Mode-controlled dataflow based buffer allocation for real-time streaming applications running on a multi-processor without back-pressure
Salunkhe, H.L.; Lele, A.; Moreira, O.; van Berkel, C.H.

Published: 01/01/2015

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Mode-controlled Dataflow based Buffer Allocation for Real-time Streaming Applications Running on a Multi-processor without Back-pressure

Hrishikesh Salunkhe, Alok Lele, Orlando Moreira and Kees van Berkel

15/03

ISSN 0926-4515

All rights reserved
editors: prof.dr. P.M.E. De Bra
prof.dr.ir. J.J. van Wijk

Reports are available at:
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Author&level=1 and
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Year&Level=1

Computer Science Reports 15-03
Eindhoven, July 2015
Mode-controlled Dataflow based Buffer Allocation for Real-time Streaming Applications Running on a Multi-processor without Back-pressure

Hrishikesh Salunkhe*, Alok Lele*, Orlando Moreira†, Kees van Berkel*
* Eindhoven University of Technology, The Netherlands; † Intel Beneleux B.V., The Netherlands
{h.l.salunkhe, a.lele, c.h.v.berkel}@tue.nl; orlando.moreira@intel.com

Abstract

Real-time streaming applications running on embedded systems such as cellular modems in smartphones have to deal with severely constrained memory resources. Buffer allocation for real-time streaming applications, modeled as dataflow graphs, minimizes the total memory consumption while reserving sufficient space for each data production without overwriting any live data and guaranteeing the satisfaction of real-time constraints. We focus on the problem of buffer allocation for systems without back-pressure. Since systems without back-pressure lack blocking behavior at the side of the producer, buffer allocation requires both best- and worst-case timing analysis. Moreover, the dynamic data-dependent (decision making) behavior in these applications makes buffer allocation challenging from the best- and worst-case timing analysis perspective. We argue that static dataflow cannot conveniently express the dynamic behavior of these applications, leading to overallocation of memory resources.

Mode-controlled Dataflow (MCDF) is a restricted form of dynamic dataflow that allows mode switching at runtime and static analysis of real-time constraints. In this paper, we address the problem of buffer allocation for MCDF graphs scheduled on systems without back-pressure. We consider practically relevant applications that can be modeled in MCDF using recurrent-choice mode sequence that consists of the mode sequences of equal length. Analysis of an MCDF graph for recurrent-choice mode sequence provides tractable analysis. Our contribution is a buffer allocation algorithm that achieves up to 36% reduction in total memory consumption compared to the current state-of-the-art for an LTE and an LTE Advanced receiver use cases.

Keywords

Buffer allocation; dataflow; back-pressure; real-time streaming;

I. INTRODUCTION

Current smartphones support multiple radio standards such as GSM, WCDMA, TDSCDMA and LTE [1] running simultaneously. The transceivers for these standards are real-time streaming applications that process a potentially infinite sequence of input data streams, and have strict timing requirements [2]. The processing workload imposed by such an application has increased tremendously since the past decade [3]. In order to meet the workload requirements, a real-time streaming application is often mapped onto a heterogeneous multi-processor platform. Meanwhile, resources such as on-chip memory on such a platform are severely constrained to enable low-cost high-volume markets. Therefore, it is essential to minimize the on-chip memory allocation while meeting the timing requirements of the application.

A real-time streaming application consists of multiple computational tasks that are mapped onto one or more processing elements of a hardware platform. The tasks communicate with each other by producing (or consuming) data values through finite First-In First-Out (FIFO) buffers, which are mapped on a on-chip memory. Buffer allocation for real-time streaming applications involve the minimization of total memory consumption by buffers while reserving sufficient space for each data value production without overwriting any live (not consumed) data values and guaranteeing the satisfaction of real-time constraints. Systems may prevent overwriting of data values by implementing a back pressure mechanism in which a producer task will be blocked from executing until there is a sufficient space available on all of its output FIFO buffers [4]. Consequently, back-pressure influences buffer sizes and task scheduling [5]. However, in systems without back-pressure, a producer task will execute as soon as it is enabled without checking for the availability of space on its output buffers. This may result in buffer overflow, where a producer task overwrites a data value in its output buffer which is not yet consumed by its consumer task. This makes buffer allocation for non-back-pressured systems non-trivial. Moreover, hardware platforms that do not support back-pressure [6] are not uncommon, since back-pressure incurs extra processing and synchronization overheads, and consumes chip area and additional energy [7]. We focus on buffer allocation for systems without back-pressure.

Dataflow is a well-known model of computation that can be used to model, program and analyze real-time streaming applications [2]. A dataflow graph consists of nodes called actors and edges. Actors communicate through edges, which represent FIFO buffers, using tokens. A token is a unit of data transfer between actors. An actor can execute (fire) when there is a sufficient number of tokens available on all of its input edges. Currently, buffer allocation techniques exist for applications modeled as dataflow graphs running on a hardware platform with [5], [8], [4], [9] and without back-pressure [10].
Static dataflow, e.g. Single-rate Dataflow (SRDF, [11]), support rigorous timing analysis of real-time streaming applications. However, it cannot conveniently express the dynamically changing behavior of such applications [1]. In the context of dataflow, in such applications, actor firings and dependencies change dynamically. Moreover, when back-pressure is not supported by the hardware both the best- and worst-case timing behavior must be considered. However, we observe that modeling the best- and worst-case [10] behavior of a real-time streaming application in static dataflow can be very pessimistic, leading to the overestimation of the necessary buffer sizes.

Dynamic dataflow, e.g. Boolean dataflow [12], can easily capture dynamic behavior. However, it cannot be subjected to temporal analysis. Mode-controlled Dataflow (MCDF, [2]) is a restricted form of boolean dataflow, that allows rigorous temporal analysis for real-time streaming applications. In an MCDF graph, a specific sub-graph is chosen per iteration, depending on a mode of its execution. We observe that MCDF not only can model dynamic behavior of applications but also their best- and worst-case timing behavior more accurately.

In this paper, we propose a buffer allocation solution for real-time streaming applications modeled as MCDF graphs running on a heterogeneous multiprocessor platform without back-pressure. We consider MCDF graphs with a class of mode sequence called Recurrent-choice Mode Sequence (RCMS) which consists of mode sequences of equal length; RCMS not only allows to model practically relevant applications but also provides tractable analysis. In Section II, we introduce SRDF and the existing SRDF-based buffer allocation technique. In Section IV, we describe the MCDF model in detail. In Section III-A, we show the advantages of modeling a real-time streaming application in MCDF over SRDF for buffer allocation. In Section IV-C, we describe our buffer allocation technique. Related work is described in Section VI. In Section V, we show that our technique provides up to 36% reduction in memory consumption compared to the existing SRDF-based technique for an LTE and an LTE Advanced receiver use cases.

II. Preliminaries

In this section, we introduce Single-rate Dataflow (SRDF). We also describe our use case: LTE receiver and its SRDF model. A dominant periodic source, one of the important properties for bounded buffers required by an application scheduled in the hardware both the best- and worst-case timing behavior must be considered. However, we observe that modeling the best- and worst-case behavior of a real-time streaming application in static dataflow can be very pessimistic, leading to the overestimation of the necessary buffer sizes.

In this paper, we propose a buffer allocation solution for real-time streaming applications modeled as MCDF graphs running on a heterogeneous multiprocessor platform without back-pressure. We consider MCDF graphs with a class of mode sequence called Recurrent-choice Mode Sequence (RCMS) which consists of mode sequences of equal length; RCMS not only allows to model practically relevant applications but also provides tractable analysis. In Section II, we introduce SRDF and the existing SRDF-based buffer allocation technique. In Section IV, we describe the MCDF model in detail. In Section III-A, we show the advantages of modeling a real-time streaming application in MCDF over SRDF for buffer allocation. In Section IV-C, we describe our buffer allocation technique. Related work is described in Section VI. In Section V, we show that our technique provides up to 36% reduction in memory consumption compared to the existing SRDF-based technique for an LTE and an LTE Advanced receiver use cases.

A. Single-rate dataflow

In Single-rate Dataflow (SRDF) [11], actors have fixed execution times and in each execution (firing), an actor consumes/produces a single token from/to its input/output edges. Fig. 2a shows an example SRDF graph. The initial state of the graph is specified by delays (solid dots). An SRDF graph, extended with the Best-case Execution Time (BCET) and Worst-case [10] behavior of a real-time streaming application in static dataflow can be very pessimistic, leading to the overestimation of the necessary buffer sizes.

Other actors process both the channels: Control (C) and Data (D) channels. Figure 1 shows an LTE receiver SRDF graph. In LTE, data is transmitted in terms of symbols. A sub-frame is a scheduling unit that consists of 14 symbols and is modeled by 14 source actors in the LTE receiver model. Each symbol in a sub-frame, depending on the type of information mapped on it, involves different signal processing. The LTE receiver has two types of channels: Control (C) and Data (D) channels. Actors shown in blue and green colors process C and D channels respectively. Other actors process both the C and D channels together. DMOD_R1 and DMOD_R2 demodulate a symbol for two receiver
antenna $R_1$ and $R_2$; we assume Multiple Input Multiple Output (MIMO) with two receiver antennas [13]. Channel estimates are computed by $CEST$. MIMO computes the response for multiple antennas. Demapper ($DMAP$) demaps symbols to softbits with the help of combined response from MIMO. The demapped symbols are then fed to respective ($CDEC$ or $DDEC$) decoders. $PCPH/DCID$ extracts the location and other $C/D$ channel related information from the sub-frame symbols. $MAC$ is a higher layer interface (sink actor). Note, for $D$ channel, like $DMOD_R2$, $DMOD_R1$ also has the same dependencies towards $DMAP_D$ which are not shown in the figure. Our LTE receiver is scheduled on the platform in [14] and is explained in [15]. Every function such as $DMOD$, $DMAP$, $CDEC$ and $DDEC$ have dedicated processing elements in our platform. The execution order for an actor e.g. $DMAP$ for 14 symbols (i.e. $DMAP_1$, $DMAP_2$, ... $DMAP_{14}$) is enforced by static-order among the copies of $DMAP$ actors, it is shown using red colored edges. For more details on the LTE receiver model, kindly refer to [15].

C. Dominant periodic source

**Definition 1: Dominant Periodic Source** We say that a graph has a dominant periodic source if it has one or more actors with constant execution times that are connected together to form a cycle having exactly one delay, and this cycle has the largest cycle ratio. Also, there must exist an actor in this cycle with a delay-less path to every other actor.

The source actors $Src_1$, $Src_2$, .. $Src_{14}$ shown in Figure 1, model the periodic arrival of symbols, and constitute the dominant periodic source. We call the cycle involving the dominant periodic source the source cycle. Such a graph have $\mu(G, \hat{\tau})$ and $\mu(G, \hat{\tau})$ equal to the cycle mean of the source cycle. This condition enables the graph to keep up with the rate of the source, which is required for a realistic application. Dominant periodic source also guarantees that the graph is connected. A directed graph is connected if replacing all of its directed edges with undirected edges produces a graph that has a path between every node pair.

D. SRDF-based buffer allocation

In this section, we describe the existing SRDF-based buffer allocation technique proposed in [10]. An STS of a strongly-connected (a directed path exists between every pair of actors) graph reaches a periodic regime, after a finite number of iterations (called transient phase) assuming constant execution time for actors across all iterations [16]. However, when back-pressure is not supported on a hardware platform then the resultant graph may not be strongly-connected graph. This property is extended for any valid STS of such a (non-strongly-connected) graph $G$ without back-pressure having a dominant periodic source $src$ [10], and is depicted by Eq. 3.

$$s(i, k + 1, \tau) = s(i, k, \tau) + \mu(src), k > K(G, \tau)$$

An iteration of an SRDF graph is a sequence of actor firings where each actor fires only once. $K(G, \tau)$ is the number of iterations in transient phase and can be computed by simulation. The period of this periodic regime is a cycle ratio of the source cycle ($\mu(src)$). The best- and worst-case STS also have the same period. An SRDF graph having a dominant periodic source will always have finite bounded buffer sizes for non-back-pressured execution.

A sufficient buffer size for an edge $(i, j)$ is the maximum number of tokens having overlapping lifetimes on $(i, j)$ across all iterations. The lifetime of a token on $(i, j)$ starts when it is produced by $i$ and ends when it is consumed by $j$. For an SRDF graph $G$ with a dominant periodic source, the maximum lifetime of any token is finite and bounded by its earliest production
and latest consumption times in the best- and worst-case STS respectively. This bounds the maximum number of live tokens and
the buffer size. The algorithmic steps are explained as follows.

1) **Graph unrolling**: Buffer sizes are computed using simulation. Simulation, in this context, means that we analyze the STS
of a graph and thus obtain the start time of actors. According to Eq.3, the simulation needs to be performed till the end of
the second periodic iteration to cover all possible overlaps of token lifetimes. Let \( n \) be the total number of iterations from the start of
simulation till the last iteration running in parallel with the second periodic iteration. Simulation is performed for the best-case
STS (\( \tilde{n} \)) and the worst-case STS (\( \hat{n} \)) separately. The original graph \( G_i \) is then unrolled \( n = \max(\tilde{n}, \hat{n}) \) times. Graph unrolling
alters a graph \( G \) such that, for \( u \in \mathbb{N} \), \( u \) consecutive iterations of \( G \) are represented as a single iteration in the unrolled graph
\( G_u \) [11].

![Fig. 2: (a) An SRDF graph of a typical radio application (b) 2 times unrolled graph of the graph 2a](image)

Fig. 2a shows an SRDF graph of a typical radio application processing. \( S \) is a dominant periodic source that models a periodic
arrival of symbols. A radio application, generally, consists of control (\( C \)) and data (\( D \)) channels. Actors \( DM, CE, DP \) and
\( Dec \) have similar functionalities as actors \( DMOD, R1/R2, CEST, DMAP \) and \( DDEC/CDEC \) respectively in Fig. 1. Fig. 2b shows the 2 times unrolled graph of the graph shown in Fig. 2a.

2) **Relative lifetime analysis**: The unrolled graph \( G_u \) has multiple copies of an edge from the input graph \( G_i \), which are
mapped on a single FIFO. Moreover, the tokens belonging to different edges of \( G_i \) having the same size can also be mapped on
a single FIFO provided that they follow FIFO ordering [10]. For each edge \((i, j)\) in \( G_u \), its lifetime is obtained using simulation.
The Buffer Interference Graph (BIG), generated using these lifetimes, is a graph \( G_B(V_B, E_B) \), where \( V_B \) and \( E_B \) are the sets of
tokens and edges respectively. If \((l, r) \in E_B\) then \( l \) and \( r \) have overlapping lifetimes, and hence cannot share the same memory
space.

![Fig. 3: Relative lifetime analysis: using best- and worst-case STS](image)

Let us consider an SRDF example shown in Fig. 2a; the BCET and WCET of all actors are shown in Table I. For simplicity,
let us consider only the first iteration of the graph. The lifetime of a token is given by its earliest production in the best-case STS
and its latest consumption in the worst-case STS. In this case, all the tokens have overlapping lifetimes. According to the graph
topology, all the paths to any actor from source \( S \) goes through \( DM \). Therefore, any execution variation occurring before \( DM \)
will shift the start times of all the actors after \( DM \) equally. Consequently, \( DM \) is a relative source or a common dominator of
all the actors except \( S \) and \( A \). Fig. 3 shows the lifetimes for some tokens that are computed with respect to \( DM \). In relative
time analysis, the lifetimes of a pair of tokens are computed with respect to their relative source which results into smaller
lifetimes. This in-turn results in fewer overlapping lifetimes and more opportunities of sharing memory space among tokens.
For instance, the lifetimes of \((DM, CE_D)\) and \((DP_D, Dec)\), computed with respect to \( DM \), do not overlap. Graph unrolling
enables the relative lifetime analysis for actors in different iterations. For every edge in the BIG, the lifetimes of its source
and sink tokens are recomputed using their relative source. If the re-computed lifetimes do not overlap then the edge can be
removed from the BIG. Then, for each edge in the input graph, the maximum number of its copies from its unrolled graph
having interferences with each other in the BIG gives the buffer size needed for that edge. For more details, kindly refer [10].

<table>
<thead>
<tr>
<th>Actor</th>
<th>BCET (( \mu \text{sec} ))</th>
<th>WCET (( \mu \text{sec} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( A )</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>( DM )</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( CE_D )</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( CE_D )</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( DP_D )</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( Dec )</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**TABLE I**: The BCET and WCET of actors shown in Fig. 2a
III. MODE-CONTROLLED DATAFLOW

Mode-controlled Dataflow (MCDF) [2] is a restricted form of boolean dataflow [12] that supports run-time mode switching as well as design-time temporal analysis. In an MCDF graph, in each iteration, based on a mode value produced by a so-called mode controller, actors belonging to a specific sub-graph are fired. A typical MCDF graph is comprised of multiple switch, select, static dataflow actors and a single mode controller actor as shown in Fig. 4a. Modal actor A/B/C fires only in the iterations of mode 1/2/3. Amodal actors S and Z fire in every iteration. In each iteration, a switch consumes a control token from the control input and a data token from the amodal input, and produces the data token on the modal output associated to the mode conveyed by the control token. Conversely, a select consumes a control token from the control input and a data token from the modal input associated to the mode of the control token, and produces the data token on its amodal output. In each iteration, a mode value is passed through a control token from MC to all switches and selects. A tunnel actor \( T \) is a shorthand used to model inter-modal communication. It is comprised of MCDF actors as shown in Fig. 4b. A tunnel passes a token from the iteration of its input mode to the iteration of its output mode. Note that a modal port of a switch/select can be unconnected, in that case, it fires without producing/consuming any token.

![Diagram](image)

Fig. 4: (a) An example MCDF graph with three modes: 1, 2 and 3 (b) Tunnel from mode 1 to mode 3

An MCDF graph \( G \), extended with the BCET and WCET for actors, is denoted by \( G = (V, E, \hat{\tau}, \hat{\tau}, d, M, am, tp) \). \( V, E, \hat{\tau}, \hat{\tau} \) and \( d \) have the same definitions as in SRDF. \( M \) is the number of modes. Valuation \( am: V \rightarrow \{1, ..., M\} \) gives the mode of an actor. Since only some actors execute for specific modes, \( am \) is a partial function. For amodal actors, \( am(i) \) is undefined, denoted by \( am(i) = \bot \). There are some actors in MCDF that have special attributes such as mode controller, switch and select actors. Valuation \( tp: V \rightarrow \{\text{mode controller}, \text{switch}, \text{select}, \text{normal}\} \) gives the type of the actor.

A mode sequence defines an order of modes in which the graph will execute. A recurring mode sequence, introduced in [17], is defined as in Definition 2.

**Definition 2: Recurring Mode Sequence** A recurring mode sequence is defined as a finite, static and self-repeating sequence of modes.

A recurring mode sequence can be captured in regular expression such as \( c = (213)^* \) for the graph shown in Fig 4a. The graph fires for each mode in \( c \) sequentially. After the last mode in \( c \), the graph execution switches back to the first mode in \( c \), and the sequence starts over. Note, the sequence is static and choice is excluded. We use \( c(k) \) to denote the \( k \)th mode value in a recurring mode sequence \( c \). For \( c \), the start time and the execution time of the \( (k+1) \)th firing of actor \( i \) is defined as \( s(i, k, c, \tau) \) and \( \tau(i, k, c) \) respectively. \( s(i, k, c, \tau) \) only exists when \( am(i) = c(k) \) or \( am(i) = \bot \). Otherwise we define \( s(i, k, c, \tau) = \bot \). \( \tau(i, k, c) \) for all recurring mode sequences \( c \), \( \forall i \in V \), \( \forall k \in \mathbb{N}_0 \) is bounded from both sides by \( \hat{\tau}(i) \) and \( \hat{\tau}(i) \) respectively: \( \hat{\tau}(i) \leq \tau(i, k, c) \leq \hat{\tau}(i) \), and written as \( \tau \preceq \tau \preceq \hat{\tau} \). The edge modality \( em(i, j) \) is given by \( am(j) \) if \( am(j) \) is defined else by \( am(i) \).

Generally, the dynamic behavior of an application can be split into one or more static behaviors, where switching from one to another static behavior is performed at runtime. Each such static behavior can be modeled as a recurring mode sequence. In a dynamic behavior, after the last mode in the current recurring mode sequence, the graph execution may non-deterministically switch to the first mode of any of the associated recurring mode sequences and must continue executing until it reaches the end of the recurring mode sequence. We introduce a choice to model such non-determinism in selecting recurring mode sequences using Recurrent-choice Mode Sequence (RCMS); it is captured in regular expression as \( (c_1|c_2|...|c_n)^* \), where \( c_1, c_2, ..., c_n \) are recurring mode sequences. Note, such non-deterministic behavior cannot be modeled in static dataflow such as Cyclo-static Dataflow (CSDF) [18], Synchronous Dataflow (SDF) [19] or SRDF [11]. In this work, we consider practically relevant applications that can be modeled using RCMS having recurring mode sequences of equal length. Unlike fully generalized regular expressions, the analysis of an MCDF graph for RCMS is tractable as we will show in Section IV.

A self-timed schedule of an MCDF graph for a specific recurring mode sequence \( c \) and execution time \( \tau \) is shown in Eq.4.

\[
s(i, k, c, \tau) = \max_{(x, i) \in E} \begin{cases} 
  s(x, k - \delta(k, c, d(x, i)), c, \tau) + \tau(x, k - \delta(k, c, d(x, i)), c), & em(x, i) = c(k) \\
  s(x, k - d(x, i), c, \tau) + \tau(x, k - d(x, i), c), & em(x, i) = \bot \\
  0, & k < d(x, i)
\end{cases}
\]

(4)
where, \( \delta(k, c, d(x, i)) \) is called modal delay. For modal edge \((x, i)\), firing of \(i\) in the \(k^{th}\) iteration depends on the firing of \(x\) in the \((k - \delta(k, c, d(x, i)))^{th}\) iteration. \(i\) fires \(d(x, i)\) times between the \((k - \delta(k, c, d(x, i)))^{th}\) and the \(k^{th}\) iterations in \(c\). The start times for best-case STS and worst-case STS are denoted as \(\tilde{s}(i, k, c)\) and \(\hat{s}(i, k, c)\).

A. Motivation (MCDF Vs. SRDF)

We start with an example radio processing SRDF model shown in Fig. 2a, where we show the control (C) and data (D) channel processing for a single symbol. In a typical radio application, a symbol may carry the C or D channel. SRDF cannot model this behavior conveniently. In SRDF, we must assume the worst-case behavior i.e. a symbol carries both the C and D channels. This leads to the overestimation of memory resources. For instance, in an iteration, a token produced by DM will be replicated on the \((DM, CE_C)\) and \((DM, CE_D)\) buffers, resulting into larger buffer sizes. Similarly, in an iteration, Dec will consume tokens from both \(DP_C\) and \(DP_D\).

Contrarily, for the MCDF model shown in Fig 5, in an iteration, only one of the modes is active and hence DM produces token either on the input of \(C\) or \(D\) depending on the chosen mode. Moreover, Dec will consume token from either \(DP_C\) or \(DP_D\) depending on the chosen mode. This results into smaller buffer sizes.

Moreover, when a back-pressure is not supported, the best- and worst-case behaviors also need to be modeled conservatively in SRDF. For instance, the worst-case start time of Dec is derived by the \(DP\) between \(C\) or \(D\) channel which finishes later than the other. This is modeled by adding dependencies from both \(DP_C\) and \(DP_D\) to Dec. Conversely, in the best-case behavior, best-case start time of Dec is derived by the \(DP\) between \(C\) and \(D\) channel that finishes earlier than the other. This is modeled by setting BCETs of \(CE\) and \(DP\) of \(C\) and \(D\) channels to zero [15]. This conservative modeling guarantees that the actual start time of Dec will never be earlier and later than its best-case and worst-case start time respectively. However, this modeling is pessimistic since it increases the estimated difference in token production and consumption times i.e. token lifetime, with the consequence that may result into larger buffer sizes.

In MCDF, the best- and worst-case timing behavior is more accurate than the SRDF model. In a single iteration and in the best-/worst-case behavior, Dec will be triggered by the earliest/latest firing of \(DP\) belonging to the selected mode. Consequently, MCDF models the data-dependent behavior of a radio application more accurately and thereby can provide smaller buffer sizes compared to SRDF.

IV. MCDF-BASED BUFFER ALLOCATION

The optimal buffer allocation algorithm has exponential time complexity [10]. Therefore, we use a heuristic method, where we extract the SRDF equivalent graphs from the MCDF model to compute the sufficient buffer sizes.

A. Recurring mode sequence graph

For each mode in a recurring mode sequence, the mode controller selects a (modal) subgraph associated to that mode, of an MCDF graph, to fire. For instance, Fig 6b shows a modal subgraph associated to mode 1. Each such modal subgraph is an SRDF graph. We can express the MCDF graph firing for a recurring mode sequence using an SRDF graph; we extract modal subgraphs for all modes and connect them in the sequence defined by the recurring mode sequence such that each inter-iteration dependency, i.e. edges with delays, from the MCDF graph is replicated across modal graphs. We term such a graph a recurring mode sequence graph.
For instance, edge \((S, S)\) in \(G\) (Fig. 6a) is replicated as \((S_1, S_2), (S_2, S_3)\) and \((S_3, S_1)\) in its recurring mode sequence graph \(G_c\) (Fig. 6c). Moreover, the inter-iteration dependencies between the modal actors of a mode are only replicated across the modal subgraphs associated to the same mode. For instance, \((B_1, B_2)\) and \((B_3, B_1)\) in \(G_c\). Note, \(S_1, S_2\) and \(S_3\) constitute a dominant periodic source. The formal conversion procedure of an MCDF into a recurring mode sequence graph is defined as follows:

**Definition 3: (MCDF to recurring mode sequence graph conversion)**

Let \(G = (V, E, r_c, \tau_c, d_m, d_a)\) be a recurring mode sequence graph in an MCDF graph \(G = (V, E, \tau, \hat{\tau}, d, M, ma, tp)\) for a recurring mode sequence \(c\) of length \(L_c\), let \(L = \{0, 1, \ldots, L_c - 1\}\) be the set of integers from 0 to \(L_c - 1\). Then the conversion function \(F: G \times c \rightarrow G_c\) is such that \(G_c = F(G, c)\) where,

\[
\begin{align*}
V_c & = \{v \in V, l \in L | \text{am}(v) \in \{\perp, c(l)\} | v_l\} \\
E_c & = \{(u, v) \in E, l \in L | m = id((u, v), l, L_c) | (u_l, v_m)\} \\
d_c & = \{(u, v) \in E, l \in L | m = id((u, v), l, L_c) | (u_l, v_m), \left((d(u, v) \div L_c + l(l, m))\right)\} \\
\hat{\tau}_c & = \{v \in V, l \in L | \text{am}(v) \in \{\perp, c(l)\} | (v_l, \hat{\tau}(v))\} \\
\hat{\tau}_c & = \{v \in V, l \in L | \text{am}(v) \in \{\perp, c(l)\} | (v_l, \hat{\tau}(v))\}
\end{align*}
\]

where,

\[
\begin{align*}
id((u, v), l, L_c) & = \begin{cases} (l + d(u, v)) \mod L_c, & \text{em}(u, v) = \perp \\ (l + \delta(l, c, d(u, v))) \mod L_c, & \text{em}(u, v) = c(l) \end{cases} \\
t(a, b) & = \begin{cases} 1, & b < a \\ 0, & \text{otherwise} \end{cases}
\end{align*}
\]

The firing of the MCDF graph \(G\) for a recurring mode sequence \(c\) for \(L_c\) iterations is equivalent to firing \(G_c\) for a single iteration. An iteration for \(G_c\) means a single firing of each actor in \(G_c\). Therefore, to differentiate between the iterations of MCDF graph \(G\) and a recurring mode sequence graph \(G_c\), henceforth, we will denote the iteration of an MCDF graph as **mode iteration** and the iteration of an SRDF graph as **iteration**. For an STS of MCDF graph \(G (s(i, k, c, \tau))\) and recurring mode sequence graph \(G_c\) \((s(i, k, c), \tau)\) graphs, by construction of \(G_c\) from \(G\), if \(s(i, k, c, \tau)\) exists then \(s(i_1, k', \tau)\) also exists such that \(s(i, k, c, \tau) = s(i_1, k', \tau)\), where \(l = k \mod L_c\) and \(k' = \lfloor l / L_c \rfloor\). We know that \(s(i, k, c, \tau)\) exists if \(am(i) = c(k \mod L_c)\) or \(am(i) = \perp\). From Definition 3, we know that \(i_t\) exists in \(V_c\), if \(am(i) = c(l)\) or \(am(i) = \perp\). From the Definition 3, Eq. 4 and 1, we know that the \(k^{th}\) firing of \(i\) in \(G\) is equivalent to the \(k^{th}\) firing of the \(t^{th}\) copy of \(i\) in \(G_c\). Therefore, \(s(i, k, c, \tau) = s(i_t, k', \tau)\).

**B. Recurrent-choice mode sequence graph**

During the execution of an MCDF graph \(G\) for the Recurrent-choice Mode Sequence (RCMS) \(C = (c_1, c_2, \ldots, c_n)^*\), one of the recurring mode sequence will be executed after every \(L_c\) mode iterations, where \(L_c\) is the length of each recurring mode sequence \(c\) in \(C\). Let us assume that the \(c_m\) (i.e. \(G_{c_m}\)) is chosen to fire for the current iteration \(k\). If there is an edge \((i, j)\) with \(d(i, j) > 0\) in \(G_{c_m}\) then actor \(i\) in \(G_{c_m}\) will impose a data dependency on actor \(j\) of \(G_{c_m}\), when \(c_n\) is chosen in the \((k + d(i, j))^{th}\) iteration. These inter-iteration (edges with delays) dependencies between recurring mode sequence graphs arise from dynamic behavior and will influence subsequent firings of some actors in future iterations. Consequently, the inter-iteration dependencies influence the earliest production and the latest consumption times of tokens across recurring mode sequences. Therefore, we have to consider the inter-iteration dependencies between all recurring mode sequence graphs to capture the best- and the worst-case behavior.

Let us assume the graph \(G\) shown in Fig. 6a with \(c_1 = (121)^*\) and \(c_2 = (112)^*\). We capture the execution behavior of the MCDF graph for the RCMS \(C = (c_1|c_2)^*\) conservatively, by allowing the selection of both \(c_1\) and \(c_2\) simultaneously [17], as shown in Fig. 7a. Each iteration for a recurring mode sequence \(c\) in Fig. 7a corresponds to the iteration of its equivalent SRDF graph \(G_c\).

1) **Worst-case behavior:** The solid red edges between \(Z_3\) and \(Z_1\), shown in Fig. 7a, depict the dependencies from the last mode to the first mode of the same recurring mode sequence; for instance the edge \((Z_3, Z_1)\) in Figure 6c. In the worst-case behavior, the firing of \(Z_1\) of \(c_1\) in the \(2^{nd}\) iteration depends on the firing of \(Z_3\) in the \(1^{st}\) iteration of the recurring mode sequence between \(c_1\) and \(c_2\) which finishes later than the other. Therefore, we add the dashed red edges, shown in Fig. 7a, from \(Z_3\) of \(c_1|c_2\) in the \(1^{st}\) iteration to \(Z_1\) of \(c_2|c_1\) in the \(2^{nd}\) iteration. Eq.8 gives the precedence constraints between all recurring mode sequence graphs \(G_{c_1}, G_{c_2}, \ldots, G_{c_n}\) of \(G\).
where, $\tilde{s}_{c_p}(j, k) \geq \max_{q \in \{1, \ldots, n\}} (\tilde{s}_{c_q}(i, k - d(i, j)) + \tilde{\tau}_{c_q}(i))$,

$$\tilde{s}_{c_p}(j, k) \geq d(i, j) \geq 1$$

$$\tilde{s}_{c_p}(j, k) = 0, \quad k < d(i, j)$$

(8)

where, $\tilde{s}_{c_p}$ and $\tilde{s}_{c_q}$ give the worst-case start times of actors belonging to $G_{c_p}$ and $G_{c_q}$ ($c_p, c_q \in C$) respectively.

Consequently, we have to consider dependencies between all possible recurring mode sequences. As a result, we create a new graph called the worst-case RCMS graph denoted by $\hat{G}_m$, shown in Fig 7b, where we merge graphs $G_{c_1}$ and $G_{c_2}$ by adding dependencies (dashed edges) between recurring mode sequence graphs. E.g. for each inter-iteration edge in $G_{c_1}\cup G_{c_2}$, an equivalent edge, i.e. having the same number of delays, is added from the source of the original edge in $G_{c_1}\cup G_{c_2}$ to the sink of the original edge in $G_{c_1}\cup G_{c_2}$. Recall Section III, this guarantees that all possible non-deterministic executional dependencies between recurring mode sequences are (conservatively) considered.

Let $L = \{0, 1, \ldots, L_c - 1\}$ be the set of integers from 0 to $L_c - 1$. The worst-case RCMS graph $\hat{G}_m = (\hat{V}_m, \hat{E}_m, \hat{\tau}_m, \hat{d}_m)$ is defined as follows:

$$\hat{V}_m = \bigcup_{c \in C, \hat{v}_c \in G_c} V_c$$

$$\hat{E}_m = \bigcup_{c \in C, E_c \in G_c} E_c \cup \bigcup_{c \in C, \hat{v}_c \in G_c} E_i$$

$$\hat{\tau}_m = \bigcup_{c \in C, \hat{v}_c \in G_c} \hat{\tau}_c$$

$$\hat{d}_m = \bigcup_{c \in C, d_c \in G_c} d_c \cup \bigcup_{c \in C, \hat{v}_c \in G_c} d_i$$

(9)

where, $E_i$ and $d_i$ are the inter-iteration dependencies between all recurring mode sequence graphs respectively. For each edge $(u_i, v_{m}) \in E_c$ with $d_c(u_i, v_{m}) > 0, E_i$ and $d_i$ are defined as follows:

$$E_i = \{c' \in C | \hat{v}_{m} = f_d(G_{c'}, c', v_{m}) \}$$

$$d_i = \{c' \in C | \hat{v}_{m} = f_d(G_{c'}, c', v_{m}) \}$$

where,

$$f_d(G_{c'}, c', v_{m}) = \begin{cases} v'_{m}, & \text{am}(v_{m}) \in \{\bot, c'(m)\}, v'_{m} \in V_{c'} \\ v_{l' \in V_{c'}, l' \in L : l > l' > m, \text{am}(v_{m}) = c'(l')} & \text{am}(v_{m}) \notin \{\bot, c'(m)\}, l \in L, \text{am}(v_{m}) = c'(l) \end{cases}$$

(10)

Consequently, in any STS of an MCDF graph $G$, the start time of actor $i \in G$ in the $k^{th}$ mode iteration can never be larger than the worst-case start time of $i_1 \in \hat{G}_m$ in the $k^{th}$ iteration: $\forall i \in G, \exists \tau \leq \hat{\tau} \leq \tilde{\tau} : s(i, k, c, \tau) \leq \hat{s}(i_1, k')$ where $l = k \mod L_c$, $i_1 \in G_c$ and $k' = \lfloor \frac{l}{L_c} \rfloor$. $s(i, k, c, \tau)$ is an STS of an MCDF graph $G$ and $\hat{s}(i_1, k')$ is the worst-case STS of the worst-case RCMS graph $\hat{G}_m$. 

Fig. 7: (a) Inter-iteration dependencies (b) Worst-case RCMS graph ($\hat{G}_m$) (c) Best-case RCMS graph ($\hat{G}_c$)
2) **Best-case behavior:** In the best-case behavior, for the inter-iteration dependencies shown in Fig. 7a, the earliest firing of \( Z_1 \) of \( c_1 \) in the 2\(^{nd} \) iteration is derived by the earliest firing of \( Z_2 \) in the 1\(^{st} \) iteration between \( c_1 \) and \( c_2 \). This firing constraint is based on the \( \text{min} \) expression, which cannot be represented in dataflow as can be seen from Eq. 1. Therefore, we capture this behavior by ignoring all dependencies between recurring mode sequence graphs. This guarantees that the consumers of these, ignored, dependencies will fire earlier than their actual earliest firing times, which is conservative. The graph shown in Fig. 7c depicts the conservative best-case behavior. Recall from Section II, that every earliest or latest actor firing is constrained by its dominant periodic source. Therefore, the dominant periodic source related dependencies, i.e. solid and dashed black edges with delays shown in Fig. 7c, between the recurring mode sequence graphs are preserved in the best-case behavior. This graph is termed the best-case RCMS graph, denoted by \( \hat{G}_m \).

### C. Buffer computation

In this section, we describe our buffer computation technique. We use the best- \( (\hat{G}_m) \) and worst-case \( (\bar{G}_m) \) RCMS graphs of the MCDF graph to perform relative lifetime analysis. Note that by construction, both the \( \hat{G}_m \) and \( \bar{G}_m \) graphs have the same dominant periodic source, as shown in Fig. 7b and 7c, and hence have the same MCR: \( \mu(\hat{G}_m) = \mu(\bar{G}_m) \). Therefore, the STSs of both the (non-strongly connected) \( \hat{G}_m \) and \( \bar{G}_m \) graphs, after a finite number of iterations, will enter into periodic regimes with the same period assuming constant actor execution times, BCET for \( \hat{G}_m \) and WCET for \( \bar{G}_m \), across iterations [10]. Moreover, the maximum lifetime of a token is finite and bounded by its earliest production time in the BC-STS of \( \hat{G}_m \) and its latest consumption time in the WC-STS of \( \bar{G}_m \). Therefore, the maximum number of live tokens i.e the buffer sizes are also bounded.

1) **Recurring mode sequence based lifetime analysis:** For a token, its lifetime is given by its earliest production time in the best-case STS of \( \hat{G}_m \) and the latest consumption time in the worst-case STS of \( \bar{G}_m \). Since the RCMS graphs mimic the behavior where all the recurring mode sequences are running in parallel, the actor firings for all recurring mode sequences are included in each iteration of these graphs. However, for any iteration, only one of the recurring mode sequences will be chosen to fire in practice. Therefore, while performing lifetime analysis based buffer computation, every recurring mode sequence is considered separately in each iteration.

Let us assume an MCDF graph \( G \) with two recurring mode sequences \( c_1 \) and \( c_2 \), and consider the first two iterations. For the two iterations, four different sequences of the recurring mode sequences are possible, since \( c_1 \) or \( c_2 \) can be selected for the first and second iteration. Let us consider the two cases as shown in Fig. 8: case 1) \( c_1 \) is selected in both the iterations; case 2) \( c_1 \) and \( c_2 \) are selected in the first and the second iteration respectively. Let us assume the case 1 shown in Fig. 8a. Each horizontal black line corresponds to the lifetime of a token on some edge \((i, j)\). We can see that there are at most 4 tokens alive at any time for this case. However, if we see case 2, shown in Fig. 8b, then the maximum number of live tokens are 6. In order to guarantee case 1 and 2, we have to choose the buffer size for \((i, j)\) as 6 tokens. As a result, we have to consider all possible sequences of recurring mode sequences for all iterations to compute buffer sizes.

Recall Section IV-C, since both the \( \hat{G}_m \) and \( \bar{G}_m \) graphs share the same dominant periodic source and hence have the same period. Therefore, recall Section II-D and Eq. 3, we only need to consider iterations till the end of the second periodic iteration. We simulate the \( \hat{G}_m/\bar{G}_m \) with the best-/worst-case STS, and count all the iterations \( n/\bar{n} \) from the start of simulation till the last iteration that is running in parallel with the second periodic iteration. Consequently, \( n = \max(\bar{n}, \hat{n}) \) iterations need to be analyzed to compute the sufficient buffer size. This makes the analysis tractable, in the sense that, we do not need to consider all the iterations to compute buffer sizes.

2) **Relative lifetime analysis:** Recall Section II-D, the \( \hat{G}_m \) and \( \bar{G}_m \) are unrolled for \( n \) iterations, the unrolled graphs are denoted as \( \hat{G}_u \) and \( \bar{G}_u \) respectively, which model all possible execution sequences of recurring mode sequences for \( n \) iterations. Now, the buffer size for each edge can be computed by performing relative lifetime analysis. We use dominator analysis to perform relative lifetime analysis [10]. We say that \( A \) dominates \( B \) (\( A \text{ dom } B \)) if every path from source \( S \) to \( B \) always goes through \( A \). We use two graphs to perform lifetime analysis. Recall Section IV-B, \( \hat{G}_u \), in contrast with \( \bar{G}_u \), does not have
inter-iteration edges, which may yield different dominators for the same actor. Hence, the definition of dominators is adapted as follows: A dominates B if every path from source S to B always goes through A in both $\hat{G}_u$ and $\hat{G}_u$ graphs. The closest common dominator is a relation $\text{cdom}: V \times V \rightarrow V$ such that c is $\text{cdom}$ of a and b if c dominates both a and b and there does not exist actor d which dominates a, b and c. $\text{cdom}$ provides the closest relative source for an actor pair. Algorithm 1 describes our buffer computation algorithm in a functional OComil like programming language syntax. Our buffer allocation algorithm is implemented in Heracles [2], a temporal analysis tool developed at Ericsson.

**Input**: MCDF graph G, set C of recurring mode sequences, unrolled RCMS graphs $\hat{G}_u$ and $\hat{G}_u$, and the best-case STS $\hat{s}$ of $G_u$ and worst-case STS $\hat{s}$ of $G_u$  

**Output**: Total buffer size $B$ for $G$

begin
    for each $(i,j) \in E$ do
        let $E_{(i,j)} = \text{extract\,copies\,of\,][(i,j), \hat{E}_u]$ in
        let $E_{(i,j)} = \text{sort\,increasing\,production\,time\,}(E_{(i,j)}, \hat{s}, \hat{s})$ in
        for each $(i', j') \in E_{(i,j)}$ do
            for each $k \in \{1, 2, \ldots\}$ do
                let $b_{k}(i', j') = \text{maximum\,live\,tokens()}$ in
                let $b_{k}(i', j') = \max_{c \in C} (b_{k}(i', j'))$ in
                let $b(i, j) = \sum_{(i', j') \in E_{(i,j)}} (b(i', j'))$ in
                $B = \sum_{(i, j) \in E} b(i, j)$;

**Algorithm 1**: Buffer size computation

For every edge $(i, j)$ in the MCDF graph G, its copies are extracted from $\hat{G}_u$, and the copies are sorted in the increasing order of their production times. Here, each copy corresponds to a token. Then the maximum number of tokens alive for edge $(i, j)$ are computed by considering all possible sequences of the recurring mode sequences for $n$ iterations as shown in Algorithm 1.

**Input**: $(i', j'), E_{(i,j)}, k, c, \hat{s}, \hat{s}$  
**Output**: Buffer size $B$ for $(i', j')$ in the $k^{th}$ iteration for recurring mode sequence $c$

begin
    let $E_{(i,j), k, c} = \text{extract\,copies\,of\,}(i, j, k, c)$ in
    let $E_{(i,j), k} = \text{extract\,copies\,produced\,after\,}(i, j) E_{(i,j), k}$ in
    let $B = 0$ in
    for each $(i,j, k) \in E_{(i,j), k}$ do
        let $\text{dom} = \text{cdom}(j', i)$ in
        let $\hat{s}_{\text{rlv}}(i) = \hat{s}(i) - \hat{s}(\text{dom})$ in
        let $\hat{f}_{\text{rlv}}(j') = \hat{f}(j') - \hat{s}(\text{dom})$ in
        if $\hat{s}_{\text{rlv}}(i) < \hat{f}_{\text{rlv}}(j')$ then
            let $B = B + 1$ in

**Algorithm 2**: Function $\text{maximum\,live\,tokens}$ in Algorithm 1

The maximum number of tokens whose lifetimes overlap with edge $(i', j')$ for the given recurring mode sequence $c$ and the iteration $k$ are computed by function $\text{maximum\,live\,tokens}$ as described in Algorithm 2. Functions $\hat{f}$ and $\hat{f}_{\text{rlv}}$ give the worst-case and relative worst-case finish times of an actor. For any two edges (tokens) $(i', j')$ and $(i, j)$, their lifetimes do not overlap if $j'$ finishes before $i'$ starts or $j$ finishes before $i'$ starts. Let us consider the first case and let $\text{cd} = \text{ccdom}(j', i)$, we compute the relative best-case start time (earliest production) of $i$ and worst-case finish time (latest consumption) of $j'$ with respect to $\text{cd}$. If the relative best-case start time of $i$ occurs after the relative worst-case finish time of $j'$, then the lifetimes of tokens produced by $i'$ and $i$, do not overlap, and hence they can share the same memory space.

3) Lifetime analysis for switch and select: Recall Section III, data input and output buffers for switch and select reuse the same memory space. The lifetime of a token produced on the input of a switch, in some mode ‘m’, is given by the start time of its source actor and the finish time of the modal actor that is connected to the modal output of mode ‘m’ of the switch.
4) **Lifetime analysis for tunnels**: A tunnel passes tokens from its input actor of one mode to its output actor of another mode. The lifetime of such token starts when it is produced by its producer i.e. input actor of the tunnel and ends when it is consumed by its consumer i.e. output actor of the tunnel. In this way, we compute lifetimes of tokens for each tunnel in $\hat{G}_u$.

Recall Section III-A, $CE_C$ and $CE_D$ consumes the same token from $DM$. In SRDF, memory space will be allocated twice for this token. Also, in MCDF, this situation may arise when involved actors are amodal or belongs to the same mode. We capture this reuse of tokens by using a broadcast ($BRD$) actor. $BRD$ actor broadcasts a token present at its input to all of its outputs. The lifetime of such token starts when it is produced by its producer i.e input actor of the $BRD$ and ends when it is consumed by all of its consumers i.e. all output actors of $BRD$.

V. **Experiments & Results**

In this section, we benchmark our technique using an LTE and LTE Advanced receivers [20]. The buffer computation is performed using the unrolled RCMS graphs. Algorithms developed for dominator based lifetime analysis have a time complexity of $O(n^3)$ for a graph having $n$ actors. Also the length of the transient phase of the RCMS graphs also influence the algorithm’s running time.

A. **LTE receiver**

The SRDF model of an LTE receiver is taken from [15] and is shown in Figure 1. The MCDF model of an LTE receiver is presented in Appendix A. These models are scheduled on the modem platform [14]. We compare the buffer sizes for the LTE receiver using three techniques: 1) the baseline (manual) technique, the buffer sizes are computed by product engineers using simulation, 2) the existing SRDF-based buffer allocation (SRDF) [10] and 3) the MCDF-based buffer allocation (MCDF).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Buffer Sizes (KBytes)</th>
<th>Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>575</td>
<td>-</td>
</tr>
<tr>
<td>SRDF</td>
<td>489</td>
<td>15</td>
</tr>
<tr>
<td>MCDF</td>
<td>433</td>
<td>25</td>
</tr>
</tbody>
</table>

As shown in Table II, the MCDF-based technique reduces the memory consumption by 25% and 11% compared to the manual and the SRDF-based technique respectively. This is because of the more accurate modeling of the LTE receiver by MCDF.

B. **LTE Advanced receiver**

An LTE carrier aggregation is one of the latest features of LTE Advanced [20] that allows to combine up to 5 carriers to enable high peak data rates. There is exactly one primary carrier and one or more secondary carriers. The maximum and minimum offsets in the arrival of symbols of the secondary carriers with respect to the primary carrier are bounded. Therefore, the source cycle of the primary carrier forms the dominant periodic source for all the carriers. A sub-frame is a scheduling unit, that consists of 14 symbols, for an equipment. In our schedule, within a sub-frame, the 1st symbol of the 1st carrier is processed then the 1st symbol of the 2nd carrier is processed and so on till the last carrier, then the 2nd symbol of the 1st carrier is processed, and so on.

![Buffer sizes: SRDF Vs. MCDF](image)

The buffer sizes for LTE Advanced receiver computed using the SRDF and MCDF-based techniques are shown in Fig. 9. The MCDF-based technique achieves from 24% to 36% reduction in memory consumption compared to the SRDF-based technique. This is because, since all the carriers share the same dominant periodic source, the best- and worst-case behavior modeling in SRDF produce pessimistic estimated token lifetimes. On the other hand, the best- and worst-case modeling in MCDF reduce the estimated token lifetimes further, resulting into smaller buffer sizes.
VI. RELATED WORK

The related work is split into two parts: buffer allocation for static and dynamic dataflow based systems, and for systems with and without back-pressure. There are several buffer allocation solutions [21], [5], [4] proposed based on static dataflow. Moreover, approaches presented in [8], [22], [9] describe buffer allocation based on restricted form of dynamic dataflow. However, these buffer allocation techniques cannot handle systems that do not support back-pressure.

In contrast to the technique shown in [10], we perform buffer allocation for applications modeled as MCDF graphs scheduled on systems without back-pressure, which results in significant reduction in buffer sizes.

Several other temporal analysis tools such as SymTA/S [23] and RTC [24] support an execution model without back-pressure and also perform buffer sizing. However, SymTA/S can only handle specific cases of cyclic dependencies whereas our technique can handle general cyclic dependencies.

VII. CONCLUSION

In this paper we provide a buffer allocation solution for applications modeled as Mode-controlled Dataflow (MCDF) graphs running on hardware without back-pressure. We consider MCDF graphs with Recurrent-choice Mode Sequence (RCMS) that consists of the mode sequences of equal length: RCMS not only allows to model practically relevant applications but also provides tractable analysis. We capture the best- and worst-case timing behavior of an MCDF graph using two Single-rate Dataflow (SRDF) graphs that provide more accurate estimation of token lifetimes, which results in up to 36% reduction in the memory expenditure compared to the existing SRDF-based buffer allocation technique for an LTE and an LTE Advanced receiver use cases.

REFERENCES

[14] M. Breschel et al., “10.8 a multi-standard 2g/3g/4g cellular modem supporting carrier aggregation in 28nm cmos,” in ISSCC.

APPENDIX

In this appendix, we describe the MCDF model of an LTE Receiver. In LTE, data is transmitted in terms of symbols. A sub-frame is a scheduling unit that consists of 14 symbols. Each symbol in a sub-frame, depending on the type of information mapped on it, involves different signal processing. The LTE receiver has two control ($C_1$ and $C_2$) channels and one data ($D$) channel. The $C_1$ channel carries a message containing information about the structure and size of $C_2$ channel. $C_2$ channel carries a message that includes resource assignments (i.e. information about $D$ channel) and other control information for one or more
user equipments. The $D$ channel is the main shared (among all UEs) data channel which carries all the user data. Henceforth, wherever necessary, we will refer both $C_1$ and $C_2$ channels as $C$ channel. In LTE, a special Reference Signals (RS) are used to facilitate channel estimation and timing synchronization.

**TABLE III: Sub-frame formats: channel mappings on the 14 symbols of a sub-frame**

<table>
<thead>
<tr>
<th>Sub-Frame formats</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2 to 14</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1 to 2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1 to 3</td>
</tr>
</tbody>
</table>

In LTE Receiver, variation in the mapping of $C_1$, $C_2$ and $D$ channels to the symbols in a sub-frame give rise to different sub-frame formats. Table III shows different sub-frame formats i.e. mapping (location) of channels on symbols for a sub-frame.

Fig. 10 shows the MCDF model of the LTE receiver. Recall Section II-B, all the LTE receiver related actors in the MCDF model have the same functionality as the actors shown in Fig. 1. We generalize switches and selects by allowing assignment of a mode set to a modal edge instead of a single mode. We call modal edges and modal actors as multi-modal edges and multi-modal actors. Each multi-modal edge of a switch/select is annotated with the mode set it belongs to. Mode sets of any two edges of a switch/select are disjoint. Union of mode sets of all the edges of a switch/select is the mode set of the input graph. For each multi-modal actor, all input/output edges must have the same mode set. We use these multi-modal abstractions when multiple modes have the same behavior thereby abstracting the graph complexity. A conversion of multi-modal MCDF graph into canonical MCDF graph is described in [1].

In our modeling work, to keep the model simple, we will not show the control edges from the mode controller. Also, we do not show unconnected modes of a switch/select. Recall Section II-B, the red edges depict static-order among actors.
TABLE IV: Modes of the LTE receiver MCDF model

<table>
<thead>
<tr>
<th>Mode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>process the symbols of the $C_1$ channel (this symbol has an RS.)</td>
</tr>
<tr>
<td>$C_2$</td>
<td>process the symbols of the $C_2$ channel (these symbols always have RSs.)</td>
</tr>
<tr>
<td>$C_{2L}$</td>
<td>process the last symbol of the $C_2$ channel that does not have an RS</td>
</tr>
<tr>
<td>$C_{2LR}$</td>
<td>process the last symbol of the $C_2$ channel that has an RS</td>
</tr>
<tr>
<td>$D_R$</td>
<td>process the symbols of the $D$ channel that do not have RSs</td>
</tr>
<tr>
<td>$D_L$</td>
<td>process the last symbol of the $D$ channel (this symbol does not have an RS.)</td>
</tr>
<tr>
<td>$D_{rop}$</td>
<td>brings $CEST$ stage 4/6 symbols ahead of the $C/D$ channel processing stage</td>
</tr>
</tbody>
</table>

Table IV shows the modes associated to the LTE receiver MCDF model. Mode $C_1$ process the $C_1$ channel. Modes $C_2$, $C_{2L}$ and $C_{2LR}$ process the symbols having $C_2$ channel. Modes $D$, $D_R$ and $D_L$ process the symbols having $D$ channel. Subscript $L$ denote the processing of the last symbol of the associated channel, and the subscript $R$ denote the processing of the symbol having an RS of the associated channel. $CEST$ only consumes the symbols from $DMOD$ that have an RS, whose positions in any sub-frame are fixed. Moreover, this behavior forces the $CEST$ stages for $C$ and $D$ channels to run 4 and 6 symbols ahead of their respective decoding stages respectively [15]. For instance, recall Section II-B, in Fig. 1, for $C/D$ channel, $DMAP_{C5}/DMAP_{D8}$ produces softbits of the 1st/2nd symbol. This is achieved in the MCDF model by putting 6 and 2 delays on ($DMOD_{R1}/R2$, $SW_{ODMP_{R1}CD/R2}$) and ($BRD$, $SW_{COWC_{C}}$) edges. $D_{rop}$ mode fires 6 times and this happens only once at the start of the graph execution; this brings the $CEST$ stage 4 and 6 symbols ahead of the $C$ and $D$ channel processing stage respectively. Consequently, 6 and 2 delays present on ($DMOD_{R1}/R2$, $SW_{ODMP_{R1}CD/R2}$) and ($BRD$, $SW_{COWC_{C}}$) respectively are discarded.

TABLE V: Recurring mode sequences

<table>
<thead>
<tr>
<th>Name</th>
<th>Recurring mode sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>($D_R$ $D$ $D_R$ $D_R$ $D_D$ $C_1$ $C_2$ $C_{2L}$ $D_D$ $D_D$ $D_R$)</td>
</tr>
<tr>
<td>$R_2$</td>
<td>($D_R$ $D$ $D_R$ $D_R$ $D_R$ $D_D$ $C_1$ $C_{2LR}$ $D_D$ $D_D$ $D_R$)</td>
</tr>
<tr>
<td>$R_3$</td>
<td>($D_R$ $D$ $D_R$ $D_R$ $D_D$ $C_{2LR}$ $D_R$ $D_D$ $D_R$ $D_R$)</td>
</tr>
</tbody>
</table>

Table V shows the recurring mode sequences associated to the LTE receiver model. Each recurring mode sequence corresponds to a sub-frame format. These recurring mode sequences denote processing with respect to the decoding stage; for instance, the first $C$ symbol is decoded at the seventh (symbol) mode which is denoted by the seventh location ($C_1$) in the given recurring mode sequences. For more details on the modeling, kindly refer [1].
In this series appeared (from 2012):

12/01 S. Cranen  
Model checking the FlexRay startup phase

12/02 U. Khadim and P.J.L. Cuijpers  
Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP

12/03 M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher  
Revised budget allocations for fixed-priority-scheduled periodic resources

12/04 Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever  
Experience Report on Designing and Developing Control Components using Formal Methods

12/05 Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse  
A cure for stuttering parity games

12/06 A.P. van der Meer  
CIF MSOS type system

12/07 Dirk Fahland and Robert Prüfer  
Data and Abstraction for Scenario-Based Modeling with Petri Nets

12/08 Luc Engelen and Anton Wijs  
Checking Property Preservation of Refining Transformations for Model-Driven Development

12/09 M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and J. Nolte  
Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version

12/10 Milosh Stoljki, Pieter J. L. Cuijpers and Johan J. Lukkien  
Efficient reprogramming of sensor networks using incremental updates and data compression

12/11 John Businge, Alexander Serebrenik and Mark van den Brand  
Survival of Eclipse Third-party Plug-ins

12/12 Jeroen J.A. Keiren and Martijn D. Klabbers  
Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2

12/13 Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever  
Evaluating the Effect of Formal Techniques in Industry

12/14 Ammar Osaiweran, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever  
Incorporating Formal Techniques into Industrial Practice

13/01 S. Cranen, M.W. Gazda, J.W. Wesselink and T.A.C. Willemse  
Abstraction in Parameterised Boolean Equation Systems

13/02 Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse  
Decomposability in Formal Conformance Testing

13/03 D. Bera, K.M. van Hee and N. Sidorova  
Discrete Timed Petri nets

13/04 A. Kota Gopalakrishna, T. Ozelebi, A. Liotta and J.J. Lukkien  
Relevance as a Metric for Evaluating Machine Learning Algorithms

13/05 T. Ozelebi, A. Weffers-Albu and J.J. Lukkien  
Proceedings of the 2012 Workshop on Ambient Intelligence Infrastructures (WAmlI)

13/06 Lotfi ben Othmane, Pelin Angin, Harold Weffers and Bharat Bhargava  
Extending the Agile Development Process to Develop Acceptably Secure Software

13/07 R.H. Mak  
Resource-aware Life Cycle Models for Service-oriented Applications managed by a Component Framework

13/08 Mark van den Brand and Jan Friso Groote  
Software Engineering: Redundancy is Key

13/09 P.J.L. Cuijpers  
Prefix Orders as a General Model of Dynamics
<table>
<thead>
<tr>
<th>Date</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>14/01</td>
<td>Jan Friso Groote, Remco van der Hofstad and Matthias Raffelsieper</td>
<td>On the Random Structure of Behavioural Transition Systems</td>
</tr>
<tr>
<td>14/02</td>
<td>Maurice H. ter Beek and Erik P. de Vink</td>
<td>Using mCRL2 for the analysis of software product lines</td>
</tr>
<tr>
<td>14/03</td>
<td>Frank Peeters, Ion Barosan, Tao Yue and Alexander Serebrenik</td>
<td>A Modeling Environment Supporting the Co-evolution of User Requirements and Design</td>
</tr>
<tr>
<td>14/04</td>
<td>Jan Friso Groote and Hans Zantema</td>
<td>A probabilistic analysis of the Game of the Goose</td>
</tr>
<tr>
<td>14/05</td>
<td>Hrishikesh Salunkhe, Orlando Moreira and Kees van Berkel</td>
<td>Buffer Allocation for Real-Time Streaming on a Multi-Processor without Back-Pressure</td>
</tr>
<tr>
<td>14/06</td>
<td>D. Bera, K.M. van Hee and H. Nijmeijer</td>
<td>Relationship between Simulink and Petri nets</td>
</tr>
<tr>
<td>14/07</td>
<td>Reinder J. Bril and Jinkyu Lee</td>
<td>CRTS 2014 - Proceedings of the 7th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems</td>
</tr>
<tr>
<td>14/08</td>
<td>Fatih Turkmen, Jerry den Hartog, Silvio Ranise and Nicola Zannone</td>
<td>Analysis of XACML Policies with SMT</td>
</tr>
<tr>
<td>14/09</td>
<td>Ana-Maria Şutîi, Tom Verhoeff and M.G.J. van den Brand</td>
<td>Ontologies in domain specific languages – A systematic literature review</td>
</tr>
<tr>
<td>14/10</td>
<td>M. Stolikj, T.M.M. Meyfroyt, P.J.L. Cuijpers and J.J. Lukkien</td>
<td>Improving the Performance of Trickle-Based Data Dissemination in Low-Power Networks</td>
</tr>
<tr>
<td>15/01</td>
<td>Önder Babur, Tom Verhoeoff and Mark van den Brand</td>
<td>Multiphysics and Multiscale Software Frameworks: An Annotated Bibliography</td>
</tr>
<tr>
<td>15/02</td>
<td>Various</td>
<td>Proceedings of the First International Workshop on Investigating Dataflow In Embedded computing Architectures (IDEA 2015)</td>
</tr>
<tr>
<td>15/03</td>
<td>Hrishikesh Salunkhe, Alok Lele, Orlando Moreira and Kees van Berkel</td>
<td>Mode-controlled Dataflow based Buffer Allocation for Realtime Streaming Applications Running on a Multi-processor without Back-pressure</td>
</tr>
</tbody>
</table>