5

Search-Based Heuristics for Modal Application

Due to the growing number of electronic control units (ECUs) in contemporary cars, sometimes reaching even 100, the automotive industry gradually resigns from their paradigm of using a separate unit for each functionality [99]. The requirement of placing a number of ever more sophisticated functionalities in one chip resulted in appearance of multi-core ECUs [158]. The AUTOSAR (AUTomotive Open System ARchitecture) standard [1] assumes a static (i.e., compile-time) mapping of atomic software components, named *runnables*, into cores since it is less complex and more predictable than dynamic resource allocation [94].

Due to the hard real-time constraint in automotive systems, the cores have to execute all the tasks on time even for their worst-case execution behavior, where they take worst-case execution time (WCET), which is usually much higher than the average execution time [152]. One possibility of decreasing the difference between the worst and average task execution times stems from the modal nature of such applications, i.e., from the fact that they can behave in a limited, known at design-time, number of ways, named *modes*. If each mode is analysed independently, the average execution time may be closer to the WCET determined for that mode [118]. In [104], six modes have been identified in a 4 cylinder gasoline torque based system, for example Cranking, Idle and Wide Open Throttle. It has been stressed there that execution times of particular runnables differ significantly for various modes of an ECU and thus applying different mappings for each operating mode may be beneficial. This way a lower number of cores could be needed than that of the corresponding system design not considering operating modes. However, introducing different mapping for modes imposes significant design complications, which have not been analysed in [104].

The contexts of runnables that are executed on different cores in different modes have to be migrated from one core to another, setting additional requirements for the available communication bandwidth. The process of mode switching usually incurs overhead (both in execution time and energy),

which is to be taken into account at run-time to decide whether to switch to a different mode or not. In hard real-time systems, it is essential to satisfy all the timing constraints even during the mode switching process, i.e., the migration time of tasks must be time bounded [146]. Therefore, the worst case switching time has to be assumed to provide the timing guarantees.

During the migration process, the taskset schedulability must not be violated. To guarantee this property, we propose to treat a migration process as any other asynchronous process in schedulability analysis, i.e., to use so-called *periodic servers*, which are periodic tasks executing aperiodic jobs. When a periodic server is executed, it processes pending task migration. If there is no pending migration, the server simply holds its capacity. To reduce the migration time, a recursive greedy algorithm for reducing the amount of data transferred during a mode change is proposed. It aims to decrease the number of periodic server instances used during a single mode switching. The proposed approach can be applied to any hard real-time systems, where different operating modes can be identified, and automotive systems in particular.

As an example, throughout this chapter we will analyse an engine ECU code named *DemoCar*. We will identify its operating modes and apply clustering to decrease their number and to eliminate task migration between neighbouring mode pairs (i.e., two modes from which at least one mode can be directly transferred to the second one) if the mode change is to be finished rapidly. The mappings for each mode will be determined using a genetic algorithm. This algorithm applies two optimizing criteria: runnable schedulability in terms of a number of deadline violations and migration cost in terms of the context length of the transmitted runnables. The typical schedulability analysis is used to determine the necessary network bandwidth to guarantee that the mode switching migration finishes in the required time.

In the next section, the state-of-the-art solutions are described followed by the proposed approach and a discussion on providing performance guarantees during mode changes.

5.1 System Model and Problem Formulation

5.1.1 Application Model

In this work we assume application model is consistent with the AUTOSAR standard [1]. A taskset Γ is comprised of an arbitrary number of periodic runnables, $\Gamma = \{\tau_1, \tau_2, \tau_3, \ldots\}$, grouped in tasks with hard real-time constraints. The j-th occurrence (j-th job) of runnable τ_i is denoted with $\tau_{i,j}$. The taskset is known in advance, including the WCET of each runnable, C_i, its period T_i, priority P_i and its relative deadline D_i equal to this period. Runnables are atomic schedulable units communicating each other with so called *labels*,

$N = \{\nu_1, \ldots, \nu_r\}$, which are memory locations of a particular length. The order of read and write operations to labels denotes the runnable dependencies, as the write operation to a particular label should be completed before its reading. Deadlines for mode changing time between each neighbouring pair of modes are also provided. We assume that the labels are stored in the same node that the runnable that reads these labels. If more than one runnable mapped to different cores read from the same label, its content is to be replicated to all the reading nodes and the writer should update the label value at all the locations. It means that the writer is aware of all its readers and knows their locations in all the possible modes.

Example 1 *Throughout this chapter, we consider a lightweight engine control system named DemoCar as an example application. The flow graph of this application is depicted in Figure 5.1. It consists of 18 runnables and 61 labels. All runnables are periodic: 8 runnables (highlighted in green) are to be executed every 10 ms, whereas period of 6 runnables (red, blue and yellow) equals 5 ms, two (violet) runnables are executed every 20 ms and the period of two (orange) runnables is 100 ms. In Figure 5.2 (upper part), 11 identified modes of this application are presented. These modes have been identified by inspecting the code of the runnable named* OperatingModeSWC*, which computes values of transaction and output functions of the Finite State Machine steering this engine. For example, label* FuelEnabled *is read by two runnables:* TransFuelMassSWC *and* ThrottleChangeSWC*. If these runnables are mapped to different cores, the label is to be replicated and kept in both the cores where these runnables were mapped to. It is a role of the writer,* OperatingModeSWC*, to update these values coherently not violating any timing constraints.*

5.1.2 Platform Model

The hardware platform assumed in this chapter is a mesh Network on Chip (NoC) with a certain number of cores $\pi \in \Pi$ and routers $\psi \in \Psi$, as shown in Figure 5.3. Each link is modelled as a single resource, so, for example, to transfer a portion of data from $\pi_{0,1}$ to appropriate sink $\pi_{2,0}$ we need such resources allocated simultaneously: $\pi_{0,1} - \psi_{0,1}$, $\psi_{0,1} - \psi_{1,1}$, $\psi_{1,1} - \psi_{2,1}$, $\psi_{2,1} - \psi_{2,0}$, $\psi_{2,0} - \pi_{2,0}$. In every mode, each runnable is mapped to one core and a label is stored in local memories of the cores requesting that label. Data transfer overhead is taken into consideration, assuming constant time for transferring a single flit (Flow control digIT, a piece of a network package whose length usually equals the data width of a single link) between two neighbouring cores if no contentions are present. Timing constants for packet

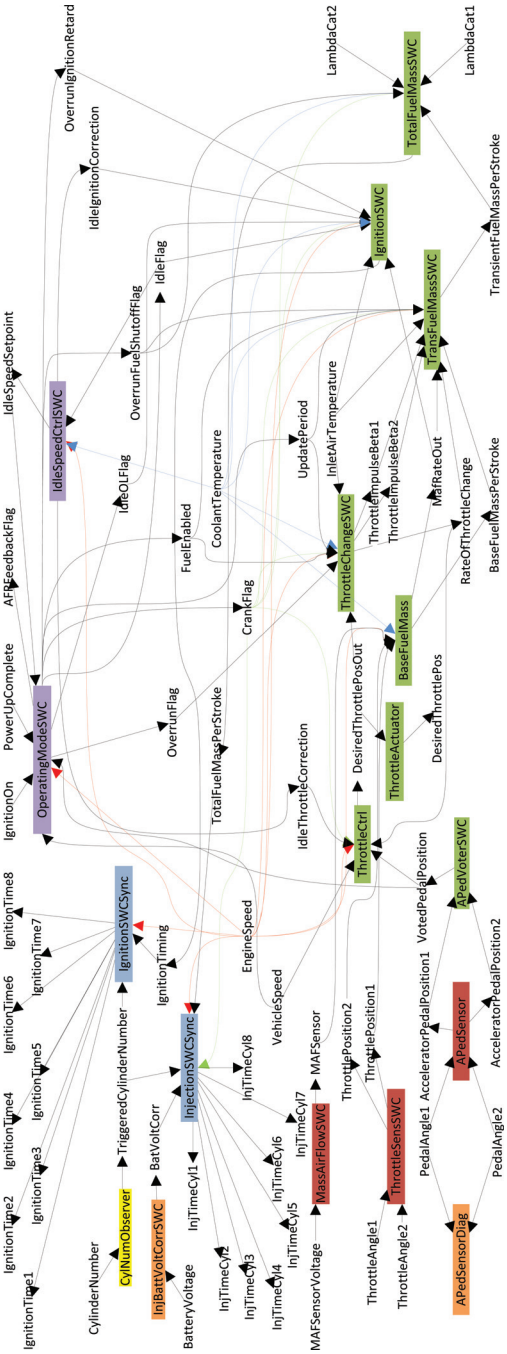

Figure 5.1 Flow graph of the DemoCar example; the runnables belonging to the same task are highlighted with the same colour, labels are not highlighted. Some flows are drawn in different colours for readability.

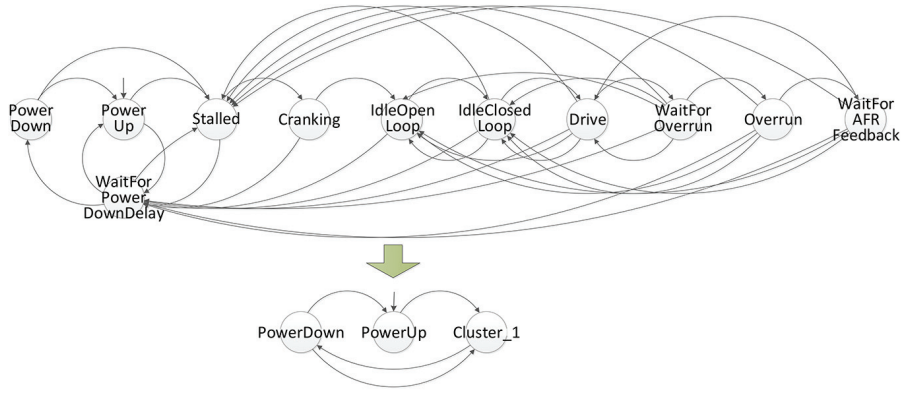

Figure 5.2 Finite State Machine describing mode changes in DemoCar use case: before (*upper part*) and after (*lower part*) the clustering step.

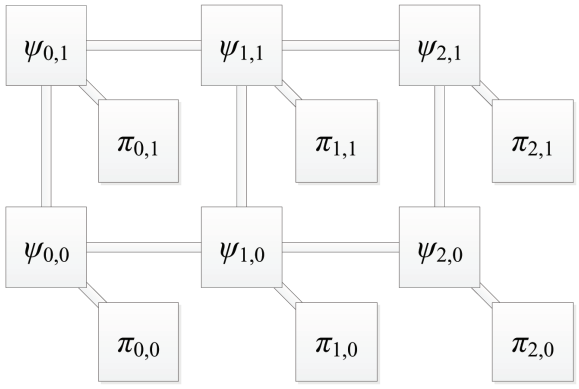

Figure 5.3 An example many-core system platform.

latencies while traversing one router and one link are denoted as d_R and d_L, respectively. The priority of data transfer packets are assumed to be equal to the priority of the runnable sending them.

5.1.3 Problem Formulation

Given a platform and an application model with a defined set of operating modes, the problem is to determine schedulable mappings for each mode so that the amount of data to be migrated during the allowed mode changes is minimized. During mode changing, the taskset should be still schedulable despite the additional network traffic generated by the task migrations. The neighbouring modes (i.e., the modes connected with a link in the FSM

describing the allowed mode transitions) with similar runnables' execution time can be clustered to decrease the frequency of task migrations. The deadlines for mode changing time between each neighbouring pair of modes must not be violated.

5.2 Proposed Approach

In this section, steps of the proposed design flow are described. Since it has been assumed that the tasksets of the considered application are known in advance, it is possible to perform some preliminary computations statically. Consequently, the mapping problem can be split into two stages: off-line (static) and on-line (dynamic), as shown in Figure 5.4. The computation time of the off-line part is not crucial and thus heuristics with even high complexity may be used for runnable and label mappings. It seems promising to combine the most effective approaches, such as multi-objective simulating annealing or genetic algorithms. The possibility of extending genetic operators benefiting from the full knowledge of the system domain, such as mutation in a way similar to [109], makes the genetic approach the first choice at this step.

During run time, detection of the current mode is assumed to be done by observing certain variable. (In DemoCar such variable is named *–sm* and is stored in runnable *OperatingModeSWC*.) When a value of this variable has been changed, the current runnable and label mapping might have to be switched. The mappings have been chosen during the design time with respect to minimize the amount of data to be migrated. Schedulability analysis guarantees that even the worst case switching time does not violate the deadline required for mode changes. If such violation is unavoidable, either the states can be clustered, or the network bandwidth is to be increased.

The off-line part of the proposed approach is comprised of five steps, which are covered in the following subsections.

5.2.1 Mode Detection/Clustering

The reasons for introducing the *mode detection & clustering* step are twofold. Firstly, some neighbouring modes can be characterized with similar runtime

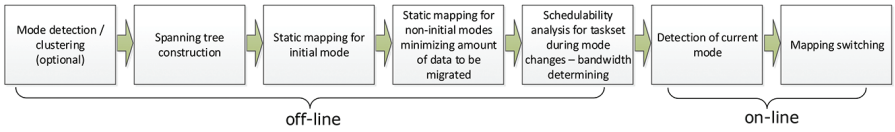

Figure 5.4 Steps of dynamic resource allocation method benefiting from modal nature of applications.

and resource consumption. Then there is little benefit in preparing different mappings for such modes and migrating the runnables when a transition between these neighbouring modes is made. Moreover, some transitions are required to be done immediately, whereas others can be less time tight. If a runnable migration is to be performed quickly, for example between two consecutive runnable occurrences, the bandwidth needed to transfer the appropriate amount of data in that time may be unreasonably high. Therefore, it may be more sensible to merge two modes with such rapid task switching time and generate only one mapping for them.

Example 2 *(continuation of Example 1) In DemoCar, transitions between modes:* Stalled*,* Cranking*,* IdleOpenLoop*,* IdleClosedLoop*,* Drive*,* WaitForOverrun*,* Overrun*,* WaitForAFRFeedback *and* WaitForPowerDownDelay *are to be performed between two consecutive executions of their runnable occurrences, which is upperbounded with 5 ms for 9 runnables. Since performing task migration during such short time window would require a bandwidth of considerable size, these modes have been clustered into* Cluster_1*. Finally, three modes can be identified after the clustering step:* PowerDown*,* PowerUp *and* Cluster_1*, as presented in Figure 5.2 (lower part).*

5.2.2 Spanning Tree Construction

To minimize the amount of data to be migrated between two consecutive modes with the technique proposed in this chapter, the FSM describing mode changes should include weights denoting state transition probabilities. Since probabilities of staying in the current mode are not relevant at this step, they can be omitted for simplicity. The probabilities can be given or determined during long simulation of the modal system. The FSM has also to have all its cycles removed to guarantee halting of the *Static mapping for non-initial modes* step. In this regard, for an FSM treated as a weighted connected graph $G(V, E)$, where V is the set of vertices and E denotes the set of edges, a maximum spanning tree can be constructed. We recollect that a spanning tree of a graph G is its subgraph $T(V, E')$, which is connected and whose number of edges is equal to the number of vertices minus 1, $|E'| = |V|-1$. If $\mathcal{T}$ denotes the set of all spanning trees of G, a maximum spanning tree $T_{max}(V, E_{max})$ of G is a spanning tree iff:

$$\underset{T(V,E')\in\mathcal{T}}{\forall} \sum_{(v,z)\in E_{max}} w(v,z) \geq \sum_{(v,z)\in E'} w(v,z),$$

where $w(v, z)$ is the weight value assigned to the edge from a vertex v to z. A maximum spanning tree can be constructed in time $O(|E|\log|V|)$, e.g., by the classic Prim's algorithm [108].

The operation performed in this step neither influences the application behaviour nor limit the possible mode transitions. It only makes the least frequent transitions not optimized during stage *Static mapping for not initial modes minimizing amount of data to be migrated* (Figure 5.4).

Example 3 *(continuation of Example 2) For DemoCar, probabilities of mode changing have been shown in Figure 5.5 (left). The maximum spanning tree, constructed with the Prim's algorithm, is presented in Figure 5.5 (right).*

5.2.3 Static Mapping for Initial Mode

Since mapping for each mode is performed off-line, even heuristics known from their high computational cost, such as genetic algorithms, can be applied. A genetic algorithm used for hard real-time systems shall guarantee that under the chosen schedule all timing constraints are satisfied. This can be performed in several ways. For example, each missed deadline can impose a certain penalty to the fitness function value, and thus each schedule with unsatisfied constraints should be eliminated during the evolutionary process. A particular mapping is portrayed as a chromosome, stored as a bit string, representing on each gene the processing core where the task would be mapped to, similarly to [61]. The bit string one-point crossover operator and flip bit mutation have been applied together with the tournament selection of the individuals.

Below, an algorithm encompassing the aforementioned properties is described.

In Algorithm 5.1, it is presented a pseudo-code of a genetic algorithm that can be used during *Static mapping for initial mode* step, the third off-line step of the proposed approach, as depicted schematically in Figure 5.4. We propose to use two fitness functions – measuring (i) the number of deadline violations and (ii) makespan (also known as response time). Both these functions apply the interval algebra described in Chapter 2. The first fitness function value is of primary importance, as in a hard real-time system no deadline violation is allowed. But among fully schedulable mappings, the one leading to a

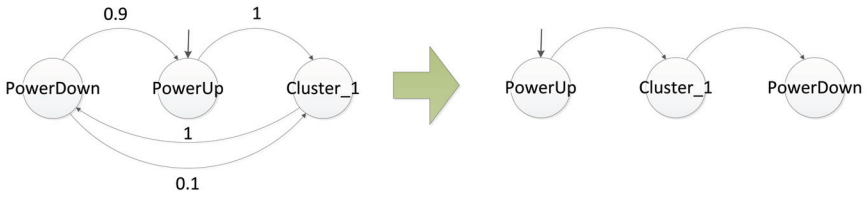

Figure 5.5 Spanning tree construction for DemoCar.

Algorithm 5.1 Pseudo-code of no deadline violation with makespan minimisation algorithm for the initial mode mapping

inputs : Workload Γ;
Resource set Π;
outputs : Task mapping;

```
1  Choose an initial random population of task mappings
2  while not termination condition do
3      Evaluate the number of deadline violations using IA; //criterion (i)
4      Evaluate the makespan using IA; //criterion (ii)
5      Create clusters of individuals with the same number of deadline violations;
6      Sort the clusters by increasing number of deadline violations;
7      Sort individuals in each cluster wrt their makespan;
8      Perform tournament selection; //criterion (i) has higher priority than criterion
         (ii)
9      Generate individuals using crossover and mutation;
10     Create a new population with the best found mappings;
11 end
```

lower makespan is chosen, since idle intervals can be used to decrease energy consumption or execute tasks of lower criticality levels.

In the algorithm, the following steps can be singled out.

Step 1. Initial population initialisation (line 1). An arbitrary number of random task mappings (individuals) is created.

Step 2. Creating a new population (lines 3–10). For each individual, values of two fitness functions are computed - the number of deadline violations and the makespan (lines 3–4). Individuals with the same number of deadline misses are grouped together (line 5). The groups are then sorted with respect to the number of deadline violations in the ascending order (line 6). Inside each group, individuals are sorted according to their growing makespan (line 7). The tournament selection is then performed – individuals from a group with lower number of deadline violations are always preferred, whereas among individuals from one group the one with the lowest makespan is to be chosen (line 8). The individuals winning the tournament are then combined using a typical crossover operation and mutated (line 9). A new population is created (line 10). Step 2 is repeated in a loop as long as a termination condition is not fulfilled, which can be a maximal number of generated populations or lack of improvement in a number of subsequent generations.

Example 4 *(continuation of Example 3) For the* PowerUp *(initial) mode of DemoCar to be executed on a multi-core embedded system, we evaluate makespan and number of violated deadlines during one hyperperiod (i.e.,*

the least common multiple of all runnables' periods) by allocating runnables and labels to different cores.

The size of the NoC mesh has been initially configured as 2x2 with no idle cores, since this size has been earlier checked (also using Algorithm 5.1) and is large enough to execute DemoCar in the most computational intensive mode, Cluster_1*, not violating any of its timing constraints. The genetic algorithm is executed again to perform assignment of tasks to cores with timing characteristics for the initial* PowerUp *mode. The genetic algorithm has been configured to generate 100 generations of 20 individuals each. The first fully schedulable allocation has been found in the 1st generation, which suggests that it might be possible to allocate the taskset to a lower number of cores.*

After performing further search it has appeared that the taskset in the initial mode is schedulable even when mapped to one (out of four) active core. The lowest makespan for the NoC with three idle cores is equal to 8622 µ*s.*

5.2.4 Static Mapping for Non-Initial Modes

It is of primary importance to migrate as little data as possible during mode changes to minimise the migration time.

Each application A includes a set of tasks and can be represented with a vector comprised of p runnables and r shared memory locations (labels) of these tasks, $\mathrm{A} = [\tau_1, \ldots, \tau_p, \nu_1, \ldots, \nu_r]$, and platform Π is composed of s processing cores, $\Pi = \{\pi_1, \ldots, \pi_s\}$. A mapping M is a vector of p core locations, $\mathrm{M} = [\pi_{\tau_1}, \ldots, \pi_{\tau_p}]$, where each element corresponds with the appropriate element of A and can be substituted with any element of set Π. Each element of weight vector W, $\mathrm{W} = [w_{\tau_1}, \ldots, w_{\tau_p}]$, is equal to the amount of data that has to be transferred when a particular runnable is migrated, including the labels to be read.

Let M_α and M_β be sets of mappings that are fully schedulable in a given system in state α and β, respectively. The elements of the difference vector $D_{m_\alpha, m_\beta} = [d_{\tau_1}, \ldots, d_{\tau_p}]$ indicate which runnables are to be migrated when the mode is changed from α to β. Each element d_δ, $\delta \in \{\tau_1, \ldots, \tau_p\}$, takes value 1 if the particular runnable/label is allocated to different cores in mappings $m_\alpha \in M_\alpha$ and $m_\beta \in M_\beta$, and 0 otherwise:

$$d_\delta = \begin{cases} 0, & \text{if } m_{\alpha,\delta} = m_{\beta,\delta}, \\ 1, & \text{otherwise} \end{cases} \tag{5.1}$$

where $m_{\alpha,\delta}$ and $m_{\beta,\delta}$ denote the δ-th element of vectors m_α and m_β, respectively. The migration cost c between two states α and β is then computed in the following way:

$$c_{m_\alpha,m_\beta} = \mathrm{D}_{m_\alpha,m_\beta} \cdot \mathrm{W}^{\mathrm{T}}. \tag{5.2}$$

A recursive greedy algorithm for reducing an amount of data transferred during mode changes is presented in Algorithm 5.2.

Since some cycles are likely to occur in a graph representing the Finite State Machine describing transitions between modes, a spanning tree (ST) is to be built, as described in the previous subsection. Then the mode corresponding to the initial state of the FSM is selected as the current mode (line 1). For this mode, a set of schedulable mappings is generated, e.g., with Algorithm 5.2 (line 2). If more than one schedulable mapping is found, an additional criterion $crit$ (e.g., minimum makespan value) is used to select one of them (line 3). Then for each direct successor of the ST node corresponding to FSM initial state, the *FindMappingMin* procedure is executed (lines 4 and 5).

In the *FindMappingMin* procedure, a set of schedulable mappings for that successor node is found using minimal migration cost criterion (5.2) (line 8). If more than one schedulable mapping is equally evaluated by this criterion, an additional criterion, $crit$, is used (line 9). The *FindMappingMin* procedure is then recursively run for each direct successor of the ST node provided as the function parameter (lines 10 and 11). More mappings could be delivered to the *FindMappingMin* procedure to browse a larger search space by skipping lines 4 and 9 in the algorithm and providing all elements of M_α instead of just one.

Algorithm 5.2 Pseudo-code of a migration data transfer minimisation algorithm

inputs : A spanning tree ST based on Finite State Machine (FSM) describing the system modes with transaction probabilities;
W - size of each runnable memory footprint;
$crit$ - mapping optimality criterion (e.g., minimum makespan value);
outputs : Runnable and label mapping for each mode;

```
1  Select the initial state of ST and assign it to α;
2  Find a set of schedulable mappings M_α;
3  Select m_α ∈ M_α wrt criterion crit;
4  forall β being a direct successor of α in ST do
5  |  FindMappingMin(α, β, m_α);
6  end
7  FindMappingMin(α, β, m_α)
8  Find a set of schedulable mappings M_β minimizing criterion (5.2) using W
9  Select m_β ∈ M_β wrt criterion crit
10 forall q being a direct successor of β in ST do
11 |  FindMappingMin(β, q, m_β)
12 end
```

Example 5 *(continuation of Example 4) Regardless of the mode, the application has been mapped in a 2 × 2 mesh Network on Chip without deadline violations. For the* PowerUp *mode, schedulable mappings have been found even if three of the four NoC cores remains idle, as shown in Example 4. It means that in this mode three cores can be switched off, leading to considerable energy savings. Similarly, two cores can remain idle in the* PowerDown *mode.* (PowerDown *requires more computations than* PowerUp *since some maintenance procedures are to be consistently performed.*) *However, despite intensive search using a genetic algorithm, all four cores are needed in the* Cluster_1 *mode to have the taskset fully schedulable. Thus, when the current mode changes from* PowerUp *to* Cluster_1*, three cores have to be activated, whereas two cores can be switched off after leaving the* Cluster_1 *mode.*

Let us focus on the transition between the PowerUp *and* Cluster_1 *modes. For* PowerUp*, only one core is active and thus all runnables are to be mapped to the only active core. However, in other cases a larger set of mappings that are fully schedulable on active NoC cores would have been identified. From these mappings, the one with the lowest makespan (an additional criterion) is chosen. This mapping has been used as a parameter of the* FindMapping-Min *procedure (from the algorithm presented in Algorithm 5.2). The set of schedulable mappings following the minimum criterion (Equation (5.2)) is identified using a genetic algorithm. By the applied criterion ($min(c_{m_\alpha,m_\beta})$), a significantly lower amount of data has to be migrated during the mode change. In the best found case, 3 runnables have to be migrated, whose total $c_{PowerUp,Cluster\_1} = 261968$ bytes. However, by not using this criterion, but the minimal makespan instead, the lowest number of runnables to be migrated equals 13, which results in $c_{PowerUp_2,Cluster\_1} = 890162$ bytes. In the second case, the amount of data to be transferred using a periodic server is about 240% higher than in the first mapping pair. Since periodic servers offer equal throughput during the system execution, the mode change between the latter mappings would last more than three times as long as between the former pair.*

During mode change from Cluster_1 *to* PowerDown*, 2 runnables have to be migrated and $c_{Cluster\_1,PowerDown} = 113568$ bytes. Although the transition between modes* PowerDown *and* PowerUp *are not optimized, in this case only 2 runnables have to be migrated with $c_{PowerDown,PowerUp} = 113536$ bytes.*

5.2.5 Schedulability Analysis for Taskset During Mode Changes

Since some runnables and labels are expected to be located at different cores in two different modes, their migration is to be performed during mode changes. A runnable migration process is schematically depicted in Figure 5.6. Two mappings m_α and m_β of runnables τ_i, τ_j, τ_k into nodes π_a and π_b are

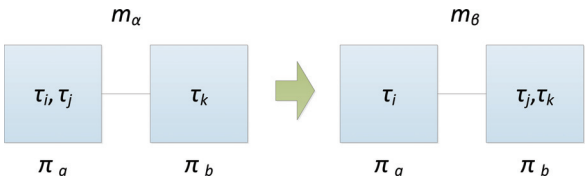

Figure 5.6 Example of two different mappings (m_α, m_β) of runnables τ_i, τ_j, τ_k into cores π_a and π_b.

used in two different system modes: α and β, respectively. The difference between these mappings is the assignment of runnable τ_j. We assume no deadline violations for both mappings m_α and m_β. During hyperperiods involved in the migration process between α and β, the schedulability analysis for communication resources should take into consideration not only all the transfers between π_a and π_b described in the workload, but also an additional periodic job, i.e., the periodic server of certain policy (polling, sporadic, deferrable, etc.) with a certain execution time in each period. A technique for determining this time is presented in this subsection.

When the mode changes from α to β, runnable τ_j is to be copied from π_a to π_b. Since the precopy strategy is applied, τ_j is still executed on core π_a during the migration. To migrate runnable τ_j, the periodic server is used. The whole context of the runnable is transferred during a number of subsequent hyperperiods. It is worth stressing that the maximal migration time can be computed statically, since the runnable context size and the periodic server time slot length and period are known. After this time, it is safe to start executing τ_j on π_b and remove its copy in π_a.

To guarantee schedulability of runnables, one of the schedulability tests, described for example in [42], shall be applied. It is possible to calculate the longest possible time interval between the release of runnable τ_i and its termination, which is referred as τ_i's worst case response time (WCRT) and is represented by R_i. The schedulability analysis is performed in the way described in [9], i.e., by checking whether WCRTs of all runnables do not exceed their deadlines. WCRT of runnable τ_i can be computed using equation:

$$R_i = C_i + \sum_{\forall \tau_j \in hp(\tau_i)} \left\lceil \frac{R_i + R_j - C_j}{T_j} \right\rceil C_j, \tag{5.3}$$

where $hp(\tau_i)$ denotes the set of all runnables that can preempt τ_i, C_i and C_j are the worst case execution time of τ_i and τ_j, respectively, and T_j denotes the period of τ_j. Similarly, the worst case latency r_k of packet φ_k transmitted over a link in a mesh NoC with wormhole switching, issued periodically every t_k,

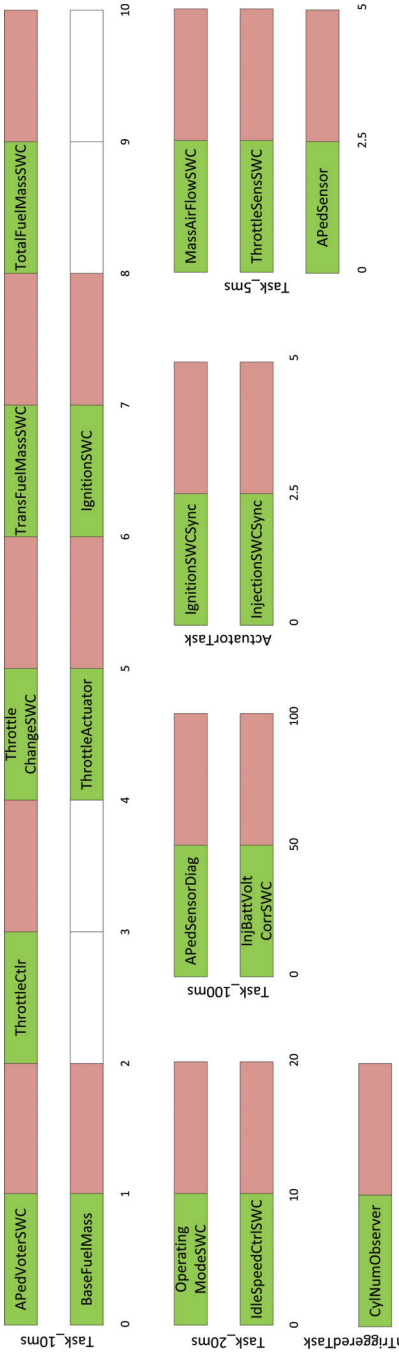

Figure 5.7 Tasks' stages in DemoCar: green – runnable execution, red – write to labels; release times and deadlines are provided in ms.

can be formulated in a similar manner as that of [61, 100, 122]:

$$r_k = c_k + b_k + l_k, \tag{5.4}$$

where c_k is a basic network latency, b_k is the maximal blocking time from lower-priority packages, and l_k is the maximal blocking time due to interference with higher-priority packets. The basic network latency can be computed with the following equation [61, 100]:

$$c_k = H \cdot (d_R + d_L) + \left\lceil \frac{PS}{FS} \right\rceil \cdot d_L, \tag{5.5}$$

where d_R and d_L denote the constant packet latencies while traversing one router and one link, respectively, PS is the number of bits in the package, and FS is the flit length in bits. H is the hop number between source and destination cores. The remaining terms of Equation (5.4) can be computed with equations [61, 100]:

$$b_k = H \cdot (d_R + d_L), \tag{5.6}$$

$$l_k = \sum_{\forall \varphi_l \in interf(\varphi_k)} \left\lceil \frac{r_k + (r_l - c_l) + R_i}{t_l} \right\rceil (c_l + b_l), \tag{5.7}$$

where $interf(\varphi_k)$ denotes the direct interference set of φ_k, which is the set of all packets that can preempt φ_k, i.e., have a higher priority and share at least one link with φ_k. The response time of task τ_i that releases φ_k, R_i, has been substituted as a maximum release jitter. The term $(r_l - c_l)$ is an upper bound of indirect interference [61].

By applying Equations (5.3) and (5.4), both schedulabilities of independent runnables executed on processing cores and packet transmissions can be verified. However, jobs in the considered applications, possibly executed on different cores, are characterised with various dependency patterns. Typically, to start a job execution it is required to have all its parent jobs executed (which contributes to so-called *computation latency*) and all the necessary data transferred to the core where this job is assigned to (*communication latency*).

The goal is thus to establish whether all task-chains of an application have their end-to-end deadlines met in a particular platform, and this assessment is referred as *end-to-end schedulability test*. Such test must consider the end-to-end latency of each task of a *task-chain*. To check schedulability of a task chain, it is sufficient and necessary to test the individual end-to-end response times of all tasks belonging to that chain [73]. In [73], a technique for end-to-end schedulability analysis is proposed, but it assumes a pipelined task

execution pattern, where multiple jobs of the same task chain are executed simultaneously over different cores, but the simultaneous execution of more than one job of the same task is not allowed. When the execution pattern does not follow this scheme, meeting end-to-end deadlines can be checked by assigning an appropriate local deadline for each job in every chain. These local deadlines shall be chosen in a way that all the jobs on every core are schedulable and the local deadlines at the chain last stage do not exceed the respective end-to-end deadline [59].

Example 6 *(continuation of Example 5) In DemoCar, each task is composed with series of three subsequent stages: read from labels, runnable execution, write to labels. Since the labels are always located in the same core as that of runnable reading these labels, the read stage can be omitted and two remaining stages are presented in Figure 5.7 for all tasks, highlighted in green and red. Runnables belonging to one task and drawn one above the other can be executed in parallel, whereas the execution order of the runnables follows dependencies defined by label write and read operations so that each label is to be written by a runnable prior to be read by another runnable. The end-to-end deadline has been divided into number of stages in each task and in that way deadlines for each stage have been determined* (*in Figure 5.7 these deadlines are written beneath the end point of each stage*).

For example, the release time of runnable ThrottleCtrl *is 2ms. By this time, all the packets with label values required by this runnable are assumed to arrive at the node executing* ThrottleCtrl. *The deadline for this runnable execution is 3 ms, so the WCRT (R_i) must not be higher than this value. The packages with data are then issued between 2 ms (the runnable release time) and R_i. They have to reach their destination nodes earlier than 4 ms.*

To check schedulability of DemoCar, it is then sufficient to check schedulability for runnables executed in all (green) stages and also data transfers to and from labels performed in the appropriate (red) stages.

Since the earliest execution starting time of each runnable is limited by the starting time of the stage including particular runnable, this stage starting time can be treated as an offset O_i as described in [143]:

$$R_i = C_i + \sum_{\forall \tau_j \in hp(\tau_i)} \left\lceil \frac{R_i + (R_j - C_j - O_j) - O_i}{T_j} \right\rceil C_j + O_i. \tag{5.8}$$

Using similar rationale, Equations (5.4) and (5.7) can be rewritten in the following way:

$$r_k = c_k + b_k + l_k + R_i, \tag{5.9}$$

$$l_k = \sum_{\forall \varphi_l \in interf(\varphi_k)} \left\lceil \frac{(r_k - O_i) + (r_l - c_l - O_{i'})}{t_l} \right\rceil (c_l + b_l), \qquad (5.10)$$

where term $(r_k - O_i)$ reflects an additional jitter imposed by the response time of task R_i that initiates this transfer and $O_{i'}$ denotes an offset of the task that releases φ_l. One more requirement has to be added to set $interf(\varphi_k)$. It includes not only packets having a higher priority and transferred via a path sharing at least one link, but also timing boundaries of both their sender executions or traffic stages have to overlap.

As mentioned earlier, the proposed task mapping technique aims to benefit from a modal nature of applications, but it also possess new challenges. If the modes are treated independently from each other, the end-to-end schedulability of runnables and packet transmission in each mode can be analysed using Equations (5.8) and (5.9). It is the instant of transition between these modes that requires special attention. The task migration time can be computed with Equations (5.5), (5.6), (5.9), (5.10), where the packet size, PS, is equal to the sum of the header length and the size of the payload including the whole context of runnables and labels to be migrated. If a relatively large runnable is to be migrated in a highly utilised platform, performing the migration when the next job of the runnable is due to start could require rather high bandwidth in order not to violate any deadlines. Thus we assume to use the precopy strategy, as described in [109]. The job is executed in its current (source) location during the mode switching, until all the runnables have been migrated to their new (destination) locations. Then the migrated runnables are removed from the source location, and their next execution will be performed in their destination locations. If a runnable is of combinational nature (its outputs depend solely on input values; all DemoCar runnables have this property), only the runnable code section is to be migrated. In case of a sequential nature of a runnable, the whole context is to be migrated.

Similarly to [98], we split a runnable context intro two parts: invariant, which is not modified at runtime, and dynamic, including all volatile memory locations. We assume that an upper bound of the dynamic part size of all runnables is known in advance. This part shall be migrated at once using the last instance of the periodic server. It means that the local memory locations that can be modified by the runnable must not be precopied, but migrated after the last execution of the runnable in the old location. This requirement can influence the minimum periodic server size and, consequently, the network bandwidth, as it must be then wide enough to guarantee migration of dynamic part before the next runnable execution (in the new location). This property shall be checked using (5.9).

In the proposed approach, any kind of periodic servers can be used, however, the trade-off between implementation complexity and ability to

guarantee the deadlines of hard real-time tasks, as described for example in [40], shall be considered.

The number of the hyperperiods required for performing task migration depends on the size of runnables and labels to be transferred, mappings, and network bandwidth, in particular flit size FS and timing constants for packet latencies while traversing one router and one link d_R and d_L.

Example 7 *(continuation of Example 6) The flit size, FS, has been fixed to 16 bits. A few examples of the number of hyperperiods required to migrate tasks from* PowerUp *to* Cluster_1*, depending on constants d_R and d_L are presented in Table 5.1. The hyperperiod length for DemoCar equals 100 ms and this time is enough to migrate all data when the router and link latencies are equal to 50 and 100 ns, respectively.*

5.2.6 On-Line Steps

In the proposed approach, only two steps are performed on-line: *Detection of current mode* and *Mapping switching*. Both of these steps are characterised with low computational complexity and thus they impose low overhead for the system during run time.

We assume that the system states are defined explicitly and there is a possibility of determining the current state by observing some system model variables, similarly to [104]. Otherwise, the most efficient multi-choice knapsack problem (MMKP) heuristics, listed in the brief survey earlier, have to be applied to identify the current mode on-line, as proposed in [73].

When the mode change is requested, an agent residing in each core prepares a set of packages with runnables to be migrated via the network. This agent is configured statically and is equipped with a table with information which runnables have to be migrated during a particular mode change. Then the precopy of these runnables is performed. In the following hyperperiods, runnables are transported using periodic servers of the length determined statically in step *Schedulability analysis for taskset during mode change*. The agent is aware of the number of periodic server instances that have to be used during the whole migration process (as in example in Table 5.1), and have the

Table 5.1 Number of hyperperiods (100 ms) required for switching between states *PowerUp* to *Cluster_1* in DemoCar depending on router (d_R) and one link latencies (d_L)

d_R [ns]	d_L [ns]	No. of Hyperperiods
50	100	1
100	200	2
100	400	3
200	500	4
400	800	6
500	1000	7

volatile portion of the context identified. If this instance number elapses, the runnables that have been migrated are killed.

Simultaneously, the same agent can receive migration data from other agents in the network. After the appropriate number of hyperperiods, the contexts of these runnables are fully migrated and are ready to be executed by the operating system.

The details of the agent depend on the underlying operating system.

5.3 Related Works

Systems with distinguishable operating modes are increasingly popular in research. A number of research activity aims at developing design-time (off-line) heuristics to reduce the number of operating points, since the amount of possible scenarios is typically prohibitively high [89]. This Design Space Exploration (DSE) process can be carried out using classic heuristic techniques (including genetic algorithms [73, 89, 116], tabu search [115], simulated annealing [160], particle swarm optimization [103], etc.), or with techniques for pruning the design space [107], performing statistical analysis for identifying potentially benefiting operating points, or use a priori knowledge of the target platform [14]. Then during run-time of that system, a run-time manager (RTM) chooses an appropriate operating point according to the available platform resources by solving an instance of multi-dimensional multi-choice knapsack problem (MMKP). Despite MMKP belongs to the NP-hard complexity class, there exists a number of light-weight greedy heuristics facilitating finding a quasi-optimal mapping during run-time [73]. Alternatively, in [104], there is a possibility of determining the current mode out of explicitly given set by observing some variables of the model. In our work, the current mode is determined in a similar way.

Two different mapping approaches are proposed in [119]. In the first one, named global static power-aware mapping, each task is assigned to one particular processing element independently from the actual scenario. This approach reduces the amount of memory required for storing the configurations and increases the efficiency of run-time management. However, it results in increased power consumption in comparison with the second approach, dynamic power-aware scenario-mapping, where this assumption is relaxed and different mappings for scenarios are stored. These approaches do not allow task migration – once a task is assigned to a processing element, it remains there until finishing its computation. In contrast, Benini et al. [15] allowed tasks to migrate between processing elements when the envisaged performance increase is higher than the precomputed migration cost. This analysis is performed at each instance of configuration change.

In order to analyse the worst case switching time between two modes, it is helpful to show the possible modes and transition between them in a formal way, using for example Finite State Machines (FSMs), as proposed in [118]. In this way it is possible to enumerate all allowed modes and transitions, and to check the cost of mode switchings. In [55], an average switching time overhead for H.264 decoder has been measured to be equal to 0.2% of the total system time. This sligh value has been caused by a low number of existing modes, obtained due to the clustering, and thus relatively rare switches. In hard real-time systems such decrease of modes by clustering is even more crucial and thus it is incorporated in the proposed design flow. In [137], the authors suggest to map as many tasks as possible to the same core in various modes to avoid the data or code items to be moved between different resources when switching between modes. In the proposed approach, we use a genetic algorithm to minimize the amount of data to be migrated.

To perform a schedulability analysis during mode changes, the data migration work is performed during time slots allocated to a periodic server. There exist different kinds of such servers, including *polling servers*, *sporadic servers* and *deferrable servers*, with different replenish policies of server execution time [40]. Despite these differences, their period and maximum execution time during one period are selected in a way that the chosen end-to-end scheduling test proves that no deadline is violated. To decrease the timing of best-effort (i.e., migration) task execution, the *best-effort bandwidth server* includes a slack reclaiming procedure and an algorithm for determining appropriate server parameters [11]. An example of a direct application of *synchronized deferrable servers* for multi-core systems has been demonstrated in [159], but its authors assumed the migration cost to be either negligible, or added to the worst-case execution time of each task, which is difficult to be applied in systems with network architecture prone to contentions. In the proposed approach, more realistic migration time is evaluated, taking into account network parameters and interference from other flows.

In [20], it was experimentally shown that even a total freezing task migration strategy, i.e., where the migrated task is stalled while all its code and data are transferred through links to the target core, can be used in a NoC-based environment and still improve the fulfilment of task deadlines in soft real-time systems. To guarantee hard timing constraints, freeze time should be bounded and possibly short. One of the possible techniques is a precopy strategy, where code and data of some tasks are copied before the actual switching. However, this technique is more complicated than total freezing and has higher migration time, as some data portions may be required to be copied more than once due to their modifications [93]. Storing task code in a few cores and transferring only the necessary data is another possibility. However, in doing so, the storage overhead at each core can increase by a large amount.

A method to guarantee hard real-time for task migration is proposed in [98]. However, a costly schedulability analysis is performed during runtime. No experiments supporting their proposed approach is provided, but one may predict that the overload of that dynamics could be considerable.

The research described in [104] is the closest to the approach proposed in this chapter. Its authors have identified mode transition points in an engine management system, and shown that a load distribution by mode-dependent task allocation is better balanced in comparison with a static task allocation. The performance has been evaluated by simulation, but, contrary to our approach, the task migration costs have not been considered.

From the literature survey it follows that designing real-time systems with distinguishable operating modes has been mainly limited to soft timing constraints. According to the authors' knowledge, there is no proposal of any method guaranteing no hard deadline violation during task migrations. In particular, schedulability analysis has not been applied to check the feasibility of task migration process or to determine the worst case switching time between two operating modes except of the positional paper [98].

5.4 Summary

An approach of task migration in a multi-core network-based embedded system has been proposed as a way to decrease the number of cores needed for guaranteing safe execution of a hard real-time software. The steps to be performed statically have been described in details using illustrative examples based on a lightweight engine control unit. A Finite State Machine describing mode changes has been extracted from the software code and transaction probabilities have been identified during simulation. The closely related modes have been merged into clusters. The most frequent transactions have been identified with the classic Prim's algorithm, and a genetic algorithm has been used to determine the runnable-to-core mapping for the initial mode. Similarly, a genetic algorithm minimizing the number of migrated data has been used for selecting the runnables to be migrated when a change of the current mode is requested. The migration time has been evaluated using schedulability analysis depending on the network bandwidth.

The proposed approach requires development of an agent realising the migration process. Since its architecture details depend on the underlying operating system, its implementation and evaluation in real embedded environments are planned as a future work.