

Automatic Generation of Machine Emulators: Efficient Synthesis of Robust Virtual Machines for Legacy Software Migration

Michael Franz

Andreas Gal

Christian W. Probst

University of California, Irvine

Technical University of Denmark

Abstract: As older mainframe architectures become obsolete, the corresponding legacy software is increasingly executed via platform emulators running on top of more modern commodity hardware. These emulators are virtual machines that often include a combination of interpreters and just-in-time compilers. Implementing interpreters and compilers for each combination of emulated and target platform independently of each other is a redundant and error-prone task. We describe an alternative approach that automatically synthesizes specialized virtual-machine interpreters and just-in-time compilers, which then execute on top of an existing software portability platform such as Java. The result is a considerably reduced implementation effort.

1 Introduction

A substantial amount of legacy software has been designed for mainframe architectures that are no longer commercially available. As the last computers implementing such an architecture reach the end of their physical life span, the best option available if one wants to continue running the legacy software is to construct a software emulator for the legacy hardware architecture. Because of the effects of Moore’s law, a legacy mainframe program that is being emulated in software on a modern microprocessor often does not even run much slower than the original program did on the old hardware.

Programs that need to be emulated in this manner are typically “niche” programs—if a program were of critical importance and used widely, it would be ported to a more modern platform. If the original target architecture were really widespread, then the manufacturer of that architecture would have provided a backward-compatibility solution of some kind. However, this still leaves a substantial number of legacy programs without the “critical mass” to make traditional porting approaches cost-effective. In fact, for many of these programs, even the cost of constructing a dedicated emulator in software might be too large when compared to the residual benefit provided by the legacy program.

A related concern when constructing a system emulator is that the hosting platform on which the emulator executes might eventually itself become obsolete. Hence one either needs good judgment and a bit of luck when picking a hosting platform for such an emulator, or in the long run the original hosting platform itself may need to be emulated on yet another machine, leading to a whole *stack* of cascaded emulators.

We aim to reduce the effort of creating emulators for legacy architectures, thereby giving a new lease of life to a wider range of existing legacy programs. Our approach is based on two strategies: First, the target of our emulation is not an actual hardware architecture, but the portable Java platform that is available on almost every conceivable system. We thereby greatly reduce the risk that any emulator constructed using our method would ever become obsolete, because Java has “critical mass”. Second, our emulator is constructed using a technique of generative programming, which greatly reduces the effort required for its construction, and simultaneously also the scope for errors.

In the following, we give an introduction to virtual machines in general and to our approach in particular. We then describe our application of generative programming for deriving an interpreter and code generator from a machine specification. After a section discussing related work, our paper ends with conclusions and an outlook to future work.

2 Virtual Machines

A software program that emulates the behavior of a specific hardware platform is also called a virtual machine (VM). There has been considerable interest in virtual machines recently, driven by the popularity of mobile-code approaches such as Java. Virtual machines emulate one platform on top of another by *interpretation*, by *(just-in-time) compilation*, or by a combination of these two strategies. An interpreter executes individual virtual machine instructions one by one, performing the corresponding functionality directly. A (just-in-time) compiler takes a sequence of virtual machine instructions and translates them to the native instruction set of the runtime platform prior to execution.

Implementing such an interpreter or just-in-time compiler is a non-trivial task. Previous research has studied the use of domain-specific languages to specify code generators, as well as the automatic generation of interpreters. Most of these solutions are complex because they need to deal simultaneously with both varying emulated architectures and varying actual execution platforms. As a result, none of these previous approaches has gained widespread popularity for the purpose of supporting legacy code.

Our approach to machine emulation uses the Java Virtual Machine (JVM) as the **target platform**. Since Java is available almost universally, this eliminates the variability problem on the hosting side: if we can build an interpreter and/or a compiler that can bridge the gap between the architecture to be emulated and the JVM, then users can deploy their legacy software on *any* platform on which the JVM is available. To base a machine emulator on the JVM, we need to build a compiler that “compiles up”, from a low-level machine language into Java’s higher-level bytecode language. It is precisely this problem that we have solved in a prototype system that we call *Virtual Execution Environment for Legacy Software (VEELS)*.

Our current prototype system executes standard Linux PowerPC, ARM, and MIPS binaries on top of the Java Virtual Machine by way of “compiling up”. It thereby makes it possible to run all three kinds of binaries on **any** platform that a JVM is available on. VEELS uses a mixed-mode execution strategy that combines an interpreter with a just-in-time

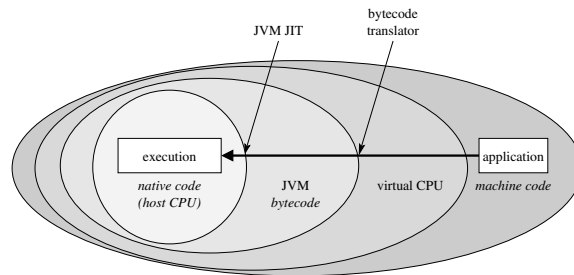


Figure 1: Life-cycle of emulated native machine code on top of a Java Virtual Machine (JVM).

compiler. Initially, the legacy machine code is interpreted by a simple CPU emulator that is implemented in Java. Whenever a “hot spot” with high execution frequency is detected, a just-in-time compiler is used to translate blocks of instructions into Java bytecode. If our “target” JVM is in turn hosted using another just-in-time compiler (from JVM bytecode to the actual native code of the underlying hardware platform), then the emulated code will eventually be compiled (via the intermediate step of JVM bytecode) to native code.

Using dynamic optimization techniques similar to those found in binary translation frameworks, we were able to reduce the performance penalty of our scheme to a surprisingly low level. Even in our worst-case benchmarks and with an *unoptimized* prototype, the slowdown of emulating PowerPC code on top of a Java VM running on a PowerPC processor (double emulation) never exceeded a factor of 70. When emulating on a modern Intel host, the slowdown was almost zero. Considering that most platform emulations will concern much older, slower platforms than the still current PowerPC, our results imply that it is perfectly reasonable to “compile up” native code so that it runs on top of a synthetic, high-level instruction set architecture (ISA) such as the Java VM. The high-level ISA in turn is then likely to be dynamically “compiled down” again into some low-level native code, giving hardware architects much more leeway as they are no longer bound by any constraints of backward compatibility (Figure 1). Even more important than pure performance, the effort required to construct such an emulator is sufficiently low to be attractive even for “niche” legacy architectures.

3 Building A Translator

Instead of manually and separately implementing the CPU emulator and code generator for each emulated platform, we use generative programming to dynamically assemble specialized interpreters and code generators at runtime. Using the same initial specification, we can dynamically instantiate an interpreter, a simple basic-block-based compiler, and a superblock-based optimizing compiler.

In order to interpret foreign code on a host machine, one can execute it instruction-by-instruction using a virtual CPU that emulates the instruction set of the guest system. This

```

void addi(int insn) {
    int D = (insn >> 21) & 0x1f;
    int A = (insn >> 16) & 0x1f;
    int imm = ((short)(insn & 0xffff));
    reg[D] = reg[A] + imm;
}

void decode(int insn) {
    switch((insn >>> 26) & 0x3f) {
        ...
        case 14: addi(insn); break;
        ...
    }
}

```

Figure 2: The hosted machine code instructions are first decoded (right hand side), then individual functions (here `addi` for the PowerPC architecture) extract the operands and emulate the instruction.

approach is not particularly efficient because in addition to the effort of actually executing the operation, each instruction also has to be decoded first. As the example in Figure 2 shows, decoding an instruction can easily be more costly than actually executing it. This approach also does not yield ideal results when dealing with irregularities in instruction sets. As an example, the `addi` instruction of the PowerPC architecture (Figure 2) has a special form that can be optimized when R_0 is used as source register, because R_0 always returns 0 on PowerPC CPUs. While this would allow to write the value of the immediate directly to the destination register, during instruction-by-instruction interpretation, any reduction in execution time resulting from this optimization is cancelled out by the increased decoding effort required to actually detect the special form.

To overcome this problem, many existing machine emulators translate blocks of foreign machine instructions directly into code executable by the host machine (“just-in-time” compilation). By doing so, the decoding effort is incurred only once and can be recuperated over time if the translated block is executed repeatedly. Block translation is also able to perform optimizations such as generating specialized code for specific instruction forms. The additional decoding effort is incurred only once, while the execution time benefit applies to each subsequent execution of the translated code block. On the other hand, block translation can decrease performance if the translated block is not executed frequently enough. To overcome this drawback, one uses “mixed-mode” execution that combines interpretation and compilation. However, constructing a mixed-mode virtual execution environment requires implementing a machine emulator at least twice: one that acts as an interpreter and one that acts as a code generator.

3.1 Using Generative Programming to Obtain a Mixed-Mode Environment

Implementing the machine emulator twice is redundant and complicated. To make matters worse, this needs to be done in two different languages. While the *interpreter* can use the Java source language to implement the semantics of the emulated machine, the *code generator* needs to emit Java bytecode instructions. To avoid this duplication of work, we use generative programming [BBCE02, EC00] to derive an interpreter and a code generator for the emulated machine from the same specification. **The solution involves using the standard `javac` compiler to translate parts of the machine emulator into the JVM bytecode language, and then extracting the bytecode that was generated.** To this end, we introduce a notion of *code templates*. A code template is a piece of Java code that performs the actions required by the semantics of a foreign machine code instruction. For

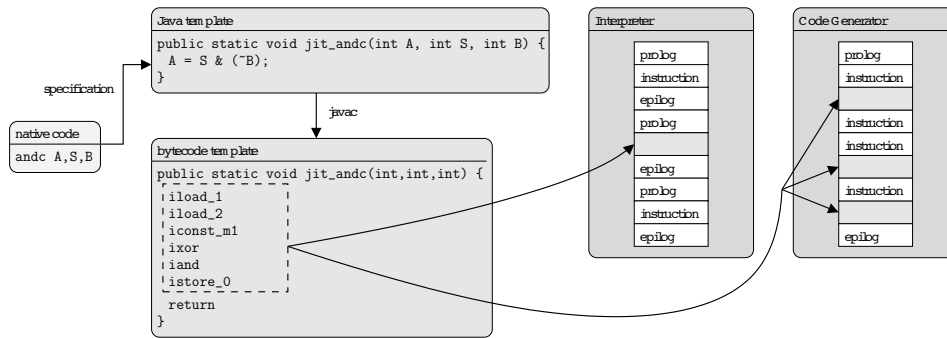


Figure 3: Overview of our approach. For each native instruction, a Java function is programmed that emulates the behavior. Using `javac`, this function is translated into a decode template, which is then used by both the interpreter and the code generator.

```

class Interpreter extends Decoder {
    ...
    void emit(String signature, String template, int[] context) {
        Instruction i = cache.get(template, signature);
        if (i == null) {
            i = compile(template, signature, context.length);
            cache.put(template, signature, i);
        }
        i.execute(context);
    }
    ...
    void interpreter_loop() {
        while (true) decode(Memory.read_word(reg.PC));
    }
}
  
```

Figure 4: Structure of the interpreter implementation. The interpreter loop decodes the next instruction using the `decode`, which uses the `emit` method to select appropriate code patterns.

each such instruction, we implement a decoder function that extracts the operands and selects one or more code templates to perform the necessary activities that together emulate the machine instruction.

To be generic enough to serve for both, building an interpreter and building a code generator for a block translator, code templates operate on JVM local variables. To turn a code template into a suitable form for the interpreter, it has to be surrounded by appropriate prologue and epilogue code to load the contents of the specified registers plus any immediate values (in the prologue) and to commit back any changes to the register file (in the epilogue). Deriving a pattern for the code generator works in a similar fashion, but the prologue and epilogue code are located at the beginning and the end of the translated block only. This allows for a more efficient execution, as register contents are exchanged between instructions in the same block via local variables without having to update the register file after each individual instruction.

The interpreter and the block translator apply different levels of specialization to code templates. The interpreter dynamically generates one Java method for each instruction of the

```

machine code:
    addi r1, r0, 15
    addi r2, r1, 30

interpreter:
    tmp1_addi.execute(int[] context)
        10 = reg[context[1]]; // prologue: 0
        11 = context[2]; // prologue: 15
        12 = 10 + 11; // tmp1_addi
        reg[context[0]] = 12; // epilogue: 1
    tmp1_addi.execute(int[] context)
        10 = reg[context[1]]; // prologue: 1
        11 = context[2]; // prologue: 30
        12 = 10 + 11; // tmp1_addi
        reg[context[0]] = 12; // epilogue: 2

block-execution:
    block.execute()
        10 = 15; // prologue: imm 15 in l1
        11 = 30; // prologue: imm 30 in l2
        // prologue: R1 in l2 (not read)
        12 = 10; // rewritten tmp1_addi_0
        13 = 12 + 11; // rewritten tmp1_addi
        reg[1] = 12; // epilogue
        reg[2] = 13; // epilogue

```

Figure 5: Comparison of the Java bytecode (shown as pseudocode) executed for two subsequent PowerPC `addi` instructions for the interpreter and the block translator.

emulated machine. This method receives *context* information containing intermediate values and register index numbers from the decoder, reads the appropriate registers, executes the code template and then commits changes back to the register file before returning to the interpreter loop. The basic structure of the interpreter is shown in Figure 4. The interpreter loop reads the next instruction from the program counter location and decodes it using the `decode` method, which is shared between interpreter and block translator. The `decode` method selects the appropriate code patterns using the `emit` method. In case of the interpreter, `emit` directly executes the requested pattern.

3.2 Block Compilation By Piggybacking On `javac`

The block compiler, in contrast, uses the code templates to generate specialized code that reads and writes from the specific registers encoded in the instruction at the current program counter location. This allows the block-compiled code to execute much faster because the *context* information is directly encoded in the resulting Java bytecode.

The specialization of code templates is done using bytecode rewriting [RW02]. Each code template expects its arguments in local variables. The block translator assigns unique local variable numbers to registers and immediate values and ensures through rewriting that specialized templates access the appropriate registers. The differences in the generated and executed bytecode for interpreter and block-translator are shown in Figure 5.

Having found a way to derive interpreter and block-compiler from the same specification we still have to find a way to actually specify the code templates. The most obvious approach would be to specify Java bytecode instructions for each code template. While this is trivial for simple instructions such as `addi`, it gets rather difficult and error-prone for more complex instructions such as `cntlzw` (Figure 6). **The key insight of our approach is that we do not need to perform this translation manually, but that we can “piggy-back” on the existing `javac` compiler.** For this, we use Java methods to code machine instructions as shown in the lower part of Figure 6. Incoming and outgoing register val-

```

tmpl_addi: 0=D 1=A 2=imm
0: iload_1
1: iload_2
2: iadd
3: istore_0

tmpl_cntlzw: 0=D 1=A
0: iload_1
1: istore_2
2: bipush 32
4: istore_3
5: bipush 32
7: istore_3
8: goto 18
11: iload_2
12: iconst_1
13: iushr
14: istore_2
15: iinc 3, -1
18: iload_2
19: ifne 11
22: iload_3
23: istore_0

public static void
    addi(int D, int A, int imm) {
    D = A + imm;
}

public static void
    cntlzw(int A, int S) {
    int w = S; int a = 32;
    for (a = 32; w != 0; --a) w >>= 1;
    A = a;
}

```

Figure 6: Java bytecode (upper part) and source code templates (lower part) for the `addi` and `cntlzw` instructions. Using `javac`, these templates are translated to class files, which are parsed at runtime to extract a code template from each method in the specification class.

ues and immediates are modeled as arguments and will be allocated to local variables by `javac`, starting at index 0. Later on, these references are rewritten to access the actual locations at which register values and immediates are stored in the generated code. The code rewriting mechanism also handles temporary values (w , a), and remaps branch instructions as needed. When the emulator starts, the generated class file is parsed to extract a code template from each method in the specification class. The parsed templates are then used for code generation.

It should be noted that we rely on `javac` to always generate code for writes to arguments, even if they are not used again in the code. The statement $D = A + imm$ in the `addi` template, for example, generates an `istore_0` instruction (Figure 6). Executing

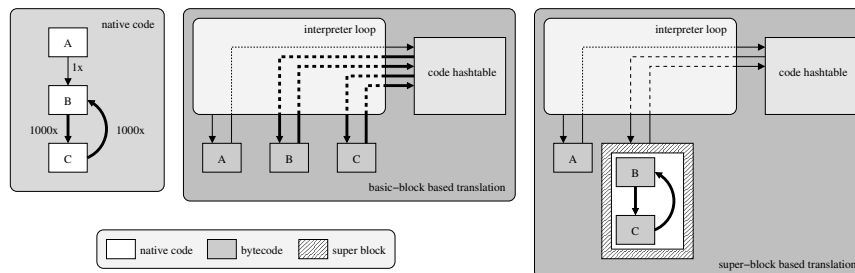


Figure 7: Comparison of basic-block-based and superblock-based translation of a loop in native machine code to Java bytecode.

the `addi` method, the value written to local variable 0 would be dead immediately after the `istore_0` instruction because of Java’s copy semantics for method arguments. Through bytecode rewriting, however, the `istore_0` instruction is remapped to a meaningful local variable index, which is then used in the epilogue to update the content of the associated register. Future versions of `javac` might detect and eliminate such redundant writes, thereby breaking our current code template mechanism. To compensate, we would need to extend our code rewriting component to use other language constructs such as `getstatic` and `putstatic` as placeholders to access register values and immediates.

3.3 Superblock Translation

Besides interpreting foreign machine code instruction-by-instruction, our system can also compile entire blocks of machine code into Java bytecode. In its simplest form this is done at basic block granularity. A hash table is used to map program counter (PC) locations in the native program to Java methods containing the equivalent “compiled up” bytecode. Whenever the emulator encounters a PC location for which no compiled Java method exists yet, it starts compiling a new block from that location until the next branch instruction. Every time a basic block completes execution, it returns to the interpreter loop and returns the updated program counter location. The interpreter loop then uses the hash table to locate the next code block to be executed (Figure 7).

Having to consult a hash table at every basic block boundary is expensive and takes up a significant fraction of the execution time. Additionally, having to exit and enter translated code blocks frequently is also costly because the CPU state has to be read into local variables at method entry and then committed back when exiting. Last but not least, going through a hash table also inhibits the just-in-time compiler of the host system from performing global optimization of the code. As far as the underlying just-in-time compiler is concerned, basic blocks are disconnected code fragments and aggressive optimizations such as loop hoisting or partial redundancy elimination cannot be applied to them. To overcome these limitations we group “hot” basic blocks and translate them to a single “*superblock*”. Ideally, such superblocks are entered and exited infrequently and most basic block transitions happen inside of the same superblock (Figure 7). If both caller and callee are located within the same superblock, the cost for the transition is reduced to the cost of executing a simple Java `goto` instruction. Even more importantly, by using actual Java control-flow constructs, loops in the native machine code are translated to loops in the generated bytecode, enabling the underlying just-in-time compiler to detect and optimize them.

3.4 Trace-based Block Selection

Our dynamic translator uses a technique similar to Dynamo [BDB99] and records a *program trace* when it detects a loop header. The path between the loop header and the corre-

<pre> for (int i = 0; i < 100000000; ++i) if ((random() & 1) == 1) a(); else b(); </pre>	<pre> recorded trace: loop: guard(i < 100000000); guard((random() & 1) == 1); a(); /* inlined code from a() */ ++i; goto loop; </pre>
---	--

Figure 8: A simple loop that randomly executes `a()` or `b()`. The recorded trace contains only one of the two possible alternatives, and as a result has to be exited frequently during execution.

sponding backward branch is then translated into a superblock. Since this enables the full loop to be executed locally within the superblock, performance is increased dramatically for simple example code.

For real-world code, however, the speedup is much smaller than expected. This is caused by the inability of our system to “grow” traces once they have been recorded. Dynamo, in contrast, grows existing traces when a guard condition fails, following the rationale that all paths leading out of a hot region (loop) are likely to be hot as well. For this, Dynamo patches the existing trace in place to target the newly recorded trace every time the associated guard instruction fails. Growing the trace cache in such a manner can deal with loops that do not consist of a single hot path but which in fact have several loop-internal hot paths (Figure 8). In contrast to Dynamo, we are emitting traces as Java bytecode and not as native machine code. We are therefore unable to grow traces by patching them in situ. Thus, whenever we encounter a non-trivial loop, we are able to execute only one path through the loop efficiently—all other paths cause the superblock to be exited.

Popularity-based block selection. To deal with the multiple-path problem described above, we are investigating block selection based on popularity (“hotness”). For this, the code is executed for a while at the basic-block level and all basic blocks reached are recorded. When a certain threshold is exceeded, this information is used to select basic blocks for a superblock.

By no longer recording a single trace, this approach should be able to improve upon the shortcomings of the trace-based basic-block collector as far as multiple paths are concerned. However, execution of the resulting superblock is likely to be somewhat slower for the path that was previously recorded as a trace, because direct branches can no longer be optimized away. With respect to the severe performance penalty associated with the multiple-path problem, however, this slowdown is probably negligible.

3.5 Emulation Environment

Executing legacy software requires more than emulating a CPU and a main memory. Instead, an appropriate legacy execution environment has to be emulated as well. Our prototype system focuses on providing Linux support and provides two emulation modes: user-space emulation and system-level emulation.

The *user-space* emulation mode is sufficient for executing most legacy Linux applications.

It does so by creating a flat process address space into which the executable is loaded. The emulation system intercepts Linux system calls and emulates them in the Java context, i.e., the operating system itself is emulated as well [Fra93]. It is sufficient to support a small simplified subset of the over 200 system calls implemented by an actual Linux kernel. Complex system calls such as *mmap*, which changes the process memory mapping, remain unimplemented and return an error code when invoked, as the standard Linux C library contains fall-back code that can emulate proper behavior even without such complex system functions.

The *system-level* emulation mode emulates the hardware features of a PowerPC architecture to a sufficient degree that a standard Linux kernel can be booted. The Linux kernel running on top then uses emulated hardware devices for I/O.

Both emulation modes have advantages and disadvantages. The user-space emulation has a performance benefit. It also integrates much better with the surrounding software environment since file I/O is handled by the host system. The system-level emulator, on the other hand, handles its own file system through emulated block devices. These emulated block devices become opaque files for the host system. An advantage of the system-level approach is that it can faithfully replicate the complete original execution environment, offering a higher degree of backward compatibility for legacy software, but at the expense of convenience and speed.

4 Related Work

There are three main bodies of related work: code-generator generators and interpreter generators each are concerned with synthesizing similar components as those generated by our system. The third area is the co-generation of both code generators (or, more generally, compilers) and abstract machines. The aspect of executing the generated program is related to binary translation.

The most prominent example for a code generator generator is Burg [FHP92], extending earlier work [AGT89, App87]. Just like BEG [ESL89] and Twig [AGT89], Burg works on tree grammars and generates a tree parser, which in turn make two passes over the tree for code generation. The main difference between these approaches and our work is that they strive to select optimal code sequences for nodes in the IR tree. In contrast, we use bytecode templates that might be globally sub-optimal and rely on the just-in-time compiler of the underlying virtual machine to optimize the program.

In the area of pure interpreter generators, VMgen [EGKP02] is the only working approach that we know of. In principle, our system provides a functionality similar to that of VMgen. However, we automatically generate the description of instructions of the interpreter from the bytecode files output by `javac`. The rest of the generated interpreter is static in that it does not change if the input language changes.

Finally, there has been work in the area of semantics-directed generation of compilers and abstract machines (e.g. [Die96]). The main concern in this area is to prove correctness of compilers and abstract machines generated from a description of the semantics of a

programming language. In contrast to this, we use a general-purpose high-level language (Java) to specify the operational semantics of VM instructions. By using `javac` for the generation of code templates for both the code generator and the virtual machine, we implicitly obtain a correctness proof “for free” with respect to the semantics of Java.

With focus on the execution of our generated system, our approach is closely related to binary translation [Pro01] systems such as VirtualPC [Mic], but in contrast to existing systems we do not need to implement a specific compiler for each host system. Once we add the necessary machine description for the guest system, we can emulate that architecture on *any* machine capable of running JVM. At the same time we are able to benefit from the extensive optimizations performed by just-in-time compilers found in modern JVMs.

Like our approach, QEMU [Bel] emulates different CPUs and has support for user level emulation as well as full system emulation. It uses native code pattern-based dynamic compilation [PR98], which motivated the bytecode template mechanism we use in VEELS. QEMU, however, only supports block-based translation and does not attempt to generate an interpreter from the same pattern-based machine specification.

5 Summary and Outlook

We have presented a novel approach to dynamically assemble interpreters and code generators from a common specification. Our simple yet elegant approach allows to specify a machine description as a series of Java source code snippets, which in turn are compiled to Java bytecode and used to generate an interpreter and various code generators. Using a general-purpose high-level language to specify the operational semantics of virtual machine instructions gives us maximum flexibility and expressibility for the implementation of each instruction template, and we can use the standard Java bytecode compiler to turn the source code snippets into bytecode templates.

Operating on the bytecode-level instead of generating Java source code permits to dynamically generate specialized interpreters and code generators at runtime. This allows to instantiate application-specific interpreters and code generators. In our prototype, we generate a minimal interpreter that covers only machine instructions that are actually used by the application. For fairly complex and irregular architectures such as PowerPC, this results in a substantially smaller and more efficient implementation.

VEELS is an ongoing research effort. Already today, we can provide emulation of older architectures that should compare well with the peak performance available on the historic hardware. For example, when comparing an emulated PowerPC hosted on a modern Intel processor with actual PowerPC hardware that is only five years old, the performance is similar. Our near-term focus is to improve performance by further developing the popularity-based superblock translator.

We are also currently exploiting how to apply our approach to Microsoft’s .NET infrastructure [MG]. Initial experiments have shown that Microsoft’s C# compiler does not optimize write operations to local variables either, which will allow us to use the same template mechanism as we do for the Java Virtual Machine.

References

- [AGT89] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.
- [App87] A. Appel. Concise Specification of Locally Optimal Code Generators. Technical Report CS-TR-080-87, Princeton University, 1987.
- [BBCE02] B. Barth, G. Butler, K. Czarnecki, and U. Eisenecker. Generative Programming. *Object-Oriented Technology: ECOOP 2001 Workshop Reader. Lecture Notes in Computer Science*, 2323:135–149, 2002.
- [BDB99] V. Bala, E. Duesterwald, and S. Banerjia. *Transparent Dynamic Optimization: The Design and Implementation of Dynamo*. Technical Report HPL-1999-78, Hewlett Packard Laboratories, June 1999.
- [Bel] F. Bellard. Qemu: Generic and Open Source Processor Emulator. <http://fabrice.bellard.free.fr/qemu>. Last visited on 6th February 2007.
- [Die96] S. Diehl. *Semantics-directed Generation of Compilers and Abstract Machines*. PhD thesis, Universität Saarbrücken, 1996.
- [EC00] U. W. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [EGKP02] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen: A Generator Of Efficient Virtual Machine Interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.
- [ESL89] H. Emmelmann, F.-W. Schröer, and L. Landwehr. BEG: A Generator for Efficient Back Ends. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 227–237, Portland, Oregon, June 1989.
- [FHP92] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
- [Fra93] M. Franz. Emulating an Operating System on Top of Another. *Software—Practice and Experience*, 23(6):677–692, June 1993.
- [MG] E. Meijer and J. Gough. Technical Overview of the Common Language Runtime. <http://research.microsoft.com/~emeijer/papers/CLR.pdf>. Last visited on 6th February 2007.
- [Mic] Microsoft Corporation. Virtual PC. <http://www.microsoft.com/virtualpc>. Last visited on 6th February 2007.
- [PR98] I. Piumarta and F. Riccardi. Optimizing Direct Threaded Code by Selective Inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 291–300, Montreal, Quebec, Canada, June 1998.
- [Pro01] M. Probst. Fast Machine-Adaptable Dynamic Binary Translation. In *Proceedings of the 3rd Workshop on Binary Translation*, Barcelona, Spain, September 2001.
- [RW02] A. Rudys and D. S. Wallach. Enforcing Java Run-Time Properties Using Bytecode Rewriting. In *International Symposium on Software Security*, Tokyo, Japan, November 2002.