A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation

X. Zhong and E. Tick

CIS-TR-93-08
April 1993

Abstract

A major implementation problem with implicitly parallel languages, is that small grain size can lead to execution overheads and reduced performance. We present a new granularity-analysis scheme that produces estimators, at compile time, of the relative execution weight of each procedure invocation. These estimators can be cheaply evaluated at runtime to approximate the relative task granularities, enabling intelligent scheduling decisions. Our method seeks to balance tradeoffs between analysis complexity, estimator accuracy, and runtime overhead of evaluating the estimator. To this end, rather than analyze data size or dependencies, we introduce *iteration parameters* to handle recursive procedures. This simplification facilitates solving the recurrence equations that describe the granularity estimators, and reduces the runtime overhead of evaluating these estimators. The algorithm is described in the context of concurrent logic programming languages, although the concepts are applicable to functional languages in general. We show, for a benchmark suite, that the method accurately estimates cost. Multiprocessor simulation results quantify the advantage of dynamically scheduling tasks with the granularity information.

This article appears in *New Generation Computing*, vol. 11, no. 3–4, June, 1993.

Printing history: TR-91-19, September 1991.
TR-91-19a, July 1992.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

1 Introduction

The importance of grain sizes of tasks in a parallel computation has been well recognized [9, 7, 12]. 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 [12]. 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 may be too large to achieve efficient speedup, in which case the approach is infeasible. A more 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. In the context of logic programming languages, such approach has been taken by Tick [16], Debray *et al.* [3], and King and Soper [6]. In this paper, we adopt this strategy.

A method for the granularity analysis of concurrent logic programs is proposed. 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 scheduling in many concurrent logic program language implementations, such as Flat Guarded Horn Clauses (FGHC) [18], can be made efficient by exploiting 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. We show that for a suite of benchmark programs, the method correctly estimates time complexity for 85%–91% of all procedures analyzed. Multiprocessor simulations show that performance improvements can be attained for an on-demand scheduler exploiting the granularity information *vs.* a naive scheduler.

The paper is organized as follows. Section 2 reviews results in the area of granularity analysis. Section 3 outlines our algorithm. Sections 4–5 give the details of the algorithm, with illustrating examples. Section 6 contains empirical results of the accuracy and effectiveness of the scheme. Conclusions and future work are summarized in

Section 7.

2 Motivations and Literature Review

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 [12] 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 [16] first proposed a method to estimate weights of procedures by statically analyzing the call graph of a program. The method, as refined by Debray [1], derives the call graph of the program, and then combines mutually-recursive procedures into clusters (i.e., strongly connected components 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 *static* estimations. Thus every procedure invocation uses the same weight estimation, irrespective of dynamic state such as input arguments. This problem is especially severe for recursive procedures.

To illustrate this and other algorithms in the paper, we now introduce concurrent logic programs, built of guarded Horn clauses of the form:

$$H \text{ :- } G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n$$

where $m \geq 0$ and $n \geq 0$. H is the clause head, G_i is a guard goal, and B_i is a body goal. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). Note that if the guard is empty, no commit operator is given. The informal semantics[1] of such programs are: to execute a procedure $p(A_1, A_2, ..., A_k)$ with arguments A_i, the arguments must match (passively unify with) the formal parameters of the head of some clause with name p. In addition, the guard goals must all succeed. If these conditions hold, the procedure invocation may "commit" to that clause (or *any* other clause so satisfying these conditions). The body of the committed clause is then executed. A program successfully executes when no goals remain to be executed. Synchronization in committed-choice programs is enforced implicitly in the commit rule. If no clause can commit to a procedure invocation, that invocation *suspends*. Suspension occurs because a passed parameter is not sufficiently instantiated to allow

[1]For more formal semantics, see for instance Shapiro [14].

```
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).
```

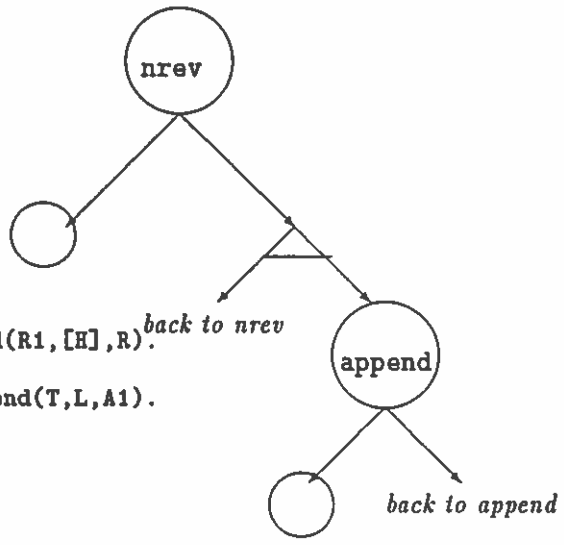

Figure 1: Naive Reverse and its Call Graph

successful head matching or guard reduction. A suspended invocation is *resumed* when a later binding is made to a variable associated with the suspended invocation.

As an example of Tick's original method, consider the naive-reverse procedure in Figure 1.[2] 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.* [3] 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 predicate costs and predicate input argument sizes. In addition, we derive the closed forms (functions) between the output and input argument sizes. Such cost and argument size functions can be evaluated at runtime to estimate the body goal costs. A similar approach was also proposed by King and Soper [6]. Such schemes 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 procedure code. Whenever nrev/2 is invoked, the cost function is evaluated, which obviously requires n. If the cost is greater than some preselected overhead threshold, the goal is executed in parallel; otherwise, the goal is executed sequentially.

[2]Procedures are named as nrev/2, where '2' is the number of formal parameters. The clauses in the nrev/2 program do not have guards, i.e., only head unification is responsible for commit.

The method described suffers from several drawbacks. First, there is a considerable runtime overhead to keep track of argument sizes, which are essential for the cost estimation at runtime. Although relationships between sizes of input and output arguments, within one goal, can be derived at compile time, such relationships must be evaluated at runtime, a potentially heavy overhead. 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, the fundamental assumption behind these methods is that the cost of executing a goal is solely a size-monotonic function of its input arguments. However, the complexity of many procedures depends on other factors, and perhaps even nonmonotonically on argument size, limiting the application of the approach. Furthermore, within the general metric of argument size, different metrics may be needed, e.g., list length, term depth, and the value of an integer argument. It is unclear (from [3, 6]) how to correctly and automatically choose metrics which are relevant for a given predicate.

Third, the resultant recurrence equations for size relationships and cost relationships can be fairly complicated. Although a conservative solution method (i.e., estimate the upper bound of a cost) is adopted, and common recurrence equation templates are used [3], it is still unclear what systematic methods can be used to solve the recurrence equations.

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 procedure, without incurring too much runtime overhead. Instead of considering data dependencies and sizes, we use only the call graph as input to our algorithm, as described in the next section.

3 Overview of the Approach

We argue here, as in our earlier work, that it is often sufficient to estimate only *relative* costs of goals. This is especially true for an on-demand runtime scheduler (e.g., [13]). 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 is how to distribute the cost of the parent goal to its subgoals properly, especially for a recursive call. For instance, reconsider the naive reverse procedure `nrev/2`. 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)`? Intuitively, the recursive call is heavier than the append (this intuition is solidified in Section 5).

The correct cost distribution depends on the runtime state of the program. For example, consider a parent goal A = `nrev([1,2,3,4],R)`, its first child goal B = `nrev([2,3,4],R1)`, and C = `nrev([3,4],R2)`, which is the first child goal of the previous goal (i.e., the grandchild). Although all goals are defined by the *same* procedure, the percentage of cost distributed to B (with respect to A's children) is *different* from the percentage of cost distributed to C (with respect to B's children). To capture the runtime state of a procedure invocation, we introduce an *iteration parameter* to model that 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 call graph of the program, which is slightly different from a standard call graph, need be considered, to obtain these iteration relationships (see Section 4.1). 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 plus a constant.

We then proceed to solve these recurrence equations for cost functions. Unknowns in the recurrence equations are eliminated in their bottom-up order in a reduced call graph which is obtained by clustering together mutually recursive nodes in the original call graph (see Section 2). The leaf nodes are solved first, and so on, in a bottom-up fashion.

After we obtain all the cost functions (which are functions of iteration parameters), cost-distribution functions between an active goal and its body goals are derived as follows. Suppose the cost of an active parent goal is given. We first solve for the parent's iteration parameter based on the cost function derived. Once the iteration parameter is solved, costs of the parent's child (body) goals, which are specified in terms of iteration-parameter functions, are derived. The assumption is that the iteration-parameter functions are simple functions of the parent's iteration parameter, as specified by heuristics. The derived child goal costs are functions of the parent's cost, and constitute the cost-distribution functions needed for runtime goal scheduling.

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

1. Form the *call graph* of the program.
2. Transform the call graph into a *reduced call graph*, by collapsing strongly-connected components (i.e., mutually recursive goals).
3. Associate an *iteration parameter* with each procedure, and use heuristics to derive the *iteration-parameter functions.*
4. Form recurrence equations for the *cost functions* of goals and subgoals.
5. Proceed bottom up in the *reduced call graph* to derive the cost functions.
6. For each clause, solve for the iteration parameter, and derive a *cost-distribution function* for each body goal.

4 Deriving Cost Relationships

In this section we describe steps (1–5) in the analysis algorithm, using an example to illustrate the technique.

4.1 Cost Functions and Their Recurrence Equations

To derive the cost relationships for a program, we use a graph G (a *call graph*) to capture the program structure. Intuitively, a call graph gives the call relationship among procedures, similar to a standard call graph. Unlike the standard call graph, however, in our call graph, calls within one clause are distinguished from calls in other clauses. Such a distinction of calls is used in our cost-analysis scheme.

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 multiset of pair nodes. E represents the call relationship among procedures. $(p_1, p_2) \in E$ represents a call of procedure p_2 from one of the clauses of procedure 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 procedure calls are within one clause. Edges exiting a node are partitioned by A into groups which correspond to clauses. Nodes with zero out degree are called *terminal nodes.* Figure 2 shows an example, where edges in the same partition represented by A are marked with a horizontal bar. For this example, the reduced call graph has the same structure without the back-edges, which are subsumed into their source nodes.

As in Debray [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 called

```
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).

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

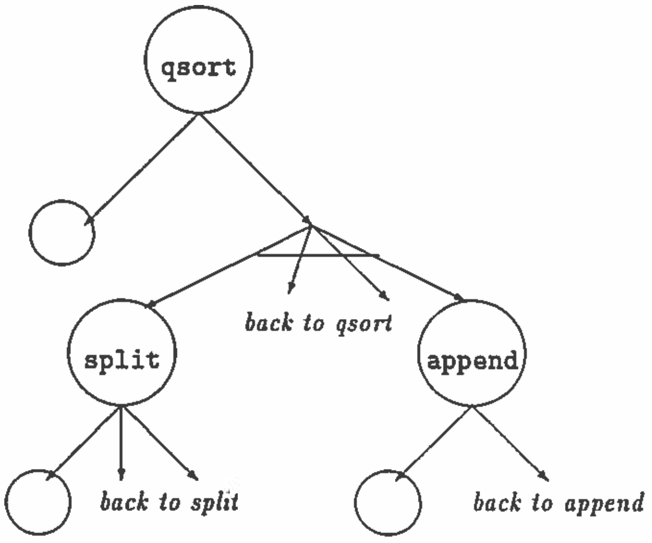

Figure 2: Quick Sort: FGHC Source Code and its Call Graph.

the *reduced call graph*, G', achieved by traversing G and finding all strongly-connected components. In this traversal, the calls from different clauses are 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 [15].

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 approximate 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 iteration-parameter function I_{ij} will be discussed in Section 4.2.

To model the structure of the program, we use the 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 examining G. Let $\text{Cost}_p(n)$ denote the cost of p. Three cases arise in this derivation:

> **Case 1:** p is a terminal node in G. This includes cases where 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-terminal nodes p, with the following clauses (i.e., OR-processes),

$$\begin{aligned} C_1 : p &\;{:}{-}\; p_{11}, \ldots, p_{1v_1}. \\ C_2 : p &\;{:}{-}\; p_{21}, \ldots, p_{2v_2}. \\ &\ldots \\ C_k : p &\;{:}{-}\; p_{k1}, \ldots, p_{kv_k}. \end{aligned}$$

Let the cost of each clause be $\text{Cost}_{C_j}(n)$ for $1 \leq j \leq k$. For flat committed-choice languages, we can safely ignore the guards: their costs are assumed to be constant (see Equation 4). For non-flat languages, we can assume that deep guards are included as goals p_{ij} above. 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. However, this doesn't account for potential suspensions and their associated overheads.
>
> 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 and we prefer to use the summation as an approximation. Another alternative is to use a weighted-arithmetic mean for the right-hand side, which does not complicate the solution. A linear mean could be used for simplicity; however, runtime profiling information [12, 19, 10] will give more accurate estimation of cost. We believe the use of profiling information within our analytical framework is an interesting issue to explore in the future.

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 estimations. This can be seen for `split/4` in the Quick Sort 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:

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

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

$$\mathrm{Cost}_p(0) = \sum_{j=1}^{u} \mathrm{Cost}_{C_j}(0). \tag{3}$$

See the beginning of Section 4.2 for the intuition behind this boundary condition.

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

$$\mathrm{Cost}_{C_j}(n) = d_j + \sum_{m=1}^{v_j} \mathrm{Cost}_{p_{jm}}(I_{jm}(n)) \tag{4}$$

where d_j is a constant reduction cost of clause C_j, and $I_{jm}(n)$ is the iteration parameter for the m^{th} body goal.[3] Substituting Equation 4 back into Equation 1 or 2 gives us the recurrence equations for cost functions of predicates.

The above recurrence equations are not complete because the iteration parameters of each body goal $I_{ij}(n)$ are still unknown. These iteration-parameter functions I_{ij} are derived based on some heuristics described next.

[3]The cost d_j accounts for matching head arguments, evaluating the guard, and enqueuing body goals.

4.2 Iteration Parameters

There are several intuitions behind the introduction of the iteration parameter. As we mentioned above, iteration parameter n represents an approximate 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., [5], especially in proofs of loop termination. A loop $\mathcal{L}$ terminates if and only it is possible to choose a function M which always maps the runtime state of $\mathcal{L}$ to a nonnegative integer such that M monotonically decreases for each iteration of $\mathcal{L}$. When the integer becomes zero the loop exits, which corresponds to the invocation of a boundary clause (Equation 3). 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 complex. 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} which 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 k times in the body, where $k > 1$, $I_{im}(n) = n/k$ (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 iteration 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 $\mathrm{Cost}_p(n)$ 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 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 alternative heuristics.

4.3 Solving for Cost Fuctions and an Example

After setting up the recurrence equations for cost functions and determining iteration-parameter functions, we have a system of recurrence equations for the cost functions. This system of recurrence equations can be solved in a bottom-up manner in graph G'. That is, we first solve equations for procedures in the leaves of G' and then solve those for procedures in higher levels of G'. It should be pointed out that the complexity of these recurrence equations depends on the iteration parameters. The problem of systematically solving recurrence equations derived with the above heuristics is discussed in the Appendix.

We now consider a complete example for the `qsort/2` program given in Figure 2. The boundary condition for $\mathrm{Cost}_{qsort}(n)$ is that $\mathrm{Cost}_{qsort}(0)$ is equal to the constant execution cost d_1 of `qsort/2` clause one. The following recurrence equations are derived:

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

With Heuristic §1 and §3, we have

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

where d_2 is the constant reduction cost of the second clause of `qsort/2`. Similarly, the recurrence equations for $\mathrm{Cost}_{split}(n)$ are

$$\begin{aligned}\mathrm{Cost}_{split}(0) &= d_3\\ \mathrm{Cost}_{split}(n) &= d_4 + (\mathrm{Cost}_{C_2}(n) + \mathrm{Cost}_{C_3}(n))/2\end{aligned}$$

Furthermore,

$$\mathrm{Cost}_{C_2}(n) = \mathrm{Cost}_{C_3}(n) = d_5 + \mathrm{Cost}_{split}(n-1)$$

where d_5 is the constant reduction cost of the second (and third) clause of `split/4`. The recurrence equations for $\text{Cost}_{append}(n)$ are

$$\begin{aligned}\text{Cost}_{append}(0) &= d_7\\ \text{Cost}_{append}(n) &= \text{Cost}_{append}(n-1)+d_6\end{aligned}$$

We first solve the recurrence equations for `split/4` and `append/3`, which are in the lower level in G', and then solve the recurrence equations for `qsort/2`. This gives $\text{Cost}_{split}(n) = d_3+(d_4+d_5)n$ and $\text{Cost}_{append}(n) = d_7+d_6n$ which can be approximated as $(d_4+d_5)n$ and d_6n respectively. We then have $\text{Cost}_{qsort}(n) = d_1 + d_2\log n + (d_4 + d_5 + d_6)n\log n$, which is the well-known average complexity of `qsort`.

It should be emphasized that it is necessary to distinguish between the recursive and nonrecursive clauses here, taking 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

$$\text{Cost}_{split}(n) = d_3+d_4+2(d_5+\text{Cost}_{split}(n-1))$$

In this case, the solution of $\text{Cost}_{split}(n)$ is an exponential function, which is not correct. Adopting Equation 2, however, the correct relationship is derived, as shown above.

5 Distributing Costs

After we have derived cost functions for each procedure, we are ready to derive cost-distribution functions for the body goals of each clause. The first step is to solve for the iteration parameter n in Equation 4 assuming that $\text{Cost}_p(n)$ is given at runtime as W_p. In other words, assuming that clause i is invoked at runtime, we approximate $\text{Cost}_{C_i}(n)$ as W_p and solve Equation 4 for n. Let $n = F(W_p)$ be the symbolic solution (i.e., $F = \text{Cost}_p^{-1}$), which depends on the runtime value of $\text{Cost}_p(n)$ (i.e., W_p). Denoting the cost distribution function for m^{th} body goal as $\text{Cost}^{(m)}(W_p)$, we can derive $\text{Cost}^{(m)}(W_p)$ by substituting n with $F(W_p)$ in $\text{Cost}_{p_{im}}(I_{im}(n))$ for m^{th} body goal. This gives rise to the cost-distribution functions that we need to evaluate at runtime.

As an example, let's reconsider the `nrev/2` procedure. The cost equations are derived as follows:

$$\text{Cost}_{nrev}(0) = d_1$$

$$\begin{aligned}
\text{Cost}_{nrev}(n) &= d_2 + \text{Cost}_{nrev}(n-1) + \text{Cost}_{append}(n) \\
\text{Cost}_{append}(0) &= d_3 \\
\text{Cost}_{append}(n) &= d_4 + \text{Cost}_{append}(n-1)
\end{aligned}$$

where d_i are constant reduction costs. We derive the closed forms for these two cost functions as $\text{Cost}_{append}(n) = nd_4 + d_3 \approx nd_4$ and $\text{Cost}_{nrev}(n) = d_1 + (d_2+d_3)n + d_4n(n-1)/2 \approx d_4n^2/2$. Now, given that $\text{Cost}_{nrev}(n) = W_r$, we solve for $n = \sqrt{\frac{2W_r}{d_4}}$. Hence, we have $\text{Cost}^{(1)}(W_r) = d_4(\sqrt{\frac{2W_r}{d_4}} - 1)^2/2$ for the first body goal nrev, and $\text{Cost}^{(2)}(W_r) = d_4\sqrt{\frac{2W_r}{d_4}}$ for the second body goal append. These are the desired cost-distribution functions. As mentioned in Section 3, intuitively, the recursive call encompasses most of the work, and the append only a small fraction (square-root of total work).

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

$$\begin{aligned}
\text{Cost}_{fib}(0) &= d_1 \\
\text{Cost}_{fib}(n) &= d_2 + 2\text{Cost}_{fib}(n/2)
\end{aligned}$$

Without actually deriving the cost functions of $\text{Cost}_{fib}(n)$, we can simply derive the cost-distribution function from the second equation as $\text{Cost}^{(1)}(W) = \text{Cost}^{(2)}(W) = (W - d_2)/2$.

Also note that at compile time, the cost distribution 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 $c_1a_1{}^n + c_2a_2{}^n + \ldots + c_ka_k{}^n$ where $a_1 \geq \ldots a_k$, we simplify it as $(c_1 + c_2 + \ldots + c_k)a_1{}^n$. This will simplify the solution of the iteration parameter and the cost-distribution function and hence lessen the runtime evaluation cost.

5.1 Quicksort Example Revisited

The **qsort** cost function, $\text{Cost}_{qsort}(n) = d_1 + d_2 \log n + (d_4 + d_5)n \log n$, is too complicated to solve for its inverse function. Therefore we conservatively approximate

```
qsort(_,[],S) :- S=[].
qsort(W,[M|T],S) :-
    DD := W//2,
    WS := sqrt(DD),
    W1 := DD-WS,
    split(WS,T,M,S,L),
    qsort(W1,S,SS),
    qsort(W1,L,LS),
    append(WS,SS,LS,S).

split(_,[],   M,S,L) :- S=[], L=[].
split(W,[H|T],M,S,L) :- H < M |
    W1 := W-1,
    S=[H|TS], split(W1,T,M,TS,L).
split(W,[H|T],M,S,L) :- H >= M |
    W1 := W-1,
    L=[H|TL], split(W1,T,M,S,TL).
```

Figure 3: Quick Sort: Annotated FGHC Source Code

$\text{Cost}_{qsort}(n) \approx (d_4 + d_5)n^2$. Furthermore, assuming unit constant costs, $d_4 = d_5 = 1$, gives $\text{Cost}_{qsort}(n) \approx 2n^2$. Assuming that $\text{Cost}_{qsort}(n) = W$ at runtime, then the qsort cost-distribution functions (for body goal 1 split, body goal 2 and 3 qsort and body goal 4 append) can be derived as $\text{Cost}^{(1)}(W)=\text{Cost}^{(4)}(W) = \sqrt{W/2}$, $\text{Cost}^{(2)}(W)=\text{Cost}^{(3)}(W) = W/2 - \text{Cost}^{(1)}(W) = W/2 - \sqrt{W/2}$.

In this derivation, the sum of the body goal weights equals the parent weight. For consistency, we wish to preserve this work-distribution invariant. With conservative estimation, if we simply substitute the iteration parameter n with the inverse of the cost function (in this case, $\sqrt{W/2}$), the invariant is not guaranteed. Therefore we obtain all *but the last* body goal cost-distribution functions by substitution. The last goal is solved by explicitly satisfying the invariant.[4]

After deriving the cost-distribution functions, the functions can be inserted into the program to be exploited by a runtime scheduler. For example, we can annotate each procedure in the source program by including a new input argument holding the granularity cost, i.e., the runtime weight of that procedure. The corresponding runtime weights of the procedure's body goals are generated by evaluating the cost-distribution functions at runtime. For example, Figure 3 shows the annotated FGHC Quick Sort source code. In this program, the weights are included as a new first argument of each procedure, and the cost-distribution functions are inserted in each clause body. Predicate :=/2 performs arithmetic evaluation and assignment.

[4]Preserving the invariant exactly is not critical for the scheme's effectiveness — approximations can be tolerated here as in other steps of the algorithm. Note however that by conservative estimation of body goal weights, we will never derive negative weights with this method.

5.2 Runtime Goal Management

The above cost relationship estimation is well suited for a runtime scheduler which adopts an on-demand scheduling policy (e.g., [13]), where processors (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. Based on an initial cost (discussed below) and the cost-distribution functions 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 our scheme. First, even though we can simplify the cost-distribution functions at compile time to some extent, this may not be sufficient. In some cases, the runtime overhead of evaluating the cost-distribution functions may offset the advantage of scheduling based on granularity information. This is especially undesirable for trivial procedures, i.e., procedures that execute little computation. On the other hand, we expect an overall benefit from this analysis for nontrivial procedures. Furthermore, in a parallel machine organization that has higher overhead for goal migration (message passing) than goal reduction, runtime weight evaluation may be well-worth the induced savings in reduced messages.

One solution to this potential 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-distribution 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 zero, i.e., the initial weight is not large enough. One solution is to ignore the problem, i.e., bottom-out at zero. Since we require only relative costs, another 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.

How should we choose the initial cost, i.e., the cost of the query? Choosing an arbitrary value 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, *choosing the initial weights as big as possible* (e.g., the maximum representable integer) is 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.

6 Empirical Results

In this section we present empirical data collected by hand-analysis and multiprocessor simulation, to demonstrate the performance potential of the granularity-analysis method.

6.1 Justifying the Heuristics

We applied our three heuristics and the cost-estimation functions to two classes of programs. The first class includes nine widely-used benchmark programs [17], 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 correctly estimates complexity. The row labeled "all" gives the total number

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

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 heuristics §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. This might be corrected by accounting for the *type* of data structures, or as in Debray's scheme [3], exploring the *size* of data structures.

For recursive procedures, §3 can capture only the fixed-degree divide and conquer programming paradigm. However, the compiler benchmark contained procedures which recursively traverse a list (or vector) and the degree of the divide and 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 again 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.

6.2 Performance Analysis

To demonstrate the utility of the granularity-estimation scheme, we simulated a multiprocessor which exploited the weights for on-demand scheduling purposes. The parameterized simulator is a high-level model of the parallel execution of fine-grain FGHC processes [18]. The base time-step, called a *cycle*, is the time needed to commit a procedure invocation to a clause or suspend the process. The model is controlled by three main input parameters: the number of PEs, the cost (in cycles) of a reduction, and the *idle delay*, i.e., the cost of switching in a remote task when a PE becomes idle. It is assumed that switching in a local task, when suspension occurs, is free. In the simulations presented here, the default value of reduction time is a number of cycles equal to the number of user-defined body goals in the matching clause — a realistic assumption. The input parameters allow us to tune the model across the range of tightly-coupled shared-memory to loosely-coupled distributed-memory multiprocessors. For instance, a large ratio of idle delay over reduction cost models a distributed system. As a performance metric, we define $C_{\text{opt}}/C_{\text{noopt}}$, the number of cycles executed by the program exploiting granularity divided by the number of cycles executed by the program with no granularity information.

The simulation proceeds on a cycle-by-cycle basis, keeping ready and suspended process queues for each individual PE. At the beginning of each cycle, all idle PEs steal goals from busy PEs. Our intention is *not* to explore load-balancing methods here, and so we distribute the goals according to one, realistic rule: the most busy (most goals in ready queue) PEs offload goals at the top of their queues to the idle PEs. Any PE with only a single ready goal does not offload. The offloaded goals are tagged with the fixed idle penalty, causing the receiving PE to subsequently spin (modeling transmission delay) for that many cycles. The naive scheduler pushes body goals onto the ready queue in the program-defined order. In the granularity scheduler, the ready queue is prioritized by goal weight, so that the top-most goal is heaviest.

Six benchmarks were chosen to allow comparison with Debray's results [3], and to illustrate characteristics particular to on-demand scheduling with relative weights. Debray's experiment involved two actual parallel Prolog systems: ROLOG and &-Prolog executing on four PEs. These systems explicitly spawned goals that were determined to be independent, i.e., no data dependencies. Of the benchmarks, four were measured on both systems, and of those four, Fibonacci was the only program to achieve consistent results: speedup of 28%. Our experiment differs most significantly in three ways. First, we use weights to decide which goal should be sent to a requesting (idle) PE, not which goals should be eagerly spawned. Second, we exploit relative weights, and therefore have no ability to cutoff recursive decent at an absolute threshold weight, e.g., fib(5). Third, the FGHC architecture exploits communicating processes, and therefore processes may suspend due to lack of data inputs. Thus if granularity information contradicts data dependencies, exploiting granularity without regard for dependencies may induce extra suspensions.

Extensive simulations of the benchmarks were conducted, a partial summary of which is given here. Sufficiently large input data was supplied in all of the runs to avoid starvation problems. All of the benchmarks retained positive weights throughout execution.

Figure 4 plots the performance data by varying PEs, keeping the idle delay = 16 cycles. The utility of the analysis is seen to be highest for eight or more PEs, but doesn't increase much beyond this. For 1–4 PEs, the naive scheduler can do reasonably well because the limited PEs have sufficient large-grain parallelism to exploit in the program. As the number of PEs increases, competition for work exhausts the large grains, and small grains are inadvertently exploited. At this point, the granularity scheduler, differentiating grain size, achieves a performance advantage.

Figure 5 replots benchmark performance with varying idle delay, keeping PE=16 (the default reduction time is used). The benchmarks again fall into roughly two groups. The lower plots represent programs that display a trend towards performance improvement with increasing idle delay. These programs illustrate that granularity information has its highest utility when interprocessor communication is slow, e.g., distributed multiprocessors. The higher plots represent programs that do not improve significantly with increasing idle delay. These programs are "accidentally" scheduled properly even without granularity information because of the program structure. For example, Matrix multiply with 30-element fixnum vectors cannot be misscheduled: the self-recursive inner-product loop comprises a large-enough granule of work that previous scheduling decisions do not much matter. Compare this to Bmatrix, a 5x5 matrix multiply with bignum elements. Here the process-tree branches are thicker, because the inner-product now entails nontrivial bignum multiplication. As a result, superior scheduling decisions can be made based on granularity information.

Figures 6 and 7 show the Fibonacci benchmark for varying PEs and idle delays. Fibonacci was chosen because of its consistent results in Debray's work, and its typical characteristics as displayed in the previous graphs. The linear relationship between performance improvement and idle time is again displayed here. The jump in performance between 4 and 8 PEs is clearly seen. In fact, granularity scheduling *slows down* the program for few PEs.

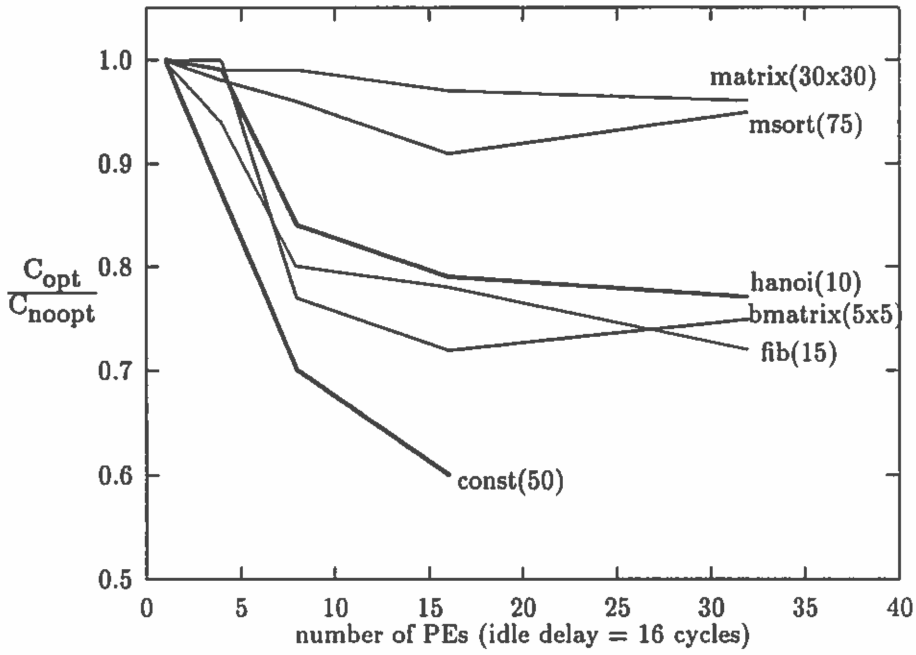

Figure 4: Performance as a Function of Number of PEs

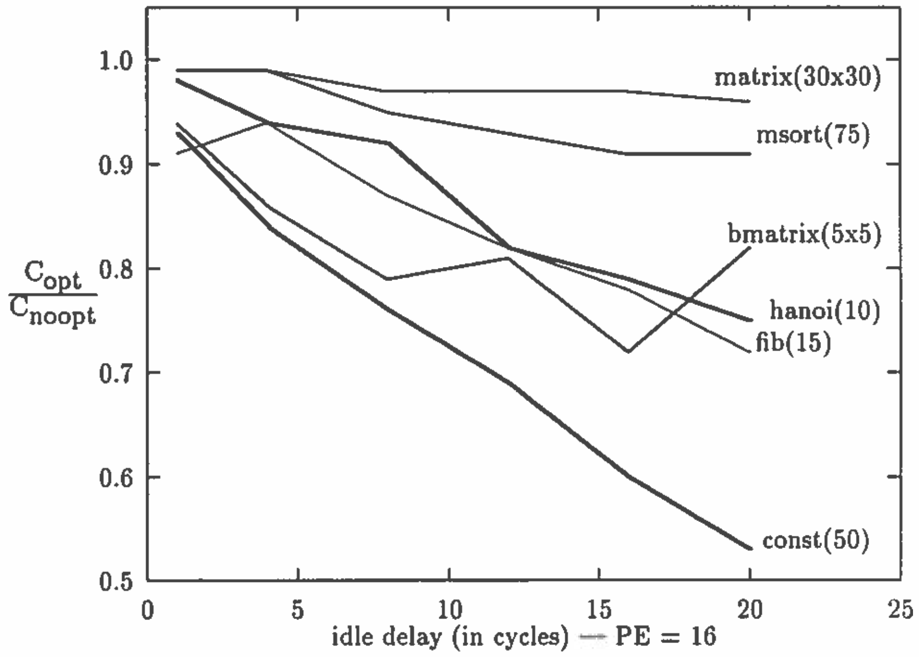

Figure 5: Performance as a Function of Idle Delay

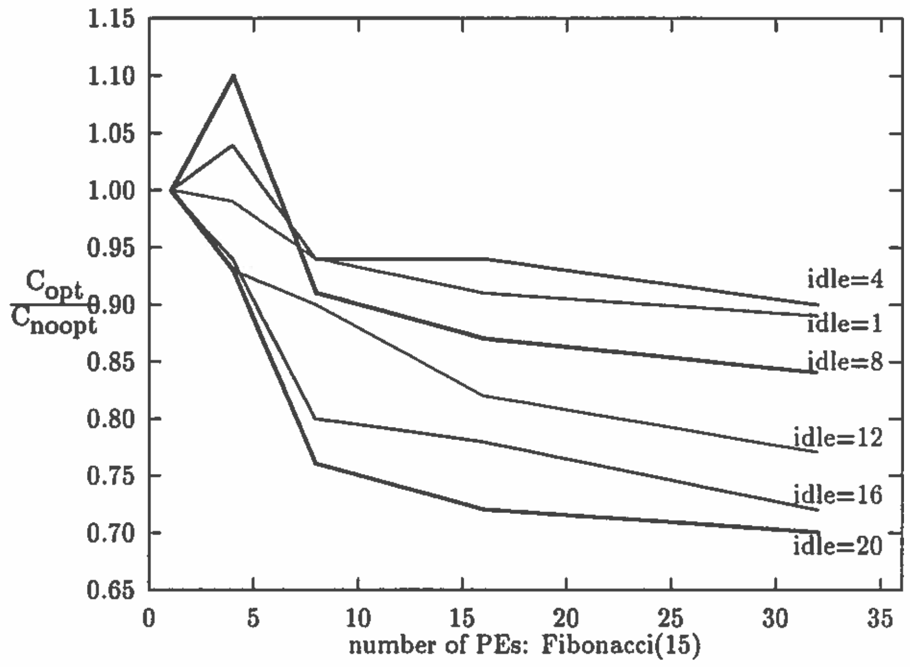

Figure 6: Performance of Fibonacci(15) vs. Idle Delay

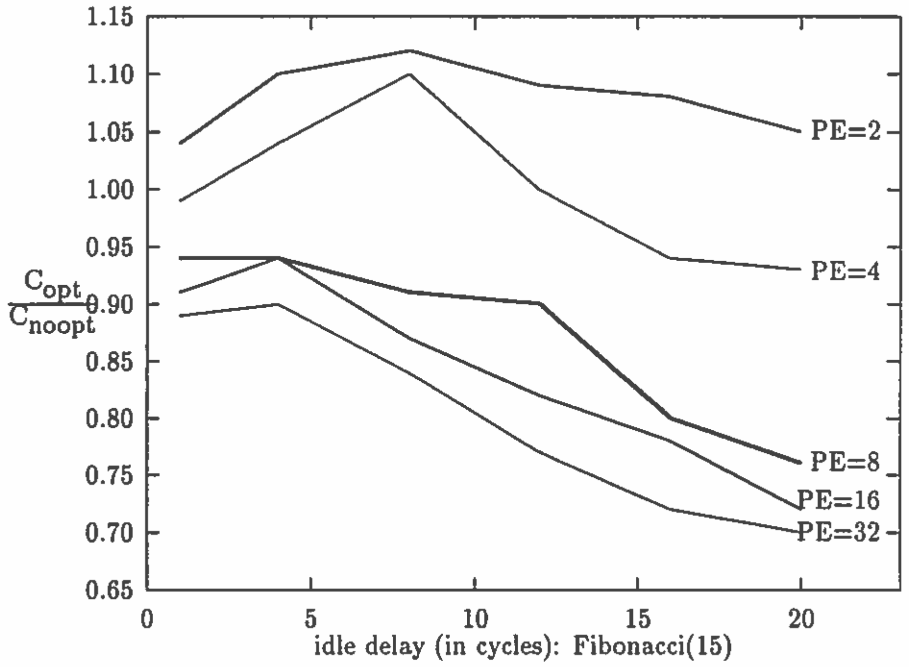

Figure 7: Performance of Fibonacci(15) vs. Number of PEs

This anomaly is due to the increase in suspensions caused by deferring small-grain processes. As seen in Fib (for small numbers of PEs) and Bmatrix (for larger numbers of PEs), certain simulations show degraded performance with increasing idle delay. One factor is nondeterministic scheduling effects. Another factor is that the granularity scheduler makes the *wrong* decision, favoring the execution of large-grain tasks *that depend on smaller-grain tasks.* The naive scheduler also makes these mistakes, but on the whole is more true to the programmer-intended execution order: the static program order usually has producers before (to the left of) consumers.

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. Simulation results illustrate the potential for the analysis to produce speedups in distributed multiprocessors where the penalty for remote task switching is medium to high.[5] Programs with "thick" branches in their process trees achieve the highest utility from granularity information because naive schedulers cannot distinguish between fine-grain work (twigs) and large-grain work (sub-branches). The experiments also uncovered a potential scheduler pitfall that must be avoided: granularity information should not be favored over data-dependency information in a dependent-AND parallel environment.

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 [16], 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 by Debray *et al.* [3, 2, 6]. In addition, we believe that, compared to the above methods, the runtime overhead incurred in exploiting the granularity is reduced.

Empirical results showed that the method (with the three heuristics proposed) correctly estimates time complexity for 85%–91% of all procedures analyzed. These iteration-parameter function heuristics estimate a child procedure's work as a function

[5]Considering that an average reduction takes 3–4 cycles (as defined in our model), an idle delay of 16 cycles represents a communication/computation speed ratio of about 5:1.

of the parent's work. Simulation studies indicate performance improvements can be attained for an on-demand scheduler exploiting the granularity information vs. a naive scheduler. The most utility was gained when interprocessor communication time was high, e.g., modeling a distributed-memory multiprocessor.

Nonetheless, our method may result in an inaccurate estimation for some programs. This is because we use only the call graph to model the program structure, not the data or dependencies. The majority of cases we have observed where our heuristics break down are tree computations, abstractly of multiway branching nodes, however the committed-choice program hides this composition by recursing on intermediate lists or structures. Furthermore, even if the heuristics are correct estimates of complexity, runtime execution may diverge from the estimates, e.g., an unbalanced OR-parallel search tree. An open problem is how to incorporate further information to derive more precise estimations, keeping analysis costs and runtime costs low.

Future work in our granularity analysis includes developing additional heuristics for iteration-parameter functions, to capture a broader class of programs. Development of a more systematic and precise method to solve the derived recurrence equations is needed. It is also necessary to examine the method for more practical programs, performing benchmark testing on a multiprocessor to conclusively demonstrate the method's utility.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award with matching funds generously supplied by Sequent Computer Inc. We thank Jonas Barklund for his extensive comments.

References

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

[2] S. K. Debray and N.-W. Lin. Automatic Complexity Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 599–613, Paris, June 1991. MIT Press.

[3] 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, White Plains, NY, June 1990. ACM Press.

[4] I. P. Goulden and D. Jackson. *Combinational Enumeration.* John Wiley and Sons, Inc., 1983.

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

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

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

[8] C. L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill, New York, 1968.

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

[10] K. Pettis and R. C. Hansen. Profile Guided Code Positioning. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, June 1990. ACM Press.

[11] P. A. Purdom and C. A. Brown. *The Analysis of Algorithms.* Holt, Rinehart and Winston, Inc., New York, 1985.

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

[13] 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.

[14] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[15] 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.

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

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

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

[19] D. W. Wall. Predicting Programming Behavior Using Real or Estimated Profiles. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–70, White Plains, June 1990. ACM Press.

Appendix: Solving the Recurrence Equations

This appendix discusses how to solve the recurrence equations derived in Section 4. Systematic methods are given to solve for equations derived based on different heuristics. In particular, a method is proposed to solve the equations where iteration-parameter functions are derived based on heuristic §3, by converting the recurrence equations into a system of standard linear recurrence equations with multiple indices. These can be solved with standard methods, e.g., generating functions.

To solve for the cost function of p from the recurrence equations derived above, we first simplify the equations by substituting those cost functions whose goals are not mutually recursive with p with their solutions. This is possible since we start from the leaves of G' and move upward. In general, the system of recurrence equations is of the following form for $i = 1, \ldots, m$,

$$C_i(n) = \sum_{j=1}^{m} u_{ij} C_j(I_{ij}(n)) + a_i(n) \tag{5}$$

where $a_i(n)$ is a function of n which is derived from goals which are not mutually recursive with p. If we use heuristic §3, we assume that $I_{ij}(n) = n/k_{ij}$. In general, there is no systematic way to solve an arbitrary system of recurrence equations [8, 11]. However, for these particular recurrence equations, although they are not standard linear recurrence equations, we can convert them into the standard ones by sampling the functions $C_i(n)$. More precisely, let b be the least common multiplier (LCM) of all k_{ij} for all $i, j = 1, \ldots, m$. Furthermore, we factorize b into its prime factors as $b = q_1^{l_1} \ldots q_u^{l_u}$ where $q_i, i = 1, \ldots, u$ are all primes. Let $k_{ij} = q_1^{l_1^{ij}} \ldots q_u^{l_u^{ij}}$ (remember that b is the LCM of all k_{ij}) where $l_h^{ij} \leq l_h$. Let $n = q_1^{n_1} \ldots q_u^{n_u}$ where $n_h \geq l_h$ for $h = 1, \ldots, u$, Equation 5 becomes

$$C_i(q_1^{n_1} \ldots q_u^{n_u}) = \sum_{j=1}^{m} u_{ij} C_j(q_1^{n_1 - l_1^{ij}} \ldots q_u^{n_u - l_u^{ij}}) + a_i(q_1^{n_1} \ldots q_u^{n_u}) \tag{6}$$

Denoting $C_i(q_1^{n_1} \ldots q_u^{n_u})$ as $C_i'(n_1, \ldots, n_u)$, and $a_i(q_1^{n_1} \ldots q_u^{n_u})$ as $a_i'(n_1, \ldots, n_u)$, Equation 6 becomes

$$C_i'(n_1, \ldots, n_u) = \sum_{j=1}^{m} u_{ij} C_i'(n_1 - l_1^{ij}, \ldots, n_u - l_u^{ij}) a_i'(n_1, \ldots, n_u) \tag{7}$$

Equation 7 is a standard linear recurrence equation with multiple indices [8]. To solve these equations, we first need to obtain boundary conditions: values of $C'_i(n_1, \ldots, n_u)$ for $n_h = 0, \ldots, l_h$, and $h = 1, \ldots, u$. Such values can be obtained by directly iterating Equations 5 from $n = 0, 1, \ldots, b$. This is feasible since in practice, b is usually not very big.

After the boundary conditions have been obtained, we can use the generating function technique to solve for C'_i. Formally, let $Z_1, \ldots, Z_u$ be the symbolic parameters, define the generating function $D(Z_1, \ldots, Z_u)$ of an arbitrary function $D(n_1, \ldots, n_u)$ as follows,

$$D(Z_1, \ldots, Z_u) = \sum_{n_1=0,\ldots,n_u=0}^{\infty} D(n_1, \ldots, n_u) Z_1^{n_1} \ldots Z_u^{n_u}$$

Multiplying $Z_1^{n_1} \ldots Z_u^{n_u}$ and summing both sides of Equation 7 for $n_i \geq l_i$, we obtain

$$\begin{aligned} C'_i(Z_1, \ldots, Z_u) - B_i(Z_1, \ldots, Z_u) &= \sum_{j=1}^{m} u_{ij} Z_1^{l_1^{ij}} \ldots Z_u^{l_u^{ij}} (C'_j(Z_1 \ldots Z_u) - B_j(Z_1, \ldots, Z_u)) \\ &\quad + A'_i(Z_1 \ldots Z_u) - A''_j(Z_1, \ldots, Z_u) \end{aligned}$$

Where $B_i(Z_1, \ldots, Z_u), A''_i(Z_1, \ldots, Z_u)$ can be obtained from the boundary values of C'_i and A''_i respectively. This results in a system of linear equations for generating functions. Such generating functions can be solved by linear algebra techniques. Furthermore, the solutions to these generating functions will have forms as $P(Z_1, \ldots, Z_u)/Q(Z_1, \ldots, Z_u)$ where both P and Q are polynomial functions of $Z_1, \ldots, Z_u$. Closed forms of C'_i can be obtained by standard techniques [4, 11, 8].

After we obtain $C'_i(n_1, \ldots, n_u)$, we can obtain $C_i(n)$ for any n by approximation. First, notice an important property for $C_i(n)$: $C_i(n)$ is a monotonically nondecreasing function. Since all we need is a conservative upper bound for $C_i(n)$, any $C'_i(n_1, \ldots, n_u)$ such that $v = q_1^{n_1} \ldots q_u^{n_u} \geq n$ is fine as an approximation of $C_i(n)$. Obviously, the best approximation is to find $C'_i(n_1, \ldots, n_u)$ such that $v = q_1^{n_1} \ldots q_u^{n_u} \geq n$ is the closest to n. However, since there is no a simple way to find those $n_1, \ldots, n_u$, we sacrifice some precision and approximate $C_i(n)$ by $C'_i(n_1, 0, \ldots, 0)$ where $n_1 = \lceil \log_{q_1}(n) \rceil$ (assuming that $q_1 < q_2 < \ldots < q_u$).

If we adopt our conservative principle and assume that $I(n) = n-1$, we get a system of linear recurrence equations. A system of linear equations of generating functions can be directly obtained from Equation 5. Closed forms for $C_i(n)$ can be derived much more easily.