Open Runtime Platform: Flexibility with Performance using Interfaces

Michal Cierniak
Intel Labs
2200 Mission College Blvd.
Santa Clara, CA 95052
michal.cierniak@intel.com

Brian T. Lewis
Intel Labs
2200 Mission College Blvd.
Santa Clara, CA 95052
brian.t.lewis@intel.com

James M. Stichnoth
Intel Labs
2200 Mission College Blvd.
Santa Clara, CA 95052
james.m.stichnoth@intel.com

ABSTRACT
According to conventional wisdom, interfaces provide flexibility at the cost of performance. Most high-performance Java virtual machines today tightly integrate their core virtual machines with their just-in-time compilers and garbage collectors to get the best performance. The Open Runtime Platform (ORP) is unusual in that it reconciles high performance with the extensive use of well-defined interfaces between its components. ORP was developed to support experiments in dynamic compilation, garbage collection, synchronization, and other technologies. To achieve this, two key interfaces were designed: one for garbage collection and another for just-in-time compilation. This paper describes some interesting features of these interfaces and discusses lessons learned in their use. One lesson we learned was to selectively expose small but frequently accessed data structures in our interfaces; this improves performance while minimizing the number of interface crossings.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors – run-time environments, code generation, memory management (garbage collection); D.2.2 [Software Engineering]: Design Tools and Techniques – modules and interfaces.

General Terms

Keywords
Java, virtual machine, JVM, dynamic compilation, just-in-time compilation, garbage collection, modular components, interfaces, interface design.

1. INTRODUCTION
The Open Runtime Platform (ORP) was developed by Intel’s Microprocessor Research Lab to support a wide range of experiments in different Java virtual machine techniques for dynamic compilation, garbage collection, synchronization, and threading. Over the past five years, researchers at Intel and elsewhere have used ORP to conduct a number of virtual machine experiments [5][6][7][8][9][11][18]. At least two different garbage collectors and eight different just-in-time (JIT) compilers have been developed and integrated with ORP. The public open source distribution of ORP [1] contains two IA-32 Java JIT compilers. There are three other JIT compilers for ORP in active use or under development in our lab.

ORP uses interfaces to support this flexibility. One interface supports JIT compilation and a second supports garbage collection (GC). As shown in Figure 1, these cleanly separate the core virtual machine (VM) from particular garbage collectors or JIT compilers, and enable modularity and substitutability. For example, different JITs may be plugged in at either build time or run time (as dynamically linked libraries). In addition, multiple JITs with different tradeoffs between compilation time and code quality may be used simultaneously.

This approach to VM design has a number of advantages over a tightly coupled system:
• A researcher can build and use a new garbage collector or JIT without modifying, or even necessarily understanding, the rest of the system. All that is required is to implement the necessary routines in the appropriate interface.
• Similarly, the core VM can be modified and restructured without modifying the JIT or GC implementations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JGI’02, November 3–5, 2002, Seattle, Washington, USA.
Copyright 2002 ACM 1-58113-599-8/02/0011…$5.00.
• Different organizations can release the core VM and the JIT or GC components separately, on different release cycles.

• The approach may allow for rapid development of new JIT and GC components to exploit new features and improvements in microprocessors, without requiring changes to the core VM. As an example, a new GC implementation could be released to exploit Intel’s HyperThreading™ capability, which effectively supports multiprocessing using a single CPU core.

The advantages of JIT and GC interfaces have been noticed before, and multiple systems have been built to support interfaces for the JIT, GC, or both [19][24]. Most of these systems used a single JIT or GC implementation specifically developed to support other system components. However, the Sun Classic VM [19] was successfully used with multiple JIT compilers [20][21], many of which were developed by groups outside of Sun. Also, the GC interface [22] of Sun’s Research VM supports a variety of different garbage collectors [28].

The challenge in building such a VM is to ensure that performance is not lost due to calls to interface routines. Performance can also suffer because the core VM and other components are not able to take advantage of knowledge about the data structures and algorithms used internally by the other components. In fact, there has been a trend in recent JVMs toward more closely integrating JITs and garbage collectors with the core JVMs to reduce overhead. ORP demonstrates that high performance is possible despite use of separate components. Its interfaces are a tradeoff between abstract and concrete: being abstract enough to allow modifications to the implementation of the components on one side of the interface without modifying anything on the other side, and being concrete enough to expose what is needed to achieve high performance.

Prior VM work using interfaces has tended to focus either on high performance or on using interfaces to simplify research experiments. That has been exemplified by the high-performance commercial JVMs that do not provide interfaces even if earlier versions of those JVMs provided them. This seems to be an admission that high performance is possible only through tight integration of all parts of the runtime systems into a large monolithic JVM. Our contribution is to demonstrate that it is possible to provide flexible interfaces that are as convenient to use as any interfaces we know of without sacrificing performance. In fact, others have shown that ORP has performance as good as the best commercial JVMs [23].

The remainder of this paper is organized as follows. Section 2 gives a basic overview of ORP and its key data structures. Sections 3, 4 and 5 describe ORP’s JIT and garbage collection interfaces. Section 6 details some of our experience using these interfaces. Section 7 evaluates ORP performance for common Java benchmarks. Finally, Sections 8 and 9 present related work and concluding remarks.

2. OVERVIEW OF ORP

ORP is a high-performance VM that features exact generational garbage collection, fast thread synchronization, and multiple JITs, including highly optimizing JIT compilers. All code is JIT-compiled; there is no interpreter. Use of a new JIT may be specified on the ORP command line.

ORP is released under an open source license [1]. It is written in about 250,000 lines of C++ and a small amount of assembly code.

It compiles under Microsoft VC++ 6.0 and GNU g++, and runs under Windows (NT/2000/XP), Linux, and FreeBSD. ORP supports both IA-32 [16] and IPF (IA-64, [17]) CPU architectures, although the public release currently only supports IA-32. ORP uses the GNU Classpath library [3], an open source implementation of the Java class libraries.

ORP is distributed with two JIT compilers for Java. The Simple Code Generator (known as the O1 JIT [5]) produces code directly from the JVML instructions [13] without applying complex optimizations. Its optimizations include strength reduction, load-after-store elimination, and simple versions of common subexpression elimination (CSE), array bounds check elimination, and register allocation.

The Optimizing Compiler (known as the O3 JIT) converts JVML instructions to an intermediate representation (IR) that can be used for more aggressive optimizations. Besides the optimizations performed by the O1 JIT, O3 applies inlining, global optimizations (e.g., copy propagation, dead code elimination, loop transformations, and constant folding), as well as more complete implementations of CSE and array bounds check elimination.

As shown in Figure 2, ORP can run in a mode that uses both the O1 and O3 compilers. In this mode, when a method is invoked for the first time, ORP uses O1 to compile the method in a way that instruments the generated code with counters that are incremented on every method call and on every back edge of a loop. When a counter reaches a predetermined threshold, ORP invokes O3 to recompile the method. The dynamic recompilation approach allows ORP to avoid the cost of expensive optimizations, while applying those optimizations to the methods where the payoff is likely to be high. It also provides the O3 JIT with profiling information that can help guide the optimizations.

The default ORP garbage collector is the train algorithm [10]. ORP may also be built [4] to implement just part of the train functionality, reducing train collection to one of the following:

• Two-semispace copying collection
• Mark and sweep fixed collection
• Generational copying collection

2.1 Basic Data Structures

ORP has a data structure for every class, field, and method loaded during its execution. The Class data structure holds fields such as a pointer to the common vtable (virtual method table) for the Java
class, specific attributes of the array if this is an array class, and referenced to inner classes, static initializers, and finalizers. Every Java field is represented internally by a Field data structure. This contains information needed for reflection (class, type, name, and constant value), as well as internal ORP information such as an instance field’s offset or a static field’s address. Similarly, every Java method is represented by a Method data structure.

These data structures are used only inside the core VM. The JIT and GC modules operate only on opaque data types, and use the functions described in Section 3 to query and update the VM data structures. However, all ORP modules share some basic information about object layout. The first two fields of every instance of a Java class are the following:

```c
typedef struct Java_java_lang_Object {
    VTable *vt;
    POINTER_SIZE_INT obj_info;
} Java_java_lang_Object;
```

Knowledge about these fields is shared between the core VM, each JIT, and each garbage collector. No other fields of the Object data structure are exposed outside the core VM. The first field is a pointer to a VTable structure. The VM creates one VTable structure for each class,1 and stores enough class specific information to perform common operations like virtual method dispatch. It is also used during GC, where it supplies information such as the instance’s class and the offset of each reference stored in the instance. The second field, obj_info, is 4 bytes long on IA-32 and 8 bytes on IPF, and is used in synchronization and garbage collection. This field also stores the instance’s default hashcode.

Class-specific instance fields immediately follow these two fields. Garbage collectors and JIT compilers also share knowledge about the representation of array instances, although the specific offsets at which the array length and the first element are stored are determined by the core VM, and are available to the GC and JIT via the VM interface (see Section 3).

Another small but important piece of shared information is the following. The GC component is expressly allowed to use a portion of the Vtable to cache frequently used information to avoid runtime overhead (see Section 5.1). This cached information is private to the given implementation of GC and is not accessed by other ORP components. Apart from the basic assumptions about object layout and this Vtable information, all interaction between major ORP components is achieved through function calls.

### 2.2 Overview of JIT Compilation

When ORP loads a class, new and overridden methods are not immediately compiled. Instead, ORP initializes the vtable entry for each of these methods to point to a small custom stub that causes it to be JIT-compiled at its first invocation. After the JIT compiles the method, the VM iterates over all vtables containing an entry for that method, and replaces the pointer to the original stub with a pointer to the newly compiled code.

ORP allows many JITs to coexist within the same VM by defining a JIT interface. Each JIT must implement this interface, which is described in more detail below. A JIT can be either linked statically or loaded dynamically from a dynamic library.

A method may be compiled in one of several ways by ORP. Normal Java methods are compiled by one of the JITs, after which the code field in ORP’s data structure representing the method points directly to the JIT-generated code. For native methods, there is still the notion of compilation: the method’s code field points to a VM-generated machine code wrapper sequence that arranges to call the corresponding native routine.

Optionally, a background thread can be created to support recompiling methods concurrently with the rest of the program.

### 2.3 Overview of Garbage Collection

The garbage collector is responsible both for allocation of new objects and for collection of dead objects. Typically, when the heap is exhausted, garbage collection proceeds by stopping all Java threads, determining the set of root references [15], performing the actual collection, and then resuming the threads. A garbage collector relies upon the core VM to enumerate the root set. The core VM enumerates the global references and thread-local references in the runtime data structures and then it uses the JIT interface to signal the JIT to enumerate the local references in each frame of each thread stack (Section 5 describes in more detail the elements of the root set).

ORP defines an interface for garbage collection. The core VM uses the GC routines, for example, to allocate new objects, and to request that the heap be garbage collected when the application issues the corresponding request through the standard Java API. Two different garbage collectors have been implemented. The first uses the train algorithm and is the default ORP collector. It can be configured to implement several different GC algorithms. A second experimental collector, Sapphire [8], supports concurrent GC.

### 3. VM Common Interface

In the following two sections, we give a detailed description of ORP’s two key interfaces, namely JIT compilation and garbage collection. In this section we begin by describing the high-level architecture of ORP’s interfaces, and the part of the VM interface that is shared by both the JIT and the GC modules.

The VM interface contains a large set of internal reflection functions that are used to query various properties of the class, method, field, and object data structures. There are functions, for example, to return the name of a class given a class handle, to resolve a field and return a field handle given a class handle and an integer index, to return the address of a static field, and to return the offset to the array length field or the first array element.

These functions are called extensively during JIT compilation, and a number are used during GC. We refer to this part of the interface as the VM common interface, as illustrated in Figure 1.

The VM interface also includes a number of JIT-specific functions, intended for use only by the JIT compilers. Some of these are provided for querying VM-provided data structures, similar to the VM common interface. A few methods are provided to allocate memory for code, data, and JIT-specific information. The VM allocates this memory, rather than the JIT,
which allows the space to be reclaimed when it is no longer needed (however, ORP does not currently implement garbage collection of methods). Other JIT-specific functions query the exception information provided in the Java class files and set the exception information for generated code. In ORP, the structure of the exception information in JIT-compiled code is similar to that in the class files [13]. However, the JIT is not required to generate code with the same number of exception handlers as the corresponding Java method — compiler optimizations may eliminate or add exception handlers.

ORP supplies a JIT with a large number of runtime helper functions that can be called from JIT-compiled code. Examples of these are functions to allocate a new object or array, throw an exception, perform a `checkcast` or `instanceof` operation, call `monitorenter` or `monitorexit`, and perform a GC write barrier. Most of these functions are implemented directly by the VM, but the GC-related calls are forwarded by the VM to the appropriate interface call in the garbage collector.

The VM interface also provides a set of functions that are intended only for a garbage collector. The most important of these are used for the root set enumeration and for stopping and resuming all threads. At this time, ORP does not provide a public interface for concurrent garbage collectors. However, one concurrent collector [8] has been implemented in ORP, and based on this experience, we are working on defining a root set enumeration interface for concurrent GC. ORP currently provides the two functions `orp_enumerate_root_set_all_threads()` and `orp_resume_threads_after()` to implement stop-the-world collections.

4. JIT INTERFACE

4.1 Description of the Interface

ORP defines the JIT interface that can be used to plug in a JIT without any knowledge of the details of ORP’s implementation. The JIT interface consists of a set of routines that a JIT is required to export and a set of routines exported by the core VM. Our JIT interface allows the core VM to support multiple JIts. The interface is CPU-independent and is used by both our IA-32 and IPF JIT compilers.

One obvious routine in the JIT interface supports requests to compile a method. The JIT interface also includes some not-so-obvious procedures that the JIT must export, which implement functionality traditionally thought of as being part of the core VM. These include routines to unwind a stack frame and to enumerate all roots in a stack frame. Stack unwinding is required for exception handling, garbage collection, and security. To allow exact GC, the JIT interface must provide a mechanism to enumerate exactly the elements of the root set in each stack frame. This is in contrast to some other JIT interfaces [19] that assume conservative scanning of the stack. Of course, if a conservative collector were used with ORP, this JIT enumeration interface would never be invoked.

There are two basic solutions to providing stack unwinding and root set enumeration from the stack:

1. The VM and JIT compiler can agree on a common format for GC maps. At compile time, the JIT creates the GC map along with the code, and then the VM can unwind and enumerate without any further help from the JIT.

2. The JIT compiler can store the GC maps in an arbitrary format understood only by that JIT. Whenever the VM subsequently requires a stack unwind or enumeration, the VM calls back into the JIT, which decodes its own GC map and performs the operation.

ORP adopted the latter scheme, the black box approach. The advantage of ORP’s approach is simplicity and flexibility in the JIT design. For example, the O3 JIT supports garbage collection at every native instruction[6], but the simpler O1 JIT only supports GC at call sites and backward branches. This is all possible through the same JIT interface.

4.2 Support for Multiple JIts

To support multiple JIts simultaneously, the core VM maintains an array of pointers to the JIT objects that represent each JIT. The standard ORP configuration includes two statically linked JIts, O1 and O3. Additional JIts may be specified on the command line by supplying the name of a library containing its implementation.

When a method is invoked for the first time, the custom stub transfers control to the core VM, which tries each JIT in turn until one returns success. If no JIT succeeds, ORP terminates with a fatal error. O1 and O3 always succeed, so if no other JIts are loaded via the command line, the first JIT in the array compiles all methods. By default, this is the optimizing O3 JIT.

4.3 Native Method Support

ORP gives each JIT wide latitude in defining how to lay out the stack frames it generates, and in determining how it uses physical registers. As a consequence, a JIT is responsible for unwinding its own stack frames and enumerating their root set references, and must implement methods for this that the VM may call. However, since a native platform compiler, not a JIT, compiles native methods, ORP cannot assume any such cooperation. As a result, ORP must generate special wrapper code for each native method. These wrappers are called when control is transferred from JIT-compiled Java code to native code. They record enough information on the stack and in thread-local storage to support unwinding past native frames and enumerating JNI-style references during garbage collection. The wrappers also include code to perform synchronization for native synchronized methods.

4.3.1 JNI vs. Direct Calls

JNI is preferred and is the only native method calling mechanism available to application programmers. However, a few native methods are called so frequently and their performance is so time-critical that ORP internally uses a `direct` call interface for better performance. The direct interface simply calls the native function without any wrapper to record the necessary information about the transition from Java code to native code. The lack of a wrapper means that ORP cannot unwind its stack frame. This means that the direct native interface can be used only for methods that are guaranteed not to require GC, exception handling, or security support.

For JNI methods, ORP generates a specialized wrapper for each method, which performs the exact amount of work needed based on its signature. This specialization approach reflects the general ORP philosophy to perform as much work as possible at compile time, so that at runtime only the minimum work is required. The wrapper first saves enough information to unwind the stack to the frame of the Java method that called the native function, and then calls the native method. This information includes callee-saved registers, the instruction pointer, the stack pointer, and the previous Last Java Frame (LJF). The LJF list is used to unwind
over a sequence of one or more native frames without it having to know anything about the size or layout of the native platform compiler’s frames (more about this below).

4.3.2 Stack Unwinding for Native Methods
Unwinding a thread’s stack proceeds by first identifying, for each frame, whether it is JIT-compiled or native. If the frame is for a JIT-compiled method, the corresponding JIT is called to unwind the frame. Otherwise, the LJF (Last Java Frame) list is used to skip over one or more native frames, arriving at the previous JIT-compiled frame. The LJF list links together the information saved on the stack by native wrappers. For every thread, a pointer to the most recently saved wrapper information is kept in a thread-local location called the LJF pointer.

Figure 3 shows a thread stack just after a call to a native method. The thread-local LJF variable points to the head of the LJF list. During unwinding, the LJF list is traversed as each native-to-Java transition is encountered, and the wrapper information pointer is used to unwind past native frames.

4.4 Performance Discussion
For the JIT, the performance impact of using interfaces is minimal, since interface methods are called relatively few times during program execution. Naturally, the compilation interface is used once for every method that is compiled (including the wrapper generation for native methods), but there are usually only a few hundred or a few thousand different methods compiled and executed in a typical Java application, and the compilation cost far exceeds the interface cost. Depending on the application, the number of calls related to exception unwinding and root set enumeration may be much higher than the compilation-related calls. Once again, though, the cost of performing these operations generally greatly exceeds the cost of using the interface.

5. GARBAGE COLLECTION INTERFACE
Heap management, including object allocation and automatic garbage collection, is implemented in a component that is separated from the rest of ORP by the GC interface. Using an interface here has a potentially much greater performance impact than using a JIT interface, since a large number of objects may be created and garbage-collected during the lifetime of a typical Java application. Calling a VM function to access type information would slow down such common GC operations as object scanning. A common solution to this problem exposes VM data structures to the GC, but has the downside that it increases the dependence between GC and the core VM. The ORP solution is to expose type information only through a call interface (which provides good separation between the core VM and GC) and to reserve space in the vtable for the GC information. In our experience, that has been a very important feature of the GC interface, and we describe it in more detail in Section 5.1.

Key GC interface methods include:

- \texttt{gc\_requires\_barriers()} – Allows the collector to return a description of what kind of read or write barriers it needs. This might be a value that indicates no barriers are needed, or, for example, that a card marking write barrier must be used. The latter records information needed to track references from objects in a generational collector’s old generation to objects in younger generations.
- \texttt{gc\_write\_barrier()} – If write barriers are needed, both the core VM and the code generated by the JIT must call this method after storing a reference into a field of an object.
- \texttt{gc\_class\_prepared()} – This function is called by the VM after a class is loaded and prepared: after the offsets have been assigned to its instance fields and the vtable has been created for that class.

In addition to using function calls to communicate between the GC and the core VM, the collector assumes the following aspects of the object layout:

- The vtable pointer, which the GC reads to identify the object’s type.
- The object info field, which it uses as a forwarding pointer when an object is copied.
- The layout of array elements, so it can calculate the size of the array object and scan the elements if they are themselves references.

Garbage collection proceeds by stopping all Java threads, enumerating the complete root set, performing the GC, and resuming the threads. The outline of the GC process is:

1. The garbage collector calls the VM procedure \texttt{orp\_enumerate\_root\_set\_all\_threads()}. This causes the VM to stop all Java threads and then enumerates the root set.
2. GC is performed. The heap is scanned, dead objects are collected, live objects may be moved, and root set pointers are updated if those objects were moved.
3. The collector calls \texttt{orp\_resume\_threads\_after()} to notify the core VM to resume the Java threads.

The core VM is responsible for providing the root set to the GC in step 1. The root set consists of global and thread-local object references. Global references are found in the static fields of Java classes, JNI global handles, interned constant strings, and class loaders. They are also found in objects that, while identified to be unreachable during previous GC cycles, have not yet been reclaimed because their finalizers have not been run. Thread-local references are found in JIT-created stack frames, local JNI handles, and the per-thread data structure maintained by ORP. The core VM identifies each JIT-created frame and calls the appropriate JIT routine to enumerate the references it contains.
5.1 Caching Type Information
The core VM does not expose the implementation of its data structures to the GC. That means that if the GC needs to obtain any information about a type (e.g., the number, types, and offsets of fields), it must do so by querying the VM via one of the interface functions. We have observed that some GC operations, such as scanning, are too frequent to allow the use of the call interface and still achieve good performance. Our solution is to reserve space in the beginning of every vtable for the private use of the GC. After a class is loaded, the VM calls gc_class_prepared. At that time, the GC is expected to query the core VM through the call interface and cache the relevant information in the vtable for later use. Because class loading is infrequent, the cost of caching the information is quickly amortized in following garbage collection cycles.

6. EXPERIENCE WITH ORP’s INTERFACES
This section describes some of our experience with the design and use of ORP’s interfaces. It starts with a number of examples where we were able to add important performance optimizations by making a few (or no) extensions to an interface, without requiring direct access by one component (e.g., a JIT) to internal details of another (e.g., data structures of the core VM). The section concludes with a discussion about the overhead of ORP’s interfaces and how we designed them to minimize their cost.

6.1 Flexibility of Stack Unwinding and Root Set Enumeration
The black box approach to stack unwinding and root set enumeration for the stack frames made it easy for us to experiment with new ideas. For instance, we were able to implement support for GC at every instruction [6] in the O3 JIT compiler by using compression for the GC maps, and this required no changes to the core VM or the JIT interface.

Similarly, in the O1 JIT, we implemented lazy GC map creation [7]. This reduces memory usage by not allocating memory for the GC map unless necessary. When the VM calls one of the JIT functions that requires the GC map, the JIT “recompiles” the method (without actually generating code) to compute the GC map, which is then stored for future use. This approach resulted in significant memory savings without any noticeable, negative performance impact, and was also implemented without modifying the JIT interface or core VM.

6.2 Deoptimizing Compiled Code
The O3 JIT performs class hierarchy analysis to determine that there is only a single target for a virtual method invocation. In such cases, the compiler generates code that takes advantage of that information (e.g., through inlining) and registers that class hierarchy assumption with the VM [7]. If the VM later detects that loading a class that overrides the method in question has invalidated this assumption, it invokes the JIT compiler to deoptimize the code and transform the code to use the standard dispatch mechanism for virtual methods. The following functions in the JIT interface are used in that scheme:

- method_is_overridden(Method_Handle m) – Checks if the method has been overridden in any of the subclasses.
- method_set_inline_assumption(Method_Handle caller, Method_Handle callee) – Informs the VM that the JIT compiler has assumed that the caller directly calls the callee.

6.3 Lazy Exceptions
Some applications make extensive use of exceptions for control flow. Often, however, the exception object is dead in the exception handler. In such cases, the work spent to create the exception object and to create and record the stack trace in the exception object is wasted. We implemented an optimization that allows the JIT and the core VM to cooperate on eliminating the creation of exception objects if the existence of the exception objects can be proven to have no observable effect on the application. We call this technique lazy exception creation [7].

To implement lazy exceptions, the JIT marks exception handlers whose exception object is dead at entry. In addition, in the code where the exception is thrown, the JIT analyzes the constructors of the exception objects. If it can prove that the constructor has no side effects, the JIT generates code that, instead of creating the exception object and throwing it, pushes all arguments to the exception constructor and calls a runtime function to perform the lazy throw. As in a normal exception throw, this function unwinds the stack to find the matching handler. If the exception object is dead at the entry to the handler, the VM transfers control to the handler without ever creating the exception object and the associated stack trace. If the exception object is live at the handler entry, the VM lazily creates the exception object and invokes the constructor, passing the appropriate arguments.

Our initial JIT interface did not support lazy exceptions, but we decided the potential performance gain was high enough (it improved the performance of 213 javac, an important SPEC JVM98 benchmark, by 8%) to justify adding the required functions to the interface. Currently only one of our JITs, the O3 JIT, implements the analysis required for lazy exceptions.

6.4 Fast Object Type Checks
For some operations, such as the checkcast and instanceof instructions, the JIT must emit code to test whether an object is assignable to a particular type. Our standard interface provides a runtime function to perform that check using data structures only visible to the VM. We observed, however, that for some benchmarks, these runtime checks are frequent and that performance could be improved by inlining the operations into JIT-generated code. In the applications we studied, these tests involve only the simple case (i.e., they do not involve interfaces or arrays) and can be implemented using the display of parent classes [29].

We allow JITs to generate code for those simple type tests by specifying that the display is located in the vtable, and by providing two functions that the JIT can query at runtime: a function to obtain the offset to the display table from the start of the vtable, and a function to obtain the depth of the class in the class hierarchy. In our implementation, the maximum size of the display is a compile-time constant, and the JIT cannot use the fast
code sequence if the depth of a class is larger than this predefined maximum. Functions to test if a class is an interface or array already existed in the interface.

This modest extension to our original interface allows us to generate fast type checks for virtually all the cases we have encountered.

6.5 Fast Constant String Instantiation

Loading constant strings is another common operation in Java applications. In our original interface, generated code had to call a runtime function to instantiate the string. We extended the interface to reduce the constant string instantiation code sequence to a single load, similar to a load of a static field.

To use this optimization, the JIT must call at compile-time the function `class_get_const_string_intern_addr()`. This function interns the string. It also returns the address of the location containing a pointer to the interned string. The core VM is responsible for reporting the location where the interned string reference is stored as part of the root set before every GC.

Because these string objects are created at compile time regardless of which control paths are actually executed, there is the possibility that applying this optimization blindly to all JIT-generated code will allocate a significant number of unnecessary string objects. Our experiments confirmed this: performance of some applications degraded when we modified the JIT to use fast constant strings. Fortunately, a simple heuristic of not using fast strings in exception handlers avoids this problem. This heuristic can be extended to any cold code, such as might be discovered through profile-based dynamic optimization.

6.6 Overhead of ORP Interfaces

In the benchmarks we run, we find that the execution time is largely dominated by JIT-generated code, and relatively little time is spent in core VM or GC functionality. Within JIT-generated code, though, there are potentially many interface crossings. The interface calls involved can be divided into a few categories, each of which is further described below: GC related, synchronization, class related, type checking, and generic helpers. Our general approach to designing interfaces has been guided by the following principles:

- We initially provide a simple, straightforward set of interface procedures for each category of required functionality. If these later prove to have a performance issue, we provide additional procedures for a “fast path” that can be inlined into JIT-generated code, with a fallback “slow path” if the fast path fails.

- We optimize only operations known (i.e., measured) to have significant overhead.

- We do not try to optimize away overhead that cannot make a noticeable difference, such as call/return instructions in an expensive synchronization code sequence.

- We consider using information known at compile time, such as the precise size of an object to be allocated, to provide specialized interface procedures for a JIT compiler to take advantage of.

GC related interface functions include object allocation (of both objects and arrays) and write barriers. The write barrier sequence consists of just a few straight-line instructions with no control flow, and the extra call and return instructions have not proven to be a performance issue. For object and array allocation, the extra call and return instructions are also not a significant source of overhead. However, at the allocation site, the JIT is always aware of the type of the object, and often knows the allocation size. Therefore, if a future benchmark warranted, it would be straightforward to extend the allocation interface to use information known to the JIT at compile time.

Synchronization functions include `monitorenter` and `monitorexit`. The same functions are sufficient for Java synchronized statements, synchronized virtual methods, and synchronized static methods. The overhead of synchronization is so high on modern processors (e.g., requiring a “lock” prefix instruction on the IA-32) that there is little point in extending its JIT/VM interface. Class related interface functions include exception throwing and constant string loading. Exception throwing is usually rare, but we provide a special interface for lazy exceptions (see Section 6.3), which offers a performance gain of 8% for the `javac` benchmark. Loading of constant strings can be frequent, so ORP provides a mechanism for pre-allocating the string at compile time, after which the JIT can generate code that simply loads a pointer to the string from a fixed memory location (see Section 6.5).

Type checking functions have proven to be executed frequently enough to justify extending the interfaces. These functions include `checkcast`, `instanceof`, `aastore` (array element type compatibility checking), and array bounds checking. For array bounds checking, the VM provides a function that tells the JIT the offset into the array at which the array length is stored, and the JIT is responsible for testing the array index and throwing an exception if necessary. Although `checkcast`, `instanceof`, and `aastore` take up only at most a couple of percent of the execution time for our benchmarks, that is enough to justify some inlining into JIT-generated code. ORP provides an interface to allow the JIT to perform a faster, inlined type check under some conditions that are common in practice (see Section 6.4).

ORP’s VM interface also provides a set of generic helper functions to the JIT, for example, to perform long shift and long multiply, or to perform `f2i/f2l/d2i/d2l` conversions. These functions are not central to a VM, and are simply provided as a convenience for all JITs to use. Therefore, the JIT is free to inline its own version if such a helper function proves to be a performance bottleneck, and no special interface into the VM is required.

7. PERFORMANCE

In our research, high performance is critical, and we put significant effort into designing the interfaces so that they impose minimal overhead. As a result, ORP rivals the best commercial JVMs across a set of standard benchmarks. Unfortunately, licensing restrictions associated with commercial JVMs do not allow us to publish a comparison of ORP’s performance with that of those systems, but we can publish such a comparison with the Jikes RVM [24][25]. Readers curious about ORP’s performance relative to commercial systems can find an extensive performance comparison in an independent study [23].

The Jikes RVM (formerly known as Jalapeño) is a research VM implemented in Java that is used by several academic and research groups. It has a design similar to ORP: it has no interpreter, and all Java code is JIT-compiled. We present the comparison in Figure 4; a lower number indicates better
performance. Those numbers are taken on a 2.0 GHz dual-processor Pentium 4 Xeon machine running Linux. We used the SPEC JVM98 [26] and SPEC JBB2000 [27] benchmarks to evaluate ORP’s performance. We are unable to strictly follow the official run rules for these benchmarks because, for example, the Java class library we use, GNU Classpath, does not support AWT and thus cannot run applets which are required for a conforming SPEC JVM98 run. However, we have tried to approximate as closely as possible the conditions required for conforming runs. We use unmodified benchmarks, each of which was run from the command line. ORP does not exploit the fact that the benchmarks are run from the command line in any way, so we believe that these numbers closely approximate what a conforming run would show. For all benchmarks, we used the same Linux machine for both ORP and Jikes. We also set the maximum heap size to the same value for both systems: 48 MB for SPEC JVM98 and 384 MB for SPEC JBB2000. A smaller heap size was chosen for SPEC JVM98 since that benchmark suite contains client applications. SPEC JBB2000 is a server benchmark and requires a larger heap.

![Performance Comparison](image)

**Figure 4. Performance comparison.**

Performance numbers are presented in a relative fashion so that the performance of ORP is normalized to 1, and lower numbers are better. Note that SPEC JBB2000 reports numbers as the throughput, whereas SPEC JVM98 reports numbers as the execution time. We present the inverse of the throughput, so that the direction of the bar on the graph is the same as for SPEC JVM98, i.e., the lower, the better. The actual SPEC JBB2000 scores were 13,420, 12,159 and 18,160 for FastSemiSpace, FastMarkSweep and ORP respectively. We tried running Jikes in various configurations, and reported results for the best available configuration for JIT compilation. However, there were two different implementations of GC that were nearly tied for top performance, so we decided to include both. In our experiments, the FastSemiSpace collector is typically faster than others (with the exception of 209 db). ORP was run in its default configuration (all methods were compiled by the O3 JIT) and the only parameter we modified was the heap size.

For all benchmarks, ORP is at least 35% faster than Jikes, and in some cases ORP has almost twice the performance. Our paper is not focused on demonstrating new optimization techniques, so the purpose of this section is not to prove that ORP is faster than other systems, but rather to show that our decision to base our design on clear component boundaries with explicit interfaces did not sacrifice performance. The Jikes RVM was designed with different goals in mind and it was originally implemented on PowerPC, so perhaps it should not be surprising that ORP’s performance is better on IA-32. Nonetheless, we believe that this performance comparison demonstrates that using interfaces can be consistent with good performance.

8. RELATED WORK

Well-defined interfaces have been used in the past both for JIT compilers [19][24] and for GC [22][24]. ORP’s interfaces have been developed independently from these efforts, but there are similarities to each of those interfaces. For example, the GC interface used by Sun’s ResearchVM [22] has been used to support the construction of a variety of different garbage collectors, such as the one described in [28]. The capabilities of the ResearchVM’s GC interface resemble those provided by the one in ORP. However, ORP uses interfaces more extensively. For example, each of the ResearchVM’s JITs must directly implement the read and write barriers needed for each collector. The ORP’s GC interface insulates each JIT (and the core VM) from details such as how GC barriers are implemented.

The Sun JDK 1.0.2 JIT interface [19] has been used by a number of JIT compilers, including the OpenJIT [20] and shuJIT [21]. ORP’s JIT interface provides more advanced features such as exact root set enumeration, write barrier support, and deoptimization. ORP’s interface has been used to implement eight different JIT compilers for two CPU architectures (IPF and IA-32) and three operating systems (Windows, Linux, and FreeBSD). An external group implemented one of these JIT compilers [18] based solely on ORP’s JIT interface; they had no access to the ORP source code.

9. CONCLUSION

Along with a general overview of ORP, we have described our use of strict interfaces between the core VM and other components, in particular the JIT compiler and the garbage collector. These interfaces have allowed us and others to construct new JIT compilers and garbage collectors without having to understand or modify the internal structure of the core VM or other components. Contrary to conventional wisdom, we are able to provide this level of abstraction and yet still maintain high performance. The performance cost of using interfaces is minor for the JIT, where interface crossings are infrequent. For a more heavily crossed interface like that of the garbage collector, we maintain high performance by exposing a small, heavily used portion of the Java object structure as part of the interface and allowing caching of frequently used information. Our experience has shown that this approach is effective in terms of both software engineering and performance.

Our experience with ORP’s component design has been positive and has encouraged us to modularize our implementation further. We are currently developing interfaces for other VM components such as ORP’s threading and synchronization subsystem, to simplify experimentation with various runtime technologies.

10. ACKNOWLEDGEMENTS

This work would not be possible without contributions from the ORP development team. We would also like to thank ORP users outside of Intel for their contributions, the Java Grande
anonymous reviewers for their feedback, and GNU Classpath developers for providing an excellent class library for Java.

11. REFERENCES