Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System

M. Hermenegildo[1,2], E. Albert[3], P. López-García[1], and G. Puebla[1]

[1] School of Comp. Sci., Technical U. of Madrid
{herme,pedro,german}@fi.upm.es
[2] Depts. of Comp. Sci. and Elec. and Comp. Eng., U. of New Mexico (UNM)
herme@unm.edu
[3] School of Comp. Sci., Complutense U. of Madrid
elvira@sip.ucm.es

Abstract. Distributed parallel execution systems speed up applications by splitting tasks into processes whose execution is assigned to different receiving nodes in a high-bandwidth network. On the distributing side, a fundamental problem is grouping and scheduling such tasks such that each one involves sufficient computational cost when compared to the task creation and communication costs and other such practical overheads. On the receiving side, an important issue is to have some assurance of the correctness and characteristics of the code received and also of the kind of load the particular task is going to pose, which can be specified by means of *certificates*. In this paper we present in a tutorial way a number of general solutions to these problems, and illustrate them through their implementation in the Ciao multi-paradigm language and program development environment. This system includes facilities for parallel and distributed execution, an assertion language for specifying complex programs properties (including safety and resource-related properties), and compile-time and run-time tools for performing automated parallelization and resource control, as well as certification of programs with resource consumption assurances and efficient checking of such certificates.

Keywords: resource awareness, granularity control, mobile code certification, distributed execution, GRIDs.

1 Introduction

Distributed parallel execution systems speed up applications by splitting tasks into processes whose execution is assigned to different nodes in a high-bandwidth network. GRID systems [12] in particular attempt to use for this purpose widely

distributed sets of machines, often crossing several administrative domain boundaries. Many interesting challenges arise in this context.

A number of now classical problems have to be solved when this process is viewed from the *producer side*, i.e., from the point of view of the machine in charge of starting and monitoring a particular execution of a given application (or a part of such an application) by splitting the tasks into processes whose execution is assigned to different nodes (i.e., *consumers*) on receiving sides of the network. A fundamental problem involved in this process is detecting which tasks composing the application are independent and can thus be executed in parallel. Much work has been done in the areas of parallelizing compilers and parallel languages in order to address this problem. While obviously interesting, herein we will concentrate instead on other issues.

In this sense, a second fundamental problem, and which has also received considerable attention (even if less than the previous one), is the problem of grouping and scheduling such tasks, i.e., assigning tasks to remote processors, and very specially the particular issue of ensuring that the tasks involve sufficient computational cost when compared to the task creation and communication costs and other such practical overheads. Due to these overheads, and if the *granularity* of parallel tasks (i.e., the work necessary for their complete execution) is too small, it may happen that the costs are larger than the benefits of their parallel execution. Of course, the concept of small granularity is relative: it depends on the concrete system or set of systems where parallel programs are running. Thus, a *resource-aware* method has to be devised whereby the granularity of parallel tasks and their number can be controlled. We will call this the *task scheduling and granularity control* problem. In order to ensure that effective speedup can be obtained from remote execution it is obviously desirable to devise a solution where load and task distribution decisions are made automatically, specially in the context of non-embarrassingly parallel and/or irregular computations in which hand-coded approaches are difficult and tedious to apply.

Interestingly, when viewed from the *consumer side*, and in an open setting such as that of the GRID and other similar overlay computing systems, additional and novel challenges arise. In more traditional distributed parallelism situations (e.g., on clusters) receivers are assumed to be either dedicated and/or to trust and simply accept (or take, in the case of work-stealing schedulers) available tasks. In a more general setting, the administrative domain of the receiver can be completely different from that of the producer. Moreover, the receiver is possibly being used for other purposes (e.g., as a general-purpose workstation) in addition to being a party to the distributed computation. In this environment, interesting security- and resource-related issues arise. In particular, in order to accept some code and a particular task to be performed, the receiver must have some assurance of the *correctness and characteristics of the code received* and also of *the kind of load the particular task is going to pose*. A receiver should be free to reject code that does not adhere to a particular *safety policy* involving more traditional safety issues (e.g., that it will not write on specific areas of the disk) or *resource*-related issues (e.g., that it will not compute for more than

a given amount of time, or that it will not take up an amount of memory or other resources above a certain threshold). Although it is obviously possible to interrupt a task after a certain time or if it starts taking too much memory, this will be wasteful of resources and require recovery measures. It is clearly more desirable to be able to detect these situations a priori.

Recent approaches to mobile code safety involve associating safety information in the form of a *certificate* to programs [28, 21, 26, 1]. The certificate (or proof) is created at compile time, and packaged along with the untrusted code. The consumer who receives or downloads the code+certificate package can then run a *verifier* which by a straightforward inspection of the code and the certificate, can verify the validity of the certificate and thus compliance with the safety policy. It appears interesting to devise means for certifying security by enhancing mobile code with certificates which guarantee that the execution of the (in principle untrusted) code received from another node in the network is *safe* but also, as mentioned above, *efficient*, according to a predefined safety policy *which includes properties related to resource consumption*.

In this paper we present in a tutorial way a number of general solutions to these problems, and illustrate them through their implementation in the context of a multi-paradigm language and program development environment that we have developed, `Ciao` [3]. This system includes facilities for parallel and distributed execution, an assertion language for specifying complex programs properties (including safety and resource-related properties), and compile-time and run-time tools for performing automated parallelization and resource control, as well as certification of programs and efficient checking of such certificates.

Our system allows coding complex programs combining the styles of logic, constraint, functional, and a particular version of object-oriented programming. Programs which include logic and constraint programming (CLP) constructs have been shown to offer a particularly interesting case for studying the issues that we are interested in [14]. These programming paradigms pose significant challenges to parallelization and task distribution, which relate closely to the more difficult problems faced in traditional parallelization. This includes the presence of highly irregular computations and dynamic control flow, non-trivial notions of independence, the presence of dynamically allocated, complex data structures containing pointers, etc. In addition, the advanced state of program analysis technology and the expressiveness of existing abstract analysis domains used in the analysis of these paradigms has become very useful for defining, manipulating, and inferring a wide range of properties including independence, bounds on data structure sizes, computational cost, etc.

After first reviewing our approach to solving the granularity control problem using program analysis and transformation techniques, we propose a technique for resource-aware security in mobile code based on safety certificates which express properties related to resource usage. Intuitively, we use the granularity information (computed by the cost analysis carried out to decide the distribution of tasks on the producer side) in order to generate so-called *cost certificates* which are packaged along with the untrusted code. The idea is that the receiving side

can reject code which brings cost certificates (which it cannot validate or) which have too large cost requirements in terms of computing resources (in time and/or space) and accept mobile code which meets the established requirements.

The rest of the paper proceeds as follows. After briefly presenting in Section 2 the basic techniques used for inferring complex properties in our approach, including upper and lower bounds on resource usage, Section 3 reviews our approach to the use of bounds on data structure sizes and computational cost to perform automatic granularity control. Section 4 then discusses our approach to resource-aware mobile code certification. Section 5 finally presents our conclusions.

2 Inferring Complex Properties Including Term Sizes and Costs

In order to illustrate our approach in a concrete setting, we will use `CiaoPP` [15] throughout the paper. `CiaoPP` is a component of the `Ciao` programming environment which performs several tasks including *automated parallelization and resource control*, as well as *certification* of programs, and efficient *checking* of such certificates. `CiaoPP` uses throughout the now well-established technique of *abstract interpretation* [5]. This technique has allowed the development of very sophisticated global static program analyses which are at the same time automatic, provably correct, and practical. The basic idea of abstract interpretation is to infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus obtaining safe approximations of program behavior. The technique allows inferring much richer information than, for example, traditional types. The fact that at the heart of `Ciao` lies an efficient logic programming-based kernel language allows the use in `CiaoPP` of the very large body of approximation domains, inference techniques and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area (see, e.g., [2, 27, 6, 16] and their references) and which are integrated in `CiaoPP`. As a result of this, `CiaoPP` can infer at compile-time, always safely, and with a significance degree of precision, a wide range of *properties* such as data structure shape (including pointer sharing), bounds on data structure sizes, determinacy, termination, non-failure, bounds on resource consumption (time or space cost), etc.

All this information is expressed by the compiler using *assertions*: syntactic objects which allow expressing "abstract"—i.e. symbolic—properties over different abstract domains. In particular, we use the high-level assertion language of [29], which actually implements a two-way communication with the system: it allows providing information to the analyzer as well as representing its results.

As a very simple example, consider the following procedure `inc_all/2`, which increments all elements of a list by adding one to each of them (we use functional notation for conciseness):

```
inc_all([])    := [].
inc_all([H|T]) := [ H+1 | inc_all(T)].
```

Assume that analysis of the rest of the program has determined that this procedure will be called providing a list of numbers as input. The output from CiaoPP for this program then includes the following assertion:

```
:- true pred inc_all(A,B)
         : ( list(A,num), var(B) )
        => ( list(A,num), list(B,num), size_lb(B,length(A))
         + ( not_fails, is_det, steps_lb(2*length(A)+1)).
```

Such "true pred" assertions specify in a combined way properties of both: ":" the entry (i.e., upon calling) and "=>" the exit (i.e., upon success) points of all calls to the procedure, as well as some global properties of its execution. The assertion expresses that procedure inc_all will produce as output a list of numbers B, whose length is at least (size_lb) equal to the length of the input list, that the procedure will never fail (i.e., an output value will be computed for any possible input), that it is deterministic (only one solution will be produced as output for any input), and that a lower bound on its computational cost (steps_lb) is 2 $length(A) + 1$ execution steps (where the cost measure used in the example is the number of procedure calls, but it can be any other arbitrary measure). This simple example illustrates type inference, non-failure and determinism analyses, as well as lower-bound argument size and computational cost inference. The same cost and size results are actually obtained from the upper bounds analyses (indicating that in this case the results are exact, rather than approximations). Note that obtaining a non-infinite upper bound on cost also implies proving *termination* of the procedure.

As can be seen from the example, in our approach cost bounds (upper or lower) are expressed as functions on the sizes of the input arguments and yield bounds on the number of execution steps required by the computation. Various measures are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. Types, modes, and size measures are first automatically inferred by the analyzers and then used in the size and cost analysis.

While it is beyond the scope of this paper to fully explain all the (generally abstract interpretation-based) techniques involved in this process (see, e.g., [15, 10, 11] and their references), we illustrate through a simple example the fundamental intuition behind our lower bound cost estimation technique.

Consider again the simple inc_all procedure above and the assumption that type and mode inference has determined that it will be called providing a list of numbers as input. Assume again that the cost unit is the number of procedure calls. In a first approximation, and for simplicity, we also assume that the cost of performing an addition is the same as that of a procedure call. With these assumptions the exact cost function of procedure inc_all is $\texttt{Cost}_{inc\_all}(n) = 2\ n + 1$, where n is the size (length) of the input list.

In order to obtain a lower bound approximation of the previous cost function, CiaoPP's analyses first determine, based on the mode and type information inferred, that the argument size metric to be used is list length. An interesting problem with estimating lower bounds is that in general it is necessary to account for the possibility of failure of a call to the procedure (because of, e.g., an

inadmissible argument) leading to a trivial lower bound of 0. For this reason, the lower bound cost analyzer uses information inferred by non-failure analysis [9], which can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution (which would indeed be the case for `inc_all`) or not terminate.

In general, in order to determine the work done by (recursive) clauses, it is necessary to be able to estimate the size of input arguments in the procedure calls in the body of the procedure, relative to the sizes of the input arguments. For this, we use an abstraction of procedure definitions called a data dependency graph. Our approach to cost analysis consists of the following steps:

1. Use data dependency graphs to determine the relative sizes of variable bindings at different program points.
2. Use the size information to set up difference equations representing the computational cost of procedures
3. Compute lower/upper bounds to the solutions of these difference equations to obtain estimates of task granularities.

The size of an output argument in a procedure call depends, in general, on the size of the input arguments in that call. For this reason, for each output argument we use an expression which yields its size as a function of the input data sizes. For the `inc_all` procedure let $\mathtt{Size}^2_{\mathtt{inc\_all}}(n)$ denote the size of the output argument (the second) as a function of the size of its first (input) argument n. Once we have determined that the size measure to use is list length, and the size relationship which says that the size of the input list to the recursive call is the size of the input list of the procedure head minus one, the following difference equation can be set up for `inc_all`/2:

$\mathtt{Size}^2_{\mathtt{inc\_all}}(0) = 0$ (boundary condition from base case),
$\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = 1 + \mathtt{Size}^2_{\mathtt{inc\_all}}(n-1)$.

The solution of this difference equation obtained is $\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = n$.

Let $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{p}}(n)$ denote a lower bound on the cost (number of resolution steps) of a call to procedure `p` with an input of size n. Given all the assumptions above, and the size relations obtained, the following difference equation can be set up for the cost of `inc_all`/2:

$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(0) = 1$ (boundary condition from base case),
$\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 1 + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n-1)$.

The solution obtained for this difference equation is $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 2\,n+1$. In this case, the lower bound inferred is the exact cost (the upper bound cost analysis also infers the same function). In our approach, sometimes the solutions of the difference equations need to be in fact approximated by a lower bound (a safe approximation) when the exact solution cannot be found. The upper bound cost estimation case is very similar to the lower bound one, although simpler, since we do not have to account for the possibility of failure.

3 Controlling Granularity in Distributed Computing

As mentioned in Section 1, and in view of the techniques introduced in Section 2, we now discuss the task scheduling and granularity control problem, assuming that the program is already parallelized.[4] The aim of such distributed granularity control is to replace parallel execution with sequential execution or vice-versa based on some conditions related to task size and overheads. 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", GRID systems, 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.

Granularity control has been studied in the context of traditional programming [20, 25], functional programming [17, 18], and also logic programming [19, 7, 30, 8, 23, 24]. In [24] we proposed a general granularity control model and reported on its application to the case of logic programs. This model proposes (efficient) conditions based on the use of information available on task granularity in order to choose between parallel and sequential execution. The problems to be solved in order to perform granularity control following this approach include, on one hand, estimating the cost of tasks, 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.

Performing accurate granularity control at compile-time is difficult because some of the information needed to evaluate communication and computational costs, as for example input data size, is only known at run-time. A useful strategy is to do as much work as possible at compile-time, and postpone some final decisions to run-time. This can be achieved by generating at compile-time cost functions which estimate task costs as a function of input data size, which are then evaluated at run-time when such size is known. Then, after comparing costs of sequential and parallel executions (including all overheads), it is possible to determine which type of execution is profitable.

The approximation of these cost functions can be based either on some heuristics (e.g., profiling) or on a *safe* approximation (i.e. an upper or lower bound). We were able to show that if upper or lower bounds on task costs are available, under a given set of assumptions, it is possible to ensure that some parallel, distributed executions will always produce speedup (and also that some others are best executed sequentially). Because of these results, we will in general require

[4] In the past two decades, quite significant progress has been made in the area of automatically parallelizing programs in the context of logic and constraint programs, and some of the challenges have been tackled quite effectively there –see, for example, [13, 14, 4] for an overview of this area.

the cost information to be not just an approximation, but rather a well-defined bound on the actual execution cost. In particular, we will use the techniques for inferring upper- and lower-bound cost functions outlined in the previous section.

Assuming that such functions or similar techniques for determining task costs and overheads are given, the remainder of the granularity control task is to devise a way to actually compute such costs and then dynamically control task creation and scheduling using such information. Again the approach of doing as much of the work as possible at compile-time seems advantageous. In our approach, a transformation of the program is performed at compile time such that the cost computations and spawning decisions are encoded in the program itself, and in the most efficient way possible. The idea is to perform any remaining computations and decisions at run-time when the parameters missing at compile-time, such as data sizes or node load are available. In particular, the transformed programs will perform (generally) the following tasks: computing the sizes of data that appear in cost functions; evaluating the cost functions of the tasks to be executed in parallel using those data sizes; safely approximating the spawning and scheduling overheads (often also a function of data sizes); comparing these quantities to decide whether to schedule tasks in parallel or sequentially; deciding whether granularity control should be continued or not; etc.

As an example, consider the `inc_all` procedure of Section 2 and the program expression:

```
..., Y = inc_all(X) & M = r(Z), ...
```

which indicates that the procedure call `inc_all(X)` is to be made available for execution in parallel with the call to `r(Z)` (we assume that analysis has determined that `inc_all(X)` and `r(Z)` are independent, by, e.g., ensuring that there are no pointers between the data structures pointed to by `X,Y` and `Z,M`. From Section 2 we know that the cost function inferred for `inc_all` is $\texttt{Cost}^{L}_{inc\_all}(n) = 2\ n + 1$. Assume also that the cost of scheduling a task is constant and equal to 100 computation steps. The previous goal would then be transformed into the following one:

```
..., ( 2*length(X)+1 > 100 -> Y = inc_all(X) & M = r(Z)
                            ;  Y = inc_all(X),  M = r(Z) ), ...
```

where (*if* -> *then* ; *else*) is syntax for an if-then-else and "`,`" denotes sequential execution as usual. Thus, when $2 * length(X) + 1$ (i.e., the lower bound on the cost of `inc_all(X)`) is greater than the threshold, the task is made available for parallel execution and not otherwise. Many optimizations are possible. In this particular case, the program expression can be simplified to:

```
..., ( length(X) > 50 -> Y = inc_all(X) & M = r(Z)
                       ;  Y = inc_all(X),  M = r(Z) ), ...
```

and, assuming that `length_gt(L,N)` succeeds if the length of `L` is greater than `N` (its implementation obviously only requires to traverse at most the n first elements of list), it can be expressed as:

```
..., ( length_gt(LX,50) -> Y = inc_all(X) & M = r(Z)
                         ;  Y = inc_all(X),  M = r(Z) ), ...
```

```
:- module(qsort, [qsort/2], [assertions]).

qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E < C,  partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C, partition(R,C,Left,Right1).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
```

Fig. 1. A qsort program.

As mentioned before, scheduling costs are often also a function of data sizes (e.g., communication costs). For example, assume that the cost of executing remotely `Y = inc_all(X)` is 0.1 ($length(X) + length(Y)$), where $length(Y)$ is the size of the result, an upper bound on which (actually, exact size) we know to be $length(X)$. Thus, our comparison would now be:

$$\begin{aligned} 2\ length(X) + 1 > 0.1\ (length(X) + length(Y)) \equiv \\ 2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \equiv \\ 2\ length(X) + 1 > 0.2\ length(X) \cong \\ 2\ length(X) > 0.2\ length(X) \equiv \\ 2 > 0.2 \end{aligned}$$

Which essentially means that the task can be scheduled for parallel execution *for any input size*. Conversely, with a communication cost greater than $0.5(length(X) + length(Y))$ the conclusion would be that it would never be profitable to run in parallel.

These ideas have been implemented and integrated in the `CiaoPP` system, which uses the information produced by its analyzers to perform combined compile–time/run–time resource control. The more realistic example in Figure 1 (a quick-sort program coded using logic programming) illustrates additional optimizations performed by `CiaoPP` in addition to cost function simplification, which include improved term size computation and stopping performing granularity control below certain thresholds. The concrete transformation produced by `CiaoPP` adds a clause: "`qsort(X1,X2) :- g_qsort(X1,X2).`" (to preserve the original entry point) and produces `g_qsort/2`, the version of `qsort/2` that performs granularity control (where `s_qsort/2` is the sequential version) is shown in Figure 2.

Note that if the lengths of the two input lists to the recursive calls to `qsort` are greater than a threshold (a list length of 7 in this case) then versions which

```
g_qsort([X|L],R) :-
        partition_o3_4(L,X,L1,L2,S1,S2),
        ( S2>7 -> (S1>7 -> g_qsort(L2,R2) & g_qsort(L1,R1)
                         ; g_qsort(L2,R2), s_qsort(L1,R1))
                ; (S1>7 -> s_qsort(L2,R2), g_qsort(L1,R1)
                         ; s_qsort(L2,R2), s_qsort(L1,R1))),
        append(R1,[X|R2],R).
g_qsort([],[]).
```

Fig. 2. The qsort program transformed for granularity control

continue performing granularity control are executed in parallel. Otherwise, the two recursive calls are executed sequentially. The executed version of each such call depends on its grain size: if the length of its input list is not greater than the threshold 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 than the threshold, and thus, for all of them, execution should be performed sequentially and, obviously, no granularity control is needed. Procedure `partition_o3_4/6`:

```
partition_o3_4([],_B,[],[],0,0).
partition_o3_4([E|R],C,[E|Left1],Right,S1,S2) :-
        E<C, partition_o3_4(R,C,Left1,Right,S3,S2), S1 is S3+1.
partition_o3_4([E|R],C,Left,[E|Right1],S1,S2) :-
        E>=C, partition_o3_4(R,C,Left,Right1,S1,S3), S2 is S3+1.
```

is the transformed version of `partition/4`, which "on the fly" computes the sizes of its third and fourth arguments (the automatically generated variables `S1` and `S2` represent these sizes respectively) [22].

4 Resource-Aware Mobile Computing

Having reviewed the issue of granularity control, and following the classification of issues of Section 1 we now turn our attention to some resource-related issues on the receiver side. In an open setting, such as that of the GRID and other similar overlay computing systems, receivers must have some assurance that the received code is safe to run, i.e., that it adheres to some conditions (the *safety policy*) regarding what it will do. We follow current approaches to mobile code safety, based on the technique of *Proof-Carrying Code* (PCC) [28], which as mentioned in Section 1 associate safety *certificates* to programs. A certificate (or proof) is created by the code supplier for each task at compile time, and packaged along with the untrusted mobile code sent to (or taken by) other nodes in the network. The consumer node who receives or takes the code+certificate package (plus a given task to do within that code) can then run a *checker* which by a straightforward inspection of the code and the certificate can verify the validity of the certificate and thus compliance with the safety policy. The key benefit of this approach is that the consumer is given by the supplier the capacity of ensuring compliance with the desired safety policy in a simple and efficient way.

Indeed the (proof) checker used at the receiving side performs a task that should be much simpler, efficient, and automatic than generating the original certificate. For instance, in the first PCC system [28], the certificate is originally a proof in first-order logic of certain *verification conditions* and the checking process involves ensuring that the certificate is indeed a valid first-order proof.

The main practical difficulty of PCC techniques is in generating safety certificates which at the same time:

- allow expressing interesting safety *properties*,
- can be generated *automatically* and,
- are easy and *efficient* to check.

Our approach to mobile code safety [1] directly addresses these problems. It uses approximation techniques, generally based on abstract interpretation, and it has been implemented using the facilities available in `CiaoPP` and discussed in the previous sections. These techniques offer a number of advantages for dealing with the aforementioned issues. The expressiveness of the properties that can be handled by the available abstract domains (and which can be used in a wide variety of assertions) will be implicitly available to define a wide range of safety conditions covering issues like independence, types, freeness from side effects, access patterns, bounds on data structure sizes, bounds on cost, etc. Furthermore, the approach inherits the inference power of the abstract interpretation engines used in CLP to automatically generate and validate the certificates. In the following, we review our standard mobile code certification process and discuss the application in parallel distributed execution.

Certification in the Supplier: The certification process starts from an initial program and a set of assertions provided by the user on the producer side, which encode the safety policy that the program should meet, and which are to be verified. Consider for example the following (naive) reverse program (where `append` is assumed to be defined as in Figure 1):

```
:- entry reverse/2 : list * var.
reverse( [] )    := [].
reverse( [H|L] ) := ~append( reverse(L), [H] ).
```

Let us assume also that we know that the consumer will only accept purely computational tasks, i.e., tasks that have no side effects, and only those of polynomial (actually, at most quadratic) complexity. This safety policy can be expressed at the producer for this particular program using the following assertions:

```
:- check comp reverse(A,B)
         + sideff(free).
:- check comp reverse(A,B)
         : list * var
         + steps_ub( o(exp(length(A),2)) ).
```

The first (computational –`comp`) assertion states that it should be verified that the computation is pure in the sense that it does not produce any side effects

(such as opening a file, etc.). The second (also computational) assertion states that it should be verified that there is an upper bound for the cost of this predicate in $O(n^2)$, i.e., quadratic in n, where n is the length of the first list (represented as `length(A)`). Implicitly, we are assuming that the code will be accepted at the receiving end, provided all assertions can be checked, i.e., the intended semantics expressed in the above assertions determines the safety condition. This can be a policy agreed a priori or exchanged dynamically.

Note that, unlike traditional safety properties such as, e.g., type correctness, which can be regarded as platform independent, resource-related properties should take into account issues such as load and available computing resources in each particular system. Thus, for resource-related properties different nodes may impose different policies for the acceptance of tasks (mobile code).

Generation of the Certificate: In our approach, given the previous assertions defining the safety policy, the certificate is automatically generated by an *analysis engine* (which in the particular case of `CiaoPP` is based on the *goal dependent*, i.e., context-sensitive, analyzer of [16]). This analysis algorithm receives as input a set of entries (included in the program like the `entry` assertion of the example above) which define the base, boundary assumptions on the input data. These base assumptions can be checked at run-time on the actual input data (in our example the type of the input is stated to be a list). The computation of the analysis process terminates when a fixpoint of a set of equations is reached. Thus, the results of analysis are often called the *analysis fixpoint*.

Due to space limitations, and given that it is now well understood, we do not describe here the analysis algorithm (details can be found in, e.g., [2, 16]). The important point to note is that the certification process is based on the idea that the role of certificate can be played by a *particular and small subset of the analysis results* (i.e., of the analysis fixpoint) computed by abstract interpretation-based analyses.

For instance, the analyzers available in `CiaoPP` infer, among others, the following information for the above program and entry:

```
:- true pred reverse(A,B)
        : ( list(A), var(B) )
       => ( list(A), list(B))
        + ( not_fails, is_det, sideff(free),
            steps_ub( 0.5*exp(length(A),2)+1.5*length(A)+1 )).
```

stating that the output is also a list, that the procedure is deterministic and will not fail, that it does not contain side-effects, and that calls to this procedure take at most $0.5\ (length(A))^2 + 1.5\ length(A) + 1$ resolution steps. In addition, given this information, the output shows that the "status" of the three `check` assertions has become `checked`, which means that they have been validated and thus the program is safe to run (according to the intended meaning):

```
:- checked comp reverse(A,B)
         + sideff(free).
```

```
:- checked comp reverse(A,B)
        : list * var
        + steps_ub( o(exp(length(A),2)) ).
```

Thus, we have verified that the safety condition is met and that the code is indeed safe to run (for now on the producer side). The analysis results above can themselves be used as the *cost and safety certificate* to attest a safe and efficient use of procedure `reverse` on the receiving side.

In general the verification process requires first generating a *verification condition* [1] that encodes the information in the check assertions to be verified and then checking this condition against the information available from analysis. This validation may yield three different possible status: i) the verification condition is indeed checked and the fixpoint is considered a *valid certificate*, ii) it is disproved, and thus the certificate is not valid and the code is definitely not safe to run (we should obviously correct the program before continuing the process); and iii) it cannot be proved nor disproved. Case iii) occurs because the most interesting properties are in general undecidable. The analysis process in order to always terminate is based on approximations, and may not be able to infer precise enough information to verify the conditions. The user can then provide a more refined description of initial entries or choose a different, finer-grained, abstract domain. However, despite the inevitable theoretical limitations, the analysis algorithms and abstract domains have been proved very effective in practice. In both the ii) and iii) cases, the certification process needs to be restarted until achieving a verification condition which meets i). If it succeeds, the fixpoint constitutes a valid certificate and can be sent to the receiving side together with the program.

Validation in the Consumer: The *validation* process performed by the consumer node is similar to the above certification process except that the analysis engine is replaced by an *analysis checker*. The definition of the analysis checker is centered around the observation that the checking algorithm can be defined as a very simplified "one-pass" analyzer. Intuitively since the certification process already provides the fixpoint result as certificate, an additional analysis pass over it cannot change the result. Thus, as long as the fixpoint is valid, one single execution of the abstract interpreter validates the certificate.

As it became apparent in the above example, the interesting point to note is that abstract interpretation-based techniques are able to reason about computational properties which can be useful for controlling efficiency issues in a mobile computing environment and in distributed parallelism platforms. We consider the case of the receiver of a task in a parallel distributed system such as a GRID. This receiver (the code consumer) could use this method to reject code which does not adhere to some specification, including usage of computing resources (in time and/or space). Reconsider for example the previous reverse program and assume that a node with very limited computing resources is assigned to perform a computation using this code. Then, the following "check" assertion can be used for such particular node:

```
:- check comp reverse(A,B)
         : ( list(A, term), var(B) )
         + steps_ub( length(A) + 1 ).
```

which expresses that the consumer node will not accept an implementation of `reverse` with complexity bigger than linear. In order to guarantee that the cost assertion holds, the certificate should contain upper bounds on computational cost. Then, the code receiver proceeds to validate the certificate. The task of checking that a given expression is an upper bound is definitely simpler than that of obtaining the most accurate possible upper bound. If the certificate is not valid, the code is discarded. If it is valid, the code will be accepted only if the upper bound in the certificate is lower or equal than that stated in the assertion. In our example, the certificate contains the (valid) information that `reverse` will take at most $0.5\ (length(A))^2 + 1.5\ length(A) + 1$ resolution steps. However, the assertion requires the cost to be at most $length(A) + 1$ resolution steps. A comparison between these cost functions does not allow proving that the code received by the consumer satisfies the efficiency requirements imposed (i.e. the assertion cannot be proved).[5] This means that the consumer will reject the code. Similar results would be obtained if the worst case complexity property `steps_ub( o(length(A)) )` was used in the above check assertion, instead of `steps_ub( length(A) + 1 )`.

Finally, and interestingly, note that the certificate can also be used to approximate the *actual costs of execution* and make decisions accordingly. Since the code receiver knows the data sizes, it can easily apply them to the cost functions (once they are verified) and obtain values that safely predict the time and space that the task received will consume.

5 Conclusions

We have presented an abstract interpretation-based approach to resource-aware distributed and mobile computing and discussed their implementation in the context of a multi-paradigm programming system. Our framework uses modular, incremental, abstract interpretation as a fundamental tool to infer resource and safety information about programs. We have shown this information, including lower bounds on cost and upper bounds on data sizes, can be used to perform high-level optimizations such as resource-aware task granularity control. Moreover, cost information and, in particular, upper bounds, inferred during the previous process are relevant to certifying and validating mobile programs which may have constraints in terms of computing resources (in time and/or space). In essence, we believe that our proposals can contribute to bringing increased flexibility, expressiveness and automation of important resource-awareness aspects in the area of mobile and distributed computing.

[5] Indeed, the lower bound cost analysis in fact disproves the assertion, which is clearly invalid.

Acknowledgments

This work has been supported in part by the European Union IST program under contracts IST-2001-38059 "ASAP" and "GridCoord", by MCYT project TIC 2002-0055 "CUBICO" and FEDER infrastructure UNPM-E012, and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico.

References

1. E. Albert, G. Puebla, and M. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, April 2004.
2. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM), June 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
4. J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.
5. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 and 3):103–179, 1992.
7. S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
8. S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
9. S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.
10. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.
11. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.
12. I. Foster, C. Kesselman, J. Nick, and S. Tuecke, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.
13. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

14. M. Hermenegildo. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *Proceedings of EUROPAR'97*, volume 1300 of *LNCS*, pages 31–46. Springer-Verlag, August 1997.
15. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.
16. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
17. L. Huelsbergen. Dynamic Language Parallelization. Technical Report 1178, Computer Science Dept. Univ. of Wisconsin, September 1993.
18. L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.
19. S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.
20. B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.
21. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
22. P. López-García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.
23. P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.
24. P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.
25. C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32, 1989.
26. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
27. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.
28. G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.
29. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
30. X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.