Position Paper submitted to the OOPSLA'99 Workshop on
Simplicity, Performance and Portability in Virtual Machine Design

Generation of Virtual Machine Code at Startup

Robert Griesemer
Sun Microsystems, Inc.

Abstract

Performance-critical components of virtual machines are often implemented in assembly language. Traditionally, this code is compiled with an assembler and linked statically with the rest of the system. Generating this code at startup time has significant engineering advantages over the conventional approach: The generated machine code is much safer, easier to maintain, and has better performance. The virtual machine is easier to build and more portable.

Introduction

Ideally, a virtual machine (VM) is written entirely in a high-level language (e.g., C++ or Java) in order to be machine-independent and easily portable. For performance reasons however, many VMs make direct use of machine code. In the following we look at the architectural implications caused by the direct use of machine code. In particular we are concerned about *static* VM code that is written in assembly language. We will refer to this code as *VM machine code*. We will refer to all other VM code as *VM high-level code*. This paper is about generation of VM machine code at startup. This paper is not about compilers or JITs in general.

Traditionally, machine code is written in assembly language, compiled via an assembler, and then linked with the rest of the system at build time. This approach has several disadvantages:

- Since two different languages are used, two different compilation tools are needed: the assembler for the machine code, and the compiler for the high-level language. Using two different programming languages for a VM may not itself be a problem, but using *real-life* assembly language is. CPU manufacturers suggest mnemonics for machine instructions, but assembly language is not standardized to the degree of high-level languages. Different assemblers for one machine architecture may use slightly different mnemonics, and the syntax of assembly language constructs (data structure definitions, labels, macros, etc.) often differs considerably. To make things worse, some assemblers do not even parse according to a defined grammar, but seem to accept variations of examples provided in the documentation. Due to restrictions in assembly languages, more complicated code cannot even be expressed in a convenient way. Using assembly language makes VM code not only non-portable from one architecture to another, but makes it also dependent on the particular compiler or assembler. If everything were written in the high-level language, the programmer's life would be much easier.
- Both the compiler and the assembler apply tool-specific name mangling to identifiers in order to create the symbols used by the linker. In order for the link step to succeed, identifiers must be chosen carefully so the corresponding symbols match. Thus, the inverse name mangling must be applied manually. This is tedious and error-prone. It too makes the VM code dependent on compiler and assembler.
- Some dependencies of the machine code from the high-level code cannot be expressed explicitly and without overhead (i.e., not just as comments or via extra indirection) in assembly language. For example, sizes or offsets of data structures defined in the high-level code but referred to in the machine code may not be accessible there since the compiler doesn't generate linker symbols for them. Simply defining the corresponding data structures or constants in assembly language works as long as they are kept up-to-date. Making changes at one place requires a manual update at another place and is error-prone. Bugs introduced this way in the machine code are very hard to find.

Some compilers for high-level languages allow machine code to be mixed in. However, the assembly language

syntax accepted by these compilers is usually restricted, not standardized, and doesn't allow the implementation of significant-size machine code components. Thus, while this option may occasionally be useful, it doesn't replace a full assembler.

It may seem that these disadvantages are caused by the fact that machine code is used, that they are essentially of an organizational nature, and that they do not have a strong impact on the architecture or design of the VM itself. We argue that the contrary is true, and suggest that machine code be generated at startup using the high-level language. Using dynamic code generation for the VM code itself, machine code can be used in a VM (and any other software for that matter) without using assembly language or an assembler.

In the following we share our experience with dynamic code generation of machine code for our Smalltalk and Java Virtual Machines.

Application

We started using dynamic code generation for machine code in our Smalltalk VM [1]. That VM was using a bytecode interpreter and employed a sophisticated dynamic compilation system. In order to have full control over the machine and performance, originally the interpreter was written in Intel x86 assembly language. The rest of the system was written in C++. During the development process we experienced all the disadvantages mentioned above.

We tried other assemblers and other C++ compilers without much success. In mid-project we decided not to use assembly language and an assembler anymore for the machine code components, but instead to generate them at startup time. In fact, the interpreter ended up being partly written in assembly language and partly generated. Generating the machine code components proved to be so much more convenient, that we used the same code generation technique for the Java™ HotSpot™ Performance Engine [2]. The system's interpreter and many stub routines are generated at startup time; and the Java VM is written entirely in C++.

The assembly code generation is based on a few C++ classes that provide the necessary infrastructure; the most prominent one being the class *Assembler*. This class implements functions to emit code for all machine instructions needed. The names of the functions match the assembly code mnemonic closely. Assembler function calls are used to emit machine instructions into a *CodeBuffer*. The CodeBuffer manages a piece of memory into which code is generated. For "static" machine code, this memory is allocated statically. The Assembler uses additional data structures such as *Registers*, *Addresses*, and *Labels*. Registers encode processor registers, Addresses express more complicated addressing modes, and Labels transparently keep track of fixup chains for jumps. A concrete implementation for the Intel x86 processor family looks as follows:

```
class Register;
const Register eax;
const Register ebx;
// etc.

enum Scale { times_1, times_2, times_4, times_8 };
class Address {
 public:
  Address(Register base, int offset);
  Address(Register base, Register index, Scale factor, int offset);
  // etc.
};

class Label;
class CodeBuffer;
class Assembler {
 public:
  Assembler(CodeBuffer* code);

  // moves
  void movl(Register dst, int imm32);            // dst = imm32
  void movl(Register dst, Register src);         // dst = src
  void movl(Register dst, Address src);          // dst = memory.at(src)

  // branches
  bind(Label& L);                                // define label L
```

```
  jmp(Label& L);                                  // jump to L always
  jnz(Label& L);                                  // jump to L if not zero
  jl(Label& L);                                   // jump to L if less

  // operations
  void addl(Register dst, Register src);          // dst = dst + src
  void decl(Register reg);                        // reg = reg - 1
  void cmpl(Register x, Register y);              // compare x & y

  // jumps and calls
  void jmp(int dest);                             // jump to address dst
  void call(int dest);                            // call address dst

  // etc.
};
```

In the Java™ HotSpot™ Performance Engine we use several subclasses of Assembler, which provide specialized "macro" functionality. A simple generator function that generates code to clear an array looks as follows when using our code generation framework:

```
#define __ asm->                                  // helps readability!

char* generate_clear_array(Assembler* asm) {      // eax holds ptr to array
  AssemblyStub as(asm, "clear array");            // stack-allocated helper
  const int lo = length_offset();                 // length field offset
  const int bo = base_offset();                   // array base offset (a[0])
  __ movl(ecx, Address(eax, n);                   // ecx = eax.length() (> 0)
  __ xorl(ebx, ebx);                              // ebx = 0
  { Label L;
    __ bind(L);                                   // do {
    __ decl(ecx);                                 //   ecx--
    __ movl(Address(eax, ecx, times_4, bo), ebx);//   eax[ecx] = ebx
    __ jnz(L);                                    // } while (ecx != 0)
  }
  return as.entry();                              // get entry address
}
```

This example also illustrates the use of helper classes such as *AssemblyStub*: AssemblyStub is a stack-allocated C++ class. It collects *reflective information* about the generated machine code which then can be queried; e.g. for debugging. A typical query is: To which machine code stub belongs this pc?

In the Java™ HotSpot™ Performance Engine, all machine code components are generated during initialization of the VM. Since the "assembly language" is C++, a high-level approach is possible. The interpreter is table-generated, with the table consisting of generator functions for individual bytecodes. All the dispatch and auxiliary code is factored out. A new bytecode can be added to the system in a matter of minutes. All machine code provides reflective information, which is used for vital VM functions but also for debugging. Offset into data structures are computed in C++ (e.g. length_offset() in the example above), making it possible to change data structure layouts without a single change in the generator functions.

Experience

Our experience with this code generation approach is very positive. The advantages are many:

- We have full control over the generated machine code (on the bit-level!).
- There are no hassles with unspecified assembler languages or unsupported machine instructions.
- The build process is much simpler.
- One executable can optimally support code for different processors of a family (e.g. Pentium vs. Pentium Pro), since instruction selection happens at runtime. The result is better performance.
- Different machine code can be generated without recompilation depending on flags provided on the command line. This advantage cannot be overestimated; it proved to be extremely helpful for debugging.

- It is possible to maintain reflective information for machine code, including relocation and GC information. For example, it is possible to embed into the machine code direct pointers to VM objects that are garbage-collected. Garbage collection is then using the reflective information to find the pointers in the code.
- Dependencies between VM high-level code and VM machine code can be expressed explicitly (e.g. field offsets of data structures). The layout of objects referred to from the VM machine code can change completely, and no changes are necessary elsewhere.
- All the features of the high-level language used for the VM implementation are available for machine code generation (e.g., loops, function calls, and recursion).
- The generated code can be optimized if desired (e.g., by using a peephole optimizer).

Among the disadvantages we have:

- The executable for the VM becomes bigger since it contains the code generation framework, which is used only once. This is not so much a problem if the VM employs a dynamic compiler or JIT anyway, in which case code can be shared. In our VM, the entire CodeBuffer and Assembler infrastructure is shared among the different VM components.
- Extra execution time is required to generate the machine code first.
- The executable cannot fit into a ROM entirely; the generated code must go into RAM.
- The generated code can only be called indirectly; i.e., via function pointers.

In our case only the first disadvantage is an issue at all. If we would use statically assembled machine code, the VM executable could be smaller. The extra execution time (slower startup time!) of the VM is negligible: Generating the whole Java interpreter takes less than 10ms on a 400MHz PII. Overall, the advantages are outweighing the disadvantages by far.

Conclusion

We have made a case for dynamic code generation of traditionally statically assembled machine code in virtual machines. If a virtual machine is employing a dynamic compiler or JIT already, the code generation infrastructure can be reused. In our experience, generating the machine code at run-time comes with significant engineering benefits. The result is a virtual machine that is easier to build, maintain and debug.

Acknowledgments

Thanks to Lars Bak, Steffen Grarup, and Srdjan Mitrovic who provided valuable comments on earlier versions of this paper.

References

[1] Smalltalk Virtual Machine developed at Longview Technologies, LLC, doing business as Animorphic Systems. The VM was exhibited publicly at OOPSLA'96 in San Jose, California. Longview Technologies was acquired by Sun Microsystems, in February 1997.

[2] Java™ HotSpot™ Performance Engine by Sun Microsystems, Inc. More information, including white papers and download information can be found at: http://java.sun.com/products/hotspot/launch/download.html.