An Analysis of Modern Compilation Techniques

To What Extent Can Just-In-Time Compilers Execute Code at Speeds Comparable to Ahead-Of-Time Compiled Code

Computer Science
3213 Words

September 30, 2021

Contents

1 Introduction

Code can be executed using either a compiler or an interpreter. Compilers are programs that read source code and emit machine code Ahead-Of-Time (AOT) to later be executed, whereas interpreters are programs which step through source code line by line executing instructions. There is consensus that compiled code will execute faster than interpreted code;[1] however, interpreters are much more versatile, portable and fast to work with.[2] There is a type of interpreter, however, which attempts to harness both the simplicity of interpreters and speed of compilers. These interpreters generate machine code and execute them as they step through the source code. This method has been dubbed Just-In-Time (JIT) compilation. This paper will examine to what extent JIT compilers can execute code at a speed comparable to AOT compilers. It is important to note that AOT compilers themselves do not execute code, but rather emit code; therefore, when analyzing execution speed of AOT compilers, this paper will be considering the speed of the code it emits, not compilation itself.

This paper will speak thoroughly about instructions, and it is important to distinguish between when they are native and virtual. A native instruction is defined in the CPU's instruction set; when one is executed, there is a physical location in the CPU which corresponds to the given instruction, and the hardware will follow a specific process in order to carry out the operation. A virtual instruction, however, is an arbitrary instruction defined independently of the CPU. Since a CPU can only execute native instructions, when a virtual instruction is executed, it is, in reality, being mapped to one or more native instructions. This process of mapping virtual instructions to native instructions introduces complexity, and, often, a reduction in code execution speed.

[1] "What is the difference between a compiled and an interpreted program?," University Information Technology Services, August 20, 2021, `https://kb.iu.edu/d/agsz`.

[2] Anton Ertl and David Gregg, "The Structure and Performance of Efficient Interpreters," Journal of Instruction-Level Parallelism, November 3, 2003, `https://jilp.org/vol5/v5paper12.pdf`.

It is also important to define abstraction. In the context of programming languages, abstraction refers to the complexity of instructions, as well as how distanced the language is from fundamental aspects of computing such as memory management. At a basic level, virtual instructions, as previously defined, are a form of abstraction. This definition allows for a general distinction between programming languages: low level, and high level. A low level programming language has relatively little abstraction – that is to say that its instructions map quite closely to a CPU's native instructions, and the user must be conscious of memory management. High level languages, however, define complex virtual instructions which are later mapped to native instructions. Finally, in the context of multi-stage compilation, the term 'lowering' simply refers to translating the current representation of the source code into the next representation – almost always one with less semantic information than the last.

This paper will first outline the most important factors to a program's execution speed, followed by a brief history of both compilers & interpreters, and an explanation of how they work. Next, the optimizations specific to, or most effective when used with, JIT compilers will be discussed. Finally, the paper will analyze the ability of JIT compilers to match the speed of AOT compiled code both in theory, and in practice.

1.1 Factors Affecting Program Speed

The CPU follows what is known as the fetch–decode–execute cycle in order to process instructions. First the next instruction to be executed is fetched from memory, then the instruction is interpreted, and finally, the decoded data is passed through to the relevant units of the CPU in order to be executed.[3] A CPU must spend at least one cycle executing a native instruction, and can only execute a certain number of cycles per unit time; therefore, the first and most important consideration for reducing code execution speed is limiting the number of native instructions in the program. Another consideration is the complexity of the

[3]Bosky Agarwal, "Instruction Fetch Execute Cycle," 2004, `https://docplayer.net/45636060-Instruction-fetch-execute-cycle.html`.

native instructions being executed: complex instructions require more cycles, and, therefore, more time.[4]

The CPU cache is another important consideration in the context of code execution speed. When a CPU is executing any arbitrary code, it will generally repeatedly access the same sets of memory; this is known as the principle of locality.[5] Computers can exploit this principle by loading neighbouring memory into cache when accessing memory.[6] A cache is a small and extremely fast set of memory, close in proximity to the processor. It stores copies of certain data which the CPU has assumed will likely be accessed in the near future. By predictively loading memory into the cache, subsequent memory accesses, ideally, should be far faster. When the CPU needs to access memory, it first searches the cache.[7] If it does not find the correct memory address, a cache miss has occurred and the data must be found in main memory which takes far longer. Limiting cache misses can have a great impact on the execution speed of a program.

1.2 Early Compilers and Interpreters

When the first computers came about, they had extremely limited speed and memory capacity. As such, writing code in anything other than assembly was not feasible.[8] For some time, there were no other languages available – some higher level ones had been written about, but never implemented. One of the first compilers to become available was FORTRAN. At its inception, FORTRAN produced code that was less efficient than what a competent

[4]Agner Fog, "List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs," Technical University of Denmark, August 17, 2021, `https://www.agner.org/optimize/instruction_tables.pdf`.

[5]William Stallings, *Computer Organization and Architecture Designing for Performance,*, 8th ed. (Upper Saddle River: Pearson, 2010).

[6]ibid

[7]ibid

[8]Maurice V. Wilkes, "Computers Then and Now," *Journal of the Association for Computing Machinery*, no. 15 (1968): 3-4, `http://www.jdl.ac.cn/turing/pdf/p1-wilkes.pdf`.

programmer could write by hand.[9] As limitations in the power of computers persisted, compilers would continue to be unable to perform as well as hand written code. As time went on, however, and hardware improved, compilers were able to become more complex and efficient. Eventually, they would become viable methods of writing code.

Interpreters were particularly difficult to implement and use in real world settings. Not only does the program being executed need to be stored on the computer's memory, but so does the interpreter itself. With memory being extremely limited both in capacity and access speed, interpreters simply were not viable. It must also be considered that there was not much of a need for interpreters at this time. Early CPUs had extremely simple instruction sets.[10] This meant that compilers were far more trivial to implement. Furthermore, computers were, often, highly specialized, and did not require the versatility of interpreters.[11] Nevertheless, with improvements in hardware came some of the first interpreters. In 1958, the first high level, interpreted programming language was implemented: LISP.[12]

1.3 Modern Compilation

Modern compilers are drastically different from the first ones presented. With no real limitations in memory usage or speed, compilers can now perform intricate processes in order to produce the best possible executable. By dividing compilation into multiple stages, the compiler can handle more complex syntaxes and optimizations.[13] Compilers first perform lexical analysis – this is the process of turning the source code into an array of tokens such as identifiers, keywords, and literals. Next, compilers parse the tokens in order to create an

[9]David Padua, "The Fortran I Compiler," University of Illinois at Urbana-Champaign, January 1, 2000, `http://www.cs.fsu.edu/~lacher/courses/COT4401/notes/cise_v2_i1/fortran.pdf`.

[10]David Patterson, "The case for the reduced instruction set computer," SIGARCH Computer Architecture News, no. 8 (1980). `https://dl.acm.org/doi/10.1145/641914.641917`.

[11]Kim Zimmermann, "History of Computers: A Brief Timeline," September 6, 2017, `https://www.livescience.com/20718-computer-history.html`.

[12]Edwin Reilly, *Milestones in computer science and information technology* (Westport: Greenwood Publishing Group, 2003), 156–157.

[13]Craig Chambers, "Staged Compilation,"ACM SIGPLAN Notices, no. 37 (2002). `https://dl.acm.org/doi/abs/10.1145/509799.503045`.

Abstract Syntax Tree (AST). The AST – made up of nodes which represent different components of the language such as variable declarations, function declarations, and expressions – represents the source code in a hierarchical fashion. Next is the type checking stage: analysis is done to contextually determine certain information such as the types of basic literals, and other things not immediately known by the AST.[14]

Finally, the compiler must translate the AST into machine code. There are a vast array of different CPU architectures each with their own instruction set. As such, lowering the AST directly to machine code would be a significant challenge: compilers would have to implement AST lowering logic for each CPU. Instead, most modern compilers first translate the AST into what is known as an Intermediate Representation (IR).[15] IRs are essentially simplified, platform independent assembly languages which can later be easily assembled into platform specific machine code. Generating IR is a crucial step for making an effective, and maintainable cross-platform compiler. Since IR is so simple, it can easily be compiled to native instructions, specific to the target CPU; furthermore, optimizations can be done quite easily. A compiler can make an arbitrary number of optimization passes on IR, making the final product more and more efficient each time. Finally, the compiler will lower the IR to native machine code.[16]

2 Downfalls of Interpretation

The slower code execution speed of interpreters can be explained by abstraction, and memory management. While every interpreter will differ in terms of performance, interpreters will always execute code slower than compilers.[17] Interpreters do not generate machine code

[14]ibid
[15]ibid
[16]ibid
[17]Theodore H. Romer et al, "The Structure and Performance of Interpreters," University of Washington, September 1, 1996, `https://dl.acm.org/doi/10.1145/237090.237175`.

– rather, they determine how to execute code as they step through the source code. This means that the CPU must execute code which, in turn, executes code. This added layer of abstraction is the most obvious and significant reason for the relatively slow speeds.[18]

Instructions executed by interpreters are defined by the interpreter, completely independently of the CPU; therefore, all instructions are virtual. Interpreters must interface all the virtual instructions with native instructions which can be executed by the CPU. The relative complexity of each virtual instruction correlates to how many native instructions must be executed.[19] Therefore the complexity of an interpreter's instruction set is a vital aspect to their performance. With less abstraction in virtual instructions comes less slowdown in execution speed.[20]

An interpreter must also have a system for storing and retrieving data. When interpreted code uses memory, it does not directly access system memory; instead, it references memory managed internally by the interpreter which, in turn, references memory. Since large amounts of virtual instructions will manage memory, this system must be efficient so as to not cause drastic performance penalties. In some cases up to 18% of all instructions executed can be related to memory management.[21]

3 Speed of AOT Compiled Code

By first compiling source code to an IR, the compiler can spend an arbitrary amount of time performing in depth static analysis. It can then follow suit with multiple passes through the IR, making aggressive optimizations each time. After the optimized IR module has been lowered to a set of native instructions, the CPU can simply execute it. In this sense, the

[18]ibid
[19]ibid
[20]ibid
[21]ibid

CPU acts as an interpreter in and of itself; however, it acts purely with hardware.

AOT compilers, inherently, do not face any of the aforementioned issues with interpreters. Unlike an interpreter, AOT compilers do not need to map virtual instructions to native instructions, nor do they have to implement a memory management system; they simply produce an array of native instructions to be executed by the CPU. During code execution, everything will be managed directly by the computer's hardware. An AOT compiler's only task is to produce the most efficient program possible.

While the time a compiler spends compiling source code is a concern when designing a compiler for practical use, compilation time is not a concern of this analysis; therefore, when comparing to JIT code execution speed, the paper will assume that AOT compiled code has been optimized to the fullest extent possible, no matter the time spent compiling.

4 Improving Upon Interpretation with JIT Compilation

JIT compilers attempt to address some of the performance issues with interpreters. Instead of handling virtual instructions directly, JIT compilers generate machine code to execute. The code generation strategy depends on the compiler's implementation: code can be generated line by line, block by block or in any other arbitrary order.[22] During compilation, there are many different optimizations that can be applied. These include redundancy elimination, dead code elimination, memory to register promotion, and many more. These optimizations, however, are done by statically analyzing the dynamically generated code. Any optimization that can be done statically, without run-time information, can also be done with AOT

[22]Terence Parr and Johannes Luber, "The Difference Between Compilers and Interpreters," Archived 2014-01-06 at the Wayback Machine: `https://web.archive.org/web/20140106012828/http://www.antlr.org/wiki/display/ANTLR3/The+difference+between+compilers+and+interpreters`.

compilers. Such optimizations, therefore, will not be the focus of this section.

In the context of code optimization, dynamically generating machine code provides one significant advantage over AOT generation: access to information only available at run-time.[23] Run-time information includes the specific architecture of the CPU executing the code, the values of variables and parameters, frequency of accesses to certain memory, and much more. By harnessing this information, JIT compilers can eliminate a vast array of time consuming instructions.[24]

Common code patterns such as null checking can often be eliminated when generating code dynamically. With access to the values in each variable, the compiler can, at times, be certain that a condition is impossible and remove the need for a value comparison entirely.[25] Not only does this eliminate a comparison instruction, but it also reduces code branching which can be costly, and, in some cases, negatively impact cache hit rates.[26] Given the specific CPU architecture, the JIT compiler can make highly specialized optimizations for that given CPU which would otherwise be impossible.[27] Devirtualization is another significant optimization enabled only by run-time information. An analysis of the code in **Appendix A** from the perspective of an AOT compiler highlights the significance of devirtualization.

When the 'foo' method is called in 'call_foo', the compiler is unsure of what function to call. An instance of the derived class is passed as type 'Base' to 'call_foo', therefore, the compiler must handle the call to 'foo' with dynamic dispatch. Dynamic dispatch requires determining what function to call by reading from a virtual table.[28] In this case, 'Derived::foo'

[23]John Aycock, "A Brief History of Just-In-Time," University of Calgary, June 1, 2003, `http://eecs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/JustInTimeCompilation.pdf`.

[24]ibid

[25]Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani, "Effective Null Pointer Check Elimination Utilizing Hardware Trap," IBM Tokyo Research Laboratory, November 12, 2000, `https://prolangs.cs.vt.edu/refs/docs/kawahito-asplos00.pdf`.

[26]ibid

[27]Aycock, "A Brief History of Just-In-Time."

[28]Scott Milton and Heinz Schmidt, "Dynamic Dispatch in Object-Oriented Languages," Australian National University, March 1994, `https://www.researchgate.net/publication/2459095_Dynamic_Dispatc`

is called instead of 'Base::foo'. Dynamic dispatch, of course, is slower than static dispatch. Since dynamic dispatch is handled at runtime, it can seldom be optimized by an AOT compiler. JIT compilers, however, have several algorithms available for de-virtualizing methods with relative ease.[29]

4.1 Code Profiling

Performing such optimizations while generating code, however, is very time consuming. Therefore, JIT compilers conduct code profiling.[30] In the context of JIT compilers, code profiling is a type of dynamic analysis run on the generated machine code. This analysis will collect data on a vast number of different categories including how long function calls take, how frequently a function is called, and what any given function callee's call chain is.[31] These statistics allow the compiler to identify code hotspots – areas where code is run most frequently and where large amounts of time are spent.[32] The compiler can then make efficient use of its time by optimizing only these areas of the program.[33] Profile guided optimizations (PGOs) are perhaps the most significant advantage JIT compilers have over AOT compilers in terms of code execution speed.[34]

4.1.1 Profile Guided Optimization

As previously mentioned, limiting cache misses can have a significant impact on code execution speed. JIT compilers have the unique opportunity to use dynamic profile information to reorder the generated code and improve locality. JIT compilers can place frequently-called

h_in_Object-Oriented_Languages.

[29]ibid

[30]Parr and Luber, "The Difference Between Compilers and Interpreters".

[31]Function callee refers to the function being called. If function A calls function B, then A is the caller and B is the callee.

[32]Susan Graham, Peter Kessler, and Marshal McKusick, "gprof: a Call Graph Execution Profiler," University of California: Berkeley, June 1, 1982, https://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf.

[33]Parr and Luber, "The Difference Between Compilers and Interpreters".

[34]April Wade, Prasad Kulkarni, and Micheal Jantz, "AOT vs JIT: Impact of Profile Data on Code Quality," University of Kansas, June 2017, https://dl.acm.org/doi/pdf/10.1145/3140582.3081037.

basic blocks (BBs) separately from infrequently-called ones.[35] This generally enables the CPU to only access memory addresses close in proximity to one another which reduces the cache miss rate. Furthermore, callees can frequently be placed near their callers, which can also improve code locality.[36]

Call instructions are relatively costly,[37] and thus an ideal target for optimization, particularly when present in a code hotspot. Function inlining is the process by which a callee function is merged into the caller, removing the need for a call instruction. There are several considerations for effective inlining. Firstly, when a callee's body is merged with the caller, its code is duplicated; this means that inlining large functions would drastically increase the size of the program. Since reducing the number of native instructions is one of the key considerations for execution speed, compilers only inline small, simple functions.[38] Secondly, after inlining, both hot and cold BBs can be placed next to each other. This leads to more frequent cache misses. In these situations, such as in figure 1, code reordering is necessary in order to improve code locality. Thirdly, the frequency of calls to a function must be considered before inlining. If an infrequently called function is inlined, the caller, after being merged, will have many more instructions that are rarely executed. In this case, the slowdown brought about by the larger BB size will outweigh any speed benefits of inlining.[39]

It is worth noting that, in some cases, AOT compilers can perform similar optimizations; however, without code profiling, function inlining and code reordering is extremely limited, and often requires the user to explicitly indicate where code can be optimized.[40]

[35] A basic block (BB) is a section of instructions with no branches other than the entry and exit. A hot BB is one that is executed frequently, whereas a cold BB is executed infrequently.

[36] Xianglong Huang, Brian Lewis, and Kathryn McKinley, "Dynamic Code Management," University of Texas at Austin, Accessed September 3, 2021, `https://www.researchgate.net/publication/228615571_Dynamic_code_management`.

[37] Fog, "List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs."

[38] Omer Boehm et al, "Aggressive Function Inlining with Global Code Reordering," IBM, November 15, 2006, `https://dominoweb.draco.res.ibm.com/reports/h-0247.pdf`.

[39] ibid

[40] Examples of this are: the 'inline' keyword and '__builtin_expect' builtin function.

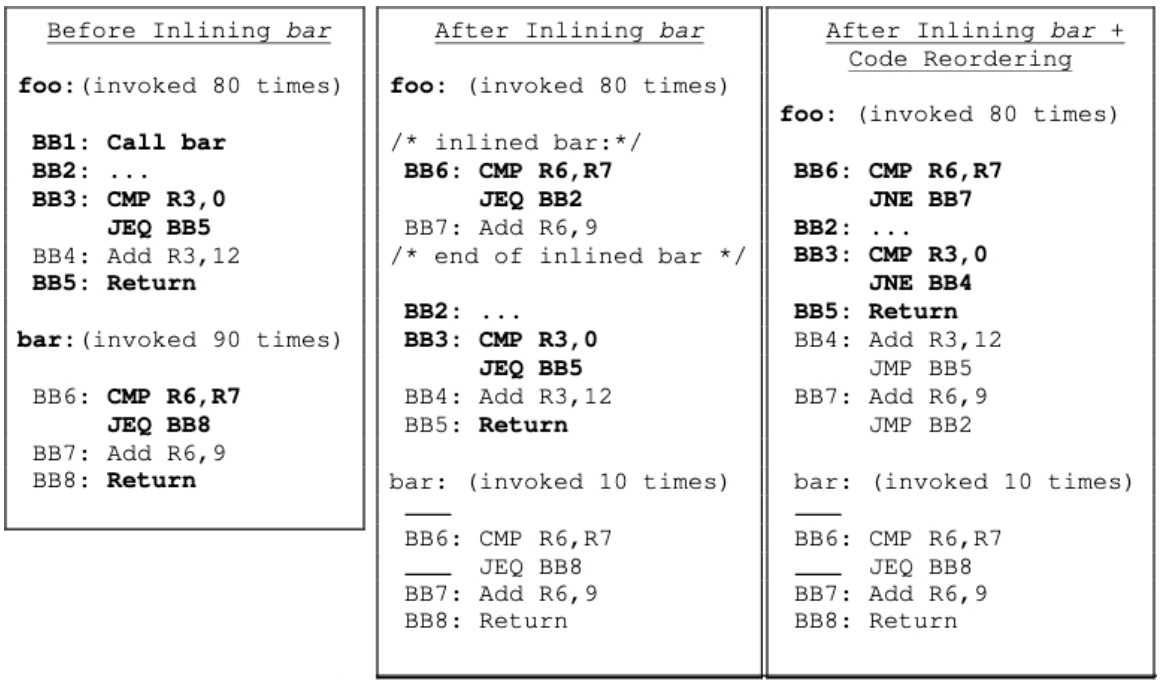

Figure 1: Code before and after being inlined, with and without code reordering. (Figure by Omer Boehm et al, *Aggressive Function Inlining with Global Code Reordering*, November 15, 2006, IBM, `https://dominoweb.draco.res.ibm.com/reports/h-0247.pdf`)

5 Conclusion

5.1 Performance Issues with Just-In-Time Compilation

JIT compilers exist in two stages: the startup state and steady state. All JIT compiled code is initially encountered as virtual instructions which must be compiled. Compilation uses time which could otherwise be used executing code, thus the startup stage is a significant barrier to achieving execution speeds comparable to AOT compiled code. Identifying hotspots with code profiling does limit the startup cost of a JIT compiler, but it cannot eliminate it entirely.[41] Performing optimizations, of course, takes time and only exacerbates startup costs. As a result, in practice, many optimizations available to JIT compilers are often not implemented.[42] Instead, startup costs are mitigated by entering directly into the

[41] Wade, Kulkarni and Jantz, "AOT vs JIT: Impact of Profile Data on Code Quality."

[42] "Understanding JIT Compilation and Optimizations," Oracle, accessed November 3, 2021, `https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html`.

steady state of code execution without optimizations.

There are many optimizations that are employed by both JIT and AOT compilers that were not discussed in this paper. This is because such optimizations would have the exact same benefits to execution speed for both types of compilers, and thus would not affect the findings of the analysis in a significant way. In fact, optimizations shared by both types of compilers aid JIT compilers less as they would only contribute to the aforementioned startup times.

It is evident that, when considering execution speed, AOT compilers hold an inherent advantage over JIT compilers. JIT compilers have a vast array of optimizations available to them: some far more, or even only, effective when used on dynamically generated code. The abilities of JIT compilers to match the code execution speeds of AOT compilers will thus depend on the availability and effectiveness of optimizations enabled by runtime information. Without sufficient opportunities to employ PGOs, it is not feasible to match AOT execution speed. The PGOs that were discussed in this paper are not usually available throughout an entire program. Devirtualization, inlining, and dynamic code reorganization – defining features of dynamic code generation – are simply not significant enough to allow JIT compilers to match the execution speed of AOT compilers.[43]

5.2 Limitations of Analysis

In practice, it is difficult to comprehensively test and compare the speeds of JIT and AOT compilers as there are not many well known languages with robust implementations of each type of compiler; furthermore, execution speed is highly dependent on the specific code written: certain programs perform far better when compiled AOT, and others when compiled

[43]Wade, Kulkarni and Jantz, "AOT vs JIT: Impact of Profile Data on Code Quality".

JIT. Therefore, this analysis remained purely theoretical.

It was assumed throughout this paper that AOT compilers could spend an arbitrary amount of time compiling code in order to produce the most efficient program possible; however, in reality, compilation speed is an important consideration and constraint. Just like a JIT compiler, certain optimizations are often left un-implemented in order to improve compilation speed.

5.3 Final Remarks

JIT compilers were, initially, not designed to perform better than AOT compilers; rather, they aim to improve upon traditional interpreters while maintaining their versatility. In this regard, JIT compilers are extremely successful. Code can theoretically be written specifically to induce JIT compiler specific optimizations; however, this is not a practical method of writing code and would most likely not be possible for complex programs. Therefore, when considering only code execution speed, JIT compilers cannot consistently match AOT compilers.

A Dynamic Dispatch for Virtual Methods

```
#include <stdio.h>

class Base {
 public:
        virtual void foo() {
                printf("Hello from Base class\n");
        }
};
```

```
class Derived : public Base {
 public:
        void foo() {
                printf("Hello from Derived class\n");
        }
};

void call_foo(Base *b) {
        b->foo();
}

int main() {
        Derived *d = new Derived();
        call_foo(d);
        return 0;
}
```

Works Cited

Agarwal, Bosky. "Instruction Fetch Execute Cycle." 2004. https://docplayer.net/45636060-Instruction-fetch-execute-cycle.html.

Wilkes, Maurice. "Computers Then and Now." Journal of the Association for Computing Machinery, no. 15 (1968): 3-4. http://www.jdl.ac.cn/turing/pdf/p1-wilkes.pdf.

Patterson, David. "The case for the reduced instruction set computer." SIGARCH Computer Architecture News, no. 8 (1980). https://dl.acm.org/doi/10.1145/641914.641917.

Zimmermann, Kim. "History of Computers: A Brief Timeline." September 6, 2017. https://www.livescience.com/20718-computer-history.html.

Reilly, Edwin. *Milestones in computer science and information technology.* Westport: Greenwood Publishing Group, 2003.

Chambers, Craig. "Staged Compilation."ACM SIGPLAN Notices, no. 37 (2002). https://dl.acm.org/doi/abs/10.1145/509799.503045.

"What is the difference between a compiled and an interpreted program?." University Information Technology Services. August 20, 2021. https://kb.iu.edu/d/agsz.

Ertl, Anton, and David Gregg. "The Structure and Performance of Efficient Interpreters." Journal of Instruction-Level Parallelism. November 3, 2003. https://jilp.org/vol5/v5paper12.pdf.

Stallings, William. Computer Organization and Architecture Designing for Performance. 8th ed. Upper Saddle River: Pearson, 2010.

Padua, David. "The Fortran I Compiler." University of Illinois at Urbana-Champaign. January 1, 2000. http://www.cs.fsu.edu/~lacher/courses/COT4401/notes/cise_v2_i1/fortran.pdf.

Romer, Theodore, Dennis Lee, Geoffrey Voelker, Alec Wolman, Wayne Wong, Jean-Loup Baer, Brian Bershad, Henry Levy. "The Structure and Performance of Interpreters." University of Washington. September 1, 1996. https://dl.acm.org/doi/10.1145/237090.237175.

Parr, Terence, and Johannes Luber. "The Difference Between Compilers and Interpreters." Archived 2014-01-06 at the Wayback Machine: https://web.archive.org/web/20140106012828/http://www.antlr.org/wiki/display/ANTLR3/The+difference+between+compilers+and+interpreters.

Kawahito, Motohiro, Hideaki Komatsu, and Toshio Nakatani. "Effective Null Pointer Check

Elimination Utilizing Hardware Trap." IBM Tokyo Research Laboratory. November 12, 2000. https://prolangs.cs.vt.edu/refs/docs/kawahito-asplos00.pdf.

Aycock, John. "A Brief History of Just-In-Time." University of Calgary. June 1, 2003. http://eecs.ucf.edu/~dcm/Teaching/COT4810-Spring2011/Literature/JustInTimeCompilation.pdf.

Milton, Scott, and Heinz Schmidt. "Dynamic Dispatch in Object-Oriented Languages." Australian National University. March 1994. https://www.researchgate.net/publication/2459095_Dynamic_Dispatch_in_Object-Oriented_Languages.

Graham, Susan, Peter Kessler, and Marshal McKusick. "gprof: a Call Graph Execution Profiler." University of California: Berkeley. June 1, 1982. https://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf.

Wade, April, Prasad Kulkarni, and Micheal Jantz. "AOT vs JIT: Impact of Profile Data on Code Quality." University of Kansas. June 2017. https://dl.acm.org/doi/pdf/10.1145/3140582.3081037.

Huang, Xianglong, Brian Lewis, and Kathryn McKinley. "Dynamic Code Management." University of Texas at Austin. Accessed September 3, 2021. https://www.researchgate.net/publication/228615571_Dynamic_code_management.

Fog, Agner. "List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs." Technical University of Denmark. August 17, 2021. https://www.agner.org/optimize/instruction_tables.pdf.

Boehm, Omer, Daniel Citron, Gadi Haber, Moshe Klausner, and Roy Levin. "Aggressive Function Inlining with Global Code Reordering." IBM. November 15, 2006. https://dominoweb.draco.res.ibm.com/reports/h-0247.pdf.