M S RAMAIAH INSTITUTE OF TECHNOLOGY

(Autonomous Institute Affiliated to VTU)

Department of Information Science and Engineering

A Report on

Compilers

Submitted in partial fulfillment of the CIE for the subject

System Software

IS61

GROUP MEMBERS:

1.AISHWARYA N (1MS17IS143)

2.SRIVIDHYA RAVICHANDRAN (1MS17IS117)

3.TARUN R (1MS17IS123)

4.SUMUKH M LOHIT (1MS17IS120)

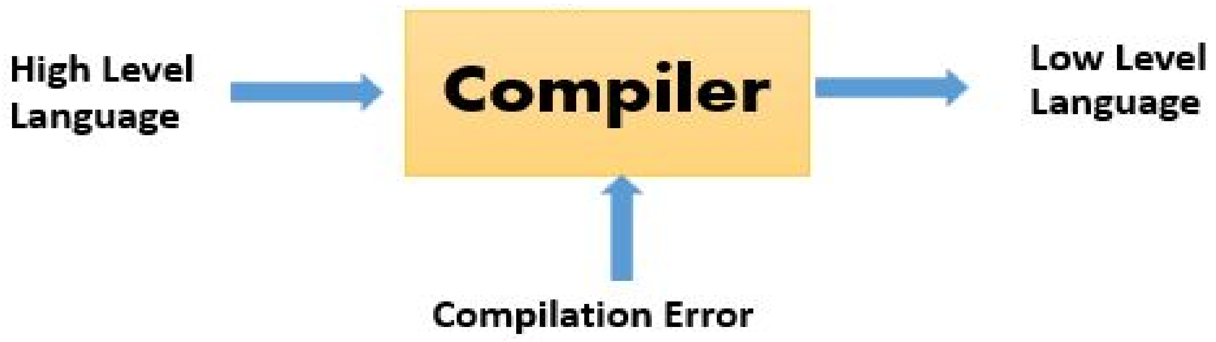
TABLE OF CONTENTS

INTRODUCTION TO COMPILERS:

What is a compiler?

A compiler is a computer program which helps to transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the code efficient which is optimized for execution time and memory space.

The compiling process includes basic translation mechanisms and error detection. Compiler process goes through lexical, syntax, and semantic analysis at the front end, and code generation and optimization at the back-end.



A Compiler

HISTORY OF COMPILERS:

Important Landmark of Compiler's history are as follows:

1.The "compiler" word was first used in the early 1950s by Grace Murray   Hopper.

2.The first compiler was built by John Backum and his group between 1954 and 1957 at IBM.

3.COBOL was the first programming language which was compiled on multiple platforms in 1960.

4.The study of the scanning and parsing issues were pursued in the 1960s and 1970s to provide a complete solution.

GENERAL STRUCTURE OF COMPILERS:

1.**Front end:**

It verifies syntax and semantics, and generates an intermediate representation of the source code for processing by the middle end and performs type checking by collecting type information.Aspects of the front end include lexical analysis,syntax analysis and semantic analysis.

2. **Middle End:**

It includes removal of useless or unreachable code and generates other intermediate representation for the back end.

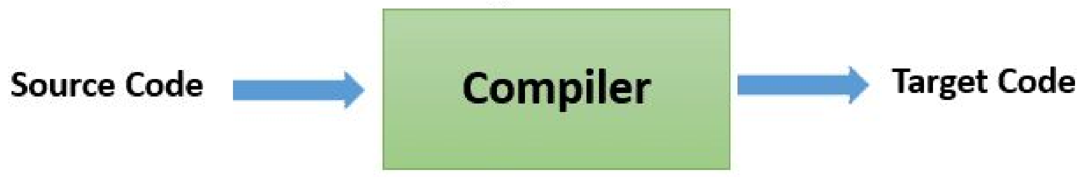
3. **Back End:**

Back end is responsible for translating source to assembly code.

TYPES OF COMPILERS:

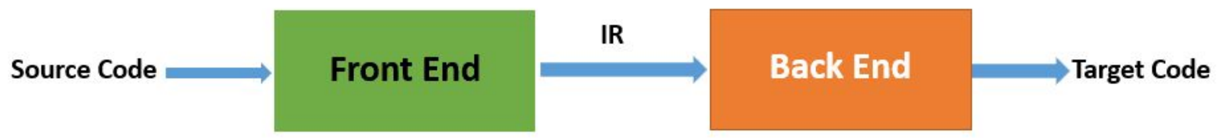
- Single Pass Compilers
- Two Pass Compilers
- Multipass Compilers

1.Single Pass Compiler



In single pass Compiler source code directly transforms into machine code. For example, Pascal language.

2.Two Pass Compiler

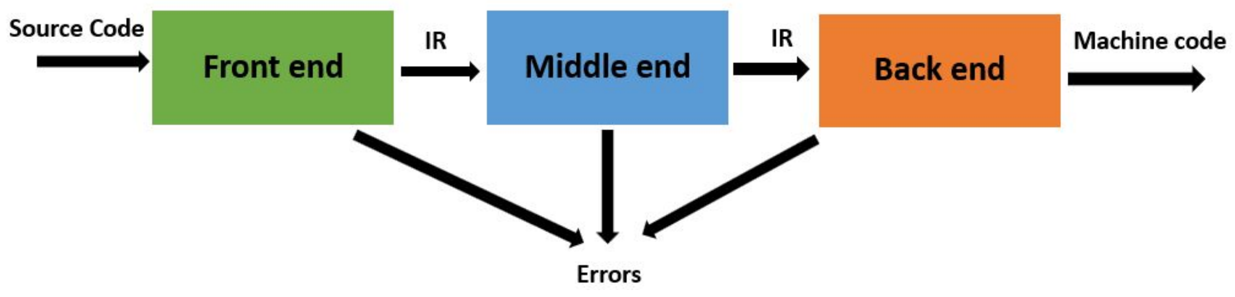


Two pass Compiler is divided into two sections, viz.

1. **Front end:** It maps legal code into Intermediate Representation (IR).
2. **Back end:** It maps IR onto the target machine

The Two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

3. Multipass Compilers



The multipass compiler processes the source code or syntax tree of a program several times. It divided a large program into multiple small programs and processed them. It develops multiple intermediate codes. All of these multipass take the output of the previous phase as an input. So it requires less memory.

Other types of compilers include:

1. Threaded code compiler: replaces given strings in the source with given binary code.
2. Incremental compiler: An incremental compiler is a kind of incremental computation applied to the field of compilation. Individual functions can be compiled in a run-time environment that also includes interpreted functions.Incremental compiler recompiles only those portions of a program that have been modified.
3. Stage compiler: compiles to assembly language of a theoreticalA **retargetable compiler** is a **compiler** that has been designed to be relatively easy to modify to generate code for different CPU instruction set architectures. machine,like some Prolog implementations.
4. Just-in-time compiler: Applications are delivered in byte code,which is compiled to native machine code just prior to execution.

5. Retargetable compiler: A **retargetable compiler** is a **compiler** that has been designed to be relatively easy to modify to generate code for different CPU instruction set architectures.
6. Parallelizing compiler: it converts a serial input program into a form suitable for efficient execution on a parallel computer architecture.

FEATURES OF COMPILERS:

- It must be bug free.
- It must generate correct machine code.
- Preserves the correct meaning of the code.
- The generated machine code must run fast.
- The compiler itself must run fast(compilation time should be proportional to program size).
- The compiler must be portable(i.e modular,supporting separate compilation)
- It must give good diagnostics and error messages.
- The generated code must work well with existing debuggers
- Recognize legal and illegal program constructs

PHASES IN COMPILER DESIGN:

Important Phases in Compiler design:

The following is a typical breakdown of the overall task of a compiler in an approximate sequence : Lexical analysis, Syntax analysis, Intermediate code generation, Code optimisation, Code generation.

A compiler usually performs the above tasks by making multiple passes over the input or some intermediate representation of the same. The compilation task calls for intensive processing of information extracted from the input programs, and hence data structures for representing such information needs to be carefully selected. During the process of translation a compiler also detects certain kinds of errors in the input, and may try to take some recovery steps for these.

PHASE 1: LEXICAL ANALYSIS

Lexical analysis is also called scanning.It breaks the source code entirely into a sequence of tokens.

The first phase of the scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

Generally in a High Level Language there are more number of tokens to be recognised - various keywords (such as, *for, while, if, else*, etc.), punctuation symbols (such as, comma, semi-colon, braces, etc.), operators (such as arithmetic operators, logical operators, etc.), identifiers, etc. Tools like *lex* or *flex* are used to create lexical analysers

PHASE 2: SYNTAX ANALYSIS

It is also called parsing.In this phase,the tokens generated by lexical analyser are grouped together to make a hierarchical structure.It determines the structure of the source string by grouping the token together.

In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.The hierarchical structure generated in this phase is called the parse or syntax tree.

During syntax analysis, the compiler tries to apply the rules of the grammar of the input HLL given using BNF, to recognise the structure of the input program. This is called *parsing* and the module that performs this task is called a *parser*. From a somewhat abstract point of view, the output of this phase is a *parse tree* that depicts how various rules of the grammar can be repetitively applied to recognise the input program. If the parser cannot create a parse tree for some given input program, then the input program is not valid according to the syntax of the HLL.

In the case of HLLs, the syntax rules are much more complicated. In most HLLs the notion of a statement itself is very flexible, and often allows *recursion*, making nested constructs valid. These languages usually support multiple data types and often allow programmers to define abstract data types to be used in the programs. These and many other such features make the process of creating software easier and less error prone compared to assembly language programming. But, on the other hand, these features make the process of compilation complicated.

PHASE 3: SEMANTIC ANALYSIS:

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

PHASE 4: Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Upon carrying out the semantic processing a more manageable equivalent form of the input program is obtained. This is stored (represented) using some *Intermediate code* representation that makes further processing easy. In this representation, the compiler often has to introduce several temporary variables to store intermediate results of various operations. The language used for the intermediate code is generally not any particular machine language, but is such which can be efficiently converted to a required machine language.

PHASE 5: Code Optimisation

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).compilers usually implement explicit steps to optimise the intermediate code.

The techniques for code optimization include:

1. Compile Time Evaluation
2. Common subexpression elimination
3. Dead Code Elimination
4. Code Movement
5. Strength Reduction

1. Compile Time Evaluation:

Two techniques that falls under compile time evaluation are-

A) Constant Folding-

In this technique,As the name suggests, it involves folding the constants.The expressions that contain the operands having constant values at compile time are evaluated.Those expressions are then replaced with their respective results.

B) Constant Propagation-

In this technique,If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.The condition is that the value of the variable must not get altered in between**.**

2. Common Subexpression Elimination-

The expression that has been already computed before and appears again in the code for computation is called Common Sub-Expression.

In this technique,As the name suggests, it involves eliminating the common sub expressions.The redundant expressions are eliminated to avoid their re-computation.The already computed result is used in the further program when required.

3. Code Movement-

In this technique,As the name suggests, it involves movement of the code.The code present inside the loop is moved out if it does not matter whether it is present inside or outside.Such a code unnecessarily gets execute again and again with each iteration of the loop.This leads to the wastage of time at run time.

4. Dead Code Elimination- In this technique,As the name suggests, it involves eliminating the dead code.The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

5. Strength Reduction-

In this technique,As the name suggests, it involves reducing the strength of expressions.This technique replaces the expensive and costly operators with the simple and cheaper ones.

PHASE 6: Code Generation

Code generation can be considered as the final phase of compilation. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.

Finally, the compiler converts the (optimised) program in the intermediate code representation to the required machine language.

The code generator should take the following things into consideration to generate the code:

- Target language : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.

- IR Type : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- Selection of instruction : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- Register allocation : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- Ordering of instructions : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

1.The output of the intermediate code generator may be given directly to code generation or may pass through code optimisation before generating the code.

2.Code produced by the compiler must be correct and be of high quality.

3.Source to target program transformation should be semantics preserving and effective use of target machine resources.

4.Heuristic techniques should be used to generate good but suboptimal code,because generating optimal code is undecidable.

APPLICATIONS OF COMPILERS:

- Compiler design helps full implementation Of High-Level Programming Languages
- Support optimization for Computer Architecture Parallelism
- Design of New Memory Hierarchies of Machines
- Widely used for Translating Programs
- Used with other Software Productivity Tools

Cross-Language Compiler Benchmarking (PAPER-1)

Comparing the performance of programming languages is difficult because they differ in many aspects including preferred programming abstractions, available frameworks, and their runtime systems. Nonetheless, the question about relative performance comes up repeatedly.

Compiler effectiveness is a term used which says the degree by which the core language abstractions are optimized by the compiler and the runtime system.

Benchmark suites are typically built around specific languages or platforms. For the Java virtual machine (JVM), there exist commercial suites such as SPECjvm20081 and academic projects such as DaCapo [1] and DaCapo con Scala [13]. JavaScript has a set of competing projects with different goals including JetStream 2, Octane 3 and Kraken 4.

However, when it comes to comparing the performance of different programming languages, only the Computer Language Benchmarks Game gained wider attention.

SPECjvm2008 is designed to measure the performance of the JVM and its libraries, focusing on processor and memory usage. The benchmarks include text, numerical, and media processing.

DaCapo [1] provides Java benchmarks and an experimental methodology to account for the JVM's dynamic compilation and garbage collection.

The Computer Language Benchmarks Game

The Computer Language Benchmarks Game is the only widely recognized resource for benchmarks in different languages. Currently, it provides a collection of 14 benchmarks for 29 language implementations. The rules for benchmark implementations state that the same algorithm should be used and that an implementation should produce the same output.

A Methodology for Cross-Language Benchmarks

Relevant Abstractions Benchmarks focus on abstractions that are likely to be relevant for the peak performance.

Portable: The benchmarks rely only on language abstractions that are part of the core language and can be mapped straightforwardly to target languages.

Focus on Compiler Effectiveness: This means, the degree by which the core language abstractions are optimized by the compiler and the runtime system

Ease of Use: To facilitate adoption, a simple and reliable methodology for executing benchmarks needs to be defined.

Requirements for Benchmark Implementations

Identical as Possible: The first and foremost requirement is that benchmarks must be as identical as possible between languages. This is achieved by relying only on a widely available and commonly used subset of language features and data types.

Well-typed Behaviour: The benchmarks are designed to be well-typed in statically-typed languages, and behave well typed in dynamic languages to ensure portability and identical behavior. For dynamic languages this means that initialization of fields and variables avoids suppressing possible optimizations.

Fully Deterministic and Identical Behaviour: The benchmarks are designed to be fully deterministic and show identical behaviour in the benchmark's code on all languages. This means, repeated executions of the benchmark take the same path through the benchmark code.

Benchmark Characterization

Metrics

1) Code Size

1. Executed Lines of Code: Compared to the classic notion of lines of code (LOC), we count only lines of code that have been executed at least once to measure the dynamic size of the program instead of its static size.

2. Classes: The number of classes includes only classes of which at least one method was executed. It is stable across languages with the exception of JavaScript

3. Executed Methods: Similar to counting executed LOC, we count methods that have been executed at least once.

4. Per Iteration Methods: In addition to executed methods, we further distinguish methods that are executed for each benchmark iteration. Thus, we separate out code that was only executed once during startup or shutdown. The number of per iteration methods indicates the methods that get likely compiled during benchmarking.

2) Dynamic Metrics: The measured dynamic metrics characterize in more detail the behaviour of the benchmarks.

Method Calls: We measure the observed variability at call sites and count the number of observed receiver types. To make the metric language-independent, classic operators such as ‘+’ or ‘*’ are excluded from the method call count. This is necessary to ensure that results for languages such as Smalltalk and Ruby are comparable to Java or JavaScript.

Maximum Stack height, loops, branches, allocations, variable accesses, array accesses are some of the dynamic metrics.

Test Case Prioritization for Compilers: A Text-Vector Based Approach (PAPER-2)

Test case prioritization aims to schedule the execution order of test cases so as to detect bugs as early as possible. For compiler testing, the demand for both effectiveness and efficiency imposes challenges to test case prioritization.

Software testing aims to guarantee software quality through the execution of test cases. To improve the efficiency of software testing.

Test case prioritization is proposed to schedule the execution order of test cases so that bugs can be detected as early as possible.

To improve the prioritization efficiency, we apply principal component analysis (PCA) to optimize our approach with the adaptive random and search strategies.

This approach consists of the following three steps:

1) This approach regards test programs as text and extracts tokens reflecting fault-relevant characteristics of the test program from text, and then transforms test programs into a set of vectors by counting the number of occurrences of each token for each test program.
2) This approach normalizes the values of elements in the vectors into an interval between 0 and 1.
3) This approach gives three prioritization strategies (greedy strategy, adaptive random strategy and search strategy) to prioritize the set of test programs.

A. Token Extraction

Our approach utilizes only test-input information to prioritize test programs. For C compilers, our approach first regards test programs as text. It recognizes test programs as token streams. Then, our approach extracts tokens reflecting fault-relevant characteristics of the test program from text.

There are three types of fault-relevant characteristics of the test program:

Statement Characteristics:

For C programs, “statement” is a kind of information reflecting fault-relevant characteristics of the test program. For example, if a test program does not have loop statements, the test program cannot detect loop optimization bugs. That is, as loop optimization bugs result from the existence of loop statements. In particular, statement keywords are the best way to extract statement characteristics from text. Therefore, our approach considers all statement keywords in C language as the first type of characteristics, e.g., for, while, if, else, goto, etc.

Type and Modifier Characteristics:

For C programs, “type” also reflects fault-relevant characteristics of the test program. For example, “struct” is a complex and error-prone variable type, and it usually leads to align bugs. Type keywords and modifier keywords in C language as the second type of characteristics, e.g., struct, union, int, static, const, etc.

Operator Characteristics:

Test programs tend to contain many operators to implement some functions. In particular, when test programs contain a large number of operators and form a series of complex operations, the bugs related to operation optimization tend to be triggered. Therefore, our approach considers all operators in C language as the third type of characteristics, e.g., ++, −−, !, ||, >>, etc. After extracting the three types of tokens reflecting fault relevant characteristics from text, our approach counts the number of occurrences of each token for each test program. Therefore, our approach transforms each test program into a text vector.

B. Normalization

As each element in a vector is numeric type, our approach normalizes each value of these elements in each vector into the interval [0, 1] using the min-max normalization [23] in order to adjust values measured on different scales to a common scale.

C. Prioritization Strategies

Based on the set of processed vectors, our approach gives three prioritization strategies to order the set of test programs, including greedy strategy, adaptive random strategy and search strategy.

1) Greedy Strategy:

Our approach gets rid of coverage information and considers only test-input information, and thus we adapt the total technique to use in our approach.

Since each test program can be represented as a vector whose elements are numeric type and the values of them are normalized, our approach gives a score to each test program by calculating the Manhattan distance between the vector and origin vector (0, 0,..., 0). The formula calculating the score for each test program is expressed as follows.

score(vi) = m k=1 |xik|

After giving a score to each test program, our approach prioritizes test programs in the descending order of their scores.

2) Adaptive Random Strategy:

The idea of adaptive random testing to prioritize test cases. The adaptive random strategy selects the next test case that has the maximum distance with the selected test cases. The set of unselected test cases is called the candidate set. Similarly, our approach adapts the adaptive random strategy to prioritize test programs. In particular, our approach uses Manhattan distance to calculate the distance between two test programs. The formula calculating the distance between two test programs is expressed as follows.

distance(vi, vj) = m k=1 |xik − xjk|

3) Search strategy:

The local beam search strategy can be regarded as the strategy between random order prioritization and the adaptive random strategy. That is, the adaptive random strategy calculates all the distances between selected test programs and the test programs in the candidate set for each selection, but the local beam search strategy samples randomly a subset of unselected test programs from the candidate set and calculates the distances between the subset of unselected test programs and all the selected test programs for each selection.

A Novel Technique for Orchestration of Compiler Optimization Functions Using Branch and Bound Strategy (PAPER-3)

Code optimisation involves the application of rules and algorithms to program code, with the goal of making it faster, smaller and more efficient. Applying the right compiler optimizations to a particular program can have a significant impact on program performance.

In current compilers, through command line arguments, the user must decide which optimizations are to be applied in a given compilation run. As compiler optimizations get increasingly numerous and complex, this problem must find an automated solution.

This research proposes a new method to use branch and bound strategy to prune the huge search space of compiler optimizations. As it is going to use branch and bound strategy .so any comparatively unfruitful paths will be eliminated automatically. Tuning time of our algorithm is hence limited by evaluating only those areas in the compiler optimization-search space which are most promising.

In this paper, the following Algorithm is described and briefly explained:

A novel performance tuning algorithm suggested, Branch and Bound based elimination (BB), which aims at picking the best set of compiler optimizations for a program.

II. PROPOSED ALGORITHM

Algorithm- B & B -search-space-pruning (BB)

Input*:* -S Set of available Optimization functions = {F_1... Fn}

Output: – B set having appropriate settings{On/OFF} of flags for all optimization options given in set S i.e unnecessary options are set OFF.

// *RIP*(*Fi*), which is the relative difference of the execution times of the two versions with and without *Fi*,. *Fi* = 1 means *Fi* is on, 0 means off.

i.e $RIP(Fi)= (RIP(F_1=0) - RIP\ (F_1=1))\ /$

$RIP\ (F_1=1)\ *100.$

If (RIP (F)) < 0, it suggests that it is beneficial to turn that function off as it takes less execution time if that optimization function is not applied to program .

//RIP_UB-indicates Upper bound on RIP. It maintains best improvementin execution times so far.

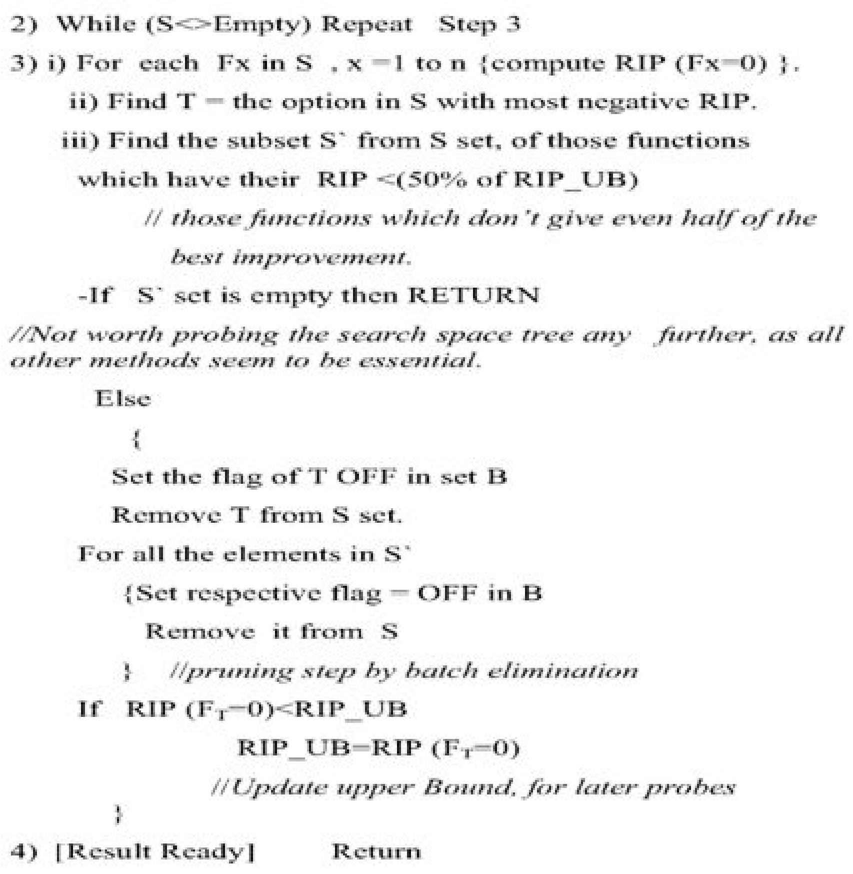
//probe- indicates one entire execution of the program P

Process: –

1. *//Initialization*

i) Initialize default individual settings of flag for each function in set B .*//Generally all ON as -03 level Optimization.*

ii) Set RIP_UB= (Time to execute P with all Flags set to ON - Time to execute P with all flags set to OFF)/ (Time to execute program P with all options OFF).



```
2) While (S<>Empty) Repeat   Step 3
3) i) For  each  Fx in S  , x =1 to n {compute RIP (Fx=0) }.
    ii) Find T = the option in S with most negative RIP.
    iii) Find the subset S` from S set, of those functions
     which have their  RIP <(50% of RIP_UB)
          // those functions which don't give even half of the
            best improvement.
     -If   S` set is empty then RETURN
//Not worth probing the search space tree any   further, as all
other methods seem to be essential.
       Else
         {
        Set the flag of T OFF in set B
        Remove T from S set.
     For all the elements in S`
         {Set respective flag = OFF in B
           Remove  it from  S
         }    //pruning step by batch elimination
     If   RIP ($F_T$=0)<RIP_UB
                    RIP_UB=RIP ($F_T$=0)
                  //Update upper Bound, for later probes
         }
4)  [Result Ready]        Return
```

<u>Features of Algorithm</u>:

a) No assumptions made regarding the compiler

b) No unnecessary time wasted in searching the unpromising portion of the optimization tree.

c) If the relative improvement in the execution time is not exceeding the limit set by the previous probe of the tree, the process is stopped, thus saving the processing time.

d)Major shortcoming is that the algorithm does not consider the interaction amongst the optimization options and hence the solution might be sub-optimal.

<u>Time Complexity</u>:

1) <u>Best Case</u>:

All the optimisation options needed to be applied to the code P.Due to which all optimisation methods become essential to be applied on the code P. Consequently no option has improvement if turned OFF.

Thus, major time is taken by the loop instep 3, shown below

(step 3) For Fx, x =1 to n {compute (RIP Fx=0) }

And later as Subset S' is EMPTY it RETURNs from the algorithm. Thus

$$\sum_{x=1}^{n} 1 = O(n\ )$$

Hence , only once for the "n" options ,computation needed But later no iterations necessary, so time complexity can be Remove T from S set. said to be 0 (n).

2) <u>Worst Case</u>:

The source code P is very perfect and has no scope of optimization further. So for each of "n" optimization methods, "n" iterations are needed. And in each of them one option is suggested to be turned OFF. Thus reducing the size of the source set by 1,in each iteration.

Thus step 3 i.e

for Fx, x =1 to n {compute(RIP Fx=0)}

iterates for (n) times and each time it looks at n, n-1, n-2,……..1 options in S.

So time needed is

$$n + (n-1) + (n-2) + (n-3) + \ldots\ldots + 1 = \frac{n(n+1)}{2} = O(n^2)$$

The time complexity is hence 0 (n2).

It is observed that the time complexity of method would be in the range 0 (n) to 0 (n2) Hence it is comparable and in some cases better than already available algorithms for optimization orchestration.

The above papers were summarised by Tarun.R - 1MS17IS123

References:

1) **Cross-Language Compiler Benchmarking**
https://www.researchgate.net/publication/310464226_Cross-Language_Compiler_Benchmarking_Are_We_Fast_Yet

2) **Test Case Prioritization for Compilers: A Text-Vector Based Approach**

   https://sci-hub.tw/10.1109/ICST.2016.19

3) **A Novel Technique for Orchestration of Compiler Optimization Functions Using Branch and Bound Strategy**

   https://sci-hub.tw/10.1109/IADCC.2009.4809056

4) **A Study on Language Processing Policies in Compiler Design (Common paper)**

   http://www.ajer.org/papers/Vol-8-issue-12/M0812105114.pdf

PAPER 1: NEW TRENDS IN COMPILER ANALYSIS AND OPTIMIZATIONS

Compiler construction primarily comprises some standard phases such as lexical analysis, syntax analysis, semantic analysis,intermediate code generation, code optimization and target code generation but due to the improvement in computer architectural designs, there is a need to improve on the code size, instruction execution speed, etc.
Hence, today better and more efficient compiler analysis and optimization techniques such as advanced dataflow analysis, leaf function optimization, cross-linking optimizations, etc. are adopted to meet with the latest trend in hardware technology and generate better target codes for recent machines.

This paper discusses new and sophisticated methods for compiler analysis and optimization techniques for modern systems against the mere redundancy elimination of the conventional compilers. We also highlight key modifications to conventional compiler techniques (such as data-flow analysis,parallel optimization, loop optimization, etc.) that are still applicable to new computing technology.The main target is to eliminate all forms of inefficiencies in a computer program, eliminate redundant operations (especially in loops and recursive evaluations), and manage resources (by reordering operations and data to better map to the target machine)

1.DATA ANALYSIS:

Data Analysis in modern compilers are beginning to incorporate some level of fuzziness in the data.

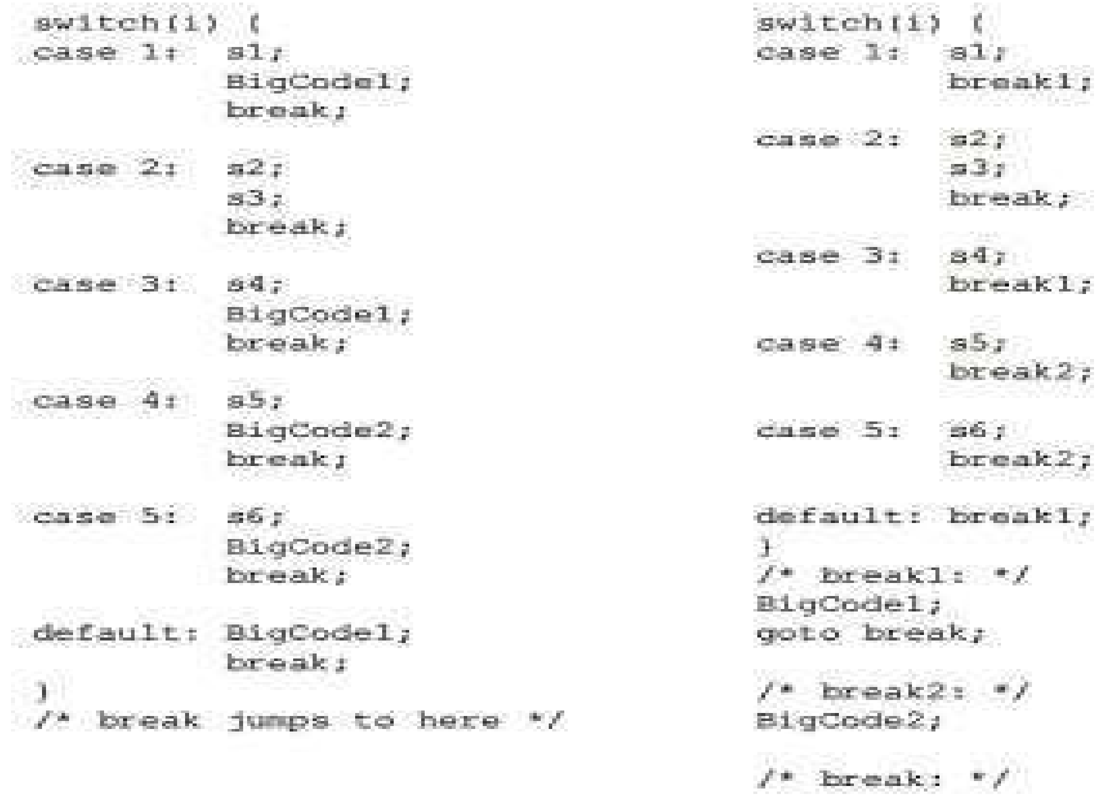
Alias Analysis: During liveness analysis in conventional compiler analysis, the two common functions applied are the Gen and Kill function but most recent techniques in advanced compiler analysis also applies the Aliases in determining the liveness of the references at the exit of the block.

2. Reverse-Inlining (Procedural Abstraction):
Reverse-inlining also considered as procedural abstraction is used to achieve code size reduction. Reverse inlining achieves this by using function calls to replace code patterns that are in the entire program.

3 .Leaf Function Optimization:
Leaf functions are those functions who do not directly call functions in a program. When represented in a call graph, leaf functions forms the leaves of the call graph.
It is easier to inline leaf functions. Hence, function entry/exit code is not required and this reduces code size greatly. During leaf optimization, register constraints are placed as function calls need to be controlled (this also reduces code size). After leaf optimization, there is further opportunity for optimization as the body of the inlined function is within the context of the parent function. Leaf function optimization can be applied to functions that do not resemble leaf function.

4. Combined code motion and register allocation:
Combined code motion and register allocation uses two conventional compiler phases: code motion and register allocation. Code-motion aims to place instructions in less frequently executed basic blocks, while instruction scheduling within blocks or regions arranges instructions such that independent computations can be performed in
parallel. This optimization is applied to theVSDG intermediate code, which greatly simplifies the task of code motion. Data dependencies are explicit within the graph,and so moving an operation node within the graph ensures that all relationships with dependent nodes are maintained.

5. Cross Linking optimization:
This method is commonly used in search engine optimization. Today, this method has also been applied in compiler optimization.
Cross-linking can be applied locally or globally to functions that contain switch statements with similar tail codes. When tail codes are spotted in a switch statement, a cross-linking optimization algorithm is used to factor out this code thereby reducing the actual size of the code.



```
switch(i) {
case 1:   s1;
          BigCode1;
          break;

case 2:   s2;
          s3;
          break;

case 3:   s4;
          BigCode1;
          break;

case 4:   s5;
          BigCode2;
          break;

case 5:   s6;
          BigCode2;
          break;

default: BigCode1;
          break;
}
/* break jumps to here */
```

```
switch(i) {
case 1:   s1;
          break1;

case 2:   s2;
          s3;
          break;

case 3:   s4;
          break1;

case 4:   s5;
          break2;

case 5:   s6;
          break2;

default: break1;
}
/* break1: */
BigCode1;
goto break;

/* break2: */
BigCode2;

/* break: */
```

Fig :code segment showing application of cross linking optimization to an un-optimized switch case statement

6. Address Code Optimization:

Address code optimization uses simple offset and general offset assignment techniques to the number of address computation code by reordering variables in memory. The main aim of address code optimization is to speed-up instructions or execution Time. As data processing expression increases in number, memory access instructions and address computation codes also increase. Hence, there is a need for rearrangement of layout of data in memory to reduce and simplify address computation.

7. Type Conversion Optimization:

Data processing also involves the physical size of the data itself, and the semantics of data processing operations. Support for a variety of data types is the target of most compilers; as such compilers insertmany implicit data type conversions, such as sign or zero extension. For instance, the C programming language specifies that arithmetic operators operate on integer-sized values (typically 32-bit).

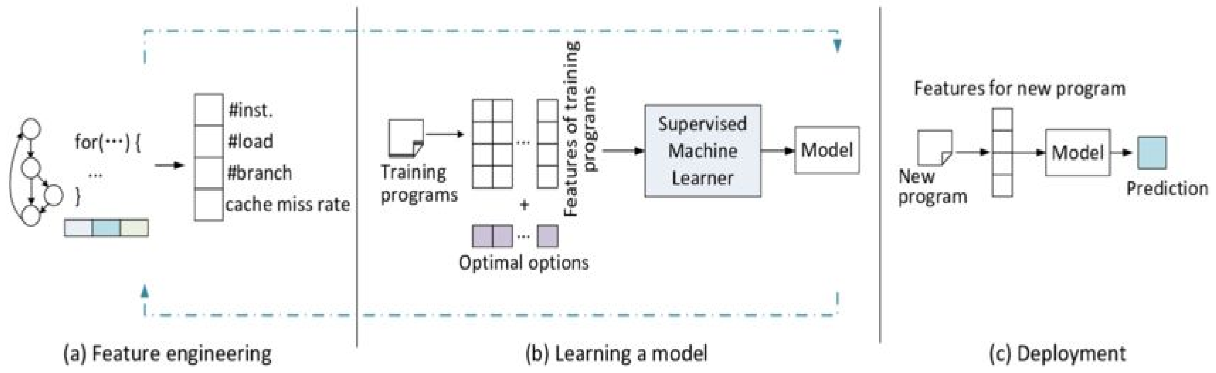
8.Multiple Memory Access Allocation:

Multiple Memory Allocation (MMA) is one of the newest optimization techniques. It involves loading and storing each instruction in multiple registers.

PAPER 2: MACHINE LEARNING IN COMPILER OPTIMIZATION:

Compilers translate programming languages written by humans into binary executable by computer hardware. Machine Learning on the other hand is an area of artificial intelligence aimed at detecting and predicting patterns.

Given a program, compiler writers would like to know what compiler heuristic or optimisation to apply in order to make the code better. Better often means to execute faster, but can also mean smaller code footprint or reduced power. Machine learning can be used to build a model used within the compiler that makes such decisions for any given program. There are two main stages involved: learning and deployment. The first stage learns the model based on training data, while the second uses the model on new unseen programs. Within the learning stage, we need a way of representing programs in a systematic way. This representation is known as the program features . The figure gives an intuitive view how machine learning can be applied to compilers. This process which includes feature engineering, learning a model and deployment is described.



A. Feature engineering:

Machine learning relies on a set of quantifiable properties, or features, to characterise the programs (Figure 1a). There are many different features that can be used. These include the static data structures extracted from the program source code or the compiler intermediate representation (such as the number of instructions or branches), dynamic profiling information (such as performance counter values) obtained through runtime profiling, or a combination of the both.

Standard machine learning algorithms typically work on fixed length inputs, so the selected properties will be summarised into a fixed length feature vector. Each element of the vector can be an integer, real or Boolean value. The process of feature selection and tuning is referred to as feature engineering. This process may need to iteratively perform multiple times to find a set of high-quality features to build an accurate machine learning model.

B. Learning a model :

The second step is to use training data to derive a model using a learning algorithm. Unlike other applications of machine learning, we typically generate our own training data using existing applications or benchmarks. The compiler developer will select training programs which are typical of the application domain. For each training program, we calculate the feature values, compiling the program with different optimisation options, and running and timing the compiled binaries to discover the best performing option. This process produces, for each training program, a training instance that consists of the feature values and the optimal compiler option for the program. The compiler developer then feeds these examples to a machine learning algorithm to automatically build a model. The learning algorithms job is to find from the training examples a correlation between the feature values and the optimal optimisation decision. The learned model can then be used to predict, for a new set of features, what the optimal optimisation option should be.

C. Deployment :

In the final step, the learned model is inserted into the compiler to predict the best optimisation decisions for new programs. To make a prediction, the compiler first extracts the features of the input program, and then feeds the extracted feature values to the learned model to make a prediction.

```
kernel void square ( global float ∗ in , globefloat ∗ out ){
int gid = get_global_id ( 0 ) ;
out[ gid ] = in[ gid] ∗ in [ gid ] ;
}
```

(a) Original OpenCL kernel

```
kernel void square ( global float ∗ in , globefloat ∗ out ){
int gid = get_global_id ( 0 ) ;
int tid0 = 2∗ gid + 0 ;
int tid1 = 2∗ gid + 1 ;
out [ tid0 ] = in [ tid0 ] ∗ in [ tid0 ] ;
out [ tid1 ] = in [ tid1 ] ∗ i n [ tid1 ] ;
}
```

(b) Code transformation with a coarsening factor of 2.

The original OpenCL code is shown at (a) where each thread takes the square of one element of the input array. When coarsened by a factor of two (b), each thread now processes two elements of the input array.

The advantage of the machine learning based approach is that the entire process of building the model can be easily repeated whenever the compiler needs to target a new hardware architecture, operating system or application domain. The model built is entirely derived from experimental results and is hence evidence based.

The above papers were summarised by Srividhya Ravichandran - 1MS17IS117

References:

1) **New trends in compiler analysis and optimizations**
https://www.researchgate.net/publication/316791432_The_New_Trends_in_Compiler_Analysis_and_Optimizations

2) **Machine Learning in Compiler Optimisation**
https://arxiv.org/pdf/1805.03441.pdf

3) **A Study on Language Processing Policies in Compiler Design(Common paper)**

http://www.ajer.org/papers/Vol-8-issue-12/M0812105114.pdf

Paper 1: Analysis of Parsing Techniques & Survey on Compiler Applications

Any program written in a high level programming language must be translated to object code before being executed. Compiler is the need to design and connectivity between the hardware and software process and grammar provides a precise way to specify the syntax and meaning of the language which are a set of rules that specify how sentences can be structured with the terminals, non-terminals and the set of productions.Code generation for embedded processors is the design of efficient compilers for target machines.

This paper describes the application specific features in a compiler and backend design that accommodates these features by means of compiler register allocation and supports the embedded systems and media applications and also presents the techniques of compiler design and also the design of network processors and embedded systems.

Design Methods of Compiler:

Microprocessor design uses CAD tools to provide a starting point in the design process. Existing processor designs provide an architectural reference point from which design modification can be made desirable architectural features. The most accurate method of architectural assessment involves circuit level timing simulation of full processor layout and cycle level simulation of full applications based on optimized compiled code.

This architecture is referred to as an architecture instance by evaluating its performance on a suite of applications using a mapping by a compiler to generate the assembly code performing analysis and evaluating the resulting code. The implications of the results can be used for iterative improvements to the architecture instance mapping or applications.

The basic compiler method enables the primary optimization of three paths:

1. Classical optimizing and procedure inlining

2. Superblock includes all optimizations in classical and adds the superblock optimization and loop unrolling

3. Hyper block includes all optimizations in superscalar and adds the hyper block optimization.

Parsing Application:

Parsing can be defined as a process of analysing a text which contains a sequence of tokens to determine its grammatical structure in given grammar.

Using Natural Language processing technique can be applied to formal language dependency structure is one way of representing the syntax of natural language. This technique automatically generates the language specific information extractor using machine learning and training of a generic parsing instead of explicitly specifying the information extractor using grammar and transformation rules.

Grammars in Compiler design:

Lexical splits the input into tokens, syntax analysis is to recombine these tokens list of characters into something that reflects the structure of the text is the syntax tree of the text. Designing a grammar to describe the syntax of a programming language is usually made able to develop a translator for the language so that programmers who use the language.

Applications of Compiler Design:

Compiler is a program that reads the many applications in networks, operating systems, and embedded systems with high positive rates.

Compiler Network Processor:

Network processor instruction set allows avoiding costly shift operations special instructions for packet level addressing, compiler bit packet manipulation is made visible to the programmer by means of compiler known functions and maps call to compiler known functions not the regular function calls into instruction sequences. Using compiler known functions the developers have detailed knowledge about the underlying processor readability of the code is improved significantly.

Compiler known functions with simple examples consider a case where we would like to add the constant to a 7-bit wide packet stored in bits of some register denoted by the C variable a. For example a = (a&0xFE03) | ((a+ (2<<2))&0x01FC).

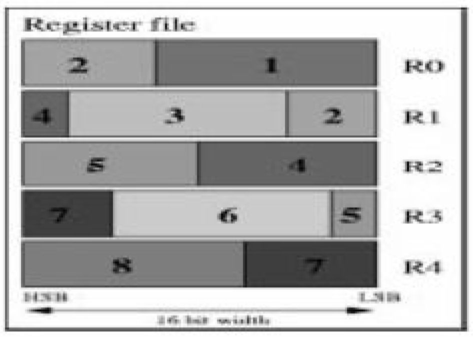


Fig shows the values allotted in Register file

This expression is simplified into standard compiler optimization techniques that are constant folding which translate into a relatively large instruction sequence on a standard processor. Network processor can implement the entire assignment within a single instruction, for this purpose can use packet access PA (int op, int var1, int off1, int var2, int off2, int width)

Collaboration of Compiler & Operating System:

System designer must define how to initiate a speed change and how to select a speed level to automatically deciding on the proper locations to insert PMPs by the compiler in an application code.The solution over here that determines how far apart any two PMPs should be placed with sequential code and an estimate of instruction latency the code is inserted. In real time the problem occurs due to the presence of branches, loops and procedure calls that eliminate the determinism of the executed path compared to sequential code. To control the overhead of speed scheduling during the execution of an application we use a compiler directed technique, during compile time the compiler inserts instrumentation code that computes information about the worst case remaining cycles of the application.

PMP interrupt service routine is periodically invoked to adjust processor speed based on WCR. Executes the power management hints at some point before a PMP to update the WCR based on the path of execution. Compiler role is to support and insert PMHs in the application code.

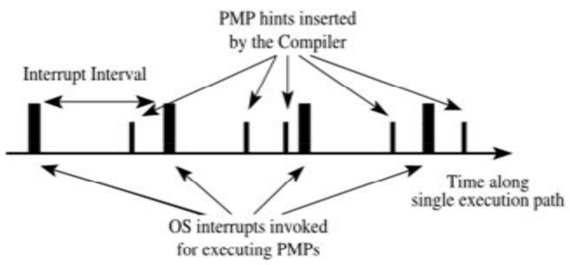

PMP hints inserted by the Compiler
Interrupt Interval
Time along single execution path
OS interrupts invoked for executing PMPs

Figure shows specific path for Invocations

Paper 2 : Towards Compiler Optimization of Codes Based on Arrays of Pointers

Languages allow the use of complex data structures usually based on pointers and dynamic memory allocation in order to speedup code development and, besides this, it also may lead to reducing the program execution time.When dealing with pointer-based data structures usually built at run time, current compilers are not able to capture, from the code text, the necessary information to exploit locality, automatically parallelize the code, or carry out other important optimizations.

Methodology:

Techniques can be included in compilers in order to allow the automatic optimization of real codes based on dynamic data structures which estimate at compile time, the shape the data will take at run time.The analyzer generates a reduced set of reference shape graphs (RSRSG) for each statement in the code.

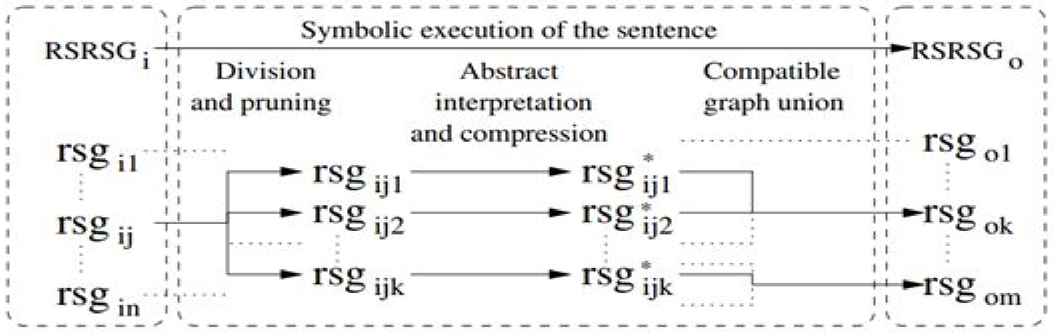
The analyzed codes are based on complex data structures such as doubly linked lists, trees, and octrees among others, and combinations of them, such as a doubly linked list of pointers to trees where the leaves point to doubly linked lists.Each statement in the code will have a set of Reference Shape Graphs (RSG) associated with it, which are called a Reduced Set of Reference Shape Graphs (RSRSG). The RSGs are graphs in which nodes represent memory locations which have similar reference patterns.To determine whether or not two memory locations should be represented by a single node, each one is annotated with a set of properties. Now, if several memory locations share the same properties, then all of them will be represented by the same node.

The symbolic execution of a statement involves the following steps :

1.**Division and pruning** - **The first step comprises graph division to better focus on the several memory configurations represented by the RSG. Pruning removes redundant or nonexistent nodes or links that may appear after the division operation.**

2.**Abstract interpretation and compression** -**Then the abstract interpretation of the statement takes place and usually the complexity of the RSGs grows. In order to counter this effect, the analysis carries out a compression operation.**

3.**Compatible graph union** - **In this phase each RSG is simplified by the summarization of compatible nodes, to obtain the rsg* ijk graphs. Furthermore, some of the $rsg*_{ijk}$ can be fused into a single rsg_{ok} if they represent similar memory configurations. This method only describes single links.**



Schematic description of the symbolic execution of a statement

We can view an array of pointers as a set of n selectors (links), all with the same name.Thus, the problem arising with the arrays of pointers is that a single selector name represents several links, and all of them belong to the same memory location (due to having been allocated by the same malloc instruction).The problem arising with the arrays of pointers is that a single selector name represents several links, and all of them belong to the same memory location.

Therefore we take into account multi selectors.

sel is a single selector which can point to a single memory location and which can be modified by statements like "x→sel=...".sel1 and sel2 represent arrays of selectors. The difference between sel1 and sel2 is that we know the size of the sel1 array at compile time, but the size of sel2 is defined at run time. In any case, we now want to deal with both types of arrays of selectors, which now have to be modified by statements like "x→sel1[i]=..." or "x→sel2[i]=...".

```
typedef struct str {

         ...

        struct str1 *sel;

        struct str2 *sel1[256];

        struct str **sel2;

 }

 x=(str *)malloc(sizeof( struct str));

 x->sel2=(str **)malloc(n* sizeof(str *));
```

Since sel1 and sel2 are not single selectors, they are called multi selectors.

Paper 3 : Performance Analysis of Symbolic Analysis Techniques for Parallelizing Compilers

Advanced analysis and transformation techniques exist today, which can optimize many programs to a degree close to that of manual parallelization. The ability of a compiler to manipulate and understand symbolic expressions is an important quality of this technology.

The accuracy of data dependence tests, array privatization, dead code elimination, and the detection of zero-trip loops increases if the techniques have knowledge of the value ranges assumed by certain variables.

Symbolic Analysis Techniques in Polaris

1. Range Propagation and the Range Test are the most important techniques that deal with symbolic terms in array subscripts.

2. Expression Propagation is a conventional technique to transform symbolic expressions into more analyzable form.

3. Symbolic Expression Simplification,provides essential functionality that several Polaris passes make use of.

Range Propagation and the Range Test

The Range Analysis technique determines the value ranges assumed by variables at each point of a program. It does this by performing abstract interpretation along the control and data flow paths. The results are kept in a range dictionary, which maps from variables to their ranges.

The Polaris compiler currently uses range dictionary information to detect zero-trip loops in the induction variable substitution pass, to determine array sections referenced by array accesses in the reduction parallelization pass, and to compare symbolic expressions in the Range Test.

The Polaris compiler is able to recognize array reductions and translate them into parallel form.

Polaris supports two levels of range dictionaries. The control range dictionary collects information by inspecting control statements, such as IF statements and DO statements. The abstract interpretation (AI) range dictionary subsumes the control range dictionary and collects additional information from all assignment statements.

Expression Propagation

By propagating the expression assigned to a variable to the variable's use sites, symbolic expressions can deliver more accurate information. Polaris can propagate constant integers, constant logical, and symbolic expressions within a procedure and across procedures. Real-valued expressions can also be propagated, but this option is switched off by default. Before analyzing individual subroutines, Polaris performs interprocedural expression propagation, during which assignments to propagated expressions are inserted at the top of each procedure. Then, these expressions are propagated to possible call sites.

Symbolic Expression Simplification

The simplification of symbolic expression is important, as compiler–manipulated expressions tend to increase in complexity, making them difficult to analyze. Nearly all Polaris passes make use of expression simplifier functions. For instance, the procedure performing symbolic expression comparison assumes that the two expressions are reduced to their simplest form. Polaris provides the following three simplifying techniques for symbolic expressions:

– Combine: A+4*A -> 5*A

– Distribute: A*(3+B) -> 3*A+A*B

– Divide: 3*A/A -> 3

The above papers were summarised by Aishwarya N - 1MS17IS143

References:

1) Analysis of Parsing Techniques & Survey on Compiler Applications

https://ijcsmc.com/docs/papers/October2013/V2I10201320.pdf

2) Towards Compiler Optimization of Codes Based on Arrays of Pointers

https://link.springer.com/chapter/10.1007/11596110_10

3) Performance Analysis of Symbolic Analysis Techniques for Parallelizing Compilers

https://link.springer.com/chapter/10.1007/11596110_19

4) A Study on Language Processing Policies in Compiler Design(Common paper)

http://www.ajer.org/papers/Vol-8-issue-12/M0812105114.pdf

(Paper 1)

PROBLEM STATEMENT

Predict the execution path from CFG, by performing Data on Abstract Syntax Tree during compilation process using static analysis by accepting the user.

Then finally the front end of a compiler generates intermediate codes for the predicted execution path.

Backend of a compiler will generate the object codes for the same execution path and the same codes will be loaded in the memory for execution. Thus program codes are reduced and automatically program occupies less memory.

PROPOSED SYSTEM

The proposed system will identify the execution path during compilation process itself by accepting the input from the user and target object codes will be generated fir the identified execution path. Proposed system will help the compiler to eliminate the generation of object codes for each statement of a source language. Finally compiler generates the target codes only for predicted execution path and same will be loaded for execution. It reduces loading time, space and execution.

An Existing compiler consists of three main parts: frontend, middle end and backend . Frontend checks whether the program is correctly written in terms of programming language and semantics. Error checking and type checking is done. Frontend generates intermediate representation for the middle-end . Backend generates machine code.

Execution of a program depends on machine architecture, machine status and the inputs. Inputs decide the execution path and the computer executes only the instruction which comes under this path.

Proposed system identifies the execution path during compiling process itself and the compiler generates object codes for the identified execution path, and this it executes faster and needs less memory. Execution path is identified in 1 phase at the end of the front-end.

STATIC ANALYSIS

Static analysis process consists of 2 steps:- Abstract Syntax Tree (AST) and abstract syntax analysis. To analyse source code, tool should parse the code and create a structure into an AST which captures the essential structure of the input. The proposed static analysis tool predicted the execution path during compilation and helped the compiler to generate object codes only for the execution path. Thus, codes can be reduced automatically. AST walker analysis is performed by walking AST and checking whether it fulfils coding standards. It accepts inputs during compilation itself. Data flow analysis algorithm operates on control flow graph (CFG) generated.

CFG represents all possible execution paths in the control graph.

Then, data flow analysis is performed to predict path-sensitive execution path of CFG., whether feasible or infeasible path.

We reduce the analysed CFG to reduce the amount of calculations so that only subsets of CFG paths are to be analysed.

On one case study, it was observed that size of a block was reduced by 486 bytes(from 1496 to 1010),a 32.48% decrease.

CONCLUSION

The proposed tool predicts the execution path by accepting inputs during compilation itself , thus resulting in reduced memory size and faster execution.

(paper 2)

Abstract

This work focuses on reusing a compiler, originally designed for compiling a language N to a target W, modified for translating a new language N' to multiple target architectures. The compiler is split into three independent objects (Front End, Back End and Interface Controller), each one working in a single pass, and each one being developed with a reuse approach. The resulting toolset is compared with a more conventional multi-pass compiler design. The use of semantic directed parsing introduces another form of reuse (that of semantic attributes) within the compiler toolset.

This paper describes an experience in the development of a compiler where an approach different from the two above has been adopted. The approach can be properly classified as *compiler reuse*.

Reuse is, in our case, based on the development of an open system (namely easy to extend and modify), with a compositional and white-box based strategy of reuse . The language for which such experience has been undertaken is called MML, which stands for Multi Micro programming Line.

Two perspectives of reuse have been considered: reuse without modification or with modification. Roughly speaking they correspond to black-box and white-box reuse respectively.

Black-box reuse in compiling is mainly based on the use of generative tools, i.e. compiler-compilers. 'In white-box reuse, the level of modifications brought span from parameterization and built-in adaptability to deep transformations.

Approach and Conclusion

The reused system is a compiler, which has been transformed to accept a different source, and extended to generate code for various target machines. An open approach has been used, contrasted with the traditional method based on the use of generative tools like compiler-compilers.

Among the advantages of the approach are: a deep and fast understanding of system architecture and increased reusability of both design and implementation. The resulting tool is as simple and reliable as the original one; simplicity has been a key factor with respect to personnel turnover throughout project development.

We have started from the Pascal-S compiler. Such a compiler is highly reusable on account of its simplicity and good design. A single-pass compiler yields to a simpler, faster to implement and more reliable system. Single pass approach yields an easy-to-reuse software system.

In our development, we have transformed the Pascal- S compiler into the MML Front End and the Program Interface Checker; and its P-Code interpreter into the Back End.

The implementation of each object in the compilation toolset has been achieved by intermediate milestones, as has been detailed for the Back End.

As for the Front End, in each sub-step of its development the source language (initially being Pascal-S) is slightly transformed into a language closer to the final MFIL language. Each ”intermediate” source language corresponding to a sub-step was consistent and generating intermediate code, thus being testable. In this way both the source and the intermediate language, the Front End and the interpreter were transformed and debugged simultaneously, thus maintaining consistent interfaces.

This strategy of design and code reuse by a sequence of transformations is generalizable to other applications with the advantage of avoiding most of the nasty design bugs relative to missing information in the specifications.
A special attention to reuse has been devoted to data structures and the most strategic parts of a system (in this case the routines for symbol table management, code generation and separate compilation handling). The role played by the Symbol Table and Code File objects in the development of the compiler toolset has been examined.

This description of this experience is the most significant but not the only one of software reuse which evolved from the MML compiler.

After project completion, it evolved towards a commercially available toolset, called MME. The academic partners undertook several projects where MML was the starting point, in retargetable kernel design, and fault tolerant software. These projects had an impact on the compiler, the code generator, and mostly on the Program Interface Checker.

Had it not been a reusable tool from the very beginning, most of the subsequent experiences would have been precluded by the need of rewriting a portion of a compiler.

(paper 3)

SWIFT/T COMPILER ARCHITECTURE AND INTERMEDIATE REPRESENTATION

The STC compiler translates a high-level Swift/T program into optimized low-level code for the runtime. The two middle stages that process an intermediate representation (IR) of the input program are of the most interest. The STC compiler uses a single IR throughout the compiler with two variants.

The frontend produces IR-1, to which optimization passes are applied to produce successively more optimized programs. IR-2 augments IR-1 with memory management and data passing bookkeeping that is needed for code generation.

A major contribution of the work is development of an IR for the execution model of data-driven task parallelism. A good IR has several attributes. First, simplicity and uniformity reduces the complexity of optimization passes. Second, the IR must be high level enough to hide irrelevant details.

STC mainly aims to optimization communication so, for example, CPU registers and memory addresses are not exposed.

Third, the IR must be low level enough to expose relevant details. STC needs to optimize communication, so the IR makes communication explicit, with explicit task creation and a distinction between local and nonlocal data.

COMPILER OPTIMIZATION

We have implemented a significant suite of optimizations in the STC compiler. Several novel optimizations have also

been implemented that are specific to task parallel execution models. We have described and evaluated several approaches that, in essence, rearrange the relationship of tasks within the IR to reduce task creation and data operations without reduction in parallelism. The remainder of optimizations are adapted from techniques used for low-level imperative languages such as C or Fortran [6]. These optimizations translate naturally to the task parallel context, since they focus on identifying and eliminating redundant operations, which serves to reduce communication as well as machine instructions.

Our contribution is in adapting these optimizations for the STC IR in particular and for data-driven task parallelism in general. Space does not permit description of each optimization. Grouped by optimization level, the optimizations are:

O0: Naïve compilation strategy with no optimization

O1: Basic optimizations: global value numbering, constant

folding, dead code elimination, and loop fusion

O2: More aggressive optimizations: asynchronous op expansion,

task coalescing, hoisting, and small loop expansion

O3: All optimizations: function inlining, pipeline fusion,

loop unrolling, intrablock instruction reordering, and algebra

Communication Reduction from Compiler Optimization

Overall we see that all applications benefit markedly from basic optimization, while more complex applications benefit greatly from each additional optimization level. With all optimizations enabled, optimized Swift code is comparable to hand-coded ADLB.

Application Speedup from Compiler Optimization

Application speedup benchmarks were run on a Cray XE6 supercomputer with 24 cores per node. Unless otherwise

indicated, 10 nodes were used for benchmarks. We measure throughput in tasks per second dispatched to worker processes. Operation count reductions gave a roughly proportional increase in throughput. In

Wavefront, the speedup was more than proportional to the reduction in runtime operations: the unoptimized code excessively taxed the data-dependency tracking at runtime, causing bottlenecks to form around some data. This result supports the hypothesis that runtime operations are the primary bottleneck for Swift/T.