International Journal of Advances in Scientific Research
ISSN: 2395-3616 (Online)
Journal DOI: 10.7439/ijasr
Short Communication

Analysis and compilation of the statement into machine code and execute that the machine code for discarding the degree to which performance changes between prototype and production system

Jyoti Dadwal and Bhubneshwar Sharma*

Department of Electronics and Communication Engineering, S.S.C.E.T, under Punjab technical university, India

***Correspondence Info:**
Er. Bhubneshwar Sharma
Assistant Professor,
Department of Electronics and Communication Engineering,
S.S.C.E.T, under Punjab technical university, India
E-mail: bhubnesh86@gmail.com

Abstract

Beyond asymptotic order of growth, the constant factors matter: an asymptotically slower algorithm may be faster or smaller (because simpler) than an asymptotically faster algorithm when they are both faced with small input, which may be the case that occurs in reality. Often a hybrid algorithm will provide the best performance, due to this tradeoff changing with size. A general technique to improve performance is to avoid work. A good example is the use of a fast path for common cases, improving performance by avoiding unnecessary work.

Keywords: Hybrid algorithms, performance parameters

1. Introduction

As performance is part of the specification of a program – a program that is unusable slow is not fit for purpose: a video game with 60 Hz (frames-per-second) is acceptable, but 6 fps is unacceptably choppy – performance is a consideration from the start, to ensure that the system is able to deliver sufficient performance, and early prototypes need to have roughly acceptable performance for there to be confidence that the final system will (with optimization) achieve acceptable performance. This is sometimes omitted in the belief that optimization can always be done later, resulting in prototype systems that are far too slow – often by an order of magnitude (factor of 10×) or more – and systems that ultimately are failures because they architecturally cannot achieve their performance goals, such as the Intel 432 (1981); or ones that take years of work to achieve acceptable performance, such as Java (1995), which only achieved acceptable performance with Hotspot (1999).

The degree to which performance changes between prototype and production system, and how amenable it is to optimization, can be a significant source of uncertainty and risk. As early as the Chou dynasty, about the fourth century B.C., a certain Sun Tzu wrote a brief treatise called "The Art of War," which made much of knowledge for the successful conduct of war. Sun Tzu's wisdom would endure. Centuries later, his treatise was consulted by Chairman Mao and memorized in its entirety by officers of the Japanese Imperial Navy in World War II; a quote from it opens a U.S. Army field manual of the 1980s that marks the first significant change in army field tactics since the U.S. Civil War. Knowledge, says Sun Tzu, is power and permits the wise sovereign and the benevolent general to attack without risk, conquer without bloodshed, and accomplish deeds surpassing all others. The New York Stock Exchange recently published its own treatise which says, less poetically, the same thing: increased productivity derives from more capital, from better capital, but most important of all, from "working smarter" with the capital at hand. American business leaders are as concerned with the art of war as Sun Tzu and his legion of international disciples have been, but in this century the battlefield has changed. Instead of the mountains and valleys of ancient China, the vital battlefield has become the international marketplace.

There are three general modes of execution for modern high-level languages:

Interpreted

When code written in a language is interpreted, its syntax is read and then executed directly, with no compilation stage. A program called

an interpreter reads each program statement, following the program flow, then decides what to do, and does it. A hybrid of an interpreter and a compiler will compile the statement into machine code and execute that; the machine code is then discarded, to be interpreted anew if the line is executed again. Interpreters are commonly the simplest implementations of the behavior of a language, compared to the other two variants listed here.

Compiled

When code written in a language is compiled, its syntax is transformed into an executable form before running. There are two types of compilation:

Machine code generation

Some compilers compile source code directly into machine code. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages. See assembly language.

Intermediate representations

When code written in a language is compiled to an intermediate representation, that representation can be optimized or saved for later execution without the need to re-read the source file. When the intermediate representation is saved, it may be in a form such as byte code. The intermediate representation must then be interpreted or further compiled to execute it. Virtual machines that execute byte code directly or transform it further into machine code have blurred the once clear distinction between intermediate representations and truly compiled languages.

In 1940s, the first recognizably modern electrically powered computers were created. The limited speed and memory capacity forced programmers to write hand tuned assembly language programs. It was eventually realized that programming in assembly language required a great deal of intellectual effort and was error-prone. The first programming languages designed to communicate instructions to a computer were written in the 1950s. An early high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000. John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer. Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

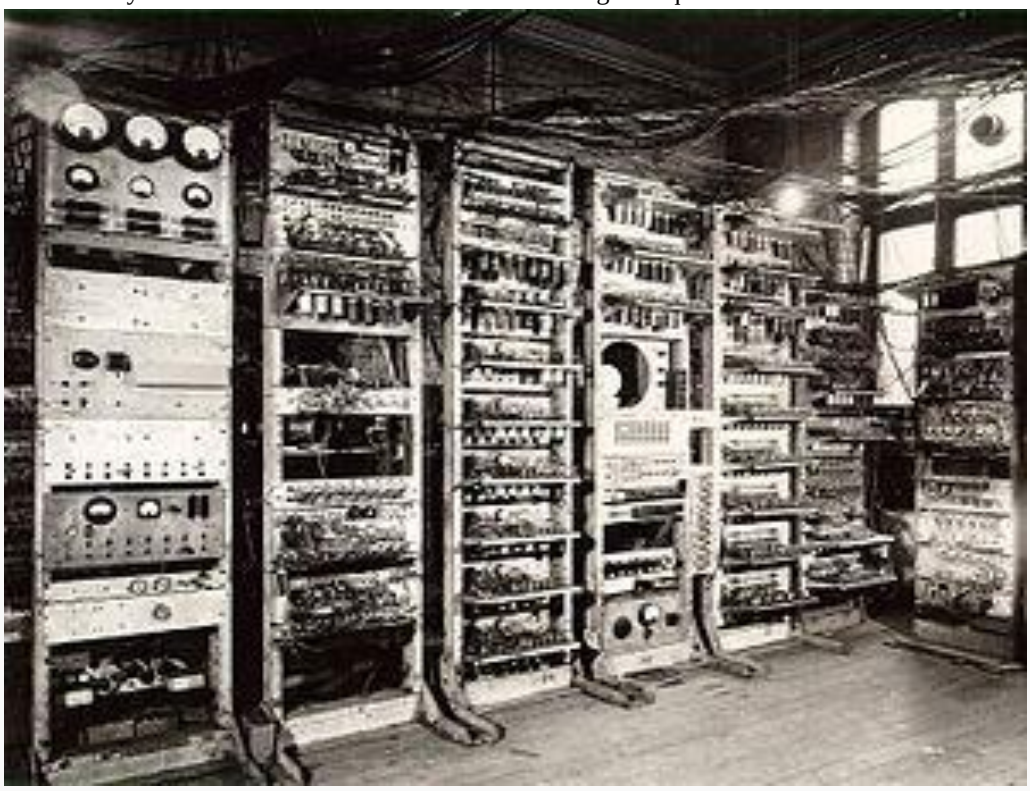

Figure 1:- The Manchester Mark 1 ran programs written in Autocode from 1952.

At the University of Manchester, Alick Glennie developed Autocode in the early 1950s. A programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first compiled high-level programming language.

2. Conclusions

The first computer codes were specialized for their applications. In the first decades of the 20th century, numerical calculations were based on decimal numbers. Eventually it was realized that logic could be represented with numbers, not only with words. For example, Alonzo Church was able to express the lambda calculus in a formulaic way. The Turing machine was an abstraction of the operation of a tape-marking machine it aimed to create an "epoch-making computer" with-supercomputer-like performance and to provide a platform for future developments in artificial intelligence. There was also an unrelated Russian project also named as fifth-generation computer

References

[1] Shapiro, Ehud Y. "The fifth generation project—a trip report." *Communications of the ACM* 1983; 26(9): 637-641.

[2] Van Emden, Maarten H., and Robert A. Kowalski. "The semantics of predicate logic as a programming language." *Journal of the ACM (JACM)* 1976; 23 (4): 733-742.

[3] Shapiro E. A subset of Concurrent Prolog and its interpreter, ICOT Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983. Also in Concurrent Prolog: Collected Papers, E. Shapiro (ed.), MIT Press, 1987, Chapter 2.

[4] Carl Hewitt Inconsistency Robustness in Logic Programming ArXiv 2009.

[5] *Hendler, James.* "Avoiding another AI Winter" *(PDF). IEEE Intelligent Systems 2*008; 23 (2): 2–4. Doi*:* 10.1109/MIS.2008.20.