Chapter 8
Schedulability Analysis and Bus Access Optimization for Multi-Cluster Systems

THIS CHAPTER PRESENTS an approach to schedulability analysis and bus access optimization for multi-cluster distributed embedded systems consisting of time-triggered and event-triggered clusters, interconnected via gateways, as introduced in Section 3.5.

On the time-triggered clusters (TTC) the processes are scheduled based on a non-preemptive static cyclic scheduling policy, and messages are sent using the TTP, while on the event-triggered clusters (ETC) we use a fixed-priority preemptive scheduling policy for processes, and messages are sent via the CAN bus.

We have proposed a schedulability analysis for multi-cluster systems, including a buffer size and worst case queuing delay analysis for the gateways, responsible for routing inter-cluster traffic. Optimization heuristics for the priority assignment and synthesis of bus access parameters aimed at producing a schedulable system with minimal buffer needs have also been developed.

This chapter is organized in five sections. The next section introduces the problems that we are addressing in this chapter. Section 8.2 presents our proposed schedulability analysis for multi-cluster systems, and Section 8.3 uses this analysis to drive the optimization heuristics used for system synthesis. The last section present the experimental results.

8.1 Problem Formulation

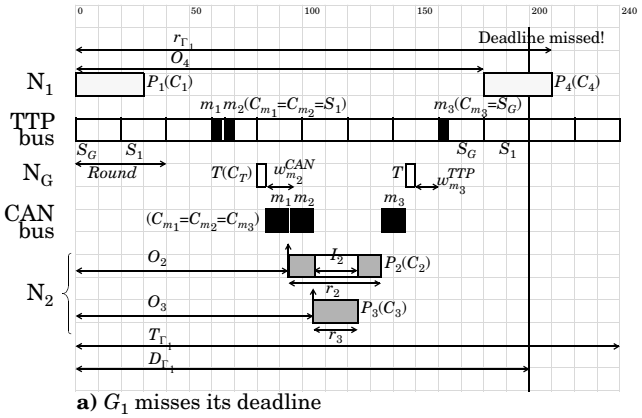
As input to our problem we have an application Γ given as a set of conditional process graphs mapped on an architecture consisting of a TTC and an ETC interconnected through a gateway node. The set of nodes on the TTC is denoted with $\mathcal{N}_T$, the ETC consists of the set of nodes $\mathcal{N}_E$, and the gateway node is denoted with N_G.

We are interested first to find a system configuration denoted by a 3-tuple $\psi = <\phi, \beta, \pi>$ such that the application Γ is schedulable. Determining a system configuration ψ means deciding on:

- The set ϕ of the offsets corresponding to each process and message in the system (see Section 6.2). The offsets of processes and messages on the TTC practically represent the local schedule tables and MEDLs.
- The TTC bus configuration β, indicating the sequence and size of the slots in a TDMA round on the TTC.
- The priorities of the processes and messages on the ETC, captured by π.

Once a configuration leading to a schedulable application is found, we are interested to find a system configuration that minimizes the total queue sizes needed to run a schedulable application. The approach presented in this chapter can be extended to cluster configurations where there are several ETCs and TTCs interconnected by gateways.

Example 8.1: Let us consider the example in Figure 8.1 where we the application G_1 mapped on the a two-cluster



a) G_1 misses its deadline

b) S_1 is the first slot, m_1, m_2 are sent sooner, G_1 meets its deadline

c) P_2 is the high priority process on N_2, G_1 meets its deadline

Figure 8.1: Scheduling Examples for Multi-Clusters

system as illustrated in Figure 3.8 on page 60. In the system configuration of Figure 8.1 we consider that, on the TTP bus, the gateway transmits in the first slot (S_G) of the TDMA round, while node N_1 transmits in the second slot (S_1). The priorities inside the ETC have been set such that $priority_{m_1} > priority_{m_2}$ and $priority_{P_3} > priority_{P_2}$.

In such a setting, G_1 will miss its deadline, which was set at 200 ms. However, changing the system configuration as in Figure 8.1b, so that slot S_1 of N_1 comes first, we are able to send m_1 and m_2 sooner, and thus reduce the response time and meet the deadline. The response times and resource usage do not, of course, depend only on the TDMA configuration. In Figure 8.1c, for example, we have modified the priorities of P_2 and P_3 so that P_2 is the higher priority process. In such a situation, P_2 is not interrupted when the delivery of message m_2 was supposed to activate P_3 and, thus, eliminating the interference, we are able to meet the deadline, even with the TTP bus configuration of Figure 8.1a.

■

8.2 Multi-Cluster Scheduling

In this section we propose an analysis for hard real-time applications mapped on multi-cluster systems. The aim of such an analysis is to find out if a system is schedulable, i.e., all the timing constraints are met. In addition to this, we are also interested to bound the queue sizes needed to run a schedulable applications.

On the TTC, an application is schedulable if it is possible to build a schedule table such that the timing requirements are satisfied. On the ETC, the answer whether or not a system is schedulable is given by a schedulability analysis, and we use the schedulability analysis outlined in Section 6.4.1.

In Section 6.4.1 the release jitter of a destination process D depends on the communication delay between sending and receiving an incoming message m: $J_{D(m)} = r_m$. However, in the case of a multi-cluster system, we will use offsets to capture the communication delays, and not the release jitter. Thus, the offset of a process will be determined such that it contains the communication delay due to the incoming message. For example, the offset O_2 of process P_2 in Figure 8.1a has been set such that it accounts for the delay due to message m_1 sent from the gateway transfer process T to the process P_2 via the CAN bus.

Moreover, determining the schedulability of an application mapped on a multi-cluster system cannot be addressed separately for each type of cluster, since the inter-cluster communication creates a circular dependency: the static schedules determined for the TTC influence through the offsets the response times of the processes on the ETC, which on their turn influence the schedule table construction on the TTC.

Example 8.2: In Figure 8.1a, placing m_1 and m_2 in the same slot leads to equal offsets for P_2 and P_3. Because of this, P_3 will interfere with P_2 (which would not be the case if m_2 sent to P_3 would be scheduled in *Round 4*) and thus the placement of P_4 in the schedule table has to be accordingly delayed to guarantee the arrival of m_3.

■

In our response time analysis we consider the influence between the two clusters by making the following observations:

- The start time of process P_i in a schedule table on the TTC is its offset O_i.
- The worst-case response time r_i of a TT process is its worst-case execution time, i.e. $r_i = C_i$ (TT processes are not preemptable).
- The worst-case response times of the messages exchanged between two clusters have to be calculated according to the schedulability analysis to be described in Section 8.2.1.

- The offsets have to be set by a scheduling algorithm such that the precedence relationships are preserved. This means that, if process P_j depends on process P_i, the following condition must hold: $O_j \geq O_i + r_i$. Note that for the processes on a TTC receiving messages from the ETC this translates to setting the start times of the processes such that a process is not activated before the worst-case arrival time of the message from the ETC. In general, offsets on the TTC are set such that all the necessary messages are present at the process invocation.

The MultiClusterScheduling algorithm in Figure 8.2 receives as input the application Γ, the mapping M, the system configuration ψ and produces the offsets ϕ and worst-case response times ρ.

The algorithm sets initially all the offsets to 0 (line 2). Then, the worst-case response times are calculated using the ResponseTimeAnalysis function (line 5) using the feasible analysis provided in [Tin94b]. The fixed-point iterations that calculate the response times at line 4 will converge if processor and bus loads are smaller than 100% [Tin94b]. Based on these worst-case response times, we determine new values ϕ^{new} for the offsets using a list scheduling algorithm (line 7).

The multi-cluster scheduling algorithm loops until the degree of schedulability δ_Γ of the application Γ cannot be further reduced (lines 9–22). In each loop iteration, we select a new offset O_i from the set of ϕ^{new} offsets (line 10), and run the response time analysis (line 12) to see if the degree of schedulability has improved (line 13). That offset O_i is selected, which corresponds to the unschedulable process P_i (i.e., its worst-case response time r_i is greater than its deadline D_i) with the largest difference $r_i - D_i$. If δ_Γ has not improved, we continue with the next offset in ϕ^{new}.

When a new offset O_i^{new} leads to an improved δ_Γ, we exit the for-each loop 10–21 that examines offsets from ϕ^{new}. The loop

```
MultiClusterScheduling(Γ, M, ψ)
1  -- determines the set of offsets φ and worst-case response times ρ
2  for each O_i ∈ φ do O_i = 0 end for -- initially all offsets are zero
3  -- determine initial values for the worst-case response times
4  -- according to the analysis in Section 8.2.1
5  ρ = ResponseTimeAnalysis(Γ, M, ψ, φ)
6  -- determine new values for the offsets, based on ρ
7  φ^new = ListScheduling(Γ, M, ψ, ρ)
8  δ_Γ = ∞-- consider the system unschedulable initially
9  repeat -- iteratively improve the degree of schedulability δ_Γ
10     for each O_i^new ∈ φ^new do -- for each newly calculated offset
11        O_i^old = φ.O_i; φ.O_i = φ^new.O_i^new -- set the new offset, remember old
12        ρ^new = ResponseTimeAnalysis(Γ, M, ψ, φ)
13        δ_Γ^new = SchedulabilityDegree(Γ, ρ)
14        if δ_Γ^new < δ_Γ then -- the schedulability has improved
15           -- offsets are recalculated using ρ^new
16           φ^new = ListScheduling(Γ, M, ψ, ρ^new)
17           break -- exit the for-each loop
18        else -- the schedulability has not improved
19           φ.O_i = O_i^old -- restore the old offset
20        end if
21     end for
22 until δ_Γ has not changed or a limit is reached
23 return ρ, φ, δ_Γ
end MultiClusterScheduling
```

Figure 8.2: The MultiClusterScheduling Algorithm

iteration 9–22 continues with a new set of offsets, determined by ListScheduling at line 16, based on the worst-case response times ρ^{new} corresponding to the previously accepted offset.

In the multi-cluster scheduling algorithm, the calculation of offsets is performed by the list scheduling algorithm presented in Figure 8.3. In each iteration, the algorithm visists the processes and messages in the ReadyList. A process or a message in the application is placed in the ReadyList if all its predecessors have been already scheduled. The list is ordered based on the priorities presented in Section 4.3.2. The algorithm terminats when all processes and messages have been visited.

In each loop iteration, the algorithm calculates the earliest time moment *offset* when the process or message $node_i$ in the application graph Γ can start (lines 5–7). There are four situations:

1. The visited node in the application graph is an ET message. In this case, the offset of message m_i is updated to *offset*.
2. The node is a TT message. In this case, the message is sched-

```
ListScheduling(Γ, M, ψ ρ) -- determines the set of offsets ϕ
1   ReadyList = source nodes of all process graphs in the application
2   while ReadyList ≠ ∅ do
3       node_i = Head(ReadyList)
4       offset = 0 -- determine the earliest time when an activity can start
5       for each direct predecessor node_j of node_i do
6           offset = max(offset, O_j + r_j)
7       end for
8       if node_i is a message m_i then
9           if m_i is an ET message then
10              O_i =offset -- update the message offset
11          else -- m_i is a TT message
12              <round, slot> = ScheduleMessage(offset, s_mi, M(S(m_i)))
13              -- set the TT message offset based on the round and slot
14              O_i = round * T_TDMA + O_slot
15          endif
16      endif
17      else  -- node_i is a process P_i
18          if M(P_i) ∈ N_E then -- process P_i is mapped on the ETC
19              O_i = offset -- the ETC process can start immediately
20          else -- process P_i is mapped on the TTC
21              -- P_i has to wait for processor M(P_i) to become available
22              O_i = max(offset, ProcessorAvailable(M(P_i)))
23          end if
24      end if
25      Update(ReadyList)
26  end while
27  return offsets ϕ
end ListScheduling
```

Figure 8.3: ListScheduling Algorithm

uled using the ScheduleMessage function from Section 4.3.1, which returns the *round* and the *slot* where the frame has been placed (line 12 in Figure 8.3). Once the message has been scheduled, we can determine its offset and worst-case response time (Figure 8.3, line 14). Thus, the offset is equal to the start of the slot in the TDMA round, and the worst-case response time is the slot length.

3. The algorithm visits a process P_i mapped on an ETC node. A process on the ETC can start as soon as its predecessors have finished and its inputs have arrived, hence O_i = *offset* (line 19). However, P_i might, later on, experience interference from higher priority processes.
4. Process P_i is mapped on a TTC node. In this case, besides waiting for the predecessors to finish executing, P_i will also have to wait for its processor $M(P_i)$ to become available (line 22). The earliest time when the processor is available is returned by the ProcessorAvailable function.

Let us now turn the attention back to the multi-cluster scheduling algorithm in Figure 8.2. The algorithm stops when the δ_Γ of the application Γ is no longer improved, or when a limit imposed on the number of iterations has been reached. Since in a loop iteration we do not accept a solution with a larger δ_Γ, the algorithm will terminate when in a loop iteration we are no longer able to improve δ_Γ by modifying the offsets.

8.2.1 SCHEDULABILITY AND RESOURCE ANALYSIS

The analysis in this section is used in the ResponseTimeAnalysis function in order to determine the response times for processes and messages on the ETC. It receives as input the application Γ, the offsets ϕ and the priorities π, and it produces the set ρ of worst case response times.

We have used the response time analysis outlined in Section 6.4.1 for the CAN bus (Equations 6.6, 6.9, 6.10, and 6.11). However, the worst-case queuing delay for a message

(Equation 6.9) is calculated differently depending on the type of message passing employed:

1. From an ETC node to another ETC node (in which case $W_m^{N_i}$ represents the worst-case time a message m has to spend in the Out_{N_i} queue on ETC node N_i). An example of such a message is m_3 in Figure 8.1, which is sent from the ETC node N_3 to the gateway node N_G.
2. From a TTC node to an ETC node (W_m^{CAN} is the worst-case time a message m has to spend in the Out_{CAN} queue). In Figure 8.1, message m_1 is sent from the TTC node N_1 to the ETC node N_2.
3. From an ETC node to a TTC node (where W_m^{TTP} captures the time m has to spend in the Out_{TTP} queue). Such a message passing happens in Figure 8.1, where message m_3 is sent from the ETC node N_3 to the TTC node N_1 through the gateway node N_G where it has to wait for a time W_m^{TTP} in the Out_{TTP} queue.

The messages sent from a TTC node to another TTC node are taken into account when determining the offsets (ListScheduling, Figure 8.2), and thus are not involved directly in the ETC analysis.

The next sections show how the worst-queuing delays and the bounds on the queue sizes are calculated for each of the previous three cases.

From ETC to ETC and from TTC to ETC

The analyses for $W_m^{N_i}$ and W_m^{CAN} are similar. Once m is the highest priority message in the Out_{CAN} queue, it will be sent by the gateway's CAN controller as a regular CAN message, therefore the same equation for W_m can be used:

$$W_m(q) = w_m(q) - qT_m \tag{8.1}$$

where q is the number of busy periods being examined, and $w_m(q)$ is the width of the level-m busy period starting at time qT_m:

$$w_m(q) = B_m + \sum_{\forall m_j \in hp(m)} \left\lceil \frac{w_m(q) + J_j}{T_j} \right\rceil C_j . \tag{8.2}$$

The intuition is that m has to wait, in the worst case, first for the largest lower priority message that is just being transmitted (B_m) as well as for the higher priority $m_j \in hp(m)$ messages that have to be transmitted ahead of m (the second term). In the worst case, the time it takes for the largest lower priority message $m_k \in lp(m)$ to be transmitted to its destination is:

$$B_m = \max_{\forall m_k \in lp(m)} (C_k) . \tag{8.3}$$

Note that in our case, $lp(m)$ and $hp(m)$ also include messages produced by the gateway node, transferred from the TTC to the ETC.

We are also interested to bound the size s_m^{CAN} of the Out_{CAN} and $s_m^{N_i}$ of the Out_{N_i} queue. In the worst case, message m, and all the messages with higher priority than m will be in the queue, awaiting transmission. Summing up their sizes, and finding out what is the most critical instant we get the worst-case queue size:

$$s_{Out} = \max_{\forall m} \left(s_m + \sum_{\forall m_j \in hp(m)} \left\lceil \frac{w_m(q) + J_j}{T_j} \right\rceil C_j \right) \tag{8.4}$$

where s_m and s_j are the sizes of message m and m_j, respectively.

From ETC to TTC

The time a message m has to spend in the Out_{TTP} queue in the worst case depends on the total size of messages queued ahead of m (Out_{TTP} is a FIFO queue), the size S_G of the gateway slot responsible for carrying the CAN messages on the TTP bus, and

the frequency T_{TDMA} with which this slot S_G is circulating on the bus, and thus, the width of the level-m busy period starting at time qT_m is:

$$w_m^{TTP}(q) = B_m + \left\lfloor \frac{(q+1)S_m + I_m(w_m(q))}{S_G} \right\rfloor T_{TDMA}, \quad (8.5)$$

where I_m is the total size of the messages queued ahead of m. Those messages $m_j \in hp(m)$ are ahead of m, which have been sent from the ETC to the TTC, and have higher priority than m:

$$I_f(w_m(q)) = \sum_{\forall m_j \in hp(f)} \left\lceil \frac{w_m(q) + J_j}{T_j} \right\rceil S_j \quad (8.6)$$

where the message jitter J_m is in the worst case the response time of the sender process, $J_m = r_{S(m)}$.

The blocking term B_m is the time interval in which m cannot be transmitted because the slot S_G of the TDMA round has not arrived yet. In the worst case (i.e., the message m has just missed the slot S_G), the frame has to wait an entire round T_{TDMA} for the slot S_G in the next TDMA round.

Determining the size of the queue needed to accommodate the worst case burst of messages sent from the CAN cluster is done by finding out the worst instant of the following sum:

$$s_{Out}^{TTP} = \max_{\forall m}(S_m + I_m). \quad (8.7)$$

8.3 Scheduling and Optimization Strategy

Once we have a technique to determine if a system is schedulable, we can concentrate on optimizing the total queue sizes. Our problem is to synthesize a system configuration ψ such that the application is schedulable, i.e., the condition[1]

$$r_{G_j} \leq D_{G_j}, \forall G_j \in \Gamma_i, \quad (8.8)$$

holds, and the total queue size s_{total} is minimized[1]:

$$s_{total} = s_{Out}^{CAN} + s_{Out}^{TTP} + \sum_{\forall N_i \in ETC} s_{Out}^{N_i}. \quad (8.9)$$

In the next section, we propose a resource optimization strategy based on a hill-climb heuristic that uses an intelligent set of initial solutions in order to efficiently explore the design space.

8.3.1 SCHEDULING AND BUFFER OPTIMIZATION HEURISTIC

The basic idea of our buffer optimization heuristic is to find, as a first step, a solution with the smallest possible response times, without considering the buffer sizes, in the hope of finding a schedulable system. This is achieved through the OptimizeSchedule function, outlined in Figure 8.4. Then, a *hill-climbing* heuristic [Ree93] iteratively performs moves intended to minimize the total buffer size while keeping the resulted system schedulable.

The OptimizeSchedule function is a greedy approach which determines an ordering of the slots and their lengths, as well as priorities of messages and processes in the ETC, such that the degree of schedulability δ_Γ (see Section 6.6.1) of the application is maximized.

As an initial TTC bus configuration β, OptimizeSchedule assigns in order nodes to the slots and fixes the slot length to the minimal allowed value, which is equal to the length of the largest message generated by a process assigned to N_i, $S_i = <N_i$,

1. The worst-case response time of a process graph G_i is calculated based on its sink node as $r_{G_i} = O_{sink} + r_{sink}$. If local deadlines are imposed, they will also have to be tested in the schedulability condition.
1. On the TTC, the synchronization between processes and the TDMA bus configuration is solved through the proper synthesis of schedule tables, thus no output queues are needed. Input buffers on both TTC and ETC nodes are local to processes. There is one buffer per input message and each buffer can store one message instance (see explanation to Figure 3.8 on page 60).

$size_{smallest}$> (line 5 in Figure 8.4). Then, the algorithm starts with the first slot (line 8) and tries to find the node which, when transmitting in this slot, will maximize the degree of schedulability δ_Γ (lines 9–37).

Simultaneously with searching for the right node to be assigned to the slot, the algorithm looks for the optimal slot length (lines 14–32). Once a node is selected for the first slot and a slot length fixed ($S_i = S_{best}$, line 36), the algorithm continues with the next slots, trying to assign nodes (and to fix slot lengths) from those nodes which have not yet been assigned.

When calculating the length of a certain slot we consider the feedback from the MultiClusterScheduling algorithm which recommends slot sizes to be tried out. Before starting the actual optimization process for the bus access scheme, a scheduling of the initial solution is performed which generates the recommended slot lengths. We refer the reader to Section 4.4.1 for details concerning the generation of the recommended slot lengths.

In the OptimizeSchedule function the degree of schedulability δ_Γ is calculated based on the response times produced by the MultiClusterScheduling algorithm (line 21). For the priorities used in the response time calculation we use the "heuristic optimized priority assignment" (HOPA) approach (line 16) from [Gut95], where priorities for processes and messages in a distributed real-time system are determined, using knowledge of the factors that influence the timing behavior, such that the degree of schedulability is improved.

The OptimizeSchedule function also records the best solutions in terms of δ_Γ and s_{total} in the seed_solutions list in order to be used as the starting point for the second step of our OptimizeResources heuristic.

In the first step of our buffer size optimization heuristic OptimizeResources, outlined in Figure 8.5, we have tried to obtain a bus configuration that improves the degree of schedulability of the application. Once a schedulable system is obtained, our goal in the second step is to minimize the buffer space. Our design

```
OptimizeSchedule(Γ, M)
1  -- given an application Γ produces the configuration ψ = <ϕ, β, π>
2  -- leading to the smallest δΓ
3
4  -- start by determining an initial TTC bus configuration β
5  for each slot Si ∈ β do Si = <Ni, sizesmallest> end for
6
7  -- find the best allocation of slots, the TDMA slot sequence
8  for each slot Si ∈ β do
9      for each node Nj ∈ TTC do
10         -- allocate Nj tentatively to Si, Ni gets slot Sj
11         Si = <Nj, sizeSj>
12         Sj = <Ni, sizeSi>
13         -- determine best size for slot Si
14         for each slot size ∈ recomended_lengths(Si) do
15             -- calculate the priorities according to HOPA heuristic
16             π = HOPA
17             -- determine the offsets ϕ,
18             -- thus obtaining a complete system configuration ψ
19             Si = <Nj, size>
20             ψcurrent = <ϕ, β, π>
21             ϕ = MultiClusterScheduling(Γ, M, ψcurrent)
22             -- remember the best configuration so far,
23             -- add it to the seed configurations
24             if δΓ(ψcurrent) is best so far then
25                 ψbest = ψcurrent
26                 Sbest = Si;
27                 add ψbest to seed_solutions
28             end if
29             determine stotal for ψcurrent
30             if stotal is best so far and Γ is schedulable
31             then add ψcurrent to seed_solutions end if
32         end for
33     end for
34     -- make binding permanent, use the Sbest corresponding to ψbest
35     if a Sbest exists
36     then Si = Sbest end if
37 end for
38
39 return ψbest, δΓ(ψbest), seed_solutions
end OptimizeSchedule
```

Figure 8.4: The OptimizeSchedule Algorithm

space exploration in the second step of OptimizeResources (lines 12–22) is based on successive design transformations (generating the neighbors of a solution) called *moves*. For our heuristics, we consider the following types of moves:

- moving a process or a message belonging to the TTC inside its [*ASAP, ALAP*] interval calculated based on the current values for the offsets and response times;
- swapping the priorities of two messages transmitted on the ETC, or of two processes mapped on the ETC;

```
OptimizeResources(Γ)
1
2   -- Step 1: try to find a schedulable system
3   seed_solutions = OptimizeSchedule(Γ, M)
4   -- if no schedulable configuration has been found,
5   -- modify mapping and/or architecture
6   if Γ is not schedulable for ψbest then
7       modify mapping
8       go to Step 1
9   end if
10
11
12  -- Step 2: try to reduce the resource need, minimize s_total
13  for each ψ in seed_solutions do
14      repeat
15          -- find moves with highest potential to minimize s_total
16          move_set = GenerateNeighbors(ψ)
17          -- select move which minimizes s_total
18          -- and does not result in an un-schedulable system
19          move = SelectMove(move_set)
20          Perform(move)
21      until s_total has not changed or limit reached
22  end for
23
24  return system configuration ψ, queue sizes
end OptimizeResources
```

Figure 8.5: The OptimizeResources Algorithm

- increasing or decreasing the size of a TDMA slot with a certain value;
- swapping two slots inside a TDMA round.

The second step of the OptimizeResources heuristic starts from the seed solutions (line 13) produced in the previous step, and iteratively preforms moves in order to reduce the total buffer size, s_{total} (Equation 8.9). The heuristic tries to improve on the total queue sizes, without producing un-schedulable systems. The neighbors of the current solution are generated in the GenerateNeighbours function (line 16), and the move with the smallest s_{total} is selected using the SelectMove function (line 19). Finally, the move is performed, and the loop reiterates. The iterative process ends when there is no improvement achieved on s_{total}, or a limit imposed on the number of iterations has been reached (line 21).

The general limitation of a hill-climbing heuristic is that it can get stuck into a local optimum. In order to improve the chances to find good values for s_{total}, the algorithm has to be executed several times, starting with a different initial solution. The intelligence of our OptimizeResources heuristic lies in the selection of the initial solutions, recorded in the seed_solutions list. The list is generated by the OptimizeSchedule function which records the best solutions in terms of δ_Γ and s_{total}.

Seeding the hill climbing heuristic with several solutions of small s_{total} will guarantee that the local optima are quickly found. However, during our experiments, we have observed that another good set of seed solutions are those that have high degree of schedulability δ_Γ. Starting from a highly schedulable system will permit more iterations until the system degrades to an un-schedulable configuration, thus the exploration of the design space is more efficient.

8.4 Experimental Evaluation

For evaluation of our algorithms we first used applications generated for experimental purpose. We considered two-cluster architectures consisting of 2, 4, 6, 8 and 10 nodes, half on the TTC and the other half on the ETC, interconnected by a gateway. Forty processes were assigned to each node, resulting in applications of 80, 160, 240, 320 and 400 processes. Message sizes were randomly chosen between 8 and 32 bytes. Thirty examples were generated for each application dimension, thus a total of 150 applications were used for experimental evaluation. Worst-case execution times and message lengths were assigned randomly using both uniform and exponential distribution. All experiments were run on a SUN Ultra 10.

In order to provide a basis for the evaluation of our heuristics we have developed two simulated annealing (SA) based algorithms (see Appendix A). Both are based on the moves presented in the previous section. The first one, named SA Schedule (SAS), was set to preform moves such that δ_Γ is minimized. The second one, SA Resources (SAR), uses s_{total} as the cost function to be minimized. Very long and expensive runs have been performed with each of the SA algorithms, and the best ever solution produced has been considered as close to the optimum value.

8.4.1 SCHEDULING AND BUS ACCESS OPTIMIZATION HEURISTICS

The first experimental result concerns the ability of our heuristics to produce schedulable solutions. We have compared the degree of schedulability δ_Γ obtained from our OptimizeSchedule (OS) heuristic (Figure 8.4) with the near-optimal values obtained by SAS. Figure 8.6 presents the average percentage deviation of the degree of schedulability produced by OS from the near-optimal values obtained with SAS. Together with OS, a straightforward approach (SF) is presented. For SF we considered a TTC bus configuration consisting of a straightforward ascending order of

allocation of the nodes to the TDMA slots; the slot lengths were selected to accommodate the largest message sent by the respective node, and the scheduling has been performed by the MultiClusterScheduling algorithm in Figure 8.2.

Figure 8.6 shows that when considering the optimization of the access to the communication channel, and of priorities, the degree of schedulability improves dramatically compared to the straightforward approach. The greedy heuristic OptimizeSchedule performs well for all the dimensions, having run-times which are more than two orders of magnitude smaller than with SAS. In the figure, only the examples where all the algorithms have obtained schedulable systems were presented. The SF approach failed to find a schedulable system in 26 out of the total 150 applications.

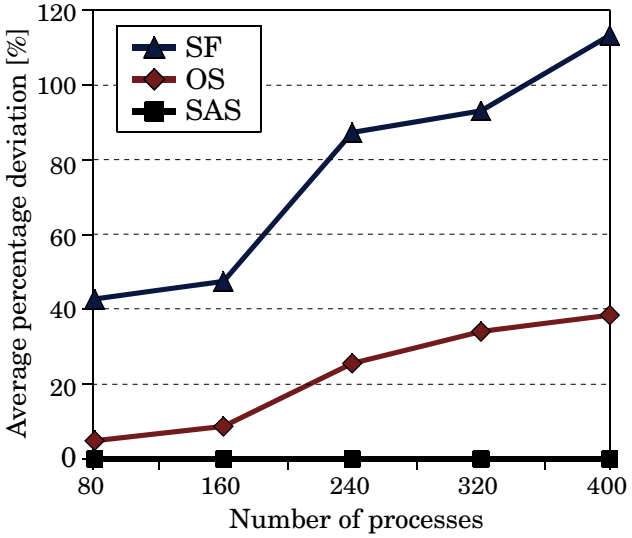


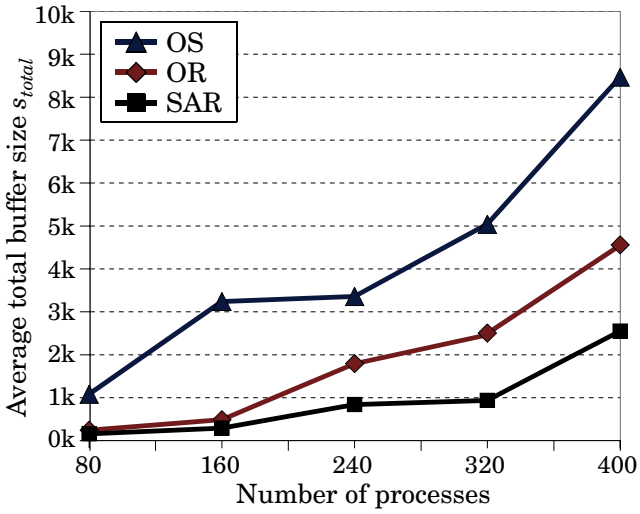
Figure 8.6: Comparison of the Scheduling Optimization Heuristics

8.4.2 BUFFER OPTIMIZATION HEURISTIC

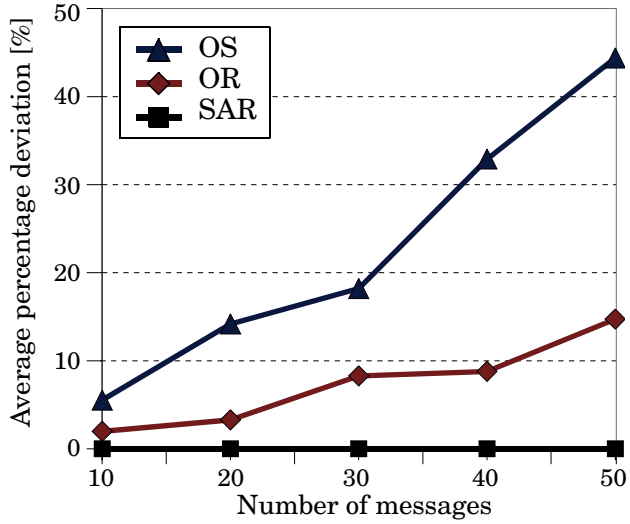
Next, we are interested to evaluate the heuristics for minimizing the buffer sizes needed to run a schedulable application. Thus, we compare the total buffer need s_{total} obtained by the OptimizeResources (OR) function with the near-optimal values obtained when using simulated annealing, this time with the cost function s_{total}. To find out how relevant the buffer optimization problem is, we have compared these results with the s_{total} obtained by the OS approach, which is interested only to obtain a schedulable system, without any other concern. As shown in Figure 8.7a, OR is able to find schedulable systems with a buffer need half of that needed by the solutions produced with OS. The quality of the solutions obtained by OR is also comparable with the one obtained with simulated annealing (SAR).

Another important aspect of our experiments was to determine the difficulty of resource minimization as the number of messages exchanged over the gateway increases. For this, we have generated applications of 160 processes with 10, 20, 30, 40, and 50 messages exchanged between the TTC and ETC clusters. Thirty applications were generated for each number of messages. Figure 8.7b shows the average percentage deviation of the buffer sizes obtained with OR and OS from the near-optimal results obtained by SAR. As the number of inter-cluster messages increases, the problem becomes more complex. The OS approach degrades very fast, in terms of buffer sizes, while OR is able to find good quality results even for intense inter-cluster traffic.

When deciding on which heuristic to use for design space exploration or system synthesis, an important issue is the execution time. In average, our optimization heuristics needed a couple of minutes to produce results, while the simulated annealing approaches (SAS and SAR) had an execution time of up to three hours.



a) Bounds on total buffer size obtained with OS, OR, SAS



b) Percentage deviations for OS, OR from SAR

Figure 8.7: Comparison of the Buffer Size Minimization Heuristics

8.4.3 THE VEHICLE CRUISE CONTROLLER

Finally, we considered a real-life example implementing a vehicle cruise controller introduced in Section 2.3.3:

- The conditional process graph that models the cruise controller has 32 processes, and is presented in Figure 2.9 on page 40,
- and it was mapped on an architecture consisting of a TTC and an ETC, each with 2 nodes, interconnected by a gateway, as in Figure 2.7b on page 37.
- The software architecture for multi-cluster systems, used by the CC, is presented in Section 3.5.
- We considered one mode of operation with a deadline of 250 ms.

The straightforward approach SF produced an end-to-end response time of 320 ms, greater than the deadline, while both the OS and SAS heuristics produced a schedulable system with a worst-case response time of 185 ms. The total buffer need of the solution determined by OS was 1020 bytes. After optimization with OR a still schedulable solution with a buffer need reduced by 24% has been generated, which is only 6% worse than the solution produced with SAR.

As a conclusion, the optimization heuristics proposed are able to increase the schedulability of the applications and reduce the buffer size needed to run a schedulable application.

In this chapter, the main contribution was the development of a schedulability analysis for multi-cluster systems. However, in the case of both TTP and CAN protocols, several messages share one frame, in the hope to utilize resources more efficiently. Therefore, in the next chapter we propose optimization heuristics for determining frame packing configurations that are able to reduce the cost of the resources needed to run a schedulable application.