A Framework for Efficient Reuse of Binary Code in Java

Pramod G. Joisha*
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208, USA.
email: pjoisha@ece.nwu.edu

Samuel P. Midkiff
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598 USA.
email: smidkiff@us.ibm.com

Mauricio J. Serrano*
Intel Microprocessor Research labs
Santa Clara, CA 95052 USA.
email: mauricio.j.serrano@intel.com

Manish Gupta
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598 USA.
email: mgupta@us.ibm.com

ABSTRACT

This paper presents a compilation framework that enables efficient sharing of executable code across distinct Java Virtual Machine (JVM) instances. High-performance JVMs rely on run-time compilation, since static compilation cannot handle many dynamic features of Java. These JVMs suffer from large memory footprints and high startup costs, which are serious problems for embedded devices (such as hand held personal digital assistants and cellular phones) and scalable servers. A recently proposed approach called quasi-static compilation overcomes these difficulties by reusing precompiled binary images after performing validation checks and stitching on them (i.e., adapting them to a new execution context), falling back to interpretation or dynamic compilation whenever necessary. However, the requirement in our previous design to duplicate and modify the executable binary image for stitching is a major drawback when targeting embedded systems and scalable servers. In this paper, we describe a new approach that allows stitching to be done on an indirection table, leaving the executable code unmodified and therefore writable to read-only memory. On embedded devices, this saves precious space in writable memory. On scalable servers, this allows a single image of the executable to be shared among multiple JVMs, thus improving scalability. Furthermore, we describe a novel technique for dynamically linking classes that uses *traps* to detect when a class should be linked and initialized. Like back-patching, the technique allows all accesses after the first to proceed at full speed, but unlike back-patching, it avoids the modification of running code. We have implemented this approach in the Quicksilver quasi-static compiler for the Jalapeño JVM. On the SPECjvm98 benchmark suite, our approach obtains code space savings, in writable memory, of between 82% to 89% of that from our previous quasi-static compiler, while delivering performance that is typically within 1% to 7% of that approach, and which is competitive with the performance of the Jalapeño adaptive optimization system.

1. INTRODUCTION

The Java Programming Language [7] is increasingly being used to develop applications for a wide variety of platforms, ranging from embedded systems to servers. A major impediment to its broader acceptance has been its poor performance, relative to languages like C and C++, in many applications. To overcome these performance woes, the simple interpretive model favored by early Java Virtual Machines (JVMs) has been replaced by dynamic compilation systems that compile methods during program execution.

To reduce the overhead of runtime compilation, adaptive optimization systems typically use interpretation or low levels of optimization for most parts of the program, and select frequently executed (or "hot") parts of the code for more extensive optimization. Examples of such systems include the IBM mixed-mode interpreter [11], the Sun Hotspot compiler [24], and the Jalapeño adaptive optimization system [3]. Nevertheless, JVMs using even the most sophisticated dynamic compilation strategies still suffer from two serious problems:

- *Large memory footprint*: Dynamic compilers that support aggressive optimizations are themselves complex software systems, and usually have a large memory footprint. This makes them unsuitable for deployment on embedded systems, where the available memory is severely limited. Therefore, all existing JVMs for embedded devices (examples include the IBM J9

*This work was done while the authors were at the IBM T. J. Watson Research Center. This paper also appears in the proceedings of the ACM 2001 International Conference on Supercomputing, Sorrento, Italy.

JVM [12], the HP ChaiVM [9], and the Sun KVM [23]) either rely on interpretation of bytecodes – abandoning good performance on compiled code, or use static compilation – abandoning compliance with dynamic Java features.

- *Startup overhead of JVMs*: Running even a tiny ("hello world" style) Java program requires execution of tens of millions of native instructions on most platforms. This is due to the interpretation and/or compilation of several core classes that are needed by the JVM. This high overhead is particularly undesirable in a transactional environment in a highly secure server, where each transaction using Java code spawns a new JVM to execute it [5].

While static compilers for Java (such as BulletTrain [17], HPCJ [21], JOVE [13], Marmot [6], and TowerJ [26]) address these problems, they are unable to support the dynamic features of Java. Static compilers cannot support dynamic loading of a class whose bytecode has not been compiled before. Furthermore, runtime binding of methods and the rules for binary compatibility in Java prevent static compilers from applying even simple optimizations, like inlining of methods across class boundaries, without breaking the language specifications [7, 20]. As a consequence, none of the JVM's mentioned above is Java compliant, and all commercial JVMs use dynamic compilation.

Recently, Serrano, et al. [20] presented *quasi-static compilation*, a hybrid compilation model that allows many desirable attributes of static compilation (like incurring the compilation overhead ahead of production runs of a program) to be incorporated into a dynamic compilation system without sacrificing compliance with Java's dynamic features. Under quasi-static compilation, methods in classes are compiled off-line into native, highly optimized executable images. The results of these compilations are saved, along with sufficient information to determine if the compilation is still valid under Java's dynamic semantics. If a pre-compiled image is invalid, the JVM falls back to dynamic compilation, allowing a correct compilation of the methods of the class to be performed. In the expected case, when the pre-compiled image is valid, the optimized native code is loaded and executed directly. During loading of the code of a method to be executed, a process called *stitching* alters the code image, adapting it to the current virtual machine instance.

This approach has several benefits, discussed in detail in [20]. From the perspective of performance, it enables expensive optimizations and whole-program analysis techniques that would be impractical during an application execution, while incurring very low run-time overheads in loading and validating the class. Moreover, because heavily used routines can be optimized off-line, the fall-back compiler that is used in the rare cases when the pre-compiled code is no longer valid can be very simple, reducing memory pressure.

For two large application domains – embedded systems and large servers where hundreds or thousands of instances of a single class may be active at any time – quasi-static compilation as just described is still deficient. The problem lies in the stitching process. Stitching, as described in [20], adapts an executable to a virtual machine instance by editing the executable code. This means that at least two versions of the method code are necessary – one as the original version, and a second as the stitched version. Moreover, each instance of a class in use by another JVM requires producing another stitched copy. Having two copies is clearly unacceptable for embedded systems where memory is a critical resource. For servers, scalability is adversely impacted by requiring that each instance of a class have, in memory, another copy of the compiled code image.

In this paper, we show how a modified form of stitching, inspired by dynamically linked shared libraries, allows us to overcome these problems. More specifically, this paper makes the following contributions:

- It presents a novel approach to get the benefits of static compilation and dynamic linking, while maintaining full compatibility with Java. We describe a solution to the problem just posed that allows a single executable to be used in an embedded quasi-static system, and for that executable to exist in immutable form, e.g. burned into a ROM. Moreover, additional such executables can be dynamically loaded and, along with byte codes, exist and interact within the same JVM. The solution also allows a single executable to be shared among multiple virtual machines in the context of a scalable Java based server environment.

- It presents a novel technique for dynamic linking of Java classes using a *trap* mechanism, which is a superior alternative to backpatching of code. Like backpatching, this technique allows all accesses to a field or method of a class to proceed at full speed after the first access that results in the class being loaded. At the same time, it does not suffer from the drawbacks of backpatching in an SMP environment [2].

- It presents extensive experimental results from an implementation of these ideas in a state-of-the-art JVM, the Jalapeño Virtual Machine [1]. On the SPECjvm98 benchmark suite, we obtain code space savings, in writable memory, of between 82% to 89% of that from the earlier Quicksilver quasi-static compiler [20], while delivering performance that is typically within 1% to 7% of it (for one of the benchmarks, it is outside that range, being worse by 24%). Our compiler delivers performance that is competitive with the performance of a sophisticated adaptive optimization system [3]. In particular, with the benefit of precompilation, for SPECjvm98 benchmarks with size 10 (the short running versions), it delivers overall performance that is 12% to 93% better, considering both compilation and execution times.

The rest of this paper is organized as follows. Section 2 gives an overview of the quasi-static compilation approach, and describes the basic indirection scheme to enable binary code reuse. Section 3 describes how the indirection table is created and used during production runs of the program, including a description of our novel technique for handling dynamic linking in Java. Section 4 presents experimental results. Section 5 describes an extension of the method described in the rest of the paper that should yield better results when whole applications are compiled as a unit. Section 6 discusses related work. Finally, Section 7 presents conclusions.

2. AN INDIRECTION SCHEME FOR CODE SHARING

In this section, we first present a brief overview of the quasi-static compiler. We then discuss techniques that enable the adaptation of immutable pre-compiled executables for use in embedded systems or by multiple, simultaneously active JVM's.

2.1 The Quicksilver Quasi-static Compiler

The Quicksilver quasi-static compiler operates in two logical phases. During the first phase (the *write* phase) methods in classes are compiled and the resulting code is stored in a persistent form, known as a quasi-static image (QSI) [20]. The current implementation forms one QSI per class. The compilation can be done either by executing the program and using dynamic compilation, or by statically compiling the methods in each class. The QSI also includes auxiliary information, such as garbage collection maps and exception tables for the binary code, that are needed by the JVM to execute that code. Furthermore, the QSI contains relocation information that allows the compiler to adapt the QSI to a new execution instance. Because the creation of QSI files is done off-line from a production run of the program, it is possible to expend sufficient resources to produce highly optimized code.

The second phase, called the *read* phase, is used during the production runs of the program. During this phase, the quasi-static compiler tries to reuse the code contained in the QSI produced during the write phase, after performing validation checks and adapting the code to the new execution instance. The validation checks include checks to see whether the original class has remained the same, verifying that the QSI has not been tampered with, and performing dependence checks to ensure that no other classes, on which interprocedural optimizations for this code are dependent, have changed [20]. Once these checks pass, the executable code and auxiliary information from the QSI is stitched, which involves modifying all entries that are no longer valid during the current execution of the program. The remainder of this paper will describe a new methodology for quasi-static compilation that allows the QSI's to be reused without having to modify them.

2.2 Entries requiring Adaptation

We shall now list the items in the executable code and auxiliary information (such as exception tables) which contain values that may be valid only during a particular execution instance of the JVM – the instance that was active during the creation of the QSI.

- *Static field and static method references*: Due to dynamic class loading in Java, a static field or method reference that is resolved during the compilation of the method containing that reference, may not be resolved during compilation in a different JVM instance. Therefore, a direct static reference cannot be reused without modification across different execution instances of the class containing that reference. Furthermore, the Jalapeño system uses a single global array called the *Jalapeño Table Of Contents* (JTOC) to store all static field and method references [1]. The offset into the JTOC for even a resolved static field or method reference depends on the order in which classes are loaded in a given JVM instance, and, in general changes across different instances.

- *Instance field and virtual method references*: Changes to an immediate or ancestral superclass can alter the field and method offsets into an instance of the class [8]. Furthermore, the resolution status of such a reference, at the time of compilation of the method containing the reference, could be different in various execution instances.

- *Symbol table identifiers*: The indices into the symbol table, where entries for types (loaded classes) and interface methods are stored in the Jalapeño JVM, can change over different JVM instances, again due to a different order in which classes are loaded.

- *String, floating-point and double literals*: These literals are interned in, and directly available from, the JTOC in the Jalapeño JVM. Therefore, the corresponding JTOC offsets could change over different JVM instances.[1]

- *Run-time field and method references*: The Jalapeño boot-image consists of a collection of predefined classes that mainly comprise the run-time library of Java. The boot-image classes are needed since Jalapeño is itself written in Java [1]. The boot-image remains "frozen" in time—that is, its classes are always loaded and resolved, with various field and method offsets being fixed across execution runs. Though not strictly necessary, the quasi-static compiler provides the option of stitching boot-image references so as to support version compatibility. In this way, the QSI files can continue to be used even when new releases of the Jalapeño boot-image are used.

2.3 Indirection Tables and Mappings

We introduce an indirection table for each class, which is used to hold, at *fixed* locations, relevant entries for the above kinds of references (that otherwise need to be stitched). This allows references in a compiled method code to remain the same, as long as the class containing that method has not changed.[2]

Figure 1 illustrates the indirection scheme. The entries in the indirection table for static fields, static methods, TIB's[3] and string literals consist of pointers to the corresponding JTOC entries. All other entries, such as offsets of instance fields and virtual methods, symbol table identifiers, and numeric literals, are stored directly in the indirection table. These entries are initially created during the write phase of the quasi-static compiler.

Before the indirection table is built for a class during the write phase, the compiler obtains an upper bound on its size using information about the bytecode size for the class and the code-expansion budget for inlining. If we were to allow the indirection table to grow during compilation, code generation would be more complicated. Consider a method `foo()` that is compiled, and during its execution, calls a method `bar()` of the same class, triggering its compilation. If the compilation of `bar()` leads to a growth in the indirection table, the table (which is an array) would be relocated in memory. Therefore, when `bar()` returns, the base address of the indirection table as previously used in `foo()`'s code would no longer be valid. By ensuring that the indirection table does not need to grow, the code generated to get the (invariant) base address of the table becomes much simpler. By the end of the write phase, the exact size of the indirection table is known for each class, and the table is stored into the QSI for that class.

During the write phase, identifying which indirection table slot to use for an entity such as a static field, static method,

[1]In some cases it is advantageous to make these constants available (and shared) across JVM's [5]. This has not been done in the work described in this paper.

[2]The quasi-static compiler checks for the class being the same before reusing a QSI.

[3]Each object in the Jalapeño implementation carries important information related to its type in an array of Java references called a *type information block* (TIB).

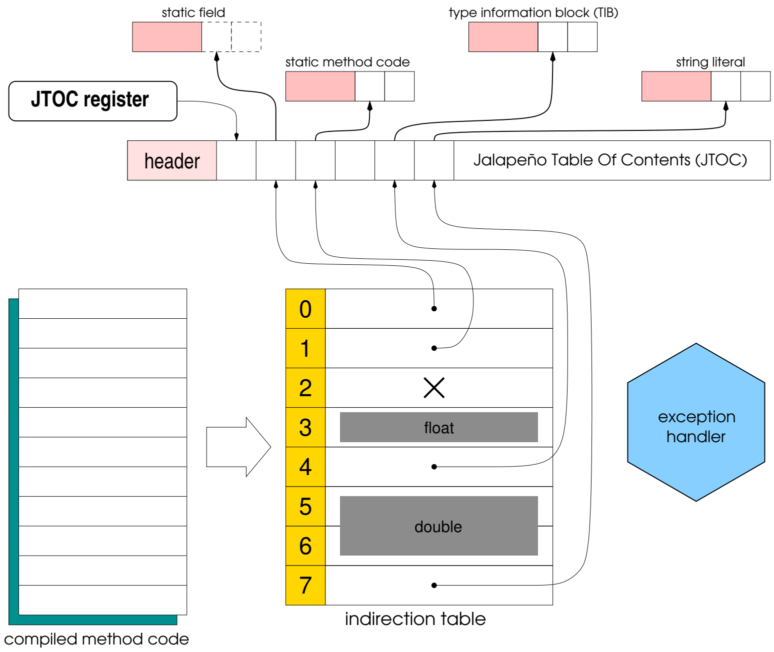

Figure 1: The Indirection Scheme to Quasi-static Compilation

or literal is facilitated by keeping a mapping H (using a hash table) from the *constant pool index* representing that entity to the indirection table index for its slot. As described in [20], for the purpose of stitching, the Quicksilver compiler creates *extended constant pools* to keep entries for methods that are inlined from other classes (for which a constant pool entry may not exist in the given class) – for those entries, H keeps the mapping from the extended constant pool index to the indirection table index.

To allow the indirection table slots to be filled up during the read phase, we record the reverse map R, from each indirection table slot to the constant pool (or extended constant pool) index for the corresponding program entity. The array representing R is stored, in addition to the indirection table, as part of the QSI.

3. USING INDIRECTION TABLES DURING PRODUCTION RUNS

In this section, we discuss how the indirection table is recreated and used during the read phase of the quasi-static compiler.

3.1 Filling the Indirection Table Entries

When a method in class `A` is invoked for the first time during the read phase, the JVM invokes the runtime (quasi-static) compiler, which causes the QSI file for `A` to be loaded and subjected to validation checks. (Note that the JVM would have already loaded, linked, and initialized the class `A` before attempting compilation of any method in it.) The indirection table that is loaded as part of the QSI file is immediately filled in by the quasi-static compiler, using the following approach. The compiler scans through the slots of the indirection table and uses the recorded mapping R to determine the program entity that a slot corresponds to. If the declaring class for that entity is loaded and resolved, the quasi-static compiler updates the contents of the slot. The value of the slot entry is found by querying the Jalapeno runtime for the appropriate JTOC offset. For example, if the indirection table slot is for a static field, the entry is updated to the absolute address of the static field in the JTOC. In this way, slots are updated at load-time. The remaining slots that correspond to program entities whose declaring classes are yet to be loaded and initialized at `A`'s loading time, are set to a special exception-throwing value, and handled, as described in Section 3.2.

The quasi-static compiler need not attempt to update all entries in the indirection table at the time of initial loading. Slots that correspond to numeric literals will have contents that remain the same with every execution run. Our current implementation saves the entire indirection table into the QSI for the sake of simplicity. In principle, it may just store *(literal,slot index)* pairs in the QSI (instead of the entire indirection table), and insert them into the indirection table when it is first created following the loading of the QSI.

String literals are treated differently, in that they are not directly stored in the indirection table. This is due to the Java language requirement that strings be interned [15] – i.e., identical string literals throughout the program must refer to the same instance of the class `java.lang.String`. Therefore, the address of the (global) JTOC slot that contains the string literal reference is stored instead in the indirection table for a class. We store the address of a reference to the string rather than the reference itself to the string because the address of a string literal object can change during program execution due to garbage collection.

3.2 Dynamic Linking using Traps

We now describe a novel technique for efficiently dealing with a potentially unresolved reference (such as a static field or method reference) to another class, without having to modify the generated code. The JVM specification requires that a class be initialized at its first active use [15]. Therefore, even though the class being referenced (say, B) may be eagerly loaded (with some restrictions on when an error during class loading may be reported), the JVM is not free to initialize the class B eagerly while compiling methods of the original class (say, A).

Two techniques have been used in most JVM's to handle accesses to unresolved references. The first is to use an explicit test using a flag to see if the reference is resolved yet. If not, a branch to a resolution code (which loads, links and initializes the corresponding class B) is taken. Subsequent accesses to the reference pass the test. This approach incurs the overhead of the test for every execution of the guarded access. Runtime compilers (such as the Jalapeño optimizing compiler) take advantage of the runtime information about the resolution status of classes to use this check only if class B has not been resolved[4] at the time of compilation of the method in class A. The second technique also uses a test in the code to determine if the class has been resolved. If it has not, the class is resolved, and then the test and access code is overwritten, or *back-patched*, to directly reference the original class member. This approach suffers from requiring the code to be mutable, and not sharable, since the resolution status of the class may differ across different JVM instances. Additionally, it suffers from all of the synchronization and I-cache issues that plague self-modifying code on a modern processor [2].

In contrast, we present a *trap*-based strategy to class resolution. The basic idea is to mark an unresolved entry with a special value, (denoted X in Figure 1), which causes a trap to be generated on the first access. The trap handler takes care of resolving the class and updating the entry in the indirection table, so that future accesses are fast. We illustrate our approach by discussing the details of code generation for a `getstatic` instruction:

```
getstatic B.f
```

where `f` is a static field declared in the class `B`. When the above instruction is compiled, class `B` could either be resolved or unresolved. If resolved, the offset *offset*[5] into the JTOC for the field in question will be known at compile-time, allowing the Jalapeño optimizing compiler to generate the following code, shown in low-level intermediate representation:

$$\texttt{int\_load r} \leftarrow \texttt{JTOC,}\ \mathit{offset}$$

The above instruction adds *offset* to the base address of the JTOC (held in a dedicated register called `JTOC`), and loads the contents at that address into register `r` (the JTOC entry contains the actual value of the static field). The Quicksilver compiler, described in [20], takes care of the following adaptation of code during the read phase: (i) if the class B is already resolved at stitching time, it updates the value of *offset*, or (ii) if the class B is not yet resolved at stitching time, it patches in a jump to code that performs the class resolution and retries the instruction with the appropriate offset.

In our new, indirection approach, *irrespective* of whether class `B` is resolved or unresolved, the following code is generated:

$$\begin{array}{l} \quad\texttt{int\_load t} \leftarrow T_A\texttt{,}\ \mathit{tableOffset} \\ \star\ \texttt{int\_load r} \leftarrow \texttt{t, 0} \end{array}$$

where T_A is a register containing the base address of the indirection table for class `A` (being compiled), and *tableOffset* is the offset into the table where the slot for the static field `B.f` is located. The second instruction in the above code sequence is annotated with a "$\star$" to indicate that it can throw an exception. If the indirection table slot at *tableOffset* contains a valid JTOC address (which will happen only if the class `B` is already resolved), the above instruction sequence will execute normally, producing the desired outcome in register `r`. If, however, the class `B` is unresolved, the associated indirection table slot will contain the special exception-throwing value X, which is an address in high memory that is not accessible to the process. Therefore, the second instruction will generate a trap. The trap handler identifies the faulting instruction (the auxiliary data stored with that instruction gives the handler sufficient information about the cause and remedy), resolves class `B`, updates the indirection table, sets the contents of register `t` to the resolved address, and retries $\langle \texttt{int\_load r} \leftarrow \texttt{t}, 0\rangle$. From then on, execution proceeds as before. Interestingly, class resolution can result both in other statics associated with the class being available, and also other classes becoming resolved [15]. Therefore, as an optimization, the trap handler also updates the indirection table with other entries that have become available.

3.3 Comparison with backpatching and explicit checks

Our trap-based technique for dynamic linking can serve as an attractive alternative to back-patching even in purely dynamic compilation systems (as long as the operating system supports precise traps). For instance, in Jalapeño, by setting the JTOC entry to an exception-throwing value if the corresponding class is not resolved, the single `int_load` instruction described above could be used to handle the `getstatic` instruction, regardless of the resolution status of the class. Our technique delivers the advantage of back-patching, in that all accesses after the first access proceed at full speed, without the drawbacks and complications of back-patching: the need to invalidate the instruction cache block and instruction prefetch buffer, and, in an SMP environment, the need to avoid data races [2].

While handling a trap is more expensive than a conditional check for whether a class is already resolved, the check-based approach has the drawback of incurring a repeated check overhead for each access during program execution. Furthermore, the check-based approach is likely to result in a slight increase in the code size. Therefore, in long-running codes, the trap-based approach can be expected to perform better. Section 4 presents experimental results comparing the performance obtained with the two approaches.

3.4 Code Generation

We now describe how code generation (performed during the write phase) is modified so that the generated code properly uses the indirection table during the read phase.

3.4.1 Setting the Base Pointer for the Indirection Table

The method prologue code is modified to set the pointer holding the base address of the indirection table for its class (which is identified using the efficient, internal support for *reflection* in Jalapeño [1]). This ensures that the base address is available before any code in that method is executed. This modification is done before the register allocation phase of the Jalapeño optimizing compiler, to avoid hardwiring a decision on whether or not to use a register to hold the base address.

3.4.2 Compiling Instructions that use Indirection

We have already discussed in Section 3.2 how references

[4] We are using the term *resolved* loosely, to indicate that the class has been loaded, linked, and initialized.

[5] We adopt the convention of italicizing *immediate values* in the code.

to static fields and methods are handled. Interface methods are invoked in Jalapeño via an interface method table. Whenever an interface method is to be invoked, the run-time system indexes into the object's interface method table using the interface method's dictionary identifier. Since the dictionary identifier can change from one JVM instance to another, we use an entry in the indirection table to keep this identifier, thus handling interface method calls without having to modify the code.

We now describe how references to instance fields and virtual method invocations are handled. We first discuss the basic approach, and then present an improved version that was used in our implementation. For the purpose of illustration, let us consider an `invokevirtual` instruction (appearing in code for a method in class `A`) for a virtual method `b.foo()`, where `b` is a reference to an object of type `B`.

The basic approach is similar to the way we handle static fields and methods – regardless of whether or not class `B` is resolved, the compiler would generate the following code sequence to load the address of the method into register `r`:

$\star$ `get_obj_tib tib` $\leftarrow$ `b`
`int_load t` $\leftarrow$ T_A, *tableOffset*
$\star$ `int_load r` $\leftarrow$ `tib, t`

The `get_obj_tib` pseudo-code instruction accesses an object's type information block (TIB), given a reference to the object. (A reference to the TIB, which contains the virtual method table, is part of a two-word header in each object in the Jalapeño VM [1].) If `b` is `null`, a `NullPointerException` exception would be thrown by the first instruction in the above sequence. If `b` is not `null`, then class `B` must be resolved. However, the third instruction would generate a trap if the indirection table slot for the virtual method (obtained by the second instruction) does not have a valid value. (The trap-generating value X is chosen such that any valid object address added to it will still lie in high memory.) If a trap is generated, the trap handler appropriately updates the indirection table slot and then successfully re-executes the instruction.

An improved approach, which we use in our implementation, is to avoid using the indirection table if the virtual method's declaring class is resolved during the write phase. The offset *vOffset* into the virtual method table for the given instance method would then be known during the write phase and can therefore be used directly in the generated code (instead of being obtained from the indirection table):

$\star$ `get_obj_tib tib` $\leftarrow$ `object`
`int_load r` $\leftarrow$ `tib,` *vOffset*

As long as there are no changes to class `B` or any of its superclasses, this offset *vOffset* will continue to remain valid in every subsequent JVM instance, independent of the order in which classes get loaded. However, to take into account the additional dependence of the generated method code on the superclasses of `B`, this dependence is recorded during the write phase, using the usual mechanism available for accommodating interprocedural optimizations in Quicksilver [20]. During the read phase, the method code is invalidated whenever a dependence check fails (due to a change in any superclass of `B`), in which case the quasi-static compiler falls back to non-optimizing compilation using the Jalapeño *baseline* compiler. For relatively stable code, in which such changes are not very frequent, this is a reasonable choice (over using indirection) to improve performance.

4. EXPERIMENTAL RESULTS

This section presents experimental results obtained from the implementation of our approach in the Jalapeño JVM as part of the Quicksilver compiler. We present results on the memory savings in code size achieved by our techniques. We then describe how the performance of this approach compares with that of the previous quasi-static compilation using in-place stitching [20] and with that of a sophisticated runtime adaptive optimization system [3]. We also present results showing the performance impact of some of the design trade-offs we have made, and present important metrics related to the composition of the indirection table.

All experiments were done using the SPECjvm98 industry-standard benchmark suite [25]. These experiments were performed using two threads on a 12-way 64-bit SMP server running the AIX 4.3.3 operating system and having a system-wide memory of 16 GB. Each node in the machine was the 450 MHz PowerPC RS64III processor with a 8MB L2 cache. The highest compilation optimization level (i.e., `-O3`) was used during the write phase for both the indirection and the in-place stitching approaches. Execution runs with the Jalapeño adaptive optimization system were performed with the adaptive code recompilation option enabled. A copying garbage collector was used in all of the executed JVMs.

4.1 Memory Savings

A sophisticated and purely dynamic compilation approach would effectively add the memory footprint of the compiler itself to that of the application, preventing it from being a viable option on a limited-memory device. Interpretation alone leads to poor performance, both in terms of time and energy consumption. Hence, we shall analyze the memory savings by comparing our approach with the previous quasi-static compilation approach that uses in-place modification of code for stitching.

To analyze the memory savings in code size made possible by the indirection approach, we introduce metrics that quantify memory overheads under an embedded system scenario. For an embedded system, let M and M' respectively represent the overall memory costs using the in-place stitching technique and the indirection scheme. Each of these memory costs can be regarded as being an additive sum of two parts: a *volatile memory cost* (M_{v} and M'_{v}) and a *nonvolatile memory cost* (M_{nv} and M'_{nv}). Thus,

$$\begin{aligned} M &= M_{\mathrm{v}} + M_{\mathrm{nv}}, \\ M' &= M'_{\mathrm{v}} + M'_{\mathrm{nv}}. \end{aligned} \tag{1}$$

Different applications running simultaneously on an embedded system, such as a handheld PDA, may use many identical classes, such as those of the JDK class library. With the indirection approach, we would only need separate indirection tables (rather than complete binary code) in volatile memory (such as RAM) for each loaded class instance, in addition to the single copy of binary code for each class in nonvolatile memory (such as ROM). Using a ROM to store the single shared program text allows precompiled software to be shipped by the manufacturer, and also prevents the accidental overwriting or tampering of code.

An exact measurement of the memory savings on a PDA would require measuring class by class savings, over some standard workload, to determine the system-wide savings. Since the infrastructure to perform these measurements does not yet exist, we approximate them with measurements of savings for whole applications. We note that for reasons discussed below, the savings of potentially more expensive volatile memory (such as RAM) per class instance are invariant of the number of instances of a class that is loaded by different applications, and therefore the numbers we present are precise, and not approximations of, the volatile memory savings for whole applications.

If an in-place stitching technique were used for code adaptation, we would have n stitched copies of the method code resident in RAM, and one original copy in ROM from which the n stitched versions are produced. Hence, the dominant contributions to each of the memory component costs in Eqs.(1) and (2) would be

$$M_{\mathrm{v}} \approx nQc_{\mathrm{v}} \qquad M_{\mathrm{nv}} \approx Qc_{\mathrm{nv}}, \tag{2}$$
$$M'_{\mathrm{v}} \approx nIc_{\mathrm{v}} \qquad M'_{\mathrm{nv}} \approx Q'c_{\mathrm{nv}}, \tag{3}$$

where Q and Q' denote the application-wide method code sizes with the in-place stitching technique and the indirection scheme respectively, I represents the *total* indirection table size per JVM instance, and c_{v} and c_{nv} respectively indicate the per-byte costs of the volatile and nonvolatile memory modules. Table 1 shows these values measured for the SPECjvm98 benchmark suite, for input sizes of 10 and 100. The measurements for Q' and I were taken when a trap-based strategy was used to update the indirection table.

If we were to determine the savings ξ_{v} in writable memory, expressed as a fraction of the writable memory cost when in-place stitching is used, we would obtain

$$\xi_{\mathrm{v}} = \frac{M_{\mathrm{v}} - M'_{\mathrm{v}}}{M_{\mathrm{v}}} \approx \frac{n(Q-I)c_{\mathrm{v}}}{nQc_{\mathrm{v}}} = \frac{Q-I}{Q}. \tag{4}$$

We could also compute the *overall* memory savings $\xi(n)$ as a fraction of the original memory cost (i.e., corresponding to in-place stitching):

$$\xi(n) = \frac{M - M'}{M} \approx \frac{(nc_{\mathrm{v}} + c_{\mathrm{nv}})(Q-I) - (Q'-I)c_{\mathrm{nv}}}{(nc_{\mathrm{v}} + c_{\mathrm{nv}})Q} \tag{5}$$
$$= \xi_{\mathrm{v}} - \frac{(Q'-I)}{(n\frac{c_{\mathrm{v}}}{c_{\mathrm{nv}}} + 1)Q}. \tag{6}$$

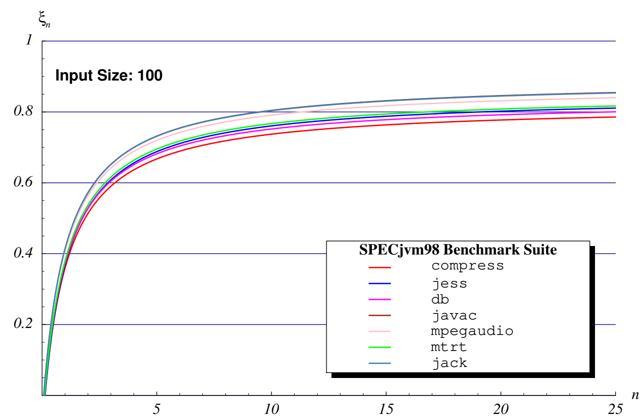

Figure 2: Overall Memory Savings $\xi(n)$

From Eq.(5), it is clear that the overall memory savings asymptotically approaches the writable memory savings. This should not be surprising since with every new JVM instance, the additional memory required is of the writable kind, and since for every Q bytes of RAM required by the in-place stitching technique, the indirection scheme requires only I bytes of RAM, where I is much smaller than Q. By a similar argument, it can be seen that the amount of RAM saved for each class that is loaded is independent of the number of instances of the class that have been loaded: each loading of a class saves $Q - I$ bytes of RAM.

The cost factors c_{v} and c_{nv} could be considered as either reflecting the power consumption levels of each type of memory, their access time costs, or simply their dollar costs. Though the ratio $c_{\mathrm{v}}/c_{\mathrm{nv}}$, which dictates the actual savings achieved, is a function of the memory technology used, its value in a given system is typically greater than 1. Therefore, we can obtain a lower bound to the savings by considering this ratio to be unity. Figure 2 graphs this lower bound for the SPECjvm98 benchmarks using the measured values Q', I and Q from Table 1, for a benchmark input size of 100. Notice from the curves that the savings initially accrue at a fast rate. For instance, with one JVM instance, the savings are between 0.36 to 0.41, while for two JVM instances, the savings jump to between 0.51 to 0.57. By the time we reach five JVM instances, the savings are already within 17% to 18% of the maximum possible. This rate of growth is basically controlled by the ratio $c_{\mathrm{v}}/c_{\mathrm{nv}}$, and the larger the ratio, the faster is the rate at which the maximum savings are approached. The curves in Figure 2 should therefore be considered as portraying a conservative estimate of the improvement in savings. In Table 1, two sample values $\xi(1)$ and ξ_{v} for $\xi(n)$ have been computed. Note that $\xi(\infty)$ is nothing but ξ_{v}.

We can similarly devise metrics that model the memory cost of sharing class libraries across JVM instances in a server setting. The trends are similar to that seen in Figure 2 except for an important departure: Rather than savings, losses occur at $n = 1$. This happens because for a single JVM instance, indirection requires $Q' + I$ bytes of main memory and Q' bytes of secondary storage, whereas in-place stitching requires only Q bytes of main memory and Q bytes of secondary storage. Since Q' always exceeds Q, memory losses, rather than memory savings, will occur in this particular setting. Actual savings begin to manifest from $n = 2$ and above, reaching the same asymptotic overall memory savings seen in the embedded system case.

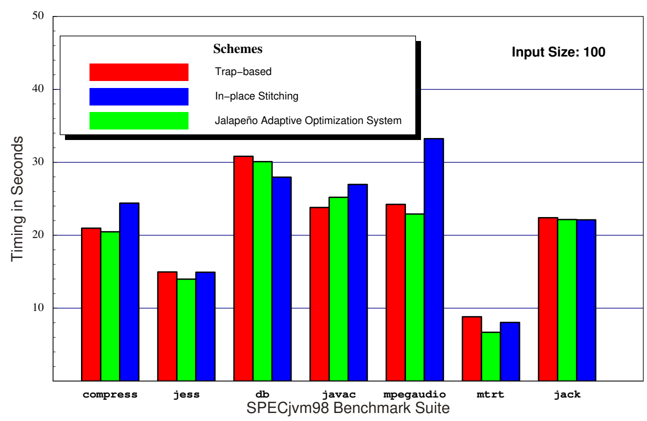

Figure 3: Timing Measurements for an Input Size of 100

4.2 Performance

The graphs in Figure 3 and Figure 4 show how the indirection approach, presented in this paper, compares in performance with the in-place stitching approach for quasi-static compilation (described in [20]), and with the Jalapeño adaptive optimization system (described in [3]). The graphs

Table 1: Method Code and Indirection Table Sizes

Benchmark	SPECjvm98 Input Size									
	10					100				
	Q' (bytes)	I (bytes)	Q (bytes)	$\xi(1)$	ξ_v	Q' (bytes)	I (bytes)	Q (bytes)	$\xi(1)$	ξ_v
`compress`	27366	4472	24842	0.359	0.820	27616	4480	25055	0.359	0.821
`jess`	59256	8200	52944	0.363	0.845	60319	8268	54299	0.368	0.848
`db`	29512	4528	27071	0.371	0.833	29719	4480	27297	0.374	0.836
`javac`	107836	11004	101307	0.413	0.891	108818	11072	101984	0.412	0.891
`mpegaudio`	73932	8652	69281	0.404	0.875	74634	8716	70318	0.407	0.876
`mtrt`	39276	5272	35938	0.380	0.853	39423	5272	35967	0.379	0.853
`jack`	66197	6832	61900	0.410	0.890	66197	6832	62268	0.414	0.890

show that in general, the indirection approach incurs a small performance degradation with respect to the in-place stitching approach. This degradation occurs when static fields, static methods, TIBs, string literals and interface method dictionary identifiers are accessed. For the aforementioned program entities, code generated in the indirection scheme goes through an extra level of indirection via the indirection table. However, code generated by the in-place stitching technique would directly access the relevant JTOC slots, perhaps after stitching the slot offsets in the generated code. For all the benchmarks except `mtrt`, this degradation was found to be between 1.1% to 6.6% for an input size of 100, and between 0.9% to 4.5% for an input size of 10. For `mtrt`, the degradation was more pronounced; it was 24.2% for an input size of 100, and 13.7% for an input size of 10.

In certain cases, the indirection scheme generates more efficient code than the in-place stitching technique. For example, if a reference to an unresolved static field is encountered in the write phase, the in-place stitching approach would generate resolution code along with code that performs the actual JTOC access. The resolution code would be executed each time prior to the actual access in the read phase, so as to ensure the resolution status of the static field involved. In the indirection scheme, the first access to the static field in the read phase will automatically force any necessary resolution using the exception-based strategy, while subsequent accesses will not incur the overhead of a resolution status check. This is the reason behind the small degradation of 1.1% for `javac` for an input size of 10, and why a performance improvement of 5.8% occurs for the same benchmark at an input size of 100.

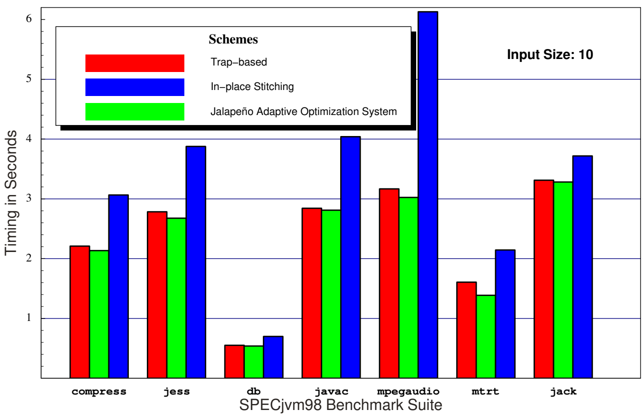

Figure 4: Timing Measurements for an Input Size of 10

Timings in Figure 3 and Figure 4 also show how the indirection scheme performs in relation to the Jalapeño adaptive optimization system. Overall, the timings are competitive, ranging between performance improvements of 37.2% for `mpegaudio`, 16.4% for `compress`, 13.3% for `javac`, to performance degradations of 0.3% for `jess`, 1.3% for `jack`, 8.9% for `mtrt` and 9.3% for `db`—all for an input size of 100. Performance degradations relative to the adaptive system are likely to be due to the overhead of indirection and the benefits of the delayed program optimizations that the Jalapeño adaptive optimization system applies at run-time, while the improvements are primarily due to the savings in run-time compilation overhead due to the reuse of QSI's. For the small input size of 10, the indirection scheme performed consistently better than the Jalapeño adaptive optimization system, registering performance improvements of 93.5% for `mpegaudio`, 42.1% for `javac`, 39.3% for `jess`, 38.7% for `compress`, 33.5% for `mtrt`, 26.9% for `db` and 12.3% for `jack`.

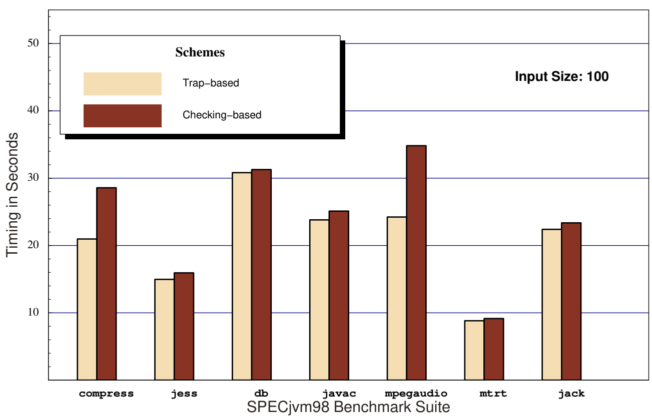

Figure 5: Comparing Indirection Table Update Strategies

Finally, Figure 5 shows how the trap-based strategy to updating the indirection table compares with a checking-based strategy in terms of performance. For the SPECjvm98 benchmark suite, the trap-based strategy we have used was found to be consistently superior. For an input size of 10, the performance improvements ranged from 0.3% to 30.5%, while for an input size of 100, the improvements were between 0.5% to 30.4%.

5. GRANULARITY OF CODE SHARING

The indirection scheme discussed so far creates and maintains an indirection table for every class in the application.

If a class is used by two or more JVM instances, each instance would have a separate indirection table for the class. In this way, sharing of code across application instances is permitted, with the granularity of sharing at the level of a class. The granularity of sharing can be made coarser: in situations in which a set of classes are always used together (such as classes that form *sealed* user packages [27]), all of the classes in the set could share the same indirection table in an application instance.

The granularity of code sharing could be further increased until the entire application code is reused across different execution instances of the application. For example, in a transactional server, numerous instances of the same application often run simultaneously, each handling a separate

client request. In this situation, each instance of the application could use a single or "global" indirection table. All code for the application would be compiled to use this global indirection table, with each application instance maintaining its own copy of the global indirection table, since the contents of this table could be different in different instances. For the special case of multiple execution instances of the same application, however, the roles of the global indirection table and the JTOC in each instance could be merged. This extension is shown in Figure 6.

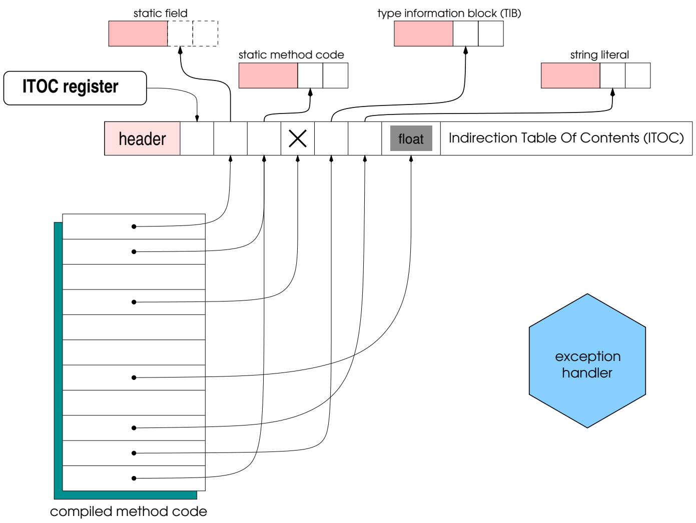

Figure 6: Using a Global Table for Indirection

As shown in the figure, the extension proposes to replace the JTOC with a global array called the Indirection Table Of Contents (ITOC). The ITOC is similar to the JTOC in that its slots hold references to static fields, static methods, TIBs, string literals, and numeric literals. In addition to these entries, the ITOC also holds other entries such as instance method offsets and dictionary identifiers not contained in the JTOC. We have made measurements which show that these fields occupy only a small percentage of the indirection table entries. Because of information overlap across local indirection tables, these percentages may be even smaller for a global indirection table. Memory savings are increased over the indirection table approach in two other ways. First, fields in the indirection table which point to fields in the JTOC are, in some sense, a redundant representation of the referenced JTOC field in that they make access to that field possible without modifying the code, but do not add to the information contained in the field. Second, references to the same JTOC entries in the indirection tables of different classes are redundant, and merged in the ITOC scheme. Thus, further improvements in memory savings are likely. Since program entities such as instance method offsets and dictionary identifiers account for a small fraction of the global indirection table, the overall size of the ITOC will be competitive in relation to that of the JTOC while providing the benefits of using indirection tables.

This representation is possible because, unlike the JTOC where slots are filled in as classes are loaded and resolved, slots in the ITOC are permanently assigned to a particular class. This assignment occurs during the write phase and remains fixed in future executions of the application. Thus, whenever a class is loaded in the read phase, it will always be assigned to the same set of slots in the ITOC. Any classes without predefined ITOC slots that are loaded during the read phase can be loaded into later, unused, slots in the ITOC and adapted to the JVM instance either by using in-place stitching or by using local indirection tables.

Adding to the attractiveness of this extension is that, in addition to further reducing the overall memory footprint of the application, it enables the generation of more efficient code sequences since a level of indirection is removed. For instance, for the `getstatic` JVM instruction discussed in Section 3.2, the following code sequence now becomes possible:

$\star$ `int_load r` $\leftarrow$ `ITOC,` *itocOffset*

where *itocOffset* remains unchanged across future executions of the application, and where `r` contains a one-word primitive data type or a reference to an object.

6. RELATED WORK

Several dynamic compilers for Java have been developed, they include the IBM JIT [22], Sun HotSpot [24], Microsoft JIT [16], and the OpenJIT [18]. We differ significantly from this work in that we allow compiled native code images and related information to persist longer than the lifetime of the JVM executing the code. Closer to our work are several static "Java" compilers most prominently HPCJ [21], JOVE [13], Marmot [6], TowerJ [26] and BulletTrain [17]. However, unlike our compiler, none of these compilers fully comply with the Java language specifications. Our work builds upon the quasi-static compilation approach, described in [20]. By developing novel techniques so that previously generated binary code can be reused without any modifications to it, we significantly enhance the utility of Java on embedded systems as well as scalable servers.

The *Chai* environment from Hewlett-Packard provides an embedded JVM [9] and byte-code interpreter. The turboChai [10] ahead-of-time compiler provides a byte-code to C compiler that runs on, and interacts with, the chaiVM JVM. It supports dynamic class loading by the pre-compiled classes. It does not support sharing of precompiled classes among multiple JVM's, or sharing of classes among multiple applications. Detailed performance numbers for applications running under chaiVM do not seem to be publicly available. In general, performance will vary depending on the quality of the C compiler used to compile the output from turboChai.

Two projects within IBM are related to the work described in this paper. The first project is a scalable Java virtual machine implementation for OS/390 [5]. This work is similar to ours in that their system allows multiple JVMs to use the same code image by putting static data in a shared heap. Because they exploit special features of OS/390 to accomplish this, rather than using indirection tables, their approach is not as general as ours. Their code images only live as long as the shared heap that contain them, and are not as long-lived as those in our approach.

The second project [19] was developed by the IBM AS/400 JVM team. A significant difference in this project from ours is that there is no central JTOC. Rather, static information for each class is a separate data structure that is associated with a loaded instance of the class. Interned data, such as strings, are stored in a separate structure. Like static compilers, their system uses explicit checks to deal with potentially unresolved references, in contrast to the trap-based dynamic linking technique that we use. Neither this project, nor the OS/390 based work, have released detailed descriptions of their implementation and performance data to characterize the savings in storage that have been accomplished, and the costs and benefits of the different design decisions.

The Forest project [14] at Sun Labs deals with *orthogonal persistence* for the Java platform (OPJ). OPJ provides support for checkpointing the state of a Java application. OPJ is a programming model that enables generation and reuse of persistent images of the Java application variables including classes, objects and threads. Our approach does not store the state of a Java application, rather it stores and reuses method binaries needed for the Java application.

Another project [4] at Sun Labs uses indirection to share code in a multitasking Java environment, i.e. where multiple applications are executing on a single JVM. This goal is different than our goal which is to efficiently support pre-compiled code in an embedded or scalable environment. They have two implementations – both of which assume bytecode interpretation, which prevents them from gaining the advantages of quasi-static compilation. In their bytecode editing implementation, indirection to allow multiple applications to share code without sharing class statics is supported by using helper classes. Their bytecode based approach also requires manual editing of a small number of system classes. Their runtime based approach replicates the static fields to a predefined maximum number, which becomes the maximum number of applications that can share the class code. Our method has no such limit.

7. CONCLUSIONS

The portability of Java has made it an attractive target for both embedded systems and large, scalable servers. Both of these domains are characterized by a wide range of hardware and software environments to be targeted, and by critical performance challenges. The approach of using dynamic compilation to achieve good performance for Java has led to severe problems in its acceptance in both embedded systems and servers. Quite simply, it is unrealistic to expect an embedded system to host a full dynamic compiler capable of performing aggressive power and space optimizations. The same dynamic compilation model presents difficulties on servers by restricting the sharing of code images, reducing the scalability of applications. Interpretation is fundamentally limited in performance, and does not provide a long term solution to executing Java on embedded and scalable platforms.

This paper has presented elegant and practical solutions to these problems, solutions that enable Java to become the preferred platform for application development on both of these platforms. By employing quasi-static compilation, our solutions provide the excellent performance of native code on both embedded systems and scalable servers, without the overhead of hosting dynamic compilers and performing dynamic compilation, and while providing full Java compliance. We allow the executable native code image for the methods of a class to be immutable by performing all stitching, or linkage operations, on an indirection table. The approach has the benefits of allowing the code to be stored in the ROM of an embedded system, and be shared by all applications on the embedded device; in a shared read-only segment on a server, and in a CD-ROM or DVD-ROM for shrink-wrapped applications. Sharing of the immutable code image by different applications only requires allocating the indirection table, which minimizes memory usage on an embedded device and increases scalability on a server. Moreover, our approach obtains these benefits while providing application performance that is, on average, better than that provided by advanced adaptive, or hot-spot, dynamic compilation systems.

Therefore, our approach provides high performance, a small memory footprint, and the portability and safety benefits of full Java compliance to the developers of applications on embedded systems and scalable servers.

In the future, we plan to investigate ways to make our techniques even more space and time efficient. In particular, by allowing indirection tables to be created for multiple classes, entries that currently appear in the tables of two or more classes can be replaced by a single entry. Going a step further, the JTOC and indirection tables can be merged, with predefined slots in the merged structure being reserved for classes with a high probability of being loaded and resolved. Merging the two structures will be more efficient than simply merging indirection tables for multiple classes. This is because indirections are no longer needed for references to the predefined slots, making the references faster and removing the need for a *de facto* indirection table slot for the referenced slot. We also plan to exploit our ability to do off-line compilation to implement optimizations and analyses that require large time and space resources at compile time.

8. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. Flynn-Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] B. Alpern, M. Charney, J.-D. Choi, A. Cocchi, and D. Lieber. Dynamic linking on a shared-memory multiprocessor. In *Proceedings of PACT'99*, Los Angeles, California, June 1999.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming and Systems, Languages, and Applications (OOPSLA) 2000*, Minneapolis, MN, October 2000.

[4] G. Czajkowski. Application isolation in the Java Virtual Machine. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 354 – 366, Minneapolis, MN, USA, Oct. 2000.

[5] D. Dillenberger, R. Bordawekar, C. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. Oliver, F. Samuel, and R. S. John. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Systems Journal*, 39(1):194–210, 2000. Java Performance Issue.

[6] R. Fitzgerald, T. Knoblock, E. Ruf, B. Steensgard, and D. Tarditi. Marmot: An optimizing compiler for Java. Technical Report 33, Microsoft Research, June 1999.

[7] J. Gosling, B. Joy, and G. Steele. *The Java$^{(TM)}$ Language Specification*. Addison-Wesley, 1996.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. The Java Series. Addison-Wesley Publishing Company, Redwood City, CA 94065, USA, 2000.

[9] HP ChaiVM white paper. `http://www.internetsolutions.enterprise.hp.com/chai/infolibrary/whitepapers/chai_vm_wp.html`.

[10] Turbochai compiler. `http://embedded.hp.com/products/devtools/turbochai.html`.

[11] IBM rewrites the book on Java performance. `http://www.developer.ibm.com/java/j2/j2perfpaper.html`.

[12] Virtual machines are not all created equal: The J9 difference. `http://www.embedded.oti.com/learn/vaesvm.html`.

[13] Instantiations, Inc. Jove, Super Optimizing Deployment Environment for Java. At `http://www.instantiations.com/_vti_bin/shtml.dll/JOVE/jovereportdownload.htm`.

[14] M. Jordan and M. Atkinson. Orthogonal Persistence for the Java Platform: Draft specification, October 1999. `http://www.sun.com/research/forest/index.html`.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. The Java Series. Addison-Wesley Publishing Company, Redwood City, CA 94065, USA, 1999.

[16] MS SDK for Java 4.0. `http://www.microsoft.com/java/vm.htm`.

[17] NaturalBridge, Inc. BulletTrain Description. At `http://www.naturalbridge.com/bullettrain.html`.

[18] H. Ogawa, K. Shumira, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura. OpenJIT : An open-ended, reflective JIT compiler framework for Java. In *Proc. European Conference on Object-Oriented Programming*, Cannes, France, June 2000.

[19] P. Richards and D. Hicks. Virtual integration. *AS/400*, pages 50–56, March 1998.

[20] M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 66 – 82, Minneapolis, MN, USA, Oct. 2000.

[21] V. Seshadri. IBM high performance compiler for Java. AIXpert Magazine, September 1997. URL: `http://www.developer.ibm.com/library/aixpert`.

[22] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[23] EmbeddedJava$^{(\mathrm{TM})}$ application environment. `http://java.sun.com/products/embeddedjava/`.

[24] Sun Microsystems, Inc. The Java HotSpot Performance Engine Architecture. At `http://java.sun.com/products/hotspot/whitepaper.html`, apr 1999.

[25] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. At `http://www.spec.org/osg/jvm98/`, 1998.

[26] Tower Technology. TowerJ3—A New Generation Native Java Compiler And Runtime Environment. At `http://www.towerj.com/products/whitepapergnj.shtml`.

[27] A. Zaks, V. Feldman, and N. Aizikowitz. Sealed calls in Java packages. In *Proceedings of the Conference on Object-oriented programming, systems, languages and application (OOPSLA'00)*, pages 83–92, Oct. 2000.