Recompilation for Debugging Support in a JIT-Compiler

Mustafa M. Tikir
Computer Science Department
University of Maryland
College Park, MD 20742
tikir@cs.umd.edu

Guei-Yuan Lueh
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052
guei-yuan.lueh@intel.com

Jeffrey K. Hollingsworth
Computer Science Department
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

ABSTRACT

A static Java compiler converts Java source code into a verifiably secure and compact architecture-neutral intermediate format, called Java *byte codes*. The Java byte codes can be either interpreted by a Java Virtual Machine or translated into native code by Java Just-In-Time compilers. Static Java compilers embed debug information in the Java class files to be used by the source level debuggers. However, the debug information is generated for architecture independent byte codes and most of the debug information is valid only when the byte codes are interpreted. Translating byte codes into native instructions puts a limitation on the amount of usable debug information that can be used by source level debuggers. In this paper, we present a new technique to generate valid debug information when Just-In-Time compilers are used. Our approach is based on the *dynamic recompilation* of Java methods by a fast code generator and lazily generates debug information when it is required. We also present three implementations for *field watch* support in the Java Virtual Machine Debugger Interface to investigate the runtime overhead and code size growth by our approach.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids.*

General Terms

Algorithms, Measurement, Performance.

Keywords

Java, Just-In-Time Compilation, Debug Information, Java Virtual Machine Debugger Interface, Field Access Watch, Dynamic Recompilation.

PASTE'02, November 18-19, 2002, Charleston, South Carolina, USA.

1. INTRODUCTION

A static Java compiler converts Java source code into a verifiably secure and compact architecture-neutral intermediate format, called Java *byte codes*. At runtime, the Java byte codes can be either interpreted by a Java Virtual Machine (JVM) or translated into native code by Java Just-In-Time (JIT) compiler to improve the runtime performance. The Java Virtual Machine Debugger Interface (JVMDI)[3] is a two-way programming interface used by debuggers and other programming tools to inspect, query and change the state of the Java programs. The JVMDI clients are notified of the interesting occurrences through events.

Static Java compilers embed debug information in Java class files in the form of attributes, which can be used by the source level debuggers. The debug information is generated for the architecture independent byte codes. That is, the debug information in a class file is sufficient if the Java byte codes are only interpreted by a JVM. However, translating byte codes into native instructions using JIT compilers puts a limitation on the amount of usable debug information. For example, the debug information in Java class files might contain a mapping from source code lines to byte code offsets but does not give any mapping from Java byte code offsets to native code segments generated by the JIT compilers.

Alternatively, the JIT compilers can dynamically generate and store additional debug information during the translation of byte codes into native code. However, this approach increases the memory requirement of the JIT compilers when the debug information is not needed. Instead, JIT compilers can simulate the translation of byte codes to native instructions and generate the debug information when it is required. This approach has to simulate byte code translations without introducing a significant overhead.

In this paper, we present a new approach to generate the necessary debug information when JIT compilers are used. Our technique generates debug information when it is required at runtime. Our technique uses dynamic recompilation of Java methods as an alternative to storing all debug information during the first compilation of the methods. To investigate the runtime overhead and code space growth for

our approach, we implemented three different algorithms to instrument programs for the field watch support in JVMDI. Our static and semi-static implementations insert all possibly needed instrumentation code during the translation of byte codes to native code. Static and semi-static approaches eliminate the need for symbol table information after the compilation of the Java methods. Our dynamic recompilation technique, however, simulates the byte code translations and lazily inserts instrumentation code at runtime when a field watch is activated first time.

The rest of the paper is organized as follows. Section 2 describes the compilation model of the Intel JVM. Section 3 presents our debugging support mechanisms using dynamic recompilation. Section 4 describes our approaches to implement the field watch support in JVMDI. Section 5 presents our experiments and results. Finally, Section 6 gives the concluding remarks.

2. COMPILATION MODEL

The Intel JVM [14] uses Java JIT compilers to improve the runtime performance. The Intel JVM consists of three major components: a fast code generator, an optimizing compiler, and profiling information[2]. Every method is compiled to native code by the fast code generator when it is first invoked. Profiling instrumentation code is inserted into the native code to collect the profiling information. Based on the collected profiles, some methods are identified as *hot* and then recompiled at runtime by the optimizing compiler. In this paper, we only used the fast code generator component of the Intel JVM and disabled the optimizing compiler component to avoid the complexity of handling the *code location* and *data-value* problems[12]. The global optimizations and instruction scheduling done by optimizing compiler can also cause the existence of the *endangered* or *non-resident* variables[13].

The fast code generator component of Intel JIT generates native IA32 instructions in two phases. The *prepass* phase performs a linear-time traversal of the byte codes to collect information and the *lazy code selection*[1] phase generates the native IA32 instructions directly from byte codes in a single pass using the collected information. Since the fast code generation takes only two linear time traversals over byte codes, the compilation is simple and fast.

3. DEBUGGING SUPPORT

In this section we broadly describe how and when the dynamic recompilation of methods is used to support stack frame accesses, data-value accesses and control breakpoints when the JIT compilers are used.

3.1 Stack Frame Accesses

To support stack frame accesses, the JIT compiler has to provide a mechanism to identify and access the caller's frame context. The stack unwinding process starts with a frame context of a thread and continues unwinding until it identifies the context of the caller. The mechanism already exists in the exception handling modules of the JVM to find the exception handler that catches the exception.

If the active frame belongs to a Java method compiled by the JIT compiler, the JVM makes calls to the JIT compiler to perform the unwinding operation. Information about the return *ip*, the location and size of the spill area, location for saving callee-save registers and outgoing arguments can only be obtained from the JIT compiler. To unwind stack frames for exceptions and compute the root set of live references for garbage collection, a Garbage Collection (GC) map is generated during the compilation for each method. In the Intel JVM, we take a lazy approach to compute the GC map [2]. A small GC map that contains some minimal but essential information is generated when a method is compiled for the first time. The JIT compiler dynamically recompiles the method to produce full GC map when it needs to compute the root set or unwind the stack frame.

3.2 Data-Value Accesses

The fast code generator uses a simple register allocation scheme to allocate the machine registers to the variables during native code generation. It maps the local variables to dedicated registers or locations in the spill area. For translated methods, only the JIT compiler stores information about the location of the variables.

To support data-value accesses, the JIT compiler provides a runtime library function that returns the address of the location where a local variable resides. The address of a variable can either be in the spill area of the stack frame or the callee-save area. The runtime library retrieves the address of the variable in three steps. 1) The full GC map is retrieved. If the full GC map is not already available, the JIT compiler dynamically recompiles the method that the variable queried is in. 2) The JIT compiler constructs the stack frame context for the current method to figure out the stack frame layout of the method. 3) The JIT compiler computes the address of the variable using the map from registers to variables, stored in the GC map.

3.3 Control Breakpoints

The JIT compilers must be able to map each byte code to its corresponding translated native instruction sequence. Intel's fast code generator performs several simple optimizations such as strength reduction and constant folding during compilation of byte codes to the native code. It does not perform any aggressive optimizations, which could potentially cause the *code location* problem (accurate mapping of source code to optimized machine code). The lazy code generator passes over the byte code sequence and generates the native instructions for each byte code sequentially.

The JIT compiler does not generate the code location map until it is needed. That is, when a breakpoint is set or cleared the code location map is generated using dynamic

recompilation. To set or clear a breakpoint, the JIT compiler provides an interface function that triggers dynamic recompilation and returns the native code offset for a given byte code in a method. Moreover, as soon as the lazy code selector reaches the byte code offset, recompilation is terminated and the next emit offset is returned. We do not store the location map generated but recompile the method every time a breakpoint is set or cleared.

3.4 Data Breakpoints

Data breakpoints, also called as *watchpoints*, provide users with a mechanism to stop the program execution in terms of the changes in program's memory states. Data breakpoints stop the execution when a specified memory location is accessed or modified. Data breakpoints can be set for accesses or modifications of local variables, class fields or arbitrary memory locations. Java is a type safe language and does not allow arbitrary memory accesses or updates in the heap. Therefore, JVMDI includes only interface functions for the *watchpoints* for the class fields.

The JVMDI specification for field watch requires an *event hook function* to be called every time the watched field is accessed or modified. Field access or modification events are generated when the field specified is about to be accessed or modified. JVMDI also supports cancellation of field watches during program execution. The hook function is called with an argument describing the event type and additional information specific to the event.

The field access events require information about the thread that the event occurred, the class/method that accessed the field, the location of the field access, the class field belongs to, the field itself, and the object of which the corresponding field is accessed. For modification events, additionally, the signature of the field and the new value are also passed.

When the byte codes are interpreted by a JVM, field watch support can be provided in a straightforward manner using the information in the constant pool of Java class files and operand stack. That is, while interpreting *getstatic*, *putstatic*, *getfield* and *putfiled* byte codes, the JVM calls the event hook function if the field watch is activated for the corresponding field. However, when the JIT compiler is used, the locations that fields are accessed or modified might not be easily identifiable. Moreover, the JIT compiler has to provide the ability to interrupt the execution before the field access and modification points to call the event hook function.

The Intel JIT compiler implements support for data breakpoints using a code-patching scheme[15]. In our implementations, we insert additional instrumentation code to the method's code space either just before the field access or modification points or at the end of the address space. Instrumentation code is inserted either during the method's first compilation or later when the field watch is activated. The instrumentation code passes necessary arguments to a runtime library function. The runtime library function calls the event hook function with the information passed to it. Moreover, to keep the state of the memory consistent before and after the call to the runtime library function, the JIT compiler generates spill code for the operands on the operand stack that are live across the call sites[1]. To prevent significant runtime overhead caused by the inactivated field watches, we guard the execution of the instrumentation code by modifying the method's address space or using Boolean flag for each Java class field.

4. Field Watch Support

To investigate the runtime overhead and code space growth for our dynamic recompilation approach, we present three different implementations to support field watch feature in JVMDI when JIT compilers are used. We believe the field watch feature is representative of other debug features for using our dynamic recompilation approach. Additionally, field watches are easy to implement and conduct experiments with. Our implementations differ in *where* and *when* the additional instrumentation code is inserted and *how* the execution of the instrumentation code is guarded.

4.1.1 Static Implementation

Static implementation inserts instrumentation code for the field watches during the compilation of a method when it is first invoked. Instrumentation code is inserted at every point the native code is generated for *getfield*, *getstatic*, *and putfield* and *putstatic* byte codes.

```
1.    aload 6
2.    getfield #42
3.           mov     ecx, DWORD PTR [esp + 10h]
4.           mov     edx, DWORD PTR [ecx + 04h]
5.           mov     DWORD PTR [esp + 08h], edx
6.           cmp     BYTE PTR [F10B78h], 01h
7.           jnz     NoWatch
8.           push    0f10b50h
9.           push    ecx
10.          call    10A837A0h
11.   iload 4
12.   if_icmpne 80
13.       NoWatch:
14.          mov     eax, DWORD PTR [esp + 18h]
15.          cmp     eax, DWORD PTR [esp + 08h]
```

Figure 1: Static Instrumentation for Field #42 Access

Figure 1 shows a byte code sequence of a method and the native instruction sequence for this segment. In Figure 1, the additional instrumentation code is given in lines 3-10. Lines 3-5 shows native instruction sequence to spill the contents of Java operand stack to their canonical spill locations. The operands that are spilled are live at the field access point. The code section in bold face, lines 6-10, is the main code segment to decide whether field access watch is activated or not.

Two values are passed to the library function as arguments, shown in lines 8-9. The first argument is the pointer to the

internal data structure for the corresponding class field in the JVM (field identifier). The second argument is the pointer to the object being accessed (object identifier).

In static implementation, to guard the execution of instrumentation code, the Intel JIT stores a Boolean variable for each class field in its internal data structures. The Boolean flag for the field is set to true when the field watch is activated and to false when it is cleared. In Figure 1, line 6 compares the value of the Boolean flag to decide whether the call to the event hook function will be executed. To activate or clear a field watch for a particular field, we simply set or clear the field's Boolean flag, respectively.

The spill code, lines 3-5 in Figure 1, is always executed during the program execution even though the field watch may not be activated. This is due to the fact that the memory state has to be kept consistent for the garbage collection and exception handling mechanism.

4.1.2 Semi-Static Implementation

Similarly, our semi-static implementation inserts instrumentation code during the first compilation of the methods. However, instead of inserting instrumentation code before the field access or modification points, it inserts all instrumentation code after the original byte code sequence is compiled and emitted. Semi-static implementation inserts *jump 0* instructions at the field access and modification points, which are used to jump into the instrumentation code when the corresponding field watch is activated.

```
1.   dconst_1
2.   aload_0
3.   getfield #159
4.       mov   eax, DWORD PTR [ebp + 08h]
5.       jmp   Watch
6.   i2d
     WatchRet:
7.       fild  DWORD PTR [eax + 44h]
8.   ldc2_w #191
9.   ddiv
10.      fld   QWORD PTR [F7DB08h]
11.      fdivr st(0), st(1)
12.      fstp  st(1)
     ___
13.  areturn
     ___

     Watch:
14.      push  eax
15.      mov   ecx, DWORD PTR [eax + 44h]
16.      mov   DWORD PTR [ebp + FFFFFFB4h], ecx
17.      mov   edx, DWORD PTR [57F710h]
18.      mov   DWORD PTR [ebp + FFFFFFB8h], edx
19.      fstp  QWORD PTR [ebp + FFFFFFC0h]
20.      push  013282a0h
21.      push  eax
22.      call  10A837A0h
23.      fld   QWORD PTR [ebp + FFFFFFC0h]
24.      pop   eax
25.      jmp   WatchRet
```

Figure 2: Semi-Static Instrumentation for Field #159

The native code shown in Figure 2 corresponds to the state when the field access watch for field #159 is activated. Instrumentation code for the field access watch is given in lines 5 and lines 14-25 in bold face. Line 5 contains the jump instruction that is used to branch into the instrumentation code. The offset of the jump instruction is 0 if the field watch is not activated. When the field watch is activated, the JIT compiler changes the offset of jump instruction into the instrumentation code.

Unlike the static implementation, during the lazy code selection, the semi-static implementation assumes that the jump to the instrumentation code is never taken. Hence, no spill code is generated for the contents of the operand stack at the field access point (line 4). Spilling is delayed until the instrumentation code is actually executed (line 14). Since the spilling of operands does not happen at the field access or modification points, the semi-static implementation saves the global state of the registers before the event hook function is called and restores them afterwards. The global memory state varies from one field access or modification point to another. For instance, saving the floating-point stack is not required if no floating-point value is live after the field access or modification point. For memory and register consistency, however, we push the live scratch registers on the call stack (line 14) and restore them before returning to the function address space.

Unlike our static implementation, our semi-static implementation executes fewer instructions when no field watch is activated. Only a jump instruction is executed. However, semi-static implementation executes more instructions when the field watch is activated due to the additional code to maintain the register and memory state consistent. Our semi-static implementation also tries to prevent cache footprint changes by moving the instrumentation code at the end of the method's code space.

For the semi-static implementation, the JIT compiler keeps track of the locations where *jump 0* instructions are inserted. Activation or cancellation of a field watch is linear in the number of locations the field is accessed or modified. During the first compilation of a method, for the activated field watches in the method, semi-static implementation inserts jump instruction with the correct offset to the instrumentation code.

4.1.3 Dynamic Recompilation

One way to be able to generate the instrumentation code for field watch support is to store necessary information about the compilation states at field access and modification points, and to lazily use this information when a field watch is activated. At each field access or modification point, the compilation state includes the state and content of the operand stack, the state of the registers, as well as the information about the field itself. However, for programs with many field access and modification byte codes, this infor-

mation may significantly increase the memory usage of the JVM.

Our dynamic recompilation approach saves the memory usage at the cost of recompilation. When a field watch is activated, the JIT compiler recompiles the methods that access or modify the field and generates the debug information necessary for the instrumentation code. Dynamic recompilation always simulates the method's first compilation. Like in our semi-static implementation, we only insert *jump 0* instructions at the field access and modification points during the method's first compilation. For the dynamic recompilation approach, the same sequence of instructions as in Figure 2 is generated for the same code segment. The only difference is that instead of inserting the instrumentation code at the end of method's code space we allocate new stub space from heap for each instrumentation code.

The JIT compiler stores a map from field access and modification points to their stub addresses, if the stubs are already generated. The generated stubs remain in the code space even after the corresponding field watch is cleared. Like in our semi-static implementation, if a field watch is already activated, the JIT compiler generates the stub and inserts the jump to the stub during the first compilation of the method.

However, insertion of *jump 0* instruction is not absolutely required for our semi-static and dynamic implementations. The JIT compiler can alternatively relocate several of the original instructions, such as in [15][17], to the code stub (line 4) and replants it with a jump instruction into the stub.

5. EXPERIMENTS AND RESULTS

To evaluate the effectiveness of our dynamic recompilation approach, we ran a set of experiments with a different number of field watch activations using our static, semi-static and dynamic approaches. We tested our implementations with the benchmark programs in the SPEC JVM98[11] suite. The experiments are conducted on a 550MHz Pentium III with 512MB of main memory running Windows NT 4.0. We repeated our experiments several times and used the average values over all results. We measured the execution time slowdown ratios with respect to the original execution times and the code space growth percentage of benchmark programs using our approaches.

Table 1 presents the number of the field access and modification byte codes and number of the actual field accesses and modifications during the execution of benchmark programs. The fourth and fifth columns show the number of field accesses and modifications during the program execution in Millions.

Table 1: Number of field access/modification byte bodes in source code and calls in program execution

Benchmark Programs	Method Count	Static Field Accesses	Static Field Modifications	Field Accesses (M)	Field Modifications (M)
compress	318	1,226	444	1,973	392
db	330	1,244	426	458	15
jack	557	2,255	991	484	280
javac	1,100	4,610	1,162	277	81
jess	744	2,137	675	248	27
mpegaudio	481	2,120	637	951	148
mtrt	449	1,480	516	241	44

5.1 Slowdown by Field Watch Support

We evaluate our implementations for the field watch support under five different scenarios. These scenarios differ by the total number of the field watches that are activated or cleared during the execution of the benchmark programs. We used the scenario where all field watches are activated during the first compilation of the methods to investigate the maximum runtime overhead introduced. We also used the scenario without any field watch activation to investigate the degradation in performance of the benchmark programs when the field watch debugging support is not used. Moreover, we used three more scenarios that randomly activate and clear the field watches to investigate the runtime overhead introduced by dynamic recompilations triggered.

5.1.1 Scenario I: Every Field Watch is Activated

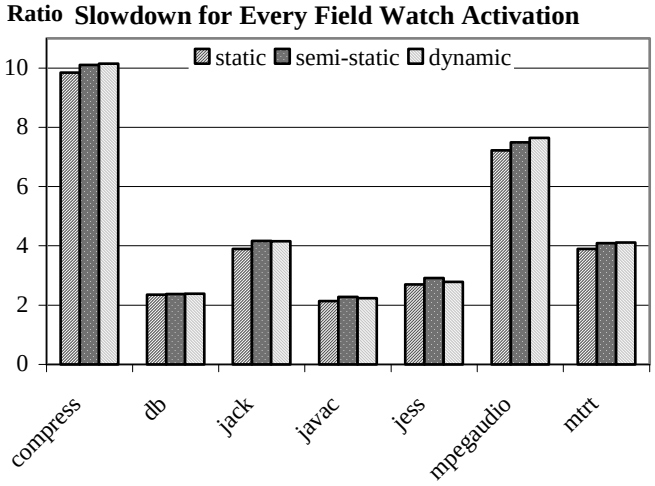

Figure 3: Slowdown when every field watch is activated during the first compilation of the methods

Figure 3 shows the execution time slowdown ratios of the benchmark programs when every field access and modification watch is activated during the first compilation of the methods. Under this scenario, no dynamic recompilation is triggered for our dynamic recompilation approach since all field watches are activated during the first compilation of

the methods. Figure 3 shows that the slowdown ratios for benchmark programs range from 2.1 to 10.2. The slowdown ratios are mainly proportional to the number of field access and modification calls during the program execution.

Figure 3 shows that all three implementations perform almost the same with slight differences. Static implementation performs slightly better compared to the others for all benchmark programs. This is due to the fact that less instrumentation code is executed for static implementation for each field watch activated.

5.1.2 Scenario II: No Field Watch is Activated

Figure 4 shows the execution time slowdown ratios for the scenario where no field access or modification watch is activated throughout the execution of the benchmark programs. Figure 4 shows that our static implementation performs significantly worse compared to the semi-static and dynamic implementations. This is due to the fact that for our static implementation, the spill code for the live operands in Java operand stack is executed for each field watch point even though it is not activated.

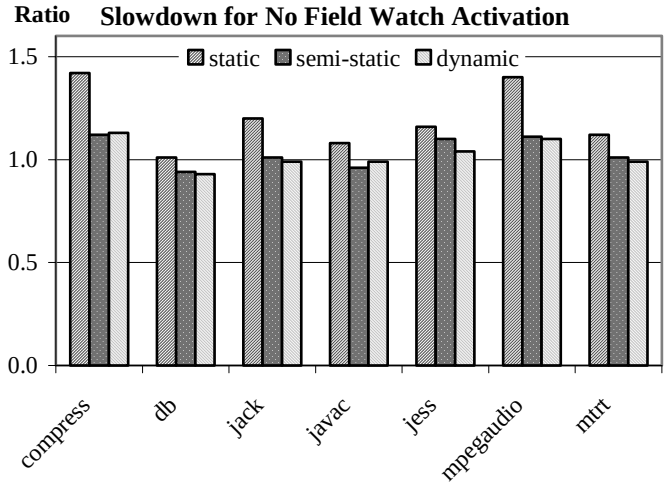

Figure 4: Execution time slowdown when no field watch is activated during the program executions.

The runtime overhead introduced by our static implementation is as high as 41.8%. The runtime overhead of our semi-static and dynamic implementations is under 12.0% and 13.0%, respectively. Moreover, runtime overhead for the semi-static and dynamic implementations can be completely eliminated by the relocation of original instructions into the stubs rather than inserting *jump 0* instructions. Figure 4 also shows negative slowdown for db with semi-static and dynamic implementations. We believe that the negative slowdown is due to the change in instruction cache footprint caused by the instrumentation code.

5.1.3 Random Field Watch Activation Scenarios

We evaluated our implementations under three more scenarios where the field access and modification watch activation occurs in a random manner. These scenarios differ in the number of requests to activate and clear the field watches. Based on the number of dynamic recompilations triggered, we label these scenarios as *heavy*, *moderate* and *light*. We believe that most of the software developers' behavior when debugging software fits into *light* scenario (Due to space consideration we present execution time slowdown and code space growth results for only *light* scenario).

Table 2: Number of recompilations triggered when field watches are activated or cleared randomly.

Benchmark Programs	#Method In Benchmark	Random Scenarios		
		Heavy	Moderate	Light
compress	318	397	90	21
db	330	372	85	21
jack	557	566	158	101
javac	1,100	1,316	514	175
jess	744	696	156	37
mpegaudio	481	611	125	31
mtrt	449	478	111	23

Table 2 presents the number of recompilations triggered by our dynamic recompilation implementation during the execution of the benchmark programs under *heavy*, *moderate*, and *light* scenarios. Table 2 shows that in *heavy* scenario, in the average, every method is compiled once more.

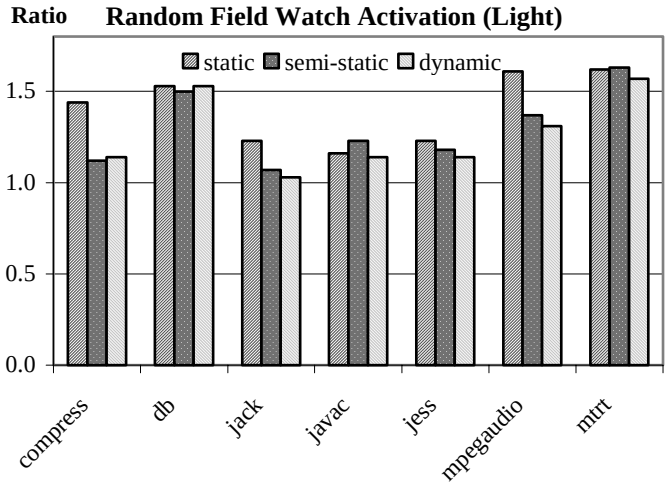

Figure 5: Execution time slowdown when a few field watches are activated.

Figure 5 shows the execution slowdown ratios of benchmark programs under the *light* scenario. In this scenario, the number of field access and modification points that call the event hook function is significantly less than in both heavy and moderate scenarios. In Figure 5, the execution time slowdown ratios for the benchmark programs range from 1.1 to 1.6. Moreover, the static and semi-static implementa-

tions are mostly outperformed by both our dynamic recompilation approach. The static implementation executes more additional instrumentation code compared to semi-static and dynamic implementations. Figure 5 also shows that dynamic recompilations do not introduce significant overhead compared to semi-static implementation, which inserts the same instrumentation code but statically during the first compilation of the methods.

5.2 Native Code Space Growth

We believe growth in generated code space is another important factor to evaluate the effectiveness of our implementations. Thus, we investigated the code space growth percentage of the benchmark programs compared to the original code space size of the translated native code.

Our static implementation inserts instrumentation code to the points where fields are accessed or modified independent from whether the corresponding field watches are activated or not. Thus, for our static implementation, the total code space size in terms of native instructions does not change under different scenarios. Similarly, the total code space size for our semi-static implementation does not change with respect to the different scenarios. For our dynamic implementation, the total code space size varies depending on the number of field watches activated. In all of our implementations, field watch deactivations do not have any affect on the size of the instrumentation code existing in the program's address space.

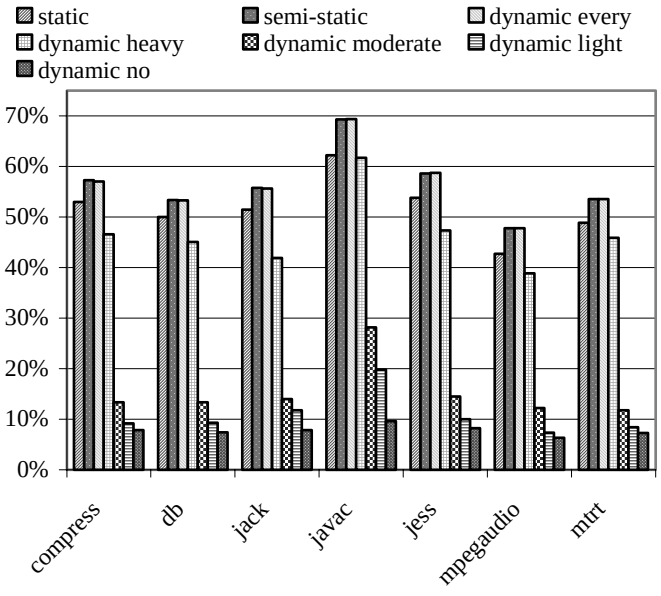

Figure 6: Code space growth percentage with respect to the original code size of the generated native code.

Figure 6 shows the code space growth percentage in terms of native instructions compared to the size of the original code without any field watch support. The first two bars in Figure 6 represent the code size growth percentage for our static and semi-static implementations for all scenarios. The third bar is used for our dynamic recompilation implementation when every field watch is activated during the first compilation. The next four bars are for dynamic recompilation for the *heavy*, *moderate*, and *light* scenarios, and the scenario for no field watch activations occur, respectively.

Figure 6 shows that the growth in code space ranges from 42.7% to 62.2% compared to the original code size for the benchmark programs when static implementation is used. Our semi-static implementation always introduces more instrumentation code compared to the static implementation, which results in a code space growth percentage that ranges from 47.8% to 69.3%. The code space growth percentage for our dynamic implementation is the same as our semi-static implementation when every field watch is activated during the first compilation of the methods.

The forth bar shows the percentage of code space growth for the heavy scenario using our dynamic implementation. Even for the *heavy* scenario, the dynamic approach is still able to reduce the code size significantly compared to the first three approaches. The code size growth drops substantially for *moderate* and *light* scenarios (fifth and sixth bars) using dynamic implementation. Under the *moderate* scenario the code space growth percentage is below 15% for all the benchmark programs except *javac*. Under the *light* scenario the code space growth is even smaller and ranges around 10% for most of the benchmark programs. The last bar in Figure 6 shows the code space growth percentage for our dynamic implementation when there is no field watch activation, which can totally be eliminated by relocation of original instructions into the stubs rather than inserting jump instructions.

6. CONCLUSIONS

In this paper we presented debugging support for the Intel JVM when JIT compilers are used. In the Intel JVM, dynamic recompilation of Java methods is used to provide debugging support for stack frame accesses, control breakpoints, data-value accesses, and data breakpoints.

Our experiments show that static implementation for field watch support significantly increases the size of the code space of the benchmark programs independent of the number of the field watches activated. Similarly, our semi-static implementation increases the code space size by 47.8-69.3% for the benchmark programs even if no field watch is activated.

Dynamic recompilation however controls the growth in code space depending on the number of the field watches activated during the program execution. For our dynamic implementation code space grows by 6.3-9.6% when there is no field watch activation. Moreover, the code space growth can totally be eliminated by relocation of the original instructions instead of inserting *jump 0* instructions.

Our experiments also show that when only a few field watches are activated, dynamic recompilation outperforms the static and semi-static implementations. Our dynamic recompilation implementation slows down the execution of benchmark programs by at most 13% when there is no field watch activation, which can also be eliminated completely by relocating original instruction(s) to the stubs instead of inserting *jump* instructions. Our experiments show that our dynamic recompilation approach is effective controlling the code size growth without degrading the performance significantly.

Acknowledgements

We thank Tatiana Shpeisman and Michal Cierniak for their valuable help in understanding the internals of the Intel JVM and JIT compilers.

7. REFERENCES

[1] A. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V.M. Parikh, and J.M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. *Conference on Programming Language Design and Implementation*, May 1998, pp. 280-290.

[2] M. Cierniak, G-Y. Lueh, J. M. Stichnoth, Practicing JUDO: Java™ Under Dynamic Optimizations, *Conference on Programming Language Design and Implementation*, June 2000, pp. 13-26

[3] Sun Microsystems, Java Virtual Machine Debug Interface Reference. http://java.sun.com/j2se/1.3/docs/guide/jpda/jvmdi-spec.html

[4] A. V. Aho, R. Sethi, and J. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[5] K. Arnold and J. Gosling. The Java Programming Language. Addison-Wesley, 1997.

[6] J. Gosling, B. Joy and G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[7] Intel Corp. Intel Architecture Software Developer's Manual, 1997 (Order number 243192).

[8] Intel Corp. Intel IA-64 Architecture Software Developer's Manual, 2000 (Order number 245319).

[9] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Second Edition. Addison-Wesley, 1999.

[10] J.M. Stichnoth, G.-Y. Lueh, and M. Cierniak. Support for Garbage Collection at Every Instruction in a Java Compiler. *Conference on Programming Language Design and Implementation*, May 1999, pp. 118-127.

[11] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98

[12] P. Zellweger. Interactive Source-Level Debugging of Optimized Programs. PhD Thesis, University of California, Berkeley, May 1984.

[13] A. Adl-Tabatabai and T. Gross. Source-Level Debugging of Scalar Optimized Code. *Conference on Programming Language Design and Implementation*, May 1996, pp. 33-42.

[14] Intel Corporation, Open Runtime Platform. http://www.intel.com/research/mrl/orp.

[15] P.B. Kessler. Fast Breakpoints: Design and Implementation. *Conference on Programming Language Design and Implementation*, June 1990, pp. 78-84.

[16] R. Wahbe. Efficient Data Breakpoints. In Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pp. 200-212.

[17] R. Buck and J.K. Hollingsworth. An API for Runtime Code Patching, Journal of High Performance Computing Applications, 14 (4), Winter 2000, pp. 317-329

[18] R. Wahbe, S. Lucco, and S.L. Graham. Practical Data Breakpoints: Design and Implementation. *Conference on Programming Language Design and Implementation*, June 1993, pp. 1-12