Improving Compiler-Runtime Separation with XIR

Ben L. Titzer
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043
(650) 919-4897
ben.titzer@gmail.com

Thomas Würthinger
Johannes Kepler University Linz
Altenbergerstr. 69
4040 Linz, Austria
+43 732-2468-7137
wuerthinger@ssw.jku.at

Doug Simon
Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025
(650) 568-4871
doug.simon@sun.com

Marcelo Cintra
Secure Systems and Software Laboratory
University of California Irvine
448B ICS Building
Irvine, CA 92717
(949) 824-9104
mcintra@uci.edu

ABSTRACT

Intense research on virtual machines has highlighted the need for flexible software architectures that allow quick evaluation of new design and implementation techniques. The interface between the compiler and runtime system is a principal factor in the flexibility of both components and is critical to enabling rapid pursuit of new optimizations and features. Although many virtual machines have demonstrated modularity for many components, significant dependencies often remain between the compiler and the runtime system components such as the object model and memory management system. This paper addresses this challenge with a carefully designed strict compiler-runtime interface and the XIR language. Instead of the compiler backend lowering object operations to machine operations using hard-wired runtime-specific logic, XIR allows the runtime system to implement this logic, simultaneously simplifying and separating the backend from runtime-system details. In this paper we describe the design and implementation of this compiler-runtime interface and the XIR language in the C1X dynamic compiler, a port of the HotSpot^{TM} Client compiler. Our results show a significant reduction in backend complexity with XIR and an overall reduction in the compiler-runtime interface complexity while still generating comparable quality code with only minor impact on compilation time.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—Code Generation; Compilers; Optimization; Runtime Environments

General Terms

Design, Experimentation, Languages, Performance

Keywords

Compilers, JIT, Java, virtual machines, lowering, software architecture, object model, virtual machine interface, intermediate representations, register allocation, runtime system

1. INTRODUCTION

The amount and importance of software running on managed runtime virtual machines continues to grow. New language features and the demand for ever-greater performance continue to drive research and development of virtual machines. Flexible software architecture is needed to support rapid exploration of new design approaches and implementation techniques. One of the key factors in achieving the necessary flexibility in virtual machines is the interface between the dynamic compiler and the rest of the runtime system, including the garbage collector, object model, synchronization mechanisms, etc. Achieving this separation without sacrificing high performance still remains elusive. Even today, industrial strength virtual machines usually require compiler changes when significant changes are made to the runtime system.

Dependencies between the compiler and runtime arise from a number of sources.

i. *Platform configuration*. The virtual machine must configure the compiler with information about the target architecture such as the instruction set, word width, reference size (which may differ from the word width when using compressed references), cache alignment, stack alignment, supported ISA extensions, allocatable registers, calling convention, etc.

ii. *Runtime data structure access*. The compiler must access a number of the runtime system's internal data structures, including some representation of methods and their code, classes, constant pools, etc. The compiler may also query the runtime about the resolution status of types, fields, and methods referenced from the bytecode.

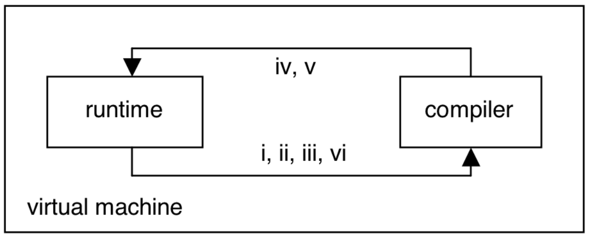
iii. *Optimization selection and tuning*. The runtime may want to selectively enable or influence different optimizations on a per-method or per-compilation basis. In particular the runtime system may wish to influence inlining decisions by making use of dynamic profiling information [1][4][15][19].

iv. *Speculative dependencies*. The compiler may optimistically assume a non-final class to be a leaf class in order to perform devirtualization and inlining, which requires communicating a dependency back to the runtime system for later deoptimization of the compiled code [18][19].

v. *Installation of compiled code*. The compiler produces machine code and metadata such as reference maps and deoptimization information that the runtime system must install into its code region(s) and internal data structures for execution.

vi. *Implementation of object model*. Lowering an object operation such as a field access, virtual invocation, synchronization operation, or memory allocation to machine-level operations such as pointer arithmetic and memory accesses is heavily runtime-dependent and comprises a large part of the compiler backend logic.

Although many virtual machines already have adequate interfaces for platform configuration, access to runtime data structures [18], and the registration of speculative dependencies [19], we believe that none of them adequately address every category. In particular, the dependencies in the compiler on the implementation of the object model present the most difficult category. This paper addresses all of these categories (except iii, control of optimizations). Our bi-directional compiler-runtime interface abstracts compiler and runtime data structures and our XIR language allows the runtime system to provide the logic for lowering all object operations. This leads to the following design (edges indicate data flow annotated with each category of dependency):



In this paper we present our techniques to separate the C1X dynamic optimizing compiler from the runtime system of the VM in which it is installed. We present the highlights of our compiler-runtime interface that separates these components from each other's internal implementation details, allowing the compiler to query runtime data structures during compilation and produce a runtime-independent representation of compiled code. The major contribution of this paper is the design and implementation of a domain-specific language called XIR which allows the runtime system to specify the implementation of object operations, completely separating the compiler from any details of the object model, garbage collector, etc.

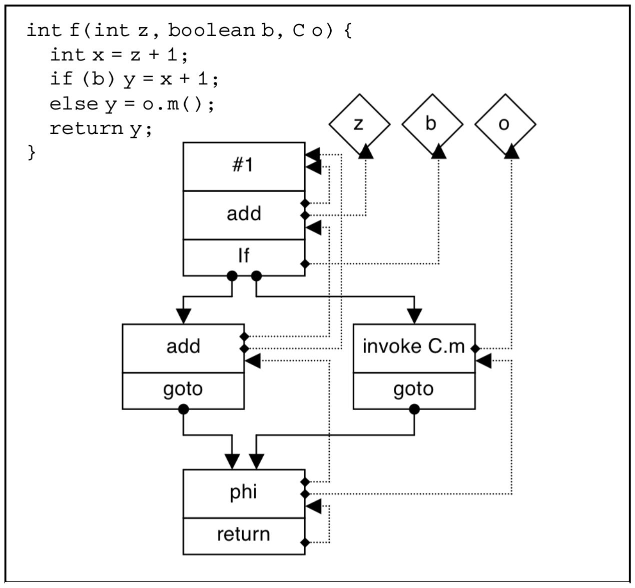
This paper is organized as follows: Section 2 provides an overview of the C1X compiler design, which remains very close to the HotSpot Client Compiler for Java from which it was ported, providing a background for discussion of XIR's contribution. Section 3 describes the separation between the compiler and runtime through interfaces representing runtime data structures and compiler data structures. Section 4 describes XIR and how it further separates the compiler from VM implementation details. Section 5 discusses results and gives metrics, both in terms of software complexity, compile time, and runtime performance on a suite of standard benchmarks. Section 6 discusses related work, and Section 7 provides a conclusion and acknowledgments.

2. COMPILER DESIGN

This section describes the origin and design of the C1X compiler to provide context for the compiler-runtime interface discussion and the presentation of XIR.

2.1 C1X Genesis

Achieving industrial-strength performance for any Java VM requires a good optimizing dynamic compiler that balances compilation time versus code quality. When building a new optimizing dynamic compiler for the Maxine VM, we began examining the compilers in the HotSpot VM. The HotSpot VM in fact includes two compilers, the client [18] and the server [19], also known as C1 and C2, respectively. Both are written in C++ and produce optimized machine code from bytecodes in response to profiling performed by the interpreter. C1 is typically the default compiler in client settings due to its fast compile speed and smaller memory footprint, while C2 is used in server settings because it performs much more thorough optimization and thus achieves maximum peak performance. C1's straightforward design makes it attractive for porting to the Maxine VM. In particular it achieves high compilation speed by focusing on "big-win" optimizations and ignores complex optimizations such as code motion. C1 performs deep inlining, eager local optimizations, some global optimizations, speculative leaf class and leaf method assumption with support for deoptimization, and a fast, well-tuned linear scan register allocator. We started with the approximately 58000 lines of C++ code in C1 from OpenJDK 7 and began rewriting it in Java to create C1X.



```
int f(int z, boolean b, C o) {
  int x = z + 1;
  if (b) y = x + 1;
  else y = o.m();
  return y;
}
```

Figure 1. Example code and its resulting HIR graph. Solid lines represent control flow edges while dotted lines represent data flow edges. Diamonds represent input parameters to the method.

2.2 FRONT END

C1X parses Java bytecodes into its main high-level IR (HIR), which consists of both a control flow graph and a value dependence graph. See Figure 1 for an example and [18] for more detail. Unlike a value dependency graph or sea of nodes representation, C1X's HIR represents control flow with basic blocks, each with an ordered list of HIR instructions. Each HIR instruction directly references the instructions that produce its input values. Phi nodes are attached to the beginning of basic blocks that are join points and merge data flow when necessary. The SSA nature of this representation is carefully constructed at bytecode parsing time by processing basic blocks in CFG reverse post-order and using a conservative estimate for phi creation in loops. The direct references from uses to definitions that are characteristic of value dependence graphs allows C1X to perform numerous optimizations while parsing, including strength reduction, constant propagation, constant folding, local value numbering, load elimination, and dead code elimination. All inlining occurs while parsing bytecodes, and values passed as arguments to inlined methods are propagated forward, allowing all of these optimizations to be applied to the inlined method when it is parsed into the HIR graph. Speculative optimizations based on class hierarchy analysis [10] allow many call sites to be devirtualized and inlined because deoptimization can later invalidate compiled code. Subsequent compiler passes remove unnecessary phis, merge blocks, remove null checks, replace some control flow with conditional moves, perform global value numbering, and remove dead code. There are minor differences in how these optimizations are implemented in C1X versus C1, but for a more detailed description, see [18].

2.3 BACK END

After optimization, C1X translates HIR into a lower level IR (LIR) before performing register allocation and then code generation. LIR consists of an ordered list of basic blocks, each with an ordered list of LIR instructions. LIR is a similar to a quad representation where each instruction has an opcode, an output operand, and multiple input operands. Operands to instructions can be physical registers, virtual registers, or constants. LIR instructions are typically machine-level operations such as pointer loads and stores, arithmetic, and moves, but some complex operations may generate multiple machine instructions and may use internal temporaries. Unlike quads, however, each LIR instruction may have an unbounded number of input operands, an unbounded number of temporaries, but at most one output operand. LIR is not in SSA form and requires phi nodes to be eliminated by inserting move instructions.

The LIR representation is machine-independent, and most object operations in HIR can be translated (lowered) to loads, stores, compares and branches in a machine-independent but runtime-dependent way. However, some operations that have architectural constraints are lowered in a machine-dependent manner (e.g. shift and divide operations on x86) and may have physical registers pre-assigned. This translation logic is written manually in C1 because it only generates code for HotSpot. However, the implementation is separated into machine-independent and machine-dependent parts. We originally ported this logic when creating C1X and modified it to generate code for the Maxine VM.

This hard-wired translation process illustrates the major problem this paper addresses: the lowering phase is the classic place in any compiler where runtime dependence and machine dependence converge. Though C1X is separated from the runtime data structures that need to be queried during compilation (as discussed in the next section), it must however generate code for object operations that use object metadata at runtime. Modifying the runtime system, e.g. to change the implementation of invokeinterface, requires modifying the compiler backend. In our port of C1 we were forced to make numerous changes to the C1X lowering phase due to the implementation differences of object operations. Although Java's primitive operations and control flow constructs can be compiled independently of the object model, we found that almost no object operations were identical (with some exceptions such as reading a resolved instance field). Further, a large amount of complexity arises from handling uncommon cases such as accesses to unresolved fields or methods and the slow paths of operations such as monitorenter. Such operations are so radically different between Maxine and HotSpot that our first solution was to implement only the slow path on Maxine by emitting a call to the runtime system (as done in [14]). Eventually a growing list of Maxine-specific backend changes led to the need for a more elegant solution: XIR, which we discuss in Section 4.

3. COMPILER RUNTIME INTERFACE

The complex interaction between the compiler and runtime system during a compilation requires a bi-directional interface, where each component provides an interface to the other. Our design explicitly separates interface elements that must be provided by the runtime from those that must be provided by the compiler and uses a naming convention where the prefix `Ci` denotes a compiler-provided interface object and the prefix `Ri` denotes a runtime-provided interface object.

First, the runtime is responsible for creating and configuring the compiler, including selecting the target architecture and configuring runtime-specific settings such as the allocatable registers and stack frame alignment. For this purpose, the compiler provides a number of concrete `Ci` classes that can be constructed by the runtime and passed to the compiler for its internal configuration. We considered a more sophisticated configuration interface that avoided exposing concrete compiler classes but in the end its complexity was not warranted.

Secondly, when the runtime system requests a compilation, it must provide a representation of the method to be compiled and related runtime data structures that the compiler will use. For this purpose the runtime must provide implementations of the `RiType`, `RiField`, `RiMethod` and `RiConstantPool` interfaces, each of which support a number of query operations used by the compiler during compilation. Java interfaces offer maximum flexibility to the runtime system; it can either expose its already-defined data structures by implementing the interfaces, or wrap them with adapters.

Thirdly, during a compilation the compiler may request additional information from the runtime system such as the calling convention for a particular call or advice on inlining decisions. For this purpose the runtime system must provide an `RiRuntime` interface to answer a number of other queries by the compiler.

Lastly, and most importantly, the compiler will produce compiled code with attached metadata. For this purpose it provides a concrete data structure, the `CiTargetMethod`, which the runtime system can reprocess into its own internal data structures for installation and execution. Similar to most other `Ci` classes, we chose to have the compiler interface provide concrete classes for this purpose, since in each case no compiler implementation details are revealed and it is not necessary to control their construction.

Selected Runtime interface classes:

- `RiConstantPool`: A constant pool associated with bytecode that the compiler uses to lookup and resolve fields, types, and methods.
- `RiExceptionHandler`: An exception handler entry, including the range of bytecode indices covered, the handler index, and the type of exception caught.
- `RiField`: A resolved or unresolved field referred to in the bytecode, including a name, a type, and the enclosing class. Resolved fields also allow queries of attribute flags such as final, private, volatile, etc.
- `RiMethod`: A resolved or unresolved method referred to in the bytecode, including a name, a signature, and the enclosing class. Resolved methods may refer to a method selector or a concrete method implementation with code and allow queries of attribute flags such as final, private, static, synchronized, etc.
- `RiRuntime`: An interface of assorted runtime services needed by the compiler, including converting a Java class object into an `RiType`, looking up a system `RiType`, getting the calling convention for a particular method signature, advice on required or disallowed inlining for certain methods, and various other queries.
- `RiSignature`: A method signature, including parameter types and return types.
- `RiType`: A resolved or unresolved Java class, interface, or array type referred to in the bytecode or from another runtime interface object. Resolved types may refer to any valid Java type and respond to queries for attribute flags, the super type, whether the type is an array or interface, etc. Resolved, instantiable types can also lookup a method implementation for an `RiMethod` selector.
- `RiXirGenerator`: The runtime class that generates XIR. It is called by the compiler during lowering of each object operation to machine-level operations (Section 4).

Selected Compiler interface classes:

- `CiArchitecture`: An object representing the machine architecture including the instruction set. Chosen by the runtime when creating and configuring a compiler.
- `CiBailout`: Exception that represents an aborted compile, either due to compiler limitations on input bytecode or unexpected internal compiler errors.
- `CiCodePos`: A code location and inlined call chain. Created by the compiler and used throughout metadata to refer to source locations in the compiled code.
- `CiCompiler`: An object capable of producing `CiTargetMethod` instances from `RiMethod` instances. Created, configured, and used by the runtime to compile methods.
- `CiConstant`: A representation of either primitive or object constants. Created and used by both the runtime and the compiler to represent and exchange constant program values.
- `CiDebugInfo`: Debug information for precise stack traces and deoptimization. Generated by the compiler and included as part of the compilation result.
- `CiKind`: An Enumeration of Java's primitive types, the object type, and a special word type. Used by both the compiler and runtime to refer to JVM-level types.
- `CiLocation`: A union of either a physical register or a stack location. Created and used by both the compiler and runtime to describe locations of parameters and temporaries within a method.
- `CiRegister`: A physical register. Created by the compiler and used by both the compiler and runtime to refer to specific machine registers.
- `CiResult`: The result of compilation, consisting of either a bailout, or a target method plus statistics. Created by the compiler as the result of compilation.
- `CiStatistics`: General statistics about the compilation, including compiled bytes, number of inlines, etc. Created by the compiler as part of the compilation result.
- `CiTarget`: A collection of settings such as the size of references, cache geometry, allocatable registers, etc. Created by the VM and used to configure a compiler.
- `CiTargetMethod`: A compiled method, including machine code and metadata such as relocation, patching, and debug information. Created by the compiler as part of the compilation result.
- `CiXirAssembler`: An assembler-oriented interface provided by the compiler to the runtime to build XIR code. Explained in more detail in Section 4.

In total, the `Ri` interface classes comprise about 1300 lines of Java source code, including documentation, while the `Ci` classes are about 2300 lines. The implementation of the requisite `Ri` interfaces in the Maxine VM totals about 3500 lines of Java source, which includes implementations of `RiType`, `RiField`, `RiConstantPool` and `RiMethod` that wrap existing data structures in the runtime system rather than modifying them to fit the `Ri` interfaces.

4. XIR

Although data structures in the compiler-runtime interface separate these components from each other's implementation details, the compiler must nevertheless generate efficient machine code to implement object operations consistent with the runtime system's design. As discussed previously in Section 2, this lowering phase from object operations to machine operations is highly runtime-dependent. Our experience porting C1X from HotSpot led us to a design where the compiler makes *no assumptions* about how the runtime system implements object operations and *all lowering logic is in the runtime system*.

C1X provides an interface to the runtime system to generate code in a small, domain-specific language called XIR that is designed explicitly for the purpose of reducing object operations. XIR is very similar to an assembly language for a RISC instruction set. Unlike RISC instruction sets however, XIR has neither a binary format nor a textual format and therefore it is more appropriately considered an intermediate representation. It is a low-level, three-address intermediate representation that has an unbounded number of virtual registers, a set of machine-level but machine-independent instructions such as 32 and 64-bit integer arithmetic, pointer load and store, and conditional branches, but no computed jumps. Local branches and jumps may divide the XIR code into basic blocks. XIR also has support for defining a *fast path* and a *slow path* (see Figure 2), which is useful for implementing

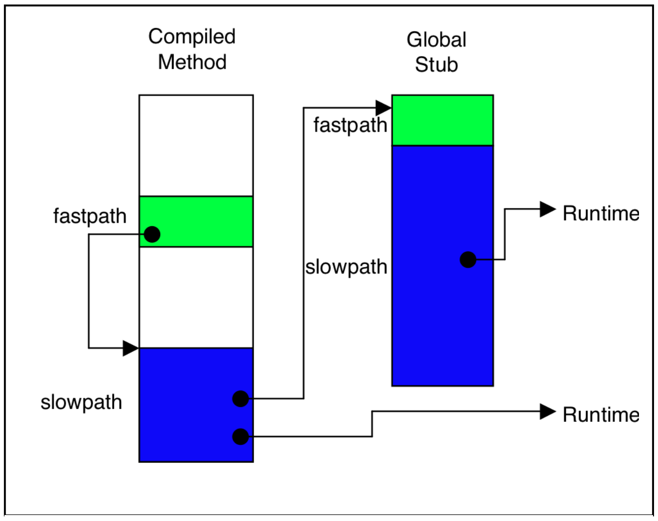


Figure 2. Different fast and slow code paths can be defined with XIR. The fastpath is generated inline and may branch to the slow path that is generated at the end of the method. Global stubs can contain common code and be called from compiled methods, and both can call the runtime system.

```
public class CiXirAssembler {
  XV     createInputParameter(CK type)
  XV     createConstantInputParameter(CK type)
  XV     createTemporary(CK type)
  XV     createFixedTemporary(CK type, CL location)
  XV     createConstant(CC constant)
  XL     createLabel(boolean fastpath)
  void   add(XV dest, XV a, XV b)
  void   sub(XV dest, XV a, XV b)
  . . .
  void   mov(XV dest, XV a)
  void   pload(XV dest, XV pointer)
  void   pstore(XV pointer, XV value)
  void   bind(XL label)
  void   jump(XL label)
  void   jeq(XL label)
  . . .
  void   callJava(XV dest)
  void   callStub(XT template, XV dest, XV[] args)
  void   callRuntime(Object rtcall, XV dest, XV[] args)
  XT     finishStub()
  XT     finishTemplate()
}
```

Figure 3. `CiXirAssembler` interface methods for creating input parameters, temporaries, constants, labels, adding instructions, branches, calls, and finishing the template. For brevity, `XV = XirVariable, XT = XirTemplate, XL = XirLabel, CK = CiKind, CL = CiLocation`.

bytecodes that require safety checks that fail infrequently and therefore must include rarely-executed failure-handling code. The compiler will always emit the fast path inline while the slow path will be generated at the end of the method for better instruction cache utilization. Further, the runtime can also define a *stub*: a global piece of XIR that can be called from the instruction-specific XIR (also in Figure 2). Stubs are useful for complex shared logic that is too large to be profitably inlined. XIR has two call instruction variants: CALL_STUB, for calling previously defined stubs, and CALL_RUNTIME, for calling any method in the runtime system that it chooses to expose to XIR.

4.1 Two Phases

In addition to separating the compiler from runtime implementation details, XIR also separates the runtime system from the specifics of the compiler's intermediate representation(s). XIR is like an assembly language that is specifically designed for the runtime implementer. One only needs to understand this language in order to describe object operations, and it is not necessary to maintain SSA form or explicitly specify data flow edges. This design choice gives rise to a mismatch between XIR and the actual intermediate representation(s) of the compiler, in particular the HIR and LIR representations of C1X.

To reduce this problem, we separated the creation of XIR by the runtime system and its use by the compiler into two distinct phases. During the first phase, when the runtime system instantiates and configures the compiler, it also creates a collection of pre-built XIR *templates* for later use. An XIR template is simply a finished piece of XIR code that has unbound input parameters known as `XirParameters`. A template can be as small as a single XIR instruction or as large as an arbitrary control flow graph of XIR instructions. In the second phase, during compilation, the compiler requests XIR from the runtime system; the runtime system responds by passing back an XIR template and its inputs, together known as an XIR *snippet*.

This phase separation has two important benefits. First, it allows the compiler to pre-process the XIR that will be used by the runtime system, before any compilations occur. During this preprocessing it may translate the XIR to an internal SSA or dataflow representation for use later (e.g. to integrate it into an HIR graph), or it may gather register allocation constraints (e.g. to generate LIR that requires certain registers on a particular architecture). Secondly, allowing the runtime system to reuse a pre-built template when lowering each object operation improves compile speed.

4.2 `CiXirAssembler` interface

The runtime system creates XIR templates in the first phase, but XIR has no textual format or binary format; it exists only inside the compiler as a graph of data structures representing XIR instructions, variables, etc. Instead, the runtime system constructs XIR using an *assembler object* provided by the compiler. Figure 3 shows a list of methods available in the `CiXirAssembler` interface. The assembler object has methods for creating XIR variables and labels as well as adding XIR instructions one by one to an internal ordered list of instructions.

When written sequentially in the runtime implementation, calls on the assembler object look much like an embedded domain-specific language with Java syntax. Figure 4 shows an example with two different implementations of the putfield operation: one with standard uncompressed references and one with compressed references. This assembler object interface also obviates the need for an XIR syntax and XIR parser, reducing the implementation burden on the compiler, unlike LIL [14], for example, which required a LIL parser built into the compiler. The implementation of `CirXirAssembler`, its internal data structures, and the interface elements `XirTemplate`, `XirSnippet` and `XirArgument` comprise a total of about 900 lines of Java source code.

```
XirTemplate buildPutFieldTemplate(CiKind kind, boolean genWriteBarrier) {
  asm.start(CiKind.Void); // putfield does not produce a value
  XirParameter object = asm.createInputParameter("object", CiKind.Object);                  // object input
  XirParameter value = asm.createInputParameter("value", kind);                              // value input
  XirParameter fieldOffset = asm.createConstantInputParameter("fieldOffset", CiKind.Int); // field offset
  asm.pstore(kind, object, fieldOffset, value, true);                                        // store field
  if (genWriteBarrier) addWriteBarrier(asm, object, value);                                  // add write barrier
  return asm.finishTemplate("putfield<" + kind + ", " + genWriteBarrier + ">");
}
XirTemplate buildCompressedPutFieldTemplate(CiKind kind, boolean genWriteBarrier) {
  asm.start(CiKind.Void); // putfield does not produce a value
  XirParameter object = asm.createInputParameter("object", CiKind.Object);                  // object input
  XirParameter value = asm.createInputParameter("value", kind);                              // value input
  XirParameter fieldOffset = asm.createConstantInputParameter("fieldOffset", CiKind.Int); // field offset
  XirVariable addr = asm.createTemporary(CiKind.Unsafe);                                     // temp for address
  XirVariable r13 = asm.createFixedTemporary(CiKind.Unsafe, AMD64Register.R13);              // R13 contains heap base
  asm.add(addr, r13, object);                                                                // add compressed oop to base
  asm.shl(addr, addr, asm.i(3));                                                             // shift left
  asm.pstore(kind, addr, fieldOffset, value, true);                                          // store field
  if (genWriteBarrier) addWriteBarrier(asm, object, value);                                  // add write barrier
  return asm.finishTemplate("putfield<" + kind + ", " + genWriteBarrier + ">");
}
```

Figure 4 shows an example usage of the `CiXirAssembler`. It builds two versions of the putfield operation: one for normal object references and one for compressed object references. Both templates take the object and the field value as inputs and have a constant input which will be the field offset. The compressed version demonstrates the usage of fixed registers; it assumes the base of the heap is always stored the AMD64 register R13. A helper method `genWriteBarrier()` adds XIR code to the template that implements the write barrier (not shown).

4.3 `RiXirGenerator` interface

The second phase happens when the compiler is lowering object operations to machine operations. Note that XIR was not designed to be an extensibility mechanism for all of Java's operations. We divided Java operations into runtime-independent operations, such as primitive arithmetic and control flow, which the compiler must handle fully, and runtime-dependent operations where the runtime system must supply XIR. This reduces the burden on the runtime system and reduces the set of XIR extension points to those listed in Figure 5.

```
public interface RiXirGenerator {
  XS genSafepoint()
  XS genResolveClassObject(RiType type)
  XS genIntrinsic(XA[] args, RiMethod method)
  XS genGetField(XA object, RiField field)
  XS genPutField(XA object, XA value, RiField field)
  XS genGetStatic(RiField field)
  XS genPutStatic(XA value, RiField field)
  XS genMonitorEnter(XA object)
  XS genMonitorExit(XA object)
  XS genNewInstance(RiType type)
  XS genNewArray(XA length, CiKind elementKind,
      RiType arrayType)
  XS genNewMultiArray(XA[] dims, RiType arrayType)
  XS genCheckCast(XA object, RiType type)
  XS genInstanceOf(XA object, RiType type)
  XS genInvokeInterface(XA receiver, RiMethod method)
  XS genInvokeVirtual(XA receiver, RiMethod method)
  XS genInvokeSpecial(XA receiver, RiMethod method)
  XS genInvokeStatic(XA receiver, RiMethod method)
  XS genArrayLoad(XA array, XA index,
      CiKind elementKind, RiType arrayType)
  XS genArrayStore(XA array, XA index, XA value,
      CiKind elementKind, RiType arrayType)
  XS genArrayLength(XA array)
}
```

Figure 5. `RiXirGenerator` interface methods. Each method accepts a number of `XirArgument` objects that represent opaque compiler IR variables or nodes, and the operands, such as the field being accessed in a getfield or the type of a checkcast. The runtime must supply an `RiXirGenerator` implementation to the C1X compiler backend. Each method returns an `XirSnippet` which contains both an `XirTemplate` and bindings for the `XirTemplate`'s inputs. For brevity, `XA = XirArgument, XS = XirSnippet`.

The compiler requires the runtime system to supply an implementation of the `RiXirGenerator` interface that has methods to generate XIR for each HIR operation. Each method on the `RiXirGenerator` interface corresponds to a Java language operation and takes two types of parameters: arguments and operands. Arguments are opaque `XirArgument` instances passed by the compiler that represent compiler variables or nodes, such as the receiver object in a field access or the value written in an array store. Operands represent the "fixed" part of an operation such as the field in a getfield operation or the type in a checkcast operation. From a bytecode perspective, arguments represent values that would be on the Java stack, and operands represent quantities in the instruction stream such as a field reference. Notice again in Figure 5 that most operands are `Ri` classes, which naturally allows the runtime to use its own data structures to decide which XIR to return to the compiler for each instruction.

Each `gen()` method in the `RiXirGenerator` interface returns an `XirSnippet`, which is simply an `XirTemplate` with each of its parameters bound to either an input `XirArgument` which was passed to the `gen()` method or an `XirArgument` representing a constant (e.g. a constant field offset). Recall that the `XirTemplate` was constructed in the previous phase, when the runtime system configures the compiler. The two-phase approach saves compilation time by reusing work from the configuration phase. Figure 6 shows an example from the Maxine `RiXirGenerator` that implements the generation of an `XirSnippet` for the putfield operation.

The implementation of the `RiXirGenerator` for the Maxine VM is 1350 lines of Java source code, including comments, blank lines, the code to build XIR templates, the data structures to store and lookup templates for different situations, the implementation of the interface methods, and the implementation of runtime calls that are called from XIR templates and stubs. It contains approximately 240 calls to the `CiXirAssembler` interface.

```
public class MaxXirGenerator extends RiXirGenerator {
  . . .
  @Override
  public XirSnippet genPutField(XirArgument receiver, RiField field, XirArgument value) {
    XirPair pair = putFieldTemplates[field.kind().ordinal()];
    if (field.isResolved()) {
      XirArgument offset = XirArgument.forInt(field.offset());
      return new XirSnippet(pair.resolved, receiver, value, offset);
    }
    XirArgument guard = XirArgument.forObject(guardFor(field));
    return new XirSnippet(pair.unresolved, receiver, value, guard);
  }
}
```

Figure 6 shows an example implementation of the `RiXirGenerator` for the putfield operation. The `genPutField()` method is passed the receiver object and the value `XirArgument` as handles and the `RiField`. This method simply looks up the correct template using the type of the field, checks whether the field is resolved, and returns either the resolved or unresolved snippet. (The `guardFor()` method creates a resolution object needed in the unresolved template, which is not shown).

4.4 Compiling XIR

We wanted to preserve as much design freedom for the compiler implementation as possible. This is done not by what is in the interface, but what is not. For example, the `RiXirGenerator` does not allow the runtime system to make any assumptions about when lowering occurs during compilation. Secondly, the `XirArgument` handles passed by the compiler hide all implementation details of the IR of the compiler. Further, the runtime cannot assume that the operations are lowered in any particular order (either according to the order in which they appear in the method or inlined methods being compiled, or any other order) and must consider each operation in isolation. It also cannot assume that all operations are lowered at the same time; the compiler might lower some operations, perform optimizations, issue other queries to the runtime interface, lower more operations, perform more optimizations, etc. In a sense, the compiler expects the `RiXirGenerator` to be *stateless*.

This allows the compiler implementation maximum freedom in ordering lowering with its other phases. Although C1 performs all lowering in its translation from HIR to LIR, other compilers might perform lowering by replacing object-operation nodes with machine-level nodes but keeping the same overall IR structure, allowing the same optimizations to be performed before or after lowering. Optimizing after lowering is especially important if the object operations become several machine operations that could be candidates for common sub-expression elimination (CSE) and/or code motion. For example, repeated accesses to an object's meta-object, such as in an invokevirtual, checkcast or instanceof operation, may reuse the load of the meta-object after the load has been generated by the runtime system through XIR. Additionally, write barriers and synchronization may produce arithmetic expressions and accesses to thread-local values that are good candidates for CSE and code motion. However, a different compiler or the same compiler on a lower optimization level may not perform many optimizations after lowering and may (like C1 and C1X) translate from a high-level IR to a low-level IR while lowering. All choices are permitted by our design.

We chose to start implementing XIR in the C1X backend only after C1X passed our regression suite with hand-written lowering logic specific to the Maxine VM. In addition to having a stable platform to work from, this allowed us to assess the implementation effort and compare the XIR results against the traditional hand-written logic.

As mentioned in Section 2, the design of C1X's LIR allows each LIR instruction to have an arbitrary number of inputs, an arbitrary number of temporaries, but at most one output. The linear scan register allocator [23] already handles such instructions, so it was straightforward to simply add a special XIR instruction that represents a complete XIR snippet including its inputs, temporaries, and output. We then modified the translation from HIR to LIR to either request XIR from the `RiXirGenerator` or perform the previous logic depending on an option; that means that both mechanisms are fully functional in the same compiler. If the XIR option is enabled, the backend will generate a single LIR instruction representing the `XirSnippet` returned from the runtime system's `RiXirGenerator`.

The two-phase approach to XIR production proved useful to our implementation. In the first phase, when the runtime system constructs `XirTemplates`, C1X preprocesses each template to gather any architectural register constraints (e.g. if it performs a division, which requires certain registers on x86) and determine whether the template has a slow path or any calls. The information computed during preprocessing is attached to the `XirTemplate` for later use. It is then transferred to the LIR instruction during lowering and used by the register allocator to allocate registers to inputs, temporaries, and output of the LIR instruction. Note that because the entire `XirSnippet` is represented as a single LIR instruction, even if it contains internal control flow, the register allocator assumes that all of a snippet's temporaries and inputs may be simultaneously live within the `XirSnippet`. Alternatively, one could generate LIR instructions from the `XIRSnippet` as soon as it is returned from the runtime system, before performing register allocation. This would allow the register allocator to compute liveness of the temporaries of a snippet just as it would for other variables and likely make better overall decisions, but at the expense of dealing with more variables and more instructions.

After register allocation is performed, C1X performs some simple optimizations such as removing useless moves and jumps and may reorder the basic blocks for short loops. C1X then visits the basic blocks of LIR instructions and generates machine code LIR instruction by LIR instruction. To support XIR, we simply added support for compiling the special LIR instructions that represent `XirSnippets`. This is done by opening up the `XirTemplate` and visiting the XIR instructions from the fast path, generating machine code for them one by one. If the `XirTemplate` has a slow path, then the slow path code will be added at the end of the method.

The implementation of XIR in the backend of C1X required a total of 600 additional lines of code.

Benchmark	methods	bytecode	code	w/XIR	time	w/XIR	HIR	LIR	w/XIR	XIR %
JDK 1.6 -O1	47252	2264265	11413306	17.8%	7.633	6.3%	962807	3171828	-4.6%	8.4%
JDK 1.6 -O3	47252	2264265	12858443	18.2%	10.247	5.0%	597182	3584689	-6.4%	10.8%
Maxine -O1	640	9050	43150	-0.1%	6.939	7.2%	853290	2804920	-0.1%	5.2%
Maxine -O3	640	9050	138844	23.9%	12.687	3.9%	818524	4131852	-1.2%	7.1%
C1X -O1	3694	255160	1069099	11.8%	0.670	5.5%	86507	293495	-4.9%	7.9%
C1X -O3	3694	255160	1363815	10.5%	1.061	2.6%	71174	369615	-9.4%	13.7%
SpecJVM98 -O1	3197	285489	1585293	4.5%	0.959	-5.7%	104216	378096	-17.0%	11.1%
SpecJVM98 -O3	3197	285489	1749927	4.6%	1.237	-5.6%	73494	423460	-18.4%	14.9%
SciMark2 -O1	157	13094	65672	3.1%	0.054	-1.2%	4996	16671	-12.1%	10.4%
SciMark2 -O3	157	13094	72717	3.3%	0.065	-1.9%	3337	18874	-14.8%	12.3%

Figure 7a gives static compilation metrics for each benchmark at two optimization levels. Level -O1 includes local optimizations and simple control flow optimizations while -O3 enables all optimizations. The columns indicate: number of methods, bytecode size, machine code size without XIR, percentage change in machine code size with XIR, compilation time in seconds without XIR, percentage change in compilation time with XIR.

Figure 7b (rows continue from 7a) includes compiler IR statistics, including the number of HIR instructions, number of LIR instructions, percentage change in LIR with XIR, and proportion of XIR instructions.

5. RESULTS

In this section we report experimental results that compare the XIR implementation in C1X to C1X without XIR. We report several static compilation metrics over several code bases, including the number of compiled methods, bytecode size and number of HIR instructions. We then compare static compilation metrics of the XIR implementation with the non-XIR implementation, including compile speed, number of LIR instructions, number of XIR-implemented instructions, and compiled code size. We then compare the code quality of the XIR implementation with the non-XIR implementation by measuring the execution time of a number of benchmarks with C1X installed into the Maxine VM as a dynamic compiler.

5.1 Static Measurements

Figure 7 gives our experimental results in gathering a number of C1X compilation metrics with and without XIR. These experiments were run on a quad-core Intel Nehalem CPU @ 2.66ghz with 8GB RAM and 64-bit OpenSolaris. We chose to run C1X as a user application on an industrial-strength VM because it allowed faster development time and allowed us to obtain more extensive measurements. For these experiments we ran C1X as a user application on the HotSpot Server VM 1.6.0_13 with a 2GB heap. To support this static compilation scenario, the Maxine class loader, class file parser, verifier and internal runtime data structures are running in user mode as well. The timing measurements were collected after allowing the host VM to "warm up" running C1X and represent an average over 10 iterations following 5 warm up iterations.

Figures 7a and 7b illustrate the effects on compilation on a number of method suites. First, notice a sizeable increase in machine code size for several benchmarks. This is due to more of the fast path operations being implemented in XIR and inlined than in our hand-written logic, which leaves many cases to slower but much smaller runtime calls. Secondly, notice that compilation time is increased by 2-7% for three benchmarks; this is because the backend must do more work to consult the runtime for lowering each operation rather than simply execute handwritten logic. However, two benchmarks actually show a reduction in compilation time; this is because for these benchmarks, XIR results in far fewer LIR instructions, which requires less work for the register allocator and subsequent optimizations on LIR.

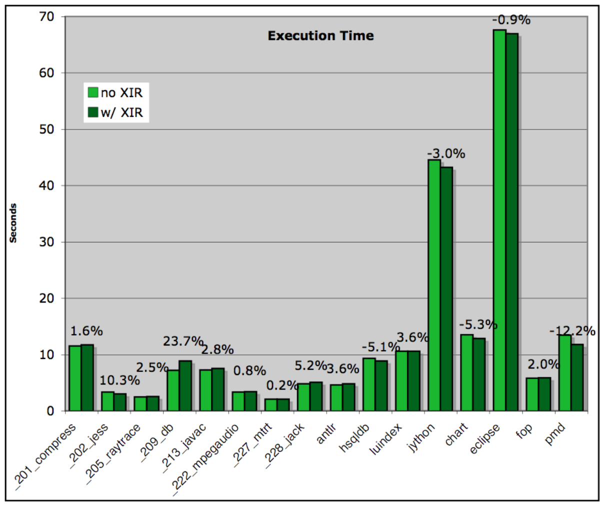


Figure 8 gives execution time results for the XIR and non-XIR implementation with C1X installed as the optimizing compiler in the Maxine VM on the SpecJVM98 benchmark suite and DaCapo benchmarks. Numbers above bars show relative change with XIR.

5.2 Dynamic Measurements

Figure 8 gives our execution time comparison with and without XIR. For this experiment, C1X is used to compile itself into the Maxine VM's boot image and is configured as the optimizing compiler. C1X is then triggered at runtime for hot methods by method invocation counters inserted by Maxine's non-optimizing compiler. We chose to run the SpecJVM98 benchmark suite and the DaCapo [7] suite. Unfortunately due to recent Maxine regressions, we cannot report results for the bloat, lusearch and xalan benchmarks.

We performed 5 runs of each benchmark, where each run was a new VM instance. For each run, the time recorded was the time from the start of the VM process until the end of the VM process; any internal timing numbers reported by the benchmarks were ignored. We ran all benchmarks in their default configurations.

Figure 8 shows that most benchmarks are affected by less than 5%, with three outliers: jess and db which are slowed down by 10% and 24% respectively, and pmd, which is sped up by 12%. Most of the DaCapo benchmarks run faster with XIR yet all of SpecJVM98 runs faster without it. Speedups from XIR are mostly because our Maxine XIR implementation provides special implementations for leaf class type tests, interface dispatch, and other operations that we did not replicate in the hand-written logic. On the other hand, the static benchmarks in Figure 7a show that XIR usually increases compilation time, which of course contributes to runtime in this scenario. Also, we notice that XIR can sometimes result in worse code because register allocation does not happen within XIR templates, but only between them. It is likely that the slowdowns experienced by some benchmarks programs are due to this effect appearing in hot loops and also due to worse instruction cache behavior with the typically larger machine code size of XIR.

5.3 A Simpler Backend

While the primary goal of XIR is to separate the compiler from the logic of lowering operations, moving this complexity to the runtime system has the side effect of simplifying the compiler. To measure the reduction in complexity, we forked the source code of C1X and created an experimental branch where we removed all hand-written lowering logic. (Note that this branch was not used to obtain any numbers in the previous section). We first removed the logic to translate HIR object operations to lower-level LIR operations and the complex LIR instructions that were only necessary to support that logic; this removed 2000 lines of Java source code from C1X, from 53000 to 51000, nearly all in the backend. We then removed 10 or so methods from the interface which were solely used by this logic; for example, the offset of a field, the index of a method in a virtual table, the size of an object header, etc. This reduced the size of the compiler-runtime interface classes from 4700 lines to 4600 lines, with a similar reduction in the Maxine implementation size (from about 5500 lines to 5400 lines). But more importantly than the number of lines of code removed from this interface, concepts such as the size and offset of object headers, the offset of fields from the start of an object, the ID of an interface, and the index of a virtual method no longer appear in the interface, providing more freedom to the runtime system and less hard-wired logic in the compiler. Our code deletion exercise was very preliminary (just a few hours); we expect that more extensive redesign and refactoring of the backend around XIR will reduce the complexity even further.

6. RELATED WORK

High-level language operations must be translated to machine-level operations at some point during compilation or interpretation. In the context of virtual machines, this translation is implemented in any or all of the JIT compiler, dynamic compilers, and the interpreter.

Jikes RVM includes two compilers: a baseline compiler that quickly translates bytecodes to machine code one-by-one, emulating the Java operand stack, and an optimizing compiler. The baseline compiler is basically a single-pass code generator and contains hard-coded semantics in its code generation pass. The Jikes RVM optimizing compiler has three representations: HIR, a high-level representation with Java-level operators and some explicit check operators; LIR, a lower-level but machine-independent representation; and MIR, a machine-specific representation. Most lowering occurs in translation from HIR to LIR where HIR instructions are expanded into LIR operations that are specific to the Jikes RVM runtime system, such as the object layout and calling conventions. Here again the specifics of the Jikes RVM runtime system are hard-coded in the translation. For example, to expand a HIR instruction that represents a call to a virtual method, an additional LIR instruction is generated to load the address of a virtual method via the object's TIB reference and the loaded address is used by a LIR call instruction. Write barriers are injected in the translation from LIR to MIR. An interface between the compilers and the GC exists for barrier injection, but it does not encompass the actual lowering of object operations.

The mostly closely related work is LIL language [9][14] for the Open Runtime Platform (ORP). LIL is a language much like XIR for describing the implementation of object operations and other runtime services. Unlike XIR, LIL is a textual language that is generated as C strings within the runtime system and fed to a parser implemented in the compiler during compilation. Though neither paper reports on the performance implications of this strategy, our assembler object interface avoids the overhead of parsing strings and (we believe) is clearer. Also, the two-phase approach to generating XIR eliminates the need to construct and verify XIR during compilation time. A second difference is that LIL stubs execute with their own activation frame, even though [9] states that they are "inlined" by the compiler. It unclear what the actual inlining mechanism is; in particular it is unclear if they treat temporaries and inputs to LIL stubs equivalently in the register allocator or whether they are required to be in particular registers or stack locations by a calling convention. In contrast, XIR inputs, temporaries, and outputs are treated equally to other variables in the register allocator. XIR is always "inlined" in this sense, and as described in Section 4, it has support for fast paths, slow paths, global stubs, and runtime calls. LIL also has some runtime-specific constructs such as access to thread locals. As shown in the putfield example, it is not necessary for XIR to have any such constructs because C1X allows the runtime to reserve some physical registers that cannot be used by the register allocator but can be used in XIR instructions. In [9] the authors describe support for other language features such as multiple inheritance and mix-ins using LIL. Exploring such ideas for XIR is future work.

The problem of translating high-level operations to machine-level operations is closely related to the problem of implementing a meta-circular virtual machine, i.e. a virtual machine implemented in the same language that it implements. There have been a number of meta-circular virtual machines [1][2][16][20][21], and inevitably the problem of expressing lower-level operations in the higher-level language [13] arises. All of these virtual machines provide low-level primitives as language extensions of one form or another, either as magic types or classes or special library routines. The ability to use low-level primitives in the source provides the opportunity to express the lowering of higher-level operations by writing an interpreter in *source code* with low-level primitives. The compiler or translator can use the interpreter's code as the specification of how to perform lowering for each object operation, e.g. by partial evaluation. This "fully metacircular" approach is taken by Maxine's previous compiler [2], the PyPy VM [20] and the Klein VM [21], which inspired Maxine. Unfortunately Maxine's meta-circular compiler produced poor quality code and had poor compilation time, thus we could not assess the effectiveness of this approach in an industrial strength, optimizing compiler. Our experience with Maxine's previous compiler was the main impetus for building C1X. The PyPy VM also has poor performance. It requires 40 minutes to translate itself to C code and the resulting interpreter-only VM has performance between 3.5 and 11 times slower than the standard CPython implementation, which is also interpreter only.

7. CONCLUSION AND FUTURE WORK

We have presented a compiler-runtime interface that separates the C1X compiler from the runtime system of the virtual machine. This includes `Ri` and `Ci` classes with well-defined roles as well as a new XIR extension mechanism that allows the runtime system to express the machine-level implementation of object operations. We implemented and evaluated XIR in C1X and have shown that XIR has a small impact on compilation time without reducing performance. In fact, we found it so much easier to express the fast path operations in XIR (as opposed to the backend of the compiler) that we implemented more fastpath variants than in the hand-written logic and achieved a significant speedup on nearly all test programs.

Porting C1X to another VM would validate the separation mechanisms discussed in this paper. One obvious choice given the origin is to back-port C1X to HotSpot. As with any port, this would require implementing the runtime interface (`Ri`) classes that expose and adapt the runtime's data structures. A significant amount of JNI would be required to access data structures and functionality in HotSpot's runtime. A previous project in 2001 by Thomas Kotzmann at Johannes Kepler University Linz took HotSpot, removed the compilers, and rewrote a simpler version of C1 in Java. He modified HotSpot to dynamically load the compiler as normal Java code, with a special JNI interface to VM internals to install compiled code. While elegant and functional, this system suffered from poor startup time due to the compiler's code initially being interpreted by the VM, requiring the compiler to also warm up and begin compiling itself to approach peak performance. To achieve competitive startup performance, we believe that C1X would have to be pre-compiled into a form suitable for linking with the HotSpot executable. Unfortunately, many issues beyond the scope of this paper arise when pre-compiling arbitrary Java code, particularly for HotSpot.

Jikes RVM [1] also represents an attractive target for porting C1X, since it is also written in Java and provides all of the necessary runtime infrastructure that is demanded by our runtime interface. We have discussed this possibility with a number of Jikes RVM core developers and identified a number of issues, most of which relate to *magic* [13] types and operations. For example, both Jikes and Maxine provide *unboxed* types, which are Java classes at the source language and bytecode level but are compiled into value types with special knowledge by the Jikes and Maxine compilers, respectively. C1X would have to recognize such magic types and produce appropriate machine operations, reference maps, etc. Another issue arises when compiling memory-model sensitive operations and restricting code motion in certain situations. We believe that the addition of an *unsafe* `CiKind` for XIR may be key to solving this issue. This unsafe type would be used by the runtime for XIR values that must not cross a safepoint (e.g. because the value represents a temporary pointer to the middle of an object) and certain compiler optimizations would be restricted for unsafe types.

The backend support in C1 for instructions with an arbitrary number of inputs and temporaries proved useful in implementing XIR by translation to LIR. However, as mentioned in Section 4, the ability to perform more optimization after lowering could significantly improve code quality. We plan to explore a lowering implementation that uses XIR to translate from object-level HIR operations to machine-level HIR operations and preserve the SSA value-dependence graph nature of HIR. This could be made efficient by preprocessing the `XirTemplates` into a small HIR graphs that can be weaved into the method's HIR at code generation time.

One lacking area of the current compiler-runtime interface is support for specifying and driving optimizations. For example, the runtime system may have information to drive inlining heuristics in the form of the class hierarchy, receiver method and type profiles, and call tree profiling. The Jikes RVM has an interface for the runtime system to make inlining decisions for the optimizing compiler, including monomorphic, n-morphic, and guarded inlining suggestions. Other optimizations that can benefit from profiling information in the runtime system include trace scheduling, block layout, and register allocation. We intend to explore interface designs for these optimizations in future work.

Our source code is freely available under the GPL version 2 license as a sub-project of the Maxine VM [2] and is separated into the compiler-runtime interface module (CRI) and the C1X module (C1X), neither of which have dependencies on any Maxine VM classes. Further, the Maxine VM does not have any source code dependencies on C1X; instead, an adapter module (MaxineC1X) that depends on both Maxine and C1X implements both the runtime interface (`Ri` interfaces) and the functionality required to use C1X as a compiler in the Maxine VM.

7.1 ACKNOWLEDGEMENTS

We would like to thank the original designers of the client compiler and many who have contributed to its maintenance and evolution over the years, including Robert Griesemer, Tom Rodriguez, David Cox, John Rose, Thomas Kotzmann, and Hanspeter Mössenböck. Special thanks to Ken Russell and Christian Wimmer for numerous fruitful discussions during the port of C1 to C1X. Thanks to Michael Van De Vanter, Jens Palsberg, Mario Wolczko, and Peter Kessler for comments on drafts of this paper.

8. REFERENCES

[1] Jikes RVM. http://jikesrvm.org

[2] The Maxine VM. http://kenai.com/projects/maxine

[3] M. Arnold, S. Fink, V. Sarkar, and P.F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*. Boston, MA. January 2000.

[4] M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call-Graph Profiles in Virtual Machines. In *International Symposium on Code Generation and Optimization (CGO '05)*. San Jose CA. March 2005.

[5] D. Ancona, M. Ancona, A Cuni, and N. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Dynamic Languages Symposium (DSL '07)*. Montreal, Canada. October 2007.

[6] D. Bacon, S. Fink, and D. Grove. Space- and Time-efficient Implementation of the Java Object Model. In *ECOOP '02,*

the 16th European Conference on Object-Oriented Programming, University of Malaga, Spain. June 2002.

[7] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*. Portland, OR. October 2006.

[8] C. Bolz and A. Rigo. How not to write Virtual Machines for Dynamic Languages. In *Dynamic Languages and Applications (DYLA '07)*. Berlin, Germany. July 2007.

[9] M. Cierniak, N. Glew, S. Triantafyllis, M. Eng, B. Lewis, and J. Stichnoth. Object-Model Independence with Code Implants. In *Multiparadigm Programming with Object Oriented Languages (MPOOL '03)*. Anaheim, CA. October 2003.

[10] C. Click and M. Paleczny. A Simple Graph-based Intermediate Representation. In *ACM SIGPLAN Workshop on Intermediate Representations*. San Francisco, CA. January 1995.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *ECOOP '95, the 9th European Conference on Object-Oriented Programming*. Aarhus, Denmark. August 1995.

[12] A. Diwan, K. McKinley, and J. E. Moss. Using Types to Analyze and Optimize Object-Oriented Programs. *In ACM Transactions on Programming Languages and Systems,* 23(1), 30-72. 2001.

[13] D. Frampton, S. Blackburn, P. Cheng, R. Garner, D. Grove, J. Moss, S. Salishev. Demystifying magic: high-level low-level programming. In *Virtual Execution Environments (VEE '09)*. Washington, DC. March 2009.

[14] N. Glew, S. Triantafyllis, M. Cierniak, M. Eng, B. Lewis and J. Stichnoth. LIL: An Architecture-Neutral Language for Virtual-Machine Stubs. In *3rd Virtual Machine Research and Technology Symposium*. San Jose, CA. May 2004.

[15] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization (CGO '03)*. San Francisco, CA. March, 2003.

[16] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. . In *Object Oriented Systems, Languages, and Applications (OOPSLA '97)*. Atlanta, GA. October 1997.

[17] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, T. Nakatani. A Study of Devirtualization Techniques for a Java Just-in-time Compiler. In *OOPSLA '00, the 15th Annual Conference on Object-Oriented Systems, Languages, and Applications*. Minneapolis, MN. October 2000.

[18] T. Kotzmann, C. Wimmer, H. Mossenbock, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. In *ACM Transactions on Architecture and Code Optimization (TACO)*, Volume 5, Issue 1. May 2008.

[19] M. Paleczny, C. Vick, and C. Click. The Java™ HotSpot Server Compiler. In *JVM '01*. Monterey CA. April 2001.

[20] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Dynamic Languages Symposium (DSL '06)*. Portland, OR. October 2006.

[21] D. Ungar, A. Spitz, and A. Ausch. Constructing a meta-circular virtual machine in an exploratory programming environment. In *Object Oriented Systems, Languages, and Applications (OOPSLA '05)*. San Diego, CA. October 2005.

[22] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value Dependency Graphs: Representation without Taxation. In *Principles of Programming Languages (POPL '94)*. Portland, OR. January 1994.

[23] C. Wimmer. Linear Scan register allocation for the Java HotSpot client compiler. Master's thesis, Institute for Systems Software, Johannes Kepler University Linz. 2004.

[24] G. Wright, M. Seidl, and M. Wolczko. An object-aware memory architecture. Science of Computer Programming 62(2): 145-163 (2006).

[25] T. Wuerthinger, C. Wimmer, and H. Mossenbock. Array bounds check elimination in the context of deoptimization. *In Science of Computer Programming*, Volume 74, Issue 5-6. March 2009.