January 31, 2006

RT0641

Computer Science 10 pages

Research Report

Problem Determination for a Java JIT Compiler using Replay Compilation

Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, Toshio Nakatani

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Problem Determination for a Java JIT Compiler using Replay Compilation

Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya,
Hideaki Komatsu, Toshio Nakatani

IBM Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan
ogatak@jp.ibm.com

ABSTRACT

The performance of Java has been tremendously improved by the advance of the compilation technology. However, debugging a dynamic compiler is much harder than a static compiler. Recompiling the problematic method again to produce a diagnostic output does not necessarily work because the compilation of a method depends on the runtime information at the time of compilation.

In this paper, we propose a new approach, called *replay JIT compilation*, to reproduce the same compilation process remotely using two compilers: *the state-saving compiler* saves all the input to the JIT compiler in the production environment, and *the replaying compiler* reproduces the same compilation process later. We reduced the overhead to save the input by using the system dump and categorizing the input based on the constancy. In our preliminary experiment, the overhead of running the state-saving compiler was negligible, and the size of the additional memory area needed for saving 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, serviceability.

1. INTRODUCTION

Over the decade, the performance of Java has been tremendously improved. Undoubtedly, the advance of the compilation technology has significantly contributed to this improvement. The Java JIT compiler performs increasingly more advanced [5,11,13], thus complicated, optimizations, and can generate more efficient code than a static compiler by taking advantage of runtime profiles.

However, debugging a dynamic compiler is much harder than a static compiler. Assume that an application crashed in a production environment, and that we identified through analysis that the code generated for a certain method may cause 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.

We could do the same when the application is written in Java. Precisely, we could have the problematic method recompiled again by rerunning the application with an option specified to produce a 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 on not only the bytecode but also on the runtime information at the time of compilation, such as the resolution status of classes referenced in the method, the class hierarchy, and the execution 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 the applied optimizations had changed 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 results of the built-in profiler.

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. Furthermore, forcing the compiler to always generate the diagnostic output would require a prohibitively large amount of the disk space since the diagnostic output for a single execution of a SPECjvm98 benchmark can be more than hundreds of megabytes.

In this paper, we propose a new approach, called *replay JIT compilation*, which allows methods to be recompiled exactly the same way as in a production environment. For the problem determination based on the system dump, our approach uses two compilers, *the state-saving compiler* and *the replaying compiler*. The state-saving compiler is run in a production environment, and, while compiling a method, records into a *repository* all of the runtime information referenced during the compilation. The repository is in the main memory, and automatically included in the system dump when the application crashes. We then run the replaying compiler with the system dump, to recompile any target method with the options for diagnostic output.

It is worth noting that using a system dump to reproduce a problem that crashed a mission-critical application is much preferable to 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 very hard 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.

We implemented our prototype based on the J9 Java VM [3] and the TR JIT compiler for AIX and successfully recreated the same compilation processes with the replaying compiler. Our preliminary experiment showed that the overhead of running the state-saving compiler is negligible, and the total size of the additional memory area required for saving the states is only 10% of that of the compiled code. This is three orders of magnitude smaller than the original diagnostic output file. To our knowledge, this is the first report describing how to successfully replay the JIT compilation process offline.

The rest of this paper is organized as follows. Section 2 summarizes related work. Section 3 discusses our approach to replaying the JIT compilation using a system dump. Section 4 describes the implementation of our prototype of the replay JIT compiler. Section 5 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 6 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, *ordering-based* and *content-based* [12] approaches.

The ordering-based approach is to record and replay the order of synchronization events [7], such as locking and message passing. For this approach, many techniques [1,2,7,9] have been developed and discussed to trace and replay the program execution with a little overhead. For a Java JIT compiler, the compiler itself is a single-threaded program and it operates deterministically. The reason for the non-deterministic operation of the JIT compiler is that the input from the Java VM changes non-deterministically during the execution of the Java program. The input to the JIT compiler is the runtime data of the Java VM. Since they are changed by many of the typical Java operations, such as object allocation, the access to a field variable, or the method invocations, it is impractical to use the ordering-based approach for a Java VM to replay the input to the JIT compiler by recording the order of those operations

The content-based approach is to save and restore the values of the input. Recap [10] records the input for a program. However, the size of the trace became large even on a slow VAX-11/780 machine, and a faster machine will produce an unacceptably huge trace. The jRapture system [15] records the parameters and the results of a Java API that interacts with the underlying system. Their prototype was three to ten times slower than the normal execution. However, the only problem of the content-based approach is the high overhead. Thus, we adopted this approach and made it practical for JIT compilers by minimizing the input that the JIT compiler must save.

Dynamic deoptmimization [4] is a technique to debug a program compiled by a JIT compiler. Tikir [16] also addressed problem determination with a JIT compiler. These techniques compile the target method again and generate new compiled code that is good for debugging. Because their objective is to debug Java applications by assuming that the JIT compiler itself is bug-free, it is not applicable to debugging the JIT compiler itself.

3. OUR APPROACH

The replay JIT compilation technique uses two compilers: the *state-saving compiler* saves all of the input for the JIT compiler into a special data area, and the *replaying compiler* reproduces the compilation process in the state-saving compiler using the saved input. In the typical usage of these compilers, the state-saving compiler runs in a production environment that executes the user application, and the replaying compiler runs in the environment of the service people who fix problems that occur in the production environment. Developers of JIT compiler can also benefit from our technique. In this usage, the state-saving compiler runs test cases and the replaying compiler reproduces the operation of the failed tests. Developers do not need to run a test case repeatedly to reproduce the error even though it is hard-to-reproduce problem.

The data area, called a *repository*, is allocated in the memory area of the process for the runtime environment, so that it is automatically saved into the system dump when the process crashes. Thus, the support people can analyze the compiled code using only a system dump from a customer. By using a system dump, we can avoid the overhead of explicitly writing the repository to disk during the execution of the user application. At the same time, the size of repository must be kept small because the repositories and the data of the user applications must co-exist in the address space of the process. We reduced the size of the repository by only storing the input for the compiler whose value may change after a compilation, because the constant values are already going to be stored in the system dump. We refer to those inputs that may change after a compilation as *the variable inputs*, and all other inputs as *the fixed inputs*.

The replaying compiler retrieves the original values of the variable input from the repository saved in the system dump. Figure 1 shows the architecture of the replay JIT compilation.

3.1 The Input for the JIT Compiler

The Java JIT compiler uses the contents of Java class files, such as the Java bytecode and string constants, and the runtime data of the Java VM as its input. Since the bytecode and the string constants are fixed during the execution of a program, they are automatically going to be saved into a system dump if they stay in the memory when the execution environment crashes. The state-saving compiler only needs to save their addresses, instead of their values, into the repository. The replaying compiler retrieves their values from the system dump using the address saved in the repository.

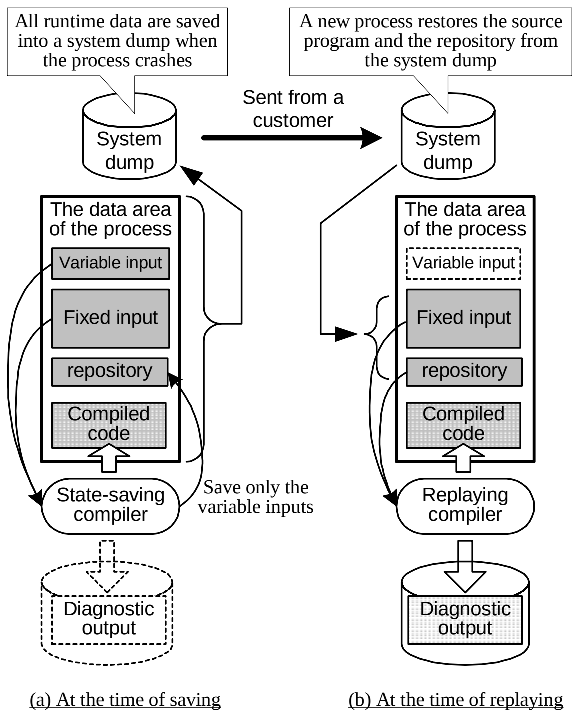
All runtime data are saved into a system dump when the process crashes
A new process restores the source program and the repository from the system dump
Sent from a customer
System dump
System dump
The data area of the process
Variable input
Fixed input
repository
Compiled code
State-saving compiler
Save only the variable inputs
Diagnostic output
The data area of the process
Variable input
Fixed input
repository
Compiled code
Replaying compiler
Diagnostic output
(a) At the time of saving
(b) At the time of replaying

Figure 1. Replay JIT compilation

The state-saving compiler always saves the values of the variable input when it gets that input from the Java VM. It saves each of the variable inputs in the smallest possible space in the repository. Since the largest parts of the input for the Java JIT compiler are the fixed input, such as the bytecode and the string constant, the state-saving compiler only has to save a small part of the input into the repository.

The factors affecting the variable input can be categorized into four groups, as shown in Table 1. This table is a good summary of what the JIT compiler uses as input. It enumerates all the factors affecting the input to the JIT compiler, because the other causes of non-determinism do not affect the JIT compiler. These irrelevant factors include inter-process communication, interrupts from devices, and operations depending on the current time. The first two reasons in the table are due to the definition of the Java language [8]. The other two reasons are due to the nature of the dynamic optimizations.

3.2 Scope of Replay Compilation

The size of the repository depends on how the input is received by the JIT compiler. For the fixed input, the state-saving compiler only saves its address in the repository, and the replaying compiler retrieves the value of the input from the system dump using the address of the fixed input. However, if the state-saving compiler does not know a persistent address for some fixed input, then the state-saving compiler needs to save the value of the input into repository as if it were a variable input. For example, if the JIT compiler gets bytecode using a function call, the state-saving compiler needs to save the value of the bytecode because the return value of a function call does not have a persistent address.

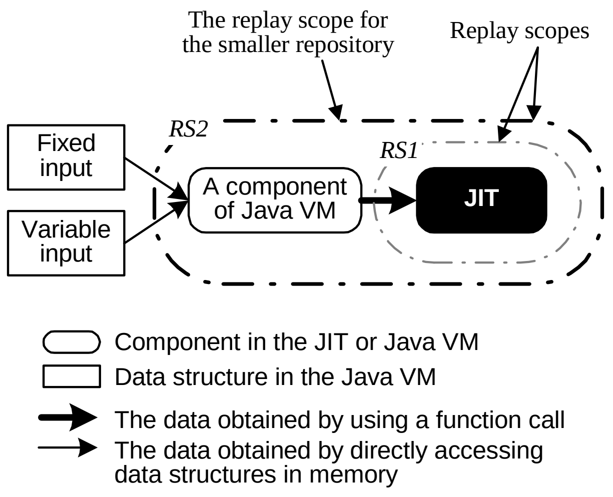
The replay scope for the smaller repository
Replay scopes
RS2
RS1
Fixed input
Variable input
A component of Java VM
JIT
Component in the JIT or Java VM
Data structure in the Java VM
The data obtained by using a function call
The data obtained by directly accessing data structures in memory

Figure 2. An example of the replay scopes

For this reason, the size of the repository can depend on the point in the code where the input is saved. Thus, the selections of those points in the code where the input are saved are important.

The JIT compiler works with the other components of the runtime environment from which the compiler obtains input. In general, a component reproduces the same sequence of outputs to the compiler by saving and restoring the input to the component if and only if it is deterministic.

If the input to the compiler from a component is large and the input to the component is small, we can reduce the size of the repository by saving only the input to the component. In this case, we must replay the group of such components together with the JIT compiler, instead of replaying only the JIT compiler. We call this group a *replay scope*.

Figure 2 shows an example of the replay scopes when a JIT compiler obtains all of the input from a component of a Java VM by using function calls. There are two possible replay scopes: RS1 and RS2. For this example, replaying RS1 saves the bytecode in the repository, as well as the runtime data, because using function calls to get the bytecode forces the compiler to save it in the repository. Thus, we can reduce the size of the repository by choosing RS2.

In general, setting the boundary of the replay scope at the point where it captures the function calls tends to create a large repository because it may save the runtime constants in the repository as if they were variable data. However, there is a trade-off between the size of the repository and the workload to implement the state-saving and replaying compilers, because expanding the replay scope increases the number of changes in the source code to implement the state-saving and replaying compilers.

3.3 Replaying the Compilation

The replaying compiler retrieves the values of the input from the repository and the data area of the Java VM that were saved in the system dump. Then it reproduces the compilation process by using that input and generates the diagnostic output.

3.3.1 Invocation of the Replaying Compiler

The replaying compiler is invoked in a bootstrapping process that executes in the environment of the support staff. The

Table 1. The variable input used by the Java JIT compiler and the reasons why its values may change

Factor	The reason for changing the input	The variable input for the JIT compiler	Possible optimizations the JIT compiler can apply
Built-in multi-threading of Java language			
	A class is initialized when it is used for the first time in the Java VM. The JIT compiler typically first compiles the method that has been executed most frequently.	• The set of initialized classes • The address of the compiled code • The saved results of the JIT optimizations (such as the results of an inter-procedural analysis)	• When the class has already been initialized, the JIT compiler can skip generating the code to initialize the class. • The JIT compiler can generate the code that directly calls the compiled code of the callee method if it is already compiled. • The JIT can reuse the saved result of inter-procedural analysis if it is stored in persistent memory.
Dynamic linking of Java classes			
	A class is loaded when it is accessed for the first time in the Java VM. An external reference is resolved when it is accessed for the first time in the class.	• The class hierarchy of the loaded classes • The resolution status of the external references at the time of the compilation	• The JIT compiler can use the class hierarchy analysis to devirtualize the method invocation of virtual and interface methods. • For each resolved reference, the JIT compiler can skip generating the code that checks if the reference is resolved. • The JIT compiler may be able to inline the callee method when the reference to it has been resolved.
On-line profiler			
	The on-line profiler continuously updates the results.	• The results of the profiler • The optimization level is decided based on the results of the profiler	• The JIT compiler will apply more aggressive optimizations to the frequently executed path, or can generate the code that is specialized for the frequently appearing values. • The JIT compiler selects the set of optimizations to apply based on the optimization level.
The configuration of the execution environment			
	The configuration of the execution environment, including both hardware and software information, is set at the time of the invocation of the Java VM.	• The processor specification (such as the model and the number of processors in the machine, and the cache size) • The type and version of the operating system • The command line options and the environment variables	• The JIT compiler can generate code that can run faster in a specific environment than generic code.

bootstrapping program can be the same Java VM as that was executed in the production environment. In this case, the replaying compiler never needs to use the runtime data of the bootstrapping Java VM, even though it is a Java VM.

The bootstrapping program can be any other program if it provides interfaces for the replaying compiler to retrieve the input from the system dump. In this case, it is possible to replay the compilation of the JIT for one operating system in another operating system if these compilers apply the same optimizations for both of the operating systems.

The bootstrapping process is invoked by specifying the name of a system dump file. The method to replay can be provided either by command line options or by reading it from console interactively. There is no restriction on the order of the methods to replay. For example, we can replay the compilation of all of the compiled methods in the reverse order of when they were compiled by the state-saving compiler.

3.3.2 Accessing the Input Saved in a System Dump

The replaying compiler reproduces the compilation process for the specified method. The function to obtain the input in the state-saving compiler must be modified for the replaying compiler to search for the results corresponding to the given parameters from the restored repository, instead of executing the function again. Any code reading directly from memory must be modified to obtain the values from the repository or from the data area of the Java VM that was saved in the system dump.

The versions of the source code for the state-saving and the replaying compilers must be synchronized, because this technique requires both of them to apply the same optimizations for the same inputs. This requirement does not cause a problem for version control because the state-saving and the replaying compilers can be implemented by modifying the same base JIT compiler. All of the source codes that obtain the inputs in the base compiler are modified to save and restore, respectively, the inputs. Using conditional compilation (i.e., #ifdef), both compilers can be built from a single source code.

One problem when modifying the code accessing the input directly is any memory dereference through a pointer restored from the system dump. Such a pointer points to an address in the address space of the process that executed the state-saving compiler. Therefore, accessing this address in the replaying compiler may access an unexpected address, which might be invalid, because the memory layout of the replaying process is not necessarily the same. Thus, the replaying compiler needs to adjust the values of all of these pointers to the corresponding addresses in the replaying process.

This adjustment can be avoided by restoring the data area of the Java VM and the repository into the same address in the replaying process. Since the inputs to the compiler are stored in the process heap and the mapped memory region, and not stored in the stack, the bootstrapping process needs to restore only the heap and the mapped memory regions. The bootstrapping process should be able to restore these areas if it restores them at the beginning of its initialization. The disadvantage of this approach is that the feasibility very much depends on the target operating system. It also prevents replaying a compilation that took place in a different operating system.

3.4 Discussion

The Java VM may also unload a class and delete the contents of the class from the memory if the class is no longer used by any class in the Java VM. The class unloading is becoming a common event in modern Java applications, especially for those that use generated bytecode or adopt a plug-in based componentized model, such as Eclipse. Since our technique uses the system dump to save the fixed input, such as Java bytecode and the string constants, the replaying compiler cannot restore the fixed input for the unloaded classes, and it fails to replay such a compilation.

However, the unloaded classes are usually unnecessary for the problem determination. A Java VM can unload a class only when all of the classes that can access the class to be unloaded can also be unloaded [8]. In other words, the classes of live objects and methods referenced from stack frames are never unloaded. When a Java VM crashes because of an error in a JIT-compiled method, the problematic method usually has a stack frame, and thus its class should not have been unloaded.

It is also possible to implement state-saving and replaying compilers that could replay the compilation of an unloaded method. To do this, the state-saving compiler could save the contents of the unloaded class to a disk when it is unloaded. The state-saving compiler would also record the timestamps when each class was unloaded and when each method was compiled. The replaying compiler would recreate the process memory image when the target method was compiled by using the system dump and the file to save the contents of the unloaded classes. The replaying compiler would use the timestamps to find the memory image each of the compilations.

Since the unloaded class would have been saved in the middle of the execution of a user application, this would cause additional overhead for the replay JIT compilation. This overhead for saving to a disk could be reduced by saving several unloaded classes at the same time, but not by a large amount, because the access to a disk for a single chunk of unloaded classes is usually faster than several separated accesses for the unloaded classes in small sizes. The repositories for the unloaded, but not yet saved to a disk, classes are saved into the system dump when the Java VM crashes before saving those classes to a disk, because they stay in memory until they are saved to a disk, though they are made invisible in the class hierarchy.

Because of this additional overhead, whether or not to support replaying the unloaded methods is an option for the compiler writer, calling for balancing between the lower execution overhead and the higher reliability for reproducing the compilation process.

Table 2. An example of the values to be saved in a repository

The input to a JIT compiler	The value to be saved into a repository
The bytecode	The address of the compiling method.
The string constant	The address of the string constant.
The status of class initialization	A flag for each class indicating if the class has been initialized.
The address of the JIT compiled code	The list of the addresses of the compiled code blocks invoked from the compiling method.
The saved results of inter-procedural analysis	The list of the pointers of the classes that holds the result of the escape analysis.
The resolution status of external references	A bitmap indicating which of the external references have been resolved.
The class hierarchy	The parameters and the result of each function call that checks if there is only a single implementation of a virtual method. It returns the address if a single implementation exists.
The results of an on-line profiler	The result and the parameters to read the result from the data structure of the on-line profiler.
The level of the optimization	The value of the optimization level.
The configuration of the execution environment	The number and the specification of the processor in the system, and the type and version of the operating system.
The command line options and the environment variables	The address of the data structure that holds the parsed command line options and the environment variables.

4. IMPLEMENTATION

This section describes our prototypes of a state-saving compiler and a replaying compiler, which we have implemented based on the J9 Java VM [3] and the TR JIT compiler for AIX.

4.1 State-Saving Compiler

The state-saving compiler saves the appropriate values for each of the input types shown in Table 1 into the repository. Table 2 shows examples of the values that are saved into the repository for each of the input types. Our state-saving compiler allocates a memory area as the repository for each compilation of the methods. The repository works as if it were a cache, so that the compiler can avoid saving duplicated input that happened to be constant during the compilation.

If the JIT compiler implements recompilation that compiles a hot method more than once to optimize it more aggressively, the state-saving compiler needs to manage the repository by associating it with the address of the compiled code rather then the address of the method. Then, it is possible to identify the repository for a particular compilation of a method at a given optimization level based on the address of the compiled code.

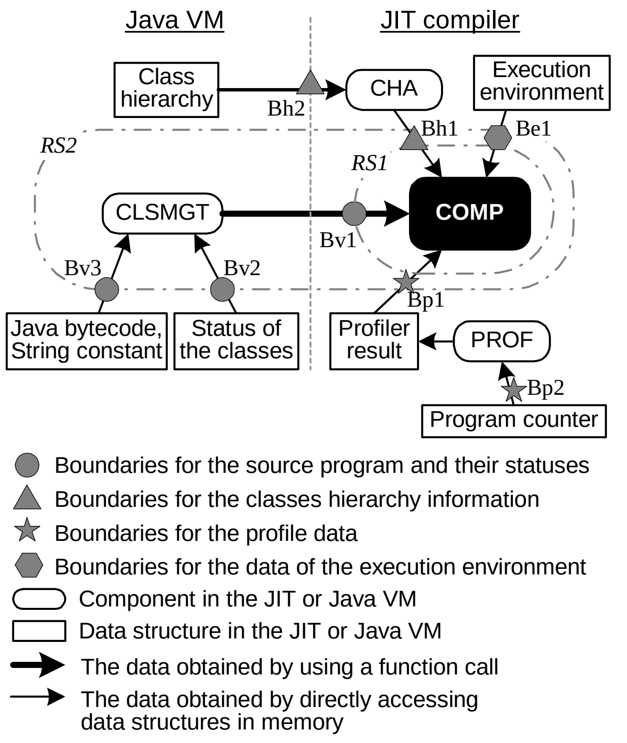
Java VM
JIT compiler
Class hierarchy
CHA
Execution environment
Bh2
Bh1
Be1
RS2
RS1
CLSMGT
COMP
Bv1
Bv3
Bv2
Bp1
Java bytecode, String constant
Status of the classes
Profiler result
PROF
Bp2
Program counter
Boundaries for the source program and their statuses
Boundaries for the classes hierarchy information
Boundaries for the profile data
Boundaries for the data of the execution environment
Component in the JIT or Java VM
Data structure in the JIT or Java VM
The data obtained by using a function call
The data obtained by directly accessing data structures in memory

Figure 3. The components of our JIT compiler

Table 3. Description of each component

Component	Description
COMP	The main component of our JIT compiler, which consists of optimizers and a code generator.
CLSMGT	The component to manage loaded classes and their status. The status includes the resolution status of the external references and the set of initialized classes.
PROF	The on-line profiler component.
CHA	The component to keep track of the current the hierarchy of loaded classes. The data is used for class hierarchy analysis so that the JIT can devirtualize virtual and interface method invocations.

Since the class unloading did not occur in our measurement environment, this prototype does not save the repositories for the unloaded classes in the middle of the execution.

4.2 Reducing the Size of the Repository

This section describes the techniques applied to reduce the total size of repository in a Java VM.

4.2.1 Choosing the Replay Scope

In this section, we will discuss the process to find the replay scope that minimizes the size of the repository in our JIT compiler. This consists of four major components as shown in Figure 3. Table 3 describes each component briefly. The main component of the compiler, COMP, obtains the input from CLSMGT and CHA by using function calls and from PROF by accessing memory directly. COMP uses five kinds of input: fixed input such as bytecode and string constants, the statuses of the classes, the class hierarchy information, the profiler results, and information about the execution environment. There are eight points that can be the boundaries of a replay scope (Bv1, Bv2, Bv3, Bh1, Bh2, Bp1, BP2, and Be1).

As a first step, we defined a replay scope RS1 that only contains COMP, and compared the repository size against that of another replay scope RS2 that contains COMP and CLSMGT. For RS1, the compiler needs to save the values of the fixed input from CLSMGT at Bv1 because COMP gets them using function calls. For RS2, the compiler saves the values of the variable input at Bv2 and the addresses of the fixed input at Bv3, but it does not save the values of the fixed input because they will be automatically saved into the system dump. As a result, the size of the repository for RS1 was approximately 1.8 times larger then the size of the JIT-compiled code, but that for RS2 was approximately 24%.

Then we compared the size of the repository for RS2 against that of other replay scopes to find the replay scope with the smallest repository. There are two more replay scopes: RS2+CHA that added CHA into RS2, and RS2+PROF that added PROF into RS2.

The difference of the size of the repositories between RS2 and RS2+CHA is the difference of the size of the input to be saved at Bh1 and Bh2, respectively. The compiler calls the functions of CHA to check if a method can be devirtualized. For RS2, the compiler needs to save the parameters and the return values of the function calls.

For RS2+CHA, the compiler needs to save the parameters and the time stamp of the function calls, and it also saves the time stamps of each class loading event to record the order of the class loading and the function calls. Thus, the size of the repository for RS2+CHA is larger than that of RS2 by at least the size of the time stamps of the class loading events, because the size of the time stamp of function calls is equal to or larger than the size of the return value for the function calls. Another drawback of RS2+CHA is that we cannot discard the time stamps of class loading even if they get old, as described in the next section, because all of the records are necessary to rebuild the class hierarchy for an error.

The size difference for the repositories between RS2 and RS2+PROF is the difference of the size of the input values saved at Bp1 and Bp2, respectively. Since there are usually a few methods that use the profiler results for their compilations, COMP needs the profile result at Bp1 only when it recompiles a very frequently executed method with more aggressive optimizations. In comparison, since PROF updates the profile data at short intervals, such as 10 ms, the size of the input at Bp2 is too large to save. Thus, the size of the repository for RS2 must be smaller.

As discussed above, RS2 is the replay scope for the smallest repository for the JIT compiler of Figure 3. On the other hand, implementing the state-saving compiler for RS2 requires more workload than that for RS1 because it needs to modify CLSMGT in addition to COMP.

4.2.2 Filtering Trustworthy Methods

We can reduce or limit the total size of the repository in a Java VM by considering how probable it is that a particular JIT-compiled method has an error and by assuming that those methods that are unlikely to have any error are unnecessary to replay for the problem determination.

The level of trustworthiness is not uniform for all JIT-compiled methods, but depends on various factors, such as the category of the method to be compiled and the time after its compilation. We call this metric the *confidence*. We can optionally adopt a filtering technique to reduce or limit the total size of the repository by discarding the repositories for those methods with high confidence. We can use multiple factors to define the confidence of a method and the order of discarding the repositories.

For example, one factor affecting the confidence is how long it has been since the method was compiled. A fatal problem in a compiled method is most likely to crash the process in the first few executions after compilation. For another example, confidence depends on the complexity of the method. Since the path length in the compiler is usually longer to compile a complex method, it is more likely to cause a problem for the compiler. We can use the size of the intermediate representation for a method as it is being compiled, the size of the compiled code, or the time taken to compile the method as a metric of complexity.

Confidence also depends on the category of a method. The methods that are commonly used in many programs can be considered less likely to cause an error because they should be well tested during the development of the JIT compiler. For example, the methods of system classes, such as the java.lang package, are used in many programs, and most of the paths in the compiler used to compile these methods should have been executed during development of the compiler. Therefore, we can reasonably believe that those methods will not cause an error.

The drawback for using this filtering technique is the possible difficulty in the problem determination if a needed repository was discarded. For example, this could occur when the confidence is defined based on the assumption that all paths in the JIT compiled code were executed, but the problem exists only in a rarely executed path, perhaps code used for exception handling. The compiler designer can conserve repository memory, but such filtering may decrease the reliability of the problem determination.

4.2.3 Compaction and Compression

The state-saving compiler can reduce the size of repository by removing any runtime datum whose value is the same as a predefined default for that input. Since the replaying compiler uses the default value if it fails to find a value for an input in the repository, it uses the same input as the state-saving compiler.

The default value is usually a conservative value subject to further improvement of this technique. The replay compilation technique may be able to narrow down the cause of a problem by changing the parameters to control the optimizations that are used. In such a case, the replaying compiler may need an input that was not accessed by the state-saving compiler, and it uses the default value. The conservative default value might disable an aggressive optimization, which might prevent us from reproducing the problem, or even cause a new problem.

The state-saving compiler can also compress a repository using a well-known compression technique, such as zlib [18].

4.3 Replaying Compiler

For the case of our implementation, the bootstrapping Java VM restores all of the repositories and the data area of the Java VM for running the state-saving compiler from a system dump during its initialization, and then invokes the replaying compiler. The bootstrapping process needs to find a repository within a block of binary data, because a system dump saves the contents of the data area of the process memory space as a block of unstructured binary data. Our state-saving compiler manages all repositories using such a data structure that all of them are reachable from a single pointer, such as a linked list. The single pointer is stored in an anchor data structure that has signature words in its header and trailer. The bootstrapping process scans the signature words in the system dump to find the saved anchor structure. When it finds the signature words, it verifies the size, finds the single starting pointer, and then finds all of the saved repositories by walking through the data structure.

The anchor structure has another pointer variable that holds the address of the repository for the currently compiling method (called the *current repository*). Since an incomplete repository will crash the replaying compiler, the current repository should not be accessible in the list of "complete" repositories. While the JIT is compiling a method, the pointer holds the address of the current repository, 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.

Table 4. Configuration 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 5. Evaluated programs

Program	Description
mtrt, jess, compress, db, mpegaudio, jack, javac	Each of the programs included in the benchmarks suite SPECjvm98 [14].
SPECjbb	The SPECjbb2000 [14] benchmark.
xml parser	The operation to parse a sample XML file using the XML parser for Java [17]. 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 [6], and load the default top page using a Web browser. The execution speed was not measured because this is an I/O bound program.

5. EXPERIMENTAL RESULTS

We measured the memory and execution speed overhead for saving the input into repositories. We used the prototypes of the state-saving and the replaying compilers described in Section 4 for these measurements. Table 4 describes the configurations of the machines used for these measurements. Table 5 describes the programs we used for the measurements.

Our prototype successfully reproduced the compilation processes for all of these programs in each of the three tested machines by executing the state-saving and the replaying compilers in the same machine. This prototype always creates a system dump when it

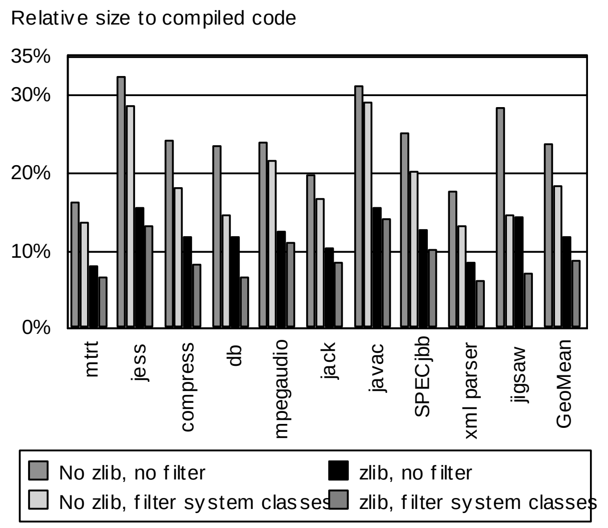

Figure 5. The sizes of repositories with compression and filtering

Table 6. The number of the saved repositories

Program	No filter	Filter system classes	Reduction of the 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%

terminates. In addition, the replaying compiler also successfully reproduced the compilation processes by using the system dump generated by a different machine. The replaying compiler succeeded in replaying all six of possible combinations of the machines to execute the state-saving compiler and the replaying compiler.

5.1 The Size of Repository

Figure 4 shows how the total size of repositories changes for the chosen replay scope. The size is relative to the total size of the compiled code. We measured the total size of the repository when we chose both of the replay scopes described in Section 4.2.1, RS1 and RS2. We applied neither compression using the zlib library nor any filtering for the methods of high confidence.

The total size of the repository for RS1 and RS2 was 180% and 24%, respectively, relative to the compiled code. (All percentages are geometric means.) Since RS1 saves the fixed input into the repository as well as the variable input, the size of the repository was much larger than that of RS2. As shown by this result, it is important to choose an appropriate replay scope that avoids saving the return values from function calls.

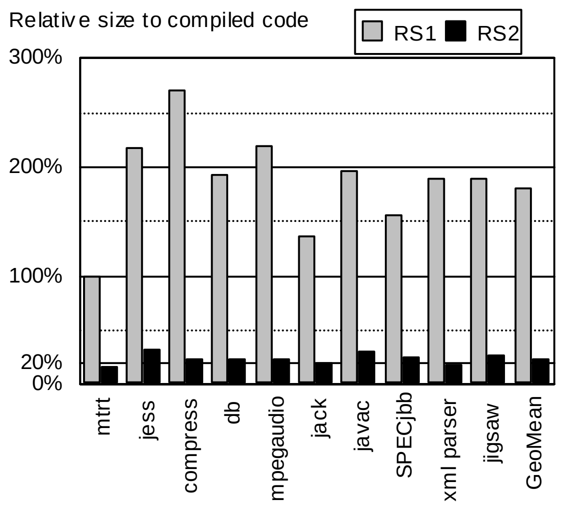

Figure 4. The sizes of repositories for both replay scopes

Figure 5 shows how the total size of the repository changes due to compression and filtering. The size is relative to the total size of the compiled code. The replay scope for this experiment is RS2. 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 filtering and holds the repositories of all of the compiled methods. 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 compiler does not discard any old repositories in these measurements.

The reduction of the size of the repository by filtering the system classes was 22.7% and 25.4% without and with compression using the zlib library, respectively. The reduction of the number of the saved repositories was 38% as shown in Table 6. This table shows the number of repositories that were saved in a process memory area and which were to be saved in a system dump when filtering is and is not used. It also shows the ratios of reduction of the number of methods by filtering and of the total size of repositories when no compression is applied. We think the reason the size reductions are relatively smaller than the reductions in the number of repositories (due to filtering out the methods of the system classes) is because many of the filtered methods are smaller than average methods.

Compression reduced the total size of the repositories by approximately half. Filtering reduced the total size of repositories by 25.4% over the compressed repositories, and the reduction made by the combination of compression and filtering was 62.9%. As a result, the geometric mean of the sizes of the repositories was reduced to less than 10% of the compiled code, which was our initial target as being acceptable for many users.

The total size of the repository is less than 0.1% of the total memory usage of the Java VM process because the size of the Java heap is much larger. Thus, either of the replay scopes may be

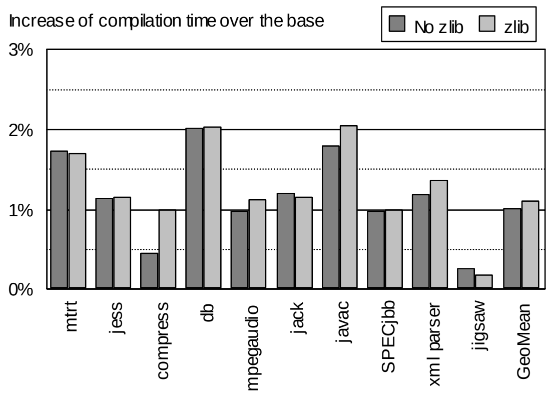

Figure 6. Compilation time increases when saving input

suitable for some users. However, we think it is still meaningful to keep the size of the repository 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, but the repository is used only for the problem determination.

5.2 Compilation Time

Figure 6 shows the increases of compilation times in the state-saving compiler over the times to compile the same methods using the base compiler. The bars labeled "No zlib" show the compilation times when no compression was used, and the bars labeled "zlib" show those times with zlib compression. Filtering was not applied for both cases in order to measure the upper bound of the overhead when compressing all of the registries.

Since our JIT compiler implements profile-based recompilation, the number of compiled methods may not always be the same. Therefore, we compared the compilation times of the methods that were compiled in both the base and the state-saving compiler.

The increase of compilation time was up to 2.0% for both cases. The geometric mean of the increase was about 1.0% and 1.1% for "No zlib" and "zlib", respectively. Thus, the increase of the compilation time was very small, and the time to save the input to the repositories was negligible.

5.3 Execution Speed

The additional overhead in the state-saving compiler against the base compiler is used to save the input into repositories, because the state-saving compiler uses the same inputs as the base compiler and generates the same compiled code. There is no additional overhead in the compiled code.

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

6. CONCLUSION

We have proposed a new approach, called *replay JIT compilation*, to reproduce the same JIT compilation process offline and remotely by using two compilers, *the state-saving compiler* and *the replaying compiler*. The state-saving compiler is designed to run in the production environment as part of the Java runtime and save into *the repository* in the memory all of the inputs to the JIT compiler that are necessary for the replaying compiler to reproduce the same compilation process later. When the Java application fails, the operating system will automatically generate a system dump that includes the repository. We developed our prototype based on the J9 Java VM and the TR JIT compiler for AIX and showed that the prototype successfully reproduces the same compilation processes done by the state-saving compiler. We did a preliminary experiment, where the overhead of running the state-saving compiler is negligible and the size of the additional memory area required for state saving was only 10% of the compiled code. This is three orders of magnitude smaller than the size of the diagnostic output file. To our knowledge, this is the first report of successfully replaying the JIT compilation process offline.

7. ACKNOWLEDGMENTS

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 difficulty of the problem determination of the JIT compiled code in the mission-critical environment, 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 the members of the Systems Group in the IBM Tokyo Research Laboratory for helpful discussions and comments on the earlier version of this paper.

8. REFERENCES

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

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

[3] 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.

[4] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (PLDI), pp. 32-43. June, 1992.

[5] 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). pp. 294-310. October, 2000.

[6] 'Jigsaw - W3C's Server'. Available at 'http://www.w3.org/Jigsaw/'

[7] 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.

[8] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, available at ' http://java.sun.com/docs/books/vmspec/index.html'

[9] 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), pp. 135-144. June, 1988.

[10] 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), pp. 124-129. May, 1988.

[11] 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). April, 2001.

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

[13] 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), February, 2000.

[14] Standard Performance Evaluation Corporation. 'SPEC JVM98 Benchmarks', available at 'http://www.spec.org/osg/jvm98/' and SPECjbb-2000, available at 'http://www.spec.org/osg/jbb2000/'

[15] 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), pp. 158-167. August, 2000.

[16] M. M. Tikir, G. Y. Lueh, and J. K. Hollingsworth. Recompilation for Debugging Support in a JIT-Compiler. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering* (PASTE), pp. 10-17. November, 2002.

[17] 'XML Parser for Java'. Available at 'http://www.alphaworks.ibm.com/tech/xml4j'

[18] zlib, available at 'http://www.gzip.org/zlib/'