PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Towards an Efficient Compile-Time Granularity Analysis Algorithm

X. Zhong, E. Tick, S. Duvvuru,
L. Hansen, A. V. S. Sastry and R. Sundararajan
Dept. of Computer Science
University of Oregon
Eugene, OR 97403

Abstract

We present a new granularity analysis scheme for concurrent logic programs. The main idea is that, instead of trying to estimate costs of goals precisely, we provide a compile-time analysis method which can efficiently and precisely estimate *relative* costs of active goals given the cost of a goal at runtime. This is achieved by estimating the cost relationship between an active goal and its subgoals at compile time, based on the call graph of the program. *Iteration parameters* are introduced to handle recursive procedures. We show that the method accurately estimates cost, for some simple benchmark programs. Compared with methods in the literature, our scheme has several advantages: it is applicable to any program, it gives a more precise cost estimation than static methods, and it has lighter runtime overheads than absolute estimation methods.

1 Introduction

The importance of grain sizes of tasks in a parallel computation has been well recognized [6, 5, 7]. In practice, the overhead to execute small grain tasks in parallel may well offset the speedup gained. Therefore, it is important to estimate the costs of the execution of tasks so that at runtime, tasks can be scheduled to execute sequentially or in parallel to achieve the maximal speedup.

Granularity analysis can be done at compile time or runtime or even both [7]. The compile-time approach estimates costs by statically analyzing program structure. The program is partitioned statically and the partitioning scheme is independent of runtime parameters. Costs of most tasks, however, are not known until parameters are instantiated at runtime and therefore, the compile-time approach may result in inaccurate estimates. The runtime approach, on the other hand, delays the cost estimation until execution and can therefore make more accurate estimates. However, the overhead to estimate costs is usually too large to achieve efficient speedup, and therefore the approach is usually infeasible. The most promising approach is to try to get as much cost estimation information as possible at compile time and make the overhead of runtime scheduling very slight. Such approach has been taken by Tick [10], Debray *et al.* [2], and King and Soper [4]. In this paper, we adopt this strategy.

A method for the granularity analysis of concurrent logic programs is proposed. Although the method can be well applied to other languages, such as functional languages, in this paper, we discuss the method only in the context of concurrent logic programs. The key observation behind this method is that task spawning in many concurrent logic program language implementations, such as Flat Guarded Horn Clauses (FGHC) [13], depends only on the *relative* costs of tasks. If the compile-time analysis can provide simple and precise cost relationships between an active goal and its subgoals, then the runtime scheduler can efficiently estimate the costs of the subgoals based on the cost of the active goal. The method achieves this by estimating, at compile time, the cost relationship based on the call graph and the introduction of iteration parameters. We show that for common benchmark programs, the method gives correct estimates.

2 Motivations

Compile-time granularity analysis is difficult because most of the information needed, such as size of a data structure and number of loop iterations, are not known until runtime. Sarkar [7] used a profiling method to get the frequency of recursive and nonrecursive function calls for a functional language. His method is simple and does not have runtime overheads, but can give only a rough estimate of the actual granularity.

In the logic programming community, Tick [10] first proposed a method to estimate weights of procedures by analyzing the call graph of a program. The method, as refined by Debray [1], derives the call graph of the program, and then combines procedures which are mutually recursive with each other into a single cluster (i.e., a strongly connected component in the call graph). Thus the call graph is converted into an acyclic graph. Procedures in a cluster are assigned the same weight

which is the sum of the weights of the cluster's children (the weights of leaf nodes are one, by definition). This method has very low runtime overhead; however, goal weights are estimated statically and thus cannot capture the dynamic change of weights at runtime. This problem is especially severe for recursive (or mutually recursive) procedures.

As an example of the method, consider the naive-reverse procedure in Figure 1. (The clauses in the `nrev/2` program do not have guards, i.e., only head unification is responsible for commit.) Examining the call graph, we find that the algorithm assigns a weight of one to `append/3` (it is a leaf), and a weight of two to `nrev/2` (one plus the weight of its child). Such weights are associated with *every* procedure invocation and thus cannot accurately reflect execute time.

Debray *et al.* [2] presented a compile-time method to derive costs of predicates. The cost of a predicate is assumed to depend solely on its input argument sizes. Relationships between input and output argument sizes in predicates are first derived based on so-called data dependency graphs and then recurrence equations of cost functions of predicates are set up. These equations are then solved at compile time to derive closed forms (functions) for the cost of predicates and their input argument sizes, together with the closed forms (functions) between the output and input argument sizes. Such cost and argument size functions can be evaluated at runtime to estimate costs of goals. A similar approach was also proposed by King and Soper [4]. Such approaches represent a trend toward precise estimation. For `nrev/2`, Debray's method gives $\text{Cost}_{nrev}(n) = 0.5n^2 + 1.5n + 1$, where n is the size of the input argument. This function can then be inserted into the runtime scheduler. Whenever `nrev/2` is invoked, the cost function is evaluated, which obviously requires the value n, the size of its first argument. If the cost is bigger than some preselected overhead threshold, the goal is executed in parallel; otherwise, it is executed sequentially.

The method described suffers from several drawbacks (see [11] for further discussion). First, there may be considerable runtime overhead to keep track of argument sizes, which are essential for the cost estimation at runtime. Furthermore, the sizes of the initial input arguments have to be given by users or estimated by the program when the program begins to execute. Second, within the umbrella of argument sizes, different metrics may be used, e.g., list length, term depth, and the value of an integer argument. It is unclear (from [2, 4]) how to correctly choose metrics which are relevant for a given predicate. Third, the resultant recurrence equations for size relationships and cost relationships can be fairly complicated.

It is therefore worth remedying the drawbacks of the above two approaches. It is also clear that there is a tradeoff between precise estimation and runtime overhead. In fact, Tick's approach and Debray's approach represent two extremes in the granularity estimation spectrum. Our intention here is to design a middle-of-the-spectrum method: fairly accurate estimation, applicable to any procedures, without incurring too much runtime overhead.

3 Overview of the Approach

We argue here, as in our earlier work, that it is sufficient to estimate only *relative* costs of goals. This is especially true for an on-demand runtime scheduler [8]. Therefore, it is important to capture the cost *changes* of a subgoal and a goal, but not necessarily the "absolute" granularity. Obviously the costs of subgoals of a parent goal are always less than the cost of the parent goal, and the sum of costs of the subgoals (plus some constant overhead) is equal to the cost of the parent goal. The challenging problem here is how to distribute the cost of the parent goal to its subgoals properly, especially for a recursive call. For instance, consider the naive reverse procedure `nrev/2` again. Suppose goal `nrev([1,2,3,4],R)` is invoked (i.e., clause two is invoked) and the cost of this query is given, what are the costs of `nrev([2,3,4],R1)` and `append(R1,[1],R)`?

It is clear that the correct cost distribution depends on the runtime state of the program. For example, the percentage of cost distributed to `nrev([1,2,3,4],R)` (i.e., as one of the subgoals of `nrev([1,2,3,4,5],T)` will be different from that of cost distributed to `nrev([1,2],R)`. To capture the runtime state, we introduce an *iteration parameter* to model the runtime state, and we associate an iteration parameter with every active goal. Since the cost of a goal depends solely on its entry runtime state, its cost is a function of its iteration parameter. Several intuitive heuristics are used to capture the relations between the iteration parameter of a parent goal and those of its children goals. To have a simple and efficient algorithm, only the AND/OR call graph of the program, which is slightly different from the standard call graph, is considered to obtain these iteration relationships. Such relations are then used in the derivation of recurrence equations of cost functions of an active goal and its subgoals. The recurrence equations are derived simply based on the above observation, i.e., the cost of an active goal is equal to the summation of the costs of its subgoals.

We then proceed to solve these recurrence equations for cost functions bottom up, first for the leaf nodes of the modified AND/OR call graph, which can be obtained in a similar way in Tick's modified algorithm by clustering those mutually recursive nodes together in the AND/OR call graph of the program (see Section 2). After we obtain all the cost functions, cost distribution functions are derived as follows. Suppose the cost of an

active goal is given, we first solve for its iteration parameter based on the cost function derived. Once the iteration parameter is solved, costs of its subgoals, which are functions of their iteration parameters, can be derived based on the assumption that these iteration parameters have relationships with the iteration parameter of their parent, which are given by the heuristics. This gives the cost distribution functions desired for the subgoals.

To recap, our compile-time granularity analysis procedure consists of the following steps:

1. Form the call graph of the program and cluster mutually recursive nodes of the modified AND/OR call graph.
2. Associate each procedure (node) in the call graph with an iteration parameter and use heuristics to derive the iteration parameter relations.
3. Form recurrence equations for the cost functions of goals and subgoals.
4. Proceed bottom up in the modified AND-OR call graph to derive cost functions.
5. Solve for iteration parameters and then derive cost distribution functions for each predicate.

4 Deriving Cost Relationships

4.1 Cost Functions and Recurrence Equations

To derive the cost relationships for a program, we use a graph G (called an AND/OR call graph) to capture the program structure. Formally, G is a triple (N, E, A), where N is a set of procedures denoted as $\{p_1, p_2, \ldots, p_n\}$ and E is a set of pair nodes such that $(p_1, p_2) \in E$ if and only if p_2 appears as one of the subgoals in one of the clauses of p_1. Notice that there might be multiple edges (p_1, p_2) because p_1 might call p_2 in multiple clauses. A is a partition of the multiple-edge set E such that (p_1, p_2) and (p_1, p_3) are in one element of A if and only if p_2 and p_3 are in the body of the same clause whose head is p_1. Intuitively, A denotes what subgoals are AND processes. After applying A to edges leaving out a node, edges are partitioned into clusters which correspond to clauses and these clauses are themselves OR processes. Figure 2 shows an example, where the OR branches are labeled with a bar, and AND branches are unmarked. Leaf facts (terminal clauses) are denoted as empty nodes.

As in [1], we modify G so that we can cluster all those recursive and mutually recursive procedures together and form a directed acyclic graph (DAG). This is achieved by traversing G and finding all strongly-connected components. In this traversing, the difference between AND and OR nodes is immaterial, and we simply discard the partition A. A procedure is recursive if and only if the procedure is in a strongly-connected component. After nodes are clustered in a strongly-connected component in G, we form a DAG G', whose nodes are those strongly-connected components of G and edges are simply the collections of the edges in G. This step can be accomplished by an efficient algorithm proposed by Tarjan [9].

The cost of an active goal p is determined by two factors: its entry runtime state s during the program execution and the structure of the program. We use an integer n, called the *iteration parameter*, to approximately represent state s. Intuitively, n can be viewed as an encoding of a program runtime state. Formally, let $\mathcal{S}$ be the set of program runtime states, M be a mapping from $\mathcal{S}$ to the set of natural numbers N such that $M(s) = n$ for $s \in \mathcal{S}$. It is easy to see that the cost of p is a function of its iteration parameter n. It is also clear that the iteration parameter of a subgoal of p is a function of n. Hereafter, suppose p_{ij} is the j^{th} subgoal in the i^{th} clause of p. We use $I_{ij}(n)$ to represent the iteration parameter of p_{ij}. The problem of how to determine function I_{ij} will be discussed in Section 4.2.

To model the structure of the program, we use the AND/OR call graph G as an approximation. In other words, we ignore the attributes of the data, such as size and dependencies. We first derive recurrence equations of cost functions between a procedure p and its subgoals by looking at G. Let $\text{Cost}_p(n)$ denote the cost of p. Three cases arise in this derivation:

> **Case 1:** p is a leaf node of G' which is non-recursive. This includes cases where that p is a built-in predicate. In this case, we simply assign a constant c as $\text{Cost}_p(n)$. c is the cost to execute p. For instance such cost can be chosen as the number of machine instructions in p.

For the next two cases, we consider non-leaf nodes p, with the following clauses (OR processes),

$$\begin{aligned} C_1 &: p \;\; :\!\!- \;\; p_{11}, \ldots, p_{1n_1}. \\ C_2 &: p \;\; :\!\!- \;\; p_{21}, \ldots, p_{2n_2}. \\ &\cdots \\ C_k &: p \;\; :\!\!- \;\; p_{k1}, \ldots, p_{kn_k}. \end{aligned}$$

Let the cost of each clause be $\text{Cost}_{C_j}(n)$ for $1 \leq j \leq k$. We now distinguish whether or not p is recursive.

> **Case 2:** p is not recursive and not mutually recursive with any other procedures. We can easily see that

$$\text{Cost}_p(n) \leq \sum_{j=1}^{k} \text{Cost}_{C_j}(n). \qquad (1)$$

Conservatively, we approximate $\text{Cost}_p(n)$ as the right-hand side of the above inequality. Notice that in a committed-choice language, the summation in the above inequality can be changed to the maximum (i.e., max) function. However this increases the difficulty of the algebraic manipulation of the resultant recurrence equations (see [11] for example) and we prefer to use the summation as an approximation.

Case 3: p is recursive or mutually recursive. In this case, we must be careful in the approximation, since minor changes in the recurrence equations can give rise to very different estimation. This can be seen for `split` in `qsort` example in Section 2.

To be more precise, we first observe that some clauses are the "boundary clauses," that is, they serve as the termination of the recursion. The other clauses, whose bodies have some goals which are mutually recursive with p, are the only clauses which will be effective for the recursion. Without loss of generality, we assume for $j > u$, C_j are all those "mutually recursive" clauses. For a nonzero iteration parameter n (i.e., $n > 0$), we take the average costs of these clauses as an approximation:

$$\text{Cost}_p(n) = \frac{1}{k-u} \sum_{j=u+1}^{k} \text{Cost}_{C_j}(n) \qquad (2)$$

and for $n = 0$, we take the sum of the costs of those "boundary clauses" as the boundary condition of $\text{Cost}_p(n)$:

$$\text{Cost}_p(0) = \sum_{j=1}^{u} \text{Cost}_{C_j}(0).$$

The above estimation only gives the relations between cost of p and those of its clauses. The cost of clause C_j can be estimated as

$$\text{Cost}_{C_j}(n) = \text{CHead}_j + \sum_{m=1}^{n_j} \text{Cost}_{p_{jm}}(I_{jm}(n)) \qquad (3)$$

where CHead_j is a constant denoting the cost for head unification of clause C_j and $I_{jm}(n)$ is the iteration parameter for the m^{th} body goal. Substituting Equation 3 back into Equation 1 or 2 gives us the recurrence equations for cost functions of predicates.

4.2 Iteration Parameters

There are several intuitions behind the introduction of the iteration parameter. As we mentioned above, iteration parameter n represents an encoding of a program runtime state as a positive integer. In fact, this type of encoding has been used extensively in program verification, e.g., [3], especially in the proof of loop termination. A loop $\mathcal{L}$ terminates if and only it is possible to choose a function M which always maps the runtime states of $\mathcal{L}$ to nonnegative integers such that M monotonically decreases for each iteration of $\mathcal{L}$. Such encoding also makes it possible to solve the problem that once the cost of an active goal is given, its iteration parameter can be obtained. This parameter can be used to derive costs of its subgoals (provided the iteration-parameter functions I_m are given), which in turn give the cost distribution functions.

Admittedly, the encoding of program states may be fairly complicated. Hence, to precisely determine the iteration-parameter functions for subgoals will be complicated too. In fact, this problem is statically undecidable since this is as complicated as to precisely determine the program runtime behavior at compile time. Fortunately, in practice, most programs exhibit regular control structures that can be captured by some intuitive heuristics.

To determine the iteration-parameter functions, we first observe that there is a simple conservative rule: for a recursive body goal p, when it recursively calls itself back again, the iteration parameter must have been decreased by one (if the recursion terminates). This is similar to the loop termination argument. Therefore, as an approximation, we can use $I_m(n) = n - 1$ as a *conservative estimation* for a subgoal p_{im} which happens to be p (self-recursive). Other heuristics are listed as follows:

§1. For a body goal p_{im} whose predicate only occurs in the body once and it is not mutually recursive with p (i.e., not in a strongly-connected component of p), $I_{im}(n) = n$.

§2. If p_{im} is mutually recursive with p and its predicate only occurs once in the body, $I_{im}(n) = n - 1$.

§3. If p_{im} is mutually recursive with p and its predicate occurs l times in the body, where $l > 1$, $I_{im}(n) = n/l$ (this is integer division, i.e., the floor function).

The intuitions behind these heuristics are simple. Heuristic §1 represents the case where a goal does not invoke its parent. In almost all programs, this goal will process information supplied by the parent, thus the it-

eration parameter remains unmodified. Heuristic §2 is based on the previous conservative principle. Heuristic §3 is based on the intuition that the iteration is divided evenly for multiple callees. Notice for the situation in heuristic §3, we can also use our conservative principle. However, we avoid use of the conservative principle, if possible, because the resultant estimation of $\text{Cost}_p(n)$ may be an exponential function of n, which, for most practical programs, is not correct.

These heuristics have been derived from experimentation with a number of programs, placing a premium on the *simplicity* of $I(n)$. A partial summary of these results is given in Section 6. A remaining goal of future research is to further justify these heuristics with larger programs, and derive alternatives.

4.3 An Example: Quicksort

After we have determined the iteration-parameter functions, we have a system of recurrence equations for cost functions. These system of recurrence equations can be solved in a bottom-up manner in the modified graph G'. The problem of systematically solving these recurrence equations in general is discussed in [11]. Here, we consider a complete example for the `qsort/2` program given in Figure 2.

The boundary condition for $\text{Cost}_{qsort}(n)$ is that $\text{Cost}_{qsort}(0)$ is equal to the constant execution cost d_1 of `qsort/2` clause one. The following recurrence equations are derived:

$$\begin{aligned} \text{Cost}_{qsort}(0) &= d_1 \\ \text{Cost}_{qsort}(n) &= \text{Cost}_{C_2} \end{aligned}$$

With Heuristic §3, we have

$$\text{Cost}_{C_2} = d_2 + \text{Cost}_{split}(n) + 2\text{Cost}_{qsort}(n/2)$$

where d_2 is the constant cost for the head unification of the second clause of `qsort/2`.

Similarly, the recurrence equations for $\text{Cost}_{split}(n)$ are

$$\begin{aligned} \text{Cost}_{split}(0) &= d_3 \\ \text{Cost}_{split}(n) &= (\text{Cost}_{C_2} + \text{Cost}_{C_3})/2 \end{aligned}$$

Furthermore,

$$\begin{aligned} \text{Cost}_{C_2} &= \text{Cost}_{C_3} \\ &= d_4 + \text{Cost}_{split}(n-1) \end{aligned}$$

where d_4 is the constant cost for the head unification of the second (and the third) clause of `split`. We first solve the recurrence equations for `split`, which is in the lower level in G' and and then solve the recurrence equations for `qsort`. This gives us $\text{Cost}_{split}(n) = d_3 + d_4 n$ which can be approximated as $d_4 n$ and $\text{Cost}_{qsort}(n) = d_1 + d_2 \log n + d_4 n \log n$, which is the well known average complexity of `qsort`.

Finally, it should be noted that it is necessary to distinguish between the recursive and nonrecursive clauses here and take the average of the recursive clause costs as an approximation. If we simply take the summation of all clause costs together as the approximation of the cost function, both cost functions for `split` and `qsort` would be exponential, which are not correct. More precisely, if the summation of all costs of clauses of `split` is taken as $\text{Cost}_{split}(n)$, we will have

$$\text{Cost}_{split}(n) = d_3 + 2(d_4 + \text{Cost}_{split}(n-1))$$

The solution of $\text{Cost}_{split}(n)$ is an exponential function, which is not correct.

5 Distributing Costs

So far, we have derived cost functions of the iteration parameter for each procedure. However, to know the cost of a procedure, we need to first know the value of its iteration parameter. This, as pointed out in our introduction, may require too much overhead. We notice that, in most scheduling policies (such as on-demand scheduling), only *relative* costs are needed. This can be relatively easily achieved in our theory since cost functions only have a single parameter (iteration parameter).

To derive cost distributing formulae for a given procedure and its body goals, the first step is to solve for the iteration parameter n in Equation 3 assuming that $\text{Cost}_p(n)$ is given at runtime as C_p. Assuming that clause i is invoked in runtime, we approximate $\text{Cost}_{C_i}(n)$ as C_p and solve Equation 3 for n. Let $n = F(C_p)$ be the symbolic solution, which depends on the runtime value of $\text{Cost}_p(n)$ (i.e., C_p), we can easily derive costs of its subgoals of clause i as we can simply substitute n with $F(C_p)$ in $\text{Cost}_{p_{im}}(I_{im}(n))$, which gives rise to the cost distributing functions we need to derive at compile time.

Let's reconsider the `nrev/2` procedure. The cost equations are derived as follows:

$$\begin{aligned} \text{Cost}_{nrev}(n) &= \text{Cost}_{nrev}(n-1) + \text{Cost}_{append}(n) \\ \text{Cost}_{nrev}(0) &= c_1 \\ \text{Cost}_{append}(n) &= \text{Cost}_{append}(n-1) + C_a \\ \text{Cost}_{append}(0) &= c_2 \end{aligned}$$

We can easily derive the closed forms for these two cost functions as $\text{Cost}_{append}(n) = n \times C_a + c_2$ which can be approximated as $C_a \times n$, and $Cost_{nrev}(n) \doteq C_a \times n^2/2$. Now, given the $\text{Cost}_{nrev}(n)$ as C_r, we solve for n and have $n = \sqrt{\frac{2C_r}{C_a}}$. Hence, we have $\text{Cost}_{nrev}(n-1) =$

$C_a(\sqrt{\frac{2C_r}{C_a}} - 1)^2/2$ and $\text{Cost}_{append}(n) = C_a\sqrt{\frac{2C_r}{C_a}}$. These are the desired cost distributing functions.

It should be pointed out that in some cases, it is not necessary to first derive the cost functions and then derive the cost distributing functions since we can simply derive the cost distributing scheme directly from the cost recurrence equations. For example, consider the Fibonacci function, where the cost equations are

$$\begin{aligned} \text{Cost}_{fib}(n) &= C_f + 2 \times \text{Cost}_{fib}(n/2) \\ \text{Cost}_{fib}(0) &= C_1 \end{aligned}$$

Without actually deriving the cost functions of $\text{Cost}_{fib}(n)$, we can simply derive the cost distributing relationship from the first equation as $\text{Cost}_{fib}(n/2) = (\text{Cost}_{fib}(n) - C_f)/2$.

Also note that at compile time, the cost distributing functions should be simplified as much as possible to reduce the runtime overhead. It is even worthwhile sacrificing precision to get a simpler function. Therefore, a conservative approach should be used to derive the upper bound of the cost functions. In fact, we can further simplify the cost function derived in the following way. If the cost function is of a polynomial form such as $c_0n^k + c_1n^{k-1} + \ldots c_k$, we simplify it as kc_0n^k and if the cost function is of several exponential components such as $c_1a^n + c_2b^n$ where $b > a$, we simplify it as $(c_1 + c_2)b^n$. This will simplify the solution of the iteration parameter and the cost distributing function and hence simplify the evaluation of them at runtime.

5.1 Runtime Goal Management

The above cost relationship estimation is well suited for a runtime scheduler which adopts an on-demand scheduling policy (e.g., [8]), where PEs maintain a local queue for active goals and once a PE becomes idle, it requests a goal from other PEs. A simple way to distribute a goal to a requesting PE is to migrate an active goal in the queue. The scheduler should adopt a policy to decide which goal is going to be sent. It is obvious that the candidate goal should have the maximal grain size among those goals in the queue. Hence, we can use a priority queue where weights of goals are their grain sizes (or costs). The priority is that the bigger the costs are, the higher priority they get. Because the scheduler only needs to know the relative costs, we can always assume the weight of the initial goal is some fixed, big-enough number. Based on this initial cost and the cost distributing formulae derived at compile time, every time a new clause is invoked, the scheduler derives the relative costs of body goals. The body goals are then enqueued into the priority queue based on their costs.

Some bookkeeping problems arise from this approach. First, even though we can simplify the cost distributing functions at compile time to some extent, the runtime overhead may still be large, since for each procedure invocation, the scheduler has to calculate the weights of the body goals. One solution to this problem is to let the scheduler keep track of a modulo counter and when the content of the counter is not zero, the scheduler simply lets the costs of the body goals be the same as that of their parent. Once the content of the counter becomes zero, the cost-distributing functions are used. If we can choose an appropriate counting period, this method is reasonable (one counter increment has less overhead than the evaluation of the cost estimate).

Another problem in this approach is that for long-running programs, costs may become negative, i.e., the initial weight is not large enough. Since we require only relative costs, a solution is to reset all costs (including those in the queue, and in suspended goals), when some cost becomes too small. Cost resetting requires the incremental overhead of testing to determine when to reset.

As stated above, we need to choose the initial cost as big as possible. However, this can introduce an anomaly for our relative cost scheme. To see this, consider the `nrev` example again. Suppose that the initial query is `nrev([1,...,50])`. The correct query cost is approximately $50 \times 50 = 2500$. The correct cost of its immediate append goal is approximately 49, and the correct cost of one of its leaf descendant goals `nrev([])` is one (the head unification cost). If we choose the initial cost as a big number, say 10^6, then the corresponding iteration parameter is 10^3. This will give the cost of `nrev([])` as $(10^3 - 50)^2$ which is bigger than the estimated cost of the initial append goal (only around 10^3). In other words, this gives an incorrect relationship between goals near the very top and near the very bottom of the proof tree.

For this particular example, the problem could be finessed by precomputing the "correct" initial value of the iteration parameter: exactly equal to the weight of the query. However, in general, a correct initial estimation is not always possible, and when it *is* possible, its computation incurs too much overhead. All compile-time granularity estimation schemes must make this tradeoff. Fortunately, in our scheme, the problem is not as serious as it first appears. For initial goals with sufficiently large cost, our scheme is still able to give correct relative cost estimation for sufficiently large goals which are not close to leaves of the execution call graph. This can be seen in the `nrev` example, where the relative costs among `nrev([2,...,50]` through `nrev([42,...,50])`, and the initial append are still correct in our scheme. Correct estimation for the large goals (those near the root of the proof tree) is more important than that for small goals (those near the leaves) because the load balance of the system is largely dependent on those big goals, and so is performance.

Heuristic	Applicable	Correct	Percentage
§1	24	21	87.5%
§2	29	26	89.6%
§3	4	2	50.0%
all	32	27	84.7%

Table 1: Statistics for Benchmark Programs

Heuristic	Applicable	Correct	Percentage
§1	64	57	89.1%
§2	49	55	87.3%
§3	6	4	66.7%
all	111	101	91.0%

Table 2: Statistics for a Compiler Front End

6 Empirical Results: Justifying the Heuristics

We applied our three heuristics and the cost estimation formulae to two classes of programs. The first class includes nine widely used benchmark programs [12], containing 32 procedures. The second class consists of 111 procedures comprising the front-end of the Monaco FGHC compiler. The results are summarized in Table 1 and Table 2. For each heuristic, the tables show the number of procedures for which the heuristic is applicable (by the syntactic rules given in Section 4.2), and the number for which the heuristic is correctly estimates complexity. The row labeled "all" gives the total number of procedures analyzed. Since more than one heuristic may be applicable in a single procedure, the total number of procedures may be less than the sum of the previous rows.

From the tables, we see that §1 and §2 apply most frequently. This indicates that most procedures are linear recursive (i.e., have a single recursive body goal) which can be estimated correctly by our scheme. The relatively low percentage of §3 correctness is because the benchmarks are biased towards procedures with exponential time complexity, whereas §3 usually gives polynomial time complexity.

Analysis of the benchmarks indicated two major anomalies in the heuristics. Although §1 may apply, a procedure may distribute a little work (say, the head of a list) to one body goal and the rest of the work (say, the tail of the list) to another goal. This cannot be captured by §1, which essentially treats the head and tail of the list as equal, i.e., a binary tree. A correct cost analysis needs to explore the data structures of the program.

For recursive procedures, §3 can capture only the fixed-degree divide & conquer programming paradigm. However, the compiler benchmark contained procedures which recursively traverse a list (or vector) and the degree of the divide & conquer dynamically depends on the number of top-level elements in the list (or vector). In this situation, the procedure may have to loop on the top level while recursively traversing down for each element (which may be tree structures). Again, this presents inherent difficulty for our scheme because we take the call graph as the sole input information for the program to be analyzed.

To summarize, our statistics show that our scheme achieves a fairly high percentage of correct estimation. However, we need to apply multiply-recursive heuristics §2 and §3 with more finesse. Further quantitative performance studies of the algorithm's utility are presented in Tick and Zhong [11]. Those multiprocessor simulation results quantify the advantage of dynamically scheduling tasks with the granularity information.

7 Conclusions and Future Work

We have proposed a new method to estimate the relative costs of procedure execution for a concurrent language. The method is similar to Tick's static scheme [10], but gives a more accurate estimation and reflects runtime weight changes. This is achieved by the introduction of an iteration parameter which is used to model recursions.

Our method is based on the idea that it is not the absolute cost, but rather the relative cost that matters for an on-demand goal scheduling policy. Our method is also amenable to implementation. First, our method can be applied to any program. Second, the resultant recurrence equations can be solved systematically. In comparison, it is unclear how to fully mechanically implement the schemes proposed in [2, 4]. Nonetheless, our method may result in an inaccurate estimation for some cases. This is because we use only the call graph to model the program structure, not the data. We admit that further static analysis of program structure such as argument-size relationships can give more precise estimations.

Future work in granularity analysis includes the development of a more systematic and precise method to solve the derived recurrence equations. It is also necessary to examine this method for more practical programs, performing benchmark testing on a multiprocessor to show the utility of the method.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with funding from Sequent Computer Systems Inc. The authors wish to thank S. Debray and the anonymous referees for their helpful criticism.

REFERENCES

[1] S. K. Debray. A Remark on Tick's Algorithm for Compile-Time Granularity Analysis. *Logic Programming Newsletter*, 3(1):9–10, 1989.

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

[3] D. Gries. *Science of Programming*. Springer-Verlag, 1989.

[4] A. King and P. Soper. Granularity Control for Concurrent Logic Programs. In *International Computer Conference*, Turkey, 1990.

[5] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, pages 23–32, January 1988.

[6] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32:1073–1978, 1989.

[7] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, Cambridge MA, 1989.

[8] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

[9] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia PA, 1983.

[10] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. *New Generation Computing*, 7(2):325–337, January 1990.

[11] E. Tick and X. Zhong. A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation. *Journal of Parallel and Distributed Computing*, submitted to special issue.

[12] E. Tick. *Parallel Logic Programming*. MIT Press, Cambridge MA, 1991.

[13] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA, 1987.

```
nrev([],R) :- R=[].
nrev([H|T],R) :- nrev(T,R1), append(R1,[H],R).

append([],L,A) :- A=L.
append([H|T],L,A) :- A=[H|A1], append(T,L,A1).
```

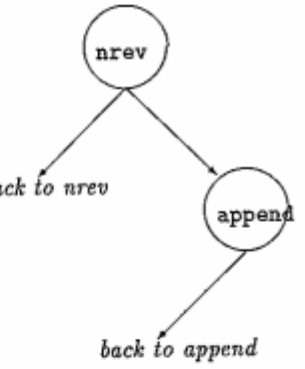

Figure 1: Naive Reverse and its Call Graph

```
qsort([], S) :- S=[].
qsort([M|T],S) :-
    split(T,M,S,L),
    qsort(S,SS),
    qsort(L,LS),
    append(SS,LS,S).

split([],   M,S,L) :- S=[], L=[].
split([H|T],M,S,L) :- H < M  |
    S=[H|TS], split(T,M,TS,L).
split([H|T],M,S,L) :- H >= M |
    L=[H|TL], split(T,M,S,TL).
```

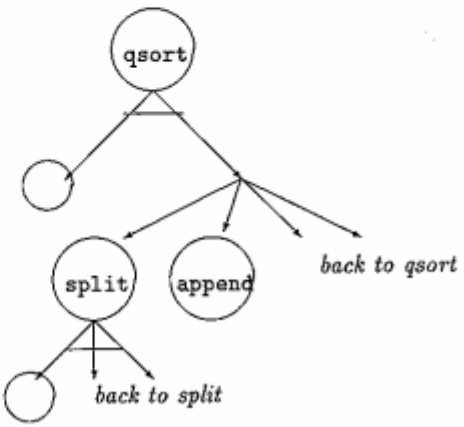

Figure 2: Quick Sort: FGHC Source Code and the AND/OR Call Graph