Retargeting JIT compilers by using C-compiler generated executable code

Mark Tokutomi

January 27, 2011

Problem: Tradeoffs in Language Implementations

- **Portability**
- **Speed of Execution**
- **Speed of Compilation**

- **Native-Code Compilers**
  - Fast compilation, fast execution, poor portability
- **Interpreters**
  - Highly portable, no compilation time, poor execution speed
- **Source-to-Source Compilers**
  - Fast execution (assuming good compiler), very portable, large compilation overhead

Application domain for this solution

- **New language implementation**
  - This approach adds little additional work beyond writing an interpreter
- **Execution speed improvement for interpreted languages**
  - This approach displays dramatic execution time improvement without writing a full native-code compiler

Overview of authors' approach

- **Modify an existing interpreter written in C**
  - Restructure the interpreter's source code to be more amenable to the rest of this process
  - Work with compiled code for the modified interpreter
  - Write a native-code compiler which pieces together fragments of this compiled code
- **Authors' description of this approach:**
  - Can be though of as turning an interpreter into a JIT compiler
  - Can also be thought of as making a native-code compiler more portable
    - This approach leaves the interpreter as a fall-back option if the compiler hasn't been written for a particular environment

Benefits of this approach

- **Portability**
  - If necessary, can fall back on the interpreter for execution
  - Much more portable than partial evaluation (specializing an interpreter for a specific program)
    - Partial evaluation approaches are generally either source-to-source or platform-targeted
- **Implementation Effort**
  - Native-code compiler implementation is labor-intensive, and may lead to inconsistencies between platforms
    - In addition to being laborious to implement, must be carefully maintained
  - Authors claim their approach is much faster to implement
- **Compilation Speed**
  - The compiler functions by concatenating pieces of compiled interpreter code, so compilation is very fast

Modifications to the Interpreter

- **Direct Threading**
  - Keep addresses of function calls in instruction pointer, jump to next address at end of function execution
  - Improvement: Static Superinstructions
    - Combine common groups of instructions into a single call
    - Shortens code, and can potentially reduce number of memory accesses
  - Improvement: Dynamic Superinstructions
    - Concatenate code for instructions when compiling
    - Doesn't allow for as many optimizations as static, but still reduces dispatch calls

Modifications to the Interpreter (cont'd)

- **Can we remove the need for the Instruction Pointer?**
  - Normally used to access immediate arguments
    - During dynamic code generation, we can patch the argument directly into the code
  - Used to return from a VM branch
    - Patch in the target address directly
- **This gives faster execution than an interpreter**
  - No longer need to access interpreted code (all arguments and branch pointers are in the code itself)
  - Superinstructions avoid the load associated with threaded dispatch
  - Not using an Instruction Pointer avoids many register updates

Implementation Issues

- **Avoiding problems due to code fragmentation**
  - When modifying the interpreter, put all instruction fragments into one function
  - Add indirect jumps after each fragment, and after branches in fragments that will be patched with jump addresses
    - Prevents register allocation problems between fragments and ensures that they can be executed in any order
- **Non-Relocatable Code**
  - Can be caused by various details in a particular code fragment
  - Instead of calling the fragment out of context with the JIT compiler, call it in the C function
    - Use the indirect jump from the previous step to return to normal execution

Implementation Issues (cont'd)

- **Determining relocatability of code fragments**
  - Create two versions of function containing all the fragments
  - Pad between the fragments with an assembly instruction
    - Moves fragments relative to each other, and can then check whether any fail due to the relocation
- **Determining how to patch code fragments**
  - Duplicate each fragment
  - In the duplicate, change the fragment's constants
    - Highlights where the constants are in the code so they can be patched
    - A similar (but more involved) approach can be used to determine information about the encodings being used for constants

Implementation Issues (cont'd)

- **VM Calls and Returns**
  - Cannot use generated C code to perform a call/return at the VM level
    - The C code clobbers the stack pointer, and may overwrite registers
  - Instead of using actual function calls and returns in C, they must be emulated
    - Save the return address, jump to the location being called, then jump to the return address
  - This approach is less efficient, but is the only portable solution to this problem
    - Better-performing solutions would rely on machine-specific instructions

Results

- **The product presented in the paper is the authors' proof-of-concept implementation**
  - It is a native-code Forth compiler created for the Athlon and PowerPC architectures using the techniques outlined in the paper
- **Benchmarks are presented comparing this compiler to a variety of other implementations**
  - Compared this approach to two Gforth interpreters, two Forth native-code compilers, and GCC (in some of the applications)
  - GCC benchmarks were based on handwritten C code
    - Since the Forth programs were not available in C, the authors compared implementations of a prime sieve, matrix multiplication, bubble sort and a recursive fibonacci function to versions written in Forth.
  - Benchmarks for the Forth systems included compile time (for the compiled systems) to more directly compare them to the interpreted systems

Results

- **The product presented in the paper is the authors' proof-of-concept implementation**
  - It is a native-code Forth compiler created for the Athlon and PowerPC architectures using the techniques outlined in the paper
- **Benchmarks are presented comparing this compiler to a variety of other implementations**
  - Compared this approach to two Gforth interpreters, two Forth native-code compilers, and GCC (in some of the applications)
  - GCC benchmarks were based on handwritten C code
    - Since the Forth programs were not available in C, the authors compared implementations of a prime sieve, matrix multiplication, bubble sort and a recursive fibonacci function to versions written in Forth.
  - Benchmarks for the Forth systems included compile time (for the compiled systems) to more directly compare them to the interpreted systems

Results cont'd

- **Comparison to interpreted Forth systems**
  - As one would expect, the authors' native-code compiler outperforms the two interpreters (compilation time + execution time vs. execution time) on every test
    - The speed increases over the plain Gforth interpreter have a median factor of 2.7, while the increases over the interpreter using superinstructions have a median of 1.32 (on an Athlon processor)
    - On a PowerPC processor, the median speedup is 1.52 over the faster interpreter
- **Comparison to native-code compilers**
  - The handwritten native-code compilers fluctuate above and below the authors' implementation in performance
    - The (generally) better-performing compiler has a median speedup over the authors' of 1.19, and performs significantly better in some cases
    - The other compiler has a median speedup factor of .93, and outperforms the authors' compiler only in only two benchmarks

Results cont'd

- **Comparison to GCC**
  - On both the Athlon and PPC platforms, GCC outperforms the authors' implementation
    - The median speedup on the Athlon is 2.44, while on the PPC it is 4.9
    - One caveat about these timings is that the authors included compilation in their timings, but not in those for GCC
  - Despite the problems with this comparison, the authors treat it as an upper-bound
    - They also mention having improved the speed of their compiler on the PPC architecture since these tests

Opinions regarding ideas, techniques, etc

- This idea is an interesting approach, and the implementation seems to accomplish the authors' stated goals
- The techniques implemented seem reasonable
  - I didn't notice anything about the authors' implementation that I would argue with
  - It's possible that there are techniques the authors could have used to improve their approach that I'm unfamiliar with

Opinions (cont'd)

- **Benefits of this approach**
  - Some of the claimed benefits are clear, while others are more situation-specific
    - Given the choice between the two systems, it seems as though few circumstances would favor an interpreter
  - The development time for this solution is clearly shorter than for a native-code compiler
    - However, the faster native-code compiler is still faster in most applications
    - Depending on how long the product would be used, and in what situations, a native-code compiler might still be preferred
    - Additionally, developing either solution would naturally require a programmer with detailed knowledge of the architecture and language; the savings is in the development time

Opinions cont'd

- This solution is undisputably faster than a source-to-source compiler in terms of compilation speed
  - However, a source-to-source compiler is similarly easier to develop than a native-code compiler
  - Additionally, since it makes use of a compiler like GCC, a source-to-source compiler has the potential to generate very fast code (one would expect benchmarks similar to those produced by GCC)
  - In situations where more time is spent executing than compiling, a source-to-source compiler might still be a valuable alternative

Opinions cont'd

- **Time investment**
  - The authors present all figures regarding this as lines of code and man-hours
    - Lines of code are a questionable measurement of complexity in most circumstances
  - This implementation (as mentioned by the authors) required detailed knowledge of the Gforth interpreter
  - Additionally, it required great familiarity with each of the architectures used
    - Porting this solution to another architecture by different developers might take significantly more time
    - When the authors ported it to the PPC, the person coding was already very familiar with their implementation
    - Additionally, the authors likely already had an idea of what modifications would be necessary for the port

Opinions cont'd

- **Benchmarking**
  - The comparisons between various Forth systems should be generally accurate
  - The comparisons to GCC seem much less direct
    - The source code is not provided, so it's difficult to know whether it's written in a way that GCC can optimize well, and determining this would require very specialized knowledge of GCC
    - The prime sieve, for example, could be implemented in a variety of ways (the paper doesn't mention which sieve was used), and the Fibonacci implementation is recursive, so GCC's relatively weak performance on that benchmark is unsurprising
    - Although this entire point is perhaps overly critical, the general technique of comparing algorithms across languages seems brittle, and very difficult to replicate (especially without the source code)