Automatic Dataflow Model Extraction from Modal Real-Time Stream Processing Applications

Stefan J. Geuns Joost P.H.M. Hausmans
University of Twente
{stefan.geuns,joost.hausmans}@utwente.nl

Marco J.G. Bekooij
NXP Semiconductors/University of Twente
marco.bekooij@nxp.com

Abstract

Many real-time stream processing applications are initially described as a sequential application containing while-loops, which execute for an unknown number of iterations. These modal applications have to be executed in parallel on an MPSoC system in order to meet their real-time throughput constraints. However, no suitable approach exists that can automatically derive a temporal analysis model from a sequential specification containing while-loops with an unknown number of iterations.

This paper introduces an approach to the automatic generation of a Structured Variable-rate Phased Dataflow (SVPDF) model from a sequential specification of a modal application. The real-time requirements of an application can be analyzed despite the presence of while-loops with an unknown number of iterations. It is shown that an algorithm that has a polynomial time computational complexity can be applied on the generated SVPDF model to determine whether a throughput constraint can be met. The enabler for the automatic generation of an SVPDF model is the decoupling of synchronization between tasks that contain different while-loops. A DVB-T radio transceiver illustrates the derivation of the SVPDF model.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Design, Theory

Keywords Real-time; Dataflow; MPSoC; Automatic Parallelization

1. Introduction

Embedded stream processing applications, such as Software Defined Radio (SDR) transceivers, are often executed on Multiprocessor System-on-Chips (MPSoCs) to meet their real-time performance requirements. These requirements are a result of interaction with the environment. In order to analyze whether these real-time requirements are met by an application, an analysis model is required. For signal processing applications without modes the synchronous data flow (SDF) model [12] is often used. However, SDF models cannot model applications containing modes. Modal behavior can be found in modern SDR transceivers, making the SDF model unsuitable to model these SDR transceivers.

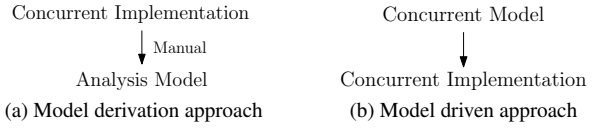
LCTES'13, June 20–21, 2013, Seattle, Washington, USA.

Various more expressive models have been proposed such that modes and mode transitions can be analyzed. Examples of such models are Scenario Aware Dataflow (SADF) [16] and Variable-rate Phased Dataflow (VPDF) [19]. Although these models are more expressive than SDF models, they suffer from a higher computational complexity or apply over-approximation to decrease the analysis time.

Even for the more expressive SADF and VPDF models, it remains a challenge to express an application such that it fits in the dataflow model. Another difficulty is that the dataflow model is often large, making it cumbersome to derive such a model manually. Furthermore, it is virtually impossible to show that a manually derived model is correct. Moreover, it is also hard to keep the model consistent with the application in case the application is modified during the design process.

This paper introduces the automatic generation of a Structured Variable-rate Phased Dataflow (SVPDF) analysis model, which is a temporal analysis model suitable to model applications containing if-statements and while-loops. The SVPDF model is similar to the VPDF model in the sense that both allow infinite iteration of actors, which is described by a parameterized consumption and production of the actors. By generating the model automatically, it is ensured that the model is correct by construction and it saves the programmer from having to rewrite the algorithm such that it can be modeled by an analysis model. The automatic generation of an SVPDF model from a sequential application is enabled by the exploitation of non-destructive read and destructive write semantics of variables inside a while-loop. This results in a decoupling of the synchronization of tasks that have statements in different while-loops. We limit the variables to scalars in this paper and exclude arrays. Our experience is that a variety of single-rate SDR applications only use scalars and this restriction increases the understandability of the concepts and proof. The extracted SVPDF model has a specific structure since it is extracted from a sequential specification. In this paper we show that this structure simplifies analysis significantly.

This paper is organized as follows. Section 2 outlines the different design approaches and motivates the approach proposed in this paper where a parallel implementation and a temporal analysis model are derived from a sequential specification. In Section 3 we describe the basic idea behind our approach. Section 4 highlights the important aspects of our sequential programming language and Section 5 explains the parallelization step in which synchronization statements are inserted such that an SVPDF temporal analysis model can be derived. This model is introduced in Section 6, and this section also explains the derivation of such a model from a task graph. In Section 7 we describe the analysis of SVPDF graphs. Section 8 illustrates the presented approach with a DVB-T application and Section 9 presents related work. Finally, Section 10 states the conclusions.



(a) Model derivation approach (b) Model driven approach

Figure 1: Traditional design approaches for real-time applications

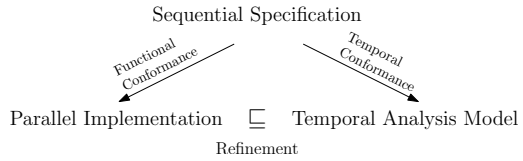


Figure 2: Proposed design approach

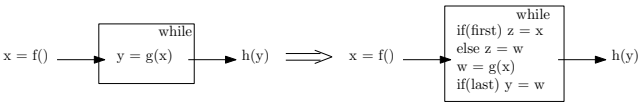


Figure 3: Decoupled synchronization

2. Design Approaches

Different design approaches can be applied to design concurrent stream processing applications. These approaches are outlined in this section, including a discussion and motivation of the approach taken in this paper.

Traditionally when software is designed, an application is written in a sequential or parallel programming language. When throughput constraints have to be taken into account, an analysis model is extracted by the programmer and this model is then analyzed. This traditional approach is illustrated in Figure 1a. A serious disadvantage of such an approach is that extracting an analysis model manually is very error prone and when the implementation changes, the programmer must manually update the model to keep it consistent. An even more serious drawback is that there are usually no guarantees that the model corresponds correctly with the implementation, thus any analysis results might not reflect the properties of the implementation.

Nowadays, model driven approaches are more often being used to implement applications with temporal constraints on MPSoCs. An example of such an approach is the PTIDES approach [6, 20]. An application is described as a concurrent model and from that model, a concurrent implementation is derived. Generating the implementation automatically ensures that the model and the implementation are consistent. This model driven approach is illustrated in Figure 1b. However, in a concurrent specification it is difficult to find a balance between expressiveness and analytical opportunities. For example deadlock can often be specified while analysis tools cannot always detect this. Reducing the expressiveness of a model might have as a consequence that practical applications can no longer be modeled.

To overcome the problems of the approaches mentioned in the previous paragraphs, the approach taken in this paper is based on the parallelization of sequential applications. An application is specified using a sequential language and a tool extracts a temporal analysis model from the sequential specification. This temporal analysis model abstracts from any functional behavior, except for the properties required for temporal analysis. At the same time, the tool also automatically extracts task-level parallelism from this specification such that the functional behavior of the parallel implementation and the sequential specification are equivalent. By deriving both the implementation and model from the same sequential specification, it is ensured that the implementation refines the temporal analysis model and thus satisfies the temporal requirements whenever the analysis model satisfies these requirements. This refinement relation $\sqsubseteq$ is the earlier-is-better refinement relation defined in [7]. The approach taken in this paper is illustrated in Figure 2.

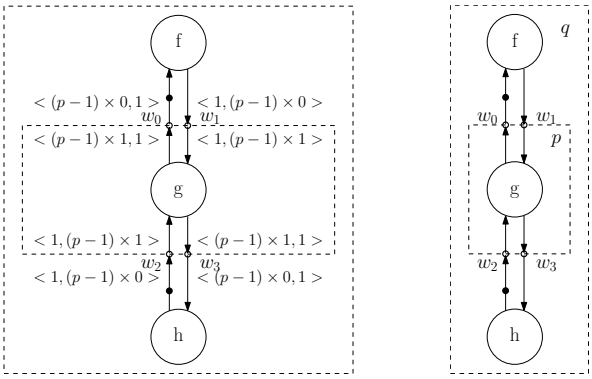
This separation into a functional implementation and temporal analysis model allows for more flexibility in both the implementation and analysis. More flexibility in the implementation is exploited by separating synchronization and communication. For if-statements, synchronization statements are executed unconditionally, whereas communication statements are executed conditionally. In the temporal analysis model only synchronization statements are modeled, not communication statements. Modeling unconditional synchronization statements prevents that the analysis model must be a model with conditions, such as the Boolean Dataflow (BDF) model [4], in which detecting deadlock is undecidable in general. Our temporal model does contain the conditional repetition of while-loops. However, despite this conditional execution deadlock and buffer sizing remain decidable.

3. Basic Idea

The approach taken in this paper is based on three key ideas. To provide an intuitive idea of our approach, these key ideas are presented informally in this section. In the following sections, these key ideas are described in more detail.

The first idea behind our approach is that in the generated implementation, synchronization is decoupled between variables accessed by statements in and around each while-loop. In every task extracted from a function in the sequential application, synchronization has to be inserted to ensure correct communication of data. Previously, correct communication was ensured by inserting synchronization in all tasks accessing a shared variable. When variables were accessed by statements in- and outside of a while-loop, synchronization must be performed for all of these accesses. Therefore, synchronization was performed at the same rate, thus coupling tasks extracted from these statements. This paper introduces an approach were synchronization can be decoupled by making use of the fact that statements around a while-loop can only read or write shared variables once with respect to this loop. By renaming the names of shared variables accessed by statements in a while-loop and adding a selection code fragment which selects between this new and the existing variable, decoupling and therefore a rate conversion is achieved. This process is illustrated in Figure 3. A more detailed discussion on how decoupling is achieved can be found in Section 5.4.

When this synchronization decoupling method is applied, it must be shown how a valid temporal analysis model can be derived such that the real-time constraints can be verified. The decoupling method is in essence implementing non-destructive read and destructive write semantics for variables accessed in and outside of while-loops. Using the variable rates of the VPDF model, non-destructive read and destructive write semantics can be modeled. A parameter can be used to indicate how often a variable is read or written. The VPDF model which models this behavior is shown in Figure 4a. The model contains hierarchical blocks, shown as dashed rectangles, with port actors w_0, w_1, w_2 and w_3 on the edges. These port actors have a variable number of phases, given by the parameter p in the example, and thus model non-destructive read and destructive write semantics. The actual number of phases is not known at compile-time. In the figure, the second phase of the edge from w_1 to g is executed $p - 1$ times and the number of tokens produced is $(p - 1) \times 1 = p - 1$ while only a single token is consumed by w_1. If a VPDF model is structured with hierarchical blocks with port actors on the edges, we call it an SVPDF model. Since the structure of a block where all port actors share the



(a) Full notation (b) Shorthand notation

Figure 4: SVPDF model with port actors w_0, w_1, w_2 and w_3, all sharing the parameter p

same parameter is always the same in an SVPDF model, we introduce the shorthand notation from Figure 4b where the parameter is shown in the top right corner of a block. This idea of modeling non-destructive read and destructive write semantics using variable phases is the second key idea of the presented approach.

The third key idea is that the structure in the generated SVPDF model can be exploited in such a way that a temporally equivalent static analysis model can be derived from this parameterized model. In the static model values of one can be chosen for all parameters, thus effectively removing all parameters from the model and simplifying analysis significantly. After showing in Section 7.1 that the throughput of an SVPDF model can be determined by assuming that all while-loops execute only once, sufficient buffer capacities can be derived for a given throughput.

4. Sequential Program Specification

The input of our parallelization tool Omphale is a sequential program specified in the Omphale Input Language (OIL) [9]. This section highlights the for this paper relevant aspects of this programming language. The OIL language is a single assignment language with support for if-statements and while-loops. These statements can have data-dependent behavior, thus the if-condition and while-loop termination conditions can be dependent on input data. Including these conditional statements allows for the description of modes, often present in stream processing applications.

The real-time throughput constraints of a stream processing application are imposed by a periodic communication with the environment. This communication is described in the OIL language with sources and sinks [9]. The environment is sampled periodically by a source, which passes its data to the application. After a sample is processed, data is periodically fed back to the environment via a sink. Sources and sinks are executed time triggered in our approach, while the algorithmic part executes event-driven [11]. Since the algorithmic does not communicate with the environment, no deadlines have to be determined for this part. The only requirement is that data is delivered in time at a sink.

A source and sink are specified in parallel to the algorithmic part of an application. Despite the parallel specification, a sequential semantics can be defined for sources and sinks. A source is assigned a new value by the environment at the beginning of a while-loop and a value is read from a sink by the environment at the end of a while-loop.

The most important reason to define the sequential semantics where every loop iteration a new value is processed, is that it can be verified at compile time whether an application will meet its throughput constraints. This also means that all values delivered by a source are processed and all values delivered to a sink will be seen by the environment. Since a while-loop can have infinitely many iterations, it must be that all sources and sinks are accessed in every loop. Otherwise, there could be an infinite amount of time between accesses, which makes it impossible to guarantee a periodic execution of sources and sinks.

5. Automatic Parallelization

In our approach, a sequential OIL specification is automatically parallelized into a task graph. The parallelization step extracts function level parallelism such that a task is created for each function in the input specification. This is unlike methods where data parallelism is achieved by unrolling a while-loop. Every task communicates with its connected tasks via Circular Buffers (CBs). Synchronization statements are inserted into each task to ensure that the functional behavior from the sequential specification is preserved. From these synchronization statements, the SVPDF model will be derived in Section 6. Section 5.1 explains how the synchronization statements operating on these CBs work. The following sections show how synchronization statements are inserted into tasks for sources and sinks, if-statements and while-loops, such that the parallelized task graph has the same functional behavior as the sequential input specification.

A task graph is a directed graph which can contain cycles. Formally, a task graph is defined as $H = (T, B, \beta)$. Here T is a set of tasks. For convenience, we define T_S as the set of source or sink tasks, with $T_S \subseteq T$. The set of hyperedges $B \subseteq \mathcal{P}(T) \times \mathcal{P}(T)$ represents the CBs and $\beta : B \to \mathbb{N}$ specifies the sizes of these CBs. From every function f in the sequential OIL program a task t_f is extracted and added to T. For every variable x in the program, a hyperedge b_x is added to B. The incoming connections to b_x are all tasks in which the corresponding functions assign a value to x. In the example these are tasks t_f and t_g. The outgoing connections from b_x are all tasks in which the corresponding functions read from the variable x, tasks t_h and t_k in the figure.

5.1 Circular Buffers

The communication buffers to handle inter task communication are CBs with sliding windows [3]. These buffers can be accessed by multiple reading and writing tasks while preserving any possibility for pipelining. However, there can be only one task writing to a location in a buffer per iteration. An iteration ends when all reading tasks no longer require a location. This requirement, equivalent to single assignment in the sequential specification, ensures that no data races occur between writing tasks.

Every task reading from or writing to a buffer can access that buffer within a window. The head of a window of a writing task (a producer) is moved one place by calling the synchronization function $acqProd$. The tail of this window is moved one place when a producer calls $relProd$. For a task reading from a buffer (a consumer) the equivalent operations are called $acqCons$ and $relCons$ respectively.

Both $acqProd$ and $acqCons$ are blocking functions, meaning they wait until the location next to the current head of the window is empty or full respectively. The $relProd$ and $relCons$ functions are non-blocking. Every producer and every consumer must move its window an equal number of times. If one producer or consumer does not synchronize on a location, and thereby not move its window, all other windows will become blocked once they need to synchronize for this location the next time. Since a synchronization statement moves a window by one location and all windows must be moved the same number of times, every synchronization statement on a buffer must be called the same number of times in every task.

Figure 5 shows an example of the operations that can be performed on a buffer b_x. This buffer is connected to one producer, t_A, and one consumer, t_B. Every buffer location written by t_A is first

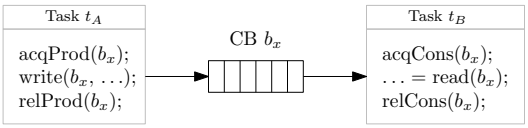


Figure 5: Operations on a CB with sliding windows.

acquired and released afterwards. The same holds for the consumer t_B. Writing to and reading from a CB is performed via $write$ and $read$ functions.

5.2 Sources and Sinks

Since sources and sinks are specified in parallel with the rest of the application, they do not have to be parallelized. However, also for the implementation of sources and sinks, synchronization statements have to be inserted in order to ensure correct functional behavior. The semantics of a source are such that the environment is sampled exactly once in every while-loop iteration. For a sink, a value is sent to the environment once in every while-loop iteration.

A source and a sink are specified by a variable and a function from which respectively a buffer and a task are created during parallelization [9]. However, a problem arises with synchronization. Consider the application from Figure 6a where a source A is read by two functions f and g. The task graph of this application is shown in Figure 7 on the left. Since the tasks t_f and t_g read from the same buffer, their synchronization rates must be coupled and thus both tasks have to synchronize, even if the task is not executing its corresponding function. In order to overcome this coupling of synchronization rates, we propose to transform the task graph such that a buffer is created for every consumer of a source and producer to a sink. The source then writes it data to, and the sink reads its data from the buffer connected to the task that is connected according to the input specification. This transformed task graph is shown in Figure 7 on the right.

Figure 6 shows how this transformation is applied in an example. When a task t_h is created for a source or sink, the structure from the input specification is copied to t_h. This process is shown for task t_h in Figure 6b. The figure shows that the three while-loops are copied and, as described in the previous paragraph, two buffers b_{A1} and b_{A2} are added to pass the source values to the functions requiring these values. The functions f and g now read from b_{A1} and b_{A2} respectively such that the functional behavior of the application remains the same. The tasks t_f and t_g created from these functions are shown in Figures 6c and 6d.

5.3 Parallelization of If-Statements

An if-statement makes assignments to and reading of variables conditional. After the parallelization to a task graph where communication between tasks is arranged via buffers with sliding windows, reading from and writing to the buffer is also conditional. Since the condition of an if-statement can be input-data dependent, it cannot be determined at compile-time whether and when a variable is accessed.

When synchronization is performed by a task on a CB, all tasks must eventually synchronize on this CB, thus all tasks must synchronize the same number of times on a CB. This would mean that if a task conditionally synchronizes on a CB, all tasks must conditionally synchronize. Therefore, all tasks should contain this if-statement. In order to prevent that all if-statements have to be included in all tasks, synchronization is made unconditional. Whenever synchronization statements are added to a branch of an if-statement, they are also added to the else-branch, thus resulting in unconditional synchronization. This approach for inserting synchronization for if-statements is detailed in [3].

```
source A = h();

loop{
  loop{
    f(A);
  } while(C1);

  loop{
    g(A);
  } while(C2);
} while(1);
```

(a) Input program

```
do{
  do{
    acqProd(bA1);
    write(bA1, h());
    relProd(bA1);
  } while(C1);

  do{
    acqProd(bA2);
    write(bA2, h());
    relProd(bA2);
  } while(C2);
} while(1);
```

(b) Task t_h

```
do{
  do{
    acqCons(bA1);
    f(read(bA1));
    relCons(bA1);
  } while(C1);
} while(1);
```

(c) Task t_f

```
do{
  do{
    acqCons(bA2);
    g(read(bA2);
    relCons(bA2);
  } while(C2);
} while(1);
```

(d) Task t_g

Figure 6: Parallelization of an application with a source

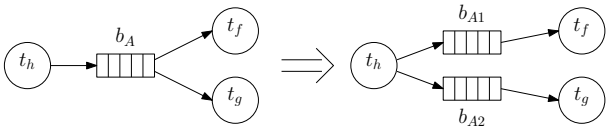


Figure 7: Task graph transformation to decouple synchronization between reading tasks

5.4 Parallelization of While-Loops

In this section, we present a method for inserting synchronization in while-loops. The basic idea behind the presented method is that a value written by an assignment before a while-loop is only needed in the first iteration of a while-loop and only the last value written by an assignment in a while-loop is visible after the while-loop. Because of the limited lifetime of variables, values are lost every loop iteration independent of whether they are written in an iteration or not [8].

For a variable written in an assignment before and in a while-loop, two new variables are created to replace the variable accessed in the loop. One variable is used for writing and one variable for reading. It is known that a variable in the loop can be written only in the next iteration due to single assignment, all occurrences where the original variable is written are be replaced by the same variable. The newly created variable used for reading, is assigned either the value assigned to the existing variable written before the loop or the value of the new variable written in the loop.

This process is illustrated by the example in Figure 8. Here Figure 8a shows an OIL program with a while-loop and Figure 8b shows the result of the aforementioned transformation process. The assignment to x' means that the value made available for the next while-loop iteration. In the program the variable x is written by an assignment before and in the inner while-loop. Therefore, the access to x by the function h is replaced by a fresh variable z. The function g reads from x in the input program and therefore a selection is made whether to read from x or z. This choice is made by the if-statement in the transformed program. Note that it is guaranteed by construction that z is only read from the second loop iteration onwards, therefore no initial write to z is required.

After this translation process, there is no variable anymore written by assignments before and in a loop. Every variable read by a statement in a loop is no longer written in that loop. Therefore, synchronization can be added around the while-loop. This is demonstrated in Figure 9, which shows the parallelized application from

```
loop{
  x = f();
  loop{
    g(x);
    x' = h();
  } while(...);
} while(1);
```

(a) Input program

```
loop{
  x = f();
  first = true;
  loop{
    if(first){
      y = x;
      first = false;
    }
    else{
      y = z;
    }
    g(y);
    z' = h();
  } while(...);
} while(1);
```

(b) Transformed program

Figure 8: Example of a variable written by an assignment before and in the inner loop

```
do{
  acqProd(b_x);
  write(b_x, f());
  relProd(b_x);
} while(1);
```

(a) Task t_f

```
do{
  first = true;
  acqCons(b_x);
  do{
    acqCons(b_z);
    if(first){
      y = read(b_x);
      first = false;
    }
    else{
      y = read(b_z));
    }
    relCons(b_z);
    g(y);
  } while(...);
  relCons(b_x);
  acqCons(b_z);
  relCons(b_z);
} while(1);
```

(b) Task t_g

```
do{
  acqProd(b_z);
  relprod(b_z);
  do{
    acqProd(b_z);
    write(b_z, h());
    relprod(b_z);
  } while(...);
} while(1);
```

(c) Task t_h

Figure 9: Parallelized task graph given the transformed program from Figure 8b

the previous example. Since the variable x is only read in the inner while-loop from Figure 8b, synchronization for this variable is added around the loop, see Figure 9b.

When a function placed after a while-loop reads from a variable written by statements inside that while-loop, only the last written value is read. To store this value, a new variable is added to the program. This new variable is only assigned a value in the last loop iteration. Since it is now known by construction that this variable is assigned a value only once during the execution of all while-loop iterations, synchronization statements for this variable can be placed around the while-loop. All statements placed after the while-loop accessing the old variable are now changed to read from the new variable. Since the producing tasks synchronizes only once for this new variable, also the consuming tasks only have to synchronize once.

This process is illustrated in Figure 10. In the input program from Figure 10a a variable x is written by a function f in a while-loop and read by a function h after that loop. In the transformed program from Figure 10b a new variable y is created and in the last while-loop iteration this variable is assigned the last produced value of x. In the function h all accesses to x are now replaced by y. The parallelization and insertion of synchronization statements is analogue to the case described before and therefore not detailed in this paper.

```
loop{
  loop{
    x' = f();
  } while(g(x'));
  h(x);
} while(1);
```

(a) Input program

```
loop{
  loop{
    x = f();
    t = g(x);
    if(!t){
      y' = x;
    }
  } while(t);
  h(y);
} while(1);
```

(b) Transformed program

Figure 10: A variable is written by an assignment in the inner loop and read by a function after the inner loop

6. Structured Variable-rate Phased Dataflow

In this section we first define the SVPDF temporal analysis model and then explain the automatic derivation of such a model from a sequential OIL specification of an application.

Formally, an SVPDF model is a directed multigraph defined as $G = (V, E, P, \delta, \rho)$, where V is a finite set of actors and E is a finite set of edges with $(v_i, v_j) \in E$ and $v_i, v_j \in V$. An SVPDF model is structured into blocks with port actors, on the block boundaries. A block is characterized by a parameter $p \in P$, defining the number of consecutive iterations the actors in a block fire, after which the parameter value changes. A port actors perform the rate conversion between the rate of a block and its parent.

Actors in an SVPDF model are not auto-concurrent, meaning there is an implicit edge back from an actor to itself with one token on this edge. All actors, except port actors, consume one token from each input edge and produce one token on each output edge during every firing. The firing duration $\rho : V \rightarrow \mathbb{R}^+$ is the time between the start and finish of a firing. Actors consume tokens at the start of a firing and produce tokens at the moment the firing finishes. On an edge e are $\delta(e)$ initial tokens, with $\delta : E \rightarrow \mathbb{N}$.

Port actors are the actors that are used to consume from and produce tokens to actors outside of a block. These port actors are divided into two categories, upscale port actors and downscale port actors. An upscale port actor consumes one token and produces p tokens whereas a downscale port actor consumes p tokens and produces one token. Figure 4 shows an example of an upscale port actor, actor w_1, and a downscale port actor, actor w_0. The phases of a port actor are always structured the same, as shown in the figure.

An upscale port actor consumes one token from outside a block and enables the use (of a "copy") of this token multiple times in the block. This is modeled with an actor that has two phases. The first phase consumes and produces one token and the second phase consists of a parameter p. This parameter reflects the rate conversion such that one token is consumed and p tokens are produced.

A downscale port actor is used to produce tokens to actors outside a block. Every token produced by a downscale port actor is always produced in the last firing of the p consecutive firings of the block. This behavior is also modeled with two phases. The first phase consumes $(p - 1)$ tokens from the block and produces 0 tokens to actors outside of the block whereas the second phase consumes and produces one token.

Thanks to the specific structure with port actors, only the first of p consecutive firings of a block depends on the consumption of tokens from outside the block. Next to that, only the last iteration of p consecutive firings of a block produces tokens to actors outside of the block. Note that a block does not act as a barrier because the moment at which different port actors consume/produce tokens from/to actors outside the component, does not have to be equal. Since a port actor is a modeling construct which has no corresponding object in a task graph, its firing duration is zero. Port actors are

left implicit in the formal description of the SVPDF model, which improves the readability of the proofs in Section 7.1.

6.1 Automatic Generation of an SVPDF Model

From every task graph extracted from a valid OIL program, a corresponding SVPDF model can be automatically generated. The dataflow model reflects the synchronization statements that are placed each task in the parallelized task graph. In a while-loop synchronization statements are executed conditionally, meaning that it is decided at run-time whether another loop iteration is executed or whether the statements after the while-loop are executed. The SVPDF model can be used to model this conditional synchronization. Synchronization statements for if-statements, functions and assignments are executed unconditionally, and thus a static dataflow model such as a Homogeneous Synchronous Dataflow (HSDF) model can be used to model these statements.

The translation of a task graph $H = (T, B, \beta)$ to an SVPDF model $G = (V, E, P, \delta, \rho)$ can be defined as follows. The function $\psi : \mathbb{N} \rightarrow \mathbb{N}$ returns the index $j = \psi(i)$-th of the block that is fired during the iteration i of the source. For every task $t_k \in T$ that is executed in block q, an actor v_k^q is added in the model, which is fired if $q = \psi(i) \ \% \ N$, where $\%$ is the modulo operation and N the number of while-loops. The firing duration of this actor, $\rho(v_k^q)$, is equal to the response time of t_k. In the case of budget-based run-time schedulers, the response time of tasks can be determined independent of the execution time and execution rate of other tasks [18].

For every source or sink task $t_s \in T_S$ there are N actors v_s^q added in the SVPDF model, with $q = \{0, 1, \ldots, N-1\}$. This is because a source or sink task contains multiple while-loops that execute at different rates, as can be seen in Figure 6b. These tasks thus belong to different blocks. The q-th block corresponds to the q-th while-loop in the OIL program. The port actors in the derived SVPDF model are left implicit.

For every actor v_i, corresponding with a task t_i in which CB b is written, and actor v_j, corresponding with a task t_j which reads from b, an edge (v_i, v_j) is added. When b corresponds with a variable representing a loop termination condition, a token is added on this edge. This token represents that a while-loop is always executed at least once. An edge $e = (v_j, v_i)$ is added to represent the empty locations in b. The number of initial tokens $\delta(e)$ on this edge is determined by the buffer capacity analysis. The sum of the initial tokens on the edges (v_i, v_j) and (v_j, v_i) is equal to the buffer capacity of the corresponding CB, $\beta(b)$. For a detailed discussion on how multiple tasks reading from and writing to the same buffer can be included in dataflow models, the reader is referred to [2].

A source or a sink places constraints on the throughput that an application must achieve. However, periodic execution constraints cannot be expressed in a dataflow graph. Therefore, separate constraints are added to the SVPDF model. Because multiple actors are derived for every source or sink task, the combined schedule of these actors must result in the periodic schedule of that task. As a transition between blocks can occur after any number of iterations of a block, a constraint is added to the SVPDF model for a repetitive firing of each source/sink actor and a constraint is also added for firings of successive actors corresponding with the same source or sink task.

For a repetitive firing of an actor in the same block it must hold that the time between firings is μ, where μ is the period of a source or sink, ie. one divided by the frequency of a source or sink. Therefore it must hold that $\sigma(v_s^q, i+1) = \sigma(v_s^q, i) + \mu$, where we define $\sigma(v, i)$ as the start time of an actor v in iteration $i \in \mathbb{N}$ if this actor fires in iteration i of a source or sink actor. For a transition between blocks a similar constraint exists. Take as the switching moment iteration i. Then it must hold that $\sigma(v_s^{(q+1) \ \% \ N}, i+1) = \sigma(v_s^q, i) + \mu$. If both constraints hold than it holds that the corresponding source or sink task t_s can execute every period μ, i.e.:

$$\forall_{t_s \in T_s, i \in \mathbb{N}} : \sigma'(t_s, i) = \sigma'(t_s, 0) + i \cdot \mu \tag{1}$$

Note that we define the start time of the first firing of the source and sink actors equal to their corresponding source or sink task. This is done irrespective of whether such an actor actually fires in iteration 0: $\forall_{v_s^q \in V_S} : \sigma(v_s^q, 0) = \sigma'(t_s, 0)$. Both of the constraints on the start times of actors corresponding to source and sink tasks can be verified using HSDF analysis techniques [14].

The first constraint (repetitive firing of the same actor) can be verified by analyzing each block in isolation. A block in isolation is obtained by removing all edges to port actors. The second constraint can be verified by analyzing a flattened version of the SVPDF model where only one iteration of each block is modeled. In the next section we prove that analyzing each block in isolation combined with analyzing the flattened graph, is sufficient to guarantee the periodicity constraint of the sources and sinks in the application.

The throughput of each block in isolation and of the flattened graph can be verified using an algorithm to find the maximum cycle mean (MCM) in HSDF graphs. A number of such algorithms exist that have a polynomial time computational complexity [5]. Because the complete SVPDF analysis method analyses N blocks, the computational complexity remains polynomial.

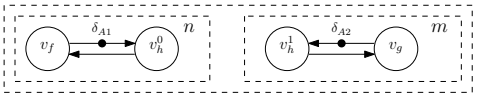


Figure 11: SVPDF model for Figure 6

Figure 11 shows the SVPDF model corresponding to the example program from Figure 6. For the source defined by function h, two actors v_h^0 and v_h^1 are added in the model. Since there are two while-loops with a source access, the constraints in Equation 2 are added. These constraints state that the time between subsequent firings of every iteration of the source h fires is μ. These constraints are only valid when the same while-loop is repeatedly executed.

$$\begin{aligned} \sigma(v_h^0, i+1) &= \sigma(v_h^0, i) + \mu \\ \sigma(v_h^1, j+1) &= \sigma(v_h^1, j) + \mu \end{aligned} \tag{2}$$

The two instances of the source function h are executed sequentially in the source program because of the order of the while-loops. This results in the additional constraints that are shown in Equation 3. The first constraint states that when the first while-loop ends at iteration i_0, the source actor in the next while-loop must fire μ time later. The second constraint states that the transition back from the second to the first while-loop must also happen in μ time.

$$\begin{aligned} \sigma(v_h^0, i_0+1) &= \sigma(v_h^1, i_0) + \mu \\ \sigma(v_h^1, i_1+1) &= \sigma(v_h^0, i_1) + \mu \end{aligned} \tag{3}$$

7. Analysis of SVPDF Graphs

7.1 Throughput Analysis

This section discusses the throughput analysis of SVPDF graphs. The strictly periodic execution of sources and sinks imposes a throughput requirement on the application. Because sources and sinks are modeled by actors, this also places throughput constraints on the SVPDF graph. It is proven in this section that we can adhere to these temporal constraints by analyzing a flattened graph where only one iteration of each block is modeled and by analyzing each block in isolation. This is possible thanks to the specific structure of SVPDF graphs.

Source and sink tasks correspond with multiple mutual exclusive actors in the model, each actor belonging to one block. Every execution of a source or sink corresponds to the firing of exactly

one such actor. The periodicity constraint of sources and sinks is thus spread over all blocks. For every block, the constraint is again split into two separate constraints. The first constraint (C1) specifies that as long as a block is repeated, the actors, corresponding to source or sink tasks, in that block can fire periodically. The second constraint (C2) specifies that when a block is finished, the actor in the next block fires one period after the last firing of the actor derived from the same source or sink but in the current block.

To prove that these two constraints hold, a parameterized schedule for the actors in the SVPDF graph is proposed. This is followed by the proof that this schedule is admissible. A schedule is admissible if sufficient tokens are present on the incoming queues in the model at the start of each firing of the actors in the schedule.

We prove that the proposed schedule is admissible by showing that the analysis results obtained by analyzing the flattened graph and every block in isolation are conservative when one or more blocks have consecutive iterations. From analyzing the flattened graph it is known whether this graph satisfies the periodicity constraint μ. For this proof we assume that this flattened graph satisfies the periodicity constraint and refer to this as Fact 1. We furthermore assume that the source and sink actors in every block in isolation can fire every μ, to which we will refer as Fact 2.

From the structure of the OIL language it is known that a block, which corresponds to a while-loop, is fired a number of times before the next block is fired. The function $\phi(j)$, with $\phi : \mathbb{N} \rightarrow \mathbb{N}^+$, returns the number of firings of block j before block $j+1$ is fired. The function $\psi : \mathbb{N} \rightarrow \mathbb{N}$, as defined in Equation 4, returns the $j = \psi(i)$-th block that is fired during iteration i of the source, with $i \in \mathbb{N}$.

$$\psi(i) = \max\{x \mid \sum_{j=0}^{x-1} \phi(j) \leq i\} \tag{4}$$

The two constraints C1 and C2 can now be formalized in terms of these functions. The first constraint is that a successive firing of an actor derived from a source or sink starts μ time later. This constraint is formalized in Equation 5. This equation says that a firing of an actor v_s^q derived from a source or sink must occur every μ time, if the actors in the enclosing block are repeatedly fired.

$$\begin{aligned} \forall_{v_s^q \in V_S} : \psi(i+1) = \psi(i) \Rightarrow \\ \sigma(v_s^q, i+1) = \sigma(v_s^q, i) + \mu \end{aligned} \tag{5}$$

The second constraint states that if a transition between blocks occurs, the time between the firings of actors derived from the same source or sink is μ. This constraint is formalized in Equation 6.

$$\begin{aligned} \forall_{v_s^q, v_s^{(q+1)\%N} \in V_S} \quad : \psi(i+1) = \psi(i) + 1 \Rightarrow \\ \sigma(v_s^{(q+1)\%N}, i+1) = \sigma(v_s^q, i) + \mu \end{aligned} \tag{6}$$

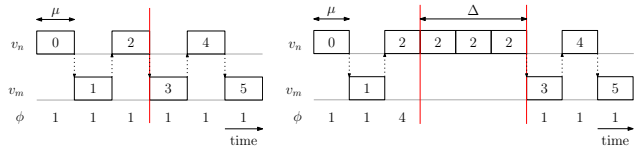
We define a parameterized schedule σ in Equation 7, where α is defined as $\alpha(i) = \sum_{j=0}^{\psi(i)-1} \phi(j)$. The value of $\alpha(i)$ corresponds with the number of iterations that are fired by the completely finished blocks, i.e. blocks that have executed $\phi(j)$ times.

$$\begin{aligned} \forall_{v^q \in V} : \sigma(v^q, i) = \sigma(v^q, 0) + \alpha(i) \cdot \mu + (i - \alpha(i)) \cdot \mu \\ \text{if } q = \psi(i) \ \% \ N \end{aligned} \tag{7}$$

In the next two lemmas we show that the schedule σ is a schedule satisfying the periodicity constraint as defined by equations (5) and (6) for arbitrary parameters i.e. $\forall_j : \phi(j) \geq 1$. Then we show in Lemma 7.4 that the schedule σ is also an admissible schedule for arbitrary parameters.

Lemma 7.1. *The schedule defined by Equation 7 satisfies the constraint defined in Equation 5 for arbitrary parameter values, i.e.* $\forall_j : \phi(j) \geq 1$.

Proof. For two subsequent firing of an actor $v^q \in V_S$ for which it holds that $\psi(i) = \psi(i+1)$, we must prove that



(a) Periodic schedule (b) A block repeated four times

Figure 12: Periodic schedule for an SVPDF graph with two actors in different blocks. In the right schedule, the actor in block $\psi(i) = 2$ fires three additional iterations

$\sigma(v^q, i+1) = \sigma(v^q, i) + \mu$. Substituting (7) in this equation results in:

$$\alpha(i+1) \cdot \mu + (i+1-\alpha(i+1)) \cdot \mu = \alpha(i) \cdot \mu + (i - \alpha(i)) \cdot \mu + \mu$$

which is true because $\alpha(i) = \alpha(i+1)$ if $\psi(i) = \psi(i+1)$ according to the definition of the function α. □

Lemma 7.2. *The schedule defined by Equation 7 satisfies the constraint defined in Equation 6 for arbitrary parameter values, i.e.* $\forall_j : \phi(j) \geq 1$.

Proof. For two subsequent firing of actors $v_s^q, v_s^{(q+1)\%N} \in V_S$, for which it holds that $\psi(i+1) = \psi(i) + 1$, we must prove that $\sigma(v_s^{(q+1)\%N}, i+1) = \sigma(v_s^q, i) + \mu$. Because the start time of the first firing of both actors is equal, ie. $\sigma(v_s^q, 0) = \sigma(v_s^{(q+1)\%N}, 0)$, substituting (7) in this equation results in:

$$\alpha(i+1) \cdot \mu + (i+1-\alpha(i+1)) \cdot \mu = \alpha(i) \cdot \mu + (i - \alpha(i)) \cdot \mu + \mu$$

Since it is known that $\psi(i+1) = \psi(i) + 1$ and using the definition of α we get $\phi(\psi(i)) \cdot \mu + (1 - \phi(\psi(i))) \cdot \mu = \mu$ which is true. □

The next lemma shows that the self-timed execution of an SVPDF has a temporally linear behavior. This fact is used to prove Lemma 7.4.

Lemma 7.3. *The self-timed execution of an SVPDF graph has a temporally linear behavior.*

Proof. An SVPDF is a special case of a functionally deterministic data flow (FDDF) graph because all SVPDF actors have sequential firing rules [13]. Because an SVPDF is a special case of an FDDF graph it has a temporally linear behavior [19]. □

In Lemma 7.4 we prove that the parameterized schedule σ as defined in Equation 7 is an admissible schedule for arbitrary parameters i.e. $\forall_j : \phi(j) \geq 1$. The intuition behind the proof is illustrated with the schedules shown in Figure 12. The numbers in the schedule of every actor indicate for a firing i the block number $\psi(i)$. The dotted arrows indicate dependencies between actors. All parameters $\phi(j)$ equal one in Figure 12a. In Figure 12b the block for which $j = \psi(i) = 2$ is fired three additional times. As a consequence the production of this block, which is indicated with an arrow, is delayed by Δ time. However the consumption by v_m is also delayed according to the parameterized schedule σ with Δ time. Therefore σ remains admissible for $\phi(j) \geq 1$ if σ is admissible for $\phi(j) = 1$. In the proof for Lemma 7.4 the parameterized schedule is split into three cases. In Figure 12b this case distinction is shown by two red lines.

Lemma 7.4. *The parameterized schedule in Equation 7 is an admissible schedule for arbitrary parameter values, i.e.* $\forall_j : \phi(j) \geq 1$.

Proof. This lemma is proven by induction. We define σ_K as $\sigma_K = \sigma$ with $\forall_{k<K} : \phi(k) \geq 1$ and $\forall_{k \geq K} : \phi(k) = 1$.

Base case: For $K = 0$ we have $\forall_{k\geq 0} : \phi(k) = 1$. We know that if $\forall_{k\geq 0} : \phi(k) = 1$ the schedule σ_K is admissible because this is given by Fact 1.

Induction step: We show that σ_{K+1} is an admissible schedule assuming that σ_K is an admissible schedule. This is shown in (8) by a case distinction in $\psi(i)$. To keep the notation compact we define $\Phi(K) = \phi(K) - 1$. Furthermore, without loss of generality, it is assumed that $\forall_{K,i\in\mathbb{N}} : \phi_K(i) = \phi(i)$.

$$\sigma_{K+1}(v,i) = \begin{cases} \sigma_K(v,i) & \text{if } \psi(i) < K \\ \sigma_K(v,\alpha(i)) + (i-\alpha(i)) \cdot \mu & \text{if } \psi(i) = K \\ \sigma_K(v, i - \Phi(K)) + \Phi(K) \cdot \mu & \text{if } \psi(i) > K \end{cases} \tag{8}$$

For the case that $\psi(i) < K$ it holds that $\sigma_{K+1}(v,i)$ is admissible because the induction hypothesis states that σ_K is admissible.

For the case that $\psi(i) = K$ we have that $\phi(K) = 1$ in $\sigma_K(v,i)$ but $\phi(K) \geq 1$ in $\sigma_{K+1}(v,i)$. By making use of Fact 2, which states that actors in a block can fire every μ, and that this block can fire $\Phi(K)$ times without consuming and producing any tokens from edges outside of that block, it follows that $\sigma_{K+1}(v,i)$ is admissible during the $\Phi(K)$ additional iterations of the block.

For the case that $\psi(i) > K$ we use the fact that only during the last iteration of a block tokens are produced via downscale port actors. Therefore, the $\Phi(K)$ additional iterations of the block delay the production of a token by at most $\Delta = \Phi(K) \cdot \mu$ time. From Lemma 7.3 we know that the self-timed execution of an SVPDF graph has a temporally linear behavior. Therefore, a production of an actor that is Δ time later will not delay the enabling of any actor firing later than that actor firing, by more than Δ. Thus is the delay of actor firings during the firing of the $\psi(i) > K$-th block, not more than Δ. In the parameterized schedule σ_{K+1} is is known that the firing of the actors during the firing of the $\psi(i) > K$-th block are delayed by $\Delta = \Phi(K) \cdot \mu$. From this it follows that $\sigma_{K+1}(v,i)$ is admissible because the actors are enabled before they are fired.

We can conclude that σ_K is an admissible schedule for every K by making use of the induction axiom because both the base case and the induction step hold. Because σ_K holds for every K, this schedule is equal to the parameterized schedule σ for arbitrary parameters $\phi(j) \geq 1$. □

Theorem 7.5. *The existence of an admissible schedule σ of the SVPDF graph with period μ as defined in Equation 7 implies that the source and sink tasks can execute strictly periodically with period μ as defined in Equation 1.*

Proof. By making use of the earlier-is-better-refinement [7] we know that tasks will not produce data later than the corresponding actors produce tokens if the actors have firing durations that are larger or equal than the response times of the tasks. From this we conclude that (1) holds if (7) holds. □

7.2 Buffer Sizing

Once it is determined using an SVPDF model whether an application can meet its throughput constraint, sufficient buffer capacities must be determined. This is because a pipelined execution of an application can require more than one value of a variable simultaneously. In the previous section it is shown that the throughput of an SVPDF model can be determined under two assumptions. To determine sufficient buffer capacities it must thus be shown that the required models meet these assumptions. To show that these models meet the throughput constraints, a sufficient number of tokens must be available on every edge. In [18] it is shown that the number of tokens relate directly to the buffer capacity of the buffer modeled by this edge.

When analyzing the flattened graph and the individual blocks, buffer capacities are computed twice if they are present in both the individual block and the flattened model. Therefore, to determine sufficient tokens, and thus buffer capacities, we take the maximum number of tokens as determined by the analysis on these models. Taking the maximum ensures that sufficient tokens are available in both the individual blocks as well as in the flattened model.

Determining sufficient buffer sizes can be done using a linear program (LP) algorithm. Because the capacity of a buffer is always a discrete number, the capacity as determined by the LP algorithm can be rounded up [18]. An LP algorithm has a polynomial time computational complexity, thus also sizing buffers in an SVPDF model has a polynomial time computational complexity.

```
source Symbol = dfe() @ 156 KHz;
loop{
  channel = selectChannel(474 MHz);
  loop{
    s = readInput(Symbol, channel);
    acquisition(s, out window');
  } while(!isValid(window'));
  loop{
    x = readInput(Symbol, channel);
    verifySync(x, out syncd, out window');
    fft(x, out y, window);
    z = equalization(y);
    demap(z);
  } while(syncd);
} while(1);
```

Figure 13: OIL specification of a simplified DVB-T receiver

8. Case Study

In this case-study a simplified DVB-T transceiver illustrates the applicability of the introduced approach. In the DVB-T application, as shown in the OIL program from Figure 13, a periodic source dfe delivers symbols to the functions that belong to the acquisition and decoding mode. These two modes are specified by making use of two while-loops. First a channel is selected in the $selectChannel$ function by passing the frequency of the channel. Next, the first symbol can be read from the source by the $readInput$ function. The $acquisition$ function parses these symbols and tries to detect a reference symbol in the input stream. If this is successful, the stream of symbols is decoded by the functions in the second while-loop. A loss of synchronization is detected by the $verifySync$ function in the second while-loop. If this function detects that synchronization in the stream is lost, the outer while-loop is repeated.

Parallelization of this application results in a task for every assignment statement and function, including the while-loop termination condition in the OIL program. Therefore, 11 tasks are created for the DVB-T transceiver. The while-loop-structure of the application is copied inside the task derived from the source function dfe. This task communicate via two buffers, one buffer ($Symbol\_1$) for the communication with the tasks that correspond to functions in the first while-loop, the other buffer ($Symbol\_2$) for communication with the tasks that correspond to the functions in the second while-loop.

Figure 14 shows the SVPDF model of the DVB-T transceiver. This SVPDF model can be divided in three parts. On the left is the block modeling the tasks created from the functions in the first while-loop. The block created from tasks derived from functions in the second inner while-loop is depicted in the block on the right. Between these blocks is the actor for the *selectChannel* function. The actors without a name are actors used to model buffers with multiple tasks reading from that buffer. Consequently, only a single edge models the buffer capacity. Note that the variable *window* is written twice, but due to the renaming step explained in Section 5 two separate variables are automatically created in our approach. When the blocks in isolation are analyzed, only the edges shown in black are included for every block. This means that the edges modeling the *channel* variable are not included when analyzing

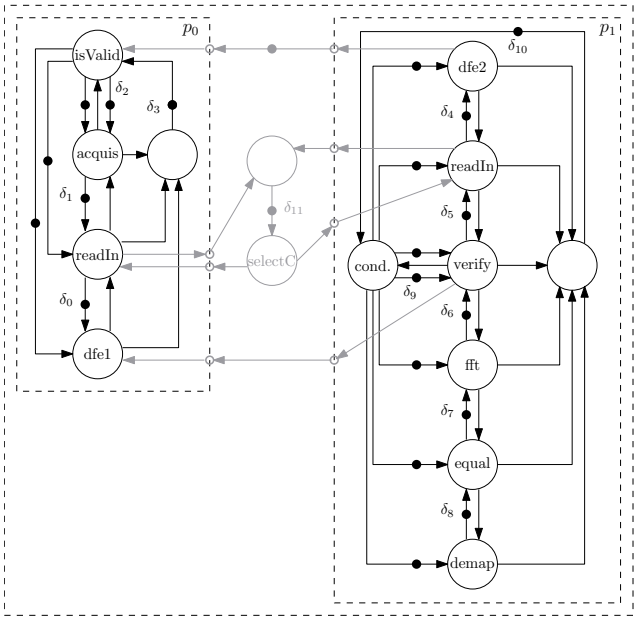


Figure 14: SVPDF graph derived from the DVB-T transceiver

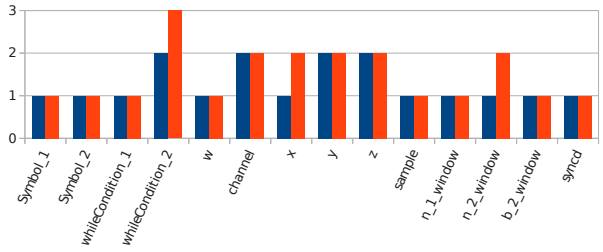


Figure 15: Buffer capacities as determined by the analysis tool

both blocks in isolation. The throughput constraint for both blocks in isolation is $\frac{1}{156\text{ kHz}} = 6.4\mu s$, which corresponds to the period of the source *Symbol*.

Using these two models and the flattened SVPDF model, when p_0 and p_1 have a value of one, buffer capacities are determined. The throughput constraint for the flattened model is $2 \cdot 6.4\mu s = 12.8\mu s$ because the source *Symbol* is accessed at least once per inner while-loop. The computed buffer capacities are shown in the graph in Figure 15 with the bars on the left. When the execution time of the *fft* and *demap* functions are set to $6.4\mu s$, the buffer capacities shown on the right in the figure are obtained. It can be seen that for the buffer of the second while-loop condition three locations are now required and two locations for the buffers containing the values for x and n_2_window.

When the application is executed using these computed buffer capacities, the schedule as shown in Figure 16 is obtained. The tasks are executed as soon as they are enabled because in this example we assume that all task do not share a processor. The bars in the schedule containing the text "XXX" represent the time before a task fires for the first time. The numbers inside the bars represent the q-th execution of that task except for the source function *dfe* where the numbers represent $\psi(i)$.

The schedule shows that the execution of both inner while-loops overlaps. When the sixth execution of the source starts, also the tasks that corresponds with functions in the first inner while-loop start executing again while the tasks corresponding with functions in the second inner while-loop are not yet finished. In this example, the tasks derived from the functions *equalization* and *demap* need to finish their execution.

Furthermore, this schedule shows that the function *selectChannel* can have an execution time of $12.8\mu s$, which is indeed twice the period of the source. However, because the first value produced by this function is required by a function in the first while-loop, the source starts executing at a time greater than zero. If the source would start executing at time zero, a periodic execution cannot be guaranteed since the termination condition of the first while-loop determines whether the second value should be written to the source buffer for the first while-loop or the buffer for the second while-loop.

9. Related Work

In this section other parallelization approaches are discussed that generate an analysis model besides a parallel implementation. We also discuss temporal analysis models that allow dynamism to be expressed.

The *pn*-tools presented in [15] automatically parallelizes sequential applications. A polyhedral model is used to analyze the application behavior and to generate the parallel implementation. In [1] it is shown that the temporal behavior of an application generated by the *pn*-tools can be analyzed with a Cyclo-Static Dataflow (CSDF) model. Support for the parallelization of while-loops is added in [15], but without the generation of a temporal analysis model.

The structured dataflow approach [10], like this work, has additional structure in the form of blocks. Also in this approach blocks represent loops. However, unlike our approach blocks behave the same as actors and have firing rules. As a consequence a block acts as a barrier, prevent pipeline parallelism over loop boundaries.

This work extends the approach presented in [3] and [8]. The approach presented in [3] generates a CSDF analysis model from a sequential specification of an application to analyze the temporal behavior of applications. This sequential specification is parallelized using function level parallelism. However, the sequential specification does not allow for while-loops with an unknown number of iterations to be specified. Also the CSDF model does not allow for dynamic behavior to be expressed. This paper extends this approach by generating a dynamic SVPDF model which does allow for the expression of while-loops. In [8] the parallelization approach from [3] is extended by allowing while-loops in the input specification. Synchronization statements are inserted in the parallelized application such that in every task synchronizes the same number of times for every variable. However, synchronization statements can therefore also be executed repeatedly in tasks not having any repetition, thus potentially resulting in a large synchronization overhead. This paper decouples synchronization between tasks and then shows how a temporal analysis model can be generated from the application.

A dataflow model which allows for dynamic behavior is the Variable-rate Dataflow (VRDF) model [17]. In the VRDF model actors can have variable rates. However, an upper bound on the parameter values must be known at compile time. For while-loops there is no such upper bound definable. The VPDF model does allow for phases to be expressed and does not require upper bounds on the parameter values [19]. However, two problems exist with analysis for VPDF models. First, analysis techniques for this model can result in a false detection of deadlock when deriving the maximum throughput because a linearization step is applied which results in an over-approximation. Furthermore, determining the parameters in an VPDF model during analysis has an exponential computational complexity in the number of parameters P: $O(V2^P + E2^P)$. The analysis algorithms for the SVPDF model have a polynomial time computational complexity and a false detection of deadlock can not occur.

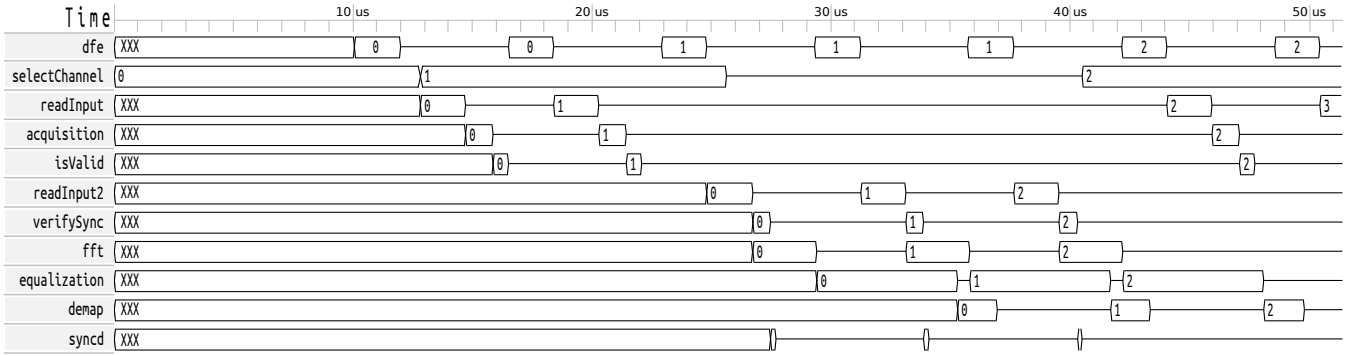


Figure 16: Fragment of a possible schedule for the DVB-T transceiver from Figure 13

The SADF model also allows for dynamic behavior to be expressed [16]. On top of an SDF model is a Finite State Machine (FSM) defined which specifies when mode transitions occur. Automatic derivation of the SDF model and the FSM has not been addressed. The SDF model is analyzed using model checking techniques which have a worst-case exponential complexity.

10. Conclusion

This paper introduced an approach in which a dataflow model is automatically derived from a sequential specification of a modal stream processing application. This derivation is enabled by a program transformation, which is introduced in this paper, where particular read operations in a while-loop are made non-destructive and particular write operations are made destructive. This results in a decoupling of the synchronization of tasks that correspond with functions in different while-loops.

The generated dataflow model is an SVPDF model which is used for the derivation of buffer capacities and the verification of temporal constraints. Because the model is generated from a sequential specification, it has a very specific structure. By making use of this structure we proved that we can verify whether the generated parallel application meets a throughput constraint using an algorithm that has a polynomial time computational complexity. The case-study shows the relevance and applicability of our approach. The generated schedule for a DVB-T application shows that the execution of tasks that belong to different modes can overlap and that the analysis using the SVPDF model takes this overlap into account.

We restricted variables in this paper to scalars for understandability of the methods and proofs. However, we consider it interesting future work to extend the presented approach to support arrays.

References

[1] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proc. of the ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 195–204. IEEE, 2011.

[2] T. Bijlsma. *Automatic parallelization of Nested Loop Programs - For non-manifest real-time stream processing applications*. PhD thesis, University of Twente, 2011.

[3] T. Bijlsma, M. Bekooij, P. Jansen, and G. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *Proc. of the Int'l Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 33–42. ACM, 2008.

[4] J. Buck and E. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 1, pages 429–432. IEEE, 1993.

[5] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):385–418, 2004.

[6] P. Derler, T. Feng, E. Lee, S. Matic, H. Patel, Y. Zhao, and J. Zou. PTIDES: A programming model for distributed real-time embedded systems. *University of California, Berkeley, EECS Technical Report. EECS-2008-72*, 2008.

[7] M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: A theory of timed actor interfaces. In *Int'l Conf. on Hybrid Systems: Computation and Control*, April 2011.

[8] S. Geuns, M. Bekooij, T. Bijlsma, and H. Corporaal. Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. *Design, Automation and Test in Europe (DATE)*, 2011.

[9] S. Geuns, J. Hausmans, and M. Bekooij. Sequential specification of time-aware stream processing applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1s):35, 2013.

[10] J. Kodosky, J. MacCrisken, and G. Rymar. Visual programming using structured data flow. In *Proc' IEEE Workshop on Visual Languages*, pages 34–39. IEEE, 1991. ISBN 0818623306.

[11] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, 2003.

[12] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.

[13] E. Lee and T. Parks. Dataflow process networks. In *Proc. of the IEEE*, May 1995.

[14] O. Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007, 2007.

[15] D. Nadezhkin and T. Stefanov. Automatic derivation of polyhedral process networks from while-loop affine programs. In *IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 102–111. IEEE, 2011.

[16] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. *Proc. Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI)*, 2011.

[17] M. Wiggers, M. Bekooij, and G. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 183–194. IEEE, 2008.

[18] M. Wiggers, M. Bekooij, and G. Smit. Monotonicity and run-time scheduling. In *Proc. of the ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 177–186. ACM, 2009.

[19] M. Wiggers, M. Bekooij, and G. Smit. Buffer capacity computation for throughput-constrained modal task graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):17, 2010. ISSN 1539-9087.

[20] J. Zou, J. Auerbach, D. Bacon, and E. Lee. PTIDES on flexible task graph: real-time embedded system building from theory to practice. In *ACM SIGPLAN Notices*, volume 44, pages 31–40. ACM, 2009.