2024-04-15-JIT vs. Fast Interpreter

A *JIT* is essentially *Eval*.

Eval is essentially the compiler.

A compiler *interprets* the source code, but, does so at “compile time” in hopes of reducing the need for some *interpretation* at “run time”.

A compiler is a preprocessor that evaluates certain bits of code at “compile time” in hopes of reducing the amount of work required at “run time”. The work doesn’t go away, it just gets moved into the preprocessor.

Additionally, repetitive work can be reduced by doing it only once, then reusing the result multiple times. If possible, the compiler performs the work itself and installs the result into the “run time” code (the “binary” executable or object file). If not possible, the compiler rearranges the “run time” code such that it needs to perform the work once at “run time”, and gets to reuse the work instead of recomputing it each time the result is needed.

Furthermore, some patterns of use, such as looking up the value of variables in some sort of table, can be reduced to simpler, cheaper code sequences that reduce the overhead of lookup. Compilers split up the lookup table under the hood, and, create little, cheaper lookup tables using the stack instead of a heap. Stack operations are often better supported by hardware (opcodes) than heap operations.

To perform such preprocessing, compilers require programmers to designate variables as being “local” and “static” and “scoped”, versus being “global” and “persistent”.

Code that contains no opportunities for optimization, such as repeated evaluations or loops, is “straight-line code”, and, may not actually benefit from being compiled.

A JIT tracks statistics about bits of code. When the statistics indicate that a piece of code is being used repetitively, the JIT compiles the piece of code in hopes of

amortizing the cost of compiling (*eval()*ing) the code versus running the code many times.

Tracking such information requires extra work.

For example

```
...
y := 1 + 2
z := 1 + 2
...
```

Might be compiled as...

```
...
temp := 1 + 2
y := temp
z := temp
...
```

Or

```
...
loop
      a [i] := 1 + 2
end loop
...
```

(Ignoring the other inner details of the loop, for example the increment and exit condition)

Might become

```
...
temp := 1 + 2
loop
      a [i] := temp
end loop
...
```

Or

```
...
local y := 1
global z := 2
...
```

Might allow the compiler to preprocess the code such that lookup up for “y” is cheaper than the lookup for “z”.

If the source code doesn't contain sequences such as the above, then adding the overhead of including JIT tracking might not improve the execution of the compiled code. In fact, adding a JIT might slow the code down, due to the extra overhead of tracking information needed by the JIT.

Byte Code Interpretation

A major use-case for compilation is to remove the repetitive work required to read and interpret a script ("code") character-by-character.

An easy optimization for these situations is to simply build a preprocessor that reads scripts character-by-character and compiles them into more efficient byte-codes.

Byte-codes are fixed-size encodings of character sequences. Hardware can usually interpret[1] fixed-size encodings more efficiently. For example, the string "`while`" might be encoded as a single integer, say 42. A hardware CPU can deal with a single number, like 42, more efficiently than with words that are sequences of characters. Often, character sequences are of unknown length and require looping for interpretation, whereas encodings of words as integers are fixed size in length, don't require looping, and, can be reduced to very efficient operations at the hardware CPU level.

Once code has been preprocessed into byte-code, it can run much faster at "run time". Can the code be further optimized to run even faster? The answer depends on the type of sequences contained in the code.

Calculations tend to require repetitive operations which tend to contain sequences that provide opportunities for greater optimization.

Other kinds of code, that does not perform a great deal of calculation, like positioning advertisements on web pages, or reacting to asynchronous events (like GUIs reacting to mouse movement, robots reacting to sensor data, etc.) don't provide as much opportunity for optimizations such as described above, and, might never benefit from the extra work of tracking JIT information at "run time".

---

[1] Note that a hardware CPU is an interpreter. It is just a really fast interpreter, because it is implemented in hardware instead of being implemented in software.

What Kinds of Code Benefit From Compilation and What Kinds Don't Benefit?

In general, code that computes values using loops and recursion can benefit from compilation.

In general, straight-line code might not benefit from compilation. Compiling this kind of code is moot and causes inefficiency in the workflow. The workflow requires an extra compilation preprocessing step before being run, instead of simply skipping the preprocessing step and running the code.

Note, too, that computer nodes physically distributed in space - like on the internet, or even in robotics - cannot, reasonably, use concepts such as *looping* or *recursion* in the large. Nodes can use these concepts internally, but must resort to *message passing* when communicating with other, distributed nodes. Attempts to fake out concepts like *looping, recursion, function-calling*[2]*, etc.,* in distributed situations, say by using concepts such as *RPC*s results in extra overheads and inefficiencies.

The "killer app" for compilation is the conversion of text into bytecodes.

Secondarily, it looks like compilation of variable lookup using static, scoped variables *should* be a "killer app", too, but, the restrictions imposed by this regime tend to restrict progress in coding. For example, lexical scoping removes

---

[2] Function-based code causes ad-hoc *blocking.* The caller must block - suspend itself - until the callee returns a result. Programmers had to expend extra effort to invent the - inefficient - concept of *preemption-based operating systems* in order to support the paradigm of function calling. CPUs support *subroutines.* The paradigm of function-calling is but one way to use the more general concepts of *subroutines*.

names from variables and converts these into simple numeric offsets into a stack of slots. This makes concepts like *introspection*[3] more difficult to conceive of.

[3] You *can* do introspection using static scoping, but, this requires extra work and creates an impedance mismatch that tends to prevent imaginative uses of the idea. Another example might be the concept of macros. Macros were invented in dynamically-scoped Lisp. A simple unhygienic, Lisp macro for an IF-THEN-ELSE construct is approximately one *line* of code, whereas the same macro expressed in hygienic form is about one *page* of code. It seems unlikely that Lispers would use macros as frequently if they had to stop and think about niggly details each time they wanted to abstract their code using a macro. In fact, hygiene is free if the macro processor is thought of as a pure preprocessor instead of being wound deeply into the bowels of the same language. In relatively rare situations, unhygienic macros result in debugging nightmares. The invention of hygienic macros prevents those specific cases of bugs by warping the base language. It might be more useful to leave the base language alone and to create error-checking *linters* that detect the potential for bugs due to the lack of hygiene. Creating such *linters* is more in line with the concepts of “design rule checking” found in electronics design practices.

Appendix

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*