Generalized strictly periodic scheduling analysis, resource optimization, and implementation of adaptive streaming applications

Niknam, S.

Chapter 6

Implementation and Execution of Adaptive Streaming Applications

Sobhan Niknam, Peng Wang, Todor Stefanov. "On the Implementation and Execution of Adaptive Streaming Applications Modeled as MADF". *In Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Sankt Goar, Germany, May 25-26, 2020.

Jiali Teddy Zhai, **Sobhan Niknam**, Todor Stefanov. "Modeling, Analysis, and Hard Real-time Scheduling of Adaptive Streaming Applications". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, No. 11, pp. 2636-2648, Nov 2018.

IN this chapter, we present our implementation and execution approach for adaptive streaming applications modeled as MADF graphs, which corresponds to the fourth research contribution, briefly introduced in Section 1.5.4, to address the research question **RQ3**, described in Section 1.4.3. The remainder of the chapter is organized as follows. Section 6.1 introduces, in more details, the problem statement and the addressed research question. It is followed by Section 6.2, which gives a summary of the contributions presented in this chapter. Section 6.3 gives an overview of the related work. Section 6.4 introduces an extra background material, on K-Periodic Schedules, needed for understanding the contributions of this chapter. Section 6.5 presents our extension of the MOO transition protocol (described in Section 2.1.2 and Section 2.4) followed by Section 6.6 presenting our proposed parallel implementation and

execution approach for the MADF MoC. Section 6.7 presents two case studies to demonstrate the practical applicability of our approach, presented in Section 6.6. Finally, Section 6.8 ends the chapter with conclusions.

6.1 Problem Statement

Recall, from Section 1.4.3, that the last phase of the design flow, considered in this thesis and shown in Figure 1.2, is to implement and execute the analyzed application on an MPSoC platform. This phase is an important step towards designing an embedded streaming system where the system should behave at run-time as expected according to the performed analysis at design-time. Concerning static streaming applications, an implementation and execution approach for such applications modeled as CSDF graphs and analyzed by the SPS framework, briefly described in Section 2.3, is presented in [7]. For adaptive streaming applications, modeled and analyzed with the MADF MoC [94], briefly described in Section 2.1.2, however, no attention has been paid so far at this implementation phase. Thus, in this chapter, we investigate the possibility to implement and execute an adaptive streaming application, modeled and analyzed with the MADF MoC, on an MPSoC platform, such that the properties of the analyzed model are preserved.

6.2 Contributions

In order to address the problem described in Section 6.1, in this chapter, we propose a simple, yet efficient, parallel implementation and execution approach for adaptive streaming applications, modeled with the MADF model, that can be easily realized on top of existing operating systems. Moreover, we extend the offset calculation of the MOO transition protocol, briefly described in Section 2.4, for the MADF model in order to enable the utilization of a wider range of schedules, i.e., K-periodic schedules [17], during the model analysis, implementation, and execution depending on the scheduling support provided by the MPSoC and its operating system onto which the streaming application runs.

More specifically, the main contributions of this chapter are as follows:

- We extend the MOO transition protocol employed by the MADF model. This extension enables the applicability of many different schedules to the MADF model, thereby generalizing the MADF model and making

MADF schedule-agnostic as long as K-periodic schedules are considered;

- We propose a generic parallel implementation and execution approach for adaptive streaming applications modeled with MADF that conforms to the analysis model and its operational semantics [94]. We demonstrate our approach on LITMUS^{RT} [22] which is one of the existing real-time extensions of the Linux kernel;
- Finally, to demonstrate the practical applicability of our parallel implementation and execution approach and its conformity to the analysis model, we present a case study (see Section 6.7.1) on a real-life adaptive streaming application. In addition, we present another case study (see Section 6.7.2) on a real-life streaming application to validate our proposed energy-efficient periodic scheduling approach, presented in Chapter 5, which adopts the MOO protocol of the MADF MoC for switching the application schedule, with a practical implementation of this approach by using our generic parallel implementation and execution approach presented in this chapter.

6.3 Related Work

In [60], the MCDF model is presented where the same application graph is used for both analysis and execution on a platform. In such graph, special actors, namely switch and select actors, are used to enable reconfiguration of the graph structure according to an identified mode by a mode controller at run-time. In the MCDF model, every mode is represented as a single-rate SDF graph and the actors are scheduled on each processor according to a precomputed static schedule, called quasi-static order schedule, in which extra switch and select actors are required to model the schedule in the graph. In contrast to MCDF, the MADF model [94], we consider in our work, is more expressive as each mode is represented as a CSDF graph. Moreover, our proposed MOO transition protocol extension and our implementation and execution approach for the MADF model are schedule agnostic and do not require extra switch and select actors. Therefore, our approach enables the utilization of many different schedules than only a static-order schedule, with no need of extra actors.

In [33], the FSM-SADF model is presented as another analysis model for adaptive streaming applications. To implement an application modeled and analyzed with FSM-SADF, two programming models have been proposed in [89, 90]. In [89], the programming model is constructed by merging the SDF

graphs of all scenarios into a single graph which may be larger than the FSM-SADF analysis graph. Then, to enable switching to a new scenario, all actors in all scenarios are constantly kept active while only those actors belonging to the identified new scenario by a detecting actor(s) will be executed after switching. In this way, a single static-order schedule can be used for the application in all scenarios. In contrast to [89], the proposed programming model in [90] uses a similar switch/select actors, as in MCDF [60], in the constructed graph for switching between scenario graphs at run-time. Then, the graph is reconfigured at run-time using the switch/select actors according to the identified scenario by a detecting actor(s) while updating the application's static-order schedule accordingly. However, the proposed programming models in [89,90] need to be derived manually, thereby requiring extra effort by the designer. More importantly, these programming models assume that actors in all scenarios of an application are active all the time. This can result in a huge overhead for applications with a high number of modes, thereby leading to inefficient resource utilization. In contrast to [89,90], our implementation and execution approach does not require derivation of an additional model and enables the utilization of many different schedules rather than only static-order schedule. Moreover, our approach (de)activates actors in different modes at run-time, so we do not need to keep all modes active all the time, thereby avoiding the unnecessary overhead imposed by the approaches in [89,90].

In [47], the task allocation of adaptive streaming applications onto MPSoC platforms under self-timed (ST) scheduling is studied when considering transition delay during mode transitions. In [47], however, the verification of the proposed approach and mode transition mechanism is limited to simulations and no implementation and execution approach is provided. In contrast, in this chapter, we propose a generic parallel implementation and execution approach for applications modeled with MADF which enables the applicability of many different schedules on the application as well as execution of the application on existing operating systems.

6.4 K-Periodic Schedules (K-PS)

In [19], K-periodic schedules (K-PS) of streaming applications modeled as CSDF graphs are introduced, implying that K_i consecutive invocations of an actor $A_i \in \mathcal{A}$ occur periodically in the schedule. For example, when $K_i = q_i$ for every actor $A_i \in \mathcal{A}$, such K-PS is equivalent to a ST schedule [85] where all q_i invocations of the actor A_i in one graph iteration occur in each period and can result in the maximum throughput for a given CSDF graph. On the other

hand, when $K_i = 1$ for every actor $A_i \in \mathcal{A}$, 1-PS is achieved in which only a single invocation of the actor occurs in each period. The SPS schedule [8], briefly described in Section 2.3, is a special case of 1-PS in which the actors are converted to real-time tasks to enable the application of classical hard real-time scheduling algorithms [29], e.g., EDF, to streaming applications modeled as CSDF graphs. Therefore, in general, the K-PS notion covers a wide set of schedules ranging between 1-PS and ST schedules.

6.5 Extension of the MOO Transition Protocol

As explained in Section 2.4, when multiple actors of an application, modeled as an MADF graph, are allocated on the same processor, the processor can be potentially overloaded during mode transitions due to simultaneous execution of actors from different modes. Therefore, a larger offset, than the offset x computed by using Equation (2.4), may be needed by the MOO protocol to delay the starting time of the new mode during a mode transition in order to avoid processor overloading. Then, this offset, represented with δ, is computed under the SPS schedule by using Equation (2.20). As the SPS schedule has the notion of a task utilization, by converting the actors in a CSDF graph to real-time (RT) tasks, the offset δ is computed, according to Equation (2.20), by making the total utilization of the RT tasks allocated on each processor during mode transition instants to not exceed the processor capacity. However, since the K-periodic schedules (K-PS), considered in this chapter and briefly introduced in Section 6.4, have no notion of a task utilization, the offset δ for any K-PS cannot be computed as in Equation (2.20). Therefore, in this section, we extend the MOO transition protocol to compute such an offset for any K-PS.

In fact, to avoid the processor overloading under any K-PS, the schedule interferences of modes (in terms of overlapping iteration period H) during mode transitions must be resolved on each processor. For instance, consider the MADF graph G_1 in Figure 6.1(a), explained in Section 2.1.2, with two operating modes SI^1 and SI^2. Figure 6.2(a) and Figure 6.2(b) show the corresponding CSDF graphs of modes SI^1 and SI^2, respectively. An execution of both modes SI^1 and SI^2 under a K-PS are shown in Figure 6.3(a) and Figure 6.3(b), respectively, as well as an execution of G_1 with two mode transitions and the computed offsets $x^{1\rightarrow 2} = 3$ and $x^{2\rightarrow 1} = 1$, for mode transitions from SI^1 to SI^2 and vice versa, according to Equation (2.4), is illustrated in Figure 6.4(a). Now, let us assume the allocation of all actors of G_1 on an MPSoC platform $\Pi = \{\pi_1, \pi_2, \pi_3, \pi_4\}$ containing four processors that is shown in Figure 6.1(b).

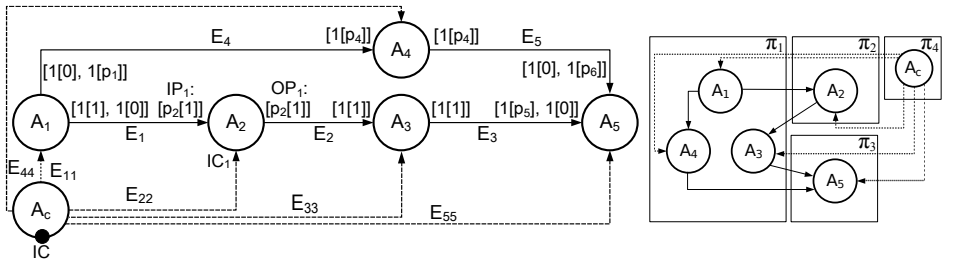
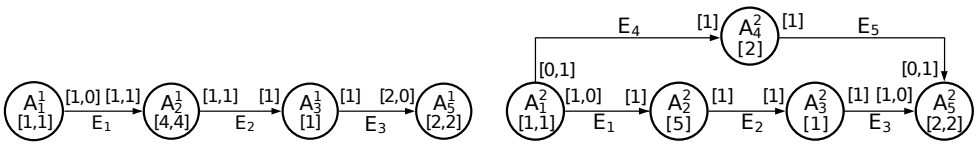


Figure 6.1: *(a) An MADF graph G_1 (taken from Section 2.1.2). (b) The allocation of actors in graph G_1 on four processors.*



(a) CSDF graph G_1^1 of mode SI^1. (b) CSDF graph G_1^2 of mode SI^2.

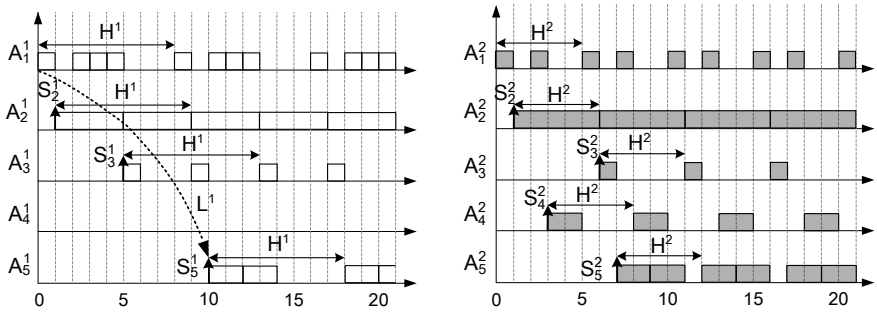
Figure 6.2: *Two modes of graph G_1 in Figure 2.1 (taken from Section 2.1.2 with modified WCET of the actors).*

Then, considering the execution of G_1 in Figure 6.4(a), the schedule interferences on π_1 happen during time periods $[6, 11]$ and $[25, 27]$ for mode transition from SI^2 to SI^1 and vice versa, respectively, while no schedule interference happens on π_2 and π_3. Obviously, to resolve the schedule interferences on π_1, the earliest start time of actors in the new mode should be further offset by the length of the time period in which the schedule interferences happen. Therefore, the extra offsets for mode transitions from SI^2 to SI^1 and vice versa on π_1 are $11 - 6 = 5$ and $27 - 25 = 2$ time units, respectively, thereby resolving the schedule interferences on π_1, as shown in Figure 6.4(b). In this example, $\delta^{2\to1} = x^{2\to1} + 5 = 6$ and $\delta^{1\to2} = x^{1\to2} + 2 = 5$.

Now, considering any K-PS, the offset $\delta^{o\to n}$ can be computed as the **maximum schedule overlap** among all processors when the new mode SI^n starts immediately after the source actor of the old mode SI^o completes its last iteration, as follows:

$$\delta^{o\to n} = \max\{x^{o\to n}, \max_{\substack{\forall {}^m\Psi_i^o \in {}^m\Psi^o \wedge {}^m\Psi_i^n \in {}^m\Psi^n \\ {}^m\Psi_i^o \neq \emptyset \wedge {}^m\Psi_i^n \neq \emptyset}} (\max_{A_j^o \in \Psi_i^o} S_j^o - \min_{A_k^n \in {}^m\Psi_i^n} S_k^n)\} \tag{6.1}$$

where ${}^m\Psi = \{{}^m\Psi_1, \ldots, {}^m\Psi_m\}$ is m-partition of all actors on m number of pro-



(a) Mode SI^1 in Figure 6.2(a) (b) Mode SI^2 in Figure 6.2(b)

Figure 6.3: *Execution of both modes SI^1 and SI^2 under a K-PS.*

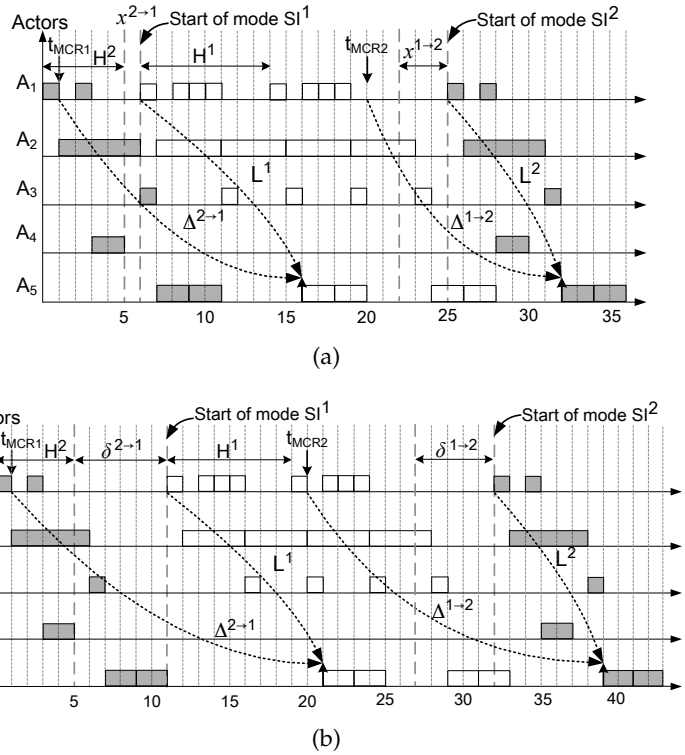


Figure 6.4: *Execution of G_1 with two mode transitions under (a) the MOO protocol, and (b) the extended MOO protocol with the allocation shown in Figure 6.1(b).*

cessors, i.e., ${}^m\Psi_i^o$ and ${}^m\Psi_i^n$ are the sets of actors allocated on the i-th processor (π_i) in the old mode SI^o and the new mode SI^n, respectively. For instance,

consider the allocation of G_1 on the four processors, shown in Figure 6.1(b), and the K-PS of modes SI^1 and SI^2 given in Figure 6.3(a) and 6.3(b), respectively. The offset $\delta^{1\rightarrow 2}$ of the mode transition from SI^1 to SI^2 on each processor is computed using Equation (6.1) as follows: (π_1) $S_3^1 - S_1^2 = 5 - 0 = 5$, (π_2) $S_2^1 - S_2^2 = 1 - 1 = 0$, and (π_3) $S_5^1 - S_5^2 = 10 - 7 = 3$, thereby resulting in the offset $\delta^{1\rightarrow 2} = \max(3, \max(5, 0, 3)) = 5$ for the start time of mode SI^2, as shown in Figure 6.4(b). Similarly, the offset $\delta^{2\rightarrow 1}$ of the mode transition from SI^2 to SI^1 on each processor is computed using Equation (6.1) as follows: (π_1) $S_3^2 - S_1^1 = 6$, (π_2) $S_2^2 - S_2^1 = 0$, and (π_3) $S_5^2 - S_5^1 = -3$, and $\delta^{2\rightarrow 1} = \max(1, \max(6, 0, -3)) = 6$.

6.6 Implementation and Execution Approach for MADF

In this section, we first present our generic parallel implementation and execution approach (Section 6.6.1) for an application modeled as an MADF. Then, in Section 6.6.2, we demonstrate our approach on LITMUS^{RT} [22].

6.6.1 Generic Parallel Implementation and Execution Approach

In this section, we will explain our approach by an illustrative example. Consider the MADF graph G_1 shown Figure 6.1(a). Our implementation consists of three main components: 1) (normal) actors, 2) a control actor, and 3) FIFO channels. We implement the actors as separate threads and the FIFO channels as circular buffers [15] with non-blocking read/write access. Thus, the execution of the threads and the read/write from/to the FIFO channels are controlled explicitly by an operating system supporting and using any K-PS, briefly introduced in Section 6.4. A valid K-PS schedule always ensures the existence of sufficient data tokens to read from all input FIFO channels and sufficient space to write data tokens to all output FIFO channels when an actor executes.

In our implementation, all FIFO channels in the MADF graph of an application are created statically before the start of the application execution to avoid duplication of FIFO channels and unnecessary use of more memory during mode transitions. On the other hand, the threads corresponding to the actors are handled at run-time. This means that when a mode change request (MCR) occurs, in order to switch the application's mode, the executing threads in the old mode are stopped and terminated whereas the threads corresponding to the actors in the requested new mode are created and launched at run-time. In this way, our implementation enables task migration during mode transitions

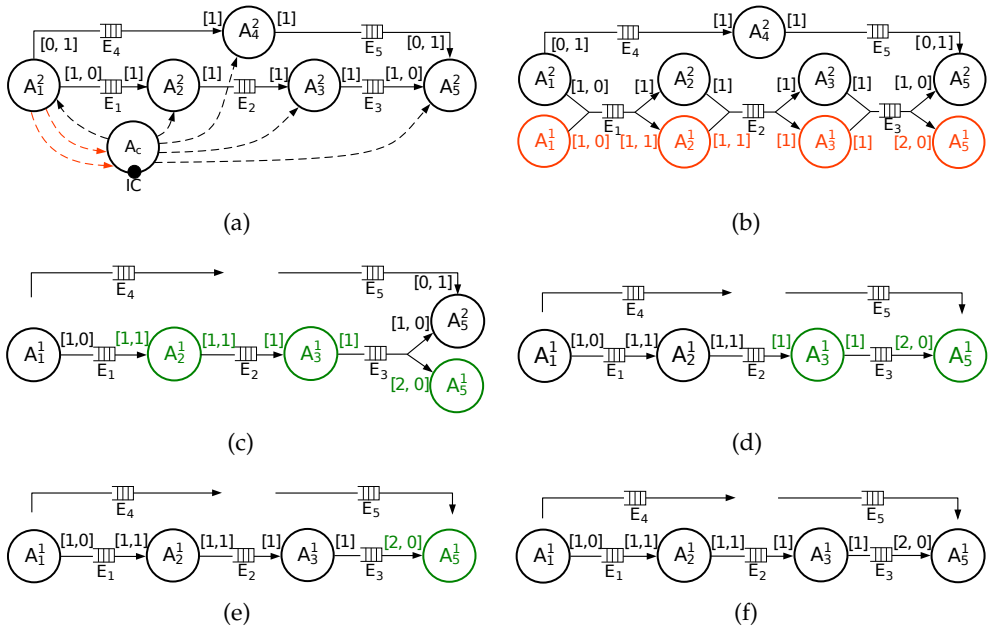


Figure 6.5: *Mode transition of G_1 from mode SI^2 to mode SI^1 (from (a) to (f)). The control actor and the control edges are omitted in figures (b) to (f) to avoid cluttering.*

by using a different task allocation in each application's mode. For instance, the implementation and execution of the mode transition from mode SI^2 to mode SI^1 of G_1, with the given schedule in Figure 6.4(b), is shown in Figure 6.5 and has the following sequence - Figure 6.5(a): The application is in mode SI^2 where the threads corresponding to the actors in this mode run. The threads are connected to the control thread A_c, which runs on a separate processor, through the control FIFO channels (the dashed arrows in Figure 6.5(a)). In our approach, two extra FIFO channels, shown in the red color in Figure 6.5(a), are required, both from the thread of source actor A_1 to control thread A_c in order to notify the control thread in which graph iteration number the source actor is currently running and the time when the thread of the source actor is terminated; Figure 6.5(b): When MCR1 occurs at time instant $t_{\text{MCR1}} = 1$ to switch to mode SI^1, the threads corresponding to the actors in mode SI^1 are created and connected to the corresponding FIFO channels. At this stage the newly-created threads (the red nodes in Figure 6.5(b)) are suspended and they wait to be released. *Note that the mode transitions cannot be performed at any moment. According to the operational semantics of the MADF model, a mode transition is only allowed in a consistent state, that is, after the graph iteration in*

which the MCR occurred, has completed and the graph has returned to its initial state. Therefore, control thread A_c needs to check the current graph iteration number of the source actor A_1^2 and notify all threads at which graph iteration number they have to be terminated; Figure 6.5(c): Next, when the thread of the source actor A_1^2 is terminated at time instant 5 (according to Figure 6.4(b)), which is notified to control thread A_c as well, the control thread signals the suspended threads to be released synchronously $\delta^{2\rightarrow 1} = 6$ time units later at time instant 11 (according to Figure 6.4(b)). At this stage, a mixture of threads in both modes may be running on processors. In the meanwhile, the threads of the actors in the old mode SI^2 are gradually finishing their execution and terminated at the same graph iteration number; Figure 6.5(d)-6.5(f): Since the actors have different start time in the new mode SI^1, as shown in Figure 6.4(b), the threads in mode SI^1 start executing accordingly after the releasing time. The threads which are released but not yet running, are shown in the green color. Then, the released threads in the new mode SI^1 gradually start running and finally, the application is switched to mode SI^1 where all created threads run and the unused channels E_4 and E_5 in this mode are left unconnected to the threads.

6.6.2 Demonstration of Our Approach on LITMUS^{RT}

In this section, we demonstrate how to realize our implementation and execution approach on LITMUS^{RT} [22] as one of the existing real-time (RT) extensions of the Linux kernel. The realizations of a normal actor and the control actor in our approach are given in C++ in Listing 6.1 and 6.2, respectively, in which the bolded primitives belong to LITMUS^{RT}. Note that, any other RT operating system which has similar primitives, e.g., FreeRTOS [72], can be used instead. We also use the standard POSIX Threads (Pthreads) and the corresponding API integrated in Linux to create the threads of the actors.

In Listing 6.1, the RT parameters of an actor, e.g., actor A_2 of graph G_1 shown in Figure 6.1(a), are set up using the data structure `threadInfo` passed to the function as argument in Lines 2-6. Under partitioned scheduling algorithms, e.g., Partitioned EDF, the processor core which the thread should be statically executing on, is set in Line 7. Then, the RT configuration of the thread is sent to the LITMUS^{RT} kernel for validation, in Line 8, in which if it is verified, the thread is admitted as a RT task in LITMUS^{RT}, in Line 9. In Line 10, the RT task is suspended, in order to synchronize the start time of the tasks, until signaled by the *control actor* to begin its execution. Next, the task enters to a `while loop` in Lines 12-31, in which iterates infinitely. At the beginning of each graph iteration, the current time instant is captured and stored in

```
1void Actor_A2(void *threadarg) {
2  threadInfo = (threadInfo *)threadarg; // Get the thread parameters
3  struct rt_task param; // Set up RT parameters
4  param.period = threadInfo.period;
5  param.relative_deadline = threadInfo.relative_deadline;
6  param.phase = threadInfo.start_time;
7  be_migrate_to_domain(threadInfo.processor_core); // For partitioned schedulers
8  set_rt_task_param(gettid(), &param));
9  task_mode(LITMUS_RT_TASK); // The actor is now executing as a RT task
10 wait_for_ts_release(); // The RT task is waiting for a release signal
11 int graph_iteration = 1;
12 while(1) { // Enter to the main body of the task
13    lt_t now = litmus_clock();
14    for(i=1; i<=threadInfo.repetition; i++){
15       lt_sleep_until(now + threadInfo.slot_offset[i]);
16       if(IC_1 is not empty) READ(& terminate, threadInfo.IC_1);
17       if(i == 1 && graph_iteration > terminate){
18          WRITE(& now, threadInfo.OC_trig);
19          task_mode(BACKGROUND_TASK); //Trans. back to non−RT mode
20          return NULL;
21       }
22       if(i == 1) WRITE(& graph_iteration, threadInfo.OC_iter);
23       if(threadInfo.mode == 1){ // Do action according to the task's mode
24          READ(& in1, threadInfo.IP_1);
25          task_function(& in1, & out1);
26          WRITE(& out1, threadInfo.OP_1);
27       }/* Actions according to the other modes */ { . . . }
28       if(i%threadInfo.K == 0) sleep_next_period();
29    }
30    graph_iteration += 1;
31 }
32}
```

Listing 6.1: *C++ code of actor* A_2

variable `now` in Line 13. Then, the task iterates as many repetition times as it has in one graph iteration in a `for loop`, in Lines 14-29. In Line 15, the task sleeps until reaching the start time of its *i*-th invocation, corresponding to the K-PS, from the time instant captured in `now`. After finishing K_i invocations, the task sleeps again, in Line 28, until finishing the current period. In fact, in this line, a kernel-space mechanism is triggered for moving the task from the ready queue to the release queue. Then, LITMUS^{RT} will move the task

```
1 void main(int argc, char **argv) {
2   /* Create FIFO channel E_1 */
3   size_E_1_in_tokens = 4;
4   size_token_E_1= sizeof(token_structure)/sizeof(int);
5   size_fifo_E_1 = size_E_1_in_tokens × size_token_E_1;
6   E_1 = calloc(size_fifo_E_1+2, sizeof(int)); // Allocate memory for E_1
7   /* Create other FIFO channels*/ {···}
8   init_litmus(); // Initialize the interface with the kernel
9   old_mode = 1, new_mode = 1;
10  while(1){
11    switch(new_mode){
12      case 1: /* Create and launch the thread of actor A_2 in mode SI^1*/
13        threadInfo.mode = 1; thread.repetition = 2; threadInfo.processor_core = 1;
14        threadInfo.IP_1 = E_1; /* Connect other FIFO channels to the thread*/ {···}
15        threadInfo.period = 8; threadInfo.relative_deadline = 8;
16        threadInfo.phase = 1; threadInfo.slot_offset = [0, 4];
17        pthread_create(&threadInfo.id, NULL, &Actor_A2, &threadInfo);
18        /* Create and launch the threads of the other actors in mode SI^1*/ {...}
19      case 2: { /* Create and launch the thread of the actors in mode SI^2*/ }
20    }
21    while(rt_task == ready_rt_tasks)
22      read_litmus_stats(&ready_rt_tasks);
23    if(new_mode != old_mode){
24      while(IC_trig is empty);
25      READ(& now, IC_trig);
26    }else now = litmus_clock();
27    release_ts(δ); old_mode = new_mode;
28    do{ READ(& new_mode, IC); } while(new_mode == old_mode)
29    READ(& graph_iteration, IC_iter);
30    t_left = H^o −(litmus_clock() − now − δ)%H^o;
31    if(t_left < t_OV) graph_iteration += ⌈(t_OV − t_left)/H^o⌉;
32    for(all active actor A_i) WRITE(& graph_iteration, OC_i);
33 }
```

Listing 6.2: *C++ code of control actor A_c*

back to the ready queue at the start time of the next period when the task will again be eligible for execution. In Line 16, the state of the input control port $\texttt{IC}_1$ is checked in which if it is not empty, the graph iteration number where the task has to be terminated is read. Then, the termination condition is checked in Line 17. If the condition holds, the mode of the thread is changed to non-RT in Line 19 and the thread is terminated in Line 20. Otherwise, the

task reads from its input FIFO channels, executes its function, and writes the result to the output FIFO channels, in Lines 23-27. Only for the source actor, the latest graph iteration number where the task is currently running and the time instant `now` are written to the output control ports $\texttt{OC}_{\texttt{iter}}$ and $\texttt{OC}_{\texttt{trig}}$, in Lines 22 and 18 highlighted with red color, respectively, which are needed by the control thread, as explained in Section 6.6.1.

In Listing 6.2, realizing control actor A_c, all FIFO channels are created and the needed memory is allocated to them using the standard `calloc()` function, in Lines 3-7. In Line 8, the interface with the LITMUS^{RT} kernel is initialized. In Lines 11-20, the data structure of `threadInfo` is initialized for each actor of the requested new mode and the corresponding threads of the actors in the new mode are created and launched. In Lines 21 and 22, the number of suspended RT tasks is checked which if is equal to the number of the actors in the new mode, they can be signaled to be released simultaneously. Therefore, in Line 27, the global release signal is sent by δ time units after receiving the time instant `now` on the input port $\texttt{IC}_{\texttt{trig}}$ from the thread of the source actor in the old mode in Line 25, implying the termination of the thread and acting as a trigger. Afterwards, the control actor continuously monitors the occurrence of a new MCR in Line 28. If an MCR occurs to a new mode which differs from the current mode, the graph iteration number in which the threads in the current mode need to be terminated is computed in Lines 29-31. The primary graph iteration number is simply the current graph iteration number of the source actor, read from the input port $\texttt{IC}_{\texttt{iter}}$ in Line 29. However, since the control actor has certain timing overhead, represented by $\texttt{t}_{\texttt{OV}}$, the primary graph iteration number needs to be revised corresponding to the time left from the current graph iteration of the source actor $\texttt{t}_{\texttt{left}}$, computed in Line 30, and $\texttt{t}_{\texttt{OV}}$, in Line 31, to ensure that all threads will be terminated in the same graph iteration number. Then, the new graph iteration number is written on the control port of all threads in the current mode in Line 32 to notify them when they have to be terminated.

6.7 Case Studies

In this section, we present two case studies using real-life streaming applications to validate the proposed implementation and execution approach in Section 6.6 as well as the proposed periodic scheduling approach in Chapter 5 by running the applications on actual hardware. We perform these case studies on the ARM big.LITTLE architecture [40], shown in Figure 1.1, including a quad-core Cortex A15 (big) cluster and a quad-core Cortex A7 (LITTLE)

Table 6.1: *Performance results of each individual mode of Vocoder.*

Mode	Analysis [94]		Implementation and execution		Number/Type of processor
	H (ms)	L (ms)	H (ms)	L (ms)	
SI^8	25	21	25	21	1 LITTLE
SI^{16}	25	19	25	19	1 big
SI^{32}	25	33	25	33	2 big
SI^{64}	25	56	25	56	3 big

cluster, that is available on the Odroid-XU4 platform [66]. The Odroid XU4 runs Ubuntu 14.04.1 LTS along with LITMUS^{RT} version 2014.2.

6.7.1 Case Study 1

In this section, we present a case study, using a real- life adaptive streaming application, to demonstrate the practical applicability of our parallel implementation and execution approach for MADF. Moreover, we show that our approach conforms to the MADF analysis model in [94] by measuring the application's performance, in terms of the achieved iteration period, iteration latency, and mode transition delay, and comparing them with the computed ones using the MADF analysis model.

In this case study, we take a real-life adaptive streaming application from the StreamIT benchmark suite [37], called Vocoder, which implements a phase voice encoder and performs pitch transposition of recorded sounds from male to female. We modeled Vocoder using the MADF graph, shown in Figure 6.6, with four modes which captures different workloads. The four modes $\{SI^8, SI^{16}, SI^{32}, SI^{64}\}$ specify different lengths of the discrete Fourier transform (DFT), denoted by $\mathsf{dl} \in \{8, 16, 32, 64\}$. Mode SI^8 ($\mathsf{dl} = 8$) requires the least amount of computation at the cost of the worst voice encoding quality among all DFT lengths. Mode SI^{64} ($\mathsf{dl} = 64$) produces the best quality of voice encoding among all modes, but is computationally intensive. The other two modes SI^{16} and SI^{32} exploit the trade-off between the quality of the encoding and the computational workload. Therefore, the resource manager of an MPSoC can take advantage of this trade-off and adjust the quality of the encoding according to the available resources, such as energy budget and number/type of processors, at run-time.

We measured the WCET of the actors in Figure 6.6 in the four modes on both big and LITTLE processors. Then, since the shortest time granularity visible to LITMUS^{RT}, i.e., the OS clock tick, is 1 millisecond (ms), the WCET of the actors are rounded up to the nearest multiple of the OS clock tick duration. This is necessary to derive the period and start time of the actors

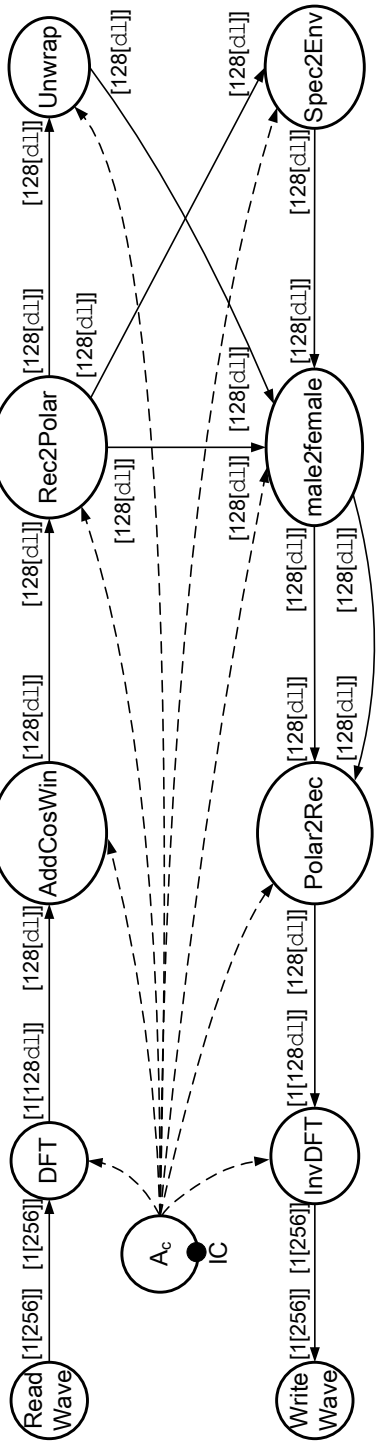


Figure 6.6: *MADF graph of the Vocoder application.*

Table 6.2: *Performance results for all mode transitions of Vocoder (in ms).*

Transition (SI^o to SI^n)	Analysis [94]		Implementation and execution
	$\Delta^{o\to n}_{min}$	$\Delta^{o\to n}_{max}$	$\Delta^{o\to n}$
$SI^8 \to SI^{64}$	146	171	160
$SI^8 \to SI^{32}$	123	148	131
$SI^8 \to SI^{16}$	111	136	122
$SI^{16} \to SI^{64}$	165	190	185
$SI^{16} \to SI^{32}$	142	167	157
$SI^{16} \to SI^8$	112	137	130
$SI^{32} \to SI^{64}$	162	187	168
$SI^{32} \to SI^{16}$	125	150	139
$SI^{32} \to SI^8$	125	150	145
$SI^{64} \to SI^{32}$	160	185	182
$SI^{64} \to SI^{16}$	146	171	162
$SI^{64} \to SI^8$	146	171	152

under any K-PS to be executed by LITMUS^{RT}. Table 6.1 shows the performance results of each individual mode under the self-timed (ST) schedule, which is a particular case of K-PS explained in Section 6.4. In this table, columns 2-3 show the iteration period H and iteration latency L of each individual application mode computed by the analysis model, respectively. The iteration period H indicates the guaranteed production of 256 samples per 25 ms, as a performance requirement, in all modes by sink actor WriteWave. Column 6 shows the number and type of processors required in each mode to guarantee the aforementioned performance requirement. On the other hand, columns 4-5 show the measured iteration period H and iteration latency L of each individual application mode achieved by our implementation and execution approach, respectively. Comparing columns 2-3 with columns 4-5, we see that the performance of Vocoder computed using the MADF analysis model is the same as the measured performance when Vocoder is implemented and executed using our approach. This is because the ST schedule of each mode is implemented in our approach by setting up, in LITMUS^{RT}, the same periods and start times of the actors as in the analysis model. Based on the results, shown in Table 6.1, we can conclude that our implementation and execution approach conforms to the MADF analysis model in terms of H and L for the Vocoder application.

Now, we focus on the performance results related to the mode transition delays for all 12 possible transitions between the four modes of Vocoder. Using the MADF analysis model in [94], the computed minimum and maximum transition delays are shown in columns 2-3 of Table 6.2, respectively. By using

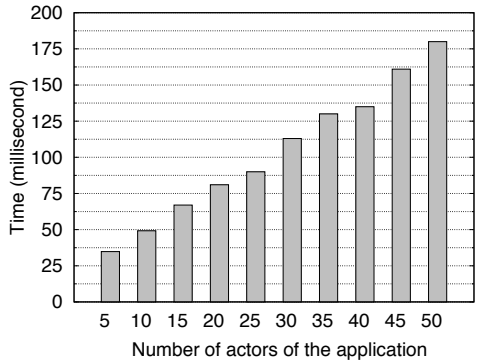


Figure 6.7: *The execution time of control actor A_c for applications with different numbers of actors.*

our implementation and execution approach, however, the measured transition delay depends on the occurrence time of the mode change request (MCR) at run-time, thus the measured transition delay could vary between the computed minimum and maximum values in each transition. For instance, column 4 in Table 6.2 shows the measured transition delay for each transition with a random occurrence time of an MCR, within the iteration period, at run-time. These measured transition delays (column 4) are within the computed bounds using the analysis model (columns 2-3). Therefore, our implementation and execution approach also conforms to the MADF analysis model in terms of mode transition delay $\Delta^{o \to n}$ for the Vocoder application.

Finally, we evaluate the scalability of our proposed implementation and execution approach in terms of the execution time t_{ov} of the control actor for applications with different numbers of actors. Since the most time-consuming and variable part of the control actor is located in Lines 11 to 22 of Listing 6.2, that is the time needed for the threads creation and the threads admission as RT tasks, we only measure the time needed for this part of the control actor. In this regard, the measured time for applications with a varying number of actors is shown in Figure 6.7. In this figure, we can clearly observe that the execution time of the control actor follows a fairly linear scalability when the number of actors in the application increases.

6.7.2 Case Study 2

In this section, we present a case study, using a real-life streaming application, for our energy-efficient periodic scheduling approach presented in Chapter 5. As explained in Chapter 5, this scheduling approach primarily selects a set

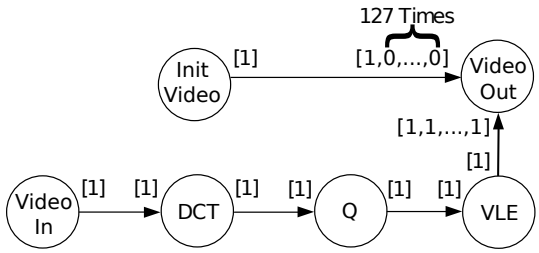


Figure 6.8: *CSDF graph of MJPEG encoder.*

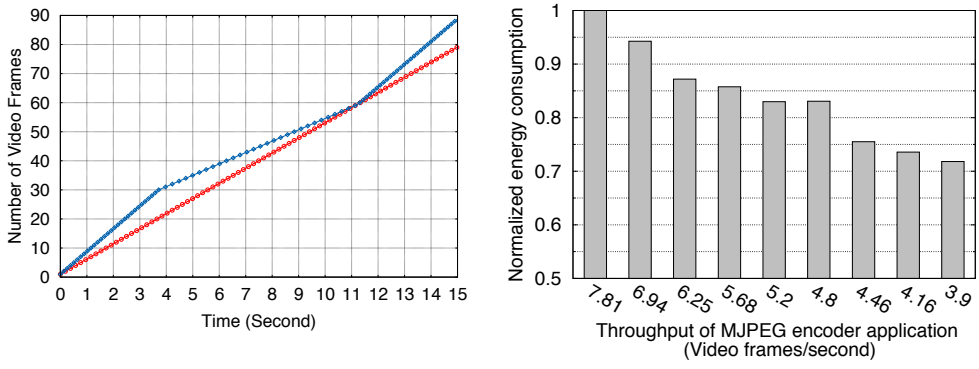
of SPS schedules, as **operating modes**, for an application modeled as a CSDF graph where each mode provides a unique pair of performance and power consumption. Then, it satisfies a given throughput requirement at a long run by switching the application's schedule periodically between modes at run-time. As this scheduling approach is evaluated using only simulations in Chapter 5, this case study aims to validate its applicability on a real hardware platform using our parallel implementation and execution approach presented in Section 6.6. To do so, we only adopt the ARM Cortex A15 cluster with four processors available on the Odroid-XU4 platform. This platform provides the DVFS mechanism per cluster in which the operating frequency of the Cortex-A15 cluster can be varied between 200 MHz to 2 GHz with a step of 100 MHz.

In this case study, we take the Motion JPEG (MJPEG) video encoder application which CSDF graph is shown in Figure 6.8. The specifications of two modes where the SPS schedule is used in each mode of this application, referred as mode SI^1 and mode SI^2, are given in Table 6.3. The iteration period H of these modes, in milliseconds, is given in the second column in Table 6.3. Mode SI^1 has an iteration period of 128 ms which results in the application throughput of $1000/128 = 7.81$ frames/second. Likewise, the iteration period of mode SI^2 is 256 ms which results in the application throughput of $1000/256 = 3.9$ frames/second. In these modes, the operating frequency of the A15 cluster is set to 1.4 GHz and 600 MHz for mode SI^1 and SI^2, respectively, while satisfying their aforementioned application throughput. As a result, these modes have different power consumption which is given in the fourth column in Table 6.3. The WCETs of all actors in these mode are also given in the fifth to tenth columns in Table 6.3. In these modes, we use the partitioned EDF scheduler plugin (PSN-EDF) in LITMUS^{RT} to schedule the actors allocated on each processor separately.

Note that modes SI^1 and SI^2 correspond to two consecutive SPS schedules

Table 6.3: *The specification of modes SI^1 and SI^2 in MJPEG encoder application*

Mode	Iteration Period (ms)	Frequency (GHz)	Power (W)	WCET of actors (ms)					
				Init Video	Video In	DCT	Q	VLE	Video Out
SI^1	128	1.4	2.24	0.003	0.139	0.272	0.136	0.267	0.779
SI^2	256	0.6	1.62	0.004	0.219	0.682	0.251	0.682	1.437



(a) (b)

Figure 6.9: *(a) The video frame production of the MJPEG encoder application over time for the throughput requirement of 5.2 frames/second. (b) Normalized energy consumption of the application for different throughput requirements.*

of the MJPEG encoder application, i.e., no other valid SPS schedule exists between them. So, to satisfy a throughput requirement between 3.9 to 7.81 frames/second, the naive solution is to constantly execute the application in mode SI^1. As a consequence, the application consumes more energy due to producing more frames/second than required. In contrast, our scheduling approach, presented in Chapter 5, can satisfy the throughput requirement at a long run by periodically switching the application execution between mode SI^1 and SI^2. For instance, let us consider the throughput requirement of 5.2 frames/second. Then, Figure 6.9(a) shows the production of video frames over time by the MJPEG encoder application under our proposed scheduling approach. The red line in this figure represents the required number of frames per second according to the throughput requirement whereas the blue curve represents the measured number of produced video frames per second by our scheduling approach implemented and executed on the real hardware platform Odroid XU4. As shown in this figure, the application executes initially in mode SI^1 for about 4 seconds while producing more video frames than required. These excessive frames are accumulated in a buffer to be

consumed when the application executes in mode SI^2 with lower throughput for the next about 7 seconds. After finishing one period of the schedule at about 11 seconds, the application delivers the throughput requirement where the red line and the blue curve in Figure 6.9(a) hit each other. This execution is then repeated indefinitely.

For different throughput requirements, we also measure the energy consumption of the Odroid XU4 platform when running the application using our periodic scheduling approach. To do so, the energy consumption of the Odroid XU4 platform is $E = V \times \int_0^t I(t)dt$, where the current $I(t)$ is obtained by precisely measuring (sampling) the current drawn by the platform during the time interval t of the application execution under the platform operating voltage V. The normalized energy consumption of the platform executing the application with different throughput requirements for a duration of one minute is shown in Figure 6.9(b). This figure clearly shows the effectiveness of our periodic scheduling approach which can reduce the energy consumption by up to 26% compared to the naive scheduling approach, mentioned earlier, where the approach constantly executes in mode SI^1 in order to satisfy any throughput requirement between 3.9 and 7.81 frames per second.

6.8 Conclusions

In this chapter, we proposed a generic parallel implementation and execution approach for adaptive streaming applications modeled with MADF. Our approach can be easily realized on top of existing operating systems and support the utilization of a wider range of schedules. In particular, we demonstrated our approach on $LITMUS^{RT}$ which is one of the existing real-time extensions of the Linux kernel. Finally, we performed a case study using a real-life adaptive streaming application and showed that our approach conforms to the analysis model for both execution of the application in each individual mode and during mode transitions. In addition, we performed another case study using a real-life streaming application to validate the practical applicability of our proposed periodic scheduling approach, presented in Chapter 5, on a real hardware platform by using our generic parallel implementation and execution approach presented in this chapter.