Implementation of an Event Driven Scheme for Visualizing Parallel Execution of Logic Programs

L. Gómez M. Carro M. V. Hermenegildo

Universidad Politécnica de Madrid (UPM),
Facultad de Informática,
28660 – Boadilla del Monte, Madrid – SPAIN
e-mail: {mcarro, lgomez, herme}@fi.upm.es

Abstract

This article presents in an informal way some early results on the design of a series of paradigms for visualization of the parallel execution of logic programs. The results presented here refer to the visualization of or-parallelism, as in MUSE and Aurora, deterministic dependent and-parallelism, as in Andorra-I, and independent and-parallelism as in &-Prolog. A tool has been implemented for this purpose and has been interfaced with these systems. Results are presented showing the visualization of executions from these systems and the usefulness of the resulting tool is briefly discussed.

1 Introduction

The difficulty of parallel programming and the lack of cost-effective parallel hardware have been traditionally considered the main hindrances to the wide spread use of parallelism. While cost-effective parallel computers are now appearing in the marketplace, the lack of software able to exploit parallelism in an efficient way is still an important bottleneck. In general, making a good program for a parallel machine is much more difficult than for a sequential one, because of the need of studying data dependencies and perhaps having to adapt the algorithm to particular topologies. High level languages have been traditionally aimed at freeing the programmer from such low-level tasks, but imperative languages do not fulfill this requirement, and machine-specific synchronization primitives have to be frequently manually used to direct flow control when such programs are parallelized.

Logic programming allows the development of a new generation of languages able to exploit the full potential of parallel computers in a transparent manner. The proximity of the declarative and procedural semantics of logic languages, together with the (theoretical) lack of flow control, makes them more amenable to analysis and automatic parallelization. This kind of evolved compiling tools, together with engines to execute parallel logic programs, could make true the users dream of being able to write a program which runs both in an sequential and in a parallel computer, achieving the best possible performance in both cases without having to give much — if any — thought to the parallel execution. But such a dream is a nightmare for the people who design and implement such systems. Programmers who develop parallel systems have to deal with concrete machines and ensure that the execution model is correct and fast, and that it has been properly implemented. Programmers developing high-level tools, such as parallelizing compilers, have to ensure that the parallelized programs retain sequential semantics, if this is desired, and that their execution is safe. Final users would also like to understand what happens in the core of the processing engine, for that understanding can help them to adopt a programming style which, while retaining sequential semantics, will be more adequate for parallel processing. Thus, it is important to develop tools to facilitate these tasks.

All programming environments have some tools to help users and developers. When creating new and more evolved programming paradigms, it is necessary to have also new and more evolved programming tools. The development of parallel logic environments has to deal actively with issues like load balancing, scheduling, correctness of parallel programs, etc. Therefore, there is a need for a new generation of debugging and analysis tools at the same level that those being developed. In this paper we will introduce one prototype of a tool of such class, aimed at exploring and understanding the intricacies of parallel logic program execution. It would be desirable for such a tool to be:

- Intuitive and understandable for people not highly skilled in parallel processing.
- Accurate and flexible, for those who will use it to verify the operational behavior of a particular implementation.
- Easy to extend, so that it can be developed as needed in order to visualize new execution paradigms.

Looking for a parallel process representation paradigm which fulfills these requirements it is natural to arrive to graphical visualization as a solution. There are two main reasons for this: on one hand, parallel process visualization has been used from the early times of parallel processing to explain parallel algorithms, and, on the other hand, visual information is easy and fast to understand. Thus, we claim that parallel visualization can be effectively used as the basis for a tool aimed at a better understanding of parallel processing from the point of view of the skilled programmer and the final user. This is the approach taken in the VisAndOr tool, which we will describe here. It should be noted that this is only a partial view of the possibilities of VisAndOr, because it is still being improved and open to the appearance of new parallel execution paradigms. In particular, in this paper we concentrate on the capabilities of the tool for analyzing the parallel execution of the MUSE, Aurora, Andorra-I, and &-Prolog systems [AK90b, AK90a, Sze89, SCWY90, SCWY91, HJ90, Her86, HG91].

The rest of this paper is structured as follows: in Section 2 there is a brief overview of some existent approaches to parallel process visualization, together with a description of the VisAndOr approach. In Sections 3, 4 and 5 the visualization of the parallel programming systems MUSE, Aurora, Andorra-I and &-Prolog from the VisAndOr perspective is discussed. Some topics in which VisAndOr can be helpful are described in Section 6. Additional topics in which the events approach can be useful are sketched in Section 7. Details about the VisAndOr implementation, capabilities and working environment are given in Section 8. Finally, Section 9 describes conclusions and future work.

2 Process Visualization

Graphical visualization has been with Computer Science since its inception, mainly because of its representation power. One example is the graphical representation of finite automata which has been found to be useful and accurate at the same time. A carefully designed representation should also lead to a better comprehension of parallel process behavior and, subsequently, to easier debugging and implementation of a real parallel system. In the next section we will have a brief look to some existing visualization paradigms, and after that we will focus on the VisAndOr approach.

2.1 A Review of Visualization Systems

Briefly, the target of visualization systems is to display data in a comprehensible and meaningful manner. The data may have been gathered prior to displaying or generated and processed just-in-time. Unlike the representation of physical phenomena, the visualization of computer processes must map abstract relations between entities (namely programs, messages, scheduling strategies. . .) into a more concrete representation — think of the finite automata example, in which discrete functions are given the appearance of a graph. Sometimes this mapping is merely a make-up (for example, bar charts obtained from load data), but even in this case, the flexibility and intuitiveness that visualization offers is enough to fill the gap between raw data and high level human perception. In this section, we will just mention some existing systems aimed at studying run time execution characteristics. We will no pay attention to *pretty-printing* programs or flow-chart generators, which give a static vision of static text. For a complete discussion on a suitable taxonomy of software visualization systems, the reader is referred to [PSB92].

One of the most well known software visualization systems is BALSA (and its most recent version, BALSA II) [Bro88]. This system allows users to watch the run-time evolution of data structures of a Pascal program. The statement being executed is highlighted in the code. Animation is achieved by calls previously inserted by the user in the source code. This is an example of a system which shows actual step-by-step sequential execution and data structure changes, but that has the drawback of not being transparent to the user, because of the need of source code transformation.

The Transparent Prolog Machine [EB88] is an interpreter which displays automatically a running trace of the program being executed. It has many good details, such as being able to show coarse-grained or fine-grained views and a good treatment of meta-predicates. This, together with a careful graphic design, makes it a very good tool to understand and study the sequential Prolog execution model. The source code being studied does not need any modification at all.

The ParaGraph[1] tool by Heath and Etheridge [HE91] is a graphical display system for visualizing the behavior and performance of parallel programs on message-passing multiprocessors. It takes as input execution profile data obtained during an actual run of a parallel program on a message-passing machine, and the resulting trace data can then be replayed pictorially with ParaGraph to provide a dynamic depiction of the behavior of the parallel program.

The ParaGraph tool by Aikawa et al. [AKK+92] is aimed at tuning the Parallel Inference Machine (PIM) [GSN+88]. Its main purposes are low-level (processor) profiling, i.e., discovering how a given (set of) processor(s) distribute its (their) time in a program execution, and high-level (shōen) profiling, which attempts to discover how many times goals are reduced or suspended. ParaGraph gathers profiling data during program execution using primitives of the KL1 [UC10] language.

Finally, the VISTA tool [Tic92] intends to give effective visual feedback to a programmer tuning a concurrent logic program. Its inputs are a trace file obtained during execution and a source program. The results obtained with VISTA have the peculiar shape of a snail shell, due to the mapping of the (parallel) search tree into a polar coordinate system. This system, which represents deterministic dependent and-parallelism, was developed using some ideas from VisAndOr's forerunner, VISIPAL.

The tools cited above are not the only ones with merit enough to be mentioned, but they give us a starting point to find where VisAndOr fits. As we can infer, there are some tools which need programs to be rewritten and some others that do not. While some tools are oriented to sequential processing, other tools are mainly devised to visualize parallel processing (although, in general, they are able to deal with sequential execution as a special case). Last, but not least, there are tools which are oriented towards machine-level profiling[2] and tools which are aimed at showing a given execution paradigm regardless the underlying architecture. This rough division is not exclusive: the classes do overlap among them.

So, where does VisAndOr fit in? We will give some overall characteristics, using the taxonomy given in [PSB92], before we expand on how VisAndOr is designed:

Scope: VisAndOr can visualize arbitrary parallel Prolog program executions: currently, or-parallel programs, Independent and-parallel programs and dependent deterministic and-parallel programs — possibly also with or-parallelism. The inner algorithms do not pose limit to the size and/or complexity of such programs. Only the available memory and numerical accuracy of the computer can introduce restrictions. The tool is still being developed to encompass more execution paradigms. In general, tree, fork-and-join, and non-structured task organizations can be represented in time.

[1] There are two different visualization tools with the same name: the one we are currently referring to and the ParaGraph by Aikawa et al., described in [AKK+92], which we refer to below.

[2] i.e., process to processor mapping, message-passing schemes, etc.

Content: VisAndOr performs program visualization, not code visualization. Data visualization can often be naturally inferred due to the nature of the Prolog execution model. The information is completely gathered at run-time. What is shown is an abstraction (a process dependency graph) of the parallel execution.

Form: The graphic output consists mainly of a sketch of the search space traversed during execution. Color is effectively used (although VisAndOr can also run on monochrome displays), together with icons and identifiers. The view is static, but conveys the whole history of the execution.

Method: No source modification is needed, and the data gathered at run-time is used to feed VisAndOr as a batch job. The user does not need to be familiar with the code being traced neither to understand the pictures nor to obtain them.

Interaction: Navigation through a large picture can be accomplished by zooming in and out. As we have said before, VisAndOr presents the whole execution in a single snapshot; accurate time measurements can be easily performed. There is as yet no automatic way to perform data elision.

Effectiveness: The visual metaphor used is quickly understood, and it has been reported to be useful.

In the following sections we will extend on the basic VisAndOr concepts, mainly the notion of event tracing and the parallelism versus time representation through a process dependency graph.

2.2 The event mechanism

The VisAndOr process visualization paradigm is based on the idea of *event*. Roughly speaking, an event describes a relevant point in the execution, together with the information needed to distinguish it from any other event in the same execution. Typically an event contains its type, a time stamp and some additional pieces of information.

VisAndOr takes as input trace files containing events, usually gathered during actual execution. An event is generated when an *interesting point* in the execution is reached. The notion of "interesting point" is, of course, different for every execution paradigm. But, as can be expected, typical events for parallel logic programs execution are fork, join, success, fail, cut, etc. Trace files are the only input that VisAndOr needs and understands. This approach has some interesting characteristics:

- Essentially no program transformation is needed (except for starting and stopping the tracing of events and saving them into a file). Other approaches to visualization need major changes to the programs to be analyzed, because they are based on the idea that the program itself draws its execution. Although such transformations can be minimized by carefully choosing drawing primitives, most times the transformation depends on the algorithm and the data structures used and needs the user to have previous knowledge of the source program. Since in the VisAndOr approach the events are generated at a lower level, the program source can remain almost unchanged. Of course, this assumes that the implementation can be modified to generate the events. Alternatively, the program transformation approach can always be taken.

- There is no need to have direct access to the program or system being studied during visualization. Since the events can be dumped to trace files, an execution can be visualized at any place, without need for the program or system that generated it.

- VisAndOr is easily extensible. When a new parallel computation paradigm is to be visualized, VisAndOr only has to be extended to understand the traces generated by it. Sometimes this is not needed, and a program can perform the necessary translation to a format understandable by VisAndOr.

- In fact, real systems are not required. Since VisAndOr only requires execution traces, such traces can be generated by a simulator or obtained through any type of processing of otherwise actual traces.

It is important to ensure that the execution is affected as little as possible by the tracing of events. This is relatively easily achieved in general by storing the events in a table in main memory (dumping only to a file at the end of the interesting part of the execution) and calling a fast, low level system primitive to obtain time stamps.

The mechanism of events has been found to be flexible and reliable, and has been used in other visualization systems.

2.3 VisAndOr basics

In this section we will describe how an execution is shown in VisAndOr, and what facilities VisAndOr offers to the user.

The current implementation of VisAndOr shows a static view of a whole execution. In order to do this, time is represented as a spatial coordinate (the Y axis), advancing from top to bottom; the execution is shown as a dependency graph. This graph is similar, but not equal, to the usual and-or tree: what is shown here is the work actually being done in parallel at each time, as well as the available work. For example, let us think of an or-parallel execution (see, for example, Figure 1). Whenever different alternatives are found for a given branch, this branch is split. Horizontal dashed lines show the point in the time where the alternative was found; vertical dashed or thick lines represent or-parallel search paths. If a given branch does not contain alternatives up to the success or failure, it will be shown as a single line. Since we are tracing or-parallel tasks, the conjunctive goals in the bodies are not shown: they appear as a single straight line, unless alternatives exist for them. A similar scheme is used to depict Andorra-I and &-Prolog parallelism.

In general, no matter what execution paradigm is being studied, the number of active tasks (i.e., tasks in which active work is being done) can be different than the total number of tasks. This can be due (as in the execution models we are studying) because the number of processors is different (in general, smaller) than the number of processes or because of some other reasons, i.e., due to suspension of goals. The strategy used in VisAndOr to signal this is very simple: a thick line means a working processor, whereas a thin dotted line means that there is work available, but no processor available to work on it. In a color display a different color is assigned to each processor.

Sometimes a more detailed view of the events which happened in an execution is needed: for example, think of the end of a single line in an or-parallel execution. To know whether wether it represents a failure or a success, additional information is needed. This information is provided by VisAndOr in the form of icons. Icons are optionally placed in the points where events have happened, so that events that have no immediate or obvious effect on the parallel execution can be easily detected. Typical examples of such events are a cut or a change in the scheduling priority for a given task. Identifiers can also be attached to the branches in order to help to identify (specially in monochrome displays) which processor is working on each task. In paradigms where the number of agents (i.e., the number of WAMs) can be different from the number of workers (i.e., processors) the agents to processors and agents to tasks mapping can also be displayed using different identifiers for the agents. VisAndOr also has a zooming capability to aid in the detailed study of local phenomena.

The shape of the graph depends on the type of execution being visualized. Or-parallel execution looks like a tree; Independent and-parallel execution has also to show the join of independent tasks, and is therefore a planar graph, and Andorra-I parallelism exhibits both or-parallelism and deterministic and-parallelism, which can be visualized as a variation of a tree

In the following three sections we will focus on how VisAndOr shows program executions, taking as examples of parallel logic languages the MUSE, Aurora, Andorra-I, and the &-Prolog models. The features above mentioned, as well as others not yet referred to will be explained with examples taken from real executions.

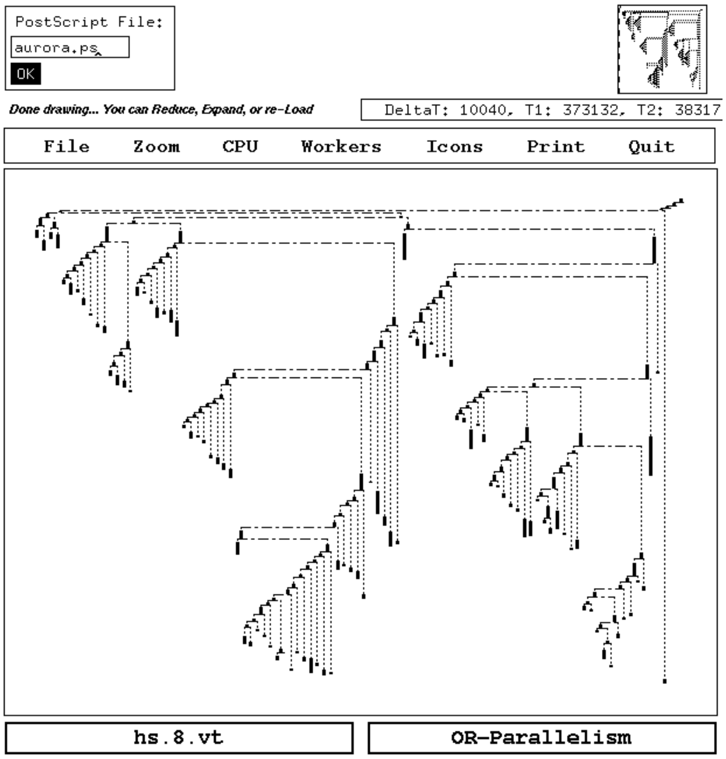

Figure 1: Or parallelism: an Aurora trace

3 Visualization of or-parallelism (MUSE and Aurora)

Or parallelism occurs when the different clauses of a given predicate are explored simultaneously, i.e. in the parallel execution of different *or* branches of the resolution tree. The entities which perform the parallel exploration of the *or* branches are called *workers*. Each worker is a complete WAM with extensions to allow parallelism.

Two models will be used to show the VisAndOr approach to or-parallelism visualization: the MUSE model, developed at SICS [AK90b, AK90a], and the Aurora system, developed at the University of Bristol [Sze89]. Results (from the purely visualization point of view) are quite similar in both cases.

3.1 VisAndOr and or-parallelism

The representation of *or* parallelism in VisAndOr takes the form of a tree (Figure 1), in which the whole execution is shown. Time is represented as a linear function of the space from the start of the tree, i.e. the time coordinate runs from top to bottom. Vertical thick lines are active *workers*, whereas vertical dashed lines represent clauses waiting for an idle worker. In a color display, each worker appears in a different color; if a black and white display is being used, the worker number can be displayed next to each execution line. Horizontal dashed lines represent the existence of multiple choices for a given predicate.

Just at first sight, the amount of thick lines provides an intuitive estimation of the degree of parallelism achieved. The information contained in the tree helps to understand the benefits of parallel execution: it is clear how multiple workers are active at a time. The overall vision of the execution helps to understand the "big picture" of the parallel execution. In order to tune the system, the zooming capabilities of VisAndOr can be used to perform micro-analysis. VisAndOr supports currently two more aids to tuning analysis: worker analysis and events analysis.

Worker analysis in intended to figure the active work that a given worker performs. Of course, this could be done by finding out the ratio between the time that worker has been active and the sum of

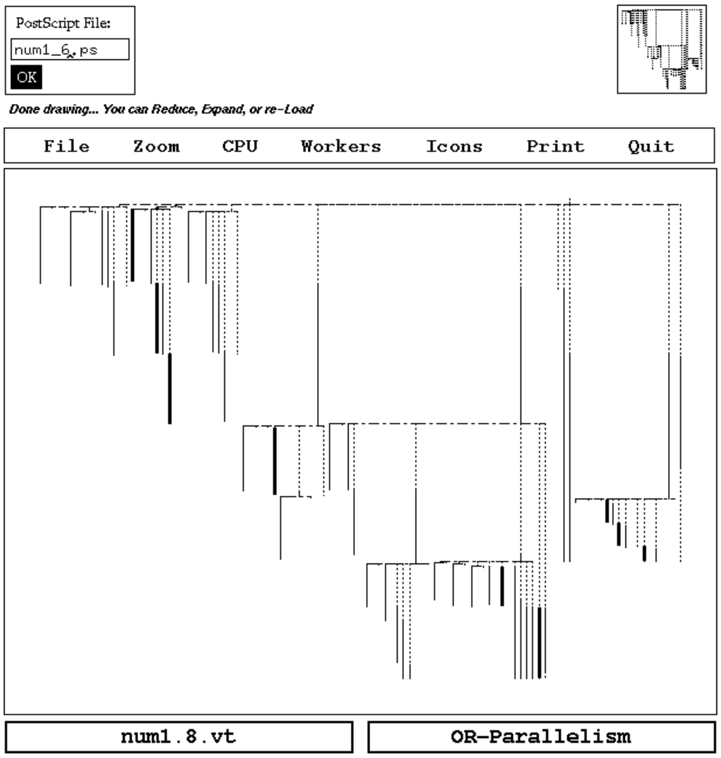

Figure 2: A working worker

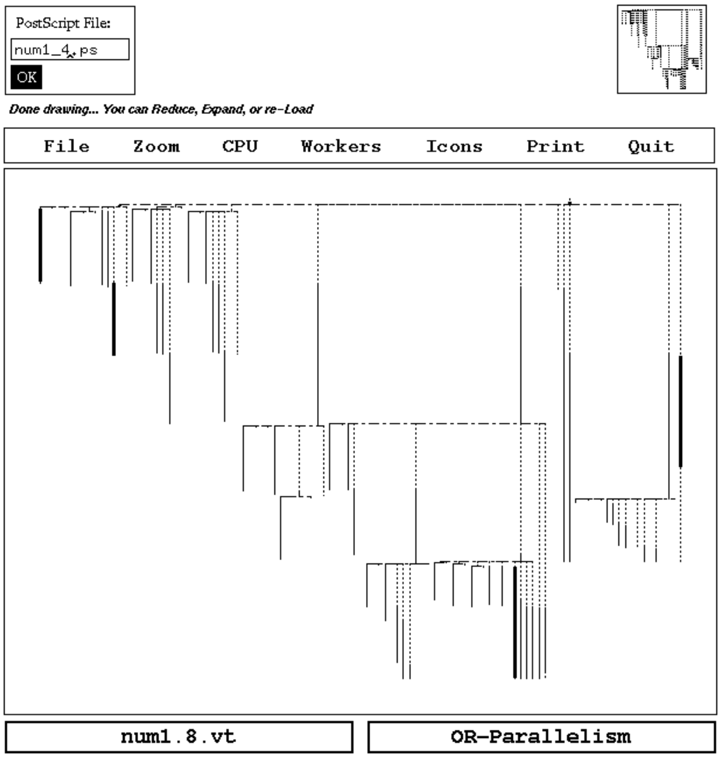

Figure 3: A lazy comrade

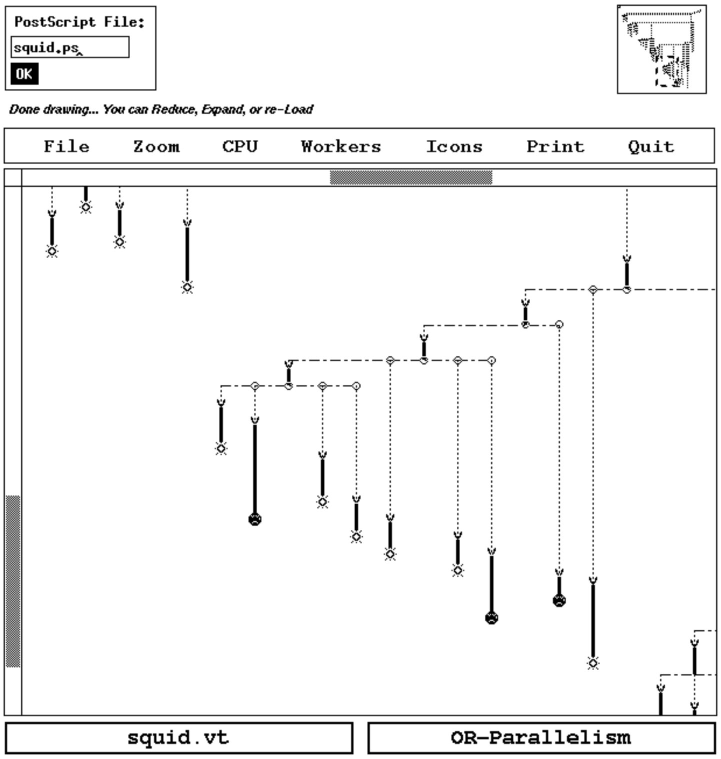

Figure 4: A zoom with icons

the times of all the workers. But this raw figure cannot give a reason why a hypothetical unbalanced load happens. Considerable help can be obtained from the previous history of such worker and from the current state of the computation. VisAndOr uses a simple mechanism to uncover the story and computational environment of a worker: to show with a thick line only the worker being studied. In Figures 2 and 3 the same MUSE execution is shown, but two different workers have been highlighted.

What can be inferred from both figures? It is easy to see that the worker in the Figure 2 has been active all the time; but a look at Figure 3 shows that there is a gap in which the selected worker has not been active. This inactivity lasts enough to perhaps conclude that there was no work available or, depending on the scheduler used, that there were considerable scheduling delays. Careful analysis of issues like this can be of invaluable help when implementing, debugging and tuning parallel systems.

Another type of analysis that VisAndOr offers is event analysis. VisAndOr has the ability of displaying the events that define the execution. Usually, such events include special points in the execution which have to be carefully treated, such as success, failure, cuts and *or* forks. VisAndOr shows such events as icons located at the appropriate points in the tree. Figure 4 is a example of a zoom with icons displayed. Start of active working is signaled with a small arrow pointing towards bottom, the creation of alternatives are shown with circles and the successful and failing *or* branches are distinguished with a sad face and a twinkling star. A way of measuring actual times using the mouse is included to aid in the analysis of such figures. The position of the section being displayed in the whole graph is depicted by the dashed square in the graph summary in the top right corner of the figure.

3.2 ViMust

The VisAndOr approach to logic programs visualization is static, i.e., the whole tree corresponding to a given execution is shown. The complementary model, in which the part of the tree being explored is shown as in a movie is being experimented at SICS, continuing along the lines of the original WAM-Trace tool [DL87] developed at Argonne National Labs in the context of Aurora. The program, called MUST [SS90], can show snapshots of MUSE executions as well as animations of such executions. The tree

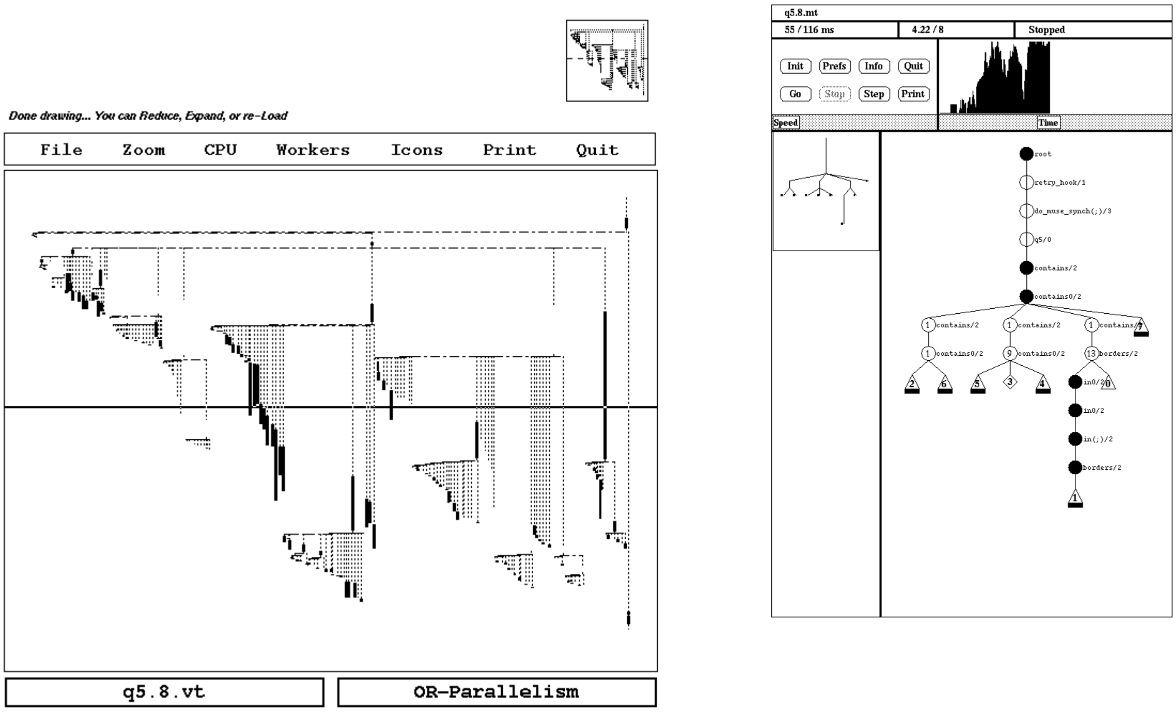

Figure 5: ViMust: MUST and VisAndOr working together

shown by MUST corresponds to the actual path being explored in parallel, and contains information about the state of each worker. VisAndOr and MUST can work together through a simple protocol which allows each one to send messages to the other asynchronously. VisAndOr indicates the point displayed by MUST with a horizontal line and answers to the messages sent by MUST to move the line. Conversely, the bar can be moved from VisAndOr with the mouse, and MUST receives the appropriate message to show a snapshot of the execution as required. Figure 5 is a snapshot of the resulting system which has shown to be of great use at SICS.

4 Visualization of Andorra-I

Andorra-I [SCWY90, SCWY91, HJ90] is a logic language which allows *or* parallelism and Dependent *and* parallelism (DAP) between a restricted class of goals, namely determinate goals. Andorra-I workers are divided into teams. Two different teams work on different *or* branches, and all workers in a given team work in the same *or* branch. The workers into a team work on determinate parallel goals. A goal can be found to be literally deterministic at compile time, and therefore it its determinate; otherwise, finding out whether a goal is determinate or not is responsibility of a worker in the team, designated as the *master*. The other workers in the same team are designated as *slaves*. Determinate parallel conjunctive goals are executed in DAP and eagerly by the master and the slaves, whereas non-determinate conjunctive goals are reduced sequentially by the master.

Figure 6 shows the execution of an Andorra-I program. *Or* parallelism visualization is similar to that of MUSE. *and* parallelism visualization of work by slaves is done by drawing lines adjacent to the one of the master. Each one of these lines represents a slave which helps the master. Figure 7 represents a zoom of the same execution where and-parallel slaves can be observed. It is to be noted that slaves finish and join their master without a "nested loops" structure, i.e., a slave can finish before an *and* sibling which started later than it. Studying the behavior of masters and slaves is very interesting in Andorra-I, in particular in the development of flexible two-level scheduling scheme (the *or* scheduler among teams and the *and* scheduler between workers of a team), an area in which VisAndOr is being

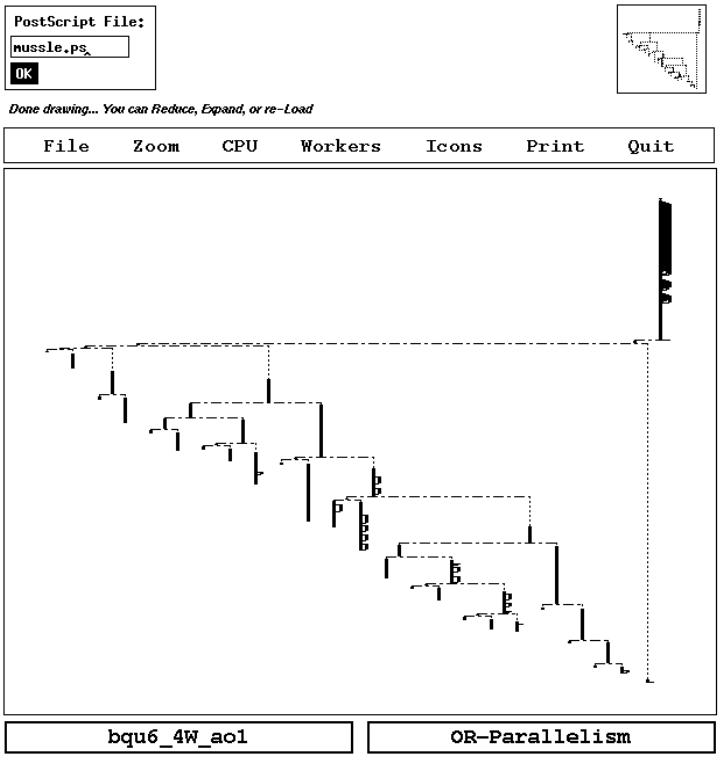

Figure 6: An Andorra-I execution

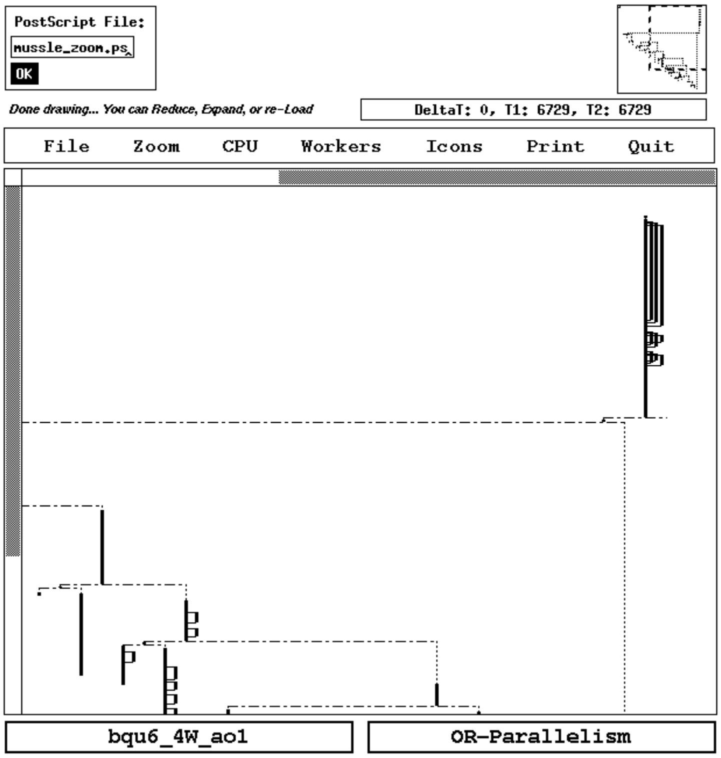

Figure 7: Andorra-I slaves

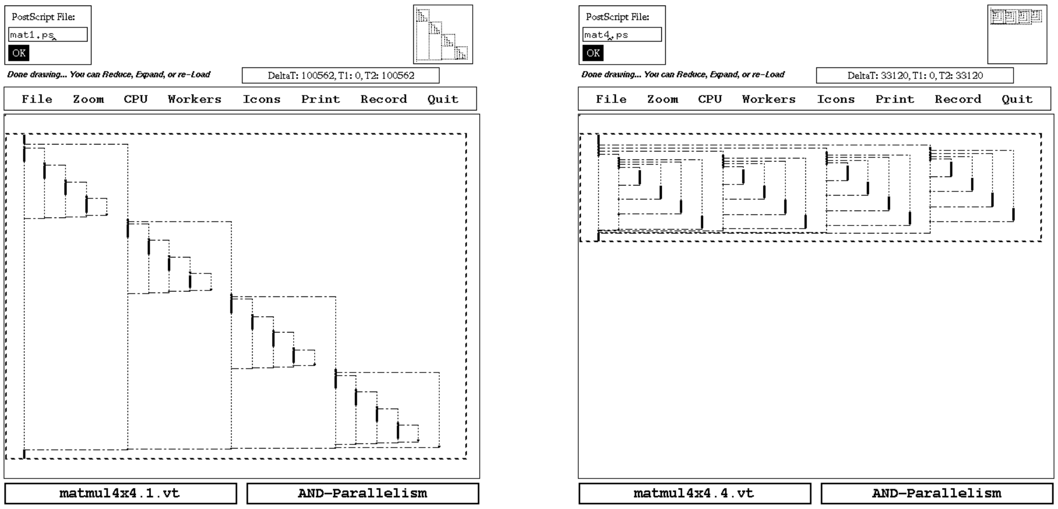

Figure 8: Sequential and Parallel execution together

effectively used at Bristol.

Another characteristic issue of Andorra-I which can benefit from the overall vision given by VisAndOr is the determinacy study. In general, determinacy of goals can only be partially determined at compile time, so run-time checks turn out to be necessary to exploit parallelism. But in some executions the time spent for such checks can be wasted, if the goals are always not determinate or, conversely, if the goals are always determinate. VisAndOr allows to see the overall effect of these issues.

5 Visualization of &-Prolog

&-Prolog [Her86, HG91] is an implementation of the Independent *and* parallelism scheme [DeG84], which relies both on compile-time analysis and on run-time checks to ensure the correctness of the parallel conjunctive execution of goals in the body of a clause [HR92].

The visualization of &-Prolog programs follows the same general patterns already seen: vertical axis is time, horizontal axis is parallelism. When two workers fork and join has also to be shown [3]. This is done by using dotted horizontal lines. Figure 8 shows, at the left, a sequential execution or a 4 x 4 matrix multiplication; it is easy to see how only one worker is active at a time, and also the general shape of the execution: four groups of four vector multiplications. At the right, the same execution, but with four processors. The processes which before were sequential now overlap in time. The rightmost snapshot retains the time scale of the leftmost one, so that execution times can be easily compared. The total amount of time in microseconds is shown in the upper right corner, and it corresponds to the height of the dashed frame surrounding the execution graph. The time with four processors is not exactly one fourth of the corresponding with only one, due to sequential components and scheduling time.

6 Applications of VisAndOr

The previous sections where mainly aimed at showing the basic characteristics of VisAndOr. Some areas in system development in which VisAndOr can be of help were briefly referred to. In this section we will address such topics as well as others which, although not being directly related to the MUSE, Aurora, and Andorra-I models, can be studied with the help of VisAndOr.

[3] When only the CGE annotation is used in &-Prolog to express run-time checks, only nested fork and join events can be expressed (for ways expressing arbitrary graphs see [MH89]). With CGE annotations, when a worker finishes a task, the task remains *suspended*, waiting for its siblings to finish.

Understanding parallel execution: As a pedagogical tool, VisAndOr can be used to help to the understanding of parallel execution of logic programs. The impact of the parallelism conditions and the number of available processors become clear when visualization is used.

Simulation and remote systems study: The VisAndOr approach does not need the real system to be useful. Trace files are enough to study the behavior of a given system. This can seem a non-scientific issue, but in practice this is a major advantage. In particular, the executions shown here were obtained at SICS (for the MUSE system) and Bristol (for Andorra-I and Aurora), and sent by e-mail. In addition, MUSE traces were initially designed for MUST, and they simply had to go through a simple translation in order to be processed by VisAndOr.

Another advantage which derives from the use of event files is that no real system is needed. A simulator could generate traces to be used in VisAndOr. It is also possible that traces obtained using real systems be processed to obtain different execution profiles. This is because a trace file includes all the relevant elements to reconstruct an execution. Once the real execution has been reconstructed, some aspects (i.e., scheduling time) could be changed in order to obtain hypothetical traces of the same program under different conditions.

Parallel system debugging: Implementing a parallel execution system is a very error-prone task, and debugging a parallel system can become agonizing using standard tools. This is specially true when such tools have been designed to work at a very low level. The generation of events in a parallel system can be used to give us a trace of the execution, where it is the programmer who decides what is relevant and what is not relevant. Under this perspective, VisAndOr can be used as a high level debugger customized to reflect the needs of the programmer. As VisAndOr shows the real execution, mistakes of a higher level of abstraction and overheads can be easily discovered and measured, because the low-level details are hidden.

Scheduling: Scheduling is a major issue in parallel execution, and the benefits of a good and fast scheduler are very important [Cal88, AK90a, Dut91, But88]. But scheduling has a dynamical nature, and understanding such behavior is a key to designing good scheduling policies or to work out improved schedulers from existent ones. For example, a flexible scheduler for Andorra-I will allow workers to migrate from one team to another depending on the amount of work available in each team. This dynamic behavior is very difficult to predict prior to real execution, and even after an actual execution, issues like paging rates and system load can be very different from one run to another. The "big picture" of VisAndOr allows the programmer to concentrate on the interaction of high-level ideas without taking into account the minor changes introduced by spurious phenomena.

Benefit of Parallel Processing: Sometimes it is not desirable to run in parallel runnable processes, because of the time and work spent in scheduling and preparing the tasks. The target of granularity analysis [DLH90, Tic88] is to find out when parallel execution is convenient in practice. The analysis performed usually uses recursion depth as the parameter which decides between parallel and sequential execution. But there are more conditions to be taken into account. In programs in which parallel execution conditions cannot be worked out at compile time, some tests have to be performed at run time. Even more, the same program with the same initial call pattern and date could, in principle, be scheduled in a different way in different machines. This could be due, for example, to different operating system capabilities, different architecture, etc. So there are at least three main threads which interact very strongly:

1. Different schedulers have different behavior according to the particular situation of the computation and working environment.

2. Systems that perform run time checks can show very different degrees of parallelism in the same program if different initial calls are performed.

3. Even with the same call pattern, initial data greatly influences the time spent in the parallelism checks and, to a lesser degree, setting up the tasks.

An analytical solution to the practical problem is almost impossible, because of the large number of independent variables involved. A possible solution is to study the behavior of programs in different machines and conditions. Again, VisAndOr turns out to be an appropriate tool. System-dependent conditions, like page faults, and data-dependent conditions, like time spent in parallelism checks, can be easily discovered with a graphical tool.

7 Extending the Event Driven Scheme

Visualization is not the only topic in which the event driven scheme is useful. Dumping data tailored to different needs is a flexible way of interfacing tools with engines, each tool having its own view of the execution's skeleton. For example, a tool to depict the inner state of the parallel WAMs in &-Prolog, for debugging and low-level tuning, is currently under development. This tool is intended to be helpful in the assesment of the correctness of parallel backtracking and garbage collecting algorithms. The approach taken to interface this tool with the &-Prolog engine is the same as in VisAndOr: an event file is generated during program execution, to be consulted later. Different events and related information are needed in this case, but this is only a minor point in the framework.

The events currently recorded for VisAndOr can be directly used for purposes other than visualization. IDRA, a tool already developed in our working group [FH92] uses as input the same traces that VisAndOr does, but with a different purpose. IDRA finds out the optimal processor allocation for a given parallel execution; all the data needed for this purpose is contained in the VisAndOr traces. With this information, an optimal scheduling and the corresponding speedup is computed for a given number of processors (finite or infinite). A new trace corresponding to that scheduling can be generated, which can in turn be visualized with VisAndOr. The speedup obtained with this trace, compared with the one obtained in a real parallel system, is a valuable indication of the quality of the actual scheduling algorithm.

8 Implementation Details and Related Problems

We have seen how VisAndOr represents a few paradigms of parallel execution of logic programs. In this section we will have a look at some implementation details and features. Some of these features have already been mentioned, but we will repeat them here, together with some others not yet alluded to. Some of them are even subject to changes and improvements.

- In a color display, VisAndOr uses a different color for each CPU.
- Numbers and characters can be displayed next to the active agent lines. This allows distinguishing among them in an black and white display. This is also useful for paradigms where WAMs and workers are not intimately associated.
- Events relevant to each execution paradigm can be displayed using icons.
- The time between two points in the display can be determined by simply selecting the two points with the mouse.
- The time scale (i.e., the time corresponding to each pixel in the screen) can be retained from one trace to another, so that execution times can be graphically compared.
- The picture can be dumped to a printer or to a PostScript file.
- Zooming in and out is possible. In this case navigation through the execution can be performed with slide bars or using the dashed rectangle which appears into the small window at the upper right corner. This small window always shows the complete execution.

- VisAndOr can communicate bidirectionally with MUST. Commands can be sent and received in order to synchronize both tools. The dialog can be maintained while MUST and VisAndOr still respond to the X-Windows system.

VisAndOr is written in C and runs under the X-Windows environment. It has been constructed using the Xt library and the Athena Widgets. They have been found to be useful, but sometimes the lack of flexibility when defining graphic objects became a problem. The inner structure is quite modular. Each feature is accessed through a call back routine activated by the corresponding button or menu item. The execution events are internally stored in a virtual space which is mapped to the real screen when a change of scale is requested. This can lead to problems when a zoom of a small region is requested, due to the lack of virtual memory in some X-Windows servers.

The difficulty of adapting the &-Prolog emulator to dump traces was not very high, although in the Sun Sparc implementation a problem arised which can also happen in other architectures. Time has to be consulted each time an event is to be recorded. Unlike machines like the Sequent Balance, in which the time is stored in a memory position, so that consulting it was very quick, Sun OS needs a system call to be performed. It was found that this system call could take a sizeable proportion of the total process time, so the absolute times were much higher when tracing. This effect was balanced by taking into account how long each system call lasted and subtracting it from the actual time.

9 Conclusions and Future Work

We have presented in an informal way some early results on the design of a series of paradigms for visualization of the parallel execution of logic programs. The results presented here refer to the visualization of or-parallelism, as in MUSE and Aurora, deterministic dependent and-parallelism, as in Andorra-I and independent and-parallelism as in &-Prolog. We have reported on a tool, VisAndOr, which has been implemented for this purpose and has been interfaced with these systems. Results have been presented showing the visualization of executions from these systems and the usefulness of the resulting tool has been briefly discussed. The approach used to interface the engines can be understood as an instance of a more general system of extracting the execution skeleton. This representation can be used for purposes other than visualization; on the other side, simulated or "massaged" executions can be depicted using the same interface.

VisAndOr is still to be improved both in the implementation dimension and in the conceptual (visualization design) dimension. In the implementation dimension, new features like processor utilization, idle times, etc., much in the style of ParaGraph [HE91] can be added. In the conceptual dimension, an extension to VisAndOr to support several other forms of parallelism and their combinations is planned. Independent *and* + *or* parallelism visualization is being studied; a 3-D scheme (one dimension for time, the other two for *and* and *or* processing) is considered. Dependent *and* parallel execution models need the producer-consumer relation and the suspension of goals to be visualized in a clear manner; this is an issue in which much can still be done.

10 Acknowledgments

The authors would like to thank Tony Beaumont, Ines Dutra, and David Warren from U. of Bristol and Roland Carlsson from SICS, for useful discussions on visualization and feedback on our tools. The connection of MuseTrace and VisAndOr was done in collaboration with Roland Carlsson of SICS. Very special thanks to Roger Nasr for his VISIPAL implementation to which our current tools owe much. Also to the members of the CLIP group at U.P.M. for their help on earlier drafts of this article and during the implementation of the tools herein presented.

References

[AK90a] K.A.M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1990. Vol. 19, No. 6, pp. 445–475.

[AK90b] K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, October 1990.

[AKK+92] Seiichi Aikawa, Mayumi Kamiko, Hideyuki Kubo, Fumiko Matsuzawa, and Takashi Chikayama. Paragraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proceedings of the Fifth Generation Computer Systems*, pages 286–293. Tokio, ICOT, June 1992.

[Bro88] M. H. Brown. Exploring Algorithms Using BALSA II. *IEEE Computer*, 21(5):14–36, 1988.

[But88] R. Butler et. Al. Scheduling Or-Parallelism: An Argonne Perspective. In *Fifth International Conference and Symposium on Logic Programming*, pages 1565–1577. University of Washington, MIT Press, August 1988.

[Cal88] A. Calderwood. Scheduling Or-Parallelism in Aurora—the Manchester Scheduler. Unpublished, July 1988.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DL87] T. Disz and E. Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 46–53, 1987.

[DLH90] 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*. ACM Press, June 1990.

[Dut91] I. Dutra. Flexible Scheduling in the Andorra-I System. In *Proc. ICLP'91 Workshop on Parallel Logic Programming, LNCS 569*. Springer Verlag, December 1991.

[EB88] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.

[FH92] M. J. Fernández and M. Hermenegildo. IDeal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal Speedups in Parallel Logic Languages. Technical Report TR Number FIM26.3/AI/92, Computer Science Faculty, Technical University of Madrid, September 1992.

[GSN+88] A. Goto, M. Sato, N. Nakajima, K. Taki, and A. Matsumoto. Overiew of the Parallel Inference Machine (PIM)8z. In *International Conference on Fifth Generation Computer Systems*. ICOT, 1988.

[HE91] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[Her86] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[HG91] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, June 1990.

[HR92] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1992. To appear (also published as Technical Report Computer Science Dept, Universidad Politecnica de Madrid, Spain, Sept 1991).

[MH89] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming*. MIT Press, June 1989.

[PSB92] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A Taxonomy of Software Visualization. In *Twenty-Fifth Hawaii International Conference on System Sciences*, January 1992.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[SS90] J. Sundberg and C. Svensson. MUSE TRACE: A Graphic Tracer for OR–Parallel Prolog. Technical Report T90003, SICS, 1990.

[Sze89] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[Tic88] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[Tic92] Evan Tick. Visualizing Parallel Logic Programming with VISTA. In *International Conference on Fifth Generation Computer Systems*, pages 934–942. Tokio, ICOT, June 1992.

[UC10] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, December 19910.