Research Paper

Volume : 4 | Issue : 1 | January 2015 • ISSN No 2277 - 8179

Mutual Interdependency between Compiler and Computer Architecture: An Overview

Computer Science

KEYWORDS : Compiler Design, Computer Architecture, Programming Language, Software, Hardware

Dr.Telkapalli Murali Krishna	Asst.Professor, Dept of CS&IT, College of Natural & Computational Sciences, Wolaita Sodo University, Ethiopia.
Dr.Sreedhar Appalabatla	Asst. Professor, Dept of CS&IT, College of Natural & Computational Sciences, Wolaita Sodo University, Ethiopia

ABSTRACT

A compiler is a software layer that helps the high level executions that are made in a programming language to be compiled and implemented by the underlying hardware computer architecture. Though a compiler is majorly designed based on the language specification, the hardware that it is going to implement on has a significant role in the compiler design. The factors in which the compiler and the computer architecture have to agree on are: regularity, orthogonality and compensability. Any issue with these factors result in a tougher task for the compiler designer as the more perfect the compilation is and less efficient it happens to result; as it would require extensive implementations in the computer architecture. In case of platform independent languages, the task is more demanding as it has to comply with the different sets of hardware, allows the programmer to utilize the full potential of the computer architecture and at the same time make the compilation part efficient. So, making an efficient compiler involves the collaborative work of both a compiler designer and the hardware architects who built the computer architecture.

INTRODUCTION

Today almost all programming is done in high-level languages for desktop and server applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times for these applications, architectural decisions were often made to ease assembly language programming or for a specific kernel. Because the compiler will significantly affect the performance of a computer, now a days, understanding compiler technology is critical to design and efficient implementation of an instruction set. Once it was popular to try to isolate the compiler technology and its effect on hardware performance from the architecture and its performance, just as it was popular to try to separate architecture from its implementation. This separation is essentially impossible with today's desktop compilers and computers. Architectural choices affect the quality of the code that can be generated for a computer and the complexity of building a good compiler for it, for better or for worse.

In this section, we discuss the critical goals in the instruction set primarily from the compiler viewpoint. It starts with a review of the anatomy of current compilers. Next we discuss how compiler technology affects the decisions of the architect, and how the architect can make it hard or easy for the compiler to produce good code. We conclude with a review of compilers and multimedia operations, which unfortunately is a bad example of cooperation between compiler writers and architects.

This paper focuses on listing the structure of a compiler and the underlying architecture, the mutual dependency between them and how the efficiency can be increased by several factors that are modified in the compiler level and also in the architectural level. Next we discuss what is compiler, hardware compilation, impact of optimizations on performance, the Impact of Compiler Technology on the Architect's Decisions, Factors in architecture that affect the compiler optimization, Wulf's views, how architect can help compiler writer and conclusion.

COMPILER:

The most known definition of compiler is that it translates the high level language into machine understandable language. In basis of computer architecture, a compiler translates the behavioral level to the structural level. The structure of the compiler can be classified as front end and backend in which the last one is more dependent on the architectural framework of the system. The front end consists of the Lexical Analyzer, Syntax Analyzer and Semantic Analyzer.

Lexical Analyzer: It isolates each part of the statement and tokenizes them as operands, operator, variable, constants etc. The lexical analysis phase reads the characters in the program and groups them into tokens that are sequence of characters having a collective meaning.

Syntax Analyzer: It parses the token sequence and identifies the syntactic structure of the program.

Semantic Analyzer: This phase of the compiler checks for type errors and adds all the necessary semantic information to the parse tree.

The back end consists of the Intermediate Code Generator, Code Optimizer and Code Generator.

Intermediate Code Generator: This phase of compiler transforms parser tree into an intermediate language representation of the source program. Intermediate codes are machine independent codes, but they are close to machine instructions.

Code Optimizer: Code optimization is the process of modifying the working code to a more optimal code based on a particular goal. The code optimization phase attempts to improve the intermediate code, so that faster running machine code will result.

Code Generator: It takes the optimized intermediate representation of the source program as input and produces a target program or object program as its output. The final phase of the compiler is the generation of the target code or machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then intermediate instructions are translated into a sequence of machine instructions that perform the same task.

The back end performs the intermediate code generation, code optimization and generation which are very significant parts in the compilation process.

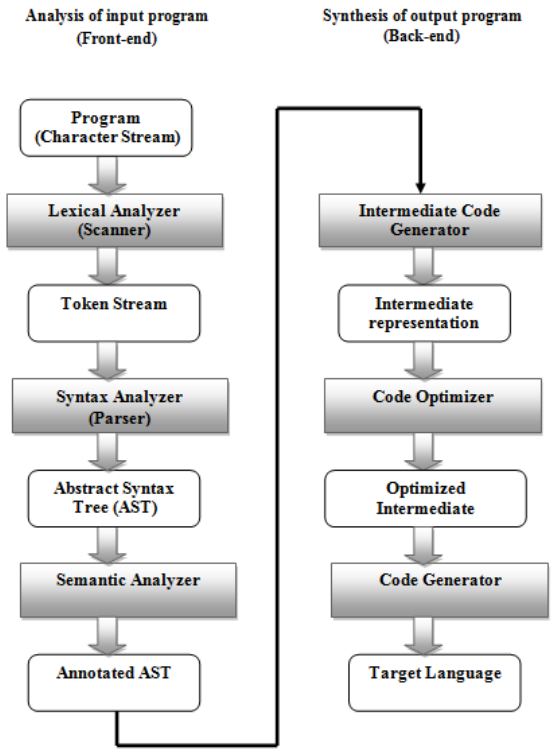

Figure 1: The Complete Structure of a compiler.

The Structure of Recent Compilers:

To begin, let's look at what optimizing compilers are like today. The figure below shows the structure of recent compilers.

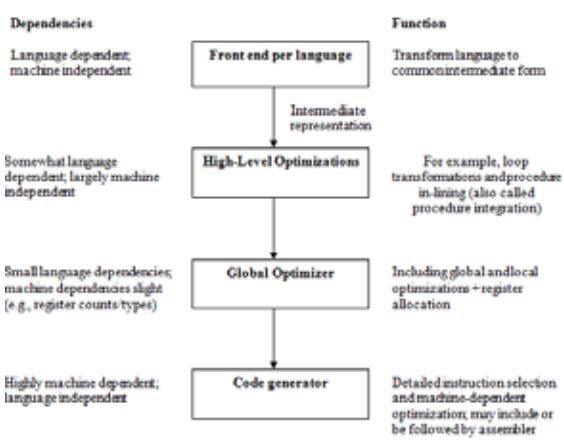

Figure 2: Structure of recent compiler

In this figure compilers typically consist of two to four passes or phrases, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A pass is simply one phase in which the compiler reads and transforms the entire program. Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes (phrases) in the compiler transform higher-level, more abstract representations into progressively lower-level representations. Eventually it reaches the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier.

The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler in Figure 2 above, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. Such iteration would be prohibitive, both in compilation time and in complexity. Thus, compilers make assumptions about the ability of later steps to deal with certain problems. For example, compilers usually have to choose which procedure calls to expand inline before they know the exact size of the procedure being called. Compiler writers call this problem the ***phase-ordering problem***.

How does this ordering of transformations interact with the instruction set architecture? A good example occurs with the optimization called ***global common sub-expression elimination***. This optimization finds two instances of an expression that compute the same value and saves the value of the first computation in a temporary. It then uses the temporary value, eliminating the second computation of the common expression.

For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading it may negate the savings gained by not recomputing the expression. There are, in fact, cases where this optimization actually slows down code when the temporary is not register allocated. Phase ordering complicates this problem because register allocation is typically done near the end of the global optimization pass, just before code generation. Thus, an optimizer that performs this optimization must assume that the register allocator will allocate the temporary to a register.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

High-level optimizations *are often done on the source with output fed to later optimization passes.*

Local optimizations *optimize code only within a straight-line code fragment (called a basic block by compiler people).*

Global optimizations *extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.*

Register allocation *associates registers with operands.*

Processor-dependent optimizations *attempt to take advantage of specific architectural knowledge.*

HARDWARE COMPILATION:

Some compilers are called the hardware compilers or synthesis tools as their output aims at a very low level where the effectively compiled source code controls the final configuration of the hardware which is at the logic (gate) level. The output of the compiled

Volume : 4 | Issue : 1 | January 2015 • ISSN No 2277 - 8179

code does not aim at executing instructions in sequence but observes the interconnection of transistors or lookup tables.

Examples: Field Programmable Gate Array (FPGA) or Structured Application Specific Integrated Circuit (ASIC) which uses a synthesis tool XST (Xilinx Synthesis Tool). Similar tools available from other vendors are Altera, Synplicity, and Synopsys.

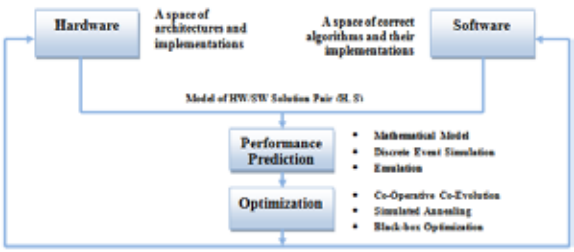

Figure 3: Co-design of Hardware/Software compilation framework

IMPACT OF OPTIMIZATIONS ON PERFORMANCE:

The unoptimized code is relatively much bigger and would involve a lot of hardware implementation for a simple process. It would increase the execution time and thus reduce the efficiency. The object code produced by a compiler using straightforward compiling techniques is often not very efficient. In many cases, the object code can be made to run faster or take up less space through program transformations known as optimizations. Compilers that improve performance through code transformations are known as optimizing compilers.

In an unoptimized program, there is generally a one-to-one correspondence between a source statement and a group of one or more machine instructions. Optimizing compilers must preserve the correctness of a program, but after optimization, this one-to-one correspondence no longer exists in many cases, and the program may no longer execute in the order implied by the source code. This complicates debugging. Programmers must often resort to debugging assembly code to figure out how an optimized program is behaving. In general, users should debug the unoptimized version of a program before using the optimizer.

Some of the code optimization techniques are:

- Algebraic simplification
- Common sub-expression elimination
- Copy propagation
- Constant propagation
- Dead-code elimination
- Inline expansion
- Loop transformation

After the front end process is completed, when the resulting intermediate representation need to be optimized, there are two other major optimization techniques used as the register allocation and instruction scheduling which are more dependent on the architecture in which the optimized code is to run on. The Figure 4 listed below shows the unoptimized and optimized code of a simple program to compute d = 4*n*n*(n+1)*(n+1) and return the same. The efficiency of implementation of the unoptimized code in the hardware in comparison to the optimized code is so low.

Figure 4: Code optimization

Unoptimized Code

```
 expr:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %edx
        imull   -4(%rbp), %edx
        movl    -4(%rbp), %eax
        incl    %eax
        imull   %eax, %edx
        movl    -4(%rbp), %eax
        incl    %eax
        imull   %edx, %eax
        sall    $2, %eax
        movl    %eax, -8(%rbp)
        movl    -8(%rbp), %eax,
        leave
        ret
```

Optimized Code

```
    expr:
.LFB2:
        movl    %edi, %eax
        imull   %edi, %eax
        incl    %edi
        imull   %edi, %eax
        imull   %edi, %eax
        sall    $2, %eax
        ret
```

The common optimization methods are analyzed on basis of their relative frequencies of occurrence in Table 1. The optimizations are classified on basis of high-level, local, global and Machine-dependent where N.M denotes the number of occurrences of those optimizations was not recorded. These optimizations were applied on a small set of 12 Pascal and FORTRAN programs in 1983 using Stanford UCODE compiler.

Table 1: Major types of optimizations and examples

Optimization Name	Explanation	Percentage of the total number of optimizing transforms
High-level	At or near the source level; machine-independent	
Procedure Integration	Replace procedure call by procedure body	N.M.
Local	Within straight-line code	
Common sub expression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	Across a branch	
Global common sub expression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., A=X) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of loop	16%
Induction variable elimination	Simplify/elimination array-addressing calculations within loops	2%
Machine-dependent	Depends on machine knowledge	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

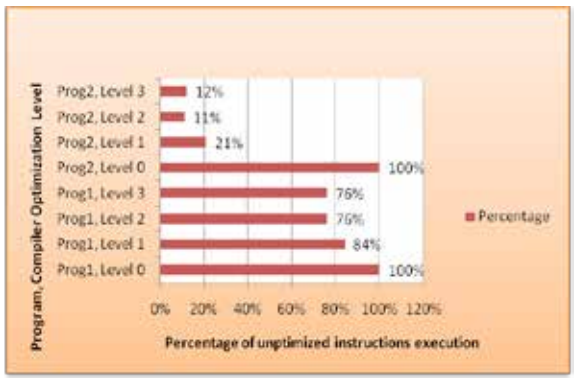

Figure 5: The effect of optimization on program execution

Figure 5 shows that the effect of various optimizations on instructions executed for two programs (i.e. Prog1 and Prog2). In this case, optimized programs executed roughly 25% to 90% fewer instructions than unoptimized programs. The figure illustrates the importance of looking at optimized code before suggesting new instruction set features, since a compiler might completely remove the instructions the architect was trying to improve. The two programs vary as compiler optimization levels vary. Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (software pipelining), and global register allocation. Level 3 adds procedure integration.

THE IMPACT OF COMPILER TECHNOLOGY ON THE ARCHITECT'S DECISIONS:

The interaction of the compiler and the high level language majorly depends on what instruction set architecture it works on. To have an understanding of how the variables are allocated and addressed and the number of registers used for it can be obtained on looking into the ways how high level language allocate their data:

Stack – used for local variable allocation which is referenced 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 stack and they are accessed using pointers.

The concept of register allocation that we earlier found as optimization technique is suitable for stack allocated objects than the other two. For heap objects it is impossible as a pointer is used to access it. Some compilers may not allocate register to any local variable if any one of them is addressed by a pointer fearing for its dependency factors.

FACTORS IN ARCHITECTURE THAT AFFECT THE COMPILER OPTIMIZATION

Generally the major factor affecting the compiler optimization is the machine on which the code is to be executed because of which few compilers like *gcc* machine description parameters that can be altered have based on the machine in which the compiled code is to be executed.

Number of CPU Registers

The more the number of registers, it makes it easier for the compiler to allocate registers. The local variables will be allocated registers instead of a stack data representation and other intermediate values can also be stored in registers. Here, it is to be noted that RISC uses the multiple register set as any instructions has to be from/to the internal registers but the CISC uses only a single register set as it can address operands directly from the memory bypassing the registers.

RISC vs. CISC

The CISC (Complex Instruction Set Computers) has a 200 to 250 instructions and has a variable length instructions and addressing modes because of which ends up providing a large case analysis for the instructions to be chosen. Even for simple function it provides several choices and the decision making part for choosing the best instruction out of it makes the compilation speed to slow down. But in RISC (Reduced Instruction Set Computers) which typically has only 20 to 30 most commonly used instructions has a fixed length of instructions and also only specific addressing modes. Any instruction has to pass through the internal registers unlike CISC which references directly to the memory for operands. The compiler has relative cost criteria based on which it chooses the necessary ISA (Instruction Set Architecture).

Pipelines

An instruction can have several execution stages in code generation in which it can be of stages: instruction decode, address decode, memory fetch, register fetch, compute, register store etc. Pipelining comes into picture when one instruction is in memory fetch stage and the other instruction is in register store stage and that they happen to be inter-dependent similar to a deadlock condition. Pipeline conflicts can lead to pipeline stalls which can be reduced by scheduling or re-ordering the instructions.

Number of Functional Units

Some of the architectures have several Arithmetic and Logical Unit (ALU) and Floating Point Unit (FPU) which process certain instructions from a program simultaneously. Analogous to the pipeline conflicts the instructions may be inter-dependent which can also be eliminated by proper instruction scheduling to different units of the CPU.

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 like inline expansion and loop unrolling require more cache space and may increase the length of code which may have trouble in fitting into the cache memory available. At time due to cache collisions which occur as it is not fully associative will make the available cache memory unproductive at the code execution time.

Cache/Memory transfer rates.

The compiler is indicated of the issue occurred because of cache collisions or unusability of the cache memory. This may cause efficiency issues in some specialized applications.

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 evolved to architectural changes that can simplify the work of the compiler and provide much efficient object code. In this paper, the author discusses a cost equation evaluating the efficiency and cost factors that are taken to consideration from the perspective of the compiler and the architecture.

6.1 Cost Equation

The author classifies the cost based on the software or the compiler design and the hardware i.e. 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 in comparison to the hardware cost. The last two costs are difficult to be related acquisitively but any flaw with it may result in decreased productivity, unavailable functionality and a fall in reliability.

Cost Equation
The author classifies the cost based on the software or the compiler design and the hardware i.e. 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 in comparison to the hardware cost. The last two costs are difficult to be related acquisitively but any flaw with it may result in decreased 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 replication of the designed implementation or in other words the manufacturing of the hardware is the only cost that has been reducing over years. The designing cost is always more as the critical part of the whole architecture is the designing and the one-time work if any irregular structure occurs then it keeps repeating as the implementation is only replicated from the initially designed flawed architecture which would turn expensive if not corrected at design level.

General Principles
Regularity
The principle of regularity states that any process if done in one way in one part of the program, then it should be done in the same way whenever the process is done. This is called "the law of least astonishment" in the computer design community.

The author applauds that regularity is used in most of the architectures as they accept several different data types and the ability of the compiler to treat the source and destination same way just like how the memory and registers can be treated alike. The irregularities that he complains are about the arithmetic shift and the "immediate mode" arithmetic.

Orthogonality
The feasibility to do a clear lexical analyzing on the machine code i.e., any instruction should provide a clear view of the function, the operands or constants involved in the process. Some machines provide different instruction sets to memory-to-memory, register-to-memory, and register-to-register operations which offers a large case analysis for the compilers to choose from. At times the instruction to be chosen also depends on the addressing mode chosen by the instruction.

Compensability
The proper usage of the regularity and orthogonality without any deviation will automatically provide compensability i.e., the ability to compose the orthogonal and regular notions in arbitrary ways.

The author identifies the root of the problem as the programming languages views data type as a property of data whereas the machine language views data type as a property of operator. Due to this problem, any data type cannot be used from any addressing mode which is viewed as a violation of the composability principle.

Other Principles
One vs. All
Any instruction must be provided with one possible way to do it or all possible ways to do it. For example, if the instruction may appear in 6 possible scenarios, the instruction set must provide one solution for all or 6 possible solutions which will reduce the work of the compiler by just routing it to the specific case or general case. Instead if there exist 4 instruction sets for the instruction, the compiler will have a tough time deciding which scenario the instruction has to be put in. Thus, either one or all will simplify the work of the compiler.

Provide primitives, not solutions
The author states that it is better to provide primitive conclusions from which the programs can synthesis its own solution. The complicated functions as procedure call or case statements the instruction set provided seem to suit few languages well but end up giving a semantic clash with other languages. This is because of providing too much of semantic content to the instruction because of which the compiler can only use those instructions in specific contexts.

Addressing
The addressing computations are paths and are not specific to array or other data structures. The referencing from a memory or through pointers appears more complex than it can be which reduces the scope and context in which they can be used.

Environment Support
The handling of run-time environment issues is the environment support expected from the architecture from the compiler's view. Some run-time environment issues are stack frames, displays or static/dynamic links, exceptions, processes etc.

The common run-time issues that need support are: (a) Uninitialized variables (b) Constraint checks (c) Exceptions

(d) Debugging support

Deviations
Only in cases where the instructions are implementation independent, the deviation of these principles should occur.

HOW ARCHITECT CAN HELP COMPILER WRITER?

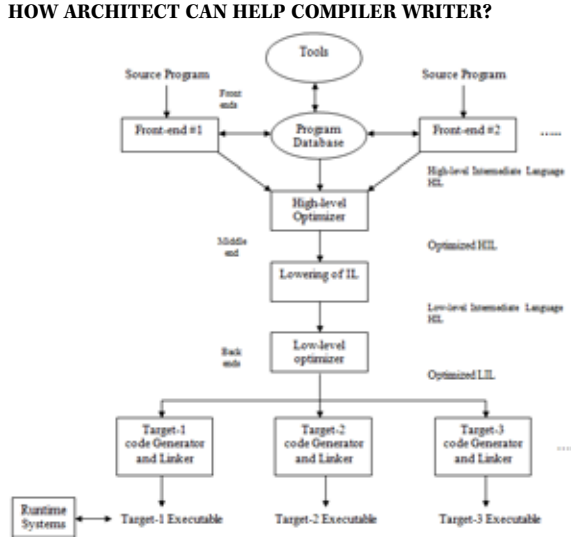

Figure 6: Structure of Optimizing Compilers

Today, the complexity of a compiler does not come from translating simple statements like A = B + C. Most programs work fine for simple translations. The complexity arises with large programs and globally complex in their interactions. The structure of compilers chooses the best instruction for the given statement in a small fraction of time.

Compiler writers usually work based on a goal where the quality is highly tested for the most frequent instructions and the accuracy is aimed at for the infrequent and complicated instructions. They consider the simple instructions to be available fast and the complex instructions though slow should give accurate results.

Some instruction set properties help the compiler writer and makes it easier to write a compiler that will generate efficient and correct code.

Providing regularity—it is important to remember that in any instruction set, the influencing factors are the operation, the data type used for the variables and the addressing mode used to call the variable must be orthogonal. In this context orthogonality more about not being inter-dependent on each other i.e., irrespective of the chosen data type any addressing mode can be used should be the condition. For example, the addressing modes and operations are orthogonal if for every operation all addressing modes are applicable. On following this type of regularity, the compiler's code generation will be simpler and much less complex it need not choose between two different passes for an instruction. One evidence of irregularity is when a specific operation demands for only specific registers. For example, in case of taking D= (A+B)*C in which the addition operation need not have to look out for an even register and on register allocation it may end up with an even or odd one but the multiplication process which chooses an even register will have to make an extra data to move to compute the expression. This would result in having available registers but not the one needed (in this case even register) for register specific applications.

Provide primitives, not solutions—the generalization of a high level language functionality often fails results in supporting one high level language well and creating a semantic clash in the other languages. So it is much better not to add the special features in the instruction sets so that the features in the architecture are much generalized for use of all languages and can be synthesizing its own necessary solution from the given set of instruction sets.

Simplify trade-offs among alternatives—the biggest challenge for current compilers is dealing with a large case analysis. When several suggestions are made for a specific statement the compiler can choose the simple instruction set and implement the goal of the statement but the decision making time itself is more to choose which is the best and efficient instruction set of those available which consumes most of the compilation time. With pipelining and cache memory too it is difficult to handle the code size that is generated after compilation which was comparatively much less earlier. It is more important for the compiler writer to understand the alternative code sequences suggested and then choosing the best instruction set for the statement. The register-memory architecture is the most difficult tradeoff in deciding how many times a variable should be referenced before it is cheaper to load it into a register. This threshold point is very hard to compute and, in fact, may vary among models of the same architecture.

The architecture provides instructions that bind the quantities known at compile time as constants. Not all variables that have a value at the compiler time can act the same way in run-time, and this case when considered vice-versa by the architecture makes the compiler writer's job more complex. Good counter examples of this principle include instructions that interpret values that were fixed at compile time. For instance, the VAX procedure call instruction (calls) dynamically interprets a mask saying what registers to save on a call, but the mask is fixed at compile time.

CONCLUSION

An effective compiler allows a more efficient execution of application programs for a given computer architecture, while well-conceived architectural feature can support more effective, compiler optimization techniques. Both the compiler writer and machine designer have multiple objectives.When the compiler writer and the machine designer work on their objectives also having in mind the objectives of the other then it is possible achieve what they need to accomplish and also what can be accomplished using the feature or optimization built by them. The compiler writer will benefit by having more space to research on the optimization techniques of the compiler and the machine designer will profit on the reduced execution time it takes to implement because of the well structured architecture that complies with the needs of a compiler writer and also achieves its primary goals. From embedded micro-controllers to large-scale multiprocessor systems, it is important to understand the interaction between compilers and computer architectures. A machine designer should upgrade his architecture in a way that helps implement the high-level languages. But the up-gradation must take into consideration cost, reliability, and customer acceptance factors also. It is of no use if it is very high level language oriented as it fails to generalize the concept of how the same is dealt in other languages. Similarly, a compiler writer should apply the semantics that are provided in the source code and translate the same to the code that can be interpreted by the machine for hardware implementation and also provide a user-friendly interface that doesn't make the work of the programmer complex. Correctness and efficiency is the major factors that are expected from an efficient compiler. To achieve both it needs a proper interaction with the computer architecture and an understanding of how things need to be conceived which would result in efficient and accurate compilers.

REFERENCE

[1] William A. Wulf IEEE 0018-9162 Compilers and Computer Architecture. | [2] John Hennessey and David Patterson, Computer Architecture - A Quantitative Approach (fifth edition). | [3] Torben Egidius Mogense, Basics of Compiler Design. | [4] John. J. Donavan, Systems Programming | [5] Gyungho Lee and Pen-Chung Yew, Interaction between Compilers and Computer Architectures. | [6] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-sma-5502-fall-2005/lecture-notes/14_wrapup.pdf. | [7] http://en.wikipedia.org/wiki/Compiler#Front_end. | [8] Kurt Keutzer , Wayne Wolf Anatomy of a Hardware Compiler AT&T Bell Laboratories Murray Hill NJ, 1988 ACM O-8979 l-269- l/88/0006/0095. | [9] http://en.wikipedia.org/wiki/Optimizing_compiler#Specific_techniques. | [10] https://www.inkling.com/read/computer-architecture-hennessy-5th/appendix-a/section-a-8. | http://www.tkt.cs.tut.fi/tools/public/tutorials/synopsys/design_compiler/gsdc.html#arch_opt. | [11] M. Morris Mano, Computer Systems Architecture | [12] Carl Hamacher, Computer Organization. |