Just-too-late compilation - An examination of a post-emptive compilation technique

Stephen Rogers

Sitting at home, Éire

Abstract

Just-in-time (JIT) compilation is a well-established method of deferring the compilation of code until it is needed at runtime, in an effort to the boost performance of interpreted systems. In this paper, we examine the concept of Just-too-late (JTL) compilation which compiles code for an application after it has already attempted to execute. JTL represents the state of the art in runtime performance in interpreted systems by offloading the compilation process to the host system too late, and ignoring any negative side-effects.

1 What's Going On?

Just-in-time (JIT) compilation is a well-established method of deferring the compilation of code until it is needed at runtime. It is particularly useful in interpreted environments, where significant performance gains can be made due to the use of native machine code, compared to a generally much slower interpreter. However, this comes at the cost of a "startup" or "warm-up" delay as any code must be compiled the first time it is to be executed. This means an application may take longer to execute the first time each JIT compiled code section is to be executed, as it has to wait for the code to be compiled.

Enter Just-too-late (JTL) compilation. As the name suggests, JTL involves compilation of code just after an application has attempted to execute it. When attempting to execute uncompiled code, the processor will typically (hopefully) encounter an invalid instruction encoding. When this happens, the processor will raise an interrupt with the host Operating System (OS) which must then handle the exception. Typically, this would involve halting the execution of the process and raising an error. However, with JTL compilation, the OS compiles the code which was attempting to execute and returns it to the process for it to continue execution.

Through this method of compilation, the "warm-up" delay technically no longer exists in your application, as it is the OS which is executing the compilation for you. Therefore, when measuring the performance of an application which uses JTL, there is a measurable improvement in runtime performance when compared with JIT. The elapsed real time may appear to be longer with JTL, but this is not relevant.

2 Explain It To Me

The first time a JTL section of code is reached for execution, it must be loaded in its entirety into memory. This region of memory should then be treated as a regular function and the host's calling convention shall be used to enter that region of memory. At this point, an invalid instruction encoding exception must be raised immediately at the start of the section so that it may be intercepted as a complete unit for the compilation process. This can be handled by inserting a known invalid instruction at the head of the section, but this comes at the cost of at least one single byte of memory. To avoid this overhead, we simply hope that the start of the uncompiled code happens to map to an invalid instruction encoding.

When an invalid instruction encoding is detected at runtime, the processor will raise an exception with the host OS. On X86, this is interrupt 6 "Invalid Opcode Exception". When this occurs, the CS (Code Segment) and EIP (Extended Instruction Pointer) registers will point to the segment of "code" which caused the exception. On Linux based systems, this interrupt will result in a SIGILL (Illegal Instruction Signal) for the process, which causes its termination.

In our modified system, rather than sending a SIGILL, the OS uses the CS and EIP registers to fetch the full uncompiled code from memory and pass it to the appropriate compiler. The compiled machine code is then taken from the compiler and overwrites the uncompiled code in memory. Control is then returned to the process to continue execution with the same CS and EIP values. This does require that the compiled code will fit into the same memory region as its uncompiled counterpart, which can be achieved using compilation techniques for code compaction. More importantly, this can be achieved with a wish and a prayer.

3 Unicorn Code

However unlikely, it is conceivable that there exists a program source code which coincidentally is encoded as valid machine code providing the exact functionality the source code represents. In this case, JTL provides the fastest possible execution path by bypassing the compilation process entirely. Such Unicorn Code represents the Holy Grail for performance in interpreted systems.

4 In The End, It Doesn't Even Matter

This paper presents the state of the art in runtime performance for interpreted systems by abusing the failure states of systems, piggybacking on a mechanism that is in no way designed for or suited to it. Bizarrely, or perhaps naively, I think this might actually possibly work.