Programming Language Implementations

Let *L* be any programming language, from high-level to intermediate to machine language. A **virtual machine** for *L* is a software program that simulates execution of any programs in *L*. A virtual machine for *L* is also referred to as a **simulator** or a **software interpreter** for *L*. A virtual machine can be constructed for any programming language whatsoever, for example for the existing machine languages for Intel and AMD hardware processors, for intermediate languages like Java bytecode, for high-level languages like Lisp and Basic. Virtual machines for *L* achieve **hardware-independent ("portable")** execution of *L*.

A **hardware machine** for *L* is a hardware processor that directly executes any programs in *L*. The Intel and AMD hardware processors execute their machine languages. A hardware machine can be built to execute Java bytecode directly; SUN Microsystems has built such hardware machines. Hardware machines can even be built to execute any high-level languages directly, although they are usually not commercially viable. In principle, hardware machines can be constructed for any programming languages whatsoever. Hardware machines for intermediate languages (like Java bytecode) and high-level languages are often called "hardware interpreters". The emergence of the field-programmable gate array technology and of hardware description languages has facilitated construction of custom hardware machines for high-level and intermediate languages.

Fig. 1 Programming Language Implementation

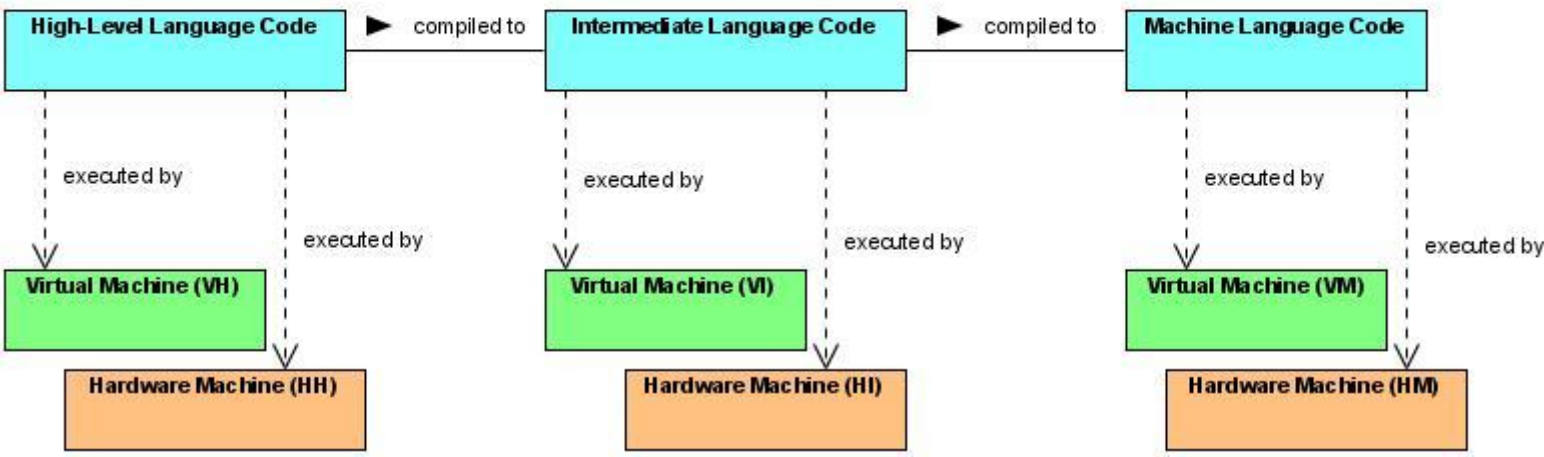
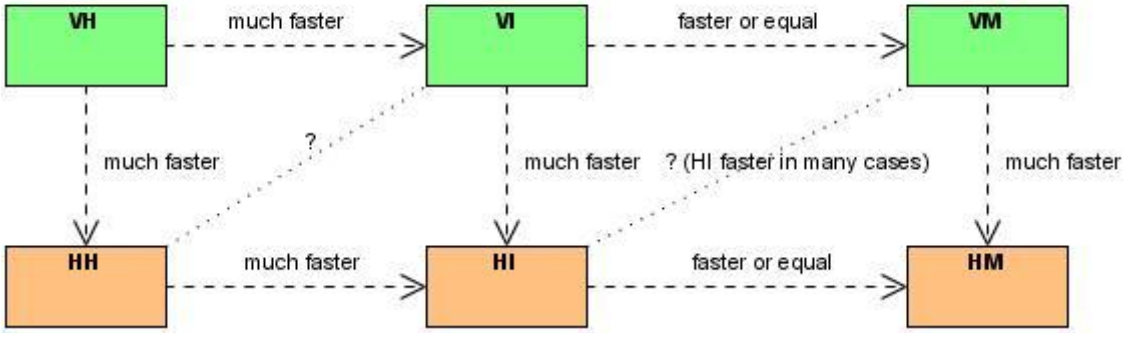


Fig. 2 Efficiency Comparison

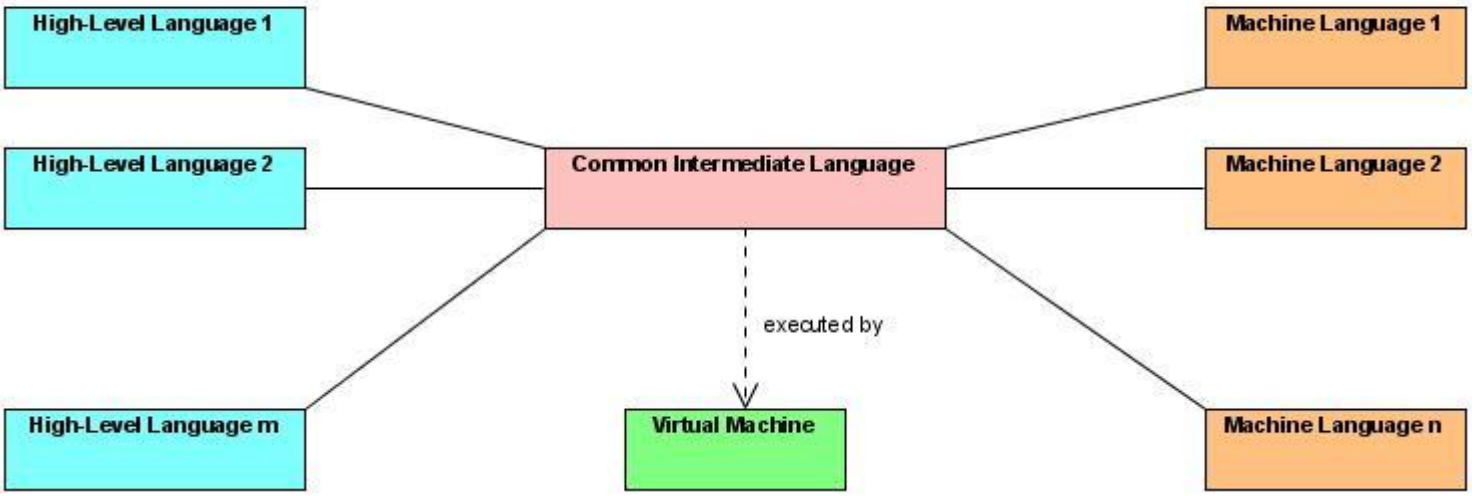


The use of **intermediate languages (ILs)** is nowadays very common in programming language implementations. There are dozens of ILs – some are close to assembly languages for actual hardware processors while others are more abstract and higher-level. Some functional and logic language implementations have even used a conventional procedural language like C as IL.

The benefits of ILs are summarized as follows.

- Two-phase generation of native machine code via IL is easier and more reliable than direct generation without IL.
- Better optimization. Optimization is the process of transforming IL code or native machine code so that it runs faster and/or has a smaller code size, for example by removing redundant instructions. Thanks to its cleaner structure, optimization of IL code is more effective than optimization of native machine code. Performing optimization to both intermediate and native machine code results in better optimized native machine code than performing optimization to native machine code only.
- Provide two execution options: IL code can be executed by a virtual machine or can be further compiled into a native machine language of a hardware processor.
- A well-designed IL can serve as a bridge for compiling multiple high-level languages into multiple native machine languages (Fig. 3). Only need to construct *m* front ends and *n* back ends with respect to the common IL. Without a common IL, need to construct *m* × *n* compilers independently, each with both front and back ends. Also, a single virtual machine for the common IL can be used to execute all *m* high-level languages.

Fig.3 Common intermediate language as a bridge between high-level languages and machine languages



Execution of intermediate code by a virtual machine is much slower than execution of the corresponding native machine code by a hardware processor – usually difference factors have been observed to be in the range of 8 to 20. On the other hand, the positive side of virtual machines is:

- They provide hardware-independent, portable execution of languages. Languages can be executed on any hardware/OS platform with a virtual machine installed. For example, an application software program can be compiled into intermediate code, then packaged with a virtual machine and distributed.

- Being software simulations, they are programmed to recognize runtime errors such as arithmetic overflows, division by 0, and array indexing out of bounds, and terminate gracefully with informative error messages. Runtime errors encountered in native machine code execution usually halt with uninformative messages. For the same reason, useful debuggers are relatively easily written for virtual machines. This is why software simulators are often used for the teaching of assembly languages of actual hardware processors.