Automatic Accurate Cost-Bound Analysis for High-Level Languages

Yanhong A. Liu
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Gustavo Gómez
Computer Science Department
Indiana University
Bloomington, IN 47405-7104

May 18, 2001

Corresponding author: Yanhong A. Liu
Email: liu@cs.sunysb.edu
Tel: 631-632-8463
Fax: 631-632-8334
URL: http://www.cs.sunysb.edu/∼liu/

©200x IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution ot servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Automatic Accurate Cost-Bound Analysis for High-Level Languages*

Yanhong A. Liu Gustavo Gómez

May 2001

Abstract—**This paper describes a language-based approach for automatic and accurate cost-bound analysis. The approach consists of transformations for building cost-bound functions in the presence of partially known input structures, symbolic evaluation of the cost-bound function based on input size parameters, and optimizations to make the overall analysis efficient as well as accurate, all at the source-language level. The calculated cost bounds are expressed in terms of primitive cost parameters. These parameters can be obtained based on the language implementation or be measured conservatively or approximately, yielding accurate, conservative, or approximate time or space bounds. We have implemented this approach and performed a number of experiments for analyzing Scheme programs. The results helped confirm the accuracy of the analysis.**

Index Terms—**Cost analysis, Cost bound, performance analysis and measurements, program analysis and transformation, program optimization, timing analysis, time analysis, space analysis, worst-case execution time.**

1 Introduction

Analysis of program cost, such as running time and space consumption, is important for real-time systems, embedded systems, interactive environments, compiler optimizations, performance evaluation, and many other computer applications. It has been extensively

*This work was supported in part by NSF under Grant CCR-9711253 and ONR under Grants N00014-99-1-0132 and N00014-01-1-0109. Yanhong A. Liu's address: Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400. Gustavo Gómez's address: Computer Science Department, Indiana University, Bloomington, IN 47405-7104. Corresponding author: Yanhong A. Liu. Email: liu@cs.sunysb.edu. Tel: 631-632-8463. Fax: 631-632-8334. URL: http://www.cs.sunysb.edu/~liu/.

studied in many fields of computer science: algorithms [25, 16, 17, 53], programming languages [50, 26, 41, 44], and systems [46, 37, 43, 42]. It is particularly important for many applications, such as real-time systems and embedded systems, to be able to predict accurate time bounds and space bounds automatically and efficiently, and it is particularly desirable to be able to do so for high-level languages [46, 37, 38].

For analyzing system running time, since Shaw proposed timing schema for high-level languages [46], a number of people have extended it for analysis in the presence of compiler optimizations [37, 12], pipelining [20, 28], cache memory [4, 28, 14], 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 [36, 3]. 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 [37], while the manual way of providing such information is potentially an even larger source of error, in addition to its inconvenience [36]. Various program analysis methods have been proposed to provide loop bounds or execution paths [3, 13, 19, 21]; 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. Similarly, loop bounds and recursion depths are needed also for space analysis [38].

This paper describes a *language-based* approach for automatic and accurate cost-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 cost-bound functions in the presence of partially known input structures, symbolic evaluation of the cost-bound function based on input size parameters, and optimizations to make overall the analysis efficient as well as accurate, all at the source-language level. We describe analysis and transformation algorithms and explain how they work. The calculated cost bounds are expressed in terms of primitive cost parameters. These parameters can be obtained based on the language implementation or be measured conservatively or approximately, yielding accurate, conservative, or approximate time or space bounds. The cost analysis currently does not include cache analysis. We have implemented this approach and performed a number of experiments for analyzing Scheme programs. The results helped confirm the accuracy of the analysis. 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 multiple kinds of cost analysis. Our main analysis sums the cost in terms of different operations performed; it gives upper bounds for all kinds of operations, such as arithmetic operations, data field selections, and constructor allocations. Variations of it can analyze stack space, live heap space, output size, etc., and can analyze lower bounds as well as upper bounds. The basic ideas also apply to other programming languages.

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

2.1 Cost and cost bound

Language-based cost-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 *cost function* that (takes the same input as the original program but) returns the cost 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 cost and by summing these parameters based on the semantics of the constructs [50, 10, 46]. We call parameters that describe the costs of program constructs *primitive cost parameters*. To calculate actual cost bounds based on the cost function, three difficult problems must be solved.

First, since the goal is to calculate cost 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 cost function. In general, due to imperfect knowledge about the input, the cost function is transformed into a *cost-bound function*.

In algorithm analysis, inputs are characterized by their size; accommodating this requires manual or semi-automatic transformation of the cost (time or space) function [50, 26, 53]. The analysis is mainly asymptotic, and primitive cost parameters are considered independent of input size, i.e., are constants while the computation iterates or recurses. Whatever values of the primitive cost parameters are assumed, a second problem arises, and it is theoretically

challenging: optimizing the cost-bound function to a closed form in terms of the input size [50, 10, 26, 41, 17, 7]. 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 [46, 37, 36, 28], even though program analyses can help in some cases [3, 13, 19, 21]. Closed forms in terms of parameters for these bounds can be obtained easily from the cost function. This isolates the third problem, which is most interesting to systems research: obtaining values of primitive cost parameters that depend on 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, for time analysis [37, 4, 28, 14]. 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.

In programming-language area, Rosendahl proposed using *partially known input structures* [41]. 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, such as n, for describing partially known input structures *input size parameters*. The cost function is then transformed automatically into a cost-bound function: at control points where decisions depend on unknown values, the maximum cost of all possible branches is computed; otherwise, the cost of the chosen branch is computed. Rosendahl concentrated on proving the correctness of this transformation. He assumed constant 1 for primitive cost parameters and relied on optimizations to obtain closed forms in terms of input size parameters, but again closed forms can not be obtained for all cost-bound functions.

2.2 Language-based cost-bound analysis

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

First, we use an automatic transformation to construct a cost-bound function from the original program based on partially known input structures. The resulting function takes input size parameters and primitive cost parameters as arguments. The only caveat here is that the cost-bound function might 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 cost bounds efficiently without relying on closed forms, we optimize the cost-bound function symbolically with respect to given values of input size parameters. This is based on partial evaluation and incremental computation. This symbolic evaluation always terminates provided the cost-bound function terminates. The resulting function expresses cost bounds as counts of different operations performed, where the cost of each kind of operations is denoted by a primitive cost parameter.

A third component consists of transformations that enable more accurate cost bounds to be computed: lifting conditions, simplifying conditionals, and inlining nonrecursive functions. The transformations should be applied on the original program before the cost-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 cost bounds, which can be computed more efficiently as well.

The approach is general because all three 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 cost-bound analysis for high-level languages is new. In the resulting cost bounds, primitive cost parameters can be obtained based on the language implementation or be measured conservatively or approximately, to give accurate, conservative, or approximate time or space bounds.

We have implemented the analyses and transformations for a subset of Scheme [2, 11, 1], a dialect of Lisp. All the transformations are done automatically, and the cost bounds, expressed as operation counts, are computed efficiently and accurately. Example programs analyzed include a number of classical sorting programs, matrix computation programs, and various list processing programs. We also estimated approximate bounds on the actual running times by measuring primitive cost parameters for running times using control loops, and calculated accurate bounds on the heap space allocated for constructors in the programs

based on the number of bytes allocated for each constructor by the compiler. We used a functional subset of Scheme 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 and space 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 estimating actual running time and space efficiently and accurately as well.

3) Analyses and transformations developed for functional language can be applied to improve analyses of imperative languages as well [52].

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

2.3 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

$$f(v_1, ..., v_n) \triangleq e$$

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

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

For binary primitive operations, we will be changing between infix and prefix notations depending on whichever is easier for the presentation. Following Lisp and Scheme, we use

[1] The keywords are taken from ML [35]. Our implementation supports both this syntax and Scheme syntax.

$cons(h, t)$ to construct a list with head h and tail t, and use $car(l)$ and $cdr(l)$ to select the head and tail, respectively, of list l. We use nil to denote an empty list, and use $null(l)$ to test whether l is an empty list. For example, the program below selects the least element in a non-empty list.

$$
\begin{array}{l}
least(x) \triangleq \ \textbf{if}\ null(cdr(x))\ \textbf{then}\ car(x) \\
\qquad\qquad\quad \textbf{else}\ \ \textbf{let}\ s = least(cdr(x)) \\
\qquad\qquad\qquad\quad \textbf{in}\ \ \textbf{if}\ car(x) \leq s\ \textbf{then}\ car(x)\ \textbf{else}\ s\ \textbf{end}
\end{array}
$$

We use *least* as a small running example. To present various analysis results, we also use several other examples: insertion sort, selection sort, merge sort, 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 cost analysis in the presence of arrays and pointers is not fundamentally harder [37], because the costs of the program constructs for them can be counted in a similar way as costs of other constructs. For example, accessing an array element $a[i]$ has the cost 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 [9, 22]. For pure functional languages, higher-order functions and lazy evaluations are important. Cost functions that accommodate these features have been studied [49, 44]. The symbolic evaluation and optimizations we describe apply to them as well.

3 Constructing cost-bound functions

3.1 Constructing cost functions

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

Given an original program, we add a set of cost functions, one for each original function, which simply count the cost while the original program executes. The algorithm, given below,

is presented as a transformation $\mathcal{C}$ on the original program, which calls a transformation $\mathcal{C}_e$ to recursively transform subexpressions. For example, a variable reference is transformed into a symbol C_{varref} representing the cost of a variable reference; a conditional statement is transformed into the cost of the test plus, if the condition is true, the cost of the true branch, otherwise, the cost of the false branch, and plus the cost for the transfers of control. We use cf to denote the cost function for f.

$$\text{program: } \mathcal{C}\left[\!\!\left[\begin{array}{l} f_1(v_1,...,v_{n_1}) \triangleq e_1; \\ ... \\ f_m(v_1,...,v_{n_m}) \triangleq e_m; \end{array}\right]\!\!\right] = \begin{array}{ll} f_1(v_1,...,v_{n_1}) \triangleq e_1; & cf_1(v_1,...,v_{n_1}) \triangleq \mathcal{C}_e[\![e_1]\!]; \\ ... & ... \\ f_m(v_1,...,v_{n_m}) \triangleq e_m; & cf_m(v_1,...,v_{n_m}) \triangleq \mathcal{C}_e[\![e_m]\!]; \end{array}$$

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

Applying this transformation to program *least*, we obtain function *least* as originally given and cost function *cleast* below, where infix notation is used for additions, and unnecessary parentheses are omitted. Note that various C's are indeed arguments to the cost function *cleast*; we omit them from argument positions for ease of reading.

$$\begin{array}{l} cleast(x) \triangleq \; C_{if} + C_{null} + C_{cdr} + C_{varref} \\ \quad +(\textbf{ if } null(cdr(x)) \textbf{ then } C_{car} + C_{varref} \\ \quad\quad \textbf{else } C_{let} + C_{call} + C_{cdr} + C_{varref} + cleast(cdr(x)) \\ \quad\quad\quad +(\textbf{ let } s = least(cdr(x)) \\ \quad\quad\quad\quad \textbf{in } C_{if} + C_{\leq} + C_{car} + C_{varref} + C_{varref} \\ \quad\quad\quad\quad\quad +(\textbf{ if } car(x) \leq s \textbf{ then } C_{car} + C_{varref} \textbf{ else } C_{varref}) \textbf{ end})) \end{array}$$

This transformation is similar to the local cost assignment [50], step-counting function [41], cost function [44], etc. in other work. Our transformation extends those methods with bindings, and makes all primitive cost parameters explicit at the source-language level. For example, each primitive operation p is given a different symbol C_p, and each constructor c is given a different symbol C_c. Note that the cost function terminates with the appropriate sum of primitive cost 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 cost function.

3.2 Constructing cost-bound functions

Characterizing program inputs and capturing them in the cost function are difficult to automate [50, 26, 46]. However, partially known input structures provide a natural means [41]. 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.

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

Similar structures can be used to describe an array of n elements, a matrix of m-by-n elements, a complete binary tree of height h, 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. For example, $f_{\leq}(v_1, v_2) \triangleq \textbf{if } v_1 = unknown \vee v_2 = unknown \textbf{ then } unknown \textbf{ else } v_1 \leq v_2$. We also define a new function lub, denoting least upper bound, that takes two values and returns the most precise partially known structure that both values conform with. For example, if $v_1 = cons(3, nil)$ and $v_2 = cons(4, nil)$, then $lub(v_1, v_2) = cons(unknown, nil)$.

$$\begin{aligned} f_p(v_1, ..., v_n) \triangleq\ & \textbf{if } v_1 = unknown \\ & \quad \vee \ ... \ \vee \\ & \quad v_n = unknown \\ & \textbf{then } unknown \\ & \textbf{else } p(v_1, ..., v_n) \end{aligned}$$

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

Also, the cost functions need to be transformed to compute an upper bound of the cost: if the truth value of a conditional test is known, then the cost of the chosen branch is computed normally, otherwise, the maximum of the costs of both branches is computed. Transformation $\mathcal{B}$, given below, embodies these algorithms, where $\mathcal{B}_e$ transforms an expression in the original functions, and $\mathcal{B}_c$ transforms an expression in the cost functions. We use uf to denote function f extended with the value *unknown*, and we use cbf to denote the cost-bound

function for f.

$$\text{program: } \mathcal{B}\left[\!\!\left[\begin{array}{ll} f_1(v_1,...,v_{n_1}) \triangleq e_1; & cf_1(v_1,...,v_{n_1}) \triangleq e'_1; \\ ... & ... \\ f_m(v_1,...,v_{n_m}) \triangleq e_m; & cf_m(v_1,...,v_{n_m}) \triangleq e'_m; \end{array}\right]\!\!\right]$$

$$= \begin{array}{lll} uf_1(v_1,...,v_{n_1}) \triangleq \mathcal{B}_e[\![e_1]\!]; & cbf_1(v_1,...,v_{n_1}) \triangleq \mathcal{B}_c[\![e'_1]\!]; & f_p(v_1,...,v_n) \triangleq \text{... as above} \\ ... & ... & \\ uf_m(v_1,...,v_{n_m}) \triangleq \mathcal{B}_e[\![e_m]\!]; & cbf_m(v_1,...,v_{n_m}) \triangleq \mathcal{B}_c[\![e'_m]\!]; & lub(v_1,v_2) \triangleq \text{... as above} \end{array}$$

variable reference:	$\mathcal{B}_e[\![v]\!]$	$= v$
data construction:	$\mathcal{B}_e[\![c(e_1,...,e_n)]\!]$	$= c(\mathcal{B}_e[\![e_1]\!],...,\mathcal{B}_e[\![e_n]\!])$
primitive operation:	$\mathcal{B}_e[\![p(e_1,...,e_n)]\!]$	$= f_p(\mathcal{B}_e[\![e_1]\!],...,\mathcal{B}_e[\![e_n]\!])$
conditional:	$\mathcal{B}_e[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]$	$= \textbf{let } v = \mathcal{B}_e[\![e_1]\!]$
		$\textbf{in if } v = unknown \textbf{ then } lub(e'_2, e'_3)$
		$\quad \textbf{else if } v \textbf{ then } e'_2 \textbf{ else } e'_3 \textbf{ end}$
		where $e'_2 = \mathcal{B}_e[\![e_2]\!]$, $e'_3 = \mathcal{B}_e[\![e_3]\!]$
binding:	$\mathcal{B}_e[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!]$	$= \textbf{let } v = \mathcal{B}_e[\![e_1]\!] \textbf{ in } \mathcal{B}_e[\![e_2]\!] \textbf{ end}$
function call:	$\mathcal{B}_e[\![f(e_1,...,e_n)]\!]$	$= uf(\mathcal{B}_e[\![e_1]\!],...,\mathcal{B}_e[\![e_n]\!])$
primitive cost parameter:	$\mathcal{B}_c[\![C]\!]$	$= C$
summation:	$\mathcal{B}_c[\![add(e_1,...,e_n)]\!]$	$= add(\mathcal{B}_c[\![e_1]\!],...,\mathcal{B}_c[\![e_n]\!])$
conditional:	$\mathcal{B}_c[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]$	$= \textbf{let } v = \mathcal{B}_e[\![e_1]\!]$
		$\textbf{in if } v = unknown \textbf{ then } max(e'_2, e'_3)$
		$\quad \textbf{else if } v \textbf{ then } e'_2 \textbf{ else } e'_3 \textbf{ end}$
		where $e'_2 = \mathcal{B}_c[\![e_2]\!]$, $e'_3 = \mathcal{B}_c[\![e_3]\!]$
binding:	$\mathcal{B}_c[\![\textbf{let } v = e_1 \textbf{ in } e_2 \textbf{ end}]\!]$	$= \textbf{let } v = \mathcal{B}_e[\![e_1]\!] \textbf{ in } \mathcal{B}_c[\![e_2]\!] \textbf{ end}$
function call:	$\mathcal{B}_c[\![cf(e_1,...,e_n)]\!]$	$= cbf(\mathcal{B}_e[\![e_1]\!],...,\mathcal{B}_e[\![e_n]\!])$

Applying this transformation on functions *least* and *cleast* yields functions *uleast* and *cbleast* below, where function f_p for each primitive operator p and function *lub* are as given above. Shared code is presented with where-clauses when this makes the code smaller.

$$\begin{array}{l} uleast(x) \triangleq \textbf{ let } v = f_{null}(f_{cdr}(x)) \\ \quad \textbf{in if } v = unknown \textbf{ then } lub(e_1, e_2) \\ \quad\quad \textbf{else if } v \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ end} \\ \quad \text{where } e_1 = f_{car}(x) \\ \quad\quad\quad e_2 = \textbf{let } s = uleast(f_{cdr}(x)) \\ \quad\quad\quad\quad \textbf{in let } v = f_{\leq}(f_{car}(x), s) \\ \quad\quad\quad\quad\quad \textbf{in if } v = unknown \textbf{ then } lub(f_{car}(x), s) \\ \quad\quad\quad\quad\quad\quad \textbf{else if } v \textbf{ then } f_{car}(x) \textbf{ else } s \textbf{ end end} \end{array}$$

$$
\begin{array}{l}
cbleast(x) \triangleq\ C_{if} + C_{null} + C_{cdr} + C_{varref} \\
\quad +(\textbf{let } v = f_{null}(f_{cdr}(x)) \\
\quad\quad \textbf{in if } v = unknown \textbf{ then } max(e_1, e_2) \\
\quad\quad\quad \textbf{else if } v \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ end}) \\
\quad \text{where } e_1 = C_{car} + C_{varref} \\
\quad\quad\quad e_2 = C_{let} + C_{call} + C_{cdr} + C_{varref} + cbleast(f_{cdr}(x)) \\
\quad\quad\quad\quad +(\textbf{let } s = uleast(f_{cdr}(x)) \\
\quad\quad\quad\quad\quad \textbf{in } C_{if} + C_{\leq} + C_{car} + C_{varref} + C_{varref} \\
\quad\quad\quad\quad\quad\quad +(\textbf{let } v = f_{\leq}(f_{car}(x), s) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{in if } v = unknown \textbf{ then } max(C_{car} + C_{varref}, C_{varref}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else if } v \textbf{ then } C_{car} + C_{varref} \textbf{ else } C_{varref} \textbf{ end}) \textbf{ end})
\end{array}
$$

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

Both transformations $\mathcal{C}$ and $\mathcal{B}$ 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 cost-bound function might 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 cost-bound function for any recursive function does not terminate, since the original program does cost infinite resource in the worst case. We can modify the analysis to detect nontermination in many cases, as for example in [27]. For the example of giving *unknown* to a recursive cost-bound function, nontermination is trivial to detect, since the arguments to recursive calls would remain *unknown*.

4 Optimizing cost-bound functions

This section describes symbolic evaluation and optimizations that make computation of cost 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 cost-bound functions might 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 size parameters, since it essentially searches for the worst-case execution path for all inputs satisfying the partially known input structures.

4.1 Partial evaluation of cost-bound functions

In practice, values of input size 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 cost-bound function symbolically in terms of primitive cost parameters given specific values of input size parameters.

The evaluation simply follows the structures of cost-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 cost parameters and maximums among alternative sums, which can easily be done symbolically. Thus, the transformation inlines all function calls, sums all primitive cost 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 (where each variable is mapped to its value) and returns as result a symbolic value that contains the primitive cost parameters. The evaluation starts with the application of the cost-bound function to a partially unknown input structure, e.g., $cbleast(list(100))$, and it starts with an empty environment. We assume that add_s is a function that symbolically sums its arguments, and max_s is a function that symbolically takes the maximum of its arguments.

variable reference:	$\mathcal{E}\,[\![v]\!]\,\rho$	$= \rho(v)$
		look up binding of v in environment
primitive cost parameter:	$\mathcal{E}\,[\![C]\!]\,\rho$	$= C$
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:	$\mathcal{E}\,[\![add(e_1, ..., e_n)]\!]\,\rho$	$= add_s(\mathcal{E}\,[\![e_1]\!]\,\rho, ..., \mathcal{E}\,[\![e_n]\!]\,\rho)$
maximum:	$\mathcal{E}\,[\![max(e_1, ..., e_n)]\!]\,\rho$	$= max_s(\mathcal{E}\,[\![e_1]\!]\,\rho, ..., \mathcal{E}\,[\![e_n]\!]\,\rho)$
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 to value of e_1 in environment
function calls:	$\mathcal{E}\,[\![f(e_1, ..., e_n)]\!]\,\rho$	$= \mathcal{E}\,[\![e]\!]\,\rho[v_1 \mapsto \mathcal{E}\,[\![e_1]\!]\,\rho, ..., v_n \mapsto \mathcal{E}\,[\![e_n]\!]\,\rho]$
	where f is defined by $f(v_1, ..., v_n) \triangleq e$	

As an example, applying symbolic evaluation to *cbleast* on a list of size 100, we obtain

the following result:

$$\begin{aligned} cbleast(list(100)) \triangleq\ & 497 * C_{varref} + 100 * C_{null} + 199 * C_{car} + 199 * C_{cdr} \\ & + 99 * C_{\leq} + 199 * C_{if} + 99 * C_{let} + 99 * C_{call} \end{aligned}$$

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

The symbolic evaluation given only values of input size parameters is inefficient compared to direct evaluation given values of both input size parameters and particular primitive cost parameters, even though the resulting function takes virtually constant time given any values of primitive cost parameters. For example, directly evaluating a quadratic-time reverse function (that uses append operation) on input of size 20 takes about 0.96 milliseconds, whereas the symbolic evaluation takes 670 milliseconds, hundreds of times slower. We propose further optimizations below that greatly speed up the symbolic evaluation.

4.2 Avoiding repeated summations over recursions

The symbolic evaluation above is a global optimization over all cost-bound functions involved. During the evaluation, summations of symbolic primitive cost 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 cost parameter. The transformation $\mathcal{S}$, given below, performs this optimization. We use $vcbf$ to denote the transformed cost-bound function of f that operates on vectors. We use function add_v to compute component-wise sum of the argument vectors, and we use function max_v to compute component-wise maximum of the argument vectors.

$$\text{program: } \mathcal{S}\left[\!\!\left[\begin{array}{l} cbf_1(v_1, ..., v_{n_1}) = e_1; \\ ... \\ cbf_m(v_1, ..., v_{n_m}) = e_m; \end{array} \right]\!\!\right] = \begin{array}{l} vcbf_1(v_1, ..., v_{n_1}) = \mathcal{S}_c[\![e_1]\!]; \\ ... \\ vcbf_m(v_1, ..., v_{n_m}) = \mathcal{S}_c[\![e_m]\!]; \end{array}$$

primitive cost parameter:	$\mathcal{S}_c[\![C]\!]$	$=$	create a vector of 0's except with the component corresponding to C set to 1
summation:	$\mathcal{S}_c[\![add(e_1, ..., e_n)]\!]$	$=$	$add_v(\mathcal{S}_c[\![e_1]\!], ..., \mathcal{S}_c[\![e_n]\!])$
maximum:	$\mathcal{S}_c[\![max(e_1, ..., e_n)]\!]$	$=$	$max_v(\mathcal{S}_c[\![e_1]\!], ..., \mathcal{S}_c[\![e_n]\!])$
all others:	$\mathcal{S}_c[\![e]\!]$	$=$	e

Let V be the following vector of primitive cost parameters:

$$\langle C_{varref}, C_{nil}, C_{cons}, C_{null}, C_{car}, C_{cdr}, C_{\leq}, C_{if}, C_{let}, C_{call} \rangle$$

Applying the above transformation on function *cbleast* yields function *vcbleast*, where components of the vectors correspond to the components of V, and infix notation $+_v$ is used for vector addition.

$$\begin{array}{l}
vcbleast(x) \triangleq \ < 1,0,0,1,0,1,0,1,0,0 > \\
\quad +_v(\textbf{let } v = f_{null}(f_{cdr}(x)) \\
\quad\quad \textbf{in if } v = unknown \textbf{ then } max_v(e_1, e_2) \\
\quad\quad\quad \textbf{else if } v \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ end}) \\
\quad \text{where } e_1 = < 1,0,0,0,1,0,0,0,0,0 > \\
\quad\quad e_2 = < 1,0,0,0,0,1,0,0,1,1 > +_v vcbleast(cdr(x)) \\
\quad\quad\quad +_v(\textbf{let } s = uleast(f_{cdr}(x)) \\
\quad\quad\quad\quad \textbf{in } < 2,0,0,0,1,0,1,1,0,0 > \\
\quad\quad\quad\quad\quad +_v(\textbf{let } v = f_{\leq}(f_{car}(x), s) \\
\quad\quad\quad\quad\quad\quad \textbf{in if } v = unknown \textbf{ then } < 1,0,0,0,1,0,0,0,0,0 > \\
\quad\quad\quad\quad\quad\quad\quad \textbf{else if } v \textbf{ then } < 1,0,0,0,1,0,0,0,0,0 > \\
\quad\quad\quad\quad\quad\quad\quad \textbf{else } < 1,0,0,0,0,0,0,0,0,0 > \textbf{ end}) \textbf{ end})
\end{array}$$

The cost-bound function $cbleast(x)$ is simply the dot product of $vcbleast(x)$ and V.

This transformation incrementalizes the computation over recursions 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 drastic speedup of the evaluation. For example, optimized symbolic evaluation of the same quadratic-time reverse on input of size 20 takes only 2.55 milliseconds, while direct evaluation takes 0.96 milliseconds, resulting in less than 3 times slow-down; it is over 260 times faster than symbolic evaluation without this optimization.

5 Making cost-bound functions accurate

While loops and recursions affect cost bounds most, the accuracy of the cost bounds calculated also depends on the handling of the conditionals in the original program, which is reflected in the cost-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 cost. This is a major source of accuracy for our worst-case bound.

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

$$\begin{array}{l} e_1 = \textbf{if } v \textbf{ then } 1 \textbf{ else } Fibonacci(1000) \\ e_2 = \textbf{if } v \textbf{ then } Fibonacci(2000) \textbf{ else } 2 \end{array}$$

If we compute the cost bound for $e_1 + e_2$ directly, the result is at least $cFibonacci(1000) + cFibonacci(2000)$. However, if we consider only the two realizable execution paths, we know that the worst case is $cFibonacci(2000)$ plus some small constants. This is known as the false-path elimination problem [3].

Two transformations, *lifting conditions* and *simplifying conditionals*, applied on the source program before constructing the cost-bound function, allow us to achieve the accurate analysis results. In each function definition, the former lifts conditions to the outermost 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

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

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

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

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 [51, 45, 47, 18, 32]. 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 cost-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.

Note that these transformations on the source program are aimed at making the cost-bound function more accurate and more efficient, not at optimizing the source program. Even though making the source program faster also makes the corresponding cost-bound function faster, these two goals are different. Optimizing the source program is meant to produce a different program that has a smaller cost. Cost analysis is meant to analyze accurately the cost of a given program.

To make use of all the techniques for making cost-bound analysis efficient and accurate, we perform an overall cost-bound analysis by applying the following transformations in order to the source program: lifting conditions and simplifying conditionals (as in Section 5), constructing cost functions and then cost-bound functions (as in Section 3), and precomputing repeated local summations and then performing global symbolic evaluation (as in Section 4).

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 experiments and obtained encouraging good results.

6.1 Implementation and experimental results

The implementation is for a subset of Scheme [2, 11, 1]. An editor for the source programs is implemented using the Synthesizer Generator [40], 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. Construction of cost-bound functions is written in SSL, a simple functional language used in the Synthesizer Generator. Lifting conditions,

simplifying conditionals, and inlining nonrecursive calls are also implemented in SSL. The symbolic evaluation and optimizations are written in Scheme.

Figure 1 gives the results of symbolic evaluation of the cost-bound functions for six example programs on inputs of sizes 10 to 2000. For example, the second row of the figure means that for insertion sort on inputs of size 10, the cost-bound function is

$$\begin{aligned} cbinsertionsort(list(10)) \triangleq\ & 321 * C_{varref} + 11 * C_{nil} + 55 * C_{cons} + 66 * C_{null} \\ & + 100 * C_{car} + 55 * C_{cdr} + 45 * C_{\leq} + 111 * C_{if} + 65 * C_{call} \end{aligned}$$

The last column lists the sums for every rows. For the set union example, we used inputs where both arguments were of the given sizes. These numbers in the figure characterize various aspects of the examples; they contribute to the actual time and space bounds discussed below. We verified that all numbers are also exact worst-case counts. For example, for insertion sort on inputs of size 10, indeed 65 function calls are made during a worst-case execution. The worst-case counts are verified by using a modified evaluator. These experiments show that our cost-bound functions can give accurate cost bounds in terms of counts of different operations performed.

Figure 2 compares the times of direct evaluation of cost-bound functions, with each primitive cost 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 main memory. They include garbage-collection time. The times without garbage-collection times are all about 1% faster, so they are not shown here. These experiments show that our optimizations of cost-bound functions allow symbolic evaluation to be only a few times slower than direct evaluation rather than hundreds of times slower.

For merge sort, the cost-bound function constructed using the algorithms in this paper takes several days to evaluate on inputs of size 50 or larger. Special but simple optimizations were done to obtain the numbers in Figure 1, namely, letting the cost-bound function for merge avoid base cases as long as possible and using sizes of lists in place of lists of unknowns; the resulting symbolic evaluation takes only seconds. Such optimizations are yet to be implemented to be performed automatically. For all other examples, it takes at most 2.7 hours to evaluate the cost-bound functions.

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

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
w/append	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 cost-bound functions.

the symbolic evaluation, cost bounds can be computed in virtually no time given values of primitive cost parameters.

	insertion sort		selection sort		merge sort		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).

Among over twenty 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. This is similar to nontermination caused by merging paths in other methods [33, 34], but nontermination happens much less often in our method, since we essentially avoid merging paths as much as possible. 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. That evaluation algorithm is not yet implemented. A future work is to exploit results from static analysis for identifying sources of nontermination [27] to make cost-bound analysis terminate more often. For practical use of a cost-bound analyzer that might not terminate on certain inputs, we can modify the evaluator so that if it is stopped at any time, it outputs the cost bound calculated till that point. This means that a longer-running analysis might yield a higher bound.

6.2 Further experiments

We also estimated approximate bounds on the actual running times by measuring primitive cost parameters for running times using control loops, and calculated accurate bounds on the heap space allocated for constructors in the programs based on the number of bytes allocated for each constructor by the compiler. For time-bound analysis, we performed two sets of experiments: the first for a machine with cache enabled, and the second for a machine with cache disabled. The first gives tight bounds in most cases but has a few underestimations for inputs that are very small or very large, which we attribute to the cache effects. The second gives conservative and tight bounds for all inputs. We first describe experiments for

time-bound analysis with cache enabled and for analysis of heap space allocation bound, and then analyze the cache effects and show results for time-bound analysis with cache disabled.

The measurements and analyses for time-bounds are performed for source programs compiled with Chez Scheme compiler [8]. The source program does not use any library; in particular, no numbers are large enough to trigger the bignum implementation of Chez Scheme. We tried to avoid compiler optimizations by setting the optimization level to 0; we view necessary optimizations as having already been applied to the program. To handle garbage-collection time, we performed separate sets of experiments: those that exclude garbage-collection times in both calculations and measurements, and those that include garbage-collection time in both.[2] Our current analysis does not handle the effects of cache memory or instruction pipelining; we approximated cache effects by taking operands circularly from a cycle of 2000 elements when measuring primitive cost parameters, as discussed further below. For time-bound analysis with cache enabled, the particular numbers reported are taken on a Sun Ultra 1 with 167MHz CPU and 64MB main memory; we have also performed the analysis for several other kinds of SPARC stations, and the results are similar.

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 cost 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 estimated and measured worst-case times for six example programs on inputs of sizes 10 to 2000. 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 very small (20, in 1 case) or very large (2000, in 3 cases), which is analyzed and addressed below. The measurements including garbage-collection times are similar except with a few more cases of underestimation. Figure 4 depicts the numbers in

[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.

	insertion sort			selection sort			merge sort		
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/append		
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) with cache enabled.

Figure 3 for inputs of sizes up to 1000. Examples such as sorting are classified as complex examples in previous study [37, 28], 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 [37] or 20 [28]) is reported in each experiment.

Using the cost bounds computed, we can also calculate, accurately instead of approximately, bounds on the heap space dynamically allocated for constructors in the source programs. The number of bytes allocated for each constructor can be obtained precisely based on the language implementation. For example, Chez Scheme allocates 8 bytes for a *cons*-cell on the heap; this information can also be obtained easily using its statistics utilities. Based on results in Figure 1, by setting C_{cons} to 8 and other primitive cost parameters to 0, we obtain exact bounds on the heap space dynamically allocated for constructors in the programs, as shown in Figure 5.

Consider the accuracy of the time-bound analysis with cache enabled. We found that when inputs are very small (20), the measured time is occasionally above the calculated time for some examples. Also, when inputs are very large (1000 for measurements including

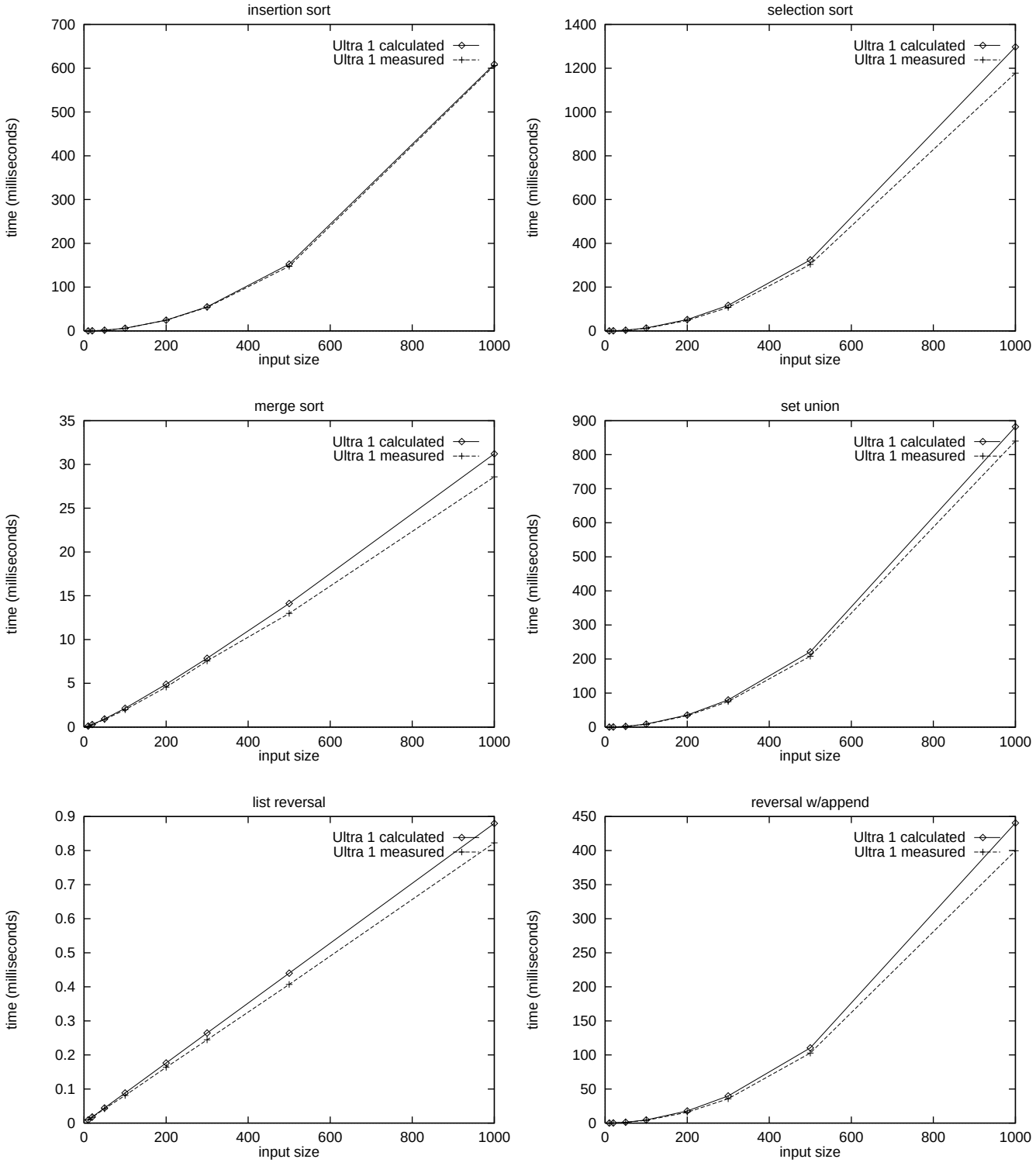

Figure 4: Comparison of calculated and measured worst-case times with cache enabled.

size	insertion sort	selection sort	merge sort	set union	list reversal	reversal w/app.
10	440	440	552	80	80	440
20	1680	1680	1416	160	160	1680
50	10200	10200	4584	400	400	10200
100	40400	40400	10760	800	800	40400
200	160800	160800	24712	1600	1600	160800
300	361200	361200	39816	2400	2400	361200
500	1002000	1002000	71816	4000	4000	1002000
1000	4004000	4004000	159624	8000	8000	4004000
2000	16008000	16008000	351240	16000	16000	16008000

Figure 5: Bounds of heap space allocated for constructors (in bytes).

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, for the following reasons. First, the initial cache misses are more likely to show up on small inputs. Second, underestimation for inputs of size 2000 in Figure 3 happens exactly for the 3 examples whose allocated heap space is very large in Figure 5, and recall that we used a cycled data structure of size 2000 when measuring primitive cost parameters. Furthermore, for programs that use less space, our calculated bounds are accuracy for even larger input sizes, and for programs that use extremely large amount of space even on small inputs, we have much worse underestimation. For example, for Cartesian product, underestimation occurs for small input sizes (50 to 200); as an example, on input of size 200, the measured time is 65% higher than the calculated time.

We performed a second set of experiments for time-bound analysis for a machine with cache disabled. The machine used is a Sun Ultra 10 with 333MHz CPU and 256MB main memory. Figure 6 shows the estimated and measured worst-case times for the same six programs on inputs of sizes 10 to 2000. These times do not include garbage-collection times. We can see that all measured times are closely bounded by the calculated times, with no underestimation. Figure 7 depicts the numbers in Figure 6.

To accommodate cache effect in time-bound analysis with cache enabled, we could adjust our measurements of primitive cost parameters on data structures of appropriate size. The appropriate size can be determined based on a precise space usage analysis. Heap-space allocation is only one less direct aspect. More directly, we can incorporate precise knowledge about compiler-generated machine instructions into our analysis method. We leave this as a future work. Our current method can be used for approximate time-bound estimation in the

	insertion sort			selection sort			merge sort		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.15222	0.14228	93.5	0.29483	0.25866	87.7	0.24955	0.23774	95.3
20	0.58026	0.53773	92.7	1.15757	1.00954	87.2	0.63029	0.60380	95.8
50	3.52196	3.22160	91.5	7.15578	6.22520	87.0	2.00717	1.91025	95.2
100	13.9499	12.6945	91.0	28.5194	24.4070	85.6	4.66697	4.38690	94.0
200	55.5253	50.5195	91.0	113.871	97.5660	85.7	10.6383	9.94885	93.5
300	124.726	113.551	91.0	256.057	219.080	85.6	17.0790	15.9820	93.6
500	346.007	315.220	91.1	710.928	610.595	85.9	30.6905	28.5640	93.1
1000	1382.66	1255.81	90.8	2842.68	2438.77	85.8	67.8999	63.3030	93.2
2000	5527.91	5053.00	91.4	11368.7	9794.00	86.1	148.836	138.786	93.2

	set union			list reversal			reversal w/append		
size	calculated	measured	me/ca	calculated	measured	me/ca	calculated	measured	me/ca
10	0.21630	0.21299	98.5	0.02065	0.02014	97.5	0.11724	0.10644	90.8
20	0.79790	0.77961	97.7	0.04051	0.03789	93.5	0.43258	0.38470	88.9
50	4.73684	4.60915	97.3	0.10007	0.09114	91.1	2.56979	2.24415	87.3
100	18.6155	18.0889	97.2	0.19933	0.17976	90.2	10.1024	8.75360	86.6
200	73.7997	71.7215	97.2	0.39786	0.35615	89.5	40.0575	34.6355	86.5
300	165.552	161.145	97.3	0.59639	0.53297	89.4	89.8657	77.8655	86.6
500	458.766	446.670	97.4	0.99345	0.88594	89.2	249.041	216.280	86.8
1000	1831.75	1784.91	97.4	1.98611	1.76579	88.9	994.409	859.320	86.4
2000	7320.41	7133.00	97.4	3.97142	3.52055	88.6	3974.12	3469.58	87.3

Figure 6: Calculated and measured worst-case times (in milliseconds) with cache disabled.

presence of low-level effects or precise analysis in their absence, and can be used for more accurate space-bound analysis that helps addressing memory issues.

7 Related work and conclusion

A preliminary version of this work appeared in [30]. An overview of comparison with related work in cost 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 [26, 44, 53, 7], this work consistently pushes through symbolic primitive cost parameters, so it allows us to calculate actual cost bounds and validate the results with experimental measurements. There is also work on analyzing average-case complexity [17], which has a different goal than worst-case bounds. Compared to work in systems [46, 37, 36, 28], 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,

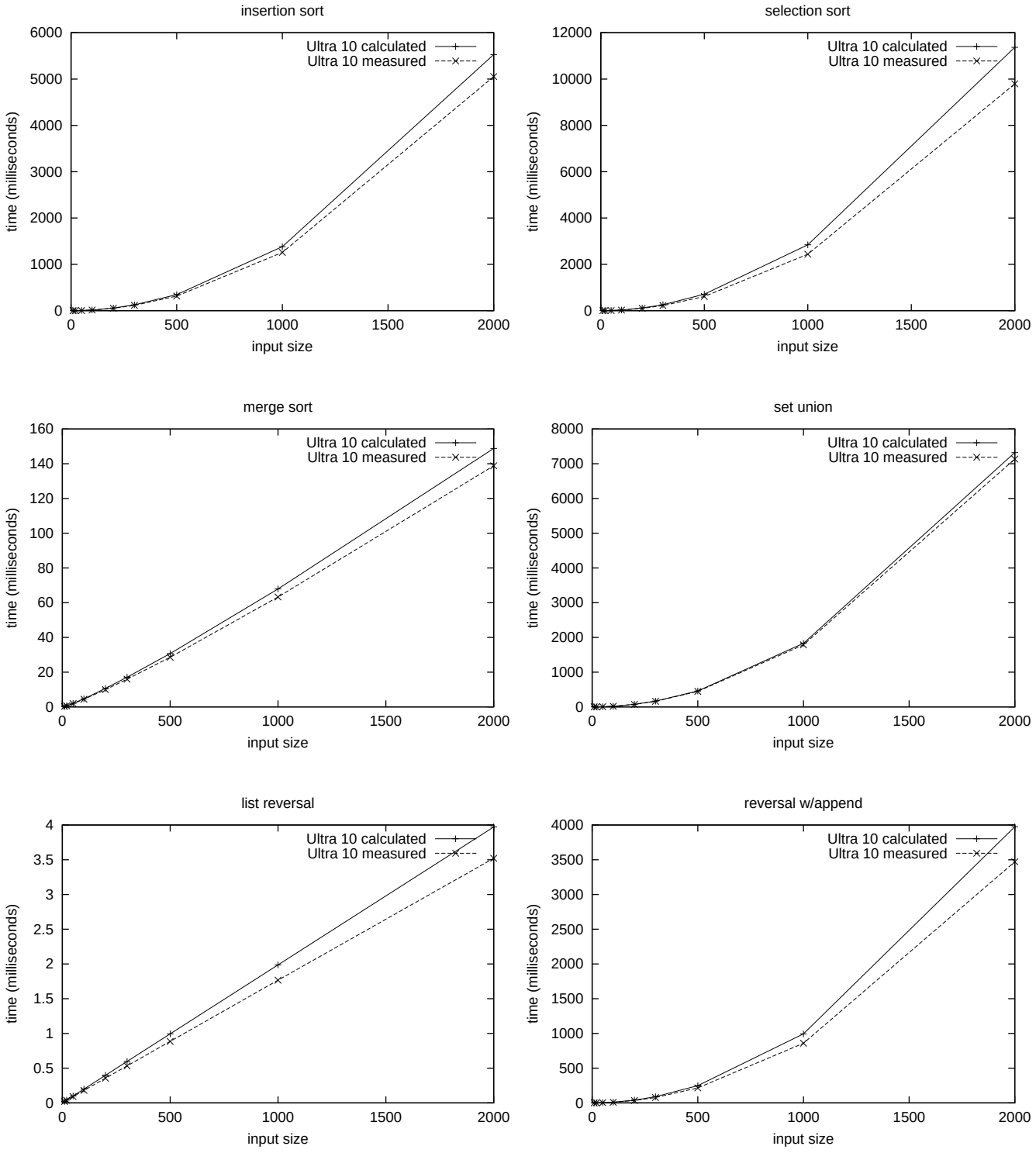

Figure 7: Comparison of calculated and measured worst-case times with cache disabled.

recursion depths, or execution paths automatically and precisely. There is also work for measuring primitive cost parameters for the purpose of general performance prediction [43, 42]. 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 for time analysis [36, 3, 13, 19, 21]. Manual annotations [36, 28] are inconvenient and error-prone [3]. 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 [13], separating the loop and path information from the rest of the analysis is in general less accurate than an integrated analysis [34]. Second, approximations for merging paths from loops, or recursions, very often lead to nontermination of the time analysis, not just looser bounds [13, 19, 34]. Some newer methods, while powerful, apply only to certain classes of programs [21]. In contrast, our method allows recursions, or loops, to be considered naturally in the overall cost analysis based on partially known input structures. In addition, our method does not merge paths from recursions, or loops; this may cause exponential time complexity of the analysis in the worst case, but our experiments on test programs show that the analysis is still feasible for inputs of sizes in the thousands. We have also studied simple but powerful optimizations to speed up the analysis dramatically.

In the analysis for cache behavior [14, 15], 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 partially known input data structures. The work by Lundqvist and Stenstrom [33, 34] is based on similar 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 they can not handle dynamically allocated data structures as we can, and their method for merging paths for loops would lead to nonterminating analysis for many more programs than our method. We apply the ideas at the source level, and our experiments show that we can calculate more accurate cost bound and for many more programs than merging paths, and the calculation is still efficient. There are also methods for time analysis based on program flow graphs [39, 6]. Unlike our method, these methods do not exploit given input sizes, and they require programmers to

give precise path information.

The idea of using partially known input structures originates from Rosendahl [41]. We have extended it to manipulate primitive cost parameters. We also handle binding constructs, which is simple but necessary for efficient computation. An innovation in our method is to optimize the cost-bound function using partial evaluation, incremental computation, and transformations of conditionals to make the analysis more efficient and more accurate. Partial evaluation [5, 24, 23], incremental computation [32, 31, 29], and other transformations have been studied intensively in programming languages. Their applications in our cost-bound analysis are particularly simple and clean; the resulting transformations are fully automatic and efficient.

We have started to explore a suite of new language-based techniques for cost analysis, in particular, analyses and optimizations for further speeding up the evaluation of the cost-bound function. We have also applied our general approach to analyze stack space and live heap space [48], which can further help predict garbage-collection and caching behavior. We can also analyze lower bounds using a symmetric method, namely by replacing maximum with minimum at all conditional points. A future work is to accommodate more lower-level dynamic factors for timing at the source-language level [28, 14], by examining the corresponding compiler generated code, where cache and pipelining effects are explicit.

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 cost bounds efficiently and accurately and can be used to accurately or approximately analyze time and space bounds.

Acknowledgment

We thank the anonymous referees for their careful reviews and many very helpful comments.

References

[1] H. Abelson, R. K. Dybvig, et al. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug. 1998.

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

[3] 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.

[4] 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.

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

[6] J. Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Systems*. To appear.

[7] J. Blieberger and R. Lieger. Worst-case space and time complexity of recursive procedures. *Real-Time Systems*, 11(2):115–144, 1996.

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, Nov. 1999.

[16] 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. Springer-Verlag, Berlin, July 1989.

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

[18] Y. Futamura and K. Nogi. Generalized partial evaluation. In Bjørner et al. [5], pages 133–151.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

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

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

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

[27] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, Jan. 2001.

[28] 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.

[29] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, Dec. 2000.

[30] Y. A. Liu and G. Gómez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 31–40. Springer-Verlag, June 1998.

[31] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998.

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

[33] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 1998.

[34] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, Nov. 1999.

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

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

[37] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Comput.*, 24(5):48–57, 1991.

[38] P. Persson. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 45–54. ACM, New York, May 1999.

[39] P. P. Puschner and A. V. Schedl. Computing maximum task execution times — a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

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

[41] 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.

[42] 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.

[43] 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.

[44] D. Sands. Complexity analysis for a lazy higher-order language. In N. D. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 361–376. Springer-Verlag, Berlin, May 1990.

[45] 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.

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

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

[48] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *Proceedings of the ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems*. ACM, New York, June 2001.

[49] 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.

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

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

[52] 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.

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