In computing, just-in-time (JIT) compilation, also known as dynamic translation, is compilation done during execution of a program – at run time – rather than prior to execution.[1] Most often this consists of translation to machine code, which is then executed directly, but can also refer to translation to another format. A system implementing a JIT compiler typically continuously analyses the code being executed and identifies parts of the code where the speedup gained from compilation would outweigh the overhead of compiling that code.

JIT compilation is a combination of the two traditional approaches to translation to machine code – ahead-of-time compilation (AOT), and interpretation – and combines some advantages and drawbacks of both.[1] Roughly, JIT compilation combines the speed of compiled code with the flexibility of interpretation, with the overhead of an interpreter and the additional overhead of compiling (not just interpreting). JIT compilation is a form of dynamic compilation, and allows adaptive optimization such as dynamic recompilation – thus in theory JIT compilation can yield faster execution than static compilation. Interpretation and JIT compilation are particularly suited for dynamic programming languages, as the runtime system can handle late-bound data types and enforce security guarantees.

Applications[edit]

JIT compilation can be applied to some programs, or can be used for certain capacities, particularly dynamic capacities such as regular expressions. For example, a text editor may compile a regular expression provided at runtime to machine code to allow faster matching – this cannot be done ahead of time, as the pattern is only provided at runtime. Several modern runtime environments rely on JIT compilation for high-speed code execution, including most implementations of Java, together with Microsoft's .NET Framework. Similarly, many regular expression libraries feature JIT compilation of regular expressions, either to bytecode or to machine code.

A common implementation of JIT compilation is to first have AOT compilation to bytecode (virtual machine code), known as bytecode compilation, and then have JIT compilation to machine code (dynamic compilation), rather than interpretation of the bytecode. This improves the runtime performance compared to interpretation, at the cost of lag due to compilation. JIT compilers translate continuously, as with interpreters, but caching of compiled code minimizes lag on future execution of the same code during a given run. Since only part of the program is compiled, there is significantly less lag than if the entire program were compiled prior to execution.

Overview[edit]

In a bytecode-compiled system, source code is translated to an intermediate representation known as bytecode. Bytecode is not the machine code for any particular computer, and may be portable among computer architectures. The bytecode may then be interpreted by, or run on a virtual machine. The JIT compiler reads the bytecodes in many sections (or in full, rarely) and compiles them dynamically into machine language so the program can run faster. This can be done per-file, per-function or even on any arbitrary code fragment; the code can be compiled when it is about to be executed (hence the name "just-in-time"), and then cached and reused later without needing to be recompiled.

In contrast, a traditional interpreted virtual machine will simply interpret the bytecode, generally with much lower performance. Some interpreters even interpret source code, without the step of first compiling to bytecode, with even worse performance. Statically compiled code or native code is compiled prior to deployment. A dynamic compilation environment is one in which the compiler can be used during execution. For instance, most Common Lisp systems have a compile function which can compile new functions created during the run. This provides many of the advantages of JIT, but the programmer, rather than the runtime, is in control of what parts of the code are compiled. This can also compile dynamically generated code, which can, in many scenarios, provide substantial performance advantages over statically compiled code[citation needed], as well as over most JIT systems.

A common goal of using JIT techniques is to reach or surpass the performance of static compilation, while maintaining the advantages of bytecode interpretation: Much of the "heavy lifting" of parsing the original source code and performing basic optimization is often handled at compile time, prior to deployment: compilation from bytecode to machine code is much faster than compiling from source. The deployed bytecode is portable, unlike native code. Since the runtime has control over the compilation, like interpreted bytecode, it can run in a secure sandbox. Compilers from bytecode to machine code are easier to write, because the portable bytecode compiler has already done much of the work.

JIT code generally offers far better performance than interpreters.[citation needed] In addition, it can in some cases offer better performance than static compilation, as many optimizations are only feasible at run-time:

The compilation can be optimized to the targeted CPU and the operating system model where the application runs. For example, JIT can choose SSE2 vector CPU instructions when it detects that the CPU supports them. However, there is currently no mainstream JIT that implements this. To obtain this level of optimization specificity with a static compiler, one must either compile a binary for each intended platform/architecture, or else include multiple versions of portions of the code within a single binary.

The system is able to collect statistics about how the program is actually running in the environment it is in, and it can rearrange and recompile for optimum performance. However, some static compilers can also take profile information as input.

The system can do global code optimizations (e.g. inlining of library functions) without losing the advantages of dynamic linking and without the overheads inherent to static compilers and linkers. Specifically, when doing global inline substitutions, a static compilation process may need run-time checks and ensure that a virtual call would occur if the actual class of the object overrides the inlined method, and boundary condition checks on array accesses may need to be processed within loops. With just-in-time compilation in many cases this processing can be moved out of loops, often giving large increases of speed.

Although this is possible with statically compiled garbage collected languages, a bytecode system can more easily rearrange executed code for better cache utilization.

Startup delay and optimizations[edit]

JIT causes a slight delay to a noticeable delay in initial execution of an application, due to the time taken to load and compile the bytecode. Sometimes this delay is called "startup time delay". In general, the more optimization JIT performs, the better the code it will generate, but the initial delay will also increase. A JIT compiler therefore has to make a trade-off between the compilation time and the quality of the code it hopes to generate. However, it seems that much of the startup time is sometimes due to IO-bound operations rather than JIT compilation (for example, the rt.jar class data file for the Java Virtual Machine (JVM) is 40 MB and the JVM must seek a lot of data in this contextually huge file).[2]

One possible optimization, used by Sun's HotSpot Java Virtual Machine, is to combine interpretation and JIT compilation. The application code is initially interpreted, but the JVM monitors which sequences of bytecode are frequently executed and translates them to machine code for direct execution on the hardware. For bytecode which is executed only a few times, this saves the compilation time and reduces the initial latency; for frequently executed bytecode, JIT compilation is used to run at high speed, after an initial phase of slow interpretation. Additionally, since a program spends most time executing a minority of its code, the reduced compilation time is significant. Finally, during the initial code interpretation, execution statistics can be collected before compilation, which helps to perform better optimization.[3]

The correct tradeoff can vary due to circumstances. For example, Sun's Java Virtual Machine has two major modes—client and server. In client mode, minimal compilation and optimization is performed, to reduce startup time. In server mode, extensive compilation and optimization is performed, to maximize performance once the application is running by sacrificing startup time. Other Java just-in-time compilers have used a runtime measurement of the number of times a method has executed combined with the bytecode size of a method as a heuristic to decide when to compile.[4] Still another uses the number of times executed combined with the detection of loops.[5] In general, it is much harder to accurately predict which methods to optimize in short-running applications than in long-running ones.[6]

Native Image Generator (Ngen) by Microsoft is another approach at reducing the initial delay.[7] Ngen pre-compiles (or "pre-JITs") bytecode in a Common Intermediate Language image into machine native code. As a result, no runtime compilation is needed. .NET framework 2.0 shipped with Visual Studio 2005 runs Ngen on all of the Microsoft library DLLs right after the installation. Pre-jitting provides a way to improve the startup time. However, the quality of code it generates might not be as good as the one that is JITed, for the same reasons why code compiled statically, without profile-guided optimization, cannot be as good as JIT compiled code in the extreme case: the lack of profiling data to drive, for instance, inline caching.[8]

There also exist Java implementations that combine an AOT (ahead-of-time) compiler with either a JIT compiler (Excelsior JET) or interpreter (GNU Compiler for Java).

History[edit]

The earliest published JIT compiler is generally attributed to work on LISP by John McCarthy in 1960.[9] In his seminal paper Recursive functions of symbolic expressions and their computation by machine, Part I, he mentions functions that are translated during runtime, thereby sparing the need to save the compiler output to punch cards[10] (although this would be more accurately known as a "Compile and go system"). Another early example was by Ken Thompson, who in 1968 gave one of the first applications of regular expressions, here for pattern matching in the text editor QED.[11] For speed, Thompson implemented regular expression matching by JITing to IBM 7094 code on the Compatible Time-Sharing System.[9] An influential technique for deriving compiled code from interpretation was pioneered by Mitchell in 1970, which he implemented for the experimental language LC^2.[12][13]

Smalltalk (c. 1983) pioneered new aspects of JIT compilations. For example, translation to machine code was done on demand, and the result was cached for later use. When memory became scarce, the system would delete some of this code and regenerate it when it was needed again.[1][14] Sun's Self language improved these techniques extensively and was at one point the fastest Smalltalk system in the world; achieving up to half the speed of optimized C[15] but with a fully object-oriented language.

Self was abandoned by Sun, but the research went into the Java language. The term "Just-in-time compilation" was borrowed from the manufacturing term "Just in time" and popularized by Java, with James Gosling using the term from 1993.[16] Currently JITing is used by most implementations of the Java Virtual Machine, as HotSpot builds on, and extensively uses, this research base.

The HP project Dynamo[17] was an experimental JIT compiler where the 'bytecode' format and the machine code format were the same; the system turned PA-6000 machine code into PA-8000 machine code. Counterintuitively, this resulted in speed ups, in some cases of 30% since doing this permitted optimizations at the machine code level, for example, inlining code for better cache usage and optimizations of calls to dynamic libraries and many other run-time optimizations which conventional compilers are not able to attempt.[18][19]

Security[edit]

JIT compilation fundamentally uses executable data, and thus poses security challenges and possible exploits.

Implementation of JIT compilation consists of compiling source code or byte code to machine code and executing it. This is generally done directly in memory – the JIT compiler outputs the machine code directly into memory and immediately executes it, rather than outputting it to disk and then invoking the code as a separate program, as in usual ahead of time compilation. In modern architectures this runs into a problem due to executable space protection – arbitrary memory cannot be executed, as otherwise there is a potential security hole. Thus memory must be marked as executable; for security reasons this should be done after the code has been written to memory, and marked read-only, as writable/executable memory is a security hole (see W^X).[20] For instance Firefox's JIT compiler for Javascript introduced this protection in a release version with Firefox 46[21]

JIT spraying is a class of computer security exploits that use JIT compilation for heap spraying – the resulting memory is then executable, which allows an exploit if execution can be moved into the heap.