OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java

Hirotaka Ogawa[1], Kouya Shimura[2], Satoshi Matsuoka[1], Fuyuhiko Maruyama[1], Yukihiko Sohda[1], and Yasunori Kimura[2]

[1] Tokyo Institute of Technology
[2] Fujitsu Laboratories Limited

Abstract. OpenJIT is an open-ended, reflective JIT compiler framework for Java being researched and developed in a joint project by Tokyo Inst. Tech. and Fujitsu Ltd. Although in general self-descriptive systems have been studied in various contexts such as reflection and interpreter/compiler bootstrapping, OpenJIT is a first system we know to date that offers a stable, full-fledged Java JIT compiler that plugs into existing monolithic JVMs, and offer competitive performance to JITs typically written in C or C++. This is in contrast to previous work where compilation did not occur in the execution phase, customized VMs being developed ground-up, performance not competing with existing optimizing JIT compilers, and/or only a subset of the Java language being supported. The main contributions of this paper are, 1) we propose an architecture for a reflective JIT compiler on a monolithic VM, and identify the technical challenges as well as the techniques employed, 2) We define an API that adds to the existing JIT compiler APIs in "classic" JVM to allow reflective JITs to be constructed, 3) We show detailed benchmarks of run-time behavior of OpenJIT to demonstrate that, while being competitive with existing JITs the time- and space-overheads of compiler metaobjects that exist in the heap are small and manageable. Being an object-oriented compiler framework, OpenJIT can be configured to be small and portable or fully-fledged optimizing compiler framework in the spirit of SUIF. It is fully JCK compliant, and runs all large Java applications we have tested to date including HotJava. We are currently distributing OpenJIT for free to foster further research into advanced compiler optimization, compile-time reflection, advanced run-time support for languages, as well as other areas such as embedded computing, metacomputing, and ubiquitous computing.

1 Introduction

Programming Languages with high-degree of portability, such as Java, typically employ portable intermediate program representations such as bytecodes, and utilize *Just-In-Time compilers (JITs)*, which compile (parts of) programs into native code at runtime. However, all the Java JITs today as well as those for other languages such as Lisp, Smalltalk, and Self, are monolithically architected without provision for user-level extension. Instead, we claim that JITs could be utilized and exploited more opportunely in the following situations:

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 362–387, 2000.

- *Platform-specific optimizations* — Execution platforms could be from embedded systems and hand-held devices to large servers and massive parallel processors (MPPs). There, requirements for optimizations differ considerably, according to particular class of applications that the platform is targeted to execute. JITs could be made to adapt to different platforms if it could be customized in a flexible way.
- *Platform-specific compilations* — Some platforms require assistance of compilers to generate platform-specific codes for execution. For example, DSM (Distributed-Shared Memory) systems and persistent object systems require specific compilations to emit code to detect remote or persistent reference operations. If one were to implement such systems on Java, one not only needs to modify the JVM, but also the JIT compiler. We note that, as far as we know, representative work on Java DSM (cJVM[2]) and persistent objects (PJama[3]) lack JIT compiler support for this very reason.
- *Application-specific optimizations* — One could be more opportunistic by performing optimizations that are specific to a particular application or a data set. This includes techniques such as selection of compilation strategies, runtime partial evaluation, as well as application-specific idiom recognition. By utilizing application-specific as well as run-time information, the compiled code could be made to execute substantially faster, or with less space, etc. compared to traditional, generalized optimizations. Although such techniques have been proposed in the past, it could become a generally-applied scheme and also an exciting research area if efficient and easily customizable JITs were available.
- *Language-extending compilations* — Some work stresses on extending Java for adding new language features and abstractions. Such extensions could be implemented as source-level or byte-code level transformations, but some low-level implementations are very difficult or inefficient to support with such higher-level transformations in Java. The abovementioned DSM is a good example: Some DSMs permit users to add control directives or storage classifiers at a program level to control the memory coherency protocols, and thus such a change must be done at JVM and native code level. One could facilitate this by encoding such extensions in bytecodes or classfile attributes, and customizing the JIT compilers accordingly to understand such extensions.
- *Environment- or Usage-specific compilations and optimizations* — Other environmental or usage factors could be considered during compilation, such as adding profiling code for performance instrumentation, debugging etc. [1]

Moreover, with Java, we would like these customizations to occur within an easy framework of portable, security-checked code downloaded across the network. Just as applets and libraries are downloadable on-the-fly, we would like the JIT compiler customization to be so as well, depending on the specific platform, application, and environment. For example, if a user wants to instrument

[1] In fact we do exactly that in the benchmarking we show later in Section 5, which for the first time characterizes the behavior of a self-descriptive JIT compiler.

his code, he will want to download the (trusted) instrumentation component on-the-fly to customize the generated code accordingly.

Unfortunately, most Java JITs today are architected to be closed and monolithic, and do not facilitate interfaces, frameworks, nor patterns as a means of customization. Moreover, JIT compilers are usually written in C or C++, and live in a completely separate scope from normal Java programs, without enjoying any of the language/systems benefits that Java provides, such as ease of programming and debugging, code safety, portability and mobility, etc. In other words, current Java JIT compilers are "black boxes", being in a sense against the principle of modular, open-ended, portable design ideals that Java itself represents.

In order to resolve such a situation, the collaborative group between Tokyo Institute of Technology and Fujitsu Limited sponsored by Information Promotion Agency of Japan, have been working on a project OpenJIT[19] for almost the past two years. OpenJIT itself is a "reflective" Just-In-Time open compiler framework for Java written almost entirely in Java itself, and plugs into the standard Sun JDK 1.1.x and 1.2 JVMs. All compiler objects coexist in the same heap space as the application objects, and are subject to execution by the same Java machinery, including having to be compiled by itself, and subject to static and dynamic customizations. At the same time, it is a fully-fledged, JCK (Java Compatibility Kit) compliant JIT compiler, able to run production Java code. In fact, as far as we know, it is the ONLY Java JIT compiler whose source code is available in public, and is JCK compliant other than that of Sun's. And, as the benchmarks will show, although being constrained by the limitations of the "classic" JVMs, and still being in development stage lacking sophisticated high-level optimizations, it is nonetheless equal to or superior to the Sun's (classic) JIT compiler on SpecJVM benchmarks, and attains about half the speed of the fastest JIT compilers that are much more complex, closed, and requires a specialized JVM. At the same time, OpenJIT is designed to be a compiler framework in the sense of Stanford SUIF[28], in that it facilitates high-level and low-level program analysis and transformation framework for the users to customize.

OpenJIT is still in active development, and we are distributing it for free for non-commercial purposes from `http://www.openjit.org/`. It has shown to be quite portable, thanks in part to being written in Java—the Sparc version of OpenJIT runs on Solaris, and the x86 version runs on different breeds of Unix including Linux, FreeBSD, and Solaris. We are hoping that it will stem and cultivate interesting and new research in the field of compiler development, reflection, portable code, language design, dynamic optimization, and other areas.

The purpose of the paper is to describe our experiences in building OpenJIT, as well as presenting the following technical contributions:

1. We propose an architecture for a reflective JIT compiler framework on a monolithic "classic" JVM, and identify the technical challenges as well as the techniques employed. The challenges exist for several reasons, that the JIT compiler is reflective, and also the characteristics of Java, such as its pointer-safe execution model, built-in multi-threading, etc.

2. We show an API that adds to the existing JIT compiler APIs in "classic" JVM to allow reflective JITs to be constructed. Although still early in its design, and requiring definitions of higher-level abstractions as well as additional APIs for supporting JITs on more modern VMs, we nonetheless present a minimal set of APIs that were necessary to be added to the Java VM in order to facilitate a Java JIT compiler in Java.
3. We perform extensive analysis of the performance characteristics of OpenJIT, both in terms of execution speed and memory consumption. In fact, as far as we know, there have not been any reports on any self-descriptive JIT compilation performance analysis, nor memory consumption reports for any JIT compilers. In particular, we show that (1) JIT compilation speed does not become a performance issue, especially during the bootstrap process when much of the OpenJIT compiler is run under interpretation, (2) memory consumption of reflective JITs, however, could be problematic due to recursive compilation, especially in embedded situations, (3) that there are effective strategies to solve the problems, which we investigate extensively, and (4) that the solutions do not add significant overhead to overall execution, due to (1). In fact, the self-compilation time of OpenJIT is quite amortizable for real applications.

2 Overview of the OpenJIT Framework

2.1 OpenJIT: The Conceptual Overview

OpenJIT is a JIT compiler written in Java to be executed on "classic" VM systems such as Sun JDK 1.1.x and JDK 1.2.x. OpenJIT allows a given Java code to be portable and maintainable with compiler customization. With standard Java, the portability of Java is effective insofar as the capabilities and features provided by the JVM (Java Virtual Machine); thus, any new features that has to be transparent from the Java source code, but which JVM does not provide, could only be implemented via non-portable means. For example, if one wishes to write a portable parallel application under multi-threaded, shared memory model, then distributed shared memory (DSM) would be required for execution under MPP and cluster platforms. However, JVM itself does not facilitate any DSM functionalities, nor provide software 'hooks' for incorporating the necessary read/write barriers for DSM implementation. As a result, one must modify the JVM, or employ some ad-hoc preprocessor solution, neither of which are satisfactory in terms of portability and/or performance. With OpenJIT, the DSM class library implementor can write a set of compiler metaclasses so that necessary read/write barriers, etc., would be appropriately inserted into critical parts of code.

Also, with OpenJIT, one could incorporate platform-, application-, or usage-specific compilation or optimization. For example, one could perform various numerical optimizations such as loop restructuring, cache blocking, etc. which have been well-studied in Fortran and C, but have not been well adopted into JITs for excessive runtime compilation cost. OpenJIT allows application of such optimizations to critical parts of code in a pinpointed fashion, specified by either

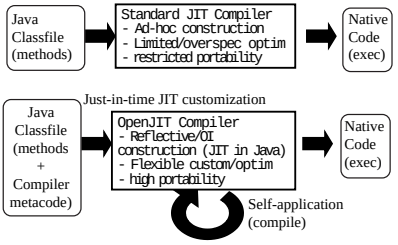

Fig. 1. Comparison of Traditional JITs and OpenJIT

the class-library builder, application writer, or the user of the program. Furthermore, it allows optimizations that are too application and/or domain specific to be incorporated as a general optimization technique for standard compilers, as has been reported by [16].

In this manner, OpenJIT allows a new style of programming for optimizations, portability, and maintainability, compared to traditional JIT compilers, by providing separations of concerns with respect to optimization and code-generation for new features. With traditional JIT compilers, we see in the upper half of Figure 1, the JIT compilers would largely be transparent from the user, and users would have to maintain code which might be too tangled to achieve portability and performance. OpenJIT, on the other hand, will allow the users to write clean code describing the base algorithm and features, and by selecting the appropriate compiler metaclasses, one could achieve optimization while maintaining appropriate separation of concerns. Furthermore, compared to previous open compiler efforts, OpenJIT could achieve better portability and performance, as source code is not necessary, and late binding at run-time allows exploitation of run-time values, as is with run-time code generators.

2.2 Architectural Overview of OpenJIT

The OpenJIT architecture is largely divided into the frontend and the backend processors. The frontend takes the Java bytecodes as input, performs higher-level optimizations involving source-to-source transformations, and passes on the intermediate code to the backend, or outputs the transformed bytecode. The backend is effectively a small JIT compiler in itself, and takes either the bytecode or the intermediate code from the frontend as input, performs lower-level optimizations, and outputs the native code for direct execution. The reason there is a separate frontend and the backend is largely due to modularity and ease of development, especially for higher-level transformations, as well as defaulting to the backend when execution speed is not of premium concern. In particular, we strive for the possibility of the two modules being able to run as independent components.

Upon invocation, the *OpenJIT frontend* system processes the bytecode of the method in the following way: The *decompiler* recovers the AST of the original Java source from the bytecode, by recreating the control-flow graph of the source

program. At the same time, the *annotation analysis module* will obtain annotating info on the class file, which will be recorded as attribute info on the AST[2]. Next, the obtained AST will be subject to optimization by the *(higher-level) optimization module*. Based on the AST and control-flow information, we compute the data & control dependency graphs, etc., and perform program transformation in a standard way with modules such as *flowgraph construction module*, *program analysis module*, and *program transformation module*. The result from the OpenJIT frontend will be a new bytecode stream, which would be output to a file for later usage, or an intermediate representation to be used directly by the OpenJIT backend.

The *OpenJIT backend* system, in turn, performs lower-level optimization over the output from the frontend system, or the bytecodes directly, and generates native code. It is in essence a small JIT compiler in itself. Firstly, when invoked as an independent JIT compiler bypassing the frontend, the *low-level IL translator* analyzes and translates the bytecode instruction streams to low-level intermediate code representation using stacks. Otherwise the IL from the frontend is utilized. Then, the *RTL Translator* translates the stack-based code to intermediate code using registers (RTL). Here, the bytecode is analyzed to divide the instruction stream into basic blocks, and by calculating the depth of the stack for each bytecode instruction, the operands are generated with assumption that we have infinite number of registers. Then, the *peephole optimizer* would eliminate redundant instructions from the RTL instruction stream, and finally, the *native code generator* would generate the target code of the CPU, allocating physical registers. Currently, OpenJIT supports the SPARC and the x86 processors as the target, but could be easily ported to other machines. The generated native code will be then invoked by the Java VM, as described earlier.

3 Overview of the OpenJIT Frontend System

As described in Section 2, the OpenJIT frontend system provides a Java class framework for higher-level, abstract analysis, transformation, and specialization of Java programs which had already been compiled by `javac`: (1) The decompiler translates the bytecode into augmented AST, (2) analysis, optimizations, and specialization are performed on the tree, and (3) the AST is converted into the low-level IL of the backend system, or optionally, a stream of bytecodes is generated.

Transformation over AST is done in a similar manner to Stanford SUIF, in that there is a method which traverses the tree and performs update on a node or a subtree when necessary. There are a set of abstract methods that are invoked as a hook. The OpenJIT frontend system, in order to utilize such a hook functionality according to user requirements, extends the class file (albeit in a conformable way so that it is compatible with other Java platforms) by adding annotation info to the classfile. Such an info is called "classfile annotation".

[2] In the current implementation, the existence of annotation is a prerequisite for frontend processing; otherwise, the frontend is bypassed, and the backend is invoked immediately.

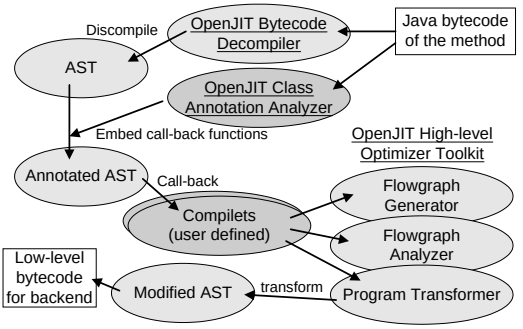

Fig. 2. Overview of OpenJIT Frontend System

The overall architecture of the OpenJIT frontend system is as illustrated in Fig. 2, and consists of the following four modules:

1. *OpenJIT Bytecode Decompiler*
   Translates the bytecode stream into augmented AST. It utilizes a new algorithm for systematic AST reconstruction using dominator trees.
2. *OpenJIT Class Annotation Analyzer*
   Extracts classfile annotation information, and adds the annotation info onto the AST.
3. *OpenJIT High-level Optimizer Toolkit*
   The toolkit to construct "compilets", which are modules to specialize the OpenJIT frontend for performing customized compilation and optimizations.
4. *Abstract Syntax Tree Package*
   Provides construction of the AST as well as rewrite utilities.

For brevity, we omit the details of the frontend system. Interested readers are referred to [20].

4 OpenJIT—Backend and Its Technical Issues

4.1 Overview of the OpenJIT Backend System

As a JIT compiler, the high-level overview of the workings of OpenJIT backend is standard. The heart of the low-level IL translator is the `parseBytecode()` method of the `ParseBytecode` class, which parses the bytecode and produces an IL stream. The IL we defined is basically an RISC-based, 3-operand instruction set, but is tailored for high affinity with direct translation of Java instructions into IL instruction set with stack manipulations for later optimizations. There are 36 IL instructions, to which each bytecode is translated into possibly a sequence of these instructions. Some complex instructions are translated into calls into run-time routines. We note that the IL translator is only executed when the OpenJIT backend is used in a standalone fashion; when used in conjunction with the frontend, the frontend directly emits IL code of the backend.

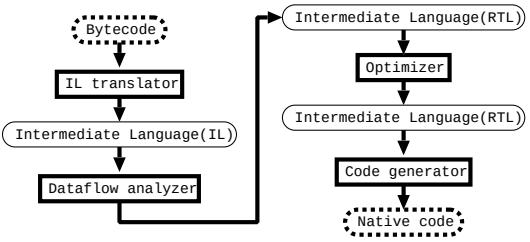

Fig. 3. Overview of the OpenJIT Backend System

Then, RTL converter translates the stack-based IL code to register based RTL code. The same IL is used, but the code is restructured to be register-based rather than encoded stack operations. Here, a dataflow analyzer is then run to determine the type and the offset of the stack operands. We assume that there are infinite number of registers in this process. In practice, we have found that 24–32 registers are sufficient for executing large Java code without spills when no aggressive optimizations are performed[24]. Then, the *peephole optimizer* would eliminate redundant instructions from the RTL instruction stream.

Finally, the *native code generator* would generate the target code of the CPU. It first converts IL restricting the number of registers, inserting appropriate spill code. Then the IL sequence is translated into native code sequence, and ISA-specific peephole optimizations are performed. Currently, OpenJIT supports the SPARC and x86 processors as the target, but could be easily ported to other machines[3]. The generated native code will be then invoked by the Java VM, upon which the *OpenJIT runtime module* will be called in a supplemental way, mostly to handle Java-level exceptions.

The architectural outline of the OpenJIT backend is illustrated in Figure 3. Further details of the backend system can be found in [23].

4.2 Technical Challenges in a Reflective Java JIT Compiler

As most of OpenJIT is written in Java, the bytecode of OpenJIT will be initially interpreted by the JVM, and gradually become compiled for faster, optimized execution. Although this allows the JIT compiler itself to adapt to the particular execution environment the JIT optimizes for, it could possibly give rise to the following set of problems:

1. *Invoking the Java-based JIT compiler from within the JVM*
   As the JIT compiler is invoked in the midst of a call chain of the base Java program. There must be a smooth way to massaging the JVM into invoking a JIT compiler in Java in a separate context.

[3] Our experience has been that it has not been too difficult to port from SPARC to x86, save for its slight peculiarities and small number of registers, due in part being able to program in Java. We expect that porting amongst RISC processors to be quite easy.

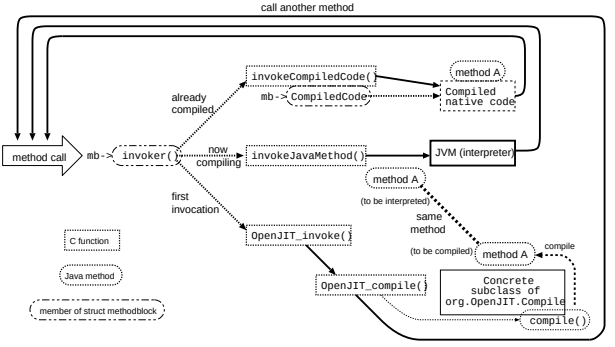

Fig. 4. Invoking OpenJIT

2. *Recursive Compilation*
   The current OpenJIT is designed to be *entirely* bootstrapped in "cold" mode, i.e., no parts of the JIT compiler are precompiled. Thus, as is with any reflective system, there must be some mechanism to stop the infinite recursive process, and "bottom out". This is a little more subtle than conventional compiler bootstrapping, as compilation occurs at runtime coexisting with compilation of applications; furthermore, the mechanism must be safe w.r.t. Java multi-threading, i.e., no deadlocks should occur.
3. *Speed and Memory Efficiency of the JIT compiler*
   A JIT compiler is beneficial only if the combined (compilation time + execution time) is smaller than the interpretation time under JVM. In more practical terms, OpenJIT must compete with traditional C-based JIT compilers for performance. Here, because of the interpretation and possible slowness of JIT execution even if itself were JIT compiled due to quality of generated native code, it is not clear if such goals could be satisfied. Moreover, memory efficiency is of primary concern, especially for embedded systems. In this regard, there is a particular issue not present in C-based JIT compilers.
4. *Lack of appropriate API for Java-written JIT compilers in standard JVM*
   A JIT compiler must be able to introspect and modify various data structures within the JVM. Unfortunately, JVM does not have any APIs for that purpose, primarily because it is likely that JIT compilers were assumed to be written with a low-level language such as C. For this purpose, there must be appropriate Java-level APIs which must be reasonably portable for JVM introspection in OpenJIT.

Again, for brevity, we only cover the most salient technical features here: for complete technical details readers are referred to [25].

Invoking the Java-based JIT compiler from within the JVM. In a "classic" JVM, for each method, both JIT compilation and transfer of control to the native method happens at the point of the subject method invocation. The JVM interpreter loop is structured as follows. When a method is invoked, the

invoker function of the methodblock structure (a structure internal to the JVM which embodies various info pertaining to a particular method) `mb` is called. Under interpretive execution, this in turn calls the JVM to generate a new Java stack frame. The first argument of `invoker()` function `o` is the class object for static method calls, and the invoked object on normal method calls. The second argument `mb` is a pointer to the methodblock structure, etc.

```
while(1) {
    get opcode from pc
    switch(opcode) {
    ...(various implementation of the JVM bytecodes)
    callmethod:
        mb->invoker(o, mb, args_size, ee);
        frame = ee->current_frame; /* setup java frame */
        pc = frame->lastpc; /* setup pc*/
        break;
    }
}
```

As showed in Figure 4, we substitute the value of the invoker in methodblock structure of every method to `OpenJIT_invoke` when a class is loaded. The `OpenJIT_invoke` function is defined as follows in C:

```
bool_t OpenJIT_invoke(JHandle *o, struct methodblock *mb,
                      int args_size, ExecEnv *ee)
```

This function in turns calls the `OpenJIT_compile()` in the C runtime to dynamically compile the method. Thereafter, the control is transferred to `mb->invoker`, transferring control to the just compiled method. The called `OpenJIT_compile()` performs the following functions:

1. *Mutual exclusion to prevent simultaneous compilation of the same method* — We must prevent multiple threads from compiling the same method at the same time with proper mutual execution using a compile lock. We reserve a bit in the methodblock structure as a lock bit.
2. *Setup of invoker and CompiledCode fields in the methodblock structure* — When a method is invoked, and subject to compilation, we reset the invoker and other fields in the methodblock so that any subsequent invocation of the method will have the method run by the interpreter during compilation. This allows natural handling of recursive self-compilation of OpenJIT compiler classes.
3. *Invocation of the body of the JIT compiler* — The Java method to invoke the compiler is then upcalled. An instance of a new JIT compiler in Java (to be more specific, its upcall entry class) is allocated and initialized for each JIT compiler invocation. Then, the `compile()` method of the instantiated entry class is up with `do_execute_java_method_vararg()`. Note that the current call context is preserved in the stack; that is to say, the same thread is utilized to make the upcall.

4. *Postprocessing of JIT compilation* — After compilation, control returns to the C runtime. At this point, most of the compiler becomes garbage, except for the persistent information that must be maintained across method compilations. This is to facilitate dynamic change in the compiler with compilets, and also to preserve space, directly exploiting the memory management feature of Java. If the compilation is successful, we set the invoker field of the methodblock structure to the compiled native code. When compilation fails: The methodblock field values are restored to their original values. [4]

In this manner, the JIT compiler in Java is smoothly invoked on the same execution thread. In practice it is much more complicated, however, due to possibility of exceptions, JIT compilation occurring even on calls from native methods, advanced features such as backpatching, inlining, and adaptive compilation. Some of the issues are further discussed below, while for the rest refer to [25].

Recursive Compilation. Recursive compilation is handled at the C runtime level of OpenJIT with simple locking mechanism, as we see in the following simplified code fragment (in practice, it would include more code such as support for adaptive compilation):

```
COMPILE_LOCK(ee);
if (COMPILE_ON_THE_WAY(mb)) {
    /* now compiling this method. avoid from double compiling */
    COMPILE_UNLOCK(ee);
    return;
}
START_COMPILE(mb);
/* reset invoker temporarily */
mb->invoker = (mb->fb.access & ACC_SYNCHRONIZED) ?
    invokeSynchronizedJavaMethod : invokeJavaMethod;
/* reset dispatcher temporarily */
mb->CompiledCode = (void *)dispatchJVM;
COMPILE_UNLOCK(ee);
```

This is essentially where the compilation "bottoms out"; once the method starts to be compiled, a lock is set, and further execution of the method will be interpreted. In fact, in Java we actually obtain this behavior *for free*, as mutual exclusion of multi-threaded compilation has to be dealt with in any case, defaulting to interpretation.

However, in the case of recursive compilation, there are some issues which do not exist for C-based JIT compilers:

[4] In practice, the invoker field is not directly substituted for the compiled native method, but rather we invoke a native code stub, depending on the type of the return argument. This is done to handle exceptions, java reflection, calls between native and interpreted code, etc.

– *Possibility of Deadlocks* — We must be assured that, as long as JIT compiler obeys the locking protocol, recursive multi-threaded compilation does not cause any deadlocking situations. This is proven by showing that cyclic resource dependencies will not exist between the multi-threaded compilations. Let the dependencies between the methods be denoted $m_1^c \longrightarrow m_2^c$, where for execution of compiled method m_1^c we need to execute a compiled method m_2^c. We further distinguish compiled and interpreted execution of methods with m^c and m^i, respectively. Then, starting from the entry method as a node, graph of dependency relations will clearly form a tree for single-threaded case. For multi-threaded case, however, it must be shown that arbitrary interleavings of the tree via possible self compilation will only create DAGs. Informally this simply holds because all m_i's will not be dependent on any other nodes, and thus the cycle will have to be formed amongst m_c's, which is not possible.

  We also note that, in practice, deadlocks could and does occur not only between the JIT compiler and the JVM. One nasty bug which took a month to discover was in fact such a deadlock bug. As it turns out, the "classic" JVM locks the constant pool for a class when its finalizer is run. This could happen just when OpenJIT tries to compile the finalizer method, resulting in a deadlock.
– *Speed and Memory Performance Problems* — Aside from the JIT compiler merely working, we must show that the JIT compiler in Java could be time and memory efficient. The issue could be subdivided into cases where the OpenJIT is compiling (1) application methods, and (2) OpenJIT methods. The former is simply shown by extensive analysis of standard benchmarks in Section 5, where it is shown that OpenJIT achieves good time and memory performance and despite being constrained by the limitations of the "classic" VM, such as handle-based memory systems implementation, non-strict and non-compacting GC, slow monitor locking, etc. The latter is much more subtle: because of recursive compilation, two undesirable phenomena occur. (A) compilation of a single application bytecode will set off a chain of recursive compilations, due to the dependency just discussed. This has the effect of accumulating compiling contexts of almost the entire OpenJIT system, putting excessive pressure on the memory system. (B) We could prevent the situation by employing adaptive compilation and defaulting back to interpretation earlier, but this will have the effect of slowing down the bootstrap time, as long as possibly having some residual effect on application compilation due to some OpenJIT compiler methods still being interpreted. (A) and (B) are strongly interrelated; in the worst case, we will be trading speed, especially the bootup time, for space. On the other hand, one could argue that little penalty is incurred by adaptive means, not because of the typical execution frequency argument, but rather, that because of recursive compilation, much of the OpenJIT system could be compiled under interpretation in the first place. We perform extensive performance analysis to investigate this issue in Section 5.

Lack of appropriate API for Java-written JIT compilers in standard JVM. None of the current Java VMs, including the "classic" VM for which OpenJIT is implemented, have sufficient APIs for implementing a JIT compiler in Java. In particular, JVM basically only provides APIs to *invoke* a C-based JIT compiler, but does not provide sufficient APIs for generalized introspection or intersession features. Note that we cannot employ the Java reflection API either, for it abstracts out the information required by the JIT compiler.

Instead, we define a set of native methods as a part of the OpenJIT runtime. The `Compile` class declares the following native methods, which are defined in `api.c` of the distribution. There are 17 methods in all, which can be categorized as follows:

– *Constant pool introspection methods*

```
public final native int ConstantPoolValue(int index)
private final native int ConstantPoolTypeTable(int index)
public final int ConstantPoolType(int index)
public final boolean ConstantPoolTypeResolved(int index)
public final String ConstantPoolClass(int index)
private native byte[] ConstantPoolClass0(int index)
public final String ConstantPoolName(int index)
private native byte[] ConstantPoolName0(int index)
public final native int ConstantPoolAccess(int index)
public final native byte[] ConstantPoolMethodDescriptor(int index)
public final native int ConstantPoolFieldOffset(int index)
public final native int ConstantPoolFieldAddress(int index)
```

– *Native method allocation and reflection*

```
public final native void NativeCodeAlloc(int size)
public final native int NativeCodeReAlloc(int size)
public final native void setNativeCode(int pc, int code)
public final native int getNativeCode(int pc)
private native byte[] MethodName()
```

– *Class resolution methods (used for inlining)*

```
public final native void initParser(int caller_cp, int index)
public final native void resolveClass(int caller_cp, int index)
```

As one can see, majority of the methods are such that either introspective or intercessive operations being performed on the JVM.

The current API is sufficient, but admittedly too low level of abstraction, in that it exposes too much of the underlying VM design; indeed, our goal is to allow JITs to be a customizable and portable hook to the Java system, and thus, have OpenJIT be portable across different kinds of VMs. For this purpose, in the next version of OpenJIT, we plan to design a substantially higher-level API, abstracting out the requirements of the different VMs. The implementation of the API for "classic" VM will sit on the current APIs, but other VMs will have different implementations of native methods.

Another issue is the safety of the API. In the current implementation, the OpenJIT native method APIs are accessible to all the classes, including the application classes. It is easy to restrict the access to just the compiler classes (those with path `org.OpenJIT.`), but this will preclude user-defined compilets. Some form of security/safety measures with scope control, such as restricting access only to signed classfiles, might be necessary. We are currently investigating this possibility to utilize the security API in JDK 1.2.

5 Performance Analysis of OpenJIT

We now analyze the behavior of OpenJIT with detailed benchmarks. As mentioned earlier, our concern is both execution speed and memory usage. The former is obvious, as the execution overhead of the JIT compiler itself as well as quality of generated code will have to match that of conventional JIT compilers. Memory usage is also important, especially in areas such as embedded computing, one of major Java targets.

All the OpenJIT objects, except for the small C runtime system, coexists in the heap with the target application. The necessary working space includes that of various intermediate structures of compiler metaobjects that the OpenJIT builds, including various flowgraphs, intermediate code, etc., and persistent data, such as the resulting native code. Standard C-based JIT compilers will have to allocate such structures outside the Java heap; thus, memory usage is fragmented, and efficient memory management of the underlying JVM is not utilized. For OpenJIT, since both the application and compiler metaobjects will coexist in the heap, it might seem that we would obtain the most efficient usage of heap space.

On the other hand, the use of Java objects, along with automated garbage collection, could be less memory efficient than C-based JITs. Moreover as mentioned in Section 4, there could be a chain of recursive compilations which will accumulate multiple compilation contexts, using up memory. It is not clear what kind of adaptive compilation techniques could be effective in decreasing the accumulation, while not resulting in substantial execution penalty.

5.1 Benchmarking Environment

As an Evaluation Environment, we employed the following platform, and pitted OpenJIT against Sun's original JIT compiler (sunwjit) on JDK 1.2.2 (ClassicVM).

- Sun Ultra60 (UltraSparc II 300MHz×2, 256MB)
- Solaris 2.6-J
- JDK 1.2.2 (ClassicVM)

We took six programs from the SPECjvm98 benchmark, as well as the simple "Hello World" benchmark. The six—_201_compress (file compression), _202_jess (expert system), _209_db (DBMS simulator), _213_javac (JDK 1.0.2 compiler),

Table 1. Code size of OpenJIT and C runtimes

	classes (files)	methods	# lines	classfile (stripped binary) bytes
Frontend	243	1,439	24,148	629,062
Backend (sparc)	23	182	7,560	118,592
Backend (x86)	21	182	8,085	118,125
C runtime(sparc)	3		3,565	42,556
C runtime(x86)	3		3,752	28,928
sunwjit (sparc)				234,112
sunwjit (x86)				146,508

and _227_mtrt (multi-threaded raytracer), and _228_jack (parser generator)—have been chosen as they are relatively compute intensive, do not involve mere simple method call loops, and not reliant on runtime native calls such as networks, graphics, etc. "Hello World" benchmark superficially only makes a call to System.out.println(), but actually it will have executed almost the entire OpenJIT system, the Java packages that OpenJIT employs, as well as the constructors of system classes. This allows us to observe the bootstrap overhead of the OpenJIT system.

In order to obtain the precise profile information for memory allocation, we employed the JVMPI (Java Virtual Machine Profiler Interface) of JDK 1.2.2. Additionally, we extended OpenJIT to output its own profile information. This is because it is difficult to determine with JVMPI whether the allocated compiler metaobject is being used to compile application methods, or used for recursive compilation, because JVMPI merely reports both to be of the same class (say, merely as instances of `ILnode`, etc.). By combining JVMPI and OpenJIT profile information, we obtain precise information of how much space the live OpenJIT compiler metaobjects occupy, how much native code is being generated, how much of the native code is that of OpenJIT, along the execution timeline. Also, how much classfiles are being loaded, how many methods are being compiled, and what is the percentage of the OpenJIT classes, can be profiled as well.

Such profiling is done in real-time, in contrast to the simulation based profiling of SpecJVM memory behavior in [6]. Such an approach is difficult to apply for our purpose, as JIT compilation is being directly involved, resulting in code not directly profilable with JVM simulation. Our compiler-assisted profiling allows us to obtain almost as precise an information as that of [6] at a fraction of time. Nevertheless, the profile information generated is quite large, reaching several hundred megabytes for each SpecJVM run.

5.2 Benchmarking Contents

The Size of OpenJIT. We first show the code size of OpenJIT compared to sunwjit. As we can see, the frontend is approximately 3 times the size of backend in terms of number of lines, and factor of approximately 8–10 larger in terms of number of classes and methods. This is because the frontend contains numerous small classes representing syntactic entities of Java, whereas the backend has much larger method size, and the backend IL does not assign a class for each instruction. We also see that the combined size of OpenJIT backend and C runtime is smaller than sunwjit, but when it is self-compiled, the x86 version

Table 2. Baseline Performance

Program	JIT	class#	alloc obj# (openjit)	allocsize[MB] (openjit)	GC#	time
Hello	interpreter	167	4,890(—)	0.273(—)	0	0.380
	sunwjit	172	5,244(—)	0.285(—)	0	0.450
	openjit	185	90,600(74,831)	2.906(2.316)	5	1.270
	openjit-int	185	37,059(31,149)	1.265(0.941)	2	1.280
_201_ compress	interpreter	224	15,547(—)	110.640(—)	20	673.910
	sunwjit	226	9,399(—)	110.266(—)	16	89.620
	openjit	241	136,328(107,197)	114.330(3.318)	21	72.530
	openjit-int	241	81,910(62,742)	112.662(1.918)	18	74.460
_202_ jess	interpreter	373	7,951,562(—)	221.919(—)	547	148.550
	sunwjit	375	7,936,214(—)	221.190(—)	565	65.750
	openjit	390	8,103,973(142,626)	226.383(4.402)	528	62.530
	openjit-int	390	8,049,403(98,045)	224.710(2.998)	532	62.160
_209_ db	interpreter	218	3,218,293(—)	63.249(—)	33	307.480
	sunwjit	220	3,213,851(—)	63.095(—)	32	142.160
	openjit	235	3,343,820(109,778)	67.104(3.398)	39	172.830
	openjit-int	235	3,289,249(65,197)	65.431(1.994)	37	182.080
_213_ javac	interpreter	386	5,972,713(—)	147.288(—)	80	200.940
	sunwjit	388	5,936,663(—)	145.458(—)	69	94.850
	openjit	403	6,181,295(208,562)	154.486(6.478)	77	102.960
	openjit-int	403	6,126,571(164,145)	151.531(5.080)	67	108.850
_227_ mtrt	interpreter	239	6,382,222(—)	84.118(—)	90	173.510
	sunwjit	241	6,376,266(—)	83.902(—)	90	59.430
	openjit	256	6,524,115(124,549)	88.467(3.855)	96	56.640
	openjit-int	256	6,469,545(79,968)	86.794(2.451)	93	56.980
_228_ jack	interpreter	270	6,878,777(—)	150.755(—)	451	196.330
	sunwjit	272	6,868,951(—)	150.353(—)	465	66.669
	openjit	287	7,046,695(152,625)	155.818(4.707)	286	66.970
	openjit-int	287	6,992,109(108,001)	154.144(3.302)	276	68.010

could get larger. Thus, this raises an interesting issue of what happens if we run the compiler always interpreted in embedded situations; in the subsequent benchmark, we will also investigate this possibility.

Baseline Performance. We next observe the baseline execution time and memory usage characteristics of OpenJIT. We set the heap limit to 32MBytes (as mandated by the SpecJVM98 benchmarks) comparing the execution of JVM interpreter, sunwjit, OpenJIT with self compilation, and OpenJIT without self compilation. Table 2 shows for each execution, how many classes are loaded and their sizes, how many objects are allocated (parenthesis indicates how many OpenJIT compiler metaobjects), how much memory size are allocated (and that of OpenJIT compiler metaobjects), wallclock execution time, and number of GCs.

Figure 5 additionally show consumed overall heap space, live OpenJIT object heap space, along the time axis. This shows the process of compiler bootstrapping. The compilation in OpenJIT was set to be most aggressive i.e., all the methods are JIT compiled on their first invocations, and the entire frontend had been turned off, and are not loaded.

The Hello benchmark exemplifies the overhead of bootstrapping openjit and openjit-int; compared to sunwjit, we see approximately 2.8 times increase in startup time, indicating that compilation with OpenJIT incurs approximately $\times 3$ overhead over sunwjit. On the other hand, difference between openjit and

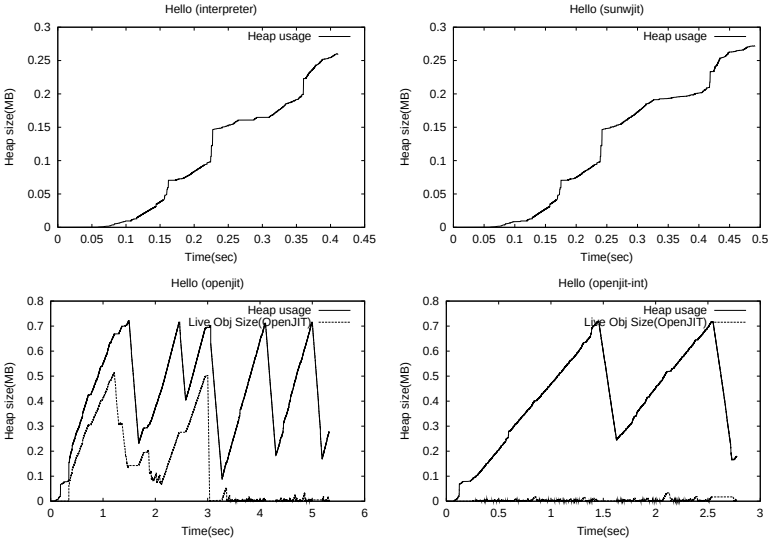

Fig. 5. Timeline behavior of heap usage and live object allocated by OpenJIT (interpreter, sunwjit, openjit, openjit-int). The measured time of this figure contains the overhead of profiling, so it did not exactly match the results of Table 2.

openjit-int is negligibly small; this indicates that overhead of self compilation is almost negligible, but rather, the overhead of system and library classes are substantial (we observe approximately 457 methods compiled, as opposed to 128 methods for OpenJIT).

For the six SPECjvm98 benchmarks, we see that the overhead is well amortized, and OpenJIT is competitive with sunwjit, sometimes superior. The running time of programs range between 56–172 seconds, so the overhead of JIT compilation is well amortized, even for openjit-int, given the relative expense of OpenJIT compilation over sunwjit. Moreover, since method-specific openjit compiler metaobjects are mostly thrown away on each compilation, in principle we do not occupy memory compared to sunwjit (Fig. 5) In fact, we may be utilizing memory better due to sharing of the heap space with the application. The runtime comparison of execution times of each program depends on each program. For compress, openjit was 20% superior, whereas sunwjit was faster by about 18%. Other benchmarks are quite similar in performance. Even small but unnegligible difference in compilation overhead, OpenJIT is likely producing slightly superior code on average.

We do observe some anomalies, however. Firstly, for most cases OpenJIT had increased invocations of GCs due to heap coexistence; but for jack and jess, OpenJIT had less GC invocations, by approx. 40% and 5%, respectively. This is attributable to unpredictable behavior of conservative GC in the "classic" JVM; it is likely that by chance, the collector happened to mistake scalars for pointers on the stack. Neither really contributes significantly to performance differences. Another anomaly is that, in many cases openjit-int was faster than

openjit with self-compilation. This somewhat contradicts our observation that compilation DOES incur some overhead, as difference between interpreted and compiled executions of OpenJIT itself should manifest, but doesn't.

Figure 5 shows the timeline track of the amount of heap usage by the entire program, OpenJIT (openjit) and interpreted OpenJIT (openjit-int), respectively, for the Hello benchmark. Again, we observe that during bootstrapping, openjit and openjit-int require approximately 700Kbytes of heap space, which is about 2.6 times the heap space as sunwjit and pure interpreter. Since openjit-int does not allocate metaobjects to compile itself, and the amount being consumed to compile methods of other classes are small, we attribute the consumption to the system objects with the libraries being called from OpenJIT, and immediately released.

The Hello benchmark also verifies that there are two phases of execution for OpenJIT. Firstly, there is a bootstrap phase where the entire OpenJIT is aggressively compiled, accumulating multiple compilation contexts in the call

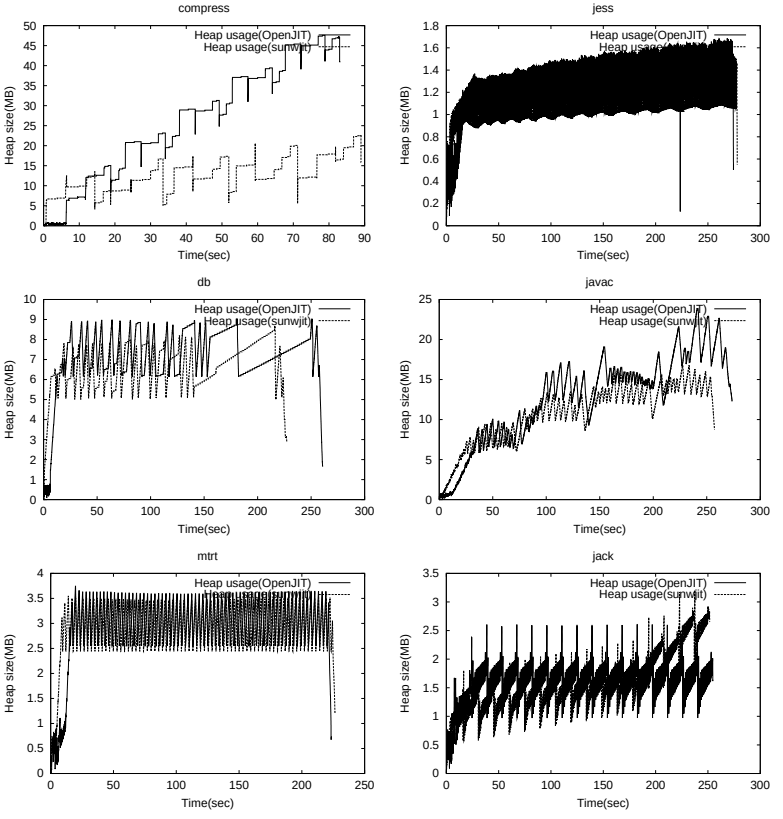

Fig. 6. Timeline behavior of heap usage with SPECjvm98 (sunwjit vs. openjit). The measured time of this figure contains the overhead of profiling, so it did not exactly match the results of Table 2.

Table 3. SPECjvm98 results on Linux (x86)

Benchmark	OpenJIT	interpreter	sunwjit	IBM1.1.8
_200_check	0.043	0.082	0.106	0.042
_201_compress	40.828	316.687	59.338	18.236
_202_jess	29.783	90.434	48.493	13.142
_209_db	85.881	203.289	119.228	40.259
_213_javac	56.227	120.199	70.698	30.964
_222_mpegaudio	40.911	263.870	41.705	11.942
_227_mtrt	30.101	96.156	37.337	17.014
_228_jack	28.403	107.049	49.176	10.751

chain of the JIT compiler. Thus, the space required is proportional to the critical path in the call chain. Then, it quickly falls off, and transcends into a stable phase where most parts of OpenJIT have been compiled, and only application methods are being compiled and executed.

No matter how the memory is being used, the amount of additional heap space required for recursively compiling OpenJIT will not be a problem for modern desktop environments, in some situations. A typical desktop applications consumes orders of magnitude more space: for example, our measurements in Figure 6 for `javac` shows it consumes more than 20 Mbytes[5]. However, for embedded applications, such an overhead might be prohibitive. As discussed earlier, this could be suppressed using less aggressive, adaptive compilation similar to the Self compiler[12], but it is not clear what strategy will achieve good suppression while not sacrificing performance. In the next section we consider several adaptive strategies for suppression.

We have also taken some benchmarks on the x86 version of OpenJIT, and compared it against IBM's JDK 1.1.8 JIT compiler, which is reputed to be the fastest JIT compiler for x86, in neck to neck with Sun's Hotspot. Table 3 shows the results: we see that, for most benchmarks, OpenJIT x86 is superior to sunwjit, and runs about half the speed of IBMs JIT, despite being constrained by the "classic" JVM.

5.3 Adaptive OpenJIT Compilation Strategies

There are several criteria in the design space for adaptive compilation in OpenJIT for memory suppression of the bootstrapping phase.

1. *Alteration of JIT compilation frequency* — The most aggressive strategy will compile each method on its first invocation. We reduce the frequency of compilation using the following strategies, with p as a parameter ($p = 2, 4, 8, 16$)
   - JIT compile on pth invocation, deferring to interpretation for the first $p - 1$ invocations (*constant delay*).
   - Assign each method a random number between [0, $p - 1$], and compile when the number of invocation reaches that number (*random delay*).

[5] [6] reports that with exact GC, the actual usage is approximately 6MBytes. The difference is likely to be an artifact of conservative GC, our close examination has shown.

   - Compile with probability $1/p$ on each invocation (*probabilistic*). reduced probability increased the execution time by
2. *Restriction of methods subject to adaptation* — We could delay compilation for all the methods, or alternatively, only those of the OpenJIT compiler metaclasses. The former obviously will likely consume less space, but the former may be sufficient and/or desirable, as it will not slow down the application itself. We verify this by comparing altering compilation frequency changes to all classes, versus only altering the frequency of OpenJIT method. For the latter, all other methods are compiled on first invocation except for class initializers, which are interpreted.
3. *Restriction of number of simultaneous compilations* — We put global restriction on how many compilations can occur simultaneously. This can be done safely without causing deadlocks. Attempt to compile exceeding this limit will default back to the interpreter. ($L = 1, 8$). Note that, although simultaneous compilation could occur for application methods under multithreading, this primarily restricts the simultaneous occurrence of deep recursive compilation chains on bootstrapping.
4. *Restricting compilation of* `org.OpenJIT.ParseBytecode.parseBytecode()` — This is a special case, as preliminary benchmarks indicated that `parseBytecode()`, is quite large for a single method, (1576 lines of source code, 6861 JVM bytecodes), and thus single compilation of this method creates a large structure in the heap space once it is subject to compilation, irrespective of the strategies used. In order to eliminate the effect, we test cases where compilation of `parseBytecode()` is restricted. In the next version of OpenJIT we plan to factor the method into smaller pieces.

According to the Hello benchmark, in when adaptaion is applied to all the methods, combinations of other schemes effectively yielded reduction in the number and size of objects that are allocated during bootstrapping, without significant increase in bootstrap time. On the other hand, restricting compilation of OpenJIT method only did not yield significant results, except for the case when the entire OpenJIT was interpreted, or when `parseBytecode()` was restricted, again, without significant loss of performance.

The table only shows the *total* memory allocated. In order to characterize the *peak* memory behavior, we present the timeline behavior in Figure 7. Here, for each scheme, the parameter with *lowest* peak is presented. We observe that, (1) probabilistically lowering the frequency helps reduce the peak usage, and (2) `parseBytecode()` dominates the peak. We are currently conducting futher analysis, but it is conclusive that naive frequency adjustment does not help to reduce the peak; rather, the best strategy seems to be to estimate the heap usage based on bytecode length, and supressing compilation once a prescribed limit is exceeded.

Table 4. Alteration of JIT compilation frequency for all methods

criteria	Hello					
	param	method# (openjit)	alloc obj# (openjit)	alloc size [MB] (openjit)	GC#	time
always	–	457(128)	90,594(74,831)	2.905(2.316)	5	1.270
constant delay	2	225(126)	64,827(52,194)	2.117(1.629)	3	1.190
	4	180(116)	57,568(46,133)	1.898(1.445)	3	1.280
	8	161(114)	55,743(44,470)	1.843(1.395)	4	1.250
	16	151(113)	54,069(43,064)	1.793(1.352)	4	1.810
random delay	2	412(127)	84,739(69,746)	2.729(2.162)	5	1.270
	4	301(123)	72,686(59,272)	2.357(1.843)	4	1.190
	8	231(120)	62,628(50,589)	2.050(1.578)	4	1.240
	16	196(115)	61,923(49,790)	2.031(1.558)	3	1.350
probability	2	170(115)	56,383(45,041)	1.862(1.412)	4	1.280
	4	190(115)	58,493(46,877)	1.924(1.466)	4	1.900
	8	149(115)	53,710(42,735)	1.780(1.341)	3	2.010
	16	112(99)	27,066(21,399)	0.955(0.664)	1	0.920

Table 5. Alteration of JIT compilation frequency for OpenJIT methods only

criteria	Hello					
	param	method# (openjit)	alloc obj# (openjit)	alloc size [MB] (openjit)	GC#	time
always	–	457(128)	90,594(74,831)	2.905(2.316)	5	1.270
constant delay	2	457(128)	90,526(74,757)	2.904(2.314)	5	1.240
	4	451(122)	89,616(73,988)	2.877(2.291)	5	1.260
	8	449(120)	88,535(73,179)	2.845(2.265)	5	1.280
	16	446(117)	85,699(71,112)	2.760(2.200)	5	1.800
random delay	2	457(128)	90,534(74,765)	2.904(2.314)	5	1.320
	4	455(126)	90,236(74,491)	2.895(2.306)	5	1.220
	8	452(123)	89,780(74,115)	2.882(2.295)	5	1.230
	16	450(121)	89,299(73,763)	2.868(2.284)	5	1.330
probability	2	450(121)	89,490(73,884)	2.873(2.288)	5	1.270
	4	429(116)	83,831(69,523)	2.703(2.152)	4	1.980
	8	446(117)	85,722(71,136)	2.760(2.201)	5	2.020
	16	441(112)	84,637(70,295)	2.728(2.117)	4	0.910
limit simultaneity	1	457(128)	90,590(74,821)	2.906(2.316)	4	2.530
	8	457(128)	90,514(74,751)	2.903(2.314)	4	1.410
no parseBytecode	–	456(127)	69,463(58,430)	2.248(1.788)	3	1.190
openjit-int	–	329(0)	37,059(31,149)	1.265(0.941)	2	1.280

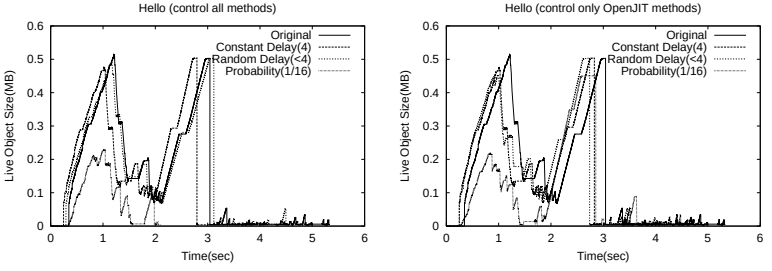

Fig. 7. Timeline behavior of heap usage for adaptive compilation (best cases).

6 Related Work

As mentioned earlier, most modern compilers and language systems are bootstrapped in a self-descriptive fashion, but they do not coexist at runtime. In fact, although Lisp and Smalltalk systems embodied their own compilers written in terms of itself and executable at run-time, they are typically source-to-bytecode compilers, and not bytecode-to-native code compilers, which JITs are. In fact, as far as we know, there have not been any reports of a JIT compiler for a particular language being reflective. Most JIT compilers we have investigated, including those for Lisp, Smalltalk, Java as well as experimental languages such as SELF, have been written in C/C++ or in assembly language.

More recent efforts in self-descriptive, practical object-oriented system is Squeak[14]. Squeak employs the Bluebook[9] self-definition of Smalltalk, then bootstraps it using C, then further optimizes the generated VM. Bootstrapping in Squeak involves the VM only, and not the JIT compiler. The recently-announced JIT Squeak compiler is written in C and basically only merges the code fragments corresponding to individual bytecode. Thus, this is not a true compiler in a sense, but rather a simple bytecode to binary translator. This was done to achieve very quick porting of Squeak to various platforms, and stems from some of the earlier work done in [21].

We know of only two other related efforts paralleling our research, namely MetaXa[10] and Jalapeño[1]. Metaxa is a comprehensive Java reflective system whereby many language features could be reified, including method invocations, variable access, and locking. MetaXa has built its own VM and a JIT compiler; as far as we have communicated with the MetaXa group, their JIT compiler is not full-fledged, and is specific to their own reflective JVM. Moreover, their JIT is reported not robust enough to compile itself.

Jalapeño[1] is a major IBM effort in implementing a self-descriptive Java system. In fact, Jalapeño is an aggressive effort in building not only the JIT compiler, but the entire JVM in Java. The fundamental difference stems from the fact that Jalapeño rests on *its own customized JVM with completely shared address space*, much the same way the C-based JIT compilers are with C-based JVMs. Thus, there is little notion of separation of the JIT compiler and the VM for achieving portability, and the required definition of clean APIs, which is mandated for OpenJIT. For example, the JIT compilers in Jalapeño can access the internal objects of the JVM freely, whereas this is not possible with OpenJIT. So, although OpenJIT did not face the challenges of JVM bootstrapping, this gave rise to investigation of an effective and efficient way of interfacing with a monolithic, existing JVMs, resulting in very different technical issues as have been described in Section 4.

The manner in which Jalapeño bootstraps is very similar to Squeak and other past systems. The way the type safety of Java is circumvented, however, is similar to the technique employed in OpenJIT: there is a class called `Magic`, which defines a set of native methods that implements operations where direct access to VM internals are required. In OpenJIT, the `Compile` class defines a set of APIs using a similar technique. Unfortunately, again there is no mention

of attempting to develop the API into a clean one for generalized purposes of self-descriptive JITs for Jalapeño.

There are other technical differences as well; OpenJIT is architected to be a compiler framework, supporting features such as decompilation, various frontend libraries, whereas it is not with Jalapeño. No performance benchmarks have been made public for Jalapeño, whereas we present detailed studies of execution performance validating the effectiveness of reflective JITs, in particular memory profiling technique which directly exploits the 'openness' of OpenJIT. Interestingly enough, Jalapeño is claimed to be only targeting server platforms, and not desktop nor embedded platforms. It would be quite interesting to investigate the memory performance of Jalapeño in the manner we have done, in particular to test whether it makes sense to target smaller platforms or not.

Still, the Jalapeño work is quite impressive, as it has a sophisticated three-level compiler system, and their integrated usage is definitely worth investigating. Moreover, there is a possibility of optimizing the the application together with the runtime system in the VM. This is akin to optimization of reflective systems using the First Futamura projection in object oriented languages, as has been demonstrated by one of the author's older work in [17] and also in [18], but could produce much more practical and interesting results. Such an optimization is more difficult with OpenJIT, although some parts of JVM could be supplanted with Java equivalents, resulting in a hybrid system.

There have been a number of work in practical reflective systems that target Java, such as OpenJava[27], Javassist[5], jContractor[15] , EPP[13], Kava[30], just to name a few. Welch and Stroud present a comprehensive survey of Java reflective systems, discussing differences and tradeoffs of where in the Java's execution process reflection should occur[30].

Although a number of work in the context of open compilers have stressed the possibility of optimization using reflection such as OpenC++[4], our work is the first to propose a system and a framework in the context of a dynamic (JIT) compiler, where run-time information could be exploited. A related work is Welsh's Jaguar system[31], where a JIT compiler is employed to optimize VIA-based communication at runtime in a parallel cluster.

From such a perspective, another related area is dynamic code generation and specialization such as [7, 11, 8]. Their intent is to mostly provide a form of run-time partial evaluation and code specialization based on runtime data and environment. They are typically not structured as a generalized compiler, but have specific libraries to manipulate source structure, and generate code in a "quick" fashion. In this sense they have high commonalities with the OpenJIT frontend system, sans decompilation and being able to handle generalized compilation. It is interesting to investigate whether specialization done with a full-fledged JIT compiler such as OpenJIT would be either be more or less beneficial compared to such specific systems. This not only includes execution times, but also ease of programming for customized compilation. Consel et. al. have investigated a hybrid compile-time and run-time specialization techniques with their Tempo/Harrisa system [29, 22], which are source-level Java specialization system written in C; techniques in their systems could be applicable for OpenJIT with some translator to add annotation info for predicated specializations.

7 Conclusion and Future Work

We have described our research and experience of designing and implementing OpenJIT, an open-ended reflective JIT compiler framework for Java. In particular, we proposed an architecture for a reflective JIT compiler framework on a monolithic VM, and identify the technical challenges as well as the techniques employed, including the minimal set of low-level APIs required that needed to be added to existing JVMs to implement a JIT compiler in Java, contrasting to similar work such as Jalapeno. We performed analysis of the performance characteristics of OpenJIT, both in terms of execution speed and memory consumption, using collaborative instrumentation technique between the JVM and OpenJIT, which allowed us to instrument the JIT performance in real-time, and showed that OpenJIT is quite competitive with existing, commercial JIT systems, and some drawbacks in memory consumption during the bootstrap process could be circumvented without performance loss. We demonstrate a small example of how reflective JITs could be useful class- or application specific customization and optimization by defining a compilet which allowed us to achieve 8-9% performance gain without changing the base-level code.

Numerous future work exists for OpenJIT. We are currently redesigning the backend so that it will be substantially extensible, and better performing. We are also investigating the port of OpenJIT to other systems, including more modern VMs such as Sun's research JVM (formerly EVM). In the due process we are investigating the high-level, generic API for portable interface to VMs. The frontend requires substantial work, including speeding up its various parts as well as adding higher-level programming interfaces. Dynamic loading of not only the compilets, but also the entire OpenJIT system, is also a major goal, for live update and live customization of the OpenJIT. We are also working on several projects using OpenJIT, including a portable DSM system[26], numerical optimizer, and a memory profiler whose early prototype we employed in this work. There are numerous other projects that other people have hinted; we hope to support those projects and keep the development going for the coming years, as open-ended JIT compilers have provided us with more challenges and applications than we had initially foreseen when we started this project two years ago.

References

[1] B. Alpern, D. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, and J. J. Barton. Implementing Jalapeno in Java. In *Proceedings of OOPSLA '99*, pages 314–324, November 1999.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proceedings of ICPP '99*, September 1999.

[3] M. Atkinson, L. Daynes, M. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4), December 1996.

[4] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA '95*, pages 285–299, 1995.

[5] S. Chiba. Javassist — A Reflection-based Programming Wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
[6] S. Dieckman and U. Holzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *Proceedings of ECOOP '99*, pages 92–115, 1999.
[7] D. R. Engler and T. A. Proebsting. vcode: a retargetable, extensible, very fast dynamic ocde generation system. In *Proceedings of PLDI '96*, 1996.
[8] N. Fujinami. Automatic and Efficient Run-Time Code Generation Using Object-Oriented Languages. In *Proceedings of ISCOPE '97*, December 1997.
[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.
[10] M. Golm. metaXa and the Futre of Reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
[11] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An Evaluation of Staged Run-time Optimization in DyC. In *Proceedings of PLDI '99*, 1999.
[12] U. Holzle. Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming. Technical Report STAN-CS-TR-94-1520, Stanford CSD, 1995.
[13] Y. Ichisugi and Y. Roudier. Extensible Java Preprocessor Kit and Tiny Data-Parallel Java. In *Proceedings of ISCOPE '97*, December 1997.
[14] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak - A Usable Small talk Written in Itself. In *Proceedings of OOPSLA '97*, pages 318–326, October 1997.
[15] M. Karaorman, U. Holzle, and J. Bruno. iContractor: A Reflective Java Library to Support Design by Contract. In *Proceedings of Reflection '99*, pages 175–196, July 1999.
[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP '97*, pages 220–242, 1997.
[17] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. In *Proceedings of OOPSLA '95*, pages 57–64, October 1995.
[18] H. Masuhara and A. Yonezawa. Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of ECOOP '98*, pages 418–439, July 1998.
[19] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, and K. Hotta. OpenJIT — A Reflective Java JIT Compiler. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
[20] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT Frontend System: an implementation of the reflective JIT compiler frontend. *LNCS 1826: Reflection and Software Engineering*, 2000 (to appear).
[21] I. Piumarta and F. Riccardi. Optimizing Direct-threaded Code by Selective Inlining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 291–300, June 1998.
[22] U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards Automatic Specialization of Java Programs. In *Proceedings of ECOOP '99*, June 1999.
[23] K. Shimura. OpenJIT Backend Compiler. http://www.openjit.org/docs/backend-compiler/openjit-shimura-doc-1.pdf, June 1998.
[24] K. Shimura and Y. Kimura. Experimental development of java jit compiler. In *IPSJ SIG Notes 96-ARC-120*, pages 37–42, October 1996.

[25] K. Shimura and S. Matsuoka. OpenJIT Backend Compiler (Runtime) Internal Specification version 1.1.7. http://www.openjit.org/docs/backend-internal/index.html, October 1999.

[26] Y. Sohda, H. Ogawa, and S. Matsuoka. OMPC++ — A Portable High-Performance Implementation of DSM using OpenC++ Reflection. In *Proceedings of Reflection '99*, pages 215–234, July 1999.

[27] M. Tatsubori and S. Chiba. Programming Support of Design Patterns with Compile-time Reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[28] Stanford University. SUIF Homepage. http://www-suif.stanford.edu/.

[29] E. N. Volanschi, C. Consel, and C. Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of OOPSLA '97*, pages 286–300, October 1997.

[30] I. Welch and R. Stroud. From Dalang to Kava - the Evolution of a Reflective Java Extention. In *Proceedings of Reflection '99*, pages 2–21, July 1999.

[31] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, December 1999. Special Issue on Java for High-Performance Applications.