CS60203: Design Optimization of Computing Systems *August 17, 2024*

How to design a JIT Compiler?

Instructor: Mainack Mondal *Scribe-By: <Sourodeep Datta >*

1 Summary

These notes will describe the design process of JIT compilers, particularly with reference to V8 and the Cpython JIT. It will cover some of the design decisions made and problems faced, along with their solutions. It will also cover Copy-and-Patch, a fairly newer form of JIT compilation.

2 Refresher: Compilation Approaches

The current approaches we know of are "full compilation" and "linearization of a high-level language to bytecode followed by assembly into machine code".

2.1 Full Compilation

Full Compilation refers to the process of converting high-level source code entirely into instructions, usually in binary. This is done by a compiler, which performs a comprehensive transformation of the source code into object files.

One of its use cases is *metaprogramming*. Metaprogramming can be used to move computations from runtime to compile time to generate code using compile-time computations.

2.2 Compilation via Bytecode

"Linearization of HLL (High-Level Language) to bytecode" refers to the process of transforming a high-level programming language (HLL) code into a sequential, linear stream of instructions (bytecode) that a virtual machine (VM) can execute. This is followed by its assembly into machine code.

This approach is used by most databases and Python. The reasons for this include:

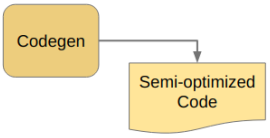
- Bytecode is platform-independent, meaning that it can be executed on any machine with the appropriate virtual machine. This makes it easier to deploy database systems and Python applications across different environments without needing to recompile the code for each platform.
- Bytecode is usually smaller than hardware instructions, and complex sets of operations can be made as a single bytecode instruction.
- Bytecode can be efficiently interpreted by a virtual machine. This is especially useful in environments where runtime flexibility is important, such as in databases where queries can be generated dynamically.

3 V8 JIT

V8 is a JavaScript and WebAssembly engine developed by Google for its Chrome browser.

3.1 V8: 2007

It consists of the "codegen" compiler, which takes the abstract syntax tree (AST) as input and emits the corresponding machine code. This code is semi-optimized, but the overall process of executing a program is slow.



3.2 V8: 2010

This version of V8 has two compilers: one that runs fast and produces generic code (Full-codegen), and one that doesn't run as fast but does try to produce optimized code (Crankshaft).

The first time V8 sees a function, it only runs the full-codegen compiler when the function is first run. Along with this, it starts a profiler thread to determine which functions are *hot*. This lets V8 record the type information flowing through it and uses it to augment the AST. So by the time it has decided that a function is hot, it has type information to give to the Crankshaft compiler [1].

Even after generating the optimized code, V8 still needs to retain a copy of the unoptimized machine code. This is due to the optimized code sometimes not being better than the unoptimized code. It is also due to the nature of JS, where attributes such as the types of objects can change during runtime, which can make the optimized code stop working. In this case, the unoptimized code will have to be executed [2].

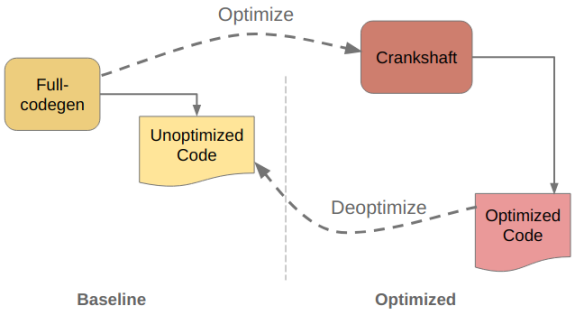
By using the "dumb" full-codegen, the code is generated faster, and is slower. As the code is generated faster, the startup time is less. This leads to a trade-off between start-up time and execution time.

3.2.1 Full-codegen

The quick-and-simple compiler is known internally as the "full-codegen" compiler. It takes as its input the abstract syntax tree (AST) of a function, walks over the nodes in the AST, and emits the machine code.

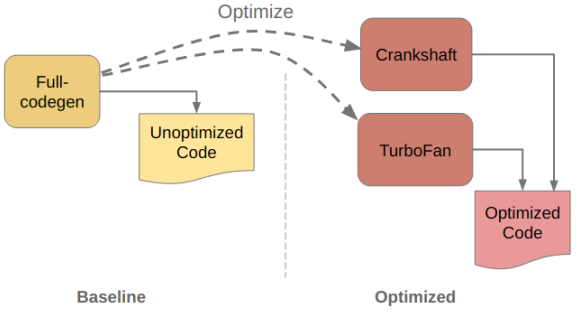
3.2.2 Crankshaft

Once V8 has identified that a function is hot and has collected some type feedback information, it tries to run the augmented AST through an optimizing compiler, which is Crankshaft.



3.3 V8: 2015

Similar to the 2010 version, but has 2 optimizing compilers, Crankshaft and Turbofan. Turbofan is slower than Crankshaft but generates more optimized code.



3.4 Problem: Optimizing Compiler Speed

One problem with optimizing compilers is that they are slow. This leads to a startup time, which mostly consists of the time taken by the optimizing compiler to compiler to code, causing *user facing latency*.

3.4.1 Optimizer Thread

One way to resolve this is to run the optimizing compiler on another thread. Therefore, while the optimizing compiler is running, the baseline code can be executed. Once the optimizing compiler finishes, the baseline code can be swapped out with the optimized code (which is a fast operation). Thus, most of the startup time is reduced, with the remaining portion being the time taken to check for hot path and sending to the optimizer thread.

3.4.2 Profiler Thread

The startup time can be further reduced by running a separate profiler thread which checks for hot path and sends the corresponding code to the optimizer thread. This is done via a *Dispatcher Queue*, where the *Code Object* is inserted. Thus, the only latency that contributes to *user facing latency* now will be the time to swap the baseline code with the optimized code. This is usually a pointer swap, so is very fast.

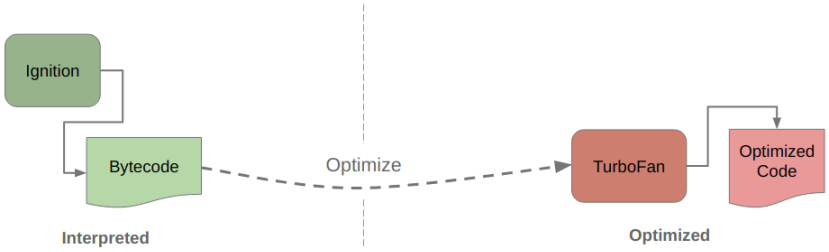
It should be noted that all of these latency optimizations come with a resource overhead, as the system will need to be running multiple threads parallelly when executing the code.

3.5 Problem: Memory Consumption of Machine Code

An issue with compiling directly to machine code is that it takes quite a significant amount of memory to store the code objects. V8 uses the baseline compiler to generate non-optimized machine code. This can consume a significant amount of memory, around 30% of the heap [2].

3.5.1 Bytecode Interpreter: Ignition

Instead of the compilers using the AST to compile to machine code, Ignition interprets the JS code into a concise bytecode, which takes up between 25% to 50% the size as the equivalent baseline machine code. TurboFan then uses this bytecode to generate optimized code.



4 CPython JIT: Copy-and-Patch

The copy-and-patch system consists of two components: the MetaVar compiler and the copy-and-patch code generator. The key to the copy-and-patch algorithm is the concept of a binary stencil, which is a partial binary implementation of a bytecode instruction or an AST node of a high-level language. The MetaVar compiler generates many binary stencils that implement different optimization cases for every bytecode or AST node. The MetaVar compiler takes as input bytecode/AST stencil generators and produces a library of binary stencils at library installation time. The stencil library becomes an input to the copy-and-patch code generator, together with a bytecode sequence or an AST that implements a function. The code generator then produces binary code that implements the function by copying and patching together stencils that implement the bytecodes or AST nodes. The patching step rewrites pre-determined places in the binary code, which are operands of machine instructions [3].

With copy-patch, you can write stencils in C. These stencils are functions with holes, and can be compiled with a typical C compiler. Then, when you want to compile something, you stitch stencils together, fill in the gaps, and jump straight into your brand new "compiled" function [4].

CPython has an experiment JIT compiler that uses copy-and-patch. It can be enabled with --enable-experimental-jit. It uses stencils written in C code. For variables that are to be determined at runtime, the code is compiled with those parameters set as 0. All of this machine code is stored as a sequence of bytes in the file jit_stencil.h.

4.1 Benefits of Copy-and-Patch

Full JIT compilers convert op-codes to an IR and then machine code. This causes a much longer startup time and significantly more RAM as compared to a JIT implementing Copy-and-Patch.

5 Parallels between a processor and a VM

Just like how Operating Systems execute instructions and have operations that they support, Virtual Machines (like the JVM) have defined instruction sets and operations that they support.

References

[1] wingolog. v8: a tale of two compilers. https://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers.

[2] BlinkOn. BlinkOn 6 Day 1 Talk 2: Ignition - an interpreter for V8. https://www.youtube.com/watch?v=r5OWCtuKiAk.

[3] HAORAN XU, FREDRIK KJOLSTAD. Copy-and-Patch Compilation. https://fredrikbk.com/publications/copy-and-patch.pdf.

[4] Pinaraf's website. Look ma, I wrote a new JIT compiler for PostgreSQL. https://www.pinaraf.info/2024/03/look-ma-i-wrote-a-new-jit-compiler-for-postgresql/.