Performance Analysis of Parallel & Embedded Real-Time Systems Based on Measurement and Visualization

Javier García Martínez, Jose Luis Díaz de Arriba, Joaquín Entrialgo Castaño, and Daniel F. García Martínez

Universidad de Oviedo, Campus de Viesques, Gijón, Spain

Abstract. This paper describes an approach to carry out performance analysis on systems which combine two major characteristics: real-time behaviour and parallel computational structure. It relies on a toolset composed of a monitoring system and a visualization tool. The steps to carry out performance analysis based on the toolset are clearly defined, and they are explained by means of a case study. First, the instrumentation process is described, and later, the performance views shown by the visualization tool are presented. It is suggested that these views are highly suitable for understanding the performance of parallel real-time systems.

1 Introduction

One major concern in real-time system design is performance. This arise from the very concept of "real-time". Real-time systems (which can normally be considered, from a behavioural perspective, as devices that react to an environment) must respond to environmental requests in real time, that is, each system response must be bound in time. Hence, this kind of systems must be developed in a performance-oriented manner, so that temporal constraints can be met.

When the computational power needed to compute responses in order to fulfil temporal constrains is high, the utilization of parallel or distributed technologies becomes necessary. In this case, the inherent complexity of real-time systems, together with the difficulties introduced by parallel or distributed design, make the use of models, techniques and tools essential, in order to make the crucial task of performance analysis feasible.

In this paper we present a method based on measurement, intensive trace analysis and visualization, which provide reasonable solutions for the challenging problem of the performance analysis posed by this kind of systems.

2 Monitoring and Visualization: Related Work

The measurement system proposed in this paper is based on a software monitor, oriented to collecting event traces during application execution. Event traces are an intrusive monitoring technique, however, they provide very detailed information, which can be indispensable in order to carry out precise analysis. This is the approach followed by a wide amount of tools devoted to monitoring parallel and distributed applications. Special mention must be made about general and portable instrumentation systems, such as PICL [6] and instrumented versions of MPI, such as MPICH [7].

After monitoring, the information collected must be analyzed. Normally, event traces contain so much information that special tools must be utilized to manage and present all that information in a comprehensible format. One pioneer tool developed to represent behavioural information of parallel programs is Paragraph [4].

Other approaches combine both monitoring and visualization in integrated toolsets. Some of them have been developed for particular systems (for example [8] and [2] for transputer based systems), or for determined parallel programming paradigms (as Apshot [3] for MPI, or Annai/PMA [9] for both MPI and HPF).

One of the main problems of analyzing and visualization tools is to interpret the large amount of data generated during monitoring, in order to find the system performance problems. On many occasions the level of behavioural information is too low, and hence, difficult to digest by the analysts. To overcome this problem, some tools, as Pablo [10], allow the analyst to develop his own high level application views. However, all the analysis tools we know, including the aforementioned, fail when they are applied to parallel real-time systems, because they do not supply the means to understand the reactive behaviour of this kind of systems.

3 The Proposed Approach for Performance Analysis of Parallel Real-Time Systems

This method assumes that an implementation (at least at prototype level) of the application under study is available, in order that it can be measured. Later, the application is instrumented and executed, and the information collected during execution is used to carry out behavioural and performance analysis. The main problem is how to achieve the application instrumentation, so that the information measured can be easily understood and serves for analysis purposes.

Firstly, we must establish a behavioural model which permits the designer to reason about application behaviour. Two considerations must be taken into account in the development of this model: 1) the application is driven by external events; and 2) the application execute a certain set of activities in response to each event. We will refer to these sets of activities as *execution paths*, which can be described by means of activity execution graphs. So the behavioural model of an application can be expressed as the set of *execution paths* that it performs in response to the external events that it must deal with.

The semantic of the activity execution graphs is described in [5]. Its main characteristics are:

1. It describes partial ordering of activities based on AND, OR and sequential relationships among them.
2. Each activity is described by means of two times:
   - *Waiting time* is the interval, between when all the precedence conditions for activity execution are met, until execution really begins. It represents resource contentions or synchronization problems.
   - *Execution time* is the time elapsed from the beginning to the end of the activity execution.

This model will drive the instrumentation process, for which two types of essential information must be put directly or indirectly in the traces:

1. The beginnings of path executions (these must be instrumented in the software points where external events are recognized), and their execution numbers (since the same path is normally executed several times during a measurement session, once for each occurrence of its associated event).
2. The beginnings and ends of activity executions, and to which path and execution path number they belong.

After executing an instrumented application the traces are analyzed with a visualization and analysis tool, which must also receive a description of all the paths existing in the software. Based on this description, the tool can carry out a powerful exploration of the trace. For example, it allows the visualization of the execution paths (with execution times for each activity) for every external event and execution number. This is a novel fashion of visualizing traces obtained by direct measurement from parallel computers. The execution paths are called *macro-activities* in the tool, and we consider both of these terms as having the same meaning.

Figure 1 summarizes the steps that an analyst must follow to carry out the behavioural and performance analysis of a parallel real-time application, using the approach we are describing:

1. An application prototype, to a greater or lesser extent, is developed.
2. The designer looks for the execution paths and the activities that belong to them in the prototype software structure. The path definitions are stored in the "path description file".
3. The software prototype is instrumented in accordance with path definitions. We refer to this type of instrumentation as "path based instrumentation".
4. The software prototype is executed. At this point, the trace files with the behaviour information are generated.
5. The trace files together with the path description file are used by the visualization tool to analyze application behaviour.

4 Monitoring Tool

4.1 Tool Description

This monitor has been developed initially for a multiprocessor based on the T9000 Transputer [1] architecture, and recently ported to a parallel architecture

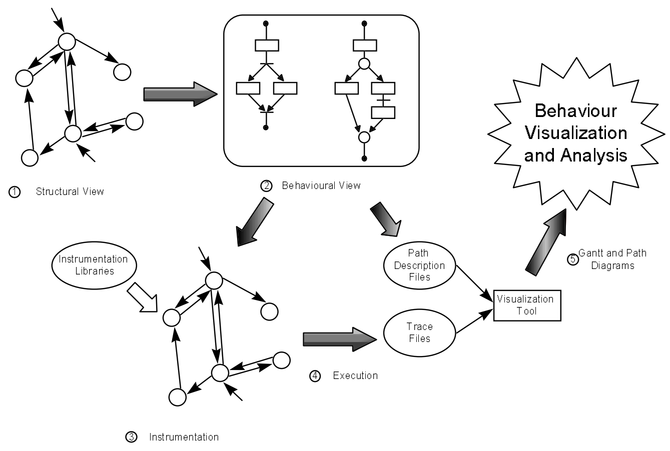


Fig. 1. Performance and behavioural analysis methodology

based on the Texas Instruments TMS320C4x. The ideas behind both implementations are the same, and the only differences affect to the low-level details (as accessing to the hardware timer, for example).

Monitor Functionality. The function of the monitoring system is to trace the occurrence of the most relevant software events during application execution, and store information related to them in a set of trace files. Therefore, in order to specify the functionality of the monitoring system, the set of relevant software events has been defined.

1. *Run-Time system events*: these represent calls to the run-time system. The main ones are: communications; synchronization operations, such as timers, and alternations; and the operations with the I/O subsystem.
2. *Monitoring system state events*: these express important states reached by the monitoring system, such as, the start and end of monitoring, or the filling of the monitoring buffer. They are very important, in order to interpret the trace correctly.
3. *Activity execution events*: these determine: the moment an activity is executed; in response to what external event; and its execution state (waiting or executing). They provide fundamental information to analyze the behaviour of the system, using the behavioural model described in section 3.

Monitor Structure. The monitoring system is structured in two main components; a *distributed monitor*, and a collection of *instrumentation probes* spread over the application processes.

The *distributed monitor* is made up of a set of *monitoring processes* (one per CPU of the multiprocessor), and a *central monitor*. The monitoring processes hold the *buffers* in which events are stored. The central monitor is devoted to synchronizing the main activities that must be carried out by the monitoring system during a measurement session.

The *instrumentation probes* are the elements that capture event occurrence. They are inserted in the source code of the application in two ways: either by using instrumented runtime functions or by using special instrumentation functions (supplied by a special purpose instrumentation library).

Monitor Intrusiveness. In order to minimize the monitor intrusiveness, the information of each event stamp is stored in the memory, in a very compact format, and only in the final stage (when the application has finished) is it transferred to the disk.

The time required for storing an event in the memory does not depend on the type of event. It was measured for our target architectures, and resulted of only 15.8μs for the TMS320C4x and 31μs for the Transputer.

4.2 The Instrumentation Process: Case Study

We will consider a synthetic example, simple enough to understand, but which exhibits behaviour complex enough to merit an in-depth analysis.

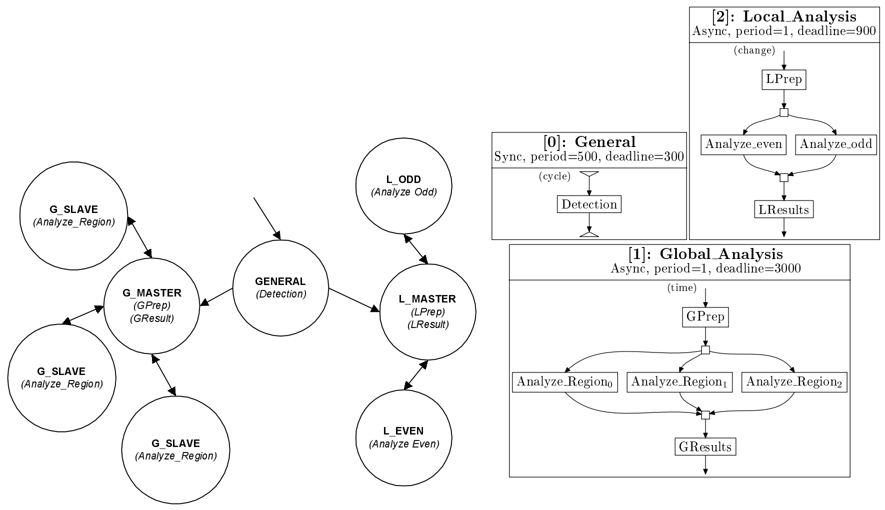
Our example involves a (hypothetic) real-time vision system, which receives an image from a video camera and performs the following tasks: first, it determines if any substantial change has occurred in the scene, in which case an object recognition task is issued on the changing region of the image. In addition, a routine task which computes some global statistics is performed on the whole image at regular intervals.

The system is implemented by means of eight processes, whose communication structure is shown in Fig. 2(a). The process called *General* is awakened each 500 milliseconds. It issues an image scan, decides if a substantial change has occurred, and if necessary, awakes the tasks *L_master* (if a change has occurred) or *G_master* (if it is time to do a global analysis).

The tasks *L_master* and *G_master* exhibit similar behaviour. Both are asleep until a message is received from *General*; then both issue a preprocessing on the data received after which they issue a partition of the data between their workers, and wait until the answer is received. Finally both processes assemble the received results and go to sleep again until the next data packet arrives from *General*.

Analyzing this implementation, the following macro-activities (*execution paths* shown in Fig. 2(b)) are determined:

1. *General*: A synchronous macro-activity (automatically triggered by a timer each 500 ms) which performs the movement detection on the image. It consists of a single activity called *Detection* (implemented in the process also called *General*).



(a) Structural view (b) Behavioural View

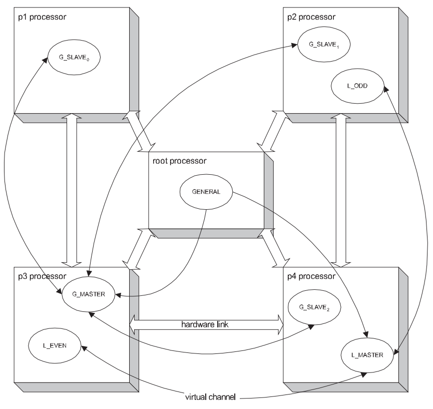
Fig. 2. Case study description

2. *Local Analysis*: An asynchronous macro-activity (triggered by the macro-activity *General* when a movement is detected) which performs a detailed analysis of the changing region. This macro-activity consists of the following activities: the preprocessing task *LPrep* (implemented in the process *L_master*); the tasks *Analyze Even* (implemented in the process *L_even*) and *Analyze Odd* (implemented in the process *L_odd*) which can be performed in parallel; and the final aggregation of the results in task *LResults* (also implemented in the process *L_master*).
3. *Global Analysis*: An asynchronous macro-activity (triggered by the macro-activity *General* at regulars intervals) which performs a global analysis of the image in order to calculate a set of statistics and store them in a database. This work split among three CPUs (*workers*). The structure of this macro-activity is similar to the structure of the *Local Analysis*.

The instrumentation of the source code is carried out in a semi-automatic way. All the communication through the virtual channels and other low-level services are instrumented automatically. However, the instrumentation related to the macro-activities must be done manually. This instrumentation consists of writing a number of “macros” in the appropriate points of the source code, as shown in Fig. 3(a). These macros are defined in external header files, and expanded by the preprocessor of the compiler. The resulting code is the *instrumentation probes* discussed in section 4.1.

```
/* ... previous listing omitted ... */
main(int argc, char *argv[], char *envp[],
     CHAN *pin[], int ins, CHAN *pout[], int outs)
{
   /* >>>>START<<<< */
   do
   {
      /* Blocking communication */
      receive_region(region, pin[0]);
    BEGIN_MACRO_ACTIVITY_Local_Analysis;
    BEGIN_ACT_LPrep;
      preprocess(region);
    END_ACT_LPrep;
      send_even_lines(region, pout[1]);
      send_odd_lines(region, pout[2]);
      /* Wait until the workers finish */
      receive_even_results(even_results, pin[1]);
      receive_odd_results(odd_results, pin[2]);
    BEGIN_ACT_LResults;
      compute_results(even_results, odd_results);
    END_ACT_LResults;
   } while (1);
   /* >>>>END<<<< */
}
```

(a) Source code instrumentation (fragment of `L_MASTER`)



(b) The mapping of the processes on the target hardware

Fig. 3. Case study instrumentation and mapping

The mapping of processes to CPUs is done statically (in compilation time), by means of a configuration file. We have chosen the mapping shown in Fig. 3(b), which also shows the hardware architecture of our target system. The process *General* needs to be mapped on the *root* CPU, because it needs to access to the video hardware intensively, and only the *root* CPU has direct access to the video memory. The other processes are distributed among the remaining CPUs.

The execution of the instrumented application will generate a set of traces with the information about where and when significant events occurred in the application (beginnings and endings of communications, activities and macro-activities, etc.) This great amount of information must be visualized off-line with a tool in order to gain understanding of the run-time behaviour of the system.

5 Visualization Tool

This tool reads the trace files originated in the execution of the instrumented application, as well as a set of configuration files (automatically generated) which describe the software structure and the relationship between the numeric identifiers stored in the trace files and the logical identifiers (such as process names and CPUs names), which are more meaningful for the user.

With the information obtained, the visualization tool builds several views of the software, which the user can browse. These are better explained in relation with our case study.

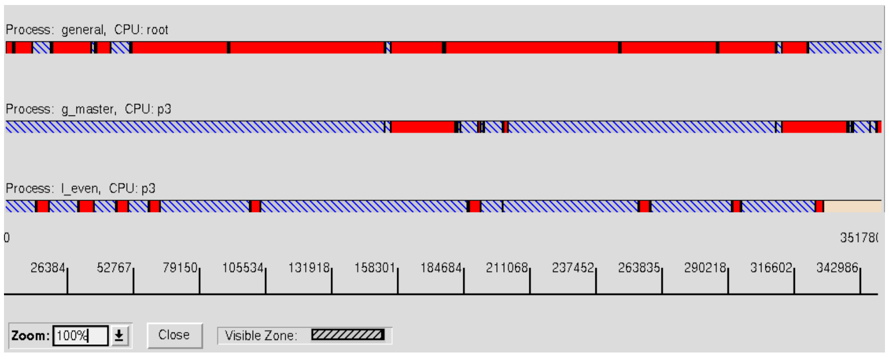


Fig. 4. Gantt diagram of processes

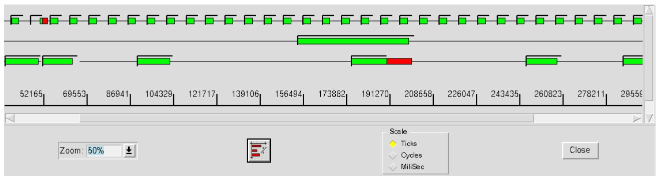


Fig. 5. Gantt diagram of macro-activity executions

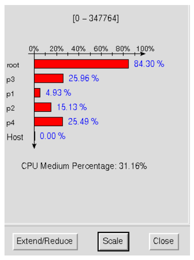
5.1 Performance Analysis: Case Study

The visualization tool provides several views of the behaviour of the system. The first is the typical Gantt diagram of processes (shown in Fig. 4). In this diagram each horizontal bar represents one process of the system. These bars change colour according to the state of the process. The most common states are: "computing" (red), "blocked for communication" (gray) and "blocked for host service" (blue).

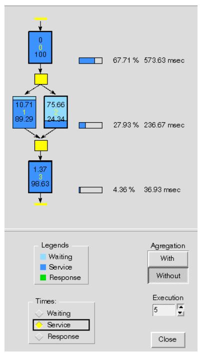
This view is useful for getting a visual impression about the relationship between computation and communication, detecting when two or more processes are competing for the CPU time.

Another view is the Gantt diagram of CPUs (not shown). In this diagram each CPU can only be in two states: busy (when one or more processes are running on it) or idle (when all its processes are blocked). This view is useful for determining the degree of utilization of the resources. This kind of information can also be shown in a bar diagram (see Fig. 6(a)).

But all this information is too low level for the designer of the application. The high level vision is provided by the Gantt diagram of the macro-activities (Fig. 5) and the execution paths diagrams (Fig. 6(b)). In the first view we can see three horizontal lines, representing the starting and ending points of each of the three macro-activities. The first line corresponds to the synchronous task "*General*".



(a) Processor resource utilization



(b) Execution path view, showing the fifth cycle of the macro-activity "Local Analysis"

Fig. 6. Other displays of the tool

Each firing of this macro-activity is represented as a green rectangle. The bracket over the rectangle shows the expected firing instant and the expected duration (deadline). If the activity ends after its corresponding deadline, the rectangle is shown in red. We can see how almost all the executions of the macro-activity number 0 fulfil their deadlines.

The second line corresponds to the macro-activity "*Global Analysis*", and in the period of time analyzed it is fired only once, and it finishes within its deadline.

Finally, the third line corresponds to the macro-activity "*Local Analysis*". The firings of this macro-activity do not exhibit a regular pattern (it is fired when a change is detected in the image). We can see how this macro-activity is fired twice in a row at the beginning of the analyzed period (causing the macro-activity *General* to wait for it, violating its deadline), but after this it is activated less frequently. In the period analyzed we can also see how an execution of the macro-activity *Global Analysis* coincides with the execution of the macro-activity *Local Analysis*. Since these macro-activities have workers that compete for CPU time (in processor *p2*), this interference causes the violation of the deadline of the macro-activity *Local Analysis* for this execution.

In Fig. 6(b) we can further investigate the consequences of this competition for resources. This figure shows the path execution for the macro-activity *Local Analysis* in its fifth cycle (which is when the conflict arises). This view represents each activity which makes up the macro-activity as a box, and each temporal dependence as an arrow, and hence it is similar to one of the behavioural views shown in Fig. 2(b). But in this case the boxes representing each activity are filled with data calculated from the trace files. We can see how much time was spent in each activity, how much was spent *waiting* (defined in section 3), and so on.

6 Conclusions and Future Work

The need of performance analysis approaches for systems that combine both parallel and real-time characteristics has been posed, and a method for addressing this kind of analysis on parallel embedded systems has been proposed.

Our approach is supported by a toolset which is composed of a monitoring system and a visualization and analysis tool. The monitoring system has been developed for two platforms: a T9000 transputer based system and a DSP based system. Their trace formats have been normalized, so that the visualization tool can be utilized with both of them.

The steps to carry out performance analysis based on the toolset have been clearly defined, and we have proposed a new way of achieving software instrumentation, referred to as "path based instrumentation". The behavioural information provided by this kind of instrumentation allows very understandable performance visualization and analysis, based on execution paths (macro-activities).

In summary, the presented approach provides highly understandable performance views with a reasonable effort on the part of the application designer.

In regard to the future work, one of the major inconveniences to deal with is intrusiveness. This cannot be eliminated when software monitors are used, but it would be very interesting to model its influence on performance analysis results.

References

1. Parsys Ltd. *SN 9500 Technical Overview*, 1994.
2. L. Schäfers and C. Scheidler. Monitoring the T9000: the TRAPPER approach. In *Proc. of Transputer Applications and Systems '94*, September 1994.
3. V. Herrarte and E. Lusk. Studying parallel program behaviour with upshot. Technical Report ANL-91/15, Argonne National Laboratory, 1991.
4. M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, September 1991.
5. C. M. Woodside. A three-view model for performance engineering of concurrent software. *IEEE Trans. Software Eng.*, 21(9), September 1995.
6. G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to picl: A portable instrumented communication library. Technical Report TM-11616, Oak Ridge National Laboratory, 1990.
7. E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. Technical report, Argonne National Laboratory and NSF Engineering Research Center (Mississippi State University), 1997.
8. R. Borgeest, B. Dimke, and O. Hansen. A trace based performance evaluation tool for parallel real-time systems. *Parallel Computing*, 21:551–564, 1995.
9. B. J. N. Wylie and E. Endo. Annai/PMA multi-level hierarchical parallel program performance engineering. In *Proc. 1st International Workshop on High-level Programming Models and Supportive Environments*, April 1996.
10. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F.Tavera. Scalable performance analysis: The pablo performance analysis environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.