Bounding the Execution Cost of WebAssembly Functions

John Shortt[1], Amy Felty[1], and Anil Somayaji[2]

[1] University of Ottawa, Ottawa ON, Canada
[jshor018,afelty]@uottawa.ca
[2] Carleton University, Ottawa ON Canada soma@scs.carleton.ca

Abstract. Bounds on worst-case execution time can improve system reliability and security in a variety of contexts. Past work on bounding execution time has faced challenges due to the lack of formal specifications of mainstream computing environments. WebAssembly is a low-level language originally designed for efficient execution in browsers that is used today in edge computing and other environments. WebAssembly has been formally specified and has a mechanized soundness proof, greatly facilitating the formal analysis of WebAssembly programs. Here, we present a new tool for bounding the worst-case execution time of WebAssembly functions that is based on the formalization of WebAssembly. We have tested our tool on significantly more and larger functions than those studied in previous work (over 107,000 functions, with the longest function being 4156 lines of WebAssembly from 392 lines of C), and it successfully and efficiently analyzed the vast majority of functions tested. Progress in our tool suggests the feasibility of calculating worst-case execution bounds on large real-world code bases.

Keywords: Program Analysis, Execution Cost Analysis, Formal Verification, WebAssembly

1 Introduction

In the context of web applications, edge computing, digital contracts, and numerous rich document formats, systems execute untrusted code. If not properly contained, such code can consume arbitrary resources, resulting in denials of service, battery exhaustion, information theft, and application and host compromises. Language and operating system sandboxes can limit the potential damage of malicious code, but to accommodate increasingly complex applications, such sandboxes must allow code to have significant CPU and memory resources. Fixed limits can reduce but not eliminate the risk of giving arbitrary code access to CPU and memory resources. Worst-case execution time (WCET), however, offers an alternative defense strategy: by calculating the worst-case execution time of a program in advance, it becomes possible to decide not to run code if it will consume too many resources. As we know, this problem cannot be solved in

the general case (otherwise we would have a solution to the halting problem [21]), but we also know that formally defined systems can be reasoned about with sophisticated tools that allow us to make interesting conclusions about the behavior of those systems.

Today WebAssembly [17,25] is the leading technology for untrusted code written in arbitrary programming languages. WebAssembly is widely deployed in web browsers, is an enabling technology for edge computing, and is finding applications in other domains. Two properties in particular contribute to offering a path toward bounding the execution time of programs: it is low level and it is formally specified. With regard to the first property, WebAssembly can serve as a compilation target much like CPU-specific assembly languages and bytecode languages like the Java Virtual Machine (JVM) and Common Language Runtime (CLR). WebAssembly's appeal comes from highly sandboxed yet very efficient runtimes, something that is not generally available for other compiler targets. The second key property, the fact that WebAssembly has been formally specified, greatly facilitates program analysis. The formal specification assures that WebAssembly has no undefined behavior, and its program control flow mechanisms use the ideas of structured programming in a way that also simplifies reasoning about them. Other compiler targets have been formalized; however, these formalizations are post-hoc and often only approximate the functioning of the language in practice. In contrast, WebAssembly has been formally specified from the beginning, thus making it an important application area for program analysis techniques both because of potential practical applications and because it has been designed to facilitate formal program analysis.

Our tool, which we call **Wanalyze**, and is available on GitHub[3], shows the potential of WebAssembly to simplify formal program analysis, for example by eliminating the need to detect loops (they are evident in the structured control flow) and reducing the need for directly analyzing complex data structures (WebAssembly has few data types and simple data structures), while still allowing large-scale production code to be analyzed. Further, because WebAssembly itself is formally specified, our results apply not to idealized execution environments (as is often the case for C variants), but to production runtimes.

As a step towards this goal, we have chosen to focus on analyzing individual functions rather than entire programs. Specifically, the scope of our work is the static analysis of each function in a WebAssembly module to determine a bound on its cost of execution. As we discuss later, previous work in the literature shows how this can be extended to whole programs [2,3,4,6]. Although we do not fully handle whole programs, we are able to analyze functions that are much larger than the programs analyzed in previous work (up to four times larger) and for over 107,000 functions from real-world code bases, giving us a breadth of experience significantly beyond that of past work.

In this paper we present the methods we have developed for calculating the worst-case cost of WebAssembly functions and our efforts to validate our methods. **Wanalyze** is implemented in OCaml, consists of approximately 5000 lines

[3] `https://www.github.com/jsCarleton/wanalyze`

of code, and is licensed under the Apache 2.0 license and available for download. We have evaluated **Wanalyze** on WebAssembly programs and libraries of varying complexity and have found that it is able to successfully produce execution bounds for the vast majority of these functions, while being able to analyze thousands of lines of code per second in our tests run on relatively modest hardware.

The contributions of this paper consist of the software tool, **Wanalyze**, which is the first published application that can bound the worst-case cost of WebAssembly functions, demonstrating the feasibility of bounding real-world code with the analysis of over 107,000 functions.

In the rest of this paper, Section 2 contains a motivating example for the techniques that we use. Section 3 contains details of our analysis techniques. In Section 4 we describe the experimental tests that have been performed and their results. We discuss related work in Section 5. Section 6 concludes.

2 Example

By way of example, we examine the WebAssembly code for the inner loop of a bubble sort implementation. Listing 1 contains a C implementation and Listing 2 contains the skeleton of the WebAssembly code to which this C code compiles. The latter shows the branching structure of the WebAssembly code and is annotated with comments that define the blocks of code in that structure. It also includes annotations (labels) that define the instructions that can be branched to and the destination of a branching instruction.

```c
void bubble(int n, int* data) {
  int i, j, temp;

  for(i = 0; i < n - 1; i++) {
    for(j = 0; j < n - i - 1; j++) {
      if(data[j] > data[j + 1]) {
          temp = data[j];
          data[j] = data[j + 1];
          data[j + 1] = temp;
      }
    }
  }
}
```

Listing 1: C code to implement bubble sort

```
  (func (;6;) (type 6) (param i32 i32)
    (local i32 i32 i32 i32 i32 i32 i32 i32)
    ;; BB 0
    block  ;;label = @1
```

```
5       ;; BB 1 (deleted 3 lines)
9         br_if 0 (;@1;)
10        ;; BB 2 (deleted 7 lines)
18        loop  ;;label = @2
19          ;; BB 3 (deleted 2 lines)
22          block  ;;label = @3
23            ;; BB 4 (deleted 7 lines)
31            br_if 0 ;;label = @3
32            ;; BB 5
33            loop  ;;label = @4
34              ;; BB 6
35              block  ;;label = @5
36              ;; BB 7
37                ;; (deleted 20 lines)
57                br_if 0 ;;label = @5
58                ;; BB 8 (deleted 6 lines)
65              end
66              ;; BB 9 (deleted 3 lines)
70              br_if 0 ;;label = @4
71              ;; BB 10
72            end
73            ;; BB 11
74          end
75          ;; BB 12 (deleted 10 lines)
86          br_if 0 ;;label = @2
87          ;; BB 13
88        end
89      ;; BB 14
90      end
91    ;; BB 15
92  )
```

Listing 2: Outline of WebAssembly code to implement bubble sort

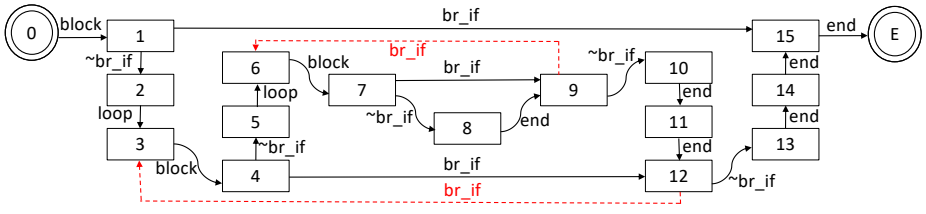

Fig. 1: Basic block control flow diagram for bubble sort

Figure 1 shows the branching structure of Listing 2 in the form of a control flow diagram. In this diagram, nodes represent *basic blocks* (BBs) and edges are possible execution paths. Red-dashed edges represent backward execution paths to the beginning of a loop. Figure 2 shows how we can further refine this

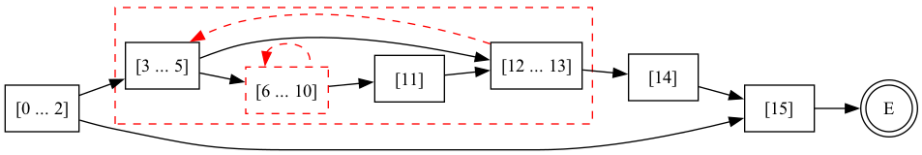

Fig. 2: Super block control flow diagram for bubble sort

block structure by merging consecutive blocks when there is only one code path through them. In doing so we retain information about the flow of control in the function and create a higher level control flow diagram with fewer nodes. In this diagram nodes represent *super blocks* (SBs) and, again, edges are possible execution paths. There are two types of superblock: those that do not contain a loop, shown as a rectangle with a black border, and those that do, shown with a red dotted border. The term basic block comes from the compiler and static analysis literature [13,14]. We define these block types precisely for WebAssembly in the next section.

Lines 37 through 70 of Listing 2 contain the code that implements the body of the inner-most loop of bubble sort. Lines of code are deleted in the listing for presentation purposes. Table 1, column (b) contains all of this code, including the missing lines from the listing, with line numbers in column (a). In this table `n[i]` and `m[i]` refer to local variable and memory location `i`, respectively. Column (c) is the WebAssembly value stack contents after the instruction is executed and any variable or memory changes that the instruction causes. We use the convention that the left-most item in the stack is the one most recently added. Lines with comments in the listing are omitted from the table. To improve readability, we also convert the label indices in branch instructions to line numbers. For this table, it is sufficient to know that n is an array containing function parameters and local variables, and m is an array representing WebAssembly memory.

In summary, this code behaves as follows: in lines 37 through 49 the 2 values to be compared are loaded into temporary variables, and the loop counter is updated (line 49), in lines 50 through 65 the two values are swapped if they are not in the correct order, and in lines 67 through 70 the loop counter is compared to the loop bound to determine if another loop pass is required. The variable `n[5]` is used for the loop counter and is modified only on line 49. The variable `n[4]` is used as the loop bound and is not modified in the loop body. Note that the code fragment uses variables like `n[4]` that have been previously initialized.

Table 1: Symbolic execution of bubble sort inner loop and SSA form

(a) Line	(b) Instruction	(c) Updated stack contents/ Instruction side effects	(d) Equivalent SSA
37	`local.get 1`	`[n[1]]`	$t_1 \leftarrow n[1]$
38	`local.get 5`	`[n[5]; n[1]]`	$t_2 \leftarrow n[5]$
39	`i32.const 2`	`[2; n[5]; n[1]]`	$t_3 \leftarrow 2$
40	`i32.shl`	`[(n[5] shl 2); n[1]]`	$t_4 \leftarrow t_2$ shl t_3
41	`i32.add`	`[n[1] + (n[5] shl 2)]`	$t_5 \leftarrow t_4 + t_1$
42	`local.tee 6`	`[n[1] + (n[5] shl 2)]` `n[6] ← n[1] + (n[5] shl 2)`	$n[6] \leftarrow t_5$
43	`i32.load`	`[m[n[1] + (n[5] shl 2)]]`	$t_6 \leftarrow m[t_5]$
44	`local.tee 7`	`[m[n[1] + (n[5] shl 2)]]` `n[7] ← m[n[1] + (n[5] shl 2)]`	$n[7] \leftarrow t_6$
45	`local.get 1`	`[n[1]; m[n[1] + (n[5] shl 2)]]`	$t_7 \leftarrow n[1]$
46	`local.get 5`	`[n[5]; n[1]; m[n[1] + (n[5] shl 2)]]`	$t_8 \leftarrow n[5]$
47	`i32.const 1`	`[1; n[5]; n[1]; m[n[1] + (n[5] shl 2)]]`	$t_9 \leftarrow 1$
48	`i32.add`	`[n[5] + 1; n[1]; m[n[1] + (n[5] shl 2)]]`	$t_{10} \leftarrow t_8 + t_9$
49	`local.tee 5`	`[n[5] + 1; n[1]; m[n[1] + (n[5] shl 2)]]` `n[5] ← n[5] + 1`	$n[5] \leftarrow t_{10}$
50	`i32.const 2`	`[2; n[5] + 1; n[1];` `m[n[1] + (n[5] shl 2)]]`	$t_{11} \leftarrow 2$
51	`i32.shl`	`[((n[5] + 1) shl 2); n[1];` `m[n[1] + (n[5] shl 2)]]`	$t_{12} \leftarrow t_{10}$ shl t_{11}
52	`i32.add`	`[n[1] + ((n[5] + 1) shl 2);` `m[n[1] + (n[5] shl 2)]]`	$t_{13} \leftarrow t_{11} + t_{12}$

53	`local.tee 8`	`[n[1] + ((n[5] + 1) shl 2);`
		`m[n[1] + (n[5] shl 2)]]`
		`n[8] ← n[1] + ((n[5] + 1) shl 2)`
			$n[8] \leftarrow t_{13}$

54	`i32.load`	`[m[n[1] + ((n[5] + 1) shl 2)];` `m[n[1] + (n[5] shl 2)]]`	$t_{14} \leftarrow m[t_{13}]$

55	`local.tee 9`	`[m[n[1] + ((n[5] + 1) shl 2)];`
		`m[n[1] + (n[5] shl 2)]]`
		`n[9] ← m[((n[5] + 1) shl 2) + n[1]]`
			$n[9] \leftarrow t_{14}$

56	`i32.le_s`	`[m[n[1] + (n[5] shl 2)] ≤` `m[n[1] + ((n[5] + 1) shl 2)]]`	$t_{15} \leftarrow t_{14} \leq t_6$
57	`br_if 67`	`[]`	t_{15}
59	`local.get 6`	`[n[6]]`	$t_{16} \leftarrow n[6]$
60	`local.get 9`	`[n[9]; n[6]]`	$t_{17} \leftarrow n[9]$
61	`i32.store`	`[]` `m[n[9]] ← n[6]`	$m[t_{17}] \leftarrow t_{16}$
62	`local.get 8`	`[n[8]]`	$t_{18} \leftarrow n[8]$
63	`local.get 7`	`[n[7]; n[8]]`	$t_{19} \leftarrow n[7]$
64	`i32.store`	`[]` `m[n[7]] ← n[8]`	$m[t_{19}] \leftarrow t_{18}$
65	`end`	`[]`	
67	`local.get 5`	`[n[5]]`	$t_{20} \leftarrow n[5]$
68	`local.get 4`	`[n[4]; n[5]]`	$t_{21} \leftarrow n[4]$
69	`i32.ne`	`[n[5] ≠ n[4]]`	$t_{22} \leftarrow t_{21} \neq t_{20}$
70	`br_if 37`	`[]`	t_{22}

We use symbolic execution and single-assignment form (SSA) [16] to determine the value of these variables when the loop is entered.

Column (c) of Table 1 can be created by symbolically executing the corresponding WebAssembly code. Symbolic execution is a method of recording the effects on the machine state symbolically, instruction by instruction, when executing a code fragment. It gives us the ability to inspect the state of the virtual machine at any point in a function's execution. For example, we can look at a basic block that ends with a conditional branch instruction and determine symbolically what expression the conditional branch is based on. Symbols shown in column (c) represent the values of the respective variables when they are placed on the stack, not subsequently updated values.

Column (d) is an expression of the effect of the corresponding instruction (from column (b)). It is in SSA form.

We can observe the following facts about the code in the table for the inner loop of bubble sort:

- The loop tests the condition $n[5] \neq n[4]$ on line 69 (we can see this on the value stack) and continues execution via the `br_if` instruction on line 70 if this condition is true. Thus the variables that determine when this loop terminates are $n[4]$ and $n[5]$.
- From the instruction side effects in column (c) we see that the variable $n[4]$ is not modified.
- Similarly, we see that the variable $n[5]$ is modified on line 49 (only), where it is incremented.

From these observations we can see that the number of times this loop body will be executed is determined by the values of $n[4]$ and $n[5]$ when the loop body is entered. In the next section, we describe how we use this information to determine a bound on the number of loop iterations, and we elaborate on how this information can be used to determine a bound on execution cost.

3 Approach

In this section we describe our approach to producing bounds on WebAssembly functions. The **Wanalyze** tool reads a WebAssembly binary module as its input and analyzes each of the functions contained in that module. It outputs an expression, in terms of the function inputs, for an upper bound on the cost of executing the function.

Note that our cost model assumes that all WebAssembly instructions and all functions called (in either WebAssembly or native code) take the same amount of time to execute, allowing us to use the executed instruction count within a function as a proxy for code execution time. We count this as one unit of execution cost. While this simplification precludes precise execution bounds, such bounds are not feasible with WebAssembly in general, due to it being an execution format designed for portable, optimizing language runtimes; however, as we show in Section 4, this simplified model is sufficient to calculate consistent execution bounds.

Algorithm 1 Get SBs from BBs

```
1: function SBsOfBBs(BBs, SBs, SB, loop_nesting)
2:     while BBs ≠ [] do
3:         BB ← car(BBs)
4:         BBs ← cdr(BBs)
5:         // are we beginning a loop?
6:         if BB.type = LOOP then
7:             // yes, close off the current SB, if any, and start a new one
8:             if SB ≠ [] then
9:                 SBs ← SBs + SB
10:            end if
11:            return SBsOfBBs(BBs, SBs, [BB], loop_nesting + 1)
12:        else
13:            // are we ending a loop?
14:            if BB.type = END and BB.nesting = loop_nesting then
15:                // yes, close off the SB and start a new SB
16:                SBs ← SBs + (SB + BB)
17:                SB.children ← SBsOFBBs(SB, [], [], loop_nesting)
18:                SB ← []
19:                loop_nesting ← loop_nesting − 1
20:            else
21:                // Does this BB have predecessors not in the current SB?
22:                if BB.predecessors \ SB ≠ [] then
23:                    // Yes, start a new SB
24:                    return SBsOfBBs(BBs, SBs + SB, [BB], loop_nesting)
25:                else
26:                    // No need to start a new SB
27:                    SB ← SB + BB
28:                end if
29:            end if
30:        end if
31:    end while
32:    // close off the current SB, if any
33:    if SB ≠ [] then
34:        SBs ← SBs + SB
35:    end if
36:    return SBs
37: end function
```

Basic Blocks We first generate a set of basic blocks [14] for each function based on the following definition:

Definition 1. *(Basic Blocks of a WebAssembly function) The (set of) Basic Blocks (BBs) of a WebAssembly function are consecutive lines of code with the following properties:*

- *each instruction in the body of the function is contained in exactly one BB,*
- *each BB ends with either a* `block, loop, if, else, end, br, br_if, br_table, return, unreachable` *instruction and contains no other occurrences of any of these instructions.*

Because a BB contains no conditional execution paths its cost can be determined precisely as the number of instructions in the BB.

Super Blocks Aggregating BBs into Super Blocks (SBs) as we did in Figure 2 allows us to analyze loop structures more readily. In our definition, an SB is composed of consecutive BBs, and, in the case of nested loops, other SBs.

Definition 2. *(Super Block of a WebAssembly function) The conditions that determine how BBs are assigned to SBs are as follows:*

- *C1 There is an SB for each* `loop/end` *structure block in the function. It contains the BBs of that code fragment beginning with the first BB after the* `loop` *instruction and ending with, and including, the BB containing the associated* `end` *instruction.*
- *C2 A function's flow of execution can only enter an SB at the first BB in the SB.*
- *C3 Consecutive BBs with no loops appear in the same SB subject to C2.*

Algorithm 1 describes how the list of SBs is determined. In addition to the code, each basic block has some attributes, including its *type* (e.g., *LOOP*, *END*), its *nesting* level, which indicates the depth of the loop (if any) that it is contained in, and its *predecessors*, which include all previous basic blocks. A single pass is made through the basic blocks that make up the function, emitting super blocks whenever the conditions to create a new one or end the current one are satisfied. These conditions are on lines 6, 14, and 21 of the algorithm. Their meaning is as follows:

- line 6: the first basic block of a loop has been reached,
- line 14: the end basic block of a loop has been reached,
- line 21: a basic block with multiple entry points has been reached.

Code Path Generation If an SB has no loops we can bound its cost as: *the maximum cost over all paths from the root of the control flow graph to a leaf node* where we consider the cost of a path to be the sum of the cost of the BBs on that path. That is, we consider all possible execution paths through the SB

and take the maximum cost of those execution paths as our upper bound on the SB cost.

In general we evaluate the cost of each code path to compute a bound on the function cost.

Symbolic Execution and Static Single-Assignment (SSA) Form Symbolic execution models the effects of code on machine state symbolically, instruction by instruction. It gives us the ability to inspect the state of the virtual machine at any point in a function's execution. Column (c) of Table 1 shows the results of symbolic execution for our example.

We can use symbolic execution to determine the condition under which a loop terminates. To find a bound on the number of loop iterations we need another technique that allows us to symbolically evaluate the values of the variables in the condition. To do this we perform *program slicing* [27] on the SSA. Column (d) of Table 1 shows this result for our example.

Approach to Determining Loop Execution Cost As mentioned, the execution cost of an SB is bounded by the cost of the maximum cost path. There is no general solution to this problem (otherwise we would have a solution to the halting problem, as mentioned earlier). Our approach is based on rules, recognizing specific patterns in loop structures.

A loop body is an SB, thus giving us a bound on the execution of a loop body. To arrive at an expression for a bound on a loop, we must determine a bound on the number of iterations. We then combine these two bounds to arrive at an expression for a bound on the loop execution cost, which will generally include the parameters of the function. We need to know the following:

- L1: conditions that cause the loop to exit;
- L2: variables that are used in evaluating these conditions;
- L3: initial values of these variables when the loop is entered;
- L4: how the variables are updated in the loop.

To determine L1, the tool determines all possible paths through the body of the loop that either exit the loop, or return to the top of the loop. The tool then symbolically executes these paths to determine the contents of the value stack when a looping condition is evaluated. Since we want conditions that cause the loop to exit, this looping condition is negated if control returns to the top of the loop. This produces a list of loop conditions, one for each path through the loop body. We can then parse these conditions to determine which variables they use, which gives us L2. From line 69 in the example, we can see that the variables in this case are $n[4]$ and $n[5]$. For each variable, we can then create the SSA form for all function execution paths to the loop SB. We can then simplify these SSA to get the possible values of the variables at loop entry. This gives us L3. Finally, we can create a SSA form for each of the variables, but this time simplify it over all execution paths from the first BB of the loop to the BB of their associated condition. This will give us L4.

This result is then used to apply a set of rules depending on the condition and the manner in which the condition variables are updated to determine the number of loop iterations. In our tests, we found that a small number of rules, usually based on observing a loop counter representing the size of a data structure or a static loop limit, allowed most loop constructs to be analyzed to produce a bound on the number of iterations. We believe that this is due to the fact that the structured control flow found in the source language (e.g. C or Rust) translates readily into WebAssembly.

Cost Analysis We compose the analyses based on bounds on the costs of BBs, SBs, and loops using code path generation, symbolic execution, and SSA to produce a bound on the execution cost of the function. In describing these costs, we use the phrase *SB path* to mean an execution path that is made up of SBs. We say *linear SB path* to mean a path that does not loop. The SBs that make up the path may contain loops, but at the level of the path there are no loops. Similarly a *BB path* is an execution path that is made up of BBs and a *linear BB path* is a path with no loops. We also introduce some notation:

- We use f to denote a function, X to denote the inputs to the function, sp for an SB path, both s and sc denote an SB, bp a BB path and b a BB.
- For an SB s, let $S(s)$ be the set of all linear SB paths of SBs in s that start at the first SB of s and exit s. That is, we restrict $S(s)$ to contain only paths that are non-looping and are made up of SBs contained in s. We extend this notation to functions; $S(f)$ denotes all linear SB paths through the function f.
- Let $nS(sp)$ be the set of SBs without a loop that are on the SB path sp.
- Let $lS(sp)$ be the set of SBs with a loop that are on the SB path sp
- For a looping SB s, let $N(s, X)$ be a bound on the number of times the loop will iterate given function inputs X.
- Let $B(s)$ be the set of all linear BB paths of the SB s.
- Let s_f be the SB that contains all of f.

With this notation, we can express the following equalities and inequalities for bounding the cost of a function:

$$cost_F(f, X) \leq \max_{sp \in S(f)} cost_{pS}(sp, X) \tag{1}$$

$$cost_{pS}(sp, X) \leq \sum_{s \in nS(sp)} cost_S(s) + \sum_{s \in lS(sp)} [\max_{sp' \in S(s)} cost_{pS}(sp', X)] * N(s, X) \tag{2}$$

$$cost_S(s) \leq \max_{bp \in B(s)} cost_{pB}(bp) \tag{3}$$

$$cost_{pB}(bp) = \sum_{b \in bp} cost_B(b) \tag{4}$$

$$cost_B(b) = \text{number of instructions in BB } b \tag{5}$$

In these equations, the subscript F is for the cost of the function, pS for the cost of an SB path, S for an SB, pB for a BB path, and B for a BB. The equations can be summarized as follows: (1) the cost of a function is bounded by the maximum of the cost of all SB paths through the function; (2) the cost of an SB path is bounded by the sum of the cost of each non-looping SBs in the path plus, inductively, the cost of each of the looping SBs times the number of loop iterations; (3) the cost of an SB is bounded by the maximum cost of all BB paths of the SB; (4) the cost of a BB path is the sum of the cost of the BBs in the path; and (5) the cost of a BB is the number of instructions in the BB.

Objects in Memory A case that warrants specific mention is that of objects in memory and functions that operate on those objects. Generally if a function contains code that loops over an object structure and has a loop termination condition that depends only on that data in the object, then **Wanalyze** will not be able to produce a bound for that function.

4 Test Results

We discuss three experiments that we have carried out, followed by an analysis of the results.

Bubble Sort The focus of the first experiment was the bubble sort code described in Section 2. Thus, we start with a relatively simple test case containing both a nested loop and conditional execution paths. We chose this example to determine if **Wanalyze** could successfully produce a prediction for the number of instructions to be executed and to determine how accurate that prediction was compared to actual measurements.

This experiment was run in two parts. The first part involved running the bubble sort WebAssembly code using the **wasmtime** [11] WebAssembly runtime. This runtime has the ability to instrument WebAssembly code to measure the amount of "fuel" the code consumes when executed. Conveniently, it measures fuel as the number of instructions executed with the exception of `nop`, `drop`, `block`, and `loop` instructions.

The bubble sort WebAssembly function takes as input a fixed length array of 32 bit integers and sorts them in place. It is well known that the performance of bubble sort is dependent on the composition of the data. Worst case performance occurs when the input data are ordered in the complete reverse of the sorted data. We ran two different tests so that we could compare results. The first test was with the data in this worst-case reverse-sorted order; the second test was done with the data in random order. Ten runs with input arrays sized between 10,000 and 100,000 in increments of 10,000 were run. In order to run this WebAssembly code with **wasmtime**, it was necessary to write "glue" code in Rust that creates the data. This code loads the WebAssembly module containing the bubble sort, runs it and reports on the amount of fuel consumed.

The second part of the experiment involved running **Wanalyze** on the WebAssembly bubble sort function. The analysis was successful and produced the following polynomial, which is typical of the general format produced:

$$\frac{1}{2}(31n^2 + 11n - 20)$$

as the bound on the number of instructions that would be executed when sorting n items. This agrees with the well-known fact that bubble sort takes $O(n^2)$ time to execute.

Table 2: Bubble sort bound vs. actual - **wasmtime** fuel

Items	**Wanalyze**	Worst-case		Random	
(K)	Bound (M)	Actual (M)	Difference	Actual (M)	Difference
10	1,595	1,550	2.90%	1,401	13.85%
20	6,800	6,200	1.61%	5,601	12.48%
30	14,105	13,950	1.11%	12,594	12.00%
40	25,010	24,800	0.85%	22,401	11.65%
50	39,015	38,750	0.68%	35,000	11.47%
60	56,120	55,800	0.57%	50,397	11.36%
70	76,325	75,950	0.49%	68,625	11.22%
80	99,630	99,200	0.43%	89,597	11.20%
90	126,035	125,550	0.39%	113,400	11.14%
100	155,540	155,000	0.35%	139,990	11.11%

Table 2 contains the results of both parts of the experiment. The column titled "**Wanalyze** Bound" shows the bound that **Wanalyze** produced for the number of instructions (in millions) executed to sort that many items. The next column "Worst-case Actual" is the number of instructions reported by **wasmtime** when sorting a set of items that is initially in the complete reverse of sorted order. The column "Random Actual" is the number of instructions, but this time the set to be sorted is initially in random order. The two percentage columns are the differences between the **Wanalyze** bound and the actual measured result in **wasmtime**.

Dhrystone Benchmark The second experiment followed a similar procedure as the first, but this time the WebAssembly code generated by the **emscripten** toolchain for version 2.1 of the Dhrystone benchmark [20] was analyzed. It is expected that the Dhrystone benchmark performance is linear in the number of iterations. Inspection of the code verifies that this is the case and **Wanalyze** produces a cost bound of:

$$1033n + 2578$$

where n is the number of Dhrystone iterations performed. See Table 3 for detailed measurements.

In this case no glue layer code was necessary because the Dhrystone program has no input data. The number of iterations to be performed is a parameter that is compiled into the program.

The results in Table 3 follow a similar format to those for bubble sort. The difference is that since there is no input data to consider, it was only necessary to run a single test for a given number of iterations.

Table 3: Dhrystone predicted / actual

wasmtime		**Wanalyze**	
Iterations	Actual	Bound	Difference
12,088	7,834	12,487	59.39%
15,110	9,792	15,609	59.41%
18,666	12,100	19,282	59.36%
20,336	13,180	21,007	59.39%
31,110	20,183	32,137	59.23%

MUSL C Library, AutoCAD application The third experiment consisted of running **Wanalyze** on the WebAssembly code for the MUSL C runtime library [23] and the AutoCAD application [8]. Each of these is a large code base, and so they demonstrate the ability of **Wanalyze** to scale. It was necessary to first compile the MUSL library to WebAssembly. AutoCAD is available for download as a WebAssembly module.

The entire MUSL library contains over 58,000 lines of code (LOC) in C which, when compiled, results in over 1.26 million lines of WebAssembly code contained in over 14,000 distinct functions. With an elapsed time of 53 minutes, this results in a rate of over 1000 lines of C code analyzed per minute and a rate of 23,000 lines of WebAssembly analyzed per minute. **Wanalyze** was able to analyze more than 94% of all functions with failure occurring in cases where the number of paths through the function exceeded the built-in limit of one million paths. This limit is used to limit the running time of the analysis and can be extended.

AutoCAD is an order of magnitude larger than the MUSL C library, when measured by lines of WebAssembly code or number of functions. Comparatively, the elapsed time required to analyze AutoCAD is nearly linear in those metrics.

Table 4: MUSL C library, AutoCAD application

	C LOC	WebAssembly LOC	# functions	Success rate	Elapsed time
MUSL	58,237	1,267,725	14,329	94.6%	53 minutes
AutoCAD	-	22,641,000	93,664	99.9%	481 minutes

Analysis As demonstrated in the final experiment, the performance of **Wanalyze** scales with larger bodies of code. This scaling is driven by these design choices:

- **Wanalyze** makes a single pass through the WebAssembly binary.
- The algorithms to produce its primary data structures, BBs and SBs, are linear in the number of instructions.
- The number of BBs and SBs produced for a given function is generally an order of magnitude less than the number of instructions in the function.
- The analysis algorithms, symbolic execution, and SSA generation operate in time that is linear in the number of paths through the function.

With both bubble sort and Dhrystone, the bound produced by **Wanalyze** is indeed a bound, in that it is greater than the measured actual value. Also, for both the difference measured is a similar percentage for tests with the same input data. However, this percentage varies across different tests and input data.

These results meet the objective for bounds to be considered reasonable that we set out in the introduction. Execution costs do not exceed the bound, and the bound has the same computational complexity as the function being analyzed. However, we must ask why does the **Wanalyze** bound differ from the actual observed instruction counts? And, why does this difference vary between applications and data sets? For example, bubble sort with random data has a difference of 14% (rounded), but Dhrystone has a difference of 59%.

The primary reason for these differences is that **Wanalyze** necessarily chooses the most costly path when evaluating alternative costs in a super block or basic block. In practice, this is a conservative approach because it will often be the case that a less costly path is taken. This means that the accuracy of **Wanalyze**'s bound will depend on how often the costliest path is taken.

This is also an explanation for the differences between applications. Not only do the applications that have lower cost paths that contribute to the **Wanalyze** estimate being higher, but also there are differences in the path structure of the functions in the applications. In particular, there is a big difference between the super block control flow diagrams for two functions. Thus, the impact of lower-cost paths will be different between the applications.

The same reasoning also explains why applications that run on different data will have different results. The effect of the lower cost paths will be different when different input data is used.

5 Related Work

The problem of automatically bounding or estimating the cost of execution of a program using static analysis methods has been well studied, starting with the work of Wegbreit [26]. In a series of papers [2,3,6], Albert et al. described their work on solving this problem for Java bytecode. Wegbreit's basic method of analysis is used but aspects of Java bytecode created additional challenges such as loop detection. A notable contribution of these papers is a theoretical

framework for formulating and solving the recurrence relations that are produced by the analysis. Determining a bounding on the cost of executing a loop is a core challenge; lexicographic ranking functions [1,2,10] are widely used both to prove loop termination and to determine loop costs.

Some authors [7,19,22,24] focus on the problem of determining the algorithmic complexity of a function and choose to test their solution against well-known algorithms with well-known complexity. Solutions also include consideration of amortized algorithmic costs which can improve precision. The example given in [19] that demonstrates this benefit is the functional queue, which performs dequeue operations in constant amortized time. The language used in that paper is OCaml. Other languages, such as Raml (resource aware ML) [18], and Solidity (a language for Ethereum smart contracts) [5] have been studied as well.

Work in the WCET domain [9,15] has focused on the analysis of loops. We demonstrate that aspects of WebAssembly simplify this problem to some extent.

Almost all of the papers mentioned performed experiments using specific data sets comprised of code that was analyzed. Our general observation is that, like the functions in the real-world applications that we analyzed, the test cases consist of a small number of small to medium sized functions or programs. The examples tested in [12] represent a particularly challenging set of loop constructs that are atypical of loop constructs we saw when analyzing real world programs. In contrast, we have taken an empirical approach of analyzing well-known functions, particularly those in an implementation of the standard C library implementation MUSL.

6 Conclusion

As we have shown, WebAssembly provides a new opportunity to revisit the problem of bounding the execution cost of a program. In particular, it has features that simplify cost analysis such as structured control flow, which means that it is not necessary to perform loop detection on the input program and translate it to an intermediate, structured form. In addition, WebAssembly only has scalar data types. As a consequence, expensive size analysis methods described in the literature can be greatly simplified.

As future work, we can extend our results to determine a bound on the cost of a whole program, which includes functions that call other functions in the module as well as imported functions, assuming that the WebAssembly hosting environment will provide the cost of those functions if required. The cost bound of a non-recursive function call will be an expression based on that function's input, which will, in turn, be expressed in terms of the inputs to the calling function. For both recursive and non-recursive calls, we can derive a set of recurrence relations between the costs of these functions. The work by Albert et al. [2,3,4,6] describes how these recurrence relations can be solved. Future work also includes correctness proofs of our results, which would provide greater assurances in our methods. A mechanization of such a proof would also allow us to add proofs of performance bounds to analyzed functions.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Static Analysis: 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings 15. pp. 221–237. Springer, Berlin Heidelberg (2008)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. Journal of automated reasoning **46**, 161–203 (2011)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Costa: Design and implementation of a cost and termination analyzer for java bytecode. In: Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures 6. pp. 113–132. Springer, Berlin Heidelberg (2008)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012)
5. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: Don't run on fumes—parametric gas bounds for smart contracts. Journal of Systems and Software **176**, 110923 (2021)
6. Albert, E., Genaim, S., Masud, A.N.: More precise yet widely applicable cost analysis. In: Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings 12. pp. 38–53. Springer, Berlin Heidelberg (2011)
7. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Static Analysis: 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings 17. pp. 117–133. Springer, Berlin Heidelberg (2010)
8. AutoCAD: Roundup: The AutoCAD Web App at Google I/O 2018. https://blogs.autodesk.com/autocad/autocad-web-app-google-io-2018/ (2018), accessed December, 2022
9. Blazy, S., Maroneze, A., Pichardie, D.: Formal verification of loop bound estimation for wcet analysis. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 281–303. Springer (2013)
10. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17. pp. 491–504. Springer, Berlin Heidelberg (2005)
11. Bytecode Alliance: wasmtime - A Standalone Runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime (2022), accessed December, 2022
12. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 467–478. ACM, New York, NY, USA (2015)
13. Cocke, J.: Global common subexpression elimination. In: Proceedings of a symposium on Compiler optimization. pp. 20–24. ACM, New York NY (1970)
14. Cooper, K.D., Torczon, L.: Engineering a Compiler. Elsevier, Burlington MA (2011)
15. Cullmann, C., Martin, F.: Data-flow based detection of loop bounds. In: 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2007)

16. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 25–35. ACM, New York, NY (1989)
17. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 185–200. ACM, New York NY (2017)
18. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 357–370. ACM, New York NY (2011)
19. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for OCaml. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 359–373. ACM, New York NY (2017)
20. Keith S. Thompson: Dhrystone v2.1. https://github.com/Keith-S-Thompson/dhrystone/tree/master/v2.1 (2022), accessed December, 2022
21. Lucas, S.: The origins of the halting problem. Journal of Logical and Algebraic Methods in Programming **121**, 100687 (2021)
22. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part I. pp. 250–269. Springer, Berlin Heidelberg (2021)
23. musl: musl libc. https://musl.libc.org/ (2023), accessed May, 2023
24. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. pp. 745–761. Springer, Berlin Heidelberg (2014)
25. WebAssembly Community Group: WebAssembly Introduction. https://webassembly.github.io/spec/core/intro/introduction.html (2020), accessed December, 2022
26. Wegbreit, B.: Mechanical program analysis. Communications of the ACM **18**(9), 528–539 (1975)
27. Weiser, M.: Program slicing. IEEE Transactions on software engineering **SE-10**(4), 352–357 (1984)