INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 4.295
(Volume 4, Issue 6)
Available online at: www.ijariit.com

Review on relationship between compiler design and computer architecture

Punit Mishra
punit.mishra2017@vitbhopal.ac.in
VIT Bhopal University, Bhopal, Madhya Pradesh

Ruturaj Sahu
ruturaj.sahu2017@vitbhopal.ac.in
VIT Bhopal University, Bhopal, Madhya Pradesh

Rahul Dev Singh
rahul.devsingh2017@vitbhopal.ac.in
VIT Bhopal University, Bhopal, Madhya Pradesh

P. Sanjeevi
sanjeevi.p@vitbhopal.ac.in
VIT Bhopal University, Bhopal, Madhya Pradesh

ABSTRACT

A compiler is a software structure which achieves the job of compiling and executing High-Level Language codes with the hardware computer architectures present. Previously, applications were often made through assembly language programming or for a specific kernel. Compiler technology and its influence on hardware performance were isolated from the architecture and its performance. However, now the compiler significantly affects the performance of a computer. Hence, understanding compiler technology is critical to the design and efficient implementation of an instruction set. Although the compilers are designed as per the language to be executed upon, their efficient functioning very much depends on the hardware used. A perfect balance between compilers and computer architectures, which varies according to the code to be executed, is the key to fruitful designing of highly efficient and effective computer systems. It is very imperative to study the relations of computer architecture and compiler from embedded microcontrollers to large-scale multiprocessor systems. There are some aspects to which the compiler and the computer architecture must agree, such as regularity, orthogonality, and compensability. So, the construction of a good compiler rests on both compiler design and computer architecture. There is an evident mutual dependency.

Keywords— *Computer Architectures, Compiler design, Compiler optimization, Pipelines*

1. INTRODUCTION

1.1. What is a compiler?

Compilers and operating systems set up the rudimentary interfaces between a programmer and the machine. The compiler is a program which translates high-level programming language into a low-level programming language or source code into machine code. Understanding these relationships simplifies the unavoidable transitions to new hardware and programming languages and develops a person's capability to make a suitable trade-off in design and implementation. Several techniques used to build a compiler are beneficial in a wide range of applications involving symbolic data. The term 'compilation' denotes the transfiguration of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. It is not difficult to see that this transfiguration process from source text to instruction sequence needs considerable effort and obeys complex rules. Programming languages are tools used for the construction of formal descriptions of finite computations (algorithms). Every computation consists of operations that transform a given initial state into a final state [1]. Compilers also perform the job of error detection and only move forward if the code is correct.

Thus, compilers have two jobs-converting High-Level Language into Machine Language and error detection as shown in figure 1.

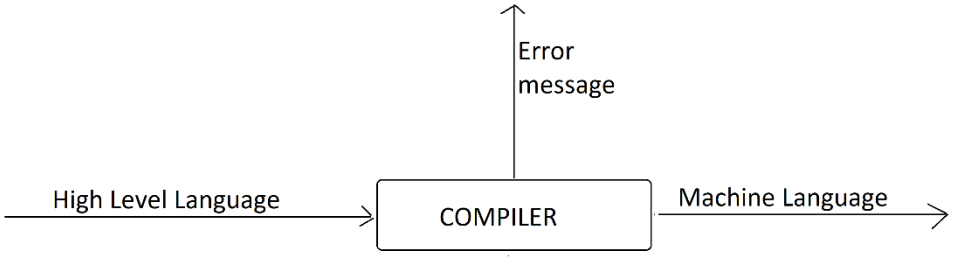

Fig. 1: Role of compiler

1.2. What is computer architecture?

"Computer Architecture is the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals."- WWW Computer Architecture Page [2].

2. PHASES OF COMPILER

The key stages of a compiler as shown in figure 2 are:
(a) Lexical Analysis
(b) Syntax Analysis
(c) Semantic Analysis
(d) Intermediate Code Generation
(e) Code Optimization
(f) Target Code Generation

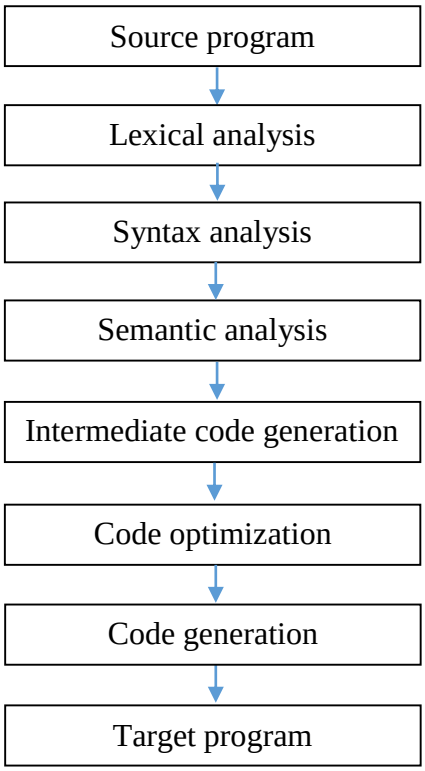

Fig. 2: Phases of Compiler [8]

2.1 Lexical Analysis

The lexical syntax (token structure), which is processed by the lexer and the phrase syntax, is in turn processed by the parser. The lexical syntax is typically a regular language, whose alphabet comprises of the individual characters of the source code text. The phrase syntax is generally a context-free language, whose alphabet consists of the tokens created by the lexer. In computer science technology, lexical analysis is the procedure of converting a sequence of characters into a sequence of tokens which are meaningful character strings. A program or function that accomplishes lexical analysis is called a lexical analyzer, lexer, tokenizer, or scanner (though "scanner" is also used in the first stage of a lexer) [3].

2.2 Syntax Analysis

Syntax analysis is the second phase of the compiler and is also called as parsing. The tokens created during the lexical analysis phase of the compiler are the inputs which are used to generate the tree-like structure of the token list. In the distinguishing structure of the syntax tree, the interior node denotes an operation and the children of the node denote the arguments of the operation. What is important in creating a syntax tree is determining how the leaves are combined to form the tree structure and how the interior node is labeled. Moreover, the parser should recognize invalid texts and reject them by reporting the recognized syntax errors [4].

2.3 Semantic Analysis

Semantic analyzer examines the following from the parse tree entries:
(a) The data type of the first operand
(b) The data type of the second operand
(c) If '+' operator is unary or binary
(d) Checks the number of operands provided to the operator depending on its type (unary or binary)
(e) The data type of identifier sum
(f) If the data type of both LHS and RHS match in the assignment statement [4].

2.4 Intermediate Code Generation

Assembly language makes sure that the computer is able to execute only a limited number of primitive operations on operands with numeric memory addresses all encoded as binary values. Code generation involves translation to machine language instructions or

assembly language in which the assembler appeals to produce the object program. For target machines with several CPU registers, the code generator is blameable for register allocation [5].

2.5 Code Optimization
The IR code which is generated by the translator is analyzed and altered into functionally equivalent but improved IR code by the optimizer. The term 'optimization' is deceptive: we do not always yield the best possible translation of a program, even post optimization by the best of compilers. Why? Some optimizations are impossible to do in every circumstance because they include an unpredictable problem. Eliminating dead code is, in common, impossible.

Further optimizations are too lavish to do in all cases. These include NP-complete problems which are alleged to be inherently exponential. Allocating registers to variables is an example of an NP-complete problem. Optimization can be complex; it may contain numerous sub-phases, which need to be applied more than once. Optimizations may be turned off for speed enhancement. Nevertheless, a well-designed optimizer can significantly enhance the program execution by simplifying, moving and eliminating unwanted computations [6].

2.6 Target Code Generation
In the ultimate phase of the compiler, the translated intermediate language is converted into the target language which forms the native machine language of the system (object code). The making of resource and storage decisions, such as deciding which variables to store into registers and memory and also the assortment and planning of the appropriate machine instructions along with their linked addressing modes is the key purpose of the final phase [7].

3. HISTORY OF COMPILERS
Grace Murray Hopper coined the term compiler in the early 1950s. The translation was back then seen as the "compilation" of a sequence of routines selected from a library. Between 1954 and 1957 a group led by John Backus at IBM developed the first compiler for High-Level Language -FORTRAN. Thus, the viability of high-level and hence less machine dependent languages was proved. The study of the scanning and parsing problems was pursued in the late 60s and 70s that led clearly to a fair solution. It became a standard part of compiler theory and resulted in scanner and parser generators that automate part of compiler development. Research still continues on the development of methods for generating efficient target code, known as optimization techniques. Compiler technology was also applied in rather unexpected areas such as text-formatting languages and hardware description languages for the automatic creation of VLSI circuits [9].

4. THE IMPACT OF COMPILER TECHNOLOGY ON THE ARCHITECT'S DECISIONS
The relations of the compiler and the high-level language majorly depends on which instruction set architecture it works on. An understanding of how the variables are allocated and addressed and the number of registers used for it can be obtained by looking into the ways of how high-level language allocate their data:

Stack– used for local variable allocation which is referred in case of procedure call or return and does not push or pop as it is used only as activation records.

Global data area– for global variables or constants and other data structures as arrays.

Heap– for dynamic objects which do not fit in the stack and they are accessed using pointers. The concept of register allocation that we earlier found as an optimization technique is suitable for stack allocated objects than the other two. For heap objects, it is impossible because a pointer is utilized to access it. Some of the compilers may not apportion register to any local variable if any one of them is addressed by a pointer being afraid of its dependency factors [9].

5. FACTORS AFFECTING COMPILER OPTIMIZATION
5.1. The architecture of the target CPU
5.1.1Number of CPU registers: To a certain limit, the more registers used the easier it is to optimize the performance. Local variables can be allocated in the registers and not in the stack. Temporary or intermediate results can be left in registers without writing to and reading back from memory [10].

5.1.2 RISC vs. CISC:
Properties of CISC
(a) Some simple and very complex instructions
(b) In CISC instructions take more than 1 clock per Cycle to execute
(c) Variable size instructions
(d) No pipelining
(e) Few registers
(f) Not a load and store machine
(g) For Compilation not so good in term of speed
(h) Emphasis of Hardware
(i) Transistors are used for storing complex instructions

Properties of RISC:
(a) Small and simple instructions
(b) In RISC Instructions are executed in one clock cycle per Instructions

(c) All instructions have the same length
(d) Load and Store architecture implemented due to the desired single-cycle operation
(e) Have Pipelining
(f) More register than CISC
(g) Optimal compilation speed as compared to CISC
(h) Emphasis on software
(i) Compared to CISC, RISC spends more transistors on memory registers [11].

Comparison:
(a) CISC is costlier than RISC.
(b) Super-scaling is possible in RISC but not in CISC.
(c) In CISC, the fewer number of registers are used in comparison to RISC.
(d) RISC has fixed format information, whereas CISC has variable format instruction.
(e) CISC has multiple clock cycles, whereas RISC has only one.
(f) Few addressing modes are present in RISC, whereas there are multiple addressing modes in CISC [12].

Table 1: Examples of RISC and CISC Processors [13]

RISC	CISC
MIPS R2000	IBM 370/168
SUN SPARC	VAX 11/780
INTEL i860	MicroVAX II
MOTOROLA 8800	INTEL 80386
PowerPC 601	INTEL 80286
IBM RS/6000	Sun-3/75
MIPS R4000	PDP-11

5.1.3. Pipelines: A pipeline is essentially a CPU fragmented into an assembly line. It allows the use of CPU parts for different instructions by fragmenting the execution of instructions into several stages: instruction decode, address decode, memory fetch, register to fetch, compute, register store, etc. One instruction can be in the register store stage, whereas another can be in the register fetch stage. Pipeline conflicts arise when an instruction in one stage of the pipeline relies on the result of another instruction ahead of it in the pipeline but not yet accomplished. Pipeline conflicts may lead to pipeline stalls: where the CPU rubbishes cycles waiting for a conflict to resolve.

Compilers can *schedule*, or reorder, instructions so that pipeline stalls seldom occur [10].

5.1.4. A number of functional units: Some CPUs have several ALUs and FPUs. This allows them to execute multiple instructions simultaneously. There may be restrictions on which instructions can pair with which other instructions ("pairing" is the simultaneous execution of two or more instructions), and which functional unit can execute which instruction. They also have issues similar to pipeline conflicts.

Here again, instructions have to be scheduled so that the various functional units are fully fed with instructions to execute [10].

5.2 The architecture of the machine:
5.2.1. Cache Size and type: The cache size that is available in current architectures is 256 KB to 12 MB and type is directly mapped, fully associative. Techniques such as inline expansion and loop unrolling necessitate more cache space and may increase the length of code which faces trouble in fitting into the cache memory available. At time due to cache collisions which occur as it is not completely associative will make the available cache memory unproductive at the time of code execution.

5.2.2. Cache/Memory transfer rates: The compiler is intimidated of the issue occurred because of cache collisions or unused cache memory. This may cause efficiency disputes in some specialized applications.

6. WULF's VIEWS

William A. Wulf published a paper on the relationship between Computer architecture and compilers in July 1981 which includes some of the principles that were advanced to architectural changes and simplifies the work of the compiler to provide much efficient object code. In this paper, the author discussed a cost equation evaluating the efficiency and cost factors that are taken into consideration from the perspective of compiler and architecture.

Cost Equation: The author classified the cost based on the software or the compiler design and the hardware or the computer architecture part.

Based on the compiler related costs:

• Designing compilers • Executing the compiler • Executing the compiled program

In the above-listed costs, the designing part is a one-time cost which is high as compared to the hardware cost. The last two costs are hard to be related acquisitively but any flaw in it may result in reduced productivity, unavailable functionality and a fall in reliability.

Based on the computer architecture costs:

• Designing the hardware architecture •Designing the hardware implementation for the architecture • Manufacturing the hardware

In the above-listed costs, the duplication of the designed implementation or in other words the manufacturing of the hardware is the only cost that has been dropping over years. The designing cost is always comparatively more like the critical part of the entire architecture is the designing and the one-time work; if any irregularity occurs then it keeps reiterating as the implementation is only replicated from the initially designed faulty architecture which would turn costly if not corrected at the design level **[14].**

6.1. General principles

6.1.1. Regularity: If something is done in one way in one place, it ought to be done the same way everywhere. This principle is known as the "law of least astonishment" in the language of the design community.

6.1.2. Orthogonality: It should be feasible to divide the machine definition (or language definition) into a set of separate concerns and define each in segregation from the others. For example, it should be possible to discuss data types, addressing, and instruction sets independently.

6.1.3. Compensability: If the principles of regularity and orthogonality have been followed, then it should also be possible to create the orthogonality and regular notions in arbitrary ways. It should be possible, for example, to use every addressing mode with every operator and every data type [15].

6.2. Other principles:

6.2.1. One vs. all: There must be precisely one way to do something, and/or all ways should be possible.

6.2.2. Provide primitives and not solutions: It is far better to provide good primitives from which solutions of code generation problems can be produced than to provide the solutions themselves.

7. STORAGE MANAGEMENT

7.1. Static storage management

Storage management is commonly used in the cloud [19-28] and static storage management means the compiler can provide fixed addresses for all objects at the time of program translation (here we assume that translation includes binding). Arrays with dynamic bounds follow recursive procedures and the use of anonymous objects are barred. The condition is satisfied for languages like FORTRAN and BASIC, and for the objects in the outermost contour of an ALGOL 60 or Pascal program. (In contrast, arrays with dynamic bounds can occur even in the outer block of an ALGOL 68 program.)If the storage for the elements of an array with dynamic bounds is managed distinctly, the condition can be enforced to hold in this case too [16].

7.2. Dynamic storage management using a stack

Every declared value in languages such as Pascal and SIMULA have restricted lifetimes. Moreover, the environments in these languages are nested: The extent of all objects fitting in the contour of a block or procedure finishes before that of objects from the dynamically enclosing contour. Thus, we can use a stack discipline to manage these objects: Upon procedure call or block entry, the activation record comprising storage for the local objects of the procedure or block is pushed into the stack. At the block end, procedure return or a jump out of these builds the activation record that is popped out of the stack. (The entire activation record is stacked, we do not deal with single objects individually!)An object of automatic extent occupies storage in the activation record of the syntactic construct with which it is associated. The position of the object is given by the base address, b, of the activation record and the relative location offset, R, of its storage within the activation record. R must be known at compile time but b cannot be known (otherwise we would have static storage allocation). To access the object, b should be determined at runtime and positioned in a register. R is then added to the register and the result is used as an indirect address, or R is seen as the constant in a direct access function of the form 'register + constant'. The extension, which may differ in size from activation to activation, is often called the second-order storage of the activation record. Storage within the extension is constantly accessed indirectly via information held in the static part; in fact, the static part of an object may solely consist of a pointer to the dynamic part [17].

7.3. Dynamic storage management using a heap

The last option is to allocate storage on a heap: The objects are allotted storage arbitrarily within an area of memory. Their addresses are determined at the time of allocation, and they can only be accessed indirectly. Examples of objects demanding heap storage are anonymous objects such as those created by the Pascal new function and objects whose size changes arbitrarily during their lifetime. (Linked lists and the flexible arrays of ALGOL 68 belong to the latter class). The use of a stack storage discipline is not necessary, but simply provides a suitable mechanism for reclaiming storage when a contour is no longer significant. By storing the activation records on a heap, we expand the possibilities for specifying the lifetimes of objects. Storage for an activation record is done by analyzing and understanding all the provided review comments carefully. Now make the obligatory amendments in your paper. If you are not assured about any review comment, then don't forget to clarify that comment. And in some cases, there could be chances where your paper receives a number of critical remarks. In that case, don't get depressed and try to improvise the maximum. Heap allocation is mostly simple if all objects required during execution can ‘tin to the designated area at the same time. In most cases, however, this is not possible. Either the area is not large enough or, in the case of virtual storage, the working set becomes too large. We should only sketch three possible recycling strategies for storage and specify the support requirements placed upon the compiler by these strategies [18].

8. CONCLUSION

The compiler is a program structure which does the task of translating high-level programming language into a low-level programming language or source code into machine code. The key phases of compiler design are Lexical Analysis, Syntax Analysis,

Semantic Analysis, Intermediate Code Generation, Code Optimization, and Target Code Generation. The architecture of the Machine and Architecture of the Target CPU affect compiler optimization. It is dependent on a number of functional units, pipelines, cache type, and size etc.

9. REFERENCES

[1] Jatin Chhabra, Hiteshi Chopra, Abhimanyu Vats, 'Research paper on Compiler Design', 'INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY', Volume 1, Issue 5, pp. 151, 2014.

[2] https://www.cis.upenn.edu/~milom/cis501-Fall11/lectures/00_intro.pdf

[3] Mahak Jain, Nidhi Sehrawat, Neha Munsi, 'COMPILER BASIC DESIGN AND CONSTRUCTION', IJCSMC, Vol. 3, Issue. 10, pp.850 – 852, 2014.

[4] Neha Pathapati1, Niharika W. M.2, Lakshmishree .C, 'Introduction to Compilers', 'International Journal of Science and Research (IJSR)', Volume 4, Issue 4, pp. 2400, 2015.

[5] Ch. Raju, Thirupathi Marupaka, Arvind Tudigani, 'Analysis of Parsing Techniques & Survey on Compiler Applications', Vol. 2, Issue. 10, pg.115 – 125, 2013.

[6] http://pages.cs.wisc.edu/~fischer/cs536.s08/lectures/Lecture04.4up.pdf

[7] www.cse.aucegypt.edu/~rafea/csce447/slides/intro.pdf

[8] Vishal Trivedi, 'Life Cycle of Source Program - Compiler Design', 'International Journal of Innovative Research in Computer and Communication Engineering', Vol. 5, Issue 12, 2017

[9] Jasveen Kaur, Amanpreet Kaur, 'Role of Compilers in Computer Architecture', 'International Journal of Engineering Research and General Science', Volume 4, Issue 3, pp 250-251, 2016.

[10] https://en.wikipedia.org/wiki/Optimizing_compiler

[11] Shahzeb, Naveed Hussain, Amanullah, Furqan Ahmad, Salman Khan, 'Comparative Study of RISC AND CISC Architectures', 'International Journal of Computer Applications Technology and Research', Volume 5, issue 7, pp. 500, 2016

[12] Kirti Tokas, Dhruv Sharma, Lokesh Yadav, 'RISC AND CISC ARCHITECTURE', 'INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY', Volume 1 Issue 6, pp. 2096, 2014.

[13] Hasan Krad, Aws Yousif Al-Taie, 'A New Trend for CISC and RISC Architectures', 'Asian Journal of Information Technology', Volume: 6, Issue: 11, Page No. 1125-1131, 2007

[14] Telkapalli Murali Krishna, Sreedhar Appalabatla, 'Mutual Interdependency between Compiler and Computer Architecture: An Overview', 'INTERNATIONAL JOURNAL OF SCIENTIFIC RESEARCH', Vol. 4, Issue no. 1, 2015

[15] William A. Wulf, 'Compilers and Computer Architecture', 'IEEE', Volume 14, Issue 7, pp. 41-47, 1981

[16] Jatin Chhabra, Hiteshi Chopra, Abhimanyu Vats, 'Research paper on Compiler Design', 'INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY', Volume 1, Issue 5, pp. 151-153, 2014.

[17] Aastha Singh, Sonam Sinha, Archana Priyadarshi, 'Compiler Construction', 'International Journal of Scientific and Research Publications', Volume 3, Issue 4, pp. 1-6, 2013.

[18] Ms.Snehal H. Chaflekar, Ms. Ashwini Lokhande, Ms. Priyanka Gomase, Ms. Rupali Shinganjude, 'Compiler Architecture and Design Issues', 'International Journal of Computer Science Trends and Technology', Volume 5 Issue 3, pp. 13-16, 2017

[19] P. Sanjeevi and P. Viswanathan, 'NUTS scheduling approach for cloud data centers to optimize energy consumption', Computing (Springer), Vol. 99, No. 12, pp. 1179-1205, 2017.

[20] P. Sanjeevi and P. Viswanathan, 'Workload Consolidation Techniques to Optimize Energy in Cloud: Review', Int. J. of Internet Protocol Technology, Vol. 10, No. 2, pp. 115–125, 2017.

[21] P. Sanjeevi and P. Viswanathan, 'DTCF: Deadline Task Consolidation First for energy minimization in cloud data centers', International Journal of Networking and Virtual Organizations, Inderscience, Vol. 19, No. 3, pp. 209–233, 2017.

[22] P. Sanjeevi and P. Viswanathan, 'Employing Smart Homes IoT Techniques for Dynamic Provision of Cloud Benefactors', Int. J. of Critical Computer-Based Systems, Inderscience, Vol. 7, No. 3, pp. 209–224, 2017.

[23] P. Sanjeevi and P. Viswanathan, 'A survey on various problems and techniques for optimizing energy efficiency in cloud architecture', Walailak Journal of Science and Technology, Vol. 14, No. 10, 2017.

[24] P. Sanjeevi, P. Viswanathan, M. R. Babu, and P. V. Krishna, 'Study and Analysis of Energy Issues in Cloud Computing', International Journal of Applied Engineering Research, Vol. 10, No. 7, pp. 16961-16969, 2015.

[25] P. Sanjeevi, G. Balamurugan, and P. Viswanathan, 'The Improved DROP Security based on Hard AI Problem in Cloud', Int. J. of Internet Protocol Technology, Vol. 9, No. 4, pp. 207-217, 2017.

[26] P. Sanjeevi and P. Viswanathan, 'A green energy optimized scheduling algorithm for cloud data centers', IEEE International Conference on Computing and Network Communications, Trivandrum, pp. 941-945, 2015.

[27] P. Sanjeevi and P. Viswanathan, 'Towards energy-aware job consolidation scheduling in the cloud', International Conference on Inventive Computation Technologies (ICICT 2016), IEEE Xplore, pp. 361- 366, 2016.

[28] G. Kesavan, P. Sanjeevi and P. Viswanathan, 'A 24 hour IoT framework for monitoring and managing home automation', International Conference on Inventive Computation Technologies (ICICT 2016), IEEE Xplore, pp. 367-371, 2016.

