Communication between Nested Loop Programs via Circular Buffers in an Embedded Multiprocessor System *

Tjerk Bijlsma[1], Marco Bekooij[2], Pierre Jansen[1], and Gerard Smit[1]
[1]University of Twente, The Netherlands, [2]NXP Semiconductors Research, The Netherlands
t.bijlsma@utwente.nl

Abstract

Multimedia applications, executed by embedded multiprocessor systems, can in some cases be represented as task graphs, with the tasks containing nested loop programs. The nested loop programs communicate via arrays and can be executed on different processors. Typically an array can be communicated via a circular buffer with a capacity smaller than the array. For such buffers, the communicating nested loop programs have to synchronize and a sufficient buffer capacity needs to be computed. In a circular buffer we use a write and a read window to support rereading, out-of-order reading or writing, and skipping of locations. A cyclo static dataflow model is derived from the application and used to compute buffer capacities that guarantee deadlock free execution. Our case-study applies circular buffers in a Digital Audio Broadcasting channel decoder application, where the frequency deinterleaver reads according to a non-affine pseudo-random function. For this application, buffer capacities are calculated that guarantee deadlock free execution.

1 Introduction

Multimedia application typically show streaming behavior and can be represented as task graphs. For performance reasons the tasks, from the task graph, are typically executed on several processors in a embedded multiprocessor systems. In the multiprocessor system that we consider, a processing tile includes a processor and a scratch pad memory (SPM), as depicted in Figure 1. Processing tiles are connected via a network on chip (NoC) that provides lossless and in-order delivery of read and write operations.

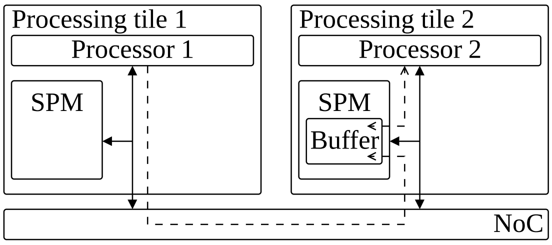


Figure 1. Multiprocessor system template

Throughout this paper we consider a task to contain a nested loop program (NLP). NLPs communicate via arrays, where one NLP writes in an array and the second NLP reads from it. A straightforward approach is to store this array in an SPM and synchronize on the entire array. In contrast, a buffer can be used that typically is smaller than the array. It is more likely that the smaller buffer fits into an SPM, but the buffer requires more synchronization and the computation of the buffer capacity is a complex problem.

If a task writes into two buffers, the capacity of one buffer may depend upon the read and write pattern in the other buffer. Choosing the buffer capacity too small, may result in a deadlock of the application. To compute sufficient buffer capacities, the read and write patterns in all arrays should be considered at once.

This paper presents an approach that uses circular buffers (CBs) for the communication between the NLPs of the tasks. In a CB, a read and a write window are used that support rereading, out-of-order reading or writing, and skipping of locations. An NLP is extended with a few statements for the synchronization. The capacities of the CBs are computed by considering the read and write patterns of all NLPs, such that deadlock free execution of the application is guaranteed. The presented approach does not require affine index expressions.

The extended NLP that writes in a CB, called the producer, uses a write window, whereas the extended NLP that reads from a CB, called the consumer, uses a read window. The consumer or producer has mutually exclusive access to the locations of the CB that are in its window. For both windows, the window size is derived by analyzing the sequence of locations accessed by the NLP in the array. The implementation of a window in a CB is by two pointers. An NLP is extended by adding a few conditional statements to update the pointers of a window, where the conditions are

---

simple, i.e. comparing a counter variable with a constant value. A cyclo static dataflow (CSDF) model [1] models the window usage of the NLPs. With the CSDF model, capacities for the CBs can be computed that guarantee deadlock free execution of the application.

In the case-study, we show how to arrange the communication in a Digital Audio Broadcasting (DAB) channel decoder via CBs. We show that our approach is applicable to the pseudo-random function used for reading by the frequency deinterleaver, which cannot be described with an affine index expression. Sufficient buffer capacities are computed, such that deadlock free execution of the DAB channel decoder is guaranteed.

This paper is organized as follows. In Section 2 related work is discussed. Section 3 discusses the characteristics of the applications that we consider. In Section 4, we explain our approach that realizes communication via CBs and calculates sufficient CB capacities. Section 5 discusses the case-study. In Section 6 we present the conclusions.

2 Related work

The methods presented in [4] and [5] use buffers for arrays that are accessed in one affine NLP that is executed on a single processor. In [4] for a read and a write access to an array, Integer Linear Programming is used to determine a window size that serves as the required capacity for the circular buffer. In [5] for a dependency between two accesses to an array, the window size is determined that serves as buffer capacity. In a cache that is used as an SPM, buffers are allocated for some of the windows. In comparison, our approach considers parallel execution of tasks and therefore uses two windows in a circular buffer.

With the method presented in [8] a Kahn Process Network (KPN) is derived from a parameterized affine NLP. In the KPN the communication between NLPs is arranged via first-in-first-out (FIFO) buffers. When the consuming NLP has to read a location in an array multiple times, or read the locations of an array out-of-order, the consumer stores the values in an additional buffer. Complex if-statements are used to write only the values to be read in a FIFO buffer. Instead of FIFO buffers we use CBs with windows. In a window we have random access and non-destructive reads, therefore there is no need to copy values in an additional buffer.

In [6] an approach is presented to use a read and a write window. A window supports reading locations multiple times, reading or writing the locations out-of-order, or skipping locations. No analysis to determine window or buffer sizes is proposed in [6]. Compared to their approach, we derive the read and write window size, given the access patterns of the NLPs and we determine sufficiently large buffer capacities such that deadlock free execution is guaranteed.

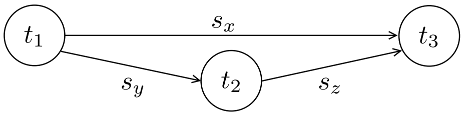
3 Target applications

Throughout this paper we assume that each application is represented by a weakly connected directed acyclic task graph $H = \{T, S, A, \alpha, \rho, \sigma, \theta\}$. In a weakly connected graph, for every pair of vertices x and y, there is a directed path from x to y or from y to x. The set of vertices is T. Each vertex $t_i \in T$ represents a task, where the functional behavior of a task is defined by an NLP. The set of arrays is A. Each array $a_j \in A$ is declared in an NLP. The set of directed edges is S. An edge $s_j = (t_h, t_i)$, with $s_j \in S$, is from task t_h to task t_i, with $t_h, t_i \in T$ and $t_h \neq t_i$. Each edge represents a buffer. In a buffer s_n the values of one corresponding array a_n are stored. The l^{th} access of task t_i in array a_j, accesses the array location with index $\alpha(t_i, a_j, l)$, with $\alpha : T \times A \times \mathbb{N} \to \mathbb{N}$. The function $\rho(t)$ is the total number of accesses performed during one execution of a task t, with $\rho : T \to \mathbb{N}$. The size, in number of locations, of the array a_j is given by $\sigma(a_j)$, with $\sigma : A \to \mathbb{N}$. The capacity of buffer s_j is $\theta(s_j)$ locations, with $\theta : S \to \mathbb{N}$.

The code of an NLP is single assignment code, this means that a location in an array is assigned a value at most once per execution of the task. We describe an NLP using the C-syntax. Figure 2 depicts a task graph together with the NLPs. To execute the application, the tasks of the directed acyclic task graph are topologically sorted. The tasks are executed an infinite number of times, according to the schedule that results from the topological sort. The NLP of a task contains statements to declare the arrays followed by nested for-loops. The arrays written and read in the NLPs are global variables and are declared in the NLPs that write them. The shorthand notation used for a for-loop is *for* $i : l : u$, where i is the iterator of the for-loop, l the lower bound and u the upper bound. The iterator is incremented with one after each iteration of the for-loop. In this paper we assume that the lower-bound and the upper-bound are constant values. The innermost for-loop contains the loop-body. The loop-body contains one or more statements. In a statement one or more arrays are read and only one array is written. In the loop-body an array a_i is either read or written once. The index of a location in an array is determined with an index expression that can only have the iterators of the nested for-loops as variables. The index expression is not limited to be affine, but it has to produce the same sequence of indices for every execution of its task. The symbol $\sim$, used in some statements in Figure 2, denotes code fragments that are omitted for clarity.

Two tasks communicate by writing and reading in the same array a_i. The function α gives the consumption or production pattern of an array. For example, the production and consumption patterns in array a_y by task t_1 and t_2 from Figure 2 are:

l	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\alpha(t_1, a_y, l)$	1	0	3	2	5	4	7	6	9	8	11	10		
$\alpha(t_2, a_y, l)$	0	1	2	3	4	5	6	5	6	7	8	9	10	11



```
int X[12];
int Y[12];                int Z[14];
for i0:0:5                for j0:0:1          for k0:0:3
 for i1:0:1 {              for j1:0:6 {        for k1:0:2 {
   X[2i0-i1+1] = ~;          Z[7j0+j1] =         ~ = X[11-3k0-k1];
   Y[2i0-i1+1] = ~;             Y[5j0+j1];       ~ = Z[3k0+k1];
 }                          }                  }
```

Figure 2. Task graph with its NLPs

Three interesting access patterns are identified: out-of-order access, multiplicity and skipping [8, 6].

The out-of-order access pattern occurs if a producer or consumer accesses non-consecutive locations in an array. An example of out-of-order production is present in the access pattern for the NLP of t_1 in array a_y. In this pattern the location with index one is written in iteration zero, $\alpha(t_1, a_y, 0) = 1$, and the location with index zero is written in iteration one, $\alpha(t_1, a_y, 1) = 0$.

The multiplicity pattern occurs if a location is accessed more than once. Only the access pattern for the consumption may contain multiplicity, since we consider single assignment code. For example, the NLP of t_2 in array a_y consumes the location with index five twice, $\alpha(t_2, a_y, 5) = \alpha(t_2, a_y, 7) = 5$.

If the skipping pattern occurs, locations from an array are not accessed. The NLP of t_3, from Figure 2, skips the locations with index 12 and 13 in array a_z.

4 Communication via circular buffers

This section describes the usage of CBs by the NLPs of tasks. In the CBs windows are introduced that support rereading, out-of-order reading or writing, and skipping of locations. Next, the NLPs are extended with a few statements for synchronization. Finally, a method to compute the capacities of the CBs is presented, such that deadlock free execution of the task graph can be guaranteed.

4.1 Communication between tasks

The communication between two tasks is performed via a buffer. This section focuses at the location of the buffer, the granularity of the communication between the tasks, the synchronization between the two tasks, and the buffer used for the communication.

By locating the buffer in the SPM of the processing tile that is executing the consuming task, as depicted in Figure 1, this task always has local memory accesses. The low latency access to the SPM minimizes the stall time of the processor that executes the consuming task. The producing task sends its values to the buffer in the SPM of the processing tile that is executing the consuming task. This is called sender initiated communication [3].

The communication between tasks is performed at a word-level granularity. For most access patterns a buffer smaller than a complete array is sufficient. Smaller buffers require less memory in an SPM.

The communication and synchronization are performed via a shared memory, which is in this case the SPM of the processing tile that executes the consuming task. In case of shared memory communication, a memory consistency model defines the ordering in which the read and write operations performed by a processor become visible to other processors. This ordering between read and write operations defines how the synchronization between tasks executed on the processors can be implemented.

We use the streaming memory consistency model [2], where acquire and release statements provide the synchronization. In this memory consistency model a location in a buffer has to be acquired before it is accessed and released after it has been written or read for the last time. The key advantage of the streaming memory consistency model is that it supports posted writes. A posted write is a write operation that allows the producer to continue execution, instead of waiting until the value to be written is stored into the memory. A posted write is sent via the NoC to the buffer, where the write operations can be pipelined in the NoC, such that a next write request can be accepted by the NoC before the previous write requests have been stored in the memory.

A *circular buffer* (CB) is used for the communication between tasks, because it can handle the multiplicity, skipping, and out-of-order access pattern. A CB can be implemented with a write and a read pointer. The synchronization of the write and read pointer, between the producing and the consuming task requires no additional hardware, because the C-HEAP [7] protocol is used.

In a CB, the producer has random write access in all locations between the write and the read pointer of the CB, whereas the consumer has random read access in all locations between the read and the write pointer. The producer makes values available to the consumer by increasing the write pointer. The consumer increases its read pointer when the value at the location of the read pointer will not be read anymore. The pointers may not overtake each other. Typically both pointers start at the same location and the first pointer to be increased is the write pointer. When a pointer reaches the end of the CB, it wraps around to the begin of the CB. Hence, both the producer and the consumer have mutually exclusive access to their part of the buffer.

The values of the arrays in Figure 2 will be communicated via CBs in combination with streaming memory consistency. This combination requires that a location in a CB has to be acquired before it is accessed and released later on. An NLP has to be extended with acquire and release statements and statements to update either the read or write pointer of the CB.

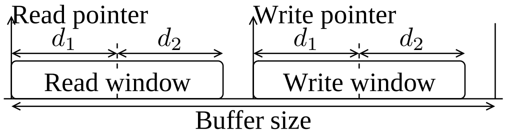


Figure 3. The read and write window in a CB

We introduce a *read window* for the consumer and a *write window* for the producer. As depicted in Figure 3, both windows contain a sequence of consecutive locations. For the write window only the index of the first location has to be stored. The index of the last location in the write window equals the write pointer. An acquire statement for n locations, adds the n locations immediately in front of the first location of the write window, to the window. When a release statement for n locations is executed for the write window, the n locations consecutive to the write pointer are released from the window, followed by incrementing the write pointer for these n locations. For the read window the acquire and release statements are executed in the same way, they add locations to and remove locations from the read window and increment the read pointer.

We try to minimize the computational overhead as caused by adding statements to an NLP, in order to communicate an array via a CB. A conditional acquire statement for one location is added at the beginning of the loop-body and a conditional release statement for one location at the end of the loop-body. The conditions are kept simple, i.e. they compare a counter variable with a constant value. Starting from the first iteration, the conditional acquire statement acquires one location per iteration, until for each locations of the array an acquire has been performed in the CB. Since every iteration acquires one location, it may be necessary to acquire n locations before the nested for-loops, to guarantee that each iteration of the loop-body can access its location. Furthermore, possibly an initial number of iterations of the loop-body should not execute the release statement for one location, to guarantee that a location is not released before it has been accessed for the last time. The conditional acquire of one location at the beginning of a loop-body and the conditional release of one location at the end of a loop-body, make the windows *sliding windows*.

4.2 Window size

The read or write window size, is determined using the access pattern of the NLP. To guarantee that the location to be accessed is acquired, it may be necessary to acquire an initial number of locations before the first read or write access of an NLP. The number of initially acquired locations by the NLP of t_i in the CB s_j is called the *lead-in* $d_1(t_i, s_j)$, with $d_1 : T \times S \rightarrow \mathbb{N}$. In the following, the location with index 0 is the first location to be acquired in a CB.

The intuition on how to determine the lead-in $d_1(t_i, s_j)$, for the array of CB s_y, that is written by NLP t_1, from Figure 2, is given in Figure 4. The upper sequence gives the indices of the locations acquired by the producer and the lower sequence provides the indices of the locations that are written. By shifting the lower sequence right we guarantee that every location is acquired before it is written. In Figure 4, a lead-in of one location is found.

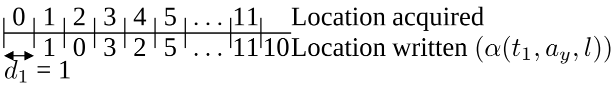


Figure 4. Lead-in $d_1(t_1, s_y)$, for Figure 2

It is possible to give an expression for the lead-in in a CB. Let t_i be the considered task, s_j the CB and l the iteration.

Lemma 1 *A lead-in* $d_1(t_i, s_j) = \max_l(\alpha(t_i, a_j, l) - l)$, *with* $0 \leq l < \rho(t_i)$, *is the minimal number of locations acquired before the nested for-loops, such that if each iteration of the loop-body acquires one location, it is ensured that every location is acquired before it is read or written.*

Proof: Iteration r, with $0 \leq r < \rho(t_i)$, accesses location $\alpha(t_i, a_j, r)$, so at least $\alpha(t_i, a_j, r) + 1$ locations should be acquired in the CB. Note that each iteration of the loop-body accesses one location in the CB s_j. Each iteration of the loop-body also acquires one location, so initially $\alpha(t_i, a_j, r) + 1 - (r + 1)$ locations should be acquired, if $\alpha(t_i, a_j, r) > r$. In case $\alpha(t_i, a_j, r) \leq r$, location $\alpha(t_i, a_j, r)$ is already acquired in iteration r of the loop-body. To guarantee that in each iteration of the loop-body the accessed location is acquired, the minimal and sufficient number of initial acquired locations $d_1(t_i, s_j)$ is found by $\max_l(\alpha(t_i, a_j, l) - l)$, with $0 \leq l < \rho(t_i)$. $\square$

No location should be released from a window until for each remaining iteration of the loop-body it can be guaranteed that the released location will not be read or written anymore in a succeeding iteration. The number of iterations without a release for the NLP of t_i in a CB s_j is called the *lead-out* $d_2(t_i, s_j)$, with $d_2 : T \times S \rightarrow \mathbb{N}$.

The intuition to determine the lead-out $d_2(t_i, s_j)$, for t_1 in CB s_y, is given in Figure 5. The upper sequence gives the indices of the written locations and the lower sequence provides the indices of the released locations. The lower sequence is shifted right to guarantee that no location is released before it is written. In Figure 5 a lead-out of one iteration is found.

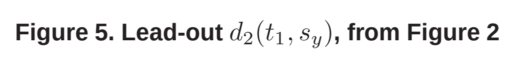


Figure 5. Lead-out $d_2(t_1, s_y)$, from Figure 2

As for the lead-in, an expression to determine the lead-out can be given. Let t_i be the considered task, s_j the CB, and l the iteration of the NLP.

Lemma 2 *A lead-out* $d_2(t_i, s_j) = \max_l(l - \alpha(t_i, a_j, l))$, *with* $0 \leq l < \rho(t_i)$, *is the minimal number of initial iterations of the loop-body of the NLP of* t_i *without a release, such that in the remaining iterations one location can be released, without releasing a location before it is written or read for the last time.*

Proof: After an initial number of iterations $d_2(t_i, s_j)$ each iteration of the loop-body releases one location in CB s_j. To guarantee that the location with index $\alpha(t_i, a_j, r)$ is still acquired in iteration r, with $0 \leq r < \rho(t_i)$, at least the first $r - \alpha(t_i, a_j, r)$ iterations should release no location, if $r \geq \alpha(t_i, a_j, r)$. In case $r < \alpha(t_i, a_j, r)$, if at the end of every iteration of the loop-body one location is released, the location with index $\alpha(t_i, a_j, r)$ is released later than the r^{th} iteration. To make sure that in each iteration the read or written location is not released yet, minimally the first $\max_l(l - \alpha(t_i, a_j, l))$ iterations, with $0 \leq l < \rho(t_i)$, of the loop-body should not release a location. □

It is possible that a negative lead-out is found, this happens if $\forall l : l - \alpha(t_i, s_j, l) < 0$. In this case one or more locations at the beginning of the array are skipped. In this case the window should be slided over the skipped locations before executing the nested for-loops. Then the skipped locations are not included in the window size.

The lead-in, lead-out, and the location acquired at the beginning of the loop-body of the NLP of t_i in CB s_j, build up a window with size $w(t_i, s_j) = \min(d_1(t_i, s_j) + d_2(t_i, s_j) + 1, \sigma(a_j))$. The window contains at most the whole array, therefore if $d_1(t_i, s_j) + d_2(t_i, s_j) + 1 > \sigma(a_j)$ the window has the size of the array $\sigma(a_j)$, as depicted in Table 1 for the window of task t_3 in CB s_x.

Theorem 1 *For the NLP of* t_i *in CB* s_j *with a lead-in* $d_1(t_i, s_j)$ *and a lead-out* $d_2(t_i, s_j)$, *a window size* $w(t_i, s_j) = \min(d_1(t_i, s_j) + d_2(t_i, s_j) + 1, \sigma(a_j))$ *guarantees that the accessed location is in the window.*

Proof: Lemmas 1 and 2 state that the locations to be accessed will be acquired and not yet released. Lemma 1 states that the lead-in is determined such that $d_1(t_i, s_j) \geq \alpha(t_i, a_j, l) - l$, this is equal to $\alpha(t_i, a_j, l) \leq d_1(t_i, s_j) + l$. Lemma 2 states that the lead-out is determined such that $d_2(t_i, s_j) \geq l - \alpha(t_i, a_j, l)$, this is equal to $l - d_2(t_i, s_j) \leq \alpha(t_i, a_j, l)$. The combination of both ensures that the accessed location is always in the window, $l - d_2(t_i, s_j) \leq \alpha(t_i, a_j, l) \leq l + d_1(t_i, s_j)$. The window size is defined by the difference between the upper bound and the lower bound plus one, $l + d_1(t_i, s_j) - (l - d_2(t_i, s_j)) + 1$. One additional location is required, because the conditional acquire is performed at the beginning of the loop-body and the release at the end. Since at most all locations of the array are acquired in CB s_j, the maximum window size is $\sigma(a_j)$. □

NLP	t_1		t_2		t_3	
CB	s_y	s_x	s_y	s_z	s_z	s_x
d_1	1	1	0	0	0	11
d_2	1	1	2	0	0	11
w	3	3	3	1	1	12
σ	12	12	12	14	14	12

Table 1. The lead-in, lead-out, window size, and array size for the NLPs in Figure 2

Table 1 contains the lead-in $d_1(t_i, s_j)$, lead-out $d_2(t_i, s_j)$, window size $w(t_i, s_j)$, and array size $\sigma(a_j)$ for the NLPs of the tasks in Figure 2. The table depicts that the windows for the CBs s_y and s_z are smaller than the size of their array. Due to the out-of-order reading pattern of the NLP of t_3, a read window of 12 locations is required for CB s_x. For this communication pattern, the whole array a_j is stored in CB s_x.

4.3 Extending the NLPs

This section discusses the statements that need to be added to the NLPs of the tasks, to let them communicate via CBs using sliding windows.

Figure 7 depicts the task graph from Figure 2, where the NLPs are extended to communicate arrays via CBs, using sliding windows. For every accessed array a_j by task t_i, the NLP of t_i is extended according to the template shown in Figure 6. In the template the structure is presented to extend the C-code of the NLP, with acquire and release statements in the initial phase, the processing phase, and the final phase. In the *initial phase* lead-in $d_1(t_i, s_j)$ locations are acquired, to guarantee that during the processing phase in each iteration the location to be accessed contains a valid value in case of reading, or can be overwritten in case of writing. During the *processing phase*, at the beginning of the loop-body an if-statement checks, whether there is a location in the CB left to acquire. At the end of the loop-body, an if-statement checks whether lead-out $d_2(t_i, s_j)$ iterations have passed, to determine if a location in the CB can be released. In the *final phase*, the remaining acquired locations in the CB are released. For every accessed array a_j, in total $\sigma(a_j)$ locations are acquired and released in the corresponding CB s_j, during the initial, processing, and final phase. Within the initial phase, processing phase, and final phase, the C-code can be partitioned in sections that can perform an acquire or release statement in a CB s_j, we will call them synchronization sections. In the template, the dummy counter variable p contains the index number of the current synchronization section, that can perform an acquire or release operation in CB s_j.

```
int $p$ = 1;
acquire($\zeta(t, s_j)$,$s_j$);
$p$++;
for (i:1:$\hat{\rho}(t)$){
  if (i $> \hat{\rho}(t) + d_2(t, s_j)$){
    acquire(1,$s_j$);                                Initial phase
    release(1,$s_j$);
  }
  $p$++;
}

int $c$ = 1;
(nested for-loops){
  if ($c \leq \sigma(a_j) - d_1(t, s_j)$)
    acquire(1,$s_j$);
  (statements)                                       Processing phase
  if ($c > d_2(t, s_j)$)
    release(1,$s_j$);
  $c$++; $p$++;
}

for (i:1:$\check{\rho}(t)$){
  if (i $\leq \sigma(a_j) - \rho(t) - d_1(t, s_j)$){
    acquire(1,$s_j$);
    release(1,$s_j$);
  }                                                  Final phase
  $p$++;
}
release($\chi(t, s_j)$,$s_j$);
```

Figure 6. Template to extend an NLP to communicate an array a_j via a CB s_j

During the *initial phase* of the NLP of t, $d_1(t, s_j)$ locations are acquired, in every CB s_j adjacent to task t. The template, as depicted in Figure 6, contains an initial phase for a CB s_j. First an acquire statement acquires the whole or a part of the lead-in $d_1(t, s_j)$ in the CB s_j. The number of acquired locations is given by $\zeta(t, s_j)$, with:

$$\zeta(t, s_j) = \begin{cases} d_1(t, s_j) + d_2(t, s_j), \textbf{if } d_2(t, s_j) < 0 \\ d_1(t, s_j), \textbf{otherwise} \end{cases}$$

The for-loop succeeding the acquire statement, acquires and releases $-d_2(t, s_j)$ locations in CB s_j, if the lead-out $d_2(t, s_j)$ is negative. The lead-out can be negative when the first locations in an array are skipped.

In the extended NLP of task t, the for-loop in the initial phase may perform acquires and releases for multiple CBs. The number of iterations $\hat{\rho}(t)$ of the initial for-loop in the NLP of a task t, is the number of acquire and release operations required by the CB with the smallest lead-out, with $\hat{\rho} : T \rightarrow \mathbb{N}$. Where $\hat{\rho}(t_i)$ of task t_i is given by:

$$\hat{\rho}(t_i) = \max\{-d_2(t_i, s_j) | \\ s_j = (t_k, t_l) \in S \wedge (t_k = t_i \vee t_l = t_i)\}$$

During the *processing phase*, for all adjacent CBs of a task t, at the beginning of an iteration at most one location is acquired and at the end of an iteration at most one location is released. In the template in Figure 6, a counter variable c is used in the loop-body of the NLP of a task t, to count the number of performed iterations of the loop-body. At the beginning of the loop-body, an if-statement is inserted that checks whether an acquire should be performed. For CB s_j this if-statement checks if the iteration counter is smaller or equal to $\sigma(a_j) - d_1(t, s_j)$, this is the case if there are locations left in the CB to acquire. The access to the array a_j is replaced by a read or a write operations in CB s_j. After performing the statements in the loop-body, an if-statement checks if a location should be released in the CB s_j. For CB s_j, one location can be released if the iteration counter is larger than $d_2(t, s_j)$, so after the number of iterations for the lead-out.

In the *final phase* of the NLP of a task t, the remaining locations in the adjacent CBs are released. In the template in Figure 6 the final phase starts with a for-loop. When not all locations in the CB s_j have been acquired, the for-loop performs $\sigma(a_j) - \rho(t) - d_1(t, s_j)$ iterations that acquire and release a location in the CB. The last release statement in the final phase, releases the locations that are still acquired. The number of locations to be released is given by the function $\chi(t, s_j)$, with:

$$\chi(t, s_j) = \begin{cases} \sigma(a_j) - (\rho(t) - d_2(t, s_j)), \\ \quad \textbf{if } \sigma(a_j) \leq \rho(t) + d_1(t, s_j) \\ \sigma(a_j) - (\rho(t) - d_2(t, s_j)) \\ \quad - (\sigma(a_j) - \rho(t) - d_1(t, s_j)), \textbf{otherwise} \end{cases}$$

The number of locations in CB s_j left to release depends upon the number of release statements executed in the for-loop of the final phase. If the for-loop did not release locations in CB s_j, so $\sigma(a_j) \leq \rho(t) + d_1(t, s_j)$, there are $\sigma(a_j) - \rho(t) + d_2(t, s_j)$ locations left to release. Otherwise the number of releases performed by the for-loop have to be subtracted from this number.

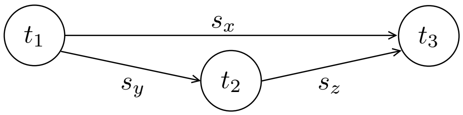
The for-loop in the final phase may contain acquire and release statements for multiple CBs. Therefore the number of iterations $\check{\rho}(t)$ performed by the for-loop in the final phase of the NLP of task t, is the maximum number of acquire and release operations to be performed for a CB, with $\check{\rho} : T \rightarrow \mathbb{N}$. Where $\check{\rho}(t_i)$ for the NLP of task t_i is given by:

$$\check{\rho}(t_i) = \max\{\sigma(a_j) - \rho(t_i) - d_1(t_i, s_j) | \\ s_j = (t_k, t_l) \in S \wedge t_k = t_i \vee t_l = t_i\}$$

Figure 7 depicts the task graph with the extended NLPs from Figure 2. The access pattern of the NLP of task t_3 in CB s_z contains skipping, therefore not all locations are acquired during the processing phase, i.e. $\sigma(a_z) - \rho(t_3) - d_1(t_3, s_z) = 14 - 12 - 0 > 0$. The for-loop in the final phase performs $\check{\rho}(t_3) = 2$ iterations that acquire and release one location per iteration in the CB s_z.

4.4 Buffer capacity per edge

In this section the buffer capacities are determined per edge, followed by an example that illustrates that the found



```
while(1){
 int c1 = 1;
 acquire(1,s_x);
 acquire(1,s_y);

 for i_0:0:5
  for i_1:0:1 {
   if(c1 ≤ 11)
     acquire(1,s_x);
   if(c1 ≤ 11)
     acquire(1,s_y);
   write(2i_0-i_1+1,s_x,∼);
   write(2i_0-i_1+1,s_y,∼);
   if(c1 > 1 )
     release(1,s_x);
   if(c1 > 1)
     release(1,s_y);
   c1++;
  }

 release(1,s_x);
 release(1,s_y);
}
```

```
while(1){
 int c2 = 1;

 for j_0:0:1
  for j_1:0:6 {
   if(c2 ≤ 12)
     acquire(1,s_y);
   acquire(1,s_z);
   write(7j_0+j_1,s_z,
      read(5j_0+j_1,s_y));
   if(c2 > 2)
     release(1,s_y);
   release(1,s_z);
   c2++;
  }

 release(1,s_y);
}
```

```
while(1){
 int c3 = 1;
 acquire(11,s_x);

 for k_0:0:3
  for k_1:0:2 {
   if(c3 ≤ 1)
     acquire(1,s_x);
   acquire(1,s_z);
   ∼ = read(11-3k_0-k_1,s_x);
   ∼ = read(3k_0+k_1,s_z);
   if(c3 > 11)
     release(1,s_x);
   release(1,s_z);
   c3++;
  }

 for k_3:1:2 {
  acquire(1,s_z);
  release(1,s_z);
 }
 release(11,s_x);}
```

Figure 7. Task graph from Figure 2, extended to communicate via CBs

buffer capacities can be insufficient to guarantee deadlock free execution of the task graph. In section 4.5 an alternative approach is described where all CBs are considered at once.

The minimal buffer capacity of a CB $s_j = (t_h, t_i)$ in isolation $\iota(s_j)$, is:

$$\iota(s_j) = w(t_h, s_j) + w(t_i, s_j) - 1$$

The found buffer capacity is minimal for a CB in isolation. Consider the NLPs of the two tasks t_h and t_i that communicate via array a_j, corresponding to CB $s_j = (t_h, t_i)$. Task t_h requires a window with $w(t_h, s_j)$ locations and task t_i a window with $w(t_i, s_j)$ locations. If the buffer contains $w(t_h, s_j) + w(t_i, s_j) - 1$ locations, either the NLP of t_h or t_i can execute its loop-body by acquiring one location, followed by reading or writing in the CB and releasing one location, next the other NLP can execute its loop-body.

The minimum per edge buffer capacities for the CBs in Figure 7 are $\iota(s_x) = 12$, $\iota(s_y) = 5$, and $\iota(s_z) = 1$. That these capacities are too small to guarantee deadlock free execution of the tasks, can be seen as follows. In the task graph, the NLP of t_3 can execute if 12 locations can be acquired in CB s_x. To provide these locations, the NLP of t_1 should perform 12 iterations. If the NLP of t_1 performs an iteration it acquires locations in both CB s_x and CB s_y. After four iterations of the NLP of t_1, the task cannot continue with writing in CB s_y before additional locations become available in CB s_y. The NLP of t_2 has to perform two iterations before it starts releasing locations in CB s_y. Since the determined buffer capacity of CB s_z is one, NLP t_2 can only acquire and release one location in CB s_z. The NLP of t_3 cannot perform an iteration, in which it acquires and releases the location in CB s_z, since it first has to acquire 12 locations in CB s_x. Since t_3 waits for t_1, t_1 waits for t_2, and t_2 waits for t_3, there is deadlock.

The deadlock is the consequence of the dependency between the acquire operations in CB s_z and the acquire operation CB s_x. Instead of computing the CB capacities per edge, the CB capacities should be calculated considering the dependencies between the acquire and release operations of all the NLPs at once. In the next section we present the derivation of a CSDF model from a task graph. This CSDF model is used to compute the buffer capacities that guarantee deadlock free execution of the task graph.

4.5 Buffer capacities for the task graph

We model the periodic acquire and release operations of the tasks in a task graph, in a cyclo static dataflow (CSDF) model [1, 9]. A CSDF model consists of a weakly directed graph $G = (V, E, \delta, \pi, \gamma, \phi)$, with V the set of actors and E the set of directed edges. An edge $e_{ij} = (v_i, v_j)$, with $e_{ij} \in E$, is from actor v_i to actor v_j, with $v_i, v_j \in V$. An edge represents an unbounded FIFO queue. Actors communicate tokens over edges. There are $\delta(e_{ij})$ initial tokens on an edge e_{ij}, with $\delta : E \rightarrow \mathbb{N}$. The period of an actor v_i contains $\phi(v_i)$ phases, with $\phi : V \rightarrow \mathbb{N}$. An actor is enabled if all its input edges contain the number of tokens that will be consumed during the next firing. At the moment actor v_j is fired for the c^{th} time, it atomically consumes $\gamma(e_{ij}, ((c-1) \mod \phi(v_j)) + 1)$ tokens from every input edge e_{ij}, with $\gamma : E \times \mathbb{N} \rightarrow \mathbb{N}$. On finishing firing c, actor v_j atomically produces $\pi(e_{ji}, ((c-1) \mod \phi(v_j)) + 1)$ tokens on its output edge e_{ji}.

A CSDF model can be constructed from a task graph with extended NLPs. Every task t_i in the task graph is modeled by an actor v_i. An edge $s_{ij} = (t_i, t_j)$ representing the buffer between two tasks, is modeled by an edge $e_{ij} = (v_i, v_j)$ and a back edge $e_{ji} = (v_j, v_i)$. Initially, edge e_{ji} contains $\delta(e_{ji})$ tokens, which is equal to the buffer capacity $\theta(s_{ij})$ of the corresponding CB s_{ij}. Edge e_{ij} contains no initial tokens. The template in Figure 6 contains the dummy counter variable p that represents the number of the synchronization section, in the NLP of a task. The number of phases $\phi(v_i)$ of an actor v_i is equal to the number of synchronization sections of the NLP of t_i, with $\phi(v_i) = \hat{\rho}(t_i) + 1 + \rho(t_i) + \check{\rho}(t_i) + 1$.

The production on and consumption from an edge adjacent to an actor v_i in the CSDF model, is derived from the number of acquired and released locations in the corresponding CB adjacent to task t_i. An acquire statement for n locations in a CB in the NLP of t_i, corresponds to the consumption of n tokens from an edge by an actor v_i in the CSDF model. The same holds for a release statement for n locations in a CB in the NLP of t_i that corresponds to the production of n tokens on an edge by actor v_i. For an edge pair e_{ij} and e_{ji} in the CSDF model that correspond to the CB s_h in the task graph, the total number of tokens consumed from e_{ji} or produced on edge e_{ij} during a period of actor v_i, equals the size of the array $\sigma(a_h)$ of CB s_h.

The number of consumed tokens in phase p by an actor v_i on an edge e_{ji} is derived from the acquire statement executed in synchronization section p in the CB s_h by the NLP of t_i, with e_{ji} either the edge or the back edge modeling CB s_h. For an actor v_i on edge e_{ji} in phase p, the number of consumed tokens $\gamma(e_{ji}, p)$ is given by:

$$\gamma(e_{ji}, p) = \begin{cases} \zeta(t_i, s_h), \textbf{if } p = 1 \\ 0, \textbf{if } 2 \leq p \leq \hat{\rho}(t_i) + 1 \wedge d_2(t_i, s_h) \geq 0 \\ 0, \textbf{if } 2 \leq p \leq \hat{\rho}(t_i) + d_2(t_i, s_h) + 1 \wedge d_2(t_i, s_h) < 0 \\ 1, \textbf{if } \hat{\rho}(t_i) + d_2(t_i, s_h) + 2 \leq p \leq \hat{\rho}(t_i) + 1 \\ \quad \wedge\, d_2(t_i, s_h) < 0 \\ 1, \textbf{if } \hat{\rho}(t_i) + 2 \leq p \leq \hat{\rho}(t_i) + \sigma(a_h) - d_1(t_i, s_h) + 1 \\ 0, \textbf{if } \hat{\rho}(t_i) + \sigma(s_h) - d_1(t_i, s_h) + 2 \leq p \leq \\ \quad \hat{\rho}(t_i) + \rho(t_i) + \check{\rho}(t_i) + 2 \end{cases} \tag{1}$$

The first four conditions in this function cover the synchronization sections in the initial phase of an NLP. The first phase of an actor consumes $\zeta(t_i, s_h)$ tokens. If the lead-out $d_2(t_i, s_h)$ of array a_h is positive, the phases of actor v_i corresponding to the iterations of the for-loop in the initial phase of the NLP of t_i consume no tokens. In case of a negative lead-out, the phases of the actor v_i that correspond to the last $-d_2(t_i, s_h)$ iterations of the for-loop in the initial phase of the NLP of t_i, consume one token. Condition five and six cover the synchronization sections in the processing phase and the final phase of the NLP of t_i. Actor v_i consumes one token from edge e_{ji} in the phases that correspond to the synchronization sections of the NLP of t_i that acquire one location in the CB s_h until all $\sigma(a_h)$ locations have been acquired. Remaining phases of actor v_i consume zero tokens.

The number of produced tokens by an actor v_i on an edge e_{ij} in phase p is derived from the release statements executed in synchronization section p in the CB s_h by NLP t_i, with e_{ij} either the edge or back edge modeling CB s_h. We make use of an additional function $\omega(t_i, s_h) = \sigma(a_h) - \rho(t_i) - d_1(t_i, s_h)$ that returns the number of values in array a_h that are left to be skipped, after performing the initial and processing phase of the NLP of t_i. For an actor v_i on an edge e_{ij} during phase p, the number of produced tokens $\pi(e_{ij}, p)$ is:

$$\pi(e_{ij}, p) = \begin{cases} 0, \textbf{if } 1 \leq p \leq \hat{\rho}(t_i) + d_2(t_i, s_h) + 1 \\ 1, \textbf{if } \hat{\rho}(t_i) + d_2(t_i, s_h) + 2 \leq p \leq \hat{\rho}(t_i) + \rho(t_i) + 1 \\ 1, \textbf{if } \hat{\rho}(t_i) + \rho(t_i) + 2 \leq p \leq \\ \quad \hat{\rho}(t_i) + \rho(t_i) + \omega(t_i, s_h) + 1 \wedge \omega(t_i, s_h) > 0 \\ 0, \textbf{if } \hat{\rho}(t_i) + \rho(t_i) + \omega(t_i, s_h) + 2 \leq p \leq \\ \quad \hat{\rho}(t_i) + \rho(t_i) + \check{\rho}(t_i) + 1 \wedge \omega(t_i, s_h) > 0 \\ 0, \textbf{if } \hat{\rho}(t_i) + \rho(t_i) + 2 \leq p \leq \\ \quad \hat{\rho}(t_i) + \rho(t_i) + \check{\rho}(t_i) + 1 \wedge \omega(t_i, s_h) \leq 0 \\ \chi(t_i, s_h), \textbf{if } p = \hat{\rho}(t_i) + \rho(t_i) + \check{\rho}(t_i) + 2 \end{cases} \tag{2}$$

In this equation, the first two conditions cover the synchronization sections of the initial and processing phase. The third, fourth, and fifth condition, correspond to the conditional release operation in the for-loop in the final phase of the NLP of t_i. If there are no locations left to acquire in CB s_h, zero tokens are produced in the phases corresponding to the for-loop of the final phase. The sixth condition produces $\chi(t_i, s_h)$ tokens, this corresponds to the number of locations to be released in the last statement of the final phase.

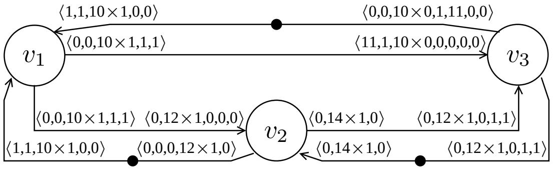


Figure 8. The CSDF model derived from the task-graph in Figure 7

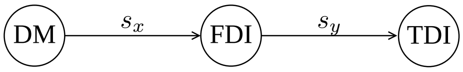
Figure 8 depicts the CSDF model derived from the task graph in Figure 7. The Equations 1 and 2 are used to construct the production and consumption sequences for the edges. For example for actor v_1, the production on edge e_{13} is $\langle \pi(e_{13}, 1), \ldots, \pi(e_{13}, \phi(v_1)) \rangle = \langle 0, 0, 10 \times 1, 1, 1 \rangle$. In this notation, 10×1 means that ten consecutive phases produce one token each. Furthermore, the edges with the black dot are the back edges that contain a number of initial available tokens.

To compute the required number of initial tokens on an edge to avoid deadlock in a CSDF model, a conservative approximation algorithm [9] is used. In [9] it is also shown that the computed number of initial tokens for the edges in a CSDF model, can be used as the buffer capacities for the buffers in the task graph, resulting in deadlock free execution of the task graph. The computational complexity of this algorithm is linear in the number of edges times the maximum number of phases of an actor in the CSDF model, i.e. $O(|V| + |E| \cdot \max_i(\phi(v_i)))$.

Sufficient initial tokens for the edges in the CSDF model in Figure 8 were computed in less than 0.1 seconds. According to this algorithm the number of initial tokens should be, $\delta(e_{31}) = 14$, $\delta(e_{21}) = 5$, and $\delta(e_{32}) = 13$. Therefore, deadlock free execution of the application is guaranteed by choosing the buffer capacities as $\theta(s_x) = 14$, $\theta(s_y) = 5$, and $\theta(s_z) = 13$.

5 Case-study

In this section, we compute the buffer capacities for a part of a task graph of a DAB channel decoder that operates in decoding mode I. Figure 9 depicts the part of the task graph with the corresponding NLPs. The demapper (DM) task contains the differential demodulator and the dequantizer. The DM writes a symbol, which consists of 2048

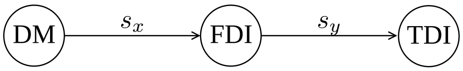


```
int x[2048];         int y[1536];
for i0:0:2047{       for j0:0:1535 {        for k0:0:1535 {
  x[i0] = ∼;           y[j0] = x[F(j0)];      ∼ = y[k0];
}                    }                      }
```

Figure 9. Task graph fragment of a DAB channel decoder, containing the demapper, the frequency deinterleaver, and the time deinterleaver

```
int F(int index){
  int counter=0,value=511;
  while (counter < index){
    value = ((value * 13) + 511)%2048;
    if ((value ≥ 256) ∧ (value ≠ 1024) ∧ (value ≤ 1792))
      counter++;
  }
  return value;
}
```

Figure 10. Pseudo-random function used for reading, by the frequency deinterleaver



```
while(1){                 while(1){                    while(1){
 for i0:0:2047{            int c2 = 1;                  for k0:0:1535 {
   acquire(1,s_x);         acquire(1713,s_x);             acquire(1,s_y);
   write(s_x,i0,∼);        for j0:0:1535 {                write(∼,∼,read(s_y,k0));
   release(1,s_x);           if (c2 ≤ 335)                release(1,s_y);
 }                             acquire(1,s_x);          }
}                            acquire(1,s_y);           }
                             write(s_y,j0,
                               read(s_x,F(j0)));
                             if (c2 > 1235)
                               release(1,s_x);
                             release(1,s_y);
                             c2++;
                           }
                           release(1748,s_x);
                          }
```

Figure 11. Task graph from Figure 9, extended to communicate via CBs

pairs of 4 bit metrics, in array a_x. The frequency deinterleaver (FDI) reads a symbol from array a_x. Index 1024 and the indices below 256 or above 1792 in array a_x are skipped by the FDI. The metric pairs read by the FDI are reordered according to the pseudo-random function *F(i)* in Figure 10. The FDI writes 1536 metric pairs in-order in array a_y. The time deinterleaver (TDI) reads the 1536 metric pairs in-order from array a_y.

Table 2 depicts the lead-in $d_1(t_i, s_j)$, lead-out $d_2(t_i, s_j)$, window size $w(t_i, s_j)$, and array size $\sigma(a_j)$ for the NLPs of the tasks in Figure 9. These results show that only the out-of-order consumption pattern of the FDI in array a_x requires a read window with the size of a symbol, which is 2048 locations.

The extended NLPs for the task graph in Figure 9, are depicted in Figure 11. The NLPs of the DM and TDI tasks, are extended with only one acquire and one release statement, since they have an in-order access pattern without the skipping and multiplicity pattern. Figure 12 depicts the CSDF model constructed from the extended task graph in Figure 11. The tasks DM, FDI, and TDI correspond to the actors v_d, v_f, and v_t, respectively. Computing the number of initial tokens for the back edges in the CSDF model, using the algorithm from [9], required 0.2 seconds and resulted in $\delta(e_{fd}) = 3464$ and $\delta(e_{tf}) = 388$. Therefore, deadlock free execution of the task graph in Figure 9 is guaranteed, when the capacity of CB s_x is $\theta(s_x) = 3464$ locations and the capacity of CB s_y is $\theta(s_y) = 388$ locations.

NLP	DM	FDI		TDI
CB	s_x	s_x	s_y	s_y
d_1	0	1713	0	0
d_2	0	1235	0	0
w	1	2048	1	1
σ	2048	2048	1536	1536

Table 2. The lead-in, lead-out, window size, and array size for the NLPs in Figure 9

6 Conclusions

We have presented an approach to arrange the communication between the nested loop programs (NLPs) of the tasks of an application via circular buffers (CBs).

In a CB, a read and a write window with random access are used that support rereading, out-of-order reading or writing and skipping of locations. The CB with windows handles the out-of-order and the multiplicity pattern without copying values into an additional buffer. The size of a window is derived from the sequence of accessed locations by an NLP in an array.

Each NLP is extended with only a few conditional acquire and release statements that guarantee mutually exclusive access in a window. A cyclo static dataflow (CSDF) model is derived from the task graph with these extended NLPs. Sufficient buffer capacities can be computed with the CSDF model, such that deadlock free execution can be

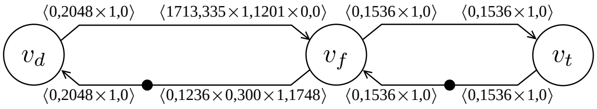


Figure 12. CSDF model of the task graph in Figure 11

guaranteed.

The approach is demonstrated on a fragment of a Digital Audio Broadcasting channel decoder that includes the frequency deinterleaver. The frequency deinterleaver uses a pseudo-random function for reading that cannot be described with an affine expression. We have extended the application to use CBs with windows, furthermore sufficient buffer capacities have been determined to guarantee deadlock free execution.

References

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44[2]:397–408, 1996.

[2] J. W. v. d. Brand and M. Bekooij. Streaming consistency: a model for efficient MPSoC design. In *Proc. Euromicro Symposium on Digital System Design*, pages 27–34, 2007.

[3] D. Culler, A. Gupta, and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[4] E. de Greef, F. Catthoor, and H. de Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Int'l Journal of Parallel Computing*, 23[12]:1811–1837, 1997.

[5] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Int'l Journal of Parallel and Distributed Computing*, 5[5]:587–616, 1988.

[6] K. Huang, D. Grünert, and L. Thiele. Windowed FIFOs for FPGA-based multiprocessor systems. In *Proc. Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 36–42, 2007.

[7] A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. In *Proc. Design Automation Conference (DAC)*, pages 233–270, 2002.

[8] A. Turjan, B. Kienhuis, and E. Deprettere. An integer linear programming approach to classify the communication in process networks. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 62–76, 2004.

[9] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 281–292, April 2007.