Asymptus - A Tool for Automatic Inference of Loop Complexity

Junio Cezar, Francisco Demontiê, Mariza Bigonha and Fernando Pereira

UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte

{juniocezar,demontie,mariza,fernando}@dcc.ufmg.br

Abstract. *Complexity analysis is an important activity for software engineers. Such an analysis can be specially useful in the identification of performance bugs. Although the research community has made significant progress in this field, existing techniques still show limitations. Purely static methods may be imprecise due to their inability to capture the dynamic behavior of programs. On the other hand, dynamic approaches usually need user intervention and/or are not effective to relate complexity bounds with the symbols in the program code. In this paper, we present a tool which uses a hybrid technique to solve these shortcomings. Statically, our tool determines: (i) the inputs of a loop, i.e., the variables that control its iterations; and (ii) an algebraic equation relating the loops within a function. We then instrument the program to output pairs relating input values and number of operations executed. By running the program over different inputs, we generate sufficient points for a polynomial interpolator in order to precisely determine a complexity function for loops. In the end, the complexity function for each loop is combined using an algebra of our own craft. We have implemented this tool using the LLVM compilation infrastructure. We correctly analyzed 99.7% of all loops available in the Polybench benchmark suite. These results indicate that our technique is an effective and useful way to find the complexity of loops in high-performance applications.*

Demo URL: https://youtu.be/pzfrIDfoCEc

1. Introduction

Complexity analyses show how algorithms scale as a function of their inputs. The importance of complexity analysis stems from the fact that such a technique helps program developers to uncover performance bugs which would otherwise be hard to find. In addition to this, complexity analysis supports the decision of offloading or not computation to the cloud or GPU. Finally, this kind of technique has implications to the theoretical computer science community, as it provides data that corroborate the formal asymptotic analysis of algorithms. Given this importance, it comes as no surprise that, since the 70s [Wegbreit 1975], large amounts of effort have been spent in the design and improvement of empirical methodologies to infer the complexity of code.

Over the time, different static approaches were proposed to analyze programs in functional [Wegbreit 1975, Le Métayer 1988, Rosendahl 1989, Debray and Lin 1993] and imperative [Gulavani and Gulwani 2008, Gulwani et al. 2009b, Gulwani et al. 2009a] languages. Although the static approaches have the benefit of running fast and may give correct upper bounds, this methodology has shortcomings.

Static analyses may yield imprecise – or even incorrect – results. This imprecision happens due to the inherently inability of purely static approaches to capture the dynamic behavior of programs. In order to circumvent this limitation of static approaches, the programming language community has resorted to profiling-based methodologies [Goldsmith et al. 2007, Zaparanuks and Hauswirth 2012, Coppa et al. 2012]. However, even these dynamic techniques are not free of limitations.

The main drawback of a profiling-based complexity analysis is the fact that it is usually ineffective to relate the symbols in the program text to the result that it delivers. For instance, the state-of-the-art tool in this field is `aprof` [Coppa et al. 2012]. Aprof furnishes programmers with a table that relates input sizes with the number of operations performed. This modus operandi has two problems, in our opinion. First, the input is provided as a number of memory cells read during the execution of a function. This number may not be meaningful to the programmer, as we will clarify in Section 2. Second, it works at the granularity of functions. However, developers are often more interested in knowing the computational complexity of small regions within a function. Such regions can be, for instance, performance-intensive loops. This paper addresses these two limitations of input sensitive profiling.

The main contribution of our work is a tool named Asymptus, which uses a novel hybrid technique to perform complexity analysis on imperative programs. Our technique is hybrid because it combines static analysis with dynamic profiling. First, we use static analysis to determine loop inputs and to find algebraic relations between these loops. Then, we use a dynamic profiler, plus polynomial interpolation, to infer the complexity of each loop in a function. Our technique is capable of generating symbolic expressions that denote the complexity of each loop, instead of the whole function. Furthermore, we combine and simplify these expressions to make them even more meaningful to the software engineer. We believe that this granularity can help developers to have a deeper understanding of a function's behaviour; hence, it provides them with the means to detect and solve performance bugs more efficiently. We also show that our technique is simpler than previous work while producing more useful results.

We have designed, tested, and implemented a tool on top of the LLVM compilation infrastructure [Lattner and Adve 2004] to infer, automatically, the complexity of loops within programs. We ran our tool over the Polybench [Pouchet 2012] and Rodinia [Che et al. 2009] benchmark suites. Our results indicate that we are capable of correctly inferring the complexity of 99.7% of the Polybench loops and 69.18% of the Rodinia loops. All the equations that we output, as explained in detail in Section 2, are written as functions of the symbols, i.e., variable names, present in the program code – that is an improvement on top of `aprof` and similar tools. Moreover, we have found that 38% of all functions in the benchmarks that we analyzed have at least two independent loops. In this case, tools that only report complexity information for entire functions may miss important details about the asymptotic behaviour of smaller regions of code.

2. Overview

In this section we give an overview of the challenges Asymptus addresses. Figure 1 shows the example we will use to illustrate our technique. Function *multiply* is a routine that performs matrix multiplication of two square matrices. For pedagogical purposes,

```
1: void multiply(int **matA, int **matB, int n){
2:     int i, j, k, sum;
3:     int **result = (int**) malloc(n * sizeof(int*));
4:     for (i = 0; i < n; i++)
5:       result[i] = (int*) malloc(n * sizeof(int));
6:
7:     for (i=0; i < n; i++) {
8:       for (j=0; j < n; j++) {
9:         sum = 0;
10:        for (k=0; k < n; k++) {
11:          sum += matA[i][k] * matB[k][j];
12:        }
13:        result[i][j] = sum;
14:      }
15:    }
16:
17:    j = 0;
18:    for (i = 0; i < n;) {
19:      if (j >= n) {
20:        j = 0;
21:        i++;
22:        printf("\n");
23:      } else {
24:        printf("%8d", result[i][j++]);
25:      }
26:    }
27:    printf("\n");
28: }
```

Figure 1. Matrix multiplication – the running example that we shall use to explain our contributions.

our function does not return the resulting matrix; instead, it prints the result. We chose to implement the function in such a way to show how our technique behaves on functions with multiple loops.

As developers, we would like to know the computational cost to execute this function. For instance, knowing the complexity of each part of the target function, we can find out performance bottlenecks and improve its implementation. Looking at the *multiply* function we can easily identify the linear behavior of the loop on line 4 and the cubic behavior of the nested loops beginning at line 7. However, a quick visual inspection on the loop at line 18 may not capture its quadratic complexity.

We can use profilers to find out where the program is spending most of its resources. However, traditional tools lack the ability to show how the program scales as a function of its inputs. For instance, Figure 2 shows the output that Gprof [Graham et al. 1982] – the most well-known profiler in the Unix systems – produces for our example. This profiler does not give us any information regarding the asymptotic complexity of the program in Figure 1. Instead, it produces a table describing

```
index % time    self  children   name
[1]    100.0    0.00    0.03     main [1]
                0.03    0.00     multiply(int**, int**, int) [2]
                0.00    0.00     initArray(int**, int, int) [9]
                0.00    0.00     free_all(int**, int**, int) [10]
-----------------------------------------------
```

Input size	Average Cost
138	1111
174	3324
286	12007
367	18576
463	26694
575	36394
701	47749
846	60781
1007	75587

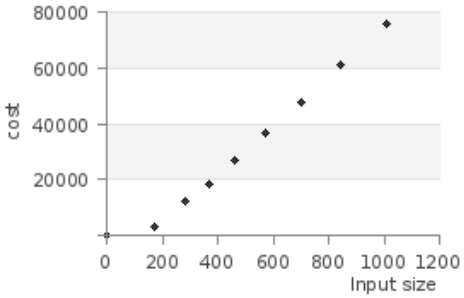

Figure 3. The output produced by the aprof input sensitive profiler.

where the program spends more time during its execution.

There exist profilers that have been designed specifically to provide developers with an idea about the asymptotic complexity of programs [Goldsmith et al. 2007, Zaparanuks and Hauswirth 2012, Coppa et al. 2012]. Nevertheless, `aprof` [Coppa et al. 2012], the state-of-the-art approach in this field, is also not very useful in this example. For instance, only looking at Figure 3, which shows `aprof`'s results for the function *multiply*, the user may not fully understand the function behaviour: this table shows numbers, but do not relate these numbers with symbols in the program text. Moreover, the complexity curve seems to be linear, since `aprof` considers the whole matrices as inputs (n^2) – usually, developers describe asymptotic complexity in terms of the matrices dimentions (n). Finally, the result generated by `aprof` describes the whole function. We believe that this granularity is too coarse, because it makes it very difficult for the user to verify the behavior of particular parts of the function.

We can do better: the technique that we describe in this paper produces one polynomial for each loop in the function. These polynomials range on symbols defined in the program text, e.g., the names of variables. Therefore, we claim that our output is clearer to the developer. For instance, considering the loop in line 7, we will state – automatically – that its complexity is:

$$n + 1$$

Furthermore, considering the loop nest starting in line 18, we produce the following equa-

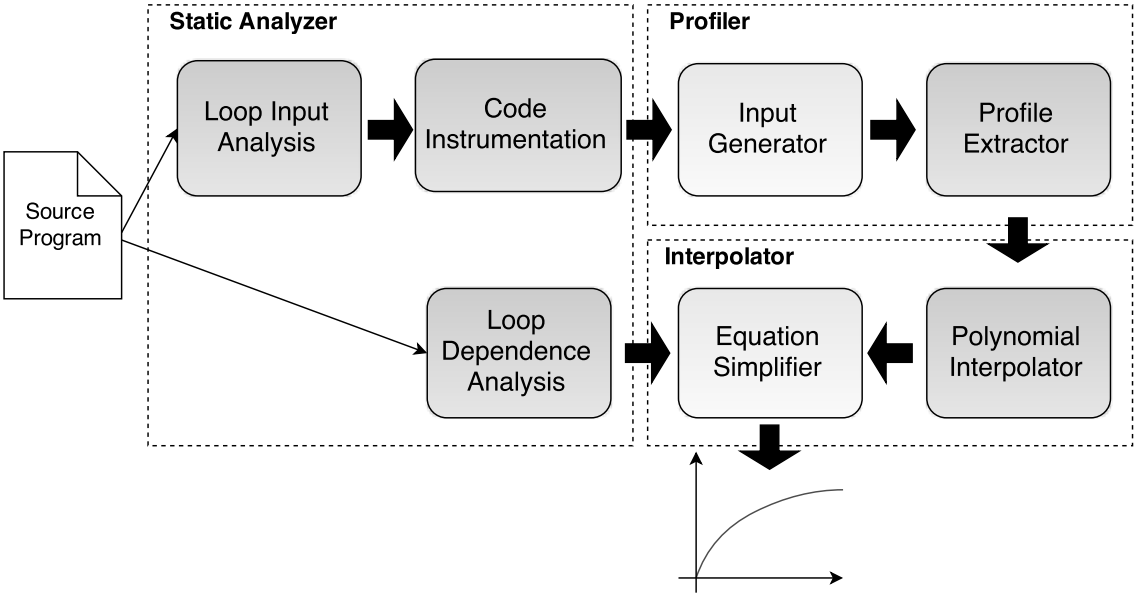

Figure 4. The Asymptus architecture. The arrows represent data flow. This diagram shows how the components of the tool work together.

tion to denote its complexity:

$$n^2 + n + 1$$

Our result is on a finer granularity, so we can combine them to generate an equation that expresses the asymptotic behavior of the whole target function. For the function in the Listing 1, our approach generates the following simplified equation, in big O:

$$O(n^3)$$

We claim that this notation, which uses the names of variables present in the program, is more meaningful to the application developer than the output produced by traditional profilers, such as *gprof* or `aprof`.

3. The Asymptus Tool

Asymptus, the tool that we describe in this paper, consists of four main steps: (1) static analysis, (2) code instrumentation, (3) dynamic information extraction and (4) polynomial interpolation. In this section we describe each one of these steps. Figure 4 shows a simple architecture of Asymptus.

3.1. Input Analysis

We start the process of inferring the complexity of code with a static analysis phase. The static analysis determines the inputs of each loop in the function. We qualify as *loop input* any data that: (i) influences the stop condition of the loop; and, (ii) is not defined within the loop. For instance, the loop at line 7 in Figure 1 is controlled by $i < n$. Variable i has two definitions: one outside the loop, which we shall call i_0, and another inside, which we shall call i_1. The former is initialized with the constant zero, which is thus considered a loop input. Variable n is a parameter of the function; hence, it is considered a symbolic

input. Therefore, the two inputs of the loop that exists at line 7 are $\{0, n\}$. Concretely, we detect inputs through a *backward* analysis, that starts at the variables used in the loop's stop condition, and ends at the definitions of variables that lay outside the loop body.

3.2. Loop Dependence Analysis

Our profiler outputs the complexity of all the loops within a program. We must combine this information to have a snapshot of the program's complexity. For doing this, we statically analyze control dependency relations between the loops within a function. We use these relations to determine an equation based on the big-O notation. For example, if two loops are in the same level and they are control equivalent, we can sum their complexities. As if we have a loop nest, the complexities can probably be multiplied. After we fill this equation with the loop complexities, the equation is simplified using an operators algebra based on the big-O semantics. For instance, for the function in Figure 1 our tool generates the following equation: $C(multiply) = C(L_{4-5}) + C(L_{7-15}) \times C(L_{8-14}) \times C(L_{10-12}) + C(L_{18-26})$. After the profiling phase, this equation is replaced by $O(n + n * n * n + n^2)$. It is easy to see that we can simplify this equation and have $O(n^3)$ as the resulting complexity.

3.3. Code Instrumentation

We infer the complexity of code by analyzing profiling data. We produce this data through code instrumentation. To be able to extract dynamic information, we instrument the target program to output: (i) the values of the loop inputs immediately before the loop execution and (ii) the number of operations performed by each loop. Loop inputs are determined by the analysis seen in Section 3.1. The execution cost is measured in terms of instructions executed. We have implemented this instrumentation framework within the LLVM compiler infrastructure. Care must be taken with regard to loops with multiple paths. Different paths may yield different costs, a fact that could hinder our interpolator from finding a perfect polynomial fit. To avoid this problem, we consider that the cost of a loop is determined by its path of highest cost, which we estimate statically. To obtain a conservative estimate of this path, we resort to a modified version of Dijkstra's algorithm, to solve the single-source largest path problem for an acyclic graph with non-negative weights assigned to edges [Dijkstra 1959]. Once we have instrumented the program, we execute it. As mentioned before, each execution of an instrumented program outputs the values of each loop input, together with the number of operations executed within that loop.

3.4. Polynomial Interpolation

We log the output of our profiler and parse it to extract pairs: input value $\times$ execution cost. With these points, we execute a polynomial interpolation method to find the curve that best fits into this set. For example, the program seen in Figure 1 has two blocks of loops; thus, we produce two polynomials. Let us take a deeper look into the polynomial that we produce for the loop that exists at lines 18-25 of Figure 1. In this example, we have obtained, after profiling the program with eight different inputs, the following pairs of size $\times$ cost: (13, 183), (50, 2,551), (72, 5,257), (80, 6,481), (98, 9,704), (115, 13,341), (139, 19,461). Our interpolator produces the polynomial $n^2 + n + 0.8$ The complexity of the loop is then $O(n^2)$, where n is the only symbolic input of the loop under analysis, as

we have explained in Section 3.1. We perform similar process to discover the polynomial that characterizes the loop nest at lines 7-15 of Figure 1.

3.5. Usage

Since Asymptus was implemented using the LLVM compilation infrastructure, it runs over, either C/C++ source files, or LLVM bytecodes. We give the user the possibility to specify the inputs for the target program in three ways: (i) providing real data as inputs for the program, (ii) letting Asymptus generate random values or (iii) a mix of real data and random values. To let Asymptus to generate random values, the user have to use the option `--args` specifying the type of each command line argument of the program. The types may be one of the following: int, long, float, double, num (a numeric value without specifying the specific type), char or string. To mix concrete and random values, there is the option `--mix`. In this case, the types for the random values have to be specified within '{}'. For example, `--mix concrete1 {int} concrete2 {string}`. To execute the program with only concrete data, the user has to use the option `--man`. When using this option, Asymptus asks the user for the input values for each execution. An empty line marks the end of the inputs. By default, Asymptus shows results in the function level. To show the results for each loop within the program, the user has to specify the option `-v`. Figure 5 shows an output example. The function *multiply* is the one of Figure 1. Function *init* creates an array with dimensions of size passed as argument.

```
Function 'multiply(int**, int**, int)':
    Loop at line 6: 1.00 * n + 1.00
    Loop at line 9: 1.00 * n + 1.00
    Loop at line 10: 1.00 * n + 1.00
    Loop at line 12: 1.00 * n + 1.00
    Loop at line 20: 1.00 * n^2 + 1.00 * n + 1.00

    Complexity: O(n^3)

Function 'init(int)':
    Loop at line 38: 1.00 * size + 1.00
    Loop at line 42: 1.00 * size + 1.00
    Loop at line 43: 1.00 * size + 1.00

    Complexity: O(size^2)
```

Figure 5. Asymptus output example using verbose mode and random inputs.

4. Conclusion

This paper presented Asymptus, a publicly available tool which uses a new technique based on a combination of profiling and static analysis, to infer the complexity of code. Static analysis gives us the names of variables that bound the trip count of loops. Profiling lets us associate these variables with the number of operations in the loops that they control. We believe that our approach, whenever applicable, yields results that are more meaningful to the application developer than the state-of-the-art tools that are currently available. Asymptus has three main limitations: (i) it can only analyze functions reached by the execution flow of the program, (ii) it only generates results for a function if it can find enough data-points (i.e. if the function is executed with different inputs) and (iii) only work for loops with polynomial complexities. There are several ways in which such a tool can be employed. Our immediate goal is to use it to help in the automatic placement of code in non-uniform memory access architectures. In this scenario, it is worthwhile to

migrate processes of high computational cost closer to the memory banks that contain the data that said processes use. A totally static solution has been devised to this problem by Piccoli *et al.* [Piccoli et al. 2014]. Our intention is to add to this solution a dynamic component based on this paper's ideas, in hopes to increase its precision.

References

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE.

Coppa, E., Demetrescu, C., and Finocchi, I. (2012). Input-sensitive profiling. In *PLDI*. ACM.

Debray, S. K. and Lin, N.-W. (1993). Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *FSE*, pages 395–404. ACM.

Graham, S. L., Kessler, P. B., and McKusick, M. K. (1982). gprof: a call graph execution profiler (with retrospective). In *Best of PLDI*, pages 49–57.

Gulavani, B. and Gulwani, S. (2008). A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, volume 5123 of *LNCS*, pages 370–384. Springer.

Gulwani, S., Jain, S., and Koskinen, E. (2009a). Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385. ACM.

Gulwani, S., Mehra, K. K., and Chilimbi, T. (2009b). SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Le Métayer, D. (1988). Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266.

Piccoli, G., Santos, H., Rodrigues, R., Pousa, C., Borin, E., and Pereira, F. M. Q. (2014). Compiler support for selective page migration in NUMA architectures. In *PACT*, pages 369–380. ACM.

Pouchet, L.-N. (2012). Polybench: The polyhedral benchmark suite. URL: http://www.cs.ucla.edu/ pouchet/software/polybench/. Last access: April, 2015.

Rosendahl, M. (1989). Automatic complexity analysis. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 144–156, New York, NY, USA. ACM.

Wegbreit, B. (1975). Mechanical program analysis. *Commun. ACM*, 18(9):528–539.

Zaparanuks, D. and Hauswirth, M. (2012). Algorithmic profiling. In *PLDI*, pages 67–76. ACM.