On the Implementation and Execution of Adaptive Streaming Applications Modeled as MADF

Sobhan Niknam, Peng Wang, Todor Stefanov
Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
Email: {s.niknam, p.wang, t.p.stefanov}@liacs.leidenuniv.nl

ABSTRACT

It has been shown that the mode-aware dataflow (MADF) is an advantageous analysis model for adaptive streaming applications. However, no attention has been paid on how to implement and execute an application, modeled and analyzed with the MADF model, on a Multi-Processor System-on-Chip, such that the properties of the analysis model are preserved. Therefore, in this paper, we consider this matter and propose 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 while supporting the utilization of a wider range of schedules. In particular, we demonstrate our approach on LITMUS^{RT} as one of the existing real-time extensions of the Linux kernel. Finally, to show the practical applicability of our approach and its conformity to the analysis model, we present a case study using a real-life adaptive streaming application.

1 INTRODUCTION

Nowadays, many modern streaming applications, in the domain of multimedia, image, and signal processing, increasingly show adaptive behavior at run-time. For example, a computer vision system processes different parts of an image continuously to obtain information from several regions of interest depending on the actions taken by the external environment. To handle the ever-increasing computational demand and real-time (RT) constraints of such adaptive streaming applications, a Multi-Processor System-on-Chip (MPSoC), which contains an increasing number of Processing Elements (PEs), has become a standard platform that is widely adopted in embedded systems design to benefit from parallel execution. Designing such an embedded MPSoC system, however, imposes two major challenges: 1) how to efficiently express the adaptive behavior and parallelism found in streaming applications in order to analyze their behavior and guarantee their timing constraints, and 2) how to implement and execute the analyzed applications on MPSoC platforms.

To address the first challenge and facilitate the analysis of adaptive streaming applications, several parallel models of computation (MoCs), referred as adaptive MoCs in this paper, such as mode-aware dataflow (MADF) [1], mode-controlled dataflow (MCDF) [2], and finite-state machine scenario-aware dataflow (FSM-SADF) [3] have been proposed. These adaptive MoCs are able to capture the behavior of an adaptive streaming application as a collection of a finite number of different dataflow graphs, called scenarios or modes, that are controlled by parameters which values need to be updated at run-time to activate different graphs. In this way, the application's behavior is switched from one scenario/mode to another scenario/mode, thereby expressing the adaptive behavior. Each scenario or mode is modeled using a static MoC such as synchronous dataflow (SDF) graph [4] or cyclo-static dataflow (CSDF) graph [5] and is individually analyzable in terms of performance and resource usage at design-time. As a result, design-time analyzability of the adaptive streaming applications, modeled with the aforementioned adaptive MoCs, can be provided to some extent, e.g., hard real-time (HRT) analysis[1], worst-case performance analysis [3], etc.

Apart from the expressive power, the major difference between the aforementioned adaptive MoCs is in how the actors in a graph, representing application's tasks in a particular mode, are scheduled during mode transitions. In FSM-SADF [3] and MCDF [2], a protocol, referred as self-timed (ST) transition protocol, has been adopted which specifies that tasks are scheduled as soon as possible during mode transitions. This protocol, however, introduces timing interference of one mode execution with another one that can significantly affect and fluctuate the latency of an adaptive streaming application across a long sequence of mode transitions. To avoid such undesirable behavior caused by the ST transition protocol, a simple, yet effective transition protocol, called maximum-overlap offset (MOO), is proposed for the MADF model [1]. The MOO protocol can resolve the timing interference between modes upon mode transitions by properly offsetting the starting time of the new mode. In [1], it has been shown that such offset for the MOO protocol can be calculated when using the strictly periodic scheduling (SPS) [6] in each mode to schedule the application tasks.

Although MADF has the above-discussed advantage compared to FSM-SADF [3] and MCDF [2] due to its powerful MOO transition protocol, so far no attention has been paid on how to address the aforementioned second major challenge, i.e., how to implement and execute an application, modeled and analyzed with the MADF model, on an MPSoC, such that the properties of the analyzed model are preserved. Therefore, in this paper, we consider this matter and propose a simple, yet efficient, parallel implementation and execution approach for adaptive streaming applications, modeled with MADF model, that can be easily realized on top of existing operating systems. Moreover, we extend the offset calculation of the MOO protocol for the MADF model in order to enable the utilization of a wider range of schedules, i.e., K-periodic schedules [7], during the mode 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 paper are summarized as follows:

- We extend the MOO 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 [1]. We demonstrate our approach on LITMUS^{RT} [8] which is one of the existing RT 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 on a real-life adaptive streaming application.

The remainder of the paper is organized as follows: Section 2 gives an overview of the related work. Section 3 introduces the background material needed for understanding the contributions of this paper. Section 4 presents the proposed extension of the MOO transition protocol followed by Section 5 presenting the parallel implementation and execution approach for the MADF MoC.

SCOPES '20, May 25–26, 2020, Sankt Goar, Germany
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7131-5/20/05…$15.00
https://doi.org/10.1145/3378678.3391876

Section 6 presents a case study to show the applicability of our approach presented in Section 5 and Section 7 ends the paper with conclusions.

2 RELATED WORK

In [2], 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 PE 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 [1], we consider in this paper, is more expressive as each mode is represented as a CSDF graph. Moreover, our proposed MOO 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 [3], 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 [9] and [10]. In [9], 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 [9], the proposed programming model in [10] uses a similar switch/select actors, as in MCDF [2], 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 [9] and [10] 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 [9] and [10], 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 [9] and [10].

In [11], the task allocation of adaptive streaming applications onto MPSoC platforms under ST scheduling is studied when considering transition delay during mode transitions. In [11], 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 paper, 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.

3 BACKGROUND

In Section 3.3, we provide an overview of the MADF model. Since each mode in the MADF model is represented as a CSDF graph, the CSDF model is introduce in Section 3.1. Then, the notion of the K-periodic schedules [7] for CSDF graphs is briefly explained in Section 3.2, which covers the wide range of schedules considered in this paper.

3.1 Cyclo-Static Data Flow (CSDF)

A CSDF graph [5] is defined as a directed graph $G = (\mathcal{A}, \mathcal{E})$ which consists of a set of actors $\mathcal{A}$ that communicate with each other through a set of edges $\mathcal{E}$. Actors represent computation and edges represent the communication using FIFO channels to transfer data tokens between actors. Each actor $A_i \in \mathcal{A}$ may consume/produce a varied but predefined number of data tokens in its consecutive executions, called consumption/production sequence. It has been proven in [5] that a valid static schedule of a CSDF graph can be generated at design-time if the graph is consistent and live. A CSDF graph is said to be consistent if a non-trivial solution exists for the repetition vector $\vec{q} = [q_1, q_2, \ldots, q_{|\mathcal{A}|}]^T \in \mathbb{N}^{|\mathcal{A}|}$. An entry $q_i \in \vec{q}$ indicates the number of invocations of actor $A_i \in \mathcal{A}$ in one graph iteration of graph G. Fig. 1(b) show a CSDF graph (G_1^1) with $\vec{q} = [4, 2, 2, 2]^T$. The worst-case execution time (WCET) of each actor is shown below its name. For instance, actor A_5^1 has WCET of 2 time units and its data consumption sequence on FIFO channel E_3 is $[2, 0]$, i.e., consuming 2 and 0 token(s) from E_3 in its every two consecutive invocations, respectively.

3.2 K-Periodic Schedules (K-PS)

In [7], 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 [12] 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 [6] is a special case of 1-PS in which the actors are converted to RT tasks to enable the application of classical HRT scheduling algorithms [13], e.g., EDF, to streaming applications modeled as CSDF graphs. In general, the K-PS notion covers a wide set of schedules ranging between 1-PS and ST schedules. In all K-PS, discussed above, the duration needed by the graph to complete one iteration is called *iteration period* and denoted by H. In addition, the time distance between the starting times of the source (input) actor and the sink (output) actor is called *iteration latency* and denoted by L. For instance, a K-PS of G_1^1 is shown in Fig. 2(a), with $H^1 = 8$ and $L^1 = 10$ time units.

3.3 Mode-Aware Data Flow (MADF)

MADF [1] is an adaptive MoC which can capture multiple application modes associated with an adaptive streaming application, where each individual mode is represented as a CSDF graph [5]. Here, we explain the MADF intuitively by an example. The MADF graph G_1 of an adaptive streaming application with two different modes is shown in Fig. 1(a). This graph consists of 5 computation actors A_1 to A_5 that communicate data over FIFO channels E_1 to E_6. Also, there is an extra actor A_c which controls the switching between modes through control FIFO channels E_{11}, E_{22}, E_{33} E_{44}, and E_{55} at run-time. Each FIFO channel contains a production and a consumption pattern, and some of these production and consumption patterns are parameterized. Having different values of parameters and WCET of the actors determine different modes. For example, to specify the consumption pattern with variable length on a FIFO channel in graph G_1, the parameterized notation $[x[y]]$ is used to represent a sequence of x elements with integer value y, e.g., $[2[1]] = [1, 1]$ and $[1[2]] = [2]$. Assume in this particular example that the parameter vector $[p_1, p_2, p_4, p_5, p_6]$ can take only two values $[0, 2, 0, 2, 0]$ and $[1, 1, 1, 1, 1]$. Then, A_c can switch the application between two corresponding modes SI^1 and SI^2 by setting the parameter vector to the first value and the second value,

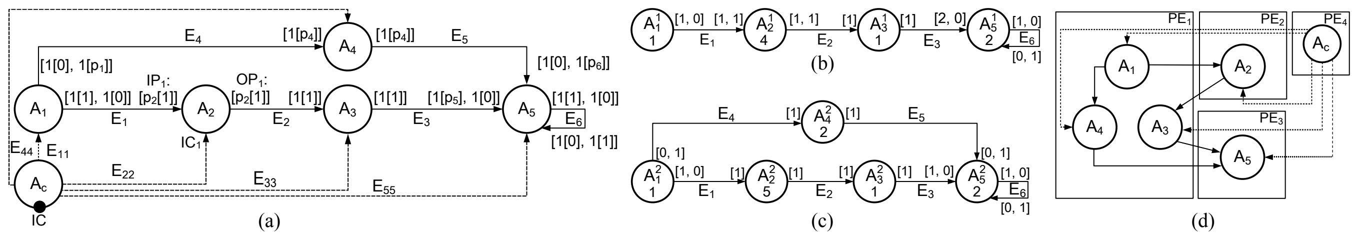


Figure 1: (a) An example of MADF graph (G_1), (b) CSDF graph G_1^1 of mode SI^1, and (c) CSDF graph G_1^2 of mode SI^2, (d) mapping of G_1 on 4 PEs.

respectively, at run-time. Fig. 1(b) and (c) show the corresponding CSDF graphs of modes SI^1 and SI^2.

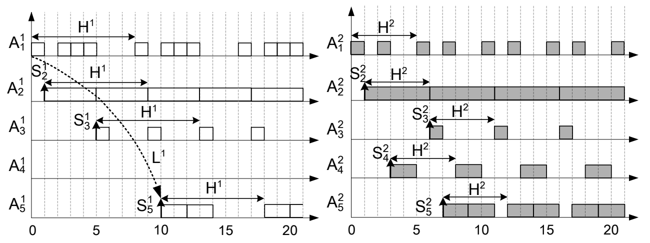
As an important part of the operational semantics of MADF, MADF employs the MOO transition protocol [1] when switching an application's mode by receiving a mode change request (MCR) from the external environment via the `IC` port of actor A_c. To resolve the timing interference of modes (in terms of overlapping iteration periods H) during mode transitions, this protocol delays the starting time of the new mode by the offset $x^{o\to n}$, computed as follows:

$$x^{o\to n} = \begin{cases} \max\limits_{A_i\in\mathcal{A}^o\cap\mathcal{A}^n} (S_i^o - S_i^n) & \text{if } \max\limits_{A_i\in\mathcal{A}^o\cap\mathcal{A}^n} (S_i^o - S_i^n) > 0 \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

where S_i^o and S_i^n are the starting times of actor A_i in modes SI^o and SI^n, i.e., the old and the new modes, respectively. For instance, consider the K-periodic schedule of modes SI^1 and SI^2 shown in Fig. 2 (a) and (b), respectively. Then, the offsets $x^{1\to2}$ and $x^{2\to1}$ for mode transitions from SI^1 to SI^2 and vice versa, computed using Eq. (1), are 3 and 1, respectively. An execution of G_1 with two mode transitions and the computed offsets is illustrated in Fig. 3(a), in which, as explained in Section 1, the iteration latency L of the K-periodic schedule of the modes, in Fig. 2 (a) and (b), are preserved during mode transitions. When multiple actors are mapped on the same PE, the PE can be potentially overloaded during mode transitions due to simultaneous execution of actors from different modes. Therefore, a larger offset may be needed to delay the starting time of the new mode during a mode transition in order to avoid processor overloading. In [1], this offset, represented with $\delta^{o\to n}$, is computed under the SPS schedule [6]. To quantify the responsiveness of a transition protocol, a metric, called *transition delay* and denoted by $\Delta^{o\to n}$, is also introduced in [1] and calculated as $\Delta^{o\to n} = \sigma_{snk}^{o\to n} - t_{\text{MCR}}$, where $\sigma_{snk}^{o\to n}$ is the earliest starting time of the sink actor in the new mode SI^n and t_{MCR} is the time when the MCR occurred. In Fig. 3(a), we can compute the transition delay for MCR1 occurred at time $t_{\text{MCR1}} = 1$ as $\Delta^{2\to1} = 16 - 1 = 15$ time units.

4 EXTENSION OF THE MOO TRANSITION PROTOCOL

As the SPS schedule has the notion of a task utilization, by converting the actors in a CSDF graph to RT tasks, the aforementioned offset $\delta^{o\to n}$ in [1] is computed by making the total utilization of the RT tasks mapped on each PE during mode transition instants to not exceed the PE capacity. However, since the K-periodic schedules, considered in this paper, have no notion of a task utilization, the offset $\delta^{o\to n}$ for any K-PS cannot be computed as in [1]. Therefore, in this section, we extend the MOO transition protocol to compute such an offset for any K-PS. In fact, to avoid the PE 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 PE. For instance, Fig. 1(d) shows an example of a mapping of all actors of G_1 to four PEs. Considering the execution of G_1 in Fig. 3(a), the schedule interferences on PE$_1$ happen during time periods $[6, 11]$ and $[25, 27]$ for mode transition from



(a) Mode SI^1 in Fig. 1(b) (b) Mode SI^2 in Fig. 1(c)

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

SI^2 to SI^1 and vice versa, respectively, while no schedule interference happens on PE$_2$ and PE$_3$. Obviously, to resolve the schedule interferences on PE$_1$, the earliest starting 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 PE$_1$ are $11 - 6 = 5$ and $27 - 25 = 2$ time units, respectively, thereby resolving the schedule interferences on PE$_1$, as shown in Fig. 3(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 PEs when the new mode SI^n starts immediately after the source actor of the old mode SI^o completes its last iteration, as follows:

$$\delta^{o\to n} = \max\{x^{o\to n}, \max_{\substack{\forall\Psi_i^o\in\Psi^o\wedge\Psi_i^n\in\Psi^n \\ \Psi_i^o\neq\emptyset\wedge\Psi_i^n\neq\emptyset}} (\max_{A_j^o\in\Psi_i^o} S_j^o - \min_{A_k^n\in\Psi_i^n} S_k^n)\} \quad (2)$$

where $\Psi = \{\Psi_1, \ldots, \Psi_m\}$ is m-partition of all actors on m number of PEs, i.e., Ψ_i^o and Ψ_i^n are the sets of actors mapped on the i-th PE (PE$_i$) in the old mode SI^o and the new mode SI^n, respectively. For instance, consider the mapping of G_1 on the four PEs, shown in Fig. 1(d), and the K-PS of modes SI^1 and SI^2 given in Fig. 2 (a) and (b), respectively. The offset $\delta^{1\to2}$ of the mode transition from SI^1 to SI^2 on each PE is computed using Eq. (2) as follows: (PE$_1$) $S_3^1 - S_1^2 = 5 - 0 = 5$, (PE$_2$) $S_2^1 - S_2^2 = 1 - 1 = 0$, and (PE$_3$) $S_5^1 - S_5^2 = 10 - 7 = 3$, thereby resulting in the offset $\delta^{1\to2} = \max(3, \max(5, 0, 3)) = 5$ for the starting time of mode SI^2, as shown in Fig. 3(b). Similarly, the offset $\delta^{2\to1}$ of the mode transition from SI^2 to SI^1 on each PE is computed using Eq. (2) as follows: (PE$_1$) $S_3^2 - S_1^1 = 6$, (PE$_2$) $S_2^2 - S_2^1 = 0$, and (PE$_3$) $S_5^2 - S_5^1 = -3$, and $\delta^{2\to1} = \max(1, \max(6, 0, -3)) = 6$.

5 IMPLEMENTATION AND EXECUTION APPROACH FOR MADF

In this section, we first present our generic parallel implementation and execution approach (Section 5.1) for an application modeled as an MADF. Then, in Section 5.2, we demonstrate our approach on LITMUS$^{\text{RT}}$ [8].

5.1 Generic parallel implementation and execution approach

In this section, we will explaine our approach by an illustrative example. Consider the MADF graph G_1 shown Fig. 1(a). Our implementation consists of three main components: 1) (normal) actors,

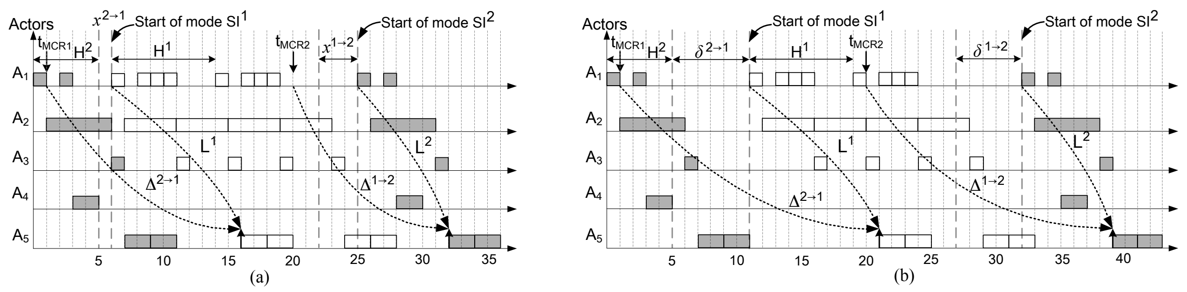


Figure 3: Execution of G_1 with two mode transitions under (a) the MOO protocol, and (b) the extended MOO protocol with the mapping shown in Fig. 1(d).

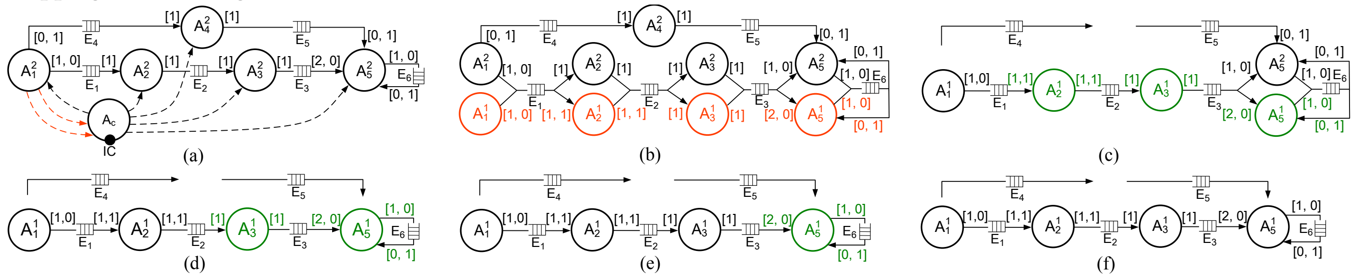


Figure 4: Mode transition of G_1 from mode SI^2 to mode SI^1 (from (a) to (f)). The control actor and the control edges are omitted in figures (b) to (f) to avoid cluttering.

2) a control actor, and 3) FIFO channels. We implement the actors as separate threads and the FIFO channels as circular buffers [14] 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 3.2. 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 an 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 by using a different task mapping 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 Fig. 3(b), is shown in Fig. 4 and has the following sequence - Fig. 4(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 PE, through the control FIFO channels (the dashed arrows in Fig. 4(a)). In our approach, two extra FIFO channels, shown in the red color in Fig. 4(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; Fig. 4(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 Fig. 4(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; Fig. 4(c): Next, when the thread of the source actor A_1^2 is terminated at time instant 5 (according to Fig. 3(b)), which is notified to control thread A_c as well, the control thread signals the suspended threads to be released synchronously $\delta^{2\to1} = 6$ time units later at time instant 11 (according to Fig. 3(b)). At this stage, a mixture of threads in both modes may be running on PEs. 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; Fig. 4(d)-(f): Since the actors have different starting time in the new mode SI^1, as shown in Fig. 3(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.

5.2 Demonstration of our approach on LITMUS$^{\text{RT}}$

In this section, we demonstrate how to realize our implementation and execution approach on LITMUS$^{\text{RT}}$ [8] as one of the existing 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 1 and 2, respectively, in which the bolded primitives belong to LITMUS$^{\text{RT}}$. Note that, any other RT operating system which has similar primitives, e.g., FreeRTOS [15], 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 1, the RT parameters of an actor, e.g., actor A_2 of graph G_1 in Fig. 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 PE 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$^{\text{RT}}$ kernel for validation, in Line 8, in which if it is verified, the thread is admitted as a RT task in LITMUS$^{\text{RT}}$, in Line 9. In Line 10, the RT

```
 1 void 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.PE_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       if(i == 1) WRITE(& graph_iteration, threadInfo.OC_iter);
22       if(threadInfo.mode == 1){ // Do action according to the task's mode
23         READ(& in1, threadInfo.IP_1);
24         task_function(& in1, & out1);
25         WRITE(& out1, threadInfo.OP_1);
26       }/* Actions according to the other modes */{ . . . }
27       if(i%threadInfo.K == 0) sleep_next_period();
28     }
29     graph_iteration += 1; } }
```

Listing 1: C++ code of actor A_2

task is suspended, in order to synchronize the starting time of the tasks, until signaled by the *control actor* to begin its execution. Next, the task enters to a `while loop` in Lines 12-29, in which iterates infinitely. At the beginning of each graph iteration, the current time instant is captured and stored in 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 14-28. In Line 15, the task sleeps until reaching the starting 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 27, 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 back to the ready queue at the starting 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 22-26. 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 21 and 18 highlighted with the red color, respectively, which are needed by the control thread, as explained in Section 5.1.

In Listing 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 2-6. In Line 7, the interface with the LITMUS^{RT} kernel is initialized. In Lines 10-19, 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 20 and 21, 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 26, 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 24, 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 27. 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

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

Listing 2: C++ code of control actor A_c

28-30. 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 28. 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 29, and $\texttt{t}_{\texttt{OV}}$, in Line 30, 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 31 to notify them when they have to be terminated.

6 CASE STUDY

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 [1] 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. We perform this case study on the ARM big.LITTLE architecture [16], including a quad-core Cortex A15 (big) cluster and a quad-core Cortex A7 (LITTLE) cluster, available on the Odroid-XU4 platform [17]. The Odroid-XU4 runs Ubuntu 14.04.1 LTS along with LITMUS^{RT} version 2014.2.

In this case study, we take a real-life adaptive streaming application from the StreamIT benchmark suit [18], 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 Fig. 5, 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 $dl \in \{8, 16, 32, 64\}$. Mode SI^8 ($dl = 8$) requires the least amount of computation at the cost of the worst voice encoding quality among all DFT lengths. Mode SI^{64} ($dl = 64$) 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 PEs, at run-time.

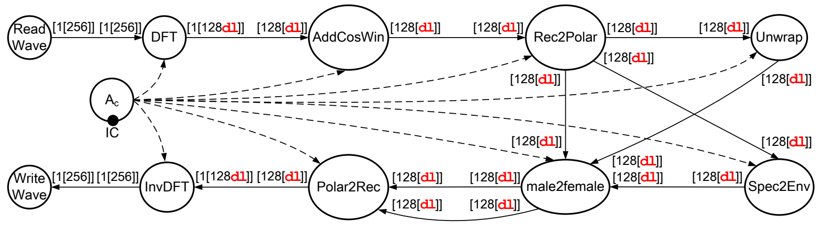


Figure 5: MADF graph of the Vocoder application.

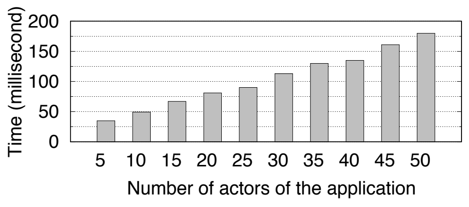


Figure 6: The execution time of control actor A_c.

Table 1: Performance results of each individual mode of Vocoder.

Mode	Analysis [1]		Implementation and execution		Number/Type of PE
	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

Table 2: Performance results for all mode transitions of Vocoder (in ms).

Transition (SI^o to SI^n)	Analysis [1] $\Delta^{o\to n}_{min}$	Analysis [1] $\Delta^{o\to n}_{max}$	Implementation and execution $\Delta^{o\to n}$
$SI^8 \to SI^{64}$	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

We measured the WCET of the actors in Fig. 5 in the four modes on both big and LITTLE PEs. 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 starting time of the actors under any K-PS to be executed by LITMUS^{RT}. Table 1 shows the performance results of each individual mode under the ST schedule, which is a particular case of K-PS explained in Section 3.2. 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/type of PEs 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 starting times of the actors as in the analysis model. Based on the results, shown in Table 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 [1], the computed minimum and maximum transition delays are shown in columns 2-3 of Table 2, respectively. By using our implementation and execution approach, however, the measured transition delay depends on the occurrence time of an 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 2 shows the measured transition delay for each transition with a random occurrence time of the 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 10 to 21 of Listing 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 Fig. 6. 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.

7 CONCLUSION

In this paper, 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.

REFERENCES

[1] J. T. Zhai et al. Modeling, analysis, and hard real-time scheduling of adaptive streaming applications. *IEEE TCAD*, 2018.
[2] O. Moreira. Temporal analysis and scheduling of hard real-time radios running on a multi-processor. *ser. PHD Thesis, TU Eindhoven*, 2012.
[3] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, 2010.
[4] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE TC*, 1987.
[5] G. Bilsen et al. Cycle-static dataflow. *IEEE Trans. Signal Process.*, 1996.
[6] M. Bamakhrama and T. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *DAES*, 17(2):221–249, 2013.
[7] B. Bodin et al. Optimal and fast throughput evaluation of csdf. In *DAC*, 2016.
[8] J. M. Calandrino et al. Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS*, 2006.
[9] R. Van Kampenhout et al. A scenario-aware dataflow programming model. In *DSD*, 2015.
[10] R. van Kampenhout et al. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *DATE*, 2017.
[11] H. Jung et al. Multiprocessor scheduling of a multi-mode dataflow graph considering mode transition delay. *ACM TODAES*, 2017.
[12] S. Stuijk et al. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Comput.*, 2008.
[13] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 2011.
[14] S. Bhattacharyya and E. Lee. Memory management for dataflow programming of multirate signal processing algorithms. *IEEE Trans. Signal Process.*, 1994.
[15] FreeRTOS. *http://www.freertos.org/*.
[16] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011.
[17] Odroid. https://www.hardkernel.com.
[18] M. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.