Automatic Accurate Time-Bound Analysis for High-Level Languages

Yanhong A. Liu* and Gustavo Gomez*

Computer Science Department, Indiana University, Bloomington, IN 47405
{liu,ggomezes}@cs.indiana.edu

Abstract. This paper describes a general approach for automatic and accurate time-bound analysis. The approach consists of transformations for building time-bound functions in the presence of partially known input structures, symbolic evaluation of the time-bound function based on input parameters, optimizations to make the overall analysis efficient as well as accurate, and measurements of primitive parameters, all at the source-language level. We have implemented this approach and performed a number of experiments for analyzing Scheme programs. The measured worst-case times are closely bounded by the calculated bounds.

1 Introduction

Analysis of program running time is important for real-time systems, interactive environments, compiler optimizations, performance evaluation, and many other computer applications. It has been extensively studied in many fields of computer science: algorithms [11, 8], programming languages [25, 12, 20, 22], and systems [23, 19, 21]. It is particularly important for many applications, such as real-time systems, to be able to predict accurate time bounds automatically and efficiently, and it is particularly desirable to be able to do so for high-level languages [23, 19].

Since Shaw proposed timing schema for analyzing system running time based on high-level languages [23], a number of people have extended it for analysis in the presence of compiler optimizations [5], pipelining [13], cache memory [13, 7], etc. However, there remains an obvious and serious limitation of the timing schema, even in the absence of low-level complications. This is the inability to provide loop bounds, recursion depths, or execution paths automatically and accurately for the analysis [18, 1]. For example, the inaccurate loop bounds cause the calculated worst-case time to be as much as 67% higher than the measured worst-case time in [19], while the manual way of providing such information is potentially an even larger source of error, in addition to its inconvenience [18]. Various program analysis methods have been proposed to provide loop bounds or execution paths [1, 6, 9]. They ameliorate the problem but can not completely solve it, because they apply only to some classes of programs or use approximations that are too crude for the analysis, and because separating the loop and path information from the rest of the analysis is in general less accurate than an integrated analysis [17].

This paper describes a general approach for automatic and accurate time-bound analysis. The approach combines methods and techniques studied in theory, languages, and systems. We call it a *language-based* approach, because it primarily exploits methods and techniques for static program analysis and transformation.

* This work was partially supported by NSF under Grant CCR-9711253.

The approach consists of transformations for building time-bound functions in the presence of partially known input structures, symbolic evaluation of the time-bound function based on input parameters, optimizations to make overall the analysis efficient as well as accurate, and measurements of primitive parameters, all at the source-language level. We describe analysis and transformation algorithms and explain how they work. We have implemented this approach and performed a large number of experiments analyzing Scheme programs. The measured worst-case times are closely bounded by the calculated bounds. We describe our prototype system, ALPA, as well as the analysis and measurement results.

This approach is general in the sense that it works for other kinds of cost analysis as well, such as space analysis and output-size analysis. The basic ideas also apply to other programming languages. Furthermore, the implementation is independent of the underlying systems (compilers, operating systems, and hardware).

2 Language-based approach

Language-based time-bound analysis starts with a given program written in a high-level language, such as C or Lisp. The first step is to build a *timing function* that (takes the same input as the original program but) returns the running time in place of (or in addition to) the original return value. This is done easily by associating a parameter with each program construct representing its running time and by summing these parameters based on the semantics of the constructs [25, 23]. We call parameters that describe the running times of program constructs *primitive parameters*.

Since the goal is to calculate running time without being given particular inputs, the calculation must be based on certain assumptions about inputs. Thus, the first problem is to characterize the input data and reflect them in the timing function. In general, due to imperfect knowledge about the input, the timing function is transformed into a *time-bound function*.

In programming-language area, Rosendahl proposed the use of *partially known input structures* to characterize input data [20]. For example, instead of replacing an input list l with its length n, as done in algorithm analysis, or annotating loops with numbers related to n, as done in systems, we simply use as input a list of n unknown elements. We call parameters for describing partially known input structures *input parameters*. The timing function is then transformed automatically into a time-bound function: at control points where decisions depend on unknown values, the maximum time of all possible branches is computed; otherwise, the time of the chosen branch is computed. Rosendahl concentrated on proving the correctness of this transformation. He assumed constant 1 for primitive parameters and relied on optimizations to obtain closed forms in terms of input parameters, but closed forms can not be obtained for all time-bound functions.

Combining results from theory to systems, we have studied a general approach for computing time bounds automatically, efficiently, and more accurately. The approach analyzes programs written in a functional subset of scheme. Functional programming languages, together with features like automatic garbage collection, have become increasingly widely used, yet work for calculating actual running time of functional programs has been lacking. Analyses and transformations developed for functional language can be applied to improve analyses of imperative languages as well [26].

Language. We use a first-order, call-by-value functional language that has structured data, primitive arithmetic, Boolean, and comparison operations, conditionals, bindings, and mutually recursive function calls. For example, the program below selects the least element in a non-empty list.

$$\begin{aligned} least(x) = \ & \textbf{if } null(cdr(x)) \textbf{ then } car(x) \\ & \textbf{else } \textbf{let } s = least(cdr(x)) \\ & \quad \textbf{in } \textbf{if } car(x) \leq s \textbf{ then } car(x) \textbf{ else } s \textbf{ end} \end{aligned}$$

To present various analysis results, we use the following examples: insertion sort, selection sort (which uses *least*), mergesort, set union, list reversal (the standard linear-time version), and reversal with append (the standard quadratic-time version).

Even though this language is small, it is sufficiently powerful and convenient to write sophisticated programs. Structured data is essentially records in Pascal and C. We can also see that time analysis in the presence of arrays and pointers is not fundamentally harder [19], because the running times of the program constructs for them can be measured in the same way as times of other constructs. Note that side effects caused by these features often cause other analysis to be difficult [4]. For pure functional languages, higher-order functions and lazy evaluations are important. Time-bound functions that accommodate these features have been studied [24, 22]. The symbolic evaluation and optimizations we describe apply to them as well.

3 Constructing time-bound functions

Constructing timing functions. We first transform the original program to construct a timing function, which takes the original input and primitive parameters as arguments and returns the running time. This is straightforward based on the semantics of the program constructs.

Given an original program, we add a set of timing functions, one for each original function, which simply count the time while the original program executes. The algorithm, given below, is presented as a transformation $\mathcal{T}$ on the original program, which calls a transformation $\mathcal{T}_e$ to recursively transform subexpressions. For example, a variable reference is transformed into a symbol T_{varref} representing the running time of a variable reference; a conditional statement is transformed into the time of the test plus, if the condition is true, the time of the true branch, otherwise, the time of the false branch, and plus the time for the transfers of control.

$$\text{program: } \mathcal{T}\left[\!\!\left[\begin{array}{l} f_1(v_1, ..., v_{n_1}) = e_1; \\ ... \\ f_m(v_1, ..., v_{n_m}) = e_m; \end{array}\right]\!\!\right] = \begin{array}{ll} f_1(v_1, ..., v_{n_1}) = e_1; & tf_1(v_1, ..., v_{n_1}) = \mathcal{T}_e[\![e_1]\!]; \\ ... & ... \\ f_m(v_1, ..., v_{n_m}) = e_m; & tf_m(v_1, ..., v_{n_m}) = \mathcal{T}_e[\![e_m]\!]; \end{array}$$

$$\begin{array}{lll} \text{variable reference:} & \mathcal{T}_e[\![v]\!] & = T_{varref} \\ \text{data construction:} & \mathcal{T}_e[\![c(e_1, ..., e_n)]\!] & = add(T_c, \mathcal{T}_e[\![e_1]\!], ..., \mathcal{T}_e[\![e_n]\!]) \\ \text{primitive operation:} & \mathcal{T}_e[\![p(e_1, ..., e_n)]\!] & = add(T_p, \mathcal{T}_e[\![e_1]\!], ..., \mathcal{T}_e[\![e_n]\!]) \\ \text{conditional:} & \mathcal{T}_e[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] & = add(T_{if}, \mathcal{T}_e[\![e_1]\!], \textbf{if } e_1 \textbf{ then } \mathcal{T}_e[\![e_2]\!] \textbf{ else } \mathcal{T}_e[\![e_3]\!]) \\ \text{binding:} & \mathcal{T}_e[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!] & = add(T_{let}, \mathcal{T}_e[\![e_1]\!], \textbf{let } v = e_1 \textbf{ in } \mathcal{T}_e[\![e_2]\!] \textbf{ end}) \\ \text{function call:} & \mathcal{T}_e[\![f(e_1, ..., e_n)]\!] & = add(T_{call}, \mathcal{T}_e[\![e_1]\!], ..., \mathcal{T}_e[\![e_n]\!], tf(e_1, ..., e_n)) \end{array}$$

This transformation is similar to the local cost assignment [25], step-counting function [20], cost function [22], etc. in other work. Our transformation handles bindings and makes all primitive parameters explicit at the source-language level. For example, each primitive operation p is given a different symbol T_p, and each

constructor c is given a different symbol T_c. Note that the timing function terminates with the appropriate sum of primitive parameters if the original program terminates, and it runs forever to sum to infinity if the original program does not terminate, which is the desired meaning of a timing function.

Constructing time-bound functions. Characterizing program inputs and capturing them in the timing function are difficult to automate [25, 12, 23]. However, partially known input structures provide a natural means [20]. A special value *unknown* represents unknown values. For example, to capture all input lists of length n, the following partially known input structure can be used.

$$\begin{aligned} list(n) = \ & \textbf{if } n = 0 \textbf{ then } nil \\ & \textbf{else } cons(unknown, list(n-1)) \end{aligned}$$

Similar structures can be used to describe an array of n elements, etc.

Since partially known input structures give incomplete knowledge about inputs, the original functions need to be transformed to handle the special value *unknown*. In particular, for each primitive function p, we define a new function f_p such that $f_p(v_1, ..., v_n)$ returns *unknown* if any v_i is *unknown* and returns $p(v_1, ..., v_n)$ as usual otherwise. We also define a new function *lub* that takes two values and returns the most precise partially known structure that both values conform with.

$$\begin{aligned} f_p(v_1, ..., v_n) = \ & \textbf{if } v_1 = unknown \\ & \quad \vee \ ... \ \vee \\ & \quad v_n = unknown \\ & \textbf{then } unknown \\ & \textbf{else } p(v_1, ..., v_n) \end{aligned} \qquad \begin{aligned} lub(v_1, v_2) = \ & \textbf{if } v_1 \textit{ is } c_1(x_1, ..., x_i) \wedge \\ & \quad v_2 \textit{ is } c_2(y_1, ..., y_j) \wedge \\ & \quad c_1 = c_2 \ \wedge \ i = j \\ & \textbf{then } c_1(lub(x_1, y_1), ..., lub(x_i, y_i)) \\ & \textbf{else } unknown \end{aligned}$$

Also, the timing functions need to be transformed to compute an upper bound of the running time: if the truth value of a conditional test is known, then the time of the chosen branch is computed normally, otherwise, the maximum of the times of both branches is computed. Transformation $\mathcal{C}$ embodies these algorithms, where $\mathcal{C}_e$ transforms an expression in the original functions, and $\mathcal{C}_t$ transforms an expression in the timing functions.

$$\text{prog: } \mathcal{C}\left[\!\!\left[\begin{array}{ll} f_1(v_1, ..., v_{n_1}) = e_1; & tf_1(v_1, ..., v_{n_1}) = e'_1; \\ ... & ... \\ f_m(v_1, ..., v_{n_m}) = e_m; & tf_m(v_1, ..., v_{n_m}) = e'_m; \end{array} \right]\!\!\right]$$

$$= \begin{array}{l} f_1(v_1, ..., v_{n_1}) = \mathcal{C}_e[\![e_1]\!]; \quad tf_1(v_1, ..., v_{n_1}) = \mathcal{C}_t[\![e'_1]\!]; \quad f_p(v_1, ..., v_n) = ... \text{ as above} \\ ... \\ f_m(v_1, ..., v_{n_m}) = \mathcal{C}_e[\![e_m]\!]; tf_m(v_1, ..., v_{n_m}) = \mathcal{C}_t[\![e'_m]\!]; lub(v_1, v_2) = ... \text{ as above} \end{array}$$

$$\begin{array}{lll} \text{variable ref.:} & \mathcal{C}_e[\![v]\!] & = v \\ \text{data const.:} & \mathcal{C}_e[\![c(e_1, ..., e_n)]\!] & = c(\mathcal{C}_e[\![e_1]\!], ..., \mathcal{C}_e[\![e_n]\!]) \\ \text{primitive op.:} & \mathcal{C}_e[\![p(e_1, ..., e_n)]\!] & = f_p(\mathcal{C}_e[\![e_1]\!], ..., \mathcal{C}_e[\![e_n]\!]) \\ \text{conditional:} & \mathcal{C}_e[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] & = \textbf{let } v = \mathcal{C}_e[\![e_1]\!] \\ & & \quad \textbf{in if } v = unknown \textbf{ then } lub(\mathcal{C}_e[\![e_2]\!], \mathcal{C}_e[\![e_3]\!]) \\ & & \quad\quad \textbf{else if } v \textbf{ then } \mathcal{C}_e[\![e_2]\!] \textbf{ else } \mathcal{C}_e[\![e_3]\!] \textbf{ end} \\ \text{binding:} & \mathcal{C}_e[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!] & = \textbf{let } v = \mathcal{C}_e[\![e_1]\!] \textbf{ in } \mathcal{C}_e[\![e_2]\!] \textbf{ end} \\ \text{function call:} & \mathcal{C}_e[\![f(e_1, ..., e_n)]\!] & = f(\mathcal{C}_e[\![e_1]\!], ..., \mathcal{C}_e[\![e_n]\!]) \end{array}$$

$$\begin{array}{lll} \text{primitive parameter:} & \mathcal{C}_t[\![T]\!] & = T \\ \text{summation:} & \mathcal{C}_t[\![add(e_1, ..., e_n)]\!] & = add(\mathcal{C}_t[\![e_1]\!], ..., \mathcal{C}_t[\![e_n]\!]) \\ \text{conditional:} & \mathcal{C}_t[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] & = \textbf{let } v = \mathcal{C}_e[\![e_1]\!] \\ & & \quad \textbf{in if } v = unknown \textbf{ then } max(\mathcal{C}_t[\![e_2]\!], \mathcal{C}_t[\![e_3]\!]) \\ & & \quad\quad \textbf{else if } v \textbf{ then } \mathcal{C}_t[\![e_2]\!] \textbf{ else } \mathcal{C}_t[\![e_3]\!] \textbf{ end} \\ \text{binding:} & \mathcal{C}_t[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!] & = \textbf{let } v = \mathcal{C}_t[\![e_1]\!] \textbf{ in } \mathcal{C}_t[\![e_2]\!] \textbf{ end} \\ \text{function call:} & \mathcal{C}_t[\![tf(e_1, ..., e_n)]\!] & = tf(\mathcal{C}_t[\![e_1]\!], ..., \mathcal{C}_t[\![e_n]\!]) \end{array}$$

The resulting time-bound function takes as arguments partially known input structures, such as $list(n)$, which take as arguments input parameters, such as n. Therefore, we can obtain a resulting function that takes as arguments input parameters and primitive parameters and computes the most accurate time bound possible.

Both transformations $\mathcal{T}$ and $\mathcal{C}$ take linear time in terms of the size of the program, so they are extremely efficient, as also seen in our prototype system ALPA. Note that the resulting time-bound function may not terminate, but this occurs only if the recursive structure of the original program depends on unknown parts in the partially known input structure. As a trivial example, if partially known input structure given is *unknown*, then the corresponding time-bound function for any recursive function does not terminate.

4 Optimizing time-bound functions

This section describes symbolic evaluation and optimizations that make computation of time bounds more efficient. The transformations consist of partial evaluation, realized as global inlining, and incremental computation, realized as local optimization. In the worst case, evaluation of the time-bound functions takes exponential time in terms of the input parameters, since it essentially searches for the worst-case execution path for all inputs satisfying the partially known input structures.

Partial evaluation of time-bound functions. In practice, values of input parameters are given for almost all applications. While in general it is not possible to obtain explicit loop bounds automatically and accurately, we can implicitly achieve the desired effect by evaluating the time-bound function symbolically in terms of primitive parameters given specific values of input parameters.

The evaluation simply follows the structures of time-bound functions. Specifically, the control structures determine conditional branches and make recursive function calls as usual, and the only primitive operations are sums of primitive parameters and maximums among alternative sums, which can easily be done symbolically. Thus, the transformation simply inlines all function calls, sums all primitive parameters symbolically, determines conditional branches if it can, and takes maximum sums among all possible branches if it can not.

The symbolic evaluation $\mathcal{E}$ defined below performs the transformations. It takes as arguments an expression e and an environment ρ of variable bindings and returns as result a symbolic value that contains the primitive parameters. The evaluation starts with the application of the program to be analyzed to a partially unknown input structure, e.g., $mergesort(list(250))$, and it starts with an empty environment. Assume *symbAdd* is a function that symbolically sums its arguments, and $symbMax$ is a function that symbolically takes the maximum of its arguments.

variable ref.:	$\mathcal{E}[\![v]\!]\rho$	$= \rho(v)$ look up binding in environment
primitive parameter:	$\mathcal{E}[\![T]\!]\rho$	$= T$
data constr.:	$\mathcal{E}[\![c(e_1,...,e_n)]\!]\rho$	$= c(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho)$
primitive op.:	$\mathcal{E}[\![p(e_1,...,e_n)]\!]\rho$	$= p(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho)$
summation:	$\mathcal{E}[\![add(e_1,...,e_n)]\!]\rho$	$= symbAdd(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho)$
maximum:	$\mathcal{E}[\![max(e_1,...,e_n)]\!]\rho$	$= symbMax(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho)$
conditional:	$\mathcal{E}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\rho$	$= \mathcal{E}[\![e_2]\!]\rho$ if $\mathcal{E}[\![e_1]\!]\rho = true$ $\mathcal{E}[\![e_3]\!]\rho$ if $\mathcal{E}[\![e_1]\!]\rho = false$
binding:	$\mathcal{E}[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!]\rho$	$= \mathcal{E}[\![e_2]\!]\rho[v \mapsto \mathcal{E}[\![e_1]\!]\rho]$ bind v in environment
function call:	$\mathcal{E}[\![f(e_1,...,e_n)]\!]\rho$	$= e[v_1 \mapsto \mathcal{E}[\![e_1]\!]\rho,...,v_n \mapsto \mathcal{E}[\![e_n]\!]\rho]$ where f is defined by $f(v_1,...,v_n) = e$

This symbolic evaluation is exactly a specialized partial evaluation. It is fully automatic and computes the most accurate time bound possible with respect to the given program structure. It always terminates as long as the time-bound function terminates.

Avoiding repeated summations over recursions. The symbolic evaluation above is a global optimization over all time-bound functions involved. During the evaluation, summations of symbolic primitive parameters within each function definition are performed repeatedly while the computation recurses. Thus, we can speed up the symbolic evaluation by first performing such summations in a preprocessing step. Specifically, we create a vector and let each element correspond to a primitive parameter. The transformation $\mathcal{S}$ performs this optimization.

$$\text{program: } \mathcal{S}\left[\!\!\left[\begin{array}{l} tf_1(v_1,...,v_{n_1}) = e_1''; \\ ... \\ tf_m(v_1,...,v_{n_m}) = e_m''; \end{array}\right]\!\!\right] = \begin{array}{l} tf_1(v_1,...,v_{n_1}) = \mathcal{S}_t[\![e_1']\!]; \\ ... \\ tf_m(v_1,...,v_{n_m}) = \mathcal{S}_t[\![e_m']\!]; \end{array}$$

primitive parameter:	$\mathcal{S}_t[\![T]\!]$	= create a vector of 0's except with the component corresponding to T set to 1
summation:	$\mathcal{S}_t[\![add(e_1,...,e_n)]\!]$	= component-wise summation of all the vectors among $\mathcal{S}_t[\![e_1]\!],...,\mathcal{S}_t[\![e_n]\!]$
maximum:	$\mathcal{S}_t[\![max(e_1,...,e_n)]\!]$	= component-wise maximum of all the vectors among $\mathcal{S}_t[\![e_1]\!],...,\mathcal{S}_t[\![e_n]\!]$
all other:	$\mathcal{S}_t[\![e]\!]$	$= e$

This incrementalizes the computation in each recursion to avoid repeated summation. Again, this is fully automatic and takes time linear in terms of the size of the cost-bound function.

The result of this optimization is dramatic. For example, optimized symbolic evaluation of the same quadratic-time reverse takes only 2.55 milliseconds, while direct evaluation takes 0.96 milliseconds, resulting in less than 3 times slowdown.

5 Making time-bound functions accurate

While loops and recursions affect time bounds most, the accuracy of the time bounds calculated also depends on the handling of the conditionals in the original program, which is reflected in the time-bound function. For conditionals whose test results are known to be true or false at the symbolic-evaluation time, the appropriate branch is chosen; so other branches, which may even take longer, are not considered for the worst-case time. This is a major source of accuracy for our worst-case bound.

For conditionals whose test results are not known at symbolic-evaluation time, we need to take the maximum time among all alternatives. The only case in which this would produce inaccurate time bound is when the test in a conditional in one subcomputation implies the test in a conditional in another subcomputation. For example, consider an expression e_0 whose value is *unknown* and

$$e_1 = \textbf{if } e_0 \textbf{ then } 1 \textbf{ else } Fibonacci(1000)$$
$$e_2 = \textbf{if } e_0 \textbf{ then } Fibonacci(2000) \textbf{ else } 2$$

If we compute time bound for $e_1 + e_2$ directly, it is at least $tFibonacci(1000) + tFibonacci(2000)$. However, if we consider only the two realizable execution paths, we know that the worst case is $tFibonacci(2000)$ plus some small constants. This is known as the false-path elimination problem [1].

Two transformations, *lifting conditions* and *simplifying conditionals*, allow us to achieve the accurate analysis results above. In each function definition, the former lifts conditions to the outmost scope that the test does not depend on, and the latter simplifies conditionals according to the lifted condition. These transformations are not needed for the examples in this paper. They are discussed further in [14].

6 Implementation and experimentation

We have implemented the analysis approach in a prototype system, ALPA (Automatic Language-based Performance Analyzer). The implementation is for a subset of Scheme. The measurements and analyses are performed for source programs compiled with Chez Scheme compiler [3]. The particular numbers below are taken on a Sun Ultra 1 with 167MHz UltraSPARC CPU and 64MB main memory, but we have also performed the analysis for several other kinds of SPARC stations, and the results are similar.

We tried to avoid compiler optimizations by setting the optimization level to 0. To handle garbage-collection time, we performed two sets of experiments: one set excludes garbage-collection times in both calculations and measurements, while the other includes them in both.

Since the minimum running time of a program construct is about 0.1 microseconds, and the precision of the timing function is 10 milliseconds, we use control/test loops that iterate 10,000,000 times, keeping measurement error under 1%. Such a loop is repeated 100 times, and the average value is taken to compute the primitive parameter for the tested construct (the variance is less than 10% in most cases). The calculation of the time bound is done by plugging these measured parameters into the optimized time-bound function. We then run each example program an appropriate number of times to measure its running time with less than 1% error.

Figure 1 shows the calculated and measured worst-case times for six example programs on inputs of size 10 to 2000. For the set union example, we used inputs where both arguments were of the given sizes. These times do not include garbage-collection times. The item me/ca is the measured time expressed as a percentage of the calculated time. In general, all measured times are closely bounded by the calculated times (with about 90-95% accuracy) except when inputs are extremely small (10 or 20, in 1 case) or extremely large (2000, in 3 cases), which is analyzed below.

For measurements that include garbage-collection times, the results are similar, except that the percentages are consistently higher and underestimates occur for a few more inputs and start on inputs of size 1000 instead of 200. We believe that this is the effect of garbage collection, which we have not analyzed specifically.

Examples such as sorting are classified as complex examples in previous study [19, 13], where calculated time is as much as 67% higher than measured time, and where only the result for one sorting program on a single input (of size 10 [19] or 20 [13]) is reported in each experiment.

We found that when inputs are extremely small, the measured time is occasionally above the calculated time for some examples. Also, when inputs are large, the measured times for some examples are above the calculated time. We attribute these to cache memory effects, and this is further confirmed by measuring programs, such as Cartesian product, that use extremely large amount of space even on small inputs (50-200). While this shows that cache effects need to be considered for larger applications, it also helps validate that our calculated results are accurate relative to our current model.

	insertion sort			selection sort			mergesort		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.06751	0.06500	96.3	0.13517	0.12551	92.9	0.11584	0.11013	95.1
20	0.25653	0.25726	100.3	0.52945	0.47750	90.2	0.29186	0.27548	94.4
50	1.55379	1.48250	95.4	3.26815	3.01125	92.1	0.92702	0.85700	92.4
100	6.14990	5.86500	95.4	13.0187	11.9650	91.9	2.15224	1.98812	92.4
200	24.4696	24.3187	99.4	51.9678	47.4750	91.4	4.90017	4.57200	93.3
300	54.9503	53.8714	98.0	116.847	107.250	91.8	7.86231	7.55600	96.1
500	152.448	147.562	96.8	324.398	304.250	93.8	14.1198	12.9800	91.9
1000	609.146	606.000	99.5	1297.06	1177.50	90.8	31.2153	28.5781	91.6
2000	2435.29	3081.25	126.5	5187.17	5482.75	105.7	68.3816	65.3750	95.6

	set union			list reversal			reversal w/app.		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.10302	0.09812	95.2	0.00918	0.00908	98.8	0.05232	0.04779	91.3
20	0.38196	0.36156	94.7	0.01798	0.01661	92.4	0.19240	0.17250	89.7
50	2.27555	2.11500	92.9	0.04436	0.04193	94.5	1.14035	1.01050	88.6
100	8.95400	8.33250	93.1	0.08834	0.08106	91.8	4.47924	3.93800	87.9
200	35.5201	33.4330	94.1	0.17629	0.16368	92.9	17.7531	15.8458	89.3
300	79.6987	75.1000	94.2	0.26424	0.24437	92.5	39.8220	35.6328	89.5
500	220.892	208.305	94.3	0.44013	0.40720	92.5	110.344	102.775	93.1
1000	882.094	839.780	95.2	0.87988	0.82280	93.5	440.561	399.700	90.7
2000	3525.42	3385.31	96.0	1.75937	1.65700	94.2	1760.61	2235.75	127.0

Fig. 1. Calculated and measured worst-case times (in milliseconds), without garbage collection.

Among fifteen programs we analyzed using ALPA, two of the time-bound functions did not terminate. One is quicksort, and the other is a contrived variation of sorting; both diverge because the recursive structure for splitting a list depends on the values of unknown list elements. We have found a different symbolic-evaluation strategy that uses a kind of incremental path selection, and the evaluation would terminate for both examples, as well as all other examples, giving accurate worst-case bounds. We are implementing that algorithm. We also noticed that static analysis can be exploited to identify sources of nontermination.

7 Related work and conclusion

Compared to work in algorithm analysis and program complexity analysis [12, 22], this work consistently pushes through symbolic primitive parameters, so it allows us to calculate actual time bounds and validate the results with experimental measurements. Compared to work in systems [23, 19, 18, 13], this work explores program analysis and transformation techniques to overcome the difficulties caused by the inability to obtain loop bounds, recursion depths, or execution paths automatically and precisely. There is also work for measuring primitive parameters of Fortran programs for the purpose of general performance prediction [21], not worst-case analysis.

A number of techniques have been studied for obtaining loop bounds or execution paths [18, 1, 6, 9]. Manual annotations [18, 13] are inconvenient and error-prone [1]. Automatic analysis of such information has two main problems. First, separating the loop and path information from the rest of the analysis [6] is in general less accurate than an integrated analysis [17]. Second, approximations for merging paths from loops, or recursions, very often lead to nontermination of the time analysis, not just looser bounds [6, 17]. Some new methods, while powerful, apply only to certain classes of programs [9]. In contrast, our method allows recursions, or loops, to be considered naturally in the overall execution-time analysis based on partially known input structures.

The most recent work by Lundqvist and Stenstrom [17] is based on essentially the same ideas as ours. They apply the ideas at machine instruction level

and can more accurately take into account the effects of instruction pipelining and data caching, but their method for merging paths for loops would lead to nonterminating analysis for many programs, e.g., a program that computes the union of two lists with no repeated elements. Our experiments show that we can calculate more accurate time bound and for many more programs than merging paths, and the calculation is still efficient.

The idea of using partially known input structures originates from Rosendahl [20]. We have extended it to manipulate primitive parameters. We also handle binding constructs, which is simple but necessary for efficient computation. An innovation in our method is to optimize the time-bound function using partial evaluation [2, 10], incremental computation [16, 15], and transformations of conditionals to make the analysis more efficient and more accurate.

We are starting to explore a suite of new language-based techniques for timing analysis, in particular, analyses and optimizations for further speeding up the evaluation of the time-bound function. To make the analysis even more accurate and efficient, we can automatically generate measurement programs for all maximum subexpressions that do not include transfers of control; this corresponds to the large atomic-blocks method [19]. We also believe that the lower-bound analysis is entirely symmetric to the upper-bound analysis, by replacing maximum with minimum at all conditional points. Finally, we plan to accommodate more lower-level dynamic factors for timing at the source-language level [13, 7]. In particular, we plan to apply our general approach to analyze space consumption and hence to help predict garbage-collection and caching behavior.

References

1. P. Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the 8th EuroMicro Workshop on Real-Time Systems*, pages 102–107, L'Aquila, June 1996.
2. B. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, Amsterdam, 1988.
3. Cadence Research Systems. *Chez Scheme System Manual*. Cadence Research Systems, Bloomington, Indiana, revision 2.4 edition, July 1994.
4. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310. ACM, New York, June 1990.
5. J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of the 10th EuroMicro Workshop on Real-Time Systems*, Berlin, Germany, June 1998.
6. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *In Proceedings of Euro-Par'97*, volume 1300 of *Lecture Notes in Computer Science*, pages 1298–1307. Springer-Verlag, Berlin, Aug. 1997.
7. C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.
8. P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science, Series A*, 79(1):37–109, Feb. 1991.
9. C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings of the IEEE Real-Time Applications Symposium*. IEEE CS Press, Los Alamitos, Calif., June 1998.
10. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, N.J., 1993.
11. D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Mass., 1968.
12. D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, Apr. 1988.

13. S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, July 1995.
14. Y. A. Liu and G. Gomezes. Automatic accurate time-bound analysis for high-level languages. Technical Report TR 508, Computer Science Department, Indiana University, Bloomington, Indiana, Apr. 1998.
15. Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3), May 1998.
16. Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.
17. T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. Technical Report No. 98-3, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1998.
18. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.
19. C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.
20. M. Rosendahl. Automatic complexity analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, Sept. 1989.
21. R. H. Saavedra and A. J. Smith. Analysis of benchmark characterization and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, Nov. 1996.
22. D. Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 361–376. Springer-Verlag, Berlin, May 1990.
23. A. Shaw. Reasoning about time in higher level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, July 1989.
24. P. Wadler. Strictness analysis aids time analysis. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 1988.
25. B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–538, Sept. 1975.
26. D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 1994.