Efficient Synchronization Methods for LET-based Applications on a Multi-Processor System on Chip

Gabriela Breaban∗, Sander Stuijk∗, Kees Goossens∗†
*Eindhoven University of Technology, The Netherlands
†Topic Embedded Products, The Netherlands

Abstract—Distributed control applications cover a wide range of areas such as automotive, avionics, and automation. The Logical Execution Time (LET) Model of Computation (MoC) was proposed as a formal method to describe the functional and timing behavior of such applications. However, modern Multi-Processor Systems on Chip (MPSoC) do not have a shared notion of time between processors, due to their use of Globally Asynchronous Locally Synchronous (GALS) architecture. In this paper we propose two methods (based on FIFO channels and barriers) to implement time and data synchronization on a MPSoC. While a barrier synchronizes the execution flows of tasks at predefined points in their executions, a FIFO is an asynchronous data communication method between two tasks. First, they are used to implement LET applications. Next, we show how dataflow applications and mixed LET-dataflow applications are supported too. We implemented both methods on a MPSoC prototyped on a FPGA, and show that the data synchronization outperforms the related work by 67% in terms of software overhead.

I. INTRODUCTION

Distributed control applications with real-time constraints consist of sensors, actuators and a set of computation tasks. While a traditional distributed system is composed of several computation nodes implemented on separate chips and communicating over a network (e.g. CAN, Ethernet), a Multi-Processor System on Chip (MPSoC) is also distributed since the processors communicate via a Network on Chip (NoC).

Model-based design [1], [2] introduced a higher level of abstraction in the systems design flow by using a MoC to describe the application behavior prior to its implementation. LET is a MoC for control applications that abstracts from the physical execution time of application tasks, which is hardware dependent, and instead characterizes the tasks relative to the instants when the inputs are read and the outputs are written [3]. The time between the instant when the inputs are read and the instant when outputs are written is called logical execution time. It is fixed and independent of any physical implementation. This requirement makes the timing and functional behavior of the application portable and reproducible, a property which is called time safety.

Several programming models are build based on the LET paradigm. Giotto [4] requires the LET to be ensured for all application tasks. PTIDES [5] extends the Discrete Event (DE) MoC to add the notion of real time at I/O boundaries.

Following the model-based design methodology, every implementation of an application has to preserve the semantics of the chosen MoC, that is, the behavior of the implemented application has to be consistent with the behavior specified by the MoC. The LET semantics require every implementation to be time safe. A time safe implementation requires a global notion of time and deterministic inter-task communication. Achieving a global notion of time poses significant challenges for GALS MPSoCs, that often feature Dynamic Voltage and Frequency Scaling (DVFS) and experience clock drift. Next to this, deterministic communication demands that the data accessed by multiple parties is handled in a safe manner that ensures its consistency and timely access. Time and data synchronization are usually addressed separately. Global time is obtained via clock synchronization [6], [7], which scales for large distributed systems, but incurs hardware and/or software costs for each processor. As for consistent data communication, the methods reported in literature for multi-processor systems include lock-based [8] methods and transactional memory [9] that typically require hardware support.

A. Contribution

In this paper we propose two software-only methods that implement time and data synchronization for a LET-based application running on a MPSoC. Both methods accommodate both LET and Dataflow semantics and can support up- and downsampling scenarios. In this way, the LET semantics can

![Design Flow for LET-Dataflow applications](image-url)
be ensured for the (I/O) tasks that require strict timing guarantees, while the other tasks, such as intermediate computation tasks can be started and finished when the data is ready. Our methods target a single GALS multi-processor platform on which the processors communicate via shared memory using a NoC. Figure 1 summarizes and positions the proposed methods in the design flow context. We evaluate both methods by conducting experiments on a FPGA platform.

B. Related work

The LET paradigm was not addressed for multi-core systems until recently. Kluge et al. present in [10] an OS extension that supports the LET communication semantics, while the time synchronization is achieved via the NoC. Our contribution differs since we address time synchronization at SW level on one hand, thus without any HW cost, and, on the other hand, we support a more flexible realization of LET that allows tasks to start or stop based on data.

PTIDES [5] is an actor-based event-triggered MoC that enforces the LET semantics only for sensors and actuators, while intermediate computation actors are executed based on the DE semantics. It was integrated in the Ptolemy simulation framework that provides also code generation for PTIDES models. It resembles our work as it combines different MoC semantics (event-triggered and LET) from the specification down to the implementation. However, our methods combine the LET and Dataflow semantics and offer a rather lightweight, not yet automated implementation.

For time synchronization, both SW [11] and HW [12] implementations of clock synchronization have been reported in the literature. The precision of the synchronization depends on the precision of the timestamps. One disadvantage is that the implementation can be subject to unpredictable message latencies [13]. Time synchronization is obtained in our methods via time-predictable communication between a selected time-aware processor and the client processors. Our methods are purely SW based and, as expected, have a coarser granularity.

The time-triggered paradigm was originally conceived for the time-triggered architecture [14]. The implementation of this paradigm on a multi-processor system needs to address data consistency for the shared memory. The lock-based and wait-free mechanisms [15] proposed in literature solve this problem for a set of tasks scheduled based on static priorities. However, the capability to preserve the LET semantics is not addressed. Our methods, in turn, preserve both the LET semantics and the dataflow, depending on the characterization of each task.

The integration of timing requirements into the Dataflow MoC has been previously addressed in works such as [16], [17]. Moreira et al. [16] define a maximum production period for each actor and propose a scheduling strategy that improves the processor usage while meeting the maximum period constraints per actor. However, all actors are started based solely on the presence of data while the absolute notion of time is not explicitly considered, as in our method. In [17] the authors extend the Synchronous Dataflow (SDF) model to accommodate for a callback-based architecture and minimize the jitter of the produced output. Both works target multimedia applications that have throughput-centric requirements rather then deadline-centric, as in LET.

II. BACKGROUND

While the LET formalism is suitable for describing applications that access periodically I/O devices such as sensors and actuators and have fixed periods and deadlines, the static dataflow formalism introduced by Lee and Messerschmitt [18] and its derivations accommodate streaming applications in which the tasks are triggered by the data arrival and for which the real-time requirements are centered on the average throughput, rather then fixed periods/deadlines. A dataflow graph consists of a set of actors connected by arcs. An actor models a computation process and an arc models communication between processes. The execution of an actor is called firing. An actor fires as soon as it is enabled, that is, when the amount of data tokens present on each of its input arcs is at least equal to the consumption rate of the arc. Then the actor will run for a finite amount of time and at the end it will produce on each output arc an amount of tokens equal to the arc production rate. In the Cyclo Static Dataflow (CSDF) variant, the actor firings follow a repetitive cycle composed of a number of phases, each one characterized by its own rate and execution time. For a more detailed explanation of the dataflow semantics, we refer the reader to [18] and [19].

Imposing either of these two formalisms can be unnecessarily strict since there are applications that exhibit mixed data and time-triggered characteristics (e.g. data-intensive control, image-based control [20]). This motivates us to find a task characterization that combines both formalisms.

We will describe the properties of the input application tasks based on the running example shown in Figure 2 that we also use for experimental evaluation. Due to space limitations, the following description is informal.

A control application typically consists of sensors, actuators and a set of software tasks that process the sensors readings and/or compute the actuator updates. Our example contains two sensors, s1 and s2, two actuators, a1 and a2 and five
tasks, $t_1$ to $t_5$. The application requirements determine the 
period with which the time-driven tasks are activated and the 
deadlines by which they have to finish execution. A data-driven 
task can perform an intermediate computation when it receives 
the inputs from an upstream task and then produce the outputs 
as soon as it finishes. We call this a dataflow task. Hence the 
start and stop condition of a task can be either time or data 
driven. We account for all possible combinations between start 
and stop conditions. Table I summarizes the task classification. 
In the example, the time-driven start/stop condition is shown 
by a solid bar, while the lack of a bar denotes a data-driven 
start/stop condition. All four tasks types are included and also 
the corresponding task dependencies: the arrows labelled ‘td’ 
denote a dependency between a task finishing on time and a 
task starting on data, the ‘dd’ ones denote the data to data 
dependency, the ‘tt’ arrows a time to time dependency and the 
dd’ ones a data to time dependency. An additional concept 
shown in the figure is the task state, denoted by an arrow 
between two successive task invocations and it is illustrated 
for tasks $t_1$ and $t_5$.

<table>
<thead>
<tr>
<th>Start Condition</th>
<th>Stop Condition</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Pure Time-triggered (ta)</td>
<td>Periodic task w/o deadline</td>
</tr>
<tr>
<td>Data</td>
<td>Data-driven task w/ deadline</td>
<td>Dataflow task</td>
</tr>
</tbody>
</table>

The proposed synchronization methods take as input a 
control application described by a task characterization as the 
one presented in the previous section. Our methods only target 
a single GALS MPSOC platform for which the clocks have 
been classified in terms of drift and we select the processor 
with the most reliable clock as the time-aware processor with 
which all the processors that need to access time, synchronize 
using blocking data communication. The advantage is that we 
don’t need to run a clock synchronization algorithm on each 
processor. The disadvantage is that such a technique can only 
be applied if the communication overhead is relatively low, to 
be able to obtain a reasonable synchronization accuracy. This 
is possible on a multi-processor platform, but for an actual 
distributed system consisting of a higher number of computa-
tion nodes connected by a network, the communication delays 
will be considerably higher and a distributed clock distribution 
algorithm would be a better option.

The time-predictable inter-processor communication is en-
abled by the Time Division Multiplexing (TDM) NoC [21] 
which provides low communication jitter. It achieves this 
by using pipelined TDM schedule for the packets, logical 
TDM slot synchronization between the communicating nodes, 
a fixed 2 cycles delay per router and a small TDM slot size 
of 2 words. These design choices determine a bounded and 
stable communication latency.

We devise two methods for implementing time and data 
synchronization. Both methods assume a MPSOC with a 
single trusted clock source that is distributed using either 
barriers or FIFOs. Both data- and time-triggered execution 
of tasks is supported. We assume the existence of a feasible 
static mapping and static order schedule for the input application 
and then derive the design parameters for each method that 
satisfy the timing requirements. The static order schedule is 
chosen due to its simplicity and composability.

Figures 3 and 4 show the result of applying the methods to 
the running example. It is important to note that our methods 
are meant to be applied to the application as described by the 
MoC , prior to the selection of a HW platform, mapping and 
schedule. Thus the two figures do not incorporate such details.

A. The Dataflow Method

This method models the input application as a dataflow 
graph that is afterwards implemented in software according to 
the dataflow semantics. The LET semantics are modeled in 
the graph and are thus implicitly realized by the implementation.

We derive a CSDF application wrapper for the input ap-
plication based on the tasks characterisation presented in 
Section II, using the algorithm 1. For each actor in the graph, 
the production rates on every output arc and the consumption 
rates on every input arc, respectively, are equal for all CSDF 
phases. We use the following notations in the algorithm: $H$ 
is the hyper-period duration, $T_i$ is the period duration for task 
t, $r_p$ and $r_c$ represent production and consumption rates.

The timeStep actor and the $t_i$ Delay actors in the algorithm 
above are time-aware actors and are mapped to the time-aware
Algorithm 1 Create CSDF application

```plaintext
for all task $t_i$ in application A do
    Create a CSDF actor $t_i$ with $\frac{H}{T_{t_i}}$ phases
    if $t_i$ has state then
        Add self arc to actor $t_i$ with 1 initial token
        $\tau_{step} \leftarrow \min(T_{t_i})$
        $m \leftarrow \frac{H}{\tau_{step}}$
        Create a CSDF actor $timeStep$ with $m$ phases
        Add state variable $s$ with initial value $-\tau_{step}$
        $s \leftarrow finishTime(timeStep)$
        Add a self arc to actor $timeStep$ for variable $s$
        $executionTime(timeStep) \leftarrow s + \tau_{step}$
    for all task $t_i$ in application A do
        $p \leftarrow numberOfPhases(t_i)$
        if $t_i$ has a time-driven start condition then
            Create a CSDF actor $t_iWrDrv$ with $p$ phases
            Add one arc from $t_iRdDrv$ to $t_i$ with $r_p = r_c = 1$
            Add one arc from $timeStep$ to $t_iRdDrv$ with $r_p = 1$,
            $r_c = \frac{T_{t_i}}{\tau_{step}}$ and $r_c = 1$ initial tokens
        if $t_i$ has a time-driven stop condition then
            Create a CSDF actor $t_iDelay$ with $p$ phases
            Create a CSDF actor $t_iDelay$ with $p$ phases
            Add an arc from $t_i$ to $t_iWrDrv$ with $r_p r_c = 1$
            Add an arc from $t_iDelay$ to $t_iWrDrv$ with $r_p r_c = 1$
            Add an arc from $timeStep$ to $t_iDelay$ with $r_p r_c = 1$
            Add an arc from $timeStep$ to $t_iDelay$ with $r_p r_c = 1$
            add 1 initial token
        if $T_{t_i} == T_{t_j}$ then
            Add an arc from $src$ to $dst$ with $r_p r_c = 1$
        if $T_{t_i} == n \cdot T_{t_j}$ then
            Add a CSDF actor $Downsample$ with 1 phase
            Add an arc from $src$ to $Downsample$ with $r_p = 1$, $r_c = n$
            Add an arc from $Downsample$ to $dst$ with $r_p = r_c = 1$
            and $n$ initial tokens
    if $T_{t_i} == n \cdot T_{t_j}$ then
        Add a CSDF actor $Upsample$ with 1 phase
        Add an arc from $src$ to $Upsample$ with $r_p = r_c = 1$
        Add an arc from $Upsample$ to $dst$ with $r_p = n$, $r_c = 1$
        and $n$ initial tokens
```

Figure 3 shows the resulting CSDF graph. In the graph, task $t_5$, with a time triggered start condition, doesn’t have a read driver. Since a driver is responsible for converting data from a sensor/actuator format to a task format, it can be dropped when the task doesn’t communicate with such a device.

B. The Barrier Method

This method uses a barrier synchronization library to achieve the time and data synchronization for the input control application. The barrier synchronizes a predefined number of clients in a blocking manner: each client updates its location in the barrier data structure and blocks until all the other clients update their locations. We use two barrier types: data and time barriers. Figure 4 shows the barriers for the running example.

A data barrier is used to synchronize two tasks that have a data to data dependency, that is, the sending task has a data-driven stop condition and the receiving one has a data-driven start condition. The sending task writes the output data, then updates the barrier and blocks. The receiving task first updates the barrier and blocks waiting for the sending task, and then reads the input data. In our example, there are two data to data dependencies, from $t_3$ to $t_2$ and from $t_2$ to $t_1$ and the data barriers for them are labelled ‘data’ in Figure 4.

A time barrier is used to implement a time-driven start or stop condition. In this case one barrier client will be the time-aware processor and the other client will be the task. For a time-driven start condition, the time-aware processor will

processor. Then a feasible static order schedule is derived. The execution time of each $t_iDelay$ actor will be determined by its position in the schedule, the communication time towards the corresponding $t_iWrDrv$ and the execution time of $t_iWrDrv$, such that the token sent to $t_iWrDrv$ will enable the driver actor to fire and finish execution just before its deadline.

One limitation of this method is that the Downsampling actor can only model the cases in which out of $n$ produced samples, all but the last are dropped.
wait for the predefined task period, update the barrier and block. The processor running the task should have finished any previous activity and be waiting for the time barrier update (this is ensured by the schedule). This can be visualized for tasks $t_5$, $t_4$ and $t_5$ in Figure 4. For a time-driven stop condition, the barrier update time is anticipated based on the worst case bounds for the communication time and the write driver execution time such that the output is made available as close as possible to the deadline. In the figure, tasks $t_1$ and $t_3$ have a time-driven stop condition.

For downsampling and upsampling of data, we allocate a set of consecutive barriers that ensure the synchronization between the sender and the receiver. For downsampling, the barriers will synchronize with the sending task on each produced sample via a data barrier followed by either a data or a time barrier to synchronize with the receiver on the right sample depending on its start condition. Figure 4 illustrates this for tasks $t_4$ and $t_5$. Upsampling is based on the same principle and it will execute the reverse operations but it is not shown in the figure for space reasons.

### C. Analytical Evaluation

As a quantitative evaluation, we observe that the total number of connections required for the time and data synchronization is almost identical. Both methods require one connection per timed start/stop condition and per data synchronization between two tasks running at the same frequency. The only difference is for the up-sampling $1:n$ and down-sampling $n:1$ where the dataflow method uses one dedicated actor with two communication FIFOs and the barrier uses $n+1$ barriers. Thus the barrier has worse scalability than dataflow for up- and downsampling since the number of barriers is proportional to the number of dropped or reused samples.

The quality of the time synchronization is given by the accuracy and the jitter. The synchronization accuracy is given by the maximum absolute difference between each period/deadline time and the actual time when the corresponding task reads its inputs or writes its outputs. It is determined by the total number of tasks that share the same start/stop time and the communication times. The time-aware processor updates the barriers for each start/stop time sequentially, thus the accuracy will be given by either the latest task read time or the earliest task write time. This can be seen in Figure 4 where at time 0, three barriers are executed sequentially and the last barrier in the sequence, the one for task $t_5$ will cause the largest time difference with respect to the start period time. For dataflow, the synchronization will be performed by the TimeStep actor and the execution order will be reflected by the order in which each output FIFO is written. The jitter represents the maximum timing variation of each task actual read/write times with respect to the reference period/deadline.

### IV. Experiments

We implemented the 2 methods for the running example on our multi-processor platform and evaluated them in terms of SW overhead and memory consumption. For this, we synthesized a platform consisting of three processor tiles all running at 100 MHz on a Xilinx ML605 FPGA.

The first required steps for both methods are finding a mapping and then a static order schedule per processor. To allow for comparison, the chosen mapping and schedules are identical for both methods. More specifically, we decided to map tasks $t_1$ and $t_3$ to the tile 1, tasks $t_2$, $t_4$ and $t_3$ to tile 2 and the timing-aware barriers or actors to tile 3, which is selected to be the time reference. The task drivers are mapped on the same tile as the tasks. Tile 3 will run the timeStep, $t1Delay$ and $t3Delay$ for the Dataflow method and the time barriers as well as the Down-sampling barriers for the barrier method. Although the mapping of the tasks $t_1$ to $t_3$ on different processors is not efficient, given the task dependencies, it’s purpose is to illustrate and evaluate the use of the data barrier.

The chosen schedules are: $\{t_3WrDrv, t_3, t_1WrDrv\}$ on tile 1, $\{t_2RdDrv, t_5, t_4, t_2RdDrv, t_4, t_5WrDrv\}$ on tile 2. As the Dataflow method uses two actors for upsampling and downsampling, they need to be added to the schedule on tile 2 and the chosen positions are: Downsample after $t_4$ and Upsample after $t_5WrDrv$. On tile 3 the timeStep actor or the start time barrier will run twice, at the beginning and the middle of the hyper-period, followed by the deadline delay actor or barrier for $t_1$ and then $t_3$, before the end of the hyperperiod. Since we use a synthetic example, we assign to each task and driver a constant execution time.

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>EXPERIMENTAL RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronization</td>
<td>Barrier Method</td>
</tr>
<tr>
<td>SW Overhead (cycles)</td>
<td>min</td>
</tr>
<tr>
<td>Period Sync</td>
<td>223</td>
</tr>
<tr>
<td>Deadline Sync</td>
<td>223</td>
</tr>
<tr>
<td>Data Sync</td>
<td>362</td>
</tr>
</tbody>
</table>

Table II shows the SW overhead for each method in clock cycles. For the barrier method, we see that the obtained overheads are in a close range. The maximum value for the period synchronization is a caused by the schedule on tile 2,
where $t_5$ runs after $t_4$ and it is delayed by the driver
execution time which was set to 2048 cycles. Similarly, the
variation of the data synchronization barrier is determined by
the specific position in the schedule of the communicating
tasks. For the dataflow method, the overhead is higher due to
the fact that each actor is executed according to the model
of execution (that implements the dataflow semantics) and
handling each communication FIFO involves several checks
and updates of the administration objects in addition to the
actual transfer of data. The large maximum value for the
period synchronization comes from the same scenario as for
the other method, the difference being caused by the higher
data communication times, which are in the range of 1500 to
2500 cycles for dataflow and 200 to 400 cycles for the barrier.

The worst-case cost of around 400 cycles (4 μs) per task for
our most efficient method, the barrier, shows that, in terms of
scalability, we can synchronize maximum 10-25 concurrent
tasks per time-aware processor with a accuracy of <100 μs.
This is also a maximum feasible task load for a typical multi-
processor platform comprising at most 10 processors. The
accuracy could be improved, when possible, by adding more
time-aware processors having synchronized clocks.

The total data memory consumption is 112 bytes for the
barrier method and 2728 bytes for the dataflow method. The
increased amount for dataflow comes from the size of the
administration structure, that is 96 bytes per FIFO. For both
methods, the data was mapped on the receiver’s local memory
to minimize the access overhead.

From the measurements we observe that the dataflow
method has a higher cost both at run-time and in terms of
memory, making it mostly suitable for applications that include
more data-intensive components than time-triggered ones. The
barrier method has a low cost, thus it fits a wider range of
applications that include time-triggered as well as data-intensive
components. The accuracy of both methods will depend on
the specific application timing requirements: the more tasks
requiring synchronization at the same time instant, the worse
the accuracy. The accuracy, as defined in section III-C is 3307
cycles (33.07 μs) for the barrier method and 6914 cycles
(69.14 μs) for the dataflow method. The jitter for the barrier
method is 123 cycles (1.23 μs) and for the dataflow method is
404 cycles (4.04 μs). Note that the reported accuracy for the
PTP SW implementation [12] is between several milliseconds
to several hundreds of microseconds. While the frequency of
running the PTP synchronization algorithm is normally set at
2 s, within our methods the synchronization is performed for
each time-driven start/stop condition.

The closest related work with respect to our methods is [10].
Their OS extension implements the LET semantics for inter-
task communication while the time synchronization is obtained
via the NoC. The reported SW overhead for this extension ranges from around 1100 (kernel executed from scratch-pad
memory) to 12000 clock cycles (kernel executed from global
memory) and it can be compared with our data synchronization
that has a lower overhead ranging between 362 and 3836
cycles, leading to a reduction of 67%. The measurements for
the related work are obtained on a Patmos 4-core processor
and the main sources of the overhead are the cache misses and
the off-chip memory accesses. In addition, the authors in [10]
do not offer a characteristic of jitter.

V. Conclusions

In this article we presented two methods for data and
time synchronization for a LET -based control application
on a multi-processor platform. Both methods relax the LET
semantics by using dataflow semantics when a strict time-
triggered start/stop condition is not required. We evaluate the
methods using a synthetic example implemented on the FPGA.
The comparison with the related work shows a 67% reduction
of the SW overhead for the data synchronization.

This work was partially funded by projects CATRENE
ARTEMIS 621429 EMC2, 621353 DEWI, 621439 AL-
MARVI.

 REFERENCES

[1] K. Balasubramanian et al., “Developing Applications Using Model-
Driven Design Environments,” IEEE Computer Society, vol. 39, no. 2,
2006.
[2] I. Sander et al., “System modeling and transformational design re-
finement in ForSyDe (formal design system).” IEEE Transactions on
Computer-Aided Design of Integrated Circuits and Systems, vol. 23,
Springer Berlin Heidelberg, 2012, pp. 103–120.
Time Embedded Systems,” EECs Department, University of California,
synchronization system based on FPGA,” in ICICIP, 2014.
[15] G. Han et al., “Experimental Evaluation and Selection of Data Consis-
tency Mechanisms for Hard Real-Time Applications on Multicore
[16] O. Moreira et al., “Supporting the Logical Execution Time model on a
Heterogeneous Multiprocessor,” in RSN. ACM, 2016.
Dataflows for Multimedia Systems,” SPIE The International Society for
and Fast Connection Set-Up,” Computers IEEE Transactions on, vol. 63,
2014.