Automatic Inference of Loop Complexity through Polynomial Interpolation

Francisco Demontiê, Junio Cezar, Mariza Bigonha, Frederico Campos and Fernando Pereira

UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte
{demontie,juniocezar,mariza,ffcampos,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 behaviour 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 hybrid technique that solves these shortcomings. Our technique uses a numeric method based on polynomial interpolation to precisely determine a complexity function for loops. Statically, we determine: (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 plot a curve relating inputs and number of operations executed. By running the program over different inputs, we generate sufficient points for our interpolator. In the end, the complexity function for each loop is combined using an algebra of our own craft. We have implemented our technique in the LLVM compiler, being able to analyse 99.7% of all loops available in the Polybench benchmark suite, and most of the loops in Rodinia. These results indicate that our technique is an effective and useful way to find the complexity of loops in high-performance applications.

1 Introduction

Complexity analyses show how algorithms scale as a function of their inputs. Its importance stems from the fact that such a technique helps program developers to uncover performance bugs which are 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 [20], large amounts of effort have been spent in the design and improvement of empirical methodologies to infer code complexity.

Over the time, different static approaches were proposed to analyze programs in functional [20, 15, 6] and imperative [11, 13, 12] 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 [9, 22, 4]. 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` [4]. 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 novel hybrid technique to perform complexity analysis on imperative programs, which we describe in Section 3. 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 [14] to infer, automatically, the complexity of loops within programs. We ran our tool over the Polybench [19] and Rodinia [3] benchmark suites. Section 4 reports our findings. 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.

```
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: }
```

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

2 Overview

In this section we give an overview of the challenges this paper 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, 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

```
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]
-----------------------------------------------
                0.03    0.00     main [1]
[2]    100.0    0.03    0.00     multiply(int**, int**, int) [2]
-----------------------------------------------
```

Fig. 2: Gprof output for a simple program containing our example function.

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

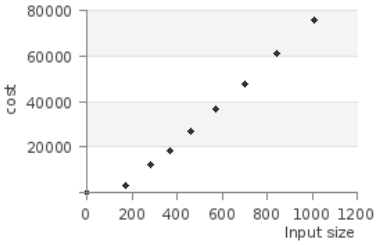

Fig. 3: The output produced by the aprof input sensitive profiler.

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 [10] – 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 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 [9, 22, 4]. Nevertheless, `aprof` [4], the state-of-the-art approach in this field, is also not very useful in this example. For instance, only looking at Figure 2, which shows `aprof`'s results for the function *multiply*, the user may not fully understand about 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 polynomial is: $n+1$. Furthermore, considering the loop nest starting in line 18, we produce the following equation to denote its complexity polynomial: $n^2 + n + 1$.

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, to denote the function's complexity:

$$O(n^3)$$

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 Complexity Analysis

We can describe our technique in 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. However, before delving into the details of our technique, we shall introduce some notation, which will guide our explanations henceforth.

Loop Jargon. Let S be a subset of nodes of a control flow graph G. S contains a special node H, which we shall call *header*, or *entry point*. Following Appel and Palsberg [2, pp.376], we say that S is a *natural loop* if, and only if, it presents the following three properties:

1. there exists a path from any node in S to H;
2. there exists a path from H to any node in S;
3. there is no path from a node of G to a node of S that does not go across H.

The last property defines S as a *single-entry* region, following Ferrante's nomenclature [8]. An edge between any node in S to H is called a *back-edge*. We adopt Wolfe's definition of *trip count* [21, pp.200]: the number of times any back-edge of a natural loop has been traversed by the program flow within a single execution of the loop. Hence, a loop that exits the first time it is executed has a trip count of zero. The number of times H is visited is one more than the trip count of the loop. We estimate the *complexity* of a loop as the product of its trip count by the number of operations in its longest path.

We call a node $L \in S$ a *latch*, or *exit point*, if there exists an edge from L to a node N, $N \in G$, $N \notin S$. We say that L is a *natural latch* if one of these two conditions applies:

– $L = H$. In this case we have a *while loop*;
– $L \neq H$, and any edge from L either leaves S or leads to H. In this case we have a *repeat loop*.

If S contains only one latch, then we call it *single exit*. In this work we consider multiple exit loops featuring only one natural latch. Code generated from typical programming language constructs, i.e., `for`, `while` and `repeat` has this property, as long as the command `goto` is not used.

Any latch contains a *stop condition*: a boolean expression whose evaluation either keeps the program flow in S or leads away from it. If the natural latch contains a stop condition that uses only one operator, which can be either $<$, $\leq$, $>$ or $\geq$, then we call S an *interval loop*. We let the operands of the stop condition be the *limits* of the interval. For instance, in the interval loop `for(i = 0; i < N; i++)`, we have the stop condition `i < N`, whose limits are `i` and `N`. Our technique handles any loop with only one input, and interval loops with up to two inputs i_1 and i_2. In this case, we consider as the input size the difference $|i_1 - i_2|$.

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:

– influences the stop condition of the loop; and,
– 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. To determine the complexity of a loop, we will plot the number of operations executed by the loop for each value bound to one of its inputs that we have observed during a profiling step. We shall describe this profiling in Section 3.3

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. However, combining the complexity of all the loops that constitute a program is not a straightforward problem. One of the main difficulties that we face in this case is how to deal with loops that may, or may not, execute, depending on the path that the program follows. In order to provide meaningful answers to the user,

```
 1: void printDups(std::vector<std::string> lines, std::string key) {
 2:   std::vector<std::string> result;
 3:   for (int i=0; i < lines.size(); i++) {
 4:     if (lines[i].find(key) != std::string::npos) {
 5:       result.push_back(lines[i]);
 6:     }
 7:   }
 8:
 9:   if (result.empty()) return;
10:
11:   // find dups in a naive way
12:   for (int i=0; i < result.size()-1; i++) {
13:     for (int j=i+1; j < result.size(); j++) {
14:       if (i != j && result[i] == result[j])
15:         std::cout << result[i] << std::endl;
16:     }
17:   }
18: }
```

Fig. 4: A function to print duplicate lines containing a given key. The second loop has a conditional execution.

we propose an algebra to simplify the equations that we produce. Our algebra has three operators: *plus* ($+$), *times* ($\times$) and *expander* ($\oplus$). The *plus* and *times* operators have the usual semantics of asymptotic analysis. The expander was proposed by us as an alternative to describe the complexity of code that may or may not execute, depending on the program's flow. Its semantics is defined in the equations 1 and 2:

$$O(x^a \oplus y^b) = O(x^a) + O(x^b), \quad \{a, b\} \in \mathbb{N} \tag{1}$$

$$\Omega(x^a \oplus y^b) = \Omega(x^a), \quad \{a, b\} \in \mathbb{N} \tag{2}$$

As a reminder, the big-Omega notation indicates a lower asymptotic bound: $\Omega(f)$ denotes a function whose growth is less than or equal to the growth of f. Expansion denotes the complexity of code that executes conditionally. Figure 4 provides an example of a situation where the expander operation is useful. The function *printDups* prints the duplicate lines containing a given substring in a naive way. Because of the conditional branch in line 9, the loop starting on line 12 may or may not execute. Because of this, the complexity of this function is $\Omega(n)$ - best case, when no line contains the key - and $O(n^2)$, where n is the size of the vector. If $C(L)$ denotes the asymptotic complexity of a given code region, then we let $C(\textit{printDups}) = C(L_{3-7}) \oplus C(L_{12-17}) = O(n \oplus n^2)$, where L_{3-7} is the loop at lines 3 to 7 in Figure 4, and L_{12-17} is the loop at lines 12 to 17.

As usual, addition and multiplication in the big-O notation are associative and commutative. Multiplication is also distributive with regard to addition. On

the other hand, *expansion* is only associative, due to Equation 2. These properties let us use typical simplification rules to provide users of our tool with more palatable results. Notice, once again, that expansion is non-commutative, and simplification only applies if the first operand has higher complexity than the second:

$$\frac{C(L) = O(x^a) + O(x^b), a \geq b}{C(L) = O(x^a)} \qquad \frac{C(L) = O(x^a) \times O(x^b)}{C(L) = O(x^{a+b})}$$

$$\frac{C(L) = O(x^a) + O(x^b), a < b}{C(L) = O(x^b)} \qquad \frac{C(L) = O(x^a) \oplus O(x^b), a \geq b}{C(L) = O(x^a)}$$

The simplification process is guaranteed to terminate, as it always reduces the size of the resulting expression. Looking back to Figure 1 it is easy to see that the complexity is $C(multiply) = C(L_{4-5}) + C(L_{7-15}) \times C(L_{8-14}) \times C(L_{10-12}) + C(L_{18-26})$, which gives us: $O(n + n * n * n + n^2)$. Using the above equations we can recursively simplify this expression. Firstly, we can simplify $n * n$ with n^2. We have now $O(n + n^2 * n + n^2)$ and we can use the same rule to simplify the remaining multiplication, resulting in n^3. It is easy to see that we can use the two rules of *plus* to simplify the two additions. Then, the resulting complexity is $O(n^3)$, as expected. Notice that n is a symbol produced by the input analysis of Section 3.1.

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. Figure 5 illustrates this shortcoming. The program seen in part (a) of the figure contains two loops, at lines 2 and 4. The loop at line 4 contains two execution paths. Let's assume that during execution, our profiler has observed that for $M = 1$, that loop executed 44 instructions, and for $M = 2$, it always took the cheapest path; hence, executing 3+3 operations. These points, $(1, 42), (2, 6)$ would confuse our interpolator, which expects more operations for larger inputs. 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 [7]. To build an acyclic graph, we consider all the paths from the loop header H to its natural latch L.

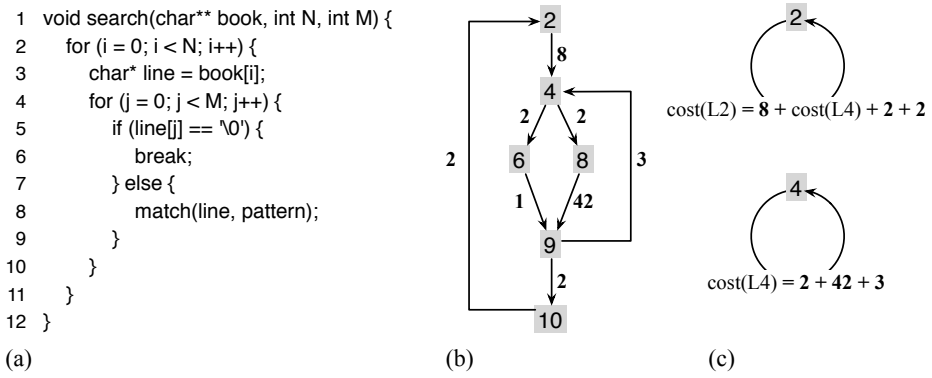

Fig. 5: (a) Program with a multi-path loop. (b) The cost-graph of the program. Nodes represent program points and the edges' weights represent the number of executed instructions between two points. (c) The cost of each loop iteration.

Once we have determined – statically – the cost of a loop iteration, we instrument it. To this end, we create a counter variable at the loop's header, and increment it by the estimated cost. Notice that incrementing this counter at the loop header will account for one more iteration than the real execution. Nevertheless, it will not affect our cost analysis. We chose to do it like this because the loop header is unique, and is always executed, independent on the way the program flows within the loop body. Figure 5 (c) shows the cost expressions that we create for each loop. In the figure, edges represent paths within the loop, and the nodes are the headers of those loops. Each one of these values is added once per iteration of the loop. 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. Our interpolation works as follows: we test different polynomials, starting from a line (degree 1) upwards until $n-1$, where n is the number of points available. At step i we need $i+1$ points to determine a polynomial. Any group of $i+1$ different points fits this purpose. We call this group of points the *guiding set*. We use the points that are left to check if we have found the correct polynomial. These remaining points are called the *verification set*. We stop interpolation if, upon finding a polynomial p, of degree k, $k < n-1$, we notice that the $n-k$ points in the verification set fit perfectly into p. Our interpolation only works for single-variable polynomials,

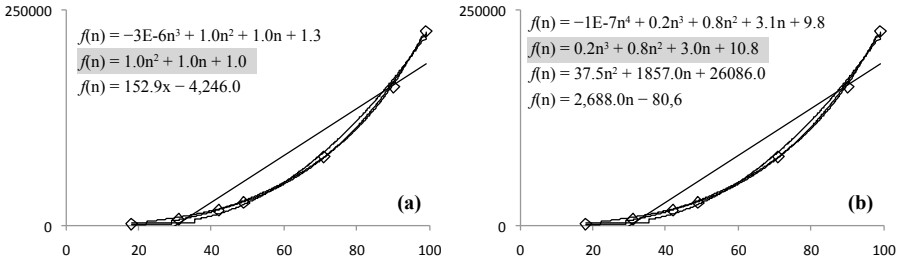

Fig. 6: (a) Polynomials found for the loop at lines 18-25 of Figure 1. (b) Polynomials found for the loop nest at lines 7-15. In each figure, the first curve that fits the points in the verification set is marked in gray.

but we can infer the complexity of nests of loops by multiplying symbolically their individual complexities.

Figure 6 illustrates this process for the program seen in Figure 1. The figure 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. This curve is shown in Figure 6 (a). In this example, we assume that we have obtained, after profiling the program with eight different inputs, the following pairs of size × cost: (13, 183), (50, 2,551), (72, 5,257), (80, 6,481), (98, 9,704), (115, 13,341), (139, 19,461). To derive a polynomial that describes the complexity of this loop, we try to interpolate a line across those points using, as our guiding set, only the first two pairs, e.g., (13, 183) and (50, 2,551). This line does not contain the other six points, which form the verification set. Thus, we move on to try a polynomial of degree two, this time, adding also the pair (72, 5,257) to our guiding set. The new polynomial, $n^2+n+0.8$ contains the points in our verification set. Hence, we let it denote the computational cost of the loop. 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. However, this time our search stabilizes in a third-degree polynomial. Figure 6 (b) shows this curve.

4 Experiments

To examine the real applicability of our technique, we have implemented it as a prototype tool. We have used the LLVM compilation infrastructure to perform the static analysis and code instrumentation phases mentioned in Sections 3.1, 3.2 and 3.3. All the experiments that we shall present in this section have been run on an Intel Xeon processor, with 16GB of RAM, running Linux Ubuntu. The main goals of these experiments are: (1) to find out how effective is the

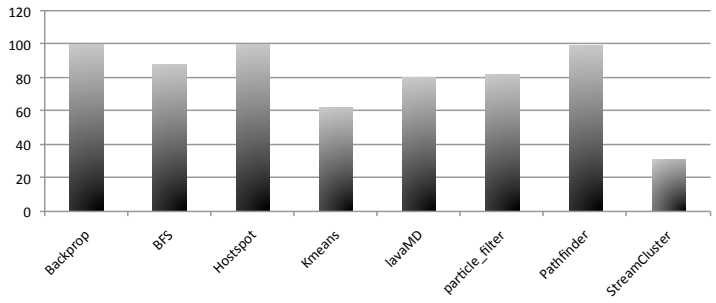

Fig. 7: Percentage of loops per benchmark of Rodinia that we could analyze. The correctness of all these results have been checked manually.

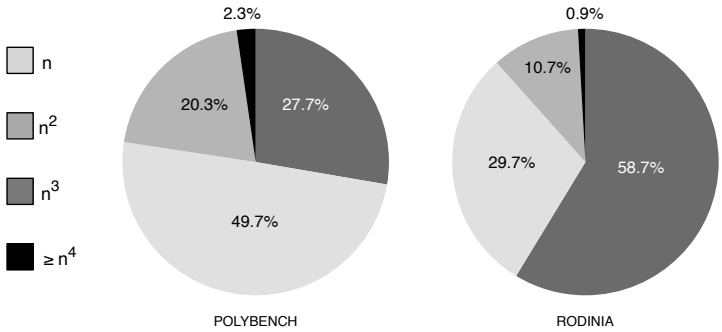

Fig. 8: Distribution of complexities. In this chart we ignore the difference between variables - we consider that $n \times m$ is equals to n^2, for instance.

technique when applied to the loops found in real-world programs; and (2) to provide a taxonomy of the loops found in real-world systems.

Effectiveness To achieve our first goal – to probe the effectiveness of our tool – we have executed it on the Polybench [19] and Rodinia [3] benchmark suites. We have checked, manually, the answers produced by our tool for every loop in these benchmarks. This exercise shows that we are able to correctly analyze 99.7% of the loops in Polybench. The remaining 0.3% is due to a single loop which is constant for the first two points, and varies for larger inputs. This behavior makes it impossible for us to get a perfect polynomial match. For Rodinia – a much bigger and general benchmark suite – our tool correctly analysed 63.58% of the loops. However, the execution flow never reached some functions during our profiling phase so we could not generate data for them. If we ignore those functions, our success rate increases to 69.18%.

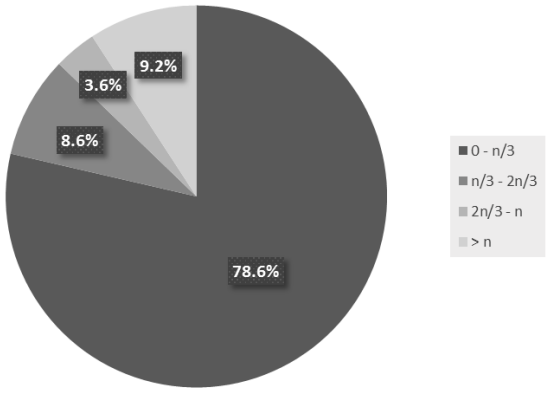

Fig. 9: Quality of the approximation heuristic seen in Section 3.3. Each slice groups a range of loops for which our approximation yielded similar results. For instance, for 78.6% of the loops the approximation yields a result that is within $[0, 1/3]$ of the observed value.

Our results are worse for Rodinia because of three reasons: (1) some loops are not polynomial, but we use a polynomial interpolation; (2) some loops iterate over structures that our technique does not handle, such as strings or files; and (3) some loops have 3 or 4 inputs that bound their execution. In this case, we do not generate pairs of input vs time for the loop. Figure 7 shows the percentage of loops that we could analyze per Rodinia benchmark. We do not show a chart for Polybench, because we believe that this chart is not interesting. It would have almost only bars at 100% of precision.

Given all the machinery that we now have in place, we thought that it would be interesting to categorize the loop nests that we have found in our benchmarks. Figure 8 shows the distribution of complexities found in both benchmark suites. The majority of loop nests in Rodinia are linear, and only a handful of them are $O(N^4)$ or higher. In Polybench, the picture is slightly different. Most of the loop nests in that collection are quadratic. This happens because Polybench has been designed to test optimizations built over the polytope model. Linear loops are simply not challenging enough to the current state-of-the-art polyhedron techniques.

The Topology of Loops. To better understand the power and limitations of the technique that we advocate in this paper, we chose to analyze in greater detail the topology of loops found in real-world programs. In addition to Rodinia and Polybench, this time we chose to study also the loops present in SPEC CPU 2006, to have a larger body of samples.

We have counted the number of independent loops within functions. We say that two loops, L_1 and L_2, are independent if one is not nested within

the other. We saw that 21.8% of the functions have at least two independent loops in Polybench, 40.4% in Rodinia, and 38.1% in SPEC. These numbers let us conclude that it is important, from a software engineering point of view, to output complexity results in a finer grain than functions, as `aprof` does. We do it at the loop level. This finer granularity gives developers more information to understand a function's behaviour. We have also counted the number of loops that are executed conditionally within a function. We found 92 control-flow breaks (e.g. returns or exit calls) in the 99 functions that we have analyzed. This data shows that if we ignore conditional execution, then we may output incomplete – or incorrect – results. That is why we use the *expander* operator.

The last metric that we have studied is the number of loops with multiple paths. We saw that 51.2% of the loops in SPEC have multiple paths. We also would like to know how far from the exact number of instructions we stay when using the approximation seen in Section 3.3. In that case, we approximate the cost of a loop as the cost of its longest path. By profiling the actual number of instructions executed in our benchmarks, we got that, most of the time, our approximation is within 33% of the actual result. This metric shows that using the heuristic from Section 3.3 increases the applicability of our analysis without compromising its results. Figure 9 shows a distribution of how distant our approximation is from the real program behavior.

5 Related Works

Recent work has attempted to improve the state of the art on complexity analysis. Particularly, profiler-based approaches have been able to give interesting results. Goldsmith *et al.* [9] proposed a technique which consists in executing the target program over workloads with different orders of magnitude and tracking how many times each program location was executed. They use polynomial regression to fit the data into a linear or power-law model. However, the user has to specify, for each workload, the value of features - a feature is an input property which affects the algorithm execution, e.g. the size of an array or the height of a tree. Our technique is able to automatically infer loops' inputs; hence, it does not require this type of user intervention.

Zaparanuks *et al.* [22] proposed the concept of algorithmic profiler. Their approach consists in grouping the basic blocks of a loop and the functions which make a cycle in the call-graph into the so called *repetition nodes*. Those nodes are then combined in units that they have named *algorithms*. The technique is able to identify if an algorithm is modifying or traversing a list or an array, for example. In order to estimate the complexity of an algorithm, they retrieve the size of the inputs and some performance metrics for each execution of the repetition nodes. This modus operandi leads to a significant overhead, since the analyzer iterates over the entire data structure to calculate its size. The automatic reconstruction of data-structures is still an incipient area of research. Therefore, Zaparanuks *et al.* have implemented a prototype which, up to this point, can analyze only toy examples. We cannot reconstruct recursive data-structures as Zaparanuks does;

however, our approach is able to infer the complexity of most of the loops in a real-world benchmark suite.

The work that is the most related to ours is Coppa *et al.*'s input sensitive profiler [4]. This work has materialized itself into `aprof` tool. Core to `aprof`'s work is the notion of Read Memory Size (RMS). This metric represents the number of memory locations which are read before they have been written inside a function. Aprof was implemented as a *Valgrind* [17] extension. We believe that `aprof` is the most practical tool available nowadays to infer the complexity of general purpose programs. Nevertheless, it has the shortcomings which we have described in Section 1: (i) the granularity of results is at the function, not at the loop, level; (ii) users have to fit equation by hand in `aprof`'s results to find the complexity of a function; and (iii) results are given in terms of RMS, which may not be significant to the developer. Our technique is capable of addressing these drawbacks.

There exists a plethora of work related to the static estimation of complexity of code [1, 5, 11, 13, 16]. Our work is essentially different from these approaches, because our results are based on program behavior observed at runtime. In other words, our approach is dynamic: we execute and profile the program to infer its computational complexity. The downside of our approach is that we are not able to *prove* properties about the program's complexity: there are no guarantees that we will be able to observe every possible execution path within the program code. The upside is precision: our approach is able to reason about typical programming language features such as dynamically allocated memory, multiple paths in loops, non-structured control flow graphs and pointer arithmetics. So far, these real-world constructs have been challenging adversaries to the purely static analyses.

6 Conclusion

This paper has presented 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. A tool that implements the technique is publicly available[1] for use. 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.* [18]. Our intention is to add to this solution a dynamic component based on this paper's ideas, in hopes to increase its precision.

[1] http://demontiejr.github.io/asymptus

References

1. Péricles Rafael Oliveira Alves, Raphael Ernani Rodrigues, Rafael Martins de Souza, and Fernando Magno Quintão Pereira. A case for a fast trip count predictor. *Inf. Process. Lett.*, 115(2):146–150, 2015.
2. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, 2nd edition, 2002.
3. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE, 2009.
4. Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In *PLDI*. ACM, 2012.
5. Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL*, pages 133–144. ACM, 2008.
6. Saumya K. Debray and Nai-Wei Lin. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826–875, November 1993.
7. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
8. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.
9. Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *FSE*, pages 395–404. ACM, 2007.
10. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler (with retrospective). In *Best of PLDI*, pages 49–57, 1982.
11. BhargavS. Gulavani and Sumit Gulwani. 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, 2008.
12. Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385. ACM, 2009.
13. Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.
14. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.
15. Daniel Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, April 1988.
16. David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *SAS*, pages 369–385. Springer, 2011.
17. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.
18. Guilherme Piccoli, Henrique Santos, Raphael Rodrigues, Christiane Pousa, Edson Borin, and Fernando Magno Quintão Pereira. Compiler support for selective page migration in NUMA architectures. In *PACT*, pages 369–380. ACM, 2014.
19. Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite, 2012. URL: http://www.cs.ucla.edu/ pouchet/software/polybench/. Last access: April, 2015.
20. Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, September 1975.
21. Michael Wolfe. *High Performance Compilers for Parallel Computing.* Adison-Wesley, 1st edition, 1996.
22. Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *PLDI*, pages 67–76. ACM, 2012.