Learning Cost Semantics for Modeling Running Time of OCaml Programs

Ankush Das Jan Hoffmann
Carnegie Mellon University

Abstract

Programmers would greatly benefit from a source-language model of the running time of compiled code that goes beyond asymptotics. However, it is a difficult problem to accurately model the running time of compiled code at the source level. In this work, we present an operational cost semantics to model the running time of OCaml programs on a specific hardware architecture. Our hypothesis is that the running time of a program can be modeled as a linear function of the number of executions of some high level constructs that we have carefully selected. We learn the coefficients of this linear function by applying traditional machine learning algorithms to automatically adapt the cost semantics to match the measured runtime of training examples on a modern hardware platform. With this learned cost semantics, we are able to model the running time of simple OCaml programs within a reasonable margin of error. Our experiments show that effects of low-level features like memory caches seem to amortize while modeling garbage collection remains a major challenge.

1 Introduction

Modeling the running time of programs has been a long standing research problem. Precise modeling of program running times can have several positive impacts, particularly in scheduling high load tasks in servers [5], efficient allocation of resources to programs [7], development-time feedback for programmers [4], etc. This problem is often considered difficult due to unpredictable cache and other memory effects.

In this work, we present a simple operational cost semantics to model the running time of sequential programs within a reasonable margin of error. We have identified several high-level constructs in the OCaml [3] language,

and keep track of the number of executions of each construct in the semantics using an OCaml interpreter that we developed ourselves. Our semantics then models the running time of the program as a linear function of these constructs. We learn the coefficients of this linear function by running a set of carefully chosen, relatively simple programs. We measure the average running times of these programs and apply a linear regression algorithm [8] on the linear function obtained by running the programs with different inputs in our interpreter.

Once we learn the semantics, we run several test programs with our interpreter along with the derived cost semantics to get an approximate measure of the actual running time. Note that our approach is dynamic, in the sense that we run the program with our interpreter to get an approximation on the running time. With this approach, we are able to model the running time of moderately involved programs which don't call the garbage collector with an accuracy of about $80 - 96\%$.

2 Example

Consider the following OCaml program, which computes the factorial.

```
let rec fact n =
        if (n = 0) then 1 else n * fact (n-1);;
(fact 10);;
```

In the above program, if we count the number of high level constructs, we get

- Function Application: 11
- Integer Equality: 11
- Integer Subtraction: 10
- Integer Multiplication: 10
- Let Rec: 1

Our model follows the hypothesis that the running time of a program is the sum of the running time of each construct multiplied by the number of times that construct is executed. For the above program, the total running time is

$$11 * T_{app} + 11 * T_{eq} + 10 * T_{sub} + 10 * T_{mult} + 1 * T_{letrec}$$

The main challenge of the problem is to define an appropriate set of these constructs. Defining too few constructs may not be able to cover the generality of a program, and having too many constructs could cause overfitting of the model, thus leading to poor results on the test programs. Our main task in this work is to learn the values of these T's to get an approximation of the program running time.

3 Cost Semantics

We have chosen 32 constructs which we briefly describe below.

- Function Application (normal, tail)
- Boolean Conditionals $(=, \&\&, ||, not)$
- Integer and Float Conditionals $(=, <, \leq, >, \geq)$
- Integer and Float Arithmetic $(+, -, *, /, mod)$
- Let and Let Rec
- Pattern Match
- Tuple and Tuple Match

During our experimental evaluation, we made the following observations.

- We do not need to model the cache or memory to make reasonably good predictions. These effects get amortized in the coefficients we learn for the above constructs.
- Normal function calls are treated differently from tail calls. A tail call occurs when the return value of a callee is directly returned by the caller. OCaml 4.02.1 statically detects tail calls, and converts them to a jump instruction. Hence, we need a special construct to model tail calls.
- OCaml 4.02.1 also inlines small functions. We currently don't support inlining, so we supress it in the compiler.

4 Training

We used a set of 64 carefully chosen, relatively simple training programs to learn the cost semantics. We have chosen a simple base file, which is a naive recursive program as follows.

```
let rec fbase n =
        if (n = 0) then 0 else fbase (n-1);;
```

We then chose two programs, both specializing in one construct to model the increase in running time over this base program. This is done in an attempt to isolate each construct from the rest, helping the machine learner to converge faster. For instance, the training program for integer addition looks as follows.

```
let rec fadd n =
        if (n = 0) then 0 else 1 + 1 + 1 + fadd (n-1);;
```

Now, during analysis, the machine learning algorithm understands that the increase in running time over the base program is caused primarily by integer addition, hence counting the number of extra additions gives us a precise estimate of the execution time for a single addition. Following this approach with each construct, we get a set of 64 simple training programs.

We have developed an OCaml interpreter which counts the number of executions of each of these constructs. We run each program with about 50 inputs from size 1000 to 50000, once with our interpreter to obtain the linear function like the one described in the example. We then run the program on 1.6 GHz Intel i5-5250U Dual Core processor with 16 GB RAM to measure its running time, i.e. the value that the linear function should have. Since the running times vary depending on OS scheduling, memory available, system calls, etc. we run each program 1000 times for each input to measure the average running time. We then run a linear regression algorithm to obtain the coefficients of this linear function, thus completing the training of the cost semantics.

5 Results

Once we learn the cost semantics, we use simple but realistic programs for testing purposes. We run these test programs with our interpreter to obtain the linear function from our semantics. Now, plugging in the coefficients, we get an approximation of the program running time. We used about 40 test programs to measure the accuracy of our model. We ran the programs with

varying input sizes in intervals of 1000, starting from 1000, but restricting the maximum size carefully to ensure that the GC is not called. As in training, we run each program 1000 times for each input to get the average running time. For each input, we compute the modeled running time and measure the average running time. The difference gives the error for one input, which is then averaged out over all input sizes. With the above configurations, we are able to model the actual running time of these test programs with an accuracy of about $80 - 96\%$. For instance, for our factorial example, the error was averaged out to $+5.7\%$. We list several other examples below.

- append (appends one list to another) - Error = $+16.1\%$
- drop (drops the last element of list) - Error = $+4.9\%$
- echelon (converts matrix to echelon form) - Error = -9.1%
- isort (insertion sort of list) - Error = -23.5%
- compress (compression algorithm on list) - Error = $+4.2\%$
- equal (checks two lists for equality) - Error = $+13.5\%$
- map (maps elements of list by an arbitrary function) - Error = -7.3%
- rotate (rotates the list by a constant) - Error = -11.6%

A positive error means our estimated time is higher than the actual running time and vice-versa. Hence, we are able to learn a cost semantics which models the running time of programs within a reasonable margin of error. One of the interesting contributions is that we are able to make predictions about complex OCaml programs, with a cost semantics learned from relatively simple training programs. Another interesting aspect is that most of the memory and cache effects get amortized in the learned coefficients, and we don't need additional modeling for these effects.

6 The Garbage Collector

With smaller inputs to the program, there are no calls to the garbage collector, and hence, the program running time can be effectively modeled using a linear function. With larger inputs, the garbage collector runs causing sudden significant jumps in the running time. Hence, we cannot model these running times using linear functions. If we just naively extend our model

with larger inputs, the margin of error increases significantly. We present some of our evaluation below.

- append - Error = -27.0%
- drop - Error = -26.2%
- isort - Error = $+142.4\%$
- factorial - Error = $+22.4\%$

Note that the functions *append* and *drop* are not written in a tail recursive manner, thus they call the GC several times during their execution. On the other hand, *isort* and $factorial$ are tail recursive, and hence, make no calls to the GC. Hence, for *append* and *drop*, the actual running time starts out to be smaller than the modeled time (since we learned larger values for coefficients) but eventually goes above the modeled time after several jumps in running time. For *isort* and $factorial$, since there are no GC calls, the estimated time is always higher than the actual running time.

It is generally a difficult problem, to be able to model the input sizes where calls to the GC are made. Also, the number of GC calls varies significantly, in the training and testing programs, thus causing an even higher margin of error. Intuitively, we believe we can model the number of heap allocations done by the program using a linear function over these constructs but modeling the number of live cells requires more work and is currently subject of our research.

7 Conclusion and Future Work

We have developed an operational cost semantics to predict the running time of programs. Using existing techniques, we can even statically compute the symbolic frequency of each construct, hence, computing the actual symbolic running time of program statically. Obviously, the cost semantics need to be learned for a specific hardware, and in our experience, memory and cache effects get amortized in our cost semantics, hence, we don't have to add these features to our training model. We also believe that this approach can easily be extended to other programming languages like SML-NJ, and imperative languages like C, C++, Java since the connection between the high-level constructs and assembly code is essentially the same in all languages.

One of the challenges that remains to be addressed is the garbage collector, because it causes significant changes to running times if the program

is memory intensive. One approach is to observe that most of the memory cells are live for a very short duration, while only a few cells are live for a longer period of time [6, 10, 2]. If we consider the distribution of lifetime of memory cells, it is, in principle, similar to distributions with a decreasing failure rate (DFR) property [9, 1], where the probability of failure (equivalent of not being live) decreases with increasing lifetime. We believe we can model the memory behavior of programs by appropriately tuning the parameters of these DFR distributions. We are currently pursuing this idea to model the garbage collector.

References

[1] K Adamidis and S Loukas. A lifetime distribution with decreasing failure rate. *Statistics & Probability Letters*, 39(1):35–42, 1998.

[2] Cagdas Bozman, Michel Mauny, Fabrice Le Fessant, and Thomas Gazagnaire. Study of ocaml programs memory behavior. *OCaml Users and Developers*, 2012.

[3] Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.03. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[4] Neil B. Harrison and Avaya Labs. A Study of Extreme Programming in a Large Company.

[5] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[6] Jin-Soo Kim and Yarsun Hsu. Memory system behavior of java programs: Methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '00, pages 264–274, New York, NY, USA, 2000. ACM.

[7] M.S. Miller, E.D. Tribble, N. Hardy, and C.T. Hibbert. Diverse goods arbitration system and method for allocating resources in a distributed computer system, June 17 1997. US Patent 5,640,569.

[8] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[9] Frank Proschan. Theoretical explanation of observed decreasing failure rate. *Technometrics*, 42(1):7–11, 2000.

[10] Timothy Sherwood and Brad Calder. Time varying behavior of programs. 1999.