Summary: Retargeting JIT compilers by using C-compiler generated executable code

Mark Tokutomi
January 27, 2011

This paper presents a novel approach to creating a Just-in-Time compiler for a language which has an interpreter whose source code is available. The idea is to leave most of the code generation up to a modified version of the interpreter, but to "patch" in necessary information (such as addresses and arguments to functions) when the code is dynamically-generated. This approach has advantages over each of the alternative methods currently in use, although it still must make tradeoffs in the same way.

The three approaches to programming language implementation discussed by the authors are native-code compilers, interpreters, and source-to-source compilation (in this paper the reasonable assumption is made that the target language is C). Though each of these approaches is useful in specific situations, each also has significant weaknesses relative to the others. By executing in a language which is presumably spread across a wide variety of platforms, both interpreters and source-to-source compilers can offer a high level of portability between systems and architectures. In the case of source-to-source compilation, all that is required is a compiler in the target language, and for an interpreter, either the source code or an executable must be provided. On the other hand, a native-code compiler is specific to the architecture for which it was written; additionally, even if ported to another architecture, fully optimizing the code it generates may require very thorough familiarity with the architecture to which it is being ported. In exchange for portability, a native-code compiler gains the notable benefits of speed in both execution and compilation. An unavoidable feature of an interpreter is slow execution: the fact that it reads the program's source as it is executing makes optimization much more difficult than in a traditional compiler, and makes some types of optimization entirely impossible. However, a corollary to this drawback of interpreters is the benefit that no time is spent compiling the code. Depending on the development environment and expected usage of a piece of software, either approach can be advantageous. Finally, a source-to-source compiler has the capacity to generate code that executes at least as fast as a native-code compiler. Since the target language is usually very widespread, highly developed compilers with strong

optimizing capabilities can be used, and additionally if the target language is low-level enough, the first-stage compiler can optimize the code before it is even passed to the target language's compiler.

By using a (modified) interpreter to generate the code used by the Virtual Machine, the authors' approach immediately gains much of the portability enjoyed by interpreted languages. Even in cases where the software hasn't been ported to the target architecture, the interpreter can be used instead, giving this approach an excellent fall-back option in terms of compatibility. Although modifying the interpreter requires some knowledge of the target architecture, the task is of much lesser magnitude than writing a native-code compiler. By using code compiled from C as the input to the Virtual Machine, the JIT approach also gains the benefit of the optimization capabilities of GCC, leading to faster execution time (although still likely slower than the code generated by a source-to-source or native-code compiler, since optimizing during JIT compilation is much more difficult than optimizing with a full compiler). Finally, due to the straightforward nature of its compilation, this approach also offers faster compile time than any other approach besides an interpreter. A final benefit to the authors' approach is the short implementation time: the authors claim that implementing their approach on the first architecture took two person-weeks, and that it was ported to a second in under a person-day.

The basic idea behind this compilation technique is to generate each Virtual Machine instruction using the source language's interpreter, and then at run-time to dynamically generate input to the Virtual Machine by combining these compiled instructions. This approach is improved through the use of static and dynamic superinstructions. Static superinstructions have the benefit of decreasing the resulting code's size; additionally, because they execute entirely as one instruction rather than linking multiple calls together, they can be rewritten to use fewer memory accesses and stack pointer updates. Dynamic superinstructions are not built-in like static superinstructions. Rather, they are generated simply by concatenating the architecture's regular instructions. This approach does not allow for the same degree of optimization as adding static superinstructions, since memory accesses can no longer be optimized, but it does greatly reduce the number of necessary dispatch calls. To further reduce the number of necessary dispatch calls, the authors place the code fragments into a monolithic C function. This enables the Virtual Machine to jump between instructions directly, rather than storing an instruction pointer to indirectly jump between code fragments. To

prevent the compiler from reorganizing the code in a way that would break this approach, each instruction is followed in the function by an indirect jump. Since the compiler does not have sufficient information to determine at compile time in what order the instructions will ultimately execute, it is forced to generate code that allows the instructions within the function to execute in any order.

An interesting side effect of the reduction in the number of dispatches that this approach achieves through the use of superinstructions is a greatly reduced use of the Virtual Machine's instruction pointer. The instruction pointer is still used to pass immediate arguments to instruction calls, as well as to continue execution after a Virtual Machine branch. However, both of these uses can be rendered unnecessary by dynamically patching the arguments into the code fragments corresponding to the instruction at execution. This patching technique is what allows the authors' approach to execute so much faster than a normal interpreter. It allows the JIT compiler to avoid threaded dispatch, accessing the interpreted code (to retrieve arguments and control pointers), and the use of the instruction pointer (which additionally frees up a register for other potential uses). Additionally, since it makes such extensive use of code generated by the interpreter, it is also what gives this approach greater portability than a native-code compiler.

Most of the implementation details presented in the paper simply deal with placing "markers" in the generated code in order to identify the proper location for arguments to instructions, as well as to determine what encodings are used by the architecture. For example, generating two code fragments which are identical save for the value of a constant being passed to the function allows the developers to identify the constant's location in the resulting code, which enables them to see where the code needs to be patched at compile time in order to modify the arguments to the instruction.

A more interesting (or at least less mechanical) implementation issue discussed in the paper, however, is that of dealing with non-relocatable code. While the approach of patching necessary arguments in at compile time and calling instructions from within a single function works in most cases, there are situations (all architecture-specific in the paper) in which a piece of code cannot be executed in this way (an example given in the paper is a program-counter-relative reference to code not contained in the given fragment). In cases where the code must be executed from its context within the compiler code, the authors use the indirect jumps between each code fragment to act as a substitute instruction pointer: the address of the next instruction is loaded

into memory, and then jumped to. The authors determine whether a given code fragment is relocatable by generating two versions of the function containing the code fragments. The first version is unmodified; the second uses the instruction *asm(".skip 16");* to pad between the fragments. This makes jump instructions within each fragment point to different relative locations (as compared to the original version), and observing what issues (if any) this modification creates allows the developers to determine which fragments may not be relocated during compilation.

A final detail that must be considered in order to implement this approach is the fact that the generated code is incapable of correctly implementing calls and returns at the Virtual-Machine-level. The generated code exists within the Virtual Machine and maintains its own stack pointer and register allocations; the authors point out that calling a function or using a return statement within the generated code will generally clobber both of these. Hence, the only portable way offered by the authors is to save the address to which we wish to return, and then use jump statements to enter and exit a Virtual Machine branch. They point out that more efficient implementations are possible, but such implementations would have to be machine-specific.

The results presented in the paper are an improvement over the existing solutions in every way one would expect; additionally, this approach stands up to native-code compilation better than (I at least) expected. The authors use a proof-of-concept implementation based on the Gforth interpreter (for the Forth programming language); they test it against two Gforth-based interpreters, and two native-code Forth compilers. They also test it against C code compiled with GCC; however, since most of the test suite is composed of programs written in Forth, the authors write some simple routines (matrix multiplication, a prime sieve, bubble sort, and a recursive fibonacci function) in order to compare their implementation to GCC's output. Predictably, the code generated by the authors' solution executes faster than either of the interpreters in every case, even taking into account compilation time. Surprisingly, however, only one of the native-code Forth compilers is on average faster; the other is actually slightly slower on average. Further, the authors' solution compiles faster than the Forth compiler it was tested against. Obviously, GCC's output executes significantly faster in every test (one of the test Forth compilers is nearly as fast at executing the recursive fibonacci function, but that's most likely the fault of C rather than GCC). However, the authors state that they tested their code against GCC mainly to determine how much faster the generated code could theoretically be (its output is something of

a practical upper bound for code executed with a Just-in-Time compiler). The tests compare the authors' initial implementation (for a processor with a 386 architecture) to both Forth compilers, both intepreters, and GCC; additionally, a version the authors ported to the PowerPC architecture is tested against both interpreters and GCC. GCC's advantage is considerably larger on the PowerPC, which the authors state can be mitigated by adding static stack caching to the PowerPC implementation.

The approach taken by the authors of this paper is an interesting one, and it certainly seems to yield observable benefits compared to an approach using just an interpreter or native-code compiler. Additionally, I can't fault their implementation, and though there are probably ways it could be improved, I'm unfamiliar with what those techniques might be. My critical opinions regarding this paper are almost entirely regarding the evaluation of their implementation, and whether the benefits it yields over source-to-source compilation and native-code compilers justify its weaknesses.

Though this implementation proved to be faster than one of the native-code compilers against which it was tested, it was slower than the other in nearly every benchmark. While the ease of implementation and shorter compile time would certainly be of great importance in some applications, it seems likely that in a situation where the software would see use over an extended period of time, and where execution speed was a higher priority than compilation time, a native-code compiler would still be of greater benefit. Additionally, both of these benefits would almost certainly be magnified further by the use of a source-to-source compiler rather than a native-code compiler. Although both the authors' implementation and the native-code compiler against which it was tested yield faster execution than an interpreter, the optimization capabilities of GCC, in addition to first-pass optimizations which could be implemented in the source-to-C translation, give such an approach a tremendous advantage in situations where execution speed is most important.

Additionally, though the solution presented here is almost certainly easier to implement than either a native-code or source-to-source compiler, the representation provided by the authors of the time taken to port the code to another architecture seems somewhat disingenuous. The port was performed by someone with detailed knowledge of the current architecture, the compiler itself, and the destination architecture; this situation seems unlikely to arise often in practice. Further, the port, though functional, runs substantially slower than the original implementation on some benchmarks due to

issues with the architecture which weren't accounted for when the compiler was ported. Finally, the benchmark comparison between the authors' solution and GCC seems somewhat tenuous. Without seeing the source code used to compare the two, it's impossible to know how well the algorithms used work with GCC's optimization capabilities. Particularly, a recursive fibonacci implementation is (to my knowledge) incapable of being efficiently implemented in C, and the prime number sieve could have been any of several implementations. While the comparison between the two wasn't a particularly pivotal point of the paper, such an opaque benchmarking system instills little confidence in the results, which would be very difficult to replicate.