduce a compilation by retrieving the inputs for the compilation. A tricky part of the replaying compiler is that if an input retrieved from the log is an address, it is the address in the process for the state-saving compiler (*state-saving process*). The replaying compiler cannot use the address to retrieve the value of a fixed input, since the system dump is not necessarily loaded at the same address as the sate-saving process. This is the reason why we need the adjustment in the `getFixedInputByDataAcc` macro.

We may be able to avoid this adjustment as follows. Assume that we can know the address range in the system dump where fixed inputs reside. If the replaying compiler can reserve the address range at start-up, it can load the data from the system dump into the address range, restoring the fixed inputs at the same addressees. However, it depends on the underlying operating system and the details of how the process for the replaying compiler is initialized.

4.5 Further Reducing the Size of Logs

The overhead of the state-saving compiler must be small enough both in terms of time and space, since the virtual machine invokes it while running an application. Exploiting the system-dump-based approach, the state-saving compiler allocates logs in the main memory, and skips saving fixed inputs by data structure access. The former significant help contributes to reducing the time overhead, while the latter contributes to reducing both the time and space overhead.

Here we show three techniques to further reduce the space overhead. The first technique is to compress the logs in memory. This is simple yet very effective in reducing the log size.

The second technique is to exploit *default* values. The state-saving compiler skips saving into a log an input when the value equals the default value, while the replaying compiler interprets the value of an input as default when it can find in the log no value corresponding to the input. Default values should be defined as frequently observed values, and different default values can be defined for different functions and data structure types.

The third technique is to skip saving all of the inputs for the compilation of a method if we have high *confidence* in the method, or if we can assume that the method is very likely to be compiled correctly. We call this technique confidence-based filtering. We define the confidence of a method as follows.

- Increase the confidence of a method, if the method is in heavily used libraries such as the Java core classes
- Decrease the confidence of a method, if the control flow of the method is complex.

The rationale behind the first criterion is that methods in heavily used libraries are frequently compiled and executed, and thus the paths to compile them are already well tested. The second criterion is simply based on our rule of thumb. We often observed that a problem in the compiler appeared when the compiler processed a complex control flow. We can approximate the complexity of the control flow of a method as the number of basic blocks. The time spent on compiling the method is also a good approximation.

4.6 Discussion

By definition, the replaying compiler uses exactly the same set of the options for compiling a method as the state-saving compiler used, except an additional option for diagnostic output. However, in some cases the support personnel may want to replay a compilation with a slightly different set of options in order to narrow down the cause of a problem. At a first glance, this would be impossible since the replaying compiler may now need to obtain the inputs which the state-saving compiler did not use and thus did not save. However, we can exploit the mechanism of default values described in Section 4.5. That is, we simply let the replaying compiler pick up the default value for an input when it fails to find in the log a value corresponding to the input.

The Java virtual machine may unload classes and delete data structures associated with them from the main memory. As a result, the system dump will not include the data structures for unloaded classes, such as bytecode for methods. This means that the replaying compiler is unable to replay the compilation of a method in an unloaded class.

We could argue that this is not a serious issue, based on the following observation. A class is unloaded only when there is no reference to the class in the virtual machine [13]. That is, there is no instance of the class, and no method of the class currently being executed. Assume that a system crash occurred and that it was actually caused by incorrectly compiled code for a method. We believe that such a method is very likely to have a stack frame, actively being executed.

We could also modify the state-saving compiler to store the data structures of a class into an external file when the class is unloaded. We could do so by generating a system dump at the time of unloading if we could unload classes as a batch process. The state-saving compiler also timestamps the logs for compilations so that the replaying compiler can properly associate the logs and the data structures for unloaded classes.

5. Implementation

This section describes our prototypes of a state-saving compiler and a replaying compiler. We implemented the prototype based on the J9 Java VM [7] and the TestaRossa (TR) JIT compiler [21,23] for AIX.

5.1 State-Saving Compiler

Our state-saving compiler allocates a memory area as the log for each compilation of a method. This compiler compresses each log using zlib library [28]. The log works as if it were a cache, so that the compiler can avoid saving duplicated input that happens to be constant during the compilation. That is, even if the state-saving compiler tries to get a variable input multiple times, it actually gets the value only on the first access, and subsequent accesses get the value saved in the log.

The state-saving compiler associates each log with the address of the JIT-compiled code. This makes it possible to identify the log for a particular compilation, even if the method is compiled multiple times for different optimization levels and there are multiple compiled code blocks for the method. Our prototype does not support replay compilation for unloaded classes. To defer dealing with this complex issue, we simply disabled class unloading in our experiments.

5.2 Replaying Compiler

Our prototype restores a system dump into the same address as the state-saving process to avoid the overhead by the adjustment of pointer variables, as described in Section 4.4.

Our state-saving compiler creates a special data structure, called an *anchor structure*, so that the replaying compiler can find the important data from a system dump. The data structure has markers in its header and trailer, and contains its size, the version number of the state-saving compiler, and a pointer to the list of logs. The state-saving compiler manages all logs as a linked list, and stores the pointer to the head into the anchor. The anchor structure also saves the fixed input that can be determined when the Java VM is initialized, such as the system configuration.

The replaying compiler scans the markers in the restored system dump, which is a block of unstructured binary data, for finding the anchor structure. When the compiler finds a marker, it verifies the size and the version of the state-saving compiler. It then finds the pointer to the head of the log list, and scans the list for the log corresponding to the compilation to be replayed.

The anchor structure has another pointer to the address of the log for currently being compiled (called the *current log*). Since an incomplete log may crash the replaying compiler, the current log

Table 2. Configurations of the tested machines

	Machine-1	Machine-2	Machine-3
CPU	POWER3, single processor	POWER4, 4-way SMP	POWER3, 2-way SMP
RAM	768 Mbytes	8 Gbytes	768 Mbytes
OS	AIX 5.2L	AIX 5.2L	AIX 4.3.3

Table 3. Evaluated programs

Program	Description
mtrt, jess, compress, db, mpegaudio, jack, javac	Each of the programs included in the benchmarks suite SPECjvm98 [19].
SPECjbb	The SPECjbb2000 [18] benchmark.
xml parser	The operation of parsing a sample XML file using the XML parser for Java [26]. The sample file is included in the package. The execution performance was measured by the elapsed time for parsing the sample file.
jigsaw	The operation to start the Jigsaw HTTP server release 2.2.5a [10], and load the default top page using a Web browser. The execution speed was not measured because this is an I/O bound program.

should not be accessible in the list of "complete" logs. While the JIT is compiling a method, the pointer holds the address of the current log, and clears it when the compilation has finished successfully. Using this pointer variable, the replaying compiler can tell if the system crashed during a JIT compilation.

6. Experimental Results

Using the prototypes of the state-saving and the replaying compilers described in Section 5, we measured size and time overhead for saving the input into logs. Various machine configurations and programs were used for the evaluation, as shown in Table 2 and Table 3. For all measurement, we used the exploitation of default values described in Section 4.5. The confidence-based filtering is not uses, unless otherwise is specified. This prototype forces the Java VM to always create a system dump when it terminates after executing the specified Java program (because none of the tested program crashes the Java VM).

6.1 Replayed Compilation

Our prototype successfully reproduced the compilation for all of the methods that were compiled for these programs. We executed the state-saving and the replaying compilers in the same machine in each of the three tested machines. We verified that the replaying compiler reproduced the same compilation by generating the JIT compiled code at the same address as that of the state-saving compiler and by comparing the generated code with the code saved in the system dump.

In addition, we also verified that the replaying compiler successfully reproduced the compilation from the system dump generated by a different machine. The replaying compiler succeeded in replaying all six possible combinations of the

Table 4. Comparison in size: diagnostic output vs. log

Program	Diagnostic output [MB]	Log [MB] (not compressed or filtered)
mtrt	378	0.054
jess	243	0.077
compress	80	0.011
db	83	0.019
mpegaudio	200	0.045
jack	360	0.075
javac	588	0.258
SPECjbb	1033	0.222
xml parser	101	0.030
jigsaw	136	0.056
Geo. mean	227	0.056

machines listed in Table 2 to execute the state-saving compiler with the replaying compiler.

6.2 The Size of a Log

Table 4 shows the size of the logs and the size of the diagnostic output for each program. We used neither compression nor the confidence-based filtering for this measurement. This result shows that the size of the diagnostic output is too large to save in memory, and it is also too large to save on disk because the overhead to save so much data on disk during the execution of the application program will slow down the program. The replay compilation technique reduced the size of the trace information so much that saving input can always be enabled even in a production environment.

Figure 3 shows how the total size of the log changes due to compression and filtering. The sizes are relative to the total size of the compiled code. The left two bars for each program show the results when no compression is used, and the right two bars show the results with compression using the zlib library. The bars labeled `no filter' (first and third) show the results when the compiler does not use the confidence-based filtering. The bars labeled `filter system classes' show the results when the compiler uses filtering of the system classes (the classes in the `java.lang`, `java.util`, `java.math`, and `java.io` packages).

The reduction in the log size by filtering the system classes was 22.7% and 25.4% without and with compression, respectively. (All percentages are geometric means.) Compression reduced the total size of the logs by approximately half, and the reduction made by the combination of compression and filtering was 62.9%. As a result, the geometric mean of the size of the logs was reduced to less than 10% of the compiled code, which was our initial target as being acceptable for many users.

Table 5 shows the number of logs when filtering the methods of the system classes is used or not used. It also shows the reduction in the numbers and the sizes of the logs by filtering. The reduction in size is the summary of Figure 3 when zlib is not used. The reduction in the number of methods was 38%, but the reduction in the size of the logs was only 23%. We think this is because many of the filtered methods are smaller than average methods.

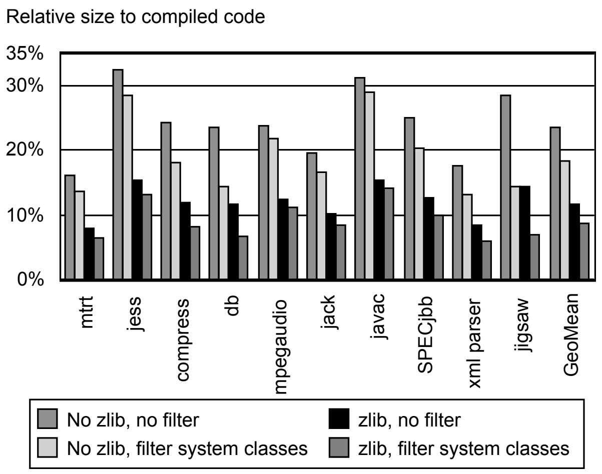

Figure 3. Reduction in the size of log with zlib and filter

Table 5. Reduction in the number of logs

Program	No filter	Filter system classes	Reduction in size by filtering (no zlib)
mtrt	153	113 (-26%)	-16%
jess	153	104 (-32%)	-12%
compress	39	22 (-44%)	-25%
db	59	22 (-63%)	-38%
mpegaudio	161	141 (-12%)	-9%
jack	201	141 (-30%)	-16%
javac	604	514 (-15%)	-7%
SPECjbb	502	345 (-31%)	-20%
xml parser	78	44 (-44%)	-25%
jigsaw	160	64 (-60%)	-49%
Geo. mean	152	94 (-38%)	-23%

The total size of the log was less than 0.1% of the total memory usage of the Java VM process for these programs when both filtering and compression are used. This is because the size of the Java heap is much larger. Thus, for the measured programs, filtering and compression may not be needed for some users because the total size is still less than about 0.3% of the total memory usage. However, the number of compiled methods will increase in large commercial applications, such as Web application servers, and thus the ratio of the size of the compiled code will be higher than for the measured programs. For such programs, it is important to keep the total size of the log as small as possible by applying filtering and compression techniques. We also think it is still beneficial to keep the size of the log much smaller than the size of the JIT compiled code, because the compiled code is used to directly benefit the user by improving the execution performance of the programs. However, the log is used only for improving debuggability, while it reduces the available memory for the user programs regardless of whether or not a problem occurs.

Table 6 shows the sizes of the system dumps and the heap sizes of the Java VM that created the system dumps. A system dump contains all of the data areas in the process memory, such as the

Table 6. Comparison in size: system dump vs. Java heap

Program	System dump [MB]	Java heap [MB]
mtrt	334	256
jess	331	256
compress	330	256
db	330	256
mpegaudio	331	256
jack	331	256
javac	336	256
SPECjbb	413	256
xml parser	124	64
jigsaw	212	128

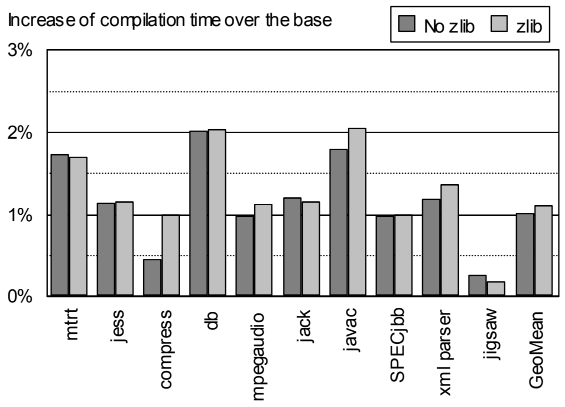

Figure 4. Increase in compilation time without and with zlib compression

Java heap, the Java stack, the native heap, the native stacks of all threads, and the JIT compiled code blocks, though the largest part is the Java heap. Note that the JIT compiler usually does not directly use the values in Java objects for compiling a method, but uses the values collected by the runtime profiler. Thus, a large portion of this large system dump is not needed by the replaying compiler. Although a large system dump requires a large free space on disk, this is considered to be acceptable in many production environments because the system dump is normally used for debugging problems that occurred in those environments.

6.3 Compilation Time

Figure 4 shows how much the compilation time is increased over the base compiler when zlib is used. Each bar labeled `no zlib' shows the compilation time when no compression is used, whereas each bar labeled `zlib' shows the compilation time when zlib compression is used. Since the set of methods compiled in an execution of the program has changed on every execution due to the non-determinism, for fair comparison, we accumulated the elapsed time of the compilations that performed the same optimizations in all executions for the base, `no zlib', and `zlib'.

The increase in compilation time was up to 2.0% in both cases. The geometric mean of the increase was about 1.0% and 1.1% for `no zlib' and `zlib', respectively. This very small increase in compilation time indicates that the time to save the input to the logs was negligible.

The increase in compilation time in the replaying compiler over the base was 9.4%, as a geometric mean, when the diagnostic output was not generated. When the diagnostic output was generated, most of the compilation time is used for writing hundreds of megabytes of text to disk, regardless of whether or not the replay compilation technique is used.

6.4 Execution Speed

The operations to save the input into logs is the only additional overhead in the state-saving compiler against the base compiler because, for the same inputs, the state-saving compiler applies the same compilation and generates the same compiled code as the base compiler. There is no additional overhead in the compiled code.

Since the increase in compilation time was small, the slowdown of execution speed was also small. The geometric mean of the slowdown was only 1%.

7. Conclusion

We have proposed a new approach, called *replay JIT compilation*, to reproduce the same JIT compilation offline and remotely by using two compilers, *the state-saving compiler* and *the replaying compiler*. The state-saving compiler is used in a normal run, and, while compiling a method, records into a *log* all of the runtime information referenced during the compilation. The log is in the main memory, and automatically included in the system dump when the application crashes. The replaying compiler is then used in a debugging run with the system dump, to recompile a method with the options for diagnostic output.

We developed our prototype based on the J9 Java VM and the TestaRossa (TR) JIT compiler for AIX and showed that the prototype successfully reproduces the same compilations done by the state-saving compiler. We also developed *confidence*-based filtering to save only the logs that are likely to be needed for debugging. The time overhead of running the state-saving compiler was only 1% and the size overhead for saving states was only 10% of the compiled code. To our knowledge, this is the first report describing how to successfully replay the JIT compilation offline.

Acknowledgements

We would like to thank Akihiko Togami of IBM Japan and Trent Gray-Donald of IBM Canada for helpful discussions on the initial idea and implementation of the replay JIT compilation. We also thank Toshimasa Shimizu of IBM Japan for pointing out the difficulties in debugging the JIT compiled code in mission-critical environments, by which we were motivated. We also thank Derek Inglis of IBM Canada for pointing out the possible problem on Java class unloading. We also thank Shannon Jacobs of IBM Japan HRS and the members of the Systems Group in the IBM Tokyo Research Laboratory for helpful discussions and comments on an earlier version of this paper.

References

[1] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '01), pp. 168-179. June, 2001.

[2] D. F. Bacon. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (PADD '91), pp. 194-205. May, 1991.

[3] M. D. Bond and K. S. McKinley. Continuous Path and Edge Profiling. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '05), pp. 130-140. November, 2005.

[4] J. D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (SPDT '98), pp. 48-59. August, 1998.

[5] Dynamic Proxy Classes, available at `http://java.sun.com/j2se/1.4.2/docs/guide/reflection/proxy.html`

[6] J. Gosling, B. Joy, and G. Steele. Java Language Specification, available at `http://java.sun.com/docs/books/jls/index.html`

[7] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Proceedings of the Third Virtual Machine Research and Technology Symposium* (VM '04), pp. 151-162. May, 2004.

[8] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '04), pp. 69-80. October, 2004.

[9] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 2000), pp. 294-310. October, 2000.

[10] Jigsaw - W3C's Server, available at `http://www.w3.org/Jigsaw/`

[11] S. Koerner, R. Bawidamann, W. Fischer, U. Helmich, D. Koldt, B. K. Tolan, and P. Wojciak. The z990 first error data capture concept. *IBM Journal of Research and Development*, Vol. 48(3/4), pp. 557-568. May, 2004.

[12] T. J. LeBlanc, J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, Vol. C-36(4), pp. 471-482. April, 1987.

[13] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, available at `http://java.sun.com/docs/books/vmspec/index.html`

[14] B. P. Miller and J. D. Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (PLDI '88), pp. 135-144. June, 1988.

[15] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (JVM '01), pp. 1-12. April, 2001.

[16] D. Z. Pan and M. A. Linton. Supporting Reverse Execution of Parallel Programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (PADD '88), pp. 124-129. May, 1988.

[17] M. Ronsse, K. D. Bosschere, and J. C. Kergommeaux. Execution replay and debugging. In *Proceedings of the Fourth International Workshop on Automated and Algorithmic Debugging* (AADEBUG 2000). August, 2000.

[18] Standard Performance Evaluation Corporation. SPEC JBB2000, available at `http://www.spec.org/osg/jbb2000/`

[19] Standard Performance Evaluation Corporation. SPEC JVM98, available at `http://www.spec.org/osg/jvm98/`

[20] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-In-Time Compiler. *IBM Systems Journal*, Java Performance Issue, Vol. 39(1), pp. 175-193. February, 2000.

[21] L. Stepanian, A. D. Brown, A, Kielstra, G. Koblents, and K. Stoodly. Inlining Java Native Calls At Runtime. In *Proceedings of ACM/Usenix International Conference On Virtual Execution Environments* (VEE '05), pp. 121-131. June, 2005.

[22] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A Capture/Replay Tool for Observation-Based Testing. In *Proceedings of International Symposium on Software Testing and Analysis* (ISSTA 2000), pp. 158-167. August, 2000.

[23] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO '06), pp. 87-97. March, 2006.

[24] UNIX System Certification, available at `http://www.opengroup.org/certification/unix-home.html`

[25] J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of ACM 2000 Java Grande Conference*, pp. 78-87. June 2000.

[26] XML Parser for Java, available at `http://www.alphaworks.ibm.com/tech/xml4j`

[27] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. In *Proceedings of the Twelfth International Conference on Parallel Architectures and Compilation Techniques* (PACT-2003), pp. 148-158. September, 2003.

[28] zlib, available at `http://www.gzip.org/zlib/`