

Should All Source Code Be Compiled?

Daniel Young,
BSc (Hons) Software Engineering
Supervisor: Rosemary Brown
Second Reader: Raj Patel
November 22nd

Abstract

The recent boom in computing has led to many high-level languages. Gone are the days when developers would have to individually write, compile and link their software. Today we use software that utilises both Interpretation and Compilation.

This paper determines if the ideal world should revolve around 100% compiled source code.

It first examines the various aspects that software developers require from their source code, ranging from efficiency to security to portability. It assesses each one and uses evidence found from sources to back up each claim.

The paper then moves onto determining why source code must be either compiled or interpreted, on the basis that modern, powerful software is too far to massive to program in machine-code.

It then moves onto the compilation technique and evaluates its advantages and disadvantages with regards to the aspects required by software developers. It is determined that compilation causes serious portability issues and that excessive compile time on large projects can reduce productivity. Compilation is found to excel in both security and efficiency.

Next the paper examines the two other competitors to compilation, Interpretation and Byte-Code. It is determined that each solve the problems caused by compilation, however interpretation poses serious speed issues, limiting its use in many scenarios. Byte-Code is found to go some way in fixing this issue, but is still not supported by many top languages.

The paper concludes that while compilation may solve the largest sub-set of aspects required by developers, in the end it does not fix them all – making it unsuitable for all source code. It recommends that developers chose their source code processing method based on the scenario on which they are working.

1. Introduction

Back in the early days of computing all software had to be compiled from either a low or high level language into a form that the computer could understand (Sim, S. (1995)). How they were compiled back then, whether it be through punch cards or a software-based compiler is not significant.

Since the early days however, compilers have evolved dramatically. In 1973, Denis Ritchie developed the basis of the C compiler / language. It is the evolution of this language, and indeed other languages that has led to more and more complex pieces of software that we use this present day (CRN (2000)).

The compiled code (also known as ‘machine code’) created by the compiler introduces a couple of problems, one of which negates the portability of the software created.

“Compiled code is translated directly from a high-level language into machine code instructions, which, by definition, are platform dependent--after all, an Intel x86 chip has a completely different instruction set to a Power-PC [Apple Macintosh] chip”

Fitzpatrick, R (2003)

Another issue, raised by many including Jim Cullinan, lead project manager on Microsoft Windows XP and echoed by others on the Internet is that compilation can take a very long time on large projects, which can have a serious impact on productivity if compilation must take place several times per day.

Various other methods of source code processing have been created and tested in industry to try and address the problems caused by compilation, but these in turn have caused their own unique disadvantages.

Section 2 of this paper identifies what developers require from their source code and in what context. Section 3 looks at why high level source code needs to be processed and takes an indepth look at the compilation technique. Section 4 looks at the available replacements to compilation and cross-references their advantages and disadvantages against that of compilation. Finally Section 5 reaches an unbiased conclusion, based on the evidence found.

In short, this paper determines the suitability of using compilation for all computer source code.

2. What Software Developers want from their Source Code

2.1. Efficiency

In Speed Terms

Personal Computers are getting faster and faster with every month that goes by. Moore’s Law states that states that computers double in speed with each passing year (Moore, G. (1965)). Computers are many hundreds of times faster than they were just a decade ago. Powerful, modern computers however don’t come cheap and many times over one will find that businesses and home users alike are using dated systems.

The reason companies and users do not upgrade their machinery is because they happily run the software that they require (SAP (2002)). In the end though, more and more is expected of both

businesses and users and they soon find that the software no longer caters for their needs. They will want updated software, but not necessarily the expense of updating their hardware too.

Efficiency in speed terms is talking about how fast a piece of software can run on a computer. It is measured on a pure time basis. For example, how long it takes the software to search for a single item within one million items. It is then up to the developer or the client to decide if how long it took is within an acceptable limit. Acceptable limits will vary from scenario to scenario, it is fair to assume that a car's 'Anti-Lock' braking system will have stricter limits than a bus time-table system.

In the end, while speed may not always be the single most crucial element to software, the challenge for developers is to write software with the aim of it running as quickly as possible as this reduces the need for clients to buy new hardware. Two things effect the speed of the software from a development point of view. The first is how well the developer wrote the software, the second is how well the source code was processed.

In Size Terms

Dated hardware, while could be slow relative to modern hardware (see above), can also be lacking in storage space. Storage space is generally referring to how much information the computer can store on its hard disk. An argument could be that storage space is relatively cheap nowadays (around £50 for a relatively large disk ~ source PC World), however, if the software has to run on many computers in a business, it is easy to see that it soon gets expensive.

If PSP (Personal Software Process) is utilised by software developers, then they should always be aiming to write software with quality in mind, and an aspect of quality is efficiency (Casey, C (2000)). Leitner states that it is often the case that well written source code is also small in size, and so it is not normally a trade off between speed or size. However, there are many situations where there will be a major trade off (Leitner, F. (2001)).

The way source code is processed after it has been written can have a great impact on the amount of space it takes up. Sections 3 and 4 look at good methods of source code processing.

2.2. Portability

Defined as,

“An application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new environment in the class is less than the effort of redevelopment.”
(Mooney, J. (2001))

How portable software is had not been a concern while there were only one or two platforms out in the public. Recently though, technological advancements have seen the release of a whole modern array of hardware platforms. The PDA (Personal Digital Assistant) and mobile phones are to name just two of the many now available. The introduction to this paper touched on how software written for an IBM based computer would not run on an Apple Macintosh, due to the different hardware they use. This statement is also true for all of the different platforms (Fitzpatrick, R (2003)).

A remark by Dahlstrand (Dahlstrand (1984)) describes portability as “having the same meaning for software as compatibility for hardware”.

Hardware however, is not the only aspect that can effect portability. The operating systems they utilise can also have a drastic effect on portability. In the case of the Apple Macintosh, there is only one operating system available so this does not cause a problem. However, for the IBM based computer there are many. Windows and Linux are just two of the many available. Each of these operating systems require software that satisfies their own unique requirements.

Portability is more of an aspect sought by Software Houses rather than the software developers themselves. They are starting to realise that they are losing custom if their software isn't available on all of the popular platforms. A recent study conducted by The University of Toledo reports that the Apple iMac is outselling PCs at a rate of four-to-one (Centre for Visual Arts (2000)). This translates to millions of potential customers on a different platform. Recently, we have seen large companies such as Microsoft, makers of the Windows Series make their software available on the Macintosh platform (Kevin Browne (2003)).

Software that needs to be written many times to cater for each of the available platforms is a waste of time, money and resources. Ideally, the source code should only need to be written once and then be available for all platforms.

2.3. Security

The final aspect wanted from source code is protection against plagiarism. Plagiarism is a severe problem in modern computing, especially with the release of many advanced reverse-engineering tools. Companies often employ talented computer programmers with the intent of determining how a piece of software works.

Fairly recently, Trend Micro filed a lawsuit against its two main competitors, McAfee and Symantec due to a patent infringement regarding their source code.

“Trend Micro Incorporated announced today that it has sued its two largest competitors, McAfee Associates (Nasdaq: MCAF) and Symantec Corporation (Nasdaq:SYMC), for infringement of its recently issued U.S. patent on computer virus detection techniques used for data carried over the Internet, electronic mail and groupware.”

(Trend Micro, 1997)

Almost all software can be copied, as the machine-code instructions are available at any time for any one to see. The only thing that can be done is to make the process of determining what they are as hard as possible. Some software houses utilise encryption techniques to encode their software. However, the encoded software has to be presented in a form the computer understands (a decrypted state), and so is ultimately insecure (Serguson, A (2002)).

3. Why Process Source Code and The Compilation Technique

This section looks at why high-level source code needs to be processed and also examines, in depth, the compilation technique, looking at how it works, its advantages and its disadvantages.

3.1. Why High Level Source Code Needs To Be Compiled

Technically, a developer does not require a compiler or an interpreter to develop software. All it would need is for the developer to write the software directly in a language that the computer can understand. This language happens to be nothing more than a series of 1's and 0's known as binary (machine language maps directly onto binary and so can be treated as one of the same). (White, R (2000)) voices the fact that everything you type, draw or do on a computer is, in the end, converted into a stream of 1's and 0's that the computer can process.

Today's software tends to be very powerful, with applications capable of creating 3D images, movies and games that look as good as real life. One would be correct to assume that powerful software like this is very large – they would be correct. Powerful software like this translates to millions and millions of computer instructions. Evidence of this can be found all around us. Take the operating system, Windows 2000. Computerhope.com states,

“Windows 2000 contains over 29 Million lines of code mainly written in C++ 8 Million of those lines alone are written for drivers.”

It is generally regarded that one line of C++ code translates into many machine code instructions (CRN (2000)). It is therefore very feasible that Windows 2000 is well over 100 Million lines of code in length.

Imagine asking a developer to write many millions of lines of code in a language restricted to 0's and 1's (Binary, the language which computers use). The time required to write the software would be unfeasible for any project and the possibility of error is massive (Risley, D (2001)). To solve this issue, other, more abstract languages were created which step away from the computer's primary language of binary.

The new languages allow developers to write software in a form closer to English rather than binary whose aims are to diminish the chance of error and reduce development time.

However the computer cannot process a high level abstract language such as C++, Pascal or Visual Basic just as a Japanese man cannot understand a French or Italian man. To allow a computer to understand the more abstract languages, compilers or interpreters are used to convert, or interpret the language and present it in a form that the computer can understand – just as the Japanese man would require a translator to translate French or Italian into Japanese.

3.2. The Compilation Technique

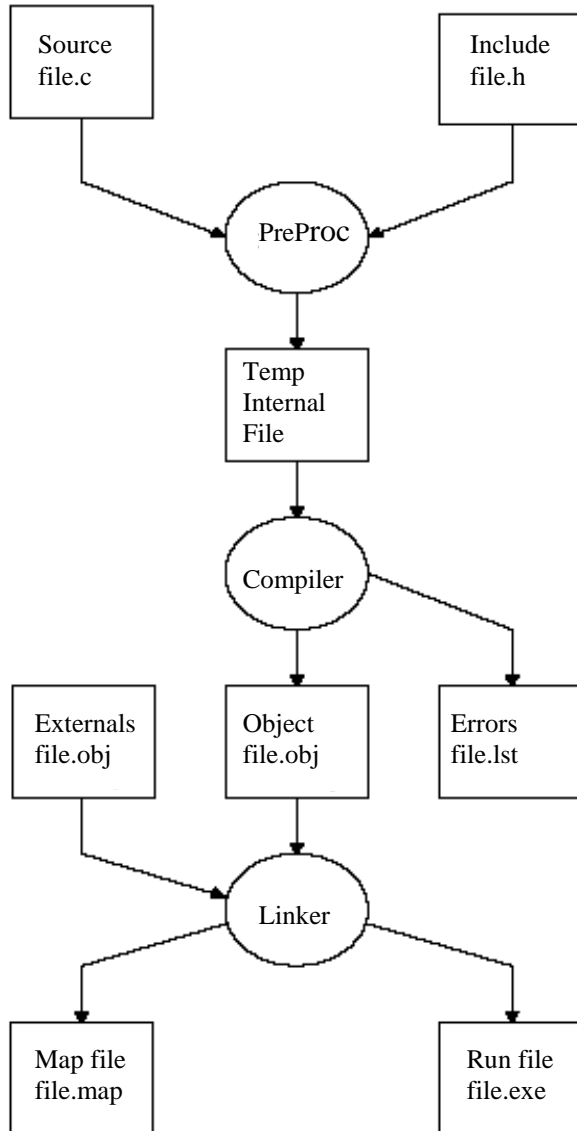
The Compiler is defined as,

“A program that translates an entire batch of source code written in a high-level language into machine language so that it can be executed at a later date.”

The tool known as the 'Compiler' is actually a term used for a collaboration of three components (CRN (2000)). The first of these components is called the 'Pre-Processor', a tool that prepares the source code so that it is ready to be fed into the second component, the 'Compiler'. The compiler processes each line of source code and converts it into equivalent machine code instructions. And

lastly the third component, the 'Linker' takes these machine code instructions and combines them with other pieces of machine code found in computer libraries.

Figure 1.1 shows the processes involved in compilation:



(Figure 1.1, source (Nicol, T (2002)))

Advantages

- **Highly Secure**
As compilers produce low level machine code, it makes the possibility of reverse-engineering very difficult or even impossible. This is one of the criteria that software developers require. Earlier in the paper Trend Micro was looked at and they showed that even compiled software can be broken and copied. As it stands though, compiled software is many times more secure than code that is to be Interpreted. (See 4.1)
- **Extremely Fast Code**

Typically, compilers convert the high level source code into highly efficient code. When it comes to running the application, no CPU (Central Processing Unit) time has to be wasted determining what the original lines of source code were instructing the processor to do, rather the CPU just has to do what it was instructed to do in the original code (Mak, R (1996)).

- **Stand Alone Software Created**

The output from the compiler is a new piece of software that is packaged in a single file. This file can be easily distributed and run on any hardware for which it was compiled.

Disadvantages

- **Portability Issues**

One of the four criteria that developers wanted from their source code was the ability for one version of the code to be compatible on all platforms. Compilation limits the application to a single platform – that being the one that the source code was compiled for (York, B (1997)). The same article discusses ways to improve portability of compiled applications, but in the end it is only improving it. There is no direct way of solving portability issues.

- **Compilation Duration**

This issue was touched on in the Introduction to this paper. Projects are getting bigger and bigger with each year. This paper has already touched on projects that have many millions of lines of code in them. The compiler must examine and process each and every line each time the project is built. It must then link in required code from libraries (Nicol, T (2002)). Even a small project can take several minutes to compile. Larger projects can take hours.

- **Debugging Issues**

Debugging Issues share partly with the time taken to compile a project. If a project took fifteen minutes to compile and a bug is found, another fifteen minutes has to be waited while the project is built again. Tamminen, Eero (2002) shows us that debugging is also hard because unless the language comes with special debugging tools, the developer has relatively little information to help track the bug. Both of these issues relate to time and therefore relate to money. However, Visual C++, (the Microsoft implementation of C++) offers dynamic compilation that allows you to compile sections of code as you are debugging. This goes a long way into reducing compilation time while debugging, but the feature is still absent from languages such as Visual Basic.

- **Large Code**

As compiled applications are stand alone, they each require a copy of all the functionality they use. A small source file (around ~10 Kilobytes) could require functionality from the compiler's library that takes up several megabytes of space. If many applications are on a computer, redundancy becomes a serious issue, with many copies of the same data on one hard disk. DLLs (Dynamic Link Libraries) help to reduce redundancy of code, but many languages do not support them.

4. Comparing Other Methods of Source Code Processing with Compilation

This section examines the main competitors to compilation. The two are Interpretation and an advanced hybrid of interpretation and compilation known as Byte-code. This section will examine their benefits and disadvantages and cross references them with Compilation in order to allow conclusions to be drawn.

4.1. Interpretation

Interpretation is just one of the available replacements to Compilation, and the dictionary defines it as,

“A program that translates an instruction into a machine language and executes it before proceeding to the next instruction.”

As the definition suggests, the Interpreter does not process the entire source code before it is executed, rather it processes each line in turn, and executes the action (Loeliger, R. G. (1981)). No stand alone software is created as with compilers; rather the Interpreter accepts the plain, readable source code and processes it each time it needs to be executed.

Advantages compared to Compilation

- **Extremely Portable**

Many of the popular languages out in the industry today are interpretation based. HTML, Perl, ASP, JavaScript and PHP just a few examples and all extremely popular with regards to websites. (Xiaoli, Z., Wong, H (1996)) showed that there was trend towards Interpretation for World Wide Web applications.

“Generally, it is because they are portable, easy to use, fast to develop and safe. And most interpreted languages are closely related to Web programming.”

Looking back at compiled applications, if they were used for websites, then that would mean that each website would be limited to viewers of a typical hardware type. This would not be ideal for obvious reasons as the Internet is information available to all. Interpretation solves this issue as the source code itself is not converted to machine-language, rather they are emulated by an Interpreter created for each platform.

In the case of the World Wide Web, a Web Browser is an Interpreter of HTML and JavaScript, just as the formatting aspect of a Word Document is interpreted by the compiled software, Microsoft Word.

- **Reduces Redundancy of Code**

As explained in Section 3, many compiled pieces of software have their own copy of library routines that it wishes to use. This means if there were many applications that all used the same functionality at some point within the software, there would be many copies of the same code. This repeated code could be as simple as an “if statement” or as complex as file handling routines.

Interpretation reduces redundancy as it is only the interpreter that contains the library routines that the source code can use. Generally this lowers the amount of disk space required.

Disadvantages compared to Compilation

- **Slow To Execute**

“The only real drawback usually is execution speed. Simple interpreters are often ten to hundred times slower than the same program written in a language such as C or Pascal and executed as a native binary.”

(Riesen, R (1997))

Interpretation techniques tend to be slower than compiled applications because of their “Interpretation Overhead”. Interpretation overhead is the time the Interpreter has to waste, per line, to determine what the line is instructing the interpreter to do (Mak, R (1996)). Compilation is by far more superior when it comes to execution speed as there is no overhead to determine what the code does which makes compilation suitable for real-time systems such as a life support machine or nuclear power plant management software. Interpretation is ideal for software which needs to run on a variety of platforms where speed is not an issue.

- **Easy to debug Interpreted Software**

Lisp is a common language/system in AI that utilises both Compilation and Interpretation to reap their advantages. For the final versions of software, Lisp provides a compiler in order to create fast code, for debugging however, they include an interpreter.

“Interpreters are best for development and debugging because they work with the source code, which programmers understand. Compilers turn nice, readable source code into the gobbledygook of machine language--but that machine language sure runs fast. Most Lisp compilers also give up some run-time checking in order to get more speed.”

(Hartheimer, A)

This is quite a contrast to compilation, which was found to have issues when it came to debugging source code. (See Section 3.2)

- **Source Code is Insecure**

With interpreted languages, the source code remains in a fully visible state when released to the public. For example, with a website, one can click “View Source” in a Web Browser and view other peoples work. This seriously promotes plagiarism. Although there is no immediate fix to this issue, developers tend to include a copyright notice to try and deter “copycats.” In the real world though, this does nothing. The following is an extract from the Sky News web site:

```
<meta name="Copyright" content="2003 BskyB; All Rights Reserved.">
```

(Sky News (2002))

4.2. Byte-code Interpretation

Byte-code Interpretation is a hybrid of Interpretation, designed to limit the “Interpretational Overhead” by pre-processing the source code before it is distributed. This means one can enjoy the

benefits of portability and source code security, yet speeding up the execution of the source code. Byte-code according to (WhatIsByteCode (2001)) is,

“... computer object code that is processed by a program, usually referred to as a virtual machine, rather than by the "real" computer machine, the hardware processor. The virtual machine converts each generalized machine instruction into a specific machine instruction or instructions that this computer's processor will understand.”

The speed of execution and security of byte-code aside, the advantages and disadvantages of byte-code compared to compilation on a whole remain the same as standard interpretation (See Section 4.1 for Advantages/Disadvantages). It is however a step in the right direction. Unfortunately, due to the fact the code is “semi-compiled,” it segregates itself from those coders who may just wish to pick up a text editor and develop simple scripts such as web sites as the pre-processing stage adds another tool, hence another layer of complexity to the development.

5. Conclusion

In order for the statement “Should All Source Code Be Compiled” to be true, either of the following two statements should be proved correct:

1. The method fully satisfies all requirements.
2. The method available is the only method.

Considering Point 2 first, the Compilation technique is not the only method available. As it happens, there are three standard methods of source code processing... Compilation, Interpretation and a hybrid of the two, known as Bytecode. Each however, just like compilation, doesn't satisfy all the requirements of a software developer. Rather they each satisfy a subset of requirements.

One would be correct to cast doubt on whether compilation is suitable for all forms of source code – after all, if it was, surely there would be no alternatives.

Regarding Point 1 last, this paper examined what compilation gave to the software developer. It was determined that compilation produced highly efficient and extremely secure software from the original source code. Compilation however, caused two problems. The first was that the software created was limited to the system for which it was compiled. Often to make the software work on other platforms, developers would have to intensely modify the code and re-compile it for the target system. This would equate to many programming hours and inevitably cost a lot of money to the employer.

The second issue with compilation is that if the source code was large, it would take a long time to compile. This produces similar issues to portability. Many programming hours would be wasted if large software had to be compiled time and again. With strict deadlines to meet, it is easy to see that time is money in the development business.

To satisfy all the requirements, compilation would have to fix both the portability issue and the time-to-compile issue. There is evidence that developers of compilers have seriously noted the time-to-compile issue and have introduced methods to limit compile time. Borland, the makers of the C++ Builder series for example include a “Make” method of compilation,

“Choose Make from the Project Manager Project context menu to compile all files in the current project that have changed since it was last built. This command is similar to Build, except that Make builds only

those files that have changed whereas Build rebuilds all files regardless of whether they have changed or not.”

(C++ Builder (2002))

This is good to see.

Even if companies did however, solve the portability issue, this paper looked at scenarios where machine-code is just not suitable. Two such scenarios were the interpreted languages of HTML and ASP, two of the most common languages used for the Internet. Website genres are extremely vast, ranging from kiddy-fun to seriously-professional. Introducing compilation to these languages could be seen as a step up in complexity which may lead to a World Wide Web that isn't as popular as it is today. Speed of execution is not essential from these languages, but the lack of security is a serious issue.

An idea would be to introduce encryption as standard to interpreters of the code, rather than the code itself, in order to promote languages that are both easy to develop with and secure.

Bytecode is, on paper, a very clever concept but it is in its infancy. It is not yet supported by many of the popular languages, such as C++, Basic and HTML. While it may satisfy all of the criteria of software developers, many developers of interpreters and compilers will not see the need to change over. C++ for example is ideal as a compiled language for fast applications, HTML as an interpreted language for portability. The developers of Java however have seen the benefits of interpreted bytecode and are using them to good effect (Corsaro, C., Schmidt, D. (2002)).

In the end, the answer to the statement “Should All Source Code Be Compiled” is “No.” Not just because it doesn't satisfy all the requirements of developers; but also because it is not ideal compile code in certain situations in an ever expanding computer-oriented society. For those where it isn't ideal, developers have two other choice methods to choose from – it is up to them to decide which best fits their own requirements.

References

Bryson, E (1995)

The World Wide Web: a Web Even a Fly Would Love

Canada-France-Hawaii Telescope Corp

C++ Builder (2002)

Borland C++ Builder Help Documentation

Borland Software Corporation

Casey, C (2000)

Do Software Developers Need The Personal Software Process

University of Central Lancashire

Centre for Visual Arts (2000)

Fight Back With the Facts

University of Toledo

Corsaro, C., Schmidt, D. (2002)

Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems

University of California, Irvine, CA

CRN (2000)

C++ Resources Network

<http://www.cplusplus.com/info/history.html>

Dahlstrand, I (1984)

Software Portability and Standards

Ellis Horwood, Chichester 1984

Dictionary

Available at <http://www.dictionary.com>

Fitzpatrick, R (2003)

Programming Methodologies

University of Texas

Hartheimer, A

Debugging in Lisp

Semantic Microsystems, Inc

<http://www.mactech.com/articles/mactech/Vol.02/02.06/DebugginginLisp/>

Kevin Browne (2000)

'Just Try Our Products'

Microsoft PressPass

Lee, R., Jordan, B. (1996)

New Bottlenecks in LAN-based Imaging Systems

Novell - <http://developer.novell.com/research/appnotes/1996/march/04/apv.htm>

Leitner, F (2001)

Writing Small And Fast Software

<http://www.fefe.de/dietlibc/diet.pdf>

Loeliger, R. G. (1981)

Threaded interpretive languages: their design and implementation.

Peterborough : Byte Pubns., 1981. - 007038360x

Mak, R (1996)

Writing Compilers and Interpreters, 2nd Edition

ISBN: 0-471-11353-0

Mooney, J. (2001)

Software Portability

<http://www.csee.wvu.edu/~jdm/jdm.html>

Moore, G (1965)

Moore's Law

Intel

Nicol, T (2002)
Advanced Programming with C++
University of Central Lancashire

Riesen, R (1997)
UNM Computer Science – Interpretation Techniques
<http://www.cs.unm.edu/~riesen/prop/node37.html>

Risley, D (2001)
Programming Languages
<http://www.pcmech.com/show/internal/105/>

SAP (2002)
A Full Life
<http://www.sap.info/public/en/article.php4/Article-115533cd293947ab9f/en>

Sersguson, A (2002)
Summary Of ‘Protecting Software Code By Guards’
Hoi Chang, Mikhail Atallah

Sim, S. (1995)
The Coming of Software Architecture: A Historical View
University of Toronto

Sky News (2002)
<http://www.sky.com/skynews/home>

Tamminen, Eero (2002)
Debugging Linux applications
http://www.movial.fi/client-data/file/movial_debugging_linux_applications.pdf

Trend Micro, 1997
Susan Orbuch
<http://www.trendmicro.com/en/about/news/pr/archive/1997/pr051497.htm>

WhatIsByteCode (2001)
Definition by WhatIs
http://whatis.techtarget.com/definition/0,,sid9_gci211722,00.html

White, R (2000)
How Computers Work: Millennium Edition
ISBN: 0789721120

York, B (1997)
C and The Portability Issue: #ifdef, or Interface Layers?
Spheringer Technologies Inc.
<http://www.gmonline.demon.co.uk/cscene/CS1/CS1-07.html>

Xiaoli, Z., Wong, H (1996)
A Report On Interpreted Programming languages

<http://www.cs.colorado.edu/~zorn/cs5535/Fall-1996/projects96/zhangx-interpret.html>

References not of web-site origin are papers available from CiteSeer or Emerald Insight.