J. Symbolic Computation (1996) **22**, 715–734

A Methodology for Granularity Based Control of Parallelism in Logic Programs†

P. LOPEZ‡, M. HERMENEGILDO‡ AND S. DEBRAY♮

‡ *Computer Science Faculty, Technical University of Madrid (UPM)*

28660-Boadilla del Monte, Madrid-Spain

pedro@dia.fi.upm.es, herme@fi.upm.es

♮ *Department of Computer Science, University of Arizona*

Tucson, AZ 85721, U.S.A.

debray@cs.arizona.edu

(Received 31 May 1995)

Several types of parallelism can be exploited in logic programs while preserving correctness and efficiency, i.e. ensuring that the parallel execution obtains the same results as the sequential one and the amount of work performed is not greater. However, such results do not take into account a number of overheads which appear in practice, such as process creation and scheduling, which can induce a slow-down, or, at least, limit speedup, if they are not controlled in some way. This paper describes a methodology whereby the granularity of parallel tasks, i.e. the work available under them, is efficiently estimated and used to limit parallelism so that the effect of such overheads is controlled. The run-time overhead associated with the approach is usually quite small, since as much work is done at compile time as possible. Also, a number of run-time optimizations are proposed. Moreover, a static analysis of the overhead associated with the granularity control process is performed in order to decide its convenience. The performance improvements resulting from the incorporation of grain size control are shown to be quite good, specially for systems with medium to large parallel execution overheads.

Keywords: Granularity Analysis and Control, Parallelism, Cost Estimation.

1. Introduction

It has been shown — e.g. by Chassin and Codognet (1994), Hermenegildo and Rossi (1995) — that several types of parallelism can be exploited in logic programs while preserving correctness (i.e. the parallel execution obtains the same results as the sequential) and efficiency (i.e. the amount of work performed is not greater or, at least, there is no

† The work of S. K. Debray was supported in part by the National Science Foundation under grant CCR-9123520. The work of M. Hermenegildo and P. López is supported in part by ESPRIT project 6707 "PARFORCE" and CICYT project number TIC93-0976-CE.

0747–7171/90/000000 + 00 $03.00/0 © 2002 Academic Press Limited

slow-down). However such results assume an idealized execution environment in which a number of practical overheads are ignored, such as those associated with task creation, possible migration of tasks to remote processors, the associated communication overheads, etc. Due to these overheads, and if the granularity of parallel tasks, i.e. the "work available" underneath them, is too small, it may happen that the costs are larger than the benefits in their parallel execution. This makes it desirable to devise a method whereby the granularity of parallel goals and their number can be controlled. Granularity control has been studied in the context of traditional programming (Kruatrachue and Lewis 1988, McGreary and Gill 1989), functional programming (Huelsbergen 1993, Huelsbergen *et al.* 1994), and also logic programming (Kaplan 1988, Debray *et al.* 1990, Zhong *et al.* 1992, Debray and Lin 1993).

The benefits from controlling parallel task size will obviously be greater for systems with greater parallel execution overheads. In fact, in many architectures (e.g. distributed memory multiprocessors, workstation "farms", etc.) such overheads can be very significant and, in them, automatic parallelization cannot in general be done realistically without granularity control. In some other architectures where the overheads for spawning goals in parallel are small (e.g. in small shared memory multiprocessors) granularity control is not essential but it can also achieve important improvements in speedup.

The aim of granularity control is to change parallel execution to sequential execution or vice-versa based on some conditions related to grain size and overheads. However, granularity control itself can induce new overheads, which should obviously be minimized. Since granularity control cannot in general be done completely at compile-time, one way to minimize its impact is to do as much work at compile-time as possible and relegate some tests and final decisions to run-time. One way to do this is by generating at compile-time cost functions which estimate grain size as a function of input data size, which are then evaluated at run-time when such size is known. This was proposed by Debray *et al.* (1990) in the context of logic programs and by Rabhi and Manson (1990) in the context of functional programs. An alternative is to determine only the *relative* cost of goals (Zhong *et al.* 1992), which can be specially useful for optimizing an on-demand run-time scheduler, but may not be as effective in reducing task creation cost. These approaches are in contrast with others, such as that of Sarkar (1989) who bases his algorithm on information obtained via runtime profiling rather than compile-time analysis. Goldberg and Hudak (1985) considers "serial combinators" with reasonable grain sizes, but does not discuss the compile time analysis necessary to estimate the amount of work that may be done by a call.

We address the problem by using the overall approach originally sketched by Debray *et al.* (1990) of computing complexity functions and performing program transformations at compile-time based on such functions, so that the transformed program automatically controls granularity. However, the central topic of such approach was really the problem of estimating *upper* bounds to task execution times, leaving as future work the determination of how that information was to be used. The method described in this paper attempts to fill this gap by illustrating and offering solutions for the many problems involved, for both the cases when upper and lower bound information regarding task granularity is available, and for a generic execution model. Such problems include on one hand estimating the cost of goals, of the overheads associated with their parallel execution, and of the granularity control technique itself. On the other hand there is also the problem of devising, given that information, efficient compile-time and run-time granularity control techniques.

We know of no other work which describes a complete granularity control system for logic programs, discusses the many problems that arise (some of them more subtle than they appear at first sight) and provides solutions to them in the generality that we present our work.

We do not discuss in detail the different types of overheads which may appear in a parallel execution when comparing it to a sequential execution, which may include not only execution time-related overheads but also, for example, memory consumption overheads, for conciseness, and because we are more concerned with speedups, we concentrate mainly on time-related overheads. However, we conjecture that a similar treatment to that which we propose can be applied to the analysis and control of memory-related overheads.

2. A General Model

We start by discussing the basic issues to be addressed in our general approach to granularity control, in terms of a generic execution model. In the following sections we will particularize to the case of logic programs.

2.1. Deriving Sufficient Conditions

We first discuss how conditions for deciding between parallel and sequential execution can be devised. We consider a generic execution model: let $g = g_1, \ldots, g_n$ be a task such that subtasks $g_1, \ldots, g_n$ are candidates for parallel execution, T_s represents the cost (execution time) of the sequential execution of g, and T_i represents the cost of the execution of subtask g_i.

There can be many different ways to execute g in parallel, involving different choices of scheduling, load balancing, etc., each having its own cost (execution time). To simplify the discussion, we will assume that T_p represents in some way all of the possible costs. More concretely, $T_p \leq T_s$ should be understood as "T_s is greater or equal than any possible value for T_p".

In a first approximation, we assume that the points of parallelization of g are fixed. We also assume, for simplicity, and without loss of generality, that no tests — such as, perhaps, "independence" tests (Chassin and Codognet 1994, Hermenegildo and Rossi 1995) — other than those related to granularity control are necessary.

Thus, the purpose of granularity control will be to determine, based on some conditions, whether the g_i's are to be executed in parallel or sequentially. In doing this, the objective is to improve the ratio between the parallel and sequential execution times. An interesting goal is to ensure that $T_p \leq T_s$. In general, this condition cannot be determined before executing g, while granularity control should intuitively be carried out ahead of time. Thus, we are forced to use approximations. At this point one clear alternative is to give up on strictly ensuring that $T_p \leq T_s$ and use some heuristics that have good average case behavior. On the other hand, it is not easy to find such heuristics and, also, it is of obvious practical importance to be able to ensure that parallel execution will not take more time than the sequential one. This suggests an alternative solution: evaluating a simpler condition which nevertheless can be proved to ensure that $T_p \leq T_s$. Such a condition can be based on computing an upper bound for T_p and a lower bound for T_s. Ensuring $T_p \leq T_s$ corresponds to the case where the action taken when the condition does not hold is to run sequentially, i.e. to a philosophy were tasks are executed sequentially unless parallel

execution can be shown to be faster. This is useful when "parallelizing a sequential program." This approach is discussed in the following section. The converse case of "sequentializing a parallel program", in which detecting when the opposite condition $T_s \leq T_p$ holds is the objective, is considered in Section 2.1.2.

2.1.1. Parallelizing a Sequential Program

In order to derive a sufficient condition for the inequality $T_p \leq T_s$, we derive upper bounds for its left-hand-side and lower bounds for its right-hand-side, i.e. a sufficient condition for $T_p \leq T_s$ is $T_p^u \leq T_s^l$, where T_p^u denotes an upper bound of T_p and T_s^l a lower bound of T_s. We will use the superscripts l and u to denote lower and upper bounds respectively throughout the discussion.

Assume that there are p free processors in the system at the instant in which task g is about to be executed. Assume also that $p \geq 2$ (if there is only one processor, then execution is performed sequentially) and let m be the lowest integer which is greater or equal than n/p, i.e. the ceiling of $\frac{n}{p}$, denoted $m = \lceil \frac{n}{p} \rceil$. We have that $T_p^u = Spaw^u + C^u$, where $Spaw^u$ is an upper bound on the cost of creating the n parallel subtasks, and C^u an upper bound on the execution of g itself. $Spaw^u$ will be dependent on the particular system in which task g is going to be executed. It can be a constant, or a function of several parameters, such as input data size, number of input arguments, number of tasks, etc. and can be experimentally determined. We now consider how C^u can be computed. Let C_i^u be an upper bound on the cost of subtask g_i, and assume that $C_1^u, \ldots, C_n^u$ are ordered in descending order of cost. Two possible ways of computing C^u are the following: $C^u = \sum_{i=1}^{m} C_i^u$; or $C^u = m\ C_1^u$. Each C_i^u can be considered as the sum of two components: $C_i^u = Sched_i^u + T_i^u$, $Sched_i^u$ denotes the time taken from the point in which the parallel subtask g_i is created until its execution is started by a processor (possibly the same processor that created the subtask), i.e. the cost of task preparation, scheduling, communication overheads, etc.† T_i^u denotes the time taken by the execution of g_i disregarding all the overheads mentioned before. We assume that the tasks $g_1, \ldots, g_n$ are guaranteed to not fail. We also assume that T_s^l can be computed as follows: $T_s^l = T_{s_1}^l + \cdots + T_{s_n}^l$, where $T_{s_i}^l$ is a lower bound of the cost of the (sequential) execution of subtask g_i.

The following two lemmas summarize the previous discussion:

Lemma 2.1. *If* $Spaw^u + \sum_{i=1}^{m} C_i^u < T_{s_1}^l + \cdots + T_{s_n}^l$, *then* $T_p \leq T_s$.

Proof. Trivial. □

Lemma 2.2. *If* $Spaw^u + m\ C_1^u < T_{s_1}^l + \cdots + T_{s_n}^l$ *then* $T_p \leq T_s$

Proof. Trivial. □

As mentioned in the introduction, bounds on execution costs often need to be evaluated totally or partially at run-time, and thus also the condition above. It would be desirable

† Note that in some parallel systems, such as &-Prolog (Hermenegildo and Greene 1991), $Sched_i^u$ can in some cases be zero, since there is no overhead associated with the preparation of a parallel task if it is executed by the same processor as the one which created the task.

to make this evaluation be as efficient as possible. There is clearly a tradeoff between the evaluation cost of such a sufficient condition and its accuracy. A sufficient condition with a simpler evaluation than those in lemmas 2.1 and 2.2 is given below, based on a series of reasonable further assumptions.

Assume that it is ensured that the tasks $g_1, \ldots, g_n$ will not take longer than they would in a sequential execution, ignoring the time to spawn them and all the associated parallel execution overheads† and that $Sched_1^u, \ldots, Sched_n^u$ are ordered in descending order of cost. Let $Thres^u$ be a threshold computed using either one of the following expressions: $Thres^u = Spaw^u + m\ Sched_1^u$; or $Thres^u = Spaw^u + \sum_{i=1}^{m} Sched_i^u$.

THEOREM 2.3. *If there exist at least $m+1$ tasks $t_1, \ldots, t_{m+1}$ among $g_1, \ldots, g_n$, such that for all i, $1 \leq i \leq (m+1)$, $Thres^u \leq Ts_{t_i}^l$, where $Ts_{t_i}^l$ denotes a lower bound of the cost of the sequential execution of task t_i, then $T_p \leq T_s$.*

PROOF. Assume that $T_{s_1}, \ldots, T_{s_n}$ are ordered in descending order of cost, where T_{s_i} denotes the cost of the sequential execution of task g_i. Consider the following condition:

$$T_p^u \leq T_{s_1} + \cdots + T_{s_m} + T_{s_{m+1}} + \cdots + T_{s_n} \tag{2.1}$$

where $T_p^u = Thres^u + T_{s_1} + \cdots + T_{s_m}$. We have that if this condition holds then $T_p \leq T_s$, since its left hand side is an upper bound of T_p. Simplifying condition 2.1 we obtain:

$$Thres^u \leq T_{s_{m+1}} + \cdots + T_{s_n} \tag{2.2}$$

If there are at least $m+1$ tasks $t_1, \ldots, t_{m+1}$ among $g_1, \ldots, g_n$, such that for all i, $1 \leq i \leq (m+1)$, $Thres^u \leq Ts_{t_i}^l$, then $Thres^u \leq Ts_{t_i}$ (where Ts_{t_i} denotes the cost of the sequential execution of task t_i), and there is some t_i, $1 \leq i \leq m+1$ which is equal to some g_j, $m+1 \leq j \leq n$ and condition 2.2 holds because $Thres^u \leq T_{s_j}$. □

We treat now a slightly more complex case in which we also consider other costs, including the cost of granularity control itself: assume now that the execution of g_i takes T_i time steps, such that $T_i = T_{s_i} + W_i$, where W_i is some "extra" work due to either parallel execution itself (for example the cost of accessing remote references) or granularity control or both of them. Let l $(0 \leq l \leq n)$ be the tasks for which we know that $W_i \neq 0$ (equivalently, $T_i > T_{s_i}$). Assume that $W_1^u, \ldots, W_l^u$ are ordered in descending order of cost, and let $r = min(l, m)$. Then, we can compute a new threshold, $Thres_w^u$, by adding W $(Thres_w^u = Thres^u + W)$ to the previous threshold $(Thres^u)$. W can be computed in two possible ways: $W = \sum_{i=1}^{r} W_i^u$; or $W = r\ W_1^u$.

THEOREM 2.4. *If there exist at least $m+1$ tasks $t_1, \ldots, t_{m+1}$ among $g_1, \ldots, g_n$, such that for all i, $1 \leq i \leq (m+1)$, $Thres_w^u \leq Ts_{t_i}^l$, where $Ts_{t_i}^l$ denotes a lower bound of the cost of the sequential execution of task t_i, then $T_p \leq T_s$.*

PROOF. The proof is similar to that of theorem 2.3. Since $Thres^u + W + T_{s_1} + \cdots + T_{s_m}$, is also an upper bound of T_p, we can follow the same argument in this proof replacing condition 2.1 by $Thres^u + W + T_{s_1} + \cdots + T_{s_m} \leq T_{s_1} + \cdots + T_{s_m} + T_{s_{m+1}} + \cdots + T_{s_n}$ □

† This can be ensured for certain execution platforms, for example if the tasks are "independent". However in some cases, if the tasks are "dependent", they may take longer than they would in a sequential execution.

Suppose now that we cannot ensure that for all i, $1 \leq i \leq n$, g_i is not going to fail. Assume that g_k is the leftmost task for which non-failure is not ensured, for some $1 \leq k \leq n$. We can modify the previous lemmas (2.1 and 2.2) and theorems (2.3 and 2.4) slightly as follows.

Lemmas 2.1 and 2.2 can be reformulated as:

LEMMA 2.5. *If* $Spaw^u + \sum_{i=1}^{m} C_i^u < T_{s_1}^l + \cdots + T_{s_k}^l$, *then* $T_p \leq T_s$.

PROOF. Trivial. □

LEMMA 2.6. *If* $Spaw^u + m\ C_1^u < T_{s_1}^l + \cdots + T_{s_k}^l$ *then* $T_p \leq T_s$

PROOF. Trivial. □

The only difference is that we consider $T_{s_1}^l + \cdots + T_{s_k}^l$ on the right hand side of the respective inequation instead of $T_{s_1}^l + \cdots + T_{s_n}^l$.

Theorems 2.3 and 2.4 can be reformulated by assuming as hypothesis that the tasks which have the m greatest costs are among $g_1, \ldots, g_k$. The proofs are similar.

THEOREM 2.7. *If there exist at least* $m+1$ *tasks* $t_1, \ldots, t_{m+1}$ *among* $g_1, \ldots, g_k$, *such that for all* i, $1 \leq i \leq (m+1)$, $Thres^u \leq Ts_{t_i}^l$, *where* $Ts_{t_i}^l$ *denotes a lower bound of the cost of the sequential execution of task* t_i, *and the tasks with the* m *greatest costs are among* $g_1, \ldots, g_k$, *then* $T_p \leq T_s$.

THEOREM 2.8. *If there exist at least* $m+1$ *tasks* $t_1, \ldots, t_{m+1}$ *among* $g_1, \ldots, g_k$, *such that for all* i, $1 \leq i \leq (m+1)$, $Thres_w^u \leq Ts_{t_i}^l$, *where* $Ts_{t_i}^l$ *denotes a lower bound of the cost of the sequential execution of task* t_i, *and the tasks with the* m *greatest costs are among* $g_1, \ldots, g_k$, *then* $T_p \leq T_s$.

2.1.2. Sequentializing a Parallel Program

Assume now that we want to detect when $T_s \leq T_p$ holds, because we have a parallel program and want to profit from performing some sequentializations. In this case we can compute T_s^u and T_p^l. Let T_i^l be a lower bound on the execution time of g_i. T_p^l can be determined in several ways:

1 If $n \leq p$ then: $T_p^l = Spaw^l + \max(T_1^l, \ldots, T_n^l)$ else: $T_p^l = Spaw^l + \lceil \frac{n}{p} \rceil \min(T_1^l, \ldots, T_n^l)$.

2 $T_p^l = Spaw^l + \sum_{i=1}^{k} T_i^l$, where $k = \lceil \frac{n}{p} \rceil$ and $T_1^l, \ldots, T_n^l$ are ordered in ascending order.

3 $T_p^l = Spaw^l + \frac{T_{s_1}^l + \cdots + T_{s_n}^l}{p}$

The determination of T_i^l will depend, of course, on the way g_i is going to be executed. If the execution is going to be performed in parallel with no granularity control, with granularity control, or sequentially, we compute $T_{p_i}^l$, $T_{g_i}^l$, or $T_{s_i}^l$ respectively. The determination of $T_{p_i}^l$ and $T_{g_i}^l$ is discussed in Section 8.

We can choose the maximum of the different possibilities for computing T_p^l. In general, if there are n different choices $x_1, \ldots, x_n$ for computing T_p^l (T_p^u, respectively) we will choose $T_p^l = \max(x_1, \ldots, x_n)$ ($T_p^u = \min(x_1, \ldots, x_n)$, respectively).

2.2. Compile-time vs. Run-time Control

The evaluation of the sufficient conditions proposed in the previous sections can in principle be performed totally at run-time, compile-time or partially at each of them. For example, it might be possible to determine at compile time if the condition expressed in Theorem 2.3 will always be true when evaluated at run-time. Let C^l be a lower bound of the cost of each g_i, $1 \leq i \leq n$, then if $Thres^u \leq (n-m)C^l$ the condition of the theorem holds, since $(n-m)C^l$ is a lower bound on $T_{s_{m+1}} + \cdots + T_{s_n}$. Clearly, in this case it is not necessary to perform any granularity control and tasks can always be executed in parallel. The converse case is also possible where tasks can be statically determined to be better executed sequentially. Thus, from the granularity control point of view program parts can be classified as *parallel* (all the performed parallelizations are unconditional), *sequential* (there are no parallel tasks), and *performing granularity control* (tests based on granularity information are performed at run-time in order to decide between parallel or sequential execution). Whether it is done at compile-time or at run-time, in order to perform granularity control two basic issues have to be addressed: how the bounds on the costs and overheads which are the parameters of the sufficient conditions are computed (*cost and overhead analysis*) and how the sufficient conditions are used to control parallelism (*granularity control*). They are the subjects of the following sections. Both of these issues imply in general both compile-time and run-time techniques in our approach.

2.2.1. Task Cost Analysis

Since *task cost* is not in general computable at compile-time, we are forced to resort to approximations and, possibly, to performing some work at run-time. In fact, as pointed out by Debray *et al.* (1990), since the work done by a call to a recursive procedure often depends on the size of its input, such work cannot in general be estimated in any reasonable way at compile time and for such calls some run-time work is necessary. The basic approach used is as follows: given a call p, an expression $\Phi_p(n)$ is computed that a) it is relatively easy to evaluate, and b) it approximates $\texttt{Cost}_p(n)$, where $\texttt{Cost}_p(n)$ denotes the cost of computing p for an input of size n. The idea is that $\Phi_p(n)$ is determined at compile time. It is then evaluated at run-time, when the size of the input is known, yielding an estimate of the cost of the call. We point out that the evaluation of $\Phi_p(n)$ will be simplified as much as possible by the compiler. In many cases it will be possible to simplify the cost function (or, more precisely, the test to be performed) to the point of being able to statically derive a threshold size for one of the input size arguments. In that case, at runtime, such input size is simply compared against the (precomputed) threshold, and thus the function does not need to be evaluated. This simplification is discussed in Section 6.1. If after simplification, the resulting expression is costly to evaluate, the compiler may decide to compute a safe approximation with a smaller evaluation cost. We would also like to point out that the cost of evaluating tests, and, in general, of performing granularity control, is also taken into account, as described in Section 7.

In the following we will refer to the compile-time computed expressions $\Phi_p(n)$ as *cost functions*.

As mentioned in Section 2 the approximation of the condition used to decide between parallelization and sequentialization can be based either on some heuristics or on a *safe* approximation (i.e. an upper or lower bound). For the latter approach we were able to show sufficient conditions for parallel execution while preserving efficiency. Because of these results, we will in general require $\Phi_p(n)$ to be not just an approximation, but also a bound on the actual execution cost. Fortunately, as mentioned before, much work has been presented on (time) complexity analysis of programs (Le Métayer 1988, Wadler 1988, Rosendhal 1989, Bjerner and Holmstrom 1989, Sarkar 1989, Zimmermann and Zimmermann 1989, Flajolet *et al.* 1991). The most directly applicable are the methods presented by Debray and Lin (1993) and Debray *et al.* (1994) for statically estimating cost functions for predicates in a logic program. The two approaches have much in common but they differ in the way the approximation is done. In the first one upper bounds of task costs are computed, that is $(\forall n)\mathtt{Cost}_p(n) \leq \Phi_p(n)$, while in the second one, to be discussed later, the converse approximation is done: $(\forall n)\mathtt{Cost}_p(n) \geq \Phi_p(n)$.

Example 2.1. Consider the procedure `q/2` defined as follows:

```
q([],[]).
q([H|T],[X|Y]):- X is H + 1, q(T,Y).
```

where the first argument is an input argument. Assume that the cost unit is the number of resolution steps. In a first approximation, and for simplicity, we suppose that the cost of a resolution step (i.e., procedure call) is the same as that of the `is/2` builtin. With these assumptions, the cost function of `q/2` is $\mathtt{Cost}_q(n) = 2\ n + 1$, where n is the size (length) of the input list (first argument). □

2.2.2. Parallelization Overhead Analysis

Regarding the determination of the overheads that appear together with the costs in the sufficient conditions of Section 2.1.1, as mentioned there, this is a more or less trivial task in systems where such costs can be considered constant. However, it is often the case that such costs have, in addition to a constant component, other components which can be a function of several parameters, such as input data size, number of input arguments, number of tasks, number of active processors in the system, type of processor, etc., in which case some run-time evaluation will be needed. For example, in a distributed system, task spawning cost is often proportional to data size, since in many models a complete closure (a call plus its arguments) is sent to the remote processor. Thus, the evaluation of the overheads also implies in general the generation at compile-time of a cost function, to be evaluated at run-time when parameters (such as data size in our previous example) are known.

2.2.3. Performing Granularity Control

Let us assume that techniques, such as those described in general terms above, for determining task costs and overheads are given. Then, the remainder of the granularity control task is to devise a way to actually compute such costs and then control task creation using such information.

We take again the approach of doing as much of the work as possible at compile-time. We propose performing a transformation of the program in such a way that the cost computations and spawning decisions are encoded in the program itself, and in the most efficient way possible. The idea is to postpone the actual computations and decisions until run-time when the parameters missing at compile-time, such as data sizes or processor load, are available. In particular, the transformed programs will perform the following tasks: compute input data sizes; use those sizes to evaluate the cost functions; estimate the spawning and scheduling overheads; decide whether to schedule tasks in parallel or sequentially; decide whether granularity control should be continued or not, etc.

3. Cost Analysis in Logic Programming

We now further discuss the cost analysis problem in the context of logic programs. We distinguish between the cases of and-parallelism and or-parallelism.

3.1. Cost Analysis for And-parallelism

In (goal level) and-parallelism the units being parallelized are goals. We have developed a lower bound goal cost analysis (which also includes a non-failure analysis) which we briefly sketch — see the work of Debray *et al.* (1994) for details. The problem when estimating lower bounds is that in general it is necessary to account for the possibility of failure of head unification, leading a naive analysis to always derive a trivial lower bound of 0. Given (an upper approximation of) mode and type information, the analysis can detect procedures and goals which can be guaranteed not to fail. The technique is based on an intuitively very simple notion, that of a (set of) tests "covering" the type of a variable. Conceptually, we can think of a clause as consisting of a set of primitive tests on the actual parameters of the call, followed by body goals. The tests at the beginning determine whether the clause should be executed or not, and in general may involve pattern matching, arithmetic tests, type tests, etc. A type refers to a set of terms. For any given clause, we refer to the conjunction of the primitive tests that determine whether it will be executed as "the tests of the clause". The disjunction of all the tests of the clauses that define a particular predicate will be referred to as "the test of that predicate." Informally, the test of a predicate covers the type of a variable if binding this variable to any value in the type, the test of the predicate succeeds (the extension of this notion to tuples of variables is straightforward).

An upper-bound cost analysis of goals has been described by Debray and Lin (1993). It is very similar and simpler than that of lower bounds, since the fact that an upper bound on the actual run-time cost is being computed allows assuming that each literal in the body of the clause succeeds and also that all clauses are executed (independently of whether all solutions are required or not).

3.2. Cost analysis for Or-parallelism

The case of or-parallelism is similar to that of and-parallelism except that the units being parallelized are branches of the computation rather than goals. However, the cost analyses of the previous sections can be adapted by simply taking into account the "continuation" of the choice points being considered. As an example, consider a clause $h :- \ldots, L, L_1, \ldots, L_n.$. Assume that the predicate of literal L is p, and the definition

of predicate p contains "c" "eligible" clauses $\{Cl_1, \ldots, Cl_c\}$, where $Cl_i = h_i :- b_i$. In the or-parallel execution of literal L, the "c" choices (each one corresponding to a clause of predicate p) and their continuations (the rest of the L_i, $1 \leq i \leq n$, and the other goals L_{n+1} to L_k that may appear after them in the resolvent at the time L is leftmost) are executed in parallel. Let $Cost_{cl_i}(x)$ and $Cost_{L_i}(x)$ denote the cost of clause Cl_i and literal L_i respectively, then the cost of the choice corresponding to clause Cl_i, denoted by $Cost_{ch_i}$ can be computed as follows: if we are computing lower bounds we have that $Cost^l_{ch_i}(x) = Cost^l_{cl_i}(x) + \sum_{j=1}^{m} Cost^l_{L_j}(x)$, if non-failure is ensured for clause Cl_i and m is the leftmost literal for which non-failure is not ensured; or, alternatively, $Cost^l_{ch_i}(x) = Cost^l_{cl_i}(x)$, if non-failure is not ensured for clause Cl_i. On the other hand, when computing upper bounds we have that $Cost^u_{ch_i}(x) = Cost^u_{cl_i}(x) + \sum_{j=1}^{k} Cost^u_{L_j}(x)$.

The determination of L_{n+1} to L_k, the continuations of the clause under consideration, cannot be obtained directly from the call graph in the presence of last call optimization. The problem is that while non-tail-calls in the body of a procedure return to the caller, because of last call optimization, a tail call does not return to its caller, but rather to the nearest ancestor procedure that made a non-tail call. Thus, while for non-tail calls the transfer of control from the caller to the callee and back is evident from the program's call graph, this is not the case for tail calls. To address this problem, given a program we construct a context-free grammar as follows: for each program clause

$$p(\bar{t}) \ :- \ \mathit{Guard} \mid q_1(\bar{t}_1), \ldots, q_n(\bar{t}_n)$$

the grammar contains a production

$$p \longrightarrow q_1 \ \mathtt{L}_1 \ q_2 \ \mathtt{L}_2 \ \ldots \ \mathtt{L}_{n-1} \ q_n$$

, where the $\mathtt{L}_i$, which are labels corresponding to procedure continuations, are the terminal symbols of the grammar. We then compute FOLLOW sets for this grammar (Aho *et al.* 1986): for any predicate p, FOLLOW(p) gives the set of possible continuations for p.

4. Granularity Control in Logic Programming: the And-Parallelism Case

We use an example to explain the basic program transformation intuitively since a formal presentation would unnecessarily make it more complex.†

EXAMPLE 4.1. Consider the predicate `q/2` defined in Example 2.1, the predicate `r/2` defined as follows:

```
r([],[]).
r([X|RX],[X2|RX1]) :- X1 is X * 2, X2 is X1 + 7, r(RX,RX1).
```

and the parallel goal: `..., q(X,Y) & r(X), ...`, in which literals `q(X,Y)` and `r(Z)` are executed in parallel, as described by the & (parallel conjunction) connective (Hermenegildo and Greene 1991).

The cost functions of `q/2` and `r/2` are $\mathtt{Cost}_q(n) = 2\ n + 1$ and $\mathtt{Cost}_r(n) = 3\ n + 1$

† Although presenting the technique proposed in terms of a source-to-source transformation is convenient for clarity and also a viable implementation technique, the transformation can also obviously be implemented at a lower level in order to reduce the run-time overheads involved even further.

respectively. Assume a number of processors $p \geq 2$. According to Theorem 2.3, the previous goal can safely be transformed into the following one:

```
..., length(X, LX), Cost_q is LX*2+1, Cost_r is LX*3+1,
     (Cost_q > 15, Cost_r > 15 -> q(X,Y) & r(X); q(X,Y), r(X)), ...
```

where a value for the threshold ($Thres^u$) of 15 units of computation is assumed, the variables `Cost_q` and `Cost_r` denote the cost of the (sequential) execution of goal `q(X,Y)` and `r(Z)` respectively, and `LX` denotes the length of the list `X`. □

5. Granularity Control in Logic Programming: the Or-Parallelism Case

Consider the clause body $\ldots, L, L_1, \ldots, L_n$. in the example in Section 3.2. This body can be transformed in order to perform granularity control as follows: $\ldots, (cond$ `->` L' `;` $L), L_1, \ldots, L_n$. Where L' is the parallel version of L, and is created by replacing the predicate name of L (p) by another one, say p', such that p' is the parallel version of p, and is obtained from p by replacing predicate name p with p' in all clauses of p. p' is then declared as "parallel" by means of the appropriate directive. If *cond* holds, then the literal L' (parallel version of L) is executed otherwise L is executed.

A problem with the use of a predicate level parallelism directive is that either all or none of its clauses are executed in parallel. Since there can be differences of costs between clauses, this can lead to worse load-balancing, so a better choice can be the use of some declaration which allows us to specify clusters of clauses such that within each cluster clauses are executed sequentially, and the different clusters are executed in parallel. That way, we can have several parallel versions of a predicate, each of them executed if a particular condition holds. This is illustrated in the following example, where a call to `p` in `...,p, q, r.` and predicate `p` are transformed as follows:

```
..., (cond_1 -> p1 ; cond_2 -> p2; p), q, r.

p:- q1, q2, q3.  p1:- q1, q2, q3 //   p2:- q1, q2, q3 //
p:- r1, r2.      p1:- r1, r2 //       p2:- r1, r2.
p:- s1, s2.      p1:- s1, s2.        p2:- s1, s2.
p.               p1.                 p2.
```

Here, the directive `//` declares three clusters for the predicate `p1`: the first and second ones composed of the first and second clauses respectively, and the third cluster composed of the third and fourth clauses (these two clauses are executed or explored sequentially). Also, for the predicate `p2` we have two clusters: the first one composed of the first clause and the second one composed of the second, third and fourth clauses.

6. Reducing Granularity Control Overhead

The transformations proposed inevitably introduce some new overheads in the execution. It would be desirable to reduce this run-time overhead as much as possible. We propose optimizations which include test simplification, improved term size computation, and stopping granularity control, where if it can be determined that a goal will not produce tasks which are candidates for parallel execution, then a version which does not perform granularity control is executed.

In order to discuss the optimizations we need to introduce some terms. We first recall the notion of "size" of a term. Various measures can be used to determine the "size"

of an input, e.g., term-size, term-depth, list-length, integer-value, etc. (Debray and Lin 1993). The measure(s) appropriate in a given situation can generally be determined by examining the operations performed in the program. Let $|\cdot|_m : \mathcal{H} \rightarrow \mathcal{N}_\perp$ be a function that maps ground terms to their sizes under a specific measure m, where $\mathcal{H}$ is the Herbrand universe, i.e. the set of ground terms of the language, and $\mathcal{N}_\perp$ the set of natural numbers augmented with a special symbol $\perp$, denoting "undefined". Examples of such functions are "list_length", which maps ground lists to their lengths and all other ground terms to $\perp$; "term_size", which maps every ground term to the number of constants and function symbols appearing in it; "term_depth", which maps every ground term to the depth of its tree representation; and so on. Thus, $|[\mathtt{a},\mathtt{b}]|_{\text{list_length}} = 2$, but $|f(a)|_{\text{list_length}} = \perp$. We extend the definition of $|\cdot|_m$ to tuples of terms in the obvious way, by defining the function $Siz_m : \mathcal{H}^n \mapsto {\mathcal{N}_\perp}^n$, such that $Siz_m((t_1,\ldots,t_n)) = (|t_1|_m,\ldots,|t_n|_m)$. Let I and I' denote two tuples of terms, Φ a set of substitutions and θ a substitution. We also define the set of states corresponding to a certain clause point as those states whose leftmost goal corresponds to the literal after that program point. We define the set of substitutions at a clause point in a similar way.

DEFINITION 6.1. [*Comp* function] Given a state s_1 corresponding to a clause point p_1, the current substitution θ corresponding to that state, and another clause point p_2, we define $comp(\theta, p_2)$ as the set of substitutions at point p_2 which correspond to states that are in the same derivation as s_1. ■

DEFINITION 6.2. [Directly computable sizes] Consider a set Φ of substitutions at a clause point p_1 and another clause point p_2. $Siz_m(I')$ is directly computable at p_2 from $Siz_m(I)$ with respect to Φ if exists a (computable) function ψ such that for all θ, θ', $\theta \in \Phi$, and $\theta' \in comp(\theta, p_2)$, $Siz_m(I\theta)$ is defined and $Siz_m(I'\theta') = \psi(Siz_m(I\theta))$. ■

DEFINITION 6.3. [Equivalence of expressions] Two expressions E and E' are equivalent with respect to the set of substitutions Φ if for all $\theta \in \Phi$ $E\theta$ yields the same value as $E'\theta$ when evaluated. ■

6.1. TEST SIMPLIFICATION

Informally, we can view test simplification as follows: the starting point is an expression which is a function of the size of a set of terms. We try to find an expression which is equivalent to it but which is a function of a smaller set of terms. Also, we apply standard arithmetic simplifications to this expression. Since this new expression will have less variables, simplification will be easier and the corresponding simplified expression will be less costly to compute.

Let us now formally describe the notion of simplification of expressions. Consider the set of substitutions Φ' at clause point p_2, just before execution of goal g. Assume that we have an expression $E(Siz_m(I'))$ to evaluate at p_2. The objective is to find a program point p_1 and a tuple of terms I such that $Siz_m(I')$ is directly computable at p_2 from $Siz_m(I)$ with respect to Φ with the function ψ, where Φ is the set of substitutions at clause point p_1 and either $p_1 = p_2$ or p_1 precedes p_2 and $E(Siz_m(I'))$ appear after p_1. We have that $E(\psi(Siz_m(I))$ is equivalent to $E(Siz_m(I'))$ with respect to Φ'. Then we can compute an expression E' which is equivalent to $E(\psi(Siz_m(I))$ (by means of simplifications) with

respect to Φ' and its evaluation cost is less than that of $E(\psi(Siz_m(I))$. The following example illustrates this kind of optimization.

EXAMPLE 6.1. Consider the goal `..., q(X,Y) & r(X), ...` in Example 4.1. In this example $I = I' = (X)$; $Siz(I')$ is directly computable from $Siz(I)$ with respect to Φ with ψ, where ψ is the identity function. $Siz(I\theta)$ is defined for all θ in Φ, since X is bound to a ground list. Thus, we have that for all $\theta \in \Phi$ and for all $\theta' \in comp(\theta, p_2)$, $Siz(I'\theta') = \psi(Siz(I\theta))$. $E(Siz(I)) \equiv max(2\ Siz(X)+1, 3\ Siz(X)+1)+15 \leq 2\ Siz(X)+1+3\ Siz(X)+1$. Let us now compute E'. We have that for all $\theta \in \Phi$, $max(2\ Siz(X)+1, 3\ Siz(X)+1) \equiv 3\ Siz(X)+1$, so we have $3\ Siz(X)+1+15 \leq 2\ Siz(X)+1+3\ Siz(X)+1$ which is simplified to $15 \leq 2\ Siz(X)+1$ and then to $7 \leq Siz(X)$ which is E'. Using this expression we get a more efficient transformed program than in Example 4.1:

```
..., length(X, LX), (LX > 7 -> q(X, Y) & r(X) ; q(X, Y), r(X)), ...
```

□

In some cases test simplification avoids evaluating cost functions, so that term sizes are compared directly with some threshold. Assume that we have a test of the form $Cost_p(n) > G$ where G is a number and $Cost_p(n)$ is a monotone cost function on one variable for some predicate p. In this case, a value k can be found such that $Cost_p(k) \leq G$ and $Cost_p(k+1) > G$, so that the expression $Cost_p(n) > G$ can be simplified to $n > k$.

6.2. STOPPING GRANULARITY CONTROL

An important optimization aimed at reducing the cost of granularity control is based on detecting when an invariant holds recursively on the condition to perform parallelization/sequentialization and executing in those cases a version of the predicate which does not perform granularity control and executes in the appropriate way which corresponds to the invariant.

EXAMPLE 6.2. Consider the predicate `qsort/2` defined as follows:

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

The following transformation will perform granularity control based on the condition given in Theorem 2.3 and the detection of an invariant (tests have already been simplified —we omit details— so that the input data sizes are directly compared with a threshold):

```
g_qsort([], []).
g_qsort([First|L1], L2) :-
  partition(First, L1, Ls, Lg),
  length(Ls, SLs), length(Lg, SLg),
  (SLs > 20 -> (SLg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2);
                            g_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))
            ;  (SLg > 20 -> s_qsort(Ls, Ls2) , g_qsort(Lg, Lg2);
                            s_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))),
```

```
  append(Ls2, [First|Lg2], L2).


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

Note that if the input size is less than the threshold (20 units of computation in this case[†]) then a (sequential) version which does not perform granularity control is executed. This is based on the detection of a recursive invariant: in subsequent recursions this goal will not produce tasks with input sizes greater or equal than the threshold, and thus, for all of them, execution should be performed sequentially and obviously no granularity control is needed. Giannotti and Hermenegildo (1991) presented techniques for detecting such invariants. □

6.3. Reducing Term Size Computation Overhead

With regard to term size computation, the standard approach is to explicitly traverse terms, using builtins such as `length`/2. However such computation can also be carried out in other ways which can potentially reduce run-time overhead:

1. In the case where input data sizes to the subgoals in the body that are candidates for parallel execution are directly computable from those in the clause head (an example of this is the classical "Fibonacci" benchmark – see Example 8.1) such sizes can be computed by evaluating an arithmetic operation. Clause heads can supply their input data size through additional arguments.
2. Otherwise term size computation can be simplified by transforming certain procedures in such a way that they compute term sizes "on the fly" (Hermenegildo and López 1995).
3. In the cases where term sizes are compared directly with a threshold it is not necessary to traverse all the terms involved, but rather only to the point at which the threshold is reached.

7. Taking Into Account the Cost of Granularity Control

As a result of the simplifications proposed in the previous sections three different types of specialized versions of a predicate can be generated: sequential, parallel with no granularity control, and parallel with granularity control. In this section we address the issue of how to select among these versions. We can view this as a reconsideration of the original problem of deciding between parallel and sequential execution, addressed in Section 2, but where we add the new issue of deciding whether to perform granularity control or not. Let T_s, T_p, and T_g denote the execution time of the sequential, parallel, and granularity control versions for the predicate corresponding to a given call. The original problem tackled in Section 2 can be viewed as determining $min(T_s, T_p, T_g)$. Again, this

[†] This threshold is determined experimentally, by taking the average value resulting from several runs.

is not computable ahead of the execution of the goals and we are once more forced to compute an approximation based on heuristics or sufficient conditions. We again take the latter approach, i.e. using sufficient conditions, which we would in principle try to compute for each of the six possible cases involved: $T_g \leq T_s$, $T_p \leq T_s$, $T_p \leq T_g$, $T_s \leq T_g$, $T_s \leq T_p$ and $T_g \leq T_p$. Since we can only approximate these conditions an important issue is the decision taken when none of such conditions can be proved to hold. One solution is to have a pre-determined order relation which is used unless another relation can be proven to be true. This corresponds to the two cases of "sequentializing by default" or "parallelizing by default" studied in Section 2, where only one condition was considered. For example, a default ordering might be: $T_g \leq T_s \leq T_p$, which essentially expresses a default assumption that the optimal execution time is achieved when execution is performed in parallel with granularity control unless the contrary is proven. Goals are also executed sequentially unless parallel execution is proven to take less time. If the "no-slowdown" condition is to be enforced, i.e. it is required that the sequential execution time not be exceeded, then, in all pre-determined order relations we must have that $T_s \leq T_g$ and $T_s \leq T_p$.

Note that these pre-determined order relations can be partial. In that case at some point a heuristic has to be applied. The order between two costs T_1 and T_2 can then be determined as follows: If T_1 and T_2 are related in the pre-determined order relation, then compute a sufficient condition to prove the opposite order; otherwise, if some sufficient condition to prove either of the relations $T_1 \leq T_2$ or $T_2 \leq T_1$ holds then we choose the corresponding one; otherwise the order can be determined by means of some heuristics. A good heuristic can be to use the average of the lower and upper bound which are already computed or take the average of the computed costs of the different clauses of a predicate.

8. Determining T_p and T_g of a call

The determination of a bound for T_s has already been addressed in the previous sections. There, T_p was simply assumed to be the same as T_s, taking as its approximation the opposite bound to that used for T_s. We now address the issue of determining T_p more precisely and also determining T_g. For conciseness, we present the techniques by means of an example.

EXAMPLE 8.1. Let us consider a transformed version `gfib/2` of the `fib/2` predicate which performs run-time granularity control:

```
gfib(0, 0).
gfib(1, 1).
gfib(N, F):- N1 is N - 1, N2 is N - 2,
             (N > 15 -> gfib(N1, F1) & gfib(N2, F2)
                      ; sfib(N1,F1), sfib(N2,F2)),
              F is F1 + F2.

sfib(0, 0).
sfib(1, 1).
sfib(N, F):- N > 1, N1 is N - 1, N2 is N - 2,
              sfib(N1, F1), sfib(N2, F2),
              F is F1+F2.
```

□

8.1. Cost of parallel execution without granularity control: T_p

8.1.1. Upper bounds

In general it is difficult to give a non-trivial upper bound on the cost of the parallel execution of a given set of tasks, since it is difficult to predict the number of free processors that will be available to them at execution time. Note that a trivial upper bound can be computed in some cases by assuming that all the potentially parallel goals are created as separate tasks but they are all executed by one processor.

Consider the predicate `fib/2` defined in Example 8.1. Let Is denote the size of the input (first argument) and $T_p(Is)$ the cost of the parallel execution without granularity control of a call to predicate `fib/2` for an input of size Is. The following difference equation can be set up for the recursive clause of `fib/2`: $T_p^u(Is) = C_b^u(Is) + Spaw^u(Is) + Sched^u(Is) + T_p^u(Is-1) + T_p^u(Is-2) + C_a^u(Is)$ for $Is > 1$, where $C_b(Is)$ and $C_a(Is)$ represent the costs of the sequential execution of the literals before and after the parallel call respectively, that is, $C_b(Is)$ represents the cost of `N1 is N-1,N2 is N-2` and $C_a(Is)$ the cost of `F is F1+F2`. The solution to this difference equation gives the cost of a call to `fib/2` for an input of size Is. The following boundary conditions for the equation are obtained from the base cases: $T_p^u(0) = 1$ and $T_p^u(1) = 1$.

8.1.2. Lower bounds

A trivial lower bound — taking non-failure into account, as discussed by Debray *et al.* (1994) — can be computed as follows: $T_p^l(Is) = \frac{W_p^l(Is)}{p}$, where W_p^l represents the work performed by the parallel execution with no granularity control of a call to predicate `fib/2` for an input of size Is, and can be computed by solving the following difference equation: $W_p^l(Is) = C_b^l(Is) + Spaw^l(Is) + Sched^l(Is) + W_p^l(Is-1) + W_p^l(Is-2) + C_a^l(Is)$ for $Is > 1$, with the boundary conditions: $W_p^l(0) = 1$ and $W_p^l(1) = 1$.

As an alternative, another value for $T_p^l(Is)$ can be obtained by solving the following difference equation: $T_p^l(Is) = C_b^l(Is) + Spaw^l(Is) + Sched^l(Is) + T_p^l(Is-1) + C_a^l(Is)$ for $Is > 1$, with the boundary conditions: $T_p^l(0) = 1$ and $T_p^l(1) = 1$. In this case, an infinite number of processors is considered. Since in each "fork" there are two branches, the longest of them ($T_p^u(Is-1)$) is chosen.

8.2. Cost of the execution with granularity control: T_g

8.2.1. Upper bounds

The following difference equation can be set up for the recursive clause of `fib/2`: $T_g^u(Is) = C_b^u(Is) + Test^u(Is) + Spaw^u(Is) + Sched^u(Is) + T_g^u(Is-1) + T_g^u(Is-2) + C_a^u(Is)$ for $Is > 15$. We assume that all the potentially parallel goals are created as separate tasks but they are all executed by one processor, as is done in Section 8.1.1.

For a call with $Is = 15$ there is no overhead associated with parallel execution since it is performed sequentially, so that the following boundary conditions are obtained: $T_g^u(15) = Test^u(15) + T_s^u(15)$; and $T_g^u(Is) = T_s^u(15)$ for $Is \leq 15$, where $T_s^u(15)$ denotes

an upper bound on the sequential execution time of a call to `fib/2` with an input of size 15.

8.2.2. Lower bounds

A trivial lower bound (taken non-failure into account) can be computed as follows: $T_g^l(Is) = \frac{W_g^l(Is)}{g}$, where W_g^l represents the work performed by the execution with granularity control of a call to `fib/2` for an input of size Is, which can be computed by solving the following difference equation: $W_g^l(Is) = C_b^l(Is) + Test^l(Is) + Spaw^l(Is) + Sched^l(Is) + W_g^l(Is-1) + W_g^l(Is-2) + C_a^l(Is)$ for $Is > 15$, with the boundary conditions: $W_g^l(15) = Test^l(15) + T_s^l(15)$, and $W_g^l(Is) = T_s^l(15)$ for $Is \leq 15$, where $T_s^l(15)$ denotes a lower bound on the sequential execution time of a call to `fib/2` with an input of size 15.

Another value for $T_g^l(Is)$ can be obtained by solving the following difference equation: $T_g^l(Is) = C_b^l(Is) + Test^l(Is) + Spaw^l(Is) + Sched^l(Is) + T_g^l(Is-1) + C_a^l(Is)$ for $Is > 15$, with the boundary conditions: $T_g^l(15) = Test^l(15) + T_s^l(15)$, and $T_g^l(Is) = T_s^l(15)$ for $Is \leq 15$.

Table 1. Experimental results for benchmarks on &-Prolog.

programs	seq	ngc	gc	gct	gcts	gctss
fib(19)	1.839	0.729	1.169	0.819	0.819	0.549
(**O**=m)		1	-60%	-12%	-12%	+24%
fib(19)	1.839	0.970	1.389	1.009	1.009	0.639
(**O**=5)		1	-43%	-4.0%	-4.0%	+34%
hanoi(13)	6.309	2.509	2.829	2.419	2.399	2.399
(**O**=m)		1	-12.8%	+3.6%	+4.4%	+4.4%
hanoi(13)	6.309	2.690	2.839	2.439	2.419	2.419
(**O**=5)		1	-5.5%	+9.3%	+10.1%	+10.1%
unb_matrix	2.099	1.009	1.339	1.259	0.870	0.870
(**O**=m)		1	-32.71%	-24.78%	+13.78%	+13.78%
unb_matrix	2.099	1.039	1.349	1.269	0.870	0.870
(**O**=5)		1	-29.84%	-22.14%	+16.27%	+16.27%
qsort(1000)	3.670	1.399	1.790	1.759	1.659	1.409
(**O**=m)		1	-28%	-20%	-19%	-0.0%
qsort(1000)	3.670	1.819	2.009	1.939	1.649	1.429
(**O**=5)		1	-11%	-6.6%	+9.3%	+21%

Table 2. Experimental results for benchmarks on Muse.

programs	seq	ngc	gctss	opt	e_1	e_2
queens(8)	17.019	2.090	1.759	1.702	+15.84 %	+86.83 %
domino(12)	37.049	4.459	4.139	3.705	+7.18 %	+42.43 %
series	22.429	7.360	4.860	2.243	+33.97 %	+48.86 %
farmer	17.929	2.170	2.149	1.793	+0.97 %	+5.57 %

9. Experimental Results

We have developed a partial prototype of a granularity control system based on the ideas presented. The current prototype has some shortcomings: it only covers the case of (independent, goal level) and-parallelism and the builtin type analyzer is comparatively simple. Despite this, it can achieve effective fully automatic granularity control on three out of the four and-parallel benchmarks (fib, hanoi, and qsort). The results are given in Table 1. For the other benchmarks (unb_matrix) and for or-parallelism we have hand-annotated the programs following the algorithms presented and assuming state of the art type inference technology. The results are given in Tables 1 and 2. We believe that by completing the prototype implementation, and incorporating existing analysis technology, the development of a fully automatic granularity control system is possible, and that our results show that such a system can result in substantial benefit in execution time.

We have first tested the granularity control system with &-Prolog (Hermenegildo and Greene 1991), a parallel Prolog system, on a Sequent Symmetry multiprocessor using 4 processors. Table 1 presents results of granularity control (showing execution times in seconds) for four representative benchmarks and for two levels of task creation and spawning overhead (**O**): minimal (**m**), representing the default overhead found in the &-Prolog shared memory implementation (which is very small – a few microseconds), and an overhead (the &-Prolog system allows adding arbitrary overheads to task creation via a run-time switch) of 5 milliseconds (**5**), which should be representative of a hierarchical shared memory system or of an efficient implementation on a multicomputer with a very fast interconnect. The program `unb_matrix` performs the multiplication of 4×2 and 2×1000 matrices. Results are given for several degrees of optimization of the granularity control process: naive granularity control (`gc`), adding test simplification (`gct`), adding stopping granularity control (`gcts`), and adding "on-the-fly" computation of data size (`gctss`). Results are also given for the sequential execution (`seq`) and the parallel execution without granularity control (`ngc`) for comparison. The obtained speedups have been computed with respect to `ngc`. The importance of the optimizations proposed is underlined by the fact that they result in steadily increasing performance as they are added. Also, except in the case of `qsort` on a very low overhead system, the fully optimized versions show substantial improvements w.r.t. performing no granularity control. Note that the situations studied are on a small shared memory machine and actually imply

very little parallel task overhead, i.e. the conditions under which granularity control offers the least advantages. Thus the results can be seen as lower bounds on the potential improvement. Obviously, on systems with higher overheads, such as distributed systems, the benefits can be much larger.

Regarding or-parallelism, Table 2 presents results of granularity control (showing execution times in seconds) for some benchmarks on the Muse (Ali and Karlsson 1990) system using 10 workers, and running on a Sequent Symmetry multiprocessor with 10 processors. **queens(8)** computes all the solutions to the 8 queens problem. **domino(12)** computes all the legal sequences of 12 dominoes. **series** computes a series whose expression is a disjunction of series. **farmer** is the "farmer, wolf, goat/goose, cabbage/grain" puzzle from ECRC. Results are given for the fully optimized versions which perform granularity control (`gctss`), the sequential execution (`seq`) and the parallel execution without granularity control (`ngc`) for comparison. **opt** is a lower bound on the optimal time, i.e. $\mathbf{opt} = \frac{\mathbf{seq}}{10}$. $\mathbf{e_1} = \frac{\mathbf{ngc-gctss}}{\mathbf{ngc}} \times 100$, and $\mathbf{e_2} = \frac{\mathbf{ngc-gctss}}{\mathbf{ngc-opt}} \times 100$ indicate the percentage of the saved time, with respect to the parallel execution time without granularity control and the ideal parallel execution time respectively, when granularity control is performed. Note that some programs do not exhibit the necessary inherent parallelism to achieve this ideal execution time even if there were no overheads associated with their parallel executions. The reason for introducing these two metrics is that the Muse system showed very good performance in the execution of the selected benchmarks. This is because the Muse scheduler performs an implicit control of parallelism depending on the load of the system. Thus, the potential benefits from applying our granularity control techniques to these benchmarks were more limited. This metric allows us to conclude that our results are in fact quite good, since in general they achieve a significant portion of the potential benefits. Note also that the situations studied are on a small shared memory machine, and, thus, the results, as in the and-parallelism case, can be seen as lower bounds on the potential improvement.

10. Conclusions

We have presented a complete granularity control system for logic programs, discussed the many problems that arise (for both the cases when upper and lower bound information regarding task granularity is available, and for a generic execution model) and provided solutions to them. We believe that the results are general enough to be of interest to researchers working on other parallel languages. We have also assessed the developed granularity control techniques for and-parallelism and or-parallelism on the &-Prolog and Muse systems respectively, and have obtained what we believe are quite encouraging results.

It appears from the sensitivity of the results that we have observed in our experiments that it is not essential to be absolutely precise in inferring the best grain size for a problem: there is a reasonable amount of leeway in how precise this information has to be. This suggests that granularity control can usefully be performed automatically by a compiler.

We can conclude that granularity analysis/control is a particularly promising technique because it has the potential of making feasible to automatically exploit low-cost parallel architectures, such as workstations on a (possibly high speed) local area network.

References

Aho, A.V., Sethi, R., Ullman, J. D. (1986). Compilers – Principles, Techniques and Tools. *Addison-Wesley*.

Ali, K.A.M., Karlsson, R. (1990). The Muse Or-Parallel Prolog Model and its Performance. 1990 North American Conference on Logic Programming, 757–776, MIT Press.

Bjerner, B., Holmstrom, S. (1989). A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, ACM Press, 157–165.

Chassin, J., Codognet, P. (1994) Parallel Logic Programming Systems. Computing Surveys, ACM, 26(3):295–336.

Debray, S. (1994). Inference of Procedure Return Points in the Presence of Last Call Optimization. Manuscript.

Debray, S., López, P., Hermenegildo, M., Lin, N. (1994). Lower Bound Cost Estimation for Logic Programs. Technical Report TR Number CLIP4/94.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, March.

Debray, S., Lin, N., Hermenegildo. M. (1990). Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, 174–188. ACM Press, June.

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

Flajolet, P., Salvy, B., Zimmermann, P. (1991). Automatic Average-Case Analysis of Algorithms. *Theor. Comp. Sci.*, (79):37–109.

Giannotti, F., Hermenegildo, M. (1991). A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, 323–335. Springer-Verlag.

Goldberg, B., Hudak, P. (1985). Serial Combinators: Optimal Grains of Parallelism. In *Proc. Functional Programming Languages and Computer Architecture, Nancy, France*, 201, 382–399. Springer-Verlag LNCS, Aug.

Hermenegildo, M., López, P. (1995). Efficient Term Size Computation for Granularity Control. In Proc. of the Twelfth International Conference on Logic Programming. The MIT Press.

Hermenegildo, M., Greene, K. (1991). The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257.

Hermenegildo, M., Rossi, F. (1995). Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45.

Huelsbergen, L. (1993). Dynamic Language Parallelization. Technical Report 1178, Computer Science Dept. Univ. of Wisconsin, September.

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

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

Kruatrachue, B., Lewis, T. (1988). Grain Size Determination for Parallel Processing. *IEEE Software*, January.

McGreary, C., Gill, H. (1989). Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32.

Le Métayer, D. (1988). ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), April.

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

Rosendhal, M. (1989). Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York.

Sarkar, V. (1989). *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, Massachusetts.

Wadler, P. (1988). Strictness analysis aids time analysis. In *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 119–132. ACM Press.

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

Zimmermann, P., Zimmermann, W. (1989). The Automatic Complexity Analysis of Divide-and-Conquer Programs. Res. Rep. 1149, INRIA, France, December.