# Quicksilver: A Quasi-Static Compiler for Java

Mauricio Serrano Rajesh Bordawekar Sam Midkiff Manish Gupta
IBM T. J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598
{mserrano, bordaw, smidkiff, mgupta}@us.ibm.com

## ABSTRACT

This paper presents the design and implementation of the Quicksilver[1] quasi-static compiler for Java. Quasi-static compilation is a new approach that combines the benefits of static and dynamic compilation, while maintaining compliance with the Java standard, including support of its dynamic features. A quasi-static compiler relies on the generation and reuse of persistent code images to reduce the overhead of compilation during program execution, and to provide identical, testable and reliable binaries over different program executions. At runtime, the quasi-static compiler adapts pre-compiled binaries to the current JVM instance, and uses dynamic compilation of the code when necessary to support dynamic Java features. Our system allows interprocedural program optimizations to be performed while maintaining binary compatibility. Experimental data obtained using a preliminary implementation of a quasi-static compiler in the Jalapeño JVM clearly demonstrates the benefits of our approach: we achieve a runtime compilation cost comparable to that of baseline (fast, non-optimizing) compilation, and deliver the runtime program performance of the highest optimization level supported by the Jalapeño optimizing compiler. For the SPECjvm98 benchmark suite, we obtain a factor of 104 to 158 reduction in the runtime compilation overhead relative to the Jalapeño optimizing compiler. Relative to the *better* of the baseline and the optimizing Jalapeño compilers, the overall performance (taking into account both runtime compilation and execution costs) is increased by 9.2% to 91.4% for the SPECjvm98 benchmarks with size 100, and by 54% to 356% for the (shorter running) SPECjvm98 benchmarks with size 10.

## 1. INTRODUCTION

The Java Programming Language [20] has become a popular language for a wide spectrum of application domains: from high-end servers [16] to client-side applications, and even for embedded devices [11, 32]. Java programs are run within a virtual machine called the Java Virtual Machine (JVM). The JVM can either interpret the Java bytecode or compile it into the native code of the target machine. Because interpretation incurs a high runtime overhead, JVMs usually rely on compilation. The most popular compilation approach is to perform dynamic or just-in-time (JIT) compilation, where translation from bytecode to native code is performed just before a method is executed. Examples of dynamic compilers available as products include the IBM JIT [39], the Sun JDK and HotSpot [31, 22], and the Microsoft JIT [13]. Several other dynamic compilers have been developed, which include Jalapeño [2], OpenJIT [34], and LaTTe [41]. The other compilation approach is to statically compile the bytecodes and then execute the compiled binary in a Java run-time environment. Examples of static compilers include HPCJ [37], Marmot [19], TowerJ [40], JOVE [26], and BulletTrain [23].

This paper presents *quasi-static* compilation, a novel approach that combines the benefits of static and dynamic compilation without sacrificing compliance with the Java standard in supporting its dynamic features. Quasi-static compilation involves two phases: offline generation of optimized binaries and runtime "compilation" of these binaries. The offline compilation can use either a static or dynamic compiler to generate these binaries. The optimized binaries generated by the offline compiler are stored in a reusable format, called the quasi-static image, or QSI. These images persist across multiple JVM instances and allow methods to be executed from the precompiled classes. The runtime compiler reads the precompiled code images and performs validation checks and on-the-fly adaptation, or *stitching*, of the images for that JVM instance. The stitching process performs a binary-to-binary transformation of code using relocation information collected during the offline compilation. Stitching can be implemented with low runtime overhead and memory footprint. If the precompiled code cannot be reused (e.g., if a class is modified after its QSI is generated), the quasi-static compiler performs compilation at runtime to generate the binary for execution. The corresponding QSI can optionally be updated to store the new version along with its relocation information. Thus QSI's may evolve over time to reflect the changing dynamic environment.

[1] The name Quicksilver was chosen because, like quicksilver, our compiler easily changes its form in response to external conditions. Quicksilver is another name for Mercury, the Roman god of merchants and merchandise – important server applications – and is identified with the Greek god Hermes [17], who, like the Quicksilver compiler, is extremely fast.

Quasi-static compilation is useful in scenarios where class files are invariant over executions of the Java bytecodes in distinct JVM instances and where response time, performance, testability, reliability, or memory requirements are important issues. For example, a transactional server can precompile an application for a particular workload and reuse the optimized binary for future invocations for a similar workload. In particular, quasi-static compilation helps mitigate the following problems with static and dynamic compilation. The problems with dynamic compilation include:

- *Performance overhead of compilation at run-time*: The overhead of compilation is incurred every time the program is executed and is reflected in the overall execution time. Therefore, dynamic compilers tend to be less aggressive in applying optimizations that require deep analysis of the program. Quasi-static compilation can employ aggressive optimizations in an offline manner; the runtime overhead of reading the quasi-static images and adapting them for use in the JVM is quite low.
- *Testability and serviceability problems of the generated code*: Dynamic compilers that make use of runtime information about data characteristics to drive optimizations can lead to a different binary executable being produced each time the program is executed. This can create reliability problems, as the code being executed may never have been tested. The code generated by the quasi-static compiler is obtained in an offline manner and can be subjected to rigorous testing (and debugging, if necessary), before being used repeatedly in production runs. Note that the dynamic nature of Java requires that a quasi-static compiler be able to generate new code at runtime in response to events like dynamic class loading (of a class not compiled before) or changes to a class file after compilation. However, an application programmer can exercise more control over these events, if having identical binaries across executions is important.
- *Large memory footprint*: A dynamic compiler is a complex software system, particularly if it supports aggressive optimizations. Hence, it usually has a large memory footprint. The memory footprint is particularly important for embedded systems, where the memory available on the device is limited. A quasi-static compiler, however, performs optimizations during an offline compilation of the program. The memory footprint of runtime adaptation of QSI's is quite low, and by using an interpreter or a simple runtime compiler to deal with the (unexpected) case where a method needs to be recompiled, the quasi-static compiler can avoid substantially increasing the memory footprint of the application program.

Some of the problems with static compilation arise out of the dynamic nature of Java:

- *Dynamic class loading*: In general, dynamic class loading, as defined in Java [20], requires the ability to handle a sequence of bytecodes representing a class (not seen earlier by the compiler) at run time. Hence, it is impossible for a JVM to support Java in its entirety with a pure static compiler. Fully compliant static systems require an interpreter or a dynamic compiler as well. The quasi-static compiler can easily handle this situation by resorting to dynamic compilation.
- *Dynamic binding*: The code for dynamically linked class libraries may not be available during static compilation of a program, causing opportunities for interprocedural optimizations to be missed. Furthermore, the rules for runtime binding and binary compatibility in Java [20] make it illegal to apply even simple interclass optimizations – e.g., method inlining across class boundaries – unless the system has the ability to undo those optimizations in the event of changes to other classes. The quasi-static compiler can perform offline compilation of a program using an optimized dynamic compiler that employs aggressive interprocedural optimizations, and generates relocatable binaries. Binary compatibility in the presence of inter-class optimizations is ensured by recording these dependences while generating the QSI's, performing the necessary checks during program execution, and recompiling the methods if those checks fail.

We describe the design of a quasi-static compiler and present details of our current implementation in the context of the Jalapeño JVM [2, 3], being developed at IBM Research. Some of the techniques we have employed in our compiler have been used individually in other systems. However, they are unique in the context of Java, and offer tremendous advantages relative to other Java compilation strategies. In particular, our paper makes the following contributions:

- It presents a new approach to Java compilation that combines the important benefits of static and dynamic compilers. Our approach sharply reduces the runtime compilation costs and improves performance and reliability by reusing highly-optimized and more easily testable binaries.
- It introduces stitching as an approach for runtime adaptation of pre-compiled binaries to a JVM instance. It describes an implementation of stitching in the Jalapeño JVM that is efficient in both space and time.
- It presents a framework for preserving optimizations across JVM instances. This framework allows interclass optimizations (e.g., inlining) to be performed offline, with reusable code being produced for execution, without breaking binary compatibility.
- It presents experimental results, in the context of a state of the art JVM, that demonstrate the performance advantages of quasi-static compilation. Our compiler achieves a runtime compilation cost comparable with that of fast non-optimizing compilation, and delivers the runtime program performance of the highest optimization level supported by the optimizing compiler, thus delivering the highest overall performance. Relative to the better of the baseline and the optimizing compilers of Jalapeño, the overall performance is increased by 9.2% to 91.4% for the SPECjvm98

benchmarks with size 100, and by 54% to 356% for the same benchmarks with size 10.

The rest of this paper is organized as follows. Section 2 presents the design of a quasi-static compiler and an overview of its implementation in the context of the Jalapeño JVM. Section 3 describes the compilation phase when QSI's are generated, and Section 4 describes how pre-existing QSI's are used during program execution. Section 5 presents experimental results from a preliminary implementation of our compiler. Section 6 describes related work. Finally, Section 7 presents conclusions and plans for future work.

## 2. OVERVIEW OF DESIGN

In this section, we present an overview of the design of the quasi-static compiler and its implementation in the context of the Jalapeño JVM. Our basic design can be applied to any JVM.

### 2.1 Basic Principles

During the compilation of a Java `class`, a quasi-static compiler saves the executable code and auxiliary information needed to reuse that code as a quasi-static image (QSI). Each QSI is closely associated with a single class. At program execution time, the compiler tries to reuse these QSI's after verifying their validity and adapting them to the new execution environment. If, however, a QSI for a `class` does not exist or is invalid, the compiler can compile the `class` at runtime. This ability clearly distinguishes a quasi-static compiler from a purely static compiler, and allows it to support dynamic features of Java like dynamic class loading. This also explains the rationale of our approach in building a quasi-static compiler by extending a dynamic compiler rather than trying to add support for the dynamic features by extending a static compiler.

The quasi-static compiler can be configured in different ways, depending on the attributes of static or dynamic compilation that are considered important in the system. The default approach presented in this paper attaches high importance to obtaining the QSI's prior to the production runs. This corresponds more closely to the static compilation model, and enables a sharp reduction in runtime compilation cost and better testability of the code to be executed (although events like dynamic class loading may still require previously untested code to be generated and used at run time). We designate the first execution of a program as a "compilation run". The quasi-static compiler is referred to as being in the *write* mode during this run, as it generates QSI's of various classes that are instantiated. An alternative to using runtime compilation during the write mode would be to compile the program statically and generate the QSI's. During normal execution of the program, the quasi-static compiler is in *read* mode, where it tries to reuse existing QSI's (and attempts to avoid invalidation of previously generated QSI's), but does not generate new QSI's.

The quasi-static compiler can also be employed in a more dynamic environment, where the main emphasis is on amortizing the overhead of compilation over different JVM executions, rather than on obtaining similar binaries across those runs or reducing the memory footprint of the dynamic optimizing compiler. The invalidation of a previously generated QSI is then treated as purely a performance issue. An adaptive compiler, in such a JVM, could choose to recompile a "hot" method in order to take advantage of runtime characteristics during a particular execution. In this system, it is not necessary to make a distinction between the *write* and *read* modes of the quasi-static compiler. A QSI may be generated during normal execution as a new image or as a replacement for an existing QSI, and the compiler may decide at runtime whether or not to reuse an existing QSI. We shall not discuss such a dynamic environment any further in this paper.

Although some individual components of our quasi-static approach (e.g., static compilation, relocatable linking, dynamic compilation) have been used in other contexts, the combined effectiveness of these components in a system to compile and execute Java are uniquely powerful. We have made the following key design decisions that allow the full power of this approach to be exploited:

1. *Dynamic compilation as the base case.* Correct behavior in the presence of dynamic activity that invalidates the correctness of a class's semantics (that were captured in the QSI) can be obtained by using the dynamic compilation system. This simplifies the design and implementation of a quasi-static compiler. Furthermore, it allows more aggressive optimizations to be attempted because a safe fall-back position is always available at effectively the same performance as if quasi-static compilation had not been used.
2. *Use of a relocatable binary format.* This eliminates QSI invalidations resulting from changes in the execution environment that are a property of the implementation and JVM environment, but that do not alter the semantics of the class captured in the QSI.
3. *Provision for multiple method versions.* This allows specialization of generated code to accommodate different workload characteristics and different constraints on optimizations, allowing quasi-static compiled code to obtain many of the performance benefits that accrue from dynamic compilation.
4. *Code generation at a finer granularity than the whole-program.* By generating code in units smaller than a whole program, the number of QSI's that must be invalidated because of changes to class files used by the program, and the opportunities for the use of code in a QSI by many applications, are increased. Both of these reduce the overhead associated with compilation and enhance the benefits of quasi-static compilation.

### 2.2 Implementation of the Quasi-Static Compiler in Jalapeño

Our quasi-static compiler is implemented within the context of the Jalapeño JVM [2, 3]. Figure 1 gives a high-level overview of the Jalapeño JVM.

The Jalapeño JVM supports a variety of compilers. The *baseline* compiler provides fast translation of bytecode to native code, without any optimizations. The *optimizing*

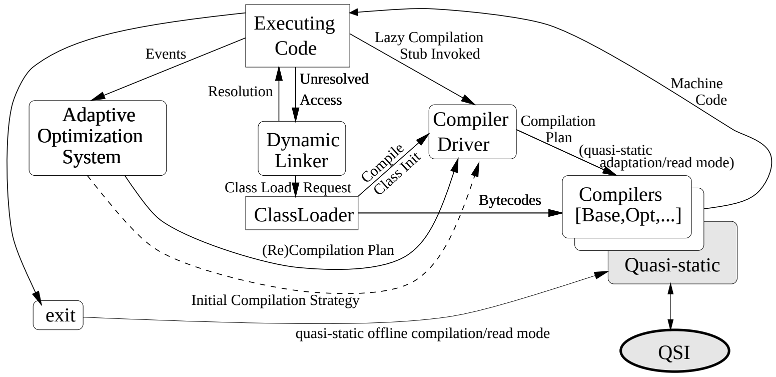

**Figure 1: The Jalapeño system including the quasi-static compiler: Overview**

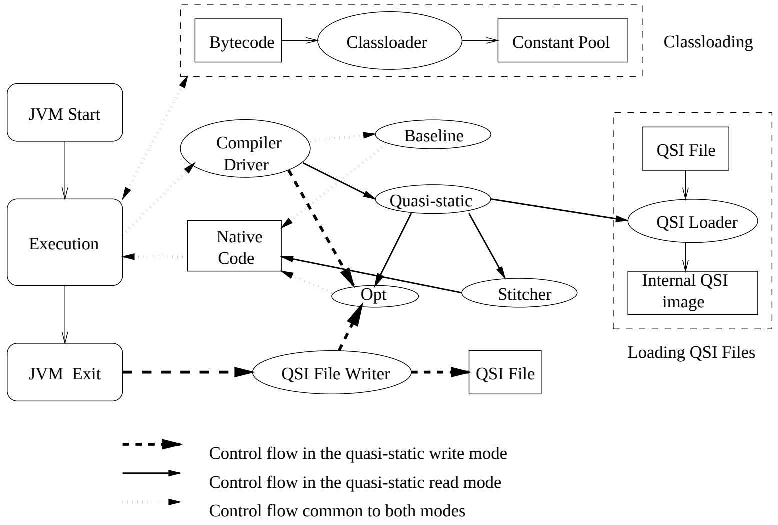

**Figure 2: The Jalapeño system including the quasi-static compiler: Details**

*(Opt)* compiler leads to the best program execution times (i.e., the time actually spent in executing the program, ignoring the runtime compilation cost), but uses more resources (both memory and time) in performing the compilation. The *adaptive* compiler [5], being developed concurrently with the quasi-static compiler, initially compiles methods using the baseline compiler, but methods whose execution is a significant part of the program execution time are recompiled using the Opt compiler. As can be seen in Figure 1, the quasi-static compiler is one of these compilers.

We use the *lazy method compilation* framework [27] of Jalapeño, in which the JVM compiles a method only when it is actually invoked, rather than compiling each method of a class when it is linked and resolved. When the Jalapeño system determines that a method needs to be compiled, it invokes the *compiler driver*. The compiler chosen by the JVM to perform runtime compilation depends on the JVM configuration. The JVM used in our system is configured to invoke the quasi-static compiler by default.

Figure 2 shows the Jalapeño components that are important to the quasi-static compiler. Ovals represent JVM components, squares are data structures, and rounded squares are phases of the JVM execution. Solid directed arcs represent read phase control flow, dashed arcs represents write phase control flow, and dotted arcs represent control flow taken by both the read and write phases. As seen in the figure, the quasi-static compiler currently uses the Opt compiler to perform compilation of the method codes during its write mode. Some of the important optimizations performed by the Opt compiler (and utilized by the quasi-static compiler) are global constant and non-null value propagation; method inlining; lock and unlock inlining; flow insensitive elimination of redundant cast checks; null check and constant sized arrays bounds checks; local bounds check elimination; and common subexpression elimination. These optimizations are discussed further in [2]. In the future, we plan to incorporate additional optimizations based on interprocedural analysis, that may be too expensive (for the Opt compiler) to perform at runtime.

During program execution (the read mode), the quasi-static compiler attempts to reuse the existing QSI's after reading their data, performing the validation checks, and stitching the binary code to the current JVM instance. As seen in Figure 2, the native code produced by the stitcher is then executed. If it determines that a method needs to be recompiled, it uses the Opt compiler (in principle, it could invoke the adaptive or baseline compiler as well).

## 2.3 Digests and QSI security models

The quasi-static compiler can use digital signatures for several purposes, including forming file names for QSI's, checking the integrity of a QSI, and determining that a QSI is the same as the one generated from a verified class file. A *digest* of the QSI file is obtained using a secure hashing function [18]. A digest of a data stream is a one-way hash function of the contents of the data stream that, with a very high probability, yields a different value if there are any changes made to the contents of the data stream [18]. We developed a preliminary implementation of our security mechanism which uses the *Secure Hashing Algorithm* (SHA) [36] to form digests. This preliminary implementation is used for measurements of digest overheads reported in Section 5, and the IBM DK1.1.8 numbers reported in this section. This preliminary implementation is not fully integrated into the Quicksilver prototype because of limitations in the Jalapeño JVM support of standard classes that make use of non-standard native code interfaces (e.g. BigInteger). Where appropriate, we describe the design of the quasi-static compiler in terms of digests while also describing what is implemented.

If a QSI is not kept in a secure area, the quasi-static compiler can ensure that it was generated by a trusted compiler from a given class file using the following procedure. In the write mode, the trusted compiler forms a digest of both the class file and the QSI, and then combines the digests. A digital signature is then computed using a cryptographically sound digital signature algorithm, (*DSA* [36] is used in our preliminary implementation), and appended to the QSI. To determine that the QSI is valid in read mode, the digest of the class file is formed and combined with the digest of the QSI (computed by excluding the stored the digital signature). The digital signature is then read from the QSI and verified using this combined digest. If either the class file or QSI has been changed, the combined signature will have changed, and verification will fail. Thus, verification makes it highly likely (to the extent allowed by the digest and digital signature algorithms) that the QSI is the QSI produced by the trusted compiler.

Although forming digests is relatively inexpensive (see Section 5 for detailed numbers) digitally signing and verifying a signature is a more expensive operation. The naive default implementation supplied with the IBM DK1.1.8 requires approximately 30–50 ms per file to perform a signature verification on our benchmarking platform (see Section 5 for details). We know of other implementations that on comparable hardware platforms can perform this operation in 10 ms [8].

The cost of signature verification can be mitigated by the following:

1. System files are typically not verified by JVM implementations. If the system files can be maliciously tampered with, then there is no reason to believe that the tampering won't also involve the files performing verification, making verification pointless. Unsigned digests can be used to detect non-malicious corruption.

2. If non-system QSI files reside on a secure server, the verification of the file can be performed once (when the QSI is received). Thereafter, the same security mechanisms used to protect the system files can be used to protect the QSI files.

3. A secure database of digest pairs (*class digest*,QSI *digest*) can be maintained, with the digest of the class and QSI being formed at class loading time, and compared to the entries in the secure database. This technique relies on the security provided by the database, rather than digital signatures, to ensure that signatures generated by the trusted compiler have not been

altered and therefor continue to accurately capture the contents of the trusted QSI file. By doing so, it is not necessary to verify a digital signature each time a class is loaded, thereby allowing verification to be done by only incurring the cost of forming digests.

4. If Java 2 security mechanisms are being enforced with digital signatures, and the class file is writable by the trusted compiler, a digest of the QSI can be placed in the class file as an annotation before signing the class file. Because the Java 2 security mechanism will verify that the class file signature is valid, there is no additional cost incurred for verifying the signature for the QSI file verification. If the class file signature verifies, it is known that the QSI digest contained within has not changed. If this digest matches the digest of the current QSI file, then the QSI file is unchanged. Thus verification of the QSI file only requires forming a digest of the QSI file, and not an additional digital signature verification.

Because of these techniques, we believe that in a server environment the overall cost of verifying the validity of class files can be kept low. Moreover, the ability to inexpensively sign messages and verify signatures has widespread and commercially important implications, and is an active research area. Therefore, the cost of performing verification will become cheaper over time, as the benefits of the aggressive analysis of programs enabled by quasi-static compilation become greater.

## 3. PROGRAM COMPILATION: GENERATION OF QUASI-STATIC IMAGES

For each class that is instantiated during the compilation run, a QSI is created to record the executable code for compiled methods in a persistent manner. The QSI is logically a part of the `classfile`. However, it is stored in a separate file with a `.qsi` suffix. The decision to store the QSI separately from the `classfile` gives the JVM greater flexibility in determining where to place the QSI. For example, the QSI of a class arriving over the network may be stored locally. Furthermore, this allows the JVM to create and update QSI's for read-only `classfiles`.

The layout of a QSI is shown in Figure 3. In our implementation, a QSI is created, just before the end of JVM execution, for each class with an Opt-compiled method. Any method in such a class that was only baseline-compiled during execution is recompiled with the Opt compiler. In the class header region for each QSI, the compiler stores information such as the environment information (JVM and OS versions, target architecture), a digital signature for the QSI and a directory area with pointers to various method code regions. For each method the compiler records additional information, such as exception tables, garbage collection (GC) maps, dependence and relocation information, as explained later in this section.

In the remainder of this section, we first describe how the correspondence is maintained between a class file and the QSI file. We then discuss how QSI's are obtained for library code. Finally, we describe how dependences on other classes, information to support speculative optimizations, and relocation information are recorded for each QSI, in addition to the binary code.

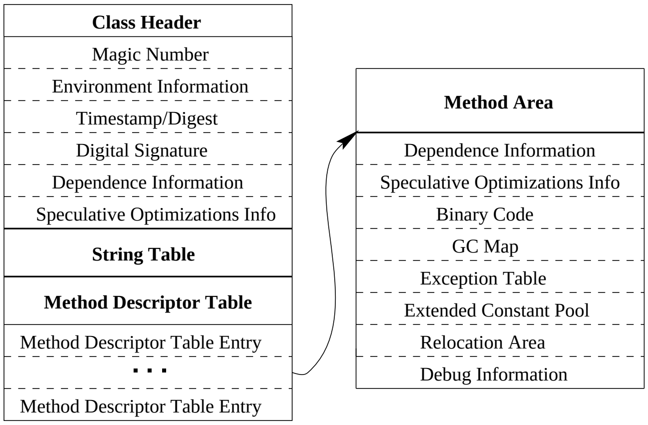

**Figure 3: Layout of the quasi-static image.**

### 3.1 Correspondence between class and quasi-static image

Given that a QSI is stored separately from a class file, the JVM needs a mechanism to uniquely identify and locate a QSI corresponding to a runtime class. In Java 2, a runtime class is uniquely identified by the pair $\langle C, D\rangle$, where $C$ is the *fully qualified name* of the class, and $D$ is the *defining class loader* of that class [29, 31]. All class loaders other than the *primordial* class loader (which is regarded as a `null` class loader) are first class Java objects.

The location in which the original class file is stored can be viewed as having two components: the *repository* containing the class, and the directory structure implied by the fully qualified name of the class. For example, consider a class with fully qualified name `MyPackage.Foo`, stored as `/jvm/Jalapeno.classes/MyPackage/Foo.class` on an AIX platform. We refer to `/jvm/Jalapeno.classes` as the repository of that class.

Consider a class $\langle C, D\rangle$ whose bytecode is stored in repository $R$. The quasi-static compiler obtains the name of the corresponding QSI as an ASCII expanded version of the 160-bit digest of the class bytecode with a `.qsi` suffix. This file is given an additional human-readable name in the form of `C_D.qsi`, which is also the name used in our current implementation.[2] The QSI is stored into the repository $f_r(R, D)$, where $f_r$ can be viewed as a set of class-to-QSI mappings defined for various class loaders. These mapping functions are chosen to be simple to compute and to not rely information about class loaders aside from the fully qualified name of their class.

Our compiler uses the local file system to store the QSI's for various classes. In principle, the QSI's may be stored in

[2] Our original design involved using the class name as the name of the QSI and storing the digest of the defining class loader in each QSI. The simplification, of using the digest of the class as the name of the QSI, was suggested by one of the anonymous OOPSLA reviewers.

other forms of persistent storage such as a database.

## 3.2 Shared class libraries

During the compilation of a program, the quasi-static compiler encounters not only classes that are unique to the program, but also classes from libraries that are shared across programs. In order to maintain a strict correspondence between a class file and a QSI file, we ensure that only a single copy of the QSI file is generated for a class. Our framework allows multiple code versions of a method to appear in a single QSI file, so, in principle, library code may be specialized for different applications. Furthermore, since compilation is done in an offline manner, the compiler has more freedom to apply potentially expensive interprocedural analysis to discover opportunities for method specialization. However, we are still in the process of implementing analysis to guide method specialization. We do currently support optimizations like inlining of methods from class libraries into methods from the user code.

## 3.3 Security attributes

If security is enforced by signing a digest of the QSI, as described in Section 2.3, a field is reserved in the QSI to hold the digital signature. We again note that this, and all features requiring digests, have not yet been implemented in our prototype.

## 3.4 Dependence information

Of particular interest from the point of view of global optimizations is the dependence information recorded by the compiler. While compiling a method `A.foo()`, for each optimization that exploits some information from a different class `B` (e.g., if a method `B.moo()` is inlined into `A.foo()`), the compiler adds class `B` to the set of classes on which the code for `A.foo()` is dependent. (This allows the quasi-static compiler to perform dependence checks during program execution to support binary compatibility, as described in the next section.) Finally, at the time of writing the QSI for class `A`, the dependences recorded for various methods of `A` are examined. The dependences that are common to all methods are recorded at the class-level, and the remaining dependences are recorded along with each method. For example, if a class `A` has two methods `foo()` and `bar()`, which are dependent, respectively, on classes {`B`,`C`} and {`B`,`D`}, the compiler would record the dependence of `A.qsi` as a whole on `B`, and additionally, the dependence of `foo()` on `C` and of `bar()` on `D`.

The dependence information of a method `A.foo()` or class `A` on another class `C` is recorded as a digest of `C` in the QSI for `A` (in either the method-level dependence area associated with `A.foo()` or in the class-level dependence area). An alternative mechanism employed in our current implementation (again, due to problems with the implementation of digests) is to simply record the name of the class `C` in the QSI. The dependence check is performed, as described in the next section, by comparing the timestamp of the QSI with the timestamp of the class on which the code in the QSI is dependent.

## 3.5 Information for Speculative Optimizations

The Jalapeño Opt compiler supports speculative inlining of virtual methods when the corresponding object pre-existence conditions are met [15]. In this case, the compiled code of a caller method needs to be invalidated if any inlined callee method is overridden due to dynamic class loading (the pre-existence conditions ensure that no stack frame rewriting needs to be done on code invalidation [15, 5]). The Opt compiler maintains an invalidation database which records the inlined callee methods along with corresponding compiled caller methods. The invalidation database is consulted by the classloader to detect if any inlined method is overridden due to the loading of a new class. If an inlined method is overridden, compiled binaries of the corresponding caller methods are invalidated.

The quasi-static compiler stores the invalidation database as a *method list* in the QSI and loads it along with the class dependence list. The invalidation database is resurrected during the *read* mode, and is used by the JVM to invalidate any previous inlining decisions due to dynamic class loading during current execution of the program. A similar mechanism can be used by the quasi-static compiler to support other speculative optimizations.

## 3.6 Relocation Information

The binary executable code, the exception table and the GC map information[3] all contain references that are specific to the execution instance in which the QSI is created. Hence, we need to record additional relocation information to allow those references to be adapted to a new execution instance at a later time. The actual information needed is closely tied to the details of the JVM implementation. Our compiler generates relocation entries for the following instructions:

- *Static field and method references*: In Jalapeño, the offsets for fields and method references are determined by the Jalapeño class loading process and the order in which classes are loaded in a particular execution of a JVM. Due to lazy class loading, classes can be loaded in a different order in different JVM instances, thus requiring different values of these offsets in different JVM instances.

- *Instance field and method references*: The offsets used for fields of objects and for virtual methods of a referenced class may change if its parent class (from which the referenced class is derived) changes between the time of writing and use of the QSI. Furthermore, the resolution status of those referenced fields and methods may change in the different JVM instances. We do not invalidate a QSI for these reasons, and instead, change the references appropriately via stitching.

- *String and floating-point constants*: These are allocated in the global table pointed to by the JTOC register. Thus these references need the proper offset in the current JVM instance.

[3] The GC map for a method invocation site contains the invoked method's id in addition to GC information. This method id is used for lazy method invocation [27] and must be stitched.

- *Runtime field and method accesses*: The Jalapeño boot image is built with several predefined Java classes that mainly consist of runtime library methods, since Jalapeño itself is written in Java [3]. Thus the boot image is a frozen core, i.e., its classes are always resolved and its offsets are always the same, independent of the run. Although it is not strictly necessary to provide relocation information for runtime references, we provide it to support version compatibility so that quasi-static images can be run with new releases of the Jalapeño boot image.

Each relocation entry contains the instruction offset in the binary code for the method, an access type, and a symbolic reference. The symbolic reference is normally expressed as an index into the *constant pool* of the given class [30]. We do not have to store the constant pool itself in the QSI, because the same constant pool is available during a later execution of the JVM (the quasi-static compiler verifies that the class has not changed), and because stitching is performed only after the class has already been loaded. Due to method inlining across class boundaries, a method may contain references that do not appear in the constant pool of its defining class. The quasi-static compiler creates an *extended constant pool* to handle relocation for references that are imported from other classes. An extended constant pool entry consists of a fully qualified name for a method or a field reference (for example, $X.foo(II)I$ for method $foo$ in class $X$ with two integer arguments and an integer return type) in a similar way to entries in the constant pool.

**Example:** Consider a method `foo()` in a class `A` that loads a static field `stats.count`). The following PowerPC load instruction is generated to access the field:

```
lwz   R1=@{JTOC +  offset of field stats.count }
```

`JTOC` is a dedicated register pointing to the table of global variables, and `offset of field stats.count` is an immediate-signed field giving the position of `stats.count` in that table. The value of the `offset` is assigned when the class `stats` is loaded. The relocation entry for the instruction contains the offset of the `lwz` instruction, the access type (static field access), and the symbolic reference to the constant pool entry in the class `A` for `stats.count`.

In addition to method variables, the compiler also generates relocation information for exception tables and GC maps. Exception tables contain references to symbolic exception types. Since Jalapeño supports type-precise garbage collection, the garbage collection (GC) maps contain information about registers and stackframe locations containing live references at program points where garbage collection can occur (See [24] for details). If new instructions are inserted, then the program points for garbage collection need to be updated (see section 4.2.2).

# 4. PROGRAM EXECUTION: REUSE OF QUASI-STATIC IMAGES

The quasi-static compiler is invoked by its parent JVM when the JVM decides to compile a method. Note that the JVM ensures that by this time, the declaring class of the method has already been loaded and resolved. The quasi-static compiler checks if a QSI file exists for that class. If the file exists, it reads data from that file and performs the steps described below to check that the QSI file is not out-of-date and has not been tampered with. If these tests show that the file is valid, the compiler accesses the code for the given method (if it exists), performs method-specific dependence tests, and adapts the code so that it can be used in the current JVM instance. If any of these steps fails, the quasi-static compiler simply compiles the method at runtime (using the Opt compiler[4] in our current Jalapeño implementation).

## 4.1 Image Loading and Validation Checks

We now describe the steps involved in loading the QSI file and performing validation checks on it.

### 4.1.1 Identification of QSI file

The quasi-static compiler first computes a digest of the bytecode for the given class $C$. It identifies the defining class loader of $C$, and also the repository from which $C$ was loaded. Based on the mapping function $f_r$ (described in Section 3.1), the compiler identifies the repository holding the QSI file for $C$. It checks if a QSI file with a name matching the digest of $C$ exists. If so, the matching of the name with the new digest for $C$ signifies that the original class has not changed since the generation of the QSI. If no such file exists, the quasi-static compiler uses dynamic compilation.

In lieu of using the digest of the class (see the discussion in Sections 2.3 and 3.1), our current implementation records the timestamp (time of last modification) of each QSI. If the timestamp of the QSI file is smaller than the timestamp of the corresponding class[5], it implies that the original class has changed since the generation (or modification[6]) of the QSI file, hence the QSI is invalidated.

### 4.1.2 Compatibility checks

First, the magic number and the execution environment information (such as the JVM version, operating system version, and target architecture) stored in the QSI file is checked to ensure that it is compatible with the current JVM.

To ensure binary compatibility while supporting global optimizations like method inlining across class boundaries, the quasi-static compiler performs dependence checks on the QSI. The recorded digest of each class $C$ on which the given QSI (as a whole) is dependent is compared with the computed digest of $C$. The check fails if any of those digests do not match. During this step, if any such class $C$ has not been already loaded by the JVM, it is loaded eagerly – in the event of an exception (or error) being thrown in

[4] We would have used the baseline compiler instead, if the target of our JVM was a memory-constrained system.

[5] The timestamp for a class file is available in Java via standard methods like `java.io.File.lastModified` and `java.util.zip.ZipEntry.getTime`.

[6] Each time an existing QSI file is modified, its original timestamp is similarly compared with the timestamp of the class file, and the QSI is invalidated if it is older than the class file.

loading this class, the exception is caught and ignored, but the dependence check fails. As discussed in Section 4.1.4, similar dependence checks are performed for finer grain dependences recorded with methods that are quasi-statically compiled.

Our current implementation performs the dependence checks by comparing the timestamps of classes on which a QSI is dependent with the timestamp of the QSI. The check fails if any class on the dependence list has a smaller timestamp than that of the QSI. As described above, a class is loaded eagerly, if it is not already loaded, to allow this check to proceed.

### 4.1.3 Security check

The validity of the QSI file can be checked by the quasi-static compiler in a variety of ways, as described in Section 2.3. This check allow the quasi-static compiler to detect corrupted or maliciously altered QSI files, and maintain the Java security requirements.

### 4.1.4 Method lookup, eager/lazy method loading, and checks

Once the validity of the QSI file has been established, a pointer to the method code is obtained by performing a lookup over the *method directory* area. The reading of method code from the QSI file may be done in a *lazy* manner (when responding to a JVM request to compile that specific method) or in an *eager* manner (when the QSI file is read for the first time, which in turn happens the first time that quasi-static compilation is attempted of *any* method in that class). We refer to these alternatives as *eager method loading* and *lazy method loading* respectively. To support lazy method loading conveniently, our implementation uses a random access file (`java.io.RandomAccessFile`) to store a QSI. Eager method loading is useful when most (or all) methods stored in the QSI are used during program execution, because sequential I/O is more efficient due to buffer prefetching. On the other hand, lazy method loading has the advantage of avoiding unnecessary I/O if relatively few methods stored in the QSI are needed during program execution. This would become particularly important when we start applying method specialization and keeping multiple method versions in the QSI. Note that even when we perform eager method loading, the stitching of method codes is still done in a lazy manner, in accordance with the lazy method compilation strategy used in the Jalapeño JVM [27].

The first item in the method code area is the information about the dependence of this method code on other Java classes. As described above for class-level dependence checks, the digests of class files corresponding to classes in this dependence list are compared with the recorded digests in the QSI file to check if the given method code should be considered invalid. (Again, as described above for class-level dependence checks, our current implementation uses timestamps of class files and the QSI for the method-level dependence checks as well.) As an optimization, the reading of the remainder of the method code can be deferred until the above validity check is performed.

## 4.2 Stitching Method Information

Stitching is the last step in the runtime adaptation of binary code. Using relocation information, various references are adapted to the new execution instance. We now describe the details of stitching in the context of our implementation in Jalapeño JVM. We discuss some interesting problems that stitching needs to handle, with the help of examples.

### 4.2.1 Stitching activities

The quasi-static compiler performs the following activities as part of stitching.

- *Updating absolute offsets*: The absolute offsets used for various references, like static field and method references, are changed using the recorded relocation information. The relocation information identifies the instruction that needs to be stitched, and also identifies the symbolic reference from which the new offset information may be generated. Note that stitching happens after the class has already been loaded and resolved. Therefore, the new table offsets to be used for field and method references are available from the mappings that the JVM maintains for the constant pools.
- *Monitoring the resolution status of method/field variables.* We say that a variable is resolved if its corresponding class has been loaded and initialized. Depending on the class loading order, some variables resolved when the binary code was generated may be unresolved when the code in the QSI is to be executed. Java semantics [30] require that the first time a variable is accessed its defining class be loaded and initialized. Thus, the compiler needs to insert extra code for an access that is resolved during the *write* mode and is unresolved during the *read* mode.
- *Monitoring pre-existing optimizations.* Some optimizations, e.g., constant propagation, produce code that may be specific to a JVM instance and hence, needs to be modified every time it is reused. In this case, instead of invalidating the code and recompiling the method, we can perform fine-grain adaptation of the code without breaking the existing optimizations (see the last example in Section 4.2.2).

Stitching requires only a single pass over the code array, even in the case where extra instructions are added, because we only add new code to the end of the method without recomputing relative branches (see Section 4.2.2). The results from our implementation in the Jalapeño JVM indicate that stitching is very fast, taking less than 65 cycles for each instruction in the method (see Section 5 for details). This speed is important because it is a runtime overhead that increases the startup time for a method, and increases the overall execution time of the program.

Stitching is different from backpatching [4] because stitching occurs at link-time (before a method is executed), while backpatching occurs during the method execution. We note that stitching of a method's code occurs only once before the method is executed for the first time. It is not necessary to update, or *backpatch* the method's code after execution begins. Hence, stitching does not suffer from dynamic linking problems which affect method backpatching on SMP machines [4].

### 4.2.2 Examples

This section presents two examples of stitching static field references to illustrate the problems that stitching can encounter. Typically, stitching only involves changing an instruction offset for a particular JVM instance. In some rare cases (which were encountered infrequently during the execution of the benchmarks described in Section 5), further processing is necessary, as described in the following examples.

Consider the following instruction to access a static field:

```
   lwz   R1=@{JTOC + offset of field}
```

At the stitching time, the original `offset` is changed to the new offset of the static field. The PowerPC load-immediate instructions use a signed 16-bit immediate field. If the new offset is too big to fit into the 16-bit immediate field used by the load instruction, we will need two instructions to perform the access:

```
   addis R1=JTOC + high(offset)
   lwz   R1=@{R1 + low(offset) }
```

The PowerPC `addis` instruction adds the content of the JTOC register with the high order bits of the offset. Then, the load instruction `lwz` adds the lower order bits of the offset. In this case, we replace the original instruction by a branch instruction to the end of the code, where we insert the two-instruction sequence followed by a branch back to the next instruction after the field access. This simplifies stitching, at the expense of introducing some inefficiency into the code. Since this case arises quite rarely, we feel this is a reasonable approach. (Some possible alternatives to this approach are: (i) insert the extra instruction in an inlined manner into the code and adjust the rest of the code accordingly, or (ii) indicate that stitching is too complicated in this case, and let the quasi-static compiler compile the method dynamically.) We may need to add new entries to the method's garbage collection map and exception table, reflecting the new instructions added at the end of the code. For example, if the original instruction is in a `try` block, the new instructions inserted at the end of the code as a replacement of the original instruction are also in that `try` block, thus we need to add a new entry in the exception table. Similarly, if the original instruction is a GC point, the new instructions contain an extra GC point at a different machine code offset.

A second complication is that a static field access may be unresolved at stitching time, requiring the static field load instruction to be replaced by the machine instruction sequence of Figure 4. The first instruction loads the offset of a table used for resolving fields. The second instruction loads an entry from that table using a unique field identification number as an index. The third instruction tests if the entry is zero (the value for unresolved fields), and the fourth instruction performs the field access. The code for field resolution is placed in line (5), but for brevity is not shown in this example. The field resolution can throw an exception if there is an error. As in the previous case, we may need to add entries to the garbage collection map and exception table for the new instruction inserted at the end of the code.

```
try:
   lwz   R1=@{JTOC + offset of field table} (1)
   lwz   R1=@{R1 + field Id}                (2)
   if (R1 == 0) goto resolve                (3)
   lwz   R1=@{JTOC + R1} // field access    (4)
   ...

resolve:
   (resolve field)                          (5)
   b     try     // goto try                (6)
```

**Figure 4: Code inserted by the Jalapeño optimizing compiler for unresolved field accesses.**

## 4.3 Preserving optimizations across JVM instances

As described in Section 2.2, the quasi-static compiler employs the Opt compiler to perform several optimizations during the write mode. The quasi-static compiler uses the QSI to propagate information necessary to preserve these optimizations across JVM instances. The optimizations, including speculative optimizations, are currently supported in two ways in the quasi-static compiler: (i) through checking dependence information and recompiling the method if there is a dependence violation, or (ii) through stitching, which performs a fine-grain adaptation at runtime. Stitching is the preferred mechanism as long as it can be done relatively inexpensively. However, dependence checking is appropriate when dependence violations are infrequent, and when stitching would be too expensive.

An interesting example of an optimization requiring a combination of the above approaches is method inlining. To preserve method inlining across class boundaries over different execution contexts, the quasi-static compiler not only records the dependence information (Section 3.4), but also creates an extended constant pool to support the stitching of inlined code and stores it in the QSI (Section 3.6). Furthermore, to support speculative inlining, it preserves the supporting data structure (the invalidation database) and stitches it such that the database can be used by the JVM during the next execution instance.

We do not foresee any problem in supporting optimizations across execution instances through a combination of stitching and dependence checking. However, the current implementation does not support the following optimizations for simplicity, although they can be supported in principle:

- Inlining of allocation code: A `new` statement is translated by the Opt compiler into a call into the allocation code; the code is later inlined. The allocation code comes in two versions, depending on whether the class was resolved at the time of the compilation of the method. If the class was not resolved, then the class resolution needs to be performed at runtime. If the quasi-static compiler inlines the allocation code, then we potentially need to replace a relatively large sequence of allocation code by different code that accounts for the class not being already resolved.

- Optimization of `static final` field references. Such references can be considered constants after the class is initialized, as in the Java code example:

  ```
  static final long time = getTime();
  ```

  In this example, `time` is assigned at class initialization time. The optimizing compiler (during the *write* mode) will replace the field references by the constant value of the field, if the class is already initialized. This may no longer be true when the QSI is loaded. This optimization could be supported in two ways, during the *read* mode:

  – Uses of the static field can be collected during the *write* mode, and stitched with the correct value during *read* mode.

  – A code dependence can be recorded on the assumption that the static field has a particular value, and the QSI invalidated at runtime if the static field takes a different value or is not initialized when the code is run.

## 5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the quasi-static compiler, we compared it with two different Jalapeño dynamic compilers: *baseline* and *Opt*. The Jalapeño baseline compiler is the simplest possible compiler – table lookups convert individual bytecodes into the corresponding sequence of machine instructions. The *Opt* compiler implements many well-known optimizations (Section 2.2). The performance of code generated using the Opt compiler has been shown to be comparable with that from the JIT used in the IBM Developer Kit for AIX, Java Technology Edition, Version 1.1.8(IBM DK1.1.8) [2].

We ran our experiments on a 4-way SMP with 166 MHz PowerPC 604e processors running AIX 4.3. For reporting the Opt compiler measurements, we used the highest compiler optimization level (-O3). In each case, the entire Jalapeño framework (i.e., the JVM runtime and compilers, which are themselves written in Java) were compiled using the same high optimization level. The Jalapeño classes were pre-compiled and stored in the JVM boot image. We used the Jalapeño JVM version with copying garbage collector (the Jalapeño JVM supports several garbage collectors [2]).

For experimental evaluation, we chose SPECjvm98, an industry-standard benchmark suite [14], and pBOB, the portable business object benchmark (version 1.2B4) [7]. pBOB follows the business model of the TPC-C benchmark and measures the scalability and throughput of the JVM. The default problem size for SPECjvm98 is 100. The pBOB application was run on 4 processors in a “minimal” mode which has a two-minute measurement period. pBOB reports maximum throughput in terms of number of transactions. In this paper, we have reported pBOB results using a different metric: we used the transaction count from the Opt compiler run as the reference value, and using the corresponding measurement duration (i.e., 120 seconds), we translated the transaction counts for the baseline and quasi-static compilers into time in seconds required to run the same number of transactions as the Opt compiled run. Using the time metric (instead of the transaction count) allows us to report the compilation times in a more meaningful manner relative to our other results.

For each benchmark, we report execution and compilation times for the baseline, Opt, and quasi-static compilers. For the baseline and Opt compilers, the runtime compilation time measures the time required to convert the Java bytecode to machine-dependent native code. For the quasi-static compiler, the runtime compilation time measures the time required to load the `qsi` files, read the relocatable binaries, perform dependence checks and adapt these binaries. It also includes the time required for compiling methods using the baseline compiler[7]. We did not find any significant performance difference between eager and lazy method loading of the QSI files. Hence, we have reported measurements only for eager method loading. We have also turned off support for speculative inlining due to a bug. We are currently working on fixing the problem.

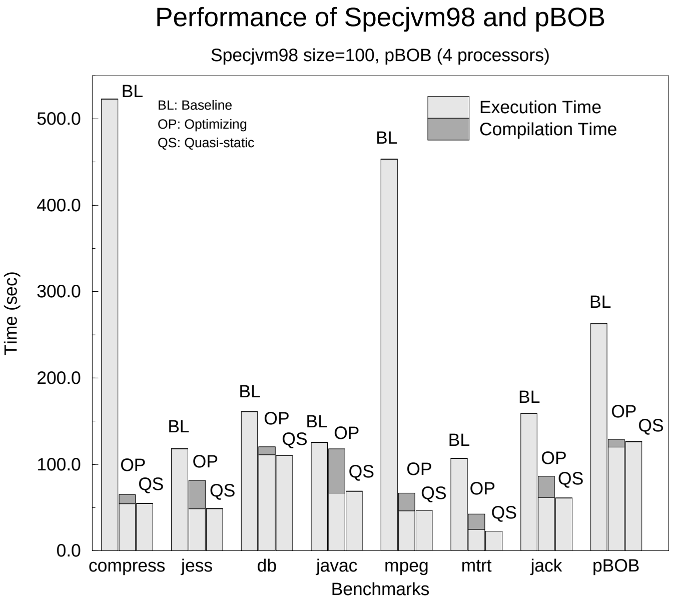

**Figure 5: Performance of the SPECjvm98 and pBOB using the Quasi-static(QS), Baseline(BL), and Opt(OP) compilers. The bottom bar denotes the execution time and the top bar denotes the compilation time.**

Figures 5 and 6 present the performance of SPECjvm98 and pBOB using the three compilers. Each stacked bar reports

[7] Our current implementation only performs quasi-static compilation of methods that are *Opt*-compiled during the *write* mode.

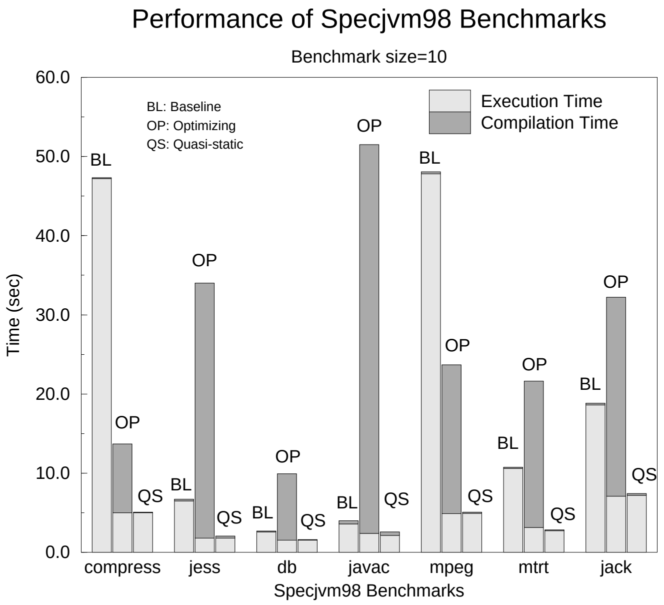

**Figure 6: Performance of the SPECjvm98 benchmarks (size=10) using the Quasi-static(QS), Baseline(BL), and Opt(OP) Compilers. The bottom bar denotes the execution time and the top bar denotes the compilation time.**

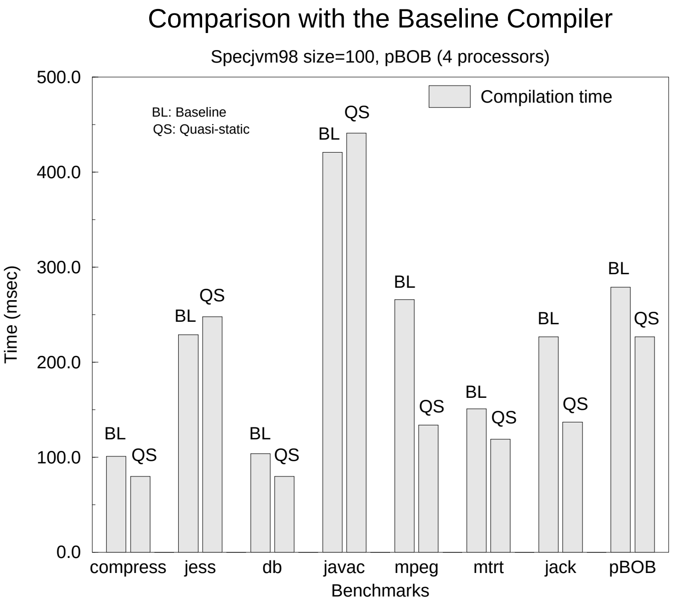

**Figure 7: Comparing Compilation costs of the Quasi-static (QS) and Baseline (BL) compilers using the SPECjvm98 benchmarks and pBOB.**

the program execution time and the runtime compilation time in seconds. The worst program execution time is observed when the baseline compiler is used. The Opt and quasi-static compilers provide comparable program execution times. In some cases the Opt compiler leads to better program performance – we believe this results from our currently turning off the optimization to inline `new` operations. In the other cases the quasi-static compiler delivers better program performance. We believe this difference is due to the smaller memory footprint of the quasi-static compiler since the Opt compiler is not invoked at runtime (resulting in fewer garbage collections and reduced memory traffic). The quasi-static compiler obtains the best overall execution times (counting both program execution and runtime compilation) in all cases. Further, the compilation cost of the Opt compiler is the highest among the three compilers. The impact of compilation cost is more pronounced for the SPECjvm98 benchmarks with the smaller size of 10 (Figure 6). In all cases, the compilation costs of the baseline and quasi-static compilers are negligible when compared to that of the Opt compiler. The runtime compilation cost of the quasi-static compiler, when compared to that of the Opt compiler, is lower by a factor of 104 to 158 for SPECjvm98. Further, relative to the *better* of the baseline and the optimizing Jalapeño compilers, the overall performance (taking into account both runtime compilation and execution costs) is increased by 9.2% to 91.4% for the SPECjvm98 benchmarks with size 100, and by 54% to 356% for the (shorter running) SPECjvm98 benchmarks with size 10.

Figure 7 presents the comparison between the compilation costs of the quasi-static read mode and the baseline compiler (which are too small to show up in Figures 5 and 6). For most benchmarks, the quasi-static compilation cost is roughly equivalent to that of the baseline compiler. As discussed earlier, the Jalapeño baseline compiler supports no optimizations and requires the least compilation cost of the dynamic Jalapeño compilers. Figures 5, 6, and 7 demonstrate that our implementation of the quasi-static compiler achieves program performance comparable to the most aggressive optimizing compiler while keeping the runtime compilation cost extremely low – comparable to the fastest (baseline) compiler.

Table 1 gives a breakup of the compilation cost during the read mode of the quasi-static compiler. For each program, the first column shows the number of QSI files that were used, and the next two columns show the number of methods that were compiled using the baseline and quasi-static compilers. The baseline compiler is used for compiling methods before the quasi-static compiler is initialized. The next four columns show the respective costs of various compilation components: the cost of baseline compilation, the I/O cost associated with accessing the QSI files, the cost of performing dependence checks, and the cost of adapting the relocatable binaries, which includes stitching the binary code and auxiliary data such as GC maps and exception tables. For each program in SPECjvm98, the same number of methods are baseline-compiled, before the quasi-static compiler is initialized. The table illustrates that the runtime quasi-static compilation times, while low in absolute terms, are dominated by I/O times. Tables 2 and 3 present more detailed measurements of the I/O and stitching costs.

Table 2 presents I/O costs associated with the quasi-static compilation. For each benchmark, we present the number of QSI files that were opened during the read mode, along with the minimum, maximum and average file sizes in bytes.

**Table 1: Quasi-static compilation costs.**

| Program | # File | # Methods | | Quasi-static Compilation Costs(ms) | | | |
|---|---|---|---|---|---|---|---|
| | | Baseline | QS | Baseline | I/O | Depend. | Stitch |
| compress | 43 | 34 | 129 | 12 | 51 | 11 | 6 |
| jess | 169 | 34 | 487 | 12 | 183 | 33 | 20 |
| db | 37 | 34 | 137 | 12 | 51 | 11 | 6 |
| javac | 172 | 34 | 743 | 12 | 334 | 56 | 39 |
| mpegaudio | 69 | 34 | 255 | 12 | 92 | 18 | 12 |
| mtrt | 59 | 34 | 208 | 12 | 81 | 16 | 10 |
| jack | 77 | 34 | 246 | 12 | 93 | 18 | 14 |
| pBOB | 82 | 32 | 422 | 12 | 163 | 29 | 23 |

**Table 2: I/O costs associated with the quasi-static compilation.**

| Program | # Files | File size(bytes) | | | I/O Costs (ms) | | | |
|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Avg | Total | File | Class | Method |
| compress | 43 | 178 | 31460 | 4254 | 51 | 7 | 22 | 22 |
| jess | 169 | 144 | 31460 | 2467 | 183 | 39 | 73 | 71 |
| db | 37 | 178 | 31460 | 5127 | 51 | 6 | 21 | 24 |
| javac | 172 | 126 | 44196 | 4328 | 334 | 39 | 182 | 113 |
| mpegaudio | 69 | 178 | 111502 | 5758 | 92 | 13 | 36 | 43 |
| mtrt | 59 | 152 | 31460 | 4433 | 81 | 10 | 38 | 33 |
| jack | 77 | 144 | 60892 | 5140 | 93 | 13 | 36 | 44 |
| pBOB | 82 | 144 | 38338 | 6396 | 163 | 13 | 90 | 60 |

In Table 2, the "Total" column denotes the overall I/O time for each application, the "File" column denotes the aggregate time required for opening the files, the "Class" column denotes the aggregate time required to read the class header, string table, and method table, and the "Method" column denotes the aggregate time required to read the relocatable method binaries along with relocation tables, GC maps, and exception tables. In our experiments, the number of QSI files varied from 37 (db) to 172 (javac). The file sizes varied from 126 bytes to 111502 bytes; the maximum average file size being 6396 bytes. For all applications, the I/O time was dominated by the time required to fetch the class data. The class I/O time includes time required for reading the class header, extended constant pool, string table and method descriptor table (see Figure 3). Note that these results were achieved only after extensive tuning of I/O, and in particular, by using unformatted I/O wherever possible.

Table 3 shows further details about stitching. Column (a) of Table 3 shows the aggregate code size (i.e., the number of machine instructions) for each benchmark. Column (b) shows the total number of relocation entries for each program. The third column shows the percentage of instructions that needed stitching. Finally, the last column presents the stitching cost per machine code instruction. Table 3 illustrates that the amount of relocation required for stitching the code is a small fraction of the code size – the maximum being 15.4% for javac (the average is around 8%). Further, the cost of stitching per machine instruction (not per instruction stitched) in the original code is very small – it varies from 0.17 microseconds (mpegaudio) to 0.39 microseconds (javac). On a 166 MHz PowerPC 604e processor, this amounts to less than 65 cycles per instruction. The total cost of runtime quasi-static compilation per instruction is also very small – it varies from 1.7 microseconds (mpegaudio) to 4.27 microseconds (javac).

Table 4 presents the times to create digests for the class and QSI files for each benchmark. Columns of the table describe the number of application (non-system) class and QSI files for which digests were formed, the total number of bytes in the class files the time taken to form the class file digests, the total number of bytes in the QSI files, and the time taken to form the QSI file digests. For reasons given in Section 2.3, only the cost of forming digests for non-system files has been counted. Because the standard digest classes are not yet supported in the Jalapeño JVM, these numbers have been collected under the IBM DK1.1.8, running on the same machine as the other benchmark results. The numbers were collected using the default SHA [36] `digest` implementation in the IBM DK1.1.8 `java.security.MessageDigest` class. Because the ability to form digests is not currently part of the Quicksilver prototype, these times are not included in the other benchmark results.

Although the cost of forming digests exceeds the combined cost of checking dependences via timestamps and stitching, it is still far less than the compilation costs of the Opt compiler (see Figure 6). Moreover, because we are using the default implementation provided with the IBM DK1.1.8 distribution, rather than the highly optimized implementations available from various providers for an additional fee, and because much of the computation occurs in native code and is JVM independent, these numbers represent upper bounds on overheads incurred from forming digests.

**Table 3: Evaluating the stitching process: Comparing code and relocation table sizes and measuring the cost of stitching per machine instruction.**

| Program | Code Size(a)<br>(# instr) | Reloc. Size(b)<br>(# reloc instr) | (b/a)<br>(%) | Total Cost<br>(ms) | Cost/intr<br>(cycles/instr) |
|---|---|---|---|---|---|
| compress | 26518 | 2386 | 9.0 | 6 | 38 |
| jess | 56495 | 6115 | 10.8 | 20 | 59 |
| db | 29025 | 2538 | 8.7 | 6 | 34 |
| javac | 100359 | 15440 | 15.4 | 39 | 65 |
| mpegaudio | 71227 | 5233 | 7.3 | 12 | 28 |
| mtrt | 37041 | 4219 | 11.4 | 10 | 45 |
| jack | 63236 | 6509 | 10.3 | 14 | 37 |
| pBOB | 81599 | 9234 | 11.3 | 23 | 47 |

**Table 4: Costs of creating digests for the application class and QSI files using the IBM DK1.1.8.**

| Program | # class | class Files | | QSI Files | |
|---|---|---|---|---|---|
| | | size (bytes) | digest time (ms) | size (bytes) | digest time (ms) |
| compress | 24 | 98623 | 20 | 120315 | 25 |
| jess | 146 | 441838 | 89 | 315476 | 80 |
| db | 16 | 92194 | 18 | 121836 | 18 |
| javac | 143 | 548659 | 108 | 636865 | 137 |
| mpeg | 50 | 192451 | 40 | 341798 | 72 |
| mtrt | 38 | 139085 | 20 | 190397 | 41 |
| jack | 57 | 204401 | 36 | 341655 | 67 |
| pBOB | 41 | 228095 | 47 | 361126 | 75 |

## 6. RELATED WORK

Several groups have developed dynamic compilers for Java, such as the IBM JIT [39], Sun HotSpot [22], Microsoft JIT [13], and the OpenJIT [34]. Several static compilers for Java are also available, such as HPCJ [37], JOVE [26], Marmot [19], TowerJ [40] and BulletTrain [23]. Our work is different, in that it attempts to incorporate the advantages of static compilation into a dynamic compilation system for Java. The IBM AS/400 JVM [35] seems to be the closest to our efforts. The AS/400 JVM supports execution of statically precompiled code (appearing in the form of a *Java program object*, which is similar to a dynamically linked library), with fallback to the JIT compiler. In contrast, our Quicksilver compiler generates code with more efficient, direct, references to fields and methods, and relies on stitching to adapt the precompiled code image to a new execution instance.

Azevedo, Nicolau, and Hummel [6] have developed the AJIT compiler. The AJIT compiler annotates the byte-code with machine independent analysis information that allows the JIT to perform some optimizations without having to dynamically perform analysis. This system differs from ours in that program transformation and code generation still occur at application execution time.

*DyC* [21] is a selective dynamic compilation system for C which shares some of our goals. DyC reduces the dynamic compilation overhead by statically preplanning the dynamic optimizations. Based on user annotations that identify variables with relatively few runtime values, it applies partial evaluation techniques to partition computations in regions affected by those variables into static and dynamic computations. DyC also supports automatic caching of dynamically compiled code. Many other systems, such as Tempo [12] and Fabius [28] support a similar staging of optimizations, triggered by user annotations. In contrast to these systems which target C and ML, we target the Java language, which presents new challenges. Our work has so far focussed on building the basic system which can support a wide range of optimizations applied by a sophisticated dynamic compiler with the attendant advantages, like compilation overhead, of a static compiler. In the future, we plan to support code specialization, although without any user annotations.

Recently, researchers have proposed techniques to detect situations where analysis based on a "closed world" view of the program continues to be valid for the "open world" scenario, where dynamic class loading may take place. Detlefs and Agesen [15] present techniques to identify opportunities for inlining virtual methods that are not affected by dynamic class loading. Sreedhar et al. [38] present *extant analysis*, which is a general framework to allow optimizations, that remain valid in the presence of dynamic class loading, to be applied based on a static analysis of the program. This work is complementary to the work described in our paper. We plan to build upon these techniques while performing interprocedural analysis in the quasi-static compiler.

Several systems for selective recompilation of programs keep track of dependences between modules to reduce the overhead of recompilation [9, 1, 10]. We apply these techniques in a different context, and also supplement them with our stitching techniques that allow previously generated code to be fixed up, rather then invalidated, based on new runtime conditions.

The Forest project [25] at Sun Labs is developing *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. The quasi-static compiler does not store the state of a Java application, rather it stores and reuses method binaries needed for the Java application.

Necula has proposed the notion of *proof-carrying code* for a simple language to allow a host system to determine if it is safe to execute a binary program supplied by an untrusted source [33]. In the context of our work, the JVM executes an adapted form of the binary executable produced earlier by the *same* JVM, and not by an untrusted source. Hence, we use the simpler approach of using digital signatures to verify that the code produced by the trusted source has not been tampered with. In the future, when the proof-carrying code methodology is well-developed for languages like Java, it should be possible to integrate it into the quasi-static compiler.

## 7. CONCLUSIONS

For performance reasons, JVMs usually rely on compilation instead of interpretation for executing the Java bytecodes. Two common compilation approaches – dynamic and static – have their drawbacks and benefits. Since static compilation is not Java-compliant, most JVMs tend to use dynamic compilation, even though the compilation time can adversely affect the overall program execution time.

In this paper, we presented a new approach for compiling Java bytecodes: quasi-static compilation. It performs offline optimizing compilation and stores the relocatable binaries into persistent code images. During program execution, it reuses previously generated code images after performing on-the-fly binary adaptation to the new JVM context, and using dynamic compilation if the relocatable binaries cannot be used. We presented details of our implementation of the quasi-static compiler using the Jalapeño JVM. We used the Jalapeño dynamic optimizing compiler for offline compilation. We evaluated our implementation using the SPECjvm98 benchmark suite and the pBOB benchmark. We demonstrated that our quasi-static compiler provides the optimization benefits of the highest optimization level (supported by the optimizing dynamic compiler) while incurring compilation costs comparable to those of the fastest, non-optimizing (baseline) dynamic compiler, thus delivering the best overall performance. The runtime compilation cost of the quasi-static compiler is lower by a factor of 104 to 158 for SPECjvm98 relative to that of the optimizing compiler. Relative to the better of the baseline and the optimizing Jalapeño compilers, the overall performance is improved by 9.2% to 91.4% for the longer running version, and by 54% to 356% for the shorter running version of the benchmark suite.

In the future, we shall investigate more general strategies for recording dependence information to explore the trade-offs between the overhead of dependence checking and the likelihood of QSI invalidation during program execution. For example, the compiler may move a dependence item shared by the important (but not all) methods to a class-level dependence, to reduce the overhead of dependence checking. This comes at the expense of possibly invalidating the entire QSI file rather than invalidating just the quasi-static code for a single method. On the other hand, the compiler may keep dependence information at a finer granularity (e.g., method-to-method dependence) to reduce the chances of invalidating a QSI, at the expense of increased overhead of dependence checking.

We are currently extending the quasi-static compiler to support global program optimizations that are often not supported by dynamic compilers due to inhibitive compilation costs. We are building upon the techniques described in [15] and [38] to support optimizations in presence of dynamic class loading. We plan to support method specialization based on partial evaluation techniques. As we incorporate optimizations performed exclusively by the quasi-static compiler that lead to execution time improvements, we expect to show even greater benefits from the quasi-static compilation approach.

## 8. ACKNOWLEDGMENTS

The help of the Jalapeño compiler and runtime groups has been invaluable in implementing our prototype. We would like to thank John Field, Vugranam Sreedhar, Chet Murthy, and the anonymous referees for their valuable comments. We would also like to thank Donna Dillenberger, Lee Nackman, Pratap Pattnaik, and Marc Snir for supporting this work.

## 9. REFERENCES

[1] R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[2] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. Russel, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–193, 2000. Java Performance Issue.

[3] B. Alpern, C. Attanasio, J. Barton, A. Cocchi, S. Flynn Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, Denver, Colorado, October 1999.

[4] 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.

[5] 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.

[6] A. Azevedo, A. Nicolau, and J. Hummel. An annotation aware Java virtual machine implementation. In *Proc. ACM SIGPLAN 1999 Java Grande Conference*, June 1999.

[7] S. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000. Java Performance Issue.

[8] Peter Buhler and Thomas Eirich. private communication.

[9] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[10] C. Chambers, J. Dean, and D. Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *Proc. 17th International Conference on Software Engineering*, Seattle, WA, April 1995.

[11] Hewlett-Packard Company. Chai products. `http://www.hp.com/emso/products/chaivm`.

[12] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 145–156, January 1996.

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

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

[15] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proc. 13th European Conference on Object-Oriented Programming*, pages 258–278, 1999.

[16] D. Dillenberger, R. Bordawekar, C. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. Oliver, F. Samuel, and R. St. 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.

[17] Encyclopedia Britannica Online. Information available via search at `http://www.eb.com`.

[18] Proposed Federal Information Processing Standard for Secure Hash Standard. *Federal Register*, 57(21):3747–3749, 31, January 1992.

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

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

[21] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An Evaluation of Staged Run-Time Optimizations in DyC. In *SIGPLAN'1999 Conf. on Programming Language Design and Implementation*, pages 293–304. ACM SIGPLAN, May 1999.

[22] The Java Hotspot Performance Engine Architecture. `http://java.sun.com/products/hotspot/whitepaper.html.`

[23] NaturalBridge Inc. BulletTrain optimizing compiler and runtime for JVM bytecodes. `http://www.naturalbridge.com/bullettrain.html`.

[24] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

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

[26] Jove. Jove, super optimizing deployment environment for Java. `http://www.instantiations.com/javaspeed/jovereport.htm`.

[27] C. Krintz, D. Grove, D. Lieber, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. Technical Report CSE2000-0648, University of California, San Diego, April 2000.

[28] M. Leone and P. Lee. Dynamic specialization in the Fabius system. In *ACM Computing Surveys*, pages 30(3es):23–28, September 1998.

[29] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, October 1998.

[30] T. Lindholm and F. Yellin, editors. *The Java Virtual Machine Specification, Second Edition.* AddisonWesley, 1998.

[31] Sun Microsystems. Java2 Platform, Standard Edition Documentation. http://java.sun.com/docs/index.html.

[32] Sun Microsystems. EmbeddedJava$^{(TM)}$ application environment. `http://java.sun.com/products/embeddedjava/`.

[33] G. Necula. Proof-carrying code. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Paris, France, January 1997.

[34] 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.

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

[36] B. Schneier. *Applied Cyptography: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, 1996.

[37] V. Seshadri. IBM High Performance Compiler for Java. *AIXpert Magazine*, September 1997. `http://www.developer.ibm.com/library/aixpert`.

[38] V. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *SIGPLAN'2000 Conf. on Programming Language Design and Implementation, To appear.* ACM SIGPLAN, 2000.

[39] 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. Java Performance Issue.

[40] Tower Technology Corporation. TowerJ3 - a new generation native Java compiler and runtime environment. http://www.towerj.com/products/whitepapergnj.shtml.

[41] B.-S Yang, S.-M Moon, S. Park, J. Lee, S. Lee, J. Park, Y Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proc. 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.