Scheduling in DACAPO

Roger Johansson
Chalmers University of Technology
Department of Computer Engineering
DRAFT(00-03-08)

Abstract

The Dacapo architecture has been designed as a platform on which distributed safety-critical applications are executed. Great care must be taken when specifying, designing and implementing such applications. The scheduled application must be analysed and proven capable to meet every time-requirement.

The Dacapo scheduler is a tool that provide such analyse and, moreover has been designed to meet special requirements introduced by control laws. This report describes the first implementation of the Dacapo scheduler.

Keywords: Distributed Real-Time Systems, Static cyclic scheduling, Fixed Priority Scheduling.

INTRODUCTION

The Dacapo architecture has been designed to meet the requirements of a hard real-time system for distributed control of safety critical applications[Bri93]. The Dacapo architecture implements hardware fault-tolerance by means of redundant units and supports software fault-tolerance by means of an application development system and a dedicated operating system. From the programmers point of view, Dacapos main features are:

- hardware error detection with high coverage
- fine grain synchronisation of global time
- predetermined "semi"-consistency of distributed global data
- fast recovery from transient faults
- short and predictable communication latencies

In this paper we discuss the application development process and focus on the scheduler. The paper is structured as follows: the next section gives an overview of the DACAPO architecture and its communication subsystem. Section 3 deals with the application design process; structuring, decomposition and task allocation. In section 4 we give an overview of the scheduler. Section 5 illustrates use of the scheduler as well as a simulator used to visualize the schedule. Finally, section 6 offers a discussion and outlines of future work.

OVERVIEW OF THE DACAPO ARCHITECTURE

A DACAPO system consists of *nodes* interconnected by a *Global Bus*. Each node has a set of sensors and actuators and the node is supposed to be located near the physical objects that it controls. In each node, there is an *embedded controller* (EC) with a local memory. Moreover there is a *dual-port RAM* (DPRAM) that interfaces the EC to the *global communication interface* (GCI) [Sn95]. Figure 1 shows an example of a DACAPO system with 4 nodes.

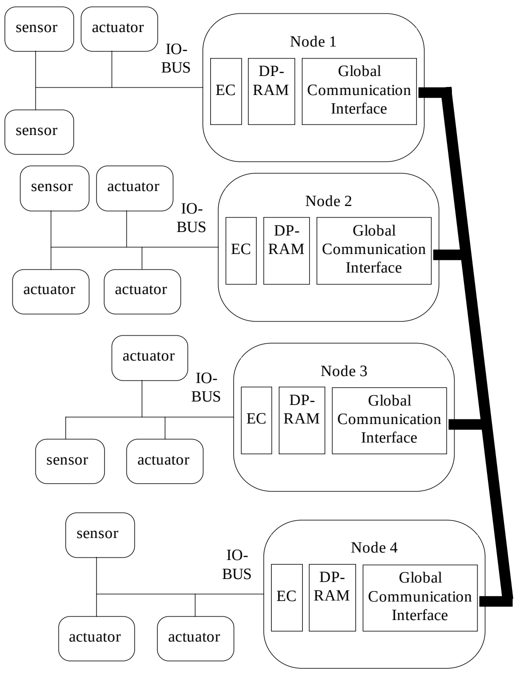

Figure 1: Sample DACAPO system

DACAPO utilises a synchronous bus protocol that repetitively transfers data on the bus. This activity is periodic and exhibits the *system cycle* (SC) (See figure 2) as its life-time. Each SC is divided into *N communication cycles* (CC), which in turn, are divided into *communication frames* (CF). During one CC; each node transmits one message and thus, the number of CF:s equals the number of nodes in the system.

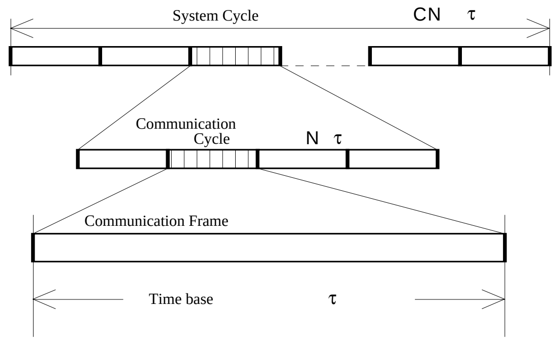

Figure 2: Periodic intervals in DACAPO

The GCI provide a fine grain global synchronisation as well as predictable internode message latency. Global time is divided into time slices and each time slice has the duration of one CF. GCI provides a signal indicating the start of each time slice in the node. Communication is pre-scheduled and fully time-deterministic. Apart from handling communication between nodes, each GCI maintains the nodes view of the global data pool. The DPRAM interfaces GCI to the node's CE and updates four entries (64 bits) in the DPRAM during each communication frame.

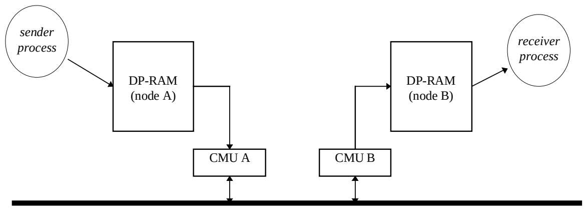

Figure 3: Principle of message passing in Dacapo

It is possible to know what data has been updated at any point in time because of the pre-scheduled communication. The following characteristics are essential for calculating the worst case response time of a message:

1. Queuing of messages from the sender node
2. Latency for the sending node to become the bus-master, i.e. await next CF for the node
3. Delay for transferring message on the bus
4. Delay for updating the receiver node's DP-RAM

Queuing of messages may result from a temporary high rate of produced messages in a single node when the required bus capacity exceeds 64 bits/CC.

Since the node is only allowed to transfer messages in one CF of a CC, this latency is determined by *in which* CF the message is produced. The Worst Case Latency, assuming no queuing, for messages from node *i* appears when the sender process response time exceeds the nodes CF (CF_i) in the current CC (Figure 4). The Worst Case Latency then becomes *one* CC. Best Case occurs when the sender process response time is less than start of CF_i (Figure 5).

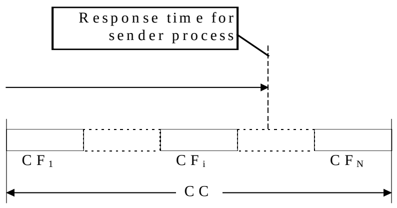

Figure 4: Worst case for message latency

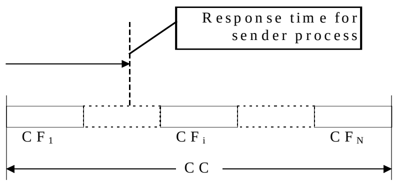

Figure 5: Best case for message latency

Delay for transferring the messages on the bus, that is the duration of one CF, is determined by hardware (currently 118 micro seconds) and the delay for updating the receiver node's DP-RAM (currently < 4 microseconds).

APPLICATION DESIGN METHODOLOGY

The vast majority of tasks in a DACAPO application are performing periodic control There is however a need for support of sporadic events as well. A task is further classified as either *safety-critical* or *not safety-critical.*

A task is "mapped" onto DACAPO processes with distinctive properties. A process may be executed repeatedly, once in each fixed period of time. We call such activity a *red* activity. On demand, or sporadic tasks, are called *blue* activities.[1] A fundamental property in DACAPO is that this important fundamental assumption that *safety critical tasks should be periodically executed and the associated processes should be guaranteed to terminate within its deadline.* As a consequence, a safety critical task is classified as a red activity and should thus be performed periodically regardless of the nature of the task in other aspects.

DACAPO uses a pre-run-time scheduling strategy to guarantee that all red activity time requirements are fulfilled while blue activities are scheduled during run-time. All messages passing between nodes are regarded as red activities in DACAPO. Since a DACAPO task, is a formal description derived from a

[1] The concept of *red* and *blue* activities was introduced in [Base95]

high level specification, a task may contain red as well as blue activities. For example, a task that is partitioned in two nodes may consists of two blue activities, one in each node. They may be connected via the message passing mechanism in DACAPO. Since global message passing is a red activity (by definition) the entire task is built from red as well as blue activities (processes).

As another example of a heterogeneous task we may consider the control of a car's side window. Start of motion of the side window elevator is triggered by a blue process since it's quit rare and certainly not safety-critical. When the elevator has started it is regarded as a red activity in order to prevent human injury or other damages. In contrast to this consider control of the "airbag" in a car which should be monitored by a red activity since the latency of a blue activity (if classified so) would *eventually* cause the airbag to blow in case of an accident.

The above examples show that there is no obvious connection between the events that invokes safety-critical handling and other events by means of periodically monitoring versus event triggered monitoring. Thus events with a periodic nature do not necessarily map onto a red activity and events of sporadic nature can be regarded as red activities. Rather, the examples suggests a design-heuristic view of the Real-Time-System design. Each task is analysed, structured and decomposed into processes to which the systems resources are allocated, made at an early state (before compile time) by the system designer. (See for example [Tö95].)

After structuring and decomposition of the applications entire set of tasks into red and blue processes, this formal description is compiled. During the compilation, program flow is analyzed and machine code is merged with directives for the scheduler. After translation the application is assembled, analyzed with respect to time properties and scheduled. The final schedule is then verified as feasible and fed into a simulator capable of visualizing the entire schedule.

Execution time analysis

Temporal aspects of the application should be analysed during, at least, three major phases. During the first phase, the analysis is performed at *basic block* level. Sequential lists of machine instructions are examined and an *Estimated Worst Case Execution Time* (EWCET) is established for each list. During the second phase a *Timing Graph* [Nie91], representing the program timing properties is established and reduced to obtain an EWCET for the program. Finally, schedulability analysis is used to verify the schedule of the entire application. Analysing the timing behaviour of a program soon becomes a non-trivial exercise for pipelined processors and in the presence of cache memories. Worst case assumptions often lead to useless results due to their too pessimistic approach. However, there exists methods by which we can reduce the overestimation of pipelined processors execution time and analyse dynamic program behaviour in order to eliminate infeasible paths thereby reducing the overestimation [Rts99].

Scheduling

A DACAPO system is always assumed to be resource adequate with respect to all safety critical processing. As a consequence, the Rate-Monotonic Criteria [JoPa86] is generally satisfied and finding a feasible schedule, is not the difficult part. The Dacapo off-line scheduler is designed to meet the special requirements emerging from typical control applications. Sensors and actuators related to the same control object may be connected to different nodes in distributed systems. Delays are introduced in the control loops, mainly, as a consequence of communication delays. These might be minimised to get a high control performance. The goal is then to schedule processes that perform sampling to complete just before the next free communication frame (CF) mastered by the node in which the process executes and to start the related actuation process (or processes) as soon as the sample has arrived. The time between sampling and actuation for a given control loop is generally required to be constant in consecutive samplings even if this means it cannot be minimal. This is because the control laws are designed with a specified delay compensation. A *varying control delay* invalidates the compensation and cause reduced performance. Reduced performance and even instability is also caused by variations in the sampling frequency, *jitter*. Periodic processes for sampling and actuation must therefore be scheduled to execute with a fixed period. This may require a higher sampling frequency to fit in the schedule and thus a modified control law.

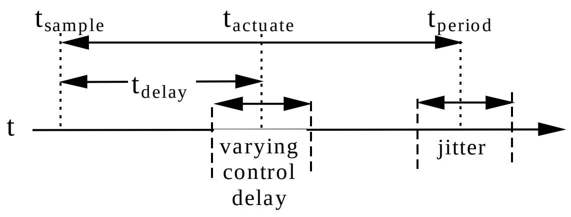

Figure 6: Scheduling limitations cause control delay and jitter.

THE SCHEDULER

For all red activities, Dacapo uses *off-line*, scheduling strategy to prepare a table (static schedule, which determines when the associated processes are executed. The table is followed with the schedule repeated cyclically at run-time. Each process that has periodicity requirements, has been assigned a runtime period that is either equal to or a multiple of another process. The duration of the schedule is thus equal to the *least common multiple* (LCM) of all the red processes in the application. The LCM is constructed, during scheduling, to fit the system cycle (SC) in Dacapo. After that, the blue processes are added assigned a lower fix priority than the red processes. Finally the resulting schedule is analysed and visualised.

To prepare a schedule, the following major actions must take place:

1. Assemble application specific tasks characteristics
2. Decompose tasks into red and blue processes
3. Define groups of processes as "transactions"
4. Establish final periods for all red activities
5. Assign priorities, starting with red processes
6. Verify the schedule using fixed priority scheduling analysis
7. Simulate and visualise the schedule output

For the analysis, in step 6 we have adopted previous work at the University of York [Tin93] which we modified and extended to fit the distributed Dacapo architecture.

Current analysis does not allow a process to receive more than one message. The workaround is to introduce "abstract" processes (with utterly short execution times) during the analysis and then assign an offset, large enough, for the real receiver process release time.

EXAMPLE:
Three different processors (in different nodes) are allocated to processes Ps1, Ps2 and Pa. Pa receives messages from Ps1 and Ps2.(Figure 7)

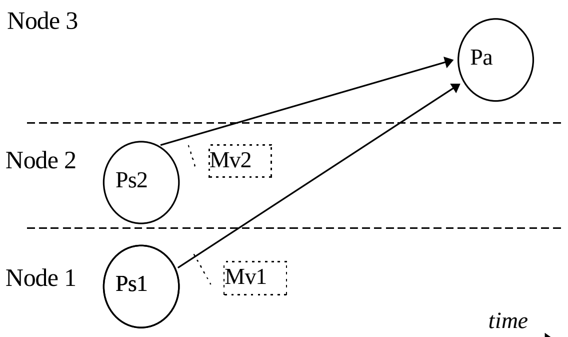

Figure 7:

Since we want the analysis to account for this message passing, we introduce the "abstract" processes Pr31 and Pr32. Pr31 receives the message (Mv1) from Ps1. Pr31 is located in the same node as the actual receiver process and the abstract process Pr32 analogous. Then we form a "barrier" built of these abstract processes and instruct the scheduler to delay the actual receiver process until the barrier is achieved.(Figure 8).

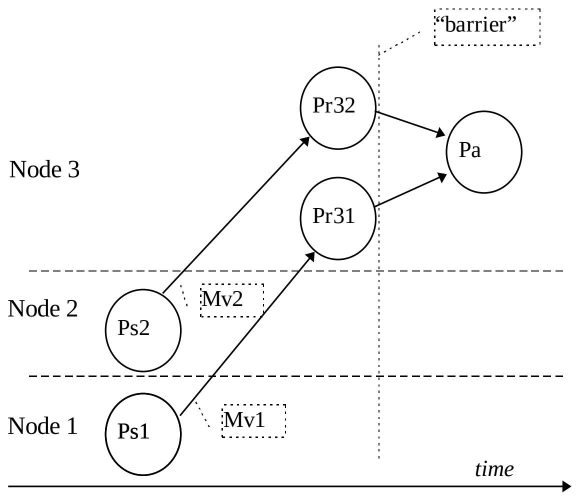

Figure 8:

Scheduler input
Input to the scheduler is provided as a text file with directives. Some of these directives are designed with application specific requirements in mind while others can be recognised as quite general. We will now turn to a brief description of those directives that are of importance for the example in the next section.

The scheduler starts with gathering information about the system. The PROCESSOR-directive is used to introduce a unique processor and connect this processor to a physical node in the system. There might be arbitrary number of processors in a node:

```
PROCESSOR   name  physical_node
```

where:
- *name*, is a unique character string identifying the processor
- *physical_node*, is an integer specifying the actual target node in the Dacapo system

---

EXAMPLE:
The directives
```
PROCESSOR   cpu1  1
PROCESSOR   cpu2  2
```
introduce two processors called "cpu1" and "cpu2" allocated to physical node 1 and 2.

The PROC-directive details a process. It is described by the following scheduler input:

```
PROC  name  cpu trans D C T J O
```

The attributes of a process are;

- *name*, a unique character string identifying the process
- *cpu*, is a unique character string denoting the processor allocated to the process
- *trans*, a unique character string denoting the transaction to which the process belongs (further discussed below)
- *D*, the process deadline
- *C*, the process estimated worst case execution time
- *J*, the process release-jitter
- *O*, the process offset within the transaction

Each process is a member of one (and only one) *transaction*. A transaction is specified by the TRANS-directive as:

```
TRANS name  T
```

where:

- *name* is a unique character string identifying the transaction
- *T* is the transaction period in micro-seconds

EXAMPLE:
Two processes "pa" and "pb" with the same periodicity (100 ms), the same deadline (10000 micros) and the same execution time (8000 micros) where we want "pa" to execute 50 ms before "pb" is expressed as:

```
....
PROCESSOR   cpu1  1
TRANS       t1    100000
PROC pa cpu1 t1 10000 8000 0 0
PROC pa cpu1 t1 10000 8000 0 50000
....
```

Message passing is introduced to the scheduler by a MESSAGE-directive:

```
....
MESSAGE mID size FROM sID to dID
```

where:

- *mID*, is a unique character string identifying the message
- *size*, is the number of bytes in the message
- *sID*, is the source of the message
- *dID*, is the destination of the message

In order to facilitate message passing of the type "several to one", two different directives is used.
The WAITBARRIER-directive instructs the scheduler to introduce an offset, large enough to await the latest process given in the corresponding BARRIER-directive.

EXAMPLE:

```
BARRIER barr1 WAITS FOR pa pb pc
```

states that termination of processes "pa", "pb" and "pc" constitutes an earliest release-time for some process or processes and the directive

```
WAITBARRIER barr1 BY pd pe END
```

identifies the processes "pd" and "pe" that have to wait. Each barrier must be uniquely defined by a character string such as "barr1" in this example.

We summarise the descriptions of these scheduler directives by the following sample input file. This input describes the first example in this paragraph:

```
-- SAMPLE1.SI
-- Models 2 to 1 message passing
tauCF          118
PROCESSOR      cpu1 1
PROCESSOR      cpu2 2
PROCESSOR      cpu3 3
TRANS   trans1 100000
TRANS   trans2 100000
TRANS   trans3 100000
PROC    Ps1     cpu1    trans1 550 500 0 0
PROC    Ps2     cpu2    trans2 620 600 0 0
PROC    Pr31    cpu3    trans3 2000 1  0 0
PROC    Pr32    cpu3    trans3 2000 1  0 0
PROC    Pc3     cpu3    trans3 3000 700 0 0
MESSAGE  Mv1 2 FROM     Ps1     TO      Pr31
MESSAGE  Mv2 2 FROM     Ps2     TO      Pr32
BARRIER        barr WAITS FOR Pr31 Pr32 END
WAITBARRIER    barr BY Pc3 END
```

Scheduler output is illustrated in Figure 9.

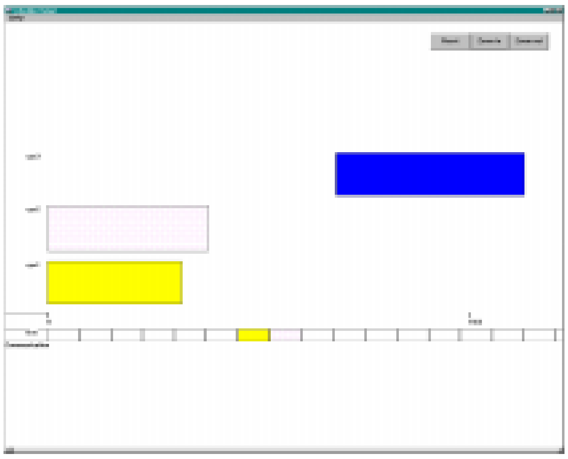

Figure 9

In the next paragraph we will turn to a more complex example and further describe and discuss the tool.

Scheduling Example

The Computational Model

Our sample model for safety-critical distributed control is a car. There are five nodes in the system. One node is located near each wheel and one node is located near the instrumental panel. Each wheel-node has a sensor, indicating the angular velocity of the wheel and an actuator that affect break pressure. The drivers break pedal is monitored by the instrumental node (Figure 10).

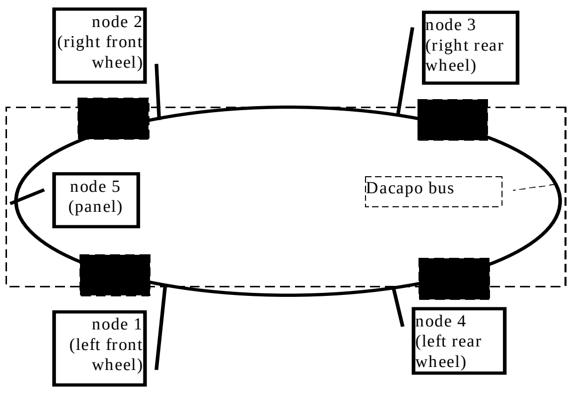
node 2 (right front wheel)
node 3 (right rear wheel)
node 5 (panel)
Dacapo bus
node 1 (left front wheel)
node 4 (left rear wheel)

Figure 10: Computational Model

We further make the following assumptions:

- In each wheel-node the velocity is sampled periodically.
- Each sample is distributed throughout the system.
- In each wheel-node, there is a process that computes a new value for the actuator. This value is based on samples from every node, i.e., other wheel's velocity as well as the break pedal pressure.
- Sampling and actuation should start concurrently, at the same time, in the wheel nodes.

We denote the sampling processes "PSx", where "x" stands for the node, the computation processes "PCx" and the actuation processes "PAx" analogous. Finally we denote the sampling process in the instrumentation node "PB". Figure 11 illustrates how the nodes are allocated to processes as well as the process communications

For the execution times we do the following general assumptions:

- Sampling processes are short. We assume different execution times for front and rear wheel sampling processes (perhaps due to different types of sensors).
- The computation processes are relatively long, but not equal, due to unique control algorithms for each wheel.
- The actuation processes are short and their execution time equals.
- All processes have the same periodicity (100 ms)

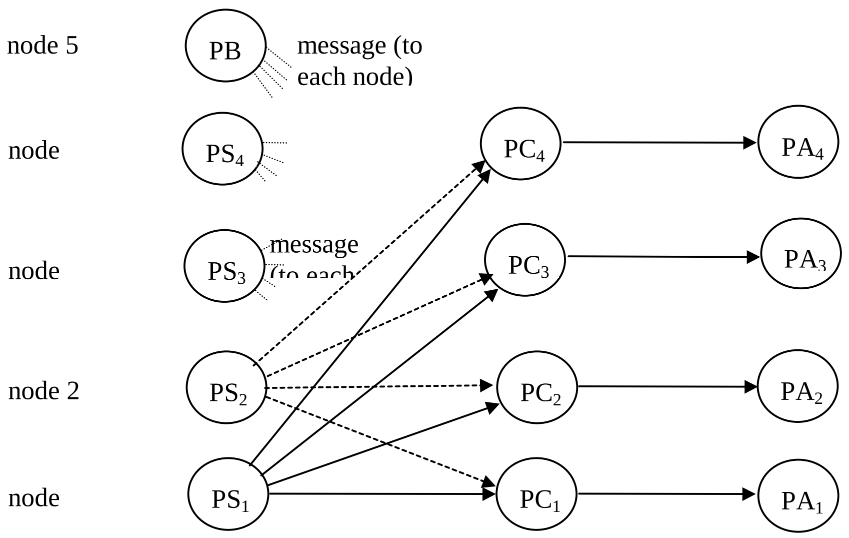

Figure 11: Process allocation and interprocess communication

We summarise our assumptions about worst case execution times and dependencies in the following scheduler input:

```
-- 4COMM.SI
-- Models 4 to 4 communication
tauCF     118
PROCESSOR cpu1 1
PROCESSOR cpu2 2
PROCESSOR cpu3 3
PROCESSOR cpu4 4
PROCESSOR cpu5 5
TRANS     trans1 100000
TRANS     trans2 100000
TRANS     trans3 100000
TRANS     trans4 100000
TRANS     trans5 100000
PROC      Ps1       cpu1      trans1  1200   900  0 0
PROC      Pr11      cpu1      trans1  3000    1   0 0
PROC      Pr12      cpu1      trans1  3000    1   0 0
PROC      Pr13      cpu1      trans1  3000    1   0 0
PROC      Pr14      cpu1      trans1  3000    1   0 0
PROC      Pr15      cpu1      trans1  3000    1   0 0
PROC      Pc1       cpu1      trans1  10000  7000 0 0
PROC      Pa1       cpu1      trans1  1000   500  0 9000
PROC      Ps2       cpu2      trans2  620    600  0 0
PROC      Pr21      cpu2      trans2  3000    1   0 0
PROC      Pr22      cpu2      trans2  3000    1   0 0
PROC      Pr23      cpu2      trans2  3000    1   0 0
PROC      Pr24      cpu2      trans2  3000    1   0 0
PROC      Pr25      cpu2      trans2  3000    1   0 0
PROC      Pc2       cpu2      trans2  13000  7000 0 0
PROC      Pa2       cpu2      trans2  1000    500 0 9000
PROC      Ps3       cpu3      trans3  800    700  0 0
PROC      Pr31      cpu3      trans3  3000    1   0 0
PROC      Pr32      cpu3      trans3  3000    1   0 0
PROC      Pr33      cpu3      trans3  3000    1   0 0
PROC      Pr34      cpu3      trans3  3000    1   0 0
PROC      Pr35      cpu3      trans3  3000    1   0 0
PROC      Pc3       cpu3      trans3  13000  6500 0 0
PROC      Pa3       cpu3      trans3  500    500  0 9000
PROC      Ps4       cpu4      trans4  800    700  0 0
PROC      Pr41      cpu4      trans4  3000    1   0 0
PROC      Pr42      cpu4      trans4  3000    1   0 0
PROC      Pr43      cpu4      trans4  3000    1   0 0
PROC      Pr44      cpu4      trans4  3000    1   0 0
PROC      Pr45      cpu4      trans4  3000    1   0 0
PROC      Pc4       cpu4      trans4  13000  7500 0 0
PROC      Pa4       cpu4      trans4  500    500  0 9000
PROC      Pb        cpu5      trans5  1000    500 0 0

BARRIER allrec1 WAITS FOR Pr11 Pr12 Pr13 Pr14 Pr15 END
WAITBARRIER         allrec1 BY Pc1 END

BARRIER             pa1_ready WAITS FOR Pc1 END
WAITBARRIER         pa1_ready BY Pa1 END

BARRIER   allrec2 WAITS FOR Pr21 Pr22 Pr23 Pr24 Pr25 END
WAITBARRIER         allrec2 BY Pc2 END

BARRIER   allrec3 WAITS FOR Pr31 Pr32 Pr33 Pr34 Pr35 END
WAITBARRIER         allrec3 BY Pc3 END

BARRIER   allrec4 WAITS FOR Pr41 Pr42 Pr43 Pr44 Pr45 END
WAITBARRIER         allrec4 BY Pc4 END

BARRIER   Pc_ready WAITS FOR  Pc1 Pc2 Pc3 Pc4 END
WAITBARRIER         Pc_ready BY Pa1 Pa2 Pa3 Pa4 END

MULTICAST Mv1 2 TO PROC Pr11 Pr21 Pr31 Pr41 FROM Ps1
MULTICAST Mv2 2 TO PROC Pr12 Pr22 Pr32 Pr42 FROM Ps2
MULTICAST Mv3 2 TO PROC Pr13 Pr23 Pr33 Pr43 FROM Ps3
MULTICAST Mv4 2 TO PROC Pr14 Pr24 Pr34 Pr44 FROM Ps4
MULTICAST Mv5 2 TO PROC Pr15 Pr25 Pr35 Pr45 FROM Pb
```

Duration of *one* communication frame (CF)

One processor in each physical node

All transactions have the same periodicity (100 ms)

“Abstract” processes “PrXX” introduced to facilitate “several to one” message passing

“BARRIER” and “WAITBARRIER”’s for the message passing

Definition of messages

We introduce the “abstract” processes “PrXX” so that the analysis will account for message delays.

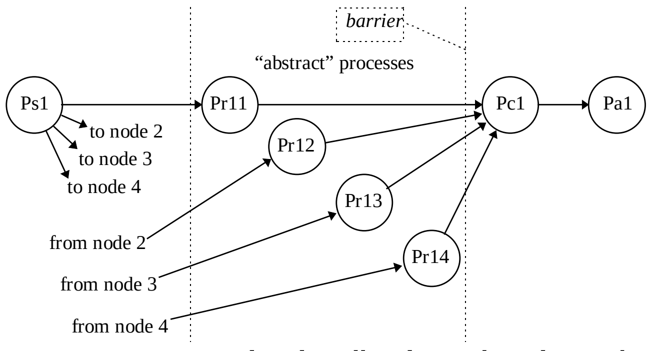

Figure 12: “Abstract” processes are introduced in all nodes. Only node 1 is shown here (for clarity)

After scheduling, analysis and simulation we obtain the following output:

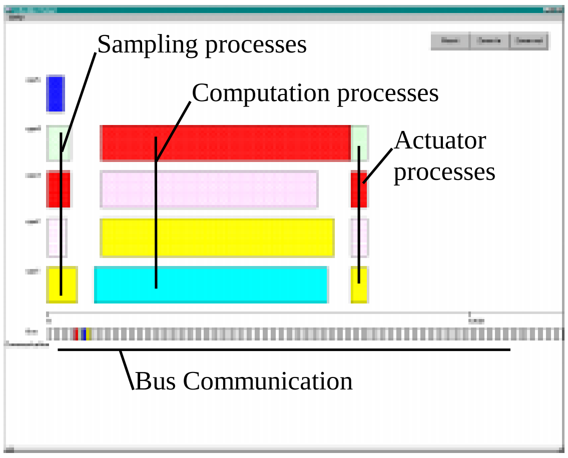

Figure 13: Output received from the scheduler

Bus allocation is illustrated by coloring the CF's with the same color (pattern) as the processes to which the slots are allocated

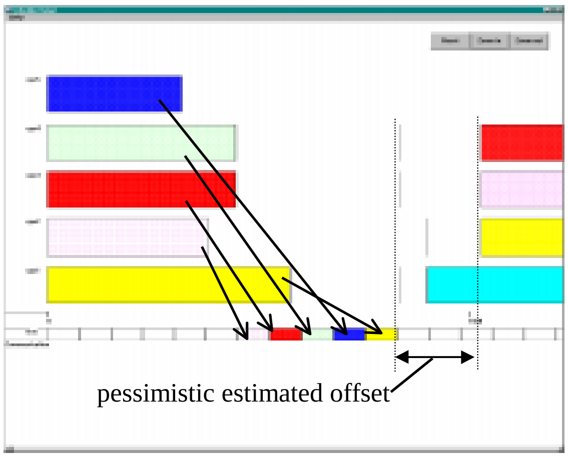

Figure 14: Bus allocation

From Figure 14 we may observe that three of the computation processes have inherited a too pessimistic offset for release. This is due to the Worst Case Assumptions concerning the bus-allocation that is done during the analysis. Since Bus Allocation is not known (at that time) during the analysis we have to assume (at least) *one Communication Cycle* as the response time for the message. This however, is not generally a good estimation as implied by Figure 14.

Summary, Conclusions and Further Work

In this report we have given an overview of the communication system architecture in Dacapo. We illustrated schedulability analysis with focus on message passing and synchronisation of processes in different nodes.

The results presented in this report was obtained with an early version of the scheduler/analyzer. The tool may be improved upon in several ways; for example, it is time consuming and error pronous to prepare scheduler input manually. The task specification should be designed to allow straight forward (automatic) decomposition and allocation. Priority assignments should be derived more or less directly from the specification. Final periods should be checked against the specification. The final schedule may be improved upon by introducing multi-pass scheduling/analysis. When all sender-processes has been scheduled and the bus-allocation is performed (message arrival time is determined) we should move processes, reallocate communication slots and invoke another pass. This would significantly reduce the currently pessimistic worst case latency for a message.

We should also construct a significantly larger and more realistic task specification in order to further test and improve the basic ideas behind the Dacapo scheduler.

References

[Bri93] Bridal, O. et alt, *"Dacapo: A dependable distributed computer architecture for control of applications with periodic operation"*, Tech. Report 165, Department of Computer Engineering, Chalmers University of Technology, Göteborg, 1993.

[Base95] Hansson,H. ,Lawson,H.W.,CUT, Strömberg, M., and Larsson, S, Mecel AB, *"BASEMENT®: A Distributed Real-Time Architecture for Vehicle Applications"*, Proc. of the IEEE Real-Time Technology and Applications Symposium (RTAS'95), IEEE Computer Society Press, Chicago, May, 1995.

[Sne95] Snedsböl, R. and Lönn, H. *"Timely fault tolerance in responsive systems for distributed control"*, Department of Computer Engineering, Chalmers University of Technology, Göteborg, 1995

[Rts99]
Thomas Lundqvist and Per Stenström: "*An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution*" Real-Time Systems, 17 (2/3):183-207, November 1999.

[Tör95] Törngren, M. *"Modelling and design of distributed real-time control applications"*, PhD thesis, DAMEK Research group, Department of Machine design, Royal Institute of Technology, KTH, Stockholm, 1995.

[Tin93] Tindell, K.,*"Fixed priority scheduling of hard real-time systems"*, PhD thesis, Department of Computer Science,University of York, 1993.