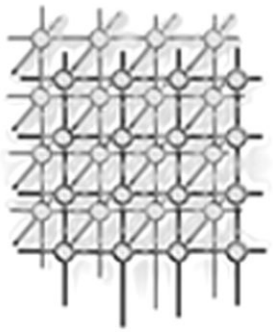
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **17**:617–637
Published online 22 February 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.852



The Open Runtime Platform: a flexible high-performance managed runtime environment‡

Michal Cierniak[1,*,†], Marsha Eng[2], Neal Glew[2], Brian Lewis[2] and James Stichnoth[2]

[1]*Microsoft Corporation, 1 Microsoft Way, Redmond, WA 98052, U.S.A.*
[2]*Microprocessor Technology Laboratory, Intel Corporation, 2200 Mission College Boulevard, Santa Clara, CA, U.S.A.*

SUMMARY

The Open Runtime Platform (ORP) is a high-performance managed runtime environment (MRTE) that features exact generational garbage collection, fast thread synchronization, and multiple coexisting just-in-time compilers (JITs). ORP was designed for flexibility in order to support experiments in dynamic compilation, garbage collection, synchronization, and other technologies. It can be built to run either Java or Common Language Infrastructure (CLI) applications, to run under the Windows or Linux operating systems, and to run on the IA-32 or Itanium processor family (IPF) architectures. Achieving high performance in a MRTE presents many challenges, particularly when flexibility is a major goal. First, to enable the use of different garbage collectors and JITs, each component must be isolated from the rest of the environment through a well-defined software interface. Without careful attention, this isolation could easily harm performance. Second, MRTEs have correctness and safety requirements that traditional languages such as C++ lack. These requirements, including null pointer checks, array bounds checks, and type checks, impose additional runtime overhead. Finally, the dynamic nature of MRTEs makes some traditional compiler optimizations, such as devirtualization of method calls, more difficult to implement or more limited in applicability. To get full performance, JITs and the core virtual machine (VM) must cooperate to reduce or eliminate (where possible) these MRTE-specific overheads. In this paper, we describe the structure of ORP in detail, paying particular attention to how it supports flexibility while preserving high performance. We describe the interfaces between the garbage collector, the JIT, and the core VM; how these interfaces enable multiple garbage collectors and JITs without sacrificing performance; and how they allow the JIT and the core VM to reduce or eliminate MRTE-specific performance issues. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: MRTE; Java; CLI; virtual machine; interface design

*Correspondence to: Michal Cierniak, Microsoft Corporation, 1 Microsoft Way, Redmond, WA 98052, U.S.A.
†E-mail: michaljc@microsoft.com
‡An earlier version of this paper was published online [1].

Received 15 January 2003
Revised 25 August 2003
Accepted 14 October 2003

INTRODUCTION

Modern languages such as Java and C# execute in a managed runtime environment that provides automatic memory management, type management, threads and synchronization, and dynamic loading facilities. These environments differ in a number of ways from traditional languages like C, C++, and Fortran, and thus provide a challenge both for language implementers and for the developers of high-performance microprocessors. This paper concentrates on language implementation challenges by describing a particular MRTE implementation developed at Intel.

Intel's Microprocessor Technology Laboratory (MTL) has developed a managed runtime environment (MRTE) implementation called Open Runtime Platform (ORP). ORP was designed to support experimentation with different technologies in just-in-time compilers (JITs), garbage collection (GC), multithreading, and synchronization. Over the past five years, researchers at Intel and elsewhere have used ORP to conduct a number of MRTE implementation experiments [1–8]. At least three different garbage collectors and eight different JITs have been developed and integrated with ORP.

Three characteristics of MRTEs provide the key challenges to their implementation. First, MRTEs dynamically load and execute code that is delivered in a portable format. This means that code must be converted into native instructions through interpretation or compilation. As a result, MRTE implementations typically include at least one JIT (and often several), and often an interpreter as well. In addition to the challenges of just-in-time compilation, dynamic loading adversely affects important object-oriented optimizations like devirtualization, which reduces the overhead of virtual method calls. Second, MRTEs provide automatic memory management and thus require a garbage collector. Since different applications may impose very different requirements on the garbage collector (e.g. raw throughput versus GC pause time constraints), garbage collector design becomes a significant challenge. Third, MRTEs are multi-threaded, providing facilities for the creation and management of threads, and facilities such as locks and monitors for synchronizing thread execution.

The design of efficient locking schemes, given the modern memory hierarchies and bus protocols of microprocessors, is a significant challenge. In addition, the garbage collector must be designed for multiple threads and may very well need to be parallel itself.

In order to provide the flexibility needed for JIT and garbage collector experiments, we designed interfaces to cleanly separate the JIT and garbage collector parts of ORP from each other and from the core virtual machine (VM) (see Figure 1). Evaluating these experiments requires performance studies, which can be meaningful only if the interfaces impose insignificant overhead. As a result, one of the key contributions of ORP is the design of clean interfaces for JITs and garbage collectors that does not sacrifice performance.

The MRTE implementation challenges described above may require cooperation between different components to achieve a good result. For example, devirtualization optimizations may require cooperation between JITs that do the optimization and the core VM that manages the class hierarchy. We had to balance the need for clean interfaces to support flexibility with the need for cooperation to overcome performance hurdles.

In the next section we elaborate on the nature of MRTEs and the challenges they provide to implementers. Then we describe ORP in detail, paying close attention to the design of interfaces that are clean and also lead to high performance.

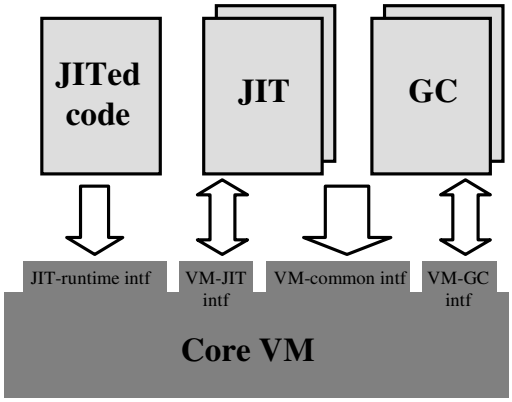


Figure 1. Block diagram of ORP.

MANAGED RUNTIME ENVIRONMENTS

In 1995, the Java programming language and the Java Virtual Machine [10] emerged as the first mainstream MRTE. In 2000, Java was joined by Common Language Infrastructure (CLI) [11], and associated languages like C# [12], as the second major MRTE in the market. Both MRTEs have significant differences over C++ compilers and runtimes; yet they are similar to each other in most important ways. In this section, we describe the terminology and key features that distinguish MRTEs from traditional C++ systems, in particular those that may require new optimization techniques to gain full performance.

Key features

MRTEs dynamically load and execute code. The code and other related data are loaded from *class files*, which can be read from disk, read from a network stream, or synthesized in memory by a running application. Concrete methods include *bytecodes* that specify what to do when that method is invoked. These bytecodes are machine independent, and are at a slightly higher level of abstraction than native instructions. As a result, MRTEs require some means to convert bytecodes into native instructions: an interpreter or a JIT.

MRTEs manage type information, that is, they store information about all the classes, fields, and methods that they have loaded, and also about other types that they define or derive automatically, such as primitive and array types. MRTEs provide reflection facilities that allow application code to enumerate and inspect all this information about types, fields, and methods.

MRTEs provide automatic memory management. There is a region of memory belonging to the MRTE called the *heap*. If the heap is full, the MRTE tries to reclaim the space of objects no longer in use, a process known as *garbage collection* (GC). The part of the MRTE that manages the heap, allocates objects, and performs GC is known as the *garbage collector*.

To perform a GC, a garbage collector must first find all direct references to objects from the currently executing program; these references are called *roots*, or the *root set*, and the process of finding them all is called *root-set enumeration*. Within one stack frame, each native instruction might have a different set of roots on the stack and in physical registers; for this purpose, a JIT usually maintains a *GC map* to provide the mapping between individual instructions and roots. Next, the garbage collector must find all objects reachable from the root set; this is called *marking* or *scanning*. Finally, the garbage collector reclaims the remaining, unreachable, objects.

Generational garbage collectors attempt to improve GC efficiency by only scanning a portion of the heap during a collection. Doing so requires additional support from the rest of the MRTE, particularly the JITs: a *write barrier* must be called whenever a reference type pointer in the heap is modified.

MRTEs provide *exceptions* to deal with errors and unusual circumstances. Exceptions can be thrown either explicitly via a 'throw' bytecode, or implicitly by the MRTE itself as a result of an illegal action such as a null pointer dereference. Each bytecode in a method has an associated list of exception handlers. When an exception is thrown, the JVM must examine each stack frame in turn, until it finds a matching exception handler among the list of associated exception handlers. This requires *stack unwinding*, the ability to examine stack frames and remove them from the stack one by one. Note that stack unwinding is also needed to implement security policies and root-set enumeration.

Most of the significant differences between CLI and Java are due to additional features in CLI. CLI has a richer set of types than Java. Key among these are *value types*, which resemble C structures and are especially useful for implementing lightweight types such as complex numbers. CLI also supports *managed pointers* that have many uses, including the implementation of call-by-reference parameters. Since these may point into the interior of heap objects, they may require special support from garbage collectors. CLI also includes several features that are especially helpful when interfacing with legacy libraries. Its platform library invocation service automates much of the work to call native library routines. CLI, unlike Java, also allows objects to be *pinned* to ensure they will not be relocated.

Optimization challenges

MRTEs (particularly Java systems) gained an early reputation for not performing as well as traditional languages like C or C++. In part, this reputation arose because the first implementations only interpreted the bytecodes. When JITs were introduced as a way to achieve better performance than interpretation, they were thought of as not optimizing code, but rather as quick producers of native code, with quick startup and response times being the driving requirements. Over time, JIT code quality has increased, due to more mature JIT technology, dynamic recompilation techniques, and a relaxation of the fast startup requirement, particularly for longer-running, server-type applications.

Despite the general maturation of JIT technology, there still remain some fundamental issues that separate a MRTE JIT from a traditional C++ compiler. One set of issues is the lack of whole-program analysis in a MRTE. Classes can be dynamically loaded into the system at any time, and new classes may invalidate assumptions made during earlier compilations of methods. When making decisions about devirtualization, inlining, and direct call conversion, JITs must take into account the possibility that a target method may be overridden in the future (even if at compile time there is only one possible target), and that a target class may be subclassed (even if the class is currently not extended).

This generally results in extra overhead for method dispatch or inlining than would typically be present in a C++ system.

Another set of issues is the safety checks required by MRTE semantics. For example, every array access must test whether the array index falls within the bounds of the array. Every type cast must test whether it is a valid cast. Every object dereference must test whether the reference is null. C and C++ lack these runtime requirements, so as to achieve competitive performance, JITs must employ additional techniques to minimize the overhead.

Further performance challenges relate to the garbage collector. Some batch-style applications may demand the highest possible throughput, while other interactive applications may require short GC pause times, possibly at the cost of some throughput. Such requirements have a profound impact on the design of the garbage collector. In addition, since the garbage collector is responsible for mapping objects into specific heap locations, it may also need to detect relationships between objects and ensure that related objects are collocated in memory, in order to maximize memory hierarchy locality.

Some of these JIT-related overheads can be reduced through compiler techniques alone. Others require some level of cooperation with the core VM. Throughout this paper, we identify such techniques and how they are implemented in ORP.

OVERVIEW OF ORP

ORP is a high-performance MRTE that features exact generational GC, fast thread synchronization, and multiple JITs, including highly optimizing JITs. All code is compiled by these compilers: there is no interpreter. ORP supports two different MRTE platforms, Java [10] and CLI [11].

Basic structure

ORP is divided into three components: the core VM, JITs, and the garbage collector. The core VM is responsible for class loading, including storing information about the classes, fields, and methods loaded. The core VM is also responsible for coordinating the compilation of methods to managed code, root-set enumeration during GC, and exception throwing. In addition, the core VM contains the thread and synchronization subsystem, although we are planning to split this into a separate component in a future version of ORP. JITs are responsible for compiling methods into native instructions. The garbage collector is responsible for managing the heap, allocating objects, and reclaiming garbage when the heap is full.

ORP is written in about 150 000 lines of C++ and a small amount of assembly code (this includes the core VM code, and excludes the JIT and garbage collector code). It compiles under Microsoft Visual C++ 6.0[§] and GNU g++, and it runs under Windows (NT/2000/XP[§]), Linux[§], and FreeBSD[§]. ORP supports both IA-32 [13] and Itanium processor family (IPF) [14] CPU architectures. ORP uses the GNU Classpath library [15], an open source implementation of the Java class libraries, and OCL [16], an open source implementation of the CLI libraries that is ECMA-335 [11] compliant.

[§]Other brands and names are the property of their respective owners.

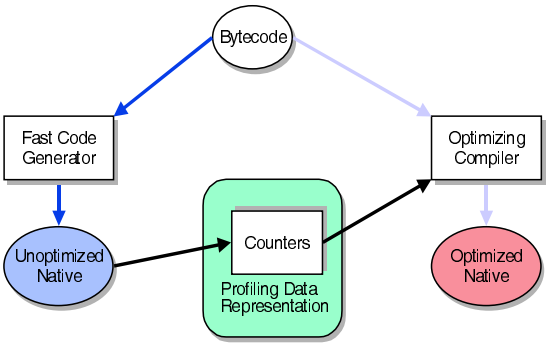


Figure 2. Structure of dynamic recompilation.

ORP was originally designed with two JITs for Java. The *Simple Code Generator* (known as the *O1 JIT* [2]) produces code directly from the JVM bytecodes [10] without applying complex optimizations. Its optimizations include strength reduction, load-after-store elimination, and simple versions of common-subexpression elimination (CSE), eliminating array-bounds checks, and register allocation.

The *Optimizing Compiler* (known as the *O3 JIT*) converts JVM bytecodes 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 elimination of array-bounds checks.

As shown in Figure 2, ORP can run in a mode that uses both the O1 and O3 JITs. 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.

ORP also supports a very simple JIT for CLI (currently only on the IA-32 platform), known as the *O0 JIT*. It does no optimizations and was designed for simplicity and to ease debugging. For each CLI bytecode instruction, it generates a sequence of machine instructions that is fixed for each set of operand types.

StarJIT [17] is a new JIT designed to plug into ORP. It supports Java and CLI, and it produces aggressively optimized code for IA-32 and IPF. It translates JVM and CLI bytecodes into a single common intermediate representation on which the rest of StarJIT operates. StarJIT includes an SSA-based optimizer and supports profile-based optimizations as well as dynamic optimizations that are based on continuous profiling and monitoring during program execution.

ORP has supported many different GC implementations over its lifetime, including a simple stop-the-world collector, an implementation of the Train Algorithm [18], and a concurrent collector [5]. There is support in the VM and JIT interfaces for moving collectors (in which objects can be relocated over their lifetimes), and for generational collectors (which require write barrier support from JITs and the core VM). ORP also supports dynamic linking of the GC module, making it possible to select a specific GC implementation via a command-line option.

Common support for Java and CLI

CLI and Java are semantically similar enough that most of ORP's implementation is common to both runtimes. Both Java and CLI require approximately the same support for class loading, exception handling, threads, reflection, runtime, and low-level (non-library specific) native methods. Of course, CLI uses a different object file format than Java, so the object file loaders are different. Similarly, the class libraries for the two runtimes are different and require a different set of native method implementations. CLI's bytecode instructions are different, so there are differences in the JITs. However, these differences are relatively minor, and most of the code in the StarJIT is common. In general, the significant differences between CLI and Java are due to additional features in CLI. This means if a MRTE (or JIT) supports CLI, it is relatively straightforward to add support for Java.

ORP has relatively few Java-specific or CLI-specific source files beyond those that load classes and those that implement the native methods required by the different CLI and Java class libraries. The MRTE-specific source changes are mostly in short sequences of code that are conditionally compiled when ORP is built. We are currently refactoring ORP to share even more code, which will significantly reduce the need for conditionally-compiled code sequences. For example, to indicate an attempt to cast an object to a class of which it is not an instance, a Java MRTE must throw an instance of *java.lang.ClassCastException*, whereas a CLI MRTE must throw *System.InvalidCastException*. Refactoring this part of ORP's implementation simply involves raising the exception stored in a variable that is initialized to the appropriate value.

THE CORE VM

The core VM is responsible for the overall coordination of the activities of ORP. It is responsible for class loading: it stores information about every class, field, and method loaded. The class data structure includes the virtual-method table (vtable) for the class (which is shared by all instances of that class), attributes of the class (public, final, abstract, the element type for an array class, etc.), information about inner classes, references to static initializers, and references to finalizers. The field data structure includes reflection information such as name, type, and containing class, as well as internal ORP information such as the field's offset from the base of the object (for instance fields) or the field's address in memory (for static fields). The method data structure contains similar information.

These data structures are hidden from components outside the core VM, but the core VM exposes their contents through functions in the VM interface. For example, when a JIT compiles an access to an instance field, it calls the VM interface function for obtaining the field's offset, and it uses the result to generate the appropriate load instruction.

There is one data structure that is shared across all ORP components, including JITs and garbage collectors, which describes the basic layout of objects. Every object in the heap, including arrays, begins with the following two fields:

```
typedef struct Managed_Object {
    VTable *vt;
    uint32 obj_info;
} Managed_Object;
```

No other fields of the Managed_Object data structure are exposed outside the core VM. The first field is a pointer to the object's vtable. There is one vtable for each class[¶], and it stores enough class-specific information to perform common operations like virtual-method dispatch. The vtable is also used during GC, where it may supply information such as the size of the object and the offset of each reference stored in the instance. The second field, *obj_info*, is 32 bits wide on both IA-32 and IPF architectures, and it 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 JITs also share knowledge about the representation of array instances. 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 garbage collector and JITs via the VM interface.

Another small but important piece of shared information is the following. The garbage collector is expressly allowed to use a portion of the vtables to cache frequently used information to avoid runtime overhead. This cached information is private to the garbage collector and is not accessed by other ORP components. Apart from the basic assumptions about object layout and this vtable cache, all interaction between major ORP components is achieved through function calls.

The VM interface also includes functions that support managed code, JITs, and the garbage collector. These functions are described as part of the discussion of the specific components, which we turn to next.

THE JIT INTERFACE

JITs are responsible for compiling bytecodes into native managed code, and for providing information about stack frames that can be used to do root-set enumeration, exception propagation, and security checks.

Compilation overview

When the core VM loads a class, new and overridden methods are not immediately compiled. Instead, the core VM initializes the vtable entry for each of these methods to point to a small custom stub

[¶]Because there is a one-to-one correspondence between a *Class* structure and a vtable, it would be possible to unify them into a single data structure. We chose to separate them to make sure that offsets to entries in the *VTable* that are used for method dispatch are small, and that instructions generated for virtual method dispatch can be encoded with shorter sequences. Also, the information in vtables is accessed more frequently, so collocating them improves spatial locality and reduces data translation lookaside buffer (DTLB) misses.

that causes the method to be compiled upon its first invocation. After a JIT compiles the method, the core VM iterates over all vtables containing an entry for that method, and it replaces the pointer to the original stub with a pointer to the newly compiled code.

ORP allows many JITs to coexist within it. Each JIT interacts with the core VM through the JIT interface, which is described in more detail below, and must provide an implementation of the JIT side of this interface. The interface is almost completely CPU independent (the only exception being the data structures used to model the set of physical registers used for stack unwinding and root-set enumeration), and it is used by both our IA-32 JITs and our IPF JITs. JITs can be either linked statically or loaded dynamically from a dynamic library.

As previously mentioned in the ORP overview, managed code may include instrumentation that causes it to be recompiled after a certain number of invocations. Another option is to have a background thread that supports recompiling methods concurrently with the rest of the program execution.

Native methods are also 'compiled' in the following sense. When a native method is invoked for the first time, the core VM generates a custom wrapper for that native method, and installs it in the appropriate vtables. The purpose of the wrapper is to resolve the different calling conventions used by managed and native code.

Interface description

The JIT interface consists of a set of functions that every JIT is required to export and a set of functions that the core VM exports. One obvious function in the JIT interface instructs the JIT to compile a method. The JIT interface also includes some not-so-obvious JIT-exported functions that implement functionality that is traditionally thought of as being part of the core VM. These include functions to unwind a stack frame and to enumerate all roots in a stack frame. Stack unwinding is required for exception handling, GC, and security. To allow exact GC, the JIT interface provides a mechanism to enumerate exactly the roots of a stack frame. Given an instruction address, the JIT consults the GC map for that method and constructs the root set for the frame. This is in contrast to some other JIT interfaces such as the Sun JDK 1.0.2[‖] JIT interface [19] that assumes conservative scanning of the stack. Of course, if a conservative collector were used with ORP, this mechanism for root-set enumeration would never be used.

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

1. A white-box approach in which the core VM and all JITs agree on a common format for GC maps. At compile time, JITs create GC maps along with native code, and then the core VM can unwind and enumerate without any further help from the JITs.
2. A black-box approach in which each JIT can store GC maps in an arbitrary format understood only by that JIT. Whenever the core VM unwinds the stack or enumerates roots, it calls back into the appropriate JIT for the frame in question, and the JIT decodes its own GC map and performs the operation.

ORP uses the latter scheme, the black-box approach. The advantage of ORP's approach is simplicity and flexibility in JIT design. For example, the O3 JIT supports GC at every native instruction [9],

[‖] Other brands and names are the property of their respective owners.

but the simpler O1 JIT only supports GC at call sites and backward branches. This is all possible through the same JIT interface.

Support for multiple JITs

To support multiple JITs simultaneously, the core VM maintains an array of pointers to *JIT* objects that represent each JIT. The standard ORP/Java/IA-32 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.

Core VM support for JITs and managed code

The VM interface includes functions to allocate memory for code, data, and JIT-specific information. The core VM allocates this memory, rather than JITs, which allows the space to be reclaimed when it is no longer needed (however, ORP does not currently implement unloading or GC of methods). The VM interface also includes functions to query the exception information provided in the application class files and to set the exception information for managed code. The core VM uses this latter information during exception propagation.

The core VM also provides runtime support functions for use by managed code. They provide functionality such as throwing exceptions, subtype checks, complex arithmetic operations, and other non-trivial operations.

Optimizations

As mentioned in the section on MRTEs, there are safety requirements and features such as dynamic class loading that can affect the applicability or effectiveness of traditional compiler optimizations. To get performance comparable to unsafe, static languages like C++, JITs must include optimizations that reduce or eliminate safety overheads, and that can work effectively even in the presence of dynamic loading. Some of these optimizations can be implemented entirely in the JITs, but some require cooperation from the core VM. Here we outline some of the key problems and their solutions, along with the additional interface functions that provide the needed cooperation.

Fast subtype checking

Both Java and CLI support single inheritance and, through interfaces, multiple supertypes. An instance of a subtype can be used where an instance of the supertype is expected. Testing whether an object is an instance of a specific supertype is frequent: many thousands of type tests might be done per second during program execution. These type tests can be the result of explicit tests in application code (e.g. Java's *checkcast* bytecode) as well as implicit checks during array stores (e.g. Java's *aastore* bytecode). These array store checks verify that the types of objects being stored into arrays are compatible with the element types of the arrays. Although *checkcast, instanceof*, and *aastore* take

up at most a couple of per cent of the execution time for our Java benchmarks, that is enough to justify some inlining into managed code. The core VM provides an interface to allow JITs to perform a faster, inlined type check under some conditions that are common in practice.

Direct-call conversion

In ORP, devirtualized calls are still by default indirect calls. Even though the target method may be precisely known, it may not have been compiled yet, or it may be recompiled in the future. By using an indirect call, the managed code for a method can easily be changed after the method is first compiled, or after it is recompiled.

Unfortunately, indirect calls may require additional instructions (at least on IPF), and may put additional pressure on the branch predictor. Thus it is important to be able to convert them into direct calls. To allow this to happen, the core VM includes a callback mechanism to allow JITs to patch direct calls when the targets change due to compilation or recompilation. Whenever a JIT produces a direct call to a method, it calls a function to inform the core VM of this fact. If the target method is (re)compiled, the core VM calls back into the JIT to patch and redirect the call.

Devirtualization and dynamic loading

The O3 JIT performs class-hierarchy analysis to determine if there is a single target for a virtual method invocation. In such cases, the compiler generates code that takes advantage of that information (e.g. direct calls or inlining) and registers that class-hierarchy assumption with the core VM. If the core VM later detects that loading a class violates a registered class-hierarchy assumption, it calls back into the JIT that registered the assumption, to instruct it to deoptimize the code to use the standard dispatch mechanism for virtual methods. This is a variant of guarded devirtualization and does not require stack frame patching (see [4] for more details). The following functions in the JIT interface are used in this scheme:

- *method_is_overridden(Method_Handle m)*: this function checks if the method has been overridden in any of the subclasses;
- *method_set_inline_assumption(Method_Handle caller, Method_Handle callee)*: this function informs the core VM that the JIT has assumed that *caller* non-virtually calls the *callee*.
- *method_was_overridden(Method_Handle caller, Method_Handle callee)*: the core VM calls this function to notify the JIT that a new class that overrides the method *callee* has just been loaded.

This small set of methods, though somewhat specialized, was sufficient to allow JITs to implement an important optimization without requiring detailed knowledge of the core VM's internal structures.

Fast constant-string instantiation

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

To use this optimization, JITs, at compile time, call the function *class_get_const_string_intern_addr()*. This function interns the string, and returns the address of a location pointing to the interned string. Note that the core VM reports this location as part of the root set during 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 managed code will allocate a significant number of unnecessary string objects. Our experiments confirmed this: performance of some applications degraded when JITs use fast constant strings. Fortunately, the simple heuristic of not using fast strings in exception handlers avoids this problem.

Native-method support

ORP gives JITs wide latitude in defining how to lay out their stack frames, and in determining how they use physical registers. As a consequence, JITs are responsible for unwinding their own stack frames and enumerating their roots, and must implement functions for this that the core VM calls. However, since a native platform compiler, not a JIT, compiles unmanaged native methods, the core VM cannot assume any such cooperation. As a result, the core VM generates special wrapper code for most native methods. These wrappers are called when control is transferred from managed to native code. They record enough information on the stack and in thread-local storage to support unwinding past native frames and enumerating Java Native Interface (JNI) references during GC. The wrappers also include code to perform synchronization for native synchronized methods.

In ORP, managed code can interact with native code using one of four native interfaces:

- direct calls;
- Raw Native Interface** (RNI);
- Java Native Interface (JNI); and
- Platform Invoke (PInvoke).

CLI code uses PInvoke, and Java code uses RNI and JNI. For optimization purposes, native methods may be called directly. RNI, JNI, and PInvoke require a customized wrapper as discussed above. In Java most of the methods use JNI.

Interestingly, we also found JNI methods to be useful for implementing CLI's *internal call* methods. These are methods implemented by the MRTE itself that provide functionality that regular managed code cannot provide, such as *System.Object.MemberwiseClone*.

Native interfaces comparison

JNI and PInvoke are the preferred interfaces and are the only native-method calling mechanisms available to application programmers. However, a few native methods are called so frequently, and their performance is so time-critical, that ORP internally uses either a *direct* call interface or RNI for better performance.

**ORP's implementation of RNI is very close to but not identical to the original Raw Native Interface that is used in the Microsoft Java SDK [20].

The direct interface simply calls the native function without any wrapper to record the necessary information about the transition from managed 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 only be used for methods that are guaranteed not to require GC, exception handling, or security support.

For the PInvoke, RNI, and JNI interfaces, ORP generates a specialized wrapper for each method. This wrapper performs the exact amount of work needed based on the method's signature. This specialization approach reflects the general ORP philosophy of performing as much work as possible at compile time, so that minimum work is required at runtime. The wrapper first saves enough information to unwind the stack to the frame of the managed code of the method that called the native function (described in more detail below), performs locking for synchronized methods, and then calls the actual native method.

RNI and JNI are very similar; the only major difference between them is how references to managed objects are handled. In RNI, references are passed to native code as raw pointers to the managed heap. In JNI, all references are passed as handles. JNI handles incur additional overhead but they make writing and debugging native methods much simpler.

CLI's PInvoke is designed to simplify the use of existing libraries of native code. It supports the look up by name of functions in specified dynamic link libraries (DLLs). It handles the details of loading DLLs, invoking functions with various calling conventions, and marshalling arguments and return values. PInvoke automatically translates (*marshals*) between the CLI and native representations for several common data types including strings and one-dimensional arrays of a small set of types.

Stack unwinding for native methods

Unwinding a thread's stack proceeds by first identifying, for each frame, whether it is managed or native. If the frame is managed, the corresponding JIT is called to unwind the frame. Otherwise, the core VM uses a last managed frame (LMF) list to find the managed frame nearest the native frame. Each thread (in thread-local storage) has a pointer to the LMF list, which links together the stack frames of the wrappers of native methods. Included in these wrapper stack frames and the LMF list is enough information to find the managed frame immediately before the wrapper frame, as well as the previous wrapper frame. Also included are the callee-saved registers and the instruction pointer needed to unwind to the managed frame.

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

JNI optimizations

The core VM generates specialized JNI wrappers to support the transition from managed to native code. The straightforward implementation of these wrappers calls a function to allocate storage and initialize JNI handles for each reference argument. However, most JNI methods have only a small number of reference parameters. To take advantage of this fact, we use an inline sequence of instructions to allocate and initialize the JNI handles directly. This can improve by several per cent the performance of applications that make many JNI calls.

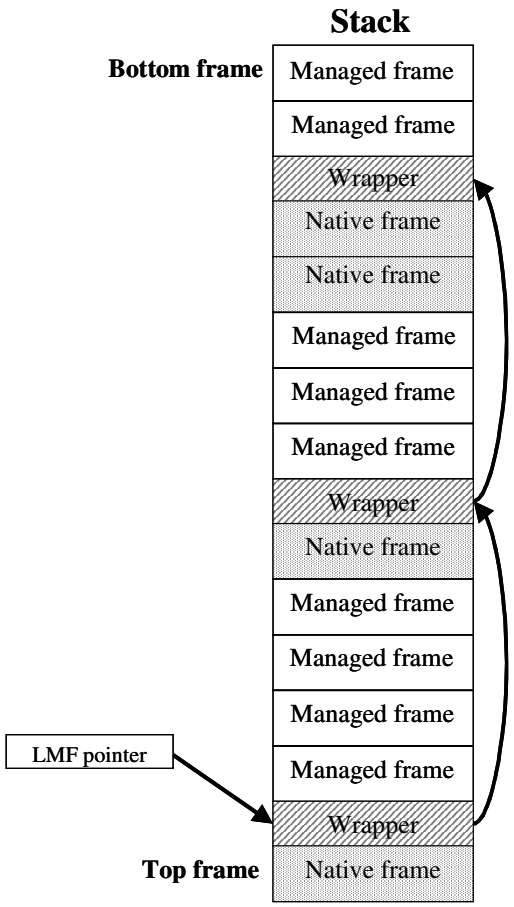


Figure 3. LMF list after the call to a native.

Flexibility versus performance

For JITs, the performance impact of using interfaces is minimal, since interface functions are called infrequently during program execution. Naturally, the compilation interface is used once for every method that is compiled (including the wrapper generation for native methods), but the number of methods executed is typically orders of magnitude greater than the number compiled, 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.

THE GC INTERFACE

The main responsibility of the garbage collector is to allocate space for objects, manage the heap, and perform GC. The GC interface defines how the garbage collector interacts with the core VM and the JITs, and it is described in detail below. First we describe the typical GC process in ORP.

Overview of GC

Typically, when the heap is exhausted, GC proceeds by stopping all managed threads at GC-safe points, determining the set of root references [21], 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. Then it enumerates each frame of each thread stack, and calls the JIT that produced the code for the frame to enumerate the roots on that frame and to unwind to the previous frame.

The garbage collector is also responsible for allocating managed objects. As such, whenever the core VM, managed code, or native methods need to allocate a new object, they call a function in the GC interface. If the heap space is exhausted, the garbage collector stops all managed threads and performs GC as described above.

A generational garbage collector also needs support from the core VM and from managed code to execute a write barrier whenever a reference field of a managed object is changed. In particular, this requires the JIT to insert calls to the write barrier function in the GC interface into managed code, where appropriate.

Overview of the interface

Using an interface for GC potentially has a much greater performance impact than using a JIT interface, since a large number of objects are created and garbage-collected during the lifetime of a typical MRTE application. Calling a core VM function to access type information would slow down common GC operations such as object scanning. A common solution to this problem is to expose core-VM data structures to the garbage collector, but this exposure increases the dependency between the garbage collector and the core VM.

The solution in ORP is to expose core-VM data structures only through a call interface (which provides good separation between the core VM and the garbage collector), but to allow the garbage collector to make certain assumptions and to have some space in vtables and thread local storage. In our experience, these non-call parts have been a very important feature of the GC interface. The following sections describe the explicit functions in the GC interface, as well as the implicit data layout assumptions shared between the core VM and the garbage collector.

Data layout assumptions

Part of the GC interface consists of an implicit agreement between the core VM and the garbage collector regarding the layout of certain data in memory. There are four classes of memory assumptions in the interface.

First, the garbage collector assumes the layout of objects described previously, in terms of the *Managed_Object* data type. This allows it to load an object's vtable without calling into the core VM. In addition, it can use the *object_info* field for certain purposes such as storing a forwarding pointer while performing GC. However, this field is also used by the synchronization subsystem, so the garbage collector must ensure it does not interfere with those uses.

Second, the core VM reserves space in each vtable for the garbage collector to cache type information it needs during GC. This cached information is used in frequent operations such as scanning, where calling the core VM would be too costly. When the core VM loads and prepares a class, it calls the GC function *gc_class_prepared* so that the garbage collector can obtain information it needs from the core VM through the VM interface and store it in the vtable.

Third, the core VM reserves space in thread-local storage for the garbage collector, and during thread creation it calls *gc_thread_init* to allow the garbage collector to initialize this space. The garbage collector typically stores a pointer to per-thread allocation areas in this space.

Fourth, the garbage collector assumes arrays are laid out in a certain way. It can call a VM function to obtain the offset of the length field in an array object, and for each array type, the offset of the first element of arrays of that type. It can further assume that the elements are laid out contiguously. Using these assumptions, the garbage collector can enumerate all references in an array without further interaction with the core VM. Note that the two offsets can be cached in vtables or other garbage collector data structures.

Initialization

The GC interface contains a number of functions that are provided to initialize certain data structures and state in the core VM and the garbage collector at specific points during execution. These points include system startup, as well as when new classes are loaded and new application threads are created.

At the startup of ORP, the core VM and the JITs call the GC interface function *gc_requires_barriers* to determine what kinds (if any) of write barriers the garbage collector requires. Write barriers are used by some generational, partial collection, and concurrent GC techniques to track the root sets of portions of the heap even in the presence of updates to those portions. If the garbage collector requires write barriers, then JITs must generate calls to the GC function *gc_write_barrier* after code that stores references into an object field.

As previously mentioned, the core VM calls *gc_class_prepared* upon loading a class, and *gc_thread_init* upon creating a thread. Also, the core VM calls *gc_init* to initialize the garbage collector, *gc_orp_initialized* to tell the garbage collector that the core VM is sufficiently initialized that it can enumerate roots, and thus that GC is allowed, and *gc_next_command_line_argument* to inform the garbage collector of command line arguments.

Allocation

There are several functions related to allocating space for objects. The function *gc_malloc* is the main function, and it allocates space for an object given the size of the object and the object's vtable. There are other functions for special cases such as pinned objects. These allocation functions are invoked by the core VM or by the managed code.

Root-set enumeration

If the garbage collector decides to do GC, it first calls the VM function *orp_enumerate_root_set_all_threads*. The core VM is then responsible for stopping all threads and enumerating all roots. These roots consist of global and thread-local object references. Global references are found in static fields of classes, JNI global handles, interned constant strings, and other core VM data structures. Thread-local references are found in managed stack frames, local JNI handles, and the per-thread data structures maintained by the core VM. The core VM and the JITs communicate the roots to the garbage collector by calling the function *gc_add_root_set_entry(Managed_Object**)*. Note that the parameter points to the root, not the object the root points to, allowing the garbage collector to update the root if it moves objects during GC.

After the core VM returns from *orp_enumerate_root_set_all_threads*, the garbage collector has all the roots and proceeds to collect objects no longer in use, possibly moving some of the live objects. Then it calls the VM function *orp_resume_threads_after*. The core VM resumes all threads; then the garbage collector can proceed with the allocation request that triggered GC.

Flexibility versus performance

Relatively few interface functions need to be called during GC, largely as a result of the cached type information. However, within managed code, there are potentially many GC interface crossings. The majority of these are object allocation (both of objects and of 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 in practice. For object and array allocation, the extra call and return instructions are also not a significant source of overhead for MRTE applications (but the same is not true in functional languages). However, if future benchmarks warranted it, the JIT and GC interfaces could be extended to allow inlining of the fast-path of allocation into managed code.

PERFORMANCE OF ORP

For our work to be relevant to other groups that we work with, and to Intel as a whole, ORP must perform as well as commercial JVMs. As a result, we have put significant effort into designing our interfaces to impose minimal overhead. The purpose of this section is not to provide any in-depth analysis of ORP's performance, but merely to show that ORP is comparable with commercial JVMs on a set of standard benchmarks. A more extensive performance analysis appears in another study [22].

Many commercial JVMs have been developed for the IA-32 platform. A few examples include IBM JDK 1.3.1†† [23], Sun HotSpot JDK 1.4.0†† [24], and BEA JRockit JVM 1.3.1†† [25].

††Other brands and names are the property of their respective owners.

We compare ORP with Sun HotSpot JDK 1.4.0[††] [24] for SPEC JVM98 [26][‡‡] which is a set of benchmarks that are designed to reflect the workload on a client machine.

Performance comparison

The comparison appears in Figure 4. These numbers are taken on a 2.0 GHz dual-processor Pentium* 4 Xeon™ machine without hyper-threading, with 1 GB of physical memory, and running RedHat Linux 7.2[†]. We set the initial and maximum heap sizes to the same value of 48 MB for both VMs by using the –Xms and –Xmx command line options.

Performance numbers are presented in a relative fashion so that the performance of ORP is normalized to 1, and numbers greater than 1 indicate better performance than ORP (the graph shows the inverse of the execution time). 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.

ORP performance compares well with Sun HotSpot on these benchmarks. We believe that this performance comparison demonstrates that using interfaces can be consistent with good performance.

Performance breakdown

For ORP performance analysis and tuning, it is important to understand the breakdown of where time is being spent in the system. We use the VTune™ Performance Analyzer [27] to identify the hot methods; ORP includes special support for registering dynamically-generated functions with VTune.

Figure 5 shows the breakdown of cycles among different logical components of ORP for the SPEC JBB2000 benchmark [28]. These measurements were taken on a 2.0 GHz quad-processor Pentium 4 Xeon™[‡] machine without hyper-threading, with 4 GB physical memory, and running the Microsoft Windows 2000 Advanced Server operating system. We separate the execution into several groups.

- Core VM (C code). This represents the statically compiled C code of the core VM.
- Core VM (asm code). This represents the assembly code that is dynamically generated and executed as part of the core VM functionality.
- GC. This represents both the allocation and the collection portions of the garbage collector, each of which occupies roughly 50% of the time in the GC component.
- Java code (lib). This represents the JIT-generated code for the core Java classes, primarily ‘java.*’ methods.

[‡‡]As a research project, the information based on the components of SPEC JVM98 are published per the guidelines listed in the SPEC JVM98 Run and Reporting rules section ‘4.1 Research Use’ (http://www.spec.org/jvm98/rules/runrules-20000427.html#Research). 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 the Abstract Window Toolkit (AWT) and thus cannot run the applets that are required for a fully compliant SPEC JVM98 run. We use unmodified benchmarks, which are run from the command line due to inadequacies in the classpath’s AWT. As such these results do NOT represent SPEC JVM98 metrics but only run times and are not directly comparable to any SPEC metrics. Enough information is being provided that would allow these results to be reproduced if ORP were publicly released.

*Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[†]Other brands and names are the property of their respective owners.

[‡]Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

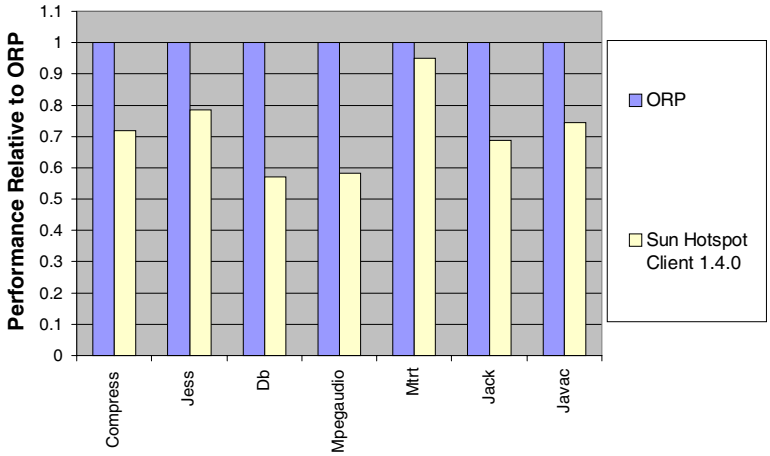


Figure 4. Relative Performance to Sun HotSpot Client.

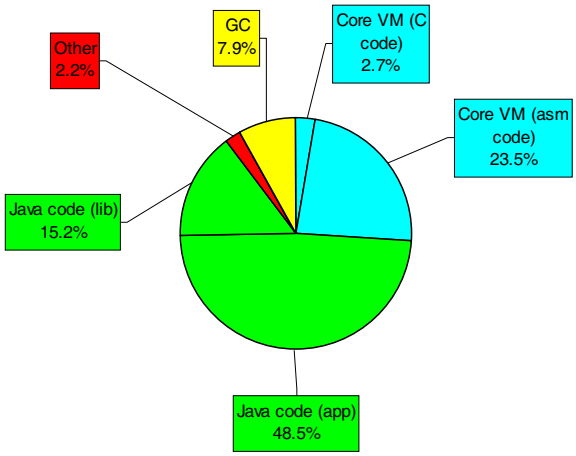


Figure 5. SPEC JBB2000 execution breakdown.

- Java code (app). This represents the JIT-generated code for the SPEC JBB2000 application classes.
- Other. This is primarily time spent in the OS or in the CPU-idle loop.

In this benchmark, 26% of the execution time is spent in the core VM (C and assembly). This further breaks down into 21% of total execution time in synchronization (monitorenter and monitorexit) code, and 5% of total execution time spread widely among other core VM routines. The vast majority of the synchronization time is spent in a single 'lock cmpxchg' instruction, and inlining the synchronization code into the JIT-generated methods does not reduce the cost.

Ignoring the inherently expensive synchronization code, 5% of execution time is spent in the core VM, 8% in the GC, and 64% in JIT-generated Java methods. Given the relatively small amount of time spent in the VM and GC components, we believe that we have achieved our goal of maintaining high performance while using strict interfaces.

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 JITs and the garbage collector. These interfaces have allowed us and others to construct new JITs 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 JITs, where interface crossings are infrequent. For the more heavily crossed interface 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 MRTE components such as ORP's threading and synchronization subsystem, to simplify experimentation with other runtime technologies.

ACKNOWLEDGEMENTS

This work would not be possible without contributions from the entire ORP development team. We also thank ORP users outside of Intel for their contributions, and GNU Classpath developers for providing an open-source class library for Java.

REFERENCES

1. Cierniak M, Eng M, Glew N, Lewis B, Stichnoth J. The Open Runtime Platform: A flexible high-performance managed runtime environment. *Intel Technology Journal* 2003; **7**(1).
Available at: http://developer.intel.com/technology/itj/2003/volume07issue01 [February 2003].

2. Adl-Tabatabai A-R, Cierniak M, Lueh G-Y, Parikh VM, Stichnoth JM. Fast, effective code generation in a just-in-time Java compiler. *ACM Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998. ACM Press: New York, 1998; 280–290.
3. Cierniak M, Lewis BT, Stichnoth JM. Open runtime platform: Flexibility with performance using interfaces. *Joint ACM Java Grande—ISCOPE 2002 Conference*, Seattle, WA, 2002. ACM Press: New York, 2002.
4. Cierniak M, Lueh G-Y, Stichnoth JM. Practicing JUDO: Java under dynamic optimizations. *ACM Conference on Programming Language Design and Implementation*, Vancouver, BC, 2000. ACM Press: New York, 2000.
5. Hudson R, Moss JEB. Sapphire: Copying GC without stopping the world. *Java Grande*. ACM Press: New York, 2001.
6. Hudson R, Moss JEB, Sreenivas S, Washburn W. Cycles to recycle: Garbage collection on the IA-64. *International Symposium on Memory Management*. ACM Press: New York, 2000.
7. Krintz C, Calder B. Using annotations to reduce dynamic optimization time. *ACM Conference on Programming Language Design and Implementation*. ACM Press: New York, 2001.
8. Shpeisman T, Lueh G-Y, Adl-Tabatabai A-R. Just-in-time Java compilation for the Itanium processor. *International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, Charlottesville, VA, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002.
9. Stichnoth JM, Lueh G-Y, Cierniak M. Support for garbage collection at every instruction in a Java compiler. *ACM Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999. ACM Press: New York, 1999; 118–127.
10. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley, 1999.
11. Common Language Infrastructure, ECMA-335. ECMA, Geneva, Switzerland, 2002.
12. C# Language Specification, ECMA-334. ECMA, Geneva, Switzerland, 2002.
13. *Intel Architecture Software Developer's Manual*. Intel Corporation, 1997.
14. *IA-64 Architecture Software Developer's Manual*. Intel Corporation, 2000.
15. GNU Classpath. http://www.classpath.org.
16. Open CLI Library (OCL). Intel Corporation, 2002. http://sf.net/projects/ocl.
17. Adl-Tabatabai A-R, Bharadwaj J, Chen D-Y, Ghuloum A, Menon V, Murphy B, Serrano MJ, Shpeisman T. StarJIT: A dynamic compiler for managed runtime environments. *Intel Technology Journal* 2003; **7**:19–31.
18. Hudson R, Moss JEB. Incremental collection of mature objects. *International Workshop on Memory Management*, 1992.
19. The JIT Compiler Interface Specification. Sun Microsystems. http://java.sun.com/docs/jit_interface.html.
20. Microsoft SDK for Java. Microsoft Corporation. http://www.microsoft.com/java/sdk.
21. Wilson PR. Uniprocessor garbage collection techniques. *Technical Report*, University of Texas, January 1994. Available at: ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps.
22. Shudo K. Performance comparison of JITs. http://www.shudo.net/jit/perf. [January 2002].
23. Jikes Research Virtual Machine. IBM. http://www.research.ibm.com/jikes.
24. Java 2 Platform, Standard Edition (J2SE) v 1.4.1. Sun Microsystems, 2002.
25. WebLogic JRockit JVM Version 1.3.1. BEA, 2002. http://commerce.bea.com/downloads/weblogic_jrockit.jsp.
26. SPEC JVM98. Standard Performance Evaluation Corporation. http://www.spec.org/jvm98.
27. VTune™ Performance Analyzer. http://developer.intel.com/software/products/vtune/.
28. SPEC Java Business Benchmark 2000. Standard Performance Evaluation Corporation. http://www.spec.org/jbb2000.

