Lower Bound Cost Estimation for Logic Programs

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
debray@cs.arizona.edu

Pedro López-García, Manuel Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid
E-28660 Madrid, Spain
pedro@dia.fi.upm.es, herme@.fi.upm.es

Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, 62107, Taiwan, R.O.C.
naiwei@cs.ccu.edu.tw

Abstract

It is generally recognized that information about the runtime cost of computations can be useful for a variety of applications, including program transformation, granularity control during parallel execution, and query optimization in deductive databases. Most of the work to date on compile-time cost estimation of logic programs has focused on the estimation of upper bounds on costs. However, in many applications, such as parallel implementations on distributed-memory machines, one would prefer to work with lower bounds instead. The problem with estimating lower bounds is that in general, it is necessary to account for the possibility of failure of head unification, leading to a trivial lower bound of 0. In this paper, we show how, given type and mode information about procedures in a logic program, it is possible to (semi-automatically) derive non-trivial lower bounds on their computational costs. We also discuss the cost analysis for the special and frequent case of divide-and-conquer programs and show how —as a pragmatic short-term solution —it may be possible to obtain useful results simply by identifying and treating divide-and-conquer programs specially. Finally, we present experimental results from a preliminary implementation of the proposed approach, applied to automatic parallel task size control.

Keywords: Cost Analysis, Lower Bound Estimation, Granularity Control, Parallelism.

1 Introduction

It is generally recognized that information about the runtime cost of computations can be useful for a variety of applications. For example, it is useful for granularity control, i.e., dynamic control of thread creation in parallel implementations of logic and functional languages [14, 4, 16, 9, 15, 12], and for query optimization in deductive databases [5]. In the context of logic programming, the work on cost estimation has generally focused on upper bound cost analyses [6]. However, in many cases one would prefer to work with lower bounds instead. As an example, consider a distributed memory implementation of Prolog: suppose that the work involved in spawning a task on a remote processor takes 1000 instructions, and that we infer that a particular procedure call in a program will execute no more than 5000 instructions. This suggests that it *may be* worth executing this call on a remote processor, but provides no assurance that doing so will not actually produce a performance degradation relative to a sequential execution (the call might terminate after executing only a small number of instructions). On the other hand, if we know that a call will execute at least 5000 instructions, we can be assured that spawning a task on a remote processor to execute this call is worthwhile. Thus, while upper bound cost information is better than no information at all, lower bounds may be more useful than upper bounds.

The biggest problem with the inference of lower bounds on the computational cost of logic programs is the possibility of failure. Any attempt to infer lower bounds has to contend with the possibility that a goal may fail during head unification, yielding a trivial lower bound of 0. An obvious solution would be to try and rule out "bad" argument values by considering the types of predicates. However, most existing type analyses provide upper approximations, in the sense that the type of a predicate is a superset of the set of argument values that are actually encountered at runtime. Unfortunately, straightforward attempts to address this issue, for example by trying to infer lower approximations to the calling types of predicates, fail to yield nontrivial lower bounds for most cases.

In [3], we showed how, given mode and (upper approximation) type information, we can detect procedures and goals that can be guaranteed to not fail. Our technique is based on an intuitively very simple notion, that of a (set of) tests "covering" the type of a variable. We showed that the problem of determining a covering is undecidable in general, and is co-NP-hard even if we have only finite types and simple equality tests. We then gave an algorithm for checking whether a set of tests covers a type, that is complete for *tuple-distributive regular types*, sound, and efficient in practice.

Based on this information, we showed how to identify goals and procedures that can be guaranteed to not fail at runtime. Note that this information is interesting in its own right, in the context of program transformations (for example, we may want to execute possibly-failing goals ahead of non-failing goals where possible) and in systems that exploit speculative parallelism.

The main contributions of this paper are as follows: (i) we show how non-failure information can be used to infer lower bounds on the computational costs of goals; (ii) discuss how to bound the chromatic polynomial of a graph from below, and thereby show how to infer lower bounds on the number of solutions a predicate can generate (this information is useful, for example, for estimating communication costs in distributed-memory implementations); (iii) show how information about the number of solutions computed can be used to improve lower bound estimates when all solutions to a goal are required; and (iv) show how to obtain improved lower bound estimates for a simple but common class of divide-and-conquer programs. We discuss the application of our ideas to granularity control for parallel programs: in this case, the use of lower bound cost estimates guarantees that no slowdowns will occur, even in systems with significant overheads associated with parallel execution. Our ideas have been implemented and the resulting lower bound cost estimates, given in Section 7, can be seen to be quite precise, especially for an automatic analysis tool. Experimental results with granularity control using lower bound cost estimates indicate that significant performance improvements can be attained using our approach.

Despite a suggestive similarity in names, our work is quite different from Basin and Ganzinger's work on complexity analysis based on ordered resolution [1]. They consider resolution based on a well-founded total ordering on ground atoms, and use this to examine the complexity of determining, given a set of Horn clauses N and a ground Horn clause C, whether $N \models C$. Our work, by contrast, is based on an operational formulation of logic program execution that is not restricted to ground queries (or, for that matter, to Horn programs, since it is easy to handle features such as cuts and negation by failure). Because operational aspects of program execution are modeled more accurately in our approach, the results obtained are considerably more precise.

2 Lower-Bound Cost Analysis: The One-Solution Case

If only one solution is required of any computation, it suffices to know whether a computation will generate at least one solution, i.e., will not fail. Assuming that this information is available, for example by using the technique mentioned in the previous section, cost analysis for a particular predicate can then proceed as follows:

1. We first determine the relative sizes of variable bindings at different program points in a clause by computing lower bounds on output argument sizes as functions of input argument sizes. This is done by solving (or estimating lower bound solutions to) the resulting difference equations: the approach is very similar to that discussed in [6], the only difference being that whereas [6] estimated upper bounds on argument sizes using the *max* function across the output sizes of different clauses in a cluster, we use the *min* function across clauses to estimate lower bounds on argument sizes.

2. The (lower bound) computational cost of a clause is then expressed as a function of the input argument size, in terms of the costs of the body literals in that clause.

   Consider a clause $C \equiv$ 'H :− $B_1, \ldots, B_m$'. Let n be the r−tuple which represents the sizes of the r input arguments for the head of the clause, and let (lower bounds on) the input argument sizes for the body literals $B_1, \ldots, B_m$ be $\phi_1(n), \ldots, \phi_m(n)$ respectively. Assume that the cost of head unification and tests for this clause is at least $h(n)$, and let $Cost_{B_i}(x)$ denote a lower bound on the cost of the body literal B_i. Then, if B_k is the rightmost body literal that is guaranteed to not fail, the following gives a lower bound on the cost $Cost_C(n)$ of the clause C on an input of size n:

   $$h(n) + \sum_{i=1}^{k} Cost_{B_i}(\phi_i(n)) \;\leq\; Cost_C(n).$$

   If the clause C corresponds to a non-failing predicate, then we take $k = m$.

3. A lower bound on the cost $Cost_p(n)$ of a predicate p on an input of size n is then given by

$$\min\{Cost_C(n) \mid C \text{ is a clause defining } p\} \;\leq\; Cost_p(n).$$

As discussed in [6], recursion is handled by expressing the cost of recursive goals symbolically as a function of the input size. From this, we can obtain a set of difference equations that can be solved (or approximated) to obtain a lower bound on the cost of a predicate in terms of the input size.

Given a predicate defined by m clauses $C_1, \ldots, C_m$, we can improve the precision of this analysis by noting that clause C_i will be tried only if clauses $C_1, \ldots, C_{i-1}$ fail to yield a solution. For an input of size n, let $\delta_i(n)$ denote the least amount of work necessary to determine that clauses $C_1, \ldots, C_{i-1}$ will not yield a solution and that C_i must be tried: the function δ_i obviously has to take into account the type and cost of the indexing scheme being used in the underlying implementation. In this case, the lower bound for p can be improved to:

$$\min\{Cost_{C_i}(n) + \delta_i(n) \mid 1 \leq i \leq m\} \;\leq\; Cost_p(n).$$

The pruning operator can also be taken into account, so that clauses which are after the first clause, say C_i, which has a non-failing sequence of literals just before the cut, are ignored, and the lower bound on the cost of the predicate is then the minimum of the costs of the clauses preceding the clause C_i and this clause itself.

3 Lower-Bound Cost Analysis: All Solutions

In many applications, it is reasonable to assume that all solutions are required. For example, in a distributed memory implementation of a logic programming system, the cost of sending or receiving a message is likely to be high enough that it makes sense for a remote computation to compute all the solutions to a query and return them in a single message instead of sending a large number of messages, each containing a single solution. For such cases, estimates of the computational cost of a goal can be improved greatly if we have lower bounds on the number of solutions—indeed, as the example of a distributed memory system suggests, in some cases the number of solutions may itself be a reasonable measure of cost.

If we obtain lower bounds on the number of solutions that can be generated by the literals in a clause (this problem is addressed in next section), we can use this information to improve lower bound cost estimates for the case where all solutions to a predicate are required. Consider a clause '$p(\bar{x})\ :-\ B_1, \ldots, B_n$' where B_k is the rightmost literal that is guaranteed to not fail. Let the input argument size for the head of the clause be n, and let (lower bounds on) the input argument sizes for the body literals $B_1, \ldots, B_m$ be $\phi_1(n), \ldots, \phi_m(n)$ respectively. Assume that the cost of head unification and tests for this clause is at least $h(n)$, and let $Cost_{B_i}(x)$ denote a lower bound on the cost of the body literal B_i. Now consider a body literal B_j, where $1 \leq j \leq k+1$, i.e., all the predecessors of B_j are guaranteed to not fail. The number of times B_j will be executed is given by the total number of solutions generated by its predecessors, i.e., the literals $B_1, \ldots, B_{j-1}$. Let this number be denoted by N_j : we can estimate N_j using Theorem 5.1 (or extensions thereof), e.g., by considering a clause whose body consists of the literals $B_1, \ldots, B_{j-1}$, and where the output variables in the head are given by $vars(B_1, \ldots, B_{j-1}) \cap vars(B_j, \ldots, B_n)$. Assume that the cost of head unification and tests for this clause is at least $h(n)$, and let $Cost_{B_i}(x)$ denote a lower bound on the cost of the body literal B_j. Then, a lower bound on the execution cost of the clause to obtain all solutions is given by

$$h(n) + \sum_{i=1}^{k}(N_i \times Cost_{B_i}(\phi_i(n)) \;\leq\; Cost_C(n).$$

4 Number of Solutions: The Single-Clause Case

In this section we address the problem of estimating lower bounds on the number of solutions which a predicate can generate.

4.1 Simple Conditions for Lower Bound Estimation

It is tempting to try and estimate a lower bound on the number of solutions generated by a clause '$H \ :- \ B_1, \ldots, B_n$' from lower bounds on the number of solutions generated by each of the body literals B_i, possibly using techniques analogous to those used in [6] for the estimation of upper bounds on the number of solutions. Unfortunately, this does not work. For example, given a clause '$p(X) \ :- \ q(X), r(X)$', where X is an output variable, and assuming that q and r generate n_q and n_r bindings, respectively, for X, then $min(n_q, n_r)$ is not a lower bound on the number of solutions the clause can generate. To see this, consider the situation where q can bind X to either a or b, while r can bind X to either b or c: thus, $min(n_q, n_r) = min(2, 2) = 2$, but the number of solutions for the clause is 1.

The following gives a simple sufficient condition for estimating a lower bound on the number of solutions generated by a clause.

Theorem 4.1 *Let $x_1, \ldots, x_m$ be distinct unaliased output variables in the head of a clause such that each of the x_i occurs at most once in the body of the clause, and x_i and x_j do not occur in the same body literal for $i \neq j$. If n_i is a lower bound on the number of bindings that can be generated for x_i by the clause body, then $\prod_{i=1}^{m} n_i$ is a lower bound on the number of solutions that can be generated by the clause.*

This result can be generalized in various ways: we do not pursue them here due to space constraints. The utility of this theorem is shown in Example 5.1.

4.2 Handling Equality and Disequality Constraints

This section presents a simple algorithm for computing a lower bound on the number of solutions for predicates which can be "unfolded" into a conjunction of binary equality and disequality constraints on a set of variables. The constraints are in the form of $X = Y$ or $X \neq Y$ for any two variables X and Y. The types of the variables in a predicate are assumed to be the same and to be given as a finite set of atoms. The problem of computing the number of bindings that satisfy a set of binary equality and disequality constraints on a set of variables with the same type can be transformed into the problem of computing the *chromatic polynomial* of a graph G, denoted by $C(G, k)$, which is a polynomial in k and represents the number of different ways G can be colored by using no more than k colors (see [6]).

Unfortunately, the problem of computing the chromatic polynomial of a graph is NP-hard, because the problem of k-colorability of a graph G is equivalent to the problem of deciding whether $C(G, k) > 0$ and the problem of graph k-colorability is NP-complete [11]. Therefore, we will develop an approximation algorithm to compute a lower bound on the chromatic polynomial of a graph. The basic idea is to start with a subgraph that consists of only a single vertex of the graph, then repeatedly build larger and larger subgraphs by adding a vertex at a time into the previous subgraph. When a vertex is added, the edges connecting that vertex to vertices in the previous subgraph are also added. At each iteration, a lower bound on the number of ways of coloring the newly added vertex can be determined by the number of edges accompanied with the vertex. Accordingly, a lower bound on the chromatic

Let $G = (V, E)$ be a graph of order n. The algorithm proceeds as follows:

begin
  compute the degree for each vertex in V;
  generate an ordering $\omega = v_1, \ldots, v_n$ of V by sorting the vertices in
    decreasing order on their degrees using the radix sort;
  $C(G, k) := k$;
  $G_1 := (\{v_1\}, \emptyset)$;
  for $i := 2$ **to** n **do**
    compute the order $|G'_i|$ of the interfacing subgraph G'_i;
    $C(G, k) := C(G, k) \times (k - |G'_i|)$;
    construct the accumulating subgraph G_i;
  od
end

Figure 1: An approximation algorithm for computing the chromatic polynomial of a graph

polynomial for the corresponding subgraph can be determined using the bound on the polynomial for the previous subgraph and the bound on the number of ways of coloring the newly added vertex.

We now describe the algorithm more formally. The *order* of a graph $G = (V, E)$, denoted by $|G|$, is the number of vertices in V. Let G be a graph of order n. Suppose $\omega = v_1, \ldots, v_n$ is an ordering of V. We define two sequences of subgraphs of G according to ω. The first is a sequence of subgraphs $G_1, \ldots, G_n$, called *accumulating subgraphs*, where $G_i = (V_i, E_i)$, $V_i = \{v_1, \ldots, v_i\}$, and E_i is the set of edges of G that join the vertices of V_i, for $1 \le i \le n$, The second is a sequence of subgraphs $G'_2, \ldots, G'_n$, called *interfacing subgraphs*, where $G'_i = (V'_i, E'_i)$, V'_i is the set of vertices of G_{i-1} that are adjacent to vertex v_i, and E'_i is the set of edges of G_{i-1} that join the vertices of V'_i, for $2 \le i \le n$.

The algorithm for computing the chromatic polynomial of a graph, based on the construction of accumulating subgraphs and interfacing subgraphs, is shown in Figure 1. This algorithm constructs the accumulating subgraphs according to an ordering of the set of vertices. At each iteration, the number of ways of coloring the newly added vertex is computed based on the order of the corresponding interfacing subgraph.

Theorem 4.2 *Let $G = (V, E)$ be a graph of order n and ω be an ordering of V. Suppose the interfacing subgraphs of G corresponding to ω are $G'_2, \ldots, G'_n$. Then:*

$$k \prod_{i=2}^{n} (k - |G'_i|) \le C(G, k).$$

The proof of this theorem is given in [13]; we omit it here due to space constraints. Since the bound obtained from this may depend on the ordering chosen for the vertices in the graph, we use a heuristic to find a "good" ordering. The intuition behind the heuristic is that if the maximum order of the interfacing subgraphs is smaller, then we can get a nontrivial lower bound ($\neq 0$) on $C(G, k)$ for more values of k. Therefore, we use the ordering that sorts the vertices in the decreasing order on the degrees of vertices.

Let the graph under consideration have n vertices and m edges. First, the computation for the degrees of vertices in the graph can be performed in $O(n + m)$.

Second, since the degrees of vertices in the graph are at most $n-1$, we can sort the vertices using radix sort in $O(n)$. Third, the total cost for the construction of accumulating subgraphs G_i, $i \leq i \leq n$, is $O(n+m)$ because each edge in the graph is examined only twice. Finally, since only the orders of the interfacing subgraphs are needed to compute the chromatic polynomial, it is not necessary to construct the interfacing graphs. The orders of the interfacing subgraphs can be obtained as a by-product of constructing the accumulating graphs. Therefore, the complexity of the whole algorithm is $O(n+m)$.

5 Number of Solutions: Multiple Clauses

The previous section discussed the estimation of lower bounds on the number of solutions computed by a single clause. In this section we discuss how we can estimate the number of solutions for a group of clauses.

Theorem 5.1 *Consider a set of clauses $S = \{C_1, \ldots, C_n\}$ that all have the same head unification and tests. If n_i is a lower bound on the number of solutions generated by C_i, $1 \leq i \leq n$, then $\sum_{i=1}^{n} n_i$ is a lower bound on the total number of (not necessarily distinct) solutions generated by the set of clauses S.*

The restrictions in this theorem can be relaxed in various ways: we do not pursue this here due to space constraints. We can use the result above to estimate a lower bound on the number of solutions generated by a predicate for an input of size n as follows: partition the clauses for the predicate into clusters such that the clauses in each cluster have the same head unification and tests, so that Theorem 5.1 is applicable, and compute lower bound estimates of the number of solutions for each cluster. Then, if a number of different clusters—say, clusters $C_1, \ldots, C_k$, with number of solutions at least $n_1, \ldots, n_k$ respectively, may be applicable to an input of size n, then the number of solutions overall for an input of size n is given by $min(n_1, \ldots, n_k)$. The utility of this approach is illustrated by the following example.

Example 5.1 Consider the following predicate to generate all subsets of a set represented as a list:

```
subset([], X) :- X = [].
subset([H|L], X) :- X = [H|X1], subset(L, X1).
subset([H|L], X) :- subset(L, X).
```

As discussed in Section 2, recursion is handled by initially using a symbolic representation to set up difference equations, and then solving, or estimating solutions to, these equations. In this case, let (a lower bound on) the number of solutions computed by `subset/2` on an input of size n be symbolically represented by $S(n)$. The first clause for the predicate yields the equation

$$S(0) = 1.$$

From Theorem 4.1, on an input of size n, $n > 0$, the second and third clauses each yield at least $S(n-1)$ solutions. Since they have the same head unification and tests, Theorem 5.1 is applicable, and the number of solutions given by these two clauses taken together is therefore at least $S(n-1) + S(n-1) = 2S(n-1)$. Thus, we have the equation

$$S(n) = 2S(n-1).$$

These difference equations can be solved to get the lower bound $S(n) = 2^n$ on the number of solutions computed by this predicate on an input of size n. $\square$

6 Cost Estimation for Divide-and-Conquer Programs

A significant shortcoming of the approach to cost estimation presented is its loss in precision in the presence of divide-and-conquer programs in which the sizes of the output arguments of the "divide" predicates are dependent. In the familiar quicksort program (see Example 6.1), for example, since either of the outputs of the partition predicate can be the empty list, the straightforward approach computes lower bounds under the assumption that both output can *simultaneously* be the empty list, and thereby significantly underestimates the cost of the program. In some sense, the reason for this loss of precision is that the approach outlined so far is essentially an independent attributes analysis [10]. However, even if we came up with a relational attributes analysis that kept track of relationships between the sizes of different output arguments of a predicate, it is not at all obvious how we might, systematically and from first principles, use this information to improve our lower bound cost estimates. For the quicksort program, for example, if the input list has length n, then the two output lists of the partition predicate have lengths m and $n - m - 1$ for some m, $0 \leq m < n$. The resulting cost equation for the recursive clause is of the form

$$C(n) = C(m) + C(n - m - 1) + \ldots \qquad (0 \leq m \leq n - 1)$$

In order to determine a worst-case lower bound solution to this equation we need to determine the value of m that minimizes the function $C(n)$, and doing this automatically, when we don't even know what $C(n)$ looks like, seems nontrivial. As a pragmatic solution, we argue that it may be possible to get quite useful results simply by identifying and treating common classes of divide-and-conquer programs specially.

In many of these programs, the sum of the sizes of the input for the "divide" predicates in the clause body is equal to the size of the input in the clause head minus some constant. This size relationship can be derived in some cases by the approach presented in [6]. However, this is not possible in other cases, since in this approach the size of each output argument is treated as a function only of the input sizes, independently of the sizes of other output arguments, and, as a result, relationships between the sizes of different output arguments are lost (consider for example the `partition/4` predicate defined in example 6.1). Although the analysis does not break down for these cases, it can lose precision. A possible solution to improve precision is to use one of the recently proposed approaches for inferring size relationships for this class of programs [2, 7].

Assuming that we have the mentioned size relationship for these programs, in the cost analysis phase we obtain an expression of the form:

$y(0) = C$,
$y(n) = y(n - 1 - k) + y(k) + g(n)$ for $n > 0$, where k is an arbitrary value such that $0 \leq k \leq n - 1$, C is a constant and $g(n)$ is any function.

where $y(n)$ denotes the cost of the divide-and-conquer predicate for an input of size n and $g(n)$ is the cost of the part of a clause body which does not contain any call to the divide-and-conquer predicate.

For each particular computation, we obtain a succession of values for k. Each succession of values for k yields a value for $y(n)$.

In the following we discuss how we can compute lower/upper bounds for expressions such as that for $\texttt{Cost}_{\texttt{qsort}}(n)$ in Example 6.1. Consider the expression:

$y(0) = C$,
$y(n) = y(n-1-k) + y(k)$ for $n > 0$, where k is an arbitrary value such that $0 \leq k \leq n-1$ and C is a constant.

A *computation tree* for such an expression is a tree in which each non-terminal node is labeled with $y(n)$, $n > 0$, and has two children $y(n-1-k)$ and $y(k)$ (left- and right-hand-side respectively), where k is an arbitrary value such that $0 \leq k \leq n-1$. Terminal nodes are labeled with $y(0)$ and have no children. Assume that we construct a tree for $y(n)$ following a depth-first traversal. In each non-terminal node, we (arbitrarily) chose a value for k such that $0 \leq k \leq n-1$. We say that the *computation succession* of the tree is the succession of values that have been chosen for k in chronological order, as the tree construction proceeds.

Lemma 6.1 *Any computation tree corresponding to the expression:*

$y(0) = C$,
$y(n) = y(n-1-k) + y(k)$ *for* $n > 0$, *where* k *is an arbitrary value such that* $0 \leq k \leq n-1$ *and* C *is a constant,*

has $n+1$ *terminal nodes and* n *non-terminal nodes.*

Proof By induction on n. For $n = 0$ the theorem holds trivially. Let us assume that the theorem holds for all m such that $0 \leq m \leq n$, then, we can prove that for all m such that $0 \leq m \leq n+1$ the theorem also holds by reasoning as follows: we have that $y(n+1) = y(n-k) + y(k)$, where k is an arbitrary value such that $0 \leq k \leq n$. Since $0 \leq k \leq n$, we also have that $0 \leq n-k \leq n$, and, by induction hypothesis, the number of terminal nodes in any computation tree of $y(n-k)$ (respectively $y(k)$) is $n-k+1$ (respectively $k+1$). The number of terminal nodes in any computation tree of $y(n+1)$ is the sum of the number of terminal nodes in the children of the node labeled with $y(n+1)$, i.e. $(n-k+1)+(k+1) = n+2$. Also, the number of non-terminal nodes in any computation tree of $y(n-k)$ (respectively $y(k)$) is $n-k$ (respectively k). The number of non-terminal nodes of any computation tree of $y(n+1)$ is the sum of the number of non-terminal nodes of the children of the node labeled with $y(n+1)$ plus one (the node $y(n+1)$ itself, since it is non-terminal) i.e. $1+(n-k)+k = n+1$. ∎

Theorem 6.2 *For any computation tree corresponding to the expression:*

$y(0) = C$,
$y(n) = y(n-1-k) + y(k)$ *for* $n > 0$, *where* k *is an arbitrary value such that* $0 \leq k \leq n-1$ *and* C *is a constant,*

it holds that $y(n) = (n+1) \times C$.

Proof By Lemma 6.1, any computation tree has $n+1$ terminal nodes labeled with $y(0)$ and the evaluation of each of these terminal nodes is C. ∎

Theorem 6.3 *Given the expression:*

$y(0) = C$,
$y(n) = y(n-1-k) + y(k) + g(k)$ *for* $n > 0$, *where* k *is an arbitrary value such that* $0 \leq k \leq n-1$, C *is a constant and* $g(k)$ *a function,*

for any computation tree corresponding to it, it holds that $y(n) = (n+1) \times C + \sum_{i=1}^{n} g(k_i)$, *where* $\{k_i\}_{i=1}^{n}$ *is the computation succession of the tree.*

Proof By Lemma 6.1, any computation tree has $n+1$ terminal nodes and n non-terminal nodes. The evaluation of each terminal node yields the value C and each time a non-terminal node i is evaluated, $g(k_i)$ is added. ■

In order to minimize (respectively maximize) $y(n)$ we can find a succession $\{k_i\}_{i=1}^{n}$ that minimizes (respectively maximizes) $\sum_{i=1}^{n} g(k_i)$. This is easy when $g(k)$ is a monotonic function, as the following corollary shows.

Corollary 6.1 *Given the expression:*

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k) + g(k)$ *for* $n > 0$, *where* k *is an arbitrary value such that* $0 \leq k \leq n-1$, C *is a constant and* $g(k)$ *an increasing monotonic function,*

Then, the succession $\{k_i\}_{i=1}^{n}$, *where* $k_i = 0$ *(respectively* $k_i = n-1$*) for all* $1 \leq i \leq n$ *gives the minimum (respectively maximum) value for* $y(n)$ *of all computation trees.*

Proof It follows from Theorem 6.3 and from the fact that $g(k)$ is an increasing monotonic function. ■

It follows from Corollary 6.1 that the solution of the difference equation (obtained by replacing k by 0):

$$y(0) = C,$$
$$y(n) = y(n-1) + y(0) + g(0) \text{ for } n > 0,$$

i.e. $(n+1) \times C + n \times g(0)$ is the minimum of $y(n)$, and the solution of the difference equation:

$$y(0) = C,$$
$$y(n) = y(0) + y(n-1) + g(n-1) \text{ for } n > 0,$$

i.e. $(n+1) \times C + n \times g(n-1)$ is the maximum of $y(n)$.

Note that we can replace $g(k)$ by any lower/upper bound on it to compute a lower/upper bound on $y(n)$. We can also take any lower/upper bound on each $g(k_i)$. For example, if $g(k)$ is an increasing monotonic function then $g(k_i) \leq g(n-1)$ and $g(k_i) \geq g(0)$ for $1 \leq i \leq n$, thus, $y(n) \leq (n+1) \times C + n \times g(n-1)$ and $y(n) \geq (n+1) \times C + n \times g(0)$.

Let's now assume that the function g depends on n and k:

Corollary 6.2 *Given the expression:*

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k) + g(n,k)$ *for* $n > 0$, *where* k *is an arbitrary value such that* $0 \leq k \leq n-1$, C *is a constant and* $g(n,k)$ *a function.*

Then, the solution of the difference equation:

$$f(0) = C,$$
$$f(n) = f(n-1) + C + L \text{ for } n > 0,$$

where L *is a lower/upper bound on* $g(n,k)$, *is a lower/upper bound on* $y(n)$ *for all* $n \geq 0$ *and for any computation tree corresponding to* $y(n)$. *In particular, if* $g(n,k)$ *is an increasing monotonic function, then* $L \equiv g(1,0)$ *(respectively* $L \equiv g(n, n-1)$*) is a lower (respectively upper) bound on* $g(n,k)$.

Example 6.1 Let us see how, using the described approach for divide-and-conquer programs, the lower-bound cost analysis can be improved. We first consider the analysis without the incorporation of the optimization, and then we compare with the result obtained when the optimization is used.

Consider the predicate qsort/2 defined as follows:

```
qsort([], []).
qsort([First|L1], L2) :-
     partition(L1, First, Ls, Lg),
     qsort(Ls, Ls2), qsort(Lg, Lg2),
     append(Ls2, [First|Lg2], L2).

partition([], F, [], []).
partition([X|Y], F, [X|Y1], Y2) :-
        X =< F,
        partition(Y, F, Y1, Y2).
partition([X|Y], F, Y1, [X|Y2]) :-
        X > F,
        partition(Y, F, Y1, Y2).

append([], L, L).
append([H|L], L1, [H|R]) :- append(L, L1, R).
```

Let $\mathtt{Cost}_{\mathtt{p}}(n)$ denote the cost (number of resolution steps) of a call to predicate p with an input of size n (in this example, the size measure used for all predicates is list length [6]). The estimation of cost functions proceeds in a "bottom-up" way as follows:

The difference equation obtained for append/3 is:

$\mathtt{Cost}_{\mathtt{append}}(0, m) = 1$ (the cost of head unification),
$\mathtt{Cost}_{\mathtt{append}}(n, m) = 1 + \mathtt{Cost}_{\mathtt{append}}(n-1, m)$.

where $\mathtt{Cost}_{\mathtt{append}}(n, m)$ is the cost of a call to append/3 with input lists of lengths n and m (first and second argument, respectively). The solution to this equation is: $\mathtt{Cost}_{\mathtt{append}}(n, m) = n + 1$. Since this function depends only on n, we use the function $\mathtt{Cost}_{\mathtt{append}}(n)$ instead.

The difference equation for partition/4 is:

$\mathtt{Cost}_{\mathtt{partition}}(0) = 1$ (the cost of head unification),
$\mathtt{Cost}_{\mathtt{partition}}(n) = 1 + \mathtt{Cost}_{\mathtt{partition}}(n-1)$.

where $\mathtt{Cost}_{\mathtt{partition}}(n)$ gives the cost of a call to partition/4 with an input list (first argument) of length n. The solution to this equation is: $\mathtt{Cost}_{\mathtt{partition}}(n) = n + 1$. For qsort/2, we have:

$\mathtt{Cost}_{\mathtt{qsort}}(0) = 1$ (the cost of head unification),
$\mathtt{Cost}_{\mathtt{qsort}}(n) = 1 + \mathtt{Cost}_{\mathtt{partition}}(n-1) + 2 \times \mathtt{Cost}_{\mathtt{qsort}}(0) + \mathtt{Cost}_{\mathtt{append}}(0)$

because the computed lower bound for the size of the input to the calls to qsort and append is 0. Thus, the cost function for qsort/2 is given by:

$\mathtt{Cost}_{\mathtt{qsort}}(0) = 1$,
$\mathtt{Cost}_{\mathtt{qsort}}(n) = n + 4$, for $n > 0$.

Now, we use the described approach for divide-and-conquer programs. Assume that we use the expression:

$\mathtt{Cost}_{\mathtt{qsort}}(0) = 1,$
$\mathtt{Cost}_{\mathtt{qsort}}(n) = 1 + \mathtt{Cost}_{\mathtt{partition}}(n-1) + \mathtt{Cost}_{\mathtt{qsort}}(k)$
$\quad + \mathtt{Cost}_{\mathtt{qsort}}(n-1-k) + \mathtt{Cost}_{\mathtt{append}}(k)$, for $0 \leq k \leq n-1$ and $n > 0$.

Replacing values, we obtain:

$\mathtt{Cost}_{\mathtt{qsort}}(n) = n + k + 2 + \mathtt{Cost}_{\mathtt{qsort}}(k) + \mathtt{Cost}_{\mathtt{qsort}}(n-1-k)$,
for $0 \leq k \leq n-1$.

According to Corollary 6.2, by giving to n and k the minimum possible value, i.e. 1 and 0 respectively, we have that $n + k + 2 \geq 3$, and thus we replace $n + k + 2$ by 3 in order to obtain a lower bound on the former expression, which yields:

$\mathtt{Cost}_{\mathtt{qsort}}(n) = 3 + \mathtt{Cost}_{\mathtt{qsort}}(k) + \mathtt{Cost}_{\mathtt{qsort}}(n-1-k)$, for $0 \leq k \leq n-1$.

which is equivalent to the difference equation:

$\mathtt{Cost}_{\mathtt{qsort}}(n) = 3 + 1 + \mathtt{Cost}_{\mathtt{qsort}}(n-1)$, for $n > 0$.

The solution of this equation is $\mathtt{Cost}_{\mathtt{qsort}}(n) = 4n + 1$, which is an improvement on the former lower bound. □

The previous results can be easily generalized to cover multiple recursive divide-and-conquer programs and programs where the sum of the sizes of the input for the "divide" predicates in the clause body is equal to the size of the input in the clause head minus some constant which is not necessarily 1.

7 Implementation

We have implemented a prototype of a lower bound size/cost analyzer, by recoding the version of CASLOG [6] currently integrated in the CIAO system [8]. The analysis is fully automatic, and only requires type information for the program entry point. Types, modes and size measures are automatically inferred by the system. Table 1 shows some accuracy and efficiency results of the lower bound cost analyzer. The second column of the table shows the cost function (which depends on the size of the input arguments) inferred by the analysis. T_{tms} is the time required by the type, mode, and size measure analysis (SPARCstation 10, 55MHz, 64Mbytes of memory), T_{nf} the time required by the non-failure analysis, and T_s and T_{ca} are the time required by the size and cost analysis respectively. **Total** is the total analysis time ($Total = T_{tms} + T_{nf} + T_s + T_{ca}$). All times are given in milliseconds.

8 Application to Automatic Parallelization

As briefly mentioned in the introduction, one of the most attractive applications of lower bound cost analysis is implementing granularity control in parallelizing compilers, an issue on which we expand in this section. Parallel execution of a task incurs various overheads, e.g. overheads associated with process creation and scheduling, the possible migration of tasks to remote processors and the associated communication overheads, etc. In general, a goal should not be a candidate for parallel execution if its granularity, i.e., the "work available" underneath it, is less than the work necessary to create a separate task for that goal. While the overheads for spawning goals in parallel in some architectures are small (e.g. in small shared memory multiprocessors), in many other architectures (e.g. distributed memory multiprocessors,

Program	Cost function	T_{tms}	T_{nf}	T_s	T_{ca}	Total
fibonacci	$\lambda x.1.447 \times 1.618^x + 0.552 \times (-0.618)^x - 1$	90	10	20	20	140
hanoi	$\lambda x.x2^x + 2^{x-1} - 2$	430	30	60	60	580
qsort	$\lambda x.4x + 1$	420	50	70	50	590
nreverse	$\lambda x.0.5x^2 + 1.5x + 1$	220	20	30	30	300
mmatrix	$\lambda \langle x, y \rangle.2xy + 2x + 1$	350	90	90	90	620
deriv	$\lambda x.x$	1010	80	170	120	1,380
addpoly	$\lambda \langle x, y \rangle x + 1$	220	70	40	30	360
append	$\lambda x.x + 1$	100	20	10	10	140
partition	$\lambda x.x + 1$	175	30	30	20	255
substitute	$\lambda \langle x, y, z \rangle.x$	70	50	110	100	330
intersection	$\lambda \langle x, y \rangle.x + 1$	150	130	20	30	260
difference	$\lambda \langle x, y \rangle.x + 1$	140	90	20	40	290

Table 1: Accuracy and efficiency of the lower bound cost analysis

workstation "farms", etc.) such overheads can be very significant. The consequence is that automatic parallelization cannot be achieved realistically in the latter without some form of granularity control.

As we have already pointed out, all of the previous work that we know of in this context involves estimating upper bounds on the cost of goals. Given a program that is already parallelized, upper bound cost information can be used to produce a program in which some of the parallel goals are forced to run sequentially, in such a way that the resulting execution can be guaranteed to be faster (or, at least, no slower) than that of the original parallel program. However, the problem faced by parallelizing compilers is in fact exactly the converse: what needs to be guaranteed is that the parallel execution will be more efficient than that of the original sequential program, rather than the other way around. This type of granularity control can be performed applying essentially the same techniques as when using upper bound information [14], but, of course, *lower bound* information on the cost of each goal is required instead, and this is where the techniques proposed in this paper fit in. The usefulness of lower bounds was already clear when the work presented in [4] was developed, but the determination of useful lower bounds was deemed too difficult at the time. Using lower bounds allows obtaining *guaranteed speedups* (or, at least, ensuring that no *slow-downs* will occur) from automatic parallelization, even in architectures for which parallel execution involves a significant overhead. We know of no other approach which can achieve this.

We have interfaced the cost analysis stage (see Section 7) with the granularity control system described in [14] (which is also integrated in the CIAO system as another stage, and which includes an annotator which transforms programs to perform granularity control). The result is a complete program parallelizer with (lower bound cost based) granularity control. Since our objective herein is simply to study the usefulness of the lower bound estimates produced, only a very simple granularity control strategy has been selected: goals are always executed in parallel provided their grain sizes are estimated to be greater than a given fixed threshold, which is a constant for all programs. Also, the versions of the programs that perform granularity control are simple source-to-source transformations which add granularity control tests to the original versions. A discussion of more advanced strategies that include variable thresholds (which depend on parameters such as data transfer cost, number of processor, system load, etc.), lower level transformations, and performing goal groupings to increase granularity can be found in [14].

programs	seq	ngc	gclb(175)	gclb(959)
mmatrix(100)	52.389	74.760 (0.70)	29.040 (1.80)	27.981 (1.87)
mmatrix(50)	6.469	5.978 (1.08)	3.378 (1.92)	3.758 (1.72)
fib(19)	0.757	1.458 (0.52)	0.128 (5.93)	0.103 (7.32)
hanoi(13)	1.442	1.464 (0.98)	0.677 (2.13)	0.619 (2.33)
qsort(1000)	0.475	0.414 (1.15)	0.230 (2.06)	0.314 (1.51)
qsort(3000)	4.142	2.423 (1.71)	1.094 (3.79)	1.575 (2.63)

Table 2: Granularity control results for benchmarks on ECL^{i}PS^e.

We have performed some preliminary experiments in which a series of benchmarks have been parallelized automatically, with the compiler option corresponding to inclusion of granularity control both enabled and disabled. The resulting programs have been executed on the ECL^{i}PS^e system using 10 workers, and running on a SUN SPARC 2000 SERVER with 10 processors. We have chosen this system, which implements and-parallelism on top of or-parallelism, because it has considerably greater parallel task execution overhead than systems which implement and-parallelism natively (such as, for example, the &-Prolog engine used in the CIAO system). As a result, this system offers an interesting challenge – it proved very difficult to achieve and-parallel speedups on it automatically with previous parallelizers.

Table 2 presents the results. It shows wall-clock execution times in seconds. Results are given for the sequential execution (**seq**), the parallel execution without granularity control (**ngc**), and the versions which perform granularity control (**gclb(175)** and **gclb(959)**). The two numbers correspond to two different choices of threshold, and illustrate the comparatively low sensitivity of the results to the choice of this parameter that we have observed.

The results of the experiments appear promising, in the sense that the granularity control does improve speedups in practice, in a quite challenging situation. On systems with higher overheads, such as distributed systems, the benefits can be much larger, although it may be difficult to achieve actual speedups in some cases (i.e., given high enough overheads, the result of the granularity analysis can often be simply a sequential program). In any case, we believe that it is possible to improve these results significantly by using more sophisticated control strategies, as mentioned above.

Acknowledgements

The work of S. Debray was supported in part by the National Science Foundation under grant CCR-9123520. The work of M. Hermenegildo and P. López-García was supported in part by ESPRIT project LTR 22532 "DiSCiPl" and CICYT proyect number TIC96-1012-C02-01. The work of N.-W. Lin was supported in part by the NSC of ROC under grant NSC83-0408-E-194-010.

References

[1] D. Basin and H. Ganzinger. Complexity Analysis based on Ordered Resolution *Proc. 11th. IEEE Symposium on Logic in Computer Science*, 1996.

[2] F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). *Proc. 6th International Workshop on Logic Program Synthesis and Transformation*, Stockholm University/Royal Intitute of Technology, 1996, pp. 134–153.

[3] S. Debray, P. López García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Leuven, Belgium, June 1997. MIT Press, Cambridge, MA.

[4] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[5] S.K. Debray and N.-W. Lin. Static estimation of query sizes in horn programs. In *Third International Conference on Database Theory*, Lecture Notes in Computer Science 470, pages 515–528, Paris, France, December 1990. Springer-Verlag.

[6] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[7] R. Giacobazzi, S.K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–248, 1995.

[8] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995. Available from `http://www.clip.dia.fi.upm.es/`.

[9] L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.

[10] Neil D. Jones and Steven S. Muchnick. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 12, pages 380–393. Prentice-Hall, 1981.

[11] R. M. Karp. Reducibility among Combinatorial Problems. *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher (eds), Plenum Press, New York, 1972, pp. 85–103.

[12] S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.

[13] N.-W. Lin. Approximating the Chromatic Polynomial of a Graph. *Proc. Nineteenth International Workshop on Graph-Theoretic Concepts in Computer Science*, Amsterdam, June 1993.

[14] P. López García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, (22):715–734, 1996.

[15] F. A. Rabhi and G. A. Manson. Using Complexity Functions to Control Parallelism in Functional Programs. Res. Rep. CS-90-1, Dept. of Computer Science, Univ. of Sheffield, England, Jan 1990.

[16] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.