Automatic Accurate Time-Bound Analysis for High-Level Languages

Yanhong A. Liu* and Gustavo Gomez*

April 1998

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 [20, 12, 13, 41], programming languages [38, 21, 30, 33], and systems [35, 28, 32, 31]. 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 [35, 28].

Since Shaw proposed timing schema for analyzing system running time based on high-level languages [35], a number of people have extended it for analysis in the presence of compiler optimizations [28, 9], pipelining [16, 22], cache memory [3, 22, 11], 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 [27, 2]. 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 [28], while the manual way of providing such information is potentially an even larger source of error, in addition to its inconvenience [27]. Various program analysis methods have been proposed to provide loop bounds or execution paths [2, 10, 15, 17]. 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 [25].

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.

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,

*This work was partially supported by NSF under Grant CCR-9711253. Authors' address: Computer Science Department, 215 Lindley Hall, Indiana University, Bloomington, IN 47405. Phone: (812)855-4373. Email: {liu,ggomezes}@cs.indiana.edu.

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).

The rest of the paper is organized as follows. Section 2 outlines our language-based approach. Sections 3, 4, and 5 present the analysis and transformation methods and techniques. Section 6 describes our implementation and experimental results. Section 7 compares with related work and concludes.

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 [38, 7, 35]. We call parameters that describe the running times of program constructs *primitive parameters*. To calculate actual time bounds based on the timing function, three difficult problems must be solved.

First, 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 algorithm analysis, inputs are characterized by their size; accommodating this requires manual or semi-automatic transformation of the timing function [38, 21, 41]. The analysis is mainly asymptotic, and primitive parameters are considered independent of input size, i.e., are constants while the computation iterates or recurses. Whatever values of the primitive parameters are assumed, a second problem arises, and it is theoretically challenging: optimizing the time-bound function to a closed form in terms of the input size [38, 7, 21, 30, 13]. Although much progress has been made in this area, closed forms are known only for subclasses of functions. Thus, such optimization can not be automatically done for analyzing general programs.

In systems, inputs are characterized indirectly using loop bounds or execution paths in programs, and such information must in general be provided by the user [35, 28, 27, 22], even though program analyses can help in some cases [2, 10, 15, 17]. Closed forms in terms of parameters for these bounds can be obtained easily from the timing function. This isolates the third problem, which is most interesting to systems research: obtaining values of primitive parameters for various compilers, run-time systems, operating systems, and machine hardwares. In recent year, much progress has been made in analyzing low-level dynamic factors, such as clock interrupt, memory refresh, cache usage, instruction scheduling, and parallel architectures [28, 3, 22, 11]. Nevertheless, inability to compute loop bounds or execution paths automatically and accurately has led calculated bounds to be much higher than measured worst-case time. Additionally, primitive parameters are difficult and expensive to obtain and can not be reused for different systems and machines.

In programming-language area, Rosendahl proposed using *partially known input structures* [30]. 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 again closed forms can not be obtained for all time-bound functions.

Combining results from theory to systems, and exploring methods and techniques for static program analysis and transformation, we have studied a general approach for computing time bounds automatically, efficiently, and more accurately. The approach has four main components.

First, we use an automatic transformation to construct a time-bound function from the original program based on partially known input structures. The resulting function takes input parameters and primitive parameters as arguments. The only caveat here is that the time-bound function may not terminate. However, nontermination occurs only if the recursive/iterative structure of the original program depends on unknown parts in the given partially known input structures.

Then, to compute worst-case time bounds efficiently without relying on closed forms, we optimize the time-bound function symbolically with respect to given values of input parameters. This is based on partial evaluation and incremental computation. This symbolic evaluation always terminates provided the time-bound function terminates. The resulting function can be used repeatedly to compute time bounds efficiently for different primitive parameters measured for different underlying systems.

A third component consists of transformations that enable more accurate time bounds to be computed: lifting conditions, simplifying conditionals, and inlining nonrecursive functions. The transformations should be applied on the original program before the time-bound function is constructed. They may result in larger code size, but they allow subcomputations based on the same control conditions to be merged, leading to more accurate time bounds, which can be computed more efficiently as well.

Finally, we use control loops to measure primitive parameters at the source-language level and use them in computing the time bound. We have implemented these transformations and the measurement procedures for a subset of Scheme. All the transformations and measurements are done automatically, and the time bound is computed efficiently and accurately. Examples analyzed include a number of classical sorting programs, matrix computation programs, and various list processing programs.

The approach is general because all four components we developed are based on general methods and techniques. Each particular component is not meant to be a new analysis or transformation, but the combination of them for the application of automatic and accurate time-bound analysis for high-level languages is new. We used a functional subset of Scheme [1, 8] for three reasons.

1) 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.

2) Much work has been done on analyzing and transforming functional programs, including complexity analysis, and it can be used for analyzing actual running times efficiently and accurately as well.

3) Analyses and transformations developed for functional language can be applied to improve analyses of imperative languages as well [40].

All our analyses and transformations are performed at source level. This allows implementations to be independent of compilers and underlying systems and allows analysis results to be understood at source level.

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. A program is a set of mutually recursive function definitions of the form

$$f(v_1, ..., v_n) = e$$

where expression e is given by the grammar below:[1]

$e ::= v$		variable reference
	$c(e_1, ..., e_n)$	data construction
	$p(e_1, ..., e_n)$	primitive operation
	if e_1 **then** e_2 **else** e_3	conditional expression
	let $v = e_1$ **in** e_2 **end**	binding expression
	$f(e_1, ..., e_n)$	function application

For binary primitive operations, we will be changing between infix and prefix notations depending on whichever is easier for the presentation. For example, the program below selects the least element in a non-empty list.

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

We use *least* as a small running example. To present various analysis results, we also use several other examples: insertion sort, selection sort, 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, structs in C, and constructor applications in ML. Conditionals and bindings easily simulate conditional statements and assignments, and recursions can simulate loops. We can also see that time analysis in the presence of arrays and pointers is not fundamentally harder [28], because the running times of the program constructs for them can be measured in the same way as times of other constructs. For example, accessing an array element $a[i]$ takes the time of accessing i, offsetting the element address from that of a, and finally getting the value from that address. Note that side effects caused by these features often cause other analysis to be difficult [6, 18]. For pure functional languages, higher-order functions and lazy evaluations are important. Time-bound functions that accommodate these features have been studied [37, 33]. 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

[1] The keywords are taken from ML [26]. Our implementation supports both this syntax and Scheme syntax.

of control.

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

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

Applying this transformation to the program *least*, we obtain function *least* as originally given and timing function *tleast*, where infix notation is used for additions, and unnecessary parentheses are omitted:

$$\begin{array}{rl} tleast(x) = & T_{if} + T_{null} + T_{cdr} + T_{varref} \\ & +(\textbf{if } null?(cdr(x)) \textbf{ then } T_{car} + T_{varref} \\ & \quad \textbf{else } T_{let} + T_{call} + T_{cdr} + T_{varref} + tleast(cdr(x)) \\ & \quad\quad +(\textbf{let } s = least(cdr(x)) \\ & \quad\quad\quad \textbf{in } T_{if} + T_{\leq} + T_{car} + T_{varref} + T_{varref} \\ & \quad\quad\quad\quad +(\textbf{if } car(x) \leq s \textbf{ then } T_{car} + T_{varref} \textbf{ else } T_{varref}) \textbf{ end})) \end{array}$$

This transformation is similar to the local cost assignment [38], step-counting function [30], cost function [33], etc. in other work. Our transformation extends those methods with 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 [38, 21, 35]. However, partially known input structures provide a natural means [30]. 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.

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

Similar structures can be used to describe an array of n elements, a matrix of m-by-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.

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

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.

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

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

variable reference:	$\mathcal{C}_e\,[\![v]\!]$	$= v$
data construction:	$\mathcal{C}_e\,[\![c(e_1,...,e_n)]\!]$	$= c(\mathcal{C}_e\,[\![e_1]\!],...,\mathcal{C}_e\,[\![e_n]\!])$
primitive operation:	$\mathcal{C}_e\,[\![p(e_1,...,e_n)]\!]$	$= f_p(\mathcal{C}_e\,[\![e_1]\!],...,\mathcal{C}_e\,[\![e_n]\!])$
conditional:	$\mathcal{C}_e\,[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]$	$= \textbf{let } v = \mathcal{C}_e\,[\![e_1]\!]$ $\textbf{in if } v = \mathit{unknown} \textbf{ then } lub(\mathcal{C}_e\,[\![e_2]\!],\mathcal{C}_e\,[\![e_3]\!])$ $\textbf{else if } v \textbf{ then } \mathcal{C}_e\,[\![e_2]\!] \textbf{ else } \mathcal{C}_e\,[\![e_3]\!] \textbf{ end}$
binding:	$\mathcal{C}_e\,[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!]$	$= \textbf{let } v = \mathcal{C}_e\,[\![e_1]\!] \textbf{ in } \mathcal{C}_e\,[\![e_2]\!] \textbf{ end}$
function call:	$\mathcal{C}_e\,[\![f(e_1,...,e_n)]\!]$	$= f(\mathcal{C}_e\,[\![e_1]\!],...,\mathcal{C}_e\,[\![e_n]\!])$
primitive parameter:	$\mathcal{C}_t\,[\![T]\!]$	$= T$
summation:	$\mathcal{C}_t\,[\![add(e_1,...,e_n)]\!]$	$= add(\mathcal{C}_t\,[\![e_1]\!],...,\mathcal{C}_t\,[\![e_n]\!])$
conditional:	$\mathcal{C}_t\,[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]$	$= \textbf{let } v = \mathcal{C}_e\,[\![e_1]\!]$ $\textbf{in if } v = \mathit{unknown} \textbf{ then } max(\mathcal{C}_t\,[\![e_2]\!],\mathcal{C}_t\,[\![e_3]\!])$ $\textbf{else if } v \textbf{ then } \mathcal{C}_t\,[\![e_2]\!] \textbf{ else } \mathcal{C}_t\,[\![e_3]\!] \textbf{ end}$
binding:	$\mathcal{C}_t\,[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!]$	$= \textbf{let } v = \mathcal{C}_t\,[\![e_1]\!] \textbf{ in } \mathcal{C}_t\,[\![e_2]\!] \textbf{ end}$
function call:	$\mathcal{C}_t\,[\![tf(e_1,...,e_n)]\!]$	$= tf(\mathcal{C}_t\,[\![e_1]\!],...,\mathcal{C}_t\,[\![e_n]\!])$

Applying the transformation on functions *least* and *tleast* yields the functions below, where function f_p for each primitive operator p and function *lub* are as given above. Note that various T's are indeed arguments to the timing function *tleast*; we omit them from argument positions for ease of reading.

$$\begin{array}{l} least(x) = \textbf{let } v = f_{null}(f_{cdr}(x)) \\ \quad \textbf{in if } v = \mathit{unknown} \textbf{ then} \\ \quad\quad lub(\, f_{car}(x), \\ \quad\quad\quad \textbf{let } s = least(f_{cdr}(x)) \\ \quad\quad\quad \textbf{in let } v = f_{\leq}(f_{car}(x), s) \\ \quad\quad\quad\quad \textbf{in if } v = \mathit{unknown} \textbf{ then } lub(f_{car}(x), s) \\ \quad\quad\quad\quad\quad \textbf{else if } v \textbf{ then } f_{car}(x) \textbf{ else } s \textbf{ end end}) \\ \quad\quad \textbf{else if } v \textbf{ then } f_{car}(x) \\ \quad\quad \textbf{else let } s = least(f_{cdr}(x)) \\ \quad\quad\quad \textbf{in let } v = f_{\leq}(f_{car}(x), s) \\ \quad\quad\quad\quad \textbf{in if } v = \mathit{unknown} \textbf{ then } lub(f_{car}(x), s) \\ \quad\quad\quad\quad\quad \textbf{else if } v \textbf{ then } f_{car}(x) \textbf{ else } s \textbf{ end end end} \end{array}$$

$$
\begin{aligned}
tleast(x) = {} & T_{if} + T_{null} + T_{cdr} + T_{varref} \\
& +(\textbf{let } v = f_{null}(f_{cdr}(x)) \\
& \quad \textbf{in if } v = unknown \textbf{ then} \\
& \qquad max(T_{car} + T_{varref}, \\
& \qquad\quad T_{let} + T_{call} + T_{cdr} + T_{varref} + tleast(cdr(x)) \\
& \qquad\quad +(\textbf{let } s = least(f_{cdr}(x)) \\
& \qquad\qquad \textbf{in } T_{if} + T_{\leq} + T_{car} + T_{varref} + T_{varref} \\
& \qquad\qquad\quad +(\textbf{let } v = f_{\leq}(f_{car}(x), s) \\
& \qquad\qquad\qquad \textbf{in if } v = unknown \textbf{ then } max(T_{car} + T_{varref}, T_{varref}) \\
& \qquad\qquad\qquad\quad \textbf{else if } v \textbf{ then } T_{car} + T_{varref} \\
& \qquad\qquad\qquad\quad \textbf{else } T_{varref} \textbf{ end}) \textbf{ end})) \\
& \quad \textbf{else if } v \textbf{ then } T_{car} + T_{varref} \\
& \quad \textbf{else } T_{let} + T_{call} + T_{cdr} + T_{varref} + tleast(cdr(x)) \\
& \qquad +(\textbf{let } s = least(f_{cdr}(x)) \\
& \qquad\quad \textbf{in } T_{if} + T_{\leq} + T_{car} + T_{varref} + T_{varref} \\
& \qquad\qquad +(\textbf{let } v = f_{\leq}(f_{car}(x), s) \\
& \qquad\qquad\quad \textbf{in if } v = unknown \textbf{ then } max(T_{car} + T_{varref}, T_{varref}) \\
& \qquad\qquad\qquad \textbf{else if } v \textbf{ then } T_{car} + T_{varref} \\
& \qquad\qquad\qquad \textbf{else } T_{varref} \textbf{ end}) \textbf{ end}) \textbf{ end})
\end{aligned}
$$

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, since the original program does take infinite time in the worst case.

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.

We first point out that time-bound functions may be extremely inefficient to evaluate given values for their parameters. In fact, in the worst case, the evaluation 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. This is why time-analysis techniques used in systems can require loop bounds from the user before time bounds are computed. 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 reference:	$\mathcal{E}[\![v]\!]\rho$	$= \rho(v)$ look up binding in environment
primitive parameter:	$\mathcal{E}[\![T]\!]\rho$	$= T$
data construction:	$\mathcal{E}[\![c(e_1,...,e_n)]\!]\rho$	$= c(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho)$
primitive operation:	$\mathcal{E}[\![p(e_1,...,e_n)]\!]\rho$	$= p(\mathcal{E}[\![e_1]\!]\rho,...,\mathcal{E}[\![e_n]\!]\rho)$
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 calls:	$\mathcal{E}[\![f(e_1,...,e_n)]\!]\rho$	$= e[v_1 \mapsto \mathcal{E}[\![e_1]\!]\rho,...,v_n \mapsto \mathcal{E}[\![e_n]\!]\rho]$ where f is defined by $f(v_1,...,v_n) = e$

As an example, applying symbolic evaluation to $tleast$ on a list of size 100, we obtain the following result:

$$\begin{aligned} tleast(list(100)) = \; & 497 * T_{varref} + 100 * T_{null} + 199 * T_{car} + 199 * T_{cdr} \\ & + 99 * T_{\leq} + 199 * T_{if} + 99 * T_{let} + 99 * T_{call} \end{aligned}$$

Figure 1 gives the results of symbolic evaluation of the timing functions for other example programs on inputs of various sizes. The last column lists the sums for every rows. All numbers are exact symbolic counts. They are verified by using a modified evaluator.

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.

The symbolic evaluation given only values of input parameters is inefficient compared to direct evaluation given values of both input parameters and particular primitive parameters, but the resulting function takes virtually constant time given any values of primitive parameters. For example, directly evaluating a quadratic-time reverse function (that uses append) on input of size 20 takes about 0.96 milliseconds, whereas the symbolic evaluation takes 670 milliseconds, hundreds of times slower. However, the resulting function can be evaluated in virtually no time given values of primitive parameters measured for any underlying systems. We propose further optimizations below that greatly speed up the symbolic evaluation.

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 $\mathcal{S}$

example	size	varref	nil	cons	null	car	cdr	$\leq$	if	let	call	total
insertion	10	321	11	55	66	100	55	45	111	0	65	829
sort	20	1241	21	210	231	400	210	190	421	0	230	3154
	50	7601	51	1275	1326	2500	1275	1225	2551	0	1325	19129
	100	30201	101	5050	5151	10000	5050	4950	10101	0	5150	75754
	200	120401	201	20100	20301	40000	20100	19900	40201	0	20300	301504
	300	270601	301	45150	45451	90000	45150	44850	90301	0	45450	677254
	500	751001	501	125250	125751	250000	125250	124750	250501	0	125750	1878754
	1000	3002001	1001	500500	501501	1000000	500500	499500	1001001	0	501500	7507504
	2000	12004001	2001	2001000	2003001	4000000	2001000	1999000	4002001	0	2003000	30015004
selection	10	576	11	55	121	190	200	90	211	55	120	1629
sort	20	2251	21	210	441	780	800	380	821	210	440	6354
	50	13876	51	1275	2601	4950	5000	2450	5051	1275	2600	39129
	100	55251	101	5050	10201	19900	20000	9900	20101	5050	10200	155754
	200	220501	201	20100	40401	79800	80000	39800	80201	20100	40400	621504
	300	495751	301	45150	90601	179700	180000	89700	180301	45150	90600	1397254
	500	1376251	501	125250	251001	499500	500000	249500	500501	125250	251000	3878754
	1000	5502501	1001	500500	1002001	1999000	2000000	999000	2001001	500500	1002000	15507504
	2000	22005001	2001	2001000	4004001	7998000	8000000	3998000	8002001	2001000	4004000	62015004
merge-	10	456	28	69	192	119	112	25	217	0	138	1356
sort	20	1154	58	177	468	315	284	69	537	0	340	3402
	50	3680	148	573	1440	1047	908	237	1677	0	1054	10764
	100	8562	298	1345	3284	2491	2116	573	3857	0	2412	24938
	200	19526	598	3089	7372	5779	4832	1345	8717	0	5428	56686
	300	31354	898	4977	11748	9355	7764	2189	13937	0	8660	90882
	500	56354	1498	8977	20948	16955	13964	3989	24937	0	15460	163082
	1000	124710	2998	19953	45900	37907	30928	8977	54877	0	33924	360174
	2000	273422	5998	43905	99804	83811	67856	19953	119757	0	73852	788358
set	10	582	10	10	121	120	110	100	231	10	120	1414
union	20	2162	20	20	441	440	420	400	861	20	440	5224
	50	12902	50	50	2601	2600	2550	2500	5151	50	2600	31054
	100	50802	100	100	10201	10200	10100	10000	20301	100	10200	122104
	200	201602	200	200	40401	40400	40200	40000	80601	200	40400	484204
	300	452402	300	300	90601	90600	90300	90000	180901	300	90600	1086304
	500	1254002	500	500	251001	251000	250500	250000	501501	500	251000	3010504
	1000	5008002	1000	1000	1002001	1002000	1001000	1000000	2003001	1000	1002000	12021004
	2000	20016002	2000	2000	4004001	4004000	4002000	4000000	8006001	2000	4004000	48042004
list	10	43	1	10	11	10	10	0	11	0	11	107
reversal	20	83	1	20	21	20	20	0	21	0	21	207
	50	203	1	50	51	50	50	0	51	0	51	507
	100	403	1	100	101	100	100	0	101	0	101	1007
	200	803	1	200	201	200	200	0	201	0	201	2007
	300	1203	1	300	301	300	300	0	301	0	301	3007
	500	2003	1	500	501	500	500	0	501	0	501	5007
	1000	4003	1	1000	1001	1000	1000	0	1001	0	1001	10007
	2000	8003	1	2000	2001	2000	2000	0	2001	0	2001	20007
reversal	10	231	11	55	66	55	55	0	66	0	65	604
with app	20	861	21	210	231	210	210	0	231	0	230	2204
	50	5151	51	1275	1326	1275	1275	0	1326	0	1325	13004
	100	20301	101	5050	5151	5050	5050	0	5151	0	5150	51004
	200	80601	201	20100	20301	20100	20100	0	20301	0	20300	202004
	300	180901	301	45150	45451	45150	45150	0	45451	0	45450	453004
	500	501501	501	125250	125751	125250	125250	0	125751	0	125750	1255004
	1000	2003001	1001	500500	501501	500500	500500	0	501501	0	501500	5010004
	2000	8006001	2001	2001000	2003001	2001000	2001000	0	2003001	0	2003000	20020004

Figure 1: Results of symbolic evaluation of time-bound functions (exact counts).

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 slow-down. Figure 2 compares the times of direct evaluation of timing functions, with each primitive parameter set to 1, and the times of optimized symbolic evaluation, obtaining the exact symbolic counts as in Figure 1. These measurements are taken on a Sun Ultra 1 with 167MHz CPU and 64MB memory. They include garbage-collection time. The times without garbage-collection times are all about 1% faster, so they are not shown here.

For mergesort, it takes several days for inputs of size 50 or larger. A special but simple optimization can be done, and resulting symbolic evaluation takes only seconds; such optimizations will be implemented in our system. For all other examples, it takes at most 2.7 hours. Note that, on small inputs, symbolic evaluation takes relatively much more time than direct evaluation, due to the relatively large overhead of vector setup; as inputs get larger, symbolic evaluation is almost as fast as direct evaluation for most examples. Again, after the symbolic evaluation, time bounds can be computed in virtually no time given primitive parameters measured on any machines.

	insertion sort		selection sort		mergesort		set union		list reversal		reversal w/app.	
size	direct	symbolic	direct	symbolic	direct	symbolic	direct	symbolic	direct	symbolic	direct	symbolic
10	0.49328	1.89057	0.71550	3.04985	1.43136	14.6666	1.44601	4.28571	0.01136	0.13916	0.25637	1.32877
20	1.93942	4.79452	3.89051	14.2352	605.714	8500.00	5.02935	10.6274	0.02113	0.26492	0.96215	2.55132
50	56.6666	87.4193	46.6666	106.451	xxxxxx	xxxxxx	134.516	192.666	0.04989	0.64224	23.2283	44.1269
100	451.428	557.142	338.571	571.428	xxxxxx	xxxxxx	1026.66	1176.66	0.09735	1.26038	178.000	231.333
500	58240.0	58080.0	39480.0	46050.0	xxxxxx	xxxxxx	125910.	117240.	0.50305	6.24266	21540.0	22180.0
2000	4024730	4039860	2666290	2761410	xxxxxx	xxxxxx	9205680	9690370	3.60703	27.4015	1810280	1711650

Figure 2: Times of direct evaluation vs. optimized symbolic evaluation (in milliseconds).

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 the time bound for $e_1 + e_2$ directly, the result 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 [2].

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. For $e_1 + e_2$ in the above example, lifting the condition for e_1, we obtain

$$\textbf{if } e_0 \textbf{ then } 1 + e_2 \textbf{ else } Fibonacci(1000) + e_2.$$

Simplifying the conditionals in the two occurrences of e_2 to $Fibonaccis(2000)$ and 2, respectively, we obtain

$$\textbf{if } e_0 \textbf{ then } 1 + Fibonacci(2000) \textbf{ else } Fibonacci(1000) + 2.$$

To facilitate these transformations, we inline all function calls where the function is not defined recursively.

The power of these transformations depends on reasonings used in simplifying the conditionals, as have been studied in many program transformation methods [39, 34, 36, 14, 24]. At least syntactic equality can be used, which identifies the most obvious source of inaccuracy. These optimizations also speed up the symbolic evaluation, since now obviously infeasible execution paths are not searched.

These transformations have been implemented and applied on many test programs. Even though the resulting programs can be analyzed more accurately and more efficiently, we have not performed separate measurements. The major reason is that our example programs do not contain conditional tests that are implied by other conditional tests. These simple transformations are just examples of many powerful program optimization techniques, especially on functional programs, that can be used to make time-bound function more accurate as well as more efficient. We plan to explore more of these optimizations and measure their effects as we experiment with more programs.

6 Implementation and experimentation

We have implemented the analysis approach in a prototype system, ALPA (Automatic Language-based Performance Analyzer). We performed a large number of measurements and obtained encouraging good results. We also used the system to obtain the exact symbolic counts and the performance measurements shown in Section 4.

The implementation is for a subset of Scheme. An editor for the source programs is implemented using the Synthesizer Generator [29], and thus we can easily change the syntax for the source programs. For example, the current implementation supports both the syntax used in this paper and Scheme syntax. Time-bound functions are constructed using SSL, a simple functional language used in the Synthesizer Generator. Lifting conditions, simplifying conditionals, and inlining nonrecursive calls are also implemented in SSL; they can be applied on the source program before constructing the time-bound function. The symbolic evaluation and optimizations, as well as measurements of primitive parameters, are written in Scheme. The measurements

and analyses are performed for source programs compiled with Chez Scheme compiler [5]. 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.[2] The source program does not use any library; in particular, no numbers are large enough to trigger the bignum implementation of Chez Scheme. Our current system does not handle the effects of cache memory or instruction pipelining; thus we tried to avoid producing large data in the example programs to minimize possible cache effects.

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 0.001 microseconds, i.e., 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 3 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 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.

	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	0.27546	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	54.9593	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	3.93600	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

Figure 3: Calculated and measured worst-case times (in milliseconds), without garbage collection.

[2] We had originally tried to avoid garbage collection by writing loops instead of recursions as much as possible and tried to exclude garbage-collection times completely. The idea of including garbage-collection times comes from an earlier experiment, where we mistakenly used a timing function of Chez Scheme that included garbage-collection time but we thought didn't and got fairly good results.

Figure 4 shows the calculated and measured worst-case times that include garbage-collection times. The results are similar to whose when garbage-collection times are excluded, except that the percentages are consistently higher than in Figure 3. In particular, underestimations occur more often for extremely small inputs, for inputs of size 1000 as well as 2000 on some examples, and for a few other inputs (about 1-2%, in 2 cases). We believe that this is the effect of garbage collection, which we have only measured in general but not analyzed specifically.

	insertion sort			selection sort			mergesort		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.06844	0.06698	97.9	0.13610	0.12778	93.9	0.11701	0.11273	96.3
20	0.26008	0.26476	101.8	0.53301	0.48645	91.3	0.29486	0.28216	95.7
50	1.57539	1.53062	97.2	3.28974	3.06625	93.2	0.93673	0.88150	94.1
100	6.23544	6.06750	97.3	13.1042	12.1850	93.0	2.17502	2.03875	93.7
200	24.8100	25.1187	101.2	52.3083	49.3375	94.3	4.95249	4.70100	94.9
300	55.7240	55.8428	100.2	117.612	115.718	98.4	7.94661	7.75000	97.5
500	154.570	153.125	99.1	326.519	320.833	98.3	14.2718	13.3200	93.3
1000	617.623	630.750	102.1	1305.53	1585.50	121.4	31.5533	29.5937	93.8
2000	2469.18	3318.50	134.3	5221.06	8376.25	160.4	69.1252	68.7000	99.4

	set union			list reversal			reversal w/app.		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.10318	0.09875	95.7	0.00935	0.00960	102.7	0.05325	0.04996	93.8
20	0.38230	0.36242	94.8	0.01832	0.01740	95.0	0.19596	0.18077	92.2
50	2.27639	2.12062	93.2	0.04521	0.04375	96.8	1.16194	1.06250	91.4
100	8.95569	8.3650	93.4	0.09003	0.08531	94.8	4.56477	4.1840	91.7
200	35.5235	33.5167	94.4	0.17967	0.17131	95.3	18.0936	16.6416	92.0
300	79.7037	75.3800	94.6	0.26932	0.25625	95.1	40.5867	37.4921	92.4
500	220.901	208.355	94.3	0.44860	0.42530	94.8	112.465	108.325	96.3
1000	882.111	839.96	95.2	0.89682	0.86580	96.5	449.038	421.8	93.9
2000	3525.45	3385.93	96.0	1.79324	1.74350	97.2	1794.50	2473.5	137.8

Figure 4: Calculated and measured worst-case times (in milliseconds), with garbage collection.

In general, the measured worst-case times are closely bounded by calculated upper bounds for all inputs of medium sizes (up to 500 for measurements including garbage-collection time, up to 1000 excluding garbage-collection time, and even larger for faster programs or programs that use less space). Figure 5 depicts the numbers in Figure 3. Examples such as sorting are classified as complex examples in previous study [28, 22], 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 [28] or 20 [22]) is reported in each experiment.

We found that when inputs are extremely small (10 or 20), the measured time is occasionally above the calculated time for some examples. Also, when inputs are large (1000 for measurements including garbage-collection time, or 2000 excluding garbage-collection time), 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); for example, on input of size 200, the measured time is 65% higher than the calculated time. 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.

Among fifteen programs we have analyzed using ALPA, two of them 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.

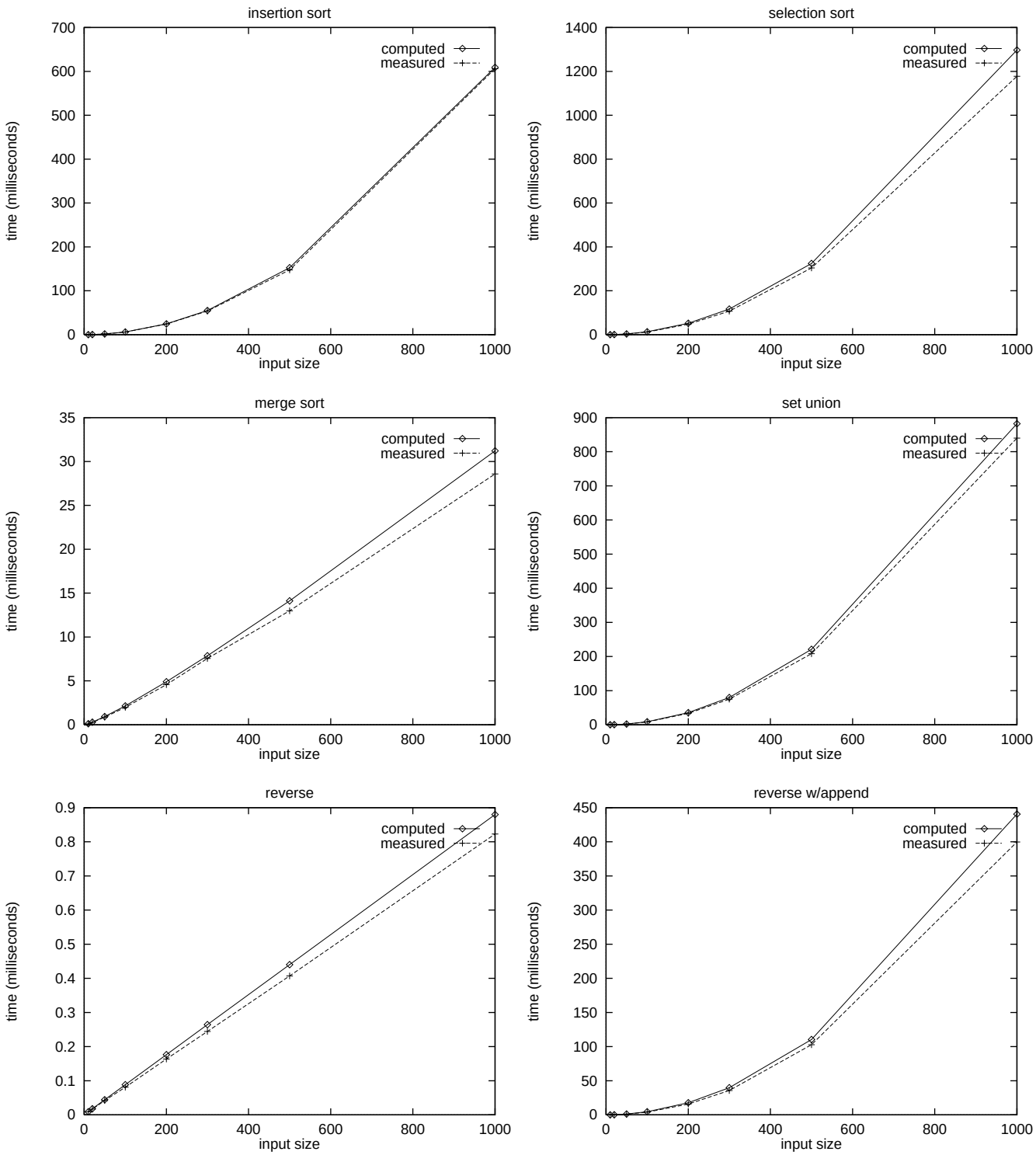

Figure 5: Comparison of calculated and measured worst-case times, without garbage collection.

7 Related work and conclusion

An overview of comparison with related work in timing analysis appears in Section 2. Certain detailed comparisons have also been discussed while presenting our method. This section summarizes them, compares with analyses for loop bounds and execution paths in more detail, and concludes.

Compared to work in algorithm analysis and program complexity analysis [21, 33, 41], this work consistently pushes through symbolic primitive parameters, so it allows us to calculate actual time bounds and validate the results with experimental measurements. There is also work on analyzing average-case complexity [13], which has a different goal than worst-case bounds. Compared to work in systems [35, 28, 27, 22], this work explores program analysis and transformation techniques to make the analysis automatic, efficient, and accurate, overcoming 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 [32, 31]. In that work, information about execution paths was obtained by running the programs on a number of inputs; for programs such as insertion sort whose best-case and worst-case execution times differ greatly, the predicted time using this method could be very inaccurate.

A number of techniques have been studied for obtaining loop bounds or execution paths [27, 2, 10, 15, 17]. Manual annotations [27, 22] are inconvenient and error-prone [2]. Automatic analysis of such information has two main problems. First, even when a precise loop bound can be obtained by symbolic evaluation of the program [10], separating the loop and path information from the rest of the analysis is in general less accurate than an integrated analysis [25]. Second, approximations for merging paths from loops, or recursions, very often lead to nontermination of the time analysis, not just looser bounds [10, 15, 25]. Some new methods, while powerful, apply only to certain classes of programs [17]. In contrast, our method allows recursions, or loops, to be considered naturally in the overall execution-time analysis based on partially known input structures. Our method also does not merge paths from recursions, or loops; this may cause exponential time complexity in the analysis, but our experiments on test programs show that the analysis is still tractable for inputs of size in the thousands. We have also studied simple but powerful optimizations to speed up the analysis.

In a new analysis for cache behavior [11], loops are transformed into recursive calls, and a predefined *callstring* level determines how many times the fixed point analysis iterates and thus how the analysis results are approximated. Our method allows the analysis to perform the exact number of recursions, or iterations, for the given partial input data structures. The most recent work by Lundqvist and Stenstrom [25] 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, for example, a program that computes the union of two lists with no repeated elements. We apply the ideas at source-level, and 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 [30]. 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, incremental computation, and transformations of conditionals to make the analysis more efficient and more accurate. Partial evaluation [4, 19], incremental computation [24, 23], and other transformations have been studied intensively in programming languages. Their applications in our time-bound analysis are particularly simple and clean; the resulting transformations are fully automatic and efficient.

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 [28]. 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 [22, 11]. In particular, we plan to apply our general approach to analyze space consumption and hence to help predict garbage-collection and caching behavior.

In conclusion, the approach we propose is based entirely on high-level programming languages. The methods and techniques are intuitive; together they produce automatic tools for analyzing time bounds efficiently and accurately. We find the accuracy of the experimental results very encouraging, especially considering that we are analyzing recursive programs at source-level, with garbage collection, and currently without special treatment for instruction pipelining or cache effects.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

[2] 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.

[3] R. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 13th IEEE Real-Time Systems Symposium.* IEEE CS Press, Los Alamitos, Calif., 1994.

[4] B. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation.* North-Holland, Amsterdam, 1988.

[5] Cadence Research Systems. *Chez Scheme System Manual.* Cadence Research Systems, Bloomington, Indiana, revision 2.4 edition, July 1994.

[6] 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.

[7] J. Cohen. Computer-assisted microanalysis of programs. *Commun. ACM*, 25(10):724–733, Oct. 1982.

[8] R. K. Dybvig. *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1987.

[9] 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.

[10] 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.

[11] 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.

[12] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-Upsilon-Omega: An assistant algorithms analyzer. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 201–212, Rome, Italy, July 1989. Springer-Verlag, Berlin.

[13] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science, Series A*, 79(1):37–109, Feb. 1991.

[14] Y. Futamura and K. Nogi. Generalized partial evaluation. In Bjørner et al. [4], pages 133–151.

[15] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2), June 1998.

[16] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 68–77. IEEE CS Press, Los Alamitos, Calif., Dec. 1992.

[17] 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.

[18] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260. ACM, New York, June 1992.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, N.J., 1993.

[20] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Mass., 1968.

[21] D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. and Syst.*, 10(2):248–266, Apr. 1988.

[22] 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.

[23] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. and Syst.*, 20(2), March 1998.

[24] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, Feb. 1995.

[25] 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.

[26] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* The MIT Press, Cambridge, Mass., 1990.

[27] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.

[28] 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.

[29] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1988.

[30] 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.

[31] 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.

[32] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38(12):1659–1679, Dec. 1989. Special issue on Performance Evaluation.

[33] 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.

[34] W. L. Scherlis. Program improvement by internal specialization. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 41–49. ACM, New York, Jan. 1981.

[35] A. Shaw. Reasoning about time in higher level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, July 1989.

[36] V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. and Syst.*, 8(3):292–325, July 1986.

[37] 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.

[38] B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–538, Sept. 1975.

[39] B. Wegbreit. Goal-directed program transformation. *IEEE Trans. Softw. Eng.*, SE-2(2):69–80, June 1976.

[40] 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.

[41] P. Zimmermann and W. Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. In *Computer and Information Sciences VI.* Elsevier, 1991.