USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA
May 6–7, 2004

LIL: An Architecture-Neutral Language for Virtual-Machine Stubs

Neal Glew Spyridon Triantafyllis* Michał Cierniak†
Marsha Eng Brian Lewis James Stichnoth

Microprocessor Technology Lab, Intel Corporation

Abstract

High-performance MREs (managed runtime environments) that run either Java[1] or CLI applications require machine code sequences, called *stubs*, to implement such runtime support operations as object allocation, synchronization, and native method invocation. Due to the frequency of these operations, implementing stubs efficiently is critical for performance. Also, the number of different stubs that have to be created and maintained makes stub creation a sizable part of an MRE's implementation. Stubs typically require access to low-level resources such as registers and the call stack, and often must be specialized at runtime for particular classes or methods. Although stubs can be implemented by generating hand-crafted machine code at runtime, this approach is tedious and error-prone, and leads to stubs that are non-portable and difficult to maintain.

To address these problems, we designed a domain-specific language, called LIL, for implementing stubs. LIL is low-level but architecture-neutral, allowing the creation of stubs that are both portable and efficient. LIL also abstracts away many implementation details, making stubs easier to read. It is lightweight enough to be used for dynamic stub generation. LIL's validity checker helps us to catch many errors early. Our preliminary experience using LIL indicates that it greatly eases development and maintenance of stubs without sacrificing performance.

1 Introduction

The *Open Runtime Platform* (ORP [5]) is an experimental managed runtime environment (MRE) developed by the Programming Systems Lab at Intel. ORP supports both Java [11] and Common Language Infrastructure (CLI [8]) applications. To provide a credible starting point for experimentation, ORP's performance is comparable to that of other state-of-the-art MREs. Also, ORP supports multiple platforms with as little effort as possible. To date, ORP runs on both Intel® IA-32 and Itanium® Processor Family (IPF) architectures, and on both the Windows and Linux operating systems.

One of the most distinctive features of ORP is its modularity. ORP comprises three components: a just-in-time compiler (JIT), a garbage collector, and a core virtual machine (VM). Interaction between these components is allowed only through strictly defined interfaces. Thus the VM, the JIT, and the garbage collector are isolated from each other's implementation details. This greatly simplifies experimentation. For example, we can experiment with new compilation techniques or garbage collection approaches by modifying just the JIT or the garbage collector respectively, without having to modify other ORP components. To date, we have implemented seven different JITs (see, *e.g.*, [2, 7, 4, 1]) and five different garbage collectors for use within ORP.

ORP uses optimized machine code sequences, called *stubs*, to implement a number of common runtime support operations including object allocation, synchronization, and exception handling. Due to ORP's modular design, the JIT does not possess enough information to directly generate the code for most of these stubs. Instead, the JIT has to rely on support from the VM, and sometimes from the garbage collector as well. These runtime support operations usually require direct access to low-level machine resources such as registers and the call stack, and often need to be generated dynamically at runtime.

In the past, the ORP VM generated stubs by directly emitting hand-written machine code. However, this approach is error prone and results in stubs that are difficult to maintain. Stubs generated in this way are also not portable: a different

*Spyridon is currently at Princeton University.

†Michał is currently at Microsoft Corporation.

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

version of each stub is needed for each of the platforms supported by ORP. This situation led us to design LIL, a domain-specific language for expressing runtime support functionality. In this paper, we present LIL and argue that it greatly simplifies the creation of stubs, both by making stubs more concise and readable, and by hiding architecture-dependent details. Furthermore, LIL provides these benefits without performance loss.[2] Indeed, using LIL facilitates experimentation with stub code and with ORP's runtime support system in general, possibly leading to performance improvements.

In the next section, we describe stubs in more detail, motivate the LIL approach, and present an object allocation stub as an example that illustrates some of the issues involved. Sections 3 and 4 present the LIL language in detail and outline its benefits. Section 5 presents preliminary results showing that LIL's performance is similar to hand-written assembly stubs. Finally, sections 6 and 7 discuss related and future work.

2 Stubs

2.1 Motivation of Our Approach

Runtime support stubs are ubiquitous in ORP. Examples include object allocation, type checking (`instanceof`, `checkcast`, and `aastore` JVM bytecodes), exception throwing, native-method invocation (JNI), delayed method compilation, synchronization, arithmetic helpers, and class initialization. Since these operations appear often in Java applications, implementing stubs efficiently is critical for performance. Also, the sheer number of different stubs that must be supported by ORP makes it necessary to generate stubs in a portable and maintainable way. The purpose of LIL is to answer both these challenges.

Several approaches have been used in the past to implement runtime support code:

1. Stubs can be written in a high-level language such as C, and compiled with the rest of the VM source code. The VM then responds to requests for runtime support routines by passing the addresses of these precompiled functions to the JIT. In the past, ORP used this approach for a few stubs.

2. Stubs can also be implemented in assembly language.

3. The VM can generate machine-code stubs at runtime. This was ORP's approach before LIL.

4. The VM can express runtime support stubs in Java bytecode. The JIT then compiles these stubs in the usual way. Operations that cannot be expressed in bytecode, such as those that violate Java's type system, can be handled through "magic" method calls that are recognized and inlined into machine code by each JIT. This approach is taken by the Jikes RVM [3], and is discussed further in Section 6.

5. The VM can generate stubs directly in the JIT's IR.

None of these existing approaches fit ORP's needs, for the following reasons.

1. **Dynamic code generation.** While some stubs can be generated statically, at ORP build time, others must be customized at runtime. For example, the stubs for invoking native methods must be generated for each native method. The set of native methods is not known until class-load time, and changes as new classes are loaded. This dynamic code generation precludes using precompiled stubs, as in the first and second approaches above.

   In addition, many frequently executed operations, such as type checking and object allocation, are significantly more efficient if they are customized on a per-type basis. For example, the ORP VM can generate a custom `checkcast` sequence for each class in a program, which is significantly more efficient than a general implementation of `checkcast`.

2. **Low-level access.** Many runtime support operations need to directly access low-level resources such as processor registers or the call stack. For example, ORP's IPF implementation keeps some global values in registers, including the pointer to the data structure for the current thread. Examples of operations that need to directly manipulate the stack include exception throwing, object allocation, and native method invocation. Such low-level operations cannot be implemented in a high-level language, Java bytecode, or most JIT IRs.

   Furthermore, certain frequently invoked stubs are so performance-critical that they must take full advantage of the underlying processor. This makes it necessary to implement such stubs in hand-crafted machine code, or at least in a sufficiently low-level language.

[2] Actually we still need a few hand-written stubs to get the best performance. We expect that future tuning of LIL will eliminate this need.

The figure below uses the following constants: `fpo` is the offset of the frontier pointer in the per-thread structure and `lpo` is the offset of the limit pointer in the per-thread structure.

	IA-32	IPF	LIL

1.	`push ebx`
	`push ebp`
	`mov ebx, [esp+20]`
	`mov ebp, [esp+16]`
			`entry 0:managed:`
			`  g4,pint:ref;`
			`locals 3;`

2.	`mov ecx, fs:[0x14]`	`adds r18 = 0h, r4`	`l0=ts;`

3.	`mov eax, [ecx+fpo]`
	`mov edx, [ecx+lpo]`
		`adds r14 = fpo, r18`
		`ld8 r8 = [r14]`
		`adds r15 = lpo, r18`
		`ld8 r16 = [r15]`
			`ld r,[l0+fpo:ref];`
			`ld l1,[l0+lpo:pint];`

4.	`add ebx, eax`
	`cmp ebx, edx`
	`ja slowpath`
		`add r17 = r8, r32`
		`cmp.ltu p0,p6=r16,r17`
			`l2=r:pint+i0;`
			`jc l2 >u l1,slowpath;`

5.	`mov [eax], ebp`
	`mov [ecx+fpo], esi`
	`pop ebp`
	`pop ebx`
	`ret 8`
		`(p6) st8 [r8] = r33`
		`(p6) st8 [r14] = r17`
		`(p6) br.ret b0`
			`st [r+0:pint],i1;`
			`st [l0+fpo:pint],l2;`
			`ret;`

6.	`slowpath:`
	`pop ebp`
	`pop ebx`
	`/* push_m2n */`
	`push [esp+36]`
	`push [esp+44]`
	`call gc_alloc`
	`add esp, 8`
	`/* pop_m2n */`
	`ret 8`
		`/* push_m2n */`
		`adds r53 = 0h, r32`
		`adds r54 = 0h, r33`
		`brl.call gc_alloc`
		`/* pop_m2n */`
		`br.ret b0`
			`:slowpath;`
			`push_m2n 0;`
			`in2out platform:ref;`
			`call gc_alloc;`
			`pop_m2n;`
			`ret;`

Figure 1: IA-32, IPF, and LIL code sequences for object allocation. The `push_m2n` and `pop_m2n` code sequences are omitted for brevity.

3. **Portability.** The requirement to generate low-level code suggests using the second and third approaches, that is, using assembly language to implement stubs. However, this causes portability and maintainability problems. New assembly-language implementations would be needed for each new processor and operating system. Since ORP contains many stubs, this would be onerous.

4. **VM/JIT dependencies.** Maintaining a clean interface between the JIT and the VM is a key ORP goal. Thus we cannot use the fifth approach, generating stubs directly in the JIT's IR, since this approach would tie the VM to a particular IR, and perhaps to a particular JIT implementation. It would be necessary to rewrite the stubs each time the JIT's IR is changed.

5. **VM/MRE dependencies.** Because ORP supports both Java and CLI, approach 4 requires that one of these languages be chosen as the stub implementation language, making ORP asymmetric and more dependent upon that language. An alternative would be to write all stubs in both languages, which is as onerous as writing them for multiple architectures.

6. **Stub inlining.** Since runtime support stubs are invoked often, implementing them using called functions can incur significant call overhead. Directly inlining runtime support stubs into JIT-generated code would reduce this overhead. Approaches 1–3 are clearly not suitable for this purpose. Although approaches 4 and 5 could achieve this, they suffer the limitations noted above. Section 4.5 discusses how LIL can be used to provide a solution to this problem.

For these reasons, none of the existing approaches for generating runtime support code is appropriate for ORP, which motivated us to develop LIL. In the rest of this paper we argue that the use of LIL provides an attractive balance between performance, portability, and modularity. The main drawback of LIL is that it requires another language and compiler within ORP. This drawback is mitigated by the simplicity of the language—the implementation adds only 8000 lines of code.

In the remainder of this section, we give an example, the object allocation stub, and show how it is implemented in IA-32 and IPF assembly code, and in LIL. This example illustrates some of the issues involved in creating stubs. Finally, before presenting the LIL language in Section 3, we discuss in more detail the thread-local storage and stack-marking issues, because they occur so often in stubs for managed runtime environments.

2.2 Example

As an example of runtime support code, consider the object allocation stub. Object allocation is implemented by the garbage collector. A high-performance garbage collector typically provides each thread with its own thread-local allocation area. This allows a thread to allocate new objects quickly, without having to acquire a global heap lock. The thread-local allocation area can be represented as a frontier pointer and a limit pointer. The allocation sequence can be divided into a fast path and a slow path. First, the requested object size (including all necessary alignment padding) is added to the frontier pointer, and the result is compared against the limit pointer. If enough space is available, the fast path is executed. This path updates the frontier pointer, initializes the newly allocated object by clearing all its fields and setting its virtual-method table (*vtable*) pointer, and returns it. If the available space is not enough, the slow path is executed. This path marks the execution stack in case garbage collection and associated stack walking are necessary, and calls into the garbage collector to perform the allocation.

Figure 1 shows the code for object allocation. Columns one and two show the hand-written IA-32 and IPF assembly code, and column three shows the equivalent LIL code, which is described in detail in Section 3. The stub is intended to be called directly from JIT-generated code, and is called with two arguments: the object size in bytes and the vtable pointer that corresponds to the object type. Notice the differences between the first two columns—the different instruction sets and the different calling conventions. The purpose of LIL is to abstract away these differences by generating platform-specific code sequences automatically. In this way, the VM could be ported to a new platform without needing to rewrite and maintain a large body of custom stub code.

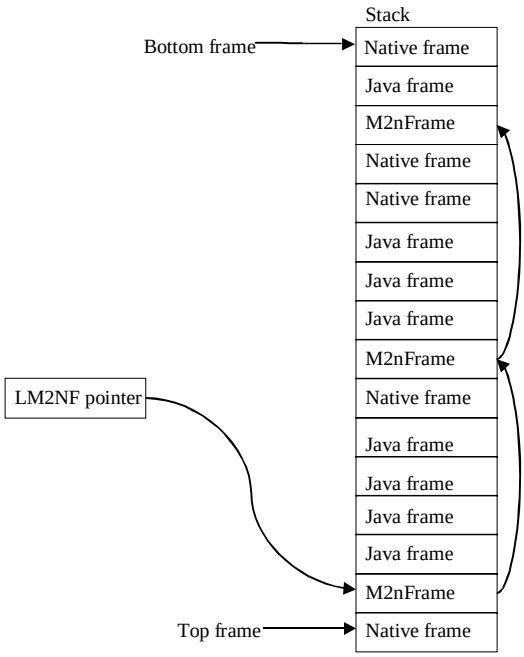

Figure 2: List of `M2nFrame`s used for stack walking.

2.3 MRE-Specific Issues

Another difference between columns one and two of Figure 1 concerns MRE-specific details. ORP has a per-thread structure that holds information about each thread, including the frontier and limit pointers for thread-local memory allocation. A pointer to this structure is stored in the thread-local storage provided by the platform's threading system. Thus, a stub can obtain a pointer to the ORP thread structure for the current thread by loading it from the thread-local storage. We call this operation *loading the thread pointer*. How it is done varies by both architecture and operating system. Row two of Figure 1 shows how it is achieved on Win32 operating systems. On the IA-32 architecture, it is loaded from a fixed offset in the `fs` segment; on the IPF architecture, ORP keeps the pointer in register `r4` (normally a preserved register). On Linux, the pointer is obtained by calling a function provided by the `pthreads` library.

Another MRE-specific issue is `M2nFrame`s. Each time JIT-generated code calls native code, ORP re-

quires that a marker be inserted between the two activation frames on the stack. This marker is necessary because of the way ORP implements stack walking, used in tasks such as root-set enumeration, exception propagation, and stack introspection. We call these markers managed-to-native frames, or `M2nFrame`s. Figure 2 shows the layout of a typical thread's stack. Combining information stored in the `M2nFrame`s with the available information on managed code, the VM can easily locate each managed activation frame on the stack. The details of this scheme are not important to this paper. What is important is that any stub that might cause stack walking, either directly or indirectly, must push an `M2nFrame` onto the stack upon entry, and pop it before returning. The `M2nFrame` is actually part of the stub's activation frame. Also important to this paper is that the details of setting up `M2nFrame`s are specific to the implementation of the VM and to the architecture. Therefore, stub code cannot be made portable unless these details are abstracted away.

3 LIL

We designed the LIL[3] language in order to enable the creation of platform-neutral stubs in ORP. We have implemented LIL, including code generators for the IA-32 and IPF architectures, within the ORP framework. This section introduces LIL by example and then provides a detailed description of the language.

3.1 Example

The last column of Figure 1 depicts a LIL stub for allocating an object. This stub is compiled into code that acts like a function. The stub's `entry` line states that it is called using the managed-code calling convention[4] with two arguments, and that it returns a result. The arguments are of type `g4` (32-bit general-purpose value) and `pint` (pointer-sized general-purpose value, often used for pointers other than object references), and the result is of type `ref` (reference to an object in the heap). The "0" is the number of standard places, which are described in Section 3.2.1. The rows in the figure do the following:

1. This row has the entry declaration and declares that the rest of the stub will use three local variables.
2. This row loads the thread pointer into `l0` (the first local variable).
3. This row loads the frontier and limit pointers. Both instructions load from the address equal to `l0` plus a constant (`fpo` and `lpo` respectively); these constants are the offsets of the frontier and limit pointers in the ORP thread structure. The first instruction loads a `ref` into `r` (the return variable); the second loads a `pint` into `l1`.
4. This row computes the new frontier pointer and compares it to the limit pointer. The first instruction adds `r` and `i0` (the first input, the size of the object to be allocated) and places the result in `l2`. The second instruction branches to label `slowpath` if `l2` is greater than `l1`, otherwise it continues with the next instruction.
5. This row initializes the object's vtable pointer, updates the frontier pointer, and returns. The first two instructions store `pint`s at the addresses `r` and `l0` plus the constant `fpo` respectively. In the first instruction, the value stored is `i1` (the vtable pointer), in the second it is `l3`. The final instruction returns, and the value returned to the caller is the current value of `r`.
6. This row is the slow path. It starts with the declaration of label `slowpath`, the target of the conditional jump above. Next it pushes an `M2nFrame`. Then it sets up to call a function according to the platform calling convention (*i.e.*, the calling convention used by the platform's C compiler), using the arguments to the stub as the arguments for the call. Then it calls `gc_alloc` and sets `r` to the value returned. Finally it pops the `M2nFrame` and returns.

3.2 The Language

A LIL stub specifies code that is like a function in a high-level language. Like a function, it takes arguments and can return a result. The stub specifies which of several calling conventions it conforms to. Also like a function, each stub executes with an activation frame on the stack. Conceptually, this activation frame is divided into a number of areas that can vary in size and type across the execution of the stub. For our purposes, these areas are: inputs, standard places, locals, outputs, and return. The inputs initially hold the arguments passed by the caller, but they can change by assignment. Their number and type is fixed across the stub. Standard places are described in Section 3.2.1. The locals hold

[3] LIL stands for Low-level Intermediate Language, and its pronunciation suggests its "little"-ness or lightweight nature.

[4] ORP specifies a calling convention that all JIT-compiled code must conform to; this is called the *managed-code calling convention*.

values local to the stub. Their number is determined by `locals` declarations, and their types are determined by a simple flow analysis. The outputs hold values passed to functions called by the stub. Their number and types are determined by `in2out` and `out` declarations. These declarations set up an output area, and various call instructions perform the actual call. In the case of `out` there must be assignments to the outputs between the declaration and the call. The return variable is a single location that is present following a `call` instruction or whenever an assignment is made to it; its type is determined by a flow analysis. Each input, standard place, local, output, and return is a LIL variable, and is referred to using the names `i0`, `i1`, ..., `sp0`, `sp1`, ..., `l0`, `l1`, ..., `o0`, `o1`, ..., and `r`, respectively.

All LIL variables and operations are typed by a simple type system. The type system makes just enough distinctions to know the width of values and where they should be placed in a given calling convention. For example, the type system distinguishes between floating-point and general-purpose values but not between signed and unsigned. In addition, the type system distinguishes various kinds of pointers (*e.g.*, pointers to heap objects versus pointers to fixed VM data structures), because in the future we may want the LIL code generator to be able to enumerate garbage-collection roots on LIL activation frames. The complete list of types is: `g1`, `g2`, `g4`, and `g8` for general-purpose values; `f4` and `f8` for floating-point values; `ref` and `pint` for pointer values; and `void` for return types only (nothing returned).

The LIL language includes a validity checker. This check makes sure that each stub is sensible, and can catch some basic errors. The conditions checked include the following. All labels used must be declared exactly once. The last instruction cannot fall through, it must jump or return (both tail calls and no return calls satisfy this requirement). Every control flow path to a point must set up the activation frame in consistent ways (*i.e.*, same number of locals, number and type of outgoing arguments, presence or absence of an `M2nFrame`, *et cetera*). This last condition is checked by a straightforward dataflow analysis. The validity checker also imposes certain restrictions on the program that make code generation easier. Space prevents a detailed description of the validity check.

Syntactically, a LIL stub consists of an entry declaration followed by a sequence of other declarations and instructions. The declarations and instructions are described in the following subsections, except that we first describe what standard places are. This section ends with a brief description of the implementation of LIL in ORP.

3.2.1 Standard Places

Standard places are a set of implicit arguments, which can be passed to a stub in addition to the normal arguments passed via the regular calling conventions. To see why standard places are necessary, consider the example of *compile-me* stubs. When a new class is loaded, its methods do not need to be compiled immediately. Instead, a compile-me stub is installed. When the method is invoked for the first time, the compile-me stub causes the method to be compiled by the JIT, after which the compiled code is invoked with the original arguments.

Conceptually, the compile-me stub for an instance method taking an integer and returning a float does the following: It pushes an `M2nFrame` in case garbage collection occurs or exceptions are thrown during compilation. Then it calls the VM to compile the method, passing a pointer to the VM data structure representing the method. This function returns a pointer to the native code, the stub pops the `M2nFrame`, and then performs a tail call to the native code. Here is the LIL code that achieves this:

```
entry 0:managed:ref,g4:f4;
push_m2n;
out platform:pint:pint;
o0=method;
call jit_method;
pop_m2n;
tailcall r;
```

where `method` is a pointer to a VM data structure representing the method in question.[5] Except for the `entry` declaration and the value of `method`, the rest of the code above is common for all compile-me stubs. Therefore it would be desirable to factor this code out into a separate, generic stub. Compile-me stubs could then call this stub and pass it the value of `method` as an argument. However, passing an argument in the usual way would upset the argument stack, which is already holding the arguments of the method to be compiled.

The easiest and most efficient way to pass this argument between the two stubs is to use a separate argument-passing mechanism that does not interfere with the call stack. Standard places serve exactly this purpose. Passing information between stubs through standard places in LIL is roughly analogous to passing information between functions through

[5] ORP's compile-me stubs also contain some exception rethrowing code not shown here.

Table 1: LIL Declarations

LIL syntax	Description
`entry n:cc:Ts:RT;`	Stub signature
`entry n:cc:arbitrary;`	Stub signature
`out cc:Ts:RT;`	Call setup
`in2out cc:RT;`	Call setup
`locals n;`	`n` locals
`std_places n;`	`n` standard places
`:label;`	Label declaration

global variables in C. In the example, the method-specific stub becomes:

```
entry 0:managed:ref,g4:f4;
std_places 1;
sp0=method;
tailcall compile_me_generic;
```

The declaration `std_places 1;` creates a standard place. The next instruction assigns `method` to it. The tail call passes this extra argument to `compile_me_generic` using fixed scratch registers. The generic compile-me stub is the following:

```
entry 1:managed:arbitrary;
push_m2n;
out platform:pint:pint;
o0=sp0;
call jit_method;
pop_m2n;
tailcall r;
```

The entry declaration declares that one extra argument will be passed in a standard place. This standard place is then used in the third instruction `o0=sp0;` as the argument to the call to `jit_method`. Notice also that the entry declaration contains the keyword `arbitrary` instead of parameter and return types. This means that the stub should work for any number and type of parameters and type of return. Such a stub cannot access the inputs and cannot return—it must end with an unconditional jump, a call that does not return, or a tail call.

The need for this extra argument-passing mechanism is important to several classes of stubs. The standard-places mechanism is one aspect of LIL that is different from traditional compiler intermediate representations.

3.2.2 Declarations

LIL declarations are summarized in Table 1. An `entry` declaration must appear at the beginning of every stub, and only there. The other declarations can appear anywhere after the entry declaration, and take effect at the point where they appear. An entry declaration contains a colon-separated list of items. The first item is the number of standard places passed to the stub. The rest is the stub's signature. A *signature* comprises a calling convention, a comma-separated list of parameter types, and a return type. As well as the `managed` and `platform` calling conventions shown in the examples, there is `jni` for the calling convention specified by JNI [10], `rth` for runtime helpers, and `stdcall` (for the `stdcall` calling convention, needed for CLI). In some stubs, such as the generic compile-me stub, the keyword `arbitrary` is used instead of parameter and return types in the entry declaration. This means that the stub works for an arbitrary number of parameters of arbitrary types and an arbitrary return type. As previously mentioned, such a stub may not access input variables nor return.

The `out` and `in2out` declarations set up an output area for a subsequent call. The `out` declaration is followed by a signature for the function being called. The `in2out` declaration is followed by the calling convention and return type; the number and types of arguments to the callee are the same as for the stub. Strictly speaking, `in2out` is both a declaration and an instruction. In addition to setting up for the call, it also copies the input variables to the output variables. This may be more complicated than it seems, since the stub and the callee may follow different calling conventions. On the IA-32 architecture, for example, an `in2out` instruction inside a `managed` stub calling a `platform` function will need to reverse the order of arguments on the stack.

The instruction `locals n;` declares `n` local variables, which are then available in subsequent instructions. Similarly, `std_places n;` creates `n` standard places. Finally, `:l;` declares a label `l`.

3.2.3 General-Purpose Instructions

LIL includes instructions typically found in a low-level compiler intermediate representation for doing arithmetic, loading, storing, and control flow. These appear in Table 2. Instructions that are specific to a VM implementation, and ORP in particular, are described in the next section.

The arithmetic instructions include unary and binary operations common on CPUs, such as addition, subtraction, negation, bitwise operations, sign and zero extension, shifts, *et cetera*. In Table 2, `v` stands for a variable (*i.e.*, input, output, local, or standard

Table 2: LIL General-Purpose Instructions

Category	LIL syntax	Description
Arithmetic	`v = o;`	Move
	`v = uop o;`	Unary
	`v = o1 op o2;`	Binary
Memory	`ld v, addr;`	Load
	`st addr, o;`	Store
	`inc addr;`	Increment
	`cas addr=o1,o2, label;`	Atomic cmp and swap
Calls	`call o;`	Ordinary
	`tailcall o;`	Tail call
	`call.noret o;`	No return
	`ret;`	Return
Branches	`jc cond, label;`	Conditional
	`j label;`	Always

Table 3: LIL VM-Specific Instructions

Category	LIL syntax
`M2nFrames`	`push_m2n o(,handles)?;`
	`pop_m2n;`
	`m2n_save_all;`
JNI Handles	`handles = o;`
Thread Pointer	`v = ts;`
Allocation	`alloc v,n;`

place), and `o` stands for an operand (*i.e.*, a variable or immediate value). In addition, a coercion can be applied to an operand to change its type to an equivalent one. On a 32-bit architecture, `g4`, `ref`, and `pint` are equivalent; on a 64-bit architecture, `g8`, `ref`, and `pint` are equivalent. A coercion has the form `:T` following the variable or immediate.

The general form for loads and stores is `ld v,addr;` and `st addr,o;`. There is also a memory increment operation, `inc addr;`. This operation is used by stubs to increment statistics and performance counters. An address consists of an optional base variable, an optional index variable, a byte offset (which can be zero), and a type. The type is the type of the memory location being accessed. The index variable can have a scale of one, two, four, or eight. Both the complex address form and the `inc` instruction match well with the IA-32 architecture's memory operations and can easily be expanded to an efficient sequence of IPF instructions. In addition, an address can specify acquire or release semantics and load instructions can specify zero or sign extension of subword values.

There is also an atomic compare-and-swap operation, `cas addr=o1,o2,label;`, for use in synchronization stubs. It compares the first operand against the memory value given by the address. If these values are equal it stores the second operand into the same memory address; otherwise it jumps to the label. The whole operation appears atomic to all threads.

The examples showed uses of call instructions, `call` and `tailcall`. The form `call.noret` is like `call` except that the LIL writer is asserting that the function does not return. The LIL code generator will not generate native code beyond the actual call instruction. It also affects the validity of a LIL stub—`call` cannot end a stub but `call.noret` can. This form is commonly used for exception throwing functions.

We have already seen examples of conditional jumps. The conditions are formed using the common relational operators (*e.g.*, `==` and `<=`). LIL also has unconditional jumps.

3.2.4 ORP-Specific Instructions

The `push_m2n` instruction sets up an `M2nFrame` on the stack. It includes an operand and an optional `handles` keyword. The operand is a pointer to a VM data structure for a method. If the stub interfaces to a native method, the operand should be this native method. All other stubs should use NULL. The `handles` keyword is explained below. The instruction `m2n_save_all;` saves additional information to the `M2nFrame` needed for exception propagation. This information is not saved automatically by `push_m2n`, because doing so is expensive on IPF, and it is very often unnecessary.

All `M2nFrames` also include a list of JNI handles. These handles are a GC-safe mechanism for referencing objects. The `handles` keyword in the `push_m2n` instruction declares that such handles will be used, and the matching `pop_m2n` instruction will free them. Otherwise the freeing step is skipped. The instruction `handles=o;` sets the list pointer in the `M2nFrame` to the operand. This instruction just sets the pointer to a list of handles in the frame; other stub code is responsible for creating the necessary handle structures.

The allocation instruction `v=alloc n;` allocates `n` bytes of memory on the stack and places a pointer to it into `v`. This space is available until the stub returns. Stubs use this instruction to create structures, such as JNI handles, and to pass pointers to them.

```
1. entry 0:managed:f8,f8:f8;
   push_m2n method,handles;
   locals 2;
   alloc l0,8;
2. ld l1,[mi:ref];
   st [l0+0:ref],l1;
   st [l0+4:pint],0;
   handles = l0;
3. out platform::void;
   call gc_enable;
4. out jni:pint,pint,f8,f8:f8;
   o0 = jni_env;
   o1 = l0;
   o2 = i0;
   o3 = i1;
   call Java_java_lang_Math_pow;
   l1 = r;
5. out platform::void;
   call gc_disable;
6. l0 = ts;
   ld l0,[l0+ceo_offset:ref];
   jc l0=0,no_exception;
7. m2n_save_all;
   out platform::void;
   call.noret rethrow_current_exception;
8. :no_exception;
   r=l1;
   pop_m2n;
   ret
```

Figure 3: LIL code sequence for the JNI wrapper to `java.lang.Math.pow`.

3.2.5 JNI Stubs

To illustrate some of the allocation and `M2nFrame` setup features, this section discusses JNI stubs, which are the most complex stubs in ORP. These stubs are responsible for a transition from managed code to native methods implemented according to the JNI specification [10]. They need to perform a number of operations, including matching calling conventions, enabling garbage collections, handling synchronization, rethrowing exceptions, and converting references to and from handles.

As an example, consider `Math.pow`, which is a static native method that takes two doubles and returns a double. Figure 3 shows the LIL code that interfaces managed code to a JNI version of `Math.pow`. In the figure, `method` stands for the VM data structure for `Math.pow`, `mi` stands for the address of the field in the VM data structure for `Math` that points to the `java.lang.Class` object for class `Math`, `jni_env` stands for the JNI environment structure in the VM, and `ceo_offset` stands for the offset of the current exception object in the thread structure.

The first block declares the inputs and locals, pushes an `M2nFrame`, and allocates 8 bytes of space for storing an object handle on the stack. The second block adds the `Math` class object to the list of local object handles. The third block calls the VM's `gc_enable` function to allow garbage collection to occur while the thread is subsequently executing within native code. The fourth block calls the actual native method. The fifth block calls `gc_disable` prior to the return to managed code. The sixth block tests whether the native method threw an exception, branching to the `no_exception` label if not. Otherwise, the seventh block calls the VM function `rethrow_current_exception` to propagate the exception. The final block pops the previously pushed `M2nFrame` and returns to managed code.

In general, generating JNI stubs is a template-driven process. Some snippets of code are included in a JNI stub only for certain types of methods. For example, a JNI stub will include a synchronization snippet only if the corresponding method is declared `synchronized`. In fact, there are two different synchronization snippets, for static and non-static methods. Other code pieces are repeated for each of a method's arguments. Actually, different pieces of code are used for reference and basic-type arguments. JNI stub generation cannot be expressed as a simple "fill in the blanks" process. It actually takes hundreds of lines of C code to generate the LIL source code for each JNI stub.

3.3 Implementation

The LIL system has two parts: a parser and a code generator. The parser takes a C string as input and produces an intermediate representation (IR) of the LIL instructions. The code generator takes the LIL IR as input and produces machine instructions for a particular architecture.

The parser includes a `printf`-like mechanism for injecting runtime constants such as addresses of functions or fixed VM data structures. For example, here is the C code that generates the method-specific compile-me stubs we encountered in Section 3.2.1.

```
NativeCodePtr
  create_compile_me(Method_Handle m)
{
  LilCodeStub* cs = lil_parse_code_stub(
    "entry 0:managed:%0m;"
    "std_places 1;"
    "sp0=%1i;"
```

```
    "tailcall %2i;",
    m,
    m,
    create_compile_me_generic());
  assert(lil_is_valid(cs));
  NativeCodePtr addr =
    LilCodeGenerator::get_platform()->
      compile(cs);
  lil_free_code_stub(cs);
  return addr;
}
```

Our code generators make a couple of prepasses to gather information and decide where to locate LIL variables, and then a main pass to generate the code. All these passes are simple sequential scans of the instructions. Each LIL instruction is translated into a sequence of machine instructions based on information gathered in the prepass. No sophisticated optimizations, intermediate languages, or peephole optimizations are used, although we are considering whether a smarter instruction scheduler and template packer would improve performance on the IPF architecture.

LIL is very lightweight: The parser and infrastructure for LIL is about 3500 lines of code, the IA-32 code generator is about 1500 lines of code, and the IPF code generator is about 2200 lines of code. The code generators also use the `M2nFrame` modules, which are about 300 lines for each architecture, and the code emitters, which are 2200 and 13500 lines respectively. However, the `M2nFrame` modules and the code emitters would be needed even without LIL.

4 Benefits

The motivation for creating LIL was to improve the portability, maintainability, and correctness of stubs. These benefits are described in Sections 4.1 to 4.4. Perhaps surprisingly, using LIL can also improve performance. Section 4.5 describes how LIL can be inlined, or *implanted*, into JIT-compiled code to make ORP's runtime support system more efficient.

4.1 CPU Independence

The most direct benefit of implementing stubs using LIL is that LIL is architecture-neutral. Each stub is written only once, instead of being reimplemented for every platform. In addition, any performance enhancements or bug fixes applied to a stub are automatically propagated to all architectures. Such modifications have been common during ORP's development. Before we switched to LIL, machine-language stubs on the IA-32 and IPF architectures diverged over time, sometimes to the extent of implementing slightly different functionality. LIL stubs do not suffer from such problems.

4.2 OS Independence

Several high-performance stubs operate directly on data structures that depend on the VM and OS. Of these data structures, the two most important are thread-local storage and `M2nFrames`. LIL hides the implementation of these constructs, and allows stubs to access them in a platform-independent manner.

Loading the thread pointer is needed for efficient implementation of object allocation and synchronization, which are significant to the overall performance of ORP. As mentioned in Section 2.3, this operation is requires different sequences on Windows versus Linux, and IA-32 versus IPF platforms. As we saw in Section 3, LIL includes a primitive for loading the thread pointer. This enables us to use the same stubs on all platforms and operating systems. Any operating system dependences are hidden within the LIL code generator.

4.3 Readability

Because LIL is more high-level than machine language, most stubs become more concise and readable when expressed in LIL. This is particularly true for complex stubs, such as the ones used to implement JNI, described in Section 3.2.5. ORP's IA-32 implementation of JNI stubs has about 700 lines of C code; ORP's IPF implementation has about 400 lines of C code. The equivalent LIL implementation is about 320 lines of C code, and is also much easier to understand and modify. In particular, the IA-32 non-LIL implementation of JNI contains extensive debugging code, because certain aspects of setting up JNI stubs were especially error-prone. One such aspect is accessing the call stack; since the stub needs to push things onto the stack, offsets of stack values keep changing during the stub's execution. Also, transitioning between managed and JNI code on the IA-32 architecture requires reversing the order of arguments on the stack, which is particularly tedious. LIL code, on the other hand, sets up the stack with simple statements such as `o3=i2;` and offsets and argument orders are automatically taken care of by the LIL code generator.

4.4 Correctness

Implementing stubs directly in machine language makes it hard to ensure their correctness. In addition to being difficult to read, machine language offers no mechanisms for automated validity checking. This is especially problematic, since even small bugs in stubs are almost certain to break the system.

Using LIL helps ensure correctness in two ways. First, some of the most tedious conventions are abstracted away by LIL. For example, the stub implementor no longer has to worry about implementing calling conventions, about which system values are in what registers or memory addresses, or about which machine registers are available for storing local variables. The implementation of such conventions needs to be checked only once, in the LIL code generator.

Second, the LIL code generator can check LIL source code for consistency, as explained in Section 3.2. Although LIL is not a strongly typed language, the limited semantic checking it supports can still catch some of the most frequent bugs, such as providing the wrong number of arguments to a function, accessing incoming arguments that do not exist, or returning without setting the return variable. In a sense, the semantic checker of the LIL language is roughly equivalent in power to that of C. This is a big improvement over using assembly language, which has no semantic checking whatsoever.

4.5 Code implants

We previously discussed LIL's value as a tool for implementing stubs that are called by JIT-generated code. But these calls introduce some inefficiencies because many stubs, including those for runtime type identification and memory allocation, are called so frequently. One way to avoid this call overhead, while maintaining ORP's strict interface between the JIT and the VM, is to move toward an *implant-based* runtime support system. In such a system, the VM passes LIL sequences to the JIT instead of translating the sequences itself. The JIT then translates LIL sequences to its own intermediate representation, implanting them into the code it generates. Much like normal method inlining, code implanting not only avoids call overheads, but also exposes more opportunities for optimization.

In general, a LIL-based code implanting system would work as follows.

1. The JIT first makes a runtime support request to the VM. Along with the request, it passes context information about the stub's call site. Such information might include which stub arguments are constant or NULL, the profile weight of the call site, *et cetera*.

2. In response to that request, the VM generates a pre-optimized LIL stub using the context information and its knowledge of its own data structures and other implementation details.

3. The JIT receives back from the VM a snippet of LIL code instead of a stub address. The JIT can then translate the LIL code to its own intermediate representation. This translation is straightforward for LIL's general-purpose instructions, but not possible for ORP-specific instructions (see Section 3.2.4). Instead, the JIT can treat such instructions as black boxes.

4. During the code generation phase, the JIT can ask the VM to expand each black-box instruction to machine code.

A preliminary version of a LIL-based code implanting system has been implemented in ORP. This system implants the VM stubs for the `checkcast` and `instanceof` bytecodes into the *O3 JIT*, which is currently ORP's best-performing IA-32 JIT. Experiments using this system show that implanting just these two stubs improves overall performance by 3% for the SPEC JVM98 [14] suite. The current system implements only part of the scheme described above. A full-fledged system would be able to implant more stubs and would have more optimization opportunities. We expect the final implanting system to provide significantly greater performance improvements.

LIL code implants, their implementation, and their performance benefits are described in more detail by Cierniak *et al.* [6].

5 LIL Performance

In designing LIL, our goal was to obtain the benefits described in the previous section without sacrificing ORP's excellent performance. Since ORP previously implemented all its stubs through hand-crafted machine-code generation, it is possible to compare the performance of LIL stubs to that of hand-coded stubs. This section will present this comparison and show that we have mostly retained performance. Figure 4 lists the LIL stubs that have been written so far, and the stubs for which LIL versions have not yet been written. Of the non-LIL stubs, the CLI stubs and atomic compare-and-swap stubs should be straightforward—we have not gotten around to them yet. The other two stubs require additional features for LIL that are highly specific to those two stubs. Having the hand-coded assembly versions of these stubs is roughly equivalent to the additions that would be needed to the LIL code generator to support them.

We evaluate the overhead of LIL by measuring ORP with hand-coded assembly versions of all stubs versus ORP with LIL versions of the stubs, on both

LIL Stubs

- **Compilation**
  Compile-me generic, compile-me specific, recompile
- **Native-method interface**
  JNI and PInvoke stubs
- **Exceptions**
  Throw, lazy throw, throw specific exceptions, throw linking exception
- **Allocation**
  New object, new array, multinewarray, load constant string
- **Synchronisation**
  Monitor enter and exit for objects and classes
- **Type tests**
  Checkcast, instanceof, aastore
- **Arithmetic helpers**
  Float to integer, double to integer, long shifts, long multiplies, long divides, *et cetera*
- **Miscellaneous**
  Load interface vtable, initialise class, character-array copy

Non-LIL Stubs

- Native to managed transition
- Transfer control to exception handler
- CLI-delegate stubs
- CLI unboxers
- Atomic compare and swap

Figure 4: LIL and non-LIL Stubs

the IA-32 and IPF architectures. We make a few exceptions for several performance-critical stubs, whose assembly versions have been highly tuned for their particular platforms. These are the object monitor enter and exit stubs on the IA-32 architecture, and the new object, new array, and character-array copy stubs on the IPF architecture.

The IA-32 performance numbers were obtained on a 4-processor, 2.0 GHz Intel® Xeon™ processor-based machine with HyperThreading disabled, with 4 GB physical memory, and running Windows 2000 Advanced Server. The IPF performance numbers were obtained on a 4-processor, 1.5 GHz Itanium® 2-based system with 6 MB L3 cache and 16 GB physical memory, running Windows Server 2003, 64-bit edition.

We measured the performance of the seven components of SPEC JVM98 [14] as well as SPEC JBB2000 [15].[6] We used a 96 MB heap for the SPEC JVM98 components on both architectures, 1 GB for SPEC JBB2000 on the IA-32 architecture, and 4 GB for SPEC JBB2000 on the IPF architecture. In general, SPEC JVM98 models client-side applications, which do not typically require significant amounts of memory. SPEC JBB2000 is designed to model more memory-intensive server applications, hence the larger heap sizes.

Figure 5 shows the relative speedup of ORP with LIL stubs over ORP with only assembly stubs on both the IA-32 and IPF architectures. On the IA-32 architecture, LIL either has no effect or improves performance for half of our benchmarks, allowing for a 0.5% experimental noise margin, and shows no more than a 7% degradation for the rest. LIL is even more promising on the IPF architecture, with less than 4% slowdown for jess and db.

When all available LIL stubs are used, the slowdowns increase substantially, particularly for SPEC JBB2000. This is because the few assembly stubs included in Figure 5 have been highly tuned for their particular platforms. The IA-32 tuning includes the use of instructions that are more efficiently implemented on the Intel® Xeon™ processor. The IPF tuning include instruction scheduling and the use of some specialized instructions. We are currently investigating whether comparable tuning is possible in LIL. The work described here has allowed us to identify the critical stubs and thereby focus the LIL optimization efforts on a few architecture-specific issues.

6 Related Work

The Jikes RVM [3] is implemented almost entirely in Java. In order to support the unsafe low-level operations it needs to directly access memory, machine registers, and operating-system resources, Jikes includes "escape" mechanisms to circumvent Java's type system and memory model. Its primary escape mechanism is the `Magic` class. This includes static methods to do typecast, compare-and-swap, and memory fence operations, read and write memory, transfer control to specified addresses, and access stack frame contents (*e.g.*, caller's frame pointer, next instruction address). Each JIT for the Jikes RVM treats a call on a method of the Magic class specially: it implements the call using a sequence of inlined machine instructions. Although support for Magic and the other Jikes escape facilities must

[6] We use the SPEC benchmarks only as benchmarks to compare the performance of the various techniques within our own VM. We are not using them to compare our VM to any other VM and are not publishing SPEC metrics of any kind.

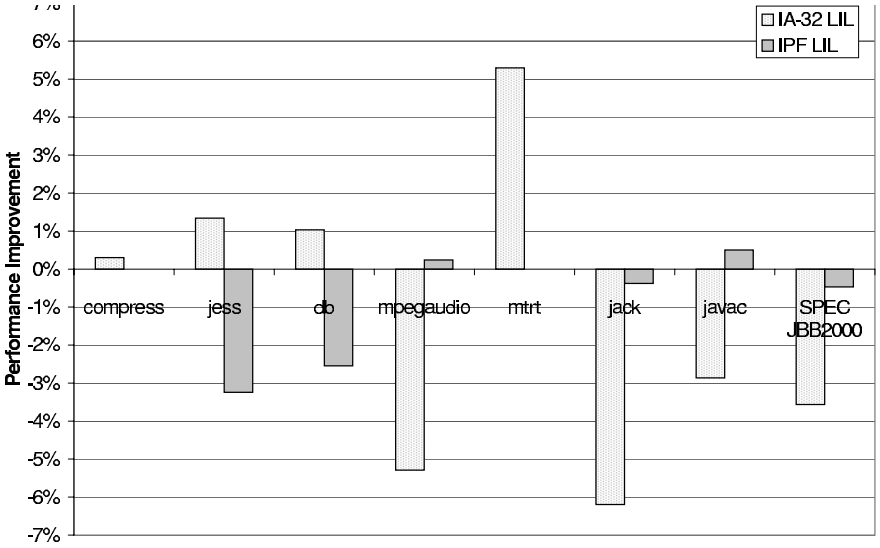

Figure 5: Performance impact of LIL.

be added to each new compiler, their implementation is relatively simple. The Magic class can be used to implement much of the same functionality as LIL. However, Magic is lower level: its operations are close to machine instructions. This makes code that uses Magic less readable. It also makes it easy to make mistakes, which can corrupt the internal state of the VM. Static analyses [12] have been implemented to check code using Jikes escape mechanisms for GC-pointer safety (that the system can determine at every GC point all the locations that contain references), but these analyses do not check for as many errors as the LIL code generator does. In addition, code using the Jikes escape mechanisms must often be reimplemented for each new machine architecture.

Griesemer [9] describes the facility used by the Hotspot virtual machine to generate machine code at runtime. C++ *Assembler* classes are used by the VM to emit stubs and other machine code: the VM calls an Assembler method to generate each machine instruction, to define labels, and other operations. So, a sequence of these calls are a kind of specification for a machine code sequence. Like LIL, the instruction-emitting Assembler methods are slightly higher-level than raw machine instructions since calling them may actually generate different instructions depending on which member of a processor family the application is running on. They also provide full control over the emitted code: assuming that the appropriate instruction-generation methods have been defined, any machine code sequence can be defined. However, these Assembler methods are architecture-specific, so different sequences of calls are needed for the IA-32 architecture versus for the IPF architecture. They are also extremely low-level, which makes it hard to understand or maintain the stub-generation code. ORP has a similar set of instruction-emitting classes for each processor architecture, and these were used before we developed LIL.

Microsoft's Shared Source Common Language Infrastructure [13] includes an interpreted language, ML, that is used to implement marshaling stubs executed during RPC calls or when transitioning between native library code and either the VM or JIT-generated code. ML includes opcodes to load and store values into marshaling buffers, convert values, and throw exceptions on errors. Unlike LIL, however, this language is specialized: it supports only argument marshaling.

Remote procedure calls (RPCs) simplify the construction of distributed programs. Part of this ease is the transparency of RPCs to the programmer, and this transparency requires the automatic construction of stubs to make calls across machines look like ordinary calls. Such stubs are similar in some ways to the native-method invocation stubs, and some of the issues are the same. However, they are different enough that the same solutions are not applicable.

7 Future Work

Many of LIL's features are the result of our experience using LIL to rewrite stubs that were formerly hand-coded. Such features include LIL's support for

loading the thread pointer, and its support for the `M2nFrames` that mark the transition between managed and native frames on the stack. However, LIL is not currently able to implement all the stubs required by ORP. We are considering whether it is worth while adding a couple of new features to LIL to support the two stubs that we cannot write in LIL.

Overall, we find the performance of LIL on both the IA-32 and IPF architectures satisfactory. However, there are still a few highly tuned stubs that LIL significantly under performs. Future work will try to improve the LIL code generators, so that LIL achieves similar performance.

Our early experiments on implanting LIL stubs into JIT-generated code have been very encouraging. We intend to continue these experiments and further develop the code implanting system. Other VM-emitted code sequences will be converted to LIL and then passed to the JIT for inlining and optimization. This will require the development of a general way for the JIT to inline LIL sequences. It will also be necessary to add to LIL any features required for implants.

8 Conclusions

The ORP managed runtime environment executes Java and CLI applications. It manages to combine high performance with modularity and experimentation ease. This is in part because of its use of optimized runtime support stubs, which are dynamically created by the VM to implement such common operations as object allocation, exception throwing, and native-method invocation. Problems with creating stubs using hand-written machine code led us to develop a new language, LIL, for specifying them. LIL is architecture-neutral and high-level than machine code, yet provides enough low-level facilities to achieve good performance and to allow low-level operations such as call-stack manipulations and register access.

We found LIL stubs to be shorter and more readable than the processor-specific machine-code sequences they replace. This has simplified their maintenance and has made it easier for us to experiment with new optimizations. We studied the performance impact of LIL stubs using SPEC JVM98 and SPEC JBB2000 running on ORP on both the IA-32 and IPF platforms. Our results demonstrate that using LIL retains most of the performance of hand-written stubs. That is, using LIL offers significant software-engineering advantages without significant performance losses.

While LIL is still under development, our experience suggests that both its performance and capabilities will continue to improve. We are particularly optimistic about the potential of *code implants*, LIL sequences passed to the JIT for optimization and inlining into the JIT-generated code. These implants not only avoid the overhead of stub calls, but expose further opportunities for optimization such as instruction reordering and scheduling.

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, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, , and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1), February 2000.

[4] A. Bik, M. Girkar, and M. Haghighat. Experiences with JAVA JIT Optimization. *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, October 1998.

[5] 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`.

[6] M. Cierniak, N. Glew, S. Triantafyllis, M. Eng, B. Lewis, and J. Stichnoth. Object-

Model Independence via Code Implants. *Proceedings of the Workshop on Multiparadigm Programming with OO Languages (MPOOL'03)*, October 2003.

[7] M. Cierniak, G.-Y. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[8] ECMA. *Common Language Infrastructure*. ECMA, 2002. Available at `http://www.ecma-international.org/publications/Standards/ecma-335.htm`.

[9] R. Griesemer. Generation of virtual machine code at startup. In *Proceedings of the OOPSLA '99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design*. Sun Microsystems, Inc., November 1999.

[10] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 1999.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[12] J.-W. Maessen, V. Sarkar, and D. Grove. Program analysis for safety guarantees in a java virtual machine written in java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 62–65. ACM Press, 2001.

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

[14] Standard Performance Evaluation Corporation. SPEC JVM98, 1998. See `http://www.spec.org/jvm98`.

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