Temporal Analysis Model Extraction for Optimizing Modal Multi-Rate 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

Modern real-time stream processing applications, such as Software Defined Radio (SDR) applications, typically have multiple modes and multi-rate behavior. Modes are often described using while-loops whereas multi-rate behavior is frequently described using arrays with pseudo-random indexing patterns. The temporal properties of these applications have to be analyzed in order to determine whether optimizations improve throughput. However, no method exists in which a temporal analysis model is derived from these applications that is suitable for temporal analysis and optimization.

In this paper an approach is presented in which a concurrency model for the temporal analysis and optimization of stream processing applications is automatically extracted from a parallelized sequential application. With this model it can be determined whether a program transformation improves the worst-case temporal behavior. The key feature of the presented approach is that arrays with arbitrary indexing patterns can be described, allowing the description of multi-rate behavior, while still supporting the description of modes using while-loops. In the model, an over-approximation of the synchronization dependencies is used in case of arrays with pseudo-random indexing patterns. Despite the use of this approximation, we show that deadlock is only concluded from the model if there is also deadlock in the parallelized application. The relevance and applicability of the presented approach are demonstrated using an Orthogonal Frequency-Division Multiplexing (OFDM) transmitter application.

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

General Terms Design, Theory, Verification

Keywords Real-time; Dataflow; Automatic Parallelization

1. Introduction

Real-time stream processing applications, such as SDR applications, are often multi-rate applications containing multiple modes. These modes can be described with while-loops in combination with if-statements. Arrays are often used to specify multi-rate behavior. Despite the presence of these programming constructs, the application has to be executed in parallel on a Multiprocessor System-on-Chip (MPSoC). Automatic optimizations have to be applied to achieve sufficient performance. Current compilers for sequential programs explore the optimization space to find transformations that are functionally equivalent to the input program [10]. The optimizations are based on the data-dependencies extracted from the sequential program. However, whether optimizations that modify the synchronization behavior introduce deadlock in a parallel program cannot be verified and can therefore not be applied automatically.

To determine whether an optimization introduces deadlock, a concurrency model is required in which deadlock analysis is decidable. Dataflow models, such as Synchronous Dataflow (SDF) models [14], can describe the synchronization behavior and can be used for deadlock analysis as well as temporal analysis. One application of these models is to verify whether an application meets its throughput constraints [12]. Also mapping an application to an MPSoC [16] and determining suitable scheduler settings can be done using dataflow models [19]. Furthermore, several optimizations already exist for these type of models. Examples are reducing synchronization overhead by moving and grouping synchronization [13], clustering tasks [9] and buffer scaling to increase pipeline parallelism [22]. However, no suitable dataflow model technique exists that allows the modeling of modes, while-loops and arrays with pseudo-random accesses. Consequently, no method exists which can derive such a model automatically from an application.

Automatic parallelization is a promising approach for programming MPSoCs [11, 15]. Starting from a sequential application description relieves the programmer of issues that make manual programming of parallel systems difficult, such as the insertion of synchronization statements. Furthermore, it provides the programmer with a familiar and simple programming model. Simultaneously with parallelizing an application, a corresponding temporal analysis model can be derived [12]. However, this method is limited to the specification of single-rate applications. Modern stream processing applications often have multi-rate behavior where array accesses can even be non-manifest, which means that these accesses are dependent on data produced by the environment. Therefore, a different modeling approach is required to model such behavior.

In this paper a method is introduced in which a deadlock-free temporal analysis model is automatically generated from a parallelized sequential application, despite the presence of non-manifest array accesses. In the dataflow analysis model only the synchronization is modeled and an abstraction is made from the functional behavior and the communication of data. This abstraction results in additional modeling freedom because synchronization behavior can be over-approximated. An over-approximation of the constraints in the parallel program is applied when modeling non-manifest array accesses because exact dependencies can not be derived at compile-time. It is shown that this conservative approximation does not introduce deadlock in the model because the approximation consists of the constraints that enforce the same execution order as in the sequential input program. Efficient analysis is possible because only a single iteration of every while-loop has to be considered during analysis.

The remainder of this paper is organized as follows. Section 2 discusses work related to the parallelization of sequential programs and model generation. In Section 3 the basic idea of our approach

SCOPES '14 June 10 - 11, 2014, Sankt Goar, Germany

http://dx.doi.org/10.1145/2609248.2609252

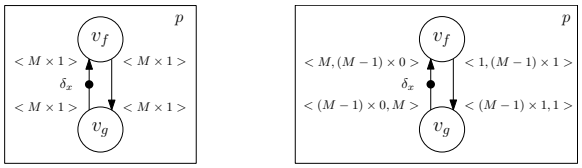
```
loop{
  forloop(0 <= i < M){
    x[i] = f();
    g(x[i]);
  }
} while(1);
```

(a) Manifest OIL program

```
loop{
  forloop(0 <= i < M){
    x[h()] = f();
    g(x[k()]);
  }
} while(1);
```

(b) Non-manifest OIL program

Figure 1: Example OIL programs where accesses are manifest (left) and non-manifest (right)



(a) Exact dependencies (b) Approximated dependencies

Figure 2: Models derived from the applications in Figure 1

is described. Section 4 discusses informally the parallelization and modeling of the for this paper relevant constructs of the sequential coordination language OIL. In Section 5 the Structured Variable-rate Phased Dataflow (SVPDF) model is introduced and it is shown how such a model can be extracted from the parallelized application. In Section 6 it is shown that the model is deadlock-free and an efficient analysis method is defined. The paper is concluded with a case-study in Section 7 and the conclusions in Section 8.

2. Related Work

Traditional methods for optimization in single-core compilers are typically targeted towards the optimization of sequential applications. These methods are based on deriving a Program Dependence Graph (PDG) from the application which describes the dependencies between the writing and reading of variables [10]. The PDG model is an untimed model. Therefore, no temporal guarantees can be given which makes this model unsuitable for the temporal optimization of concurrent applications. A PDG model can also be represented as a Boolean Dataflow (BDF) model, in which time can be added [7]. However, verifying whether an application modeled as a BDF can be executed in bounded memory or satisfies its throughput constraint is in general undecidable. For a real-time embedded system it is required that applications can be executed in bounded memory and that throughput constraints are met.

Polyhedral analysis techniques are often used for analyzing and reordering communication patterns in multi-core systems [2, 15]. The polyhedral model is a compact representation of the PDG. From a polyhedral model a dataflow temporal analysis model is derived. However, only affine array accesses are supported in the polyhedral model. Our method allows for arbitrary, and thus also non-affine, array accesses. Methods not based on polyhedral analysis also exist where non-affine array accesses can be taken into account [5]. The implementation used in this method guarantees a deadlock-free execution. However, no corresponding temporal analysis model is defined.

A temporal analysis model for analyzing applications containing while-loops is introduced in [12]. In their approach, a sequential specification is parallelized, which enables the derivation of a corresponding analysis model. However, the proposed analysis model is limited to the modeling of scalar variables whereas the approach proposed in this paper also allows array variables. Also, in their approach variables are scoped explicitly by while-loop iterations which limits the lifetime of a variable to a single while-loop iteration. Scoping allows for describing programs in single assignment (SA). In our approach, the scope of a variable is the entire program and describing a program as SA is not required, but additional costs may apply if the compiler cannot determine the lifetime of a variable.

Dataflow models such as Variable-rate Phased Dataflow (VPDF) do allow for the modeling of arrays and while-loops [22]. However, the known analysis method for this model applies a linearization step which can result in an over-approximation that can cause a false detection of deadlock. This over-approximation is not needed for SVPDF and no false deadlock detection will occur. The Scenario Aware Dataflow (SADF) model also allows for the specification of while-loops describing modes [18]. In this method a Finite State Machine (FSM) is defined on top of an SDF model to describe such modes. However, it has not been shown how such a model can be automatically derived from a sequential application.

3. Basic Idea

The basic idea behind the approach presented in this paper is illustrated by the two example OIL programs in Figure 1. These programs illustrate the difference between exact data-dependencies (manifest) and when only approximations can be derived (non-manifest). Both programs assign M times a value to an element of the array x. For the program in Figure 1a the exact dependencies are known at compile-time. These dependencies can be determined by inspecting the array index: the result of function $f()$ is assigned to element $x[i]$ and function $g()$ reads from element $x[i]$, where i is the iterator from the for-loop. Thus every read action depends on the write action in the same for-loop iteration. However, in the non-manifest program from Figure 1b it is unknown at compile-time which elements are written and read. Here, the function $f()$ assigns a value to element $x[h()]$ and $g()$ reads from element $x[k()]$, where $h()$ and $k()$ are black-box functions. Therefore, the written and read array elements can be any element in x. Deriving a static temporal analysis model requires constraints known at design time.

In our approach we derive a static analysis model based on the number of elements communicated. An important property that is exploited when deriving an analysis model from non-manifest applications is that a value can only be read after it is written as defined by the sequential semantics of the input specification. In the non-manifest example in Figure 1b, the array element read in iteration i of the for-loop can only be a value written prior to or during iteration i. During every iteration of the for-loop one additional element compared to the previous iteration is written and therefore only one new element can be read. In the analysis model we therefore model the constraint that only one new location can be read every iteration and we abstract from the actual location. This results in an over-approximation of the dependencies because it can not always be determined whether the last written location is read or that a location written in a previous iteration is read. For manifest programs the compiler can always determine exact dependencies. We illustrate the difference in modeling manifest and non-manifest programs using the two code examples shown in Figure 1.

If exact data-dependencies are known a corresponding temporal analysis model can be extracted which exactly models these dependencies. The corresponding analysis model for the manifest program in Figure 1a is shown in Figure 2a. In the model, which is visualized as a graph, the two vertices, or actors, represent the tasks from the parallelized application. The two edges together represent the buffer created for variable x, where the number of tokens δ_x represents the buffer capacity. Before an actor can fire, it requires a number of tokens. This number of tokens is specified at the input of an actor as a list of phases which is traversed in a cyclic order. In this paper we use the short-hand notation $M \times 1$ to denote a list of M items of value 1, where M is a constant number. At the end of an actor firing a given number of tokens is produced. This number is given by a list specified at an outgoing edge.

Before every firing of the actor v_f in the example, an empty buffer location (or a token in the model) is required. In the model this is reflected by the first 1 in the list at v_f on the edge from v_g to v_f. At the end of a firing a buffer location is written and thus a token is produced by v_f. This is modeled by a 1 on the edge from v_f to v_g. The actor v_g consumes this produced token and, due to the exact data-dependencies, it is known that this value is never read again and can therefore be released immediately.

In contrast to the previous model, Figure 2b shows the model derived from the non-manifest program in Figure 1b. In this model an abstraction is made from the buffer location that is written. This abstraction can be made because the model only reflects the temporal behavior and not the functional behavior. Before a buffer location can be written, it must be empty. Because any array element can be written, all corresponding buffer locations must be empty. Assuming the array size is M, also M locations must be acquired and thus M tokens must be consumed by v_f. After M tokens are consumed, the first location can be written, thus the first token is produced by v_f on the edge from v_f to v_g. This token represents the location corresponding with $x[h()]$, even though the result of calling $h()$ is not known.

When modeling the locations read by $g()$, we model the maximum number of locations that are potentially written before the element is read by $g()$. In the example a location is written and read in every iteration of the for-loop. Therefore, in the model one token is consumed every firing of v_g. After M for-loop iterations all read values are released. Releasing locations destroys the stored values. This is required because a finite buffer capacity is required and thus a finite lifetime of variables. If values are potentially required later the compiler must reinsert the current buffer contents back in the buffer. When and where this reinsertion action is added is shown in Section 4.4. For simplicity we assume in this example that no value is needed across iterations of the infinite while-loop.

This second example shows the heterogeneous semantics that this paper introduces for tokens in the dataflow model. On the edge from v_g to v_f the actual buffer locations that are acquired and released are modeled. In contrast, on the edge from v_f to v_g, it is modeled only that a location is written and read, but it is undefined which location.

4. Parallelization and Model Generation

In this section we will informally discuss the semantics of the constructs in the sequential language OIL and show how these constructs effect parallelization and model generation. In Section 5 we present the temporal analysis model and show how such a model can be derived. The OIL language is similar to C with a few exceptions. In OIL there is no recursion, dynamic memory allocation and there are no pointers. These restrictions ensure that every OIL program can be executed in bounded memory and no aliasing of variables occurs.

4.1 Functions and Assignments

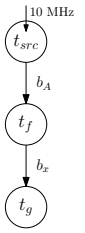
Function and assignment statements are the statements which will be executed in parallel. Functions can be implemented in a foreign language such as C. The assumption is that these functions are side-effect free such that they can be reordered during parallelization. From every function and assignment in the input program a task is extracted. An example of this parallelization step is shown in Figure 3b where the task graph corresponding to the input program from Figure 3a is shown. From both functions $f()$ and $g()$ a task is created and added to the task graph.

From the task graph and the precedence constraints in the sequential application an SVPDF model is derived. For every task in the task graph an actor is added to the model. The firing duration of such an actor is based on the execution time of the task and is independent of the execution time and execution rate of other tasks because of the usage of pre-emptive schedulers with budget guarantees [21]. The precedence constraints are used in the next section to derive the constraints between actors in case of non-manifest variable accesses. The SVPDF model derived from the example in Figure 3b is shown in Figure 3c. For both tasks t_f and t_g an actor is added to the model, named respectively v_f and v_g.

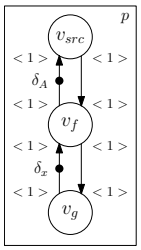
```
source A @ 10 MHz;

loop{
  x = f(A);
  if(...) g(x);
} while(1);
```

(a) OIL program



(b) Task graph



(c) SVPDF model

Figure 3: Producer with conditional-consumer example

1ML ⊂ FIFO ⊂ CB-S ⊂ CB-O ⊂ RIRO

Figure 4: Ordering on the generality of several buffer types

4.2 Variables

Variables in OIL are virtual locations in which computed values are stored. Unlike many other programming languages, a variable does not represent a physical memory location, but only a value. In our parallelization approach, a buffer is created for every variable. Such a buffer can be one of several types. A subset of these types is shown in Figure 4. Here the buffer types are ordered according to their generality. A more general buffer comes with a higher implementation cost of the required synchronization statements.

The most simple deterministic buffer is a single memory location (1ML) where arbitration is performed via a single bit indicating whether the producer or consumer can access the memory location [8]. A First-In First-Out (FIFO) buffer can have multiple locations and guarantees in-order access. FIFO buffers also enable a parallel execution of tasks communicating via this buffer. The more generic Circular Buffers with Sliding Windows (CB-Ss) allow out-of-order access and multiple producers and consumers [4]. However, when used in a parallel task graph deadlock can occur in some cases. Circular Buffers with Overlapping Windows (CB-Os) [5] do not introduce deadlock, but the lifetime of values stored in the buffer must be finite and known at compile-time such that the capacity of the buffer is finite. A Random-In Random-Out (RIRO) buffer does not have this requirement. Per array element, a list is maintained with produced values and the consumers can decide if and when values are no longer required and the next produced value will be used. In this paper we use buffers of type CB-O.

Synchronization for CB-Os is per element and such that no dependencies are created with other buffers. Synchronization is split into synchronization for tasks writing to a CB-O (producers) and tasks reading from a CB-O (consumers). A producer calls *acqProd* to request, or acquire, access to a buffer location for writing and calls *relProd* to release access to a location. Analogously, a consumer calls *acqCons* to acquire a location for reading and *relCons* to release it. A consumer cannot acquire a location before a producer has released it. The *acqProd* and *relCons* functions can only acquire and release consecutive locations in the buffer while the *acqCons* and *relProd* can acquire and release any location.

When inserting these synchronization statements in a task, it must be ensured that the functional behavior from the sequential specification is equivalent to that of the parallel task graph. The synchronization statements presented above only guarantee that a value is written before it is read. They do not guarantee that the correct value is available when multiple statements write the same location. Therefore, additional steps have to be taken to ensure that

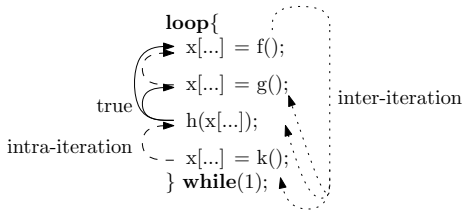


Figure 5: Different types of dependencies

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

(a) Original program

$\Longrightarrow$

```
x = f();
loop{
  if(first) y = x;
  else y = z;
  k(y);
  z = g();
  if(last) w = z;
} while(...);
h(w);
```

(b) Transformed program

Figure 6: Transformation for decoupling synchronization

variables are not overwritten before they are read. For non-manifest array accesses sequence constraints are added between the multiple producers and between producers and consumers such that the sequential ordering is retained for these tasks. For scalars or manifest array accesses, exact data-dependencies can be derived from analyzing or executing the sequential specification. For scalars this means that renaming variables is sufficient to preserve the functional behavior of the sequential program. For arrays a combination of renaming and retaining sequence is sufficient.

Consider the example program in Figure 5 where the three types of dependencies that we distinguish in this paper are shown. The true dependencies are the data-dependencies of read operations on write operations. The intra-iteration dependencies are dependencies from a producer on either a consumer or on an other producer within one iteration of the surrounding while-loop. They are caused by a producer overwriting a variable within one loop iteration. An inter-iteration dependency is the same as an intra-iteration dependency, but over the loop iteration boundary. Also such a dependency is caused by producers overwriting variables. Note that there are also inter-iteration constraints from the other producers in the example to all other statements, but they have been omitted from the figure for clarity. The sequence constraints that have to be added to preserve the functional behavior of the sequential application are caused either by intra or inter-iteration dependencies.

When the parallelization tool can assume that a part or the complete specification is in SA form, adding sequence or renaming is not required for this part since variables are never overwritten. Therefore, potentially more parallelism is available, which can increase the throughput of the parallelized application. In OIL there is an SA-statement to explicitly annotate that a part of the program is in SA form. For an example of the usage of this statement, see Figure 12. If this SA-statement was placed around the assignment statements located in the while-loop in Figure 5, the intra-iteration dependencies would no longer exist because none of the producers can overwrite an array element written by another producer. The inter-iteration dependencies remain because SA was only specified within a loop iteration, thus an assignment in another iteration can overwrite an array element. Placing an SA-statement around the complete program is not possible because this would require infinitely many elements in x and thus an an infinitely large memory.

4.3 If-Else Statements

An if-else statement provides a mutual-exclusive choice of execution within one iteration of a loop. After parallelization, synchronization for statements in the if-else statement is placed such that it is executed unconditionally. Since in the model only this synchronization is modeled, this prevents the need for a model where even deadlock analysis is not always decidable, such as the BDF model [7].

In Figure 3a the function $g()$ is conditionally executed. However, during parallelization, synchronization is placed around the if-statement to make it unconditional. Since this synchronization placement is modeled in Figure 3c, the execution of the actor v_g, representing task t_g, is unconditional. This can be seen since a token is unconditionally consumed and produced by v_g.

4.4 While-Loops

A while-loop in OIL is a block of statements which is executed repeatedly, similar to a do-while-loop in C. Its syntax is *loop...while* to distinguish it from loops in for example C. The main difference is that a loop in OIL suffers from a performance penalty in case values assigned to variables are potentially used during later loop iterations. As explained before, the CB-Os require that the lifetime of variables is finite and known at compile-time. In a while-loop the largest scope which is guaranteed to be finite is a single loop iteration. The transformation described below ensures that the lifetime of variables is bounded to one loop iteration. If variables are potentially used over iterations, the compiler has to reinsert values from the current iteration into the CB-O. Alternatively a RIRO buffer could be used, but than the overhead would be in the buffer implementation itself instead of in an additional reinsertion operation.

The termination condition of a while-loop can be any function or variable. This means that this condition can be non-manifest and therefore does not need to be known at compile-time. Despite this non-manifest behavior, the parallel execution of tasks executing different loop iterations can still overlap, provided there is no inter-iteration dependency preventing this parallelism.

Before an application with such while-loops can be parallelized, a transformation must be applied to ensure that every variable is written in only one while-loop. This will ensure that after parallelization a buffer is written and read the same number of times by all tasks and thus the synchronization behavior between these tasks is consistent. Reading a variable in deeper nested loops is allowed because values can be read multiple times by placing synchronization statements around the while-loop. Figure 6 illustrates this transformation with an example. The variable x is written before and in a loop-while statement, causing a rate difference when writing to the buffer b_x by tasks t_f and t_g, extracted from the functions $f()$ and $g()$. The buffer b_x is extracted from the variable x,

On the right in Figure 6 the program is shown after the transformation is applied. An additional variable z is added which replaces any assignments to x in the loop-while and the variable y replaces any reads of x. Since the two if-statements are inserted automatically, it is known that their conditions are true only once during the entire execution of the loop-while. Because the variable z is now written and read in different iterations, the compiler must add a copy statement reinserting the value from z in the buffer. Because the variable w is only written at the end of the last loop iteration because of the loop transformation, the compiler does not have to copy its value to make it available to the function $h()$.

Figure 7 shows the generated code for two tasks derived from Figure 6. As is shown in the code for task t_g, synchronization statements for buffer b_w are placed around the loop. The result of the last execution of $g()$ is copied to b_w during the last iteration of the loop. Because of the added copy statement which reinserts the value of the variable z into the corresponding buffer b_z, there are two assignments to z. Note that the task generated for this inserted copy statement is not shown in the figure. To ensure that the correct value is available to any read action, these assignments are performed sequentially. In the code for t_g in Figure 7a this

```
acqProd(t_g, b_w);
do{
  acqSeq(t_g, sg_z);
  acqProd(t_g, b_z);
  v = g();
  write(b_z, 0, v);
  relProd(t_g, b_z, 0);
  relSeq(t_g, sg_z);
  if(last){
    write(b_w, 0, v);
  }
} while(...);
relProd(t_g, b_w, 0);
```

(a) Task t_g

```
acqCons(t_k, b_x, 0);
do{
  acqCons(t_k, b_z, 0);
  if(first){
    y = read(b_x, 0);
  }
  else{
    y = read(b_z);
  }
  relCons(t_k, b_z);
  k(y);
} while(...);
relCons(t_k, b_x);
```

(b) Task t_k

Figure 7: Two generated tasks from the example of Figure 6

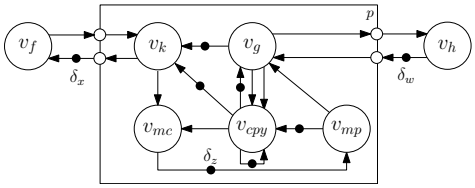


Figure 8: Model corresponding to the application in Figure 6

sequence is enforced by calling the functions *acqSeq* and *relSeq*. Both of these functions have two parameters, the first parameter is the calling task and the second parameter denotes the sequence group. All tasks which are in the same sequence group execute in the sequence as imposed by the ordering on this group. This ordering is determined by the compiler and will be the order of the corresponding statements in the sequential input program. In the example the sequence group sg_z consists of t_g and t_{cpy} because these tasks must retain sequence.

After this transformation is applied, an SVPDF model can be derived from the application. For every while-loop in the application a block is added to the model. For example in Figure 6 there is only one while-loop and thus also one block in the model, as shown in Figure 8. The parameter p of the block symbolically specifies how many iterations of a block are performed.

As shown in the generated tasks in Figure 7a, synchronization for different buffers can be placed in different loops. This can cause a rate difference during execution. In the derived model this rate conversion is modeled by special actors at the border of a block. These actors indicate that a variable can be read multiple times during a loop execution and written multiple times. Such an actor therefore represents a synchronization statement placed around a loop. In the example the actor between v_g and v_h on the border of the block represents the *acqCons(b_w)* statement. The actor between v_h and v_g represents the *relCons(b_w)* statement.

The model also shows the actor v_{cpy} which models the copy task. Because this task copies the value stored in z there are intra-iteration dependencies on all producers on a buffer, in the example v_g. The task t_g can overwrite a copied value, therefore there is also an inter-iteration dependency from v_g to v_{cpy}.

Furthermore, this model also shows how the sequence constraints imposed by the *acqSeq* and *relSeq* statements are modeled. A cycle is added between all actors corresponding with the tasks in a sequence group. This cycle contains one token such that only one actor can be enabled at any point in time. In the example the sequence group sg_z contains the tasks t g and t cpy, thus a cycle with one token is added between v_g and v_{cpy} respectively.

4.5 For-Loops

For-loops are similar to while-loops with the exception that an upper-bound on the number of loop iterations is known at compile-time. Because of this finite upper-bound, the lifetime of variables written in a for-loop can be defined over all iterations of the for-loop. Consequently, also bounded buffer capacities can be derived. The advantage is that no reinsertion has to be performed in every for-loop iteration. For-loops are suitable to express rate conversions in applications. Loops with different bounds can be used to iterate over differently sized arrays.

In the example in Figure 1b, a for-loop is shown where the number of loop iterations is M. When deriving the performance analysis model, a phase is added for every loop iteration. Figure 2b shows how M phases are added for M loop iterations such that one phase represents one loop iteration.

4.6 Sources and Sinks

The real-time constraints that are imposed by the environment on the application are described by sources and sinks [12]. Sources and sinks allow for communication between an application and its environment. A source samples the environment periodically and transfers every sample to the application. A sink periodically transfers a sample from the application to the environment. Sources and sinks execute time-triggered with a fixed period, in contrast to the data-driven execution of all other tasks. They are therefore specified in parallel with the sequential program. Communication between sources and sinks and tasks occurs via buffers. Despite the difference in execution style, they do not impose constraints on statements not accessing them. Therefore, they do not hinder parallelism or pipelining in particular. An example of a source is shown in the program in Figure 3a.

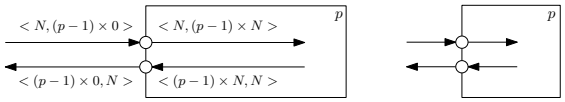
An application can have multiple sources and sinks, but the execution rates of these sources and sinks must be consistent with the sequential semantics of the application. A source delivers a new sample to the application every while-loop iteration. During a loop iteration a source delivers the same value for every read operation. A sink fetches a sample from the application also every while-loop iteration. When writing multiple times to a sink during a loop iteration, only the last written value will be visible to the environment. These semantics impose throughput constraints on an application. If no sample produced by a source should be lost, a while-loop iteration should be executed for every source sample. Therefore, the rate of the while-loop should be the same as the rate of the source. Analogously, if every sample produced by an application should be visible to the environment, the application and sink should execute on average at the same rate.

Accesses to a source or sink have to occur in every while-loop in an application. Otherwise, it cannot be verified at compile-time whether an application accesses the source or sink in time since while-loops can potentially execute infinitely long. However, to keep synchronization consistent across tasks accessing sources and sinks, also the variables used by them have to be decoupled. The implementation of a source or sink therefore consists of multiple variables, one variable for each while-loop specified in the sequential application. A source or sink task is also structured with the same while-loops as the sequential specification.

Deriving an analysis model for a source or sink is similar to a function or assignment. However, because a source or sink task contains multiple assignments to different variables, one actor is added to the model for every while-loop. These actors are connected by edges modeling the sequence and together model a source or sink. The example program in Figure 3a contains only one while-loop and therefore there is only one actor added to the corresponding model in Figure 3c.

5. SVPDF Model

In this section we introduce the SVPDF model and a derivation of an SVPDF model from a parallelized application. An SVPDF model is a directed graph defined by $G = (V, E, P, \theta, \pi, \gamma, \delta, \rho)$. Here V is a set of actors and $(v_i, v_j) \in E$ describes the set of



(a) Extended form (b) Abbreviated form

Figure 9: Port actors for up- and downscaling synchronization events

edges, with $v_i, v_j \in V$. We use the abbreviation e_{ij} to denote an edge (v_i, v_j). Actors in an SVPDF model are not auto-concurrent, at most one firing of every actor can occur simultaneously. This is modeled as an implicit edge from an actor back to itself with one token on this edge. The firing of an actor goes in a cyclo-static fashion where a list of phases is traversed and at the end of the list, the first phase is fired again. The number of phases of an actor is defined by $\theta : V \rightarrow \mathbb{N} \cup P$. On an outgoing edge this phase list is given by $\pi : E \times \mathbb{N} \rightarrow \mathbb{N}$. On an incoming edge this list is given by $\gamma : E \times \mathbb{N} \rightarrow \mathbb{N}$. The firing time for every phase is defined as $\rho : V \times \mathbb{N} \rightarrow \mathbb{N}$. The initial number of tokens on an edge is given by $\delta : E \rightarrow \mathbb{N}$. An actor can fire a firing number k if $\gamma(e_{ij}, k \bmod \theta(v_j))$ tokens are available on all incoming edges e_{ij}. After $\rho(v_j, k \bmod \theta(v_j))$ time, $\pi(e_{jl}, k \bmod \theta(v_j))$ tokens are produced on every outgoing edge e_{jl}.

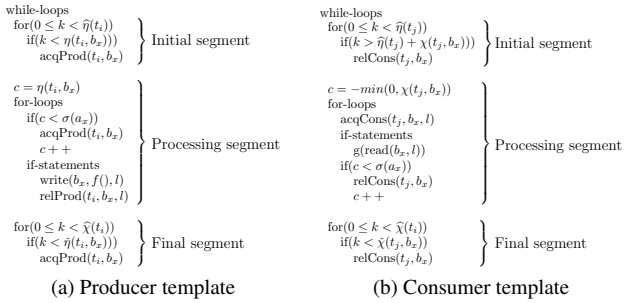
An SVPDF model is structured into blocks with port actors on the edges of a block. A block is characterized by a parameter $p \in P$. The parameter p defines the number of consecutive firings of actors in the block with respect to actors surrounding it. The value of a parameter is undefined during analysis and can be infinite. A block introduces only structure in the model and can not fire itself. Blocks are visualized as rectangles in the figures in this paper. Port actors provide communication between actors in and outside of a block. A port actor is either an upscale or a downscale port actor. Figure 9a shows these two types of port actors at the top and bottom respectively. The phases of a port actor always have the same structure as shown in the figure, being parameterized in the parameter p of the corresponding block and with a constant N. The constant N is dependent on the number of tokens produced by an actor v_i and is defined for a port actor v_q as $N = \sum_{k=0}^{\theta(v_i)-1} \pi(e_{iq}, k)$. Since the structure of port actors is always the same, it is omitted from figures and only the parameter is shown in a block, see Figure 9b.

5.1 Code Templates

As explained in Section 4, the derivation of an SVPDF model is based on the synchronization statements that are inserted by the compiler in all tasks in the task graph. Inserting these statements is based on two templates, one for a producer and one for a consumer. Inserting synchronization statements is described in more detail in [11], but because these templates form the base for deriving an SVPDF model we will briefly describe them in this paper.

Figure 10 shows the synchronization templates for a producer and consumer. Both templates consist of three segments, an initial, processing and final segment. If a task is both a producer and consumer, these templates are combined. In the initial phase of a producer, sufficient space is acquired for a task to write in. After processing, any remaining space is acquired such that a complete array is acquired. In the initial phase in the consumer, any space that can be released because the locations are never read is released. Synchronization for data communication is done in the processing phase. After processing, the remaining space is released such that the size of a complete array is released.

For the derivation of the parameters in the synchronization templates, we define for every buffer b_x a list S_x as a list of tuples of an array element index and a corresponding sequence number. These sequence numbers define an ordering on the execution of



(a) Producer template (b) Consumer template

Figure 10: Templates for synchronization statement insertion

variable assignments and functions in a sequential program. Every execution of an assignment or function is assigned its own sequence number in ascending order. We now define the list S_x such that it contains an element (l, s) if the element $x[l]$ is accessed by a statement with a sequence number s. If l is non-manifest, it is defined as $\perp$. If x is a scalar, l is zero. All elements in S_x are sorted according to their sequence number. For example in Figure 1a the list S_x is given as $\langle (n, 2 \cdot n) \bullet 0 \leq n < M \rangle$, where $\bullet$ separates the list elements from the proposition and M is the number of loop iterations. We define the sorted list A_x^i, with A_x^i being a sub-list of S_x thus $A_x^i \sqsubseteq S_x$, as the accesses to a variable x in task t_i from the parallelized task graph. Furthermore, we define the set B_P^i as the buffers written by t_i and the set B_C^i as the buffer read by t_i.

Using these lists and sets we define the size of the initial segment $\widehat{\eta}$ for both a producer and consumer. The statements *acqProd* and *relCons* acquire respectively release consecutive locations in a circular buffer. The initial segment contains synchronization for unused buffer locations such that later locations are always acquired or released in time. First, the size of the initial segment is defined per buffer. For the acquire statements this is called the lead-in. For a task t_i producing values in a buffer b_x, the lead-in is defined by (1). This equation returns the maximum of the differences between the n-th index accessed and the number of previously executed synchronization statements. We use the function ε_1 to return the first element of a tuple.

$$\eta(t_i, b_x) = max\{\varepsilon_1(A_x^i[n]) - n \bullet 0 \leq n < |A_x^i|\} \quad (1)$$

The starting point for releasing locations in a buffer b_x is called the lead-out. If elements in an array x are not accessed, and thus can be released before computation starts, the lead-out will be negative. This early releasing is done in the initial segment. The lead-out is defined as the difference between the number of previously executed synchronization statements n and the n-th accessed array index.

$$\chi(t_i, b_x) = max\{n - \varepsilon_1(A_x^i[n]) \bullet 0 \leq n < |A_x^i|\} \quad (2)$$

The size of the initial segment is defined in (3) in terms of the lead-in of every buffer and the lead-out, if locations can be released in advance.

$$\widehat{\eta}(t_i) = max\{\{\eta(t_i, b_x) \bullet b_x \in B_P^i\} \cup \{-\chi(t_i, b_y) \bullet b_y \in B_C^i\}\} \quad (3)$$

In the final segment the remaining acquires and releases are performed such that synchronization is performed for a complete array. The number of remaining acquires $\check{\eta}$ is defined as the array size σ minus the number of acquires already performed in the initial and processing segment.

$$\check{\eta}(t_i, b_x) = \sigma(a_x) - \eta(t_i, b_x) - |A_x^i| \quad (4)$$

The number of remaining releases $\check{\chi}$ that have to be performed in the final segment is also the number of array elements minus the

releases already performed in the initial and processing segments.

$$\check{\chi}(t_i, b_x) = \sigma(a_x) - (|A_x^i| - \chi(t_i, b_x)) \quad (5)$$

The size of the final segment can now be defined as the maximum of the remaining acquires and releases for every buffer.

$$\widehat{\chi}(t_i) = max\{\{\check{\chi}(t_i, b_x) \bullet b_x \in B_C^i\} \cup \{\check{\eta}(t_i, b_y) \bullet b_y \in B_P^i\}\} \quad (6)$$

5.2 Automatic Model Generation

Based on the synchronization templates presented in the previous section and the informal derivation of the graph topology of the SVPDF model in Section 4, we will refine the model derivation in this section and show how the phases of the actors are derived using the synchronization templates from the previous section.

For every task in a task graph, an actor is added to the model. In the example from Figure 3 actors are added for tasks t_f and t_g. Furthermore, for every while-loop in a source and sink task an actor is added. In the example there is only one while-loop, thus only one such actor is added. A block, modeling a data-dependent rate conversion, is added to the SVPDF model for every while-loop in the sequential program. All actors derived from different tasks, but containing the same loop-while statement, are placed in the same block. In the example there is only one while-loop, thus only one block. Since both functions $f()$ and $g()$ are located in this while-loop, the actors v_f and v_g are placed in the added block.

If tasks with non-manifest variable accesses are not in SA form, sequence constraints are added between these tasks. These sequence constraints are modeled in the SVPDF model by a cycle between the corresponding actors. This cycle contains a single token to enforce that only one task is active at any time. This token is placed on the incoming edge of the actor with the lowest sequence number of the corresponding variable accesses.

If multiple tasks write to the same buffer, a so-called merging producer actor is added to the model [3]. If multiple tasks read from the same buffer, a merging consumer actor is added. These merging actors are used to model the buffer capacity using a single edge. This edge is added from the merging consumer actor to merging producer actor. When a statement writing to a buffer is surrounded by synchronization statements, edges are added from this producer to every consumer on that buffer. Furthermore, an edge is added from every merging producer for a buffer to a producer writing to this buffer and from every consumer to the corresponding merging consumer. An example of these merging actors is shown in Figure 8. The merging actors are the actors v_{mp} and v_{mc}.

The derivation of the phases of SVPDF edges modeling buffers is based on the synchronization templates presented in the previous section and the sequential ordering in the OIL program. For every execution of a synchronization statement in a task, a phase is added to the model. The total number of phases of an actor v_i is therefore $\theta(v_i) = \widehat{\eta}(v_i) + max\{|A_x^i| \bullet b_x \in B\} + \widehat{\chi}(v_i)$.

In the following four subsections we will define the production and consumption phases of actors in the model. The first two sections describe the modeling of the synchronization for data items released by a producing task and acquired by a consuming task. For a producer t_p and consumer t_c, the first two sections thus describe the phase lists $\pi(e_{pc})$ and $\gamma(e_{pc})$. In the last two sections we describe the modeling of releasing space back into the buffer and acquiring free space from the buffer. These sections thus describe the phase lists $\pi(e_{cp})$ and $\gamma(e_{cp})$.

5.2.1 Data Production

The phases $\pi(e_{ij})$ for an actor v_i on an edge e_{ij} model the release of data by a task t_i. Therefore, these phases model the location of the *relProd* statement. Because a consumer does not always consume all produced values and because all produced tokens must be consumed, tokens may only be produced if they are consumed by a consumer. The function $\zeta_x^j(l, s)$ defines whether a location l will potentially be read after sequence number s by a task t_j.

$$\zeta_x^j(l, s) = \exists_{u,k} \bullet (u, k) \in A_x^j \wedge k > s \wedge (u = l \vee l = \perp \vee u = \perp) \quad (7)$$

The set $O_x^{i,j}(s)$ contains all values written to buffer b_x by task t_i and which are potentially read by task t_j. The set only contains values with a sequence number higher than s.

$$O_x^{i,j}(s) = \{l \bullet (l, s) \in A_x^i \wedge \zeta_x^j(l, s)\} \quad (8)$$

The phases modeling the production of data are defined in (9). Every phase k models the number of locations written at the corresponding sequence number by task t_i and which are potentially read by task t_j. The helper function $\varpi(t_i, k)$ defines the mapping of phase k of an actor v_i to a sequence number. Note that there is no lead-in or lead-out for releasing data by a producer because these synchronization statements are performed for the communicated locations immediately after communication.

$$\pi(e_{ij}, k) = \begin{cases} 0 & \textbf{if } 0 \leq k < \widehat{\eta}(t_i) \\ |O_x^{i,j}(\varpi(t_i, k))| & \textbf{if } \widehat{\eta}(t_i) \leq k < \widehat{\eta}(t_i) + |A_x^i| \\ 0 & \textbf{if } \widehat{\eta}(t_i) + |A_x^i| \leq k < \theta(v_i) \end{cases} \quad (9)$$

5.2.2 Data Consumption

The phases $\gamma(e_{ij})$ model the synchronization of a consumer t_j, thus the execution of the *acqCons* statement. Every phase k models the number of locations potentially read by t_j and produced by t_i. In order to derive these phases, we define $\alpha(e_{ij}, l, k)$ as the difference between the number of tokens produced up to and including phase l of an actor v_i and the number of tokens consumed up to phase k of an actor v_j. The order of producing buffer locations can differ from the reading order. This function ensures that the token corresponding with the read buffer location is consumed.

$$\alpha(e_{ij}, l, k) = max\{0, \sum_{n=0}^{l} \pi(e_{ij}, n) - \sum_{n=0}^{k-1} \gamma(e_{ij}, n)\} \quad (10)$$

We distinguish two cases when deriving the number of consumed tokens in each phase. In the first case there is either no data read from b_x by t_j or exact data-dependencies are known and it can be determined at compile-time no data is produced by t_i which is required by t_j. Because there is no constraint between t_i and t_j for this phase, no token needs to be consumed.

In the second case there is potentially a value produced by t_i and read by t_j. An approximation is made here to allow for non-manifest productions and consumptions. This approximation uses the sequential ordering to give an upper-bound on the last sequence number in which potentially a value was produced for the consumption with sequence number s. Equation 11 expresses this constraint. Note that when SA is required in the input specification, more information is available about the relation between producers and consumers and tighter bounds can be given here, resulting in a potentially higher throughput or lower latency.

$$\lambda(s) = max\{k \bullet (b, k) \in A_x^i \wedge (a, s) \in A_x^j \wedge k < s \wedge (a = b \vee b = \perp \vee a = \perp)\} \quad (11)$$

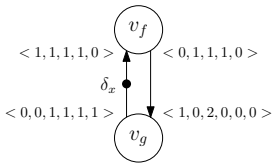
Equation 12 defines these two cases as a single constraint.

$$\beta(e_{ij}, k) = \begin{cases} 0 & \textbf{if } \lambda(K) = -\infty \\ \alpha(e_{ij}, \lambda(K), k) & \textbf{if } \lambda(K) \neq -\infty \end{cases} \quad (12)$$
$$\textbf{with } K = \varpi(t_j, k)$$

Finally, the number of tokens consumed in every phase k of an actor v_j is given in (13). The initial and final segments contain no *acqCons* statement and thus no tokens are consumed in the corresponding phases. In the processing segment the phases are

A_x^f	(1,0)	(2,2)	(0,4)	(3,6)
A_x^g	(1,1)	(1,3)	(0,5)	(2,7)

(a) Access pattern



(b) SVPDF model

Figure 11: SVPDF model generated from a manifest application

defined by $\beta(e_{ij}, k)$.

$$\gamma(e_{ij}, k) = \begin{cases} 0 & \textbf{if } 0 \leq k < \widehat{\eta}(t_j) \\ \beta(e_{ij}, k) & \textbf{if } \widehat{\eta}(t_j) \leq k < \widehat{\eta}(t_j) + |A_x^j| \\ 0 & \textbf{if } \widehat{\eta}(t_j) + |A_x^j| \leq k < \theta(v_j) \end{cases} \quad (13)$$

5.2.3 Empty Space Production

The edge e_{ji} models the empty locations in the corresponding buffer b_x. The phases on this edge correspond to the execution of the *relCons* and *acqProd* functions. These functions access consecutive locations in the buffer and thus synchronization can be done later or earlier then communication. In the SVPDF model this is also modeled. In (14) the execution of the *relCons* statement are modeled. The starting point of releasing locations is given by $\chi(t_j, b_x)$, which is an offset with respect to $\widehat{\eta}(t_j)$. Before this offset no location is released, and thus no token is produced. Next, the number of locations released, and thus tokens produced, equals the array size. Finally, there can be stuffing zeros to ensure the number of phases is equal for all phase lists of v_j.

$$\pi(e_{ji}, k) = \begin{cases} 0 & \textbf{if } 0 \leq k < \widehat{\eta}(t_j) + \chi(t_j, b_x) \\ 1 & \textbf{if } 0 \leq k - \widehat{\eta}(t_j) - \chi(t_j, b_x) < \sigma(a_x) \\ 0 & \textbf{if } \widehat{\eta}(t_j) + \chi(t_j, b_x) + \sigma(a_x) \leq k < \theta(v_j) \end{cases} \quad (14)$$

5.2.4 Empty Space Consumption

The consumption of tokens on e_{ji} models the execution of the *acqProd* statement. As can be seen in the template from Figure 10a the *acqProd* statement is fired $\eta(t_i, b_x)$ number of times in the initial segment. This can also be seen in (15). After stuffing zeros to complete the initial segment, the remainder of the array is acquired, followed by stuffing zeros to fill the phase list. Also here, b_x is the buffer corresponding to edge e_{ji}.

$$\gamma(e_{ji}, k) = \begin{cases} 1 & \textbf{if } 0 \leq k < \eta(t_i, b_x) \\ 0 & \textbf{if } \eta(t_i, b_x) \leq k < \widehat{\eta}(t_i) \\ 1 & \textbf{if } \widehat{\eta}(t_i) \leq k < \widehat{\eta}(t_i) + \sigma(a_x) - \eta(t_i, b_x) \\ 0 & \textbf{if } \widehat{\eta}(t_i) + \sigma(a_x) - \eta(t_i, b_x) \leq k < \theta(v_i) \end{cases} \quad (15)$$

EXAMPLE 1. *Figure 11 shows an example where the access pattern of the functions is known at compile-time. Assume the array size is four. Given the access pattern it can be derived that* $\widehat{\eta}(t_f) = \eta(t_f, b_x) = 1$*, thus* $\gamma(e_{gf}) = \langle 1,1,1,1,0 \rangle$ *and also* $\pi(e_{fg}, 0) = 0$*. Elements* $x[0]$*,* $x[1]$*,* $x[2]$ *are read whereas* $x[3]$ *is never read, therefore* $\pi(e_{fg}) = \langle 0,1,1,1,0 \rangle$*. For the consumer,* $\widehat{\eta}(t_g) = 0$ *and* $\chi(t_g, b_x) = 2$*, and thus* $\pi(e_{gf}) = \langle 0,0,1,1,1,1 \rangle$*. Since* $x[1]$ *is written and the first read value,* $\gamma(e_{fg}, 0) = 1$*. Then* $x[1]$ *is read again, thus* $\gamma(e_{fg}, 1) = 0$*. Before* $x[0]$ *can be consumed,* $x[2]$ *must be consumed, thus* $\gamma(e_{fg}, 2) = 2$ *and* $\gamma(e_{fg}, 3) = 0$*.*

6. Throughput Analysis

In this section it is first shown that every SVPDF model extracted using the approach described in Section 5.2 is deadlock-free. Then it is shown that it is sufficient to analyze every block in an SVPDF model in isolation, with all sub-blocks having a parameter value of one.

6.1 Deadlock-freedom

In this section we show that an SVPDF model generated from a sequential OIL program is always deadlock-free, given sufficient buffer capacities. A sequential OIL program specifies a set of precedence constraints Υ_{OIL}. These constraints are defined by the sequential ordering of statements in a program, i.e. line two can only execute after line one but also transitively line three can only execute after line one. Parallelization only preserves the data-dependency constraints from the input specification. This set is defined as $\Upsilon_{TG} \subseteq \Upsilon_{OIL}$. Satisfaction of the data-dependencies is ensured by the inserted acquire and release synchronization primitives. In case of non-manifest behavior where sequence must be retained, as explained in Section 4.4, also the sequence constraints from Υ_{OIL} are preserved in Υ_{TG}.

In case of non-manifest variable accesses or variable accesses guarded by if-statements, the set of constraints in the dataflow model, Υ_{DF}, is an over-approximation of Υ_{TG}, that is $\Upsilon_{TG} \subseteq \Upsilon_{DF}$. If an access is guarded by an if-statement an acquire or release statement is executed conditionally. However, in the SVPDF model this access is modeled as if it was unconditional. If a read access is non-manifest the number of writes prior to this read access are counted in the SVPDF model. Here the model also applies an over-approximation because in the task graph the reading task only has to wait for the task actually writing this location. However, for both non-manifest and guarded accesses it holds that the additional constraints in Υ_{DF} are also present in Υ_{OIL} because the additional constraints model the sequential precedence constraints. Because this reasoning can be applied to all variables it holds that $\Upsilon_{TG} \subseteq \Upsilon_{DF} \subseteq \Upsilon_{OIL}$. Since a sequential program is by definition deadlock-free and removing constraints cannot introduce deadlock in a functionally deterministic dataflow graph [21], also an SVPDF model derived from an OIL program must be deadlock-free.

Finite and sufficient buffer capacities also exist for which an SVPDF model is deadlock-free. Edges have by construction the same sum of production and consumption quanta within one repetition period. Therefore, after one firing of the complete cyclo-static period of all actors, the initial state is reached again. Because on the edge modeling empty space in the buffer every actor consumes or produces within one cyclo-static period the same number of tokens as the size of the corresponding array, the array size is sufficient as the capacity for the corresponding buffer for a deadlock-free execution.

Modeling buffer communication can introduce deadlock if an over-approximation is made which is not contained in the sequential precedence constraints. CB-Os allow for an immediate release of data to the buffer and acquiring data can succeed even if data is not produced in FIFO order in the buffer [5]. As is shown in Section 5.2.1, data items are released by actors in the model when the corresponding release statement is executed by the corresponding task. In the model the dependencies are made unconditional, i.e. they are over-approximated, in case it is unknown whether there will be a corresponding acquire action. This additional dependency is constructed from the sequential program and can therefore not introduce deadlock. To form the number of tokens acquired by a consuming actor the number of tokens produced by actors firings with lower sequence numbers is counted. Because there is never a token acquired which is produced with a higher sequence number acquiring tokens is captured by precedence constraints. From this we conclude that dependencies can be over-approximated by no

more than the precedence constraints from the sequential program and therefore deadlock is not introduced in an SVPDF model.

6.2 Minimum Throughput Analysis

Efficient analysis of the minimum throughput of an application is based on the method presented in [12]. Conditions are presented under which an SVPDF model can be analyzed using a flattened SVPDF model, i.e. a model where all parameters equal one. If an SVPDF model is flattened in their case, it is equivalent to an Homogeneous Synchronous Dataflow (HSDF) model. Throughput analysis must be done recursively for every block in an SVPDF model, given that all sub-blocks are flattened. Thus first the most deeply nested block must be analyzed and then its parent block until the root of the flattened SVPDF model is analyzed.

The proof that flattening an SVPDF model is allowed is based on the structuring of the dataflow model in blocks. If a parameter value increases by k a delay of $\Delta \cdot k$ is induced on any later actors. It is shown that this does not violate the throughput constraint because all sources and sinks are accessed by their corresponding actors in the current block instead of the corresponding actors in later blocks.

The SVPDF model as proposed in [12] contains, apart from port actors, only single-rate actors, i.e. actors which consume and produce one token. The model as proposed in this paper contains also more complex actors which include cyclo-static behavior. However, the delay introduced by repeating blocks is dependent only on the parameter values of blocks and the period of sources and sinks. It is however independent on the behavior of all other actors. Therefore, a similar proof as presented in [12] can be given for the SVPDF model presented in this paper. Flattening such an SVPDF model then results in a Cyclo-Static Dataflow (CSDF) model [6] where actors have static behavior, i.e. they are not parameterized.

Using the flattened SVPDF model sufficient buffer capacities can be determined for a given throughput constraint. Because a flattened model is equivalent to a CSDF model, existing analysis techniques can be used to determine buffer capacities [17, 20]. Buffer capacities can be either smaller or larger than the size of the corresponding array in the input OIL program. To increase the amount of pipelining buffer capacities can be increased. However, if the amount of pipelining is already sufficient, buffer capacities can be smaller than the array size. This is possible because there can be less constraints in the task graph than in the OIL program.

7. Case-Study

This case-study illustrates the derivation of an SVPDF model from a simplified OFDM transmitter [1]. The control flow of this application is shown in Figure 12. In the first nested for-loop the input bytes are delivered to the transmitter in 125 groups of two values. This input is QAM4 modulated and stored in the *mod* array. Next, this modulated data is interleaved and supplemented with 50 reference symbols. The interleaved data is then split into two parts and padded with zeros resulting in 512 symbols per part. An IFFT operation translates this data into the time domain and a cycle prefix is added giving 640 samples per part. Finally these two parts are upsampled and transferred to the digital-analog (DA) converter.

During parallelization, for every assignment and function statement and the sink a task is created and for every variable a buffer. Therefore, the task graph consists of 10 tasks and nine buffers. After generating the task graph, a corresponding SVPDF model is derived. A fragment of the model illustrating the model derivation is shown in Figure 13. The actors v_{mod} and v_{ref}, model the interleaving of the QAM4 modulated data with the reference symbols. These actors write into the *inter* array in an unknown order. Therefore, the complete array is acquired in advance, which is modeled by requiring 300 tokens for both actors. Locations are written one-by-one, thus respectively 150 and 50 tokens are produced to both consumers. Which array element is represented by these tokens is defined by the *interleave* function and is unknown during analysis.

```
sink DA[640*2] @ 50 MHz;

loop{
  forloop(0 <= j < 125){
    forloop(0 <= i < 2){
      mod[j*2 + i] = qam4(input);
  } }

  SA(inter){
    forloop(0 <= i < 250){
      inter[interleave(i)] = mod[i];

      if(i < 50){
        inter[interleave(250 + i)] = ref_symb[i];
  } } }

  forloop(0 <= i < 512){
    if(i < 150){
      real[i] = inter[i];
      imag[i] = inter[i+150];
    }
    else{
      real[i] = 0;
      imag[i] = 0;
  } }

  ifft_cp(real, imag, out tr, out ti);
  merge(tr, ti, out AD);
} while(1);
```

Figure 12: Simplified OFDM transmitter

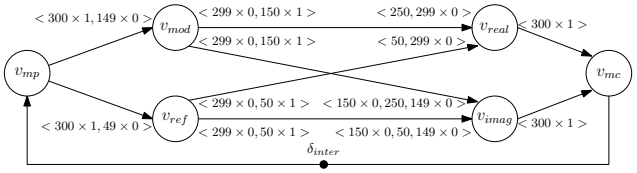


Figure 13: Fragment of the SVPDF model derived from the program in Figure 12, showing b_{inter} and connected tasks

Buffer	b_{DA}	b_{ref_symb}	b_{mod}	b_{inter}	b_{real}	b_{imag}	b_{tr}	b_{ti}
Capacity	3071	75	2	599	730	730	1215	1215

Table 1: Buffer capacities for the application from Figure 12

The SA-statement around the two producers on the *inter* variable allows these producers to be executed in parallel. Therefore, no sequence constraint needs to be added between v_{mod} and v_{ref}.

Array elements are consumed consecutively by the two readers of the *inter* variable. However, because of the unknown order of the producers, in the worst-case the last element written is the first one read. Therefore, both consumers require this last value to be available during the first iteration of the for-loop. Furthermore, because it is unknown which producer writes the required element, all values from both producers are required.

Using the generated SVPDF model, sufficient buffer capacities can be derived such that pipeline parallelism can be exploited to meet the throughput constraint. This throughput constraint is imposed on the application by the sink, which requires 1280 elements every 20 ns. After applying the step described in Section 6.2, the CSDF analysis method from [20] can be used to efficiently derive sufficient buffer capacities. These are shown in Table 1. As can be seen in the table, the buffer capacity for b_{mod} is two, which is less than the array size, which is 250. This shows that buffer capacity analysis techniques can optimize the required buffer memory below the array size.

Figure 14: Fragment of an execution trace of the application from Figure 12

Using the computed buffer capacities, the parallelized application can be executed on a multi-core system. Figure 14 shows a fragment of a trace of such an execution. This trace was captured on a system where the number of cores was equal to the number of tasks. Note that for some tasks, for example the *task_qam* task, the execution time is so small that the individual executions in the visualization can no longer be distinguished. The schedule shows that distinct iterations of the infinite while-loop can overlap during parallel execution. For example the third execution of the *task_dac* task overlaps partly with the fourth and fifth execution of the *task_ifftcp* task.

8. Conclusion

In this paper it is shown that a corresponding SVPDF analysis model can be automatically derived for an automatically parallelized multi-rate stream processing application. The automatic generation of the SVPDF model is enabled by an over-approximation of the synchronization dependencies. The generated SVPDF model is intended to verify whether an optimization improves the temporal behavior of such applications. It is shown that the SVPDF model that is generated from a parallelized application is always deadlock-free despite an over-approximation of the precedence constraints in the task graph. Furthermore, this model can be efficiently analyzed because only a single iteration of each block in the model has to be considered. This results in a model where CSDF analysis techniques can be used. The derivation of an SVPDF model requires a decoupling of the synchronization between tasks executing at different rates.

The key feature of the approach is that applications can contain modes and arrays which have arbitrary access patterns. If it can not be determined whether assignments to array elements are in single assignment form, precedence constraints from the sequential program are used to ensure functional correctness.

An OFDM transmitter is used to illustrate the relevance and applicability of the presented approach. It is shown that buffer sizes can be determined which result in a pipelined execution.

References

[1] H. Andrade et al. From streaming models to fpga implementations. In *Proc. of the Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2012.

[2] M. Bamakhrama, J. Zhai, H. Nikolov, and T. Stefanov. A methodology for automated design of hard-real-time embedded streaming systems. In *Design, Automation and Test in Europe (DATE)*, pages 941–946, 2012.

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

[4] 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.

[5] T. Bijlsma, M. Bekooij, and G. Smit. Circular Buffers with Multiple Overlapping Windows for Cyclic Task Graphs. *Proc. of the Int'l Conf. on High-Performance Embedded Architectures and Compilers (HiPEAC)*, 5(3), 2011.

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Trans. on Signal Processing*, 44(2):397–408, 1996.

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

[8] D. Culler, J. Singh, and A. Gupta. *Parallel computer architecture: A hardware/-software approach*. Morgan Kaufmann Publishers, San Francisco, 1999.

[9] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya. A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Proc. of the Int'l Conf. on Embedded Software (EMSOFT)*, pages 189–198. ACM, 2008.

[10] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[11] 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.

[12] S. Geuns, J. Hausmans, and M. Bekooij. Automatic dataflow model extraction from modal real-time stream processing applications. In *Proc. of the Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 143–152, 2013.

[13] J. Hausmans, M. Bekooij, and H. Corporaal. Resynchronization of cyclo-static dataflow graphs. In *Design, Automation and Test in Europe (DATE)*. IEEE, 2011.

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

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

[16] S. Stuijk. *Predictable mapping of streaming applications on multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007.

[17] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. on Computers*, 57(10):1331–1345, 2008.

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

[19] M. Wiggers, M. Bekooij, M. Geilen, and T. Basten. Simultaneous budget and buffer size computation for throughput-constrained task graphs. In *Design, Automation and Test in Europe (DATE)*, pages 1669–1672, 2010.

[20] M. Wiggers, M. Bekooij, and G. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proc. of the Design Automation Conf. (DAC)*, pages 658–663. ACM, 2007.

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

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