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

Todd Anderson Marsha Eng Neal Glew
Brian Lewis Vijay Menon James Stichnoth

Microprocessor Technology Lab, Intel Corporation
2200 Mission College Blvd., Santa Clara, CA, 95054, U.S.A.
james.m.stichnoth@intel.com

ABSTRACT

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 (*e.g.*, just-in-time compilers). Our group has independently developed a high-performance just-in-time (JIT) compiler and garbage collector, and wanted to take advantage of ROTOR as a platform for experimenting with these components. In this paper, we describe our experiences integrating these components into ROTOR, and evaluate the flexibility of ROTOR's modular design toward this goal.

We found the just-in-time (JIT) compiler easier to integrate than the garbage collector because ROTOR has a well defined interface for the former but not the latter. However, the JIT integration required changes to ROTOR to support multiple JITs, which included implementing a new code manager and supporting a second JIT manager. We detail the changes to our just-in-time compiler to support ROTOR's calling conventions, helper functions, and exception model. The garbage collector integration was complicated by the many places in ROTOR where components make assumptions about the garbage collector's implementation. It was also necessary to reconcile the different assumptions made by our garbage collector and ROTOR about object layout, virtual-method table layout, and thread structures.

Keywords

CLI, Java, virtual machine, just-in-time compilation, dynamic compilation, garbage collection, calling conventions, software interfaces

.NET Technologies'2004 workshop proceedings,
ISBN 80-903100-4-4

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 is encouraging this use of ROTOR.

Our group has been doing research for a number of years on the implementation of managed runtime environments for Java and CLI on Intel platforms. 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-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. 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 originally 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 interfaces.[1] We hoped the use of these interfaces by STARJIT and GCV4 would simplify our integration. ROTOR also has a well-defined JIT interface, but not one for GCs. Some of ROTOR's interfaces are defined directly in terms of internal details of the VM, but others are more abstract, using structures like 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 optimizations apply to CLI and what further

[1]The only exceptions to ORP's well-defined interfaces are its assumptions about the layouts of performance-critical data structures including object headers, vtables (virtual-method tables), and some GC information stored in vtables.

optimizations for CLI can be developed. StarJit includes advanced optimizations such as guarded devirtualization, synchronization optimization, Class Hierarchy Analysis (CHA [4]), runtime-check elimination (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, we found the JIT integration more straightforward because the Rotor JIT interface is well defined. In contrast, integrating the collector required many intricate changes, and these are interspersed throughout the Rotor source code. In both cases, however, we found our work complicated by missing functionality. We begin with some background about the integration effort, then describe in detail what was required for the JIT and the GC.

2. INTEGRATION OVERVIEW

A key goal of our JIT and GC integration efforts was to minimize changes to Rotor's code base. We also wanted to avoid making extensive changes to our own StarJit and GcV4 code bases.

2.1 JIT-Related Modifications

Rotor divides the compilation and management of compiled code into three components: JITs, JIT managers, and code managers. JITs compile CLI bytecodes into native code. JIT managers allocate and manage space for compiled code, data, exception-handler information, and garbage-collection information. Code managers are responsible for stack operations involving the frames of compiled code that they manage. The Rotor 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 class that implements the appropriate interface. The JIT interface is designed for implementation in DLLs. It also 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`. On the other hand, Rotor's JIT manager and code manager interfaces use Rotor's internal data structures directly and so are 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 only these interface functions for method compilation, and Rotor modifications have not yet been necessary.

However, certain StarJit optimizations 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 CHA.

Because of our past experience building new JITs, we implemented support for multiple JITs in Rotor. This approach allows several different JITs to be present in the system at the same time. For each new method, the VM calls the first JIT to compile the method. If the JIT is unsuccessful the VM calls the next JIT and so on until one JIT reports success. In our implementation, we give StarJit the first opportunity to compile a method. StarJit has "method table" code that allows the user to specify which methods StarJit should compile; other methods are rejected, and compiled by FJIT. As a result, if a bug is encountered, we can gradually reduce the set of methods compiled by StarJit until we locate the single method that caused the problem. This technique of debugging a new JIT with the use of a robust backup JIT proved invaluable in our integration effort.

2.2 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, 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. 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 setting we found especially useful 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 is adapting StarJit to Rotor's JIT interface. This adaptation includes providing the function to compile a method for Rotor and adapting StarJit to use the set of functions that Rotor provides for querying classes, fields, methods, *et cetera*.

While 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 experience and notes the few problems we found.

3.1 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 that, for instance, return a method's "heat" (an indication of the amount of execution time spent in the method) for 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 optimizations that are not presently enabled.

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, Rotor does not provide a way for JITs to get a handle (`CorInfoHandle`) for a primitive class. Since StarJit preloads the primitive classes at startup, we defined a new `RotorTypeInfo` data structure to represent types that include enough information to describe primitive types. When, for example, StarJit passes a `RotorTypeInfo` to the `VMInterface` method `typeHandleIsUnboxed`, the latter can recognize if the `RotorTypeInfo` represents a primitive class, and in that case return `true`. In the second case, StarJit needs the type of the `this` argument for many methods. In Rotor, this type is not derivable from the signature information (`CORINFO_SIG_INFO`) for a method. Our solution is to represent a method's signature using a structure that contains both a `CORINFO_SIG_INFO` and a `CORINFO_METHOD_HANDLE`. This technique is similar to the `OpType` tuple class used in Rotor's built-in FJIT.

3.2 Experience

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 (e.g., methods, modules, fields) and areas of runtime information (e.g., 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

This section describes the runtime support needed to integrate StarJit into Rotor. 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 relies on Rotor for such VM-specific issues as object allocation. Similarly, the Rotor VM handles stack unwinding and root-set enumeration but it relies on the JIT to interpret a given stack frame.

4.1 Helper Calls

The JIT-compiled code of StarJit and Rotor's FJIT both rely on helper calls to perform VM-specific operations (*e.g.*, to allocate objects, throw exceptions, do `castclass` or `isinst` operations, and acquire or release locks) and, in some cases, to perform common complex operations (*e.g.*, 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. To modify StarJit to use Rotor's calling conventions, an `#ifdef` was used to control the conventions it employs.

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 overheads. We instead modified StarJit to use Rotor's conventions.

There are a couple of differences between Rotor and StarJit 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. For any VM (*e.g.*, ORP) where the type is not required, that VM's StarJit`VMInterface` imple-

mentation ignores the type. 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 IA-32-specific and PowerPC-specific assembly sequences). 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.

4.2 Code and JIT Managers

As part of our implementation of 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.*, the location of callee saves registers and where local and temporary variables of reference type are located) 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:

- The code for each method is expected to consist of a prologue, followed by the body, followed by an epilogue.
- Multiple epilogues and epilogues interspersed in the body are not allowed.
- 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 allow a DLL to export a runtime manager as well as a JIT. This approach was mostly 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 the address just above the arguments of the out-going call and sometimes equal to the lowest address of the outgoing 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.

4.3 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.

4.4 Experience

ROTOR should include better support for multiple JITs. We had to modify ROTOR to try more than one JIT. We also had to add a second JIT manager, and to write our own code manager. ROTOR would be better if JIT managers and code managers could be packaged with JITs in a separate DLL, and if these could interact with the VM through an abstract interface such as those in `corjit.h` and `corinfo.h`. Furthermore, the code manager functions should have a more consistent set of parameters 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 does not have an explicit, cleanly-defined GC interface that resembles its JIT interface. ROTOR also does not support the dynamic loading of garbage collectors from DLLs. As a result, to integrate our GCV4 garbage collector into ROTOR, we added GCV4 directly to the ROTOR VM code base. Much of our effort involved adapting GCV4 to run in ROTOR and reconciling the different assumptions made by ROTOR and GCV4. This section discusses our experience integrating this collector, including the issues we encountered and our solutions.

Probably the most significant issue we found was that ROTOR exposes the implementation of its collector to other components in the system. For example, ROTOR's `GCHeap` class reveals that ROTOR uses a generational collector that treats large objects differently than small ones, and 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.

The ROTOR VM requires that the garbage collector export a number of functions. We modified ROTOR to invoke GCV4's functions instead of the corresponding ROTOR ones. The ROTOR VM now calls the GCV4 initialization function and object allocator instead of the ROTOR equivalents. We also 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. The code it generates currently directly invokes the "slow" allocation function, which we modified to call GCV4's allocator. We intend to add fastpath allocation back into the generated code, but we hope to develop an interface that will allow this to happen in a generic way to support other collectors in the future.

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.[2]

[2]In fact, `CNameSpace::GcScanHandles` ignores its callback parameter, but we modified the two functions it calls to invoke our callback function instead.

5.1 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.

  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.

- *Vtable Layout.* GCV4 assumed that the first 4 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 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.

  We also could not store the pointer at a negative offset from the start of the vtable. 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 the thread objects.

5.2 Experience

ROTOR should include a GC interface that resembles its JIT interface. That is, 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 an interface would make it easier to modify the collector and to experiment with new implementations without affecting other components. For example, ROTOR's GC interface could include a function that returns the offset of its sync block index. This would avoid other components assuming a fixed constant for that value. Our experience with ORP's GC interface has been strongly positive, and it has allowed us to use several different collector implementations without changing its 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 high throughput more than short GC pause times. Changing ROTOR to dynamically load its GC would also help to minimize assumptions made by the VM or other components.

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 initially to add STARJIT and GCV4, then later implement in ROTOR a number of optimizations such as our synchronization techniques, prefetching, and DPGO. This paper described 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 needed changes 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. This is 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 [8] benchmark. 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. Similarly, we will add support to GCV4 for managed pointers. We are optimistic about being able to complete this work and look forward to exploring other opportunities for improving ROTOR's performance.

7. 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.