Inter-Task Communication via Overlapping Read and Write Windows for Deadlock-Free Execution of Cyclic Task Graphs

Tjerk Bijlsma\textsuperscript{1}, Marco J.G. Bekooij\textsuperscript{2}, and Gerard J.M. Smit\textsuperscript{1}
\textsuperscript{1}University of Twente, The Netherlands, \textsuperscript{2}NXP Semiconductors Research, The Netherlands

tjerk.bijlsma@utwente.nl, marco.bekooij@nxp.com, g.j.m.smit@utwente.nl

Abstract—Multimedia applications process streams of values and can often be represented as task graphs. For performance reasons, these task graphs are executed on multiprocessor systems. Inter-task communication is performed via buffers, where the order in which values are written into a buffer can differ from the order in which they are read. Some existing approaches perform inter-task communication with first-in-first-out buffers and reordering tasks and require applications with affine index expressions. Other approaches communicate containers, in which values can be accessed in any order, such that a reordering task is not required. However, these containers delay the release of locations, which can cause deadlock in cyclic task graphs.

In this paper, we introduce circular buffers with overlapping windows for deadlock-free execution of cyclic task graphs that may contain non-affine index expressions. Inside the windows, values can be written or read in an arbitrary order, such that a reordering task is not required. Deadlock is avoided by releasing a written location directly from the write window. The approach is demonstrated for the cyclic task graph of an orthogonal frequency-division multiplexing (OFDM) receiver application, containing non-affine index expressions.

I. INTRODUCTION

Multimedia applications are often executed on multiprocessor systems for performance reasons. These applications process streams of values and can be represented as task graphs. The tasks in these task graphs are executed in parallel, possibly on different processors, and communicate values via buffers. A value can be read from a buffer after it has been written, otherwise the reading task has to be blocked until the value has been written, this requires synchronization between the tasks.

In existing approaches \cite{1}\textendash\cite{3}, inter-task communication is performed via first-in-first-out (FIFO) buffers. Therefore, if the write order of values in a FIFO buffer differs from the order in which the values have to be read, a reordering task has to reorder the values in a reordering memory. This task becomes complex if it has to keep track of values that are read multiple times. To determine the behavior of the reordering task, affine index expressions are required for the two communicating tasks, where an affine index expression is limited to a summation of variables multiplied with constants plus an additional constant.

Another approach for inter-task communication \cite{4} uses containers, where a container is a place holder for values. Inside a container, values can be accessed in any order and therefore a reordering task is not required. After values are written in a container, the container is released such that the values in it can be read.

However, for a cyclic task graph the use of containers with more than one value can lead to deadlock, as demonstrated with the didactic example in Figure 1. The tasks \(t_1\) and \(t_2\) communicate via the buffers \(s_x\) and \(s_y\), according to the sequential code given beside the tasks, in which \(\sim\) depicts code that is omitted for clarity. The tasks have a cyclic dependency, because the assignment-statements of \(t_1\) depend on values written by \(t_2\) and vice versa. The read order from \(s_x\) can be captured in a container with two locations and the read order from \(s_y\) in a container with one location. This means that \(t_2\) can read the values of \(x\) by reading consecutive containers from \(s_x\). However, task \(t_1\) can release its first container, with the values \(x[0]\) and \(x[1]\), in \(s_x\) after \(t_2\) released its first container in \(s_y\), with the value \(y[0]\), but to release its first container in \(s_y\), \(t_2\) requires the first container released by \(t_1\). Both tasks are waiting for a container of the other, resulting in deadlock for this cyclic task graph.

In this paper, we present a so-called circular buffer (CB) with an overlapping read and write window for deadlock-free inter-task communication in cyclic task graphs with non-affine index expressions. A read (write) window contains locations for reading (writing) that a task can access multiple times in an arbitrary order, such that a reordering task is not required. With overlapping windows, deadlock is avoided for cyclic task graphs by releasing a location from the write window directly after it is written.

In a CB, each location has a full-bit that is set if the location contains a value. The novelty of the full-bit is that it does not require atomic read-modify-write operations, because it is only set and cleared by the writing task. In a CB, the writing task, called the producer, has a write window (WW). Before the producer writes a location in its window, the location consecutive to the head of the WW has its full-bit cleared and is added to the WW. After writing a value to a location, the producer releases this location directly from its WW by setting its full-bit.

In a CB, the reading task, called the consumer, has a read window (RW) in which the locations with a set full-bit can be read. The RW can overlap with the WW, because there can be a sequence of locations from which some can be read while...
other locations still have to be written. After reading a location in the RW, the consumer releases the location at the tail of its RW. In contrast to releasing the read location immediately, releasing the location at the tail of the RW makes it possible to read locations in the RW multiple times. The release for the location at tail of the RW is executed conditionally, where the simple condition compares a constant with a variable.

We will extend tasks to perform inter-task communication via CBs with overlapping windows. Determining sufficient buffer capacities to guarantee deadlock-free execution of a task graph is a problem that cannot be solved by computing a sufficient capacity for each buffer in isolation, but requires the whole task graph to be considered at once, this is illustrated with an example. We show that the communication via overlapping windows can be captured in a cyclo static dataflow (CSDF) model [5]. Using this CSDF model, we can compute sufficient buffer capacities to guarantee deadlock-free execution of the extended task graph. In the case study, we demonstrate our approach for a fragment of an orthogonal frequency-division multiplexing (OFDM) receiver application that has a cyclic task graph.

The organization of this paper is as follows. In Section II, the related work is discussed. Subsequently, Section III presents the supported applications. CBs with overlapping windows are explained in Section IV, before Section V discusses their usage. In Section VI, the extension of the tasks is presented. It is shown that buffer capacities for deadlock-free execution of a task graph cannot be computed per buffer in Section VII. Section VIII illustrates how capacities are determined for the CBs. The case study is presented in Section IX. Finally, conclusions are drawn in Section X.

II. RELATED WORK

A CB with a non-overlapping read and write window for the inter-task communication and synchronization in an acyclic task graph is presented in [6]. The synchronization is captured in a CSDF model, with which sufficient buffer capacities for deadlock-free execution are determined. In contrast, we present the extension of tasks to communicate via buffers with overlapping windows, in which a location is directly released from the WW after it is written, this is mandatory to guarantee deadlock-free execution of cyclic task graphs. The additional costs for overlapping windows are the full-bits.

The synchronization-statements for our approach are not supported by current streaming libraries, as e.g. [7]. Their APIs only support the addition of a location with a value to the head of the RW, this results in non-overlapping windows. In contrast, our approach requires a synchronization-statement that verifies the location to be read to contain a value, so if its full-bit is set.

A full-empty bit for each location in an inter-task communication buffer is proposed in [8]. The producer sets the full-empty bit of a location after writing and the consumer clears the full-empty bit after reading a location for the last time. In contrast, we use a full-bit for each location that is only set or cleared by the producer when the location is added to or removed from the WW, respectively. Because only the producer sets and clears the full-bits, no atomic read-modify-write operations are required.

In [9], a buffer with a window to be used by a single task with an affine index expression is described. This approach is extended in [10] such that the buffer can be allocated over multiple memories. In contrast, we present a buffer for inter-task communication and synchronization between two tasks that can be executed on different processors.

III. INPUT APPLICATIONS

Throughout this paper, we assume that an application is represented by a directed task graph $H = \{T, S, A, \alpha, \rho, \sigma, \theta\}$ that may contain cycles. 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 a nested loop program (NLP). For a stream, a task is executed an infinite number of times. 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 $s_{ij} = (t_h, t_l)$, with $s_j \in S$, is from task $t_h$ to task $t_l$, with $t_h, t_l \in T$ and $t_h \neq t_l$. Each edge represents a buffer. In a buffer $s_j$, the values of the corresponding array $a_j$ 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 $\alpha : T \times A \times N \rightarrow N$. The function $\rho(t_i, a_j)$ returns the total number of accesses performed during one execution of a task $t_i$ in array $a_j$, with $\rho : T \times A \rightarrow N$. The size, in number of locations, of the array $a_j$ is given by $\sigma(a_j)$, with $\sigma : A \rightarrow N$. The capacity of buffer $s_j$ is the number of locations $\theta(s_j)$, with $\theta : S \rightarrow N$.

We describe the NLP that defines the behavior of a task using a C-like-syntax. The NLPs define the inter-task communication by reading and writing arrays. An NLP contains assignment-statements and for-loops. We use $for : i : l : u$ as a shorthand notation for a for-loop, with $i$ 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. The upper-bound and the lower-bound are constant values.

An array is either read or written in an NLP. Furthermore, an NLP should contain single assignment code, this means that a location in an array is assigned a value at most once per execution of the task. The index of a location in an array is determined with an index expression that can have the iterators of nested for-loops as variables. The index expression is not limited to be affine, but the result of the index expression should be a function of the used variables. Therefore, by executing each task once, we can derive for every array the sequence of written and read locations. These are the locations that are returned by the $\alpha(t_i, a_j, l)$ function.

Figure 2 depicts a synthetic task graph that is used in a number of examples in this paper. In this task graph, task $t_2$ reads from array $a_0$ using the non-affine function $F$. The symbol $\sim$ denotes a code fragment that is omitted for clarity.

For the accesses of tasks in arrays, three interesting access patterns are identified, being out-of-order access, multiplicity, and skipping [2], [4].

For the out-of-order access pattern, non-consecutive locations in an array are accessed. In Figure 2, $t_1$ writes out-of-order in array $a_0$, because location two is written during access zero of $t_1$, $\alpha(t_1, a_0, 0) = 2$, and the non-consecutive location one is written during access one, $\alpha(t_1, a_0, 1) = 1$. The multiplicity access pattern occurs if a location is accessed more than once. Figure 2 shows an example of multiplicity for the access of $t_2$ in $a_0$, where location two is read during access zero and five, $\alpha(t_2, a_0, 0) = \alpha(t_2, a_0, 5) = 2$ with $F(0, 0) = F(1, 2) = 2$. 


The skipping pattern occurs if a location is written in the array, but not read. An example is shown in Figure 2, where \( t_1 \) writes location three, seven and eight in array \( \hat{a}_a \), but \( t_2 \) never reads these locations.

IV. OVERLAPPING WINDOWS IN A CIRCULAR BUFFER

We perform inter-task communication via a CB in which both tasks have a window, a task can access the locations in its window in an arbitrary order. This section first explains CBs with non-overlapping windows that guarantee deadlock-free execution of acyclic task graphs. Subsequently, CBs with overlapping windows that guarantee deadlock-free execution of cyclic task graphs are explained.

The two tasks that communicate by reading from and writing in a buffer are executed in parallel, possibly on different processors. A value can only be read after it has been written, therefore the reading task should be blocked if it attempts to read an unwritten location, this because our processors do not run in lockstep \([11]\). The order in which read and write operations in a buffer become visible and accessible for other processors is defined by a memory consistency model. We use a memory consistency model that synchronizes using acquire and release calls, as the memory consistency models in \([12]\), \([13]\) do. Before accessing a location we perform an acquire call for it, this function blocks until the location is signaled to be available. Succeeding the access to a location a release call signals that the location is available. A location acquired for writing cannot be acquired by the consumer for reading before the producer released it.

We use a CB for the inter-task communication. A CB can be implemented with a read pointer \( r \) and a write pointer \( w \); as depicted in Figure 3. Arbitrary locations can be read between \( r \) and \( w \) in the CB, therefore allowing the multiplicity, skipping, and out-of-order access patterns. Between \( w \) and \( r \) in the CB arbitrary locations can be written. The pointer \( w \) or \( r \) can be incremented to make a location available for reading or writing, respectively. A pointer that reaches the end of the CB is wrapped around.

In a CB, starting at \( r \) a number of consecutive locations are acquired that form a RW, where \( \hat{r} \) points to the location at the head of this window. Similarly, starting at \( w \) a number of consecutive acquired locations form a WW, with \( \hat{w} \) pointing to the head of the window. For a window the pointer to its head and the pointer to its tail administrate the consecutive acquired locations. Both tasks have random access in their window.

In \([6]\), inter-task communication is performed via a CB with a non-overlapping RW and WW. Preceding an access to a location in the window, a task acquires the location consecutive to the head of the window by incrementing the pointer to the head of the window. Succeeding an access, the location at the tail of the window is released by incrementing the pointer to the tail of the window. This results in a sliding RW and WW, as depicted in Figure 3. The main advantage of using a sliding RW is that it allows locations in the RW to be read multiple times without requiring a complex reordering task. Both the acquire and release operation are executed conditionally, where the simple condition compares a counter variable with a constant. The main drawback is that non-overlapping windows can cause deadlock in a cyclic task graph, because the location written in the WW is not necessarily the location that is released. The delayed release of a location from the WW and the cyclic dependencies can cause deadlock, as illustrated in the example in the introduction.

For cyclic dependencies, as in Figure 1, a value should be available for reading directly after it has been written. This requires the producer to release a written location directly from its WW, such that the consumer can acquire it for reading. For non-overlapping windows the location at \( w \) is released after a write access. Because a written location is not necessarily at location \( w \), we have to allow reading past \( w \) in the CB, this results in an overlapping RW and WW, as depicted in Figure 4.

For overlapping windows, per location in the WW it should be administrated if it can be acquired for reading. This can be done with a full-bit that is cleared when its location is acquired for writing and set directly after a value is written at its location. A location in the RW with a set full-bit can be acquired for reading.

A full-bit can either be stored along with its location or in the buffer administration. Some architectures \([14]\), \([15]\) provide an additional bit for every location in the shared memory that can be used as a full-bit. An alternative is to store full-bits in the buffer administration by using a bit vector, with a full-bit for each location in the CB.

Before writing a location, the producer adds a location to the WW. To add a location to WW, the producer clears the full-bit of the location consecutive to \( \hat{w} \) and acquires this location by incrementing \( \hat{w} \). Note that \( \hat{w} \) cannot overtake \( r \), therefore if \( r \) is the location consecutive to \( \hat{w} \), the clearing of the full-bit and the acquire are blocked until \( r \) is incremented. After writing a location, it is released from the WW by setting the full-bit of this location.

To read a location in a CB the consumer acquires this location. The acquire call for a location checks if the full-bit of the location is set and that the location is not past \( \hat{w} \). After reading a location, the consumer releases the location at
by incrementing it. Because locations in the RW can still be read multiple times, overlapping windows do also not require a complex reordering task.

Updating the read pointer \( r \), write pointer \( w \), and full-bits requires no atomic read-modify-write operations, as for example test-and-set and fetch-and-add. These operations are not required, because \( r \) is only updated by the consumer and \( w \) and the full-bits only by the producer. Note that due to the full-bits, overlapping windows do not need \( r \) and \( w \).

V. USING OVERLAPPING WINDOWS

In a CB with an overlapping RW and WW, the producer acquires a consecutive location preceding a write access and the consumer releases a consecutive location succeeding a read access. The producer may need to acquire a number of locations before its first write, to make sure that the location to be written is acquired during each write. The consumer should not succeed each read with a release, to make sure that no location is released before it is read for the last time. In this section, we will determine the number of locations acquired before the first write and the number of reads that should not be succeeded with a release, these will be used in Section VI to extend tasks to use overlapping windows in a CB.

Preceding a write access, a producer acquires the location consecutive to \( w \), until for all locations from the communicated array \( a \) a location has been acquired in \( CB \) \( s \). Because the first location to be written in \( s \) is not necessarily the location consecutive to \( w \), more than one location may need to be acquired before the first write. It can be guaranteed that before each write access of the producer the location to be written is acquired, by acquiring a number of locations preceding the first write and its acquire. For a producer \( t_p \) that writes in \( s \), this number of acquired locations preceding the first write access and its acquire is called the lead-in \( d_1(t_p, s) \), with \( d_1 : T \times S \rightarrow \mathbb{N} \).

Figure 5 depicts the intuition behind the lead-in, for the writing in \( s_a \) by \( t_1 \) from Figure 2. The upper sequence in the figure contains the acquired locations and the lower sequence the written locations. The sequence with acquired locations is shifted left, such that no location is written before it is acquired. In this figure, the locations in bold are acquired and written during the same access, they determine the lead-in. For this example we find that by acquiring two locations preceding the first write and its acquire, so \( d_1(t_1, s_a) = 2 \), during each write access the written location is acquired.

\[
\begin{align*}
\alpha(t_1, a_1) &\text{ Location acquired} \\
&= 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \\
\end{align*}
\]

Fig. 5. The lead-in \( d_1 \) for \( t_1 \) in \( s_a \), from Figure 2

Given a task \( t_p \), a CB \( s_j \) with its corresponding array \( a_j \), and an access counter \( l \) for which it holds that \( 0 \leq l < \rho(t_p, a_j) \), the expression for the lead-in is:

\[
d_1(t_p, s_j) = \max_l (\alpha(t_p, a_j, l) - l)
\]

The validity of this expression is proven in [6].

In a CB with overlapping windows, the consumer can succeed a read access by releasing the location at \( r \). It is possible that the first location read by a consumer from \( s \) is not the first location in \( s \), which is equal to \( r \). The first location in \( s \) is only acquired for reading during the second read, if the first read is not succeeded by a release. To make sure that during each read of the consumer the location to be read is still acquired, possibly a number of the first reads should not be succeeded by a release. The number of reads of a consumer \( t_c \) in \( s \) without a release is called the lead-out \( d_2(t_c, s) \), with \( d_2 : T \times S \rightarrow \mathbb{N} \).

Figure 6 depicts the intuition behind the lead-out, for the reading in \( s_a \) by \( t_2 \) from Figure 2. In this figure, the upper sequence represents the read locations and the lower sequence the released locations. The sequence with released locations is shifted right such that no location is released before it has been read for the last time. Location two is depicted in bold, because it determines the lead-out, which is three, so \( d_2(t_2, s_a) = 3 \).

\[
\begin{align*}
\alpha(t_2, a_1, l) &\text{ Location read} \\
&= 2 \quad 1 \quad 0 \quad 5 \quad 4 \quad 2 \quad 4 \quad 5 \quad 6 \\
\end{align*}
\]

\[
\begin{align*}
\alpha(t_2, a_1, l) &\text{ Location released} \\
&= 2 \quad 1 \quad 0 \quad 5 \quad 4 \quad 2 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \\
\end{align*}
\]

Fig. 6. The lead-out \( d_2 \) for \( t_2 \) in \( s_a \), from Figure 2

Given a consumer \( t_c \), a CB \( s_j \), and an access counter \( l \) for which it holds that \( 0 \leq l < \rho(t_c, a_j) \), the expression for the lead-out is:

\[
d_2(t_c, s_j) = \max_l (l - \alpha(t_c, a_j, l))
\]

The correctness of this expression is proven in [6].

If a consumer skips the first locations in a CB its lead-out can be negative. For example, a consumer \( t_c \) that only reads location two from \( s_j \) has a lead-out of minus two, \( d_2(t_c, s_j) = -2 \), this lead-out is found by applying Equation 2 with \( l = 0 \) and \( \alpha(t_c, a_j, 0) = 2 \).

VI. EXTENDING THE NLP

To communicate and synchronize via CBs with overlapping windows, the C-code of the NLP that defines a task is extended with synchronization-statements and statements for communication. Acquire-statements and release-statements are added to the code for synchronization and assignment-statements are adjusted for the communication via CBs instead of arrays, as presented in this section.

For overlapping windows two different acquire-statements and release-statements are required. The statement to acquire (release) the location consecutive to the head (tail) of the window in \( s \) is acquire(s) (release(s)). In contrast, the statement to acquire (release) a location \( l \), as given by an index-expression, in \( s \) is acquireL(l,s) (releaseL(l,s)). Both acquire-statements are blocking, this means that they do not return until they succeed.

A template to extend the C-code of the NLP that defines a task \( t \) is depicted in Figure 7. In this template, \( t \) reads from CB \( s_r \) using index expression \( m_r \) with the read statement. Task \( t \) writes a value in \( s_w \) using index expression \( m_w \) with the write statement. For CB \( s_r \) the producer is task \( t_p \) and for CB \( s_w \) the consumer is \( t_c \).

Three phases are depicted by the template in Figure 7, the initial phase, the processing phase, and the final phase. In the initial phase, lead-in \( d_1(t, s_w) \) locations are acquired for
![](https://latex.codecogs.com/png.latex?\text{int} \ p = 0;  \\
\text{for} \ (i:\beta(t))\{}  \\
\quad \text{if} \ (i \leq \sigma(a_w) - d_1(t, s_w))  \\
\quad \quad \acquire(s_w);  \\
\quad p++_; \\
\text{\} \int \ l_w = 1;  \\
\int \ l_r = 1;  \\
\text{for-loops } \{}  \\
\quad \text{if} \ (l_w \leq \sigma(s_w) - d_1(t, s_w))  \\
\quad \quad \acquire(s_w);  \\
\quad \text{acquire(m_s,s_w);}  \\
\quad \text{write(m_w,s_w,F_0(read(m_r,s_r)));}  \\
\quad \text{release(m_w,s_w);}  \\
\quad \text{if} \ (l_r > d_2(t, s_r))  \\
\quad \quad \text{release}(s_r);  \\
\quad \quad \int \ l_r = \int \ l_r + 1;  \\
\quad p++;  \\
\text{\} \text{Processing phase}  \\
\text{\} \text{Initial phase}  \\
\text{for \ (i:\beta(t))}\{}  \\
\quad \text{if} \ (i \leq \sigma(a_r) - \rho(t, a_r) - d_1(t, s_w))  \\
\quad \quad \acquire(s_r);  \\
\quad \text{if} \ (i \leq \sigma(a_r) - \rho(t, a_r) + \max(0,d_2(t,s_r)))  \\
\quad \quad \text{release}(s_r);  \\
\quad p++;  \\
\text{\} \text{Final phase}  \\
\text{Fig. 7. Template for extending the NLP of } t, \text{ where CB } s_r \text{ is read and CB } s_w \text{ is written}

all written CBs, to guarantee that during a write access the location to be written is acquired. In the processing phase, the assignment-statements of the NLP are adjusted by adding read and write statements for the accessed CBs. To synchronize via these CBs the assignment-statements are encapsulated by acquire-statements and release-statements. Note that in this template a single assignment-statement is depicted. For an NLP that contains more than one assignment-statement, each assignment-statement is adjusted and encapsulated in acquire-statements and release-statements. In this case, it is possible that a CB is either read or written by multiple assignment-statements. During the final phase, the remaining locations in the read CBs are released. During these three phases, in each CB s in total \( \sigma(a) \) consecutive locations are acquired and released, where \( \sigma(a) \) is the number of locations in the array \( a \) that corresponds to \( s \).

We define a synchronization section of an extended task, as a sequence of executed statements together with acquire-statements and/or release-statements. During the initial phase and the final phase of an extended task, each iteration of the for-loop is a synchronization section. In the processing phase, each assignment-statement with its encapsulating acquire-statements and release-statements is a synchronization section. The template in Figure 7 contains a dummy counter \( p \). During the execution of a task, the value of \( p \) represents the number of the current synchronization section. In section VIII, synchronization sections will be used to derive a CSDF model from the extended task graph.

The initial phase, as depicted in Figure 7, is executed at the beginning of a task. This phase introduces counters for the accessed CBs and contains a for-loop that acquires locations in the written CBs. In a written CB \( s_w \), lead-in \( d_1(t, s_w) \) locations are acquired. Note that every iteration of the for-loop acquires at most one location in a written CB. To acquire more locations at once requires knowledge of the buffer capacity, to avoid an acquire-statements for more locations than available in the CB. The same holds for releasing locations.

The number of iterations performed by the for-loop in the initial phase, depends on the number of locations to be acquired for the lead-in among the written CBs, with:

\[
\beta(t) = \max(d_1(t, s_w) \mid s_w = (t, t_e) \in S)
\] (3)

For a CB an access counter is introduced that counts the number of accesses in it. An access counter is incremented after each access to the corresponding CB, where different assignment-statements in the NLP can access the CB. Because not all assignment-statements have to access the same CBs, each CB has its own counter. In the template in Figure 7, the counter \( l_w \) is introduced for \( s_w \) and \( l_r \) for \( s_r \). Note that since this example contains only one assignment-statement that accesses both \( s_w \) and \( s_r \), a single counter would also have been sufficient.

During the processing phase of a task \( t \), each assignment-statement is preceded by acquire-statements and succeeded by release-statements for the accessed CBs. The template of Figure 7 depicts that if a written CB conditionally a consecutive location is acquired and that the written location is released. For a read CB, the location to be read is acquired, this verifies that the full-bit of the location is set, and conditionally a consecutive location is released. Succeeding the last release-statements, the access counter of each accessed CB is incremented.

Preceding a write access to \( s_w \), an if-statement determines whether there are locations left to acquire using the access counter \( l_w \), so if \( l_w \leq \sigma(t, s_w) - d_1(t, s_w) \). In the assignment-statement, the write access to array \( a_w \) at location \( m_w \) is replaced with \( \text{write}(m_w,s_w,x) \), where \( s_w \) is the CB corresponding to \( a_w \) and \( x \) the value to be written. Succeeding an assignment-statement that writes location \( m_w \) in \( s_w \), location \( m_w \) is released by a releaseL(m_w,s_w) statement.

Preceding the assignment-statement that reads location \( m_r \) from \( s_r \) in the processing phase, an acquireL(m_r,s_r) statement acquires this location. The part of the assignment-statement that reads location \( m_r \) from \( a_r \) is replaced with \( \text{read}(m_r,s_r) \), to read location \( m_r \) from CB \( s_r \). Succeeding the assignment-statement, an if-statement checks if lead-out \( d_2(t, s_r) \) accesses have been performed in \( s_r \) by using its access counter \( l_r \), to verify whether a location can be released.

The last phase depicted in Figure 7 is the final phase, during which a for-loop acquires and releases the remaining locations for the arrays communicated via the CBs. Due to skipping, possibly not all locations were acquired in a written CB \( s_w \). The for-loop of the final phase acquires the remaining \( \sigma(a_r) - \rho(t, a_r) - d_1(t, s_w) \) locations in \( s_w \), where \( \rho(t, a_r) \) returns for one execution of \( t \) the number of accesses in \( a_w \). For a read CB \( s_r \) there can be remaining locations that have to be released, due to multiplicity, skipping, or out-of-order access. The for-loop releases the remaining \( \sigma(a_r) - \rho(t, a_r) + \max(0,d_2(t, s_r)) \) locations in \( s_r \), where the maximum of 0 and \( d_2(t, s_r) \) is taken to cover the case that the lead-out is negative.

The number of iterations performed by the for-loop in the final phase is determined by the maximum number of locations to be released in the read CBs or the maximum number of locations to be acquired in the written CBs, with:
In its WW, reads from s because the producer writes the values in the same order as in
if the CBs t omitted because they are not used in any of the conditions.

have a cyclic dependency due to communication via execution of an application. This will be demonstrated with
adjust the assignment-statements. Therefore, the extension of
and the adjustment of the assignment-statements to access CBs
and b instead of arrays. Furthermore, the last statement of
Note that for extended task graph, task t requires an empty location in s, and t requires an empty location in s, these are not available, so the task graph deadlocks. The buffers are too small for deadlock-free execution, due to the order in which the read and write operations of both tasks are performed.

VII. Determining Capacities per Buffer

In this section we will demonstrate that determining sufficient buffer capacities per CB cannot guarantee deadlock-free execution of an application. This will be demonstrated with an example in which the locations in the CBs are accessed in-order.

Figure 9 depicts an extended task graph, where the tasks have a cyclic dependency due to communication via s and s. Both t and t access the locations in s and s in-order, without skipping and multiplicity, therefore there is FIFO communication via both s and s. A CB in isolation FIFO communication requires only one location in the buffer, because the producer writes the values in the same order as the consumer reads them, so θ(s) = θ(s) = 1.

Due to the cyclic dependency and the execution order of the sequential code, the extended task graph in Figure 9 deadlocks if the CBs s and s both have a capacity of one location. The reason is as follows. Task t starts by acquiring the location in s for its WW, writes a value on it and releases it. Task t2 acquires the location in s in its RW, the location in s in its WW, reads from s, writes in s, and releases both
locations. Now the location in s contains a value and the location in s is empty. Task t1 acquires the empty location in s in its WW, writes the location, and releases it. Now both the location in s and s contain a value. To continue their execution, task t1 requires an empty location in s, and t2 requires an empty location in s, these are not available, so the task graph deadlocks. The buffers are too small for deadlock-free execution, due to the order in which the read and write operations of both tasks are performed.

VIII. Buffer Capacity Computation

This section illustrates the derivation of a CSDF model from the synchronization sections in an extended task graph. With a CSDF model we can determine sufficient buffer capacities for an extended task graph to guarantee deadlock free execution. We start by describing the CSDF model. Following, we first derive a CSDF model from an extended task graph that only accesses locations in-order, without skipping and multiplicity, subsequently we derive a CSDF model from an extended task graph with out-of-order access.

We model the synchronization sections of the tasks in an extended task graph in a cyclo static dataflow (CSDF) model [5], [16]. A CSDF model consists of a directed graph G = (V, E, δ, φ), with V the set of actors and E the set of directed edges. An edge ε = (v, v), with ε ∈ E, is from actor v to actor v, with v, v ∈ V. An edge represents an unbounded queue. Actors communicate tokens over edges. There are δ(ε) initial tokens on an edge ε, with δ : E → N. An actor v has a period that contains φ(v) phases, with φ : V → N. The first phase is phase 0. For an actor, per phase a number of consumed tokens is given for each input edge and a number of produced tokens for each output edge. An actor is fired for each phase. At the moment an actor v is fired, it atomically consumes the tokens for the current phase from its input edges. On finishing a firing, an actor atomically produces the tokens for the current phase on its output edges.

To derive a CSDF model from an extended task graph, every task t is modeled by an actor v. A CB s = (l, l) is modeled by an edge pair, with an edge ε = (v, v) and a back-edge ε = (v, v) between the actors v and v. Initially ε contains δ(ε) tokens, which corresponds to the capacity θ(s) of the modeled CB. Edge ε contains no initial tokens. Each synchronization section in an extended task t, as depicted by the counter p in the template in Figure 7, corresponds with a phase of v. The number of phases (p(v)) of an actor v is

VII. Determining Capacities per Buffer

In this section we will demonstrate that determining sufficient buffer capacities per CB cannot guarantee deadlock-free execution of an application. This will be demonstrated with an example in which the locations in the CBs are accessed in-order.

Figure 9 depicts an extended task graph, where the tasks have a cyclic dependency due to communication via s and s. Both t and t access the locations in s and s in-order, without skipping and multiplicity, therefore there is FIFO communication via both s and s. For a CB in isolation FIFO communication requires only one location in the buffer, because the producer writes the values in the same order as the consumer reads them, so θ(s) = θ(s) = 1.

Due to the cyclic dependency and the execution order of the sequential code, the extended task graph in Figure 9 deadlocks if the CBs s and s both have a capacity of one location. The reason is as follows. Task t starts by acquiring the location in s for its WW, writes a value on it and releases it. Task t2 acquires the location in s in its RW, the location in s in its WW, reads from s, writes in s, and releases both
locations. Now the location in s contains a value and the location in s is empty. Task t1 acquires the empty location in s in its WW, writes the location, and releases it. Now both the location in s and s contain a value. To continue their execution, task t1 requires an empty location in s, and t2 requires an empty location in s, these are not available, so the task graph deadlocks. The buffers are too small for deadlock-free execution, due to the order in which the read and write operations of both tasks are performed.

VIII. Buffer Capacity Computation

This section illustrates the derivation of a CSDF model from the synchronization sections in an extended task graph. With a CSDF model we can determine sufficient buffer capacities for an extended task graph to guarantee deadlock-free execution. We start by describing the CSDF model. Following, we first derive a CSDF model from an extended task graph that only accesses locations in-order, without skipping and multiplicity, subsequently we derive a CSDF model from an extended task graph with out-of-order access.

We model the synchronization sections of the tasks in an extended task graph in a cyclo static dataflow (CSDF) model [5], [16]. A CSDF model consists of a directed graph G = (V, E, δ, φ), with V the set of actors and E the set of directed edges. An edge ε = (v, v), with ε ∈ E, is from actor v to actor v, with v, v ∈ V. An edge represents an unbounded queue. Actors communicate tokens over edges. There are δ(ε) initial tokens on an edge ε, with δ : E → N. An actor v has a period that contains φ(v) phases, with φ : V → N. The first phase is phase 0. For an actor, per phase a number of consumed tokens is given for each input edge and a number of produced tokens for each output edge. An actor is fired for each phase. At the moment an actor v is fired, it atomically consumes the tokens for the current phase from its input edges. On finishing a firing, an actor atomically produces the tokens for the current phase on its output edges.

To derive a CSDF model from an extended task graph, every task t is modeled by an actor v. A CB s = (l, l) is modeled by an edge pair, with an edge ε = (v, v) and a back-edge ε = (v, v) between the actors v and v. Initially ε contains δ(ε) tokens, which corresponds to the capacity θ(s) of the modeled CB. Edge ε contains no initial tokens. Each synchronization section in an extended task t, as depicted by the counter p in the template in Figure 7, corresponds with a phase of v. The number of phases (p(v)) of an actor v is
equal to the total number of synchronization sections of the corresponding extended task.

First we will discuss the derivation of the CSDF model depicted in Figure 10 from the extended task graph in Figure 9. In this extended task graph both tasks access the locations in their CBs in order without skipping and multiplicity. The acquireL-statements (releaseL-statements) in this task graph behave as acquire-statements (release-statements), because they only acquire (release) consecutive locations.

The CSDF model in Figure 10 models task $t_1$ with actor $v_1$ and $t_2$ with $v_2$. CB $s_x$ is modeled by the edge pair $e_x$ and $e_x'$, with $e_x'$ being the back-edge. A back-edge contains a black dot that represents a number of initial tokens on this edge. Above the black dot of $e_x'$ the number of initial tokens $\delta(e_x')$ is depicted. CB $s_y$ is modeled by the edge pair $e_y$ and $e_y'$.

The $n$ consecutive locations acquired in $s$ during synchronization section $p$ of $t$, are modeled by the consumption of $n$ tokens by $v$ during phase $p$ from the incoming edge $e$, with $e$ from the edge pair that models $s$. In Figure 10 the incoming edge $e_y$ at actor $v_1$ contains the list $(0,0,1,1)$, in this list each element corresponds with a phase of actor $v_1$ and the elements in the list corresponds with the number of consumed tokens during that phase. This list shows that during synchronization sections 0 and 1, task $t_1$ acquires no locations, and in both the sections 2 and 3, one location is acquired in $s_y$, as depicted in Figure 9. For the consumption of actor $v_1$ from $e_x'$ the list (4-1) is a shorthand notation for four phases that consume one token.

The $n$ consecutive locations released in $s$ during synchronization section $p$ of task $t$, are modeled by the production of $n$ tokens on the outgoing edge $e$ by actor $v$ during phase $p$. In Figure 10 the outgoing edge $e_y$ of $v_2$ contains the list $(1,1,0,0)$ that represents the number of tokens produced during the four phases. From Figure 9, we see that both synchronization sections zero and one of task $t_2$ release one location in $s_y$ and sections two and three do not release a location, which corresponds with the number of produced tokens on $e_y$ by $v_2$ during its four phases.

The CSDF model in Figure 10 and the firing sequence in Table I depict, in a more explicit way than the textual description in Section VII, that assigning both back-edges one initial token leads to deadlock. Table I depicts that initially $e_x'$ and $e_y'$ contain one token. After firing actor $v_1$, one token is consumed from $e_x'$ and produced on $e_x$. The token on $e_x$ is consumed by firing actor $v_2$, which also consumes the token on $e_y'$ and produces a token on $e_y'$ and $e_y''$. For its second phase actor $v_1$ consumes the token from $e_x'$, and produces one token on $e_x$. Actor $v_1$ cannot fire for its third phase, because there is no token on $e_x$ and actor $v_2$ cannot fire for its second phase because there is no token on $e_y'$, this is a deadlock situation.

In [16] an algorithm is presented that can determine sufficient initial tokens in a CSDF model for deadlock freedom. Applying this algorithm to the CSDF model in Figure 10 results in 2 initial tokens being sufficient for $e_x'$ ($\delta(e_x) = 2$) and one for $e_y'$ ($\delta(e_y') = 1$), to guarantee deadlock freedom. So, deadlock-free execution of the extended task graph in Figure 9 is guaranteed with CB capacities of at least $\theta(s_x) = 2$ and $\theta(s_y) = 1$. The derivation of the CSDF model from the extended task graph in Figure 11 is less straightforward than for the previous example, due to the out-of-order write access of $t_1$ in $s_a$. The remainder of this section presents how acquireL-statements and releaseL-statements are captured in a CSDF model.

As presented above, a release statement or an acquire statement for a consecutive location in $s$ is modeled by the production or consumption of one token on $e'$ in a CSDF model, with $e'$ being a back-edge. In a CSDF model tokens are consumed in FIFO order from an edge. The tokens produced on a back-edge by an actor that models a release statement always represent consecutive locations, therefore the tokens consumed from such an edge represent consecutive locations. In contrast, a releaseL-statement or an acquireL-statement can acquire or release an arbitrary location between $w$ and $r$. To model these statements the order in which locations are acquired and released must be considered.

A releaseL-statement in $s$ is modeled by the production of a token on $e$, where the produced token represents the released location. The basic idea of modeling an acquireL-statement in $s$, is to consume tokens from $e$ until the token that represents the location to be acquired is consumed. For example, for a modeled releaseL-statement that produces a token representing location 0 followed by a token representing location 1 on an edge $e$, the acquireL-statement for location 1 is modeled by consuming both tokens from $e$. If the next acquireL-statement should acquire location 0, this is modeled by consuming zero tokens, because the token representing location 0 has already been consumed in the previous phase.

Modeling an acquireL-statement requires the lists with released and acquired locations. For $s_a$ in Figure 11, the list of locations released by $t_1$ is $\{1,0,3,2\}$ and the list...
of locations acquired by $t_2$ is $\{0,1,2,3\}$. Task $t_2$ does not release a location in synchronization section zero, releases location 1 during synchronization section one, and releases location 0 during synchronization section two. In the CSDF model this is captured by $v_1$ producing no tokens in phase zero, one token representing location 1 on $e_a$ in phase one, and one token representing location 0 on $e_a$ in phase two. The acquireL-statement in $s_a$ by task $t_2$, is captured by the consumption from $e_a$ by $v_2$. Actor $v_2$ consumes two tokens from $e_a$ during phase zero to model the acquire of location 0, first the token that represents location 1 and next the token that represents location 0. The acquireL-statement in synchronization section one of $t_2$ acquires location 1, actor $v_2$ captures this by consuming zero tokens from $e_a$ during phase one, because it already consumed the token representing location 1.

To model an acquireL-statement of a consumer $t_c$ from a CB $s$ we specify a function that returns the number of tokens to be consumed from an edge $e$. First the function $\omega(t_c, s, i)$ is specified that returns the list with locations released by the producer $t_p$ in $s$, before the location read by the consumer $t_c$ in synchronization section $i$ is released, with $\omega : T \times S \times N \rightarrow \{\mathbb{N}\}$.

$$\omega(t_c, s, i) = \{ \alpha(t_p, a, j) \mid 0 \leq j \leq g; \alpha(t_p, a, g) = \omega(t_c, s, i - \beta(t_c)) \}$$ (5)

For task $t_2$ from Figure 11, $\omega(t_2, s_a, 0)$ results in $\{\}$, because the synchronization section is in the initial phase, and $\omega(t_2, s_a, 1)$ results in $\{1,0\}$, because the producer writes locations 1 and 0 before the consumer can read location 1 in its synchronization section one.

We have to determine the locations that have to be acquired between synchronization section $i - 1$ and $i$. This list is found by taking the relative complement in $\omega(t_c, s, i)$ and the union of all locations released preceding the location to be acquired in synchronization section $i$ ($\omega(t_c, s, i)$) and the union of all locations released preceding the already acquired locations ($\bigcup_{j=0}^{i-1} \omega(t_c, s, j)$)). By taking the cardinality ($|\cdot|$) of the resulting set, i.e. the number of elements in this set, the function $\lambda(t_c, s, i)$ returns the number of tokens to be consumed from $e$ by actor $v_c$ that models $t_c$ during phase $i$, with $\lambda : T \times S \times N \rightarrow \mathbb{N}$.

$$\lambda(t_c, s, i) = |\omega(t_c, s, i) \cup \bigcup_{j=0}^{i-1} \omega(t_c, s, j)|$$ (6)

For task $t_2$ from Figure 11, $\lambda(t_2, s_a, 1)$ results in 2, because $|\{1,0\} \setminus \{\}| = 2$ and $\lambda(t_2, s_a, 2)$ results in 0, because $|\{1,0\} \setminus \{1,0\}| = |\{0\}| = 0$.

Figure 12 depicts the CSDF model derived from the extended task graph in Figure 8. Sufficient initial tokens to cause deadlock freedom are derived using the approach in [16]. We found that a sufficient number of initial locations is $\delta(e_a) = 6$ and $\delta(e_b) = 1$. This corresponds to sufficient buffer capacities of $\theta(s_a) = 6$ and $\theta(s_b) = 1$ for deadlock-free execution.

IX. Case study

In this section, we demonstrate our approach for a compacted OFDM receiver application for digital video broadcasting [17], similar to the one described in [18]. A fragment of the OFDM receiver application is presented to keep the extended task graph and its CSDF model understandable.

Figure 13 depicts the task graph of an OFDM receiver application, where a timing corrector (TC) task reads values from array $a_x$. In this task graph, one execution of TC reads eight values from $a_x$, whereas an OFDM receiver operating in 2K mode would read 2048 values. Using the values from array $a_x$, the values read from $a_x$ are adjusted using the function $adjustTiming$ before they are written in $a_y$. Initially $a_y$ contains eight zeros. The fast fourier transformation (FFT) task reads eight values from $a_y$, stores them in the array $a_t$, and applies the function $FFT$ to $a_t$ from which the result is stored in $a_v$. The values from $a_v$ are written in $a_y$ and $a_z$, using the bit-reverse function $B$ that results in the write order $\{0,4,2,6,1,5,3,7\}$. The demodulator (DM) task demodulates the values it reads from $a_z$, using the function $demodulate$, before writing them in $a_y$.

Figure 14 depicts the extended task graph of Figure 13. The FFT task writes in bit-reversed order in both $s_b$ and $s_c$, and has therefore a lead-in of three locations in both CBs, thus an initial phase with three iterations. Furthermore, in the FFT task the assignment-statements that write in $s_b$ and $s_c$ are in the same loop-body, both assignment-statements are encapsulated with an acquireL-statement and a releaseL-statement.

Figure 15 depicts the CSDF model that is derived from the extended task graph in Figure 14. The CBs $s_x$ and $s_y$ are not included in this model to obtain a more compacted figure. Task TC is modeled by actor $v_t$, task FFT by $v_f$, and task DM by $v_d$. The consumption by $v_f$ from $e_x$ is given by the list $\langle 11\cdot 0, 8\cdot 0\rangle$, in which $8\cdot 0$ is a shorthand notation for an eight times repetition of the consumption list $\langle 0, 1\rangle$. Actor $v_f$ has 27 phases that model the 27 synchronization sections of the FFT task. The FFT task has three synchronization sections for the initial phase and three times eight synchronization sections in the processing phase.

During its processing phase, the FFT task executes releaseL-statements in $s_c$ that release locations out-of-order. This
Fig. 14. Extended task graph of Figure 13

Fig. 15. CSDF model derived from the OFDM receiver in Figure 14

releaseL-statement is modeled by the production of one token by \( v_f \) on \( e_v \) during every corresponding phase. In contrast, the in-order acquiring of locations by the DM task from \( s_v \), using an acquireL-statements, is modeled by the consumption from \( e_v \) by \( v_d \). The consumption order \( (1, 4, 0, 2, 3, 0, 1) \) from \( e_v \) by \( v_d \) is a consequence of the out-of-order production of tokens by \( v_f \).

For the CSDF model depicted in Figure 15, we have determined sufficient initial tokens. We found that \( \delta(e_v) = 1 \), \( \delta(e_v') = 7 \), and \( \delta(e_v'') = 7 \) are sufficient initial tokens to guarantee deadlock freedom in the CSDF model. This corresponds with \( \theta(s_v) = 1 \), \( \theta(s_v') = 7 \), and \( \theta(s_v'') = 7 \) being sufficient buffer capacities to guarantee deadlock-free execution of the OFDM receiver.

For overlapping windows, compared to non-overlapping windows, the administration overhead is one full-bit per location in the CB. For the OFDM receiver each location in a CB stores a 32-bit complex number, therefore the administration overhead for \( s_v \), \( s_v' \), and \( s_v'' \) is one full-bit per 32-bit complex number, which is \( 1 \times 32 = 100 \approx 3\% \).

The cyclic task graph of an OFDM receiver application apply overlapping windows, because they guarantee deadlock-free execution. It might be possible to use non-overlapping windows for some inter-task communication buffers in a cyclic task graph, but this requires verification for deadlock freedom. If a deadlock situation is encountered, it may not be clear which buffer causes it.

X. Conclusion

In this paper, we introduced a circular buffer with an overlapping read window and write window that can be used for the inter-task communication and synchronization in cyclic task graphs, where the tasks may contain non-affine index expressions. The novelty of these buffers is that a location is directly released from the write window after it is written, which is required to guarantee deadlock-free execution of cyclic task graphs.

An important difference with current approaches is that we use windows in which the locations can be accessed in an arbitrary order. Therefore, we do not require a reordering task.

To administrate whether a location can be acquired for reading, we introduced the concept of a full-bit. Each location in a buffer has a full-bit that the producer clears when the location is added to the write window and sets when a value is written at the location. No atomic read-modify-write operations are required, because only the producer clears and sets the full-bits.

We have demonstrated that computing a sufficient buffer capacity for each buffer in isolation does not always result in deadlock-free execution of the task graph. Therefore, the synchronization performed by all tasks in a task graph is captured in a data flow model, with which sufficient buffer-capacities for deadlock-free execution can be computed.

The presented buffers with overlapping windows enable deadlock-free execution of cyclic task graphs. In the future, we plan to use these buffers for deadlock-free execution of task graphs, with non-affine index expressions, of which the tasks are automatically derived from sequential code.

REFERENCES


