ML for ML: Learning Cost Semantics by Experiment

Ankush Das and Jan Hoffmann

Carnegie Mellon University

Abstract. It is an open problem in static resource bound analysis to connect high-level resource bounds with the actual execution time and memory usage of compiled machine code. This paper proposes to use machine learning to derive a cost model for a high-level source language that approximates the execution cost of compiled programs on a specific hardware platform. The proposed technique starts by fixing a cost semantics for the source language in which certain constants are unknown. To learn the constants for a specific hardware, a machine learning algorithm measures the resource cost of a set of training programs and compares the cost with the prediction of the cost semantics. The quality of the learned cost model is evaluated by comparing the model with the measured cost on a set of independent control programs. The technique has been implemented for a subset of OCaml using Inria's OCaml compiler on an Intel x86-64 and ARM 64-bit v8-A platform. The considered resources in the implementation are heap allocations and execution time. The training programs are deliberately simple, handwritten micro benchmarks and the control programs are retrieved from the standard library, an OCaml online tutorial, and local OCaml projects. Different machine learning techniques are applied, including (weighted) linear regression and (weighted) robust regression. To model the execution time of programs with garbage collection (GC), the system combines models for memory allocations and executions without GC, which are derived first. Experiments indicate that the derived cost semantics for the number of heap allocations on both hardware platforms is accurate. The accuracy of the cost semantics on the control programs for the x86-64 architecture for execution time with and without GC is about 19.80% and 13.04%, respectively. The derived cost semantics are combined with RAML, a state-of-the-art system for automatically deriving resource bounds for OCaml programs. Using these semantics, RAML is for the first time able to make predictions about the actual worst-case execution time.

1 Introduction

Motivated by longstanding problems such as performance bugs [32], side-channel attacks [31, 10], and to provide development-time feedback to programmers, the programming language community is developing tools that help programmers understand the resource usage of code at compile time. There has been great progress on automatically determining loop bounds in sequential C-like programs [22, 13, 35, 15], deriving, solving recurrence relations [4, 6, 19, 9], and automating amortized analysis [25, 23, 24]. There exist several tools that can automatically derive loop and recursion bounds, including SPEED [21, 22], KoAT [13], PUBS [5], Rank [7], ABC [11], LOOPUS [38, 35], C4B [14], and RAML [23, 24].

Most of these resource analysis tools use high-level cost models, like number of loop iterations and function calls, and it is often unclear how the derived bounds relate to the machine code executing on a specific hardware. To make the connection, one has to take into account compilation, hardware specific cache and memory effects, instruction pipelines, and garbage collection cycles. While there exist tools and techniques for analyzing low-level assembly code to produce worst-case execution time bounds for concrete hardware [37], they are limited in their expressive power, as analyzing assembly code is a complicated problem.

In this article, we propose a novel technique to derive cost models that can link high-level resource bounds to the execution of low-level code. We present a simple operational cost semantics for a subset of OCaml [20] that have been learned using standard machine learning techniques like linear regression. The resources we are considering are heap allocations, execution time without garbage collection (GC), and execution time with GC. The subset of OCaml we are considering is purely functional and includes lists, tuples and pattern matching. However, the technique is also applicable to programs with side effects.

To learn a cost semantics, we first define an operational big-step semantics that assign a parametric cost expression to a well-formed expression. This cost expression is parametric in (yet unknown) constants that correspond to high-level constructs in OCaml. The assumption is that the number of executions of each of these constructs constitutes the majority of the execution time of the expression. We keep track of the number of executions of each of these constructs in the cost semantics, which has been implemented in an OCaml interpreter. Our semantics then models the execution time of the program as a linear sum of the number of executions of each construct. The (unknown) coefficients of this linear function intuitively represent the execution time of each construct.

We then determine the average values of the coefficients on a specific hardware by experiment. We carefully select a set of relatively simple training programs and measure the median execution time of these programs on the hardware of interest. To this end, each program is executed with the OCaml native code compiler 500 times on an Intel x86-64 and a ARM 64-bit v8-A platform. We then apply machine learning techniques, such as linear regression [30], on the linear functions obtained by running the cost semantics to determine the constant costs of the constructs by fitting the prediction of the cost semantics to the measured costs. We measure the execution time using the Unix library in OCaml, which is hardware independent. We measure the number of allocation by relying on the OCaml GC library, which is again hardware independent. As a result, our approach is completely hardware independent and can be easily extended to different architectures, as we demonstrate in this work.

Of course, the execution time of, say, an addition cannot be described by a constant. In fact, it can vary by a large margin depending on whether the arguments are stored on the stack or in a register. Similarly, a cons operation can be costly if one of the arguments is not in the cache has to be fetched from memory. So the constants that we learn in our experiment will represent roughly the *average cost* of the operations on the training programs.

Once we have learned these cost coefficients for a specific hardware and resource metric, we validate our cost with control (or test) programs, retrieved from the standard library, an OCaml online tutorial, and local OCaml projects. Each control program is first compiled and executed on the hardware and the median execution cost is measured in the same way we did for training programs. The program is then run on the interpreter to obtain the parametric linear cost function. By plugging in the learned coefficients, we get a prediction for the execution time or memory usage. We compare the predictions of the cost semantics with the median cost, and report the percentage error on test programs. We use the median instead of the mean because it is more resilient against outliers which are often caused by context switches in the OS.

The result of the experiments with the control programs are surprisingly encouraging. We precisely learn the amount of memory that is allocated by each construct. For execution time of programs that do not trigger GC, the accuracy of our model is between 1% and 43%.

In memory intensive programs, the impact of garbage collection cycles on the execution time is significant. So, we adapt our cost semantics to account for the time taken by the GC. We make two simplifying assumptions to model the GC time. One of them is that the time taken by each GC cycle is a constant and the other is that each GC cycle starts with a full heap, and ends with an empty heap. These assumptions, as we will see in the experiments and the OCaml documentation, are quite close to the collections of the *minor heap.* To model this behavior, we combine the cost semantics for memory allocations and the cost semantics for programs without GC. Since the GC cycle occurs periodically when the minor heap is full, we can predict the number of minor GC cycles in the lifetime of a program using the allocation semantics. To determine the time needed for a minor garbage collection, we just measure the median GC time taken by a GC cycle for the training programs.

The main application of our cost semantics is the integration into Resource Aware ML (RAML), a state-of-the-art tool for automatic resource analysis. Using the semantics for execution time on x86, RAML can automatically derive worst-case bounds for many functions that are close to the measured execution time of the compiled code. Our results are precise enough, to statically determine the faster versions of different implementations of list append, Sieve of Eratosthenes, and factorial.

2 Method and Experimental Setup

In this section, we describe our experimental setup and training method. The main hypothesis, going into this experiment, is that the resource consumption of a program, whether time or memory, is a linear combination of the number of executions of each construct in the program. Moreover, the time (or memory) consumed by each construct is averaged out to be a constant. Hence, the execution time of a program is

$$T = \sum_{c \in \mathcal{C}} n_c T_c \tag{1}$$

where $\mathcal{C}$ represents the set of constructs and n_c is the count of each construct during program execution, and T_c is the execution time of the respective construct. Clearly, these assumptions do not, in general, hold for most of the relevant platforms. Hence, we will analyze the error incurred by these simplifying assumptions on the execution time (or memory allocation) of a program.

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 10 function calls, 11 equality checks, 10 subtractions and multiplications and 1 "let rec" that defines the function. In our model the execution time of a program is the sum of the execution time of each construct multiplied by the number of times that construct is executed. For the above program, the total execution time is

$$11 * T_{FunApp} + 11 * T_{IntEq} + 10 * T_{IntSub} + 10 * T_{IntMult} + 1 * T_{letrec}$$

We are interested in the resources costs T_i that best approximate the actual cost. With this representative example, we describe our experimental setup.

Language Description We have chosen a subset of OCaml as our modeling language. In this subset, we include the following program constructs: recursive functions, conditionals, boolean, integer and float comparisons and arithmetic, pattern matching and tuples. With this fairly general subset, we can write a variety of programs including list manipulation, matrix operations and other numeric programs, as we will demonstrate in our results. We chose OCaml as the source language for several reasons. For one, OCaml is a widely used language for functional programming which is quite efficient in practice. Moreover, we wanted to demonstrate that it is possible to define a practical cost semantics for a high-level functional language with a sophisticated compiler and automatic memory management. We assume that defining such a semantics would be easier for imperative programs, which are closer to assembly code.

A major obstacle when analyzing high-level languages is compiler optimization. The resource usage of the generated target assembly code depends greatly on the choices that are made by the compiler and cannot directly derived from the original OCaml program. Hence, the cost semantics need to account for compiler optimizations. In our experience, we found two compiler optimizations with a significant impact on the execution time.

– *Tail Call Optimization* [36] - If the final action of a function body is a function call, it is optimized to a jump instruction. This is relevant for the cost semantics because a jump is faster than a call, hence we need two different costs for usual function calls and tail calls. Moreover, separating these costs in the semantics will later help us validate the claim that tail call optimization indeed reduces the execution time.
– *Function Inlining* [16] - OCaml compiler inlines functions as an optimization. Instead of accounting for inlining in our interpreter, we forced the compiler to not inline any function when generating the native code. We will demonstrate the effect of this optimization when describing training programs. Conceptually, inlining is not a problem for our approach since it is possible to track at compile time which function applications have been inlined.

Training Algorithm We formally describe the algorithm we use to learn the values of the constructs. Consider again our cost expression $T = \sum_{c \in \mathcal{C}} n_c T_c$. Essentially, we execute the native code obtained from the compiler to obtain the T value, and we execute the program on the interpreter to obtain the n_c values. Let there be P training programs and suppose we generate M instances of training data from each training program using the above method. Let us denote the count of each construct and execution time of the i-th instance of training data generated by j-th training program by $(n_c^{(i,j)})_{c \in \mathcal{C}}$ and $T_{(i,j)}$ respectively. Since we need to learn a linear model on T_c's, linear regression is the natural choice of machine learning algorithm. A simple linear regression [33] would produce the following objective function.

$$S = \sum_{j=1}^{P} \sum_{i=1}^{M} \left(T_{(i,j)} - \sum_{c \in \mathcal{C}} n_c^{(i,j)} T_c \right)^2$$

where T_c are the unknowns that need to be learned. However, this approach is useful only when the error is additive, i.e. the error is independent of n_c. Unfortunately, in our case, each instruction has an inherent noise, which is independent of the instruction. So, as the number of instructions executed increases, the error in execution time also increases. Such an error is called *multiplicative*, and a solution by simple linear regression is skewed towards the constructs which have a higher cost, leading to inaccurate results. To overcome this problem, we need to normalize our objective function. We normalize the objective function by the sum of the execution time for each training program over all inputs. Hence, the new objective function is

$$S = \sum_{j=1}^{P} \sum_{i=1}^{M} \left(\frac{T_{(i,j)}}{S_j} - \sum_{c \in \mathcal{C}} \frac{n_c^{(i,j)}}{S_j} T_c \right)^2$$

where $S_j = \sum_{i=1}^{M} T_{(i,j)}$, i.e. the total execution time of the j-th training program. We learn the cost of each construct using the weighted linear regression technique. In addition to the above method, we also employ two other regression techniques. One is robust regression [34], where the objective function is the L_1-norm, instead of the L_2-norm (written as S_{RR} below).

$$S_{RR} = \sum_{j=1}^{P} \sum_{i=1}^{M} \left| \frac{T_{(i,j)}}{S_j} - \sum_{c \in \mathcal{C}} \frac{n_c^{(i,j)}}{S_j} T_c \right|$$

And the other is the non-negative least squares method [28], where the sum of squares is minimized under the constraint that all constants need to be non-negative. We evaluate each algorithm, and compare the results obtained for each regression technique in Section 7.

Training Programs The main goal of training is to learn appropriate values T_c for each program construct c in the language described above. Since we have $|\mathcal{C}|$ variables that we need to learn, all we need are at least $|\mathcal{C}|$ training programs to get a sound cost semantics. However, there is a pitfall here that we want to avoid. Most of the typical machine learning algorithms suffer from the problem of overfitting, i.e. when the model learned is biased towards the training data, and performs poorly on the testing data. Specifically, in our case, the function call construct exists in all the training programs, hence, the learning algorithm overfits the data w.r.t. the cost for function call. Moreover, the regression algorithm is unaware of the fact that these costs need to all be positive. To overcome these issues, we need to linearly separate out the cost of each construct. To this end, we create one training program for each construct. Such a program has a significant count of one construct while being moderate in other constructs. For example, the training program for function call is

let id n = n;;
let rec fapp x = **if** (x = 0) **then** 0 **else** fapp (id (id (id (id (x−1)))));

If we don't account for function inlining, the function `id` gets inlined, and the above is treated as 1 application instead of 5. This massively impacts our training, and we obtain an incorrect cost for function application. Similarly, the training program for integer addition is

let rec fintadd x = **if** (x = 0) **then** 0 **else** x + x + x + x + fintadd (x−1);;

Once we decided the training programs, we ran each training program with 20 inputs, ranging from 1000 to 20000. In this experimentation, we have a total of 36 programs, and with 20 inputs for each program, we have a total of 720 training points for our linear regression algorithm. With this small training set, the regression techniques learn the cost model in less than 1 second.

This set of training might appear overly simplistic but our results show that this simple setup produces already surprisingly satisfying results.

Hardware Platforms All of the above experiments have been performed on two separate platforms. One is an Intel NUC5i5RYH which has a 1.6 GHz 5th generation Intel Core i5-5250U processor based on the x86-64 instruction set. Another is a Raspberry Pi 3 which has a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor based on the ARM v8-A instruction set. We will report the results for both these platforms.

3 Operational Cost Semantics

In the following, we define the big-step operational cost semantics. A value environment V maps variables to values. An evaluation judgment $V \vdash e \Downarrow v \mid t$ denotes that in the environment V, the expression e evaluates to the value v with resource cost t. To differentiate between normal function calls and tail calls, we perform a semantics-preserving program transformation, which adds a tag to all function calls. A tag `tail` is added to tail calls and a tag `normal` is added to all other function calls. Our decision to give a positive cost to the constructs below, while a zero cost to other constructs comes from analyzing the compiled assembly code. Only the constructs below generated assembly instructions with a significant relative execution time. Intuitively, the cost of other constructs can be thought of as absorbed in these constructs. For example, the cost for addition absorbs the cost for the two addends.

We show some illustrative example rules of the semantics, the full set of rules can be foundin Appendix A. We start with operations on booleans and integers.

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{B}}{V \vdash e_1 \;\&\&\; e_2 \Downarrow v_1 \;\&\&\; v_2 \mid t_1 + t_2 + T_{BoolAnd}} \;(\textsc{BoolAnd})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 + e_2 \Downarrow v_1 + v_2 \mid t_1 + t_2 + T_{IntAdd}} \;(\textsc{IntAdd})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 \leq e_2 \Downarrow v_1 \leq v_2 \mid t_1 + t_2 + T_{IntCondLE}} \;(\textsc{IntCondLE})$$

The rules for other boolean, integer and float operations are very similar. For tuples, we introduce two constants $T_{tupleHead}$ and $T_{tupleElem}$. $T_{tupleHead}$ is counted every time we create a tuple, and $T_{tupleElem}$ is counted for the length of the tuple. Similarly, for tuple matching, we count a $T_{tupleMatch}$ for every element in the tuple being matched. When creating a tuple, there is an additional instruction, which assigns a tag to the tuple that represents the tuple constructor. Since there is no such instruction during a tuple match, we have an extra $T_{tupleHead}$ for creating tuples, but not when matching on it.

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \quad \ldots \quad V \vdash e_n \Downarrow v_n \mid t_n}{V \vdash (e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_n) \mid t_1 + \ldots + t_n + T_{tupleHead} + nT_{tupleElem}} \;(\textsc{Tuple})$$

$$\frac{V \vdash tup \Downarrow (v_1, \ldots v_n) \mid t_1 \quad V[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \vdash e \Downarrow v \mid t_2}{V \vdash \text{let } (x_1, \ldots, x_n) = tup \text{ in } e \Downarrow v \mid t_1 + t_2 + n\, T_{tupleMatch}} \;(\textsc{TupleMatch})$$

Since we support higher order functions, we need an evaluation rule for function definition that accounts for closures. We again introduce two constants to deal with function definitions, T_{funDef} for creating a function, and $T_{closure}$ for creating a closure and capturing the free variables.

$$\frac{v = (V, \lambda x.e) \quad |FV(e) \setminus \{x\}| = n}{V \vdash \lambda x.e \Downarrow v \mid T_{funDef} + n\, T_{closure}} \text{ (Closure)}$$

Here, $FV(e)$ denotes the set of free variables of e. Since x is already bounded as the function argument, we remove it from the set of free variables of e'. Finally, we provide the rules for function calls.

$$\frac{\begin{array}{c} V \vdash e_1 \Downarrow (V', \lambda x.e') \mid t_1 \quad V \vdash e_2 \Downarrow v_2 \mid t_2 \\ V'[x \mapsto v_2] \vdash e' \Downarrow v \mid t_3 \quad \texttt{tag}(e_1) = \texttt{tail} \end{array}}{V; TM \vdash \text{app}(e_1, e_2) \Downarrow v \mid t_1 + t_2 + t_3 + T_{TailApp}} \text{ (TailApp)}$$

$$\frac{\begin{array}{c} V \vdash e_1 \Downarrow (V', \lambda x.e') \mid t_1 \quad V \vdash e_2 \Downarrow v_2 \mid t_2 \\ V'[x \mapsto v_2] \vdash e' \Downarrow v \mid t_3 \quad \texttt{tag}(e_1) = \texttt{normal} \end{array}}{V; TM \vdash \text{app}(e_1, e_2) \Downarrow v \mid t_1 + t_2 + t_3 + T_{FunApp}} \text{ (FunApp)}$$

Lastly, we added a constant T_{base} to account for initializations made by each program, irrespective of the program code. Hence, we say that the execution cost of program p is $t + T_{base}$ if $\cdot \vdash p \Downarrow v \mid t$. With these evaluation rules for the cost semantics, we are ready to train our model by learning the values of the constructs described in this section.

4 Learning Memory Allocations

Before analyzing the accuracy of our analysis on execution times, we will demonstrate the effectiveness of our approach by learning a cost semantics for memory allocations. During the experiments we realized that we did not get accurate results for floating point and tuple operations. The reason is that they are generally stored on the heap, but often, optimized to be stored on the stack or the registers:

– The OCaml compiler performs constant propagation to determine which floats and tuples can be treated as globals, and need not be allocated on the heap, every time they are defined.
– If tuples only appear as arguments of a function, they are passed via registers and not allocated on the heap.

To accurately learn a cost semantics for floats and tuples we would need feedback from the OCaml compiler about the optimizations that have been performed. That is why we leave floats and tuples to future work and focus on booleans, integers, and lists. We use the cost semantics that is described in the previous section. According to this semantics, M, the number of bytes allocated by a program is a linear combination $M = \sum_{c \in \mathcal{C}} n_c M_c$.

We use the same training programs for learning memory allocations as for execution times. An interesting point is that the count n_c for each construct remains the same whether executing the training programs for time or memory. Hence, while performing the linear regression, we only need to execute the program on the interpreter once to obtain the counts n_c. We then use the *Gc* module in OCaml to obtain the number M of bytes allocated by the program. Since the memory allocation of a program is constant over different runs, we only need to measure the memory consumption once. For the Intel x86-64 platform, the memory costs of each construct obtained by the linear regression are as follows where $M_x = 0.00$ for all constants M_x that are not listed.

$$M_{base} = 96.03 \qquad M_{FunDef} = 24.00 \qquad M_{closure} = 7.99 \qquad M_{cons} = 24.00$$

An analysis of the OCaml compiler indicates that rounding the learned constants to the nearest integer corresponds exactly to the number of bytes that are allocated by the corresponding construct. For example, our model implies that integers and booleans are not stored on the heap. And the OCaml manual [20] indeed confirms that all integers and booleans are immediate, i.e., stored in registers and on the stack. The value 96 for the constant M_{base} is also confirmed as each program, even without memory allocations, has an initial heap consumption of 96 bytes. The cost $M_{FunDef} = 24$ and $M_{closure} = 8$ for function closures is also confirmed by the OCaml manual. If there are free variables trapped in the closure of a function, there is is an additional memory allocation of 24 bytes on the heap to indicate that the program has a non-empty closure. Every cons constructor consumes 24 bytes on the heap; 8 bytes each for the head and tail, and 8 bytes for the tag to indicate the cons constructor. The empty list ([]) is represented as an immediate integer 0. Hence, the memory consumption of a list of size n is $24n$ bytes.

Similarly, for the ARM v8-A platform, the memory costs of the non-zero constants obtained by the same linear regression are as follows. The results are also as expected and the data size seems to be 4 words.

$$M_{base} = 64.05 \qquad M_{FunDef} = 12.00 \qquad M_{closure} = 3.99 \qquad M_{cons} = 12.00$$

It is notable that we are able to automatically learn the memory representation of OCaml values in the heap. Note that we perform a simple linear regression without the constraint that the learned coefficients need to be integral or non-negative.

5 Learning Execution Times

As mentioned earlier, we used several regression techniques to train our cost semantics: linear regression, robust regression, and non-negative least squares. The accuracy of all three approaches is similar. (See Appendix B for the numbers.) Also, we train on the median execution times since they are less prone to noise than the mean. Below we give the cost of each high-level construct (in nanoseconds) trained using the normalized linear regression technique for the Intel x86-64 architecture. Intuitively, these constants define the median execution time of the respective construct on this specific platform.

$T_{base} = 832.691$	$T_{FunApp} = 1.505$	$T_{TailApp} = 0.156$
$T_{FunDef} = 0.000$	$T_{closure} = 2.921$	$T_{BoolNot} = 0.424$
$T_{BoolAnd} = 0.184$	$T_{BoolOr} = 0.183$	$T_{IntUMinus} = 0.419$
$T_{IntAdd} = 0.297$	$T_{IntSub} = 0.278$	$T_{IntMult} = 1.299$
$T_{IntMod} = 19.231$	$T_{IntDiv} = 19.011$	$T_{FloatUMinus} = 1.232$
$T_{FloatAdd} = 2.102$	$T_{FloatSub} = 2.116$	$T_{FloatMult} = 1.737$
$T_{FloatDiv} = 8.575$	$T_{IntCondEq} = 0.382$	$T_{IntCondLT} = 0.381$
$T_{IntCondLE} = 0.381$	$T_{IntCondGT} = 0.375$	$T_{IntCondGE} = 0.381$
$T_{FloatCondEq} = 0.582$	$T_{FloatCondLT} = 0.619$	$T_{FloatCondLE} = 0.625$
$T_{FloatCondGT} = 0.585$	$T_{FloatCondGE} = 0.629$	$T_{letdata} = 2.828$
$T_{letlambda} = 1.312$	$T_{letrec} = 1.312$	$T_{patternMatch} = 0.223$
$T_{tupleHead} = 5.892$	$T_{tupleElem} = 1.717$	$T_{tupleMatch} = 0.237$

We make several qualitative observations about the learned cost semantics.

- $T_{FunApp} > T_{TailApp}$ indicates that a tail call is cheaper than a normal function call, confirming that tail call optimization reduces the execution time.
- $T_{BoolOr} \approx T_{BoolAnd}$, which is again expected as the && and $\|$ operators just place jump instructions at appropriate locations in the assembly.
- $T_{IntMod} \approx T_{IntDiv} \gg T_{IntMult} > T_{IntAdd} \approx T_{IntSub}$. This is again expected, since whenever we perform integer division or modulo, a check is performed to see if the denominator is 0 and raise an appropriate exception. Hence, division and modulo are much more expensive than multiplication. The latter is more expensive than addition and subtraction.
- $T_{IntCondEq} \approx T_{IntCondLE} \approx T_{IntCondLT} \approx T_{IntCondGT} \approx T_{IntCondGE}$ is confirmed by studying the generated assembly code. A comparison is compiled to comparing the two integers and storing the result in a register, followed by a conditional jump. The analogous observation holds for floating point comparisons.

6 Garbage Collection

The OCaml garbage collector (GC) has 2 major components, the variable size major heap and the fixed size minor heap. Allocations occur first on the minor heap. If the minor heap is full, the GC is invoked. Roughly, the GC frees unreachable cells, and promotes all live variables on the minor heap to the major heap, essentially emptying the minor heap. OCaml employs a generational hypothesis, which states that young memory cells tend to die young, and old cells tend to stay around for longer than young ones.

We roughly model this behavior of the GC. Our hypothesis is that every call to the GC roughly starts with the full minor heap and empties the minor heap. We currently do not model the major heap. Hence, the time taken for each call to the GC in our model is roughly the same. We need two parameters to model the time taken by the GC, one is n_{gc}, which is the number of calls to the GC, and the other is T_{gc}, which is the time taken by 1 call to the GC. Our hypothesis states that $n_{gc} = \left\lfloor \frac{M}{H_0} \right\rfloor$, where H_0 is the size of the minor heap, and M is the total number of memory allocations. Since we can already model the number of heap allocations, all we need is the size of the minor heap.

Conesquently, the two parameters T_{gc} and H_0 can be learnt from our training programs. OCaml offers a *Gc* module, which provides the number of calls to the GC. We use this module to find out the first call to the GC, the number of memory allocations and the time taken by the GC call, thereby learning H_0, which is equal to the number of memory allocations (due to our hypothesis), and T_{gc}. With these parameters learned, the total execution time of the program is

$$T = \sum_{c \in \mathcal{C}} n_c T_c + \left\lfloor \frac{\sum_{c \in \mathcal{C}} n_c M_c}{H_0} \right\rfloor \cdot T_{gc}$$

7 Experiments with Control Programs

For our testing experiment, we used programs from the standard *List* library [2] and an OCaml tutorial [1].

Testing Method For each test program, we first fix inputs of different sizes. For input i, we execute the compiled test program 500 times, measure the execution time, and compute the median T_i^{actual}. We then execute the test program on our interpreter to obtain the count of each construct n_c for $c \in \mathcal{C}$, and calculate $T_i^{expected} = \sum_{c \in \mathcal{C}} n_c T_c$. The average percentage error for each test program is

$$\text{Error}(\%) = \frac{1}{n} \left(\sum_{i=1}^{n} \frac{|T_i^{actual} - T_i^{expected}|}{T_i^{actual}} \right) \times 100$$

Experiments Figure 1 shows the results on a compilation of control programs. `factorialTR` is a tail recursive implementation of factorial. `append` concatenates two lists. `map` is a higher-order map function, which maps a list of integers to booleans. Positive integers are mapped to `true` and negative integers to `false`. `bubblesort` sorts a list using the bubblesort algorithm.

The measurement noise is significant, particularly in cases where high overhead is caused by context switches in the OS. Nevertheless, our prediction is surprisingly accurate given the simplicity of our model. This is particularly true for functions without allocations, like `factorialTR`. For `append` our prediction is very accurate till the point of the first GC cycle ($x \approx 2.9$). The execution time after the first GC cycle is very unpredictable, as we observe that often there are unexpected jumps (e.g. $x \approx 7$) and drops (e.g. $x \approx 14$). For `bubblesort` the GC jumps become invisible since the runtime of the GC is dominated by the actual computation. For all functions, we can very accurately predict the input size at which the GC cycles are triggered. This validates the accuracy of our model for learning memory allocations.

Table 1 summarizes the results obtained by evaluating our approach on 43 control programs. We implemented 3 different training algorithms for training in programs without GC, linear regression (LR), robust regression (RR) and non-negative least squares (NNLS). Each column represents the average percentage

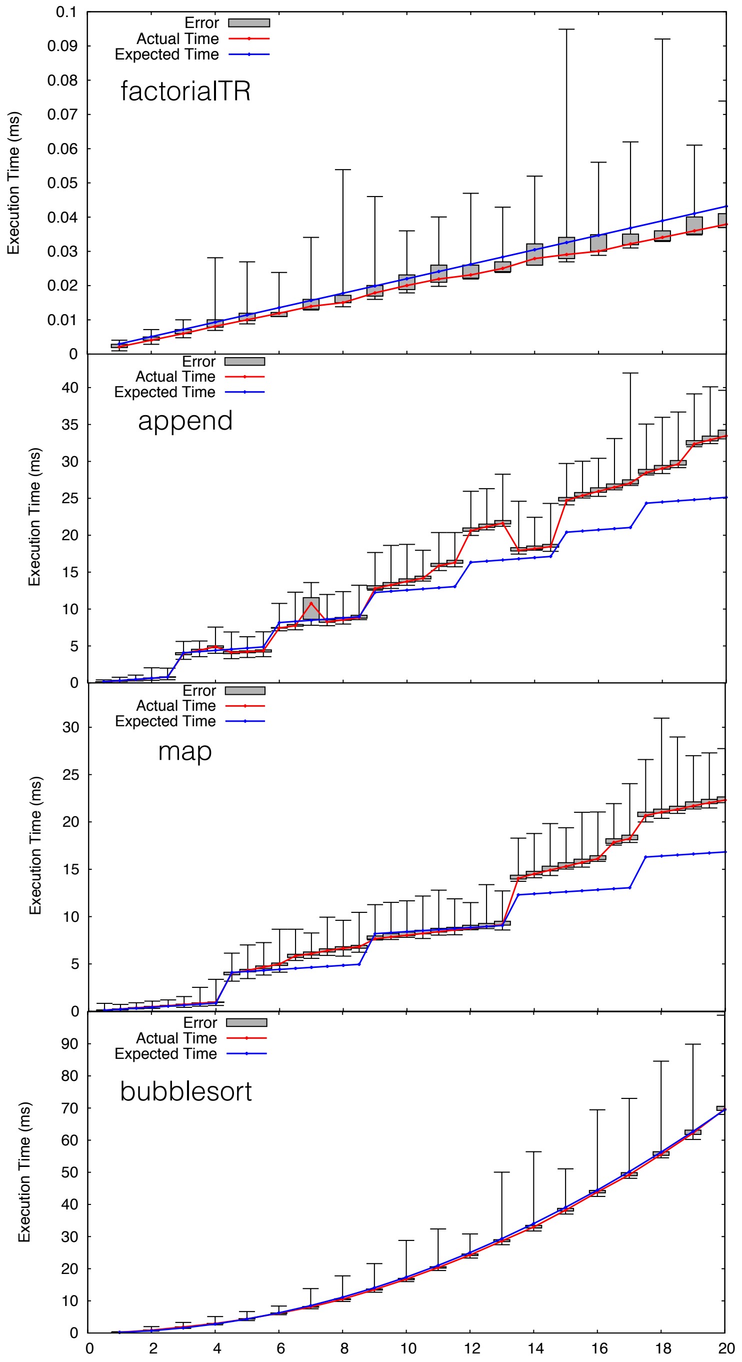

Fig. 1. Graph showing actual and expected time for factorialTR (input sizes $\times 10^3$) (top), append (input $\times 10^4$) (2nd), map (input $\times 10^4$) (3rd), and bubblesort (input $\times 10^2$) (bottom). The red and blue lines denote the actual and expected times, respectively. The vertical bars show the inherent noise in execution time. The lower and upper end of the vertical line denote the minimum and maximum execution time, while the lower and upper end of the box denotes the 1st and 3rd quartile of the execution time.

Architecture	Err (LR)	Err (RR)	Err (NNLS)	Err (GC)
x86-64	13.29	13.04	13.32	19.80
ARM v8-A	21.81	22.94	21.36	20.12

Table 1. Results on x86-64 and ARM architectures

error for both architectures. We also tested all memory-intensive programs with larger inputs to evaluate our cost model for the GC. The last column presents the percentage error for programs with GC cycles. Note that the error increase of programs with GC cycles is not significant, and it indeed decreases for ARM architecture, indicating that our accuracy for GC cycles is also accurate. A detailed description of our experimental results is given in Appendix B.

8 Applications

Integration with Resource Aware ML We have integrated our learned cost semantics into Resource Aware ML[24], a static resource analysis tool for OCaml. RAML is based on an automatic amortized resource analysis (AARA) [25, 23, 24] that is parametric in a user defined cost metric. Such a metric can be defined by providing a constant cost (in floating point) for each syntactic form. This parametricity in a resource metric, and the ability to provide a cost to each syntactic form makes RAML very suitable for our integration purposes. Given an OCaml function, a resource metric, and a maximal degree of the search space of bounds, RAML statically derives a *multivariate resource polynomial* that is an upper bound on the resource usage as defined by the metric. The resource polynomial is parametric in the input sizes of the functions and contains concrete constant factors. The analysis is fully automatic and reduces bound inference to off-the-shelf LP solving. The subset of OCaml that is currently supported by RAML contains all language constructs that we consider in this paper. We used the experiments performed for this work to further refine the cost semantics and automatic analysis. For example, we added an analysis phase prior to the analysis that marks tail calls.

With the new cost metrics, we can use RAML for the first time to make predictions about the worst-case behavior of compiled code. For example, if we use the new execution-time metric (without GC) for x86 then we derive the following bounds in Table 2. The variables in the bounds are the input sizes. The table also contains runtime of the analysis and the number of generated constraint (cumulative for both bounds).

Program	Time Bound (ns)	Heap Bound (B)	Time (s)	#Cons
append	$0.45 + 11.28M$	$24M$	0.02	50
map	$0.60 + 13.16M$	$24M$	0.02	59
insertion sort	$0.45 + 6.06M + 5.83M^2$	$12M + 12M^2$	0.04	298
echelon	$0.60 + 17.29LM^2 + 23.11M + 37.38M^2$	$24LM^2 + 24M + 72M^2$	0.59	16297

Table 2. Symbolic bounds from RAML on x86-64

We can also use RAML to predict the execution time of programs with GC. To this end, we just derive two bounds using the execution metric and the metric for the number of allocations, respectively. We then combine the two bounds using our model for GC which basically, accounts for a constant cost after a constant number of allocations. For example for `append` we obtain the following bound (heap size = 2097448 bytes, GC cycle = 3125429.15 ns on x86).

$$0.45 + 11.28M + \left\lfloor \frac{2097448 \times 24M}{3125429.15} \right\rfloor$$

Since the derived bounds for execution time and allocations are tight, this bound precisely corresponds to the prediction of our model as plotted in Figure 1.

Qualitative Analysis In addition to quantitative validation, we can also infer qualitative results from our learned cost semantics. For instance, we can compare two semantically-equivalent programs, and determine which one is more efficient on a specific hardware. Our model, predicts for example correctly the fastest version of different implementations of *factorial*, *append*, and *sieve of Eratosthenes*. Consider for example our Intel x86 machine and the following versions of `append`. (The other two examples can be found in Appendix C.)

```
let rec append1 l1 l2 =
  match l1 with
    | [] -> l2
    | hd::tl -> hd::(append1 tl l2);;
```

```
let rec append2 l1 l2 = match l1 with
      | [] -> l2
      | x::[] -> x::l2
      | x::y::[] -> x::y::l2
      | x::y::tl -> x::y::(append2 tl l2);;
```

The trade-off in the above implementations is that the first has twice the number of function calls but half the number of pattern matches, as the second one. Since $T_{FunApp} = 1.505 > 4 \times 0.223 = 2 \times T_{patternMatch}$, hence, using our cost semantics concludes that the second version is more efficient. To reach this conclusion we can now analyze the two programs in RAML and automatically derive the execution-time bounds $0.45 + 11.28M$ and $0.45 + 10.53M$ for append1 and append2, respectively. The fact that append2 is faster carries over to the execution-time bounds with GC since the memory allocation bound for both functions is $24M$ bytes.

9 Related Work

The problem of modeling and execution time of programs has been extensively studied for several decades. Static bound analysis on the source level [22, 13, 35, 15, 4, 6, 19, 9, 25] do not take into account compilation and concrete hardware.

Closer related are analyses that target real-time systems by modeling and analyzing worst case execution times (WCET) of programs. Wilhelm et al. [37] provides an overview of these techniques, which can be classified into static [18], measurement-based methods, and simulation [8]. Lim et al. [29] associate a worst case timing abstraction containing detailed information of every execution path to get tighter WCET bounds. Colin and Puaut [17] study the effect of branch prediction on WCET. The goals of our work are different since we are not aiming at a sound bound of the worst-case but rather an approximation of the average case. Advantages of our approach include hardware independence, modeling of GC, and little manual effort after the cost semantics is defined.

Lambert et al. [27] introduced a hardware independent method of estimating time cost of Java bytecode instructions. Unlike our work, they do not take into account GC and compilation. Huang et al. [26] build accurate prediction models of program performance using program execution on sample inputs using sparse polynomial regression. The difference to our work is that they build a model for one specific program, are not interested in low-level features, and mainly want to predict the (high-level) execution time for a given input.

Acar et al. [3] learn cost models for execution time to determine whether tasks need to run sequentially or in parallel. They observe executions to learn the cost of one specific program. In contrast, we build a cost semantics to make predictions for all programs. There exist many works that build on high-level cost semantics[23], for instance to model cache and I/O effects [12]. However, these semantics do not incorporate concrete constants for specific hardware.

10 Conclusion and Future Work

We have presented an operational cost semantics learned using standard machine learning techniques, like linear regression, robust regression, etc. These semantics were able to model the execution time of programs with surprising accuracy; even in the presence of compilation and garbage collection. Since both these models can be learned without relying on hardware specifics, our method is completely hardware independent and easily extensible to other hardware platforms. We have also presented an integration of the cost semantics with RAML, hence, allowing static analyzers to predict the execution time and heap allocations of assembly code for the first time.

One of the significant future directions is a more precise model for the garbage collector. Our model is limited to the minor heap, we need a model for the major heap and heap compactions as well. The size of the major heap is variable, hence, modeling the major heap is an important and complicated problem. We also need to incorporate other language features, especially user-defined data types in our semantics. Another challenge with real-world languages is compiler optimizations. We modeled a few optimizations in these semantics, but we should extend our semantics to incorporate more. Since these optimizations are performed at compile time, using static analysis techniques, it should be possible to model all of them. Finally, we think that it is possible to extend this technique to other programming languages, and we only need an appropriate interpreter to achieve that. We would like to validate this claim. We believe this connection between high-level program constructs and low-level program resources like time and memory is a first step towards connecting theoretical features of a language and its practical applications.

References

1. 99 problems (solved) in ocaml. `https://ocaml.org/learn/tutorials/99problems.html`, accessed: 2016-08-16
2. Module list. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`, accessed: 2016-08-16
3. Acar, U.A., Charguéraud, A., Rainey, M.: Oracle scheduling: Controlling granularity in implicitly parallel languages. In: Proc. of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 499–518. OOPSLA '11, ACM, New York, NY, USA (2011)
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Automatic Inference of Resource Consumption Bounds. In: Logic for Programming, Artificial Intelligence, and Reasoning, 18th Conference (LPAR'12). pp. 1–11 (2012)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: 16th Euro. Symp. on Prog. (ESOP'07). pp. 157–172 (2007)
6. Albert, E., Fernández, J.C., Román-Díez, G.: Non-cumulative Resource Analysis. In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15). pp. 85–100 (2015)
7. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In: 17th Int. Static Analysis Symposium (SAS'10). pp. 117–133 (2010)
8. Austin, T., Larson, E., Ernst, D.: Simplescalar: An infrastructure for computer system modeling. Computer 35(2), 59–67 (Feb 2002)
9. Avanzini, M., Lago, U.D., Moser, G.: Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In: 29th Int. Conf. on Functional Programming (ICFP'15) (2012)
10. Backes, M., Doychev, G., Köpf, B.: Preventing Side-Channel Leaks in Web Traffic: A Formal Approach. In: Proc. 20th Network and Distributed Systems Security Symposium (NDSS) (2013)
11. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: ABC: Algebraic Bound Computation for Loops. In: Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10). pp. 103–118 (2010)
12. Blelloch, G.E., Harper, R.: Cache and i/o efficent functional algorithms. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 39–50. POPL '13, ACM, New York, NY, USA (2013)
13. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating Runtime and Size Complexity Analysis of Integer Programs. In: Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14). pp. 140–155 (2014)
14. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional Certified Resource Bounds. In: 36th Conference on Programming Language Design and Implementation (PLDI'15) (2015)
15. Cerný, P., Henzinger, T.A., Kovács, L., Radhakrishna, A., Zwirchmayr, J.: Segment Abstraction for Worst-Case Execution Time Analysis. In: 24th European Symposium on Programming (ESOP'15). pp. 105–131 (2015)
16. Chen, W.Y., Chang, P.P., Conte, T.M., Hwu, W.W.: The effect of code expanding optimizations on instruction cache design. IEEE Transactions on Computers 42(9), 1045–1057 (Sep 1993)
17. Colin, A., Puaut, I.: Worst case execution time analysis for a processor with branch prediction. Real-Time Systems 18(2), 249–274 (2000)
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977)

19. Danner, N., Licata, D.R., Ramyaa, R.: Denotational Cost Semantics for Functional Languages with Inductive Types. In: 29th Int. Conf. on Functional Programming (ICFP'15) (2012)
20. Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The ocaml system release 4.03. `http://caml.inria.fr/pub/docs/manual-ocaml/`
21. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09). pp. 127–139 (2009)
22. Gulwani, S., Zuleger, F.: The Reachability-Bound Problem. In: Conf. on Prog. Lang. Design and Impl. (PLDI'10). pp. 292–304 (2010)
23. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th Symposium on Principles of Programming Languages (POPL'11) (2011)
24. Hoffmann, J., Das, A., Weng, S.C.: Towards Automatic Resource Bound Analysis for OCaml. In: 44th Symposium on Principles of Programming Languages (POPL'17) (2017), forthcoming
25. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03). pp. 185–197 (2003)
26. Huang, L., Jia, J., Yu, B., gon Chun, B., Maniatis, P., Naik, M.: Predicting execution time of computer programs using sparse polynomial regression. In: Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S., Culotta, A. (eds.) Advances in Neural Information Processing Systems 23, pp. 883–891. Curran Associates, Inc. (2010)
27. Lambert, J.M., Power, J.F.: Platform independent timing of java virtual machine bytecode instructions. Electronic Notes in Theoretical Computer Science pp. 97 – 113 (2008)
28. Lawson, C., Hanson, R.: Solving Least Squares Problems. Classics in Applied Mathematics, Society for Industrial and Applied Mathematics (1995)
29. Lim, S.S., Bae, Y.H., Jang, G.T., Rhee, B.D., Min, S.L., Park, C.Y., Shin, H., Park, K., Moon, S.M., Kim, C.S.: An accurate worst case timing analysis for risc processors. IEEE Transactions on Software Engineering 21(7), 593–604 (Jul 1995)
30. Neter, J., Kutner, M.H., Nachtsheim, C.J., Wasserman, W.: Applied linear statistical models, vol. 4. Irwin Chicago (1996)
31. Ngo, V.C., Dehesa-Azuara, M., Fredrikson, M., Hoffmann, J.: Quantifying and Preventing Side Channels with Substructural Type Systems (2016), working paper
32. Olivo, O., Dillig, I., Lin, C.: Static Detection of Asymptotic Performance Bugs in Collection Traversals. In: Conference on Programming Language Design and Implementation (PLDI'15). pp. 369–378 (2015)
33. Rencher, A.C., Christensen, W.F.: Multivariate Regression, pp. 339–383. John Wiley & Sons, Inc. (2012), `http://dx.doi.org/10.1002/9781118391686.ch10`
34. Rousseeuw, P.J., Leroy, A.M.: Robust Regression and Outlier Detection. John Wiley & Sons, Inc., New York, NY, USA (1987)
35. Sinn, M., Zuleger, F., Veith, H.: A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In: Computer Aided Verification - 26th Int. Conf. (CAV'14). p. 743759 (2014)
36. Steele, Jr., G.L.: Debunking the 'expensive procedure call' myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In: Proceedings of the 1977 Annual Conference. pp. 153–162. ACM '77, ACM, New York, NY, USA (1977)
37. Wilhelm, R., et al.: The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008)
38. Zuleger, F., Sinn, M., Gulwani, S., Veith, H.: Bound Analysis of Imperative Programs with the Size-change Abstraction. In: 18th Int. Static Analysis Symp. (SAS'11). pp. 280–297 (2011)

A Detailed Cost Semantics

This section gives the complete cost semantics. We denote the value environment by V and we write $V \vdash e \Downarrow v \mid t$ for the expression e evaluating to the value v in cost t under environment V. Before evaluating the program, we perform a simple semantics-preserving program transformation, where we add a tag `tail` to all tail calls, and a tag `normal` to all the other function calls.

$$\frac{v = (V, \lambda x.e) \quad |FV(e) \setminus \{x\}| = n}{V \vdash \lambda x.e \Downarrow v \mid T_{funDef} + n\, T_{closure}} \ (\textsc{Closure})$$

$$\frac{\begin{array}{c} V \vdash e_1 \Downarrow (V', \lambda x.e') \mid t_1 \quad V \vdash e_2 \Downarrow v_2 \mid t_2 \\ V'[x \mapsto v_2] \vdash e' \Downarrow v \mid t_3 \quad \texttt{tag}(e_1) = \texttt{tail} \end{array}}{V; TM \vdash \text{app}(e_1, e_2) \Downarrow v \mid t_1 + t_2 + t_3 + T_{TailApp}} \ (\textsc{TailApp})$$

$$\frac{\begin{array}{c} V \vdash e_1 \Downarrow (V', \lambda x.e') \mid t_1 \quad V \vdash e_2 \Downarrow v_2 \mid t_2 \\ V'[x \mapsto v_2] \vdash e' \Downarrow v \mid t_3 \quad \texttt{tag}(e_1) = \texttt{normal} \end{array}}{V; TM \vdash \text{app}(e_1, e_2) \Downarrow v \mid t_1 + t_2 + t_3 + T_{FunApp}} \ (\textsc{FunApp})$$

These rules define functions. We now define boolean arithmetic.

$$\frac{V \vdash e \Downarrow v \mid t \qquad v \in \mathbb{B}}{V \vdash \text{not } e \Downarrow \neg v \mid t + T_{BoolNot}} \ (\textsc{BoolNot})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{B}}{V \vdash e_1 \ \&\& \ e_2 \Downarrow v_1 \wedge v_2 \mid t_1 + t_2 + T_{BoolAnd}} \ (\textsc{BoolAnd})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{B}}{V \vdash e_1 \ || \ e_2 \Downarrow v_1 \vee v_2 \mid t_1 + t_2 + T_{BoolOr}} \ (\textsc{BoolOr})$$

We now consider rules for integer arithmetic.

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 + e_2 \Downarrow v_1 + v_2 \mid t_1 + t_2 + T_{IntAdd}} \ (\textsc{IntAdd})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 - e_2 \Downarrow v_1 - v_2 \mid t_1 + t_2 + T_{IntSub}} \ (\textsc{IntSub})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 * e_2 \Downarrow v_1 * v_2 \mid t_1 + t_2 + T_{IntMult}} \ (\textsc{IntMult})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 \text{ mod } e_2 \Downarrow v_1 \text{ mod } v_2 \mid t_1 + t_2 + T_{IntMod}} \ (\textsc{IntMod})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 / e_2 \Downarrow v_1 / v_2 \mid t_1 + t_2 + T_{IntDiv}} \text{ (IntDiv)}$$

$$\frac{V \vdash e \Downarrow v \mid t \qquad v \in \mathbb{Z}}{V \vdash -e \Downarrow -v \mid t + T_{IntUMinus}} \text{ (IntUMinus)}$$

We skip float arithmetic, which is completely analogous to integer arithmetic as defined above. We move on to defining integer comparisons.

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 = e_2 \Downarrow v_1 = v_2 \mid t_1 + t_2 + T_{IntCondEq}} \text{ (IntCondEq)}$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 < e_2 \Downarrow v_1 < v_2 \mid t_1 + t_2 + T_{IntCondLT}} \text{ (IntCondLT)}$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 \leq e_2 \Downarrow v_1 \leq v_2 \mid t_1 + t_2 + T_{IntCondLE}} \text{ (IntCondLE)}$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 > e_2 \Downarrow v_1 > v_2 \mid t_1 + t_2 + T_{IntCondGT}} \text{ (IntCondGT)}$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V \vdash e_2 \Downarrow v_2 \mid t_2 \qquad v_1, v_2 \in \mathbb{Z}}{V \vdash e_1 \geq e_2 \Downarrow v_1 \geq v_2 \mid t_1 + t_2 + T_{IntCondGE}} \text{ (IntCondGE)}$$

The semantics float comparisons are analogous to integer comparisons, so we will omit them. We instead move on to tuples and pattern matches.

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad \ldots \qquad V \vdash e_n \Downarrow v_n \mid t_n}{V \vdash (e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_n) \mid t_1 + \ldots + t_n + T_{tupleHead} + n\ T_{tupleElem}} \text{ (Tuple)}$$

$$\frac{\begin{array}{c} V \vdash e' \Downarrow (v_1, \ldots v_n) \mid t_1 \\ V[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n] \vdash e \Downarrow v \mid t_2 \end{array}}{V \vdash \text{let } (x_1, \ldots, x_n) = e' \text{ in } e \Downarrow v \mid t_1 + t_2 + n\ T_{tupleMatch}} \text{ (TupleMatch)}$$

$$\frac{V \vdash e \Downarrow \text{nil} \mid t_1 \qquad V \vdash e_0 \Downarrow v \mid t_2}{V \vdash \text{match } (e, e_0, x.y.e_1) \Downarrow v \mid t_1 + t_2 + T_{patternMatch}} \text{ (PatMatchNil)}$$

$$\frac{V \vdash e \Downarrow \text{cons}(h, t) \mid t_1 \qquad V[x \mapsto h][y \mapsto t] \vdash e_1 \Downarrow v \mid t_2}{V \vdash \text{match } (e, e_0, x.y.e_1) \Downarrow v \mid t_1 + t_2 + 2\ T_{patternMatch}} \text{ (PatMatchCons)}$$

The reason we count $T_{patternMatch}$ differently in the nil and cons case is because in the assembly, pattern matches are compiled down to conditionals. In the nil case, there is only one conditional check, while in the cons case, there are two conditional checks. Moving on to function definitions, we observe that in terms of assembly code, let is treated differently depending on whether defining a function or an expression of a data type. Hence, we perform another program transformation, basically tagging the expressions of function type as `lambda`, and expressions of data type as `data`.

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V[x \mapsto v_1] \Downarrow v_2 \mid t_2 \qquad \texttt{tag}(e_1) = \texttt{lambda}}{V \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v_2 \mid t_1 + t_2 + T_{letlambda}} \;(\textsc{LetLambda})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V[x \mapsto v_1] \Downarrow v_2 \mid t_2 \qquad \texttt{tag}(e_1) = \texttt{data}}{V \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v_2 \mid t_1 + t_2 + T_{letdata}} \;(\textsc{LetBase})$$

$$\frac{V \vdash e_1 \Downarrow v_1 \mid t_1 \qquad V[x \mapsto v_1] \Downarrow v_2 \mid t_2}{V \vdash \text{let rec } x = e_1 \text{ in } e_2 \Downarrow v_2 \mid t_1 + t_2 + T_{letrec}} \;(\textsc{LetRec})$$

B Detailed Experimentation

We give a detailed description of results in this section. Tables 3 and 4 describe our test functions. We test each program on several inputs. The start input, the end input and the increment are described in columns Start, End and Inc respectively. LOC denotes the lines of code for each test program. Table 5 describes the results obtained on each test program for the Intel x86-64 architecture. We implemented 3 training algorithms, linear regression (LR), robust regression (RR) and non-negative least squares (NNLS). Table 5 describes the percentage error obtained on each training algorithm. The percentage error is calculated by

$$Error(\%) = \frac{1}{n}\left(\sum_{i=1}^{n} \frac{|T_i^{actual} - T_i^{expected}|}{T_i^{actual}}\right) \times 100$$

where T_i^{actual} and $T_i^{expected}$ denote the actual and expected execution time for the i-th input of the test program. Table 6 describes the same results for the ARM architecture. Note that since our training and testing algorithm are completely hardware independent, we simply implement the training and test programs on a new architecture to test the effectiveness of our method. Table 7 describes the percentage error for the same programs with larger inputs. Experiments for the garbage collector are performed by increasing the size of the inputs by a factor of 10. Some of the functions are marked with (HOF) indicating that these functions are higher-order functions, taking a function as an argument. Figures 2-7 present the graphs for more experiments with our cost semantics. As in Figure 1, these experiments are also performed on x86 using linear regression as the training algorithm.

C Qualitative Analysis

We consider two more examples of functions with same asymptotic complexity, and determine which one is faster on a specific hardware.

Factorial

Consider the following two versions of the `factorial` function.

```
let rec fact1 n =
        if (n = 0) then 1 else n * fact1 (n−1);;

let rec facth n res =
        if (n = 0) then 1 else facth (n−1) (n*res);;

let fact2 n = facth n 1;;
```

Observing the cost semantics, we see that $T_{FunApp} = 1.505 >> 0.156 = T_{TailApp}$. Since function application is the dominant cost for this function, our cost model indeed confirms that `fact2` is more time efficient than `fact1`.

Sieve of Eratosthenes

Sieve of Eratosthenes is a standard method for computing the list of prime numbers from the list of all natural numbers. This method involves removing all multiples of a prime number successively from the list. Consider two implementations of the `remove` function.

```
let rec remove1 l n =
        match l with
        | [] −> []
        | hd::tl −>
                if (hd mod n = 0) then remove1 tl n
                else hd::(remove1 tl n);;

let rec removeh l mul n =
        match l with
        | [] −> []
        | hd::tl −>
                if (hd = mul) then removeh tl (mul+n) n
                else if (hd > mul) then
                        hd::(removeh tl (mul+n) n)
                else hd::(removeh tl (mul+n) n);;

let remove2 l n = removeh l (2*n) n;;
```

Again, testing the two functions with our cost semantics, we infer that the second implementation is faster because $T_{IntMod} = 19.231 >> 0.382 + 0.375 = T_{IntEq} + T_{IntCondGt}$.

D Execution Times

We give the intuitive definition of the cost associated with each construct here.

1. T_{base} denotes the base time of a program, irrespective of the program contents, i.e. the time taken for minimum initializations.
2. T_{FunApp} and $T_{TailApp}$ denote the time for usual function call and tail call, respectively.
3. T_{FunDef} and $T_{closure}$ denote the time for defining a function, and its closure, respectively.
4. $T_{BoolNot}$, $T_{BoolAnd}$, T_{BoolOr} denote the time for boolean computations.
5. $T_{IntUMinus}$, T_{IntAdd}, T_{IntSub}, $T_{IntMult}$, T_{IntMod} and T_{IntDiv} denote the time for integer arithmetic computation.
6. $T_{FloatUMinus}$, $T_{FloatAdd}$, $T_{FloatSub}$, $T_{FloatMult}$ and $T_{FloatDiv}$ denote the time for floating point arithmetic computation.
7. $T_{IntCondEq}$, $T_{IntCondLT}$, $T_{IntCondLE}$, $T_{IntCondGT}$ and $T_{IntCondGE}$ denote the time for integer comparisons.
8. $T_{FloatCondEq}$, $T_{FloatCondLT}$, $T_{FloatCondLE}$, $T_{FloatCondGT}$ and $T_{FloatCondGE}$ denote the time for integer comparisons.
9. $T_{letdata}$ and $T_{letlambda}$ denote the time for a `let` statement for a base type (int, float, bool, etc.) and for a function respectively. T_{letrec} denotes the time for defining a recursive function.
10. $T_{tupleHead}$ and $T_{tupleElem}$ denote the time for creating tuples.
11. $T_{patternMatch}$ and $T_{tupleMatch}$ denote the time for pattern matches and tuple matches, respectively.

Name	Start	End	Inc	LOC	Description
Standard List library and OCaml tutorial (list)					
append	1000	20000	1000	12	appends one list to another
all (HOF)	1000	20000	1000	15	checks if all elements of list satisfy a predicate
compress	1000	20000	1000	19	eliminates consecutive duplicates of a list
drop	1000	20000	1000	15	drops every N-th element of list
duplicate	1000	20000	1000	14	duplicates elements of list
encode	1000	20000	1000	30	performs run-length encoding of list
eq	1000	20000	1000	24	checks if two lists are equal
exists (HOF)	1000	20000	1000	15	checks if there exists an element in the list satisfying a particular predicate
fastappend	1000	20000	1000	24	appends one list to another by pattern matching on several elements in each iteration
filter (HOF)	1000	20000	1000	15	filters out elements not satisfying a particular predicate
flatten	1000	20000	1000	26	flatten a nested list structure
foldl (HOF)	1000	20000	1000	15	standard foldl implementation from list library
foldr (HOF)	1000	20000	1000	16	standard foldr implementation from list library
insertAt	1000	20000	1000	12	insert an element at a given position
last	1000	20000	1000	17	returns the last element of list
listiter	1000	20000	1000	13	iterates over list and returns unit
map (HOF)	1000	20000	1000	18	standard map implementation from list library
pack	1000	20000	1000	30	packs consecutive duplicates of list elements into sublists
removeat	1000	20000	1000	12	removes the element at a given position
replicate	500	12000	500	27	replicates the elements of a list a given number of times
reverse	1000	20000	1000	15	reverses a list
rotate	1000	20000	1000	40	rotates a list by a given quantity
slice	1000	20000	1000	20	extracts a slice from a list
split	1000	20000	1000	24	split a list into two parts, given the size of first part
appendTR	1000	20000	1000	22	tail recursive implementation of append
isort	100	1000	100	19	insertion sort on a list
isort (HOF)	50	400	20	35	higher order insertion sort on a list of lists sorted by list size
last_two	1000	20000	1000	20	extracts last two elements of list
at	1000	20000	1000	13	returns the element at a given position in a list
length	1000	20000	1000	14	returns the length of list
reverseTR	1000	20000	1000	13	tail recursive implementation of reverse
palindrome	1000	20000	1000	27	checks if a list is a palindrome
range	1000	20000	1000	7	creates a list from a range of numbers

Table 3. Description of Test Programs

Name	**Start**	**End**	**Inc**	**LOC**	**Description**
OCaml tutorial (arithmetic)					
add	100000	2000000	100000	7	adds two numbers by recursively taking successor
factorialTR	1000	20000	1000	7	tail recursive implementation of factorial
factorial	100	2000	100	7	factorial of a number
fibonacci	8	20	1	7	computes numbers of Fibonacci sequence
mult	1000	20000	1000	8	multiplies a list by successively adding
phi	1000	20000	1000	15	computes the Euler's totient function
factors	1001	20001	1000	9	creates a list of all factors of a given number
OCaml project (matrix)					
matrix add	100	2000	100	31	adds two n-dimensional matrices
matrix sub	100	2000	100	31	subtracts two n-dimensional matrices
matrix mult	10	40	2	36	multiplies two n-dimensional matrices

Table 4. Description of Test Programs

Name	Error (%) (LR)	Error (%) (RR)	Error (%) (NNLS)
Standard List library and OCaml tutorial (list)			
append	13.68	15.91	13.37
all (HOF)	1.21	1.54	1.29
compress	7.02	6.69	5.17
drop	15.39	17.61	15.04
duplicate	15.40	17.05	15.68
encode	4.56	2.02	2.92
eq	37.29	42.24	40.44
exists (HOF)	1.20	1.51	1.59
fastappend	11.55	12.94	10.27
filter (HOF)	31.17	17.13	31.44
flatten	17.21	32.50	17.69
foldl (HOF)	1.12	1.56	1.60
foldr (HOF)	12.14	12.36	13.15
insertAt	5.73	1.90	6.40
last	22.92	21.41	21.94
listiter	1.60	2.19	0.99
map (HOF)	19.80	20.96	19.08
pack	2.32	1.72	1.84
removeat	5.51	1.64	5.63
replicate	2.85	0.85	3.64
reverse	1.94	1.27	4.41
rotate	16.27	17.98	16.46
slice	6.29	5.31	6.72
split	6.89	1.38	5.96
appendTR	2.93	1.32	1.43
isort	26.61	27.30	26.49
isort (HOF)	23.33	20.81	21.18
last_two	42.62	46.96	41
at	5.52	1.98	6.66
length	3.44	1.25	1.76
reverseTR	2.44	0.97	2.34
palindrome	9.12	8.91	8.84
range	30.90	33.03	30.48
OCaml tutorial (arithmetic)			
add	0.76	4.22	0.74
factorialTR	11.77	7.88	10.67
factorial	9.03	8.31	8.15
fibonacci	8.87	4.66	9.31
mult	2.81	6.72	2.08
phi	10.43	10.50	10.46
factors	76.05	74.68	73.99
OCaml project (matrix)			
matrix add	11.91	12.37	10.11
matrix sub	11.55	12.32	9.76
matrix mult	20.70	19.02	34.69

Table 5. Results of different ML techniques on Intel x86-64 architecture

Name	Error (%) (LR)	Error (%) (RR)	Error (%) (NNLS)
Standard List library and OCaml tutorial (list)			
append	26.56	30.55	26.57
all (HOF)	14.35	15.81	12.56
compress	29.88	30.15	31.54
drop	30.84	33.57	31.48
duplicate	30.74	34.61	31.20
encode	18.62	20.43	19.76
eq	8.63	13.98	4.94
exists (HOF)	22.02	23.31	20.26
fastappend	19.35	24.02	20.04
filter (HOF)	44.48	46.96	44.16
flatten	38.04	39.65	38.52
foldl (HOF)	39.99	41.04	38.56
foldr (HOF)	33.52	33.75	32.73
insertAt	1.85	7.08	1.72
last	4.91	1.22	7.20
listiter	6.07	7.06	5.81
map (HOF)	33.44	36.46	37.52
pack	19.23	21.00	20.48
removeat	1.88	8.04	1.86
replicate	8.60	13.74	7.56
reverse	6.56	11.52	6.86
rotate	35.15	35.34	35.26
slice	0.69	0.95	0.98
split	0.96	8.12	1.05
appendTR	10.72	13.76	10.85
isort	59.48	59.65	59.87
isort (HOF)	61.31	64.46	27.93
last_two	8.26	3.79	12.44
at	1.14	8.04	0.87
length	8.29	9.43	8.09
reverseTR	7.43	11.00	5.66
palindrome	1.31	4.75	6.02
range	33.22	38.79	33.21
OCaml tutorial (arithmetic)			
add	14.80	15.53	2.89
factorialTR	105.56	98.66	107.16
factorial	30.71	16.42	11.03
fibonacci	28.18	30.23	32.03
mult	13.70	4.70	15.34
phi	0.86	1.04	2.22
factors	6.53	5.62	5.05

Table 6. Results of different ML techniques on ARM architecture

Name	Start	End	Inc	Error (%) (x86)	Error (%) (ARM)
append	1000	200000	1000	16.02	18.60
all (HOF)	1000	200000	1000	11.60	5.76
compress	1000	200000	1000	17.54	17.87
drop	1000	200000	1000	15.57	18.99
duplicate	1000	200000	1000	9.65	9.78
encode	1000	200000	1000	10.70	18.84
eq	1000	200000	1000	17.88	8.40
exists (HOF)	1000	200000	1000	11.54	4.44
fastappend	1000	200000	1000	10.54	10.53
filter (HOF)	1000	200000	1000	12.34	14.20
flatten	1000	200000	1000	19.87	26.91
foldl (HOF)	1000	200000	1000	10.63	6.66
foldr (HOF)	1000	200000	1000	3.24	14.52
insertAt	1000	200000	1000	21.10	8.65
last	1000	200000	1000	12.89	5.99
listiter	1000	200000	1000	10.80	5.17
map (HOF)	1000	200000	1000	17.32	21.60
pack	1000	200000	1000	10.69	19.08
removeat	1000	200000	1000	20.55	8.99
replicate	500	120000	500	8.67	7.98
reverse	1000	200000	1000	20.69	14.43
rotate	1000	200000	1000	18.06	23.33
slice	1000	200000	1000	12.48	6.97
split	1000	200000	1000	20.77	8.47
appendTR	1000	200000	1000	25.31	23.83
isort	100	10000	100	190.78	212.51
isort (HOF)	50	4000	20	27.93	32.17
last_two	1000	200000	1000	13.42	5.38
at	1000	200000	1000	20.99	8.63
length	1000	200000	1000	11.03	5.18
reverseTR	1000	200000	1000	21.71	14.84
palindrome	1000	200000	1000	6.02	4.32
range	1000	200000	1000	14.33	15.72

Table 7. Results on x86-64 and ARM with garbage collector

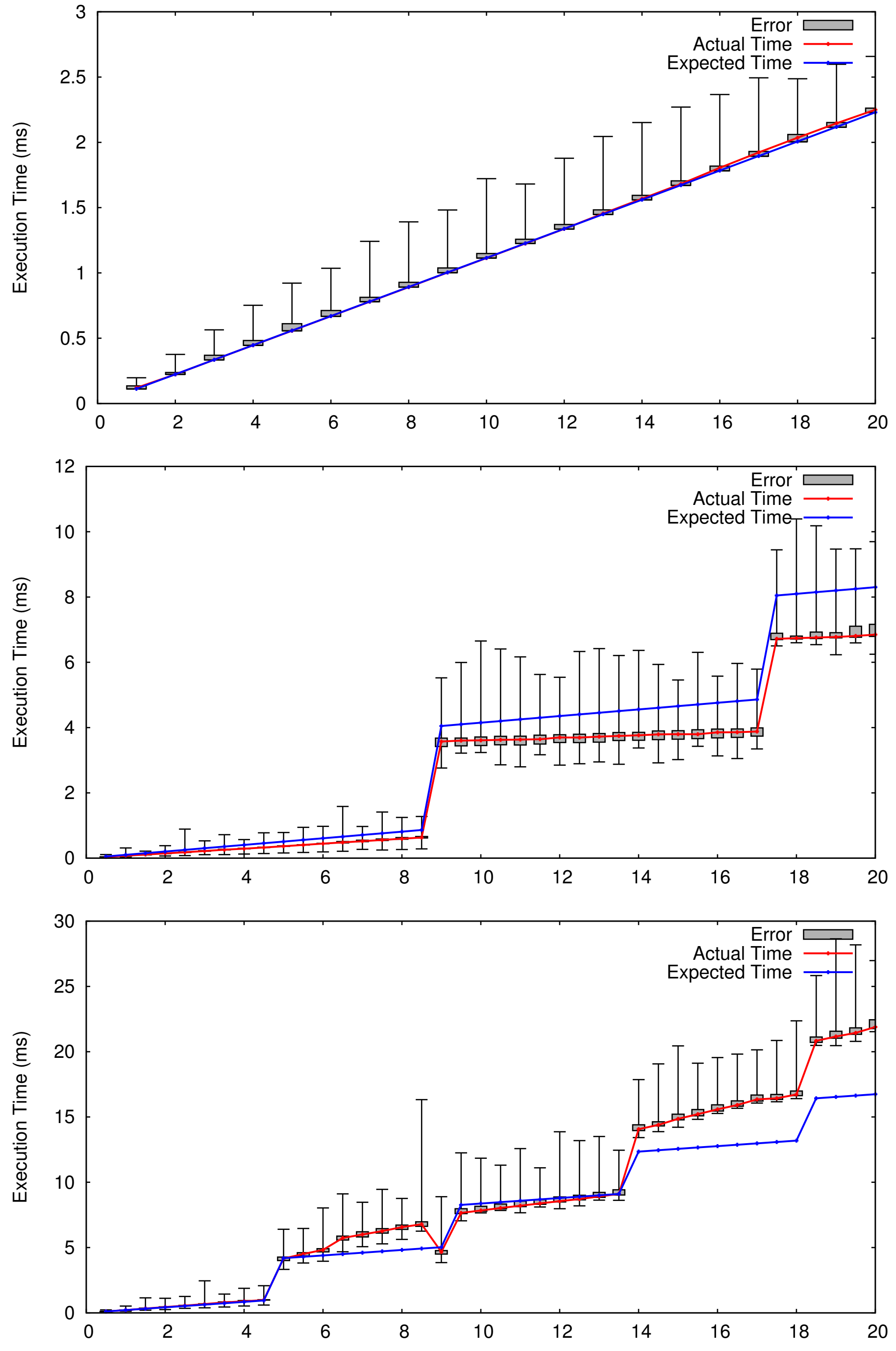

Fig. 2. Graph showing actual and expected time for add (input $\times 10^5$) (top), at (input $\times 10^4$) (middle) and drop (input $\times 10^4$) (bottom)

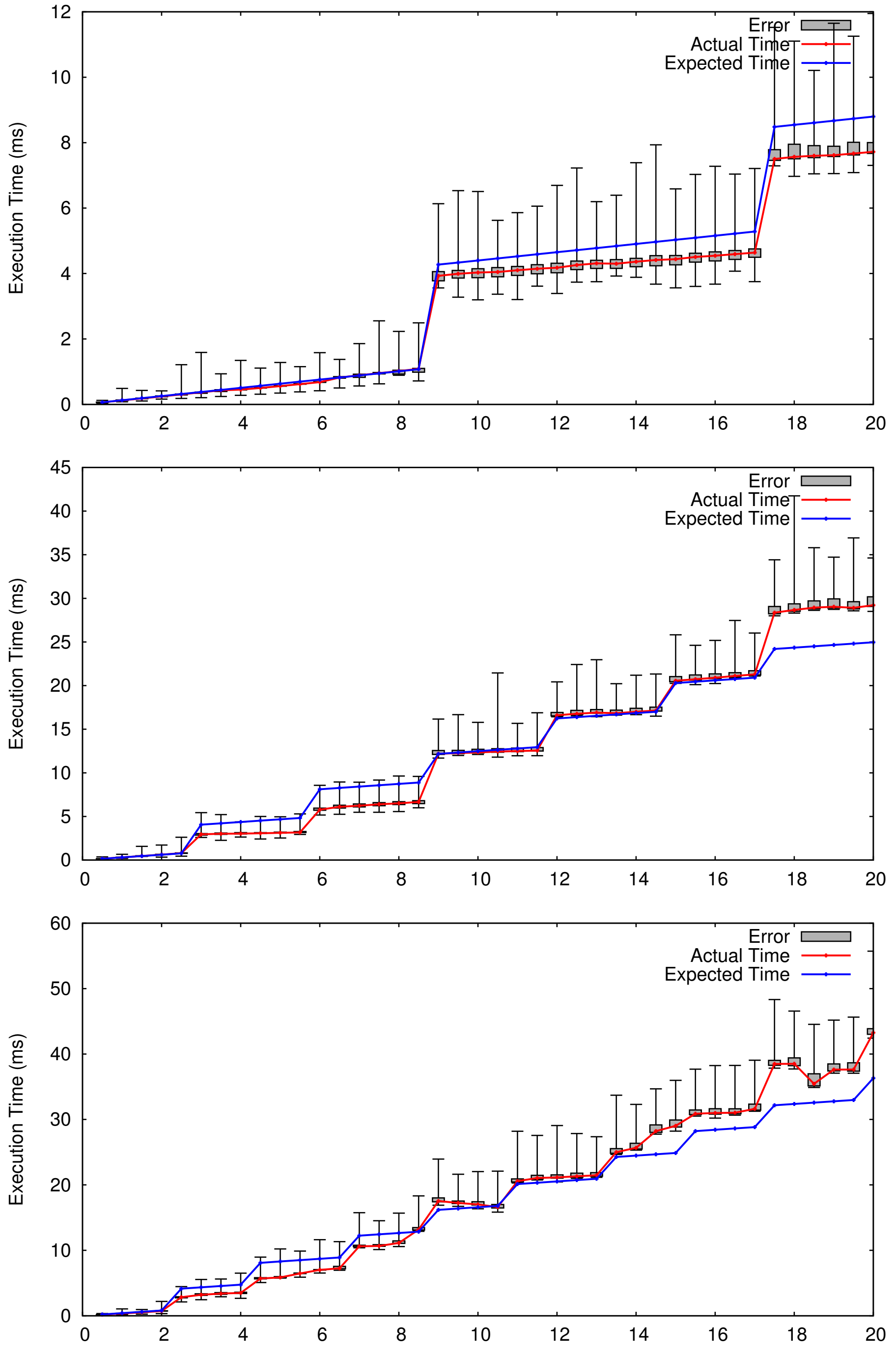

Fig. 3. Graph showing actual and expected time for all (HOF) (input $\times 10^4$) (top), duplicate (input $\times 10^4$) (middle) and encode (input $\times 10^4$) (bottom)

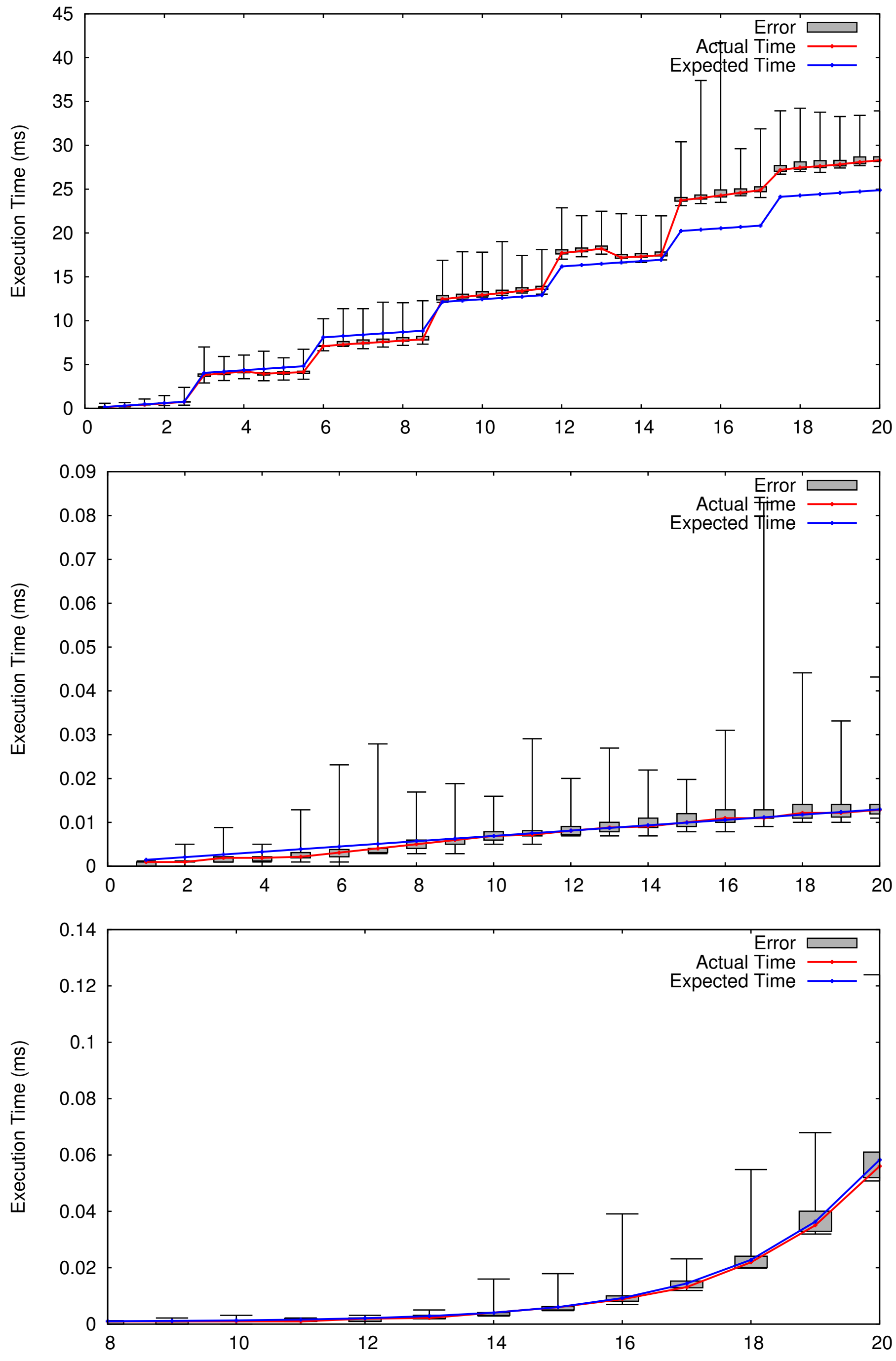

Fig. 4. Graph showing actual and expected time for fastappend (input $\times 10^4$) (top), factorial (input $\times 10^2$) (middle) and fibonacci (bottom)

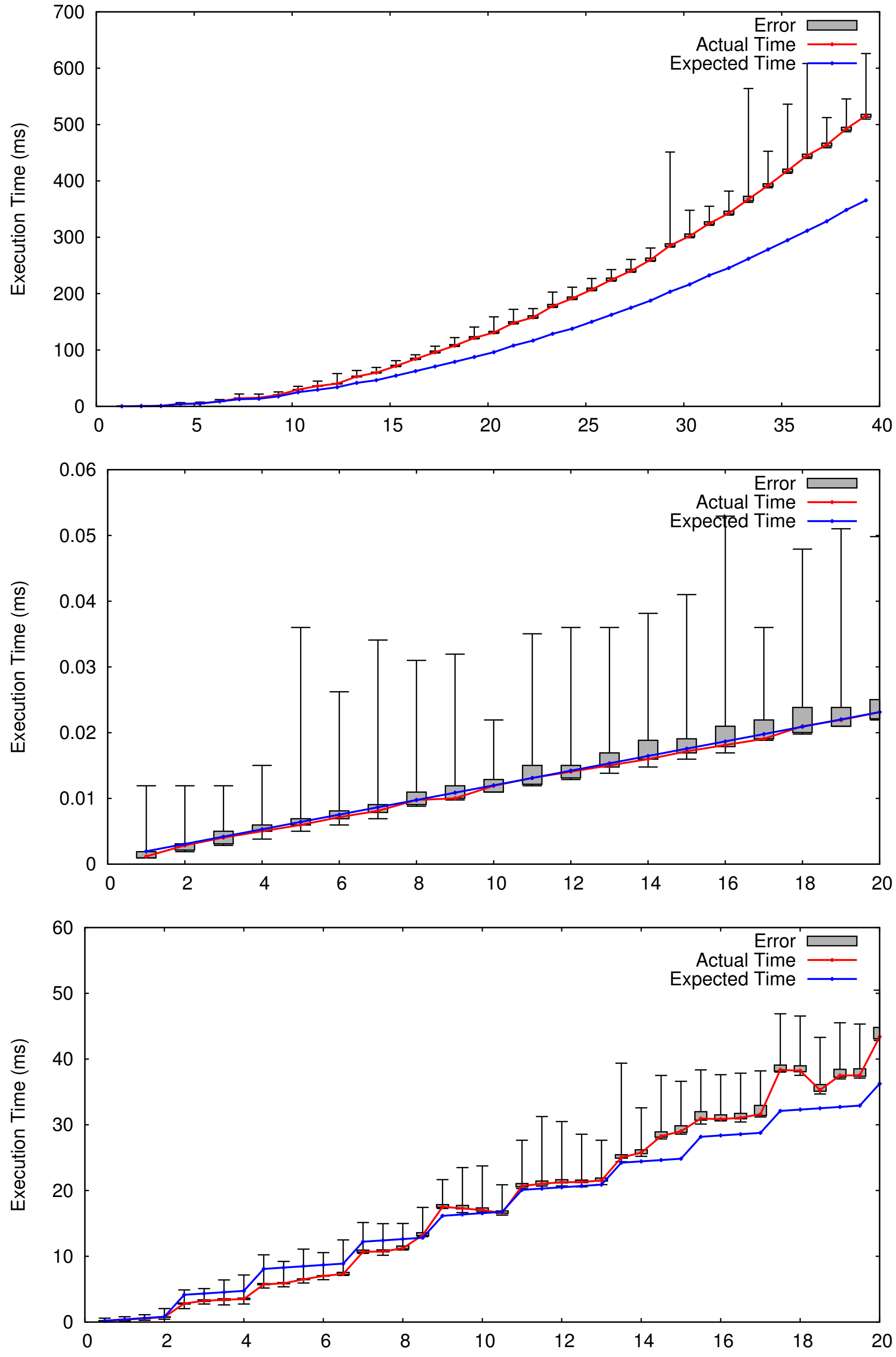

Fig. 5. Graph showing actual and expected time for isort (HOF) (input $\times 10^2$) (top), mult (input $\times 10^3$) (middle) and pack (input $\times 10^4$) (bottom)

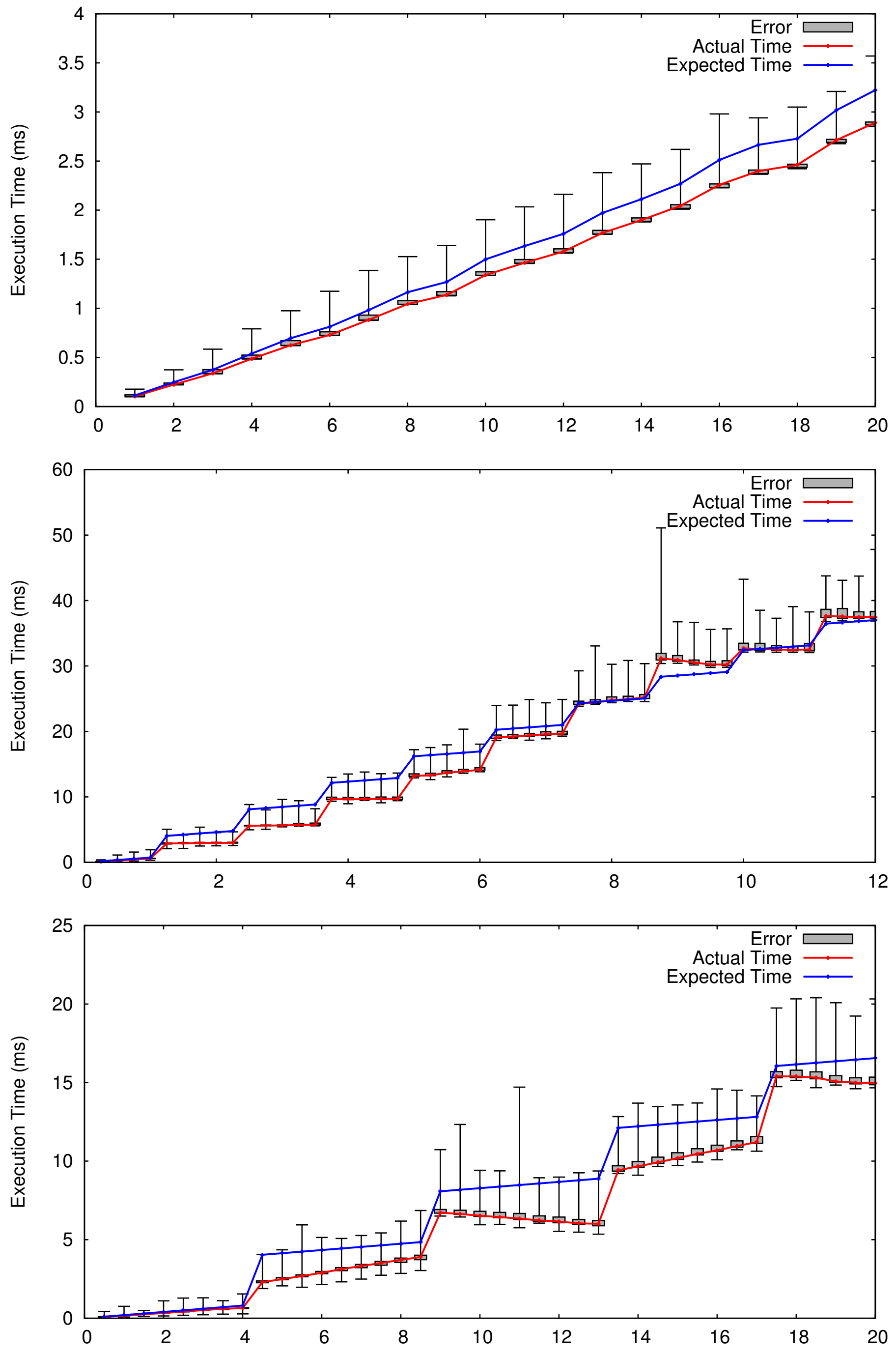

Fig. 6. Graph showing actual and expected time for phi (input $\times 10^3$) (top), replicate (input $\times 10^4$) (middle) and reverseTR (input $\times 10^4$) (bottom)

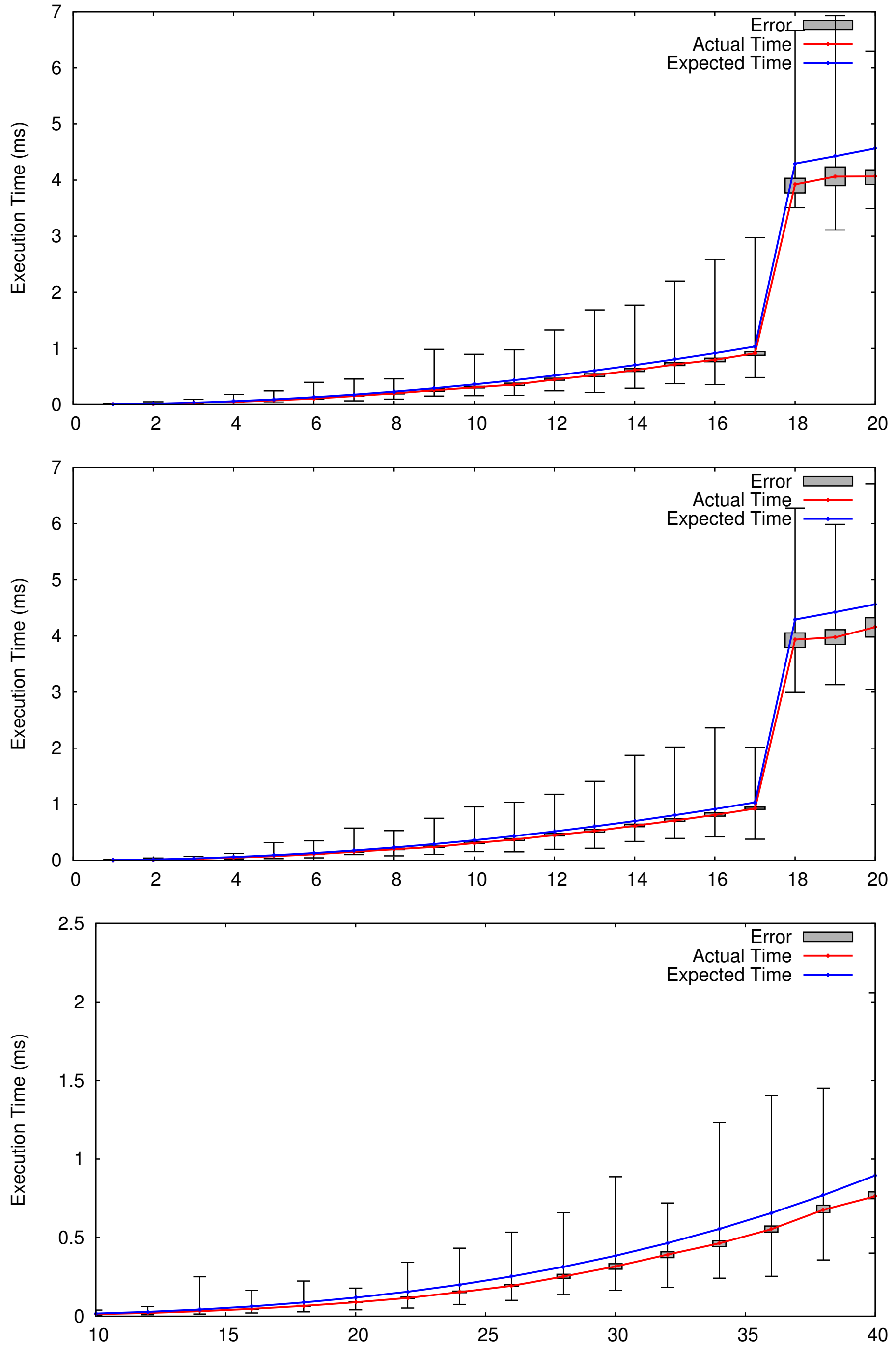

Fig. 7. Graph showing actual and expected time for matrix add (input ×10) (top), matrix sub (input ×10) (middle) and matrix mult (bottom)