JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 9
Special issue: .NET Technologies 2004 workshop, Pilsen

Experience Integrating a New Compiler and a New Garbage Collector Into Rotor

Todd Anderson, Marsha Eng, Neal Glew,
Brian Lewis, Vijay Menon, and James Stichnoth
Microprocessor Technology Lab, Intel Corporation

Microsoft's ROTOR is a shared-source CLI implementation intended for use as a research platform. It is particularly attractive for research because of its complete implementation and extensive libraries, and because its modular design allows different implementations of certain components such as just-in-time compilers (JITs). Our group has independently developed our own high-performance JIT and garbage collector (GC) and wanted to take advantage of ROTOR to experiment with these components in a CLI environment. In this paper, we describe our experience integrating these components into ROTOR and evaluate the flexibility of ROTOR's design toward this goal.

We found it easier to integrate our JIT than our GC because ROTOR has a well-defined interface for the former but not the latter. However, our JIT integration still required significant changes to both ROTOR and our JIT. For example, we modified ROTOR to support multiple JITs. We also added support for a second JIT manager in ROTOR, and implemented a new code manager compatible with our JIT. We had to change our JIT compiler to support ROTOR's calling conventions, helper functions, and exception model. Our GC integration was complicated by the many places in ROTOR where components make assumptions about how its garbage collector is implemented, as well as ROTOR's lack of a well-defined GC interface. We also had to reconcile the different assumptions made by ROTOR and our garbage collector about the layout of objects, virtual-method tables, and thread structures.

1 INTRODUCTION

ROTOR, Microsoft's Shared Source Common Language Infrastructure [7, 9], is an implementation of CLI (the Common Language Infrastructure [6]) and C# [5]. It includes a CLI execution engine, a C# compiler, various tools, and a set of libraries suitable for research purposes (it omits a few security and other commercially important libraries). As such, it provides a basis for doing research in CLI implementation, and Microsoft encourages such use of ROTOR.

For a number of years, our group has been researching and implementing managed runtime environments for Java and CLI on Intel platforms. Recently, as part of this effort, we developed a high-performance just-in-time compiler (JIT), called STARJIT [1], that can compile both Java and CLI applications, and a high-

Cite this article as follows: Todd Anderson, Marsha Eng, Neal Glew, Brian Lewis, Vijay Menon, James Stichnoth: "
Experience Integrating a New Compiler and a New Garbage Collector Into Rotor", in *Journal of Object Technology*, vol. 3, no. 9, October 2004, Special issue: .NET Technologies 2004 workshop, pp. 53–70, http://www.jot.fm/issues/issue_2004_10/article3

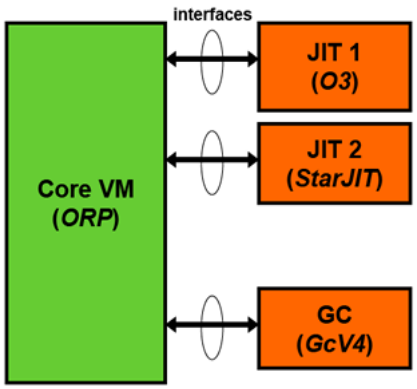


Figure 1: ORP's use of interfaces for modularity

performance garbage collector (GC), called GcV4. Because Rotor provides a complete platform for CLI experimentation, we set out to integrate StarJIT and GcV4 with Rotor on the IA-32 architecture and to see how well our technologies work in the Rotor environment. The integrated Rotor/StarJIT/GcV4 uses StarJIT instead of FJIT (Rotor's own JIT), GcV4 instead of Rotor's own garbage collector, and Rotor as the core VM. Ideally the latter would be unchanged. This paper describes our experience and presents our observations on the suitability of Rotor as a research platform.

StarJIT and GcV4 were originally developed for use with our virtual machine, ORP (the Open Runtime Platform [3]). ORP was first designed for Java and later adapted to support CLI as well. One of ORP's key characteristics is its modularity: ORP interacts with JITs and GCs almost exclusively through well-defined function-call interfaces. Each component uses these interfaces to request action from other components. For example, ORP calls an interface function to request a JIT to compile a method or to request the GC to allocate heap storage. Similarly, a JIT can request ORP to look up method names, and the GC can have ORP enumerate the on-stack and in-register object references during a garbage collection. As shown in Figure 1, ORP's use of interfaces cleanly separates the core VM from particular JITs or GCs, and enables component substitutability. The only exceptions to ORP's strict use of interfaces are ORP's assumptions about the layout of a small number of performance-critical data structures including object headers, *vtables* (virtual-method tables), and some GC information stored in vtables. Cierniak *et al.*describe in detail how ORP uses interfaces to support flexibility while preserving high application performance [3].

We hoped the use of these interfaces by StarJIT and GcV4 would simplify their integration into Rotor. Rotor also has a well-defined JIT interface, but it lacks a well-defined interface for garbage collectors. Some of Rotor's interfaces are defined directly in terms of internal data structures and other details of the VM, but others are more abstract, using opaque handles and separating the VM cleanly

from other components. Using these abstract interfaces, JITs such as StarJIT can be built independently of Rotor itself, and loaded as DLLs (dynamically-linked libraries) at runtime.

Our ultimate goal is to see how well StarJIT's and GcV4's Java optimizations apply to CLI and what further optimizations for CLI can be developed. StarJIT includes advanced optimizations such as guarded devirtualization; synchronization optimization; Class Hierarchy Analysis (CHA [4]); elimination of runtime null pointer, array index, and array-store checks; and dynamic profile-guided optimization (DPGO). GcV4 performs parallel sliding compaction to maximize application throughput. StarJIT and GcV4 can collaborate to insert prefetching based on dynamic profiles of cache misses [2]. All these optimizations are important to managed languages like Java and C#.

Overall, JIT integration was more straightforward than GC integration because the Rotor JIT interface is well defined. In contrast, integrating the GC required many intricate changes that were interspersed throughout the Rotor source code. In both cases, however, we found our work complicated by missing functionality. We start by describing our integration effort at the conceptual module level, then delve into the details of which methods and data structures were modified to make integration possible at for the JIT during both compile and run time, and for the GC.

In the descriptions in the remainder of this paper, we explicitly provide the names of Rotor data structures and source files, to provide specific landmarks for others who would like to make similar modifications and experiments on the Rotor code base.

2 INTEGRATION AT THE MODULE LEVEL

A key goal of our JIT and GC integration efforts was to minimize changes to Rotor's code base, especially the core VM. We realized some modifications to StarJIT and GcV4 would be necessary as a result, so our secondary goal was to avoid making extensive changes to these modules.

Figure 2 shows the general structure of Rotor. The core VM components that, for example, load assemblies and types, return information about methods and fields, and handle exceptions are highlighted in the middle of the figure. Rotor's GC is shown on the left-hand side, as is the garbage collected heap. The three components that together implement Rotor's FJIT ("fast JIT") and manage its compiled code are highlighted on the right-hand side of the figure.

JIT-Related Modifications

Rotor divides the compilation and management of compiled code into three com-

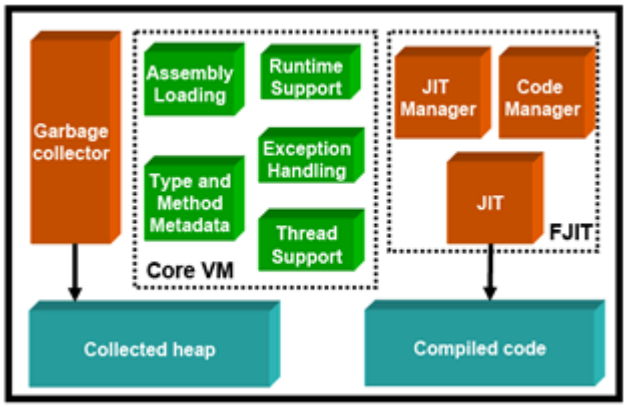


Figure 2: The structure of ROTOR including the three components of FJIT

ponents: JITs, JIT managers, and code managers. A *JIT* compiles CLI bytecodes into native code. A *JIT manager* allocates and manages space for a JIT's compiled code, data, and exception-handler and garbage-collection information. A *code manager* is responsible for stack operations involving the frames of compiled code that it manages. Each of these components implements the associated ROTOR interface, which is a C++ abstract base class. The ROTOR JIT design is general, and there is no reason why it cannot support multiple JITs, multiple JIT managers, multiple code managers, JITs that share JIT and code managers, *et cetera*. Currently, ROTOR has one JIT, two JIT managers, and one code manager.

To implement a JIT, JIT manager, or code manager, one writes a C++ class that implements the appropriate interface (abstract base class). ROTOR also defines another interface layer on top of the JIT interface. This layer allows JITs to be implemented in DLLs and hides the details of ROTOR's types for classes, methods, fields, *et cetera*, with the use of handles such as `CORINFO_CLASS_HANDLE`, `CORINFO_METHOD_HANDLE`, and `CORINFO_FIELD_HANDLE`. Throughout the paper, we use the term *JIT interface* to refer to this layer. In contrast, ROTOR's JIT manager and code manager interfaces have no such layer above them and use ROTOR's internal data structures directly, making them difficult to place in DLLs.

We found that most of the STARJIT integration effort centered around the JIT interface, which is defined in `corjit.h` and `corinfo.h`. These files define a number of interface classes, all of whose names begin with the letter `I` (*e.g.*, `ICorClassInfo`). The JIT must implement the interface class in `corjit.h` and can communicate with the VM using the interface classes in `corinfo.h`.

To date, we have succeeded in using ROTOR's existing interface functions unmodified for method compilation. However, to do this we had to disable some STARJIT optimizations that will require extensions to this interface. For example, CHA requires the JIT to examine the currently loaded class hierarchy to detect whether a particular method in a class has been overridden by a subclass. While ROTOR's

JIT interface allows exploration *up* the class hierarchy, it currently does not allow exploration *down* the class hierarchy, precluding StarJIT's CHA.

Over the years, we have implemented at least four separate JITs for ORP. One of our most valuable JIT debugging tools has been support for multiple JITs. This approach allows several different JITs to be present in the system at the same time and allows different methods to be compiled by different JITs. Although our multiple JIT support allows more than two JITs to coexist, for our integration work we use just two JITs: StarJIT and Rotor's built-in FJIT. We define an ordering for the JITs: the experimental JIT (StarJIT) comes first, the next JITs are more stable (not applicable for this project), and the most reliable JIT (FJIT) comes last. Rotor first calls StarJIT to compile each new method. If this is unsuccessful (either because of missing StarJIT functionality, or because StarJIT generates incorrect code, or because StarJIT otherwise refuses to compile the method), it returns a special error code and Rotor calls FJIT. If there were more than two JITs in the system, the VM would call each JIT in order until one successfully compiled the method. This support for multiple JITs allows us to continue debugging the experimental JIT without being stuck if it fails to compile some method because of a hard-to-fix bug. We can continue to identify and fix other problems with the experimental JIT while waiting for a fix to the first problem.

StarJIT includes a *method table* mechanism for providing fine-grain control over which methods it chooses to compile and which it rejects. This mechanism is controlled via a property file loaded by StarJIT at runtime. Typically, we first run StarJIT configured to generate a method table file. This file is simply a dump of all the methods names, one per line of the file. Then we can configure StarJIT to compile specific methods only by specifying the lines in the file that name the methods we are interested in. For example, the configuration string "`METHODS=methods.txt:10-20,30,40-50`" instructs StarJIT to compile only the methods listed in lines 10–20, line 30, and lines 40–50 of the `methods.txt` file, and to reject all other methods. We use binary search to isolate bugs. If we know that StarJIT has a bug when run with certain lines of the file, we can run with half of these lines specified. If the bug manifests then we repeat with these lines, otherwise we repeat with the other half. The configuration string also allows specifying methods by name, in addition to referencing methods listed in the file. This technique of debugging a new JIT with the use of method tables and a robust backup JIT proved to be invaluable to our integration effort.

GC-Related Modifications

Unlike for JITs, there is no clean interface in Rotor for a garbage collector to communicate with the rest of the system. The Rotor GC is responsible for both object allocation and garbage collection, and also interacts with the threading subsystem. As such, it has many touch points with the VM and more extensive modifications of Rotor were required for integrating GcV4.

Garbage-collection problems can be notoriously difficult to debug, since a problem introduced during a collection may not manifest itself until much later, and the method where the problem manifests itself may have little to do with the method in which the problem actually arose. For debugging such problems, we found it useful to use built-in Rotor functionality for forcing collections at more regular intervals. Rotor has a `GCStress` parameter that can be given various settings. One especially useful setting forces a collection every time an object is allocated. This setting often causes garbage collection problems to show up soon after they occur, when the information needed to debug them is still available.

3 JIT COMPILE-TIME INTERFACE

As previously mentioned, a major part of the StarJIT integration was adapting StarJIT to Rotor's JIT interface. This adaptation included implementing the function to compile a method, and modifying StarJIT to use the set of functions that Rotor provides for querying classes, fields, methods, *et cetera*.

Although the StarJIT integration is still under development, we have successfully compiled and run enough programs that we believe the integration is nearly complete. Despite some initial difficulty understanding the semantics of a few of Rotor's JIT interface functions, our experience has been predominantly positive. This section discusses our modifications and the problems we found.

Supporting the JIT Compile-Time Interface

StarJIT already includes an internal interface, `VMInterface`, that it uses to isolate itself from any particular VM. The ORP version of StarJIT, for example, is built with an ORP-specific implementation of this interface. The main part of our effort was spent implementing a Rotor-specific implementation of `VMInterface`.

`VMInterface` includes about 160 methods. The majority of these methods resolve classes and get information about methods, fields, and other items during compilation. One `VMInterface` method returns the address of the different runtime helpers, and is described in detail in the next section. Various StarJIT optimizations are supported by other `VMInterface` methods. One such support method returns a method's execution frequency and is used in profile-based recompilation.

Most of the `VMInterface` implementation for Rotor was straightforward. However, the `VMInterface` implementation is not yet complete—we have not implemented specialized support for certain optimizations that are currently disabled.

To support StarJIT's requirements, we found two cases where it was necessary to define new data structures in the `VMInterface` implementation to augment the corresponding Rotor information. In the first case, while Rotor provides a way to break types down, it does not provide a way to build them up. At start up time,

StarJIT builds data structures that represent the primitive types, but Rotor provides no way for another component to obtain handles (CORINFO_CLASS_HANDLE) for these types. This means that StarJIT's type data structures cannot simply be Rotor handles, and instead we designed a RotorTypeInfo data structure that includes enough information for StarJIT's needs. We can build one of these structures without a Rotor class handle. A RotorTypeInfo cannot answer all queries on types, but in these cases StarJIT would have been given a Rotor handle and Rotor can be queried instead.

In the second case, StarJIT needs the type of the this argument for many methods. In Rotor, this type cannot be obtained using the signature information (CORINFO_SIG_INFO) for a method. Our solution is to represent a method's signature using a tuple that contains both a CORINFO_SIG_INFO (for arguments other than this) and a CORINFO_METHOD_HANDLE (to get the type for this). This tuple and the RotorTypeInfo tuple are similar to the OpType tuple class used in Rotor's built-in FJIT.

Conclusions About the JIT Compile-Time Interface

In summary, we found Rotor's compile-time JIT interface (ICorJitInfo) generally well designed. However, some information needed for optimizations is missing. It was also necessary to work around some limitations such as the inability of a JIT to get handles for primitive classes. We have the impression that ICorJitInfo is narrowly defined to provide just the functionality needed for FJIT. While this makes the interface simple, it complicates adding new, more optimizing JITs to Rotor.

The ICorJitInfo class inherits from a number of abstract superclasses that each define functions in various areas of compile-time information (such as methods, modules, fields) and areas of runtime information (such as helper functions and profiling data). We expect to add support for our optimizations by adding a new superclass. This will contain, for example, methods to get class hierarchy and profile-based recompilation information.

The lack of documentation about Rotor's internals was another obstacle. While the book, *Shared Source CLI Essentials* [9], is a great help, too often we resorted to experimentation to discover what Rotor functions to use. To be more widely successful as a VM intended for research, Rotor needs better documentation.

4 JIT RUNTIME INTERFACE

Besides the compile-time cooperation described earlier, StarJIT and Rotor must also cooperate at runtime. For example, although StarJIT generates code for managed methods, StarJIT and its generated code rely on Rotor for VM-specific operations such as object allocation. Similarly, the Rotor VM handles stack unwinding and root-set enumeration but it relies on the JIT to interpret individual

stack frames. This section describes the runtime support needed to integrate StarJIT into Rotor.

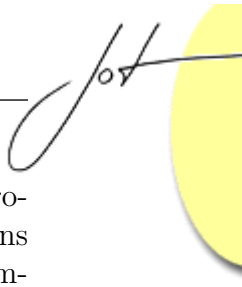
Helper Calls

The JIT-compiled code of StarJIT and Rotor's FJIT both rely on calls to runtime-helper routines to perform VM-specific operations and to perform some commonly-used complex operations. These helper calls do such things as allocate objects, throw exceptions, do `castclass` or `isinst` operations, and acquire or release locks. Common complex operations include 64-bit operations on a 32-bit architecture. Rotor provides a mechanism to query for helpers in its `ICorInfo` interface. StarJIT's Rotor-specific `VMInterface`, in turn, maps StarJIT helpers to Rotor ones. During our integration work, we encountered several issues specific to helper calls. In most cases, we were able to solve these issues within the Rotor-specific `VMInterface` layer.

The first issue we encountered involved the different calling conventions used by ORP and Rotor. StarJIT had been hardwired to use the ORP conventions when calling VM helper functions as well as other managed code. Instead, we modified StarJIT to use a new interface that allows the VM to abstractly describe the calling conventions to the JIT for any call. This description includes a specification of which arguments are passed in which registers, which are passed on the stack and in which order, how return values are returned, and whether the caller or callee pops arguments from the stack.

A second issue we discovered involved differences in both the required parameters and their order for different helpers. For example, ORP's `rethrow` helper requires the exception as a parameter but Rotor's does not. In addition, ORP and Rotor's `castclass` helpers have the object and type descriptor in different orders. We considered the use of wrapper stubs to convert between one set of conventions and the others. However, these wrappers complicate stack unwinding and incur additional performance overhead. Instead, we modified StarJIT via `#ifdef` to use Rotor's conventions. A more general approach would have an interface that allows the VM to communicate the parameter conventions for the VM helper calls to the JIT.

There are a couple of differences between Rotor and ORP related to type-specific helpers. A number of helpers, including the ones for object allocation, type checks, and interface table lookups, involve types that are known at compile time. In these cases, Rotor returns different helpers for different types, based on a type passed in at compile time. Accordingly, we modified the helper function lookup in StarJIT's `VMInterface` to require a type for all type-related helpers. There are also differences in exactly which of several type-related data structures are passed at compile time or runtime to these helpers. We abstracted this detail into `VMInterface` so that the VM-specific code can give StarJIT the correct data structure to pass.



Another challenge involved helpers that StarJIT expected that were not provided by Rotor. In most cases, these were helpers for 64-bit integer operations (*e.g.*, shifts) not provided by Rotor. In these cases, the helper could easily be implemented within the Rotor-specific `VMInterface`. Some other cases reflect a more serious mismatch between StarJIT and Rotor. For example, Rotor provides an `unbox` helper that performs the necessary type check on a reference and then unboxes it. In StarJIT, however, the type check and the actual unbox are broken into separate operations at an early point with the hope of statically removing the type check via optimization. StarJIT expects a helper to perform the unbox-specific type check but generates a simple address calculation to do the actual unboxing. Rotor, on the other hand, only provides a helper to perform the entire unbox. For now, we use the `castclass` helper instead to perform the unbox type check. However, this approach fails when the unboxed reference is a boxed enumeration type and will have to be corrected.

Finally, there are a number of helpers that Rotor provides that are not currently invoked by StarJIT. Some of these additional helpers are provided only to simplify portability: without them, Rotor's FJIT would need assembly sequences specific to IA-32 and to PowerPC. Other helpers assist in debugging, while still more support additional functionality such as remoting. Up to this point, none of the applications that we have tried to execute with StarJIT have needed the additional functionality provided by these helpers. However, in the future, we plan to extend ORP's `VMInterface` to enable StarJIT to query the VM and discover which of these additional helper functions must be called.

Code and JIT Managers

As part of our implementation of the multiple JIT support, we found we needed to use the other JIT manager in Rotor. We could not use a second instance of FJIT's JIT manager because its implementation uses global variables to, for example, map program counters to methods and to manage memory. Two instances would have conflicting uses of these variables.

Another part of the runtime interface concerns stack walking activities such as root-set enumeration, exception propagation, and stack inspection. The Rotor design, like many other VMs, divides this task into one part that loops over the stack as a whole and another part that deals with individual stack frames. The loop part is in the VM proper and rightly so. Conversely, processing an individual stack frame depends upon the JIT's stack conventions (*e.g.*, where local and temporary variables of reference type are located and the location of callee-saves registers) and therefore requires the JIT's cooperation. In Rotor, all processing of individual stack frames is done by the code manager.

The code manager that comes with Rotor makes many assumptions about JIT-compiled code for the IA-32 architecture:

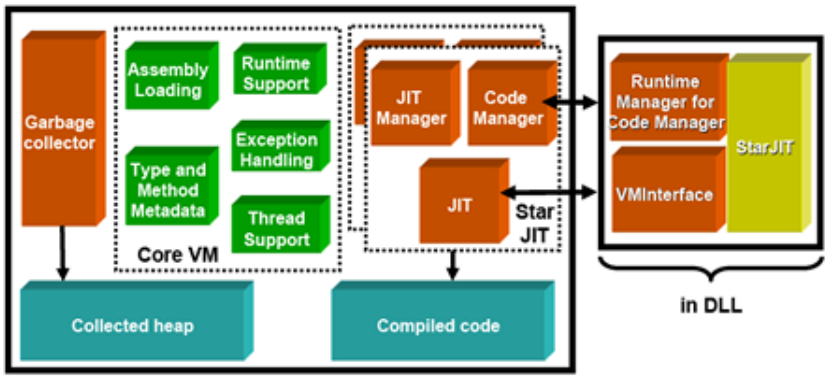


Figure 3: The structure of Rotor with StarJIT loaded from a separate DLL

- The code for each method is expected to consist of a prologue, followed by the body, followed by an epilogue.
- Only one epilogue may exist, and it must appear at the end of the compiled code.
- The prologue and epilogue are precisely defined code sequences; no deviations are allowed.
- Only `ebp` and `esi` are saved and available for use; `ebp` is used as a frame pointer, while `esi` is always a valid object reference (but possibly NULL). Registers `ebx` and `edi` may not be used.
- The security object is at address `ebp-8`.
- JITs give root-set information to the JIT manager in the form of an *info block*, which the JIT manager then passes to the code manager during root-set enumeration. This information is expected to match the particular structure of Rotor's JIT.

These assumptions of Rotor's code manager fundamentally conflict with those of StarJIT. We therefore decided to write our own code manager. This code manager has to be part of the VM, but we decided to try emulating Rotor's interaction with the JIT by having this new code manager simply convert all its calls into calls to a runtime manager placed in the same DLL as the matching JIT. We defined an interface along the lines of `corjit.h`, and we allow a DLL to export a runtime manager as well as a JIT. The resulting architecture of Rotor with StarJIT is shown in Figure 3. Note that StarJIT uses a separate set of the three JIT-related components from FJIT, and that StarJIT's JIT and Code Manager components communicate with the actual implementations that are loaded from a separate StarJIT DLL.

We found this approach generally straightforward. However, the parameters passed to different code manager methods are inconsistent. For example, the method `UnwindStackFrame` gets an `ICodeInfo` object, which can be used to identify the method and some of its attributes, but `FixContext` does not. Also, these methods need to know the current values of registers for the frame that they are unwinding, fixing up, or enumerating the roots of, and there are different types of contexts for `FixContext` versus `UnwindStackFrame` and most of the other methods. We decided to reflect these inconsistencies in the external interface. Since StarJIT's runtime interface is more uniform and requires the method handle for the method of the frame, we used the info block to pass the missing information from compile time to run time.

Another minor point is that `UnwindStackFrame` is sometimes called with the context `esp` equal to either the address just above the arguments of the out-going call, or the lowest address of the out-going arguments. In general, there is no way to tell which of the two cases holds. This situation is fine if frame pointers are used; the context `ebp` can be used to find everything in the frame. However, requiring frame pointers on IA-32 reduces the number of usable registers from 7 to 6. For now, we have modified StarJIT to use frame pointers.

Exception Handling

Another significant difference between Rotor and StarJIT concerns the details of exception propagation. Here, the differences stem directly from the characteristics of CLI and Java. In CLI, there are exception handlers, filters, finally blocks, and fault blocks. Each of these is a separate block of bytecode from the region being protected, and control cannot enter these blocks except through the exception mechanism. Conversely, in Java, there are only exception handlers and these protect a region of bytecode. When an exception is caught in Java, control is transfered to a handler address which can be anywhere in the method's bytecode.

Since StarJIT was developed against the interfaces of ORP, which originally supported Java and was later adapted to also support CLI, StarJIT's design reflects the Java exception mechanism. First, StarJIT implements finally and fault blocks by catching all exceptions and then rethrowing them. This behavior is close to but not exactly that required by the CLI specification, although it is correct for code compiled from C#. Second, there is a particular bytecode for leaving an exception handler and returning to the "main" code (a `leave`). Rotor requires the JIT at such a bytecode to call the runtime helper `EndCatch`. This helper cleans up stack state generated by the VM for exception handling and ensures that finally blocks are called. We modified StarJIT to call this helper since ORP does not have a corresponding helper. Finally, Rotor needs an exception handler to be compiled to a contiguous region of native code and it needs to know the start and end addresses of that region. StarJIT knows the start address, but not the end address, and might rearrange blocks so that a handler is no longer contiguous. We do not have

a solution for this problem yet. For now, we give a zero end address—this causes Rotor to compute incorrect handler nesting depths, but otherwise seems to have no ill effect.

Conclusions About the JIT Runtime Interface

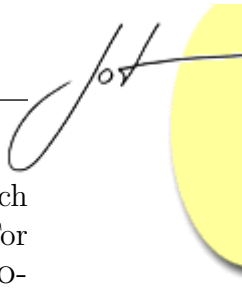
Rotor could be significantly improved through better support for multiple JITs. Key additions would be the following, all of which we have prototyped:

- A mechanism for loading more than one JIT from a DLL and trying each JIT in turn when compiling a new method.
- The ability to load an independently written JIT manager and code manager from a separate DLL (presumably packaged in the same DLL that holds the corresponding JIT). These should interact with the VM through an abstract interface such as those in `corjit.h` and `corinfo.h`.
- A more consistent set of parameters for the code manager functions, to make for a more uniform interface.

Our experience integrating StarJIT with Rotor also led to changes in StarJIT. For example, StarJIT's `VMInterface` had to be generalized to better support requests for type-specific helpers. We also found that StarJIT should allow calling conventions to be specified by the VM. Currently, we use `#ifdef`s in StarJIT's source code to control calling conventions, but this makes the code hard to maintain and the resulting code less flexible. If StarJIT queried the VM about the calling conventions to use, it could adapt itself dynamically to the needs of the VM. Also, the design of a clean and flexible runtime helper interface is an interesting problem, and one we would like to address.

5 GC INTEGRATION

Rotor includes a C++ interface class, `GCHeap`, that declares most of the GC-related methods that the rest of Rotor uses. However, this GC interface is not as complete, explicit, or cleanly-defined as its JIT interface. In addition, Rotor does not support the dynamic loading of garbage collectors from DLLs. As a result, to integrate our GcV4 garbage collector into Rotor, we needed to add the GcV4 code directly to the Rotor VM code base. Our integration work involved three sets of changes: revising the implementation of Rotor's GC interface to use GcV4, changing the rest of Rotor to use a new GC with different assumptions, and modifying GcV4 to run inside Rotor. This section discusses our experience integrating this collector, including the issues we encountered and our solutions.



Probably the most significant issue we found is that ROTOR exposes too much about the implementation of its collector to other components in the system. For example, examining the methods in ROTOR's GCHeap interface class reveals that ROTOR assumes a generational collector that treats large objects differently than small ones, and that allows clients to query whether an object is part of the ephemeral generation. Much of this is likely to change if ROTOR's GC is replaced with another GC. As another example, the ROTOR VM uses knowledge about the collector's implementation to allow JITs to emit optimized code. The VM's function JIT_TrialAlloc::EmitCore can be called by JITs to emit code for the allocation fast path for many types of objects. That code assumes intimate knowledge of the GC's data structures and object-allocation strategies.

Although ROTOR's GCHeap interface does not hide enough about the implementation of its GC, the GCHeap interface did help to reduce the number of changes we needed to make to the rest of ROTOR. GCHeap is a public interface that gives other components access to most of ROTOR's memory management functionality, for example, object allocation and the registration of objects to be finalized. The GCHeap implementation often makes calls on a low-level C++ class, gc_heap, that exposes the low-level methods and data structures of ROTOR's own GC implementation. By modifying GCHeap's implementation to call GCV4 methods instead of those in gc_heap, we minimized the number of changes required to ROTOR and were able to localize many of the changes to just the GC-related files.

Other changes were needed to the ROTOR VM. For example, we added calls to initialize and close down GCV4. We modified ROTOR's thread constructors and destructors to keep GCV4 up-to-date with respect to thread existence. Finally, we modified JIT_TrialAlloc::EmitCore to no longer make assumptions about the collector's data structures. Instead, the allocation stub emitted by EmitCore now first calls a "fast" GCV4 allocation function that succeeds if sufficient memory is readily available, but returns NULL otherwise. If this fast allocation routine fails, the stub emitted by EmitCore falls through to a slow-path allocation routine that may trigger a garbage collection. While we initially hard-coded the name of the GCV4 fast-path allocation routine in EmitCore, we soon realized that this interaction between EmitCore and the GC could be generalized by adding a method to GCHeap that returns a pointer to the GC's fast allocation function.

Similarly, GCV4 expects the VM to supply a number of functions. One especially important function, used at the start of a garbage collection, requests that the VM stop all threads and enumerate all roots. Since stopping (and restarting) threads in ROTOR requires a very specific sequence of events, we reused much of the existing ROTOR code for this purpose. We also reused the two CNameSpace methods GcScanRoots and GcScanHandles to do root-set enumeration by passing them our own GCV4 callback function instead of ROTOR's one.

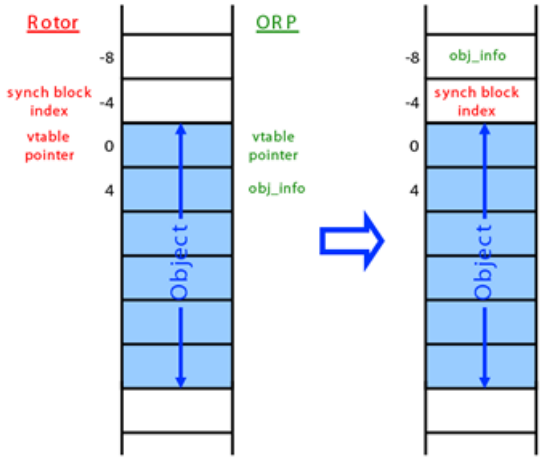


Figure 4: ROTOR and ORP expect different object layouts; our final layout

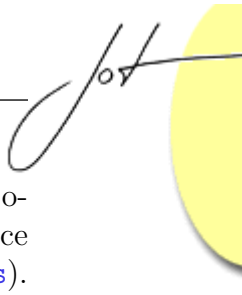
Integration Issues and Solutions

In the course of our integration, we found a number of conflicts between the assumptions made by GCV4 and ROTOR about the layout of several key data structures. These are listed below along with our solutions.

- *Object Layout.* Since GCV4 was originally developed for ORP, GCV4 expected objects to use ORP's memory layout. Moreover, GCV4 assumed that each object began with a pointer to the vtable, followed immediately by ORP's multi-use `obj_info` field. This field holds synchronization, hash code, and garbage collection state, and so resembles ROTOR's *sync block index*. However, ROTOR places other object data at a four byte offset while ROTOR expects the sync block index to be at a four byte *negative* offset from the start of an object. Realistically, too many parts of ROTOR depend on this layout to change it. Also too many parts of ROTOR use the sync block index in ways incompatible with GCV4's use of the `obj_info` field, so mapping `obj_info` to the sync block index is not a solution. The different layouts expected by ROTOR and ORP are shown in the left-hand side of Figure 4.

  Our solution was to place the ORP `obj_info` field before each object, at a negative eight offset from the object's vtable pointer. This offset does not conflict with any part of ROTOR's object layout. As a result, no ROTOR component is aware of the extra field. The right-hand side of Figure 4 shows our final object layout layout.

- *Vtable Layout.* GCV4 assumed that the first four bytes of each vtable is a pointer to a structure containing GC-related information that indicates, for example, whether the object contains pointers and if so, the offset of each pointer.



The start of ROTOR's `MethodTable` structure, however, contains the component size (for array objects and value classes), the base size of each instance of this class, and a pointer to the corresponding class structure (`EEClass`). There are many places in ROTOR that assume specific offsets to these fields, so changing the field layout would raise many problems.

Storing the pointer at a negative offset from the start of the vtable is also not an option. That would interfere with ROTOR's `CGCDesc` and `CGCDescSeries` structures, which are stored before the vtable if the class contains pointers. These structures are used by FJIT as well as ROTOR's collector, so we could not use that space for our pointer.

We solved this by reserving space in ROTOR's `MethodTable` class at a sufficiently high offset to avoid conflicts with ROTOR's fields.

- *Thread Layout.* GCV4 assumed that a portion of each thread's data structure is storage reserved for its use, which is is an essential part of ORP's object allocation and garbage collection strategies. However, ROTOR does not have an analogous field in its thread data structure. Our solution was to add the extra storage at the end of thread objects.

Conclusions About the GC Integration

ROTOR should expand and abstract its GC interface so that it more resembles its JIT interface. The parts of the current interface that are only applicable to one style of garbage collector should be removed and replaced with more generic versions that support the introduction of different kinds of garbage collectors. Moreover, ROTOR should use functions to abstract the interactions between the collector and other ROTOR components. These functions would hide details about the collector's implementation and help to make explicit the assumptions it makes. Such a GC interface would make it easier to modify the collector and to experiment with new implementations without affecting other components. Our experience with ORP's GC interface has been strongly positive, and it has allowed us to use several different collector implementations without changing the VM or JITs.

To enable easier GC experimentation, it would help if ROTOR's GC could be dynamically loaded like its JITs. New collectors could be plugged in to ROTOR including ones tailored for particular needs, such as when an application needs short GC pause times more than high throughput. Changing ROTOR to dynamically load its GC would also help to minimize assumptions made by the VM or other components.

To minimize the problems caused by assumptions about the layout of key data structures, ROTOR's GC interface could include a function that returns the offset of such fields as its sync block index. This would avoid other components assuming a fixed constant for that value. A similar interface function would also help ORP, GCV4, and STARJIT by reducing the number of their layout assumptions.

6 STATUS AND FUTURE WORK

When we started our integration work, we wondered how suitable Rotor would be as a research platform. That is, how difficult would it be to add our optimizations and what changes to Rotor would be needed to support them? Our plans were to add StarJIT and GcV4, then later implement in Rotor a number of optimizations such as our synchronization techniques, prefetching, and DPGO. This paper describes the approaches we took to integrate StarJIT and GcV4, and our experience with that effort.

The StarJIT integration was straightforward except for a few issues. While most of the changes needed were within StarJIT, we found that we had to modify Rotor to add support for multiple JITs and to add a new code manager for StarJIT. We also needed to support another JIT manager in Rotor—because we could not create another instance of FJIT's JIT manager since its implementation depends on global variables. Although Rotor allows JITs to be loaded dynamically, and communicates with those JITs using its abstract JIT interface, Rotor does not allow JIT or code managers to be loaded dynamically. Adding new code or JIT managers requires modifying Rotor itself, although abstract interfaces for these managers could be added to Rotor without much trouble. Later, we expect to add support for some of the more sophisticated StarJIT optimizations such as DPGO by augmenting Rotor's JIT interface with a new abstract superclass that defines the required functions.

We found that adding a new garbage collector to Rotor was much more difficult than integrating a new JIT. Rotor does not have a clean interface for GCs that resembles its JIT interface. Its `GCHeap` class, for example, exposes details about the GC's implementation that are used by several other parts of the system including FJIT, so adding a different implementation required changing those parts. We tried to minimize the changes to Rotor, but a number of changes were needed, for example, to have Rotor call functions in the GC interface that GcV4 exports. Both Rotor and GcV4 make assumptions about the layout of objects and virtual-method tables, so it was necessary to modify our GcV4 implementation to place the fields that GcV4 needs (such as one used to hold a forwarding pointer during collections) in locations that do not conflict with fields required by Rotor.

Our work integrating StarJIT and GcV4 with Rotor is ongoing. We can run a number of test programs and are currently getting our modified Rotor to work with the C# version of the SPEC JBB2000 benchmark [8]. Our plans for StarJIT include adding support for pinned objects and full support for CLI exceptions (such as filters), as well as support for our optimization technologies such as DPGO and prefetching. We are optimistic about being able to complete this work and look forward to exploring other opportunities for improving Rotor's performance.

REFERENCES

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1), February 2003. Available at http://intel.com/technology/itj/2003/volume07issue01/art02_starjit/p01_abstract.htm.

[2] A.-R. Adl-Tabatabai, R. Hudson, M. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *SIGPLAN Conference on Programming Language Design and Implementation*, Washington, DC, USA, June 2004.

[3] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003. Available at http://intel.com/technology/itj/2003/volume07issue01/art01_orp/p01_abstract.htm.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, Aug. 1995. Springer-Verlag (*LNCS* 952).

[5] ISO/IEC 23270 (C#). ISO/IEC standard, 2003.

[6] ISO/IEC 23271 (CLI). ISO/IEC standard, 2003.

[7] Microsoft. Shared source common language infrastructure. Published as a Web page, 2002. See http://msdn.microsoft.com/net/sscli.

[8] Standard Performance Evaluation Corporation. SPEC JBB2000, 2000. See http://www.spec.org/jbb2000.

[9] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly, Mar. 2003.

ABOUT THE AUTHORS

Todd Anderson is a staff researcher in Intel's Programming Systems Lab after joining Intel in 1999. He received his M.S. and Ph.D. degrees in Computer Science from the University of Kentucky. Todd has worked in a variety of areas including distributed file systems, distributed computing, IETF forwarding/control separation and is currently focused on memory management in managed runtime environments.

Marsha Eng is a researcher in Intel's Programming Systems Lab. Marsha joined

Intel in 2001, with an M.S. degree in Computer Engineering from the University of California, San Diego, and a B.S. degree, also in Computer Engineering, from the University of Washington.

Neal Glew is a staff researcher in Intel's Programming Systems Lab. He received a Ph.D. degree in Computer Science from Cornell University in January 2000.

Brian Lewis is a senior staff researcher in Intel's Programming Systems Lab. Brian joined Intel in 2002. He previously worked at Sun, Olivetti Research, and Xerox. While at Sun Microsystems Laboratories, Brian worked on the development of virtual machines for several languages. He also worked on techniques for binary translation as well as portions of the Spring research operating system. Brian received a Ph.D. and M.S. degree in Computer Science and a B.S. degree in Mathematics from the University of Washington.

Vijay Menon is a staff researcher in Intel's Programming Systems Lab. He received a B.S. from the University of California, Berkeley in Electrical Engineering and Computer Science and a Ph.D. from Cornell in Computer Science. His current research interests include program analysis, dynamic compilation, and managed runtime environments.

James Stichnoth s a senior staff researcher in Intel's Programming Systems Lab. Jim joined Intel in 1997, with a Ph.D. degree in Computer Science from Carnegie Mellon University and a B.S. degree in Computer Science from the University of Illinois at Urbana-Champaign. He has worked extensively on Virtual Machines, Just-In-Time Compilers, and development of enterprise Java applications. Jim currently leads a group researching Virtual Machine technology.