AESOP LLVM and Compilation Benchmarks

Gilroy Gordon
1101304
Computer Science
University of Technology, Jamaica
gilroygordon@gmail.com

Moore's Law describes a future trend where the number of transistors placed inexpensively on an integrated circuit will double approximately every two years. This law has been observed for decades as transistors increase on smaller integrated circuits to the point where processors are now being developed with a multiplicity of cores to deliver new power at a significant cost of degradation to decrease the overall electrical power required by the circuits and its trade offs. However, with such new advances in hardware, complex compilers become outdated similarly to the programmers who are not equipped to handle these new achievements. The AESOP project is a project initiated to solve these problems. The project aims to automatically translate existing serial code into parallel code automatically when being compiled and optimize this for the specific hardware the compiler is be executed on while not focusing a build for any specific hardware. Moreover, it is a derivative of the LLVM project and hence harnesses it benefits and disadvantages. This research will seek to evaluate this project, the problems it aims to solve, its solutions and its compiler translation benchmarks with respect to CLANG.

Keywords—Aesop, LLVM, CLANG, Compiler Translation

I. Introduction

The Adaptive Environment for Supercompiling with Optimized Parallelism (AESOP) project aims to serve as an automatic parallizer, transforming serial code written at the current moment in languages such as C, C++ and Fortran to parallel code into parallel code[1]. The project was developed at the University of Maryland, College Park but has noted significant contributions from Wojciech Matyjewicz for his input in base loop memory dependence and research and funding from the NASA Office of the Chief Technologist's Space Technology, Defense Advanced Research Projects Agency (DARPA), Architecture-Aware Compiler Environment (AACE) programme. The project seeks to leverage the benefits of already created work such as the LLVM project, CLANG and DRAGONEGG project in an attempt to focus on optimizing LLVM Intermediate Representation (IR) through traditional methods instead of polyhydral techniques.

II. The Problem

To derive a true appreciation for the AESOP project, one should take some background knowledge into consideration.The first consideration is the advances in processor development over the past fifty years. In order to achieve more processing power, more transistors had been integrated in the integrated circuits, following Moore's Law while also increasing the clock speeds of these processors.

With an increase in clock speed, the need for more power and this introduced design hazards and considerations for heat. When Intel® introduced the dual core processor; other companies saw this as an optimal option to increase hardware power. However, with more cores available, the clock speed of each had to be reduced [2]. This solution ignited another problem however, as now thousands of programs with millions of lines of code had been implemented to run on one processor and were not taking true advantage of the additional processors available.

These new advancements also meant that programmers had to be introduced to a new style of programming, parallel programming in order to truly reap the benefits of the Instruction Set Architecture (ISA). This created a significant learning curve and was increasingly difficult and time consuming as programmers also had to re-engineer existing programs to utilize this architecture. Programmers were then unequipped to handle [2].

Moreover, compilers remain a complex means of converting source code to object code to run at its best on their target hardware. Hardware development has outpaced compiler development and as such there is a considerable amount of time spent to create new compilers to take advantage of the hardware features [2].

Finally, software has become an important part of our society focusing on many areas requiring optimal solutions to problems such as signal processing, video processing, real-time analysis [2].

III. Solution

As a solution to the above stated issues, the AESOP project sought to develop a new optimizer that considered automatic parallelization of serial programming languages such as C++, C and Fortran instead of focusing on specialized languages designed specifically for each new machine [2]. The prototype proposed and implemented only supports shared memory Multiple Instruction Multiple Data systems (MIMD).

Moreover, the tool has made considerations for codes that have already been parallelized such those employing the OpenMP or Message Passing Interface (MPI) annotations[2]. There are three major aspects of the compiler which assists it in solving the problems stated above. These include system characterization, automatic parallelization and continuous optimization and learning exploits as illustrated in Figure 1 below.

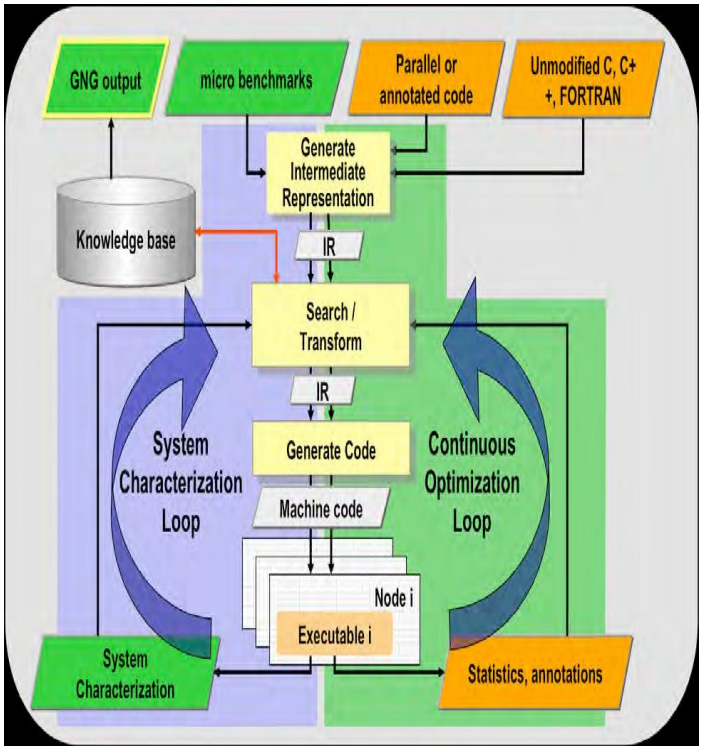

Figure 1: A Diagram illustrating the application of the major design considerations of AESOP

In Figure 1, the system characterization loop illustrated on the left is driven by a nonlinear optimization engine and seeded by micro benchmarks. This includes the receipt of key descriptive elements about the computing platform (memory and core configuration) which is fed to the continuous optimization loop which exploits parallel hardware to simultaneously experiment and estimate performance. In particular, the continuous optimization loop includes the dynamic runtime system, which includes the application of transformations (code hoisting, concurrent threads, loop enrolling), instrumentation to code, monitoring transformation results and applying further transformations based on the optimizations considered.

Integral in this process is the utilization of a bit of artificial intelligence with the learning and reasoning element. This element possesses a persistent knowledge base receiving inputs from system characterization and application instrumentation in addition to its Application Programming Interface (API) to the search/transform segment. The learned correlations among machine, code, transform characteristics and performance results inform the optimization loop.

These allow AESOP to automate the optimization and parallelization of predominantly sequential applications while also the extraction of fine grained parallization of already parellized code. In particular, the project was attributed with the following:

- Automatic parallelization of sequential code enabled more than ten (10) times productivity for the software life cycle [2].
- Heuristics involved achieved an increase of 20% in performance [2].
- The design combines the aforementioned elements seamlessly adapting codes to new hardware systems with zero modifications [2].
- From tests cited at the University of Maryland, after analyzing benchmarks compiled on an AMD Opteron™ 6212 a speedup of 1.88X was achieved on all benchmarks [1]

IV. Limitations of Tool

The following lists current limitations of the tool:

1. Only supports shared memory multiple instruction multiple data systems [2].
2. The project is built on top of the open source LLVM project is subject to change depending on any advancement in this project. Being that the project is dependent on this project, it also inherits the projects benefits as well as challenges.
3. The project depends heavily on the DRAGONEGG plugin that integrates the LLVM optimizers and code generators to generate LLVM IR before feeding it into the affine parallelizer.

V. Additional Research

The projects lifetime has been short, however it has yielded many interests, solutions to many needs and lends itself to further research in many areas. With more advancements and research in parallel computing which considers the use of Graphic Processing Units (GPUs) alongside Central Processing Units (CPUs) or the combination which created the General Purpose CPUs, there is room to identify how the tool optimizes existing serial code to utilize these hardware features. Moreover, the use of Field Programming Gate Arrays (FPGAs) has proven beneficial where optimized processing is concerned, however it has proven even more difficult to target from a programmers stand point. Research questions have been developed with the aim of determining how the tool may be used to utilize these features of hardware [2].

VI. Compiler Benchmarks

To take a closer look at the compiler benchmarks as performed by AESOP is to acknowledge that AESOP truly does not convert source code is to LLVM IR to be fed to the LLVM Core as depicted in Figure 2. AESOP currently

supports the C, C++ and Fortran Languages by leveraging the Clang LLVM and DRAGONEGG LLVM to perform translation at the LLVM front end. Therefore to detail the compiler benchmarks for the tool a closer look at DRAGONEGG or clang's translation process must be take. For the purposes of this research, the latter will be used.

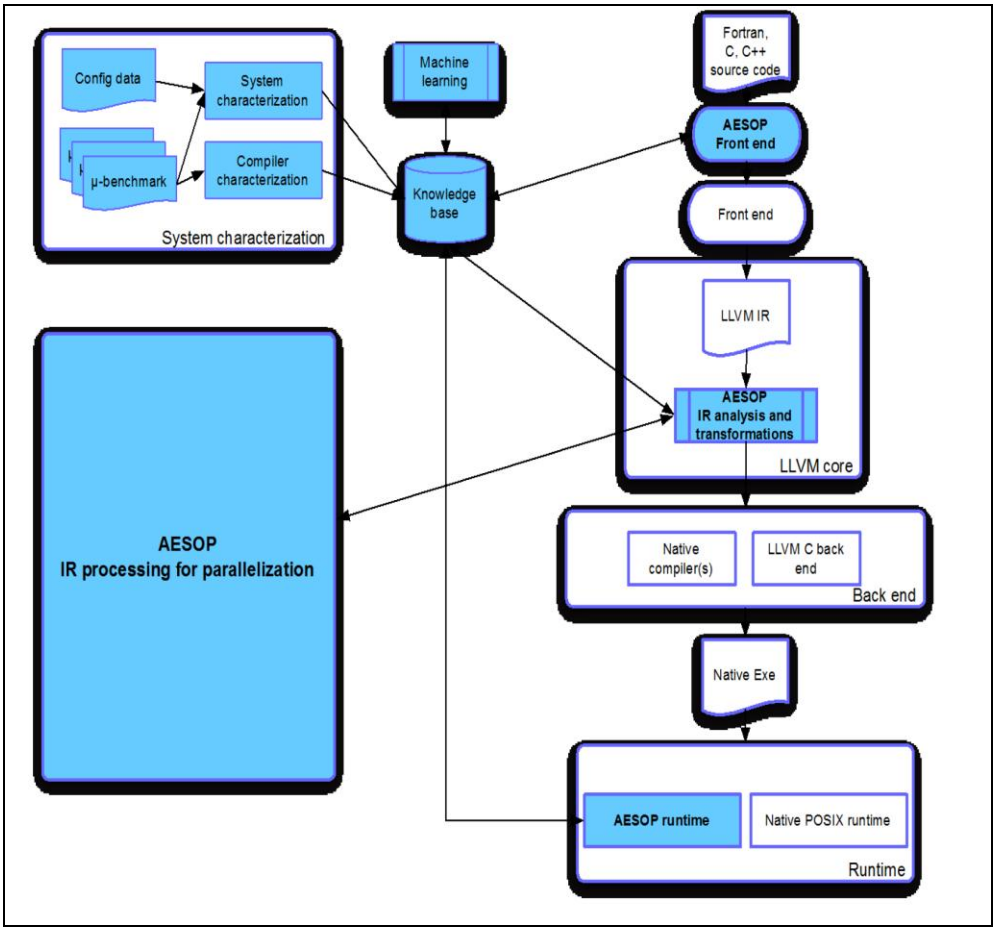

Figure 2. A Diagram depicting the integration of AESOP modules (blue) with LLVM (white) [2]

Moreover, Figure 3 which illustrated the compiler translation phases will be used as a guide for describing the phases included by CLANG and incorporated to an extent in AESOP.

According to Steve Naroff in a presentation, CLANG has five main core libraries: Basic, Lex, Parse, AST and Sema which allows it to implement the first three steps (Lexical Analyis, Syntactic Analysis and Semantic Analysis) as illustrated in Figure 3 as its Parser, Preprocessor and Lexer modules [4]. This is complemented by Guntli's research into the architecture of CLANG [5].

Guntli describes the lexer's unorthodox behavior of calling the preprocessor upon encountering preprocessor directives while it is in the process of parsing the source code into tokens or lexemes. This operation involves passing the identifier of the lexer and the directive to the preprocessor which reads all tokens, saves and adds them to a symbol table which may be used a reference in the event that the lexer encounters these directives again. While parsing the parser stores each AST declaration in a DeclContext (Class Definition) thus creating a parse tree. This may be illustrated by Clang's built in xml writer for ASTs [5].

Furthermore, although clang has a symbol map storing the memory layout for each user defined type, actual symbol lookup is done with the assistance of the AST in its ASTContext(Class Definition) since each node has a reference to its parent. Moreover, if a symbol lookup is found in a parent node, the DeclContext implements scoping. To assist this process, Clang uses an identifier table which essentially maps the location of where identifiers were parsed to verify if an identifier exists. This is useful when libraries are included. Clang also accommodates function overloading through the addition of overload declaration to the function declaration already found. Clang therefore is only able to create an identifier expression using the correct declaration once the parameter list is identified which is called Argument Dependent Lookup.

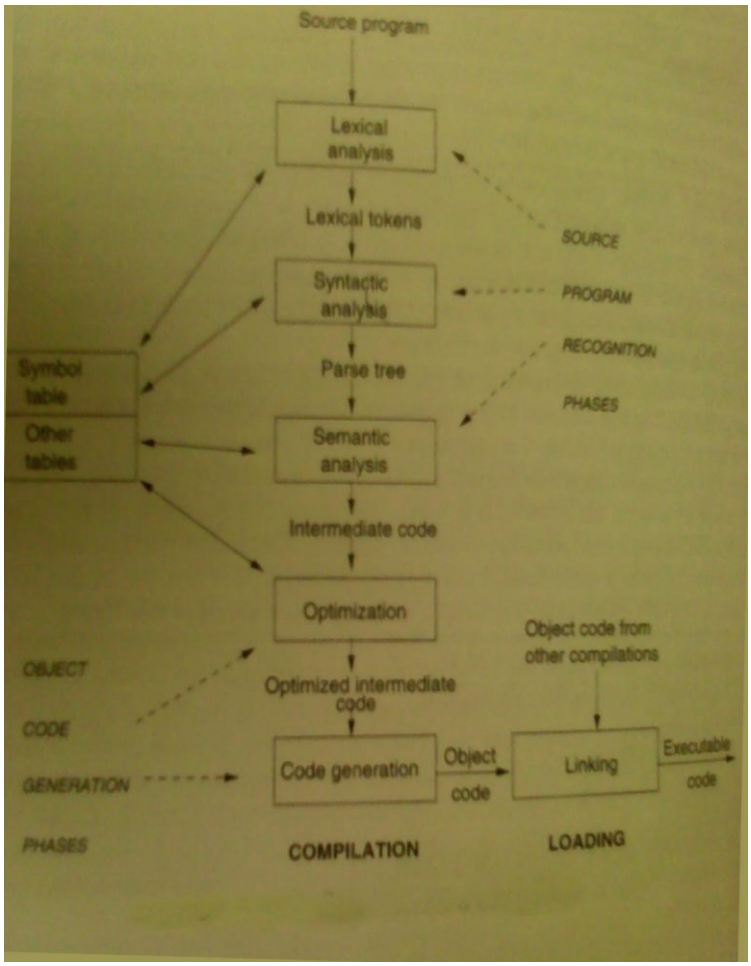

Figure 3: Compiler Translation Phases [3]

Essentially, throughout the process of creating AST Clang is able to perform syntactic and semantic checks before translating into LLVM IR.

When considering optimization, it is important to note that Clang builds upon LLVM's optimization thus allowing the following table to describe its optimization levels [6].

Optimization Levels	Description
o0	No Optimization performed
o1	Between o1 and o2
o2	Moderate level of optimization, enabling most optimizations
o3	Similar to o2 except that its optimizations generate larger code and often takes longer to perform
o4	Enables link time optimization
os	Similar to o2 but RISC

Table 1: Optimization Levels

At this point, we can make reference to the fact that AESOP leverages optimization in Figure 2 and notably operates with optimization level 3 as described in Table 1.

At this stage AESOP takes the Optimized Serial IR and performs its analysis and transformations using its three phases as described in II. More importantly, AESOP's approach to affine parallelization seeks to utilize symbols to identify dependencies within programs using traditional methods instead of polyhydral techniques. Traditional methods allow the tool to benchmark over two million lines of code that are not necessarily constrained to a particular standard and not operate in polynomial time[1]. This benefit however, makes some representations of loop representation difficult. AESOP's parallelization module is illustrated in Figure 4. The diagram, although descriptive, details the initial alias and distance vector analysis (LLVM modules) done by the parallizer which seeks to identify variable dependenies based on aliases found in the IR. This information is pertinent as when passed to the parallizer, the decision module can effectively decide what to parallelize and when to consider scalar dependencies before passing this to the parallel code generator which generates the code.

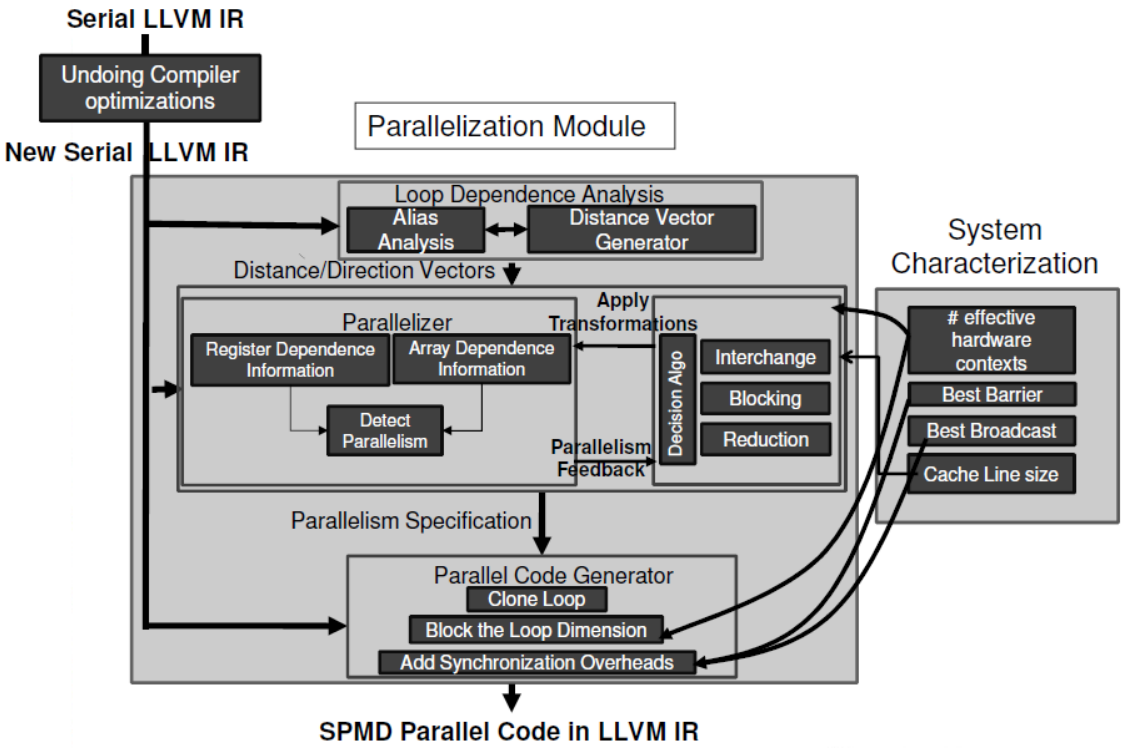

Figure 4: AESOP's Parallelization Module [1]

The Simple Program Data Model is then used when generating code with the hope of utilizing multiple threads on a shared memory system.

VII. References

[1]R. Barua, T. Creech and A. Kotha, 'AESOP: The Autoparallelizing Compiler for Shared Memory Computers', Doctorate, University of Maryland, College Park, 2011.

[2] Defense Advanced Research Projects Agency/ Information Processing Techniques Office, 'ADAPTIVE ENVIRONMENT FOR SUPERCOMPILING WITH OPTIMIZED PARALLELISM (AESOP)', BAE Systems, Inc., Guildford, UK, 2011.

[3]D. Watson, High-level languages and their compilers. Wokingham, England: Addison-Wesley, 1989.

[4]S. Naroff, 'Clang Intro', Apple[sn], 2015.

[5]C. Guntli, 'Architecture of clang: Analyze an open source compiler based on LLVM', University of Applied Science in Rapperswil, 2011.

[6] Clang.llvm.org, 'Clang Compiler User's Manual — Clang 3.7 documentation', 2015. [Online]. Available: http://clang.llvm.org/docs/UsersManual.html. [Accessed: 27- Feb- 2015].